Porting the MPI Parallelized LES Model PALM to Multi-GPU Systems – An Experience Report

Helge Knoop[1(✉)], Tobias Gronemeier[1], Christoph Knigge[1], and Peter Steinbach[2]

[1] Institute of Meteorology and Climatology, Leibniz Universität Hannover, Hannover, Germany
knoop@muk.uni-hannover.de
[2] Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
https://palm.muk.uni-hannover.de

Abstract. The computational power of graphics processing units (GPUs) and their availability on high performance computing (HPC) systems is rapidly evolving. However, HPC applications need to be ported to be executable on such hardware. This paper is a report on our experience of porting the MPI + OpenMP parallelized large-eddy simulation model (PALM) to a multi-GPU environment using the directive based high level programming paradigm OpenACC. PALM is a Fortran-based computational fluid dynamics software package, used for the simulation of atmospheric and oceanic boundary layers to answer questions linked to fundamental atmospheric turbulence research, urban climate, wind energy and cloud physics. Development on PALM started in 1997, the project currently entails 140 kLOC and is used on HPC farms of up to 43200 cores. The porting took place during the GPU Hackathon TU Dresden/Forschungszentrum Jülich in Dresden, Germany, in 2016. The main challenges we faced are the legacy code base of PALM and its size. We report the methods used to disentangle performance effects from logical code defects as well as our experiences with state-of-the-art profiling tools. We present detailed performance tests showing an overall performance on one GPU that can easily compete with up to ten CPU cores.

Keywords: CFD · GPU · HPC · LES · MPI · OpenACC · PGI · Porting

1 Introduction

High performance computing (HPC) systems for scientific applications are rapidly gaining size, complexity and adoption in various fields of academia and industry. Recently, an increasing number of these systems provide access to graphics processing units (GPU) [1], adding additional computational power to the available CPU based system performance. Equipping small clusters or even workstations with multiple GPUs enables access to considerable computational power, even for, e.g., small businesses without access to HPC installations.

M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 508–523, 2016.
DOI: 10.1007/978-3-319-46079-6_35

Applications running on such equipped systems need to be capable of dealing with the GPU architecture in order to benefit. Examples of affected applications are machine learning applications [11], molecular dynamics simulations [18] and (besides many others) large-eddy simulation (LES) models. The LES method is a computational fluid dynamics (CFD) simulation technique which is computational expensive and thus its effective use is currently still limited to HPC systems. However, GPUs are a potential enabler for the operational application of this technique in smaller businesses and institutions e.g., for urban planning or wind-energy site assessment. In order to exploit the power of GPUs with existing LES models, they need to be ported to such a computer architecture.

This paper summarizes our experiences during the porting process of the **pa**rallelized **l**arge-eddy simulation **m**odel (PALM) for atmospheric and oceanic flows to a GPU environment. In order to minimize portability loss and porting workload, the directive-based high-level programming model OpenACC [17] was chosen. The porting took place during the one-week GPU Hackathon TU Dresden/Forschungszentrum Jülich in Dresden, Germany, in 2016. During the Hackathon, we were supported by three experienced mentors. With this report we intend to provide aid and guidance for other GPU porting endeavors on code bases similar to the LES technique.

The article itself is structured in four parts: First, PALM and its state-of-art prior to the Hackathon is described. Second, a chronological report is given on the efforts at the OpenACC Hackathon 2016 and afterwards. This is followed by a detailed performance analysis and at the end, we reflect on the progress we made during the Hackathon, discuss technical aspects and draw conclusions for their influence on our future code development.

2 PALM

PALM is an atmospheric CFD application and in particular an LES model to simulate the turbulent flows of atmosphere and ocean. PALM has been applied to answer questions linked to a variety of topics including fundamental atmospheric turbulence research (e.g., [8,14]), urban climate modeling (e.g., [5,12]), wind energy [7] and cloud physics [6]. PALM solves the Boussinesq-approximated Navier-Stokes equations on a discrete three-dimensional (3D) grid for a time dependent flow. A detailed description of the physics used in PALM can be found in [13].

PALM is written in Fortran95 [2] with some Fortran 2003 [20] extensions. It is optimized for running on massively parallel computer architectures. In order to distribute data and work across multiple cores and nodes the message passing interface (MPI) and the directive-based high level programming paradigm OpenMP are used. Parallelization is realized by a two-dimensional (2D) domain decomposition of the underlying Cartesian grid. The computational domain, which consists of a large cuboid representing a portion of the atmosphere, is divided into small vertical columns and each core solves the equations inside one of these vertical columns. After each time step, data situated at the borders of the columns are exchanged with the neighboring cores using MPI. The work flow of PALM is illustrated in Fig. 1 and can be paraphrased as follows: First, the model is initialized by setting up all relevant 3D arrays and distributing necessary data to each core (initialize). Second, the time dependent loop is executed (time loop), in which the prognostic equations are solved for wind-velocity components, temperature, kinetic energy, humidity and others. Following the prognostic equations, a Poisson equation for the perturbation pressure needs to be solved (pressure solver) during each iteration of the time loop. To do this, the data arrays have to be transformed via a fast Fourier transformation (FFT), which requires several calls of MPI routines due to the domain decomposition. At the end of each loop cycle data output is done by calling the output routines. After the time loop is finished the simulation gets finalized and additional output is done.

