DOI: 10.14529/jsfi140103

Towards a performance portable, architecture agnostic implementation strategy for weather and climate models

Oliver Fuhrer[1], *Carlos Osuna*[2], *Xavier Lapillonne*[2], *Tobias Gysi*[3,4], *Ben Cumming*[5], *Mauro Bianco*[5], *Andrea Arteaga*[2], *Thomas C. Schulthess*[5,6,7]

We propose a software implementation strategy for complex weather and climate models that produces performance portable, architecture agnostic codes. It relies on domain and data structure specific tools that are usable within common model development frameworks – Fortran today and possibly high-level programming environments like Python in the future. We present the strategy in terms of a refactoring project of the atmospheric model COSMO, where we have rewritten the dynamical core and refactored the remaining Fortran code. The dynamical core is built on top of the domain specific "Stencil Loop Language" for stencil computations on structured grids, a generic framework for halo exchange and boundary conditions, as well as a generic communication library that handles data exchange on a distributed memory system. All these tools are implemented in C++ making extensive use of generic programming and template metaprogramming. The refactored code is shown to outperform the current production code and is performance portable to various hybrid CPU-GPU node architectures.

Keywords: numerical weather prediction, climate modeling, hybrid computing, programming models.

1. Introduction

The socio-economic value of numerical weather prediction and climate modelling is driving the continuous development and improvement of atmospheric models on current and emerging supercomputing architectures [22, 33]. However, attaining high scalability and efficiency with atmospheric models is a challenging task for several reasons. The typical codebase of an atmospheric model has several hundred thousand to million lines of Fortran code. Atmospheric models are community codes with large developer communities, and even larger user communities. As a consequence, strong commitments towards a single source code, performance portability across different high performance computing architectures, and a high usability of the source code by non-expert programmers are required. The life cycle of an atmospheric model consists of a development phase of approximately 10 years and a deployment phase with continuous development of 20 years or more. Due to the non-linear nature of the Earth's atmosphere, the large range of spatial and temporal scales involved and the complexity of the underlying physical processes, assessing the correctness and predictive capabilities of an atmospheric model is in itself a scientifically and computationally arduous task. Furthermore, many atmospheric models are characterized by a low arithmetic density and limited scalability. The low arithmetic density results from the algorithmic motifs in the most time-consuming parts of the codes, typically finite difference, finite element or finite volume discretizations on a structured or unstructured grid. As a consequence of these motifs, scalability is limited by the total number of horizontal

[1]Federal Office of Meteorology and Climatology MeteoSwiss, Switzerland, oliver.fuhrer@meteoswiss.ch
[2]Center for Climate Systems Modeling C2SM, ETH Zurich, Switzerland
[3]Department of Computer Science, ETH Zurich, Switzerland
[4]Supercomputing Systems AG, Zurich, Switzerland
[5]Swiss National Supercomputing Centre, ETH Zurich, Switzerland
[6]Institute for Theoretical Physics, ETH Zurich, Switzerland
[7]Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA

grid points employed in a specific simulation, which can be low for multi-decadal global climate simulations.

Given these constraints, weather and climate codes are struggling to adapt to current and emerging hardware architectures. Adaptation of productive models to traditional multi-core systems typically use flat MPI parallelization, i.e. where each individual core is considered an independent compute node with no utilization of shared memory on a physical node of the computer system at the algorithmic level[8]. Attempts to migrate codes to a hybrid programming model such as MPI+OpenMP typically result in disappointing performance gains [6, 11, 26] or require substantial software refactoring that are not simpler than efforts to migrate models to hybrid architectures with GPU accelerators [19].

An extensive development effort based purely on Fortran plus compiler directives (OpenMP, F2C-ACC [13], and OpenACC) is currently ongoing for the global Non-hydrostatic Icosahedral Model (NIM) being developed at the National Oceanic and Atmospheric Administration (NOAA) [15, 17]. For inter-node communication NIM employs the Scalable Modeling System [14]. Using this approach, the dynamical core of NIM obtains good performance on multi-core CPUs, NVIDIA GPUs and Intel Xeon Phi accelerators. The team is currently working on porting the physics using the same approach.

A different approach was taken by Norman et al. [24] when porting the Fortran-based CAM-SE model to hybrid CPU-GPU architectures. CUDA Fortran is used for the parts of the model that run on GPUs, and the resulting port scaled and performed well on ORNL's Titan supercomputer (a Cray XK7 with Hybrid CPU-GPU nodes that are based on the NVIDIA Kepler architecture). The question of how such a port will migrate onto hybrid nodes with different accelerators that do not support the CUDA programming model is still to be addressed.

The Japanese Meteorological Agency is developing ASUCA, a new local-area numerical weather prediction model. In parallel, a group at Tokyo Institute of Technology developed another version of this Fortran/OpenMP code using C++/CUDA, which allowed them to run scale ASUCA up to 4000 GPUs on the TSUBAME 2.0 supercomputer [30, 31]. However, it is unclear how sustainable an approach that relies on different code bases for different architectures will be, and whether it will be supported by the climate and numerical weather prediction community.

Here we propose a different strategy: An existing monolithic Fortran code is partially rewritten based upon re-usable, domain or data structure specific tools that aim to separate concerns of domain decomposition for distributed memory and multi-threading on novel, heterogeneous node architectures. Our approach was applied to the regional weather and climate model COSMO, and the implementation is being fully integrated into the production trunk of the code. Our target is to run COSMO with a single code base at scale and in a performance portable manner both on CPU-based systems as well as different designs of hybrid CPU-GPU systems.

The problem of maintaining complex models on modern computing architectures is particularly acute in the context of the challenges posed by future exa-scale computing systems. With this paper we hope to make a positive contribution to this more general discussion over programming models. Using COSMO as a demonstration vehicle, we show how multiple architectures with their corresponding programming models can be supported with an implementation approach that shields the developers of the atmospheric model from the architectural complexities. The discussion is based on full production problems, rather than simplified benchmarks, in order

[8]The backend of an MPI implementation may still take advantage of a specific node architecture

to highlight all challenges of the refactoring effort. We demonstrate benchmarking results for various architectures that outperform the existing implementation of the model.

The paper is organized as follows. In Section 2 the COSMO model is briefly introduced. Section 3 will present the software strategy that underpins the implementation for different programming models with a single source code. In Section 4 performance results that affirm the different design choices, along with benchmarks of typical use-cases, are presented. Finally, we conclude with a summary and a discussion of future work in Section 5.

