Master in High Performance Computing

GPU Porting and Optimization of the RegCM5

Supervisor:
Ivan Girotto, Ph.D.
Serafina Di Gioa, Ph.D.
Co-supervisor:
Graziano Giuliani

Candidate:
Christian D. Tica Ph.D.

11th edition
2024–2026

Master in
High Performance Computing

Abstract

We implemented and tested various optimizations and GPU offloading strategies on the coupled configuration of the RegCM5 regional climate model based on OpenACC and Fortran standard language parallelism through `do concurrent`. For an idealized, uncoupled, no-IO configuration, in which RegCM5 simulates a free-running atmosphere, initial benchmarks showed that a GPU-enabled run on 2 Leonardo Booster nodes using 8 A100 GPUs and 8 CPU cores outperformed the baseline CPU-only configuration running on 8 nodes of the Leonardo DCGP partition with 896 CPU cores. However, when RegCM5 was coupled with the Community Land Model CLM4.5 for a scientifically realistic production run, we found that the number of GPUs required to outperform the CPU-only configuration running on 800 CPU cores increased to 64.

Initial profiling confirmed that the delayed crossover point, at which GPU-enabled runs begin to outperform the CPU-only baseline in the coupled configuration, is caused by several code paths that are activated only in this mode. The first performance hotspot accounted for 20% of the initial total wall-clock time and originated from the noncontiguous row-slice initialization of two dimensional arrays in the hydrology tracer routine inside the module `mod_clm_hydrology2`. These whole-array initializations are lowered by the nvfortran compiler into vectorized memory-set operation under the symbol `__c_memset_avx`. To eliminate this hotspot, we replaced the implicit array operation with an explicit loop. This optimization yielded a 1.77× speedup for the GPU-enabled run.

The next hotspot accounted for 54% of the resulting wall-clock time was associated with the thermodynamic solver used to compute the Convective Available Potential Energy (Cape) and Convective Inhibition (CIN). The CAPE/CIN post-processing diagnostic was ported to GPU by refactoring the original column-based routine into an accelerator-compatible version using explicit work arrays and by restructuring the output code so that each atmospheric column can be processed independently. This preserves the original physical formulation while exposing fine-grained parallelism suitable for GPU execution. We applied the same porting strategy to the call sites of two additional procedures `interp1d_r8` and `heatindex` which together contributed another 20% of the wall-clock time. These optimizations and porting strategies produced an overall 7.10× speedup relative to the initial code. As a result, the GPU-enabled run using 16 GPUs on Booster now runs 2.4× faster than the CPU-only configuration using 800 CPU cores on DCGP. Additional optimizations and porting efforts, not discussed in the present work further increased the total speedup to 14.17× relative to the initial version.

We also studied the performance portability of the coupled configurations across two other platforms running on H100 GPUs. We observed that the coupled configuration exhibited the same superlinear scaling across three computing platforms the origin of which needs to be investigated in future work.

Ensuring that the GPU ported code reproduces the CPU-only results at the bitwise level is not yet part of the present work. At this stage, we observed only statistical reproducibility between the ported and CPU-only runs. After 7 model days, the numerical divergence remained below 0.1% for the near-surface air temperature and surface pressure while for the near-surface specific humidity remained within 5%.

Contents

1 Background and Introduction **1**
1.1 Kilometer-Scale Climate Models . . . 1
1.2 The RegCM5 . . . 2
1.2.1 Configuration and Building RegCM5 . . . 3
1.3 OpenACC . . . 4
1.4 The Fortran `do concurrent` Construct . . . 5

2 Baseline Benchmarking and Profiling **9**
2.1 Hardware and Software Environment . . . 10
2.2 Performance Benchmark of RegCM5 in RCE Configuration . . . 10
2.3 Performance Benchmark of RegCM5 in a coupled configuration with CLM4.5 . . . 14
2.4 Profiling . . . 17
2.4.1 The `__c_mset16_avx` Hotspot. . . . 18

3 GPU porting of the `getcape` subroutine **23**
3.1 Offloading using `OpenACC` . . . 24
3.2 Offloading with `do concurrent` . . . 25
3.3 Offloading with `do concurrent` and `OpenACC` . . . 26
3.4 Post-offload profiling and benchmarking . . . 27
3.5 Removing `Hotspot 4` and `5` . . . 29
3.6 Post-offload Profiling and Benchmarking . . . 31

4 Performance Portability **35**

5 Verification,Validation and Reproducibility **39**
5.1 Assessing Numerical Divergence . . . 39
5.2 Numerical divergence in `tas` . . . 40
5.3 Numerical divergence in `huss` . . . 43
5.4 Numerical divergence in `ps` . . . 44

6 Conclusions, Recommendations and Future Outlook **47**

A Profiling Tools **49**
A.1 `Perf` . . . 49
A.2 Flame graphs . . . 49
A.3 Nvidia Nsight systems . . . 52

B Reproducibility Diagnostic Workflow **55**
B.0.1 File naming and directory conventions . . . 55
B.0.2 Step 1: Concatenation and GPU–CPU difference . . . 56
B.0.3 Step 2: Multi-day temporal statistics . . . 57
B.0.4 Step 3: Spatial maps of absolute and relative error . . . 58

B.0.5 Recipe for adding a new variable 59

List of Figures **61**

References **65**

Chapter 1

Background and Introduction

1.1 Kilometer-Scale Climate Models

Major efforts in the climate modeling community are presently directed at refining the horizontal resolution of climate models to about 1 km either by refining the grid spacing employed on global climate models (GCM) or expanding the computational domain of high-resolution regional climate models (RCM) [1, 2]. In both cases, this development will enable the replacement of the parametrization of moist convection by an explicit treatment which is expected to improve simulation for the water cycle, precipitation extremes, and aspects of climate sensitivity tied to cloud processes [1].

Kilometer-scale simulations will dramatically increase the computational cost as a consequence of keeping the time steps correspondingly reduced for numerical stability. In addition, the total operation count also increases steeply with decreasing grid spacing. The community also faces an emerging data avalanche: high-frequency, high-resolution output quickly reaches petabyte to exabyte scales, making traditional "store-then-analyze" workflows increasingly impractical [1, 3].

These constraints coincide with the evolution of flagship High Performance Computing (HPC) platforms toward heterogeneous architectures, a trend that began in 2012 with the announcement of the first GPU-based system at the top of TOP500 list [4]. This shift has had significant impact on the various parallel programming models used for writing simulation software. OpenMP introduced standardized support for heterogeneous systems and became a shared-memory, directive-based parallel programming model both for multicore CPUs and offloading to GPUs with version 4.0 in 2013 [5]. During the same year, GPU-aware MPI is introduced to allow MPI libraries to directly handle device memory pointers, enabling features such as GPUDirect Remote Direct Memory Access (RDMA) and avoiding redundant host–device copies, thereby improving communication efficiency and scalability in GPU-accelerated applications [6, 7, 8].

In addition, GPU programming models and ecosystems from various vendors such as the directive-based OpenACC and OpenMP and low-level programming frameworks like NVIDIA CUDA and AMD ROCm are tightly co-designed and continuously being developed over the years in order to exploit hardware features that come with every generation of GPU architectures being released. These tools allowed application programmers to fully expose various levels of parallelism in their algorithms by mapping specific portions of the code to run on specialized massively parallel processors and achieve orders of magnitude speedups [9]. At present this trend towards heterogenous parallel computing has not waned. This is clear from the fact that 9 out of the top 10 systems which dominate the TOP500 list in 2025 are all GPU-accelerated, heterogeneous platforms [10].

The urgency and feasibility of kilometer-scale simulation of climate systems on modern heterogeneous supercomputing platforms is reflected in recent HPC milestones, including the 2025 ACM Gordon Bell Prize for Climate modelling for a novel kilometer-scale configuration of the ICON Earth system model [11]. To enable running the global model in a fully coupled

configuration and at this unprecedented resolution, the simulation is run on 8192 GPUs on Alps and 4096 on Jupiter, the first ever exascale supercomputer in Europe. Full earth simulation is made possible by mapping components to specialized heterogeneous systems and an optimization strategy that separates the implementation of model components in Fortan from the optimization details of the target architecture[2].

For their part, running RCMs at convection-permitting (CP) resolutions (CP-RCM) of a few kilometers also present novel challenges. The variability of many climate variables are expected to increase with a refined scale which implies the need to run large model ensembles in order to identify local-scale forced signals. Thus CP-RCM is estimated to increase both storage and compute requirements of 2-3 orders of magnitude. Furthermore, CP-RCMs require the availability of high resolution data set for model assessment and evaluation which is not available for most regions of the world. For these reasons, the use of CP-RCMs over other traditional downscaling techniques must be strongly justified in terms of added value [12].

This added value comes from the enhanced quality of fine-scale climate simulations offered by RCMs for a fraction of resources that GCMs require. While fully-coupled high resolutions GCMs are now a reality, running them for longer simulation runs require massive computational and storage resources that are presently not yet available in most parts of the world. Hence optimizing RCMs and their acceleration via GPU porting process remains an important development for the regional climate modeling community especially for the global south where self-organized engagement and participation of scientific communities are an important component of what drives model development [12].

In this thesis, we aim to contribute to this goal by carrying out a series of performance profiling, benchmarking and the subsequent optimizations and acceleration via GPU porting of the latest version of the RegCM regional climate modeling system, RegCM5. This new model version is based on the implementation of the non-hydrodynamic dynamical core which allows the model to be used at CP resolutions of few kilometers with increased accuracy and efficiency [13].

1.2 The RegCM5

The Regional Climate Model (RegCM5) is a limited-area regional climate model developed at the Abdus Salam International Centre for Theoretical Physics (ICTP). It runs long-term climate simulations through dynamical downscaling of global climate model outputs or reanalysis datasets and operates under a one-way nesting framework. That is, the time-dependent lateral boundary conditions are prescribed by a driving global model, with no feedback from the regional model to the large-scale circulation [12, 14, 13].

The RegCM modeling system is composed of four main components: a terrain preprocessor, an initial and boundary condition (ICBC) preprocessor, the RegCM atmospheric core, and a postprocessing system. The preprocessors perform horizontal and vertical interpolation of static and meteorological input fields from global datasets onto the regional model grid, while the core integrates the governing atmospheric equations and physical parameterizations forward in time [14].

RegCM5 supports multiple dynamical cores, including a hydrostatic solver derived from the MM5 mesoscale model, a hydrostatic solver using pressure-based vertical coordinates, and the recently introduced MOLOCH non-hydrostatic dynamical core employing height-based terrain-following coordinates [13]. These cores are discretized using finite-difference methods on staggered horizontal grids, with either Arakawa B-grid or Arakawa C-grid configurations depending on the selected dynamical formulation [14].

The model includes a comprehensive suite of physical parameterizations representing radiative transfer, land–atmosphere interactions, planetary boundary layer processes, moist convection, cloud microphysics, surface fluxes, and aerosol and chemistry processes. This modular design

allows RegCM5 to be configured for a wide range of regional climate applications and spatial resolutions, from continental-scale simulations to convection-permitting regional studies [13].

Table 1.1: Top-level structure of the RegCM5 source tree.

Name	Description
`bin/`	Installed executables after `make install`.
`Doc/`	User documentation, manuals, and reference material.
`external/`	Third-party libraries, data utilities, or bundled tools.
`Main/`	Core RegCM5 model source code (dynamics, physics, I/O).
`PostProc/`	Post-processing scripts and diagnostic utilities.
`PreProc/`	Preprocessors: terrain, ICBC, SST, CLM preprocessing.
`Share/`	Shared modules and common utilities across components.
`Testing/`	Test cases, validation runs, sample namelists.
`Tools/`	Auxiliary scripts and utilities for building or running RegCM.
`acinclude.m4`	Autotools macro definitions used by `configure.ac`.
`bootstrap.sh`	Developer script to regenerate the autotools system.
`CITATION.cff`	Standard citation metadata for RegCM.
`configure.ac`	Autotools configuration script defining build logic.
`INSTALL`	Installation instructions for users.
`LICENSE`	License information for RegCM.
`Makefile.am`	Automake input file describing build targets.
`makeinc`	Makefile fragment controlling executable naming and flags.
`README`	Project overview and basic instructions.
`regcm.png`	Logo or diagram included in documentation.

1.2.1 Configuration and Building RegCM5

A detailed guide to getting started with RegCM5 is given in [15]. To configure and install the RegCM source code, the following software stack is needed.

1. GNU build system
2. GNU `make` program
3. A Fortran 2003 Compiler
4. A `C` compiler for the option `-enable-mpi-serial` during configuration
5. An implementation of the the Message Passing Interface (MPI) library compiled with the fortran compiler for running parallel runs on multicore CPUs and across multiple compute nodes.
6. netCDF format I/O library compiled with the above fortran compiler with netCDF4 format support.

In addition, specific tools used in this thesis which will be relevant for visualizations and post-processing of netCDF output files, as well as for performance profiling of the simulation runs, are discussed in detail in the appendix section.

RegCM5 relies on the `GNU` autotools build system to provide portable configuration and compilation for different platforms. The files that are relevant in this build process are listed in table 1.1. The most important element of this system is the `configure.ac` file in the root directory. It encapsulates all the build-time logic and is used to generate the `configure` script and the `Makefile.in` template which are both used to generate a `Makefile` that orchestrates the compilation and linking of the model components. We provide the schematics of this process in figure 1.1.

The file `configure.ac` performs three main tasks: (i) it declares basic project metadata and programming language environment; (ii) it detects the available compilers and external libraries; and (iii) it exposes model features through a set of `-enable-*` and `-with-*` options, translating these choices into compiler flags, preprocessor definitions and lists of executables to be built.

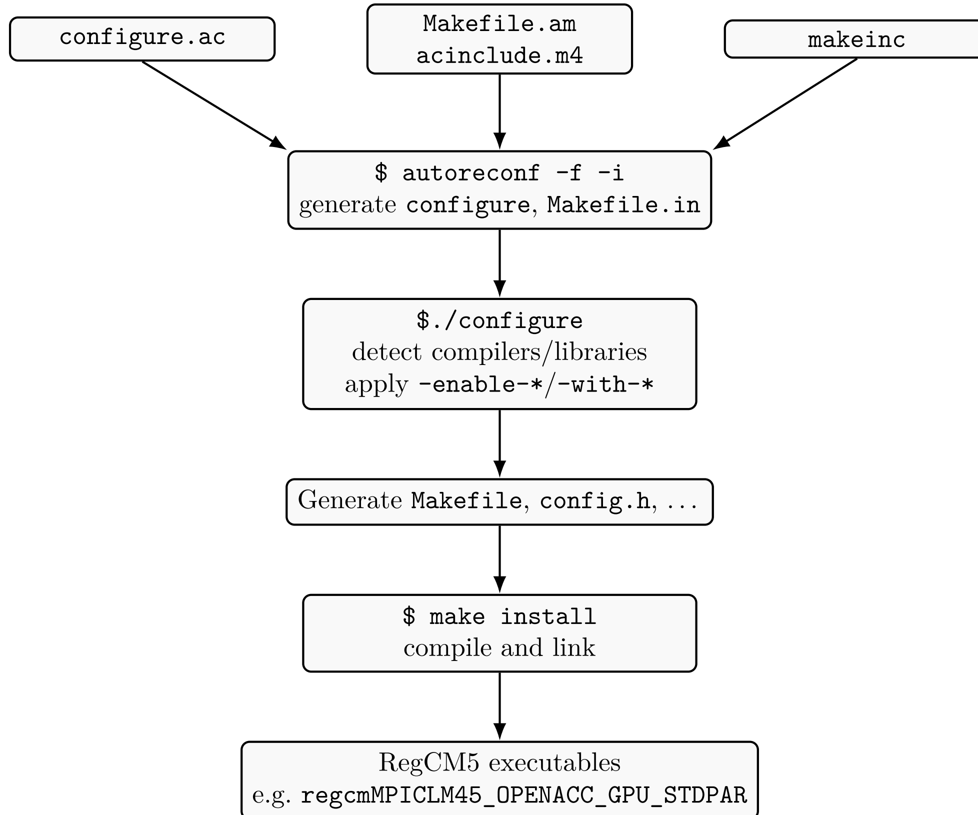


Fig. 1.1: Build pipeline from `configure.ac` and related build-system files to the RegCM5 executables used in this thesis. The files `configure.ac`, `Makefile.am`, `acinclude.m4` and `makeinc` are processed by the autotools tools (`autoreconf`, `autoconf`, `automake`) to generate the `configure` script and the `Makefile.in` templates.

1.3 OpenACC

OpenACC is a directive-based parallel programming model compatible with C/C++ and Fortran designed for programming heterogeneous HPC systems consisting of CPUs as the host device and an accelerator device [16, 17]. It is introduced with the aim of enabling domain experts and scientific application developers to quickly introduce GPU-acceleration to legacy scientific applications without the need for major code refactors or implementing vendor-specific code paths. Such features make it the suitable choice for adding accelerator support for climate models like the RegCM5 which are often structured around loop-based numerical kernels, and maintained across decades of institutional development.

Parallelism is exposed in OpenACC by means of compiler directives that describe how loop iterations and code regions may execute concurrently on accelerators such as a GPU. The execution model of OpenACC is hierarchical in that work units are divided among threads of execution that are organized into `gangs`, `workers`, and `vector` levels. These levels of organizations are in turn mapped by the compiler to actual physical hardware resources, for example thread blocks, warps, and individual threads in an NVIDIA GPU. The offloading process is facilitated by OpenACC compute directives such as `parallel loop` and `kernels` which the programmer use to annotate compute intensive loops [18].

Hence in OpenACC, unlike in explicit GPU programming models like CUDA and HIP, the programmer specifies the parallel regions with compiler directives and the compiler determines how this parallelism is executed on the GPU. This abstraction of the GPU hardware from the programmer's perspective allows for a comparable gain in performance for a fraction of cost in programming time and effort.

When executed on systems with discrete GPUs, OpenACC follows a host-device memory model: the CPU manages host memory while the GPU manages a separate device memory, and the data is transferred between them via PCI Express (PCIe) or NVLink. To reduce overheads introduced by data movement, OpenACC also provides data directives and clauses like `copy`, `copyin`, `copyout`, `create`, `present` that specify how variables are allocated and their persistence on either device. Most of the performance gain in using OpenACC come from how the data regions is structured which influences the efficiency of data movement between host and accelerator [19].

Directive-based programming models like OpenACC has many features that have direct impact on productivity, portability, and maintainability. It allows for an incremental and modular offloading process, so that portions of the code base can be ported without reorganizing the entire model. In addition, OpenACC directives are simply ignored by compilers that lack OpenACC support, preserving source compatibility with CPU-only systems. More importantly, OpenACC is designed so the GPU porting process introduces minimal code intrusion and the scientific logic remains largely unchanged. This greatly reduces chances of numerical divergence, which is important for the numerical reproducibility of the results.

The strength of OpenACC is also balanced by a few limitations. The compiler ecosystem support for OpenACC is uneven, with the strongest support limited to NVIDIA platforms. Consequently, portable performance across other vendors and platforms is not always guaranteed. Still for large Fortran climate models, OpenACC remains a pragmatic tool for accelerating kernels without disrupting scientific correctness and hurting long-term maintainability and code longevity.

1.4 The Fortran `do concurrent` Construct

Recently, programming languages themselves started adding features within their standards that compilers can use as hints to enable parallel execution on multi-core CPU and accelerators. The aim of standard language parallelism is to reduce the explicit reliance on external, vendor-specific APIs at the programming layer; and to write codes with greater performance portability, maintainability and longevity while maintaining speedups comparable to that of other programming models. Example of these language features include the C++ 17's standard parallel algorithms and Fortran's `do concurrent` (DC) construct.

DC is introduced in the Fortran 2008 standard to provide a language-level mechanism to indicate that the iterations of a `do` loop have no data dependency and can therefore be executed out-of-order. Its use for accelerated computing started in 2020 with NVIDIA HPC SDK's compiler support which relies on OpenACC backend for offloading DC loops to NVIDIA GPUs. Table 1.2 summarizes the current level of compiler support for Fortran's DC construct across major vendors and compilers [20].

For CPU execution, most compilers are able to parallelize DC loops, whereas GPU offloading support is still evolving. NVIDIA's `nvfortran` and Intel's `ifx` currently provide the most mature GPU paths, while GNU gfortran and LLVM Flang lack complete GPU offload support at the time of writing.

Table 1.2: Compiler support for Fortran `DO CONCURRENT` on CPUs and GPUs.

Vendor / Compiler	**CPU**	**GPU**	**Notes**
NVIDIA `nvfortran`	Yes	Yes	OpenACC backend for NVIDIA GPUs. (2020)
Intel `ifx`	Yes	Yes	OpenMP backend for Intel GPUs (2022).
HPE Cray	Yes	Yes	Supports AMD & NVIDIA GPUs.
GNU `gfortran`	Partial	No	CPU auto-parallelization only.
LLVM Flang	Emerging	Pending	Active development.

In the case of GPU-accelerated codes, the DC construct relies on a memory model that automatically pages data between the CPU and GPU on demand. Since this capability is neither part of the Fortran nor OpenACC standards, specific implementations are provided by the vendor in their software stack and natively in their hardware. The NVIDIA implementation for the memory model is the CUDA unified memory (UM) or Unified Virtual Memory (UVM) and more recently, Heterogeneous Memory Management (HMM) [21]. It is called Unified Shared Memory for Intel OneAPI and Smart Access Memory for AMD.

In the case of NVIDIA GPUs, hardware support for the UVM is implemented via a dedicated page migration engine. This capability is first realized in their Pascal GPUs such as the Titan X and the Tesla P100 [22] and presently with their most advanced implementation in their Grace Hopper super chip [23]. Depending on the hardware support, the NVIDIA implementation of UVM can be activated with either `gpu=mem:manged` or the more advanced `gpu:mem=unified` compiler options on NVHPC [24].

Listing 1.1 is a stencil operation in a triple-nested `do loop` written with DC. The corresponding `nvfortran` GPU offload diagnostic for the DC kernel is given listing 1.2. This compiler feedback is generated by passing the `-Minfo=accel` compilation flag. This diagnostic confirms that the DC loop is offloaded as a GPU kernel, with the three nested indices collapsed and distributed across CUDA thread blocks and threads, and with implicit `copyin`/`copyout` of the involved arrays.

Listing 1.1: Example DO CONCURRENT

```
1 do concurrent ( j = jci1:jci2, i = ici1:ici2, k = 2:kz )
2     zuh = (u(j,i,k)   + u(j,i,k-1))   * hx(j,i) +    &
3                (u(j+1,i,k) + u(j+1,i,k-1)) * hx(j+1,i)
4     zvh = (v(j,i,k)   + v(j,i,k-1))   * hy(j,i) +    &
5                (v(j,i+1,k) + v(j,i+1,k-1)) * hy(j,i+1)
6     s(j,i,k) = -0.25_rkx * (zuh+zvh) * gzitak(k)
7 end do
```

Listing 1.2: Compilation report generated by `nvfortran` with the `stdpar=gpu` and `minfo=accel` options enabled.

```
1 ! blockidx%x threadidx%x auto-collapsed
2 Loop parallelized across CUDA thread blocks, CUDA threads(128) collapse
     (3) ! blockidx%x threadidx%x collapsed-innermost
3 Generating implicit copyin(hx(jci1:jci2+1,ici1:ici2),gzitak(2:kz)) [if
     not already present]
4 Generating implicit copyout(s(jci1:jci2,ici1:ici2,2:kz)) [if not
     already present]
5 Generating implicit copyin(hy(jci1:jci2,ici1:ici2+1)) [if not already
     present]
```

There are recent investigations [25, 26] on the performance and portability of the `do concurrent` construct for accelerated computing on small to medium real-world production code bases. These studies demonstrate that DC still need to rely on interoperability with other

programming models in order to support more complex parallel programming paradigms. In the case of running on multiple GPUs with MPI for instance, Fortran has no concept of multiple accelerator devices in the standard, so that one must then depend on external API's to implement device selection by each rank either programmatically or at runtime through environment variables like the `CUDA_VISIBLE_DEVICES` in the NVIDIA programming environment. The same can be said about GPU-aware MPI, and other parallel programming features and constructs like reduction, atomic operations, asynchrony, and explicit memory management all of which are crucial for performant and semantically correct parallel code execution.

The DC construct represents and important long-term direction for portability and language-level expressiveness for production-scale HPC applications. Implementing `loops` in `do concurrent` is also ideal and easy in initial stages of code development as it forces the programmer to think in parallel early on. However for production scale GPU-accelerated workoads, these recent studies show that `do concurrent` by itself still lacks the operational maturity and the strong compiler support across vendors to qualify as a reliable programming model on par with other existing alternatives.

