Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Modern GPGPU Frameworks and their Application to the Physical Core of the ASUCA Weather Prediction Model

Master Thesis

Michel Müller

michel@typhooncomputing.com

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Dr. Takashi Shimokawabe (Tokyo Institute of Technology)
Prof. Dr. Takayuki Aoki (Tokyo Institute of Technology)
Prof. Dr. Roger Wattenhofer

March 24, 2013

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

Modern GPGPU Frameworks and their Application to the Physical Core of the ASUCA Weather Prediction Model

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name
Müller	Michel

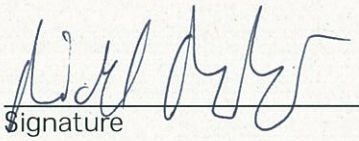
Supervising lecturer

Last name	First name
Wattenhofer	Roger

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Tokyo, 2012-10-12
Place and date


Signature

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Acknowledgements

This thesis would not have been possible without the guidance and support of several individuals to whom I will always be grateful for their contributions.

I would like to thank Prof. Dr. Takayuki Aoki for the invaluable opportunity to pursue this master thesis as his guest at the Tokyo Institute of Technology, and to be able to work together with his incredibly motivated and talented research group.

My utmost gratitude goes to Dr. Takashi Shimokawabe for his most valuable guidance throughout this master thesis. He has inspired me from the start to pursue new ideas and to not be satisfied with the ordinary.

A thousand thanks and more to Dr. Christian Feichtinger of the Friedrich-Alexander University Erlangen-Nürnbern, who helped me out many times by nudging me in the right direction and sharing his wealth of experience in the world of high performance computing.

I would also like to thank Prof. Dr. Roger Wattenhofer very much for offering me the possibility to pursue this work abroad in this still young field of GPGPU computing.

Mr. Tobias Gysi of Super Computing Systems AG, Zurich, for his steadfast encouragement and the sharing of useful experiences from one of Switzerland's leading companies in high performance computing software development.

Mr. Oliver Fuhrer of MeteoSwiss for his very interesting insight about the state of the COSMO model as well as the direction of this thesis.

Mr. Dave Norton of The Portland Group, for the sharing of his very valuable expertise when it comes to PGI compilers.

All members of the Aoki Lab group for their steady friendship, curiosity and support.

And last but not least my wife Mihoko and my parents, for their unconditional support and loving care throughout this period of hard work, as well as my Japanese in-laws for their warm welcomes and steadfast help during my stay in the breathtaking city of Tokyo.

Abstract

One of today's biggest challenges in the field of high performance computing is the efficient exploitation of the heavily increasing parallelism on socket level, especially when both CPU and GPU resources are to be applied – a challenge becoming very real for the physical processes of ASUCA. ASUCA is the Japan Meteorological Agency's next-generation weather prediction model, which is to be accelerated using GPU while keeping CPU compatibility, high CPU performance as well as an easily adaptable implementation. In this thesis we will examine the new OpenACC industry standard for hybrid GPGPU/CPU codebases, show why it is not an ideal solution for our use case and instead propose the "Hybrid Fortran" framework. This new framework will be shown to offer superior usability while enabling optimal GPU performance as well as near-optimal CPU performance through compile-time reordering of loop positions and data access patterns. A complex, bandwidth limited example module from ASUCA performs with 5 *times* speedup on Tesla M2050 versus six core Westmere Xeon while only loosing 5% of performance when executing the same codebase on CPU.

Keywords: Weather Prediction, GPGPU, Hybrid, OpenACC, CUDA, TSUBAME, Fermi, Westmere, ASUCA

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The years 2004 through 2006 mark an interesting shift in the world of computing: Single threaded performance growth has slowed down significantly since that period in time when CPU architectures arrived at their thermal design limits. Where single threaded floating point performance would grow by 64% per year before, it only gains roughly 21% per year since then [1]. This lead to a paradigm shift towards more parallelism on the socket level, a development for which Graphics Processors, out of necessity, have always been at the forefront.

In recent years there has been a push in the high performance computing (HPC) community into using Graphics Processing Units for General Purpose (GPGPU) Computing. The Tokyo Institute of Technology's TSUBAME 2.0 Supercomputer (currently ranking 14th in the TOP500 list [2]) is a prime example of this development.

The GPU instruction sets - and in extent the software development models - have not yet come to a point where they are generally applicable to programs orignally written for CPUs without significant restructurings however. Key players of the HPC industry have at the end of 2011 pushed towards a higher level programming model for GPUs through the "OpenACC" standard, and their compilers have now reached a point where this new GPGPU programming model becomes of interest for HPC software development.

1.1 Structure of this Thesis

In this thesis the applicability of GPGPU computing to the physical core of Japan's next generation weather prediction model "ASUCA" will be examined in light of these recent developments in GPGPU programming models (cha. 2). A suitable high performance toolset based on these examinations will be developed for this usecase with possible applications in other areas of scientific computing (cha. 3, cha. 4). The usability and performance of this framework will be verified (cha. 5), before pointing out the achievements and future work of this thesis (cha. 6).

The following chapter will describe the ASUCA weather prediction model (sec. 1.2), state the motivation and goals for this thesis (sec. 1.3), introduce the key technologies and terms used throughout (sec. 1.4) and point to related work (sec. 1.5).

1.2 ASUCA Weather Prediction Model

The Japan Meteorological Agency (JMA) currently operates a nonhydrostatic [1] regional weather prediction model called "JMANHM". This model has been used since the 1990s. The rapid increase of parallel architectures on the socket level, as mentioned in the preface, have lead to the motivation to renovate the weather model in order to make best use of these new levels of parallelism [4, p. 3].

The new nonhydrostatic model, to be called "ASUCA"[2], is being developed with the following objectives [5, p. 1],[4, p. 4]:

1. Higher accuracy.
2. Higher efficiency.
   (a) Less data communication.
   (b) Suitable to the current computer architecture.

ASUCA uses a three-dimensional, horizontally and vertically staggered grid [3], consisting of the two relatively widely spaced horicontal dimensions I and J and the densely spaced vertical dimension K. The equations are discretized using the finite volume method (FVM). [7, p. 2]

Fig. 1.1 gives an overview over the program flow in the ASUCA model. Physical and dynamical processes[4] are executed for each timestep. Each long timestep also contains a short time integration loop for processes that require a higher time resolution[5]. These short timestep processes also employ a second-order Runge-Kutta scheme as depicted in the figure. [7, p. 2]

[1] Nonhydrostatic as in the opposite of models using the hydrostatic approximation, i.e. the entire vertical momentum equation is being kept in the model [3, p. 138].

[2] **A**SUCA is a **S**ystem based on a **U**nified **C**oncept for **A**tmosphere.

[3] The type of grid used for ASUCA is called "Arakawa C", also being used by the European Consortium for Small Scale Modeling (COSMO) for the weather prediciton model of multiple European nations [6, p. 4].

[4] "Physical" processes can be understood as in they produce input parameters for the dynamical processes based on the last timestep's dynamics results, boundary conditions and constant factors.

[5] Horizontal sound wave and gravity wave propagation.

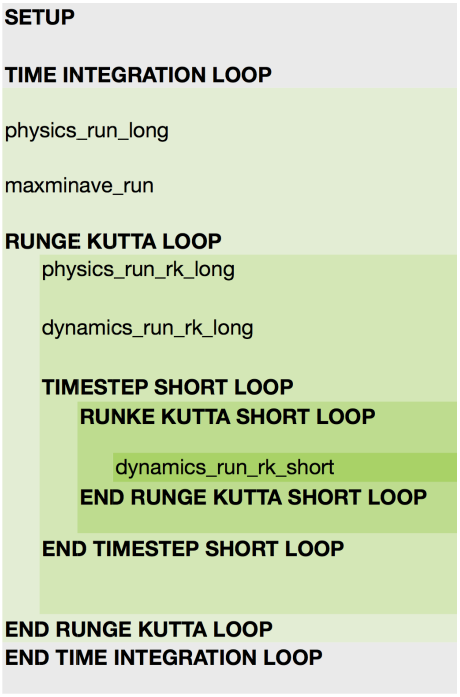


Figure 1.1: Overview ASUCA Model Program Flow.

1.3 Motivation and Goal of Thesis

The dynamical core (containing the dynamical processes depicted in fig. 1.1) was already extended for the GPU using CUDA C [6] by Shimokawabe et al. [8, p. 2]

Consequently, the next task will be the development of algorithms for the physical processes suitable for GPGPU computing. The main **motivation** for this development is to

1. eliminate host-to-device communication during the entire time integration loop.
2. gain speedups in execution time.

Being able to run this process faster on the same amount of computing nodes by employing GPGPU technology would lead to cost and energy savings and possibly an increased grid resolution, giving the model a higher accuracy.

The following **characteristics** can be observed about the dynamical in comparison to the physical core:

1. The dynamical core accounts for a higher fraction of the runtime.

[6] See sec. 1.4.2.

2. The dynamical core employs less lines of code.

3. The dynamical core tends to be more "stable" over time, i.e. the dynamical processes are not under constant development while this is the case for various physical modules.

For these reasons the dynamical processes are a prime candidate for deep performance optimizations, while porting the physical processes brings different **challenges**:

- How can the physical processes be ported while keeping a familiar development environment for the JMA researchers, i.e. Fortran 90?
- How can the physical processes be ported with the least amount of code changes compared to the current code?
- How can the physical processes be ported to the GPU while still keeping the code executable on CPU?
- How can all of the above be achieved while still enabling a high GPU performance?
- How can the CPU performance of the hybridized code be kept at the same level as with code that has been specifically optimized for the CPU? This would enable a heterogenous execution of the code on systems with both highly capable CPU and GPU resources such as TSUBAME 2.0. Specifically for the case of JMA's ASUCA this is a very important aspect, since it is planned to gradually move towards GPU execution while still maintaining CPU compatibility.

The **main goal for this thesis** is to find and verify a strategy with which the questions above can be answered. In this thesis single socket parallelism will be discussed. Multi socket / multi node parallelism will be solved through MPI communication, which will not be covered here.

1.4 GPGPU Computing on the NVIDIA Fermi Architecture

The Tokyo Institute of Technology's TSUBAME 2.0 Supercomputer has been used as a development and test platform. This system currently uses NVIDIA Fermi[7] based GPUs, of which three are added to each computing node, as well as "Westmere" generation Intel Xeon CPUs. More details about the hardware specifications and models can be found in sec. 2.4.

When it comes to GPGPU software library and tooling support, NVIDIA's Fermi GPU architecture is currently regarded as the most mature. Therefore, this thesis mainly concentrates on the available software technologies for NVIDIA GPUs on TSUBAME 2.0. However, it must be noted that vendor independant solutions would naturally be of an advantage - something that will be kept in mind for later conclusions.

1.4.1 NVIDIA Fermi

The main difference between CPU and GPU computing lies in the GPU's massively parallel nature. NVIDIA Fermi employs 16 so called "Streaming Multiprocessors" (SMs) with 32 computing cores each, resulting in 512 threads running in parallel[8]. Each core, naturally, has a much lower complexity than any x86 core, which is reflected in the reduced scheduler, cache and register resources available to it. Figure 1.2 shows the ressources of one SM.

The following hardware characteristics are very important when it comes to discussing the programming model:

1. GPUs employ a scheduling scheme similar to SIMD[9]: Each instruction is executed by a number of threads multiple of 32 (up to 512 in Fermi's case) for a whole block of data. However, compared to SIMD, these threads can be scheduled more freely, e.g. allowing branching[10]. The memory architecture is optimized for coalesced loads / stores of data blocks with a multiple of 32×4 byte size. A 32 thread unit, reflected in the SM architecture, is thus very important for Fermi GPGPU computing, and is being called "Warp".

[7] The Fermi architecture has been introduced in 2010 and has been highly popular in the HPC world, in part because of high double precision performance and support for ECC Memory. Four systems in the current top 20 use Fermi cards [9].

[8] The number of possible threads is half of that number for double precision execution in case of Tesla branded Fermi GPUs, significantly lower even for the GeForce brands.

[9] **S**ingle **I**nstruction **M**ultiple **D**ata, the classic scheme for vector processors.

[10] Branches will however degrade the performance of GPU code, since only one code path of the branch will be executed in parallel - threads that are not in that path will simply execute "nop" instructions during that time.

