University Politehnica of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Master’s Thesis

WRF Acceleration and Coupling with FLEXPART

by

Valentin Marcu

Supervisor: Associate Prof. Dr. Eng. Emil Slușanschi,

Bucharest, July 2014

Contents

List of Figures

List of Tables

Abstract

This paper discusses various software solutions, such as CUDA and OpenACC, used to accelerate one of the most computationally intensive modules of the Weather Research and Forecasting Model (WRF), which is the Rapid Radiative Transfer Module (RRTM). Although manual implementations of accelerated code proved to be faster and more stable, automatic and directive-based generation of accelerated routines has a lot of potential for growth and adoption by the entire weather prediction community.

Obtaining faster weather forecasts with WRF facilitates the coupling with a particle dispersion model called FLEXPART. The latest version of FLEXPART-WRF is briefly discussed in this paper, along with some sample plots resulted after running the coupled model with a Black Sea - centered domain.

Chapter 1

Introduction

Modern numerical weather and climate prediction models are expected to provide faster, more accurate and long-term forecasts, but traditional CPU-based clusters and their suport for large-scale coarse-grain parallelism are becoming insufficient and inefficient in terms of power consumption, cooling, reliability and cost. Instead, more and more weather and ocean modeling applications are searching for ways to effectively offload their computationally intensive modules to various accelerators, such as the latest architectures based on graphics processing units (GPUs), thus exploiting their abundant fine-grained parallelism at much lower costs and with support for peta-scale computing.

WRF and FLEXPART are among the most advanced, flexible and stable models in their fields (regional weather forecast and particle dispersion simulation), with large development communities and significant support for parallelization. Therefore, many efforts have been made in the recent years in order to improve these models and thus obtain faster, more accurate forecasts and simulations, with higher resolution domains.

The next chapter presents some basic WRF and FLEXPART features, especially in terms of parallelism and domain mapping to the hardware configuration, along with some state-of-the-art software tools used to help accelerate WRF on GPUs. Also, results obtained in the recent years are briefly discussed. Next, the hardware and software configuration is presented, with emphasis on the GPU architectures and the way they are connected with the CPUs. Detailed explanations on how the Rapid Radiative Transfer Model (RRTM) was accelerated on GPUs are provided in the Implementation chapter. Results of this optimizations are also shown and interpreted, as well as a case study of FLEXPART-WRF coupling on a domain comprising the Black Sea region.

Finally, conclusions are drawn and some potentially useful software technologies are briefly discussed in terms of their future impact in WRF and FLEXPART.

Chapter 2

State of the Art and Related Work

2.1 WRF

WRF (Weather Research and Forecast) Model is a mesoscale numerical weather prediction code designed for both operational forecasting and the research community with a broad spectrum of applications and multiple physics options.

WRF allows researchers the ability to conduct simulations reflecting either real data or idealized configurations. WRF provides operational forecasting that is flexible and computationally efficient, while offering the advances in physics, numerics, and data assimilation contributed by the research community [UCA12].

The current WRF model has two dynamic cores; the Advanced Research WRF (ARW) and the Nonhydrostatic Mesoscale Model (NMM). In addition to the dynamic core, several other elements of the WRF model are user configurable. These elements include the domain (i.e., location, resolution, and nesting) and physical parameterizations. Ultimately, the resulting model forecasts will vary depending on how the model is configured by the user [NCA11].

2.1.1 Main dataflow

The WRF Preprocessing System (WPS) is used primarily for real-data simulations. Figure 2.1 shows its main functionalities: [UCA12]

- degribbing and interpolating meteorological data from another model to this simulation domain (ungrib.exe);
- defining simulation domains (geogrid.exe)

- interpolating terrestrial data (such as terrain, landuse, and soil types) to the simulation domain (metgrid.exe)

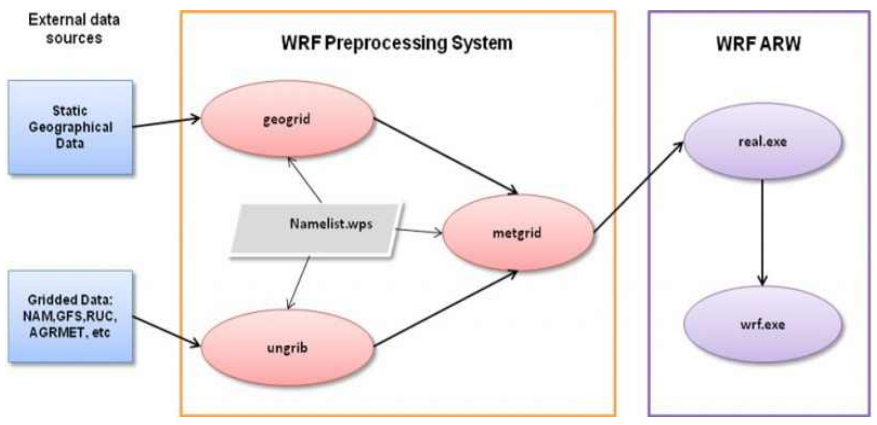


Figure 2.1: WRF flowchart

The Advanced Research WRF (ARW) solver is the key component of the modeling system, which is composed of several initialization programs for idealized, and real-data simulations (real.exe), and the numerical integration program (wrf.exe).

The key features of the WRF model include [NCA11]:

- Regional and global applications
- Fully compressible nonhydrostatic equations with hydrostatic option
- Runge-Kutta 2nd and 3rd order time integration options
- Lateral boundary conditions (for real cases: specified with relaxation zone)
- Two-way nesting with multiple nests and nest levels

The output (generated by wrf.exe) is in NetCDF format, which is widely used to store meteorological data, being nowadays compatible with almost all numerical weather forecast models.

2.1.2 Nesting domains

Nesting is a form of mesh refinement that allows costly higher resolution computation to be focused over a region of interest, as shown in Figure 2.2.

Nests in WRF are nonrotated and aligned like in Figure 2.3, so that parent mesh points are coincident with a point on the underlying nest, which eliminates the need for more complicated generalized regridding calculations. Nest configurations are specified at run-time through the namelist [YHK08].

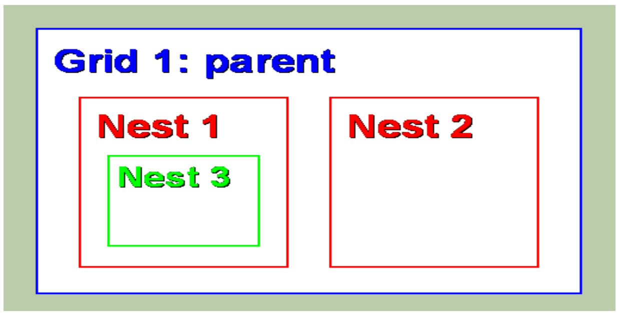


Figure 2.2: Nesting domains principle

A two-way nested run is a run where multiple domains at different grid resolutions are run simultaneously and communicate with each other: The coarser domain provides boundary values for the nest, and the nest feeds its calculation back to the coarser domain. The model can handle multiple domains at the same nest level (no overlapping nest), and multiple nest levels (telescoping) [NCA11].

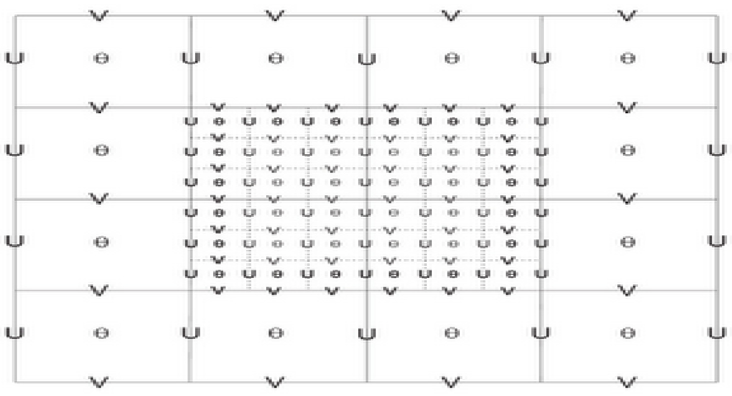


Figure 2.3: Nest versus parent domain resolution

WRF generates separate output files for each domain for which the simulation is set.