2. COSMO

The Consortium for Small-Scale Modeling (COSMO) is a consortium of seven European national weather services which aims to develop, improve and maintain a non-hydrostatic local area atmospheric model [9, 12, 32]. The COSMO model is used for both operational [4, 35] and research [20, 29] applications by the members of the consortium and many universities worldwide. Separate to the consortium, the CLM-Community [8] is an open international network of scientists are applying and developing the COSMO model for regional climate simulations. The current trunk version of the COSMO model is a Fortran code that relies on external libraries only for message passing (MPI) and I/O (grib, NetCDF). Being a community code used both for numerical weather prediction and climate simulation, COSMO is deployed on leadership class computing facilities at the largest compute centers, on medium sized research clusters at universities as well as on individual laptops for development.

The code base of COSMO can be divided into different components. The *dynamics* solves the governing fluid and thermodynamic equations on resolved scales. These partial differential equations (PDEs) are discretized on a three dimensional structured grid using finite difference methods. The so-called *physics* are modules representing sub-gridscale processes that cannot be resolved by the dynamics, such as cloud microphysics, turbulence and radiative transfer. For numerical weather prediction the incorporation of observational data – for example from ground measurement stations or weather radars – is handled by the *assimilation* code. Finally, the *diagnostics* compute derived quantities from the main prognostic variables and the *I/O* code handles input/output operations to files. A Runge-Kutta scheme is used to integrate the state variables forward in time [3], and Fig. 6 illustrates the order of operations on a time step. The COSMO model is a local area model, meaning that simulations are driven by external boundary conditions from a global model. The horizontal grid is a regular grid in a rotated latitude/longitude coordinate system. In this paper `i` and `j` refer to the indices enumerating horizontal longitude and latitude cells, and `k` refers to the vertical levels.

2.1. Benchmarks

Two model setups are used to generate the performance results presented in this paper. First, COSMO supports an initialization mode without requiring reading of external data from file by internally generating a model state and external boundary conditions from an analytical description of the atmosphere with random, three-dimensional perturbations. This configuration was used for the weak and strong scaling results presented in Section 4. This has the advantage that we can eliminate strong regional contrasts typically present in real data, which would inhibit a comparison between different domain sizes. Considerable care has been taken to ensure that the performance results with this model setup are representative for simulations with real data.

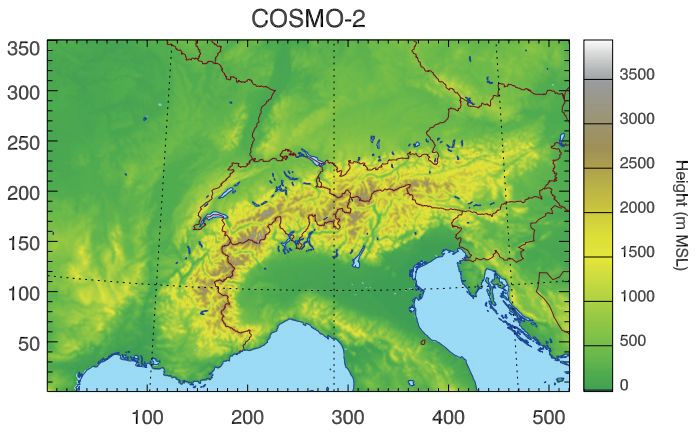

Figure 1. Computational domain of COSMO-2 benchmark. Contour shading shows model topography in meters above mean sea level (MSL)

Secondly, we use the 2.2 km operational setup of COSMO currently deployed at the Swiss national weather service MeteoSwiss. This setup, named COSMO-2, has a computational domain covering the greater Alpine region (see Fig. 1), with a total number of 520×350 horizontal gridpoints and 60 vertical levels. The COSMO-2 benchmark exercises the model in a representative way including all key physics modules. The COSMO-2 benchmark has to run with a time-compression factor of approximately 70 to meet the operational requirements of MeteoSwiss, meaning that the time-to-solution of a 24 h and 33 h simulation should correspond to 21 minutes and 29 minutes of wall-clock time, respectively. Such strict time-to-solution constraints are often the main driver choosing the size of computer systems used for operational weather forecasting.

3. Software Strategy

Like most weather and climate models, COSMO is implemented as a monolithic Fortran code that is highly optimized for the computer architectures in use at the weather services. At the time of writing these computer architectures are vector processors and distributed multi-core systems. The model is under continuous development, which means that there are many developers contributing code and who have a basic knowledge of the software architecture. Migrating such an implementation to new computer architectures that require different programming models is extraordinarily laborious. This was true in the past, when massively parallel computing systems appeared that have distributed memory with a message passing model. It continues to be a challenge today with multi-core processors that call for a threading model, as well as emerging architectures with hybrid CPU-GPU nodes.

In addition to software refactoring costs, the migration of a large weather and climate model like COSMO to new programming models requires a significant investment in training the developer community that has to adopt these models. Since the majority of this community are not computer scientists with neither extensive experience in software engineering or interests in new programming models, but domain scientists and applied mathematicians that have to be familiar with the underlying equations and their numerical discretizations, the adoption of new architectures can face significant resistance.

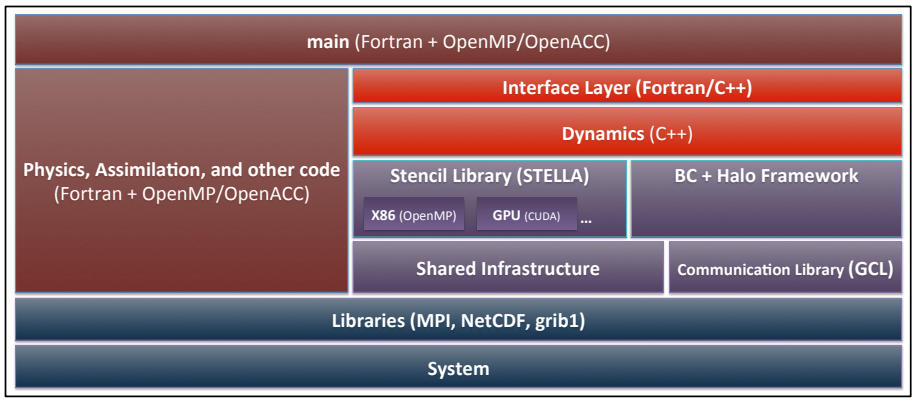

Figure 2. Software architecture of the new COSMO code. Red colors correspond to user code, purple colors are for re-usable middleware tools which have been developed, blue is for I/O and system libraries.

