Performance Portability of Earth System Models with User-Controlled GGDML code Translation

Nabeeh Jum'ah[1] and Julian Kunkel[2]

[1] Universität Hamburg, Jumah@informatik.uni-hamburg.de
[2] University of Reading, j.m.kunkel@reading.ac.uk

Abstract. The increasing need for performance of earth system modeling and other scientific domains pushes the computing technologies in diverse architectural directions. The development of models needs technical expertise and skills of using tools that are able to exploit the hardware capabilities. The heterogeneity of architectures complicates the development and the maintainability of the models.
To improve the software development process of earth system models, we provide an approach that simplifies the code maintainability by fostering separation of concerns while providing performance portability. We propose the use of high-level language extensions that reflect scientific concepts. The scientists can use the programming language of their own choice to develop models, however, they can use the language extensions optionally wherever they need. The code translation is driven by configurations that are separated from the model source code. These configurations are prepared by scientific programmers to optimally use the machine's features.
The main contribution of this paper is the demonstration of a user-controlled source-to-source translation technique of earth system models that are written with higher-level semantics. We discuss a flexible code translation technique that is driven by the users through a configuration input that is prepared especially to transform the code, and we use this technique to produce OpenMP or OpenACC enabled codes besides MPI to support multi-node configurations.

Keywords: DSL; Meta-Compiler; Earth system modeling; Software development; Performance portability

1 Introduction

The diversity of the hardware architectures that arise to fulfill the performance needs of scientific applications represents a challenge for the scientists developing these applications. The maintainability and continuous development of applications suffers when optimizing code for several architectures. The semantics of a general-purpose language (GPL) limit the compiler's ability to exploit the underlying machine capabilities. Therefore, developers need to manually adapt the code to the target machine that will run the simulation, in order to make use

The final publication is available at Springer via https://doi.org/10.1007/978-3-030-02465-9_50

of the performance capabilities of the machine. Running a model on many different machines requires to redesign the code to fit the features of the different architectures and hardware configurations. This leads to a more complicated development process where code sections are redundantly coded with machine-dependent adaptations.

Another point that complicates the development process in this context is the technical knowledge necessary during software development. Scientists developing such models need to have deep technical knowledge of the lower-level details of the computer architectures that will run the model and the necessary software development skills to exploit their features.

We suggest an approach to improve the software development process of earth system models, which are a family of high-performance scientific applications. In our approach, the model's code is written mainly with a general-purpose language. Besides, we rely on the GGDML *(General Grid Definition and Manipulation Language)* extensions from Jumah et. al [10] that can be used to write parts of the model code.

GGDML, which was previously developed as part of the approach, provides a set of semantically-higher-level extensions. As those extensions are not part of the standard syntax of the general-purpose language, we need some processing before passing the model to the general-purpose language compiler. In our approach, the source code written with GGDML syntax is passed to a source-to-source translation tool. This tool processes the code and translates it into the modeling general-purpose language. During the processing phase by the source-to-source translation tool, the semantically higher-level language extensions allow the tool to perform transformations of the source code that could not be done by a GPL compiler. An example is changing the memory layout to generate optimized code for some hardware. The transformations applied during the source-to-source translation phase are primarily defined in a configuration file which is controlled by a scientific programmer. This user-controlled source-to-source translation process is the scope of this paper.

The **main contribution** of this work is a novel user-controlled code translation technique that relies on a configurable code transformation which makes use of the higher-level semantics exhibited by the language extensions. This technique allows the user to control how the language extensions affect the code translation and optimization process. The suggested approach still permits to write manually optimized codes, e.g. using pragmas, and supports incremental rewriting to some extent. In fact, an important aspect of the approach and the technique we suggest here is the opportunity that it gives to quickly explore different configurations and options and the corresponding performance issues without the need to change the source code or rewriting the kernels.