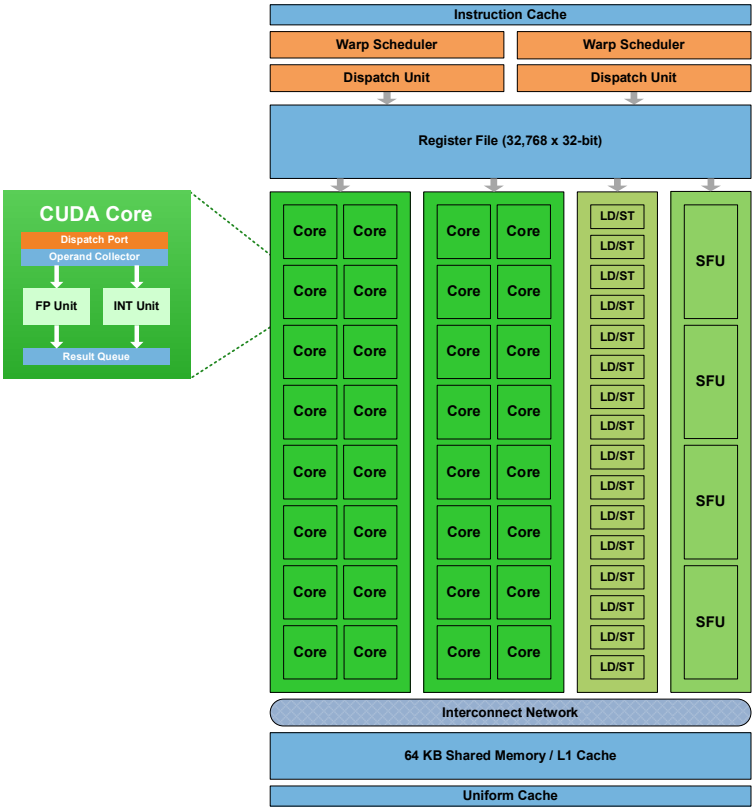


Figure 1.2: One SM contains 32 computational cores, 16 load/store units, 4 "Special Function" units, 32'768 4-byte-registers and 64kB L1 cache / Shared memory [10, p. 8].

2. The cost of thread context switching is orders of magnitude lower than on x86, to a point where it becomes negligible for most use cases.

1.4.2 CUDA Programming Model

"Compute Unified Device Architecture" (CUDA) is an integrated hardware and software technology developed by NVIDIA for writing general purpose programs on their GPUs. Since thread switching is very cheap, as discussed in sec. 1.4.1, CUDA programs are usually written in a way, such that for every data point in the parallel domain one thread (or a series of threads, executed sequentially) is created. Threads are bundled into one-, two- or three dimensional thread blocks, such that each block is executed on one SM. One of the first tasks when adapting a program for CUDA, is usually to map the involved data structures onto threadblocks, such that memory reads can be optimized into coalesced fetches of at least 32×4 bytes. This is usually accomplished using simple one dimensional or multi dimensional arrays.

CUDA Programs consist of the following parts:

Host code is code that will be executed on the host system CPU(s). It is being compiled through third party compilers, with NVIDIA providing a runtime library for interactions with the device[11], such as device memory allocations, device data transfer and kernel calls.

Kernel functions define the program of one GPU thread. Memory access is usually specified by using an index that encodes thread ID and block ID of one thread. In conventional programming terms, CUDA kernel definitions best correspond to parallelizable `for` loops.

Device functions define subroutines that can be called from a Kernel function. These subroutines need to be inlineable by the CUDA compiler.

Kernels (including the inlined device functions) will be compiled to an intermediate device code representation called "PTX" through the CUDA compiler.

1.4.3 OpenACC Programming Model

OpenACC is a relatively new standard introduced in 2011, with the intention to bring the power of directive based parallelization, such as OpenMP, to GPGPU computing [11]. The standard has been developed by CAPS, Cray and The Portland Group in association with NVIDIA [12]. Before the OpenACC standardization, the aforementioned companies each developed their own proprietary

[11] In GPGPU Computing, the word "device" is generally used for the GPU.

GPU parallelization directives for their compiler solutions. Current OpenACC compilers only support NVIDIA CUDA implementations of OpenACC user code.

The main functionality of OpenACC is the abstraction of GPU kernels through directives added to `for` loops (`do` loops in case of Fortran code). Adding such directives delegates the responsibility of implementing a GPU kernel to the OpenACC compiler. As an added benefit, the codebase becomes hybrid, i.e. executable on both CPU and GPU. Device data handling is solved through additional directives.

1.5 Related Work

Naoya Maruyama, Tatsuo Nomura et al. *Physis: An Implicitely Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers*: This paper describes a compiler-based programming framework that automatically translates user-written structured grid code into a parallel implementation code for multi-GPUs, using a DSL embedded in C language code. [13]

Tobias Gysi (Super Computing Systems AG) *A Stencil Library for the New Dynamic Core of COSMO*: This talk examines the stencil DSL library that has been developed for the European Consortium for Small Scale Modeling (COSMO), the maintaining gremium of the climate model for seven European countries. This stencil DSL library heavily relying on C++ templates has been applied to the Dynamical Core of the COSMO model and enables hybrid execution on CPU and GPU. [14]

Takashi Shimokawabe, Takayuki Aoki et al. *An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code*: This paper examines the portation of the dynamical core of the ASUCA weather prediction model to GPGPUs on TSUBAME 1.2 (Tesla S1070, AMD Opteron). Single GPU vs. single core CPU execution achieved a speedup of 26.3 in double precision. The multi-GPU implementation achieved an overall single precision performance of 15.0 TFlops on only 528 GPUs - a remarkable achievement compared to the 50 TFlops record at that time on more than 18'000 Jaguar nodes with nearly 150'000 CPU cores. To achieve this performance, optimization methods for overlapping communication and computation, as well as asynchronous computation of variables were explored. [7]

Takashi Shimokawabe, Takayuki Aoki et al. *145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction*: This paper examines the results of the ASUCA dynamical core portation, first introduced in [7], applied to the TSUBAME 2.0 architecture. Single precision performance improved by 10% while double precision performance improved by 65% on single Tesla M2050 compared to the Tesla S1070 of TSUBAME 1.0. On multi-GPUs, 145 single precision TFlops on 3990 GPUs and 76.1 double precision TFlops on 3936 GPUs have been achieved. As a future real world application, a typhoon over Japan has been simulated on a very high mesh resolution of 500m. [8]

John Michalakes, Manish Vachharajani *GPU Acceleration of Numerical Weather Prediction*: In this paper Michalakes and Vachharajani describe the GPU

portation of the so-called WRF[12] Single Moment 5-tracer (WSM5) module using NVIDIA CUDA C. This lead to a 7.7× performance increase for that module and 1.23× performance increase for the overall WRF model. [15]

John C. Linford, John Michalakes et al. *Multi-core Acceleration of Chemical Kinetics for Simulation and Prediction*: In this paper the implementation of a computationally expensive chemical kinetics kernel was compared on three multicore platforms: NVIDIA Tesla C1060 GPUs, Cell Broadband Engine (CBEA) and Intel Quad-Core Xeon. The CBEA achieved the highest speedup of 41.1× compared to the serial implementation in single precision. The GPU architecture was hampered because of low availability of on-chip memory and achieved a speedup of only 8.5× compared to the serial implementation in single precision. [16]

Stefan Kronig, Michel Müller *Feasibility and Performance of Weather Computations using GPGPU Programming*: This semester thesis, conducted in collaboration with Tobias Gysi of Super Computing Systems AG, contains a preliminary examination for a GPU port of the Dynamical Core of the COSMO Weather model, using a sample algorithm. [17]

[12] "The Weather Research and Forecast" model, the world's most widely used weather prediction model.

Chapter 2

Evaluation of Existing Frameworks

In order to determine a viable GPU implementation strategy for the physical core of ASUCA it has been necessary to analyse the available industry standards and whether they are already suited for JMA's needs as is (see also sec. 1.3).

Sec. 2.1 lists the evaluation criteria. In sec. 2.2 the investigated GPGPU software frameworks are being introduced. The CPU compilers used for comparison are listed in sec. 2.3. Sec. 2.4 introduces a performance model for the test hardware. Sec. 2.5 lists the test cases used for the evaluation. Sec. 2.6 compares the frameworks in terms of usability while sec. 2.7 compares the frameworks by performance. Sec. 2.8 draws a preliminary conclusion about the viability and performance of the tested frameworks for the purpose of an ASUCA physical process implementation.

2.1 Criteria for an ASUCA Physical Process GPU Portation

The following criteria are important to consider when choosing a framework for the GPGPU portation of the ASUCA physical process codebase:

1. The framework should offer a GPGPU interface for a Fortran 90 codebase.
2. The framework should make a hybrid codebase possible, i.e. the user code should be compilable to both GPU and CPU. This allows the verification of results on the CPU before porting, testing and debugging the GPU implementation.
3. Required code changes to the existing codebase should be as small as possible in order to lower the portation cost.

4. It should offer viable execution time performance, i.e. it should be as close to fully platform optimized code as possible. If possible, the performance should be viable both on CPU and GPU. This would give two additional advantages:
   (a) The modules could be ported one-by-one and integrated into the current production environment, allowing for a smoother transition and easier validation.
   (b) Heterogenous execution would become an option at a later point, such that the code would become portable to a wide range of clusters / supercomputers.
5. It should be as platform and software vendor agnostic as possible, e.g. Open Source frameworks and industry standards are preferred.

2.2 Investigated GPU Frameworks and Compilers

The frameworks presented in this section have been evaluated with respect to the criteria stated in sec. 2.1.

2.2.1 OpenACC

For an introduction to CUDA please refer to sec. 1.4.3. For the implementation criterias stated in sec. 2.1, OpenACC is the solution the most promoted by the industry right now. Though it must be noted that currently, no compiler completely supports the OpenACC 1.0 standard. The support has been improved over the course of this thesis, however. It is also promising, that MeteoSwiss, responsible for the Swiss weather prediction model in association with the European Consortium for Small Scale Modelling (COSMO), is also involved in an OpenACC portation project for their physical processes. Lst. 5.2 in cha. 5 shows a sample OpenACC implementation.

For this evaluation the following OpenACC capable compilers have been tested:

1. PGI Workstation (`pgcc` and `pgf90`), version 12.4
2. CAPS HMPP, version 3.1.0

Both compilers offer C as well as Fortran frontends for OpenACC. PGI compilers have been executed with the following flags when used for OpenACC compilation, if not stated otherwise:

-acc Enables OpenACC compilation.

-ta=nvidia,cc20 Specifies the target platform, in this case NVIDIA GPUs with computing capabilities version 2.0.

-O4 The highest optimization level for PGI compilers.

The HMPP framework is essentially a preprocessor that creates CUDA kernels for parsed OpenACC directives and passes the host code to an underlying compiler of the user's choice. For fair comparison, `pgcc` was chosen as the underlying compiler using the same settings as presented above.

2.2.2 CUDA C

CUDA C is NVIDIA's proprietary C language extension. For an introduction to CUDA, please refer to sec. 1.4.2. As an implementation strategy for the ASUCA

physical process it is not viable, since it violates the criterias 1 and 3 stated in sec. 2.1. However, it was used as a performance validation, since CUDA C implementations can generally be optimized the closest to GPUs from NVIDIA, other than writing PTX device instructions.

For this thesis we have tested NVIDIA's CUDA framework in version 4.1.

CUDA C requires a third party C compiler for host code compilation. For the tests discussed in this chapter, `pgcc` has been used with the following compiler flags:

-Mcuda=cc20 Enables CUDA compilation for 2.0 CUDA device architecture.

-O4 The highest optimization level for PGI compilers.

2.2.3 PGI CUDA Fortran

PGI CUDA Fortran is a Fortran 90 wrapper framework developed and maintained by The Portland Group. It creates CUDA C code from Fortran kernel and device function definitions. The CUDA C version is then compiled using a NVIDIA CUDA compiler. In that regard, CUDA Fortran can be expected to perform on the same level as CUDA C for a given feature set - however the implemented features tend to lag behind those released by NVIDIA, such as support for `printf` in kernels as well as device code debugging.

PGI CUDA Fortran, like CUDA C, would require deep restructuring of the ASUCA Physical Process codebase, however it has been used in order to compare the usability and performance to OpenACC implementations.

Again, PGI Workstation version 12.4 has been used for testing the CUDA Fortran framework. The following compiler flags have been used for `pgf90` when compiling for CUDA Fortran:

-Mcuda=cc20 Enables CUDA Fortran compilation for 2.0 CUDA device architecture.

-O4 The highest optimization level for PGI compilers.

-Minline=levels:5,reshape Enables inlining for routines defined in the same module as their caller. The `reshape` option enables the compiler to perform array reshaping when passing them to the callee.

2.3 Investigated CPU Compilers

The version 11.1 of Intel C (`icc`) and Intel Fortran (`ifort`) compilers have been used for reference purposes throughout this thesis.