There is thus a natural restoring force in favor incremental over disruptive or revolutionary changes to computer architectures and programming models. This situation is common for high-performance computing in general, and has to be addressed, because rapid gains in performance and improvements in production costs can only be achieved if new architectures are adopted in a timely manner. For COSMO, the root cause of this restoring force is the present software model: the monolithic Fortran code that prohibits a separation of concerns between the code that implements the atmospheric model the computer architecture-specific implementation.

A central emphasis of our refactoring effort of COSMO was to introduce new, domain or data structure specific tools that allow hiding architectural details and the corresponding programming models from the user code that implements the model. In order to minimize the cost of adoption, these new tools had to be integrated with the original Fortran code base.

Figure 2 illustrates the software architecture of the refactored COSMO code. Only the dynamics, which comprises about 20% of the 250'000 line trunk of COSMO but requires approximately 60% of the runtime, was completely rewritten [16]. To this end two domain-specific tools were developed and used.

The first of these, named STELLA (Stencil Loop Language), is a domain-specific embedded languages (DSELs) that leverage C++ and CUDA. STELLA was designed to implement the different stencil motifs for structured grids that are used in COSMO. The user uses a simple abstraction to specify stencil operations, without having to be concerned by the programming model and hardware specific loop and implementation details, which are generated by the DSEL using template metaprogramming. A short description of STELLA is given in Section 3.1.

The second library provides abstractions for operations at domain and sub-domain boundaries, specifically boundary conditions and the exchange of halo grid points. This framework leverages the Generic Communication Library (GCL [5]), which is a generic C++ library that implements a flexible and performance portable interface to specify halo-exchange patterns for regular grids. A short description of the communication framework and GCL is given in Section 3.3.

For all remaining parts of the COSMO code (physics, assimilation, etc.) we have used a less disruptive porting strategy based on compiler directives [21]. In Section 3.2, we will briefly discuss the implementation of the physics code that is based on OpenACC directives allowing the code to run on GPUs. A similar extension in terms of OpenMP directives that supports

threading on a multi-core processors has been developed and will be provided in a future release of the code.

The choice to leave the physics, assimilation, and other parts of the code in their original Fortran form was a pragmatic one to minimize the impact of changes on the existing developer community. However, the elegant separation of architectural concerns from the atmospheric model specification is not possible with this approach, as compiler directives necessarily introduce details of the underlying programming model. The new architecture of the dynamics has the following benefits over this directives-based approach:

- Application developers (domain scientists) use an abstract hardware model (instead of a specific hardware architecture)
- Fine-grained parallelization at the node level is separated from course-grained data distribution across nodes.
- Separation of concerns, whereby we avoid exposing hardware specific constructs/optimizations to the user code.
- Single source code which can be maintained and developed by domain scientists.
- Re-usable software components (tools) which can be shared with other codes from the same domain, or other domains that use similar algorithmic motifs and data structures.
- Use of rigorous unit testing and regression testing that allows scalable development with large teams.

3.1. Dynamics rewrite (STELLA)

STELLA [16] is a DSEL for stencil codes on structured grids, that uses template metaprogramming [2] [1] to embed the domain-specific language within the C++ host language. The DSEL abstracts the stencil formulation from its architecture-specific implementation, so that users of the library write a single, performance-portable code. At compile-time the DSEL is translated into an executable with performance comparable to hand-written code optimized for the target architecture.

A typical stencil operator in COSMO concatenates multiple smaller stencil operators (so-called stencil stages). A stencil is made up of two logical components: 1) the loop-logic that iterates over the physical domain of the model represented by a structured grid; and 2) the loop body implementing the actual stencil update function. In STELLA an update function is implemented with a function object (functor), while the architecture specific loop-logic is defined using a DSEL that targets an abstract hardware model. Data locality is leveraged by fusing multiple stencil stages into a single kernel. Figure 3 depicts an example implemented using STELLA.

Table 1. Data layout of STELLA fields for the CPU and GPU backends

Backend	CPU	GPU
programming model	OpenMP	CUDA
storage order (by stride)	$j > i > k$	$k > j > i$

STELLA provides multiple backends for specific hardware architectures that utilize the corresponding programming model. Each backend employs its own data layout (see Table 1) and parallelization strategy that best maps onto the targeted architecture. At the time of writing,

```
// declarations
IJKRealField data;
Stencil horizontalDiffusion;

// declare stencil stage
template<typename TEnv>
struct Laplace {
  STENCIL_STAGE(TEnv)

  STAGE_PARAMETER(FullDomain, phi)
  STAGE_PARAMETER(FullDomain, lap)

  static void Do(Context ctx, FullDomain) {
  ctx[lap::Center()] = −4.0∗ctx[phi::Center()] +
      ctx[phi::At(iplus1)] + ctx[phi::At(iminus1)] +
      ctx[phi::At(jplus1)] + ctx[phi::At(jminus1)] ;
  }
};
```

```
//define and initialize the stencil
StencilCompiler::Build(
  horizontalDiffusion,
  // define the input/output parameters,
  pack_parameters(
    Param<res, cInOut>(dataOut), Param<phi, cIn)(data)
  ),
  define_temporaries(
    StencilBuffer<lap, double, KRange<FullDomain,0,0> >(),
  ),
  define_loops(
    define_sweep<cKIncrement>(
      define_stages(
        StencilStage<Laplace, IJRange<cIndented,−1,1,−1,1>,
          KRange<FullDomain,0,0> >(),
        StencilStage<Divergence, IJRange<cIndented,0,0,0,0>,
          KRange<FullDomain,0,0> >(),
      )
    )
  )
);

// execute the stencil instance
horizontalDiffusion.Apply();
```

Figure 3. Example of a STELLA stencil that implements a diffusion operator based on two stages: first a Laplace operator followed by a divergence operator. The left panel is a stencil stage for the Laplacian (note that the Divergence stage is not shown). The right panel shows the definition of the full stencil. It specifies the field parameters and temporary fields and assembles all the loops, defining its properties (ranges, ordering, etc.), by concatenating the two stages.

STELLA provides two backends, one based on OpenMP [34] for multi-core CPUs, and one based on CUDA [25] for GPUs (see Tab. 1).