This paper is structured as follows: First, we discuss related work in Section 2. We then describe the approach in Section 3. In Section 4, we review the GGDML language extensions. Next, in Section 5 the configuration of the code translation process is discussed. In Section 6, the approach is evaluated with an example application that consists of various relevant kernels. Finally, we give a summary of the work presented in this paper in Section 7.

2 Related Work

To exploit the performance of the underlying hardware that runs a scientific application, different techniques have been used over time. The use of libraries which provide optimized codes like BLAS/LAPACK that can be called by applications is one way to allow applications to run efficiently. Annotating an application's code and processing the annotated code is another technique that allows performance improvement by processing the annotated code in a specific way to fit the features of the hardware. Code generation and language-specific features, e.g. generic programming with templates in C++, are techniques that are used to generate code for a specific machine. Domain-specific languages (DSLs) can be developed to describe the problem, and then a machine-specific code is generated based on the problem description that is being provided by the DSL. DSLs have been used in different ways. For example, some solutions depend on writing the application code completely with a stand-alone DSL. On the contrary, some solutions use DSLs to describe specific parts of the application. In such case, the DSL drives a code generation process that results of optimized code that will then be integrated within the other parts of the application. Some DSLs extend an existing language with technical constructs that allow users to direct specific optimizations, like OpenMP/OpenACC.

Libraries are widely used to provide performance-portable code for applications. A library implements a set of functions that an application can call to provide its functionality. Libraries like ELLPACK [15], BLAS [6], Intel's MKL [9] provide a set of mathematical functions that can be used by applications to solve mathematical problems (e.g., elliptic differential equations and linear algebra). Such libraries implement the functions that they provide to the applications in a way that is aware of the architectural features and capabilities of a specific type of machines. The developers of such libraries port them to different types of machines (e.g. [18] ports the BLAS 3 to multi-GPU platforms) to allow applications to be developed for different machines while still using the hardware efficiently.

The implementations provided by the libraries are normally highly optimized, however, the calling application could still miss some optimization opportunities particularly between subsequent library calls. Such lost opportunities for optimization can be avoided when using active libraries. The source-to-source translation features in the active libraries OP2 [12] and OPS [14] analyzes the behavior of the application. An application calls the APIs that the library provides and the source-to-source translation procedure handles the optimization of the code for a specific target architecture. OPS uses its own DSL that allows the source-to-source translation process to generate optimized code.

The use of DSLs to drive the code generation of machine-specific optimized code was employed in different ways. Stella [8] for example, provides an embedded DSL for C++ that allows describing stencil codes for structured grids. The stencils and the grids are specified using the DSL, which mainly uses the C++ syntax, and the machine-specific code is generated by a special backend for the supported hardware. GridTools [2] also builds on Stella. It provides a generic C++ API to provide performance portability for grid-based codes. GridTools use a DSL for stencil computations and for halo exchange to provide codes for machines with multiple nodes.

Intel's YASK [20] is built on a DSL that enables users to define the technical decisions to control how the stencil operation is applied to the grid, including the domain decomposition of the grid over multiple nodes. The specification of stencil operations is enabled by the use of object-oriented C++ features and generic programming. The generated code is optimized for Xeon and Xeon Phi processors. Optimization techniques like vector folding [19] and cache blocking are used to optimize memory bandwidth usage.

Some DSLs are more tightly based on scientific domains, e.g., Atmol [3] and Liszt [4], in contrast to the focus on technical details. However, the move to the declarative programming needs the model developers to move to a new paradigm for software development.

In contrast to standalone DSLs, there are DSLs that added some extensions to existing general-purpose languages, e.g. Physis [11] which adds some extensions to C++, and Icon DSL [16] which adds some extensions to Fortran. [10] suggested a set of language extensions that are language neutral. The suggested DSL provides a set of extensions that can be used regardless of the general purpose language that is used to develop the model.

