# Porting the COSMO Weather Model to Manycore CPUs

Felix Thaler
Swiss National Supercomputing Centre, CSCS
Zurich, Switzerland
felix.thaler@cscs.ch

Stefan Moosbrugger
Federal Institute of Meteorology and Climatology, MeteoSwiss
Zurich, Switzerland

Carlos Osuna
Federal Institute of Meteorology and Climatology, MeteoSwiss
Zurich, Switzerland
carlos.osuna@meteoswiss.ch

Mauro Bianco
Swiss National Supercomputing Centre, CSCS
Lugano, Switzerland
mauro.bianco@cscs.ch

Hannes Vogt
Swiss National Supercomputing Centre, CSCS
Zurich, Switzerland
hannes.vogt@cscs.ch

Anton Afanasyev
Swiss National Supercomputing Centre, CSCS
Zurich, Switzerland
anton.afanasyev@cscs.ch

Lukas Mosimann
Swiss National Supercomputing Centre, CSCS
Zurich, Switzerland
lukas.mosimann@cscs.ch

Oliver Fuhrer
Federal Institute of Meteorology and Climatology, MeteoSwiss
Zurich, Switzerland
oliver.fuhrer@meteoswiss.ch

Thomas C. Schulthess
Swiss National Supercomputing Centre, CSCS
Zurich, Switzerland
schultho@ethz.ch

Torsten Hoefler
Scalable Parallel Computing Lab, ETH Zurich
Zurich, Switzerland
torsten.hoefler@inf.ethz.ch

## ABSTRACT

Weather and climate simulations are a major application driver in high-performance computing (HPC). With the end of Dennard scaling and Moore's law, the HPC industry increasingly employs specialized computation accelerators to increase computational throughput. Manycore architectures, such as Intel's Knights Landing (KNL), are a representative example of future processing devices. However, software has to be modified to use these devices efficiently. In this work, we demonstrate how an existing domain-specific language that has been designed for CPUs and GPUs can be extended to Manycore architectures such as KNL. We achieve comparable performance to the NVIDIA Tesla P100 GPU architecture on hand-tuned representative stencils of the dynamical core of the COSMO weather model and its radiation code. Further, we present performance within a factor of two of the P100 of the full DSL-based GPU-optimized COSMO dycore code. We find that optimizing code to full performance on modern manycore architectures requires similar effort and hardware knowledge as for GPUs. Further, we show limitations of the present approaches, and outline our lessons learned and possible principles for design of future DSLs for accelerators in the weather and climate domain.

## CCS CONCEPTS

• **Applied computing** → **Earth and atmospheric sciences**; • **Software and its engineering** → *Domain specific languages*; • **Computing methodologies** → Massively parallel and high-performance simulations.

## KEYWORDS

COSMO, KNL, Supercomputing, Weather Forecasting, Domain-Specific Languanges

**ACM Reference Format:**
Felix Thaler, Stefan Moosbrugger, Carlos Osuna, Mauro Bianco, Hannes Vogt, Anton Afanasyev, Lukas Mosimann, Oliver Fuhrer, Thomas C. Schulthess, and Torsten Hoefler. 2019. Porting the COSMO Weather Model to Manycore CPUs. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19), June 12–14, 2019, Zurich, Switzerland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3324989.3325723


*PASC '19, June 12–14, 2019, Zurich, Switzerland*

ACM ISBN 978-1-4503-6770-7/19/06…$15.00
https://doi.org/10.1145/3324989.3325723

## 1 INTRODUCTION

Weather and climate simulations have been a major user of high-performance computing systems for decades. Increased model resolution is a key factor to improved predictions of climate change and global high-resolution simulations will require a new class of supercomputers and software to be run in a reasonable time frame [6, 20]. Classically implemented simulation software — that is, hand-tuned architecture-specific code — is rapidly falling behind the quickly

evolving hardware architectures and compute accelerators. In the first ten ranks of the TOP500 list of November 2018, host processors and accelerators of no less than five different vendors are found [22]. A possible approach to making HPC software more future-proof is to use domain-specific languages (DSLs) or libraries, that allow to compile the same user source code to optimized architecture-dependent binary code This should significantly reduce the effort required to move existing software to new hardware platforms, as no code changes are required from the DSL user side. Further, traditional approaches often lead to largely separate code bases for each supported hardware architecture and thus maintainability effort increases linearly with the number of supported architectures. The STELLA-library [10] is an example of a C++-embedded DSL developed to run the dynamical core of the weather model COSMO of the Consortium for Small-scale Modeling [3, 4] on CPUs and GPU accelerators. It is used to run the operational forecasts of the Swiss Federal Office of Meteorology and Climatology (MeteoSwiss) on an accelerated GPU system since 2016 [7].

GridTools is a follow-up project based on the success of STELLA. It is a further step towards exascale computing for the weather and climate domain and the foundation of the present work. Among other libraries, GridTools includes a C++-embedded DSL similar to STELLA, currently supporting CPUs and NVIDIA GPUs. To expand the number of supported hardware types and gain experience with additional modern high-bandwidth manycore architectures, we introduce a new back-end to GridTools, optimized for the Intel's Knights Landing (KNL) and future manycore architectures. To evaluate the real-world performance portability and user experience, we tested the back-end on the newly developed GridTools-based dynamical core of the COSMO weather model.

Further, to get a more complete picture of the KNL performance characteristics when running COSMO, one of the most relevant and computationally expensive physical parameterizations of the model, i.e. radiation, was ported to and optimized for KNL using OpenMP compiler directives.

Despite the announced discontinuation of the Intel Xeon Phi product line [13], this work will be relevant for upcoming many-core architectures such as Intel's next generation CPUs that base on similar principles. Further, we could already observe large speedups compared to the old CPU implementation on modern general-purpose architectures like Intel Skylake with 512 bit vector registers.

## 1.1 Related Work

The literature on accelerating high-performance computations is vast. Thus, we focus on projects in the context of accelerating weather and climate applications. Due to the relatively short lifetime of the KNL line, there are few examples in the literature that demonstrate a full port of a weather model to Intel many-core architectures. Pioneering work was done for individual components of the WRF model. Specific optimizations and performance evaluations on many-core have been reported in [15] for the RRTMGP radiative transfer and for WSM6 [17] and Goddard [16] microphysics scheme.

The only report on a full weather model ported to KNL known to the authors is [8] for the NIM model. The model was accelerated using OpenACC for NVIDIA GPUs and OpenMP for KNL. The performance evaluation on the full model show speedup of up to 2x for KNL compared to the CPU baseline, and 0.8x with respect to the P100 NVIDIA GPU.

Further, some work was done on the MPDATA advection scheme [23]. The authors report a speedup of 2x as compared to a reference multi-core CPU version.