Various experiences have shown that this compiler offers the most consistant optimizations (without the need to adjust and experiment with various tuning screws) when used on the vendor provided hardware. For this reason the Intel compilers appear to be a reasonable choice for the determination of the CPU performance reference on Intel hardware. `icc` and `ifort` have been used with the following compiler flags, if not stated otherwise:

-fast Enables the compiler to use the optimization settings leading to the highest performance based on its internal heuristics. This includes vectorization and inlining of routines as well as enabling the `-O3` arithmetic optimization level.

2.4 Hardware Model

Since the scope of this thesis has been to implement and analyse single GPU as well as single CPU performance, only one CPU socket and one GPU has been used for the performance analyses in sec 2.7 as well as chapter 5. This section gives an overview over the hardware performance model used for performance estimates throughout this thesis.

2.4.1 Overview FP Performance and Memory Bandwidth

For all tests, single Tsubame 2.0 nodes have been used in the following configuration [18, p. 4]:

1. One Dual socket Intel Xeon X5670 CPU with 6 cores per socket. One Xeon X5670 socket offers the following performance metrics [1]:
   (a) 1 Core Sustained Memory Bandwidth[2]: `9.8 GB/s`
   (b) 1 Socket Sustained Memory Bandwidth: `20.5 GB/s`
   (c) 1 Core Single Precision Peak Performance: `23.4 GFLOPS`
   (d) 1 Core Double Precision Peak Performance: `11.7 GFLOPS`
   (e) 1 Socket (6 Core) Single Precision Peak Performance: `140.1 GFLOPS`
   (f) 1 Socket (6 Core) Double Precision Peak Performance: `70.1 GFLOPS`
2. Three NVIDIA Tesla M2050 GPUs. One Tesla M2050 offers the following performance metrics [22]:
   (a) Sustained Memory Bandwidth[3]: `108.4 GB/s`
   (b) Single Precision Peak Performance: `1030 GFLOPS`
   (c) Double Precision Peak Performance: `515 GFLOPS`

It is important to note that the peak computational performance usually does not directly translate into sustained performance, however it is reasonable to take the relative ratios into consideration: One might expect a Tesla M2050 to enable the following speedups:

11.1 times faster for memory bandwidth bounded algorithms compared to single core CPU execution.

[1] Theoretical CPU performance is based on the assessment for X5650 in [19, p. 4], with the computational performance numbers scaled by 2.93 GHz/2.66 GHz in order to adjust for the Xeon X5670's higher frequency [20].

[2] Single core stream memory bandwidth is based on [21, p. 7].

[3] The Tesla M2050's sustained Memory bandwidth has been evaluated using the bandwidth test program provided in the CUDA SDK.

5.3 times faster for memory bandwidth bounded algorithms compared to single socket / six core CPU execution.

44 times faster for computationally bounded algorithms compared to single core CPU execution.

7.4 times faster for computationally bounded algorithms compared to single socket / six core CPU execution.

However there are many architectural differences that have not been taken into consideration for this line of thought. For this reason it is necessary to do tests with the actual programs in order to make reasonably well founded performance predictions.

2.4.2 System Balance

For further discussions the notion of "System Balance" will be defined as follows:

System Balance is the number of floating point instructions per floating point fetch or store, that leads to a maximum floating point throughput while utilizing the maximum sustained memory bandwidth.

That is, system balance is defined using the following formula[4]:

$$\text{System Balance} = \frac{\text{Peak FLOPS}}{\frac{\text{Sustained Memory Throughput}}{\text{Size of Floating Point Variables}}} \tag{2.1}$$

One issue to keep in mind when taking into account peak `GFLOPS` numbers: Vendors calculate these by doubling the effective throughput of floating point operations per cycle, assuming a program would only generate `multiply-add` instructions (which can be scheduled in one cycle by modern high performance architectures). Assuming that effectively `10%` of the instructions are `multiply-add`, one gets a more reasonable prediction of fp performance using the following calculation:

$$\text{Peak FLOPS} = \text{Peak FLOPS}_{\text{vendor}} \cdot (0.1 \cdot 1 + 0.9 \cdot \frac{1}{2}) \approx \text{Peak FLOPS}_{\text{vendor}} \cdot 0.55 \tag{2.2}$$

Considering the metrics introduced in sec. 2.4.1, the system balances become

[4]This notion of System Balance is based on the Roofline model by Patterson [23]. Patterson uses the notion of "operations per byte" as an indicator for boundedness. The test hardware used for this thesis offers double precision peak performance exactly half of single precision peak performance, both for CPU and GPU - therefore the memory bandwidth is being adjusted by the length of the floating point values such that the balance estimate can be used both for single and double precision execution.

5.3 for single core CPU execution (using the following calculation:

$$\frac{23.4\,\text{GFLOPS} \cdot 0.55}{\frac{9.8\,\text{GB/s}}{4\,\text{Bytes/FLOP}}} \tag{2.3}$$

).

15.0 for six core CPU execution (using the following calculation:

$$\frac{140.1\,\text{GFLOPS} \cdot 0.55}{\frac{20.5\,\text{GB/s}}{4\,\text{Bytes/FLOP}}} \tag{2.4}$$

).

20.9 for GPU execution (using the following calculation:

$$\frac{1030\,\text{GFLOPS} \cdot 0.55}{\frac{108.4\,\text{GB/s}}{4\,\text{Bytes/FLOP}}} \tag{2.5}$$

).

These assumptions are valid for both double and single precision since both the computational power and the fetched data per floating point value scale by a factor of two. One must note however, that the above calculations are rather simplistic. One would have to account for many details specific to the architectures, such as operation scheduling, in order to get a more precise performance model - especially in the CPU case, because of its rather intricate architecture, this would be an undertaking not feasible within the scope of this evaluation.

For qualitative measures it is enough to say that algorithms whose fp instruction to fetch/store ratio is significantly below System Balance, are expected to be bounded by memory bandwidth while algorithms above System Balance are expected to be computationally bounded - the most important distinction to make when trying to optimize for performance.

2.5 Test Cases Used for Comparison

As discussed in sec. 2.2.1, OpenACC appears to be the most obvious choice for a portation of the ASUCA physical process. Three modules have been used for evaluating performance and usability of OpenACC:

1. 3D Diffusion
2. (Single Stage) Particle Push
3. ASUCA Shortwave Radiation

CUDA C is generally considered to perform the closest to the hardware limit on NVIDIA GPUs since its hardware vendor maintained compiler usually receives new features first. For this reason, CUDA C implementations of the algorithms for **single stage particle push** as well as **3D diffusion** have been used in order to determine a baseline performance benchmark for the OpenACC frameworks to compare to.

Since it is easier to compare two C implementations instead of C vs. Fortran implementations, the OpenACC implementations for these two algorithms have been made using the OpenACC C language frontends as well. It is reasonable to assume the results of that comparison in C to be an indicator for Fortran GPU implementations as well, since at least PGI CUDA Fortran and PGI OpenACC use an intermediate CUDA C code representation in order to make use of NVIDIA's `nvcc` compiler.

The **ASUCA shortwave radiation** module has been used as a benchmark for a real world application, which lead to insights not only for performance but also for the usability of the tested frameworks. The frameworks have been tested in their Fortran incarnations (as described in sec. 2.2.1 and sec 2.2.3) for this module since the final goal is a GPGPU implementation of the ASUCA Physical Process using Fortran.

The following subsections offer more details on three test modules introduced above.

2.5.1 3D Diffusion

Lst. 2.2 shows the 3D diffusion routine used as the first performance benchmark.

This routine is executed with the following specifics:

- $256 \times 256 \times 256$ data points are being calculated for each timestep. This number has been chosen to be divisible by 32 (to make it best suitable for GPU warps as introduced in sec. 1.4) and to result in a long enough execution time such that measuring inaccuracies are not influencing the conclusion.
- The input and output pointers are being swapped after every timestep. No data is being copied in between timesteps.
- 6553 timesteps are being calculated.
- Single as well as double precision arithmetics have been evaluated.

A few notes about this algorithm:

- The algorithm is expected to be bounded by memory bandwidth. Disregarding the boundary computations (which only account for $\frac{3}{3+258} \approx 1\%$ of calcutions in this configuration) the algorithm leads to seven fetches and one write per 13 floating point computations, giving a computation-to-fetch ratio of 1.85, which is clearly in the memory bandwidth bounded region for all metrics displayed in sec. 2.4.

- The computation results have been validated against an analytical solution. In case of double precision numerical calculation, the result had a root mean square error of $1.1 \cdot 10^{-6}$, which is expected to be the error of the numerical approximation used here. For single precision calculation, the error was in the order of 10^{-5}, with slight differences between CPU and GPU execution. (The GPU interestingly calculated with a higher accuracy, $1 \cdot 10^{-5}$ vs. $2 \cdot 10^{-5}$ root mean square error).

```
1
2  /* ======================
3     SETUP OF ONE XY-PLANE
4     ======================
5    x marks the origin of the coordinate system (beginning of halo)
6        ------------------------------------
7  ______x|______haly_____________|pad+halx|
8  pad+hal|                       |        |
9         |                       |        |
10        |        dimX * dimY    |        |
11        |                       |        |
12        |_______________________|________|
13 _______|______haly_____________|
14
15 */
16
17 /*
18 #define ADDR_FROM_XYZ(x, y, z): calculate coordinates in the data
19 structure shown above, using X-,Y-,Z-dimensions, haloes and
       paddings
20 */
21
22 void  diffusion3d_timestep (
23    float    *f,    /* dependent variable */
24    float    *fn,   /* updated dependent variable */
25    int      nx,    /* x-dimensional grid size */
26    int      ny,    /* y-dimensional grid size */
27    int      nz,    /* z-dimensional grid size */
28    float    kappa, /* diffusion coefficient */
29    float    dt,    /* time step interval */
30    float    dx,    /* grid spacing in the x-direction */
31    float    dy,    /* grid spacing in the y-direction */
32    float    dz     /* grid spacing in the z-direction */
33 ) {
34
35     int    j,  jx,  jy,  jz,  NX = nx + 2,  NY = ny + 2;
```

```
36      float ce = kappa*dt/(dx*dx), cw = kappa*dt/(dx*dx),
37      cn = kappa*dt/(dy*dy), cs = kappa*dt/(dy*dy),
38      ct = kappa*dt/(dz*dz), cb = kappa*dt/(dz*dz),
39      cc = 1.0 - (ce + cw + cn + cs + ct + cb);
40
41      for(jz = 1; jz < nz + 1; jz++) {
42        for(jy = 1; jy < ny + 1; jy++) {
43          for(jx = 1; jx < nx + 1; jx++) {
44            j = ADDR_FROM_XYZ(jx, jy, jz);
45            fn[j] = cc*f[j]
46            + ce*f[j+1] + cw*f[j-1]
47            + cn*f[j+NX] + cs*f[j-NX]
48            + ct*f[j+(NX * NY)] + cb*f[j-(NX * NY)];
49          }
50        }
51      }
52
53      // Wall Boundary Condition
54      for(jz = 1; jz < nz + 1; jz++) {
55        for(jy = 1; jy < ny + 1; jy++) {
56          j = (NX * NY)*jz + NX*jy + 0;
57          fn[j] = fn[j+1];
58          j = (NX * NY)*jz + NX*jy + nx + 1;
59          fn[j] = fn[j-1];
60        }
61      }
62
63      for(jz = 1; jz < nz + 1; jz++) {
64        for(jx = 1; jx < nx + 1; jx++) {
65          j = (NX * NY)*jz + NX*0 + jx;
66          fn[j] = fn[j+NX];
67          j = (NX * NY)*jz + NX*(ny + 1) + jx;
68          fn[j] = fn[j-NX];
69        }
70      }
71
72      for(jy = 1; jy < ny + 1; jy++) {
73        for(jx = 1; jx < nx + 1; jx++) {
74          j = (NX * NY)*0 + NX*jy + jx;
75          fn[j] = fn[j+(NX * NY)];
76          j = (NX * NY)*(nz + 1) + NX*jy + jx;
77          fn[j] = fn[j-(NX * NY)];
78        }
79      }
80  }
```

Listing 2.1: 3D Diffusion Example Program.

2.5.2 Particle Push

Lst. 2.2 shows the very simple particle push timestep routine used as the second performance benchmark.

This routine is executed with the following specifics:

- 6'553'600 ($256 \cdot 256 \cdot 100$) particles are being calculated for each timestep. This number has been chosen to be divisible by 32 (in order to make it best suitable for GPU Warps as introduced in sec. 1.4) and to result in a long enough execution time such that measuring inaccuracies can be neglected.
- The input and output pointers are being swapped after every timestep. No data is being copied in between timesteps.
- 800 timesteps are being calculated.
- Single precision arithmetics have been used.

