D4.2 Annual report

Julian Kunkel Nabeeh Jum'ah Anastasiia Novikova Michel Müller
Hisashi Yashiro Thomas Ludwig

Workpackage:	WP4 Project coordination
Responsible institution:	Ludwig
Contributing institutions:	RIKEN, IPSL
Date of submission:	March 2018

Contents

Abstract

1 Relation to the Project

The annual report for the second year is split into two parts: The public part contains information about the project progress, status and next step and is published on the project webpage. A confidential section contains aspects relevant only for DFG project management.
The following text is the description of the project proposal for this task and deliverable:

- *Project management*
  In the project management, the overall project status and progress is monitored and checked biannually by the project executive. Every half year the status and arising risks are determined and discussed with the partners. This task allows proactive resolving of issues that might lead to a delay of deliverables and the overall project. Our management approach is based on management by exception with delegated responsibility. The project executive is named by UHHSC and responsible to meet the overall project goals. The task leader is responsible for the timely progress within the task, the WP leader for the success of the whole WP. Periodically, the success of tasks are periodically checked by the WP leader and similarly the progress of WPs is verified by the project executive. Conflicts are handled by the responsible leader of the tasks and, if escalated to the WP leader, if that is not possible conflicts between stakeholders are resolved by the project executive.

- *Internal communication*
  In this task, the project-internal communication channels and tools are established. The communication within the consortium are organized by regular phone (video) conferences, a Jabber chat room, mailing lists and via a Redmine project management system. All artifacts are hosted on central (Git) version control repositories at the University of Hamburg. This collaborative workspace is provided at M2 and allow secure communication of results, deliverables and meeting notes. The project executive organizes bi-annual face-to-face meetings and monthly (video) phone conferences and document the discussion. In 2018 we will have only one face-to-face meeting because of the reduced runtime of the proposal of our Japanese colleagues. Also, the project executive is the point of contact between the DFG and the project partners and prepare the project reports and deliverables for release.

- *Quality assurance*
  Within this task's responsibility, quality control is systematically performed to validate the correctness and appropriateness of all generated deliverables and dissemination artifacts. For every artifact at least one uninvolved partner is obliged to spot mistakes and comment the quality and that they are conform to the rules of good scientific practice from DFG; we systematically evaluate scientific accuracy, completeness and readability for documentation and papers while correctness, maintainability and performance aspects are covered for code reviews.

- *Dissemination*
  The project dissemination publishes results in scientific journals and proceedings of workshops and conferences. We intend to submit 10 publications to relevant journals and events. Since the con- sortium recognized the importance of open access to scientific results, we use self archiving where permitted. All produced documents and code is made available for the general public. UHHSC establishes a website to host the project information and all generated artifacts (with the exception of the confidential management report that is provided only to DFG). We encourage participation of third parties by permitting access to our code base under an open source license. The website also links to related activities including other SPPEXA projects.

- *Communication with third-parties*
  Firstly, the project executive establishes communication channels to vendors, consortia and standardization bodies responsible for standards such as scientific file formats and DSLs and manage the communication efforts of task leaders to these groups. Secondly, our supporters are included, the group of scientists that is interested in project results but not actively contributing. See Section 6 for a description of our supporters and their letters of support. We also establish communication channels to relevant SPPEXA and H2020 projects – such as the ESiWaCE Center of Excellence for numerical weather predication and climate, and participate in their workshops and activities. In conjunction with our bi-annual face-to-face meetings, we

organize a workshop to facility knowledge exchange between related projects, supporting organizations and individuals.

2 WP1: Higher-Level Code Design

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.

2.1 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 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

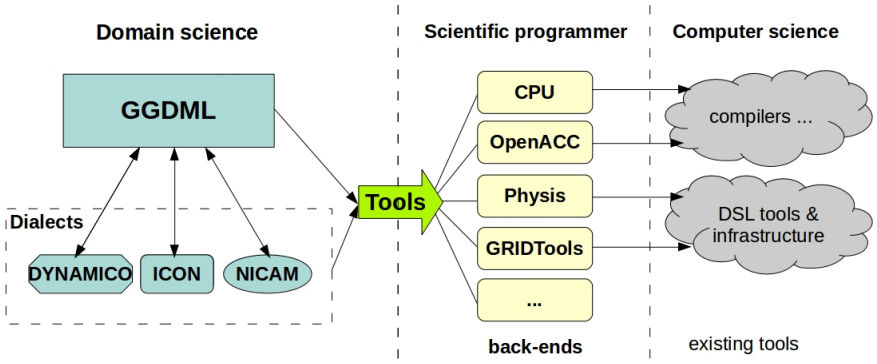


Figure 1: Separation of Concerns

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.

2.2 Computation in Icosahedral Models

In climate models, grids help the development of codes which compute variables values. Grids are used to discretize the space over which the variables are measured. In various models we see different kinds of grids. Some models use structured grids which simply address data by Euclidean space coordinates with longitudinal and latitudinal surface dimensions. However, there are some shortcomings of such grids, which limit a model's capability to provide some functionality. For example, a rectangular grid for the whole earth surface contains rectangles with different sizes which vary according to the location of the rectangle. The need of some models to offer some functions which can not be considered with structured grids, led to go beyond such grids. Among those models are icosahedral models.

An icosahedral model is one that uses an icosahedral grid, which 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. 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 between the cells, or at the vertices (Figure 2).

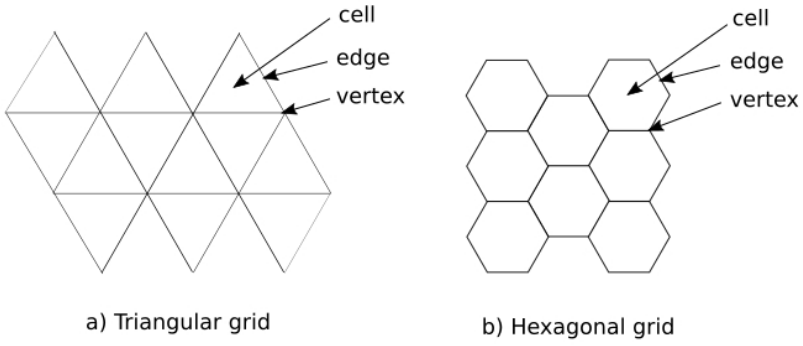


Figure 2: Icosahedral grids and variables

Moving from structured grids to icosahedral grids allows a model to provide new functionalities. However, it complicates the storage of the data of the variables. N-dimensional arrays do not directly support the storage of the icosahedral-grid-bound variables like they do in the structured grids. A transformation function is then needed to address the data of the grid-bound variables. HEVI methods with a space-filling curve (e.g. Hilbert

speace-filling curve) for the horizontal surface enable the data addressing. Some considerations like caching affect the choice of the space-filling curve.

2.3 GGDML

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 with 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 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. 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.