Even if portability of a weather model retaining a single source code is achievable using a combination of OpenMP + OpenACC + MPI, successful implementations delivering performance portability with directives across multiple architectures are rarely found in the literature. In order to address the performance portability problem, there are numerous approaches in the literature that provide programming models that hide implementation details and hardware dependent optimizations using higher level abstractions. An example is the GridTools library (see Section 1.2). Here, we briefly summarize other approaches. The Kokkos library [5] provides C++ constructs for abstracting data layouts and parallel loops to generate efficient implementations for multiple architectures. It is more general and low level than GridTools, which provides higher abstractions specific for weather codes as well as problem-specific optimizations. The CLAW compiler [2] is a Fortran DSL for column based weather code computations like those of the physical parameterizations. Parallel loops and promotions of column data is automatically performed by the compiler. A similar approach, the PSyclone DSL [1] provides a Fortran DSL for finite elements/finite differences dynamical cores. A different approach is the Polly-ACC compiler framework [9] which automatically maps a sequential code into an accelerated code using the llvm tools, and avoids the use of specific programming models. We currently investigate the applicability of this approach to the COSMO code base.

## 1.2 GridTools

GridTools is a set of software libraries for weather and climate applications. It is a joint development effort of the Swiss National Supercomputing Centre (CSCS) and MeteoSwiss. GridTools provides an embedded domain-specific language (EDSL) written in C++, designed for stencil computations. This EDSL can be seen as the successor of the Stencil Loop Language (STELLA).

When we refer to GridTools in this paper, we consider the stencil composition EDSL only. This template-based C++ library allows to define stencils as a composition of several smaller stencil operators. It allows to run the same user code without any changes on various hardware architectures. The desired back-end can be chosen at compile time, currently available are a CPU back-end, a CUDA-based NVIDIA GPU back-end and now a KNL or more generally a manycore processor back-end. GridTools takes care of managing temporary data fields that are passed from operator to operator in a composed stencil. The user additionally has the possibility to activate various per-stencil optimizations at compile time, for example caching of temporary fields, that is highly dependent on the stencil-specific data access pattern.

## 1.3 The COSMO Weather Model

The COSMO model is a non-hydrostatic limited area atmospheric model implemented using finite difference methods in Fortran 90.

Like other weather and climate models, it is divided into three main parts: a dynamical core, a set of physical parameterizations and data assimilation. The dynamical core solves the Euler equations on a curvilinear grid, using a split-explicit time integration scheme [25] with a multistage Runge-Kutta method for integration of slow processes. Due to the finer grid spacing in the vertical dimension, the dynamical core employs implicit discretizations in the vertical dimension and explicit discretizations in the horizontal. This choice leads to three dominant computational patterns: horizontal stencils, tridiagonal solvers along the vertical dimension (with no horizontal data dependencies) and point-wise computations. Figure 1 shows an overview of the code flow in COSMO.

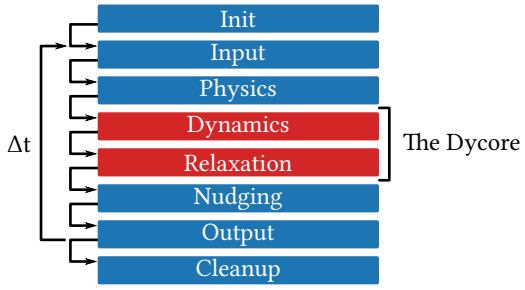


**Figure 1: COSMO model code flow. The computations inside the dynamical core are the most expensive part of a time step, roughly accounting for 60% of the run time. Graphics based on [21].**

The physical parameterizations solve the processes that are not resolved within the scale of the grid or source and sink terms that are not represented by the equations of motion, like the radiation. All physical parameterizations of COSMO contain column based computational patterns, where computations exhibit only vertical data dependencies.

A GPU capable version of the model was developed some years ago and has been used operationally on a supercomputer based on NVIDIA K80 GPUs at MeteoSwiss since 2016 [7, 10]. The model was enabled on GPUs based on several technologies and programming models used to retain a single source code that provides a significant performance improvement with respect to the CPU baseline. The dynamical core was ported to GPU using the STELLA C++ embedded DSL, which provides optimized hardware-dependent codes for the dynamical core for x86 and GPU architectures [7]. Recent work ported the complete dynamical core using the more general GridTools library (see Section 1.2), which allows extensions to other architectures by incorporation of new back-end specific templates for kernel computations. This allowed to evaluate the performance of the dynamical core on KNL by means of a newly developed GridTools back-end presented here. The physical parameterizations have been fully ported to GPU using OpenACC, retaining Fortran as the programming language for the model. The GPU implementation will also serve as a reference for comparison of the KNL performance results.

## 1.4 Intel Xeon Phi "Knights Landing"

The Intel Xeon Phi codenamed Knights Landing (KNL) is a x86-based manycore processor architecture with up to 72 cores connected by an on-chip mesh network, High Bandwidth Memory and several boot-time memory configuration options [14, 18]. The Knights Landing architecture is mainly available as a standalone CPU — unlike the predecessor Knights Corner that was only available as a PCIe accelerator. An accelerator version of the KNL architecture was canceled before release to the public market. In this work we always refer to the standalone CPU version.

The KNL is organized in 38 tiles, each hosting two x86-compliant cores with up to four hyper threads. Each core features a private 32 KB L1 data cache, 32 KB L1 instruction cache and two vector-processing units with 512 bit vector registers and support the AVX-512F instruction set. This includes exponential and reciprocal instructions, gather and scatter instructions with masking support, and software prefetching instructions.

The tiles are connected by a 2D mesh of rows and columns. The two cores on each tile share a 1 MB L2 cache. Cache coherency across the tiles is guaranteed by a distributed tag directory using the MESIF protocol. Memory IO controllers for the high-bandwidth MCDRAM and lower bandwidth DDR4 memory also reside in the mesh. Messages on the mesh can be sent along rows and columns, which should allow for efficient communication between all tiles. Out of the 38 tiles, only 32, 34 or 36 are activated, depending on the exact processor model. See Figure 2 for a graphical depiction of the architecture.

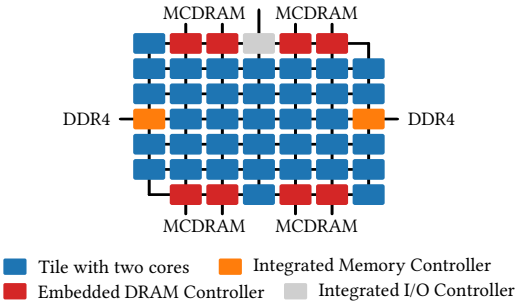


**Figure 2: Intel Knights Landing architecture. Graphics based on [12].**

Similar to GPUs, the KNL features High Bandwidth Memory, namely 16 GB of multi-channel DRAM (MCDRAM). The peak bandwidth of the MCDRAM is more than 400 GB/s, values depend on the exact processor model. Additionally, up to 384 GB of DDR4 memory with a bandwidth of close to 100 GB/s are available. The two memory types can be configured at boot time to be either in *flat* mode, *cache* mode, or *hybrid* mode. In *flat* mode, the MCDRAM and DDR memory are exposed to the OS as two different NUMA domains, and thus it allows to explicitly allocate data either in MCDRAM or DDR memory. In *cache* mode, the MCDRAM acts as a direct-mapped hardware cache for the DDR memory. This is especially useful for running software that requires a working set of more than 16 GB but still can profit from the higher MCDRAM bandwidth. The *hybrid* mode allows to use either 4 GB or 8 GB of

the MCDRAM in cache mode and the remaining 12 GB or 8 GB in flat mode.