Chapter 2

Baseline Benchmarking and Profiling

In this chapter we establish a baseline performance benchmark and profiling of RegCM5 under two distinct configurations. In the first case, we configured RegCM5 in radiative-convective equilibrium (RCE) for a minimal and idealized modeling scenario. In accordance with the Radiative-Convective Equilibrium Model Intercomparison Project (RCEMIP) [27], this is done by prescribing, a uniform solar insolation, a constant sea surface temperature (SST) or a slab ocean model beneath the atmosphere, and initializing the model with random noise.

The second configuration couples the RegCM5 with the community land model, CLM4.5 [28], for a more realistic and production modeling scenario. The coupled configuration relies on extensive input datasets, including terrain, land-use, and time-varying atmospheric boundary conditions, and produces large volumes of NetCDF output encompassing atmospheric, surface, and land-state variables. As a result, I/O is negligible in RCE benchmarks but constitutes a major component of runtime and data movement in the coupled configuration. Table 2.1 summarizes the essential physical and dynamical difference between these two configurations.

Table 2.1: Comparison of the coupled RegCM5–CLM4.5 (ERA5-driven) configuration and the RCE configuration.

Aspect	**Coupled RegCM5–CLM4.5 (ERA5-driven)**	**RCE Configuration**
Scientific goal	Physically realistic regional climate simulation and performance benchmarking	Idealized radiative–convective equilibrium and performance benchmarking
Domain and geography	European regional domain with realistic orography and land–sea contrasts	Horizontally uniform, flat domain with idealized surface conditions
Large-scale forcing	Time-dependent lateral boundary conditions from ERA5 reanalysis	No lateral forcing or relaxation; free-running atmosphere
Surface model	Interactive land surface via CLM4.5 with heterogeneous soil and vegetation	Simplified and spatially homogeneous surface conditions
Dynamics	Constrained by large-scale circulation entering the domain	Internally generated circulation only
Physics packages	Full physics suite for climate realism, including land–atmosphere coupling	Radiation and convection retained; non-essential components disabled
I/O behavior	Regular I/O for ICBC updates, coupling fields, and diagnostics	I/O minimized or disabled to isolate computational kernels
Intended interpretation	Physical climate and process analysis, performance and scalability	Mainly performance and scalability analysis

2.1 Hardware and Software Environment

The results of numerical experiments reported in this section are performed on the Leonardo supercomputing platform at CINECA, one the the EuroHPC pre-exascale systems hosted at Bologna Technopole [29]. Leonardo comprises two main partitions tailored for different workloads: a Data-Centric General Purpose (DCGP) partition intended for CPU-only jobs, and a Booster partition equipped with GPUs for accelerated computing.

Each of the 1536 compute nodes on the DCGP partition consists of a dual socket 56-core 2.0 GHz Intel Sapphire Rapids. The Booster partition consists of 3456 compute nodes each equipped with single socket 32-core Intel CPU at 2.60 GHz and four NVIDIA A100 GPUs, delivering significantly higher floating-point performance suited for GPU-accelerated code. Both partitions share a low-latency 200Gb/s Dragonfly+ Infiniband interconnect and access to multi-tier high-capacity parallel storage. For the software stack making up the runtime environment, we list all the components in table 2.2 along with a corresponding short description.

Table 2.2: Software components and their purpose in the RegCM5 workflow.

Software Component	Purpose
NVHPC v24.5 (nvfortran/nvc)	Fortran/C compilers; enable OpenACC and DO CONCURRENT offloading.
HPC-X OpenMPI v2.19	MPI library for distributed-memory parallelism.
NetCDF (Fortran + C) v4.6.1 and v4.9.1	Model I/O for atmospheric and surface fields; for model output and restart files
Parallel-NetCDF v1.12.3	Optimized MPI-IO for parallel output.
HDF5	Backend for NetCDF-C structured datasets.
CUDA Toolkit v12.4	GPU backend and instrumentation support.
CLM4.5	Land surface coupling component.

2.2 Performance Benchmark of RegCM5 in RCE Configuration

The RCE configuration is activated in the RegCM5 framework at compile time via a dedicated preprocessor macro. The build is performed with the `RCEMIP` macro enabled through the compilation flag `CPPFLAGS += $(DEFINE)RCEMIP` defined in the `makeinc` file. This macro selects code paths intended for numerical experiments where the atmospheric condition is set in a state of idealized radiative-convective equilibrium.

At runtime, the RCE configuration is completed through an idealized namelist configuration specified in the files `isc24_small.in` and `profile.in`. We provided a line-by-line annotation in table 2.3 of the various options specified in each file and an overview of a typical runtime pipeline of RegCM5 in RCE configuration in figure 2.1.

Table 2.3: Key namelist parameters in the namelist files `isc24_small.in` and `profile.in` for RegCM5 configured in RCE.

Category	Parameter	Value; Description
Domain	`domname`	`IDEAL`; idealized, non-geographical domain
Grid	`jx`, `iy`	Horizontal grid size: 1024×512 grid points
Grid	`kz`	60 vertical levels
Grid	`iproj`	`NORMER`; idealized Cartesian projection
Grid	`ds`	Horizontal grid spacing: 2.0 km
Dynamics	`idynamic`	(`idynamic=3`); Nonhydrostatic dynamical core
Initial/boundary forcing	`iboudy`	0; no lateral boundary forcing/relaxation
RCE mode	`irceideal`	1; idealized radiative–convective equilibrium mode enabled
Surface forcing	`scenario`	`CONST`; constant surface forcing

Continued on next page

Category	Parameter	Value; Description
Radiation forcing	`ifixsolar`	1; fixed solar forcing
Radiation forcing	`fixedsolarval`	340 W m^{-2}; Constant insolation
Greenhouse gases	`ghg_year_const`	Constant GHG concentrations (year 1900)
Convection	`icup_lnd`, `icup_ocn`	0; convective parameterization disabled
Ocean fluxes	`iocnflx`	2; simplified ocean surface flux formulation
Time stepping	`dt`	30 s; Model time step
Output controls	`ifatm`, `ifrad`, `ifsrf`, `ifchem`	Disabled; no diagnostic output written
Restart/save	`ifrest`, `ifsave`	Disabled; no restart/save files generated
I/O	`do_parallel_netcdf_in`	Parallel NetCDF enabled for input
I/O	`do_parallel_netcdf_out`	Parallel NetCDF disabled for output
Auxiliary init	`profile.in`	Prescribes uniform surface state (`ps`, `ts`) and small perturbations for RCEMIP

Figure 2.2 summarizes the results of running RegCM5 in RCE configuration on a $512 \times 1024 \times 60$ grid for a single-day simulation run using various compute resource configurations on Leonardo. The CPU-only runs (blue bars) employed MPI domain decomposition across either 256 or 896 CPU cores on Booster and DCGP nodes, respectively. For all numerical experiments, the MPI ranks were mapped one-to-one onto physical CPU cores to avoid oversubscription of the CPU cores by multiple MPI processes.

The GPU-enabled configurations (green bars) utilized MPI in combination with OpenACC and Fortran `do concurrent` offloading on configurations involving 4, 8, and 16 NVIDIA A100 GPUs. Each GPU is bound exclusively to a single MPI rank. Device selection by each MPI rank is implemented at runtime via per-rank launch wrappers that dynamically set `CUDA_VISIBLE_DEVICES` based on the `SLURM_LOCAL_ID` when launching the application with `srun` or when using `mpirun`, `OMPI_COMM_WORLD_LOCAL_RANK`. This ensures that each rank held exclusive access to a unique accelerator in a setup where there is one-to-one mapping between CPU processes and GPUs. To build the binary, the `configure` script is invoked with the option:

```
$ ./configure --enable-openacc-stdpar
```

to enable device offloading via `do concurrent` and to enable `OpenACC`.

The results indicate that the GPU-enabled configurations significantly reduce time-to-solution relative to CPU-only execution, achieving speedups with respect to the 896 cores CPU-only baseline of 1.75×, 2.78×, and 3.63× when using 4, 8, and 16 GPUs, respectively. The increase in speedup is nearly linear in the number of GPUs indicating efficient strong scaling in this idealized configuration.

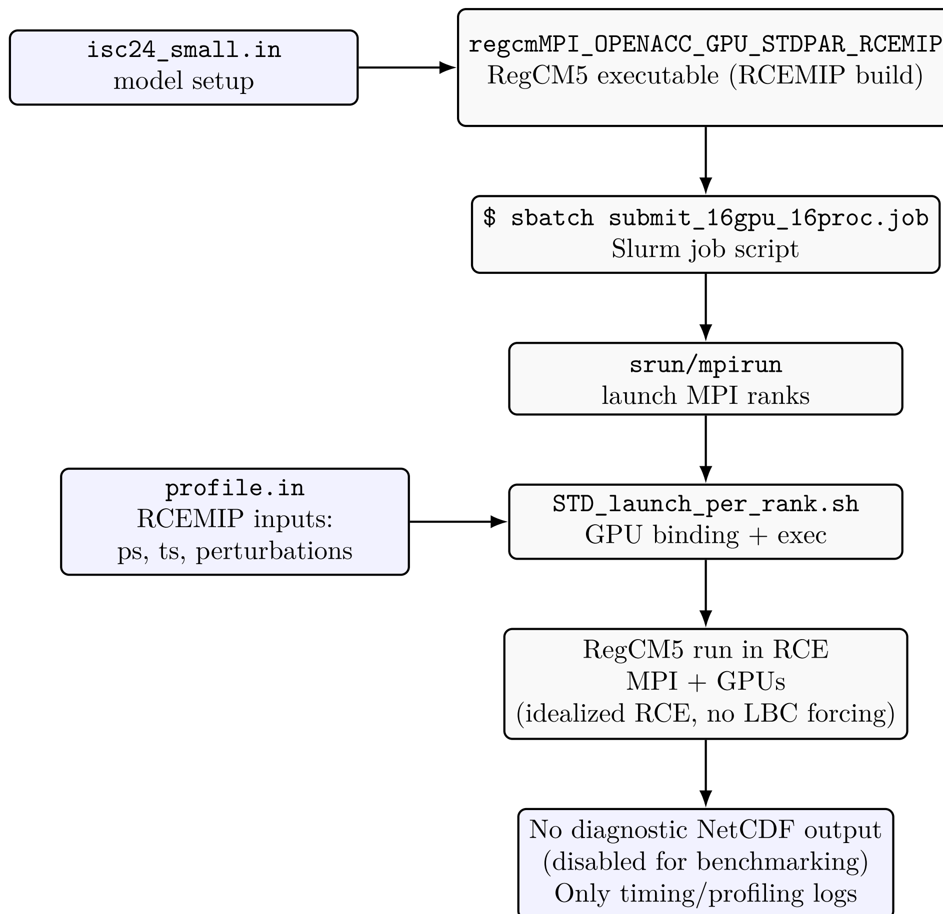


Fig. 2.1: Runtime pipeline for the RCE configuration of RegCM5. The run is configured by `isc24_small.in` and initialized by `profile.in` (uniform surface state and small perturbations). Diagnostic NetCDF output is disabled to isolate computational performance; only logs and profiling artefacts are produced.

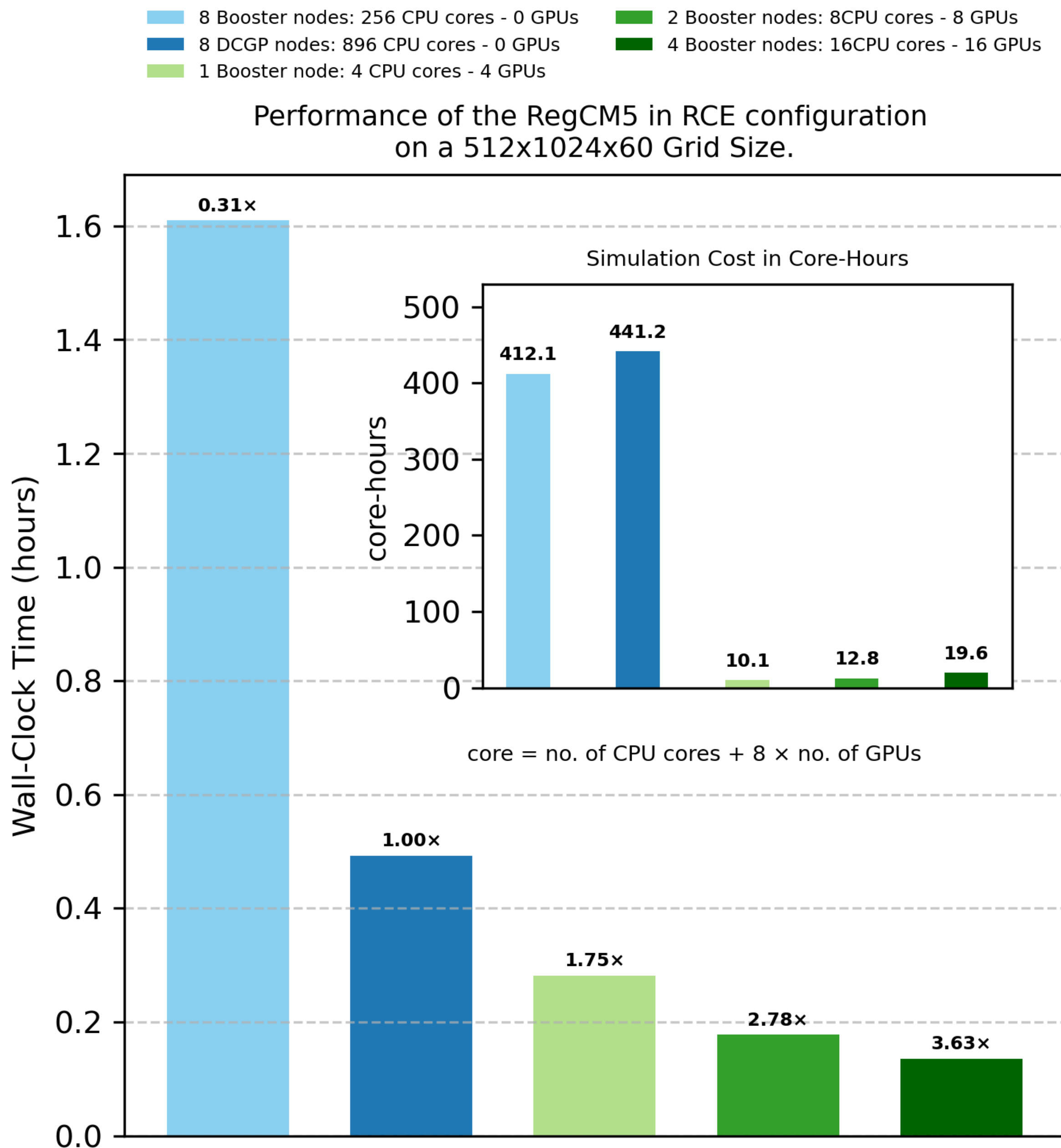


Fig. 2.2: Baseline performance of RegCM5 in the Radiative–Convective Equilibrium (RCE) configuration executed on a 512×1024×60 grid on Leonardo. CPU-only runs (blue) utilize MPI across 256 and 896 cores on Booster and DCGP partitions, respectively. GPU-enabled runs (green) combine MPI with OpenACC and Fortran `do concurrent` offloading to 4, 8, and 16 NVIDIA A100 GPUs. The inset panel reports the simulation cost in core-hours, defined as $(\text{CPU cores} + 8 \cdot \text{GPUs}) \times$ wall-clock time, demonstrating that GPU offloading provides substantial reductions in both wall-clock time and resource cost. The speedups are computed with respect to CPU-only run on 8 nodes of the DCGP partition using 896 CPU cores.

2.3 Performance Benchmark of RegCM5 in a coupled configuration with CLM4.5

In addition to the RCE configuration, we also performed a baseline benchmarking of the performance of RegCM5 in a coupled configuration with the Community Land Model, CLM4.5, on a European regional domain. In this configuration, the model is driven by external atmospheric forcing derived from the ERA5 reanalysis [30], which provides time-dependent initial and lateral boundary conditions.

In this coupled setup, ERA5 supplies the large-scale atmospheric state at the domain boundaries, while RegCM dynamically downscales these fields over the regional grid. CLM4.5 computes land–surface energy and water fluxes, which are exchanged with RegCM5 at each coupling time step. This configuration is designed for physically realistic regional climate simulations and stands in contrast to the RCE configuration, which deliberately removes external forcing and land–surface heterogeneity in order to isolate computational behavior of the atmospheric model.

Table 2.4 summarises the most relevant namelist settings defining the coupled RegCM5–CLM4.5 configuration used in this work, including the ERA5-driven boundary forcing, grid geometry, land–surface coupling, and I/O characteristics. Figure 2.3 gives a schematic view of a typical GPU-enabled pipeline for running RegCM5 in this coupled configuration.

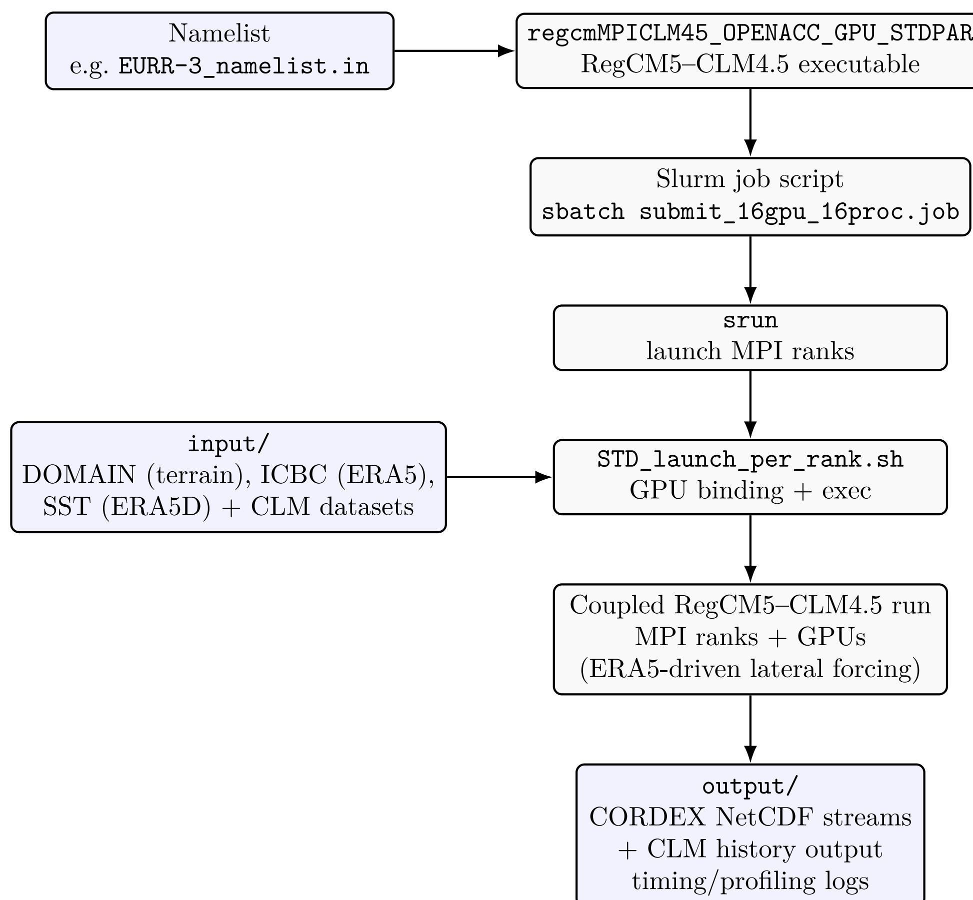


Fig. 2.3: Runtime pipeline for the coupled RegCM5–CLM4.5 configuration used in this work. The simulation is driven by ERA5 reanalysis (including ERA5 daily SST), launched via Slurm and a per-rank wrapper for GPU binding, and produces CORDEX-compliant atmospheric output as well as CLM history fields.

The binary for the coupled configuration is generated by invoking the `configure` script with `-enable-clm45` option:

```
$ ./configure --enable-clm45
```

As before, to enable support for OpenACC and offload via Fortran `do concurrent` construct in the GPU-enabled runs, the option `-enable-openacc-stdpar` is also passed.

Figure 2.4 summarizes the performance of the RegCM5 in the coupled configuration with CLM4.5 for various compute resource configuration and underlying parallel I/O strategy. Three I/O frameworks are tested: (i) Using NetCDF with HDF5 backend with parallel read and sequential write (green), (ii) parallel write (blue), and (iii) using PnetCDF library with parallel write (orange)[31]. For the latter case, the additional option `-enable-pnetcdf` is also passed when invoking the configure script. For all cases, switching between parallel and sequential read and write are done at runtime via the `do_parallel_netcdf_in` and `do_parallel_netcdf_out` namelist options, respectively.

A notable contrast emerges when comparing the performance of RegCM5 shown in figure 2.4 in the coupled configuration with that shown in figure 2.2 for the RCE configuration. In the idealized RCE configuration, 8 GPUs with 8CPU cores already outperform the CPU-only baseline which uses 896 CPU cores demonstrating the effectiveness of GPU-offloading. However this behavior does not carry over to the coupled case. In the RegCM5–CLM4.5 coupled configuration, the 8 GPU run remains slower than the 800-core CPU baseline, and accelerator benefits only appear at 32–64 GPUs. This is true regardless of the underlying I/O strategy used. Understanding this delay in the crossover point at which GPUs become advantageous motivated the profiling effort which is detailed in the next section.

Table 2.4: Key namelist parameters in the namelist file `EUR-3_namelist.in` for the coupled RegCM5–CLM4.5 simulation driven by ERA5 reanalysis on the EURR-3 domain.

Category	Parameter	Description / Value
Domain	`domname`	`EURR-3`; European regional domain
Grid	`jx`, `iy`	Horizontal grid size: 1003×1003 grid points
Grid	`kz`	50 vertical levels
Grid	`iproj`	Rotated latitude–longitude projection (`ROTLLR`)
Grid	`ds`	Horizontal grid spacing $\approx 0.03°$
Forcing	`dattyp`	`ERA5`; atmospheric initial and lateral boundary conditions
Forcing	`ssttyp`	`ERA5D`; daily ERA5 sea-surface temperatures
Forcing	`ibdyfrq`	Lateral boundary update frequency: 6 hours
Boundary relaxation	`nspgx`, `nspgd`	Sponge zone width: 31 grid points
Boundary relaxation	`high_nudge`, `medium_nudge`, `low_nudge`	Scale-dependent lateral relaxation strengths
Simulation period	`gdate1`, `gdate2`	ERA5 forcing window: 2009-09-01 to 2009-11-01
Land surface	CLM4.5 enabled	Interactive land–atmosphere coupling via CLM4.5
CLM output	`hist_nhtfrq`	CLM history output frequencies (multiple streams)
CLM diagnostics	`hist_fincl2`	Land-surface fluxes and state variables (e.g. latent/sensible heat, surface temperature)
Atmospheric output	`ifcordex`	CORDEX-compliant output enabled
I/O	`do_parallel_netcdf_in`	Parallel NetCDF enabled for input
I/O	`do_parallel_netcdf_out`	Parallel NetCDF disabled for output

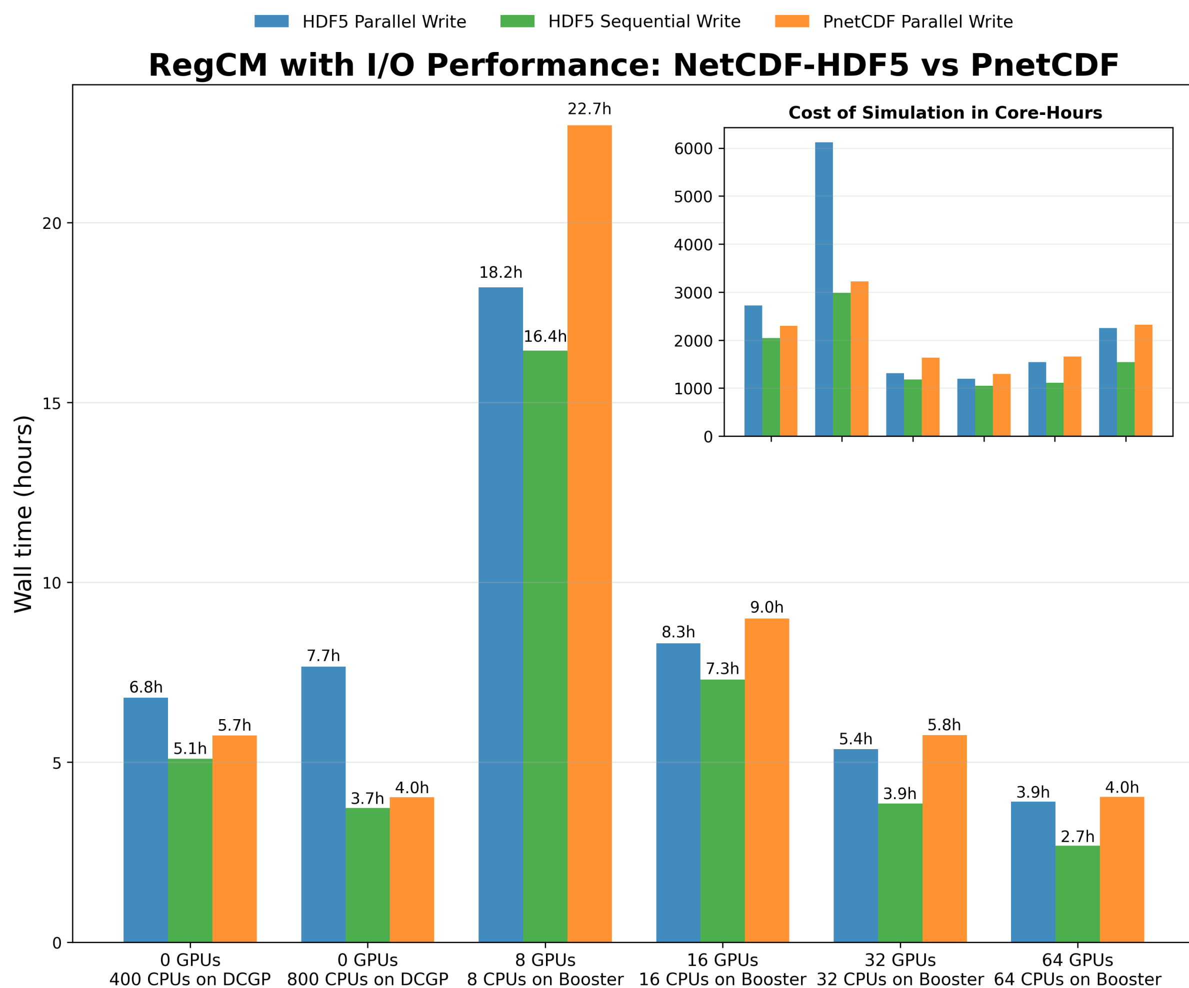


Fig. 2.4: Performance of the RegCM5–CLM4.5 coupled configuration on $1003 \times 1003 \times 50$ grid for 4 model days on Leonardo for different compute resource allocations and three I/O backends: HDF5 with sequential write (green), HDF5 with parallel write (blue), and PnetCDF with parallel write (orange). The reads are parallel for all cases. CPU-only baselines (0 GPUs) were executed on DCGP nodes using 400 and 800 CPU cores, while GPU-enabled runs used 8–64 A100 GPUs on Booster nodes. The coupled configuration exhibits reduced GPU performance compared with the RCE configuration, and only achieves lower wall-clock times and competitive core-hour costs at ≥ 32 GPUs. Inset shows total simulation cost in core-hours

2.4 Profiling

The goal of this section is to gain a clear insight into the observed degradation of relative performance of the GPU-enabled runs with respect to the CPU-only baseline in the coupled configuration shown in figure 2.4 when compared to the significant performance gain of GPU-accelerated runs in the RCE configuration in figure 2.2. We obtained a performance profile that helped us pinpoint various performance bottlenecks present in the coupled configuration which we subsequently optimized or, when possible, accelerated by offloading the relevant computations to the GPU.