Some notes about this algorithm:

- Execution of the `US` and `VS` functions results in two `sincos` calculations (sine and cosine calculated at the same time), assuming the compiler is able to optimize for this. Otherwise, two sine and two cosine operations are being executed for each data point. Since compilers will be shown to have different characteristics with respect to that arithmetical optimization, two code versions have been implemented and tested. Version 1 uses the code as shown in lst. 2.2 while version 2 is hand-optimized to use `sincos` operations instead of `sine` and `cosine`.
- The algorithm is expected to be computationally bounded, using the following assumptions:

  1. A `sincos` operation takes ≈ 30 cycles to complete with single precision on modern CPU architectures[5] and 60 cycles on GPU architectures[6] without use of the Special Function Units.
  2. There are 13 additional floating point operations present per particle and timestep.
  3. Two memory loads as well as two store operations are needed per particle and timestep.

  This yields an operation-to-fetch ratio of $\frac{73}{4} \approx 18.25$ for the CPU and $\frac{133}{4} \approx 33.25$ for the GPU - which is well above the system balance calculated in sec. 2.4.2, thus it is safe to assume that this algorithm is computationally bounded.
- Since there is no analytical result available, the computation results have been validated against a reference single threaded CPU implementation, yielding root mean square deviations in order of 10^{-6} for all results.

[5] Assumption based on [24]

[6] Assumption based on the fact that the GPU ALUs are much simpler in their architecture compared to modern x86 ALUs, thus needing more cycles per operation for complex operations.

```
1
2 #define US(x, y, t) ( - 2.0 * cos(M_PI*(t)/TAU) *
3         sin(M_PI*x) * sin(M_PI*x) * cos(M_PI*y) * sin(M_PI*y)
4         )
5 #define VS(x, y, t) ( 2.0 * cos(M_PI*(t)/TAU) *
6         cos(M_PI*x) * sin(M_PI*x) * sin(M_PI*y) * sin(M_PI*y)
7         )
8
9 void  particle_push_timestep (
10    int       np,      /* number of the particles */
11    float     *x,      /* x-coordinate of the particles */
12    float     *y,      /* y-coordinate of the particles */
13    float     *x_out,  /* updated x-coordinate of the particles */
14    float     *y_out,  /* updated y-coordinate of the particles */
15    float     time,    /* time */
16    float     dt       /* time step interval */
17 ) {
18   int      j;
19   float    xt,  yt;
20
21   for(j = 0; j < np; j++) {
22
23       xt = US(x[j], y[j], time);
24       yt = VS(x[j], y[j], time);
25       x_out[j] = x[j] + xt*dt;
26       y_out[j] = y[j] + yt*dt;
27
28   }
29 }
```

Listing 2.2: Particle Push Example Program.

2.5.3 ASUCA Shortwave Radiation

This test case has been used as a real world application (as opposed to the two rather "academic" tests presented in sections 2.5.2 and 2.5.1). Implementing this module for the GPU has lead to insights for both performance and usability when applying GPGPU framework to larger codebases.

Diverting Goals for CPU and GPU

Fig. 2.1 depicts a reduced call graph showing only the modules concerning Shortwave Radiation and its parent callers as well as the placement of the loops. In the original code version, the loops over the `I` and `J` domains are placed in outside position, within the physical process interface module called by the main program loop. The data is then passed `K`-column by `K`-column to multiple routines, each starting a call tree. This is well suited for multicore CPU execution for two reasons:

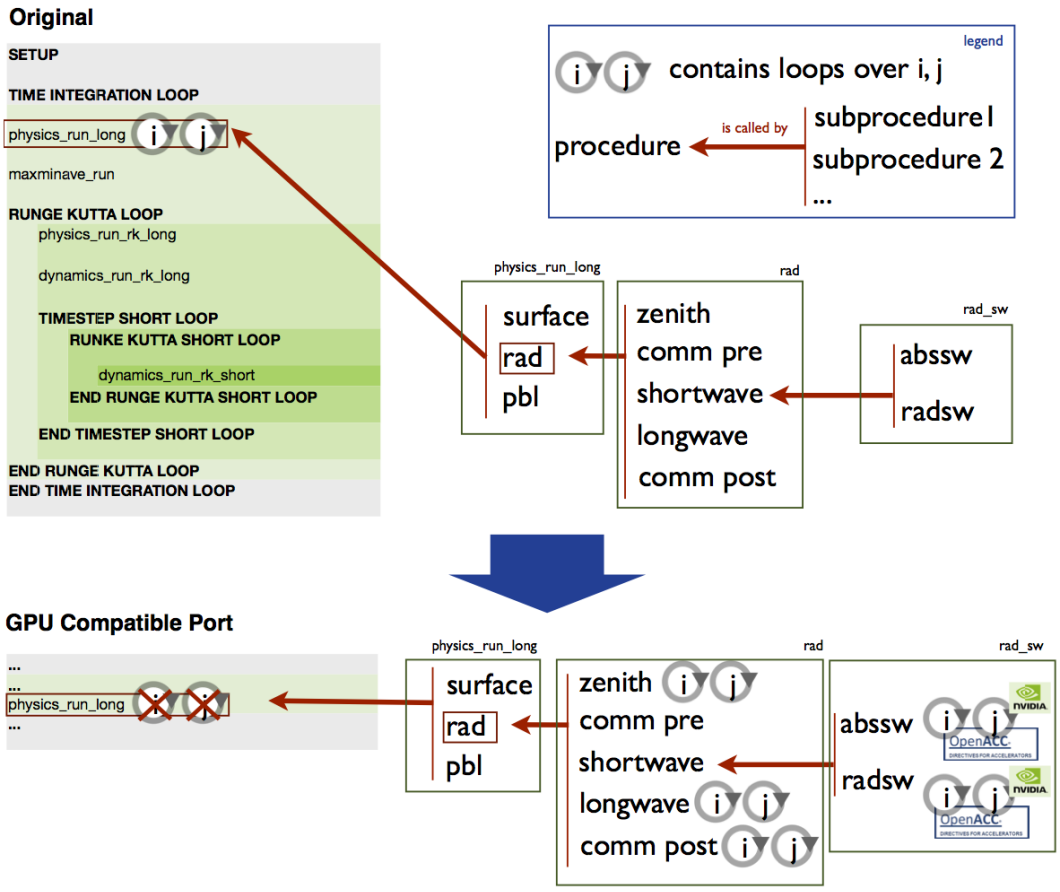


Figure 2.1: Overview ASUCA Physical Process and Shortwave Radiation.

1. For parallel execution on CPU, "work packages" with long execution times (in the order of 10^{-3} s or above) are preferred. The computation of one `K`-column for all physical modules is fullfilling this requirement.
2. In case of data sharing between modules, the chance for cache hits is higher if only `K`-columns are being shared instead of an entire `IJK`-grid of data.

For GPUs however, this program structure is ill suited for the following reasons:

1. The number of hardware registers per kernel thread (the equivalent of a an `IJ` loop run in this case) is very limited (a maximum of 63 registers are available). Having code with too much complexity inside one kernel will lead to the swapping of register data to the global memory, resulting in performance degradation in case of memory bandwidth limited programs.
2. Kernels should ideally always operate over the same code branches. Having branches inside kernels leads to performance degradation as well, since the CUDA cores of one Warp always need to either operate over the same branch or do `nop` for the diverted cycles.
3. NVIDIA GPU architectures up until Kepler with CUDA version 5.0 (in development at the time of this thesis) do not allow native context switching for calling subroutines. Subroutine calls within kernels need to be resolved by the compiler through inlining, a process that has many limitations. As a rule of thumb, it is best to keep kernel subroutines in the same modules (source files) as the kernel and to keep only one level of subroutines below the kernel.

Therefore, in order to create GPGPU implementations, the loops over the `IJ` domains need to be implemented at a deeper call level, as shown in fig. 2.1.

Execution Characteristics

This module offers the following execution characteristics:

- The `IJ` dimensions are set to $256 \cdot 256$.
- The grid size in `K` direction is 63.
- The module is executed over one timestep.
- Double precision arithmetics have been used.
- The results have been verified using a reference CPU implementation - the root mean square deviation is in the order of 10^{-10}.

- Over 95% of the computational time for Shortwave radiation is being spent within the `radsw` subroutine. This routine consists of approximately 300 lines of code, executing multiple sweeps over the `K` dimension for all `IJ` data points and aggregating over 22 spectra, thus containing 4th order loops. A total of 30 `ijk` data grids are accessed twice (read and write) at each data point for each of the spectra.

- As of expensive operations there are 22 reciprocals being executed per data point per spectrum. NVIDIA does not specify the number of cycles needed for double precision reciprocals (they are implemented in software by the CUDA compiler). Assuming 40 cycles per reciprocal[7] this results in $\frac{22 \cdot 40}{60} \approx 14.7$ operations per fetch / write, keeping this kernel well within the memory bandwidth bounded region for the GPU. For multicore CPU we expect this module to be at the edge of being computationally bounded (since the cheap `add` and `mult` operations also add to the balance of the module) while for single core CPU we expect it to be clearly computationally bounded (see also sec. 2.4.2).

- There are no stencil accesses with offsets in the `I` or `J` domain - only the `K` data dimension is accessed with offsets. This is a common characteristic for all ASUCA Physical Process modules that have been examined over the course of this thesis.

Compiler flags additional to those stated in sec. 2.2:

-Minline=levels:5,reshape enables inlining for routines defined in the same module as their caller. The `reshape` option enables the compiler to perform array reshaping when passing them to the callee.

-Mipa=inline,reshape enables inlining for routines defined in different modules from their caller.

-r8 sets double precision as the default option for `real` symbols.

[7] double the number of cycles needed by the AMD K7 CPU architecture [25, p. 1]

2.6 Comparison of Usability

In this section we compare the usability of the evaluated solutions for Fortran GPU code: CUDA Fortran, PGI OpenACC and HMPP OpenACC.

2.6.1 HMPP OpenACC versus PGI OpenACC

The two simple examples (particle push and 3D diffusion) have not shown major differences between HMPP OpenACC and PGI OpenACC in terms of usability. The Fortran implementation of shortwave radiation, however, has revealed some problems when using HMPP:

1. Compiler errors in the tested HMPP versions often do not show line numbers, making it unnecessarily hard to find the errors in large modules.
2. HMPP implemented OpenACC kernels are not able to reference scalars, both imported from other modules and local ones, that haven't been explicitely declared with OpenACC directives or passed as arguments.
3. The HMPP compiler did not accept the `-r8` compiler flag which tells the compiler to keep `real` variables by default in double precision, a functionality that has been used within the ASUCA Physical Process.

Especially item 2 in the above list would have lead to major restructuring of the OpenACC code in order to ensure compatibility with HMPP. In light of the performance results shown in sec. 2.7.2 and 2.7.3 it has been decided to not pursue this endeavor.

2.6.2 Kernel Subprocedure Inlining versus Loop Restructuring

In sec. 2.5.3 a target conflict in terms of loop positionings for CPU and GPU has been introduced. Porting a CPU implementation to GPU with the parallel region loops in an outside position often leads to decisions between

- trying to inline the subprocedures using compiler inlining.
- inlining the subprocedures manually.
- moving the parallel loop regions into the subprocedures, thus passing the complete data grids down to these subprocedures.

Experience has shown that inlining from other Fortran modules using Interprocedural Analysis (IPA) compiler options does not work reliably in conjunction with GPU kernel generation with `pgf90` and it is best not to go that path, i.e.

subprocedure calls into different modules are a clear indicator for the necessity of moving the loops into a deeper callgraph position for GPU execution. This leaves the decision for inlining within the same module. Fig. 2.2 depicts the compatibility for inlining subprocedures with CUDA while fig. 2.3 shows the same for PGI OpenACC.

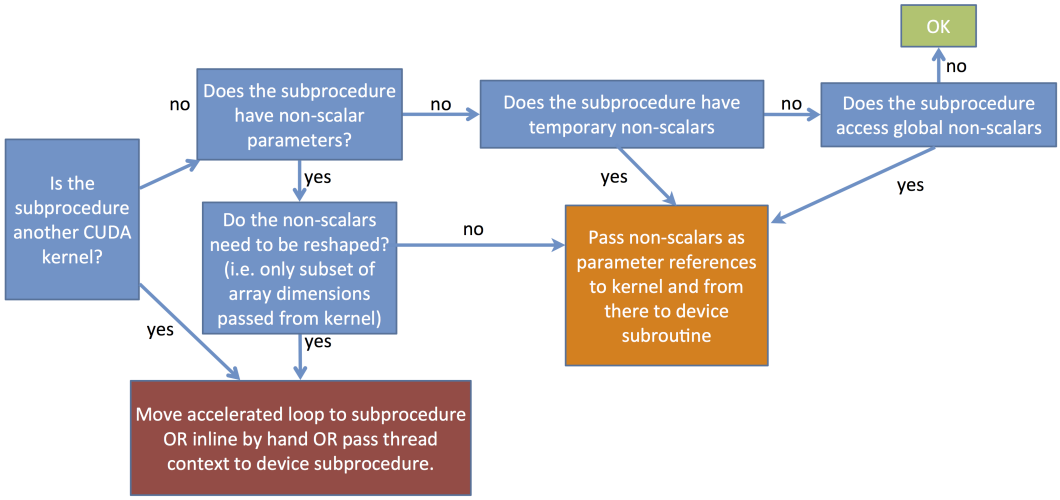


Figure 2.2: CUDA Fortran kernel contains a device subprocedure call. Can it be inlined and therefore be compiled to device code?