2.1.3 Domain decomposition

Model domains are decomposed for parallelism on two-levels, as shown in Figure 2.4:

- Patch: section of model domain allocated to a distributed memory node
- Tile: section of a patch allocated to a shared-memory processor within a node; this is also the scope of a model layer subroutine

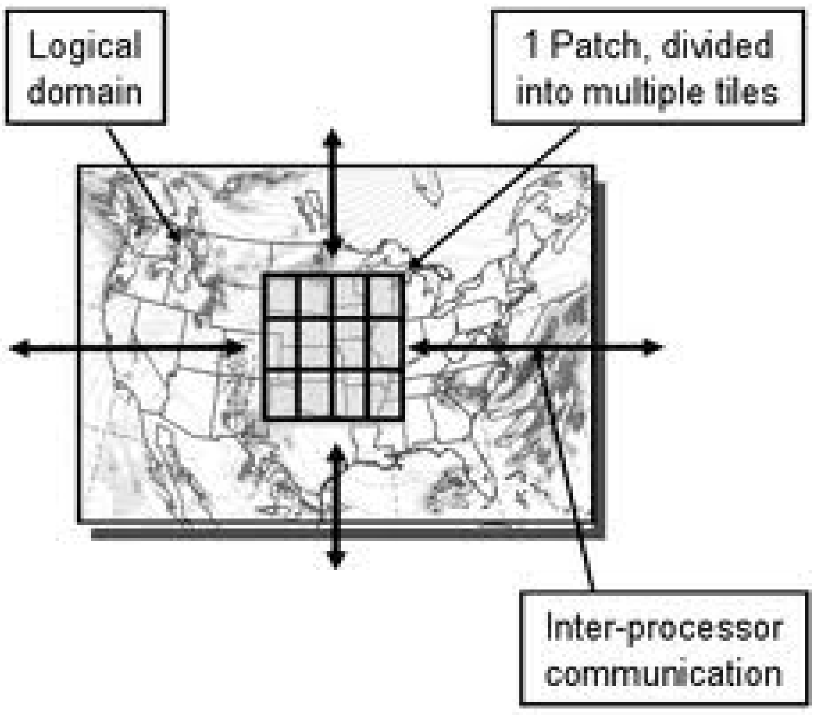

Logical domain
1 Patch, divided into multiple tiles
Inter-processor communication

Figure 2.4: Domain decomposition

A flexible approach for parallelism is achieved through a two-level decomposition in which the model domain may be subdivided into patches that are assigned to distributed-memory nodes and then may be further subdivided into tiles that are allocated to shared-memory processors within a node.

This approach addresses all current models for parallelism (single processor, shared memory, distributed memory, and hybrid) and also provides adaptivity with respect to processor type: tiles may be sized and shaped for cache blocking or to preserve maximum vector length.

Model layer subroutines are required to be tile callable, that is, callable for an arbitrarily sized and shaped subdomain. All data must be passed through the argument list (state data) or defined locally within the subroutine [JM04].

Thus, the WRF software architecture and two-level decomposition strategy provides a flexible, modular approach to performance portability across a range of different platforms, as well as promoting software reuse. It will facilitate use of other framework packages at the WRF driver layer as well as the reverse, the integration of other models at the model layer within the WRF framework [YHK08].

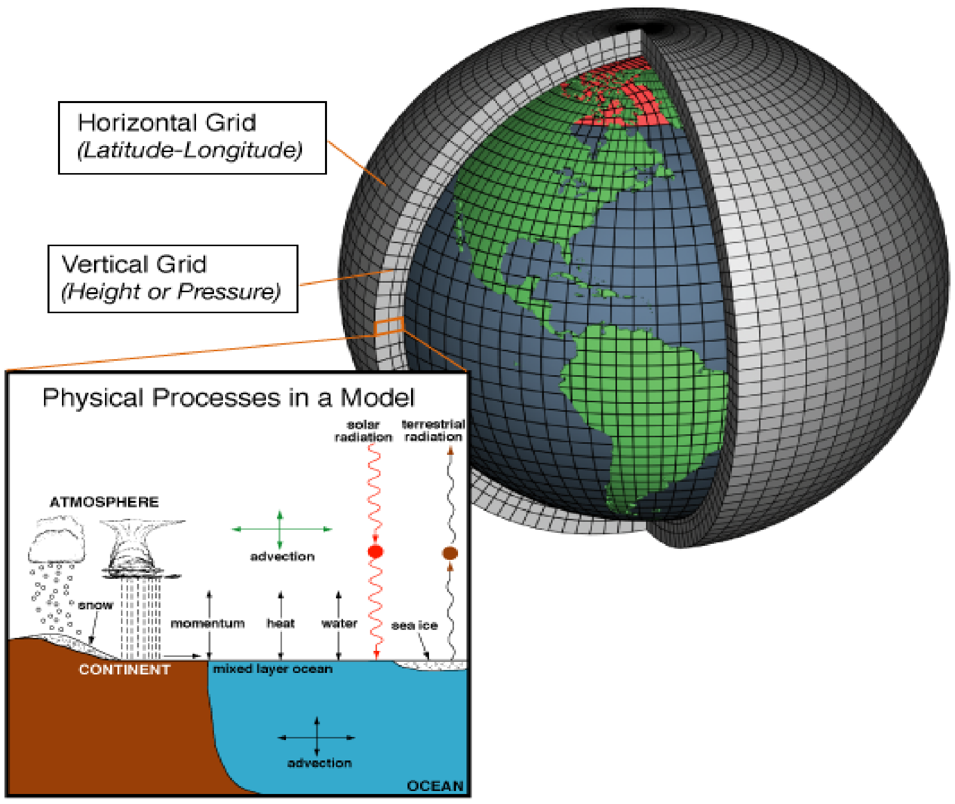


Figure 2.5: WRF radiative transfers accross horizontal and vertical grids

2.1.4 RRTM Overview

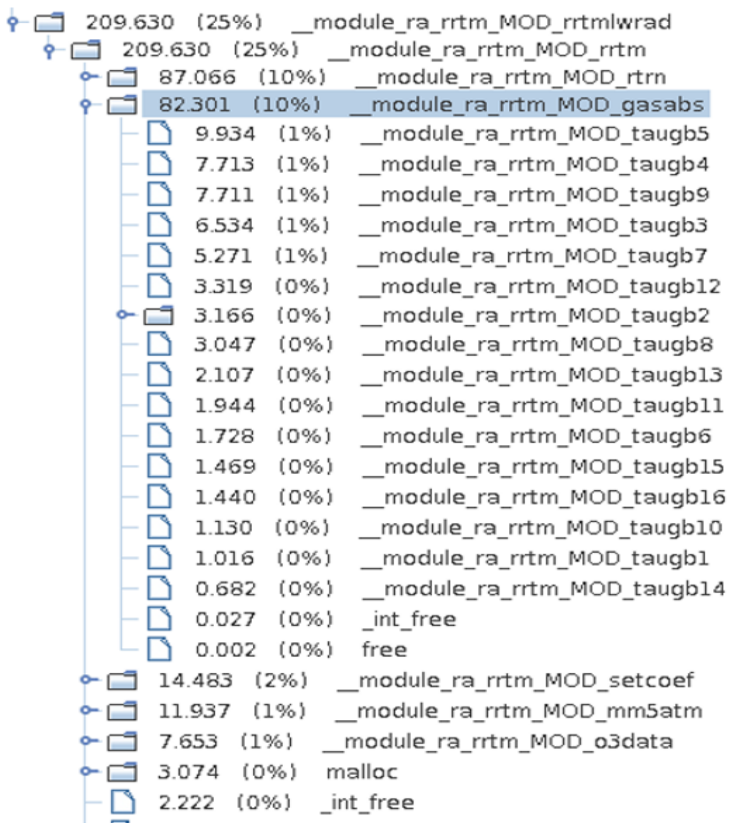
The Rapid Radiative Transfer Model (RRTM) is an accurate scheme using look-up tables for effciency. It accounts for multiple (16) spectral bands, trace gases, and microphysics species [G.R11].

As stated earlier, a WRF domain is a geographic region of interest partitioned in a 2-dimensional grid parallel to the ground. The 2D grid has multiple levels, corresponding to various vertical heights in the atmosphere.