The dynamics of COSMO was rewritten in C++ using the STELLA DSEL in order to provide a single source code that performs well both on CPU and GPU architectures. The dynamics is composed of ∼50 STELLA stencils of different size and shape. Additional C++ code organizes data fields in repositories and orchestrates the stencil calls along with calls to the halo exchange and boundary condition framework (see Section 3.3).

3.2. Fortran code refactoring (OpenACC)

The parts of the code that have been left in their original Fortran form were ported to GPUs with OpenACC compiler directives [27]. The OpenACC Application Programming Interface allows the programmer to specify, mostly at the loop level, which regions of a program should be run in parallel on the accelerator. The OpenACC programming model also provides control over data allocation and data movement between the host CPU and the accelerator. OpenACC is an open standard which is currently supported by three compiler vendors: Cray, PGI, and CAPS [7, 10, 28].

Figure 4 shows a code example which adds two two-dimensional arrays in Fortran. The compiler generates accelerator code for the region enclosed by the `parallel` and `end parallel` directives. The `data create` and `update host/device` directives control memory allocation on the device and data transfer between the CPU and the accelerator.

Atmospheric models such as COSMO have a relatively low arithmetic intensity, and the gain in performance obtained by running the computation on a GPU can be lost if data has to be moved back and forth between host and GPU memory. To avoid such data transfers wherever

```
!$acc update device(b,c)
!$acc parallel
do j = 1, NI
  do i = 1, NJ
    a(i,j) = b(i,j) + c(i,j)
  enddo
enddo
!$acc end parallel
!$acc update host(a)
!$acc end data
```

```
!$acc host_data use_device(a)
call c_wrapper(a)
!$acc end host_data
!$acc end data
```

Figure 4. Left: example of a two dimensional arrays addition in Fortran using OpenACC. Loops between the `parallel` and `end parallel` will be executed on parallel on the accelerator. Right: example using `host_data` directive to interface OpenACC and C/C++ code. The corresponding GPU address of variable `a` is passed to the routine c_wrapper.

possible all computations that require two-dimensional or three-dimensional fields are executed on the GPU.

The OpenACC approach is particularly convenient, as it allows porting large parts of the model in a relatively short time by simply adding the directives to the existing code. This naïve porting approach was used for large parts of the code which are not performance critical. For the performance critical parts, mostly in the physics, code refactoring was required to get acceptable performance with OpenACC. The main optimizations consist of restructuring loops in order to better expose parallelism and using larger kernels. A detailed description of the OpenACC porting and optimizations of the main components of the physics can be found in [21]. Some parts of the code which are not executed at every time step, such as I/O related computations, are still partially run on the CPU without degrading overall performance (see Figure 6).

3.3. Inter-node parallelization (GCL)

COSMO uses a two-dimensional domain decomposition in the horizontal plane to distribute data and work across nodes. The size and shape of overlap regions between domains (called halos) vary from field to field, as they are determined by the size of the stencils that require data from neighboring nodes. The current production version of COSMO uses MPI to exchange the data in the halo regions between nodes. Further methods are also needed in order to impose boundary conditions on the global boundary of the simulation domain.

The *halo update and boundary condition framework* combines these two operations (boundary conditions and halo exchanges) into a single framework. This allows the user to provide a unique description of these operations by defining the halo regions, fields and boundary conditions with one user interface. Just like the STELLA library, the framework abstracts implementation details of internode communication and application of boundary conditions. It supports the various memory layouts used in the backends for STELLA. For optimal performance, memory layout and boundary condition type are generated at compile time for a particular architecture. Furthermore, the framework provides Fortran bindings that allow it to be used from the Fortran side of the code as well. Figure 5 shows an example of the definition of a halo update and boundary condition object that can be executed any time a halo update is required by a stencil.

The data exchange in the halo updates is handled using the Generic Communication Library (GCL [5]). GCL is a C++ library with which users specify at compile time the data layout of the fields, the mapping of processes and data coordinates, data field value types, and methods for packing/unpacking halos. The library supports both host and accelerator memory spaces. If the

```
// define the ghost points that require updating
IJBoundary boundaryRegion;
boundaryRegion.Init(−2,2,−2,2);

// define a BC and halo−update job
HaloUpdateManager haloUpdate;
haloUpdate.AddJob(zeroGradientBCType, phi, boundaryRegion);

// apply a stencil to the data field phi
verticalDiffusion.Apply();

// start BC and halo−update job
haloUpdate.Start();

// do additional computation that does not modify data field "phi"

// complete BC and halo−update job
haloUpdate.Wait();

// apply stencil to the data field phi (consuming the halo points)
horizontalDiffusion.Apply();
```

Figure 5. Example of a configuration of a boundary condition and halo-update required for updating the halo points required by the diffusion operator introduced in Figure 3

underlying communication libraries are GPU-aware, the data exchange between accelerators will take advantage of direct GPU to GPU communication, bypassing the host memory. Presently GCL uses MPI, but other communication libraries for distributed memory architectures could be used.

3.4. Integration

As discussed above, a complete rewrite of the COSMO model was not an option. Thus the inter-operability of rewritten code parts or newly developed tools and the existing frameworks is of paramount importance. The integration of different programming models (OpenACC and STELLA) as well as flexible tools (STELLA and GCL libraries) with the legacy Fortran code framework of COSMO called for particular attention. The main challenge was that while the memory layout is fixed in the Fortran code, the memory layouts of STELLA fields are configurable and depend on the backend. Figure 6 shows a diagram of the COSMO time loop together with the porting approach for each part of the model.

In order to avoid expensive data transfers between the GPU and the CPU within each step of the time loop, all required data fields are allocated on the GPU at startup in the Fortran code using the OpenACC `data create` constructs during startup, where they remain for the entire run. The allocated fields are then reused in different parts of the Fortran code by using the `present` directive. Inter-operability with C++/CUDA can be achieved via the `host_data` OpenACC directive (see right panel of Figure 4) which allows access to the data field GPU pointers that can then be passed to a library function call.