Figures 2.5 and 2.6 are flame graph visualizations of the population of stack traces generated from running RegCM5 using 16 GPUs and 16 CPU processes on 4 Booster nodes on Leonardo using the `perf` profiling tool for the RCE and coupled configurations, respectively. Both graphs give an overall picture by showing the cumulative call stacks generated from all the CPU processes that participated in the run. The flame graphs constitute several columns each correspond to a distinct stack trace that are arranged in alphabetical order from left to right. From the bottom up, each frame in the stack column corresponds to subroutines in the user space, calls from external libraries, compiler-emitted symbols all the way up to low-level calls and functionalities living in the kernel space. More importantly, the width of each rectangular frame corresponds to the number of CPU samplings and hence the larger the share of the on-CPU time.

A major difference between these two initial performance profiles is the appearance of conspicuously wide stack plateaus in the coupled run in figure 2.6 which are absent in figure 2.5 for the RCE configuration. These are performance hotspots appearing under the symbol `__c_mset16_avx`, labeled `Hotspot 1` and a second call stack, labeled `Hotspot 2`, originating in the user space from subroutine `output` in the module `mod_output`. We dedicate the rest of the section to the treatment of the first hotspot and leave the second one for the next chapter.

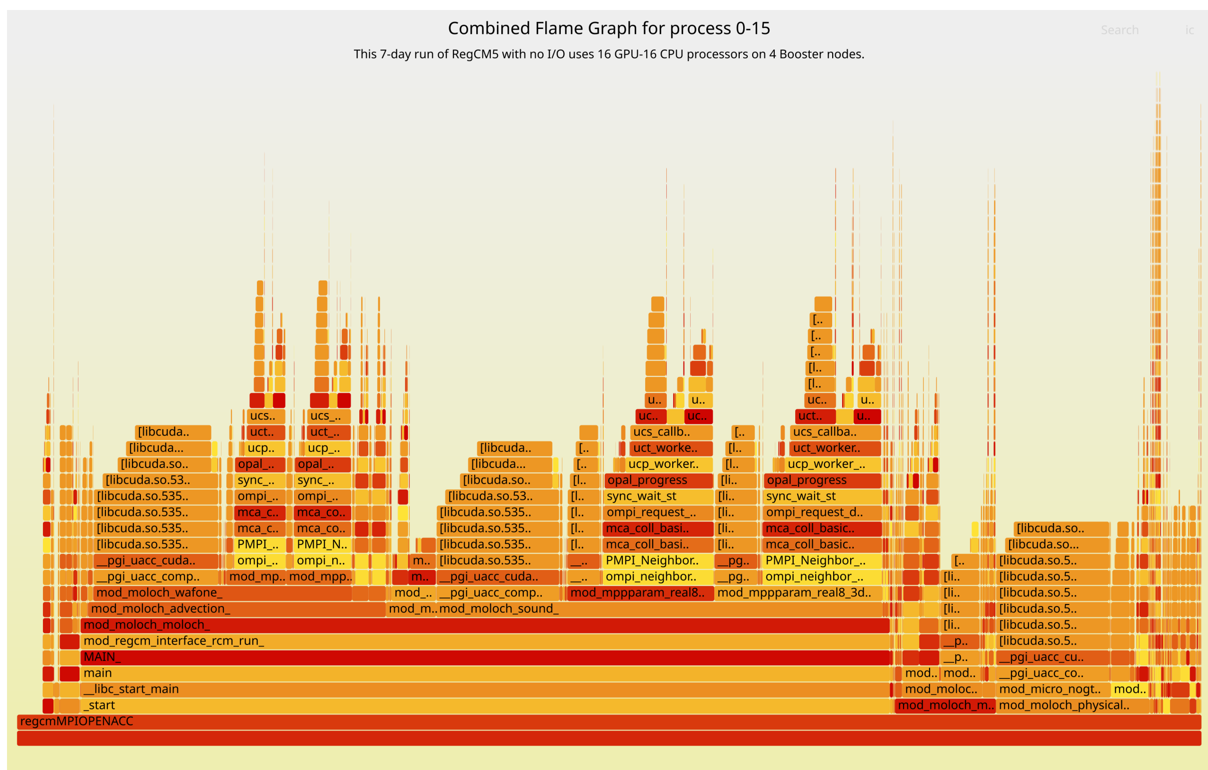


Fig. 2.5: Flame graph visualization of a population of call stacks for a 7-day simulation run on a $512 \times 1024 \times 60$ grid of the RegCM5 in RCE configuration using 16 GPU - 16 CPU processes which is 4 nodes on the Booster partition on Leonardo.

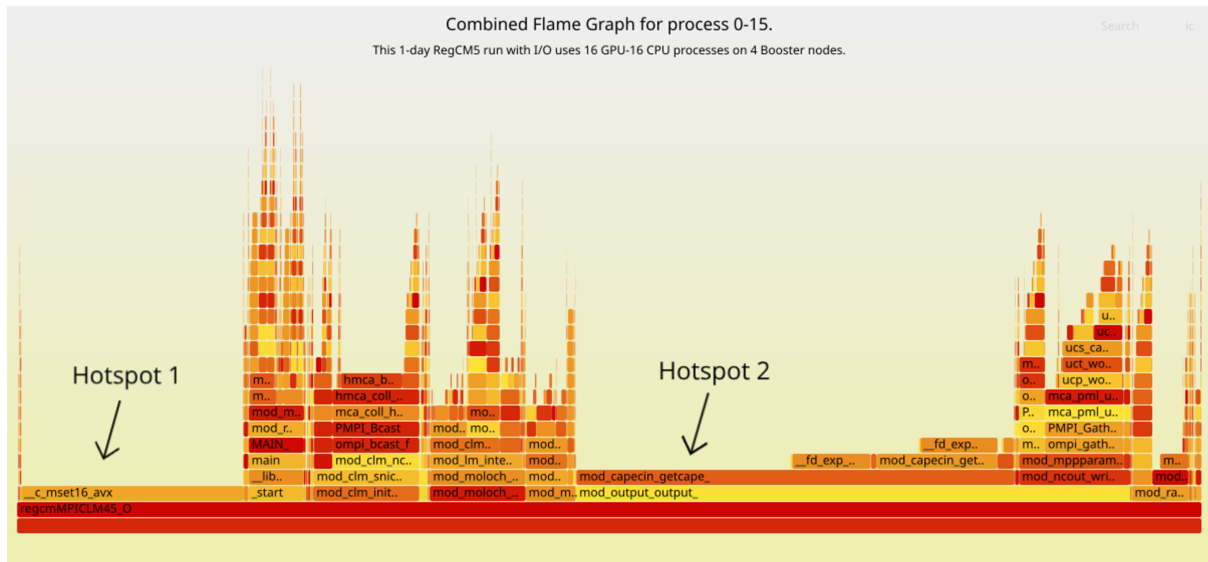


Fig. 2.6: Flame graph for the 1-day simulation run on a 1003 × 1003 × 50 grid of the RegCM5 coupled with the CLM4.5 using 16 GPU with 16 CPU processes which is 4 nodes on the Booster partition on Leonardo. Two conspicuous call stacks represent two main hotspots. The lone `__c_mset16_avx` frame constitutes 19% of the on-CPU time while the callstack starting on the `mod_capecin_getcape` frame constitutes 37%.

2.4.1 The `__c_mset16_avx` Hotspot.

Unlike `Hotspot 2`, the flame graph visualization tool did not expose a complete stack trace for `Hotspot 1` in figure 2.6. This makes it difficult to determine precisely the chain of procedure calls in the user space that could have triggered this symbol. In general, the appearance of a calling context especially for compiler-generated routines in sampling-based profilers are affected by several factors such as the profiling tool, compiler version, and optimization settings like inlining and frame-pointer omissions.

We successfully recovered the missing information by generating another performance profile of the same run using NVIDIA's NSight Systems profiling tool. Listing A.4 in the appendix A.3 gives the relevant profiling configuration with the NSight Systems in which we also used `perf` as a backend for generating the stack traces. The NSight tool traces the `__c_mset16_avx` symbol's machine code to the `NVHPC` compiler shared runtime library `libnvc.so` residing in the module path

```
/leonardo/prod/spack/06/install/0.22/linux-rhel8-icelake/
gcc-8.5.0/nvhpc-24.5-torlmnyzcexnrs6pq4cccabv7ehkv3xy/
Linux_x86_64/24.5/compilers/lib/libnvc.so
```

We confirmed that this symbol is indeed exported by this shared runtime library with the command:

```
$ nm -D libnvc.so | grep __c_mset16_avx
000000000001a370 T __c_mset16_avx
```

The presence of `libnvc.so` in the executable's runtime dependencies was also confirmed using `ldd` command,

```
$ldd ./regcmMPICLM45_OPENACC_GPU_STDPAR | grep -E "nvc|nvhpc|pg"
```

which confirms that the module path given above is correct.

NSight also managed to reconstruct several stack traces triggered by multiple subroutine calls across the RegCM5 source code all converging to `__c_mset16_avx` instruction:

```
__c_mset16_avx (35.6%)
 |
 +- mod_clm_hydrology2_hydrology2_ (0.00%)
 |    -> mod_clm_driver_clm_drv_ (0.00%)
```

```
|       -> mod_lm_interface_surface_model_ (0.00%)
|          -> mod_moloch_physical_parametrizations_ (0.00%)
|
+- mod_clm_slakehydrology_slakehydrology_ (0.00%)
|    -> mod_clm_driver_clm_drv_ (0.00%)
|       -> mod_clm_regcm_runclm45_ (0.00%)
|          -> mod_lm_interface_surface_model_ (0.00%)
|             -> mod_moloch_physical_parametrizations_ (0.00%)
|
+- mod_rad_interface_allocate_radiation_ (0.00%)
|    -> mod_params_param_ (0.00%)
|       -> mod_regcm_interface_rcm_initialize_ (0.00%)
|
+- mod_atm_interface_allocate_mod_atm_interface_ (0.00%)
|    -> mod_params_param_ (0.00%)
|       -> mod_regcm_interface_rcm_initialize_ (0.00%)
+- ...
```

Most of these code paths lead up to a set of subroutines residing in `mod_memutil.F90`, RegCM5's memory-management module, which are invoked during model initialization in both the RCE and coupled configurations. An example of such subroutine in RegCM5's memory management layer is given in listing 2.1.

This specific example manages dynamically allocated 3-D work arrays using a linked-list bookkeeping structure. If the pointer `a` is already associated, it is first released via `relmem3d(a)` to avoid memory leaks. The requested bounds are then packaged into a bounds array `b(1:3)` and passed to `getspc3d`, which allocates/reallocates the underlying storage `c3dd%a%space` with the desired extents. The caller's pointer `a` is then associated with this storage (`a => c3dd%a%space`), and the entire array is initialized to `d_zero` using whole-array assignments. Finally, a new list node is allocated (`c3dd%next`) and the tail pointer (`l3dd`) is advanced, preparing the allocator for subsequent requests.

Listing 2.1: RegCM memory utility `getmem3d_d` showing typed 3-D allocation, pointer association, and whole-array initialization (`a(:,:,:) = d_zero`) that the compiler may lower to an optimized memory-set routine (observed as `__c_mset16_avx` in profiling)

```
1  subroutine getmem3d_d(a,l1,h1,l2,h2,l3,h3,vn)
2      implicit none
3      real(rk8), pointer, contiguous,
4
5      dimension(:,:,:), intent(inout) :: a
6      integer(ik4), intent(in) :: l1, h1, l2, h2, l3, h3
7      character (len=*), intent(in) :: vn
8      type (bounds), dimension(3) :: b
9      if ( associated(a) ) call relmem3d(a)
10     b(1) = bounds(l1,h1)
11     b(2) = bounds(l2,h2)
12     b(3) = bounds(l3,h3)
13     c3dd => l3dd
14     call getspc3d(c3dd%a,b,ista)
15     call checkalloc(ista,__FILE__,__LINE__,vn)
16     a => c3dd%a%space
17     a(:,:,:) = d_zero
18     allocate(c3dd%next, stat=ista)
19     call checkalloc(ista,__FILE__,__LINE__,'c3dd%next')
20     l3dd => c3dd%next
21   end subroutine getmem3d_d
```

Because `a(:,:,:) = d_zero` is a whole-array assignment over a contiguous region, the compiler lowers it into a vectorized memset-like kernel for bulk memory initialization. This shifts the measured self-time from Fortran subroutines in the user space to compiler runtime in the profiling outputs. This

compiler lowering explains why NSight attributes 0.00% self-time for all the subroutines calling the array allocations and initialization functionalities.

In other words, since `__c_mset16_avx` is internal to NVHPC's runtime system and the compiler may omit standard stack frame information due to aggressive optimization, `perf` profiling tool is unable to reconstruct the calling context. As a result, the symbol appears in the flame graph as an isolated top-level hotspot, without attribution to a higher-level RegCM subroutine. The observed cost therefore reflects the cumulative time spent initializing or clearing large arrays during model execution, rather than time spent in a specific physical parameterization or dynamical kernel.

Since the `__c_mset16_avx` symbol only became prominent in the coupled configurations, we identified specific code paths in this stack that are only activated in the coupled configurations. We found such code paths involving the hydrology tracer routine `hydrology2` inside the module `mod_clm_hydrology2` as shown in the quoted call stack above. The subroutine contained several non-contiguous initializations executed over every row slices at each call.

```
...
snw_rds(c,:)       = 0._rk8
mss_bcpho(c,:)  = 0._rk8
mss_bcphi(c,:)  = 0._rk8
mss_bctot(c,:)  = 0._rk8
mss_cnc_bcphi(c,:) = 0._rk8
...
mss_dst1(c,:)      = 0._rk8
mss_dst2(c,:)      = 0._rk8
mss_dst3(c,:)      = 0._rk8
mss_dst4(c,:)      = 0._rk8
mss_dsttot(c,:)    = 0._rk8
mss_cnc_dst1(c,:)  = 0._rk8
mss_cnc_dst2(c,:)  = 0._rk8
...
```

Clearly, the compiler also lowers these assignments into the AVX memory set routine `_c__mset16_avx`. The contribution to the runtime due to this non-contiguous initialization pattern is considerable resulting in the `_c__mset16_avx` becoming prominent in the flame graph. After the refactor, the initializations is expressed as explicit loops over snow layers indexed as `j`, performing an element-wise initialization inside an OpenACC marked loop:

```
!$acc loop seq
do j = -nlevsno+1, 0
  snw_rds(c,j)       = 0._rk8
  mss_bcpho(c,j)     = 0._rk8
  mss_bcphi(c,j)     = 0._rk8
  mss_bctot(c,j)     = 0._rk8
  mss_cnc_bcphi(c,j) = 0._rk8
  ...
  mss_dst4(c,j)      = 0._rk8
  mss_dsttot(c,j)    = 0._rk8
  mss_cnc_dst1(c,j)  = 0._rk8
  mss_cnc_dst2(c,j)  = 0._rk8
  mss_cnc_dst3(c,j)  = 0._rk8
  mss_cnc_dst4(c,j)  = 0._rk8
  ...
end do
```

Figure 2.7 shows the resulting flamegraph of the same run under the same resource configuration after eliminating the `__c_mset16_avx` hotspot. Figure 2.8 shows a substantial reduction in wall-clock time, achieving a 1.77× speedup relative to the initial GPU configuration. However, at this stage the GPU-enabled run still does not surpass the performance of the CPU-only baseline running on 800 MPI processes.

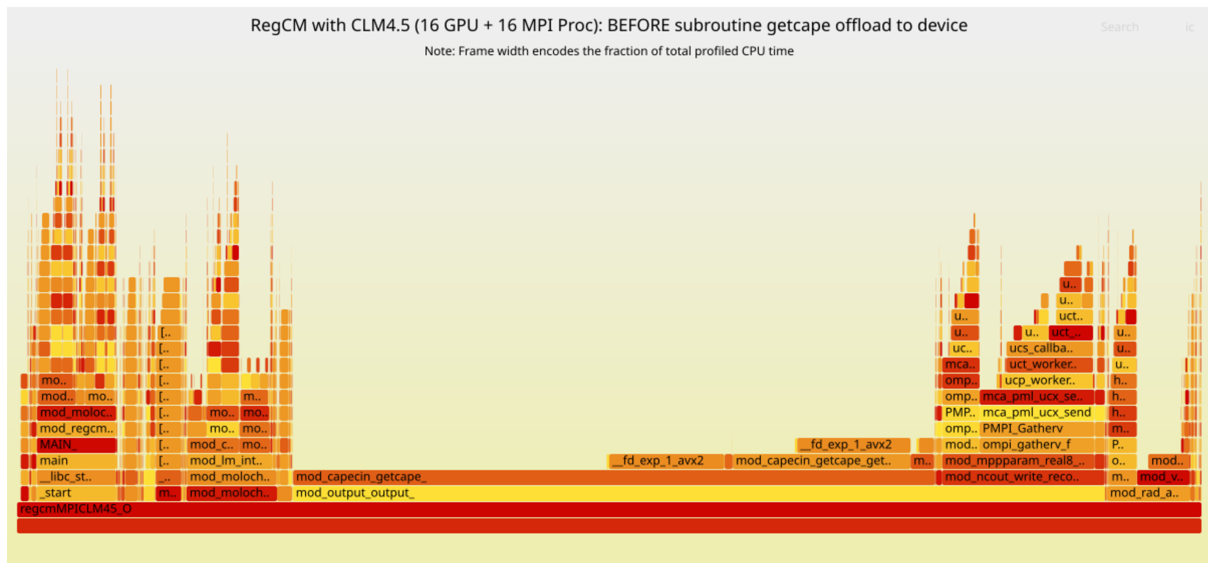


Fig. 2.7: Flame graph for the 1-day simulation run for the RegCM5 coupled with the CLM4.5 land model after removing the `hotspot 1`. The call site for the `getcape` subroutine in the subroutine `output` inside the module `mod_output` now takes up a significant 54% of the CPU sampling done by `perf`.

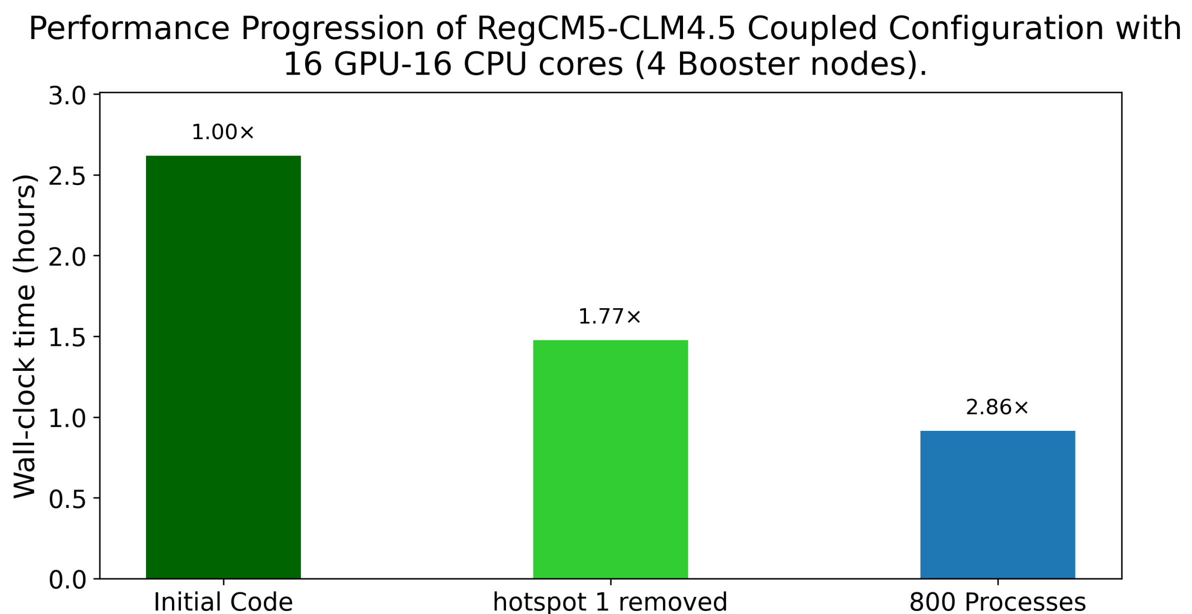


Fig. 2.8: Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run using 6 GPU + 16 CPU cores on (4 Booster nodes) on a 1003×1003×50 grid and a 1-day simulation. There is substantial performance improvements after removing, the `__c_mset16_avx` hotspot achieving up to 1.77× speedup relative to the initial performance. This GPU-enabled run is still outperformed by the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes.

Chapter 3

GPU porting of the getcape subroutine

In this chapter, we discuss the steps taken to successfully port the call site for the subroutine `getcape` defined in the module file `Share/mod_capecin.F90` to the GPU. This subroutine is used to calculate the Convective Available Potential Energy (CAPE) and the Convective Inhibition (CIN) from a sounding. Figure 2.7 is a flame graph which visualizes the call stack for a 1-day simulation of the RegCM coupled with CLM4.5 land model. The call site for the subroutine `getcape` takes up a significant 54% of the total profiled on-CPU time. This makes it an ideal region for offloading to the device.

As shown in listing 3.1, the call site for the `getcape` is within a triple-nested loop inside the subroutine `output` in the module `mod_output.F90` inside the `Main` directory. This nested computation is done entirely on the CPU. To successfully offload the call for `getcape` to the device, it needs to be refactored to help the compiler generate a corresponding device kernel. The host version of the `getcape` subroutine performs an allocation within the procedure for automatic arrays. We found that when the subroutine is offloaded as a kernel to the device, such automatic allocations are not supported. Hence, in each of the offloading strategies considered below, we allocated these arrays outside `getcape` and pass it to the subroutine as arguments.