As shown in Figure 2.5, for each grid point, radiative transfer computation proceeds along the vertical column for that point. The number of cells in a vertical column corresponds to the number of levels in the grid. The memory footprint is large with approximately 40 single-precision floating point variables per grid cell. [Mic11]

At the highest level, the model calls an initialization routine rrtminit(), and then calculates the long-wave radiation transfer by calling the routine rrtmlwrad(). [G.R11]

The call graph presented in Figure 2.6 was obtained using Oracle Studio Profiler on the original version of the module. It also reveales which calls are the



```
209.630  (25%)  __module_ra_rrtm_MOD_rrtmlwrad
  209.630  (25%)  __module_ra_rrtm_MOD_rrtm
    87.066  (10%)  __module_ra_rrtm_MOD_rrtn
    82.301  (10%)  __module_ra_rrtm_MOD_gasabs
      9.934  (1%)  __module_ra_rrtm_MOD_taugb5
      7.713  (1%)  __module_ra_rrtm_MOD_taugb4
      7.711  (1%)  __module_ra_rrtm_MOD_taugb9
      6.534  (1%)  __module_ra_rrtm_MOD_taugb3
      5.271  (1%)  __module_ra_rrtm_MOD_taugb7
      3.319  (0%)  __module_ra_rrtm_MOD_taugb12
      3.166  (0%)  __module_ra_rrtm_MOD_taugb2
      3.047  (0%)  __module_ra_rrtm_MOD_taugb8
      2.107  (0%)  __module_ra_rrtm_MOD_taugb13
      1.944  (0%)  __module_ra_rrtm_MOD_taugb11
      1.728  (0%)  __module_ra_rrtm_MOD_taugb6
      1.469  (0%)  __module_ra_rrtm_MOD_taugb15
      1.440  (0%)  __module_ra_rrtm_MOD_taugb16
      1.130  (0%)  __module_ra_rrtm_MOD_taugb10
      1.016  (0%)  __module_ra_rrtm_MOD_taugb1
      0.682  (0%)  __module_ra_rrtm_MOD_taugb14
      0.027  (0%)  _int_free
      0.002  (0%)  free
    14.483  (2%)  __module_ra_rrtm_MOD_setcoef
    11.937  (1%)  __module_ra_rrtm_MOD_mm5atm
    7.653  (1%)  __module_ra_rrtm_MOD_o3data
    3.074  (0%)  malloc
    2.222  (0%)  _int_free
```

Figure 2.6: RRTM call graph and percentage of application

most time consuming and thus may be a candidate for acceleration on GPU.

In the CPU version of rrtmlwrad(), a doubly nested loop over horizontal dimensions extracts and scales one-dimensional vertical columns from the three-dimensional input arrays and then calls the radiation model subrou- tine, rrtm(), which calculates the radiative transfer one column at a time. [G.R11]

The radiation model subroutine rrtm() has these basic steps, which are executed for each 2D point in WRF's horizontal grid:

- inirad() - computes the ozone mixing ratio distribution
- mm5atm() - prepares atmospheric profiles
- setcoef(): calculates various quantities needed for the radiative transfer algorithm specific to the atmosphere
- gasabs() - calculates gaseous optical depths
- rtrn() - calculates the radiative transfer for both clear and cloudy columns

The last two routines (gasabs and rtrn) are the most computational intensive, but only gasabs(which consists of almost half of RRTM's run time, according to

Figure 2.6) is suitable for porting to GPU, due to its 16 independent subroutine calls (taugb1, ..., taugb16), each of which initializes different regions of two large 2D output arrays (PFRAC and TAUG).

Based on the various benchmarks and parallelization strategies, the impact of gasabs() call time on the entire application may significantly vary. For example, the benchmark [oA12] used in this case revealed a decreasing percentage of gasabs() routine from 20%-25% (serial or few parallel threads) to 12%-15% (significant MPI and OpenMP parallelization). More details may be consulted in Figure 5.3, in the Results section of this paper.

2.2 FLEXPART-WRF

The Lagrangian particle dispersion model FLEXPART was originally designed for calculating long-range and mesoscale dispersion of air pollutants from point sources, such that occurring after an accident in a nuclear power plant. In the meantime, FLEXPART has evolved into a comprehensive tool for atmospheric transport modeling and analysis at different scales. Its application fields were extended from air pollution studies to other topics where atmospheric transport plays a role (e.g., exchange between the stratosphere and troposphere, or the global water cycle). [FE06]

Although FLEXPART has mostly been used with input data from global meteorological models, the planetary boundary layer (PBL) turbulence parameterizations implemented are based on data obtained from small-scale field experiments, and hence are valid for the meso- and local scales. This has led to several attempts to provide a mesoscale version of FLEXPART driven by mesoscale meteorological model output, such as the Mesoscale Meteorological (MM5) model, the Weather Research and Forecasting (WRF) model, or the weather model prediction COSMO. [JB13]

Still, many features of the standard code still remain in the coupled version: [FE06]

- forward/backward dispersion;
- removal by radioactive decay, wet/dry deposition;
- mimic atmospheric trace gases;
- concentration, uncertainties, age spectra, and mass flux calculations.

The latest version of FLEXPART-WRF is 3.1 [JB13] and was released in 2013. It introduces new features such as:

- new options for using different wind data (e.g. instantaneous or time-averaged winds);

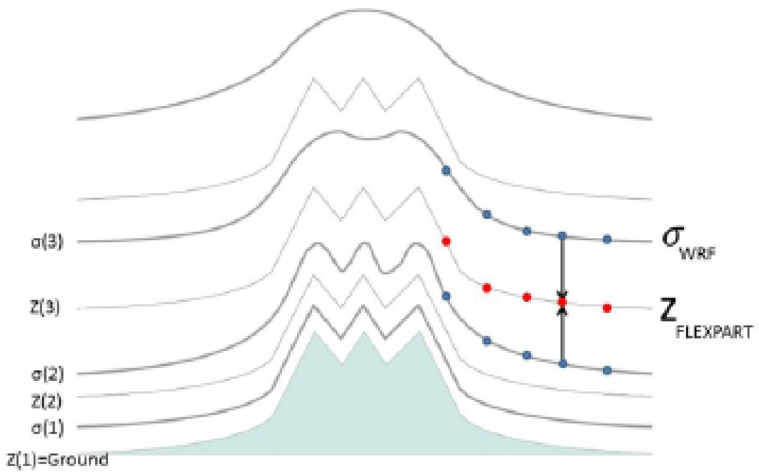


Figure 2.7: Terrain-following coordinates used in WRF and FLEXPART

- output grid projections;
- numerical parallelization for computation efficiency;
- netcdf format for output files.

Figure 2.7 is an example of how the terrain-following coordinates used in WRF (sigma) and FLEXPART (z). WRF fields are interpolated vertically onto the terrain following Cartesian coordinate z in subroutine verttransform.f90 .

To conduct a FLEXPART experiment, different meteorological fields from WRF are needed. The most important meteorological fields used to calculate the advection of air by resolved winds are the horizontal and vertical wind 3-D fields.

As shown in Figure 1.6, the geopotential is used to interpolate the WRF vertical coordinate onto the FLEXPART vertical coordinate. Also, the latitude and longitude 2-D fields are used to validate the projection calculation in FLEXPART. [JB13]

FLEXPART-WRF was programmed keeping the end-user needs in mind; it was intended for a range of different computational resources, from single CPU computers to multithread clusters with distributed memory. The source code thus consists of some common routines for both parallel and serial runs, and some specific routines either for serial or for parallel compilation. [JB13]

For example, the interpolation from the WRF coordinates onto FLEXPART coordinates, the concentration calculations, and the trajectory integrations have all been parallelized in OPENMP [Bar14b] compiler directives for parallel programming. The reading of the WRF input file could be parallelized as well using a special Network Common Data Form (NetCDF) library that handles parallel

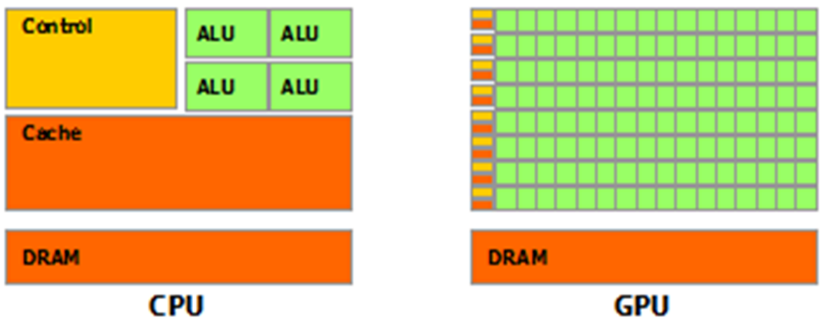


Figure 2.8: CPU vs GPU Hardware

reading. However, this has not been implemented yet. Routines with OPENMP instructions do not have a specific name, since the compiler will interpret the OPENMP instructions only when OPENMP is used. Routines marked by the mpi label, however, are the specific routines that use Message Passing Interface (MPI) [Bar14a] parallel programming in distributed memory. [JB13]

2.3 NVIDIA CUDA

2.3.1 General features

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth [Cor14b].

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as shown in Figure 2.8.

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches [Cor14b].