An interface layer between the C++ and Fortran code was developed in C++ to provide Fortran bindings to the dynamics and boundary/halo framework. At every time step the C++ dynamics is invoked via the interface layer. The memory layout in the Fortran code is `ijk`, *i.e.* `i` is the stride-one dimension. The dynamics uses an `kij` storage order on the CPU for performance reasons (see Table 1). For the GPU backend the storage order is `ijk`, like the Fortran part, however the STELLA library can use aligned memory fields that offer better performance on the GPU. The interface layer supports two different modes: 1) shared fields, *i.e.* memory layouts are equivalent, the pointer allocated in the Fortran parts is directly reused in the dynamics, and

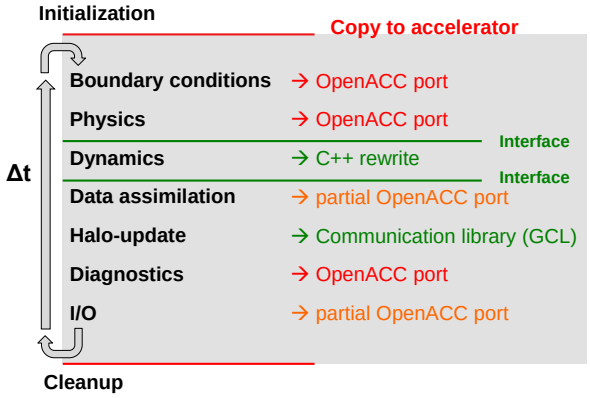

Figure 6. Workflow of the COSMO model on GPU. After the initialization phase on CPU all required data is transferred to the GPU

no data transformation or copies are required; 2) copy fields, *i.e.* memory layouts of fields are different (this includes different alignments), the call to the dynamics will trigger a copy (and transformation) of fields upon entry and exit.

Table 2 summarizes the effect of copying data between the Fortran and C++ sections in the GPU backend. In this example, using an optimally aligned data structure in the dynamics (copy fields mode) is on par with sharing fields with the Fortran code (shared fields mode). Previous generation GPUs had a more significant penalty for un-aligned memory accesses and using copy mode was typically advantageous. In any case, the specific choice of data storage layout can be chosen freely in the C++ part of the code and the interface layer automatically recognizes if copies are required or pointers can be shared.

The Fortran code uses a two time-level scheme whereby each variable has two fields, and at every time step the pointers for the fields are swapped in order to reuse the allocated memory. Similarly, STELLA uses double buffering to swap the pointers between time steps. An important function of the interface layer is to ensure consistency of all the field pointers between the Fortran parts and the dynamics, which is non trivial.

The interface layer also provides Fortran bindings to the boundary condition and halo exchanges framework (Section 3.3). This allows the Fortran parts of COSMO to apply halo exchanges and the boundary conditions strategies using a unique implementation which can deal with any underlying memory layout and architecture backend.

Finally, considering the important refactoring effort for COSMO, systematic unit testing and regression testing was introduced and has become an integral part of the software strategy. All tests are executed on a daily basis using a continuous integration tool [18].

4. Performance Results

In this section we compare the performance of the refactored code (referred to as *new*) with with the current COSMO production code (referred to as *current*). We will also present detailed experiments which demonstrate the performance impact of specific design choices outlined in the previous section.

Table 2. Performance comparison of the copy-field and the shared-fields modes for passing fields between Fortran and C++ via the interface layer on a K20X GPU. All measurements are time per time step. Shown is stencil computation time, time for copying fields in the interface layer and the total dynamics time.

	stencil time [ms]	copy time [ms]	total [ms]
shared-field mode	148	0.200	204
copy-field mode	142	6.38	206

Unless explicitly stated, performance results have been measured on a hybrid Cray XC30 named Piz Daint[9] at the Swiss National Supercomputing Centre (CSCS) in Lugano. This system has a hybrid node architecture with a Intel Xeon E5-2670 CPU with 32 GB host memory and a NVIDIA Tesla K20X GPU with 6 GB GDDR5 memory. The compute nodes are interconnected with Cray's Aries network that has a three-level dragonfly topology.

4.1. Weak and strong scaling

The left panel in Fig. 7 shows weak scaling results of the current and new code on Piz Daint. Two configurations with 128x128x60 and 64x64x60 gridpoints are shown by the solid and dashed lines (as well as the solid and empty markers, respectively), respectively. The colors correspond to the current code running purely on the CPUs (blue squares), the new code running purely on the CPUs (red circles) and the new code running with use of the GPU (orange triangles). For all cases, the 3D domain is decomposed only over the horizontal i, j-dimensions. For the CPU cases, the per-node sub-domain is further decomposed onto 8 MPI ranks that are placed on the individual cores (we don't use OpenMP threading or hyper-threading). When running in hybrid mode, only a single MPI rank is placed on a node (one per GPU). The runtime shown on the y-axis corresponds to a forward integration of 100 timesteps with I/O disabled. All curves are almost flat, which corresponds to an almost perfect linear scaling up to 1000 nodes for both per-node domain sizes. Focusing on the 128x128x60 case using 9 nodes, see also table 4, we observe a 1.5x speedup when running the new code (red solid circles) on the CPU compared to the current code (blue solid squares) for the 1.5x problem size. Moving the new code from the CPU (red solid circles) to the hybrid mode with GPUs (orange solid triangles) leads to an additional speedup of 2.9x.

Strong scaling performance with the total problem size fixed at 512x512x60 (solid line) and 256x256x60 (dashed lines) for an integration of 100 time steps is shown in the right panel of Fig. 7. Since we decompose the computational domain only along the horizontal i, j-dimensions, the number of horizontal grid points decreases with increasing node count, while the number of vertical levels remains constant at 60 levels. Analogous to the weak scaling experiment presented above, the pure CPU runs are done with 8 MPI ranks per socket, leading to a finer decomposition than for the GPU runs. While the CPU results for both the current and new code (blue squares and red circles) show very good strong scaling up to more than 100 nodes, the runs in hybrid mode (orange triangles) saturate at much lower nodes counts. The reason for the rapid saturation of the hybrid architecture is that a minimal number of horizontal grid points are required to efficiently use of the GPU that requires minimal number of concurrent threads to saturate the