Secondly, there were helper functions that are contained inside `getcape` which needs to be taken out to the module scope in order for the compiler to create their corresponding device kernels. Lastly, we removed the explicit public declaration of the `getcape` subroutine as this prevents it from being offloaded successfully as device kernel. The `getcape` subroutine and its helper functions are then annotated with the `OpenACC` directive `!$acc routine() seq` and the `pure` attribute was added since the standard specifies that only pure procedures can be called within a `do concurrent` region.

We employ three different strategies to distribute the two outer most loops among GPU threads: i.) Using OpenACC directives, ii.) Using `do concurrent` construct, and iii.) using `do concurrent` construct with an `OpenACC data` directive to control the data movement explicitly. They are shown, for the case of `idynamic == 3`, in listings 3.2, 3.3 and 3.5, respectively.

Listing 3.1: The call site for the subroutine `getcape`

```
if ( associated(srf_cape_out) .and. &

associated(srf_cin_out) ) then
  if ( idynamic == 3 ) then
    do i = ici1, ici2
      do j = jci1, jci2
        do k = 1, kz
          kk = kzp1 - k
          p1d(kk) = mo_atm%p(j,i,k)
          t1d(kk) = mo_atm%t(j,i,k)
          rh1d(kk) = min(d_one,max(d_zero,(mo_atm%qx(j,i,k,iqv) / &
              pfwsat(mo_atm%t(j,i,k),mo_atm%p(j,i,k)))))
        end do
        call getcape(kz,p1d,t1d,rh1d, &
          srf_cape_out(j,i),srf_cin_out(j,i))
```

```
16         end do
17       end do
18     else
19       do i = ici1, ici2
20         do j = jci1, jci2
21           do k = 1, kz
22             kk = kzp1 - k
23             p1d(kk) = atm1%pr(j,i,k)
24             t1d(kk) = atm1%t(j,i,k)/sfs%psa(j,i)
25             rh1d(kk) = min(d_one,max(d_zero, &
26                 (atm1%qx(j,i,k,iqv)/ps_out(j,i)) / &
27                 pfwsat(atm1%t(j,i,k)/ps_out(j,i),atm1%pr(j,i,k))))
28           end do
29           call getcape(kz,p1d,t1d,rh1d, &
30             srf_cape_out(j,i),srf_cin_out(j,i))
31         end do
32       end do
33     end if
34 end if
```

3.1 Offloading using OpenACC

The first strategy we used to offload the computation of the nested loops is with OpenACC directives. As shown in listing 3.2, we placed the nested loops inside a data region and used the `copyin` clause to allocate memory on the device for the arrays `mo_atm%p, mo_atm%t, mo_atm%qx` and copy their values from their host counterpart at the beginning of the data region. These arrays are not modified inside the compute region so that there is no need to update their host counterpart after exiting the data region.

Within the same data region, we then use the `create` clause to allocate memory on the device for the arrays `srf_cape_out`, `srf_cin_cout` and `p1d, t3d,rh3d piw, qw, tdw, thw, thvw, zw`. We then encapsulate the nested loops inside a `parallel` region and collapse the nested loops to gain a larger single iteration space. We then map work items to gangs of threads and implement a vector-level parallelization. Each GPU thread then executes instructions in the inner-most loop sequentially. In addition, we also declared the iteration variables `i,j,k` and the variables `cape_loc, cin_loc` local to each thread. Finally, before exiting the data region, we update the host counterpart of the the arrays `srf_cape_out, srf_cin_out` which are needed for some downstream computations.

Listing 3.2: GPU offloading of the `getcape` call site using OpenACC directives.

```
1  allocate(p3d(kz,jci1:jci2,ici1:ici2), t3d(kz,jci1:jci2,ici1:ici2), rh3d
     (kz,jci1:jci2,ici1:ici2))
2  allocate(piw (kz,jci1:jci2,ici1:ici2), tdw (kz,jci1:jci2,ici1:ici2), qw
       (kz,jci1:jci2,ici1:ici2))
3  allocate(thw (kz,jci1:jci2,ici1:ici2), thvw(kz,jci1:jci2,ici1:ici2), zw
       (kz,jci1:jci2,ici1:ici2))
4  if ( idynamic == 3 ) then
5        !$acc data copyin(mo_atm%p(jci1:jci2,ici1:ici2,1:kz),
6        !$acc mo_atm%t(jci1:jci2,ici1:ici2,1:kz), &
7        !$acc mo_atm%qx(jci1:jci2,ici1:ici2,1:kz,iqv:iqv)) &
8        !$acc create(srf_cape_out(jci1:jci2,ici1:ici2), &
9        !$acc srf_cin_out(jci1:jci2,ici1:ici2), p3d,t3d,rh3d,tdw,piw,qw,thw
            ,thvw,zw)
10       !$acc parallel loop collapse(2) gang vector private(i,j,k,kk,
            cape_loc,cin_loc)
11       do i = ici1, ici2
12         do j = jci1, jci2
13           do k = 1, kz
14             kk =(kz + 1) - k
15             p3d(kk,j,i) = mo_atm%p(j,i,k)
```

```
16            t3d(kk,j,i) = mo_atm%t(j,i,k)
17            rh3d(kk,j,i) = min(d_one,max(d_zero,(mo_atm%qx(j,i,k,iqv) / &
18              pfwsat(mo_atm%t(j,i,k),mo_atm%p(j,i,k)))))
19          end do
20          call getcape_gpu(p3d(:,j,i),t3d(:,j,i),rh3d(:,j,i),tdw(:,j,i),
               piw(:,j,i),qw(:,j,i),thw(:,j,i),thvw(:,j,i),zw(:,j,i),
               cape_loc,cin_loc)
21          srf_cape_out(j,i) = cape_loc
22          srf_cin_out (j,i) = cin_loc
23        end do
24      end do
25      !$acc update self(srf_cape_out, srf_cin_out)
26      !$acc end data
27      ...
28  end if
29  deallocate(p3d,t3d,rh3d,tdw,piw,qw,thw,thvw,zw)
```

3.2 Offloading with do concurrent

In this section, we rely on the `do concurrent` construct to offload to the device the call site for the `getcape` subroutine. The resulting refactored call site is given in listing 3.3. As before, we allocated 3-dimensional arrays `p3d,t3d,rh3d,piw,tdw,qw` and `thw,thvw,zw` and give contiguous column slices to each of the thread. We then rely on the `NVIDIA CUDA` managed/unified memory model to allocate their device counterparts. We then passed the contiguous column slices of these arrays as arguments to the `getcape` subroutine. Finally, we used the `local` clause to specify a private thread affinity for the variables `k,kk,cape_loc,cin_loc`.

Listing 3.4 shows the compiler output of a compilation instance of this offloaded region. We observed that in some instance, the `Nvfortran` compiler emitted an implicit `OpenACC copy` of the two dimensional arrays `srf_cape_out` and `srf_cin_cout` while in some instances the more optimal alternative `copyout` is emitted like that shown in Listing 3.4. The compiler also emitted implicit copies of the 3-dimensional arrays `t3d,p3d,rh3d,thvw,tdw,zw,thw,qw` instead of the optimal `copyout`.

Listing 3.3: GPU offloading of the `getcape` call site using `do concurrent`.

```
1   allocate(p3d(kz,jci1:jci2,ici1:ici2), t3d(kz,jci1:jci2,ici1:ici2), rh3d
       (kz,jci1:jci2,ici1:ici2))
2   allocate(piw (kz,jci1:jci2,ici1:ici2), tdw (kz,jci1:jci2,ici1:ici2), qw
        (kz,jci1:jci2,ici1:ici2))
3   allocate(thw (kz,jci1:jci2,ici1:ici2), thvw(kz,jci1:jci2,ici1:ici2), zw
        (kz,jci1:jci2,ici1:ici2))
4
5   if ( idynamic == 3 ) then
6   do concurrent(i = ici1:ici2, j = jci1:jci2) local(k,kk,cape_loc,
       cin_loc)
7     do k = 1, kz
8       kk = kzp1 - k
9       p3d(kk,j,i) = mo_atm%p(j,i,k)
10      t3d(kk,j,i) = mo_atm%t(j,i,k)
11      rh3d(kk,j,i) = min(d_one,max(d_zero,(mo_atm%qx(j,i,k,iqv) / &
12      pfwsat(mo_atm%t(j,i,k),mo_atm%p(j,i,k)))))
13    end do
14    call getcape_gpu(int(kz,ik4),p3d(:,j,i),t3d(:,j,i), rh3d(:,j,i),tdw
         (:,j,i),piw(:,j,i),qw(:,j,i),thw(:,j,i),thvw(:,j,i), &
15    zw(:,j,i), cape_loc,cin_loc)
16    srf_cape_out(j,i) = cape_loc
17    srf_cin_out(j,i) = cin_loc
18  end do
19  end if
```

```
20 deallocate(p3d,t3d,rh3d,tdw,piw,qw,thw,thvw,zw)
```

Listing 3.4: Compiler output.

```
1  1055, Generating NVIDIA GPU code
2  1055, Loop parallelized across CUDA thread blocks, CUDA threads(128)
       collapse(2) ! blockidx%x threadidx%x
3          ! blockidx%x threadidx%x auto-collapsed
4  1056, Loop run sequentially
5  1055, Generating implicit copyout(srf_cape_out(:,:)) [if not already
       present]
6          Generating implicit copy(t3d(:,:,:),p3d(:,:,:),piw(:,:,:),thvw
              (:,:,:),tdw(:,:,:),zw(:,:,:),thw(:,:,:),qw(:,:,:)) [if not
              already present]
7          Generating implicit copyin(mo_atm%p(:,:,:),mo_atm%t(:,:,:),mo_atm
              ,mo_atm%qx(:,:,:,1)) [if not already present]
8          Generating implicit copyout(srf_cin_out(:,:)) [if not already
              present]
9          Generating implicit copy(rh3d(:,:,:)) [if not already present]
10 1056, Complex loop carried dependence of mo_atm%p$p,mo_atm%t$p,mo_atm%
       qx$p prevents parallelization
11         Loop carried scalar dependence for k at line
              1057,1059,1060,1063,1061
12         Parallelization would require privatization of array p3d(:,:,:),
              t3d(:,:,:),rh3d(:,:,:)
13         Inner sequential loop scheduled on accelerator
```

3.3 Offloading with do concurrent and OpenACC

Here we augment the implementation in the previous section by not relying on CUDA unified memory for the implicitly managing the data movement between the host and device. This is done by enclosing the `do concurrent` construct within an `OpenACC data` region and use the `create` clause to explicitly prevent the dynamically allocated arrays from being copied back to the host at the end of the compute region.

Listing 3.5: GPU offloading of the `getcape` call site using `do concurrent` with `OpenACC` data directive.

```
1  allocate(p3d(kz,jci1:jci2,ici1:ici2), t3d(kz,jci1:jci2,ici1:ici2), rh3d
       (kz,jci1:jci2,ici1:ici2))
2  allocate(piw (kz,jci1:jci2,ici1:ici2), tdw (kz,jci1:jci2,ici1:ici2), qw
        (kz,jci1:jci2,ici1:ici2))
3  allocate(thw (kz,jci1:jci2,ici1:ici2), thvw(kz,jci1:jci2,ici1:ici2), zw
        (kz,jci1:jci2,ici1:ici2))
4
5  if ( idynamic == 3 ) then
6  !$acc data create(p3d,t3d,rh3d,tdw,piw,qw,thw,thvw,zw)
7      do concurrent(i = ici1:ici2, j = jci1:jci2) local(k,kk,cape_loc,
          cin_loc)
8        do k = 1, kz
9          kk = kzp1 - k
10         p3d(kk,j,i) = mo_atm%p(j,i,k)
11         t3d(kk,j,i) = mo_atm%t(j,i,k)
12         rh3d(kk,j,i) = min(d_one,max(d_zero,(mo_atm%qx(j,i,k,iqv) / &
13             pfwsat(mo_atm%t(j,i,k),mo_atm%p(j,i,k)))))
14       end do
15       call getcape_gpu(int(kz,ik4),p3d(:,j,i),t3d(:,j,i),rh3d(:,j,i),&
16         tdw(:,j,i),piw(:,j,i),qw(:,j,i),thw(:,j,i),thvw(:,j,i),zw(:,j,i
              ), &
17         cape_loc,cin_loc)
```

```
18
19          srf_cape_out(j,i) = cape_loc
20          srf_cin_out(j,i) = cin_loc
21      end do
22 !$acc end data
23      ...
24 end if
```

3.4 Post-offload profiling and benchmarking

After successfully offloading the call site for the `getcape` subroutine, we did another round of profiling and benchmarking. The resulting CPU flame graph is shown in figure 3.1 for a 1-day simulation run on a $1003 \times 1003 \times 50$ grid of RegCM5 in a coupled configuration with CLM4.5 using 16 GPUs and 16 CPU cores. The call stack involving the `getcape` subroutine no longer makes an appearance on the profiled CPU time since the relevant computation has already been offloaded to the device. In the absence of the previous two major hotspots, we to noticed new stack plateaus, labeled `Hotspot 3` and `Hotspot 4`, coming to light in the flame graph. `Hotspot 3` starts with the call to the subroutine `read_aeroppdata` from the module `mod_rad_aerosol.F90`. Walking the call stack leads to the subroutine `interp1d_r8` from the module `mod_interp.F90` which by itself takes up 12% of the total on-CPU time. The call site for `interp1d_r8` is inside the subroutine `intp1_pointer` from `mod_vertint.F90`. The call site is inside a double-nested loop as shown in listing 3.6 where the call to the generic interface `interp1d` on line 27 resolves to the `interp1d_r8` subroutine.

Similarly, `Hotspot 4` is attributed to the call site of the function `heatindex` living in the module `mod_heatindex`. The call site is a double-nested loop inside the subroutine `collect_output` from the module `mod_lm_interface`. The next section is dedicated to the treatment and elimination of these two new hotspots. As before we track the subsequent performance gain resulting from the recent GPU offload. Figure 3.2 shows significant performance boost of $4.96\times$ speedup in time-to-solution relative to the starting setup when both hotspots are present. Moreover, this run using 16 GPU with 16 CPU core has already surpassed CPU-only baseline running on 800 CPU cores with a $4.96/2.86 \approx 1.73\times$ speedup.

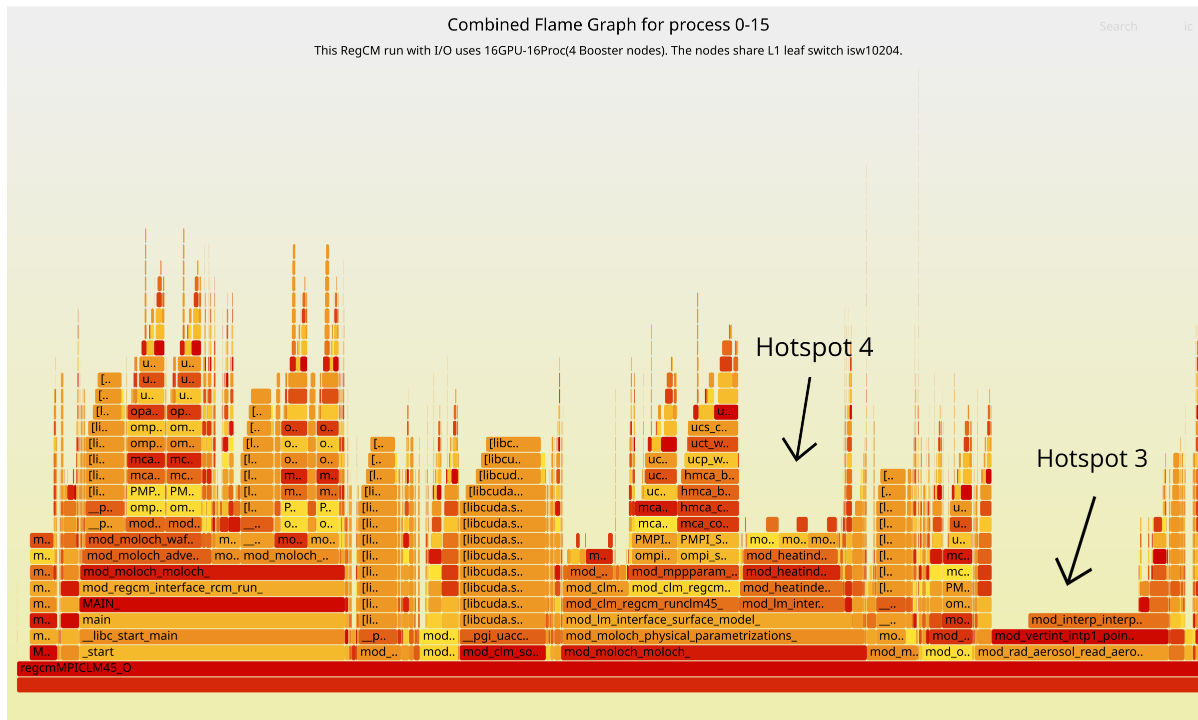


Fig. 3.1: Flame graph for the 1-day simulation run for the RegCM5 coupled with the CLM4.5 after offloading `getcape` subroutine using OpenACC. The resource configuration for this run is 16 GPUs with 16 CPU processes using 1003 × 1003 × 50 grid size. In the absence of the previous hotspots, new dominant stack plateaus labeled `Hotspot 3` taking up about 11% of the on-CPU time and `Hotspot 4` taking up 8%, both come to light.

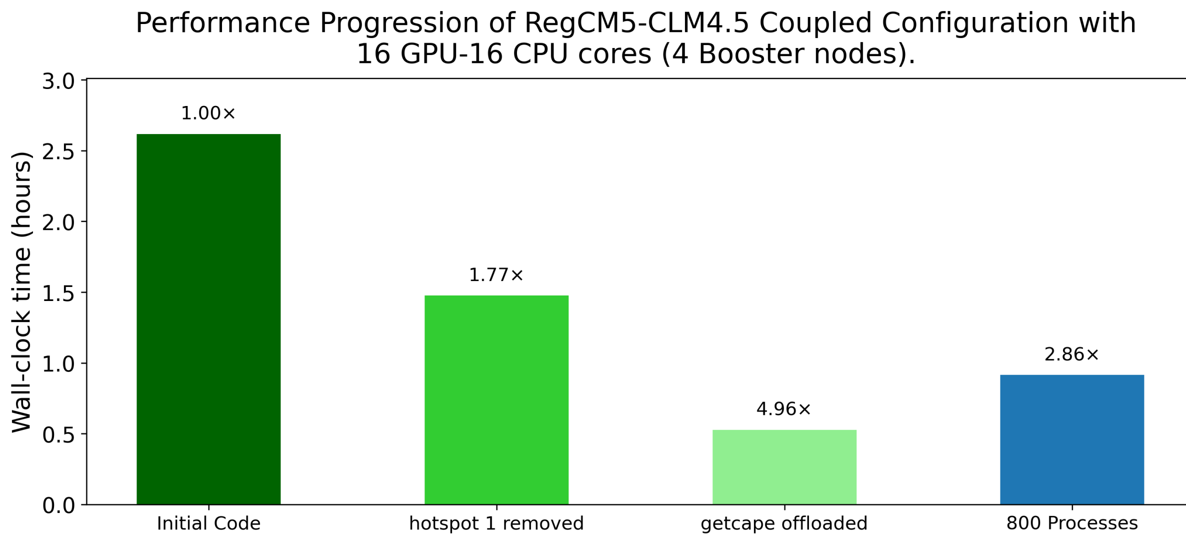


Fig. 3.2: Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run (16 GPU + 16 CPU cores on 4 Booster nodes) on a 1003×1003×50 grid on per 1 model day. Further substantial performance improvement is gained after offloading the call site for `getcape` subroutine, achieving up to 4.96× speedup relative to the initial performance. This GPU-enabled run now outperforms the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes.

Listing 3.6: The call for the general interface `interp1d` resolves to the subroutine `interp1d_r8`. The call site is situated inside the double loop within the subroutine `intp1_pointer`.

```
subroutine intp1_pointer(frcm,fccm,prcm,pccm,i1,i2,j1,j2,krcm,kccm,a,e1
    ,e2)
  implicit none
  integer(ik4), intent(in) :: kccm, krcm, i1, i2, j1, j2
  real(rkx), intent(in) :: a, e1, e2
  real(rkx), pointer, contiguous, dimension(:,:,:), intent(in) :: fccm,
      pccm
  real(rkx), pointer, contiguous, dimension(:,:,:), intent(in) :: prcm
  real(rkx), pointer, contiguous, dimension(:,:,:), intent(inout) ::
     frcm
  real(rkx), dimension(kccm) :: xc, fc
  real(rkx), dimension(krcm) :: xr, fr
  integer(ik4) :: i, j, k, kt, kb
  if ( pccm(i1,j1,1) > pccm(i1,j1,kccm) ) then
    kt = kccm
    kb = 1
  else
    kt = 1
    kb = kccm
  end if
  do j = j1, j2
    do i = i1, i2
      do k = 1 , kccm
        xc(k) = (pccm(i,j,k)-pccm(i,j,kt))/(pccm(i,j,kb)-pccm(i,j,kt))
        fc(k) = fccm(i,j,k)
      end do
      do k = 1 , krcm
        xr(k) = (prcm(i,j,k)-pccm(i,j,kt))/(pccm(i,j,kb)-pccm(i,j,kt))
      end do
      call interp1d(xc,fc,xr,fr,a,e1,e2)
      do k = 1 , krcm
        frcm(i,j,k) = fr(k)
      end do
    end do
  end do
end subroutine intp1_pointer
```

3.5 Removing Hotspot 4 and 5

In this section, we give the details for offloading using OpenACC the call sites for `interp1d_r8` and `heatindex` corresponding to `Hotspots` 3 and 4, respectively. As in the case for the `getcape`, we had to modify the subroutine `interp1d_r8` and pass scratch arrays to it instead of doing an internal allocation for automatic arrays. The offloaded call site inside the `intp1_pointer` subroutine is shown in listing 3.7. The scratch arrays `zi` and `zg` are defined in line 6. We encapsulated the triple nested loop inside a `OpenACC parallel` region. The arrays `fccm, pccm, prcm` and `frcm` are initialized externally and are passed to `intp1_pointer` as input arguments. Hence, we used the `copyin` clause to copy them to their mapped counterparts on the device. Only the array `frcm` is needed at the end of the compute region so we used `copy` to update their host counterpart.

Listing 3.7: Offloading the call site for `interp1d` using OpenACC. We passed scratch arrays to the `interp1d` subroutine as an alternative to the local allocations of automatic arrays.

```


subroutine intp1_pointer(frcm,fccm,prcm,pccm,i1,i2,j1,j2,krcm,kccm,a,e1
    ,e2)
```

```
...
! Scratch arrays
real(rkx), dimension(kccm) :: zi, zg
...
  !$acc parallel loop gang vector collapse(2)  &
  !$acc copyin(fccm,pccm,prcm) copy(frcm) &
  !$acc private(xc,fc,xr,fr,zi,zg)
  do j = j1, j2
    do i = i1, i2
      do k = 1 , kccm
        xc(k) = (pccm(i,j,k)-pccm(i,j,kt))/(pccm(i,j,kb)-pccm(i,j,kt))
        fc(k) = fccm(i,j,k)
      end do
      do k = 1 , krcm
        xr(k) = (prcm(i,j,k)-pccm(i,j,kt))/(pccm(i,j,kb)-pccm(i,j,kt))
      end do
      call interp1d_r8_gpu(xc,fc,xr,fr,a,e1,e2,zi,zg)
      do k = 1 , krcm
        frcm(i,j,k) = fr(k)
      end do
    end do
  end do
!$acc end parallel loop
end subroutine intp1_pointer
```

Similarly, the double-nested loop containing the call site of `heatindex` inside the subroutine `collect_output` from the module `mod_lm_interface` is given in listing 3.8. Offloading the call site is less involved than that of `intp1_pointer`. We simply annotated the regions with the `parallel loop` directive and with the `collapse` clause to map the loop iterations across `gang` and `vector`. In addition, we also annotated all the procedures that are called inside the parallel region with `$acc routine seq` for the compiler to generate their corresponding device-callable counterparts. From call stack in the flame graph 3.1, these procedures are `heatindex`, `find_eqvar`, `find_t`, `solveii`, `initial_find_eqvar`, `solve1`, `solve2`, `solve3`, `ra`, and `ra_bar` all living inside the module `mod_heatindex`.