In code annotation techniques, descriptions can be added to the source code to provide further information about it, and to tell how it could be optimized. Such information is provided by the developers within the source code. Hybrid Fortran [13], HMPP [5], Mint [17], CLAW [1] use annotation to drive the optimization process. In Gung Ho [7] the scientific code is separated into higher level algorithms and lower-level kernels. Directives drive the generation of the code between the two layers to handle loops and parallelization. The code annotation technique needs to push the technical details within the source code. However, with these technical annotations, the scientists need to care for the lower-level optimization details.

In this paper we extend the work that has been done with the GGDML DSL [10]. We use the higher-level semantics offered by the extensions of GGDML along with a highly configurable code translation technique to transform the source code of a model into a machine-specific code that exploits the features of the underlying architecture where the model would be run.

3 Approach

In this section we review our approach to improve the software development process and discuss the user-controlled translation technique.

3.1 The General Approach

The approach is built around using higher-level language extensions. This 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. This eliminates the need to provide technical details about optimizations within the source code. Thus, the model developer, that is usually a trained scientist in the domain field, does not need to think about the hardware and performance details. In our approach we commit to the principle of separation of concerns:

- Domain scientists write the scientific problem from a scientific perspective
- Scientific programmers provide the DSL and machine-specific optimization

The scientists formulate the scientific problem within the source code based on the scientific concepts of their domain science. 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).

3.2 The User-Controlled Source-to-Source Code Translation

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 described in this paper is guided by a configuration information that allows the translation process to make the necessary transformations in order to exploit the capabilities of the machine. This information is prepared by scientific programmers who have the necessary technical knowledge to harness the power of the underlying architectures and hardware configurations.

To enable the developers to use scientific concepts while programming, the language extensions of the DSL are developed in collaboration between the scientists and the scientific programmers. Then, the developed extensions are defined within the configuration information. For example, they can define 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 configuration allows the user to control how 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 GGDML Review

In this section we review the GGDML, which was developed as one part of our approach to provide the higher-level language semantics.

The GGDML DSL has been developed as a set of language extensions to support the development of icosahedral-grid-based earth system models. Although the extensions have been developed based on the three icosahedral models Dynamico, ICON, and NICAM which are written in the Fortran language, we use the extensions to develop a testbed application in the C language. 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.

GGDML offers a set of declaration specifiers that allow to mark a variable to contain values 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. 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 translation technique is designed to support extending the set of specifiers. This dynamic support for the extensibility of the tool stems from the highly configurable translation technique that is described in the next section.

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 a specific set of grid elements. 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. 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.

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 how the data is accessed or where it is stored.

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 write kernels independently from 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[3]. 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;
}
```

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 one level below the last level. 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.

5 Machine-Specific Configuration

The extensibility of the DSL, i.e., the set of the language extensions and the configurability of the code transformation process are key parts of our approach. The basic set of language extensions provided by GGDML are not applied by the source-to-source translation process as a constant set of extensions. Instead,

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

the translation technique allows to define new extensions and how they affect the code transformation process.

We mentioned in the previous section the declaration specifiers that mark the variables. The translation process accepts configuration information that defines named sets of specifiers and a set of specifiers under each set. For example, the basic set of specifiers that GGDML provides are implemented as a set of specifiers called 'loc' which includes the 'cell' and 'edge' specifiers to specify the spatial position of the variables value with respect to the grid, and another set that is called 'dim' which includes the specifiers '3D' and '2D' to specify the dimensionality of the grid over which the variable is defined. So, the specifiers are not built into the translation tool as compilers do usually. To the contrary, the users can define any set of specifiers as they need. To demonstrate the idea, the line