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 the how the data are accessed or where they are 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 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:

```
1 extern GVAL EDGE 3D gv_grad;
2 extern GVAL CELL 2D gv_o8param[3];
3 extern GVAL CELL 3D gv_o8par2;
4 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:

```
1  FOREACH cell IN grid|height{1..(g->height-1)}
2  {
3    GVAL v0 = REDUCE(+,N={0..2},
4       gv_o8param[N][cell] * gv_grad[cell.edge(N)]);
5
6    GVAL v1 = REDUCE(+,N={0..2},
7       gv_o8param[N][cell] * gv_grad[cell.edge(N).below()]);
8
9    gv_o8var[cell] = gv_o8par2[cell]* v0
10                  + (1-gv_o8par2[cell]) * v1;
11 }
```

[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.

2.4 Hybrid Fortran

There is a gap in HPC studies and frameworks geared for climate- and weather models where productivity plays a central role. In this work we therefore wanted to close this gap. We have used the Japanese regional weather prediction model "ASUCA" as a case study on how to design a maintainable and efficient GPU codebase for NWP. Two technical challenges stand out in terms of their potential impact on productivity and code integrity:

1. The granularity of physical processes tends to be too coarse for GPU (in ASUCA's example a single parallelization is used for all physical processes run once per long timestep, as shown in Figure 3).
2. The memory layout of data structures often needs to be reordered, as these codes have been developed with CPUs in mind. As an example, ASUCA, WRF and COSMO all use k-first storage order in their original code, while i-first or j-first is required for efficient use of GPUs [SAO14] [MLB01] [COG+13].

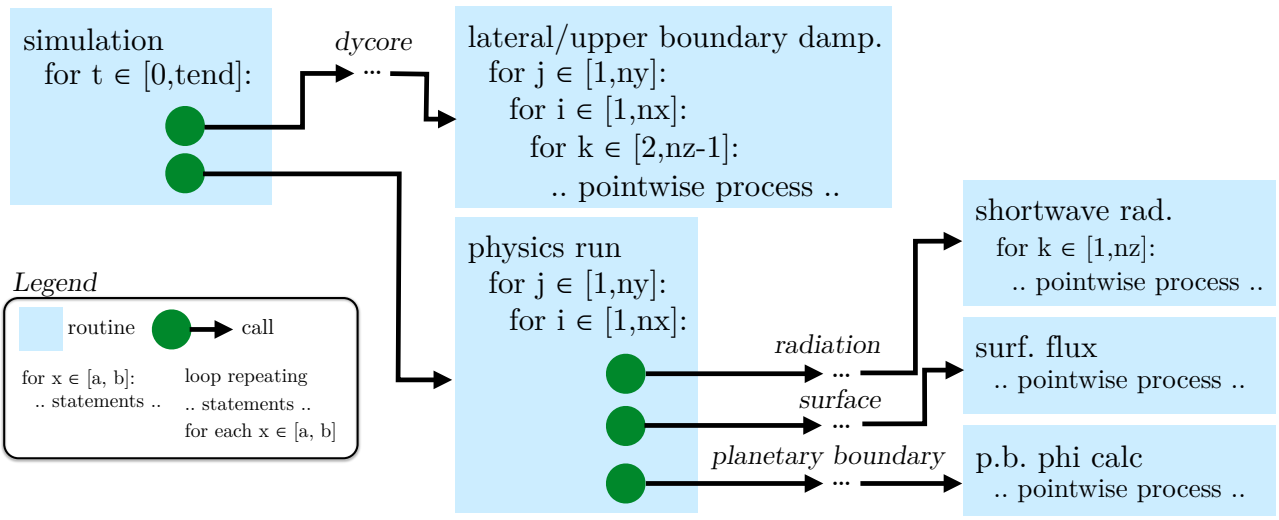


Figure 3: ASUCA original code structure with coarse granularity for physical processes.

Existing approaches to solve these issues have led to varying combinations of limited performance, poor maintainability and invasive rewrites. For ASUCA, an approach has been sought that comes with none of these drawbacks.

2.4.1 Granularity Abstraction- and Layout Transformation Method

Our proposed method to enable GPUs in NWP has been implemented as a language extension and transformation framework called "Hybrid Fortran". This technique requires only minimal changes for a CPU targeted codebase, a significant advancement in terms of productivity. The aforementioned technical challenges are addressed as follows: Parallel loops are abstracted through the means of a new language construct ("parallel region"), allowing to specify multiple granularity levels with the same code, depending on the target architecture (thus keeping support for CPU). Figure 4 gives an overview of this approach for ASUCA. Memory layout is transformed at compile-time, enabling the reuse of existing code with zero run-time overhead. In the backend the parallelization is implemented through source-to-source translation for many-core GPUs or multi-core CPUs using an original Python-based framework.

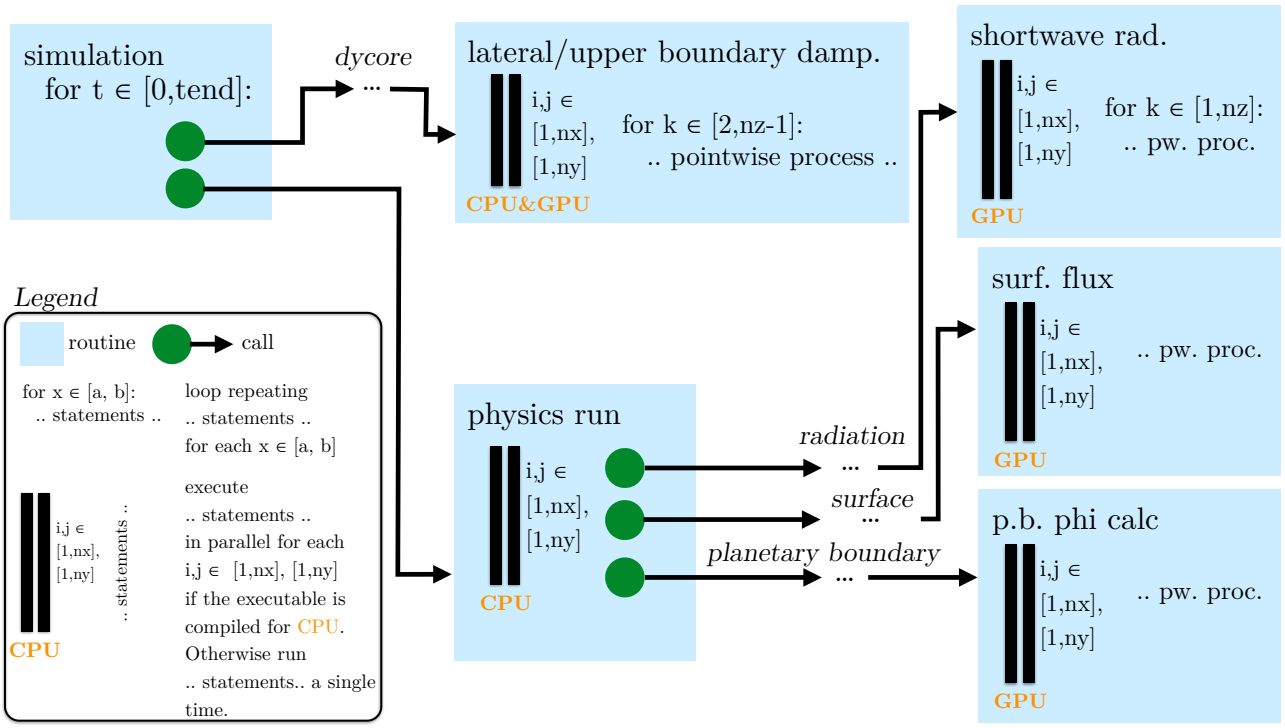


Figure 4: ASUCA hybrid code structure with two granularity levels depending on target architecture.

2.5 Experiments

In this section we discuss some experiments and results related to the use of GGDML and Hybrid Fortran in the development of model codes.

2.5.1 GGDML Experiments

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.

2.5.1.1 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.

2.5.1.2 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.

2.5.1.3 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

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

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.

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