An additional configuration capability of the KNL are the *cluster modes*. The cluster modes change the way, how cache lines are assigned to the distributed tag directories spread over the tiles. There are the five modes *All-to-all*, *Quadrant*, *Hemisphere*, *Sub-NUMA Clustering (SNC) 2*, and *SNC 4* available. The first three expose the memory as single NUMA domain per memory type, the *SNC* modes divide the mesh into 2 respectively 4 NUMA domains. The different cluster modes can influence application performance significantly and the *SNC* modes require NUMA-aware memory handling. We refer to Ramos and Hoefler [18] for a more complete description and very detailed performance evaluation including complete memory and cache latency, and bandwidth numbers.

### 1.5 Experimental Setup

Experimental results are collected on the CSCS Cray XC40 KNL system *Grand Tavé* featuring 164 Intel KNL compute nodes. Each node contains an Intel Xeon Phi CPU 7230 processor with 64 cores running at 1.30 GHz. As the working data set of the COSMO simulations performed in production at MeteoSwiss fits into 16 GB and the stencil operations of the COSMO dynamical core are mostly memory bound, we used the KNL in *flat* memory configuration with all allocations performed on MCDRAM for the present work. Further, due to simplicity and good overall performance reported in [18], we use the *Quadrant* cluster mode.

For performance comparison, results collected on the CSCS flagship machine *Piz Daint* are also presented. The Piz Daint GPU partition is a Cray XC50 system and consists of 5704 nodes with NVIDIA Tesla P100 GPUs featuring 16 GB of High Bandwidth Memory.

Additionally, some benchmarks were run on Intel Xeon Gold 6130 CPU. This 16-core chip based on the Intel Skylake general purpose CPU architecture features AVX-512, similar to the KNL. Nevertheless, worse performance for stencil codes is expected because of the smaller peak memory bandwidth due to the lack of High Bandwidth Memory.

Unless otherwise stated, we used the Intel C++ Compiler version 18.0.2 for compilation, together with the Intel OpenMP implementation.

All measurements on the benchmark stencils were taken on 100 consecutive runs. On the NVIDIA P100, the GPU frequency was fixed to the highest possible value (1328 MHz). Note that we used CUDA managed memory. The required data was always prefetched to GPU memory before stencil runs. On the KNL, caches were flushed between all runs. For both platforms, time was measured using standard C++ facilities. That is, a synchronization (*cudaDeviceSynchronize()*) was placed after each GPU kernel call as we wanted to measure the execution time of single kernel executions for run time variance estimation. We verified that the synchronization overhead is small compared to the kernel run time.

The data shown in the plots are the median results, error bars indicate the 90% confidence intervals. Note that in general the KNL shows larger variability in the performance than the P100. Most of the time the performance is stable (that is, the top results and median are close together in the plots), but some outliers exist depending on number of hyper threads and the data access pattern (caches). The darker grey areas in the plots highlight the relevant domain sizes for the production weather forecasts at MeteoSwiss.

## 2 PERFORMANCE EVALUATION ON REPRESENTATIVE STENCILS

### 2.1 Methodology

To understand the complex behavior of the Knights Landing architecture in context of stencil operations, we implemented two stencils of the COSMO weather model in hand-optimized C++ code. We chose one horizontal and one vertical stencil, representing the typical code patterns found in the dycore. The horizontal stencil pattern has data dependencies only in a compact *horizontal* neighborhood of the mesh and is embarrassingly parallel along all three axes. Vertical stencils are not stencils in the classical sense, because they carry dependencies in the vertical direction. They can only be parallelized along the horizontal dimensions. An example are tridiagonal linear system solvers [24] that occur due to the implicit discretization along the vertical.

A fourth-order horizontal diffusion operator and a vertical advection solver of COSMO were chosen as representative examples of horizontal and vertical computations respectively.

(1)–(4) show the equations of the horizontal diffusion stencil. $\phi^n_{i,j}$ refers to the input data field at (sub-)time step $n$ and spatial point $i, j$. $\phi^{n+1}_{i,j}$ is the diffused output at step $n+1$. $L^n_{i,j}$ is the discretized Laplacian of $\phi^n_{i,j}$ as defined in (1). $F^n_{i+1/2,j}$ and $G^n_{i,j+1/2}$ defined in (2) and (3) are limited fluxes along horizontal axes (note the staggered placement of the fluxes). (4) defines the actual time stepping, where $D^n_{i,j}$ is the diffusion coefficient incorporating the time step size $\Delta t$. Note that the vertical index $k$ was omitted for brevity in all equations as no offsets are present along the vertical axis, though the data is of course dependent on $k$.

$$L^n_{i,j} = 4\phi^n_{i,j} - \phi^n_{i+1,j} - \phi^n_{i-1,j} - \phi^n_{i,j+1} - \phi^n_{i,j-1}, \tag{1}$$

$$F^n_{i+\frac{1}{2},j} = \begin{cases} L^n_{i+1,j} - L^n_{i,j} & \text{if } (L^n_{i+1,j} - L^n_{i,j})(\phi^n_{i+1,j} - \phi^n_{i,j}) \le 0, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

$$G^n_{i,j+\frac{1}{2}} = \begin{cases} L^n_{i,j+1} - L^n_{i,j} & \text{if } (L^n_{i,j+1} - L^n_{i,j})(\phi^n_{i,j+1} - \phi^n_{i,j}) \le 0, \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} - D^n_{i,j}\left(F^n_{i+\frac{1}{2},j} - F^n_{i-\frac{1}{2},j} + G^n_{i,j+\frac{1}{2}} - G^n_{i,j-\frac{1}{2}}\right). \tag{4}$$

The equations for the vertical advection stencil are omitted due to higher complexity. It solves the implicitly discretized advection equation of the velocity field along the vertical axis using three tridiagonal systems, one for each velocity component.

Multiple variants were implemented and experiments were performed on the hand-optimized kernels in order to identify the characteristics of the most efficient implementation for KNL. Among others, different strategies for data storage layouts, stencil operator fusion with temporary buffers and redundant (on-the-fly) computations, alignment, blocking, loop order, non-temporal stores, (software) prefetching, and different thread counts were evaluated.

## 2.2 Results

In general, the investigated storage layout of input and output data was restricted to globally contiguous linear arrays (possibly with padding), that is, arrays with a linear memory index computed as $n(i,j,k) = s_i i + s_j j + s_k k$, where $i, j$ are the storage indices along the horizontal axis, $k$ is the index along the vertical axis and $s_i, s_j, s_k$ are the corresponding strides. This is due a design restriction in the current implementation of GridTools. The limitation was chosen due to the large variety of stencils in a typical dycore, that would require different storage block sizes (depending on the access pattern) for optimal performance and thus layout transformations. Linearly indexed arrays combined with blocked looping avoid this difficulty.