```
SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=3D|2D)
```

shows how we configured the translation process to translate our test application.

The sets of the specifiers provide information that enables the translation tool to handle further code transformation steps. So, whenever a variable is declared with any of the defined specifiers, the tool would use that information to handle any transformations related to that variable. Among the code processing that uses such information are the allocation and deallocation of the variables' data in memory, and the transformation of the addresses of the variables' data in memory, and choosing the consistent memory layout to access the variables data.

The translation tool uses configuration information to control the allocation and the deallocation of the variables. The allocation/deallocation codes are generated based on this configuration input and the specifiers used to declare the variable.

5.1 Grid Configuration

The configuration provides information that describes the grids that are used in the model. This includes the definition of the grid's components, e.g., the cells of the three-dimensional grid or the cells of the grid's two-dimensional surface.

The tool allows the configuration to specify defaults to simplify expressing the intended grids to traverse when writing a kernel. For example, in a test code we have used the defaults to write kernels with a simple iterator expression that consists only of the word 'grid'. In the configuration that we prepared for the test application, we have

```
GLOBALDOMAIN:
 ...
 DEFAULT=CELL3D[CELL3D:cell,ce,c][EDGE3D:edge,ed,e]
```

This allows us to traverse the cells of the three-dimensional grid by default or when we use one of the words 'cell','ce' or 'c' as an iterator index. Thus, the iterator

```
foreach cell in grid
```

traverses the cells of the three-dimensional grid. Likewise, this example configuration allows to traverse the edges of the three-dimensional grid when we use one of the words 'edge','ed' or 'e' as an iterator index. Thus, the iterator

```
foreach edge in grid
```

traverses the edges of the three-dimensional grid. If the defaults are not intended to be used, then the name of the grid to be traversed, or additionally the grid specification operators should be used.

The definition of the grids describes the grids that are used in the model regardless of how the processing will be divided between nodes. The details of the domain decomposition and halo exchange are provided in its own part of the configuration information.

The definition of the grids allows using variables defined in the source code. For example, to define the height of the grid in a test code we defined it to take the values between 1 and *height*, which is a variable used in the source code. This allows us in the experiments to pass the height as a command line parameter to run the model with multiple heights without recompiling the code.

5.2 Configurable Access Operators

To access the value of a variable at a grid element when applying a computation within an iterator, we refer to it by the iterator's index. In stencil codes, it is essential to access the neighboring elements. To handle this, GGDML provides a basic set of operators. For example, the *cell.neighbor* allows to refer to the neighbor cell of the current cell, assuming that *cell* is the name of the iterator's index.

The set of access operators are not limited to the basic set that GGDML provides. In fact, the access operators – even the basic set of GGDML access operators – are defined in the configurations that the translation tool uses. For example, the operator *above* is defined by

```
above(): height=$height+1
```

to tell the tool how to access the element that we refer to with this operator. In a test code, besides to this operator definition, we also defined the same operator with an overloaded form that takes a parameter to specify a number of levels above the current element, e.g., the cell above some levels to allow references like *cell.above*(2). The access operators define the relationships between the grids. The connectivity of the unstructured grids is defined by the definitions of the access operators.

5.3 Memory Layout

The variables are accessed by the iterator index, which abstracts an element among a set of elements of a grid. This abstraction does not specify where the data are stored in memory and how to access them. The translation tool uses the user-provided configuration information to define the mapping and know how to access the data.

The allocation, that is guided by the configuration, allows to control the placement of the variable's data. Several information allow the translation tool to decide how to access the data of a variable: Firstly, the information about a variable that is provided by the specifiers used to declare the variable. The iterator's index and the access operators that are used to refer to the variable's data are translated into the indices that address the data in memory in a step that includes some transformations. The applied transformations use the grid definitions as part of the transformation process. Further mathematical transformations on the indices can be controlled by the configuration information. For example, the interchange of the indices, or even transforming a three-index-based address (space) into a one-index-based address according to a formula like

```
 INDEX=$0*g->blkSize*g->height+$1*g->blkSize+$2
```

or even more complex formulas including functions, e.g. a filling curve, are possible. The expression

```
gv_temp[cell.above()]
```

is transformed into the three-index address

```
gv_temp[(block_index)][((height_index) + 1)][(cell_index)]
```

and applying the mentioned address transformation formula transforms the address into