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 5 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.

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 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

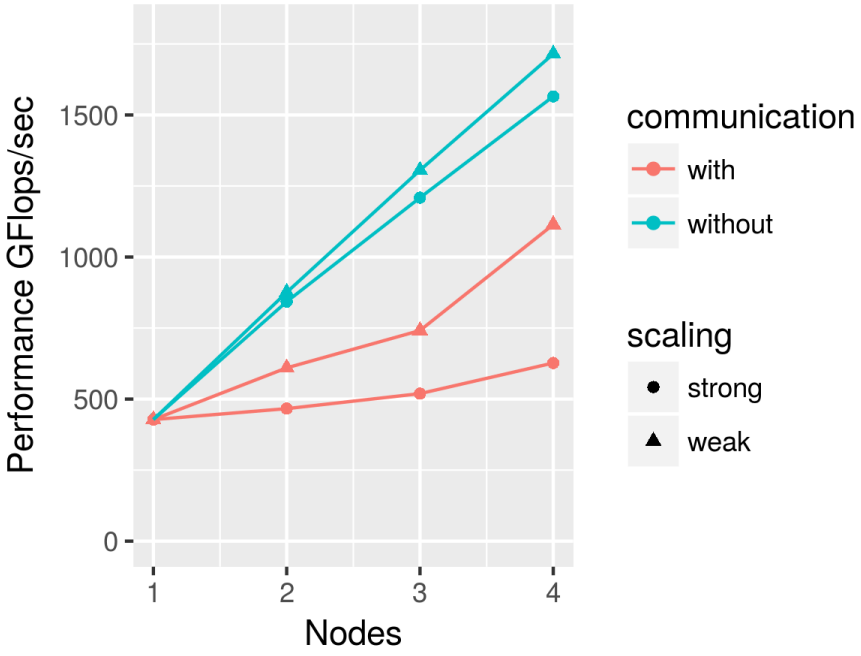


Figure 5: P100 scalability

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

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 6.
Both the strong and the weak scalability efficiency are calculated according to the equations

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

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

and the results are shown in Figure 7. 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 8.

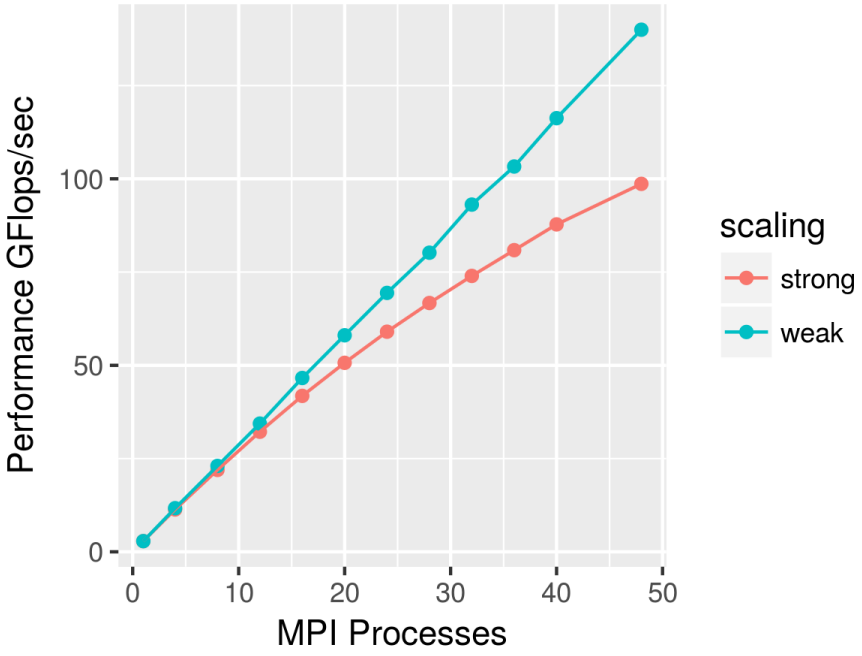
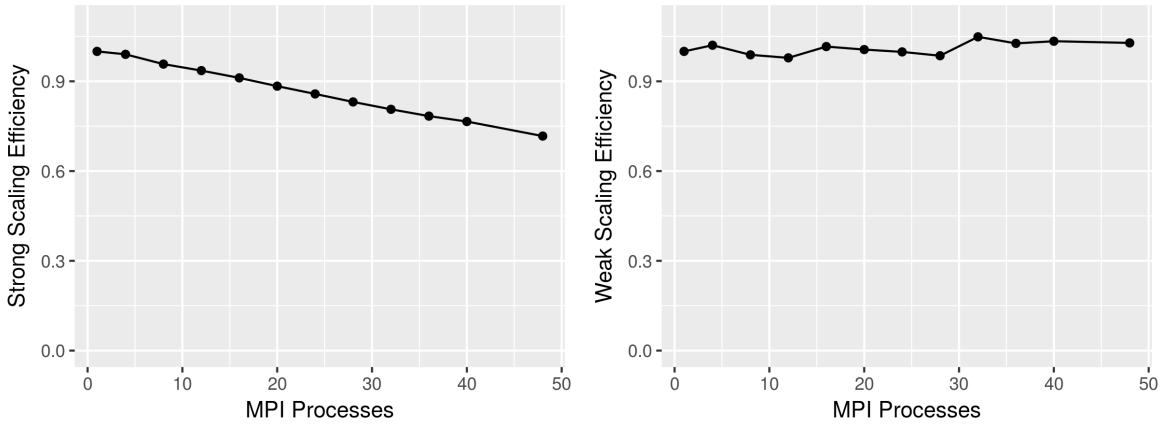


Figure 6: MPI process scalability



(a) Strong Scaling Efficiency

(b) Weak Scaling Efficiency

Figure 7: Scaling Efficiency

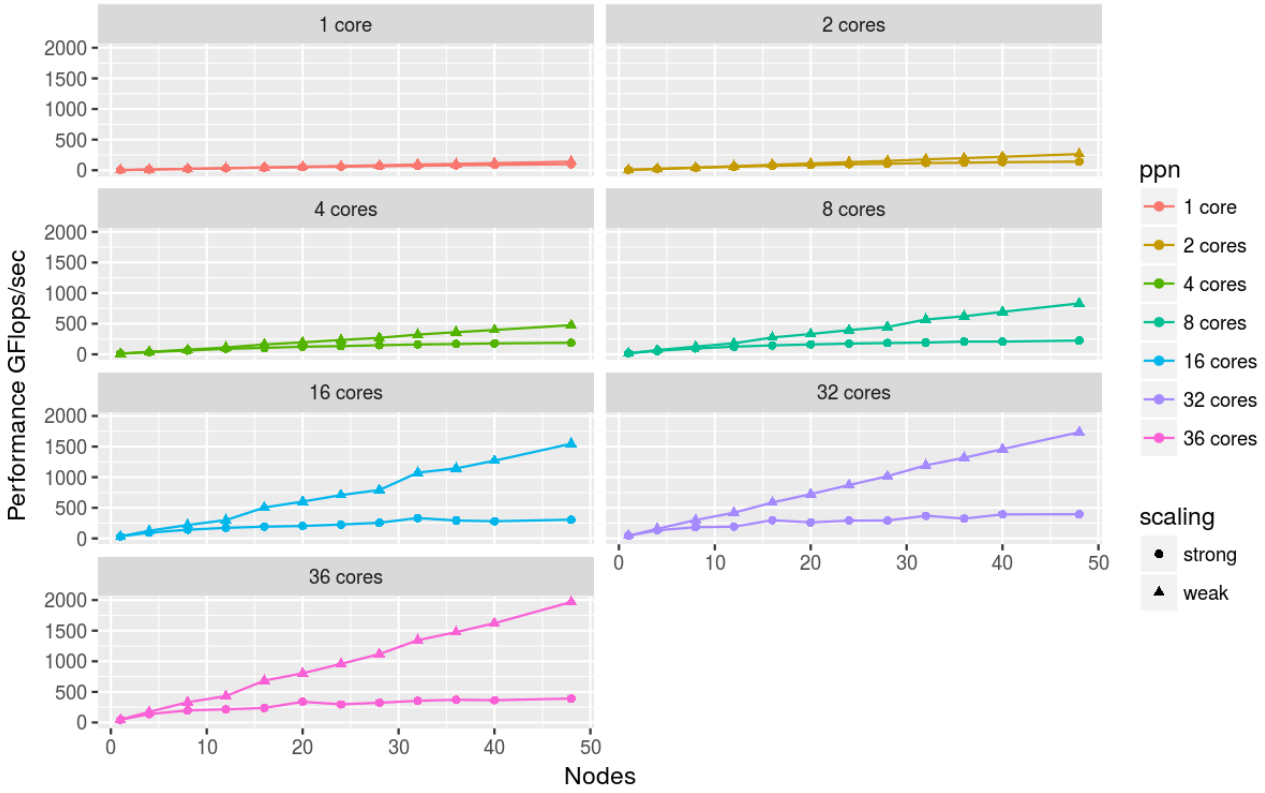


Figure 8: MPI+OpenMP scalability

2.5.2 Results regarding Hybrid Fortran and ASUCA

Hybrid Fortran has been applied to ASUCA's dynamical core as well as planetary boundary layer-, surface and radiation processes, cloud microphysics, precipitation. Using the aforementioned memory layout transformation, the new hybrid version of ASUCA is using IJK order for GPU and KIJ order for CPU. Physical processes are transformed at compile-time from coarse-grained- to fine-grained parallelization for GPU.

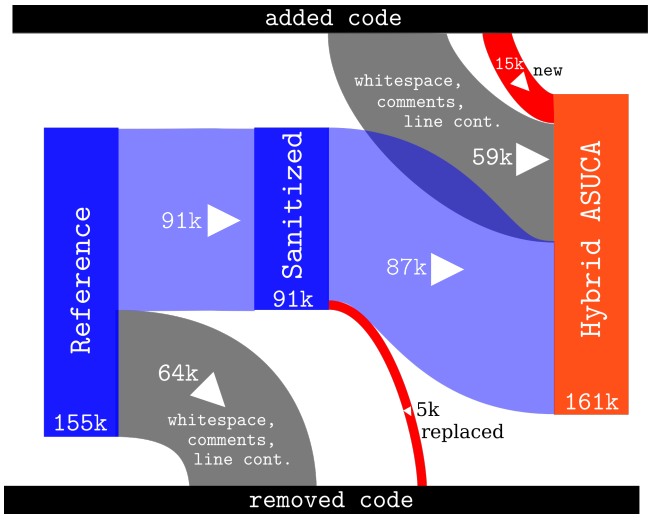


Figure 9: Changes to ASUCA from reference- to hybrid version by number of lines of code.

2.5.2.1 Productivity ASUCA's codebase encompasses more than 150k lines of code. By using extensive code transformation techniques, more than 85% of the original codebase was left unchanged and the overall code size has been extended by less than 4% (as shown in Figure 9), even though the new codebase targets two very different hardware architectures instead of one. This shows very promising productivity, maintainability and ease-of-adoption of the proposed method in an operational setting [MA18a].

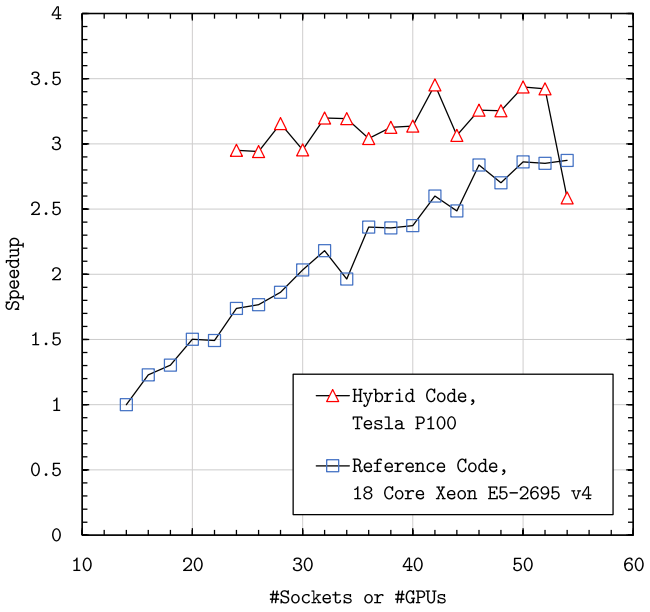


Figure 10: Strong scaling results for ASUCA on Reedbush-H.

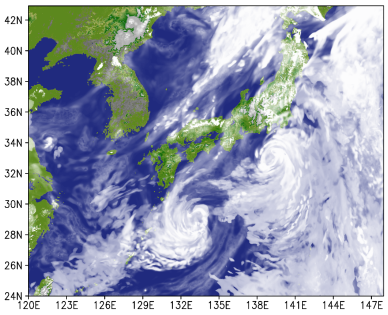
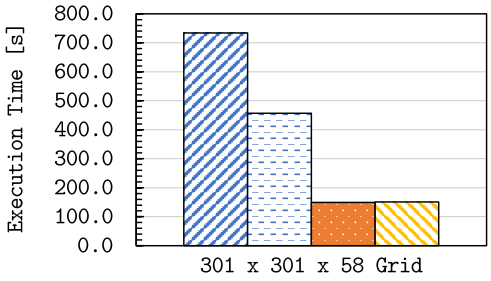


Figure 11: Cloud cover result from real data simulation.

2.5.2.2 Performance By means of a minimal weather application that resembles ASUCA's code structure we have compared our method to a performance model as well as today's commonly used method, OpenACC. As a result, the Hybrid Fortran-based implementation is shown to deliver the same or better performance than OpenACC for GPU (many-core) and OpenMP for CPU (multi-core) and its performance agrees with the model both on CPU and GPU. In a full-scale production run, using an ASUCA grid with 1581 x 1301 x 58 cells and real-world weather data in 2km resolution (see Figure 11 for the resulting cloud cover), 24 NVIDIA Tesla P100 running the Hybrid Fortran based GPU port are shown to replace more than 50 18-core Intel Xeon Broadwell E5-2695 v4 running the reference implementation (Figure 10). In terms of kernel performance, a speedup of more than 3x is shown between latest-generation CPUs and GPUs (Reedbush-H) and a speedup of 4.9x is shown on TSUBAME 2.5, comparing Kepler K20x GPUs to six-core Westmere CPUs (see Figure 12) [MA18b].



ASUCA Reference, 4 x 6-core Xeon X5670	734.0
ASUCA Reference, 1 x 18-core Xeon E5-2695 v4	456.7
Hybrid ASUCA, 4 x Tesla K20x	148.9
Hybrid ASUCA, 1 x Tesla P100	151.1

Figure 12: Kernel performance for ASUCA.

3 WP2: Massive I/O