All data fields were stored in structure-of-array manner, vector field components in separate scalar arrays. To achieve best possible memory bandwidth, huge pages were allocated, the unit stride dimension was padded to 64 bytes and the allocated data was shifted by some bytes (multiple of 64) to reduce L1 cache set conflicts as proposed in the Intel 64 and IA-32 Architectures Optimization Reference Manual [11].

To get an idea of the achievable peak bandwidth[1] of both the KNL and the P100 architecture, we first evaluated the performance on a copy "stencil", i.e., just a simple memory copy using a linear loop over the whole domain. Additionally, we implemented the copy stencil as a blocked 3D loop on KNL (blocked only along horizontal axes) respectively using a 3D grid in CUDA. The results for different domain sizes (all with 80 levels along the vertical dimension) when using double precision (64 bit) floating point numbers are shown in Figure 3. The P100 shows a higher bandwidth in general, but the overall curve looks very similar: bandwidth is only saturated for large domain sizes on both hardware architectures. Further, blocking reduces performance slightly for the copy stencil on small domains, but does not have much impact on the overall picture. Note that the block sizes were auto-tuned using a full 2D search (restricted to power-of-two sizes along each dimension) and depend on the domain size. Note also that on KNL some considerations need to be made to achieve full bandwidth: first, it is mandatory to use non-temporal store instructions for output data fields and second, storage strides of power-of-two bytes should be avoided as they may lead to a large number of L1 or L2 conflict misses. On smaller domain sizes, the difference in achievable bandwidth between normal store instructions and non-temporal ("streaming") stores is less pronounced as the bandwidth is not limited by the cache but by the memory itself.

Despite the clear advantage of the P100 on the copy stencil, for the *horizontal diffusion* stencil, we achieved performance on the KNL comparable to the available optimized GridTools implementations for NVIDIAs Tesla P100 GPU in the best configuration. Due to the purely horizontal access pattern this stencil prefers a data layout that is contiguous along the horizontal directions, the largest stride is along the vertical (i.e., $1 = s_i < s_j < s_k$ where $s_i$ and $s_j$ are the horizontal strides and $s_k$ is the vertical one). The loop order follows the same layout, i.e., the loop along the contiguous stride $s_i$ is innermost. Full parallelism was applied along the vertical dimension, the horizontal domain is divided into blocks. With the given storage layout, large block sizes along the contiguous

[1]We calculated the bandwidth from total data size and measured run time.

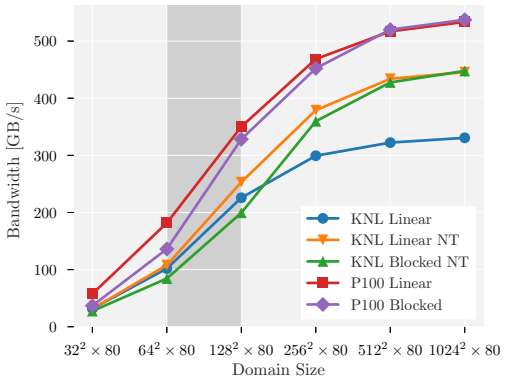


**Figure 3: Domain-dependent peak bandwidth for copy stencil on Intel KNL and NVIDIA P100. *NT* means non-temporal stores are used. The darker grey area highlights the relevant domain size range for current production forecasts.**

direction are important to exploit hardware prefetching. The best achieved run times for different domain sizes on KNL and P100 are shown in Figure 4. Note that the measurements were again taken using auto-tuned block sizes, found by a full parameter scan. On the KNL, this always led to a block size filling the whole domain on the contiguous direction (which shows the importance of hardware prefetching) and small block sizes along the non-contiguous dimension. 128 threads were used. For comparison, a simple addition of two fields with the same blocked data traversal was added. This has two input and one output fields, like the horizontal diffusion implementation using on-the-fly computations, but very simple access pattern and only a single FLOP per grid point.

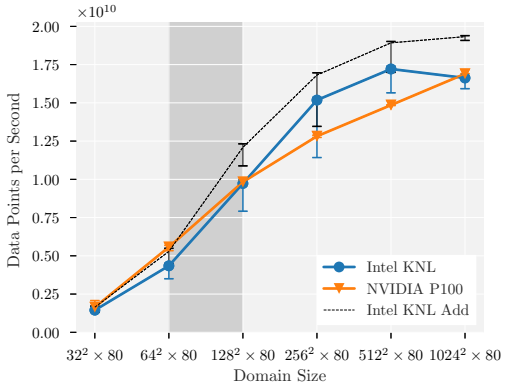


**Figure 4: Grid point updates per second for the optimized horizontal diffusion stencil on the Intel KNL and NVIDIA P100 architectures for different domain sizes. As a reference, the performance of a (blocked) addition of to fields on the KNL is provided.**

For the *vertical advection* stencil, the access pattern is a bit more complex than for horizontal diffusion. The stencil consists of a forward sweep followed by a backward sweep along the vertical

dimension, solving a tridiagonal linear system using the Thomas algorithm [24]. Due to data dependencies, vectorization along the vertical axis is not possible. Thus, the unit stride is again placed on one of the horizontal axes. On the other hand, to allow for efficient in-register storage of the intermediate coefficients appearing in the Thomas algorithm, we want the innermost loop to be along the vertical axis. Thus, we vectorize along one of the horizontal directions but have an innermost loop along the vertical. As the KNL's hardware prefetcher only recognizes unit stride (in cache line units) access patterns [11], software prefetching must be used to get good bandwidth numbers. While this works well with relatively small strides, we could not get any significant improvements from software prefetching on larger strides. That is, we had to reduce the strides along the vertical dimension and thus use a different data layout than for the horizontal diffusion stencil for optimal performance. The data was allocated such that $1 = s_i < s_k < s_j$. Using this storage layout, for domain sizes of up to $128 \times 128 \times 80$, we get almost 60% performance improvements due to software prefetching. On the $256 \times 256 \times 80$ domain its impact already drops slightly and for the even larger tested sizes it delivers only about 10% of performance improvement. The exact reason for the failure of software prefetching on the large domains is unknown but we expect some prefetcher limitations to be responsible.

Due to the very limited impact of software prefetching on the large domain sizes, we could not reach competitive results with the just presented strategy. There we achieved the best results by swapping the two innermost loops to exploit hardware prefetching along the contiguous (horizontal) axis while keeping the same data layout. But as this requires memory buffers for the coefficients used in the Thomas algorithms and thus puts more pressure on the caches, the run times on small domains are higher compared to the first strategy with working software prefetching (i.e., on small domains) and are not fully competitive with the P100 on any domain size. However, this change actually simplifies the user implementation and does not require any fine tuning of software prefetching distance as it solely relies on hardware prefetching.

Figure 5 shows the performance comparison between the strategies chosen for small domains (also shown with disabled software prefetching) and large domains, respectively. Shown is the median of 100 runs.