PALM has basic integrated profiling capabilities that are helpful for analysis during porting as well as for monitoring performance regressions between releases. The compute time consumed by each individual routine is measured using a built-in function named `cpulog`. It measures the execution time between two positions in the code by using the intrinsic Fortran function `SYSTEM_CLOCK`. At the end of a simulation, a list of time measurements and calling counts containing the most time-consuming routines is saved.

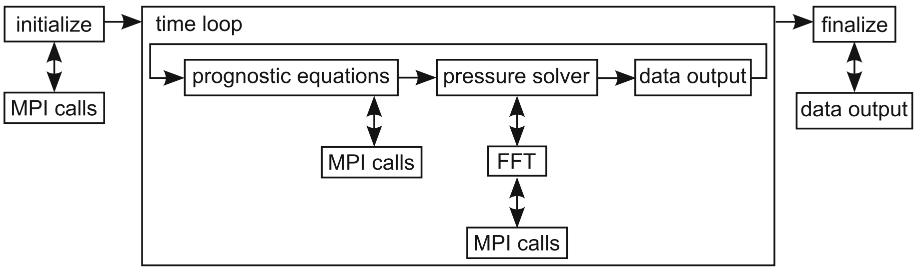

Fig. 1. Schematic work flow of PALM showing the most important parts of the model.

3 Porting PALM

3.1 Preparations

Optimizing an existing production-ready HPC source code base for performance should not be underestimated with regard to a high number of influential parameters (social, technical and design based) and time-consuming subtasks. Thus, automating a majority of the necessary steps to validate or falsify optimization hypotheses is crucial. The common work flow for this can be modeled as:

1. compile,
2. run or profile,

3. validate results,
4. interpret runtime,
5. update code and return to 1. If needed.

As the first three points do not require human intervention, automation of compilation, execution and validation yields a high return-on-investment. The automation does not only allow individual developers or developer groups to move forward autonomously, but it also ensures that the application logic is retained throughout the process. In order to achieve this, a set of shell scripts was created for building PALM, executing it with predefined parameter sets, and validating the results thereof.

To automate the compilation step, a build script was designed to flexibly and transparently adapt to the HPC environment of Taurus at the Center for Information Services and High Performance Computing (ZIH) Dresden, where the porting work took place. It allows to switch compilers and their parameters, switch between available MPI libraries and enable or disable profiling and debugging tools. The actual translation of source code to the PALM binary (using GNU make [21]) is performed in a setup-specific build directory to allow parallel testing of different setups. All these customizing options are available through the aforementioned build setup and each setup is labeled individually. The different build setups can then be chosen by providing the label as an argument to the build script, e.g.:

```
$ ./COMPILE palm_setup_pgi161_openmpi_scorep
```

where PALM is built using the PGI compiler [19] using the OpenMPI library and instrumenting the code with Score-P [16] markers. While we are aware that alternative multi-purpose build engines [3] are available, such as cmake [15] and others, this lightweight and custom approach allowed us to rapidly adapt and identify improvements to code translation and to be flexible in terms of the profiling tool-chain to be used. Given the time constraints of the Hackathon, no incentive urged us to invest resources in refactoring this build mechanism.

Our execution script is designed in a similar manner and based on the same keywords as the build script. Execution setups can be defined and customized for all possible steering parameters of the batch system. It is essential to have an execution setup available that allows a rapid testing of all parts of the LES model which are relevant to the profiling and code optimization process. The build and run script provided a common reference for the team. These scripts can be considered as an essential building block of the optimization process. Also, the PALM integrated automatic runtime measurements of all time-consuming routines (`cpulog`) are in line with this idea of receiving feedback quickly.

Further, a set of execution setups was created to test PALM regarding memory size and execution time. We found that a good execution setup during the porting process should complete in a fairly small walltime envelope but has a computational complexity that utilizes most of the available resources on the targeted GPU in terms of memory usage and occupancy. In other words, a balance has to be maintained between obtaining a representative sample of profiling data and yet retain a quick turn-around of optimization feedback to all developers involved. We thus agreed on two setups: a *small* one that would complete within one minute of walltime and yet force PALM to perform four iterations; and a *large* configuration that would complete within 5 min of walltime to allow a more global view on the impact of code optimization.