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA

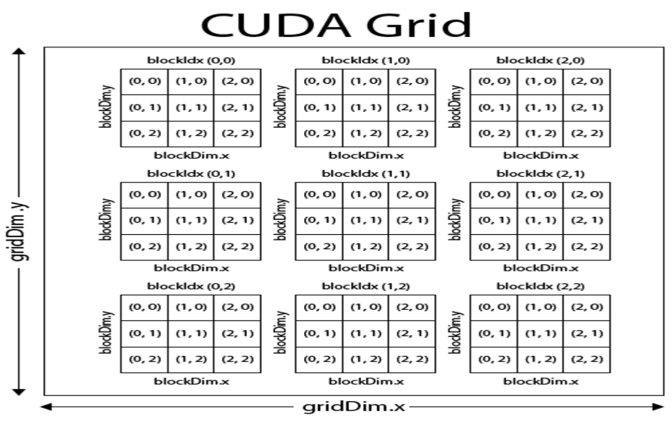


Figure 2.9: CUDA Grid

threads, as opposed to only once like regular C functions. Figure 2.9 shows how threads are grouped into blocks, which in turn are executed in a time-sharing manner within the GPU's streaming multiprocessors (SM), but under the same logical group of blocks, or grid.

2.3.2 Cuda 6

Nvidia has recently released CUDA 6, an upgrade to its proprietary GPU programming language that it says "includes some of the most significant new functionality in the history of CUDA" [Cor14b].

The five most important new features of CUDA 6 are:

- support for Unified Memory;
- CUDA on Tegra K1 mobile/embedded system-on-a-chip
- XT and Drop-In library interfaces;
- remote development in NSight Eclipse Edition;
- many improvements to the CUDA developer tools.

Unified memory implies that the underlying system manages data access and locality within a CUDA program without need for explicit memory copy calls. This benefits GPU programming in two primary ways:

- GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers;
- Data access speed is maximized by transparently migrating data towards the processor using it.

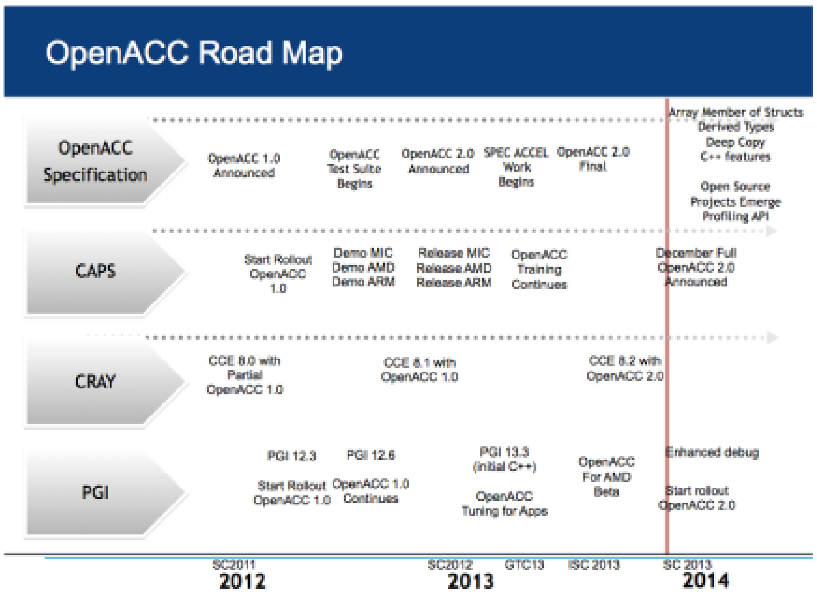

OpenACC Road Map
OpenACC Specification
CAPS
CRAY
PGI
OpenACC 1.0 Announced
OpenACC Test Suite Begins
OpenACC 2.0 Announced
SPEC ACCEL Work Begins
OpenACC 2.0 Final
Array Member of Structs Derived Types Deep Copy C++ features
Open Source Projects Emerge Profiling API
Start Rollout OpenACC 1.0
Demo MIC Demo AMD Demo ARM
Release MIC Release AMD Release ARM
OpenACC Training Continues
December Full OpenACC 2.0 Announced
CCE 8.0 with Partial OpenACC 1.0
CCE 8.1 with OpenACC 1.0
CCE 8.2 with OpenACC 2.0
PGI 12.3
PGI 12.6
Start Rollout OpenACC 1.0
OpenACC 1.0 Continues
PGI 13.3 (initial C++)
OpenACC Tuning for Apps
OpenACC For AMD Beta
Enhanced debug
Start rollout OpenACC 2.0
SC2011
2012
SC2012
2013
GTC13
ISC 2013
SC 2013
2014

Figure 2.10: OpenACC Road Map [Hem13]

In simple terms, Unified Memory eliminates the need for explicit data movement via the cudaMemcpy() routines without the performance penalty incurred by placing all data into zero-copy memory. Data movement, of course, still takes place, so a programs run time typically does not decrease; Unified Memory instead enables the writing of simpler and more maintainable code. [Cor14b]

Unified Memory features have been used to modify an accelerated module of WRF, but only for testing purposes, to estimate the local impact of some minor updates in data transfers on the performance.

2.4 OpenACC

OpenACC (for Open Accelerators) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

Like in OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using PRAGMA compiler directives and additional functions. Unlike OpenMP in versions before 4.0, code can be started not only on the CPU, but also on the GPU [Cor13].

Features of OpenACC include:

- Productivity - higher level programming model and openMP-style directives
- Portability - NVIDIA, AMD, Intel Xeon Phi

- Performance feedback - profiling results with low overhead and preliminary analysis may be obtained for each compiler-generated device function.

The execution model targeted by OpenACC API-enabled compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The device executes parallel regions, which typically contain work-sharing loops, or kernels regions, which typically contains one or more loops which are executed as kernels.

Even in accelerator-targeted regions, the host must orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the parallel region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. In most cases, the host can queue a sequence of operations to be executed on the device, one after the other.[Cor13]

One of the main features of the new version (2.0) of OpenACC is the possibility to call functions and subroutines within compute regions. OpenACC 1.0 compilers rely on inlining function and subroutine calls within compute regions. This means that unless the compiler can automatically inline a function call, the programmer must manually inline the function. This limitation proved to be difficult for applications, so OpenACC 2.0 introduces the acc routine directive, which instructs the compiler to build a device version of the function or subroutine so that it may be called from a device region. [Lar13]

The main developpers of OpenACC have already established how this standard is going to evolve in the near future [Hem13]. Figure 2.10 shows a summary of these trends.

2.5 Related Work

2.5.1 RRTM porting using PGI CUDA-Fortran

In 2011, Greg Ruetsch, Everett Phillips, and Massimiliano Fatica, from Nvidia Corporation, ported the entire RRTM module in CUDA-Fortran [G.R11], using the PGI compiler [Gro14]. Their tests revealed a 10x acceleration of RRTM routines, resulting in an approximate improvement of the entire application (WRF) of 1.2x.

2.5.2 WSM5 porting using CUDA C

John Michalakes, from National Center for Atmospheric Research, USA, ported to GPU using CUDA C another computationally-intensive module of WRF, from the Microphysics routines, called WSM5 Micro-Physics [Mic11]. The speedup for WSM5 alone was of 5x, while the obtained overall speedup of WRF was of 1.25x.

2.5.3 WSM5 accelerated with OpenACC

In 2013, W. A. Haines [Hai13] used openacc 1.0 Fortran directives to boost the performance of the WSM5 module already mentioned above. He obtained a 1.2x speed-up of the overall model by simply adding around 20 lines of easy-to-understand compiler directives to this physics scheme.

2.5.4 Other WRF modules accelerated using OpenACC

In 2012, CRAY [PJ12] used OpenACC v1.0 directives and their own compiler suite to accelerate various 3D loops within the dynamics routine of WRF (the entire module consists of more than 50% of the total run time of WRF, depending on the chosen benchmark). Their tests (which were run only on the dynamics module, not on the entire model) revealed a speedup of approximately 12x, but without measuring serial, I/O, MPI or exposed PCIe times [PJ12]. To our knowledge, no updates have been made since then regarding this test case.

Chapter 3

Hardware and Software Configuration

This chapter describes the HW and SW configurations used in the project, with emphasis on the GPU architectures and the way they are connected with the CPUs.

3.1 Hardware Configuration

We used two sets of hardware configuration, one for implementation and testing purposes and another for large-scale benchmarking and analysis. We also used the testing environment to implement and test at a small scale some new features of CUDA 6, due to its hardware support (compute capability 3.0).

3.1.1 The NCIT Cluster

The NCIT (National Center for Information Technology) [Pol13] of Politehnica University of Bucharest is composed of various Intel Xeon, AMD and Tesla processors.