The performance comparison to P100 is shown in Figure 6. For KNL, the best strategy for the particular domain size is shown. Note that KNL is faster than the P100 on domains up to a size of 128×128×80 grid points, where software prefetching is working well. On larger domain sizes, we employ the alternative strategy that does not require software prefetching but does not achieve the same performance as the P100. 128 threads were used on the smaller domains (up to 256×256×80), 64 threads were used on the two largest domains.

A roofline plot for the two representative stencils is shown in Figure 7. Arithmetic intensity and required data transfer are manually computed values, based on the minimally required data transfers and floating point operations of the present implementations. While our implementation of the horizontal diffusion stencil has a much higher arithmetic intensity than a naive implementation due to the redundant on-the-fly computations and the avoidance of temporary buffers, it is not compute bound. In case of the vertical advection

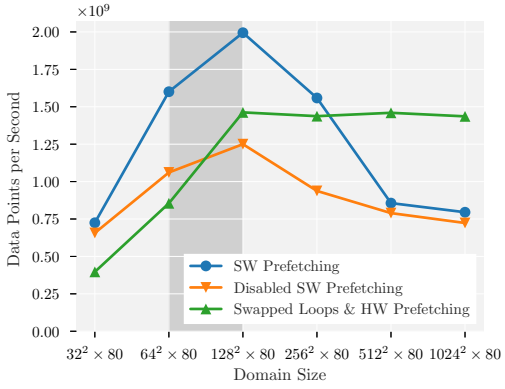


**Figure 5: Grid point updates per second for the vertical advection stencils on the Intel KNL with and without software prefetching as described in the text.**

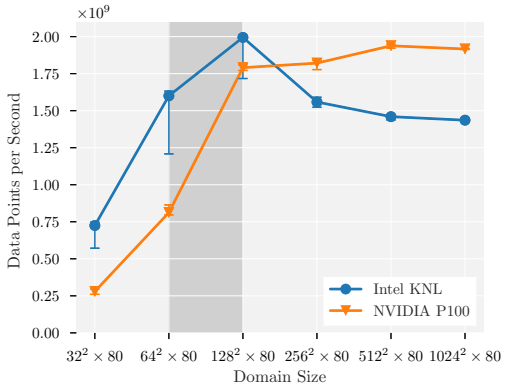


**Figure 6: Grid point updates per second for the optimized vertical advection stencil on the Intel KNL and NVIDIA P100 architectures for different domain sizes.**

stencil, the strategy changes and thus less optimal memory usage for large domain sizes is clearly visible.

Simplified code listings to clarify the looping strategies for both stencils are given in Appendix A.

## 2.3 Summary of Optimization Strategies

Here, we outline some general optimization strategies for the Intel KNL architecture, that we found to be crucial during our work and might be helpful for other optimization efforts.

Due to the high core count, setting the thread affinity correctly is very important. Further, the number of threads is a simple-to-tune parameter. We found 128 threads to perform reasonably well in most cases, sometimes 64 threads performed better. Using MCDRAM significantly improves performance on bandwidth-bound problems without any code changes.

Vectorization is crucial for reasonable performance. To reach high memory bandwidth, hardware prefetching should be exploited wherever possible. Otherwise, software prefetching might or might

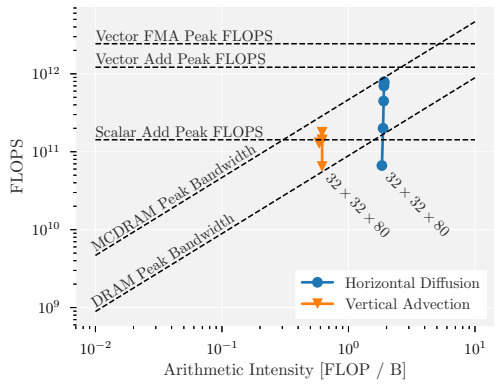


**Figure 7: Roofline plot for the same domain sizes used in the previous plots. The larger the domain size, the closer the results come to the peak MCDRAM bandwidth (see annotations to find the smallest domain size).**

not help. To reach peak bandwidth, non-temporal stores have to be employed, though using them correctly in a large code base might be non-trivial. Wrong usage can lead to large performance losses. Currently, no major compiler seems to apply them without user intervention.

If possible, using more compute resources instead of memory resources is often beneficial. Not only to reduce memory transfers, but also to reduce pressure on the sometimes limiting caching subsystem. For multidimensional arrays, padding the contiguous dimension to cache line size might improve performance. Further, using huge pages may reduce the pressure on the TLB. Power-of-two strides can lead to increased L1 and L2 cache conflicts or TLB conflicts and thus impact performance. Reducing the number of L1 set conflicts by slightly shifting allocated data as proposed in the Intel 64 and IA-32 Architectures Optimization Reference Manual [11] is recommended if multiple fields are accessed in the same loop.

# 3 GRIDTOOLS BACK-END IMPLEMENTATION

## 3.1 Methodology

Next we developed a back-end for the GridTools C++ EDSL framework, that implements the most efficient optimization strategies for KNL obtained in the hand-tuned codes. OpenMP was chosen for thread-level parallelism.

The new back-end is expected to deliver much improved performance as compared to the previously available CPU back-end not only on the Intel KNL platform but also on any similar architecture with wide vector registers, many cores and/or High Bandwidth Memory that support standard OpenMP code, for example the Intel Skylake and newer architectures. The reason is that the previous CPU back-end was optimized for legacy CPUs with small vector width and fewer cores. Its design does not allow vectorization on the vertical stencil codes in the COSMO dycore. Due to the usage of (pragma guided) auto-vectorization inside the back-end, compilation of the GridTools code on any platform with good C++ and OpenMP support should be possible, thus also non-Intel architectures are supported.

The parallel kernel skeletons and performance-relevant optimizations of the new KNL back-end of GridTools are based on the experiments with the hand-optimized representative stencils. Results are evaluated on the full COSMO dynamical core implemented using GridTools, which will replace the operational STELLA version (see Section 1.3), and will be discussed in Section 3.2.

In order to cover a representative configuration of components of the whole COSMO model, a physical parameterization, namely the Ritter and Geleyn radiation scheme [19] was ported to and optimized for KNL architecture. The current operational code is parallelized for CPU using only MPI parallelization, which is not compatible with the Dycore's OpenMP parallelization. In order to evaluate an efficient implementation of the physical parameterization on KNL, the radiation scheme was ported to OpenMP. Performance results and comparisons with the existing GPU-enabled operational code (that uses OpenACC) are shown in Section 3.4.