Finally, a lightweight automated testing process of the simulation results in terms of their correctness was introduced. For convenience, the evaluation result is summarized to check whether the execution was successful or not by textually comparing the ASCII output files produced in a given PALM run. PALM is neither equipped with a unit test suite nor are integration tests available so far. Any ambitious performance tuning of PALM should consider providing a comprehensive unit test suite commonly referred to as *test harness* [4]. This does not only ensure correct simulation results after optimizations were applied, but also exerts a high pressure to modularize the application even more, so that autonomous code modules can be extracted from the code base and be optimized independently.

3.2 Starting Position

The model PALM is optimized to run on computer architectures exposing a multi-tier cache hierarchy as well as on vector-based hardware [13]. Depending on the architecture used different branches of the code are executed to gain the best performance.

Before joining the Hackathon, PALM already contained GPU targeted code which based on the vector-optimized branch. This was mainly done by placing a data region around the whole program and adding OpenACC directives to single loops. GPU architectures are based on the parallelization paradigm "Single instruction, multiple threads" (SIMT). In hardware, this relates to the GPU executing one instruction by a group of threads at a single point in time (the hardware used for the Hackathon exposed a minimum group size of 32 threads). The more iterations a loop has, the more it benefits from SIMT architecture. In nature, a GPU is therefore much closer to a vector computer architecture than it is to a cache-optimized computer architecture. Hence, the GPU-optimized branch of PALM based on the vector-optimized branch of PALM with some slight changes to the vector-optimized branch. The GPU-optimized branch was maintained in parallel to the two already existing CPU targeted branches. However, the OpenACC-enabled code was only able to run on a single core while operating on a GPU. The goal for the Hackathon was to continue porting every routine of the program and get a fully functional version of PALM running on a multi-node multi-GPU system.

3.3 The First Unsuccessful Attempt

At the beginning of the Hackathon, it turned out that the GPU targeted routines did not produce the same results as the CPU-only routines. Therefore, during the first two days, the GPU routines were searched for source code defects (bugs) related to the existing GPU code base. This was a very time-consuming process, because the former porting turned out to be unstructured and poorly documented. Having three separate code bases made code debugging even more complicated. Also the size of PALM itself, with its 140 kLOC distributed over 472 routines made the debugging challenging. After two days trying to get the correct results within the GPU branch using the former implemented OpenACC directives, the results still differed from the CPU-only version.

Due to the difficulties regarding the partly ported code base mentioned above, it was decided to do a fresh start. All existing OpenACC directives were commented out to disable their functionality while still having them available during the upcoming second porting attempt. This helped to avoid recoding of already correctly ported parts of the code.

3.4 Starting a Structured Porting Attempt

Our porting effort from scratch focused on the GPU-optimized branch of PALM with all former OpenACC directives commented out.

The first step in a successful porting attempt is an extensive application run-time analysis using a sophisticated profiling tool. During the porting of PALM we used the Score-P measurement infrastructure, which is a highly scalable tool suite for profiling, event tracing, and online analysis of HPC applications [10]. We started several runs with different setups on multiple CPU cores using MPI in order to identify the top subroutines that consumed the most run time during the simulation. Score-P instrumentation and Vampir visualization [9] were applied. In Fig. 2, a Vampir visualization of the evolution of the PALM call stack during one cycle of the time loop is shown. It enables an easy identification of the hot-spot subroutines (marked in blue). The visualized run was performed on a single CPU core. The `time_integration` subroutine contains the whole time loop and is called directly from the main routine `palm`. During a time-loop cycle `time_integration` calls the subroutines `prognostic_equations` and `pres`. The subroutine `prognostic_equations` (marked in red) contains calls to several subroutines dealing with different terms of all required prognostic equations and the subroutine `pres` calls the pressure solver of choice. The chosen pressure solver, which is `poisfft` (marked in yellow) in our case, contains multiple FFT calls (`fft_x` and `fft_y`) and 3D array transpositions with heavy MPI communication. At the end of a time-loop cycle, the data output and other optional parts of the model, e.g., a soil model, are called.

We started this porting attempt by adding `!$acc kernels` directives to the hot-spot subroutines and kept profiling to see how the performance of the code evolved. We quickly realized that this work-flow cycle is quite time-consuming as some of the traces took 5–10 min to load in Vampir. This was mostly due to the fact that the hot-spot subroutines were called at a very high frequency on the used CPU cores and thus the number of traces exceeded an acceptable size for an undisturbed execution of Vampir.