In the AIMES project we develop libraries and methods to utilize lossy compression. The SCIL library[2] provides a rich set of user quantities to define from, e.g., HDF5. Once set, the library shall ensure that the defined data quality meets all criteria. Its plugin architecture utilizes existing algorithms and aims to select the best algorithm depending on the user qualities and the data properties.

3.1 Design

The main goal of the compression library SCIL is to provide a framework to compress structured and unstructured data using the best available (lossy) compression algorithms. SCIL offers a user interface for defining the tolerable loss of accuracy and expected performance as various quantities. It supports various data types. In Figure 13, the data path is illustrated. An application can either use the NetCDF4, HDF5 or the SCIL C interface, directly. SCIL acts as a meta-compressor providing various backends such as the existing algorithms: LZ4, ZFP, FPZIP, and SZ. Based on the defined quantities, their values and the characteristics of the data to compress, the appropriate compression algorithm is chosen. SCIL also comes with a pattern library to generate various relevant synthetic test patterns. Further tools are provided to plot, to add noise or to compress CSV and NetCDF3 files. Internally, support functions simplify the development of new algorithms and the testing.

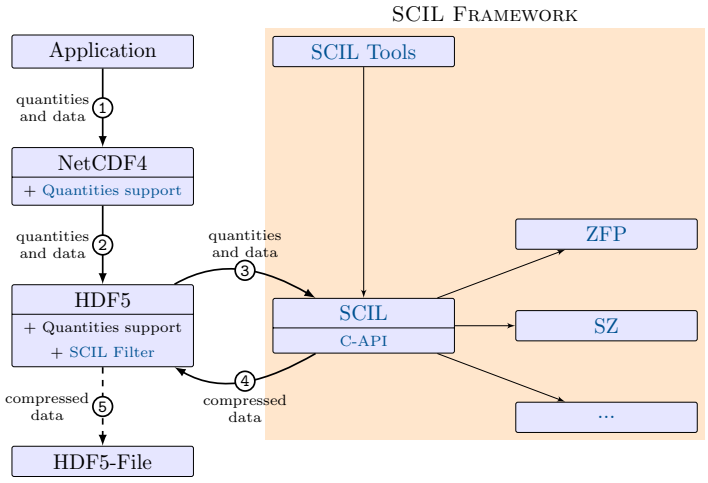


Figure 13: SCIL compression path and components

3.1.1 Supported Quantities

There are three types of quantities supported:

[2]The current version of the library is publicly available under LGPL license:
`https://github.com/JulianKunkel/scil`

3.1.1.1 Accuracy quantities define the tolerable error on lossy compression. When compressing the value v to $\hat{v}$ it bounds the residual error ($r = v - \hat{v}$):

- **absolute tolerance**: $v - \text{abstol} \leq \hat{v} \leq v + \text{abstol}$
- **relative tolerance**: $v/(1 + reltol) \leq \hat{v} \leq v \cdot (1 + reltol)$
- **relative error finest tolerance**: used together with rel tolerance; absolute tolerable error for small v's. If relfinest $> |v \cdot (1 \pm reltol)|$, then $v - \text{relfinest} \leq \hat{v} \leq v + \text{relfinest}$
- **significant digits**: number of significant decimal digits
- **significant bits**: number of significant digits in bits

SCIL must ensure that all the set accuracy quantities are honored, meaning that one can set, e.g., absolute and relative tolerance, and the value that is most strict quantity is chosen.

3.1.1.2 Performance quantities define the expected performance behavior for both compression and decompression (on the same system). The value can be defined according to: 1) absolute throughput in MiB or GiB; or 2) relative to network or storage speed. It is considered to be the expected performance for SCIL, but it may not be as strictly handled as the qualities – there may be some cases in which performance is lower. Thus, SCIL must estimate the compression rates for the data, this value is not yet covered by the algorithm selection, but will be in the future. The system's performance must be trained for each system using machine learning.

3.1.1.3 Supplementary quantities: An orthogonal quantity that can be set is the so called *fill value*, a value that scientists use to mark special data points. This value must be preserved accurately and usually is a specific high or low value that may disturb a smooth compression algorithm.

3.1.2 Algorithms

The development of the two algorithms sigbits and abstol has been guided by the definition of the user quantities. Both algorithms aim to pack the number of required bits as tightly as possible into the data buffer. We also consider these algorithms useful baselines when comparing any other algorithm.

3.1.2.1 Abstol This algorithm guarantees the defined absolute tolerance. Pseudocode for the Abstol algorithm is provided in Figure 14.