It should become clear that with these restrictions present, porting those parts of the ASUCA physical process with deep call graphs becomes a time consuming process. PGI OpenACC only offers limited help in that regard. Its advantages in comparison to CUDA Fortran in terms of porting code of the nature of the ASUCA physical process, are as follows:

1. The ability to introduce multiple parallel loops within one subprocedure.
2. The ability to use one dimensional module arrays.
3. The ability to use local arrays within kernel subroutines (not within device subroutines called by kernels however).

Item 1 saves the introduction of some subprocedures, however one could argue that the splitting of code into multiple subprocedures may in that case lead to a clearer, more maintainable codebase. Items 2 and 3 allow the use of preinitialised data and local arrays without passing them as parameters to kernels. However experience has shown that the most significant portation cost lays in the restructuring of data accesses, data parameters and loops to meet the requirements of GPU execution - an area in which OpenACC does not offer an advantage over CUDA Fortran.

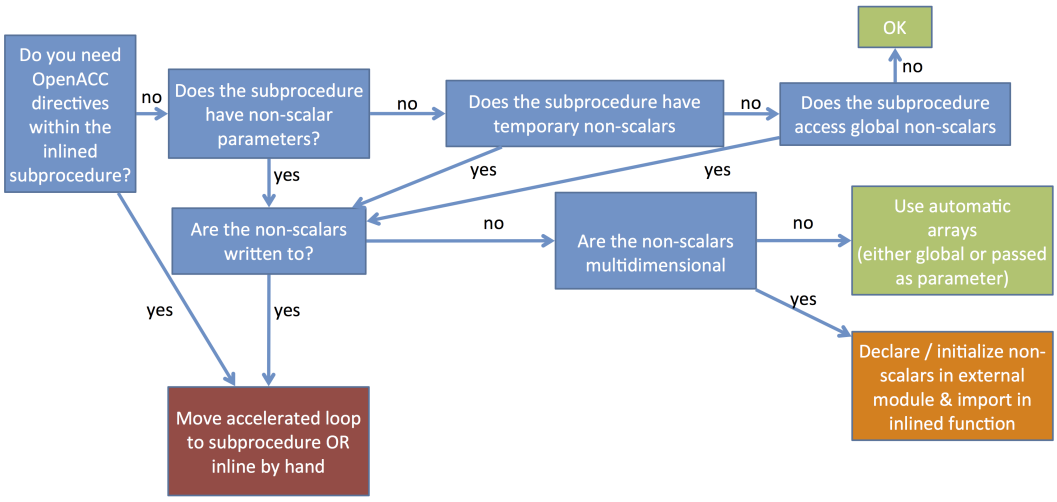


Figure 2.3: PGI OpenACC accelerated region contains a subprocedure call. Can it be inlined and therefore be compiled to device code?

2.6.3 Overview Usability

Tab. 2.1 gives an overview over the usability issues we have found when porting CPU code to the GPU using one of the aforementioned frameworks.

Issue	CUDA Fortran	PGI OpenACC	HMPP OpenACC
Subprocedure calls in parallel regions possible	see figure 2.2	see figure 2.3	Not examined
Subprocedure calls from different modules possible	No	No	Not examined
Multiple parallel regions per subroutine	No	Yes	Yes
Automatic device data copies	No	Yes	Yes
Use of local scalars	Yes	Yes	Needs directives
Use of local arrays	No	Yes	Not examined
Use of scalars imported from external modules	No	Yes	No
Recursive subprocedures	No	No	No
Inline array initialisations	No	No	No
`SAVE`, `DATA` attributes	No	No	No
Abstraction of parallel domain dependency	No	No	No

Table 2.1: Usability issues with GPGPU in Fortran.

2.7 Comparison of Performance

In this section the performance comparisons between CPU execution, OpenACC GPU execution and CUDA will be discussed.

2.7.1 Overview of Tests

The following table gives an overview over the preliminarily investigated GPGPU technologies and the applied test cases.

Framework	Version	Maintainer	Test Cases
PGI OpenACC For C Language	12.4	PGI	Particle Push, 3D Diffusion
PGI OpenACC For Fortran Language	12.4	PGI	ASUCA Shortwave Radiation
HMPP OpenACC For C Language	3.1.0	CAPS	Particle Push, 3D Diffusion
HMPP OpenACC For Fortran Language	3.1.0	CAPS	None, see sec. 2.6.1
CUDA C	4.1	NVIDIA	Particle Push, 3D Diffusion
CUDA Fortran	12.4	PGI	ASUCA Shortwave Radiation

Table 2.2: Overview frameworks and test cases.

For the test platform's hardware specifications please refer to sec. 2.4.

All results shown here have been averaged over 5 runs. Only computation time, no host-to-device data copy time has been counted, since the host-to-device bus performance is not expected to be relevant for ASUCA (the goal is a complete GPU portation, making data copying only necessary at the beginning and end of the simulation, thus significantly reducing communication overhead).

2.7.2 3D Diffusion

Fig. 2.4 and fig. 2.5 illustrate the following examinations:

1. The magnitude of speedups further indicate that this algorithm is being bounded by memory bandwidth (as estimated in sec. 2.5.1).

2. The speedups of GPU implementations versus the CPU implementation are in line with the hardware model introduced in sec. 2.4.1 for the single precision results. The double precision speedup is lower than expected however (speedups of the same magnitude as single precision execution are expected for double precision execution). Considering the execution time results shown in fig. 2.4 the CUDA and PGI OpenACC implementations

scale as expected (around a factor of two), however CPU execution as well as HMPP do not scale as expected, indicating some overhead being present in their single precision versions. Considering, that this algorithm is most likely heavily bounded by memory bandwidth, overfetches are a likely candidate for the deviations from the model - i.e. data is being read into cache that is not needed by the active thread - a phenomenon that is less strong with double precision execution, since the payload data is always twice the size. It is therefore likely that this algorithm could be further optimized by optimizing the code for cache accesses - however this kind of optimization is very specific to the architecture.

3. PGI OpenACC performs significantly better than HMPP for this example.

4. CUDA Fortran performs significantly better than the OpenACC implementations in the single precision case. In double precision, CUDA Fortran and PGI Fortran perform very similarly.

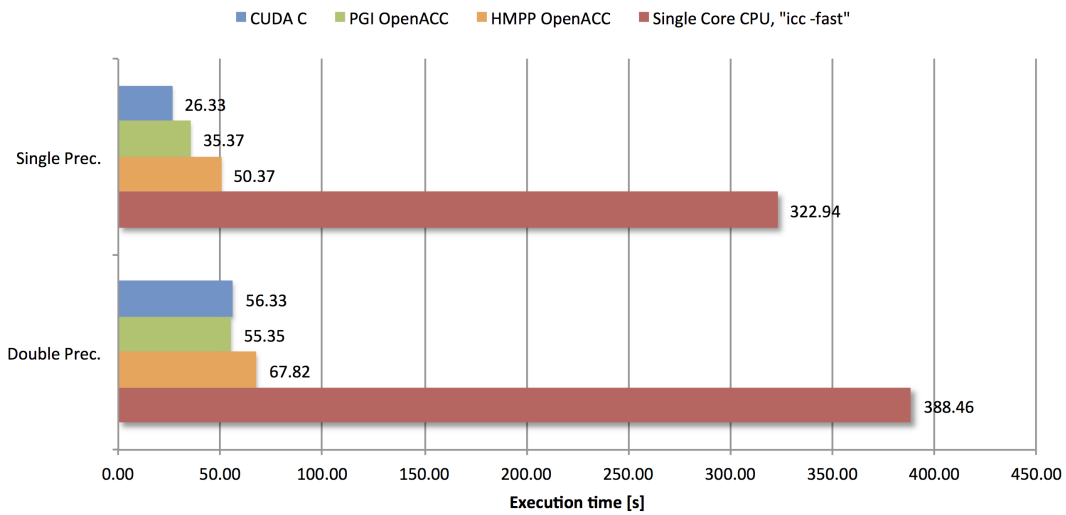


Figure 2.4: Execution time results for the 3D Diffusion Algorithm.

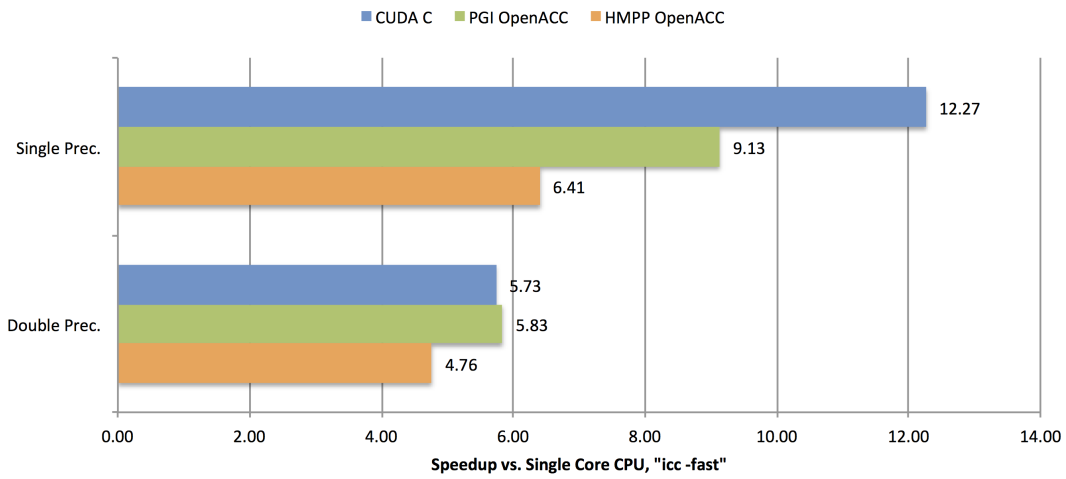


Figure 2.5: Speedup results for the 3D Diffusion Algorithm when compared to Single Core CPU, “icc -fast” compiled.

2.7.3 Particle Push

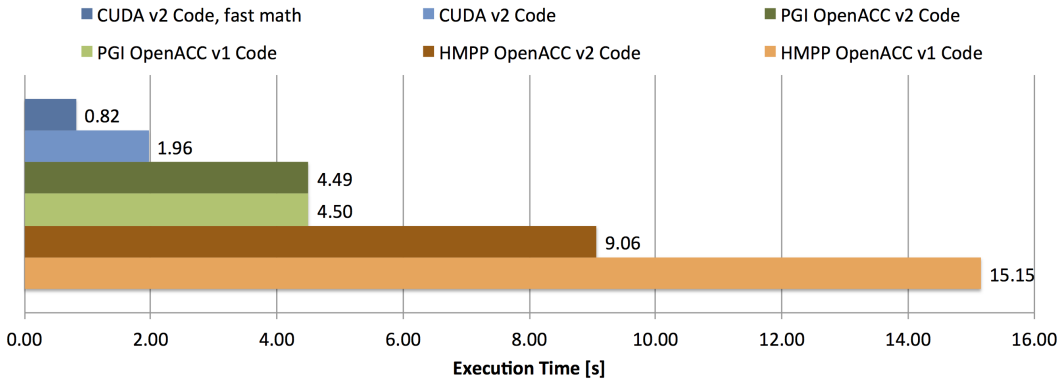


Figure 2.6: Single precision execution time results for the Particle Push Algorithm. CPU results are not shown here since the two orders of magnitude difference would make it hard to distinguish the GPU results.