The PGI compiler translates the OpenACC code to CUDA internally and emits CUDA PTX binary objects, which allows us to use the CUDA profiler as

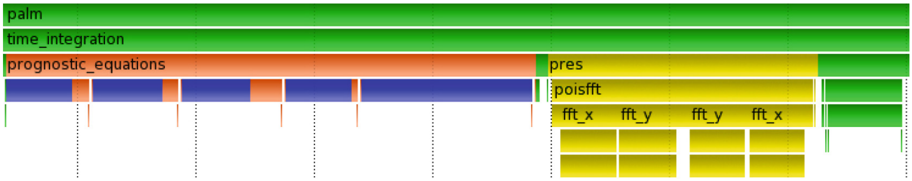

Fig. 2. Vampir screen-shot with the call-stack analysis of one cycle of the time loop in the PALM simulation. The prognostic equations (red) with their most time consuming subroutines (blue) are followed by the pressure solver (yellow). (Color figure online)

an alternative profiling tool. To realize this, the code was compiled using the options `-acc` and `-ta=tesla` to allow OpenACC interpretation, `-Minfo=acc` for OpenACC related compiler logging and `-fastsse` to enable fast SSE instructions for CPU based code.

Before profiling the application, the environment needs to be equipped with the following variables:

```
$ export COMPUTE_PROFILE=1        # 1 is on, 0 is off
$ export PGI_ACC_TIME=0           # 1 is on, 0 is off
$ export CUDA_PROFILE_LOG=./cuda_profile_out
$ export CUDA_PROFILE_CONFIG=${HOME}/cuda_prof.config
```

where `COMPUTE_PROFILE` enables the profiling (setting `PGI_ACC_TIME` would print a sum of the time needed for data movement between CPU and GPU and the time needed for computation on the GPU to the terminal). Once the application runs, it will store all relevant profiling output in `cuda_profile_out` inside the current working directory. The variable `CUDA_PROFILE_CONFIG` points to a configuration file that controls what metric is to be included in the profiling output (for more options see the online CUDA profiler documentation[1]). In our case, we added `regperthread` to the configuration file to extend the default output by the number of registers used per kernel.

For our large execution setup, which ran for 60 s on one node with one rank and one GPU, this produced a 210 MB ASCII text file. Amongst others, it contains the following information:

```
method=[ advec_u_ws_acc_2234_gpu ]
gputime=[ 1557.472 ]
regperthread=[ 160 ]
occupancy=[ 0.375 ]
```

The metrics in the above list give information about the name of the kernel which was profiled (`method`) and the time measured in microseconds spent on the GPU device during execution (`gputime`). Also the number of registers required by a kernel is given by `regperthread`, and `occupancy` gives the occupancy of the

[1] http://docs.nvidia.com/cuda/profiler-users-guide/#command-line-profiler-control.

GPU, which is used to determine how much of the computational capacity of the GPU is used by this single kernel. All of this information helped us to have a rapid turn-around frequency while doing GPU dedicated optimizations. So, rather than applying a sophisticated profiler like Score-P and Vampir, we had a very quick feedback to the changes we just made to our code. However, as the command-line based CUDA profiler can be difficult to use, one has to know exactly how to extract the needed information. Otherwise, using the NVIDIA Visual Profiler is inevitable as it contains very helpful visualizations, occupancy calculation, automatic kernel runtime analysis, etc.

Our porting approach was to add OpenACC directives to the hot-spot subroutines and wrap them into data regions. Inside a data region data are kept on the GPU and data transfer is limited to the beginning and the end of the data region as long as it is not explicitly initiated by, e.g., `!$acc update`. As porting progressed, the data regions grew and were pushed upward in the call stack, and as soon as the boundaries of two data regions collided, they were joined into one bigger data region. The code parts containing the OpenACC directives look mostly as follows:

```
!$acc data copyin( temp, u, v, w )
[...]
!$acc kernels present( temp, u, v, w )
DO  i = i_left, i_right
   DO  j = j_south, j_north
      DO  k = 1, nzt
       temp(k,j,i) = u(k,j,i) + v(k,j,i) + w(k,j,i)  !some work
      ENDDO
   ENDDO
ENDDO
!$acc end kernels

CALL completely_ported_subroutine
[...]
!$acc end data
```

where `!$acc data copyin( temp, u, v, w )` initiates the data region and copies the arrays `temp`, `u`, `v`, and `w` onto the GPU. The loops are surrounded by a kernel construct using `!$acc kernels` and `!$acc end kernels`. This enables the compiler to optimize the loop for the GPU. Additionally, `present( temp, u, v, w)` informs the compiler which variables are already present on the GPU to avoid unnecessary data transfer from the CPU to the GPU.