```
compress(data, abstol, outData){
  (min,max) = computeMinMax(data)
  // quantize the data converting it to integer, according to abstol
  tmp[i] = round((data[i] - min) * abstol)
  // compute numbers of mantissa bits needed to store the data
  bits = ceil(log2(1.0 + (max - min) / abstol))
  // now pack the neccessary bits from the integers tightly
  outData = packData(tmp, bits)
}
```

Figure 14: Pseudocode for the Abstol algorithm

3.1.2.2 Sigbits This algorithm preserves the user-defined number of precision bits from the floating point data but also can honor the relative tolerance. One precision bit means we preserve the floating point's exponent and sign bit as floating point implicitly adds one point of precision. All other precision bits are taken from the mantissa of the floating point data. Notice that, for a denormalized number (the leading "hidden" bit is always 0), for example, mantissa 0.0000001 with 5 precision bits will be cut to 0.00000. That means that the significand part is missing. Note that the sign bit must only be preserved, if it is not constant in the data. Pseudocode for the Sigbits algorithm is illustrated in Figure 15. When a relative tolerance is given, it is converted to the number of precision bits.
Since the values around the 0 contain a huge range of exponents in IEEE floating point representation, e.g., 0 ± 10^{-324} which is usually not needed, the *relative error finest tolerance* limits the precision around 0 to remove exponents and preserve space.