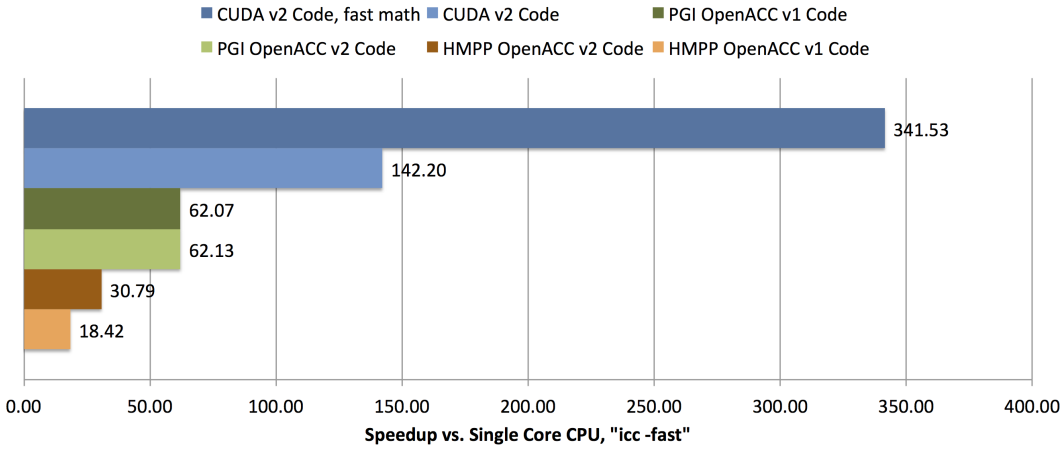


Figure 2.7: Single precision speedup results for the Particle Push Algorithm when compared to Single Core CPU, “icc -fast” compiled, v1 Code.

Fig. 2.6 and fig. 2.7 illustrate the following examinations:

1. The speedups of GPU implementations versus the CPU implementation observed with this example are significantly higher than what one could expect from the performance metrics assessed in sec. 2.4. This indicates the following:
   (a) This algorithm is indeed computationally bounded as assumed in sec. 2.5.2. Speedups of this magnitude for memory bandwidth bounded problems are highly unlikely.

   (b) The provided GPU architecture as well as the CUDA compiler appear to be more optimized for Sine- and Cosine operations. This is a reasonable assumption to make since Sine- and Cosine operations are very important for geometrical computations, one of the key areas of the GPU.

   (c) It is likely that the CPU implementation could still be optimized by a significant margin.

2. PGI OpenACC performs significantly better than HMPP for this example.

3. PGI OpenACC doesn't show a significant improvement between the code versions 1 and 2 (see sec. 2.5.2). An examination of the CUDA C code created by the PGI OpenACC compiler has revealed that the PGI compiler is able to perform these optimizations for the code version 1 as well.

4. The CUDA version is still faster by a factor of 2.3 compared to the PGI OpenACC code. The code examination of PGI OpenACC's created code has revealed that there are additional safety branches being added to the kernel code - which is not needed for the CUDA code implemented by hand, since the programmer is able to configure the kernel in a way that it will not exceed the data boundaries. One would expect this to be the reason for the slowdown of the PGI OpenACC version.

5. The `fastmath` special function units (a hardware implementation of Sine- and Cosine functions among others) are not responsible for the above mentioned speedup of CUDA vs. OpenACC, since enabling `fastmath` gives an additional speedup factor of 2.4. Note: Doing so results in a slight loss in fidelity (the GPU's root mean square error versus reference result changes from $1.62 \cdot 10^{-6}$ to $2.33 \cdot 10^{-6}$).

2.7.4 ASUCA Shortwave Radiation

This section shows the performance test results for the ASUCA shortwave radiation module. Since OpenACC allows hybrid execution, it was also tested how a hybridized CUDA version compares in terms of CPU performance. Therefore the CUDA Fortran version has been implemented using preprocessor directives in order to hybridize that codebase[8], rendering the kernel code compatible for CPU execution after the preprocessor. This hybridized CUDA version will be referred to as "Preprocessed CUDA Fortran". Some performance loss on the CPU was expected since the loop structure for these implementations is optimized for GPU execution (see also sec. 2.5.3).

Fig. 2.8 and fig. 2.9 illustrate the following:

1. The magnitude of speedups further indicate that this algorithm is being bounded by memory bandwidth (as estimated in sec. 2.5.3 for GPU execution).

2. The speedups of GPU implementations versus the CPU implementation are only slightly below what is expected for memory bandwidth limited modules as outlined in sec. 2.4.1. Judging from the hardware model another 10% speedup could be expected from further optimizing the GPU version.

3. The GPU results of PGI OpenACC are only approximately 5% below those of CUDA Fortran. This indicates that the advantage of CUDA Fortran vs. OpenACC shrinks for larger kernels, when the additions of safety branches (as discussed in sec. 2.7.3) are not as significant anymore compared to the overall load.

4. Executing code with loops optimized for GPU execution on the CPU results in a 2× performance loss in this case. The preprocessed CUDA Fortran code handles this slightly worse than the PGI OpenACC version, loosing another 15%.

[8] E.g. the CUDA kernels have been wrapped in "do"/"end do" loops over the "IJ" domain for CPU execution.

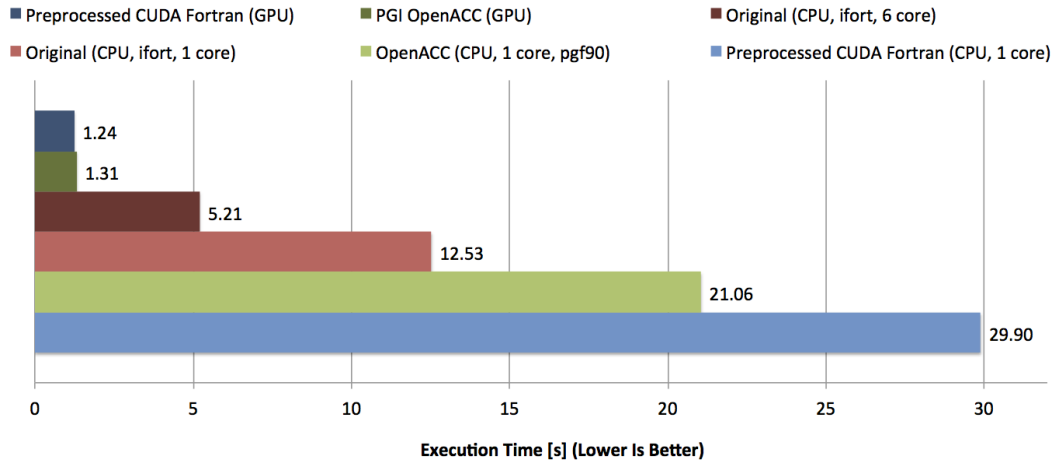


Figure 2.8: Double precision execution time results for the shortwave radiation module.

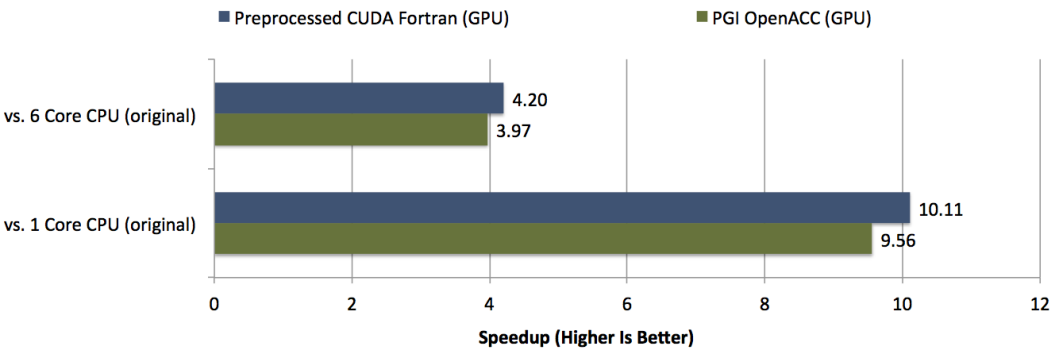


Figure 2.9: Double precision speedup results for the shortwave radiation module when compared to CPU execution of the original code version, "icc -fast" compiled.

2.8 Preliminary Conclusions

I would like to draw the following preliminary conclusions regarding the results from sec. 2.6 and sec. 2.7:

1. Porting code with complex call graphs inside parallel regions to GPU will result in the following target conflict:
   (a) Optimization for GPU results in low performance on the CPU. Fig. 2.8 illustrates this point conclusively.
   (b) Optimization for CPU lowers GPU performance and even breaks GPU compatibility at some point.
2. Changing the loop structure and data accesses is considered to be the most time consuming task for porting the ASUCA physical process to GPU.
3. In case the OpenACC route is to be pursued, the results shown in this thesis indicate a clear advantage of PGI over HMPP, both in usability and performance. Please note, however, that OpenACC support has still been under heavy development during the timeframe of this thesis - later versions would have to be reexamined.
4. PGI OpenACC is able to perform rather intricate optimizations of arithmetics, as shown in sec. 2.7.3. Kernel code created by PGI OpenACC might be helpful to examine in case of computationally bounded code.
5. The speedup estimates for the three test examples based on the hardware model introduced in sec. 2.4 have proven to be reasonable and may be taken into consideration for further analysis.

Items 1 and 2 in the above list lead to the following question: Is there a solution offering

1. lower portation cost than OpenACC,
2. high GPU performance and
3. high CPU performance?

The following chapters will answer this question.

Chapter 3

The Hybrid Fortran Framework

Chapter 2 has shown that OpenACC has some positive aspects, it will however in many cases not achieve optimal performance and the portation in case of the ASUCA physical process codebase is not significantly easier than the portation to CUDA Fortran. Over the course of the evaluation phase this insight lead to the birth of a new idea: A framework, specifically targeted at, but not limited to, the ASUCA Physical Process, that gives the advantages of a hybrid codebase and offers fully optimized performance both on CPU and GPU. This chapter will describe the functionality of the **Hybrid Fortran** framework from a user perspective, while chapter 4 will go into implementation details.

3.1 Design Goals

This section gives an overview over the design goals of the **Hybrid Fortran** framework.

1. **Hybrid Fortran** offers a unified codebase for both CPU and GPU execution.
2. The storage order must be compile time defined since CPU and GPU implementations usually have different ideal storage orders.
3. The rules for creating the GPU code versions should have a low complexity, such that optimizations performed by the user lead to predictable results.
4. The framework enables GPU performance at the level of hand optimized CUDA Fortran while maintaining CPU performance as close as possible to the original ASUCA physical process code.

5. The implementation details are abstracted from the rest of the system, such that other parallel programming frameworks can be supported in the future without changes in the user code.

6. Since the ASUCA Physical Process' stencil computations do not have dependencies at offset positions in `I` and `J` direction and its original source tree is programmed to operate over one `K` column with the `IJ` loops outside, it is beneficial to the portation process and possibly the CPU performance to hide or abstract `IJ` dependencies of arrays.

7. Since the `IJ` dependencies are abstracted, it becomes possible to define the loops over the `IJ` domains at compile time. In other words, the framework is to be able to switch at compile time between outside loops (better suited for CPU caching) and inside loops (better suited for the GPU streaming computation model).

3.2 Hybrid Fortran Directives

The directives introduced with the **Hybrid Fortran** framework are the only additions that have been made to the Fortran 90 language in order to achieve the objectives stated in sec. 3.1. All other syntax elements in **Hybrid Fortran** are a strict subset of Fortran 90.

It is necessary to introduce the following denotations before introducing the new directives:

A domain in this context denotes a tuple containing a data dimension and its size. For example, if we have an array `a` declared within the range (1, NX) and looped over using the iterator `x`, we call this array to be **domain dependant** in domain `x`. For simplicity, we assume that iterators over the same data dimension and range are always named consistantly the same - a coding style that has been present within all the modules examined in the ASUCA Physical Process.

A parallel region is a code region that can be executed in parallel over a one or more domains.

The following section lists the two directives and their available options for later reference.

3.2.1 Domain Dependant Directive

Listing 3.1 shows the "domain dependant" directive. They are used to specify the involved symbols and their domain dependencies. This information then allows the framework to rewrite the symbol accesses and declarations for the GPU and CPU cases (see example in section 3.2.3).

Notes:

1. For symbols the framework only operates on local information available for each subroutine. As an example, whether a symbol has already been copied to the GPU is not being analyzed. For this reason the `present` flag has been introduced (see below).

2. Domain Dependant Directives need to be specified between the specification and the implementation part of a Fortran 90 subroutine.

```
1 @domainDependant{ATTRIBUTE_NAME1(MEMBER1, MEMBER2, ...), ...}
2 ! symbols that share the attributes !
3 ! defined above to be defined here, separated  !
4 ! by commas                                     !
5 ...
```

```
6 @end domainDependant
7
8 !Minimal Example:
9 @domainDependant{domName(x), domSize(NX)}
10 a, b, c
11 @end domainDependant
12 !-> Defines the three arrays a, b, c to be dependant in domain x.
```

Listing 3.1: Domain dependant directive syntax.