Organizing the data regions efficiently is essential to gain additional speedup, as the data transfer between the CPU and the GPU can impose a bottleneck. This means that data transfer should be limited to a minimum and as much data as possible should be kept on the GPU. This, however, is not always feasible. Especially data output or MPI communication requires some sort of data transfer between CPU and GPU. Therefore, particular attention was needed as the data regions arrived at the MPI calls. In order to utilize GPUs on a multiple-node setup while minimizing performance loss due to the data-transfer bottleneck, it was necessary to implement CUDA-aware MPI. CUDA-aware MPI can be realized by employing the OpenACC directive `!$acc host_data`. It essentially makes the address of data located on the GPU available on the CPU. For a `MPI_SENDRECV` call the directive can be used as follows:

```
!$acc host_data use_device( ar )
CALL MPI_SENDRECV( ar, size, MPI_REAL, left,  0,          &
                   ar, size, MPI_REAL, right, 0,          &
                   comm2d, status, ierr )
!$acc end host_data
```

The directive `!$acc host_data use_device( ar )` followed by an MPI call involving the array `ar` enables data transfer of `ar` directly between GPUs without a detour via their related host CPUs. The MPI call shown above, however, is a fairly simple example. PALM utilizes many different MPI functions in several parts of the code. MPI derived data types are heavily used in order to transfer slices of 2D and 3D arrays. These data types usually represent data that is non-contiguous in memory. We found that current MPI implementation releases like Open MPI v1.10.3 are showing a severe loss of performance as soon as non-contiguous derived data types are used in CUDA aware MPI calls. In case of 3D array slices this even resulted in a termination of the program due to a segmentation fault. Therefore we were not able to port these MPI calls to become CUDA aware. Instead we were forced to employ the OpenACC directive `!$acc update` in order to transfer the respective data to the CPU in advance of the MPI calls and back to the GPU thereafter.

```
!$acc update host( ar )
CALL MPI_SENDRECV(                                          &
    ar(nzb,nys-nbgp_local,nxl),   1, type_yz(grid_level),  &
                                  pleft,  0,               &
    ar(nzb,nys-nbgp_local,nxr+1), 1, type_yz(grid_level),  &
                                  pright, 0,               &
    comm2d, status, ierr )
!$acc update device( ar )
```

The penalty imposed by the data-transfer bottleneck greatly reduced our expected final speedup.

Finally, the FFT operations had to be ported to utilize the CUDA FFT library (cuFFT). As cuFFT functions are not available in Fortran, a C interface is required. We used the Fortran 2003 `bind` feature and the intrinsic module `ISO_C_BINDING` to make the cuFFT library available. After that work was limited to calling the `cufftPlan1D` routine to generate all required cuFFT plans which were then used in the subsequent calls of `cufftExecD2Z` and `cufftExecZ2D` for forward and backward transformation, respectively. At the end, the `cufftDestroy` function is called to release the resources allocated for the plans.

As soon as a routine was running entirely on a GPU, replacing the directives `!$acc kernels` with proper `!$acc loop` constructs enabled advanced loop tuning options. Assigning and varying the gang and vector size, we quickly realized that most of the times a simple `!$acc kernels` directive does the porting job quite well. On occasion, the data independency of loops was not detected correctly by the compiler. This is shown by the output, which is generated by using the `$ -Minfo=acc` flag of the PGI Fortran compiler. Adding some `!$acc loop independent` directives quickly solved this issue.

4 Performance Tests

With every check-in, PALM was getting faster on the GPU. In the end, the code ran and produced correct results. Within one week, we were able to port almost all the major routines of PALM to the GPU. Unfortunately, we were not able to finish all the porting work during the Hackathon. Back home we had to invest another couple of days in order to push the data region out of the main time loop.

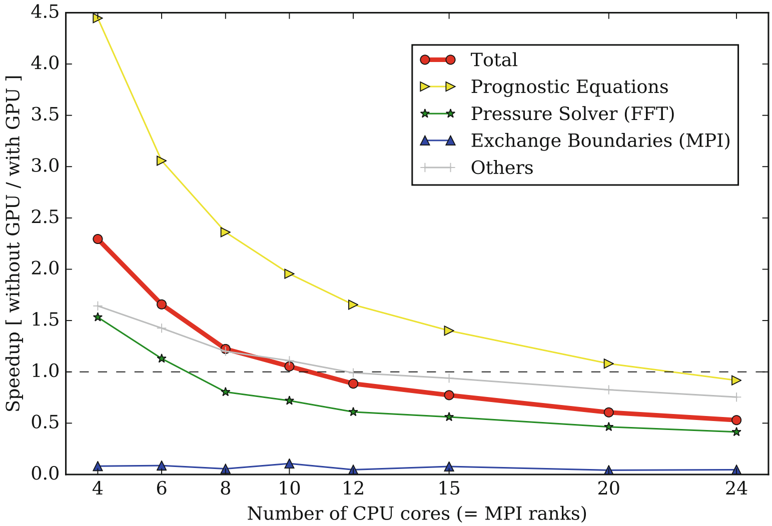

Fig. 3. The speedup factor (OpenACC-disabled runtime divided by OpenACC-enabled runtime) depending on the number of CPU cores (MPI ranks). The total speedup, the individual speedup of the tree most time consuming PALM routines and the combined speedup of the rest of the PALM routines are shown. Values greater than one (dashed line) indicate a performance gain and values smaller than one indicate a performance loss.