```
compress(data, precisionBits, outData){
  // preserve the exponent always
  (sign, min, max) = computeExponentMinMax(data)
  // compute numbers of bits needed to preserve the data
  bits = sign + bits for the exponent + precisionBits - 1
  // convert preserved bits into an integer using bitshift operators
  tmp[i] = sign | exponent range used | precision Bits
  // now pack the bits tightly
  outData = packData(tmp, bits)
}
```

Figure 15: Pseudocode for the Sigbits algorithm

3.1.3 Synthetic data creator

Synthetic data covers patterns such as constant, random, linear steps (creates a hyperplane in the diagonal), polynomial, sinusoidal or by the OpenSimplex algorithm. OpenSimplex implements a procedural noise generation [LLC+10]. Each pattern can be parameterized by the min/max value, random seed and two pattern-specific arguments:

- sin: Base frequency (1 means to have one sine wave spanning all dimensions) and number of recursive iterations by which the frequency is doubled each time and the amplitude is halved
- poly4: Initial number for random number generator and degree of the polynomial
- steps: Number of steps between min/max
- simplex: Initial frequency and number of iterations, in each iteration the frequency is doubled and the amplitude halved

Additionally, mutators can be applied to these patterns such as step (creating a linear N-dimensional interpolation for the given number of data points) and repeat which repeats a number N-times. Explanation of file names are listed in Table 5.

Pattern name	Example
random[mutator][repeat]-[min-max]	randomRep10-100
random[min]-[max]	random0-1
steps[number]	steps2
sin[frequency][iterations]	sin35
poly4-[random]-[degree]	poly4-65432-14
simplex[frequency][iterations]	simplex102

If not specified otherwise, min=0 and max=100.

Table 5: Schemas for pattern names

3.2 Evaluation

In the evaluation, we utilize SCIL to compress the data with various algorithms. In all cases, we manually select the algorithm. The test system is an Intel i7-6700 CPU (Skylake) with 4 cores @ 3.40GHz; one core is used for the testing and turbo boost is disabled.

3.2.1 Test Data

All data use single precision floating point (32 bit) representation. A pool of data is created 10 times with different random seed numbers from several synthetic patterns generated by SCIL's pattern library and kept in CSV-files. Synthetic data has the dimensionality of (300 x 300 x 100 = 36 MB).
Additionally, we utilize the output of the ECHAM atmospheric model[RBB+03] which stored 123 different scientific variables for a single timestep as NetCDF and the output of the hurricane Isabel model which stored 633 variables for a single timestep as binary[3]. The scientific data varies in terms of properties and in particular, the expected data locality. For example, in the Isabel data many variables are between 0 and 0.02 many between -80 and +80 and some are between -5000 and 3000.

[3] http://vis.computer.org/vis2004contest/data.html

3.2.2 Experiments

For each of the test files, the following setups are run[4]:

- Lossy compression preserving T significant bits
  - Tolerance (T): 3, 6, 9, 15, 20 bits
  - Algorithms: zfp, sigbits, sigbits+lz4[5]
- Lossy compression with a fixed absolute tolerance
  - Tolerance: 10%, 2%, 1%, 0.2%, 0.1% of the data maximum value [6]
  - Algorithms: zfp, sz, abstol, abstol+lz4

Each configuration is run 3 times measuring compression and decompression time.

3.2.3 Synthetic Patterns

The compression ratios for two synthetic patterns are shown in Figure 16, the figure shows a variable absolute tolerance and the number of precision bits, respectively.

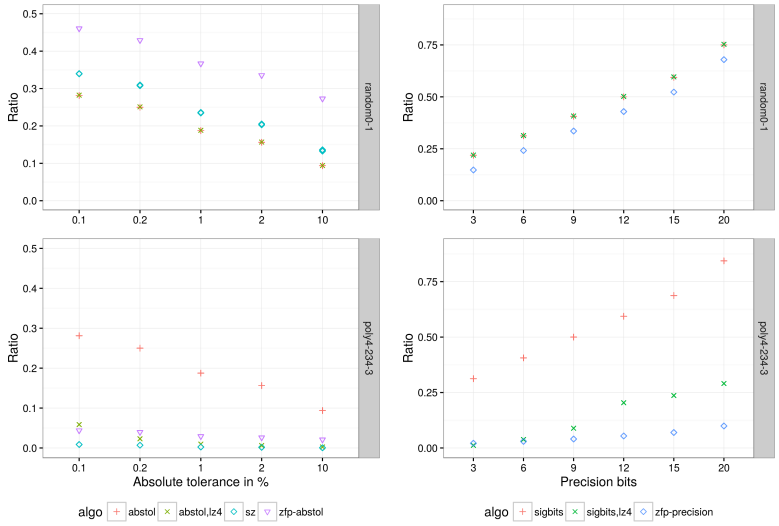


Figure 16: Mean harmonic compression factor for synthetic data based on user settings

3.2.3.1 Absolute tolerance: First, we look at the random patterns in Figure 16 and the absolute tolerance (left column). The x-axes contains the absolute tolerance between 0.001 and 0.1. In this experiment, the actual tolerance is computed based on the maximum value to enable comparison between different datasets. A value of 0.1 means that 10% of the maximum is used as absolute tolerance. Thus, data can be quantized and encoded in 5 values representing the mean of (0-20%, 20-40%, ...), i.e., 3 bits are needed out of 32 bits ⇒ a compression ratio of 9.4% can be achieved. Therefore, as abstol performs this quantization, it is expected to reach that level regardless of the redundancy in the data which it does. ZFP and SZ predict the next data point based on the experience, therefore, as expected the achievable ratio for random data is worse than for abstol by a constant of 0.05. Applying a lossless compressor on pure random data does not help, too.
The polynomial of degree 3 is compressed well by all algorithms, SZ benefits from the locality.

3.2.3.2 Precision bits: Sigbits uses a number of bits needed for encoding the sign bit and exponent range (max - min) and the precision bits as defined by the user. For the random data between 0 to 1, 5 bits are used to represent the exponent[7] leading to a ratio of 22% for 3 precision bits (= 2 mantissa bits). The regular polynomial benefits for a low precision from the repeatability, but behaves non-linearly.

[4]The versions used are SZ from Aug 14 2017 (git hash 29e3ca1), zfp 0.5.0, LZ4 (Aug 12 2017, 930a692).
[5]This applies first the Sigbits algorithm and then the lossless LZ4 compression.
[6]This is done to allow comparison across variables regardless of their min/max. In practice, a scientist would set the reltol or define the abstol depending on the variable.
[7]Since the function `rand()` is used to create the test data.

ZFP cannot be compared easily as it does not support the precision bit quantity but a fixed ratio and compresses data blockwise – this leads to validation errors. For comparison reasons, we use `zfp_stream_set_precision()` with the same number of bits as utilized by Sigbits. Still with a fixed setting, ZFP typically yields a better ratio at the cost of precision.

3.2.4 Scientific Data

3.2.4.1 Ratio Depending On Tolerance Next, we investigate the compression factor depending on the tolerance level for the scientific data. The graphs in Figures 17 and 18 show the mean compression factor for two types of climate data files varying the precision for the algorithms ZFP, SZ, Sigbits and Abstol. The mean is computed on the pool of data, i.e., after compression, a factor of 50:1 means all compressed files occupy only 2% of the original size.