The following attributes are supported for this directive:

domName Set of all domain names in which the symbol needs to be privatized. This needs to be a superset of the domains that are being declared as the symbol's dimensions in the specification part of the current subroutine (except if the $|autoDom|$ attribute flag is used (see below)). More specifically, the domain names specified here must the set of domains from the specification part plus the parallel domains (as specified using the parallel region directive, see section 3.2.2) for which privatization is needed.

domSize Set of the domain dimensions in the same order as their respective domain names specified using the `domName` attribute. It is required that $|domName| = |domSize|$.

accPP Preprocessor macro name that takes $|domSize|$ arguments and outputs them comma separated in the current storage order for symbol accesses. This macro must be defined in the file `storage_order.F90` (see section 4.1).

domPP Preprocessor macro name that takes $|domSize|$ arguments and outputs them comma separated in the current storage order for the symbol declaration. This macro must be defined in the file `storage_order.F90` (see section 4.1). Note: This preprocessor macro is usually identical to the one defined in `accPP`.

attribute Attribute flags for these symbols. Currently the following flags are supported:

present In case this flag is specified, the framework assumes array data to be already present on the device memory for GPU compilation.

autoDom In case this flag is specified, the framework will use the array dimensions that have been declared using standard Fortran 90 syntax to determine the non parallel domains. Parallel domains (as specified using the parallel region directive, see section 3.2.2) still need to be defined using the $|domName|$ and $|domSize|$ attributes. In addition, using $|autoDom|$ will by default enable standard $|accPP|$ and $|domPP|$ settings, if not specified otherwise. This means the framework assumes

|*DOM*|/|*AT*| (3D), |*DOM*4|/|*AT*4| (4D), |*DOM*5|/|*AT*5| (5D) (etc.) preprocessor directives to be specified for the compile time defined storage order (below three dimensional symbols don't get any directive added). Using this flag then greatly simplifies the |*domainDependant*| specification part, since the directive template (everything between the directive and the corresponding |*enddomainDependant*| statement) can be reused by symbols of different domains.

An important special case are scalar parameters: In case of a CUDA Fortran implementation, scalars must be passed by value in device functions. The framework must for that reason be aware of scalar symbols, such that their specification can be adjusted accordingly. For simplicity reasons, the `@domainDependant` directive has been reused for scalars and can be used in the following way:

```
1 @domainDependant{}
2 scalar1, scalar2, ...
3 @end domainDependant
```

Listing 3.2: Domain dependant directive syntax for scalars.

In **Hybrid Fortran** terms then, a scalar is a domain dependant without domains.

3.2.2 Parallel Region Directive

Listing 3.3 shows the parallel region directive. This directive is an abstraction of for-loops as well as CUDA kernels that allows the framework to define these structures at compile time. It is only allowed to be inserted in the implementation part of a subroutine.

```
1 @parallelRegion{ATTRIBUTE_NAME1(MEMBER1, MEMBER2, ...), ...}
2 ! code to be executed in parallel !
3 symbol1, symbol2, ...
4 @end parallelRegion
```

Listing 3.3: Parallel region directive syntax.

The following attributes are supported for this directive:

appliesTo Specify one or more of the following attribute members in order to set this parallel region to apply to either the CPU code version, the GPU version or both.

1. CPU
2. GPU

domName Specify one or more domain names over which the code can be executed in parallel. These domain names are being used as iterator names for the respective loops or CUDA kernels.

domSize Set of the domain dimensions in the same order as their respective domain names specified using the `domName` attribute. It is required that $|domName| = |domSize|$.

3.2.3 Example

Let's look at the following example module that performs matrix element addition as well as multiplication. Please note, that the storage order in this example is defined at compile-time using the preprocessor macros `DOM` and `AT`. These macros simply reorder the arguments in order to reflect the optimal storage order for CPU and GPU case.

```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      real, intent(in) :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY, NZ))
5      real, intent(out) :: c(DOM(NX, NY, NZ)), d(DOM(NX, NY, NZ))
6      integer(4) :: x, y
7
8      do y=1,NY
9        do x=1,NX
10         call add(a(AT(x,y,:)), b(AT(x,y,:)), c(AT(x,y,:)))
11         call mult(a(AT(x,y,:)), b(AT(x,y,:)), d(AT(x,y,:)))
12       end do
13     end do
14   end subroutine
15
16   subroutine add(a, b, c)
17     real, intent(in) :: a(NZ), b(NZ)
18     real, intent(out) :: c(NZ)
19     integer :: z
20
21     do z=1,NZ
22       c(z) = a(z) + b(z)
23     end do
24   end subroutine
25
26   subroutine mult(a, b, d)
27     real, intent(in) :: a(NZ), b(NZ)
28     real, intent(out) :: d(NZ)
29     integer :: z
30
31     do z=1,NZ
32       d(z) = a(z) * b(z)
33     end do
34   end subroutine
35 end module example
```

Listing 3.4: CPU version of matrix element module

Porting this to the GPU, one might want to move the loops over the `x` and `y` domains to the `add` and `mult` subroutines, in order to eliminate the need for

inlining and optimize for register usage.

The following listing shows how this module looks like in **Hybrid Fortran**. Figure 4.3 in chap. 4 shows the (programmatically created) callgraph of this module.

```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      real, dimension(NZ), intent(in) :: a, b
5      real, dimension(NZ), intent(out) :: c, d
6
7      @domainDependant{domName(x,y,z), domSize(NX,NY,NZ), domPP(DOM),
           accPP(AT)}
8      a, b, c, d
9      @end domainDependant
10
11     @parallelRegion{appliesTo(CPU), domName(x,y), domSize(NX, NY)}
12     call add(a, b, c)
13     call mult(a, b, d)
14     @end parallelRegion
15   end subroutine
16
17   subroutine add(a, b, c)
18     real, dimension(NZ), intent(in) :: a, b
19     real, dimension(NZ), intent(out) :: c
20     integer :: z
21
22     @domainDependant{domName(x,y,z), domSize(NX,NY,NZ), domPP(DOM),
           accPP(AT)}
23     a, b, c
24     @end domainDependant
25
26     @parallelRegion{appliesTo(GPU), domName(x,y), domSize(NX, NY)}
27     do z=1,NZ
28       c(z) = a(z) + b(z)
29     end do
30     @end parallelRegion
31   end subroutine
32
33   subroutine mult(a, b, d)
34     real, dimension(NZ), intent(in) :: a, b
35     real, dimension(NZ), intent(out) :: d
36     integer :: z
37
38     @domainDependant{domName(x,y,z), domSize(NX,NY,NZ), domPP(DOM),
           accPP(AT)}
39     a, b, d
40     @end domainDependant
41
42     @parallelRegion{appliesTo(GPU), domName(x,y), domSize(NX, NY)}
43     do z=1,NZ
44       d(z) = a(z) * b(z)
45     end do
```

```
46     @end parallelRegion
47   end subroutine
48 end module example
```

Listing 3.5: example of a Hybrid Fortran subroutine with a parallel region

This will be rewritten by the **Hybrid Fortran** framework into two versions. The CPU version will be exactly like the original, while the GPU version is shown below. Please note, that

1. the two implementations loop over the `xy` domains at different points, according to the `appliedTo` attribute defined in the `@parallelRegion` directives. See figure 4.3 in cha. 4 in order to get an overview.
2. the declarations and accessors for the arrays `a`, `b`, `c` and `d` did not need to be changed in the `add` and `mult` subroutines.
3. the left whitespace is not always preserved in the actual implementation, it has been slightly reformatted here for improved readability.

```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      use cudafor
5      real, intent(in) :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY, NZ))
6      real ,device :: a_d(DOM(NX, NY, NZ))
7      real ,device :: b_d(DOM(NX, NY, NZ))
8      real, intent(out) :: c(DOM(NX, NY, NZ)), d(DOM(NX, NY, NZ))
9      real ,device :: c_d(DOM(NX, NY, NZ))
10     real ,device :: d_d(DOM(NX, NY, NZ))
11
12     type(dim3) :: cugrid, cublock
13     integer(4) :: cuerror
14     a_d(:,:,:) = a(:,:,:)
15     c_d(:,:,:) = 0
16     b_d(:,:,:) = b(:,:,:)
17     d_d(:,:,:) = 0
18
19     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
20     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
21     call add <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:))
           , c_d(AT(:,:,:)))
22     ! **** error handling left away to improve readability *** !
23
24     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
25     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
26     call mult <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:)
           ), d_d(AT(:,:,:)))
27     ! **** error handling left away to improve readability *** !
28
29     c(:,:,:) = c_d(:,:,:)
30     d(:,:,:) = d_d(:,:,:)
```

```
31   end subroutine
32
33   attributes(global)   subroutine add(a, b, c)
34     use cudafor
35     real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
         NZ))
36     real, intent(out) ,device :: c(DOM(NX, NY, NZ))
37     integer :: z
38     integer(4) :: x, y
39
40     x = (blockidx%x - 1) * blockDim%x + threadidx%x
41     y = (blockidx%y - 1) * blockDim%y + threadidx%y
42     do z=1,NZ
43       c(AT(x,y,z)) = a(AT(x,y,z)) + b(AT(x,y,z))
44     end do
45   end subroutine
46
47   attributes(global) subroutine mult(a, b, d)
48     use cudafor
49     real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
         NZ))
50     real, intent(out) ,device :: d(DOM(NX, NY, NZ))
51     integer :: z
52     integer(4) :: x, y
53
54     x = (blockidx%x - 1) * blockDim%x + threadidx%x
55     y = (blockidx%y - 1) * blockDim%y + threadidx%y
56     do z=1,NZ
57       d(AT(x,y,z)) = a(AT(x,y,z)) * b(AT(x,y,z))
58     end do
59   end subroutine
60 end module example
```

Listing 3.6: GPU version of the hybrid code shown above

3.3 Restrictions

The following restrictions will need to be applied to standard Fortran 90 syntax in order to make it compatible with the **Hybrid Fortran** framework in its current state. For the most part these restrictions are necessary in order to ensure CUDA Fortran compatibility. Other restrictions have been introduced in order to reduce the program complexity while still maintaining suitability for the ASUCA physical process.

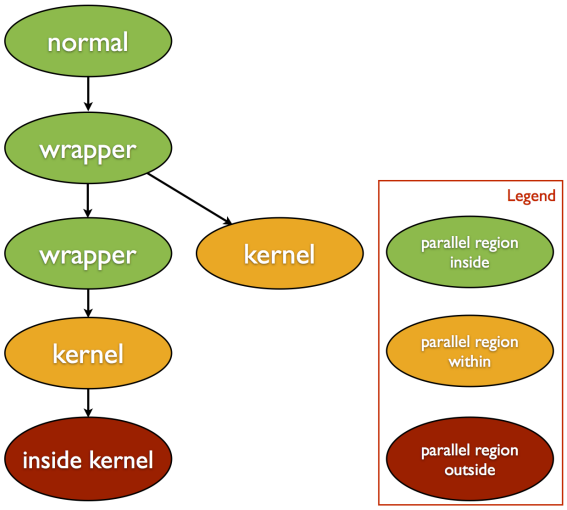


Figure 3.1: Callgraph showing subroutine types with restrictions for GPU compilation.

1. No dependencies at offsets in I,J-directions (only vertical dependencies).

2. Hybrid Fortran has only been tested using Fortran 90 syntax and its GNU Make based build system only supports Fortran 90 files (f90 / F90). Since the Hybrid Fortran preprocessor only operates on subroutines (i.e. it is not affected by OOP specific syntax), this restriction can be lifted soon.

3. Currently no line continuations are supported for the attribute definitions of directives. Instead, directives are designed to only require a limited amount of attributes.

4. CUDA Fortran differentiates between different subroutine types [26, p. 4]. Following its design goal of keeping a low complexity, **Hybrid Fortran** simply rewrites subroutines definitions to one of the CUDA subroutine types, depending on the subroutine's position relative to the parallel region (determined through metainformation about the entire visible source

code). This introduces some restrictions for subroutines calling, containing or being called by GPU parallel regions. For future reference these restricted subroutines are named in the following way (see figure 3.1):

(a) Subroutines that call one or more subroutines containing a GPU parallel region are called "wrapper subroutines".

(b) Subroutines that contain a GPU parallel region are called "kernel subroutines".

(c) Subroutines that are called inside a GPU parallel region are called "inside kernel subroutines".

Now that the names of those special subroutines are defined, it is possible to state the following restrictions. Because of CUDA Fortran restrictions, kernel- and inside kernel subroutines may not

(a) contain symbols declared with the `DATA` or `SAVE` attribute.

(b) contain multiple parallel regions.

(c) be recursive.

(d) call other kernel subroutines.

(e) contain the `recursive`, `pure` and `elemental` keywords.