Finally we conducted performance tests based on a setup that filled most of the GPU memory and required a runtime of about 300 s on twenty-four CPU cores. We performed tests with the number of CPU cores ranging from four to twenty-four. Four is the minimum number of CPU cores required for the ported MPI code parts and twenty-four is the maximum allowed number of CPU cores for the setup, which still fits on one node. Each test is repeated with one GPU (OpenACC enabled) and without GPU (OpenACC disabled). The tests were conducted on a Cray XC30 at the North-German Supercomputing Alliance (HLRN). The compute node was a symmetric multiprocessing (SMP) node with four Intel Xeon 8-core SandyBridge Processors and one NVidia Tesla K40 attached. The node was exclusively used for the test and each test was repeated ten times in order to level out performance fluctuation. The tests were performed using the double precision floating point format. The results are presented in Fig. 3. The speedup factor calculated by comparing the runtime of the OpenACC-enabled tests and the OpenACC-disabled tests is shown depending on the number of CPU cores (MPI ranks). In total a solid advantage of the OpenACC-enabled runs on up to ten CPU cores can be observed. Increasing the number of CPU cores further resulted in a speedup factor of less than one which is a performance loss induced by utilizing the GPU. Using twenty-four CPU cores resulted in an OpenACC-enabled runtime that was almost doubled compared to the OpenACC-disabled runtime. In order to find the cause for this limited performance gain, a more detailed analysis is required. Figure 3 also provide individual speedup factors of the three most time consuming parts of PALM. Additionally Fig. 4 provides an overview of the individual runtime share each of these routines contribute to the total. This information is provided separately for OpenACC-disabled and OpenACC-enabled runs and has negligible dependency on the number of CPU cores used. The prognostic equations routines have the biggest share of the total runtime and they perform much better with a solid advantage of the OpenACC-enabled runs on up to twenty CPU cores. Theses routines largely consist of 3 nested loops working heavily on the big 3D data arrays and the good performance of this part of PALM is also due to the absence of MPI calls. The pressure solver on the other hand is performing very poorly on the GPU. Running on more than six MPI ranks already results in a performance loss if OpenACC is enabled. The heavy use of the `MPI_ALLTOALL` function in order to transpose 3D arrays across the 2D domain decomposition could be an explanation but in this case the profiling shows that CUDA-aware MPI is working. About three quarters of the pressure solver runtime is dedicated to the transpositioning and one quarter is dedicated to the cuFFT calls. As the runtime share of the pressure solver gradually increases with high numbers of MPI ranks, the impact of the observed performance loss with OpenACC enabled could be lethal in production runs (more than thousand MPI ranks). By far the worst performance loss, however, can be observed during the exchange of the horizontal boundaries between the MPI ranks. This routine only consists of a series of MPI calls that utilize non-contiguous derived data types with 3D arrays (see previous section). As we were not able to make these MPI calls CUDA aware, the loss can completely be blamed to the data-transfer bottleneck between host and device. We are aware that this issue could potentially be solved by wrapping the data array slices into separate buffers and unrolling the complex MPI calls into a series of simple MPI calls with MPI derived data types that are contiguous in memory. The complex data types however are deeply integrated into the software and any unrolling or change related to them entails a lot of effort. We therefore refrained from investing time into this approach. As MPI implementations gain capability in handling non-contiguous derived data types, we hope to see further speed improvements on the GPU. Finally it should be noted that the runtime of the OpenACC-enabled tests were not depending on the number of CPU cores used. The runtime variations between four CPU cores and one GPU versus twenty-four CPU cores and one GPU was around one percent. This shows that nearly the entire program is executed on the GPU. Brief tests comparing one node using twenty-four CPU cores and one GPU to two nodes using twelve CPU cores and one GPU on each node were conducted as well. As expected the runtime of the OpenACC-enabled tests were nearly cut in half as available GPU resources are doubled.