As far as GPU architectures are concerned, there is an IBM iDataPlex dx360M3 system, consisting of 4 nodes, each with 2 NVIDIA Tesla M2070 GPUs and 2 6-core Intel Xeon X5650 CPUs with Hyper-Threading technology. In other words, up to 24 threads or processes can run concurrently on each node.

```
// sample record from /proc/cpuinfo

vendor_id       : GenuineIntel
cpu family      : 6
```

```
model           : 44
model name      : Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
cpu MHz         : 2666.588
cache size      : 12288 KB
siblings        : 12
cpu cores       : 6
```

An important metric of the GPUs is the number of Streaming Multiprocessors (SM) of each device. This configuration consists of 4 nodes with 2 GPUs of 14 SMs each.

```
// output of pgaccelinfo command (pgi suite)

Device Name:                     Tesla M2070
Device Revision Number:          2.0
Global Memory Size:              5636554752
Number of Multiprocessors:       14
Number of Cores:                 448
Concurrent Copy and Execution:   Yes
Total Constant Memory:           65536
Total Shared Memory per Block:   49152
Registers per Block:             32768
Warp Size:                       32
Maximum Threads per Block:       1024
```

3.1.2 Nvidia GeForce650M

The second system consisted of an ASUS N56VZ laptop, with Intel Core i7 3610QM Processor (quad-core with Hyper Threading) and an NVIDIA GeForce GT 650M GPU with 2GB DDR3 VRAM. The reason for this choice was that the GPU has compute capability 3.0 and therefore supports CUDA 6 API.

```
// output of pgaccelinfo command (pgi suite)
Device Name:                     GeForce650M
Device Revision Number:          3.0
Global Memory Size:              2GB
Number of Multiprocessors:       2
Number of Cores:                 384
Concurrent Copy and Execution:   Yes
Total Constant Memory:           65536
Total Shared Memory per Block:   49152
Registers per Block:             65536
Warp Size:                       32
```

```
Maximum Threads per Block:     1024
```

3.2 Software Configuration

Table 3.1 summarizes the software configuration essentials used during the development of the mentioned applications.

C / Fortran Compiler 1	GNU C/Fortran v. 4.4.6
C / Fortran Compiler 2	PGI Compiler v. 11.7
C / Fortran Compiler 3	PGI Compiler v. 14.1 (trial)
NetCDF	v. 4.1.3
MPI	OPENMPI v. 1.6
Profiler 1	Oracle Parallel Studio v. 12.3
Profiler 2	Nvidia Visual Profiler (NVVP) v.6.0
Operating System 1	Linux RHEL6, kernel version 2.6.32
Operating System 2	Linux OpenSUSE 13.1, kernel version 3.12.1

Table 3.1: Software configuration

A sample script used to launch multiple jobs running various configurations is provided below.

```
QUEUE=ibm-dp.q
for MPI in 1 2 3 4; do
  for OMP in 1 2 4 6 8; do
    qsub -hold_jid gasabs -N gasabs -cwd \
    -q $QUEUE -pe openmpi*1 $MPI -b n s.sh $MPI $OMP
  done
done
```

Chapter 4

Project implementation

This chapter presents implementation details of various GPU acceleration attempts of the previously mentioned Rapid Radiative Transfer Model (RRTM) module.

4.1 RRTM Acceleration using CUDA C + gfortran

4.1.1 Identifying the most promising areas of parallelization

As stated earlier, two of RRTM's routines (gasabs and rtrn) are the most computationally intensive, but only gasabs(which consists of almost half of RRTM's run time, according to Figure 2.6) is suitable for immediate porting to GPU, due to its 16 independent subroutine calls (taugb1, ..., taugb16), each of which initializes different regions of two large 2D output arrays (TAUGB and PFRAC).

Early attempts to accelerate the rtrn() subroutine resulted in very poor results, due to the loop dependencies which affected both the rtrn()'s nested do-loops.

Therefore, the other computationally intensive subroutine of RRTM, gasabs(), was translated from Fortran 90 to CUDA C, with promising results.

4.1.2 Using the hardware

In order to fully utilize the hardware resources of the testing environment (the front-end processor gpu.grid.pub.ro with its 4 nodes and the corresponding queue: ibm-dp.q), we combined some smaller taugb subroutines in order to be executed by the threads within the same thread block. The goal was to reduce the total number of blocks to 14, which is the number of a streaming multiprocessors (SM) within a GPU device. Thus, as shown in Figure 4.1, each thread block may be

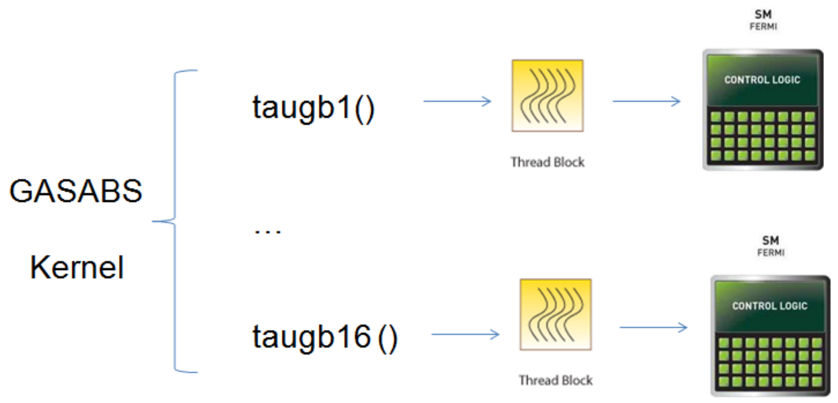


Figure 4.1: GASABS kernel

scheduled on a different SM and all the blocks may run concurrently on the same GPU.

Moreover, in order to maximize the use of the entire cluster, as well as the concurrent execution possibilities, each CPU thread (openMP) was configured to use a separate CUDA stream, which in turn was uniformly spread accross all the GPUs within a node.

```
#define DEVICES_PER_NODE 2
...
int tid = omp_get_thread_num();
...
 CHECK(cudaSetDevice(tid % DEVICES_PER_NODE));
CHECK( cudaStreamCreate(&stream[tid]) );
```

Finally, MPI was used to communicate between different nodes of the cluster (one MPI process per node).

4.1.3 Data organization

Porting the code from Fortran to CUDA C proved to be challenging especially in terms of memory usage, due to the enormous amount of data used by the module. Fortunately, most of this data was constant (lookup tables) and needed to be transfered only at the beginning of the application. Lines 6-8 from the following listing show how each constant array is copied into GPU's constant memory only when the first_call flag is set to true, which happens only when the function is first called.

Moreover, since in the original fortran implementation all these constant arrays were accessed globally (without being passed as arguments to subroutines), the CUDA host function had to reference them with their global names as seen by the gfortran linker (line 4 from the listing below).

```
1  #define FULL(A) __module_ra_rrtm_MOD_##A
2  ...
3  // global name of absa1, as seen by the compiler
4  extern "C" float FULL(absa1)[5*13*NG1];
5  ...
6  #define TODEVCONST(A,s,ss) if(first_call[tid]==true) \
7    { CHECK( cudaMemcpyToSymbolAsync (A##_global,A##_host,
8      (s)*sizeof(float),0,H2D,stream[tid][ss]) ); }
```

The other arrays, due to their large number, could not be passed by reference to gasabs_host() and had to be grouped together in structures, based on their purpose: in, out or inout.

Separate structures of data had to be allocated for constant data used by each taugbX_gpu() function. Since this data proved to be the largest, we paid extra attention when accessing it from the kernel and we observed a lot of uncoalesced accesses of some of the arrays. Therefore, we decided to send to GPU the transposed version of these problematic array, thus obtaining an impressive speedup of the entire kernel.

```
if(first_call[tid]==true)
  { // declare on cpu a transposed version of absa12
  float absa12_t[9*5*13*NG12];
  int i,j;
  // copy the contents of absa12 to its transpose
  for(i=0; i<NG12; i++){
    for(j=0;j<9*5*13; j++){
      absa12_t[i+NG12*j] = FULL(absa12)[j+(9*5*13)*i];
    }
  //send to GPU the transpose
  TODEV3(absa12_t,12,9*5*13*NG12);
  }
```

4.1.4 Other optimizations

Asynchronous data copies between host and device were imperative in order to maintain a good scalability of the program. Below are listed some other important optimizations that we successfully implemented:

- Overlap data transfers and kernel execution using streams

- Transpose some input arrays before sending them to GPU's global memory for coalesced accesses from threads (minimize global memory transactions)
- Allocate memory on GPU only at first kernel call
- Transfer data to/from GPU only when needed (constant data is transferred only once) and in a minimum of transactions (maximize bandwidth)
- set environment to use more L1 cache memory than shared memory

4.2 RRTM Acceleration using CUDA C v. 6.0 + gfortran

Unified Memory creates a pool of managed memory shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU. [Cor14a]

Figure 5.5 from the Results chapter of this thesis shows that the calls to cudaMemcpyDeviceToHost() take almost half of the time spent on CUDA API calls. We used unified memory features of CUDA 6 as an atempt to let the system improve data transfers from device to host.