```
gv_temp[(block_index) * g->blkSize * g->height +
        ((height_index) + 1) * g->blkSize + (cell_index)]
```

The example also demonstrates the use of the access operator *above* that is mentioned in Section 5.2.

The memory layout is a key factor to exploit hardware configurations. The choice of the transformation formulas is an important decision to improve the performance of an application. Fortunately, the simple and quick configurability of the memory layout makes the exploration of the memory layouts and the corresponding performance on different architectures a simple task.

5.4 Parallelization

The parallelization of the kernels is an important part of the code translation for the iterators. That is essential to use the hardware features to improve the performance. The translation tool allows to provide configuration information to control the parallelization process. For example, the user uses the following line

```
 0:pragma omp parallel for
```

to let the tool use the OpenMP scheduler. The line tells that an OpenMP pragma is used and that the blocks are mapped to the 'for' iterations that will be run in parallel. Alternatively, on GPUs we can map the blocks to the OpenACC 'gangs' for example.

In our test codes, we have generated MPI codes with OpenMP to target multicore processors, and OpenACC to target GPU-accelerated machines. We could annotate the loops to use the cores of the multicore processors and the streaming multiprocessors of the GPUs to run the kernels in parallel. The easy configuration change of the parallelization allowed us to easily explore different parallelization alternatives to explore performance impact.

To enable the models to run on multiple nodes, the translation tool provides the necessary code translation like domain decomposition and halo exchange. The processing load of the kernels over grids that are defined in the configuration to support the model is divided between the nodes that run the model.

The code transformation of the iterators includes the grids decomposition such that each node is responsible to process its own part of the grid. As part of the code transformation, the translation tool analyzes the kernels and generates the necessary halo exchange code. However, this is controlled by the user-provided configuration information. This information controls the initialization and finalization of the communication library, and the transmission/reception of the halo data. This includes a completely configurable halo pattern definition and initialization.

In the test codes, we have used the MPI library to handle the communications of the halo data between the nodes.

6 Evaluation

In this section, we discuss some experiments to evaluate the work described in this paper. 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.

6.1 Test Application

A testbed code in the C-programming language is used to demonstrate and 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 explicit time steps during each of which the model components are called to do their computations – a component can be considered a scientific process. 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.

6.2 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.

6.3 Results

In the first 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.

Table 1: Performance in GFLOPS on a 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