[9] `http://user.cscs.ch/computing_resources/piz_daint/index.html`

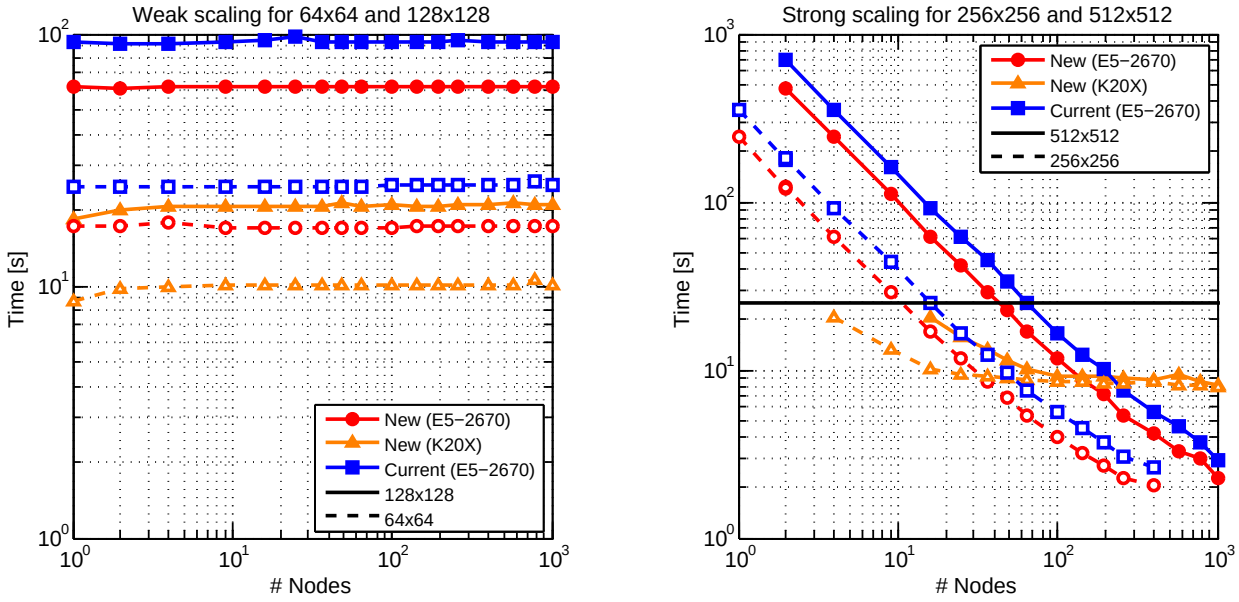

Figure 7. Left: weak scaling for two different per-node grid sizes (128x128 solid line, 64x64 dashed line). Right: strong scaling for two different total grid sizes (512x512 solid line, 256x256 dashed line). Both show the current trunk version of the code (red) and the refactored code running on CPU (blue) or GPU (green). Runtime is given in seconds and corresponds to a total of 100 timesteps. All simulations have been done with disabled I/O.

available memory bandwidth. The upper limit of the strong scaling curves is given by a minimum of 3x3 horizontal grid points per node which are required to run the COSMO model. The lower limit of the strong scaling curves is given by the available memory.

The horizontal solid black line delimits the time-to-solution requirement for the 2 km model of numerical weather prediction of MeteoSwiss. With this fixed time-to-solution requirement, one can map a given simulation to significantly less computational resources on the hybrid CPU-GPU architecture. For the 512x512x60 problem, the current code requires 64 CPU nodes (with one Intel Xeon E5-2670), while the new code requires 48 CPU nodes and only 14 hybrid nodes (with one Intel Xeon E5-2670 and a K20X GPU).

Table 3. Runtime with 128x128x60 grid points per node using 9 nodes, comparing the current code to the new code on CPU and GPU.

Code	Architecture	Time [s]	Speedup
Current	E5-2670	92.3	REF
New	E5-2670	61.6	1.5x
New	K20X	20.8	4.4x

4.2. Different node architectures

Simply put, the superior efficiency of the hybrid CPU-GPU nodes is due to the higher memory bandwidth of the GPU. This suggests that nodes with a different CPU to GPU ratios

might be more attractive. In this section, we investigate the performance of the COSMO-2 benchmark on two different node architectures. We compare runs on Piz Daint that use 8 nodes, each with one Intel Xeon E5-2670 and one NVIDIA K20X socket, to runs on a single "fat node" system named *Opcode* that has a 4:1 GPU to CPU ratio. Specifically, Opcode is a Tyan FT77A server with two Intel Xeon E5-2670 sockets, each connected via the PCIe Gen3 bus to two PLX8747 PCIe switches which each are connected to two NVIDIA Tesla K20 GPUs, i.e. 4 GPUs per Intel Xeon socket or a total of 8 Tesla K20 GPUs per node. Figure 8 gives an overview of the different node architectures.

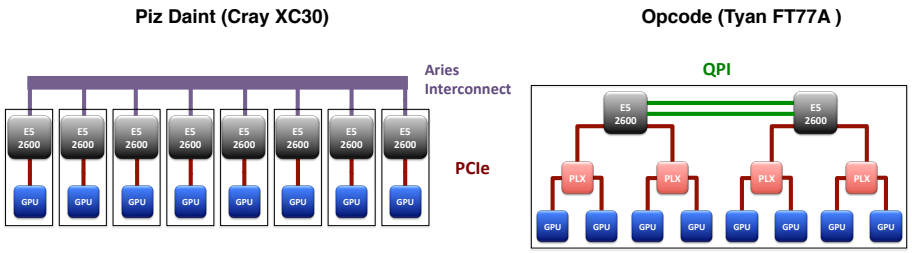

Figure 8. Comparison of the Piz Daint and Opcode systems used for the COSMO-2 benchmarks. Piz Daint nodes each have a CPU and GPU and are inter-connected via the Aries network. Opcode is a single node system with 2 CPUs and 8 GPUs, where 4 GPUs are connected to each CPU via PCIe.

Table 4 shows the runtime of the Fortran, C++ and communication components individually as well as the total runtime of the COSMO-2 benchmark. The simulation has been integrated up to +33 simulation hours. Note that the benchmark has a required time-to-solution of 29 minutes (1740 s), which on Piz Daint is almost achieved for this run using only 8 nodes.

For the dynamics, we would not expect large performance differences since STELLA uses the same backend and compiler for both architectures. The results indicate that the dynamics runs 10% slower on the Opcode system as compared to Piz Daint. This can be explained by the slightly lower peak memory bandwidth and floating-point performance of the K20 cards as compared to the K20X on Piz Daint.

On the other hand, the Fortran code which has been ported using OpenACC directives is 53% slower on the Opcode system. The rather large difference is due to the different Fortran compilers (Cray's in the case of Piz Daint and PGI on Opcode). As OpenACC compilers mature, it is to be expected that these large performance differences will decrease.