The following listing shows how we used cudaMallocManaged() and cudaStreamAttachMemAsync() calls to inform the compiler that a particular CPU array is to be used by the device.

```
outData *outdata;
if(first_call[tid]==true)
{
 global_outdata_host[tid] = new outData();
 // tell the compiler to deal with the array from now on
 cudaMallocManaged((void **)&global_outdata_host[tid], \
    sizeof(outData));
 // make sure data transfers will be async
 cudaStreamAttachMemAsync(stream[tid], \
    global_outdata_host[tid]);
 cudaStreamSynchronize(stream[tid]);
}
outdata = (outData *) global_outdata_host[tid];
```

Unfortunately, although the results obtained with this approach were correct, the overall execution time increased for all the tested scenarios.

4.3 RRTM Acceleration using OpenACC 2.0 + pgi

One of the most important features introduced in the 2.0 version of OpenACC was the possibility to call other functions and subroutines from within the accelerator regions. We implemented this feature as follows:

- placed "!$acc kernels" or "!$acc parallel" directives higher in the call graph (inside subroutines which call other subroutines)
- added "!$acc routine" directives at the beginning of each subroutine which was called from within compute regions
- marked each global function used in the accelerator regions as "!$acc managed"

Unfortunately, the latest PGI compiler (14.6) still cannot compile !$acc managed routines unless the arrays that are marked in this way are allocatable. Since all the global data used by the RRTM subroutines is composed of statically allocated arrays, the compilation was unsuccessful.

A second approach was to automaticaly inline all the subroutines called within each accelerator region, by using the -Minline compiler flag. By using this technique we obtained correct results only for small accelerated regions, while the larger regions returned either incorrect or slow results.

Chapter 5

Case studies and Results

This chapter describes the benchmark used to test the correctness and scalability of RRTM's acceleration on GPUs (together with the corresponding results), as well as a case study of FLEXPART-WRF coupling on a domain comprising the Black Sea region.

The most important performance metrics that were measured were:

- total run time
- speedup
- impact of RRTM's routines on the entire WRF model

5.1 Results of RRTM Acceleration on GPU

5.1.1 Europe Benchmark

Recognizing the need for a diverse but unified set of benchmark cases in the WRF modeling user community, the User-Oriented WRF Benchmarking Collection represents a joint effort of the University of Alaska's Arctic Region Supercomputing Center and Vienna's University of Natural Resources and Life Sciences Institute of Meteorology [oA12].

The vision behind this effort is that users are often interested in determining how much their particular model set up will cost, from a computational perspective. This information will influence decisions in setting up model domains for operational and research WRF simulations, along with aiding in decisions related to necessary hardware and software resources for various model configurations.

The primary function of this initiative is to provide the WRF user community with a centralised repository of easy-to-run benchmark cases for a variety of needs, and a reporting facility for users to submit their results and to query results of

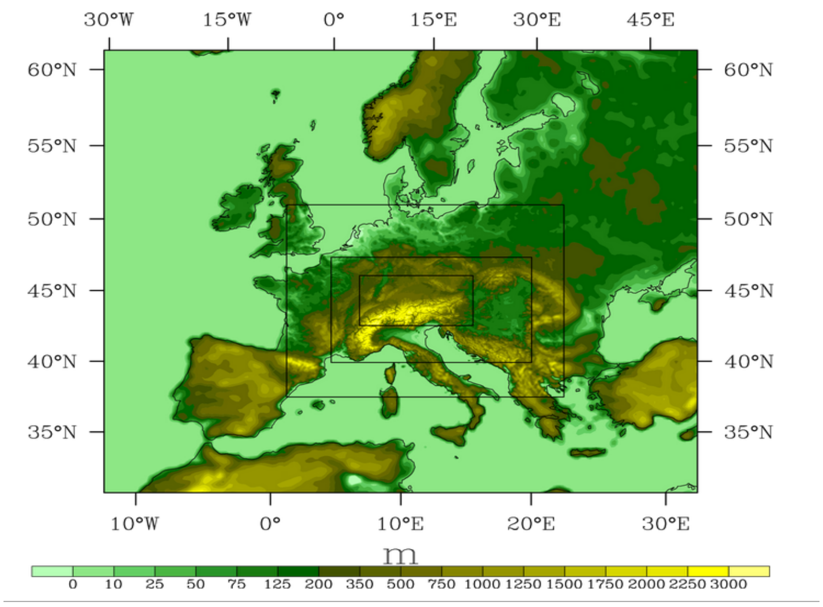
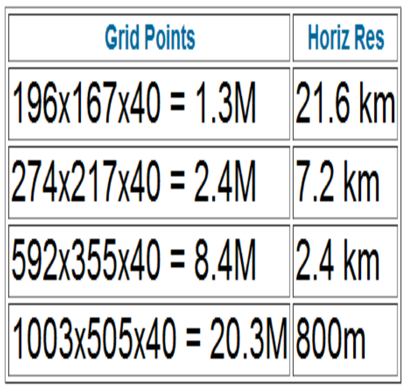


Figure 5.1: The Europe Benchmark domains

other users. Each set of benchmarks is intended to address a particular niche of user needs, and is presented in a relatively consistent manner so that users may use common procedures and tools to utilise the collection [oA12].



Grid Points	Horiz Res
196x167x40 = 1.3M	21.6 km
274x217x40 = 2.4M	7.2 km
592x355x40 = 8.4M	2.4 km
1003x505x40 = 20.3M	800m

Figure 5.2: The Europe Benchmark grids

European WRF Benchmark Suite is intended to represent typical user needs in complicated and demanding multi-nest and multi-shaped configurations used in regional modeling applications. This case is set up with an emphasis on the complete but more complex evaluation of real-world model runs that feature not only nesting structures, but numerous model parameterisations, input/output issues, etc [oA12].

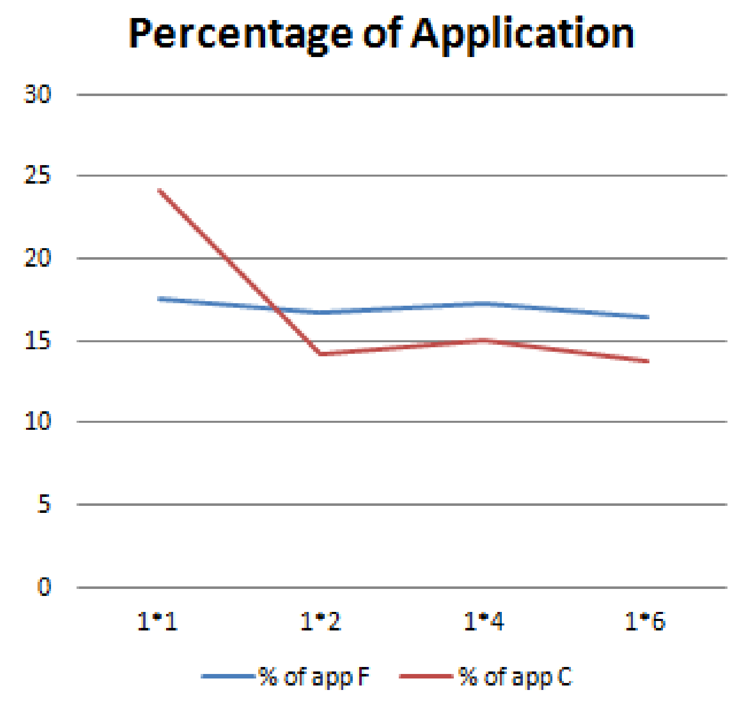


Figure 5.3: Comparative impact on WRF (X axis: MPI x OMP)

Figure 5.1 shows the geographical areas comprised by the benchmark's domains, while Figure 5.2 presents each domain's resolution and number of points in latitude*longitude*vertical dimensions. Only the outer-most domain was used during testing.

5.1.2 RRTM To CUDA C

First of all, using Oracle Studio profiler, we compared the impact that each implementation has on WRF total execution time. For example, Figure 5.3 compares the durations of the gasabs() calls with respect to the entire application (as percentages).

Secondly, we compared the total WRF execution times with the compiler optimization flags enabled and without debug options. Figure 5.4 presents the behavior of the application when the gasabs() subroutine was accelerated on GPU versus the original Fortran implementation. Also, Figure 5.5 shows only the CUDA API calls, revealing that the cudaMemcpy routines have almost the same execution times as the kernel.