We will discuss the absolute tolerance first: With 1% of tolerance, a compression factor of more than 15 can be achieved on both data sets. For the ECHAM dataset, Abstol+LZ4 outperforms SZ, for the Isabel data SZ outperforms Abstol+LZ4 initially clearly. In both cases, the ratio of Abstol+LZ4 improves with the absolute tolerance.
When using precision bits, ZFP yields a better factor on the Isabel data than Sigbits. However, note that since this dataset uses high fill values, the result cannot be trusted. With 3 precision bits (relative error about 12.5%), a ratio of around 10 is achievable. It can be concluded that the ECHAM dataset is more random compared to the Isabel data which stores a finer continuous grid.

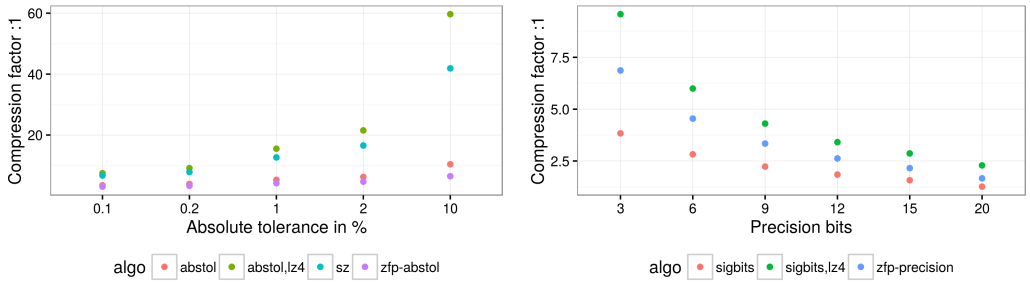


Figure 17: Mean harmonic compression factor for ECHAM data based on user settings

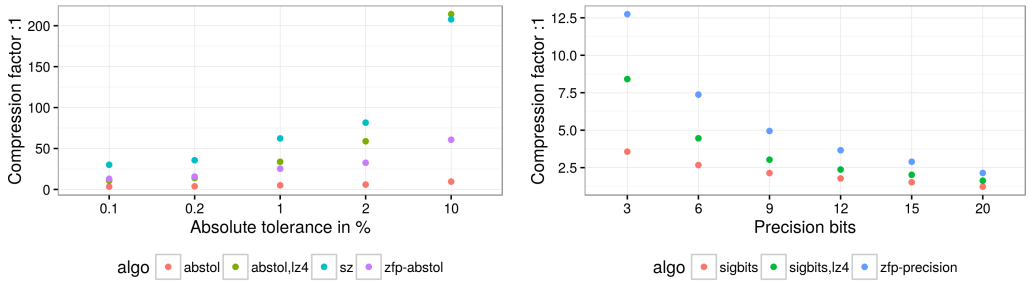


Figure 18: Mean harmonic compression factor for Isabel data based on user settings

3.2.4.2 Fixed Absolute Tolerance To analyze throughput and compression ratio across variables, we selected an absolute tolerance of 1% of the maximum value.
Mean values are shown in Table 6a. The synthetic random patterns serve as baseline to understand the benefit of the lossy compression; we provide the means for the different random patterns. Abstol+LZ4 yields a 20% reduction compared to SZ for ECHAM and the random data while for Isabel data it needs about 50% more space. Compression and decompression speed of Abstol+LZ4 is at least 2x the speed of SZ. LZ4 supports the detection of random data and avoids compression in that case increasing performance for random data significantly. ZFP achieves worse compression ratios but with a better compression speed than SZ. For Isabel data the decompression is even a bit higher than when using Abstol+LZ4.

3.2.4.3 Fixed Precision Bits Similarly to our previous experiment, we now aim to preserve 9 precision bits. mean values are given in Table 6b. The Sigbits algorithm is generally a bit faster than Abstol. It can

	Algorithm	Ratio	Compr. MiB/s	Decomp. MiB/s
ECHAM	abstol	0.190	260	456
	abstol,lz4	0.062	196	400
	sz	0.078	81	169
	zfp-abstol	0.239	185	301
Isabel	abstol	0.190	352	403
	abstol,lz4	0.029	279	356
	sz	0.016	70	187
	zfp-abstol	0.039	239	428
Random	abstol	0.190	365	382
	abstol,lz4	0.194	356	382
	sz	0.242	54	125
	zfp-abstol	0.355	145	241

(a) 1% absolute tolerance

	Algorithm	Ratio	Compr. MiB/s	Decomp. MiB/s
ECHAM	sigbits	0.448	462	615
	sigbits,lz4	0.228	227	479
	zfp-precision	0.299	155	252
Isabel	sigbits	0.467	301	506
	sigbits,lz4	0.329	197	366
	zfp-precision	0.202	133	281
Random	sigbits	0.346	358	511
	sigbits,lz4	0.348	346	459
	zfp-precision	0.252	151	251

(b) 9 bits precision

Table 6: Harmonic mean compression of scientific data

be seen that Sigbits+LZ4 outperforms ZFP mostly for ECHAM data, although ZFP does typically not hold the defined tolerance. For Isabel data ZFP is typically better than Sigbits+LZ4 (but again does not hold the precision bits). In this case, the LZ4 compression step is beneficial for a small fraction of files, for 50% it does not bring any benefit.

4 WP3: Benchmarking

To assess an applicability of DSL and new I/O methods, sample programs of the Earth system models, which have lesser complexity than original mode, is useful. We developed kernel benchmark suite and mini-app package in the AIMES project. The benchmark codes are extracted from atmospheric models participating this project; ICON, DYNAMICO, and NICAM. The kernel suite named "IcoAtmosBenchmark v1" is expected to use not only for this project but also for the similar studies in the future. As shown in Fig. Figure 19, we prepared input data and reference output data for easy validation of results. We also arranged documentation about the kernels.

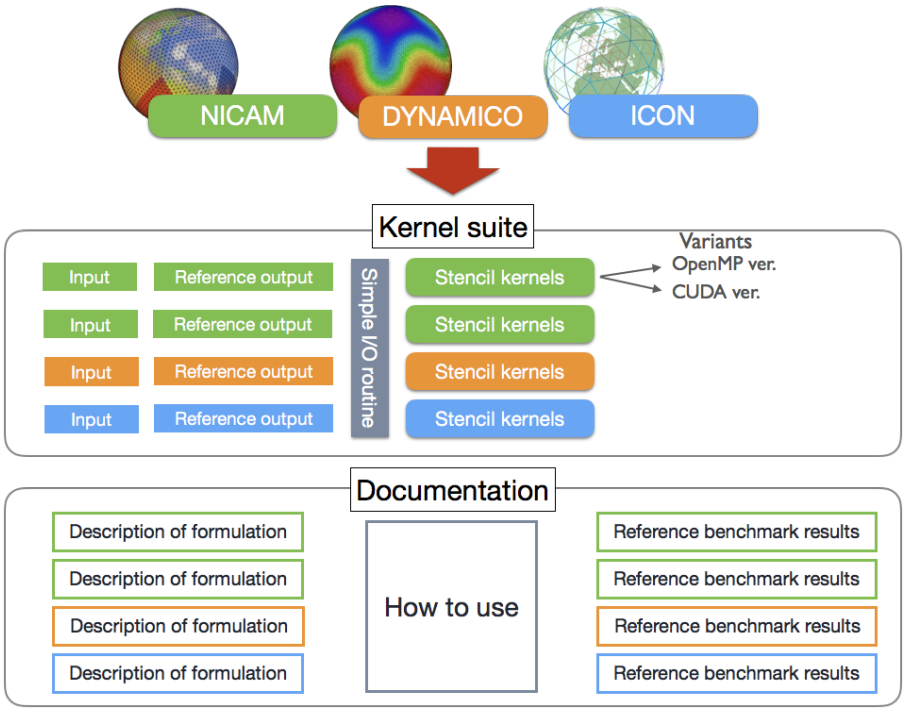


Figure 19: Overview of IcoAtmosBenchmark v1

4.1 kernels

4.1.1 NICAM kernels

An icosahedral grid system is used in NICAM to cover whole domain of the Earth's surface. We separate the complex topology of the grid system into the structured and unstructured part. The grids are decomposed into the tiles, and one or more tiles are allocated to each MPI process. The horizontal grids in the tile are kept in a 2-dimensional structure. On the other hand, a topology of the network of the tiles is complex. We selected and extracted six+1 computational kernels from the dynamical core of NICAM, as samples of the stencil calculation on the structured grid system. We also extracted a communication kernel, as a sample of halo exchange in the complex node topology. The feature of each kernel is as follows.

4.1.1.1 dyn_diffusion 2-D horizontal diffusion operator is a typical sample of stencil calculation on the icosahedral grid. The computation is mainly divided into the two parts: horizontal interpolation and flux convergence with diffusion coefficients. In NICAM, all of the variables are co-located in horizontal direction. The values are interpolated from the center of hexagonal control cell to the cell vertex. And then, the values are interpolated into the cell edge. The fluxes are calculated on each edge of hexagon. The diffusion term is provided from the total of the flux of all cell edge.

4.1.1.2 dyn_divdamp 3-D flux divergence damping operator is used for the numerical stability in the dynamical core. This scheme calculates the gradient of divergence of the horizontal and vertical wind vector. Similar to the diffusion kernel, the kernel has interpolation part and flux convergence part. The value is interpolated into the edge of the control cell in horizontal direction and the interface of the cell in vertical direction. The memory footprint of this 3-D kernel is larger than the 2-D operator kernels. Therefore this kernel is suitable to evaluate techniques for performance optimization such as cash blocking and the setting of the parallelism.

4.1.1.3 dyn_vi_rhow_solver This kernel is the tridiagonal matrix solver for vertical component of momentum. This is the part of the horizontal-explicit vertical-implicit solver, which is commonly used in the numerical weather prediction model. The computation has recurrence in vertical direction. However, dimension of horizontal direction is first in the data layout. The method of performance optimization of this kernel will be different between CPU and GPU.