Listing 3.8: Offloading the call site for `heatindex` using OpenACC.

```
!$acc parallel loop collapse(2) gang vector
do i = ici1, ici2
    do j = jci1, jci2
        tas = sum(lms%t2m(:,j,i))*rdnnsg
        ps = sum(lms%sfcp(:,j,i))*rdnnsg
        qs = pfwsat(tas,ps)
        qas = lm%q2m(j,i)
        rh = min(max((qas/qs),d_zero),d_one)
        srf_htindx_out(j,i) = heatindex(tas,rh)
    end do
end do
```

Listing 3.9 shows the NVHPC compiler feedback for the kernel for the parallel region. The compiler reports successful GPU code generation, collapses the two-dimensional loop into a single gang-level iteration space, and introduces implicit reductions for the vertical sums. The diagnostic also reveals implicit copyin/copyout of several derived-type components, indicating that the arrays are not yet resident on the device at the entry to this region

Listing 3.9: NVHPC OpenACC compiler diagnostic for the `srf_htindx_out` loop

```
1105, Generating implicit private(ici1,ici2,jci1,jci2)
      Generating NVIDIA GPU code

1106, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
1107,       ! blockidx%x threadidx%x collapsed
```

```
6  1108, !$acc loop seq
7
8  1109, Generating implicit reduction(+:t2m$r479)
9        !$acc loop seq
10       Generating implicit reduction(+:sfcp$r480)
11
12 1105, Generating implicit copyout(srf_htindx_out(jci1:jci2,ici1:ici2))
13       [if not already present]
14       Generating implicit copyin(
15         lms%t2m(:,jci1:jci2,ici1:ici2),
16         lm%q2m(jci1:jci2,ici1:ici2),
17         lms%sfcp(:,jci1:jci2,ici1:ici2),
18         lms
19       ) [if not already present]
```

3.6 Post-offload Profiling and Benchmarking

The resulting flame graph visualization of the stack traces for the RegCM5 coupled configuration with CLM4.5 after removing `Hotspot 3` and `4` is given in figure 3.3. As before, this run uses 16 GPUs with 16 CPU cores. After eliminating the hotspots appearing as wide stack plateaus that were initially present, the resulting flame graph for the coupled configuration is now more compact similar to figure 2.5 for the RCE configuration. We also updated the plot for the performance progression in figure 3.4 and added the same 1-day simulation run on $1003 \times 1003 \times 50$ after offloading the call site for `interp_1d`. We achieved further performance gain from $4.96\times$ to $7.10\times$ speedup relative to the initial code version. Relative to the CPU-only benchmark running on 800 processes, the GPU enabled run with 16 GPUs and 16 CPU processes gained an advantage of $7.10/2.86 = 2.48\times$ speedup.

The performance progression documented in figure 3.4 corresponds exclusively to the offloading of the call site for `getcape` with OpenACC. In figure 3.5, we also plot the performance of the coupled configuration of the RegCM5 using 16 GPUs and 16 CPU processes after offloading the `getcape` subroutine using `do concurrent` and the hybrid scheme where we used OpenACC to control the data movement within the `do concurrent` construct. In general, the performance gains from all three schemes are comparable with each other.

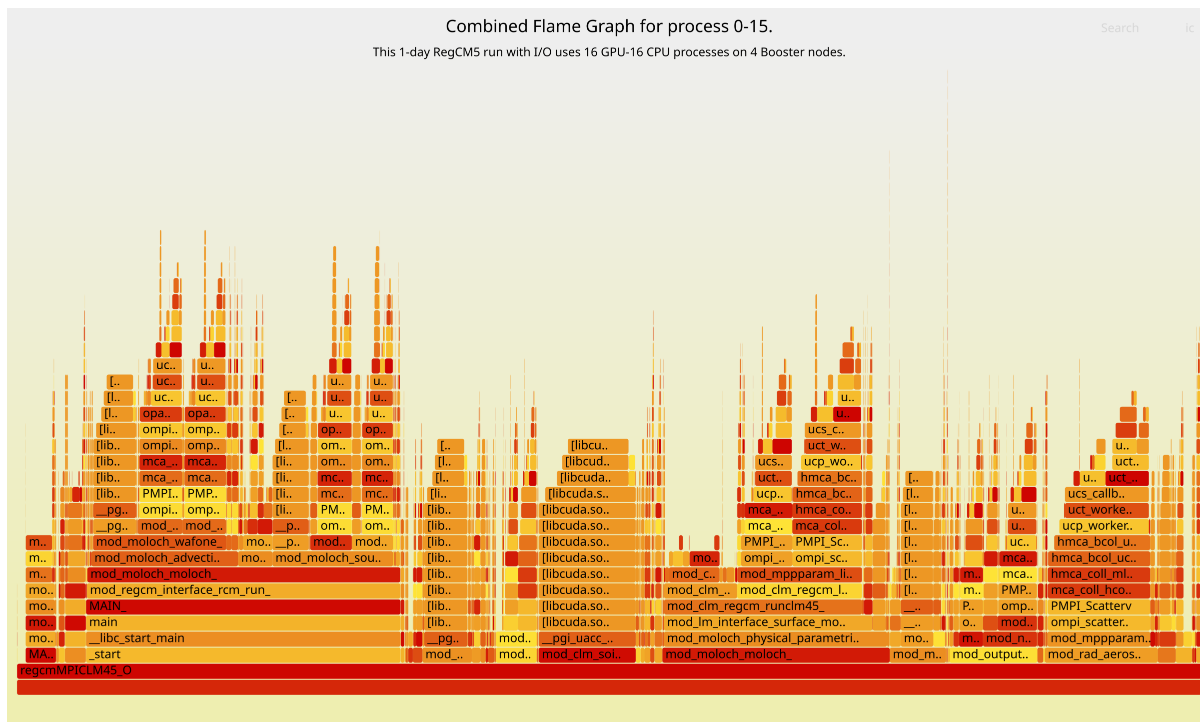


Fig. 3.3: Flame graph for the 1-day simulation run on a $1003 \times 1003 \times 50$ grid of the RegCM5 in a coupled configuration with the CLM4.5 after offloading the call site for `interp1d_r8` and `heatindex` subroutines. The resource configuration for this run is 16 GPU with 16 CPU cores which is 4 nodes on the Booster partition on Leonardo. Visually this flame graph appears more compact compared to that in figure 3.1.

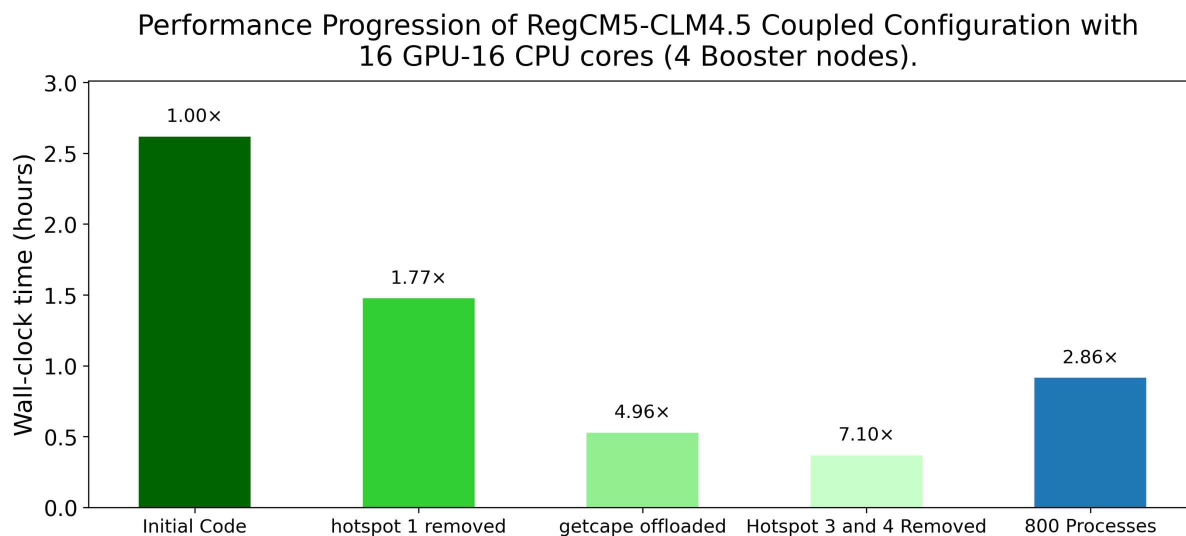


Fig. 3.4: Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run using 16 GPUs with 16 CPU cores on 4 Booster nodes on a $1003 \times 1003 \times 50$ grid per 1 model day. Further performance improvement is gained after offloading the call sites for `interp1d` and `heatindex` subroutines, achieving up to $7.10\times$ speedup relative to the initial performance. This GPU-enabled run now outperforms the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes.

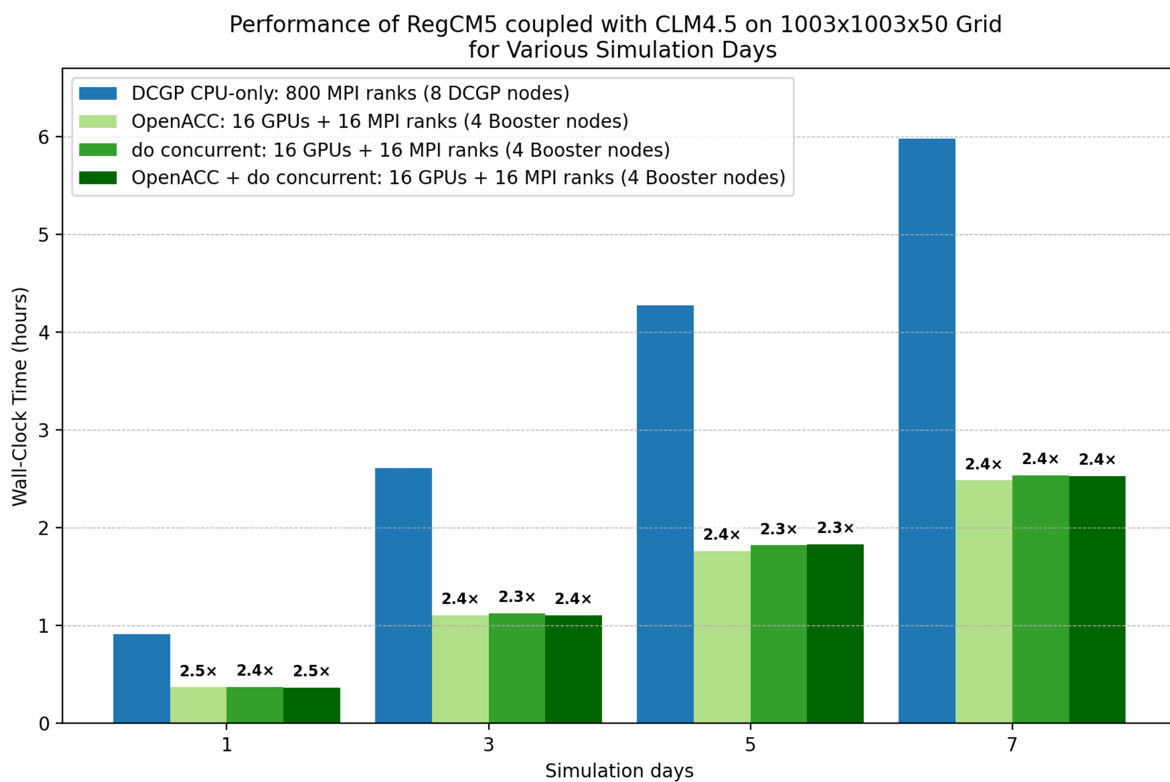


Fig. 3.5: Performance for the coupled RegCM5–CLM4.5 simulations on a 1003×1003×50 grid across varying simulation lengths (1–14 model days). The CPU-only baseline uses 800 MPI ranks on 8 DCGP nodes, while GPU-enabled runs use 16 GPUs and 16 MPI ranks on 4 Booster nodes. Three GPU implementations are compared: (i) OpenACC, (ii) Fortran do concurrent, and (iii) the combined OpenACC + do concurrent configuration. Annotated values denote speedup factors relative to the CPU-only baseline for each duration.

Chapter 4

Performance Portability

The performance progression reported in figure 3.4 and the performance comparisons given in figure 3.5 against the CPU-only baseline at the end of the previous chapter summarize the performance boost achieved from implementing on Leonardo the various GPU porting strategies and optimizations we've discussed so far. The current version of the RegCM5 code base have also undergone further optimization and GPU porting that we do not have the space to discuss in detail in the current work. The resulting performance gain achieved from these continued porting and optimization is summarized in figure 4.1. The final code version has achieved a 14.17× speedups relative to initial version in wallclock time per simulation day on a $1003 \times 1003 \times 50$ grid in the coupled configuration.

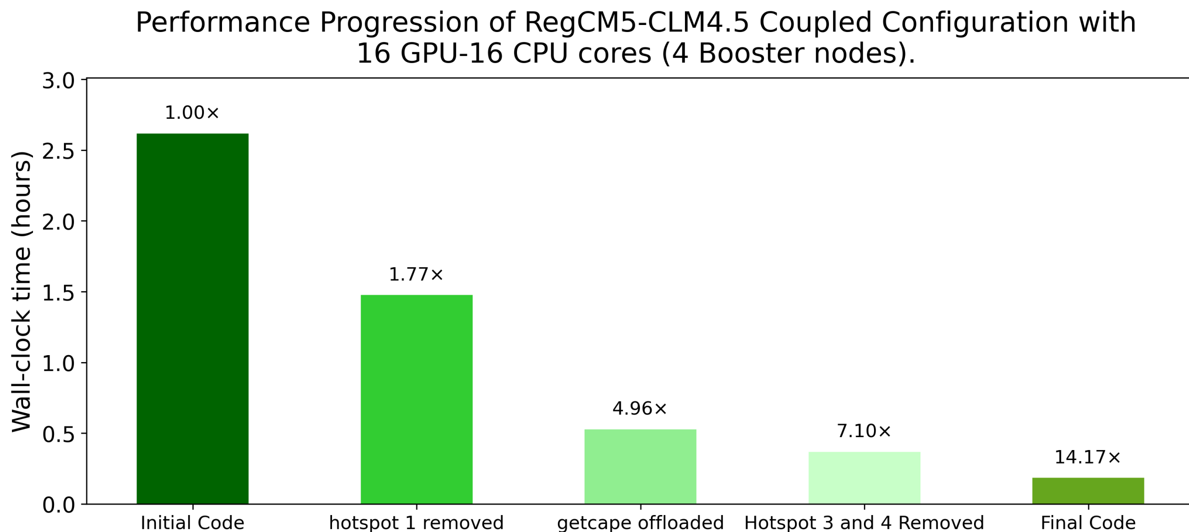
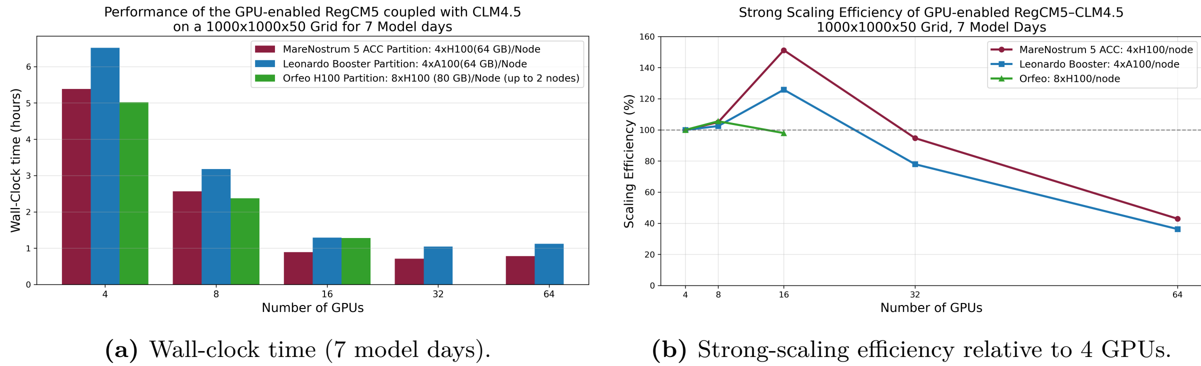


Fig. 4.1: Full performance progression of the RegCM5-CLM4.5 coupled configuration on 4 Booster nodes (16 GPUs and 16 CPU cores) on $1003 \times 1003 \times 50$ grid per 1 simulation day. The final implementation achieves a 14.17× speedup relative to the initial code.

In addition, we also present in this chapter an investigation of the performance portability of the current version of RegCM5 for the case of the GPU-enabled, RegCM5-CLM4.5 coupled configuration across various HPC platforms. We run the coupled configuration on the Accelerated (ACC) partition of MareNostrum 5 platform, another EuroHPC platform hosted on Barcenola Supercomputing Center [32, 33] , and on the H100 partition of the Open Research Facility for Epigenomics and Other (ORFEO) data center hosted within the AREA Science Park [34]. The differences in theses platforms in terms of compute and network specifications is summarized in table 4.1

The results are summarized as a strong scaling plot in figure 4.2-a for all three platforms and the corresponding efficiency plots given in figure 4.2-b. For this experiment, we run the model on $1003 \times 1003 \times 50$ grid for 7 model days. Figure 4.2 shows that the coupled RegCM5–CLM4.5 configuration benefits significantly from increasing the number of GPUs from 4 to 16, with the largest gains observed between 4 and 8 GPUs. ORFEO consistently provides the lowest wall-clock time at small scale, whereas



(a) Wall-clock time (7 model days).

(b) Strong-scaling efficiency relative to 4 GPUs.

Fig. 4.2: Strong-scaling performance of the GPU-enabled RegCM5–CLM4.5 coupled configuration on a 1003×1003×50 grid for 7 model days across MareNostrum 5 ACC, Leonardo Booster, and ORFEO H100. (a) Wall-clock time as a function of GPU count. (b) Strong-scaling efficiency relative to the 4-GPU case. All systems show substantial speedup from 4 to 16 GPUs, with apparent superlinear scaling on MareNostrum 5 and Leonardo. Beyond this range, efficiency degrades as communication overheads become increasingly significant

Leonardo is the slowest among the three platforms. The corresponding efficiency plot shows apparent superlinear scaling on MareNostrum 5 and Leonardo up to 16 GPUs, with efficiencies exceeding 100% relative to the 4-GPU baseline. A thorough investigation and profiling work is needed to fully track the origin of this behavior but is likely due to improved decomposition and memory-system effects rather than a true change in algorithmic complexity. After 16 GPUs, efficiency declines quickly, especially on MareNostrum 5 and Leonardo, indicating that communication and coupling overheads become increasingly dominant. Thus, while larger GPU counts still reduce time-to-solution, the marginal benefit decreases substantially beyond 16 GPUs for this fixed problem size

Table 4.1: Node-level hardware characteristics of the three GPU platforms used in this work. Specifications were verified via direct node introspection tools (`nvidia-smi`, `lscpu`, `ibv_devinfo`) and official documentation.

Component	**MareNostrum-5 ACC**	**Leonardo Booster**	**ORFEO H100**
Node model	BullSequana SH	Bull Sequana XH2000	Dell PowerEdge XE9680
CPU model	2× Intel Xeon Platinum 8460Y+	1× Intel Xeon Platinum 8358	2× Intel Xeon Platinum 8480+
CPU cores/node	80 (160 threads)	32 (1 thread/core)	112
NUMA domains	4	2	8
System memory	512 GB DDR5	512 GB DDR4	1 TB DDR5
GPUs/node	4× NVIDIA H100	4× NVIDIA A100	8× NVIDIA H100
GPU memory	64 GB HBM2e	64 GB HBM2e	80 GB HBM3
GPU interconnect	NVLink mesh	NVLink mesh	NVSwitch + NVLink
PCIe generation	Gen5 ×16	Gen4 ×16	Gen5 ×16
Network adapters	4× ConnectX-7	4× ConnectX-6	8× ConnectX-7
Network generation	InfiniBand NDR	InfiniBand HDR100	InfiniBand NDR
Operating system	RHEL 9.2	RHEL 8.8	Rocky Linux 9.7

Figure 4.3 highlights the performance gap between CPU-only and GPU-enabled executions of the coupled RegCM5–CLM4.5 configuration. On the Leonardo DCGP partition, increasing the CPU-only allocation from 2 to 8 nodes progressively lowers the wall-clock time, with the 8-node case serving as the reference baseline. In contrast, the GPU-enabled runs on the MareNostrum 5 ACC partition achieve superior performance once multiple accelerated nodes are employed. While the 1-node ACC configuration

does not yet surpass the CPU baseline, the 2-node and 4-node ACC cases do so clearly, with the latter delivering the best overall result. This indicates that, for this problem size, GPU acceleration becomes particularly effective at sufficiently large accelerated-node counts, where the coupled model can better exploit the available device-level parallelism.

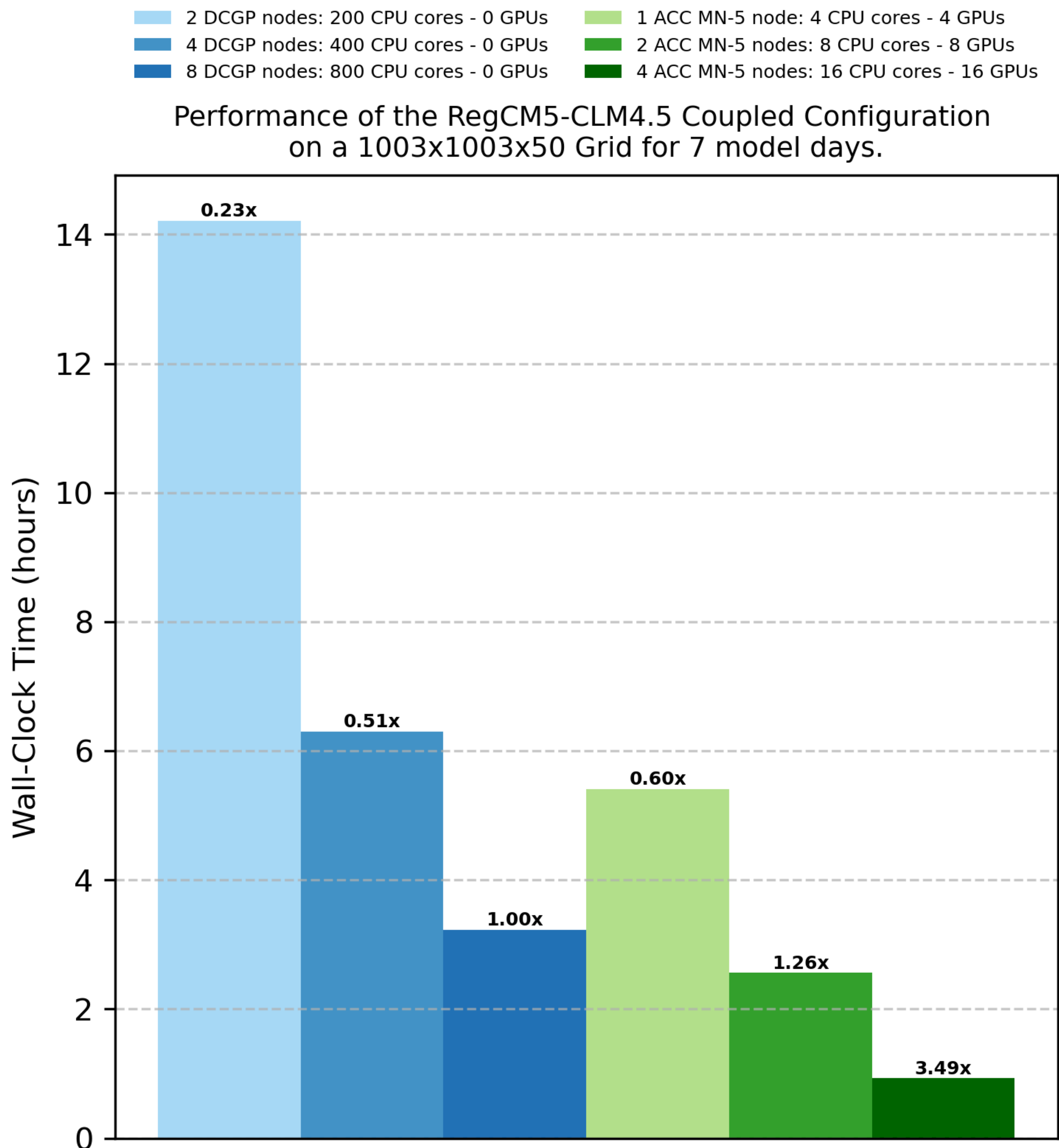


Fig. 4.3: Wall-clock time for the RegCM5–CLM4.5 coupled configuration on a $1003 \times 1003 \times 50$ grid for 7 model days, comparing CPU-only runs on the Leonardo DCGP partition with GPU-enabled runs on the MareNostrum 5 ACC partition. The CPU-only configuration uses 2, 4, and 8 DCGP nodes (200, 400, and 800 CPU cores), while the GPU-enabled configuration uses 1, 2, and 4 ACC nodes (4, 8, and 16 CPU cores with 4, 8, and 16 GPUs, respectively). The labels above the bars indicate speedup relative to the 8-node DCGP CPU-only baseline. The results show that the GPU-enabled runs outperform the CPU-only runs, with the 4-node ACC configuration achieving the shortest wall-clock time

Chapter 5

Verification,Validation and Reproducibility

One challenging aspect of heterogeneous parallel computing is reproducibility. It is defined as getting identical results from multiple runs of the same program, perhaps with different hardware resources or other changes that should not affect the answer [35]. In scientific computing experiments, reproducibility at the bit level is crucial for producing important events in simulations which needs to be replicated and reexamined more carefully. It is also required for testing and debugging programs on different processors where a quantity of interest is assumed to be identical across tests and branches.