Table 4. COSMO-2 benchmark on different hybrid node architectures. The simulation time in [s] is given for a 33h forecast. The COSMO-2 computational domain (520x350) is decomposed to 2x4 GPUs on both platforms. On Piz Daint all the 8 GPUs are on different nodes, while on Opcode all the GPUs are on the same node

	Computation [s]		Communication [s]	Total [s]
	Dynamics (Stella)	Fortran (OpenACC)		
Piz Daint (K20X)	890	574	426	1889
Opcode (K20)	975	878	431	2283

In spite of very different interconnect hardware between the GPUs, the inter-GPU communication times are almost on par. On the Opcode system, communication is over the PCIe

bus leveraging GPUDirect [23] or - in case two communicating GPUs are attached to different sockets - over the Intel QuickPath interconnect (QPI). Despite performing inter-GPU communication over the non-dedicated PCIe bus or the QPI, results for the Opcode system show a total communication time comparable to Piz Daint, where GPUs communicate with the network interface controller using GPUDirect over the PCIe, and different nodes are interconnected with the Cray's Aries network.

Finally, it should be noted that for running the refactored version of COSMO a system with fat-nodes is considerably more cost-effective and energy efficient. Furthermore, the refactored code is found to be performance portable to different node architectures.

4.3. GPU-to-GPU vs. communication over host

As has been mentioned in the previous section, GPUDirect allows communication between GPUs either on the same node or on different nodes without requiring copies to the CPU. Several MPI libraries leverage GPUDirect to enable faster GPU to GPU communication (G2G). G2G-enabled MPI libraries allow directly passing in GPU memory addresses to a restricted set of the library functions (e.g. MPI_Send/MPI_Recv). In this section, we compare the time of individual components of the halo-update on Piz Daint for different setups of the COSMO-2 benchmark.

Table 5. Shown are networking plus synchronization time, time for packing the data into messages, and time for copying message buffers to/from the CPU for different configurations of the dynamics of the COSMO-2 benchmark. The first two data columns show results for runs using the hybrid version of the code, with and without G2G (see text). The last column are measurements gathered for a CPU version of the code with exactly the same domain decomposition and setup. All values are in seconds.

Part	**GPU (with G2G)**	**GPU (without G2G)**	**CPU**
Network + Sync [s]	280.5	245.3	195.6
Packing [s]	33.2	33.5	52.9
Copy [s]	–	89.6	–
Total [s]	315.5	368.4	248.5

Table 5 shows time spent in the different parts of the halo-updates contained in the dynamics for a 33 h integration of the COSMO-2 benchmark. Over the whole simulation, a total of 7.19 GB of data is exchanged by each of the 8 MPI ranks used for the benchmark. For every halo-update, first the halos of the data fields have to be packed into linear buffers (Packing) the halos of the fields are packed into linear buffers. For runs using the hybrid code with G2G disabled, these linear buffers have to be copied to the CPU (Copy). Finally, the communication can be started by passing the data to the MPI library and waiting for the communication to finish (Network + Sync). There is a considerable load imbalance due to different computational loads for the MPI ranks which are situated at the corners of the computational domain, and the synchronization time is not negligible.

First, one can see that the packing time is lower on the GPU as compared to packing on the CPU, which can be explained by the higher memory bandwidth of the GPU. The large copy time required when disabling G2G indicates that it is worth avoiding these copies if a MPI library with G2G support is available. Finally, one can observe that the network plus synchronization

time is considerably smaller for the CPU runs. Further investigations indicate that this is mostly due to a larger sensitivity of the GPU communication to load imbalance.

5. Summary and future work

We have discussed a reimplementation of the COSMO numerical weather prediction and regional climate model that is performance portable across multiple hardware architectures such as distributed multi-core and hybrid CPU-GPU systems. The dynamics has been rewritten entirely in C++. It is implemented using a domain-specific embedded language named STELLA [16] which is targeted for stencil computations on structured grids. STELLA is implemented using template metaprogramming and supports a multi-threaded model with fine grained parallelism on a variety of node architectures supported through architecture specific backends to STELLA The dynamics is also built on top of a generic halo exchange and boundary condition framework that provides the necessary functionality for domain decomposition on distributed memory systems. The data exchange between nodes is implemented in terms of a Generic Communication Library (GCL [16]). With this strategy, the domain scientists develop methods against an abstract machine model, rather than architecture specific programming models, allowing full flexibility in the choice of underlying architectures and programming models. For practical reasons concerning the user community, the physics and other parts of the COSMO code were left in Fortran and ported to hybrid CPU-GPU platforms with OpenACC directives. The performance results we presented show the refactored code to outperform the current production code by a factor 1.5 for full production problems. Scalability and performance of the new code on various architectures corresponds to expectations based on algorithmic motifs of the code and hardware characteristics. This demonstrates that an innovative, tools-based approach to software can exploit the full performance of a system just as well as a code that has been implemented in Fortran or C.

The reimplementation of COSMO started in Autumn 2010 after a brief feasibility study and negotiations with the community. Design and implementation of STELLA as well as the rewrite of the dynamics for a single node took about two years with two staff. Refactoring the physics parts took about three man-years, and the communication and boundary condition framework as well as the GCL took another two man- years. The overall integration and testing with the legacy Fortran code started immediately after completion of the first version of the rewritten dynamics and communication framework. It took two years and a total of three man-years. Intensive co-development of the G2G communication libraries at Cray and the GCL at CSCS during the last 9 months of the project led to the successful deployment of Piz Daint in Fall 2013 with the refactored COSMO code being one of the first applications to run at scale on the system after acceptance in January 2014.

Thus, the three-year reimplementation project of COSMO took about as long as the planing and procurement of a new supercomputer. It seems important that the innovation of the software is kept in sync with the hardware cycle. More important for future development, however, is the strategy we employed: breaking up a monolithic Fortran code into tools that are reusable in other domains. Libraries like STELLA, GCL, as well as the halo update and boundary condition framework can be readily adapted to new emerging hardware architectures. Furthermore, with these tools it will be easier to implement new dynamical cores, allowing faster innovation on the numerical methods and the models as well.

A new project is under way to extend STELLA to types of block-structured grids used in global models, such as icosahedral and cubed-sphere meshes. The goal of this project is to provide performance portable tools to implement global weather, climate and seismic models. Furthermore, the tools discussed here will be extended with Python bindings, in order to support future model development directly within a high-level and object-oriented language environment. The aim of such projects will be to cut down on the integration cost, which in the case of the COSMO reimplementation project were considerable. We propose that future implementations of complex models such as COSMO should be based on performance portable tools and descriptive high-productivity languages like Python that allow for more dynamic development, rather than monolithic codes implemented in prescriptive languages such as Fortran or C/C++. This will make model development much more efficient and facilitate the co-design of tools and hardware architectures that are much more performant and efficient than today's code. Furthermore, it will simplify the introduction of new programming models and languages, as their use can be limited to the appropriate part of the toolchain and there will be no immediate need to change the entire code base of a model.