## 3.2 GridTools Back-End Design

According to the results described in section 2.2, we designed the GridTools back-end for KNL. As the optimal performance for horizontal and vertical stencil patterns is obtained using different storage layouts on the KNL (horizontal: $1 = s_i < s_j < s_k$; vertical: $1 = s_i < s_k < s_j$) due to different access patterns, we had to choose a compromise for the GridTools back-end. About half of the stencils in the COSMO dycore follow the horizontal pattern and half the vertical, so layout transformations are no option in this case. Due to the smaller performances loss of the vertically-optimal layout on the horizontal stencils than vice versa, the best layout for the vertical stencils has been chosen as the default in the GridTools back-end. On horizontal stencils this gives a loss of 15% – 25% in performance compared to the ideal layout, depending on the stencil. For data allocation, the same strategy as in the benchmarks is applied, 2 MB huge pages are allocated, the unit stride dimension is padded to cache line size and the data pointers get shifted such that L1 set conflicts are reduced.

GridTools' general design is based on element-wise operations. This allows to easily use all normal language constructs like *if*-statements and loops inside the stencil functions, which leads to well-readable code. On the other hand, it forbids to expose SIMD vector intrinsics (or wrappers around them) to the user. Thus, all vectorization relies on the compiler's auto-vectorization, guided by OpenMP pragmas. Enabling auto-vectorization in the complex C++ template based code of GridTools proved to be much harder than on the simpler code developed for the benchmarking presented before. Several compiler-specific workarounds in existing GridTools code had to be implemented inside the back-end code to allow for vectorization even of simple stencils and to work around compiler bugs. Nevertheless, with those changes, the Intel compiler is able to vectorize the majority of the tested stencils.

Data for intermediate temporary data fields is stored in per-thread private buffers. For temporary fields in purely horizontal stencils, the user can activate more efficient caching of 2D data planes, similar to the available GPU strategy.

Developing a fully automatic software prefetching strategy using GridTools is not possible in general. Depending on the exact stencil pattern, prefetching distance, the location of the prefetching instructions as well as the selection of which data even has to be prefetched might change heavily and is difficult to determine automatically. Despite the performance implications on small domain sizes, we decided to abandon this strategy in favor of implementing the best pattern found for large domain sizes, where software prefetching does not give a significant performance improvement. Those decisions give a run time penalty of estimated of up to 30% – 40% on vertical stencils on smaller domains compared to the hand-tuned code.

A further limitation is the lack of non-temporal stores in the GridTools back-end. Currently the only possibility to enable non-temporal stores on x86 hardware in a compiler-independent way is to use SIMD intrinsics, but the GridTools framework does not support the use of intrinsics since it fully relies on auto-vectorization. The Intel C++ Compiler provides two additional possibilities: using compiler pragmas or globally enabling them by a compilation flag. However, the latter is rarely useful as it also makes desired cache reuse difficult, while the former requires a pragma not where the data is really accessed, but *outside* the loop around the data access. However, we do not expect a large performance degradation in real-world applications due to missing non-temporal store instructions as often the output of one stencil is the input of the following stencil and thus caching is desired.

The last difference to the benchmark implementations is the selection of the per-thread block size: while we auto-tuned the representative stencil implementations for optimal domain-dependent block sizes, GridTools relies on a simple heuristic that tries to exploit hardware prefetching along the contiguous data dimension, which proved to be essential on the benchmark stencils, while respecting the (user-defined) number of threads.

## 3.3 Dycore Stencils Performance Results

While the benchmark results on the representative stencils presented in Section 2.2 look very promising, the compromise in the storage layout selection, reliance on auto-vectorization and sub-optimal implementation of vertical stencils as illustrated in the previous section, are clear points that limit the practical performance in the full COSMO dycore implementation. In addition to the GridTools-specific performance penalties, the dycore code itself was tuned to deliver close to optimal performance on GPUs. First, intermediate results of horizontal stencils are often cached in shared memory on the GPU, this caching is user-defined in GridTools code and will also be translated to "caching" in memory buffers on the KNL. But the experiments on the representative stencils have clearly shown that in most cases avoiding any unnecessary memory accesses is beneficial on the KNL. Thus, if intermediate results are accessed multiple times, it is often faster to compute those multiple times redundantly instead of caching them in buffers. With this caching strategy we experienced a performance loss of 30% – 100% compared to the redundant on-the-fly computations. Second, some stencils are optimized to benefit from the register-caching available in the GPU back-end along the vertical axis, even if there are no vertical dependencies. This strategy of course might only work on the GPUs as the KNL is lacking those caches. Third, several stencils contain code that is not auto-vectorized by the compiler, but after the applied code changes only a small number without major run time contribution is affected. For those reasons, a considerable slow down on the KNL compared to the P100 has to be accepted without introducing major code changes inside the GridTools dycore.

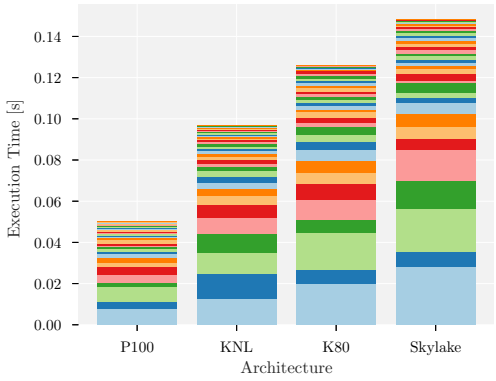


**Figure 8: GridTools COSMO dycore stencil execution times of one time step on different hardware platforms. The colors mark different stencils. Architectures are sorted by peak memory bandwidth. Domain size is 129×129×80 grid points using double precision floating point numbers.**

Figure 8 shows the total stencil run times on four different hardware platforms, namely the NVIDIA Tesla P100 and Tesla K80 and the Intel KNL and Intel Skylake architecture (see Section 1.5 for more detailed specifications and exact models of the Intel processors). The shown numbers are for a benchmark setup with a domain size of 129×129×80 grid points using double precision arithmetic, run with 128 threads. Relative numbers for other domain sizes and single precision are comparable. The total stencil execution times of the COSMO dycore on the KNL architecture is approximately a factor two of the run times on the P100 for the benchmark setup. Still, KNL is considerably faster than the NVIDIA K80 and Intel Skylake architectures, due to higher available memory bandwidth.

To see the per-stencil performance loss compared to the P100, Figure 9 shows a histogram including all stencils and relative run times compared to the P100. The median is 175%. 34 out of the 40 stencils executed in this benchmark lie within a factor 2.4 of the P100. The performance penalty for those stencils is mostly related to the non-optimal data layout (horizontal stencils only), missing register caching and software prefetching (vertical only) and usage of in-memory caching instead of on-the-fly computations. The very few stencils with an even larger slow down normally suffer from issues with auto-vectorization, other compiler limitations or GPU-specific optimizations that are counterproductive for KNL performance. Still, the newly implemented back-end is more than 10 times faster than the old CPU back-end already available in GridTools, when considering the total stencil run time of the COSMO dycore on the KNL. We expect similar results on comparable hardware, for example on the Intel Skylake or newer architectures (though to a lesser extent there due to missing high-bandwidth memory).