In HPC however, with the constant pursuit for optimization and acceleration of time to solution, bitwise reproducibility can be challenging to implement in practice. Out-of-order execution in parallel computation involving non-associative floating-point operations; various compiler optimizations like (Fused Multiply Add), AVX instructions and dynamic memory management collectively introduce nondeterministic behaviors that harm reproducibility and present a unique and formidable challenge whose impact are magnified in the era of exascale computing [36].

In addition, it is well-known that climate simulations exhibit sensitive dependence on various perturbations to initial conditions and lateral boundary conditions [37]. Small perturbations in these values are sufficient to induce unintended effects that manifest as a coherent response pattern comparable in magnitude to actual modifications of model parameters. RCMs in particular are documented to exhibit a certain amount of internal variability due to nonlinearities in model physics and dynamics. [12]. As a result, two simulations that begin from identical initial states but differ slightly in some floating-point operations will inevitably diverge from each other over time [38, 39].

Presently, it is not part of the current work to ensure that the results obtained after porting the computations to the GPU successfully reproduces, at the bit level, the ones obtained before porting. Various techniques and best practices do exist to analyze and mitigate the effects of non-reproducibility in the various domains of HPC and deep learning [39, 40, 41]. There are also efforts, workshops, and investigations [36, 42] raising awareness on this important issue of GPU acceleration of climate models as well as in the broader context of HPC.

5.1 Assessing Numerical Divergence

In this section, we validate the results obtained in the GPU-enabled configurations of RegCM5 by measuring a statistical level of reproducibility of the results. To assess the extent of numerical divergence, introduced in the accelerated code, we made use of the following metrics: for a model variable $x(i,t)$, computed in a GPU-enabled run, for a specified a grid point i at any given model time t, the mean absolute difference, $\mathrm{MAD}(t)$, is defined as

$$\mathrm{MAD}(t) = \frac{1}{N}\sum_{i=1}^{N} \left| x_{\mathrm{GPU}}(i,t) - x_{\mathrm{ref}}(i,t) \right|, \tag{5.1}$$

with respect to the same model variable computed on a reference run $x_{\mathrm{ref}}(i,t)$. Similarly, the root mean square error (RMSE) is defined as

$$\mathrm{RMSE}(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_{\mathrm{GPU}}(i,t) - x_{\mathrm{ref}}(i,t)\right)^2}. \tag{5.2}$$

We also normalized the metrics to account for the different natural scales of climate variables and come up with a measure of numerical variability in relative terms. To do this, we divide the metrics above by a representative magnitude for the reference field to set the reference scale. For this reference magnitude, we computed the domain mean absolute value (MAV), $S_{\mathrm{MAV}}(t)$,

$$S_{\mathrm{MAV}}(t) = \frac{1}{N}\sum_{i=1}^{N}\left|x_{\mathrm{ref}}(i,t)\right|, \tag{5.3}$$

and the domain root-mean-square(RMS), $S_{\mathrm{RMS}}(t)$, of the reference field $x_{\mathrm{ref}}(i,t)$,

$$S_{\mathrm{RMS}}(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}x_{\mathrm{ref}}(i,t)^2}. \tag{5.4}$$

We thus define the corresponding relative metrics rMAD(t) and rRMSE(t) as

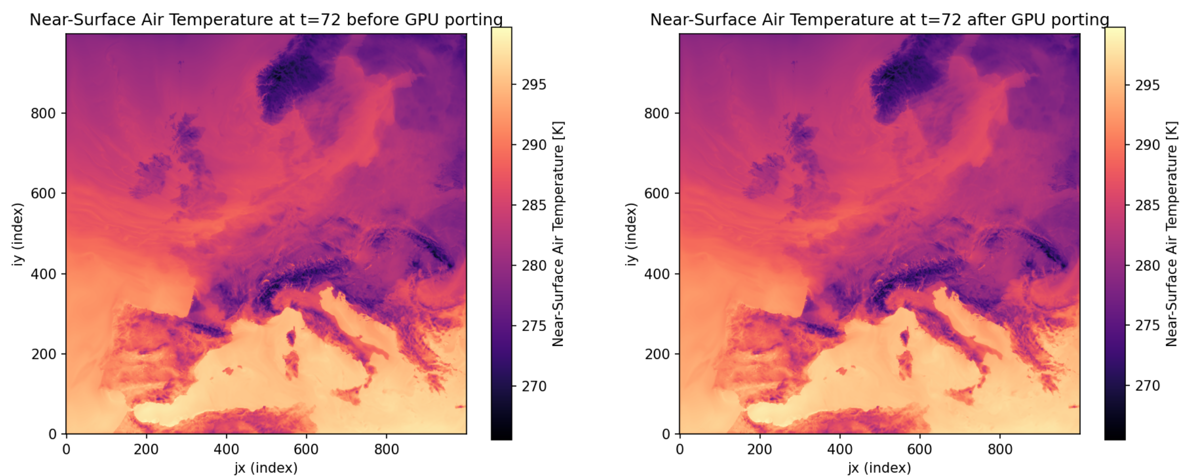
$$\mathrm{rMAD}(t) = \frac{\mathrm{MAD}(t)}{S_{\mathrm{MAV}}(t)}, \qquad \mathrm{rRMSE}(t) = \frac{\mathrm{RMSE}(t)}{S_{\mathrm{RMS}}(t)}. \tag{5.5}$$

With these formulations, these relative metrics are well-behaved and serve to quantify the divergence between one run to the next in relative terms. We then report the hourly and daily values of MAD(t), RMSE(t), rMAD(t), rRMSE(t) on three different 2-D fields: near-surface air temperature (`tas`), near-surface specific humiduty (`huss`), and surface pressure (`ps`). In addition, we also generated spatial maps of the absolute and relative difference of these fields at specified hours into the simulation.

5.2 Numerical divergence in `tas`

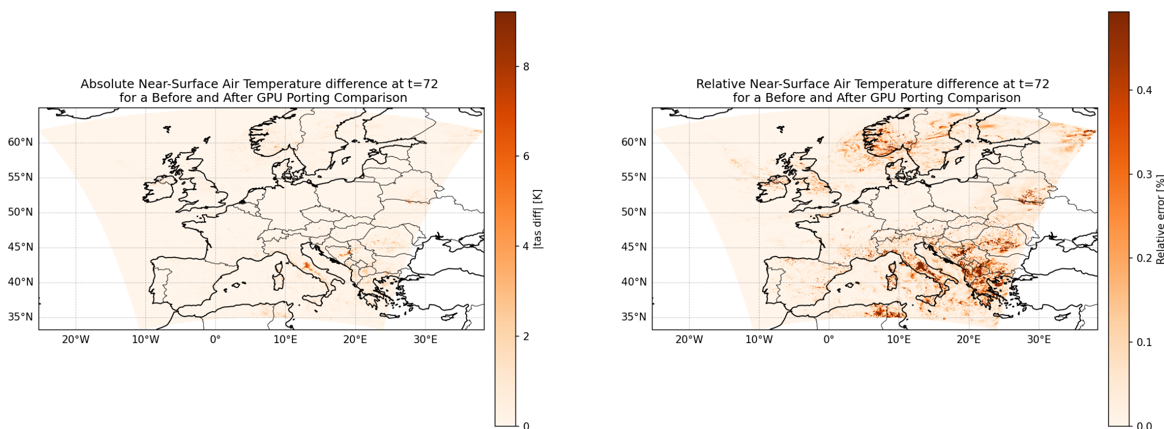
Figure 5.1 presents a spatial comparison of the near-surface air temperature (tas) field between the CPU reference simulation and the GPU-offloaded implementation at hour 72 of the experiment. The top panels show the full fields produced by the two configurations, while the bottom panels display the absolute and relative differences. The two temperature fields are visually indistinguishable over the entire domain, indicating that the GPU port preserves the large-scale thermodynamic structure of the simulation. The absolute differences remain below approximately 9 K, with the majority of the domain exhibiting errors well below 1 K. Relative differences are generally below 0.5%, confirming that the deviations between the CPU and GPU solutions are extremely small compared to the typical temperature magnitude (270–300 K).

Figure 5.2 summarizes the temporal evolution of the differences between the CPU and GPU simulations. The hourly statistics (top row) show that the mean absolute difference remains below 0.2 K, while the RMSE remains below 0.5 K throughout the entire seven-day simulation. When normalized by the magnitude of the reference field, the relative errors remain below 0.06% for rMAD and 0.17% for rRMSE. The error curves exhibit a gradual increase during the first few simulation days followed by a stabilization phase. Importantly, the magnitude of the divergence remains extremely small relative to the physical variability of the temperature field, confirming that the GPU implementation faithfully reproduces the CPU reference solution.



(a) Near-surface air temperature before GPU porting (reference).

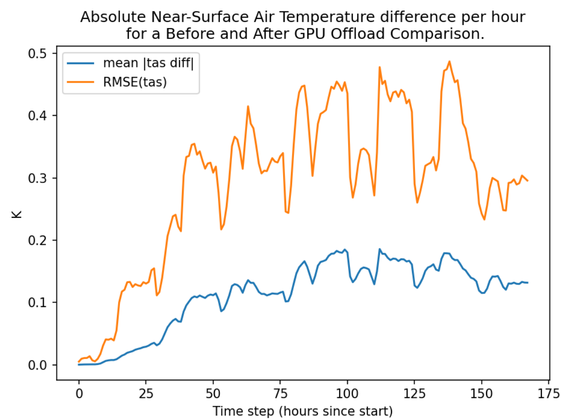
(b) Near-surface air temperature computed after GPU porting.



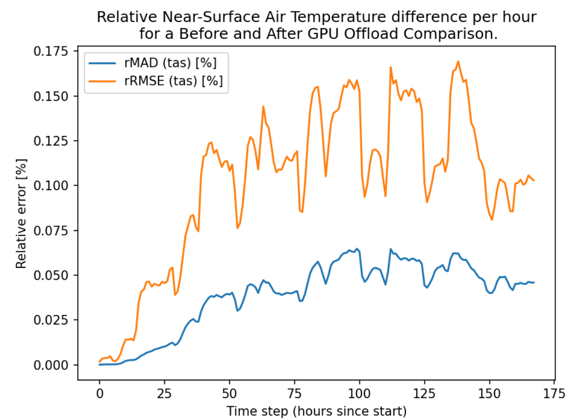
(c) Spatial map of absolute difference $|T_{\mathrm{GPU}} - T_{\mathrm{Ref}}|(i, t = 72)$.

(d) Relative difference with respect to the domain average at $t = 72$.

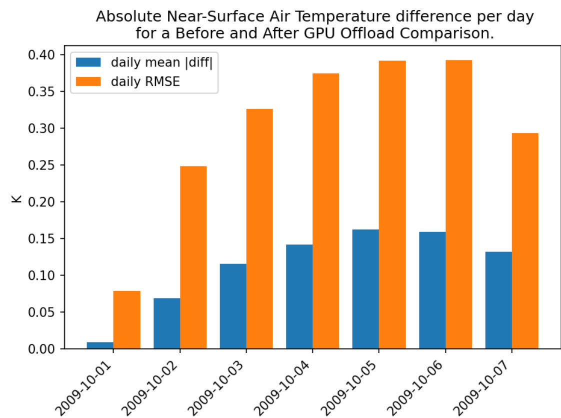
Fig. 5.1: Spatial comparison of near-surface air temperature (tas) between the CPU reference and GPU-accelerated simulations at hour 72 of the 7-day experiment. The top row shows the fields values, while the bottom row shows the corresponding absolute and relative differences. Differences remain below 10 K and 0.5% over the entire domain.



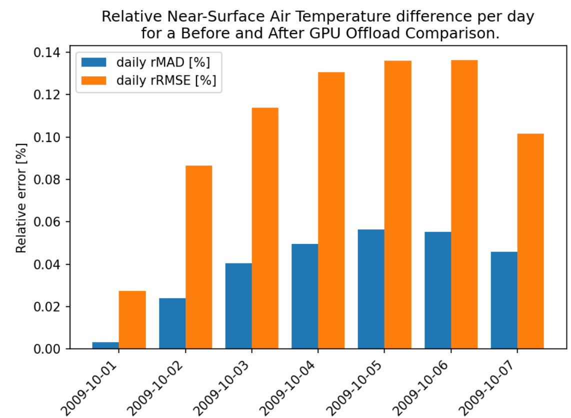
(a) Hourly absolute mean and RMSE of near-surface air temperature differences.



(b) Hourly relative rMAD and rRMSE of near-surface air temperature.



(c) Daily mean absolute difference and RMSE of near-surface air temperature.

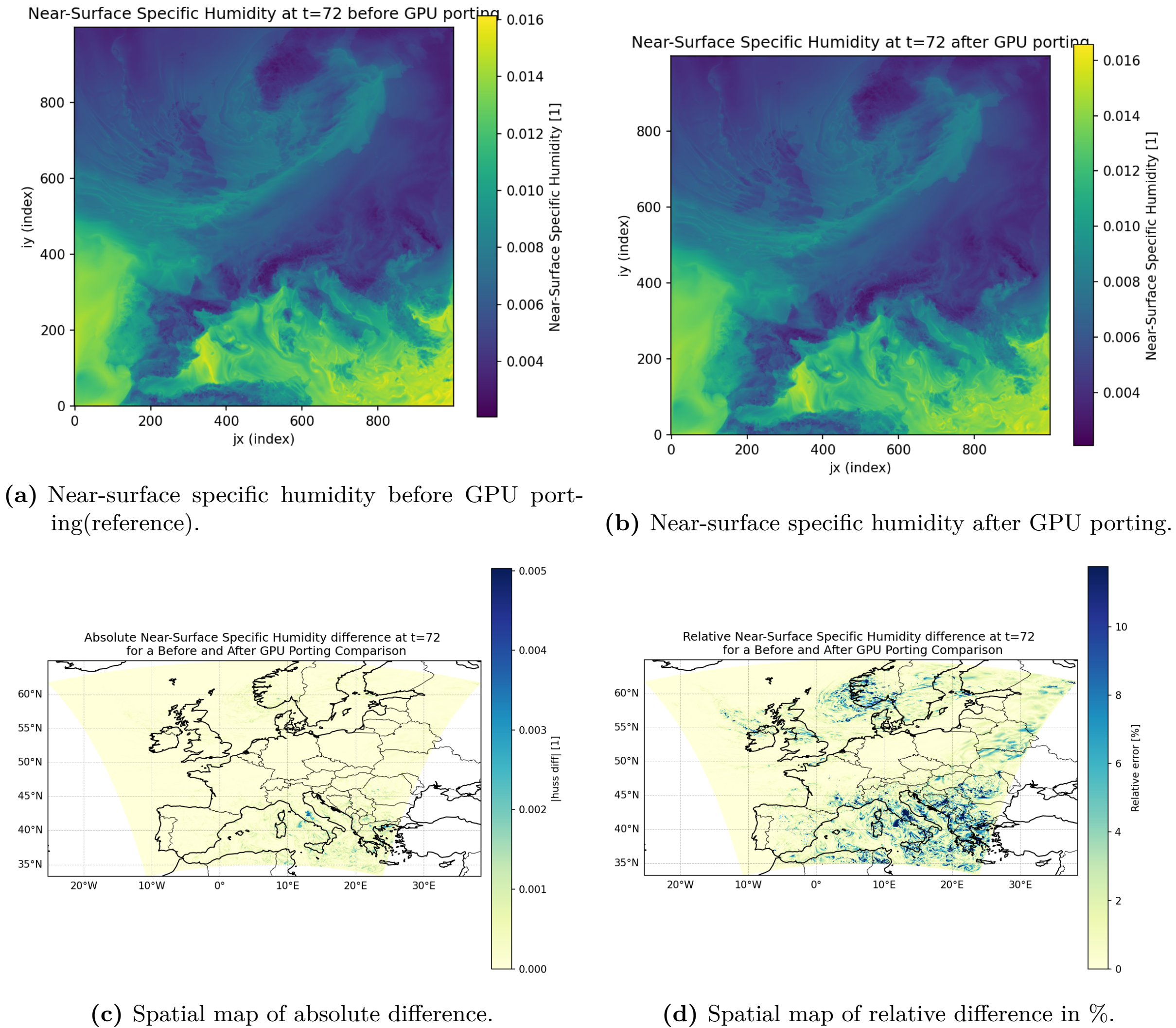


(d) Daily rMAD and rRMSE of near-surface air temperature.

Fig. 5.2: Temporal statistics of near-surface air temperature (tas) differences generated between the CPU-only and GPU-offloaded simulations over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics.

5.3 Numerical divergence in `huss`

The corresponding spatial comparison plots for the near-surface humidity (huss) are shown in figure 5.3. The top-row maps show that the near-surface specific humidity field before and after GPU porting is visually indistinguishable at hour 72. The bottom-row maps show that the differences are spatially localized rather than domain-wide. The largest absolute and relative discrepancies occur in regions around the Balkans, the eastern Mediterranean, and parts of southeastern Europe. Absolute difference remain below approximately 5×10^{-3}, while relative differences remain below about 10%, with most of the domain exhibiting substantially smaller errors.



(a) Near-surface specific humidity before GPU porting(reference).

(b) Near-surface specific humidity after GPU porting.

(c) Spatial map of absolute difference.

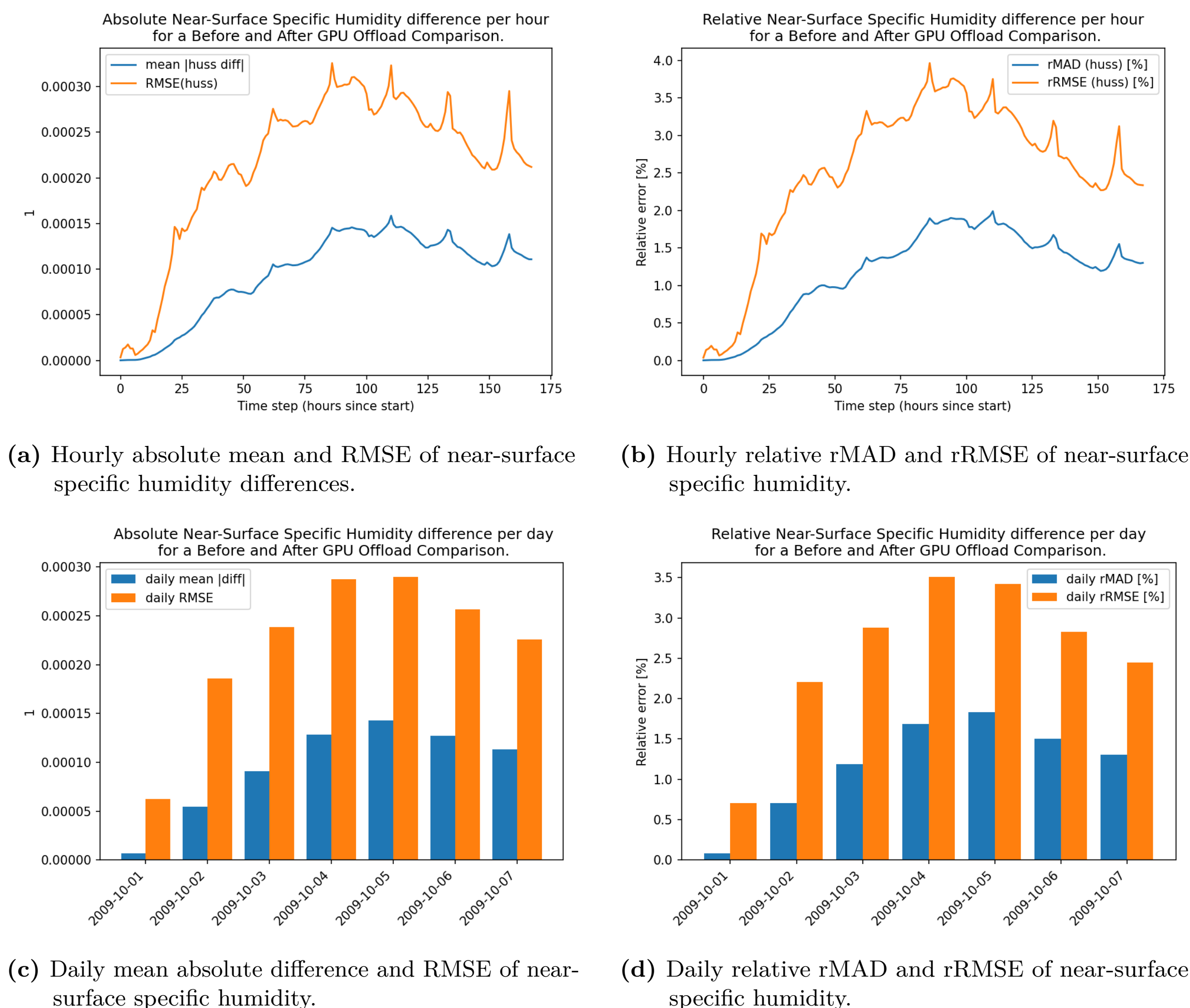
(d) Spatial map of relative difference in %.

Fig. 5.3: Spatial comparison of near-surface specific humidity (`huss`) from runs before and after GPU porting at hour 72 of the 7-day experiment. The top row shows the spatial map of the fields, while the bottom row shows the map of absolute and and relative differences. Across the domain, absolute differences remain below $5{\times}10^{-3}$ kg kg^{-1} and relative differences below 10%.

Figure 5.4 summarizes the temporal evolution of `huss` differences over the seven-day simulation. The hourly statistics show that the mean absolute difference remains below approximately 1.5×10^{-4}kgkg^{-1}, wile the RMSE remains below about 3.3×10^{-4}kgkg^{-1}. The relative metrics remain below about 2% for rMAD and 4% for rRMSE. The daily aggregated statistics exhibit the same overall behavior, with the largest differences occurring around days 4–5 and remaining bounded thereafter.

Compared with tas, huss exhibits somewhat larger relative errors, which may be attributed to the greater sensitivity of moisture fields to small perturbations in transport, boundary-layer exchange, and moist thermodynamics. Nevertheless, the errors remain small in both absolute and relative terms, and the temporal evolution remains smooth and bounded, confirming that the GPU implementation preserves

the physical integrity of the near-surface humidity field.



(a) Hourly absolute mean and RMSE of near-surface specific humidity differences.

(b) Hourly relative rMAD and rRMSE of near-surface specific humidity.

(c) Daily mean absolute difference and RMSE of near-surface specific humidity.

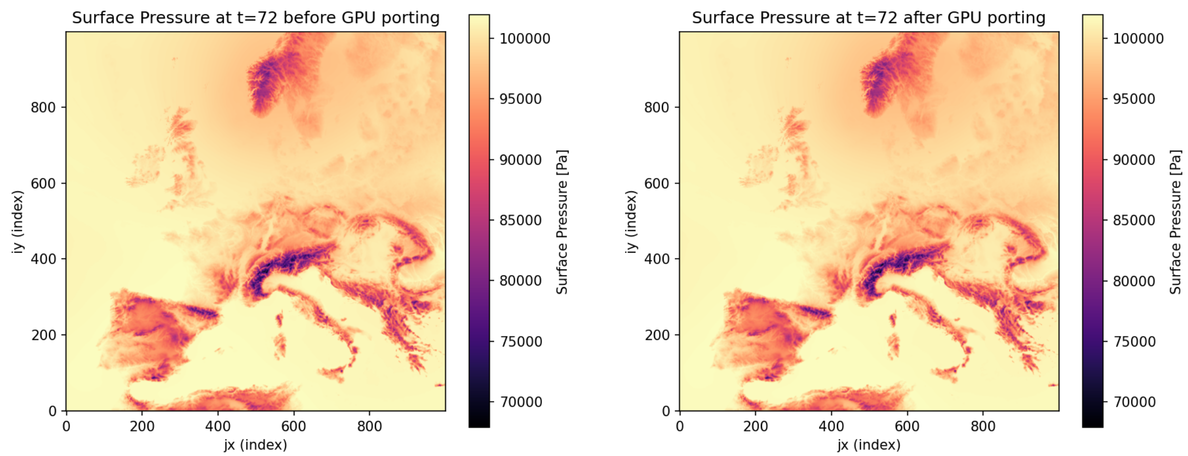
(d) Daily relative rMAD and rRMSE of near-surface specific humidity.

Fig. 5.4: Temporal statistics of near-surface specific humidity (`huss`) differences between the baseline and GPU-offloaded simulations over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics.