4.1.1.4 dyn_vert_adv_limiter This and following two kernels are used in tracer advection scheme. The tracers such as water vapor should be kept positive value in advection scheme. The conservation of the total mass of each tracer is needed, too. To satisfy these requirements, "flux limiter" is commonly used. In NICAM, a flux limiting scheme proposed by Thuburn (1996)[8] is used. This kernel calculates corrected quantity of tracer for 1-D vertical advection. The computation is more complex than the simple stencil operators. It contains intrinsic math function such as max() and min().

4.1.1.5 dyn_horiz_adv_flux This kernel calculates 2-D horizontal tracer mass fluxes on the cell edge. For the horizontal direction, NICAM uses upwind-biased advection scheme proposed by Miura (2007)[9]. The mass centroid of parallelogram (eq.(9) in Miura (2007)) is also calculated in this kernel.

4.1.1.6 dyn_holiz_adv_limiter This kernel is horizontal version of the tracer mass flux limiter scheme. This kernel is largest in the six stencil kernels from NICAM. It contains big loop body, conditional branch, and max/min functions.

4.1.1.7 dyn_metrics This is the setup routine for the stencil operators. The spatial metrics, such as the area of the control cell, the length of the cell edges, and the factor of interpolation from cell center to cell vertex, appear in the stencil operators. In NICAM, the combination of the metrics are pre-calculated in order to reduce the computation in each operator kernel. This kernel is provided supplementary for the DSL implementation.

[8] Thuburn, J.(1996): Multidimensional flux-limited advection schemes. J. Comput. Phys. 123, 74-83.

[9] Miura, H.(2007): An upwind-biased conservative advection scheme for spherical hexagonal–pentagonal grids. Mon. Wea. Rev., 135, 4038-4044.

4.1.1.8 communication Communication kernel contains subroutines of setup, halo exchange, and unit test. We prepared different settings of process number. In NICAM, the base units of the tile are ten diamonds, which are made by pairs of triangles of the icosahedron. Each diamond is subdivided according to the domain decomposition level. The node topology changes drastically by the decomposition level and process allocation. Such a complicated topology is a pending problem of the implementation for not only the stencil framework but also the PGAS language.

4.1.2 DYNAMICO kernels

The grid structure of DYNAMICO is similar to NICAM; an icosahedral grid system is used, and the decomposed grid groups have 2-D structure. However, the feature of the stencil calculation scheme is different between the two models. As discussed in WP1, the time-developing (prognostic) variables and the located point of them in the grid system are different. We selected and extracted four kernels from the dynamical core of DYNAMICO as follows.

4.1.2.1 comp_pvort This module defines subroutine caldyn, which is the main subroutines for dynamics part of the model, and several sub-subroutines for various terms in the governing equation, such as potential vorticity, geopotential, etc. This subroutine calculates potential vorticity.

4.1.2.2 comp_geopot This subroutine calculates potential geopotential.

4.1.2.3 comp_caldyn_horiz This subroutine calculates several horizontal terms, including mass flux, Bernouilli term, etc.

4.1.2.4 comp_caldyn_vert This subroutine calculates vertical mass flux and vertical transport.

4.2 input data and reference data

All kernels are single subroutines and almost same as the source codes in the original model. They are put into the wrapper for the kernel program. Values of input variables in the argument list of the kernel subroutine are stored as a data file just before the call in the execution of the original model, and they are read and given to the kernel subroutine in the kernel program. Similarly, the values of output variables in the argument list are stored just after the call in execution, and they are read and compared to the actual output values of kernel subroutine, the differences are written to the standard output for validation. For easy handling of the input/reference data by both the Fortran program and C program, we prepared an I/O routine written in C.

4.3 documents

We provided a user manual, which contains the brief introduction of each model, the description of each kernel, usage of kernel programs and sample results. This information is helpful for users of this kernel suite in the future.

5 Summary and Conclusions

We used GGDML as a higher-level set of language extensions to write a test application with scientific abstractions. This allowd us to write the code without the need to do any optimization within the source code. The code in general is small compared to hand-written code, and code redundancies do not exist as in hand-written code.

As the source code was developed with GGDML, the translation configurations were prepared for different run configurations on multi-core processors and GPUs. Different memory layouts were prepared also to explore performance impact.

The GGDML source-to-source translation tool could translate the test code into multi-core CPU code and into GPU code both with multiple node support. The 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

[COG+13] Ben Cumming, Carlos Osuna, Tobias Gysi, Mauro Bianco, Xavier Lapillonne, Oliver Fuhrer, and Thomas C Schulthess. A review of the challenges and results of refactoring the community climate code COSMO for hybrid Cray HPC systems. *Proceedings of Cray User Group*, 2013.

[LLC+10] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. A survey of procedural noise functions. In *Computer Graphics Forum*, volume 29, pages 2579–2600. Wiley Online Library, 2010.

[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.

[MLB01] J Michalakes, R Loft, and A Bourgeois. Performance-portability and the Weather Research and Forecast Model. *HPC Asia 2001*, 2001.

[RBB+03] Erich Roeckner, G Bäuml, L Bonaventura, Renate Brokopf, Monika Esch, Marco Giorgetta, Stefan Hagemann, Ingo Kirchner, Luis Kornblueh, Elisa Manzini, et al. The atmospheric general circulation model ECHAM 5. PART I: Model description, 2003.

[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*, SC '14, pages 251–261, Piscataway, NJ, USA, 2014. IEEE Press.