The last scenario revealed a speedup of our implementation between 1.0 and 1.1, with a maximum obtained when on each node there were 2 openMP threads running, in which case both GPU devices on each node were fully utilized and

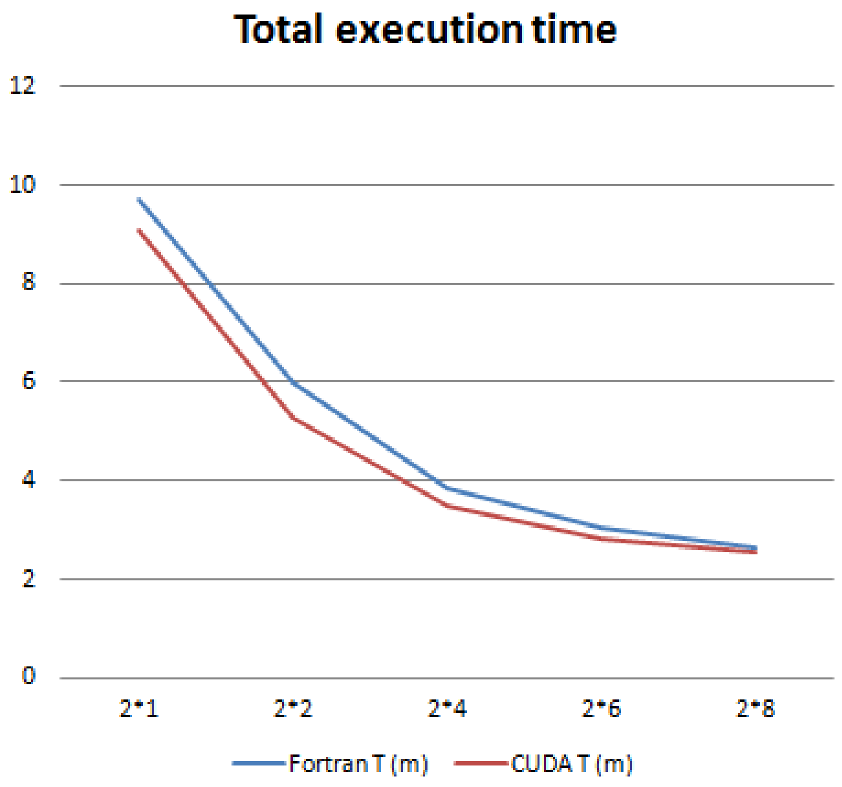
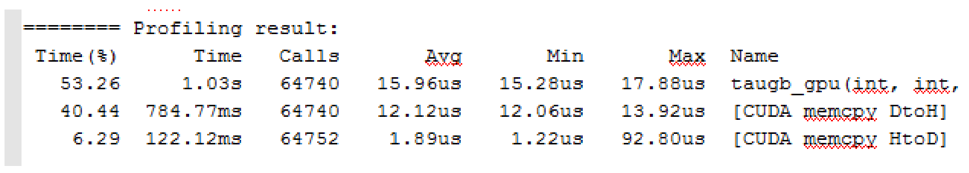


Figure 5.4: Comparative times of WRF total execution times (X axis: MPIxOMP)

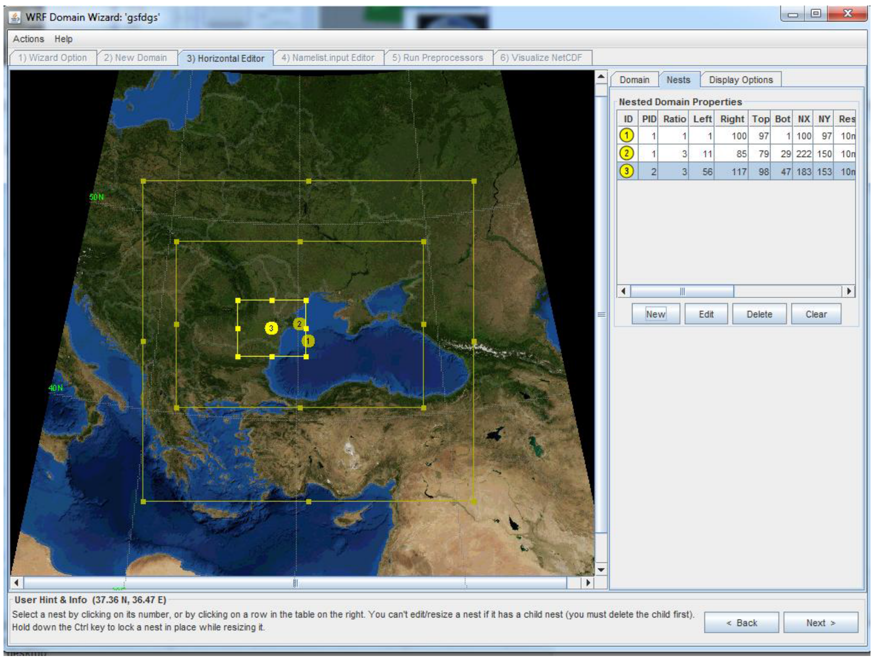


```
======== Profiling result:
 Time(%)      Time    Calls       Avg       Min       Max  Name
   53.26     1.03s    64740   15.96us   15.28us   17.88us  taugb_gpu(int, int,
   40.44  784.77ms    64740   12.12us   12.06us   13.92us  [CUDA memcpy DtoH]
    6.29  122.12ms    64752    1.89us    1.22us   92.80us  [CUDA memcpy HtoD]
```

Figure 5.5: Profiling with CUDA nvprof

no threads were waiting for resources in order to run their kernels.

As a reminder, the gasabs() impact on WRF's total execution time does not exceed 20% in the original implementation. According to Amdahl's law, the maximum theoretical speedup which can be obtained from accelerating this routine is approximately 1.2 - 1.25, depending on the benchmark. Thus, our implementation obtained approximately half of the maximum theoretical speedup.



Nested Domain Properties

ID	PID	Ratio	Left	Right	Top	Bot	NX	NY	Res
1	1	1	1	100	97	1	100	97	10n
2	1	3	11	85	79	29	222	150	10n
3	2	3	56	117	98	47	183	153	10n

User Hint & Info (37.36 N, 36.47 E)

Select a nest by clicking on its number, or by clicking on a row in the table on the right. You can't edit/resize a nest if it has a child nest (you must delete the child first).
Hold down the Ctrl key to lock a nest in place while resizing it.

Figure 5.6: Black Sea domain generated using WRF Domain Wizard

5.2 FLEXPART-WRF Coupling

5.2.1 Domain Generation

A simple way to generate any domain for a future WRF testcase setup is WRF Domain Wizard. It is in fact a graphical user interface (GUI) for the WRF Preprocessing System (WPS). It enables users to easily define and localize domains (cases) by selecting a region of the Earth and choosing a map projection. Users can also define nests using the nests editor, edit namelist.input, run the WPS programs (geogrid, ungrib, and metgrid) through the GUI, and visualize the NetCDF output.[Adm12] Figure 5.6 show such a domain, corresponding to the Black Sea region.

Of course, users also need to set many other physics parameters, which are required to completely and accurately define a test case. WPS is required to be installed on the system, together with a ”geog” folder containing static geographical data.

Furthermore, in order to generate a valid run for WRF, a series of grib files (containing input meteorological data) must be provided to WPS. Only then will WPS be able to finish the pre-processing stage of WRF and permit the model to run and generate forecasts. Of course, FLEXPART-WRF settings also need to be adjusted accordingly with the new scenario.

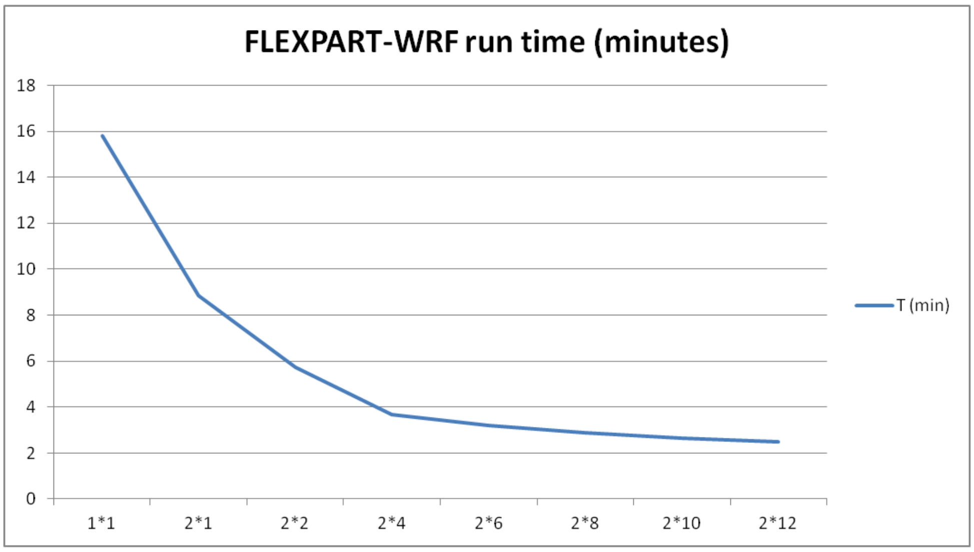


Figure 5.7: FLEXPART-WRF Scalability (MPI x OpenMP)

5.2.2 Parallelization tests

First of all, the sample benchmarks provided by the FLEXPART-WRF developer were run on the NCIT cluster, leading to approximately the same results as those mentioned in the technical report. Figure 5.9 shows how the application behaves when it is run on up to 24 cores of the same node.