5.4 Numerical divergence in ps

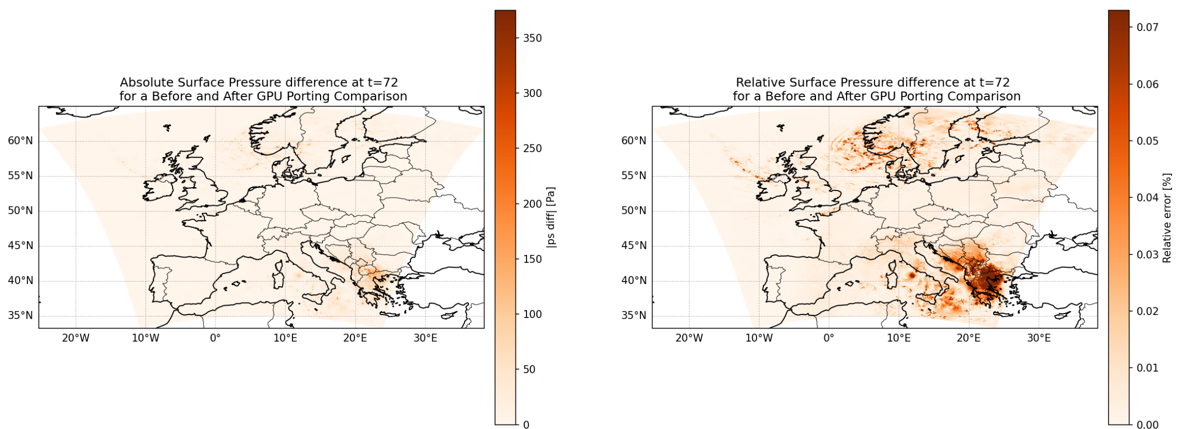
As in the previous fields, the surface pressure fields before and after GPU porting are visually indistinguishable as shown in figure 5.5. The absolute-difference map indicates that discrepancies are small over most of the domain and remain spatially localized. The largest differences appear mainly in limited areas over southeastern Europe and, more weakly, in northern portions of the domain, but even there they remain below about 350 Pa. Given that surface pressure itself is on the order of 10^5Pa, these differences are very small in relative terms. This is confirmed by the relative-difference map, where the domain-wide relative error stays below about 0.07%.

The temporal statistics support the same interpretation. During the first part of the simulation, both the hourly mean absolute difference and RMSE are near zero, after which they increase and then settle on bounded regime. Over most of the 7-day experiment, the hourly mean absolute difference remains around 6–7 Pa, while the hourly RMSE fluctuates around 15–19 Pa. The relative metrics are even more reassuring: hourly rMAD stays around 0.006–0.008%, while hourly rRMSE remains around 0.015–0.019%. These are extremely small compared with the magnitude of the pressure field itself. The same behavior is seen in the daily aggregates, which remain of similar magnitude throughout the experiment and do not exhibit any runaway growth.



(a) Surface pressure before GPU porting (reference).

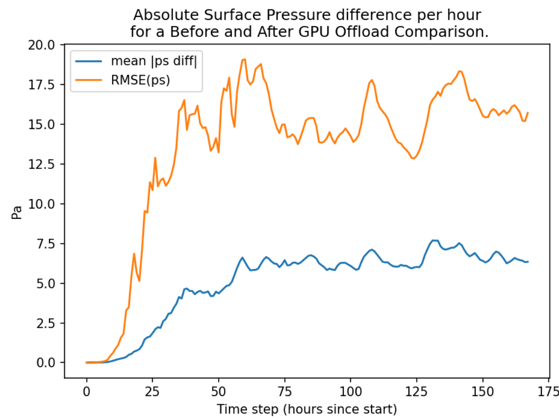
(b) Surface Pressure after GPU porting.



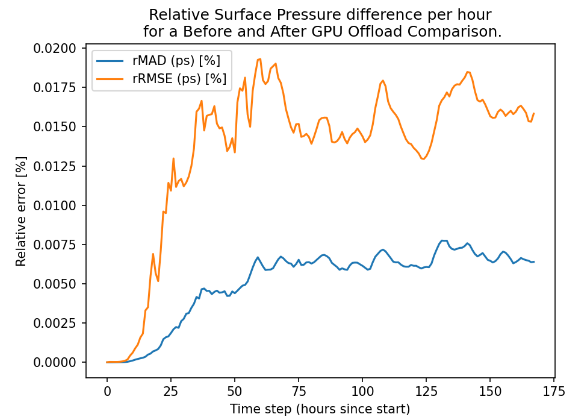
(c) Spatial map of absolute difference $|P_{\text{GPU}} - P_{\text{Ref}}|(i, t = 72)$.

(d) Relative difference with respect to the domain average at $t = 72$.

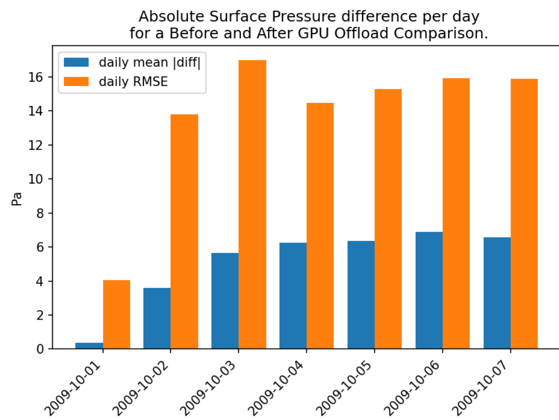
Fig. 5.5: Spatial comparison of surface pressure `ps` computed before and after GPU-porting at hour 72 of the 7-day experiment. The top row shows the absolute fields, while the bottom row shows the corresponding absolute and relative differences. Differences remain below 350 Pa and 0.07% over the entire domain.



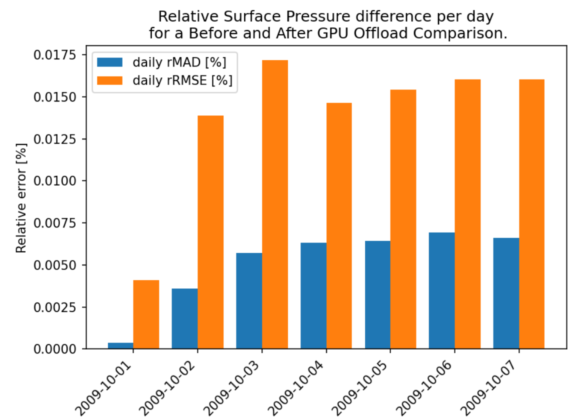
(a) Hourly absolute mean and RMSE of surface pressure differences.



(b) Hourly relative rMAD and rRMSE of surface pressure.



(c) Daily mean absolute difference and RMSE of surface pressure.



(d) Daily relative rMAD and rRMSE of surface pressure.

Fig. 5.6: Temporal statistics of surface pressure (ps) differences computed before and after GPU-offloading over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics.

Chapter 6

Conclusions, Recommendations and Future Outlook

In this work we carried out various optimizations and GPU offloading strategies on the coupled configuration of the RegCM5 regional climate model based on OpenACC and Fortran standard language parallelism through `do concurrent`. This was motivated by an observed performance degradation of the GPU-enabled runs of RegCM5 when coupled with the Community Land Model CLM4.5 as compared to running it in an idealized RCE configuration to simulate a free-running atmosphere. Profiling helped identified several code paths that are activated only in the the coupled configuration which are responsible for the observed performance degradation. The first performance hotspot accounted for 20% of the initial total wall-clock time and originated from the noncontiguous row-slice initialization of two dimensional arrays in the hydrology tracer routine inside the module `mod_clm_hydrology2`. Eliminating this performance hotspot yielded a 1.77× speedup for the GPU-enabled run.

The rest of the performance hotspots collectively account for 70% of the resulting wall-clock time and consists of several procedures whose call sites occur inside nested `do` loops and are computed entirely on the host. We refactored these subroutines to exposed fine-grained parallelism suitable for GPU execution while keeping the original physical formulation. These optimizations and porting strategies produced an overall 7.10× speedup relative to the initial code. As a result, the GPU-enabled run using 16 GPUs on Booster now runs 2.4× faster than the CPU-only configuration using 800 CPU cores on DCGP. Additional optimizations and porting efforts, not discussed in the present work further increased the total speedup to 14.17× relative to the initial version. We also studied the performance portability of the coupled configurations across two other platforms running on H100 GPUs. We observed that the coupled configuration exhibited the same superlinear scaling across three computing platforms the origin of which needs to be investigated in future work.

Ensuring that the GPU ported code reproduces the CPU-only results at the bit level is not yet part of the present work. At this stage, we observed only statistical reproducibility between the ported and CPU-only runs. After 7 model days, the numerical divergence remained below 0.1% for the near-surface air temperature and surface pressure while for the near-surface specific humidity remained within 5%. Future work should focus on taking measures to mitigate or reduce the observed numerical divergence between results obtained from the CPU-only runs and from the GPU ported codes.

All the numerical experiments reported in the present work made use of the CUDA managed memory option for the memory model. This capability is activated at compile time using the option `mem:managed` of the `nvfortran` compiler. Future investigations can also focus on the performance benefits of using the more advanced implementation of the CUDA unified virtual memory which is natively supported on the latest generation of NVIDIA devices and platforms such as the H100 and the GH200.

Finally, another important point we leave for future work is to investigate the behavior reported in figure 2.4 on the performance of running RegCM5 in the coupled configuration using various parallel output strategy. Future work on optimization can aim to identify and understand the origin of the consistent performance degradation when enabling parallel write regardless of which backend I/O library is used. Shedding light into this behavior could be a potential source for more substantial performance gain.

Appendix A

Profiling Tools

A.1 `Perf`

For Linux 2.6+ based systems, `Perf_events` or `perf` for short is a profiler tool that abstracts away CPU hardware differences in Linux performance measurements [43]. It includes a set of commands to collect and analyze performance and trace data. `Perf_events` is based on event-based sampling. This means that the `perf` tool and the underlying kernel interface uses events to trigger a sampling whereby the kernel interrupts the running program and record a snapshot of the program's execution state.

Events in `perf` are classified according its source. For instance, events like context-switch and minor faults are pure kernel counters and are classified as software events. Other events like number of cycles, instructions retired, and L1 cache misses can come from the processor itself and its Performance Monitoring Unit (PMU). These events are referred to as hardware events. The `perf_events` interface also provide a list of small set of common hardware events that gets mapped to an actual event provided by the CPU if they exist. Lastly, there are also trace events which are implemented by the kernel `ftrace` infrastructure. To obtain a complete list of supported events run the command `perf list`.

During sampling, a sampling period can be expressed as the number of occurrences of an event. A sampling period is specified using the `-c <period>` option. In this mode, a sample is recorded every `period` occurrences of an event. The kernel then records information about the state of execution of the program. This information may include the current instruction pointer, stack trace(when `-g` is enabled) or CPU/register state. What gets recorded depends on the type of measurement specified by the user and the tool. The key information that is common in all samples is the instruction pointer, i.e. where was the program when it was interrupted. Alternatively, the sampling period can also be expressed in frequency, that is the average rate of samples/sec. It is set to 1000Hz or 1000 samples/sec by default and can be specified with the `-F` option. In this case the kernel adjusts the period dynamically in order achieve the target average sampling frequency.

Here, we use the `perf` tools to count events on a per-process level. In this mode, all threads of a process is monitored and counts and samples are aggregated at the process level. The general format for command line usage is the generic tool, `perf`, followed by a command like `record`, `report`, and `stat` for collecting and analyzing data. A list of options to modify the behavior of each command can be queried by typing the command followed by `-h`.

Listing A.1, encapsulates the setup for using `perf` to profile a typical run for RegCM5. The command used to collect samples is `perf record`. By default, `perf record` uses the `cycles` event as the sampling event. This is a generic hardware event that is mapped to a hardware-specific PMU event by the kernel. We also measured the `instruction` event and added the modifier `:u` to measure both events at the user level. We also set the sampling frequency to 200Hz.

A.2 Flame graphs

Flame graphs are a convenient tool for visualizing a population of stack traces of a profiled software so that the most frequent code paths can be identified quickly and accurately. There are many types of flame graphs and its implementations [44]. Here, we used CPU flame graphs to visualize the stack traces collected by `perf record` to quickly spot which code paths are hot, that is busy on the CPU. This will help us identify which regions or procedures in the code takes up most of the on-CPU time.

To generate deep, unbroken call stacks and visualize them as informative flame graphs, it is important that RegCM5 along with the libraries linked are compiled with the debug option `-g` enabled. The option tells the compiler to emit debug information into the binary which enables `perf` to unwind the stack from the current instruction pointer backward during sampling to attribute the sample to a full stack. This is what `perf` does when we pass the `-call-graph` option to `perf record` and specify `DWARF` as the stack-unwinding mechanism to use. `DWARF` is a standardized format for storing debug information in executable and linkable files (ELF). `DWARF`-based mechanism for unwinding stack is slower but more robust even if the code is heavily optimized and inlines many routines.

The instructions encapsulated in listing A.1 generates a `perf` file for each process in the directory `/$SCRATCH/perf/with_IO/sequential_ write/`. The perf file corresponding to process 0 is thus `perf.data.rank0`. This is helpful when we wish to perform comparative analysis of the stack traces recorded in each individual process for understanding the load imbalance among the processes.

Listing A.2 is a slurm script that encpasulates information for post-processing the files generated by `perf report` to create a flame graph. We post processed the files in two ways. One way is to obtain a separate flame graphs from each individual perf file for each process. The second one is to visualize the aggregated stack traces from all processes to obtain an overall picture. In both cases, the command `perf script` expands binary samples into human-readable stack traces. The command `stackcollapse-perf.pl` converts multi-line stacks into folded lines with counts. Finally, the command `flamgraph.pl` aggregates identical stacks

The width of the boxes correspond to more samples and hence more time spent. The height of the stack corresponds to the call depth. The call stack begins at the most bottom box where the first module containing the procedure that initiated the call stack.

To generate flame graphs, we used the implementation specific for `perf` written by Brendan Greggs. The repository also contains binaries for various flame graph implementation for other profilers. We wrote a slurm job script that encapsulates the steps to generate a cpu flame graph. It is a visualization tool to quickly identify which code paths are hot.

```
1 $ cd # cd to $HOME
2 $ git clone https://github.com/brendangregg/FlameGraph.git
3 $ export PATH=$/FlameGraph:$PATH
```

Then

Listing A.1: Wrapper script for running RegCM5 with `perf` profiler.

```
1  #!/bin/bash
2  # launch_per_rank.sh
3
4  # Get the rank of the current MPI process
5  RANK=${SLURM_LOCALID}
6
7  # Optional safety check
8  if [ -z "$RANK" ]; then
9    echo "SLURM_LOCALID not set; this script should be run under srun."
10   exit 1
11 fi
12
13 # Assign each local rank to a unique GPU on this node
14 export CUDA_VISIBLE_DEVICES=$RANK
15 export ACC_DEVICE_TYPE=nvidia
16 export ACC_DEVICE_NUM=0
17 export BINDIR=/leonardo/home/userexternal/ctica000/MS_thesis/RegCM/bin
18
19 # For Perf
20 FREQ="${PERF_FREQ:-200}"  # samples/seconds
21
22 # output directory per job
23 OUTDIR="${PERF_OUTDIR:-$SCRATCH/perf/with_IO/sequential_write/perf_${
     SLURM_JOB_ID}}"
24 mkdir -p "$OUTDIR"
```

```
umask 007   # files created with 600 permissions

# Unique filename per rank & node
OUTFILE="${OUTDIR}/perf.data.rank${SLURM_PROCID}"

 # common perf args
common_args=(
        -F "$FREQ"
        --output="$OUTFILE"
        --call-graph dwarf
        -e cycles:u,instructions:u
)

echo "[Wrapper][Rank $SLURM_PROCID][Local $SLURM_LOCALID] on $(hostname
    ): CUDA_VISIBLE_DEVICES=$CUDA_VISIBLE_DEVICES"

# Launch the binary
exec perf record "${common_args[@]}" -- $BINDIR/
    regcmMPICLM45_OPENACC_GPU_STDPAR EURR-3_namelist.in
```

Listing A.2: Job script for generating the flame graph.

```
#!/bin/bash
#SBATCH --account=ICT25_MHPC
#SBATCH --out=./profile/LOG_%x_%j.out
#SBATCH --err=./profile/LOG_%x_%j.err
#SBATCH --gres=tmpfs:200g
#SBATCH --job-name create_flame_graph
#SBATCH --ntasks-per-node=16
#SBATCH --partition=dcgp_usr_prod
#SBATCH --nodes=1
#SBATCH --mem=0G
#SBATCH --time=04:00:00

export PATH=$PATH:/leonardo/home/userexternal/ctica000/MS_thesis/
    FlameGraph

echo " Generating the Flame Graph for Rank 0 "

# Generate flame graph for rank 0
perf script -i perf.data.rank0 > rank0.perf
stackcollapse-perf.pl rank0.perf > rank0.folded
flamegraph.pl rank0.folded > flame_rank0.svg

echo " Generating the combined Flame Graph for Ranks 0 to 15 "

# Combine perf data from rank 0 to rank 15
ls perf.data.rank{0..15} | parallel -j16 'perf script -i {} --max-stack
     128 --no-demangle' > subset_0_15.perf

# Generate flame graph for ranks 0 to 15
stackcollapse-perf.pl subset_0_15.perf > subset0_15.folded
flamegraph.pl subset0_15.folded > flame_rank0_15.svg
```

Listing A.3: Workflow for generating the flame graph.

```
# Step 1. Allocate resources interactively.
$srun -A ICT25_MHPC -p dcgp_usr_prod -N1 -n1 --cpus-per-task=50 --mem
    =300G --gres=tmpfs:100g --time=03:00:00 --pty bash
```

```
3
4  srun: job 24683785 queued and waiting for resources
5  srun: job 24683785 has been allocated resources
6
7  # Step 2. Collect the stacks for a particular process, say process 0.
       Run this in the backgorund and detach.
8
9  $ nohup perf script -i perf.data.rank0 > rank15.perf 2>
       perf_script_rank0.log &
10
11 # Step 3. Collapse stack into folded form.
12 $ nohup stackcollapse-perf.pl rank0.perf > rank0.folded 2>
       stack_collapse_rank0.log &
13
14 # Step 4. Generate the flame graph.
15 nohup flamegraph rank0.folded > flame_rank0.svg 2> flame_graph.log &
16
17 # Step 5. On local host, visualize the flame graph.
18 $ rsync -PravzHS ctica000@data.leonardo.cineca.it:/path/to/perf/
       flame_rank0.svg .
19
20 $ google-chrome flame_rank0.svg
```

A.3 Nvidia Nsight systems

To profile the code using Nvidia Nsight systems, we made use of the following script. For multiple GPU runs with equal number of CPU processes, this wrapper script will collect performance data for each participating process.

Listing A.4: Wrapper script for runnig the RegCM 5.0 with Nsight profiler.

```
1  #!/bin/bash
2
3  # Get local rank
4  RANK=${SLURM_LOCALID}
5
6  if [ -z "$RANK" ]; then
7    echo "SLURM_LOCALID not set; this script should be run under srun."
8    exit 1
9  fi
10
11 # GPU binding
12 export CUDA_VISIBLE_DEVICES=$RANK
13 export ACC_DEVICE_TYPE=nvidia
14 export ACC_DEVICE_NUM=0
15
16 # ----------------------------
17 # Nsight Systems (newer CLI)
18 # ----------------------------
19 NSYS_BIN="/leonardo/prod/opt/compilers/nvhpc/25.3/binary/Linux_x86_64
       /25.3/compilers/bin/nsys"
20
21 # Set binary and input paths
22 export BINDIR=/leonardo/home/userexternal/ctica000/MS_thesis/RegCM/bin/
       profile_binary
23 INPUT_FILE=EURR-3_namelist.in
24
25 # Set a unique temp directory using TMPDIR only
```

```
26 TMPDIR_BASE=$SCRATCH/tmp_nsys
27 export TMPDIR=$TMPDIR_BASE/rank_${SLURM_PROCID}
28 mkdir -p "$TMPDIR"
29
30 # Set output path for Nsight profile
31 NSYS_OUT="$SCRATCH/profile_new_nsys_16gpu/nsys_rank${SLURM_PROCID}"
32
33 # Launch with Nsight profiling
34 $NSYS_BIN profile \
35   --force-overwrite true \
36   --stats=true \
37   --sample=process-tree \
38   --backtrace=dwarf \
39   --output "$NSYS_OUT" \
40   --sampling-trigger=perf \
41   --sampling-period=1000000 \
42   --trace=nvtx,mpi,cuda,osrt \
43   --mpi-impl=openmpi \
44   --cuda-um-cpu-page-faults=true \
45   --cuda-um-gpu-page-faults=true \
46   --cuda-memory-usage=true \
47   "$BINDIR/regcmMPICLM45_OPENACC_GPU_STDPAR" "$INPUT_FILE"
```

Appendix B

Reproducibility Diagnostic Workflow

This appendix details the workflow used to quantify numerical differences between CPU and GPU (or pre– and post –offload) runs of RegCM5 on the SRF NetCDF output files. The workflow consists of three Python utilities:

1. `concat_diff_var.py`: concatenate daily NetCDF files and compute GPU–CPU differences for a single variable.
2. `multi_day_stats_generic.py`: compute multi-day temporal statistics (MAD, RMSE, rMAD, rRMSE) from the concatenated files.
3. `spatial_maps_generic.py`: generate spatial error maps on a geographical grid (and in index space) for a selected time step.

The scripts are meant to be re-usable for different variables such as `tas`, `ts`, `ps`, `huss`, and `cape`. All the examples we considered were run inside the `nco-tools` Conda environment, where the NCO utilities `ncrcat`, `ncdiff` and `ncap2` are available.

B.0.1 File naming and directory conventions

For each experiment (e.g. 7-day SRF runs) I assume:

- A directory with the CPU (or baseline) NetCDF files:
  `CPU_DIR=/path/to/cpu/output`.
- A directory with the GPU (or modified) NetCDF files:
  `GPU_DIR=/path/to/gpu/output`.
- An output directory for concatenated and diagnostic files:
  `OUT_DIR=/path/to/concatenated_general_<var>`.
- Daily SRF files following a common pattern such as
  `EURR-3_SRF.20*.nc` in both `CPU_DIR` and `GPU_DIR`.

The scripts adopt a consistent naming scheme for concatenated files:

- `<var>_CPU_all_days.nc`: concatenated reference field.
- `<var>_GPU_all_days.nc`: concatenated new field.
- `<var>_DIFF_all_days.nc`: concatenated difference field.

The difference is always defined as

$$\mathrm{diff} = \mathrm{new} - \mathrm{ref}, \tag{B.1}$$

so that the new field is recovered as

$$\mathrm{new} = \mathrm{ref} + \mathrm{diff}. \tag{B.2}$$

B.0.2 Step 1: Concatenation and GPU–CPU difference

The script `concat_diff_var.py` consolidates daily files and computes the GPU–CPU difference for a chosen variable using NCO.

Usage

```
python concat_diff_var.py \
  CPU_DIR \
  GPU_DIR \
  OUT_DIR \
  VAR_NAME \
  [--pattern "EURR-3_SRF.20*.nc"]
```

Arguments:

`CPU_DIR` directory containing the CPU (reference) SRF files.

`GPU_DIR` directory containing the GPU (or modified) SRF files.

`OUT_DIR` directory where concatenated files are written.

`VAR_NAME` NetCDF variable name to process (e.g. `tas`, `huss`).

`-pattern` optional shell glob to select SRF files (default: `"EURR-3_SRF.20*.nc"`).

The script performs:

1. Build sorted file lists in `CPU_DIR` and `GPU_DIR` matching the given pattern.
2. Check that the two lists have the same length and that corresponding filenames (basenames) are identical.
3. Call

   ```
   ncrcat -O -v VAR_NAME CPU_FILES...  <var>_CPU_all_days.nc
   ncrcat -O -v VAR_NAME GPU_FILES...  <var>_GPU_all_days.nc
   ```

   to concatenate the time dimension over all days.
4. Call

   ```
   ncdiff -O -v VAR_NAME \
     <var>_GPU_all_days.nc \
     <var>_CPU_all_days.nc \
     <var>_DIFF_all_days.nc
   ```

   to compute the GPU–CPU difference.
5. Add two diagnostic variables to the diff file using `ncap2`:

$$\texttt{<var>_diff_abs} = |\texttt{<var>}|,$$
$$\texttt{<var>_diff_sq} = (\texttt{<var>})^2.$$

Example: concatenating huss

```
$ python concat_diff_var.py \
  /.../before_offload/16gpu_16proc/output \
  /.../after_offload/16gpu_16proc/output  \
  /.../concatenated_general_huss          \
  huss \
  --pattern "EURR-3_SRF.20*.nc"
```

This produces `huss_CPU_all_days.nc`, `huss_GPU_all_days.nc`, and `huss_DIFF_all_days.nc` in `OUT_DIR`.