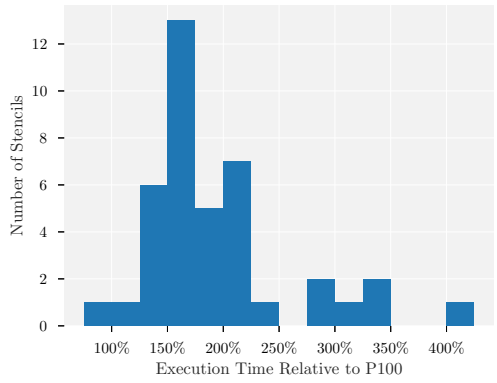


**Figure 9: Histogram of dycore stencil execution times run on KNL vs. P100. Most stencils are placed between 100% and 200%, i.e., one to two times slower on the KNL.**

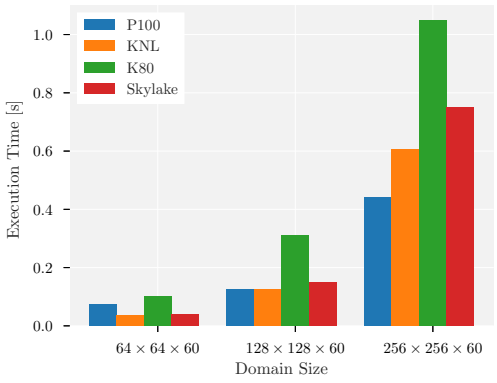


**Figure 10: Measured run times of the radiation code on different hardware. Architectures are sorted by peak bandwidth.**

Experiments have shown that, despite the considerable performance degradation compared to the P100, the performance is on par with the original COSMO dycore Fortran code for vertical stencils and an order of magnitude faster for horizontal stencils on the KNL for the given benchmark data. Due to lacking OpenMP parallelism in the original COSMO code, the Fortran dycore was executed using pure MPI. This leads to large communication costs for the horizontal stencils.

### 3.4 Physical Parameterization Performance Results

In order to have a comprehensive set of components for the COSMO performance evaluation on KNL, a relevant and computationally expensive parameterization, i.e. the radiative transfer, was ported to OpenMP, starting from the existing MPI + OpenACC implementation. The optimized code was compiled with Intel Fortran compiler 18.0. In contrast to the experience with auto-vectorization on the GridTools kernels of the dynamical core, the compiler could auto-vectorize all the parallel Fortran loops without reorganizations of the user code. This is due to two reasons: a much easier data dependencies pattern of the physical parameterizations (column based) as compared to the dynamical core stencil computations and the fact that Fortran implementations are easier to analyze and optimize than the heavy-templated C++ code used in the GridTools implementation. In order to maximize the performance of the OpenMP port used on KNL the OpenMP runtime parameters like thread affinity were tuned.

Figure 10 shows the performance comparison (median of 10 runs) of the OpenMP version measured for KNL and Skylake and the OpenACC version measured for K80 and P100 (compiled with Cray compiler v8.7.3). The figure shows that when using small domain sizes, CPU architectures outperform GPU architectures. Reason for this is that KNL and Skylake benefit from large caches, while GPU architectures suffer from a lack of parallelism. For large domains this is not true. As shown, P100 provides the best performance for large domains that provide a reasonable level of parallelism.

To compare the existing MPI-based implementation to the newly ported OpenMP, we tried different configurations of combined MPI + OpenMP parallelization. For this (communication-free) code, both approaches show very similar performance results if affinity is properly configured.

## 4 DISCUSSION

### 4.1 Achievements

We have shown performance results on hand-optimized stencil codes representative for weather and climate simulation codes on the Intel Xeon Phi KNL many core processor that are on par with the NVIDIA Tesla P100 GPU, both released in the same year. Our investigations show that the KNL architecture allows for good performance results, but reaching execution times on par with GPUs requires a detailed knowledge of the hardware and highly architecture-specific code optimization.

By implementing a back-end for the GridTools C++ EDSL optimized for the KNL, we could run the full COSMO dycore on KNL while requiring only small changes to the user code (ideally none, see next section). While we could observe a performance gap between the KNL and the P100 in the full dycore, mainly due to technical limitations and design limitations of the template-metaprogramming approach used by GridTools, this shows that reasonable performance (on par with the previously available Fortran implementation) on multiple architectures with the same user DSL code is possible. Porting the GridTools dycore to a new hardware architecture required, apart from workarounds for compiler issues, zero effort on the user code.

### 4.2 Limitations

The code complexity of the GridTools back-end design seems to overburden the current C++ compilers. A total redesign of the internals of the existing GridTools CPU back-end was necessary and compiler-specific workarounds had to be placed throughout the code to even enable auto-vectorization in most common cases. Guaranteeing successful auto-vectorization of even relatively simple user-code is still impossible. Several compiler bugs were found

and reported during the back-end development, but not all could be reproduced in simplified code. Five stencils of the dycore initially showed disastrous performance — one was more than 800 times slower than the GPU version — due to compiler issues. Those stencils required manual interventions in the *user* (dycore) code. Attempts to fix them inside the library code failed. The issues were triggered seemingly randomly. One observed trigger was for example a too deeply nested GridTools stencil function[2] call, which could be fixed by manually "inlining" the callee code (i.e., copying and pasting code). But detecting the reason for such problems is very difficult for the average user and good knowledge of the GridTools library and used compilers is often necessary for solving them.

While GridTools allows to run the same user code on multiple hardware platforms, the EDSL is not fully declarative. The user is still responsible for choosing the most effective implementation of a stencil *on a specific architecture*. This limits the practical performance portability of user code. Though, the achieved portability is still much better than for classical, fully platform-specific hand-optimized code and allows at least a certain degree of performance on multiple, totally different hardware architectures. A fully declarative stencil-DSL could possibly overcome these limits in the future, but fully automatic hardware-dependent optimization techniques specific to stencil codes would be required and are — according to our best knowledge — not available today.

## ACKNOWLEDGMENTS

This project was partially funded by the PASC initiative (www.pasc-ch.org) in form of the PASCHA project (Portability and Scalability of COSMO on Heterogeneous Architectures) and the Intel Parallel Computing Center (IPCC) program (PI: T. Hoefler). We would like to thank John Pennycook from Intel for his visit, valuable feedback on optimization strategies and support on compiler issues as well as Jason Sewell (also from Intel) for further input.

## REFERENCES

[1] Samantha V. Adams, Rupert W. Ford, M. Hambley, J. M. Hobson, I. Kavcic, C. M. Maynard, T. Melvin, Eike Hermann Müller, S. Mullerworth, A. R. Porter, Mike Rezny, Ben Shipway, and R. Wong. 2018. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *CoRR* abs/1809.07267 (2018). arXiv:1809.07267 http://arxiv.org/abs/1809.07267

[2] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. 2018. The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '18)*. ACM, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3218176.3218226

[3] COSMO. 1998. Consortium for Small-scale Modeling. http://www.cosmo-model.org/

[4] G Doms and M Baldauf. 2018. A Description of the Nonhydrostatic Regional COSMO-Model. http://www.cosmo-model.org/content/model/documentation/core/default.htm

[5] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. 2012. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming* 20 (2012), 89–114.