Several people contributed to the new version COSMO and we would like to acknowledge them here: Tiziano Diamanti, Florian Dörfler, David Leutwyler, Peter Messmer, Katharina Riedinger, Anne Roches, Stefan Rüdisühli, and Sander Schaffner. Also, the authors would like to acknowledge the Center for Climate Systems Modeling (C2SM) and specifically Isabelle Bey for administrative support. We thank the Swiss National Supercomputing Centre (CSCS) for providing access to the compute resources used to carry out the benchmarks. Finally, we would like to acknowledge the High Performance and High Productivity (HP2C) Initiative for funding this research.

References

1. I. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond*. The C++ in-Depth Series. Addison Wesley Professional, 2005.

2. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

3. M. Baldauf. Stability analysis for linear discretisations of the advection equation with runge–kutta time integration. *Journal of Computational Physics*, 227(13):6638 – 6659, 2008.

4. M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer. Operational convective-scale numerical weather prediction with the cosmo model: Description and sensitivities. *Monthly Weather Review*, 139:3387–3905, 2011.

5. M. Bianco. An interface for halo exchange pattern, 2012.

6. F. Cappelo and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proceedings of 2000 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'00. ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2000.

7. CAPS. *CAPS the many core company*, 2014. `http://www.caps-entreprise.com/`.

8. Climate Limited-area Modelling Community. http://www.clm-community.eu/.

9. Consortium for Small-Scale Modeling. http://www.cosmo-model.org/.

10. Cray Inc. *Cray Fortran Reference Manual*, 2014. http://docs.cray.com/books/S-3901-82/S-3901-82.pdf.

11. M. J. Djomehri and H. H. Jin. Hybrid MPI+OpenMP programming of an overset CFD solver and performance investigations. NASA/NAS Technical Report NAS-02-002, NASA Ames Research Center, 2002.

12. G. Doms and U. Schättler. The nonhydrostatic limited-area model LM (lokal-model) of the DWD. Part I: Scientific documentation. Technical report, German Weather Service (DWD), Offenbach, Germany, 1999.

13. M. Govett. *F2C-ACC Users Guide, Version 4.2*, 2012. http://www.esrl.noaa.gov/gsd/ab/ac/Accelerators.html.

14. M. Govett, L. Hart, T. Henderson, J. Middlecoff, and D. Schaffer. The scalable modeling system: directive-based code parallelization for distributed and shared memory computers. *Parallel Computing*, 29(8):995–1020, 2003.

15. M. Govett, J. Middlecoff, and T. Henderson. Running the NIM next-generation weather model on gpus. In *Proceedings 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 792–796, 2010.

16. T. Gysi, O. Fuhrer, C. Osuna, M. Bianco, and T. Schulthess. Stella: A domain-specific language and tool for structured grid methods. *submitted*.

17. T. Henderson, J. Middlecoff, J. Rosinski, M. Govett, and P. Madden. Experience applying fortran GPU compilers to numerical weather prediction. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC 2011)*, pages 34–41, 2011.

18. Jenkins CI. *Jenkins the continuous integration tool*, 2014. http://jenkins-ci.org/.

19. H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, 1999.

20. W. Langhans, J. Schmidli, O. Fuhrer, S. Bieri, and C. Schär. Long-term simulations of thermally driven flows and orographic convection at convection-parameterizing and cloud-resolving resolutions. *Journal of Applied Meteorology and Climatology*, 52:1490–1510, 2013.

21. X. Lapillonne and O. Fuhrer. Using compiler directives to port large scientific applications to GPUs: An example from atmospheric science. *Parallel Processing Letters*, 24(1):1450003, 2014.

22. J. K. Lazo, J. S. Rice, and M. L. Hagenstad. Benefits of investing in a supercomputer to support weather forecasting research: An example of benefit cost analysis. *Yuejiang Academic Journal*, 1:1–22, 2010.

23. Mellanox. Nvidia gpudirect technology—accelerating gpu-based systems. http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf.

24. M. Norman, J. Larkin, R. Archibald, V. Anantharaj, I. Carpenter, P. Micikevicius, and K. Evans. Porting the spectral element community atmosphere model (CAM-SE) to hybrid gpu platforms. In *2012 SC COMPANION: HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SCC)*, pages 1788–1798, Salt Lake City, UT, 2012.

25. NVIDIA. *CUDA Parallel Computing Platform*. `https://developer.nvidia.com/cuda`.

26. L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

27. OpenACC Corporation. *The OpenACC Application Programing Interface*, 2011. `http://www.openacc.org/`.

28. Portland Group Inc. *PGI Compiler Reference Manual*, 2014. `http://www.pgroup.com/doc/pgiref.pdf`.

29. A. Possner, E. Zubler, O. Fuhrer, U. Lohmann, and C. Schär. A case study in modeling low-lying inversions and stratocumulus cloud cover in the bay of biscay. *Weather and Forecasting*, 29(2):289–304, 2014/06/01 2014.

30. T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Mtiroi. 145 tflops performance on 3990 gpus of tsubame 2.0 supercomputer for an operational weather prediction. In *Proc. Int. Conf. Comp. Sci.*, volume 4 of *Procedia Computer Science*, pages 1535–1544, 2011.

31. T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

32. J. Steppeler, G. Doms, U. Schättler, H. Bitzer, A. Gassmann, U. Damrath, and G. Gregoric. Meso gamma scale forecasts using the nonhydrostatic model LM. *Meteor. Atmos. Phys.*, 82, 2002.

33. N. Stern. Review on the economics of climate change. Technical report, HM Treasury, London, UK, 2006. `http://www.hm-treasury.gov.uk/stern_review_report.htm`.

34. The OpenMP ARB. *The OpenMP API Specification for Parallel Programming*, 2013. `http://www.openmp.org`.

35. T. Weusthoff, F. Ament, M. Arpagaus, and M. W. Rotach. Assessing the benefits of convection-permitting models by neighborhood verification: Examples from map d-phase. *Monthly Weather Review*, 138:3418–3433, 2010.

Received June 6, 2014.