B.0.3 Step 2: Multi-day temporal statistics

The script `multi_day_stats_generic.py` computes temporal statistics of the difference field for a given variable. It assumes that `<var>_CPU_all_days.nc` and `<var>_DIFF_all_days.nc` exist in `BASE_DIR`.

Usage

```
python multi_day_stats_generic.py \
  --base-dir BASE_DIR \
  --var VAR_NAME \
  --units UNITS
```

Arguments:

-base-dir directory containing the concatenated files.

-var variable name (e.g. `tas`, `huss`).

-units units string for plotting (e.g. `K`, `"1"`, `Pa`).

The script:

1. Reads `<var>_CPU_all_days.nc` and `<var>_DIFF_all_days.nc` into memory. Variables with an extra singleton vertical level (e.g. `(time, m2, iy, jx)`) are automatically squeezed to `(time, iy, jx)`.
2. Computes per-time statistics over all horizontal grid points:

$$\begin{aligned} \mathrm{MAD}(t) &= \langle |d(t)| \rangle, \\ \mathrm{RMSE}(t) &= \sqrt{\langle d(t)^2 \rangle}, \\ \max |d(t)| &= \max_{i,j} |d(t,i,j)|, \end{aligned}$$

   where $d = \text{new} - \text{ref}$ and $\langle \cdot \rangle$ denotes a grid-cell mean.
3. Computes per-time reference scales:

$$\begin{aligned} S_{\mathrm{MAD}}(t) &= \langle |x_{\mathrm{ref}}(t)| \rangle, \\ S_{\mathrm{RMSE}}(t) &= \sqrt{\langle x_{\mathrm{ref}}(t)^2 \rangle}, \end{aligned}$$

   and defines relative metrics

$$\begin{aligned} \mathrm{rMAD}(t) &= \frac{\mathrm{MAD}(t)}{S_{\mathrm{MAD}}(t)}, \\ \mathrm{rRMSE}(t) &= \frac{\mathrm{RMSE}(t)}{S_{\mathrm{RMSE}}(t)}. \end{aligned}$$

4. Aggregates statistics to full days (blocks of 24 time steps) to obtain daily mean absolute error, daily RMSE, daily maximum error, and their relative counterparts.
5. Writes two CSV files:
   - `<var>_diff_stats.csv` (per-time statistics),
   - `<var>_diff_stats_per_day.csv` (per-day statistics).
6. Produces four diagnostic plots:
   - `<var>_diff_timeseries.png` (absolute MAD and RMSE per hour),
   - `<var>_diff_timeseries_relative.png` (rMAD and rRMSE per hour),
   - `<var>_diff_per_day.png` (daily absolute MAD and RMSE),
   - `<var>_diff_per_day_relative.png` (daily rMAD and rRMSE).

Example: statistics for `tas`

```
python multi_day_stats_generic.py \
  --base-dir /.../concatenated_general_tas \
  --var tas \
  --units K
```

B.0.4 Step 3: Spatial maps of absolute and relative error

The script `spatial_maps_generic.py` generates spatial maps of the reference field, new field, absolute difference, and relative difference at a chosen time index. It uses the `xlat` and `xlon` fields in the SRF files to plot data on a true geographical grid via Cartopy.

Usage

```
python spatial_maps_generic.py \
  --base-dir BASE_DIR \
  --var VAR_NAME \
  --units UNITS \
  [--ref-tag CPU] \
  [--diff-tag DIFF] \
  [--t-idx 72] \
  [--threshold THRESH] \
  [--rel-vmax VMAX]
```

Arguments:

-base-dir directory containing `<var>_CPU_all_days.nc` and `<var>_DIFF_all_days.nc`.

-var variable name (e.g. `tas`, `huss`, `cape`).

-units units string for the absolute-error color bar.

-ref-tag file tag for the reference run (default: `CPU`, expecting `<var>_CPU_all_days.nc`).

-diff-tag file tag for the difference run (default: `DIFF`, expecting `<var>_DIFF_all_days.nc`).

-t-idx zero-based time index to plot (default: 0).

-threshold optional threshold for the relative error denominator; where $|x_{\mathrm{ref}}| \leq$ `threshold`, the relative error is masked (used for diagnostic fields such as CAPE).

-rel-vmax optional upper limit for the relative error color scale in percent.

For each variable and time index, the script:

1. Reads the reference field x_{ref} and difference field $d = x_{\mathrm{new}} - x_{\mathrm{ref}}$ at time index `t-idx`, squeezes any singleton vertical dims to get a 2-D array $(\mathrm{iy}, \mathrm{jx})$, and reconstructs $x_{\mathrm{new}} = x_{\mathrm{ref}} + d$.
2. Reads `xlat` and `xlon`, checks that their shapes match the field.
3. Computes the absolute error abs_err $= |d|$, and the relative error rel_err $= |d|/|x_{\mathrm{ref}}|$, optionally masking where the reference field magnitude falls below `threshold`. The plotted relative error is expressed as a percentage.
4. Produces four geographic maps using Cartopy (Plate Carrée projection):
   - reference field,
   - new field,
   - absolute error $|d|$,

- relative error in percent.

Coastlines and national borders are drawn, and the domain is constrained to the min/max of the SRF longitude/latitude grids.

5. Produces four corresponding index-space plots using `imshow` (field as a function of `iy` and `jx`), mainly for debugging and quick inspection.

All eight plots are written to `BASE_DIR` with filenames of the form `<var>_XXX_map_geo_t<idx>.png` and `<var>_XXX_map_idx_t<idx>.png` for `XXX` in {`ref`, `new`, `abs_error`, `rel_error`}.

Example: spatial maps for tas at hour 72

```
python spatial_maps_generic.py \
  --base-dir /.../concatenated_general_tas \
  --var tas \
  --units K \
  --t-idx 72
```

B.0.5 Recipe for adding a new variable

To apply this workflow to a new SRF variable (e.g. `ps` or `pr`), the steps are:

1. Identify the variable name in the NetCDF file (e.g. `ncdump -h`).
2. Choose a new concatenation directory `OUT_DIR`, such as `.../concatenated_general_ps`.
3. Run `concat_diff_var.py` with the appropriate CPU/GPU directories and variable name:

```
python concat_diff_var.py CPU_DIR GPU_DIR OUT_DIR ps
```

4. Run `multi_day_stats_generic.py` to compute temporal statistics:

```
python multi_day_stats_generic.py \
  --base-dir OUT_DIR \
  --var ps \
  --units Pa
```

5. Optionally, generate spatial maps for one or more time indices:

```
python spatial_maps_generic.py \
  --base-dir OUT_DIR \
  --var ps \
  --units Pa \
  --t-idx 72
```

This appendix therefore provides a reproducible and extensible pattern for verifying GPU offloading changes against a CPU (or baseline GPU) reference for a wide range of SRF diagnostic and prognostic fields.

List of Figures

1.1 Build pipeline from `configure.ac` and related build-system files to the RegCM5 executables used in this thesis. The files `configure.ac`, `Makefile.am`, `acinclude.m4` and `makeinc` are processed by the autotools tools (`autoreconf`, `autoconf`, `automake`) to generate the `configure` script and the `Makefile.in` templates. 4

2.1 Runtime pipeline for the RCE configuration of RegCM5. The run is configured by `isc24_small.in` and initialized by `profile.in` (uniform surface state and small perturbations). Diagnostic NetCDF output is disabled to isolate computational performance; only logs and profiling artefacts are produced. 12

2.2 Baseline performance of RegCM5 in the Radiative–Convective Equilibrium (RCE) configuration executed on a 512×1024×60 grid on Leonardo. CPU-only runs (blue) utilize MPI across 256 and 896 cores on Booster and DCGP partitions, respectively. GPU-enabled runs (green) combine MPI with OpenACC and Fortran `do concurrent` offloading to 4, 8, and 16 NVIDIA A100 GPUs. The inset panel reports the simulation cost in core-hours, defined as (CPU cores + $8 \cdot$ GPUs) × wall-clock time, demonstrating that GPU offloading provides substantial reductions in both wall-clock time and resource cost. The speedups are computed with respect to CPU-only run on 8 nodes of the DCGP partition using 896 CPU cores. 13

2.3 Runtime pipeline for the coupled RegCM5–CLM4.5 configuration used in this work. The simulation is driven by ERA5 reanalysis (including ERA5 daily SST), launched via Slurm and a per-rank wrapper for GPU binding, and produces CORDEX-compliant atmospheric output as well as CLM history fields. 14

2.4 Performance of the RegCM5–CLM4.5 coupled configuration on 1003 × 1003 × 50 grid for 4 model days on Leonardo for different compute resource allocations and three I/O backends: HDF5 with sequential write (green), HDF5 with parallel write (blue), and PnetCDF with parallel write (orange). The reads are parallel for all cases. CPU-only baselines (0 GPUs) were executed on DCGP nodes using 400 and 800 CPU cores, while GPU-enabled runs used 8–64 A100 GPUs on Booster nodes. The coupled configuration exhibits reduced GPU performance compared with the RCE configuration, and only achieves lower wall-clock times and competitive core-hour costs at ≥ 32 GPUs. Inset shows total simulation cost in core-hours 16

2.5 Flame graph visualization of a population of call stacks for a 7-day simulation run on a 512 × 1024 × 60 grid of the RegCM5 in RCE configuration using 16 GPU - 16 CPU processes which is 4 nodes on the Booster partition on Leonardo. 17

2.6 Flame graph for the 1-day simulation run on a 1003 × 1003 × 50 grid of the RegCM5 coupled with the CLM4.5 using 16 GPU with 16 CPU processes which is 4 nodes on the Booster partition on Leonardo. Two conspicuous call stacks represent two main hotspots. The lone `__c_mset16_avx` frame constitutes 19% of the on-CPU time while the callstack starting on the `mod_capecin_getcape` frame constitutes 37%. 18

2.7 Flame graph for the 1-day simulation run for the RegCM5 coupled with the CLM4.5 land model after removing the `hotspot 1`. The call site for the `getcape` subroutine in the subroutine `output` inside the module `mod_output` now takes up a significant 54% of the CPU sampling done by `perf`. 21

2.8 Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run using 6 GPU + 16 CPU cores on (4 Booster nodes) on a 1003×1003×50 grid and a 1-day simulation. There is substantial performance improvements after removing, the `__c_mset16_avx` hotspot achieving up to 1.77× speedup relative to the initial performance. This GPU-enabled run is still outperformed by the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes. 21

3.1 Flame graph for the 1-day simulation run for the RegCM5 coupled with the CLM4.5 after offloading `getcape` subroutine using OpenACC. The resource configuration for this run is 16 GPUs with 16 CPU processes using 1003 × 1003 × 50 grid size. In the absence of the previous hotspots, new dominant stack plateaus labeled `Hotspot 3` taking up about 11% of the on-CPU time and `Hotspot 4` taking up 8%, both come to light. 28

3.2 Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run (16 GPU + 16 CPU cores on 4 Booster nodes) on a 1003×1003×50 grid on per 1 model day. Further substantial performance improvement is gained after offloading the call site for `getcape` subroutine, achieving up to 4.96× speedup relative to the initial performance. This GPU-enabled run now outperforms the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes. 28

3.3 Flame graph for the 1-day simulation run on a 1003 × 1003 × 50 grid of the RegCM5 in a coupled configuration with the CLM4.5 after offloading the call site for `interp1d_r8` and `heatindex` subroutines. The resource configuration for this run is 16 GPU with 16 CPU cores which is 4 nodes on the Booster partition on Leonardo. Visually this flame graph appears more compact compared to that in figure 3.1. 32

3.4 Performance progression of RegCM5-CLM4.5 coupled configuration for the GPU-enabled run using 16 GPUs with 16 CPU cores on 4 Booster nodes on a 1003×1003×50 grid per 1 model day. Further performance improvement is gained after offloading the call sites for `interp1d` and `heatindex` subroutines, achieving up to 7.10× speedup relative to the initial performance. This GPU-enabled run now outperforms the CPU-only baseline (blue) which runs on 800 MPI processes on 8 DCGP nodes. 32

3.5 Performance for the coupled RegCM5–CLM4.5 simulations on a 1003×1003×50 grid across varying simulation lengths (1–14 model days). The CPU-only baseline uses 800 MPI ranks on 8 DCGP nodes, while GPU-enabled runs use 16 GPUs and 16 MPI ranks on 4 Booster nodes. Three GPU implementations are compared: (i) OpenACC, (ii) Fortran do concurrent, and (iii) the combined OpenACC + do concurrent configuration. Annotated values denote speedup factors relative to the CPU-only baseline for each duration. 33

4.1 Full performance progression of the RegCM5-CLM4.5 coupled configuration on 4 Booster nodes (16 GPUs and 16 CPU cores) on 1003 × 1003 × 50 grid per 1 simulation day. The final implementation achieves a 14.17× speedup relative to the initial code. 35

4.2 Strong-scaling performance of the GPU-enabled RegCM5–CLM4.5 coupled configuration on a 1003×1003×50 grid for 7 model days across MareNostrum 5 ACC, Leonardo Booster, and ORFEO H100. (a) Wall-clock time as a function of GPU count. (b) Strong-scaling efficiency relative to the 4-GPU case. All systems show substantial speedup from 4 to 16 GPUs, with apparent superlinear scaling on MareNostrum 5 and Leonardo. Beyond this range, efficiency degrades as communication overheads become increasingly significant . . 36

4.3 Wall-clock time for the RegCM5–CLM4.5 coupled configuration on a 1003 × 1003 × 50 grid for 7 model days, comparing CPU-only runs on the Leonardo DCGP partition with GPU-enabled runs on the MareNostrum 5 ACC partition. The CPU-only configuration uses 2, 4, and 8 DCGP nodes (200, 400, and 800 CPU cores), while the GPU-enabled configuration uses 1, 2, and 4 ACC nodes (4, 8, and 16 CPU cores with 4, 8, and 16 GPUs, respectively). The labels above the bars indicate speedup relative to the 8-node DCGP CPU-only baseline. The results show that the GPU-enabled runs outperform the CPU-only runs, with the 4-node ACC configuration achieving the shortest wall-clock time 38

5.1 Spatial comparison of near-surface air temperature (tas) between the CPU reference and GPU-accelerated simulations at hour 72 of the 7-day experiment. The top row shows the fields values, while the bottom row shows the corresponding absolute and relative differences. Differences remain below 10 K and 0.5% over the entire domain. 41

5.2 Temporal statistics of near-surface air temperature (tas) differences generated between the CPU-only and GPU-offloaded simulations over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics. 42
5.3 Spatial comparison of near-surface specific humidity (`huss`) from runs before and after GPU porting at hour 72 of the 7-day experiment. The top row shows the spatial map of the fields, while the bottom row shows the map of absolute and and relative differences. Across the domain, absolute differences remain below $5{\times}10^{-3}\,\mathrm{kg\,kg^{-1}}$ and relative differences below 10%. 43
5.4 Temporal statistics of near-surface specific humidity (`huss`) differences between the baseline and GPU-offloaded simulations over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics. 44
5.5 Spatial comparison of surface pressure `ps` computed before and after GPU-porting at hour 72 of the 7-day experiment. The top row shows the absolute fields, while the bottom row shows the corresponding absolute and relative differences. Differences remain below 350 Pa and 0.07% over the entire domain. 45
5.6 Temporal statistics of surface pressure (ps) differences computed before and after GPU-offloading over the 7-day experiment. The top row shows hourly domain-mean absolute and relative errors, while the bottom row shows the corresponding daily aggregated metrics. 46

References

[1] Christoph Schär, Oliver Fuhrer, Andrea Arteaga, Nikolina Ban, Christophe Charpilloz, Salvatore Di Girolamo, Laureline Hentgen, Torsten Hoefler, Xavier Lapillonne, David Leutwyler, Katherine Osterried, Davide Panosetti, Stefan Rüdisühli, Linda Schlemmer, Thomas C. Schulthess, Michael Sprenger, Stefano Ubbiali, and Heini Wernli. Kilometer-scale climate models: Prospects and challenges. *Bulletin of the American Meteorological Society*, 2020.

[2] Daniel Klocke, Claudia Frauen, Jan Frederik Engels, Dmitry Alexeev, René Redler, Reiner Schnur, Helmuth Haak, Luis Kornblueh, Nils Brüggemann, Fatemeh Chegini, et al. Computing the full earth system at 1km resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 125–136, 2025.

[3] German Climate Computing Center (DKRZ). EERIE — Eddy Rich Earth System Models. https://www.dkrz.de/en/projects-and-partners/projects-1/eerie, 2026. Accessed: 2026-01-25.

[4] Erich Strohmaier, Jack Dongarra, Horst Simon, Martin Meuer, and Hans Meuer. TOP500 supercomputer list – november 2012. https://www.top500.org/lists/top500/2012/11/, 2012. Accessed: 2025-12-08.

[5] Tom Deakin and Timothy G. Mattson. *Programming Your GPU with OpenMP: Performance Portability for GPUs*. Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 2023.

[6] Sreeram Potluri, D. Bureddy, Hao Wang, K. Tomko, and D. K. Panda. Design and implementation of gpu-aware mpi. In *Proceedings of the IEEE International Conference on Parallel Processing (ICPP)*. IEEE, 2013.

[7] NVIDIA Corporation. Gpudirect rdma: A direct path from gpu memory to the network. Technical report, NVIDIA, 2013.

[8] NVIDIA Corporation. An introduction to cuda-aware mpi, March 2013. Accessed: 2025-12-08.

[9] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 4th edition, 2022.

[10] TOP500.org. The TOP500 list of the world's most powerful supercomputers. https://top500.org/, 2025. Accessed: 2025-12-08.

[11] Association for Computing Machinery (ACM). ACM Gordon Bell Prize for Climate Modelling. https://awards.acm.org/bell-climate, 2026. Accessed: 2026-01-30.

[12] Filippo Giorgi. Thirty years of regional climate modeling: where are we and where are we going next? *Journal of Geophysical Research: Atmospheres*, 124(11):5696–5723, 2019.

[13] Filippo Giorgi, Erika Coppola, Graziano Giuliani, James M. Ciarlo', Emanuela Pichelli, Rita Nogherotto, Francesca Raffaele, Piero Malguzzi, Silvio Davolio, Paolo Stocchi, and Oxana Drofa. The fifth generation regional climate modeling system, regcm5: Description and illustrative examples at parameterized convection and convection-permitting resolutions. *Journal of Geophysical Research: Atmospheres*, 128(6):e2022JD038199, 2023.

[14] Nellie Elguindi, Xunqiang Bi, Filippo Giorgi, Badrinath Nagarajan, Jeremy Pal, Fabien Solmon, Sara Rauscher, Ashraf Zakey, Travis O'Brien, Rita Nogherotto, and Graziano Giuliani. *Regional Climate Model RegCM: Reference Manual, Version 5.0.* Trieste, Italy, 2023.

[15] RegCM Development Team. *RegCM User Guide.* Abdus Salam International Centre for Theoretical Physics (ICTP), 2023. Version 5.0. User manual for the Regional Climate Model (RegCM).

[16] Sunita Chandrasekaran and Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies.* Addison–Wesley Professional, Boston, MA, USA, 1st edition, 2017.

[17] Rob Farber. *Parallel Programming with OpenACC.* Morgan Kaufmann, Boston, MA, USA, 1st edition, 2016.

[18] OpenACC-Standard.org. *The OpenACC® Application Programming Interface*, version 3.4 edition, June 2025. Accessed: 2026-01-14.

[19] OpenACC Organization. *OpenACC Programming and Best Practices Guide*, April 2022. Version April 2022.

[20] Andreas Herten. Many cores, many models: Gpu programming model vs. vendor compatibility overview. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1019–1026, 2023.

[21] John Hubbard, Gonzalo Brito, Chirayu Garg, Nikolay Sakharnykh, and Fred Oh. Simplifying GPU Application Development with Heterogeneous Memory Management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/, August 2023. NVIDIA Developer Blog. Accessed: 2026-01-28.

[22] Mark Harris. Unified memory for cuda beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, June 2017. NVIDIA Developer Blog.

[23] Graham Lopez, Jeff Larkin, Anastasia Stulova, Mathew Colgrove, and Jeff Hammond. Simplifying gpu programming for hpc with nvidia grace hopper superchip. https://developer.nvidia.com/blog/simplifying-gpu-programming-for-hpc-with-the-nvidia-grace-hopper-superchip/, November 2023. NVIDIA Developer Blog.

[24] NVIDIA. *NVIDIA HPC Compilers User's Guide*, 2025.

[25] Miko M. Stulajter, Ronald M. Caplan, and Jon A. Linker. Can fortran's 'do concurrent' replace directives for accelerated computing? In Sridutt Bhalachandra, Christopher Daley, and Verónica Melesse Vergara, editors, *Accelerator Programming Using Directives*, pages 3–21, Cham, 2022. Springer International Publishing.

[26] Ronald M Caplan, Miko M Stulajter, and Jon A Linker. Acceleration of a production solar mhd code with fortran standard parallelism: From openacc to 'do concurrent'. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 582–590. IEEE, 2023.

[27] Allison A Wing, Kevin A Reed, Masaki Satoh, Bjorn Stevens, Sandrine Bony, and Tomoki Ohno. Radiative–convective equilibrium model intercomparison project. *Geoscientific Model Development*, 11(2):793–813, 2018.

[28] K. W. Oleson, D. M. Lawrence, G. B. Bonan, B. Drewniak, M. Huang, C. Koven, S. Levis, F. Li, W. Riley, Z. Subin, S. Swenson, P. Thornton, A. Bozbiyik, R. A. Fisher, C. Heald, E. Kluzek, J.-F. Lamarque, P. J. Lawrence, L. Leung, and Z. Yang. Technical description of version 4.5 of the community land model (clm). Technical report, National Center for Atmospheric Research (NCAR), 2013.

[29] Matteo Turisini, Mirko Cestari, and Giorgio Amati. Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications. *Journal of large-scale research facilities (JLSRF)*, 9(1):A186, 2024.

[30] Hans Hersbach, Bill Bell, Paul Berrisford, and et al. The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, 146(730):1999–2049, 2020.

[31] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel NetCDF: A scientific high-performance I/O interface. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, page 39. IEEE Computer Society, November 2003.

[32] Fabio Banchelli, Marta Garcia-Gasulla, Filippo Mantovani, Joan Vinyals, Josep Pocurull, David Vicente, Beatriz Eguzkitza, Flavio C. C. Galeazzo, Mario C. Acosta, and Sergi Girona. Introducing MareNostrum5: A european pre-exascale energy-efficient system designed to serve a broad spectrum of scientific workloads. *arXiv preprint*, abs/2503.09917, 2025.

[33] Barcelona Supercomputing Center (BSC). Marenostrum 5 supercomputer. https://www.bsc.es/marenostrum/marenostrum-5. Accessed: March 23, 2026. MareNostrum 5 is a EuroHPC pre-exascale supercomputer hosted at the Barcelona Supercomputing Center, featuring Intel Xeon Platinum CPUs and NVIDIA Hopper GPUs with InfiniBand NDR interconnects, with a peak computational throughput of hundreds of petaflops and multi-petabyte memory and storage capacity.

[34] Area Science Park HPC Center. Orfeo hpc computational resources and partition information. https://orfeo-doc.areasciencepark.it/HPC/computational-resources/?h=partition#logical-partitions, 02 2026. Accessed: 2026-02-23.

[35] Willow Ahrens, James Demmel, and Hong Diep Nguyen. Algorithms for efficient reproducible floating point summation. *ACM Transactions on Mathematical Software (TOMS)*, 46(3):1–49, 2020.

[36] Thomas Ludwig. Bitwise reproducibility (with exascale machines). https://www.cs.fsu.edu/~nre/nre-2019/presentations/03-Ludwig.2019-06-20-ISC-Reproducibility.pdf. Accessed: 2025-11-16.

[37] Filippo Giorgi and Xunqiang Bi. A study of internal variability of a regional climate model. *Journal of Geophysical Research: Atmospheres*, 105(D24):29503–29521, 2000.

[38] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[39] Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for hpc and deep learning applications. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 170–179. IEEE, 2024.

[40] Benjamin Antunes and David RC Hill. Reproducibility, replicability and repeatability: A survey of reproducible research with a focus on high performance computing. *Computer Science Review*, 53:100655, 2024.

[41] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[42] Juan Escobar, Philippe Wautelet, Joris Pianezze, Florian Pantillon, Thibaut Dauhut, Christelle Barthe, and Jean-Pierre Chaboureau. Porting the meso-nh atmospheric model on different gpu architectures for the next generation of supercomputers (version mesonh-v55-openacc). *Geoscientific Model Development*, 18(9):2679–2700, 2025.

[43] perfwiki. Linux kernel profiling with `perf`: Sampling with `perf record`. https://perfwiki.github.io/main/tutorial/#sampling-with-perf-record. Accessed: 2025-11-16.

[44] Brendan Gregg. Flame Graphs. https://www.brendangregg.com/flamegraphs.html. Accessed: 2025-11-16.