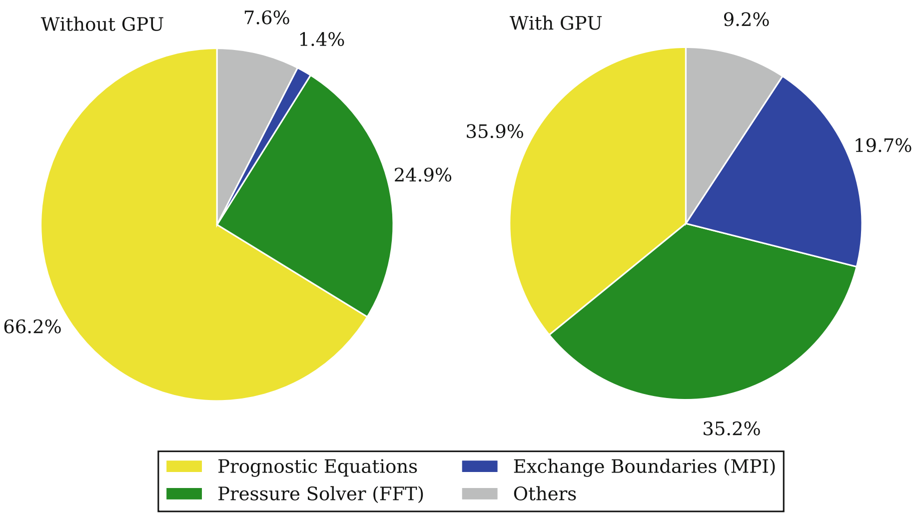

Fig. 4. Pie chart of the share the three most time consuming parts of PALM contribute to the total runtime for OpenACC-disabled tests (left) and openAcc-enabled tests (right). The share of all other routine runtimes is combined under “Others”.

5 Summary

5.1 Porting Experience

We started our porting endeavor at the GPU Hackathon in Dresden with a partly and not correctly ported version of the computational fluid dynamics model PALM. During the Hackathon, which lasted for one week, we were able to port most of the routines of PALM to the GPU. We learned to use advanced profiling tools as a guide through such a porting process and we highly recommend their use for this purpose. The problems we faced taught us an important lesson about the significance of sophisticated testing capabilities in any software project similar in size and complexity to PALM. Also maintaining several mostly redundant branches of a code base in order to port or optimize for different computer architectures significantly reduced our productivity. Due to the difficulties we faced, we were unable to finish all the porting work during the Hackathon. However, the extensive and very helpful mentoring during the entire week gave us all the necessary knowledge and tooling to finish the leftover action items within a short time frame. The practical orientation of the Hackathon was responsible for an effective and valuable first-hand knowledge transfer. A deeper understanding of the OpenACC based porting concept and the architecture of GPUs in general was our reward.

5.2 Technical Conclusions

The absence of a sophisticated unit and integration test suite was a major drawback for our porting productivity. Although we tried to compensate by preparing testing scripts and setups, we still spent much time struggling with unguided bug hunting. After the end of the Hachathon we started a discussion inside the PALM developer community whether to adapt the unit test approach. On the one hand we learned how difficult it can be to extend untested code. Not changing this situation will only result in accumulation of more technological debt over time. On the other hand the development of a test suite for PALM is a very large work package for a team that largely consists of people whose primary task is the science and PALM is the tool they apply. We are aware that this is a common conundrum in projects with large code bases in an academic context (like PALM) and the discussion on how to solve it continues to the day. So far, we refrained from investing work into a solution to this problem even though we know that it will make future improvements to the code even harder and on top, decrease the return-on-investment of our time at the Hackathon. We highly recommend the introduction of unit tests to projects that consider porting their code-base to GPUs.

Our suggestion to institutions that are aiming to provide help for code-development teams like us would be to increase funding for manpower especially dedicated to code development and training therein. We suggest more research on how to make the introduction of unit and integration testing easier for existing scientific code bases especially in an academic and/or performance critical context (e.g., [4]). Additionally, a more detailed documentation of all the available OpenACC features could improve the effectiveness of any GPU porting work (e.g., online documentation including simple examples similar to the C++ reference[2]). With such a documentation at hand our first porting attempt might have been successful and reduced the time-expensive bug hunting. Especially the latest status of compiler implementation(s) and current limitations

[2] The C++ reference is available online at http://en.cppreference.com/w/.

are a valuable information and should be provided on a central website. For us this shortcoming, however, was greatly reduced by the extensive mentoring at the Hackathon. Therefore, we would like to encourage all involved institutions to continue organizing similar events. Their potential for first-hand knowledge transfer should not be underestimated.

5.3 Performance

The completely ported code was tested regarding its performance on one GPU and showed a solid speed improvement compared to the performance on up to ten CPU cores. Even though parts of the code are showing solid speed improvements compared to up to twenty CPU cores, the MPI heavy routines consume this advantage. We would like to emphasize our strong demand for a MPI implementation that is capable of handling non-contiguous derived data types in CUDA-aware MPI calls correctly and efficiently. Without such a capability the utilization of GPUs for large production runs of PALM and probably many other similar CFD applications will not be profitable.

Acknowledgments. We would like to thank the Oak Ridge National Laboratory (US), Nvidia Corporation Inc. (US), the Portland Group Inc. (US), the standards OpenACC committee as well as the Center for Information Services and High Performance Computing (ZIH) at Technische Universität Dresden and the Forschungszentrum Jülich for organizing the OpenACC Hackathon in March 2016. We would like to thank personally Fernanda Foertter, Guido Juckeland and Dirk Pleiter for organizing the Hackathon in Dresden. Further, we express our deep gratitude to Dave Norton (Portland Group) and Alexander Grund (HZDR; Rossendorf) for their instrumental contribution as members of the mentoring team during the Hackathon. The author team consists of three PALM developers (Knoop, Gronemeier, and Knigge) and one mentor of the Hackathon (Steinbach).