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 1 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. 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.

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 2.

Both the strong and the weak scalability efficiency are calculated according to the equations

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

Table 2: Performance in GFLOPS on a Single node with a P100/V100 GPU

	Serial	P100			V100		
		performance GFLOPS	Memory throughput GB/s		performance GFLOPS	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

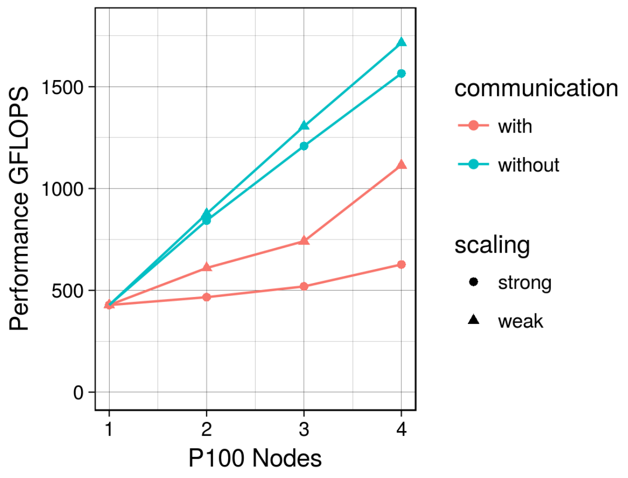


Fig. 1: Performance Scalability on nodes with P100 GPUs (performance is measured in GFLOPS)

Table 3: Performance Scalability on nodes with P100 GPUs (performance is measured in GFLOPS)

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%

$$Efficiency_{weak} = T_1/T_N \cdot 100\% \qquad (2)$$

where N is the number of processes, T_1 is the execution time on one process, and T_N is the execution time on N processes. The results are shown in Figure 3. 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 nodes and 1,2,4,8,16,32, and 36 cores per node. The measurements are shown in Figure 4.

7 Summary

In this paper, we discussed an approach to improve the software development process of icosahedral-grid-based earth system models. We investigated the ex-

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

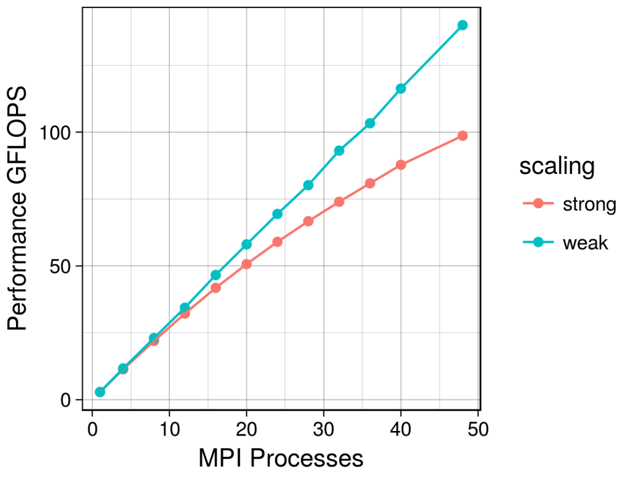
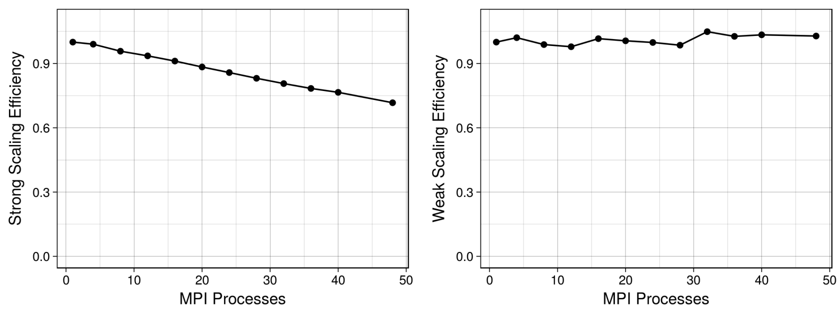


Fig. 2: MPI process scalability

tensibility of the model's programming language with higher-level extensions abstracting scientific concepts regardless of the technical concepts related to the machine and the performance optimization. The approach relies on using a source-to-source translation process that uses the higher semantics of the extensions besides to user-provided configuration information together to transform the source code into a target-machine-optimized code. The configuration information allows the users to control the code transformation process. The performance portability is an important feature of the approach. The source code of a model is written once, and the translation procedure can use many configuration files to generate code versions which work on different machines while exploiting the features of those machines to run with high performance.

More extensions can be added depending on the needs of the scientists. This is possible as the translation process is configurable by information that controls the code transformation. Among the configuration information, the translation



(a) Strong Scaling Efficiency (b) Weak Scaling Efficiency

Fig. 3: Scaling Efficiency

process accepts information that defines new extensions and how they affect the code transformation.

The approach is evaluated with a test application that has been written in the C language, with the use of the GGDML extensions. Different configurations were prepared to translate the source code into different targets. The resulting code versions were run on two machines. The results showed the impact of transforming the code to support different memory layouts, and the performance gain when using the cores of a multicore processor and the streaming multiprocessors of a GPU to apply the computations in parallel. In addition, the results showed the scalability when running on multiple nodes. The translation process was successfully used to generate codes that run on multiple multicore nodes and multiple GPU-accelerated nodes and the evaluation shows that the approach could provide performance portability for the software development of the models which need scalability.