5.2.3 Mapping WRF forecast data over Black Sea domain

To demonstrate the possibility of running FLEXPART-WRF on any predefined domain, we simply used the forecasts generated by WRF for the European benchmark [oA12] as if they were corresponding to the Black Sea region and then we ran the coupled model FLEXPART-WRF with one of the predefined particle release setup. In other words, we ”mixed” the two main input data sources of the coupled model in order to obtain virtual trajectories, particle concentrations and dry depositions in the Black Sea region. Figure 5.9 and Figure **??** show some of the plots obtained with the mentioned settings. The simulated period of time was of 36 hours from the particle release moment.

In order to obtain a realistic particle dispersion forecast for this region, both WRF data for this region and specific release information is needed (for example, a release setup corresponding to a volcano eruption will behave totally different than a radioactive release from a nuclear power plant).

However, this mix of inputs for the coupled model facilitates testing various emission scenarios in variable atmospheric conditions over the Black Sea.

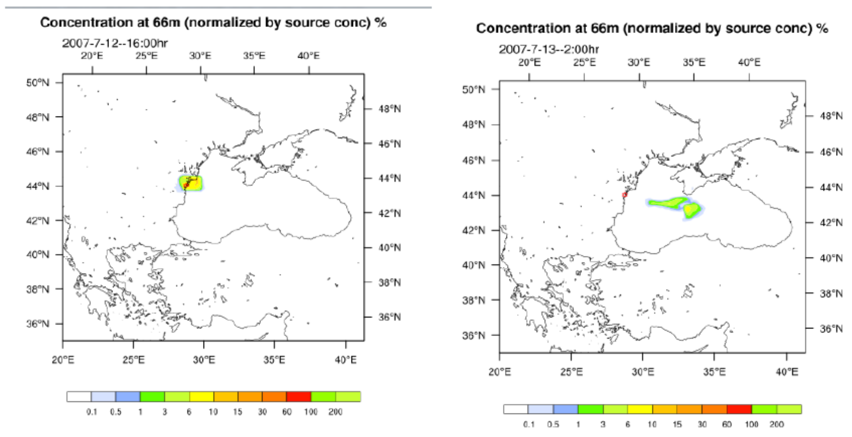

Concentration at 66m (normalized by source conc) %
2007-7-12--16:00hr
Concentration at 66m (normalized by source conc) %
2007-7-13--2:00hr
20°E 25°E 30°E 35°E 40°E
50°N 48°N 46°N 44°N 42°N 40°N 38°N 36°N
0.1 0.5 1 3 6 10 15 30 60 100 200

Figure 5.8: FLEXPART-WRF Particle Concentrations (a)

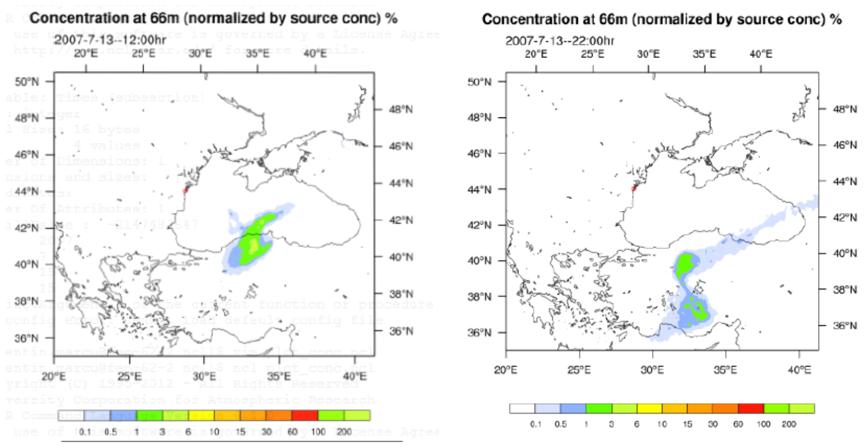

Concentration at 66m (normalized by source conc) %
2007-7-13--12:00hr
Concentration at 66m (normalized by source conc) %
2007-7-13--22:00hr
20°E 25°E 30°E 35°E 40°E
50°N 48°N 46°N 44°N 42°N 40°N 38°N 36°N
0.1 0.5 1 3 6 10 15 30 60 100 200

Figure 5.9: FLEXPART-WRF Particle Concentrations (b)

Chapter 6

Conclusions and Future Work

Porting Fortran to CUDA C tends to result in better performance than using accelerator directives, but is very time consuming and prone to various implementation errors and vulnerabilities.

The obtained overall speedup of WRF is not very high (1.05x - 1.1 x), but taking into account that the theoretical maximum speedup (obtained by diminishing RRTM's impact close to zero) does not exceed 1.3 and that we only implemented a subset of RRTM routines, the results seem satisfactory.

This performance can be easily improved once the new directive-based accelerators are developped, permitting the programmer to address some higher level modules of WRF. OpenACC 2.0 accelerator and CUDA 6+ API are the most promising tools in terms of speeding up applications accross multiple and even heterogenous high performance computing environments. OpenMP version 4.0 [Boa13] also contains directives intended to offload computation on various accelerators, but the standard is still under development and hasn't been adopted yet by any compilers.

The new version of FLEXPART-WRF uses an efficient parallelization strategy and is compatible with a wider range of input data, domain configurations and output formats, which makes it increasingly attractive for usage in various regional particle dispersion simulations.

Bibliography

[Adm12] National Oceanic & Atmospheric Administration. Wrf domain wizard. http://esrl.noaa.gov/gsd/wrfportal/DomainWizard.html, 2012.

[Bar14a] Blaise Barney. Message passing interface (mpi). Technical report, Lawrence Livermore National Laboratory, 2014.

[Bar14b] Blaise Barney. Openmp. Technical report, Lawrence Livermore National Laboratory, 2014.

[Boa13] OpenMP Architecture Review Board. Openmp application program interface version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[Cor13] OpenACC Corporation. The openacc application programming interface v.2.0. http://www.openacc.org, 2013.

[Cor14a] NVIDIA Corporation. 5 powerful new features in cuda 6. http://devblogs.nvidia.com/parallelforall/powerful-new-features-cuda-6/, 2014.

[Cor14b] NVIDIA Corporation. Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2014.

[FE06] Jerome D. Fast and Richard C. Easter. A lagrangian particle dispersion model compatible with wrf. Technical report, Pacific Northwest National Laboratory, Boulder, Colorado, USA, 2006.

[G.R11] G.Reutsch. Gpu acceleration of the long-wave rapid radiative transfer model in wrf using cuda fortran. Technical report, NVIDIA Corporation, 2011.

[Gro14] The Portland Group. Pgi compilers and tools. http://www.pgroup.com/index.htm, 2014.

[Hai13] Wesley Adam Haines. Acceleration of the weather research & forecasting (wrf) model using openacc. case study of the august 2012 great arctic cyclone. Technical report, The Ohio State University, USA, 2013.

[Hem13] Nicole Hemsoth. Openacc broadens appeal with gcc compiler support. http://www.hpcwire.com/2013/11/14/openacc-broadens-appeal-gcc-compiler-support/, 2013.

[JB13] A. Stohl J. Brioude, D. Arnold. The lagrangian particle dispersion model flexpart-wrf version 3.1. Technical report, Cooperative Institute for Research in Environmental Sciences, University of Colorado, Boulder, Colorado, USA, 2013.

[JM04] D. Gill J. Michalakes, J. Dudhia. The weather research and forecast model: Software architecture and performance. Technical report, National Center for Athmospheric Research, Boulder, CO, 2004.

[Lar13] Jeff Larkin. 7 powerful new features in openacc 2.0. https://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/, 2013.

[Mic11] John Michalakes. Gpu acceleration of numerical weather prediction. Technical report, National Center for Atmospheric Research, 2011.

[NCA11] NCAR. *Advanced Research WRF User's Guide*, January 2011.

[oA12] University of Alaska. User-oriented wrf benchmarking collectio. http://weather.arsc.edu/WRFBenchmarking/index.html, 2012.

[PJ12] Jim Schwarzmeier Pete Johnsen. Porting wrf to openacc. Technical report, CRAY Inc., 2012.

[Pol13] Politehnica University of Bucharest. *The NCIT Cluster Resources User's Guide v. 4.0*, 2013.

[UCA12] UCAR. Wrf user's guide. http://www.mmm.ucar.edu/wrf/users/docs/user_guide_V3/users_guide_chap8.htm, 2012.

[YHK08] John Michalakes Ying-Hwa Kuo, Joseph B. Klemp. Mesoscale numerical weather prediction with the wrf modal. Technical report, National Center for Athmospheric Research (NCAR)., 2008.