References

1. TOP500 Supercomputer Site. http://www.top500.org/list/2015/11/
2. Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T., Wagener, J.L.: Fortran 95 Handbook: Complete ISO/ANSI Reference. MIT Press, Cambridge (1998)
3. Doar, M.B.: Practical Development Environments. O'Reilly Media Inc., Sebastopol (2005)
4. Feathers, M.C.: Working effectively with legacy code. In: Zannier, C., Erdogmus, H., Lindstrom, L. (eds.) XP/Agile Universe 2004. LNCS, vol. 3134, p. 217. Springer, Heidelberg (2004). http://dx.doi.org/10.1007/978-3-540-27777-4_42
5. Gronemeier, T., Inagaki, A., Gryschka, M., Kanda, M.: Large-eddy simulation of an urban canopy using a synthetic turbulence inflow generation method. JJSCE B1 **71**(4), I_43–I_48 (2015). http://dx.doi.org/10.2208/jscejhe.71.i_43
6. Hoffmann, F., Raasch, S., Noh, Y.: Entrainment of aerosols and their activation in a shallow cumulus cloud studied with a coupled LCM-LES approach. Atmos. Res. **156**, 43–57 (2015). http://dx.doi.org/10.1016/j.atmosres.2014.12.008

7. Knigge, C., Raasch, S.: Improvement and development of one- and two-dimensional discrete gust models using a large-eddy simulation model. J. Wind Eng. Ind. Aerodyn. **153**, 46–59 (2016). http://dx.doi.org/10.1016/j.jweia.2016.03.004
8. Knigge, C., Auerswald, T., Raasch, S., Bange, J.: Comparison of two methods simulating highly resolved atmospheric turbulence data for study of stall effects. Comput. Fluids **108**, 57–66 (2015). http://dx.doi.org/10.1016/j.compfluid.2014.11.005
9. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008). http://dx.doi.org/10.1007/978-3-540-68564-7_9
10. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-31476-6_7
11. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105. Curran Associates, Inc. (2012). http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
12. Letzel, M.O., Helmke, C., Ng, E., An, X., Lai, A., Raasch, S.: LES case study on pedestrian level ventilation in two neighbourhoods in Hong Kong. Meteorol. Z. **21**(6), 575–589 (2012). http://dx.doi.org/10.1127/0941-2948/2012/0356
13. Maronga, B., Gryschka, M., Heinze, R., Hoffmann, F., Kanani-Sühring, F., Keck, M., Ketelsen, K., Letzel, M.O., Sühring, M., Raasch, S.: The Parallelized Large-Eddy Simulation Model (PALM) version 4.0 for atmospheric and oceanic flows: model formulation recent developments, and future perspectives. Geosci. Model Dev. **8**(8), 2515–2551 (2015). http://dx.doi.org/10.5194/gmd-8-2515-2015
14. Maronga, B., Hartogensis, O.K., Raasch, S., Beyrich, F.: The effect of surface heterogeneity on the structure parameters of temperature and specific humidity: a large-eddy simulation case study for the LITFASS-2003 experiment. Bound. Layer Meteorol. **153**(3), 441–470 (2014). http://dx.doi.org/10.1007/s10546-014-9955-x
15. Martin, K., Hoffman, B.: Mastering CMake, 4th edn. Kitware Inc., New York (2008)
16. an Mey, D., Biersdorff, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S.S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: a unified performance measurement system for petascale applications. In: Bischof, C., Hegering, H.-G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 85–97. Springer, Heidelberg (2012). http://www.springerlink.com/content/t041605372024474/?MUD=MP
17. OpenACC-Standard.org: The OpenACC Application Programming Interface, 2.5 edn. (2015). http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf
18. Páll, S., Abraham, M.J., Kutzner, C., Hess, B., Lindahl, E.: Tackling exascale software challenges in molecular dynamics simulations with GROMACS. In: Markidis, S., Laure, E. (eds.) EASC 2014. LNCS, vol. 8759, pp. 3–27. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-15976-8_1

19. PGI: PGI CUDA Fortran Compiler. http://www.pgroup.com/resources/cudafortran.htm
20. Reid, J.: The new features of Fortran 2003. SIGPLAN Fortran Forum **26**(1), 10–33 (2007). http://dx.doi.org/10.1145/1243413.1243415
21. Stallman, R.M., McGrath, R., Smith, P.D.: GNU make: a program for directing recompilation, for version 3.81. Free Software Foundation (2004)