[6] Oliver Fuhrer, Tarun Chadha, Torsten Hoefler, Grzegorz Kwasniewski, Xavier Lapillonne, David Leutwyler, Daniel Lüthi, Carlos Osuna, Christoph Schär, Thomas C. Schulthess, and Hannes Vogt. 2018. Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development* 11, 4 (May 2018), 1665–1681. https://doi.org/10.5194/gmd-11-1665-2018

[7] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Christoph Schulthess. 2014. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing Frontiers and Innovations* 1, 1 (June 2014), 45–62–62. https://doi.org/10.14529/jsfi140103

[8] Mark Govett, Jim Rosinski, Jacques Middlecoff, Tom Henderson, Jin Lee, Alexander MacDonald, Ning Wang, Paul Madden, Julie Schramm, and Antonio Duarte. 2017. Parallelization and Performance of the NIM Weather Model on CPU, GPU, and MIC Processors. *Bulletin of the American Meteorological Society* 98, 10 (2017), 2201–2213. https://doi.org/10.1175/BAMS-D-15-00278.1 arXiv:https://doi.org/10.1175/BAMS-D-15-00278.1

[9] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2925426.2926286

[10] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. 2015. STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, 41:1–41:12. https://doi.org/10.1145/2807591.2807627

[11] Intel Corporation. 2016. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[12] Intel Corporation. 2017. Intel® Xeon Phi™ Coprocessor x200 Product Family Datasheet. https://www.intel.com.br/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-x200-family-datasheet.pdf

[13] Intel Corporation. 2018. Product Change Notification 116378 - 00. https://qdms.intel.com/dm/i.aspx/9C54A9A7-BF37-4496-B268-BD2746EA54D3/PCN116378-00.pdf

[14] Jim Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming (Knights Landing Edition)*. Morgan Kaufmann, Boston. https://doi.org/10.1016/B978-0-12-809194-4.09988-9

[15] John Michalakes, Michael J. Iacono, and Elizabeth R. Jessup. 2016. Optimizing Weather Model Radiative Transfer Physics for Intel's Many Integrated Core (MIC) Architecture. *Parallel Processing Letters* 26 (2016), 1–16.

[16] J. Mielikainen, B. Huang, and A. H.-L. Huang. 2014. Intel Xeon Phi accelerated Weather Research and Forecasting (WRF) Goddard microphysics scheme. *Geoscientific Model Development Discussions* 7, 6 (Dec. 2014), 8941–8973. https://doi.org/10.5194/gmdd-7-8941-2014

[17] T. A. J. Ouermi, Aaron Knoll, Robert Michael Kirby, and Martin Berzins. 2017. OpenMP 4 Fortran Modernization of WSM6 for KNL. In *PEARC*.

[18] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Orlando, FL, USA, 297–306. https://doi.org/10.1109/IPDPS.2017.30

[19] Bodo Ritter and Jean-Francois Geleyn. 1992. A Comprehensive Radiation Scheme for Numerical Weather Prediction Models with Potential Applications in Climate Simulations. *Monthly Weather Review* 120, 2 (Feb. 1992), 303–325. https://doi.org/10.1175/1520-0493(1992)120<0303:ACRSFN>2.0.CO;2

[20] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär. 2019. Reflecting on the Goal and Baseline for Exascale Computing: A Roadmap Based on Weather and Climate Simulations. *Computing in Science Engineering* 21, 1 (Jan. 2019), 30–41. https://doi.org/10.1109/MCSE.2018.2888788

[21] Pascal Spörri. 2017. COSMO C++ Dynamical Core Training Course - Introduction and Code Flow. https://wiki.c2sm.ethz.ch/pub/COSMO/CXXDynamicalCore/20170403_-_1_-_CPP_Dycore_Intro_Code_Flow.pdf

[22] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2018. TOP500 List – November 2018. https://www.top500.org/lists/2018/11/

[23] Lukasz Szustak, Krzysztof Rojek, and Pawel Gepner. 2014. Using Intel Xeon Phi Coprocessor to Accelerate Computations in MPDATA Algorithm. In *Parallel Processing and Applied Mathematics (Lecture Notes in Computer Science)*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, 582–592.

[24] Llewellyn H. Thomas. 1949. *Elliptic Problems in Linear Differential Equations over a Network*. Watson Science Computer Laboratory Report. Columbia University, New York, NY, USA.

[25] Louis J. Wicker and William C. Skamarock. 2002. Time-Splitting Methods for Elastic Models Using Forward Time Schemes. *Monthly Weather Review* 130, 8 (Aug. 2002), 2088–2097. https://doi.org/10.1175/1520-0493(2002)130<2088:TSMFEM>2.0.CO;2

[2] Stencil operators in GridTools can be called similar to normal functions from inside other stencil operators. Information about the current iteration point and neighborhood of the caller get passed on to the callee.

# A REPRESENTATIVE STENCIL CODES

**Listing 1: Simplified horizontal diffusion stencil implementation.**

```
// loop over blocks
#pragma omp for collapse(3)
for (int k = 0; k < ksize; ++k) {
    for (int jb = 0; j < jsize; jb += jbsize) {
        for (int ib = 0; ib < isize; ib += ibsize) {

            // loop inside block
            for (int j = jb; j < jb + jbsize; ++j) {
                // vectorization (contiguous axis)
                // exploiting hardware prefetching
                #pragma omp simd
                #pragma vector nontemporal
                for (int i = ib; i < ib + ibsize; ++i) {
                    // stencil computation
                }
            }
        }
    }
}
```

**Listing 2: Simplified vertical advection stencil implementation. Version for small domains.**

```
// loop over blocks
#pragma omp for collapse(2)
for (int jb = 0; j < jsize; jb += jbsize) {
    for (int ib = 0; ib < isize; ib += ibsize) {

        // loop inside block
        for (int j = jb; j < jb + jbsize; ++j) {

            // vectorization (contiguous axis)
            #pragma omp simd
            for (int i = ib; i < ib + ibsize; ++i) {
                for (int k = 0; k < ksize; ++k) {
                    // software prefetching to L2 cache
                    // forward loop operations
                }

                for (int k = ksize - 1; k >= 0; --k) {
                    // backward loop operations
                }
            }
        }
    }
}
```

**Listing 3: Simplified vertical advection stencil implementation. Version for large domains.**

```
// loop over blocks
#pragma omp for collapse(2)
for (int jb = 0; j < jsize; jb += jbsize) {
    for (int ib = 0; ib < isize; ib += ibsize) {

        // loop inside block
        for (int j = jb; j < jb + jbsize; ++j) {
            for (int k = 0; k < ksize; ++k) {
                // vectorization (contiguous axis)
                // exploiting hardware prefetching
                #pragma omp simd
                for (int i = ib; i < ib + ibsize; ++i) {
                    // forward loop operations
                }
            }

            for (int k = ksize - 1; k >= 0; --k) {
                // vectorization (contiguous axis)
                // exploiting hardware prefetching
                #pragma omp simd
                for (int i = ib; i < ib + ibsize; ++i) {
                    // backward loop operations
                }
            }
        }
    }
}
```