7.1 Future Work

We are working further on the optimization of the halo data communication and the minimization of the communication overhead particularly to reduce the costs for the GPU version. Another important path for the research we intend to continue is the improvement of the inter-kernel and inter-module optimization. We currently provide with the translation tool some basic fusion of the kernel loops. Also, the tool currently carries out some analysis of the kernel computations. However, further work can be done to investigate the optimization opportunities over the set of kernels that are called in each time step, or even between time steps.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA)

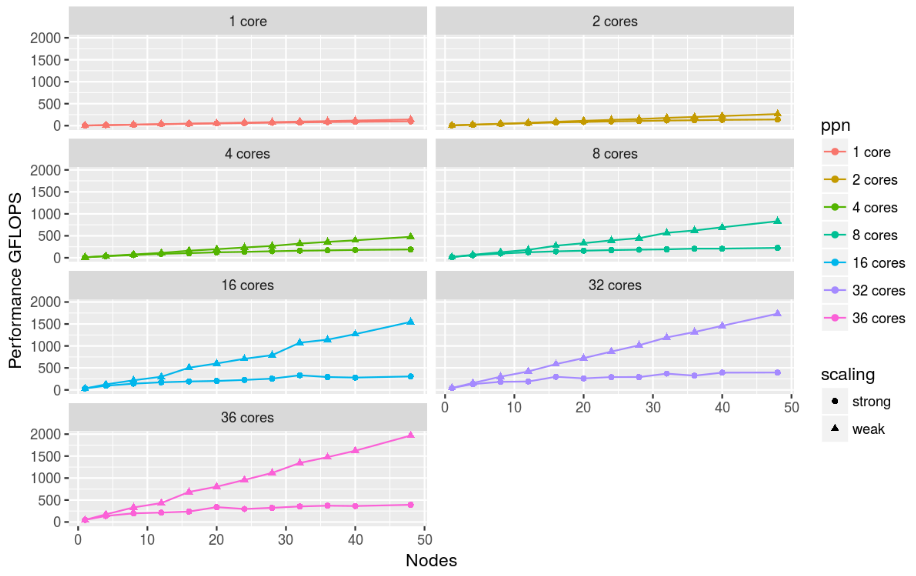


Fig. 4: MPI+OpenMP scalability

(GZ: LU 1353/11-1). We would like to thank NVIDIA who supported this work with allowing to run some tests on their PSG cluster, and the German Climate Computig-Center (DKRZ) where we also have run some tests on the Mistral supercomputer.

References

1. CSCS Claw. `http://www.xcalablemp.org/download/workshop/4th/Valentin.pdf`. Accessed: 2017-12-22.
2. CSCS GridTools. `https://pasc17.pasc-conference.org/fileadmin/user_upload/pasc17/program/post144s2.pdf`. Accessed: 2017-12-22.
3. Robert A van Engelen. Atmol: A domain-specific language for atmospheric modeling. *CIT. Journal of computing and information technology*, 9(4):289–303, 2001.
4. 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.
5. 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.
6. Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

7. 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.
8. Tobias Gysi, Oliver Fuhrer, Carlos Osuna, Benjamin Cumming, and Thomas Schulthess. Stella: A domain-specific embedded language for stencil codes on structured grids. In *EGU General Assembly Conference Abstracts*, volume 16, 2014.
9. MKL Intel. Intel math kernel library. 2007.
10. Nabeeh Jumah, Julian Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Yann Meurdesoif. GGDML: Icosahedral models language extensions. *Journal of Computer Science Technology Updates*, 4(1):1–10, 2017.
11. 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.
12. GR Mudalige, MB Giles, I Reguly, C Bertolli, and PHJ 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.
13. Michel Müller and Takayuki Aoki. Hybrid fortran: High productivity gpu porting framework applied to japanese weather prediction model. *arXiv preprint arXiv:1710.08616*, 2017.
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.
15. John R Rice and Ronald F Boisvert. *Solving elliptic problems using ELLPACK*, volume 2. Springer Science & Business Media, 2012.
16. 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.
17. 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.
18. Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.
19. Charles Yount. Vector folding: improving stencil performance via multi-dimensional simd-vector representation. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 865–870. IEEE, 2015.
20. 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: 2017-12-22).