(f) contain inline array initialisations. Static data initialisations are ideally being done outside the parallel loop, typically in `init` subroutines for each module that are always executed on the CPU. **Hybrid Fortran** directives are not needed for code parts that are in no case to be executed on the GPU, however the compile time storage order needs to be be respected by using the same storage order macros as specified in the domPP and accPP attributes for the involved arrays.

5. Inside kernel subroutines called by kernel subroutines must reside in the same Fortran module as their caller.

6. All non scalar symbols with attributes added through `@domainDependant` directives may only be accessed and set inside parallel regions.

7. Arrays that are declared as domain dependant using `@domainDependant` directives must be of `integer` or `real` type (however any byte length within the Fortran 90 specification is allowed).

8. Arrays that are declared as domain dependant using `@domainDependant` directives may not appear in declaration lines with mixed domain dependance. Example:

```
1 ..
2 real(8), dimension(nz) :: a, b
3 real(8), dimension(nz) :: c
4 ..
```

```
5  @domainDependant {domName(x), domSize(nx)}
6  a, b
7  @end domainDependant
8
9  @domainDependant {domName(y), domSize(ny)}
10 c
11 @end domainDependant
12 ..
```

Listing 3.7: This is ok.

```
1  ..
2  real(8) dimension(nz) :: a, b, c
3  ..
4  @domainDependant {domName(x), domSize(nx)}
5  a, b
6  @end domainDependant
7
8  @domainDependant {domName(y), domSize(ny)}
9  c
10 @end domainDependant
11 ..
```

Listing 3.8: This is not ok.

9. The regular Fortran 90 declarations of any symbols declared as domain dependant may not contain line continuations.
10. All source files (h90[1], H90, f90 and F90) need to have distinctive filenames since they will be copied into flat source directories by the build system.
11. Only subroutines are supported together with **Hybrid Fortran** directives, e.g. functions are not supported.
12. Preprocessor directives that affect the Hybrid Fortran preprocessing (such as code macros) must be expandable from definitions within the same H90 file. Use the H90 file suffix (instead of h90) in case you want to use macros in your code.
13. `@domainDependant` directives are required for all arrays in all subroutines called within parallel regions (the preprocessor operates only on local symbol information within each subroutine).

In general, since

1. GPU execution currently requires subroutine calls to be inlined and
2. the number of registers per GPU Streaming Multiprocessor is very limited

it is best to split deep callgraphs and large computations into multiple smaller kernels (i.e. `@parallelDomain{ appliedTo(GPU), ..}`).

[1] h90 is the file extension used for Hybrid Fortran source files.

3.4 Device Data Handling

The goal of device data handling is to hide and abstract code that is only necessary for the CUDA execution path. **Hybrid Fortran** works similarly to OpenACC in that respect. For all non-scalars that are marked as domain dependant using the `@domainDependant` directive, the following rules apply with respect to device data in case of GPU compilation:

1. If the current subroutine contains calls to kernel subroutines and the domain dependant symbol is declared using the `intent(in)` or `intent(inout)` statement, a device version of the symbol will be allocated and its content will be copied to that device array at the beginning of the subroutine.

2. If the current subroutine contains calls to kernel subroutines and the domain dependant symbol is declared using the `intent(out)` or `intent(inout)` statement, a device version of the symbol will be allocated and set to zero and the device array's content will be copied to the original array at the end of the subroutine.

3. If the domain dependant symbol is local for this subroutine, it will be allocated as a device symbol and its content will be set to zero at the beginning of the subroutine.

4. In case the domain dependant directive contains an `attribute(present)` statement, no data will be copied and the original symbol will be declared as a device symbol.

3.5 Feature Comparison between Hybrid Fortran and OpenACC

The following table gives an overview over the differences between OpenACC and the **Hybrid Fortran** framework.

Feature	OpenACC	Hybrid Fortran 90	Comments
Enables close to fully optimized Fortran code for GPU execution		✓	Details, see section 5.3.2
Enables close to fully optimized Fortran code for CPU execution		✓	Details, see section 5.3.1
Automatic device data copying	✓	✓	
Allows adjusted looping patterns for CPU and GPU execution		✓	
Allows changing the looping patterns with minimal adjustments in user code		✓	
Handles compile time defined storage order		✓	
Allows to adapt for other technologies without changing the user code (e.g. switching to OpenCL)		✓	Details, see section 4.5
Allows arbitrary access patterns in parallel domains	✓		Hybrid Fortran initially designed for ASUCA physical process access patterns (no offsets in I,J domains)
Allows multiple parallel regions per subroutine	✓		
Allows arbitrary CPU compilers	(✓)	✓	OpenACC: Only HMPP
Generated GPU syntax is a direct mapping of Fortran 90 user syntax		✓	OpenACC compiles to CUDA C (PGI), introduces new functions for device kernels. Hybrid Fortran translates to CUDA Fortran, syntax remains easily readable.
Allows debugging of device data		✓	
Framework Sourcecode available		✓	

Table 3.1: Feature Comparison OpenACC vs. Hybrid Fortran

Chapter 4

Framework Implementation

In this chapter the design of the **Hybrid Fortran** framework is presented from the implementation perspective. It will discuss the architecture that has been introduced for implementing the **Hybrid Fortran** framework, in order to achieve the goals and behaviour outlined in sec. 3.1 and sec. 3.2.

4.1 Overview and Build Workflow

The **Hybrid Fortran** build system involves the following components (depicted in fig. 4.1):

project-dir/Makefile offers a convenient interface to the build system. Please refer to appendix A.3 for the usage of this build interface. It performs the following operations (assuming a clean rebuild):

1. Creates a build directory containing subdirectories for CPU and GPU builds.
2. Copies all `f90` and `F90` source files (pure Fortran 90 sources without **Hybrid Fortran** directives) into the CPU and GPU build directories using a flat file hierarchy.
3. Creates the callgraph `xml` file as well as the colored CPU and GPU callgraph versions in the callgraph subdirectory within the build directory.
4. Creates the graphical callgraph representations in the callgraph directory using **Graphviz** libraries.
5. Converts each `h90` source file into `F90` source files, using different implementations and callgraph colorings for the CPU and GPU case. The `F90` files are created in their respective build subdirectories (CPU or GPU).

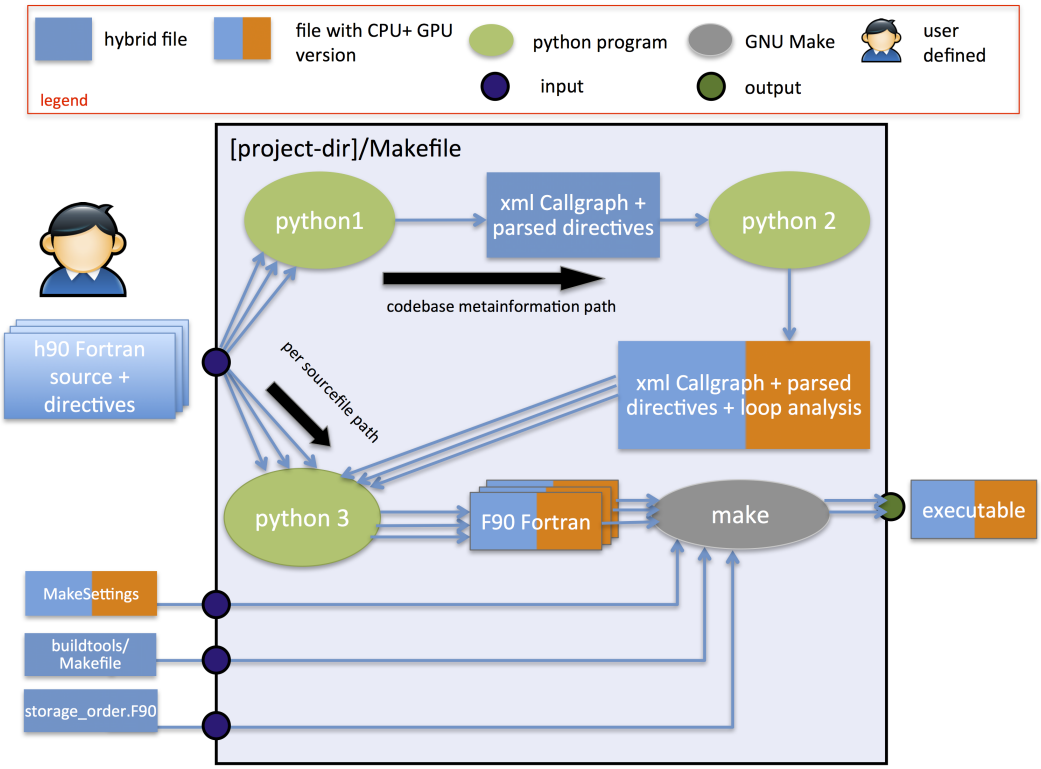


Figure 4.1: **Hybrid Fortran** Components and Information Flow.

6. Copies the `project-dir/buildtools/Makefile` into the CPU and GPU source directories.
7. Copies either `project-dir/buildtools/MakesettingsCPU` and `project-dir/buildtools/MakesettingsGPU` into the respective build subdirectory.
8. Executes `make` within the build subdirectories.
9. Installs the resulting executables into the test directory, using `cpu` or `gpu` as a postfix in the executable filename.

project-dir/buildtools/Makefile defines the dependencies between the Fortran 90 and **Hybrid Fortran** sources.

project-dir/buildtools/MakesettingsCPU defines the compiler name, compiler flags and linker flags for the CPU case.

project-dir/buildtools/MakesettingsGPU defines the compiler name, compiler flags and linker flags for the GPU case.

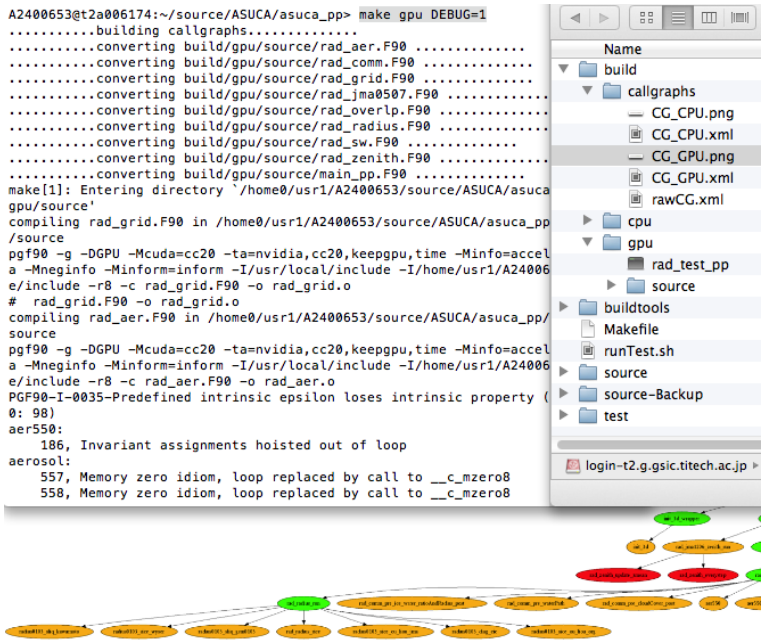


Figure 4.2: Screenshot of the **Hybrid Fortran** build system in action.

4.2 Python Build Scripts

The following python command line interface programs are part of the **Hybrid Fortran** build system:

annotatedCallGraphFromH90SourceDir.py goes through all h90 files in a given source directory and builds an xml file containing meta information about that source tree. The extracted meta information includes the call-graph visible from h90 files as well as a parsed version of the **Hybrid Fortran** directives inserted by the user. Figure 4.1 depicts this program in node `python 1`.

loopAnalysisWithAnnotatedCallGraph.py takes the meta information xml file from the previous script as its input and analyses the positioning of the user defined parallel regions relative to all subprocedures. Depending on its input arguments it performs this analysis for either the CPU or GPU version of the program in order for the framework to support compile time defined positioning of loops (kernel regions in the CUDA implementation). Figure 4.1 depicts this program in node `python 2`.

generateF90fromH90AndAnalyzedCallGraph.py takes one h90 source file as well as the analyzed meta information xml file as its inputs. It goes through the source file line by line and rewrites it in order to create compatible versions for CPU and GPU. The following operations are most essential to this module:

- Rewriting of parallel region definitions to conventional loops for the CPU or CUDA Fortran kernels for the GPU.
- Mutation of declarations and accesses of domain dependant arrays according to their position relative to the currently active parallel region.
- Insertion of statements to copy array data to and from the device in the GPU case.

Parallel domain dependant Figure 4.1 depicts this program in node `python 3`.

graphVizGraphWithAnalyzedCallGraph.py This program has been created in order to make debugging easier and to give the user an overview over the codebase and the involved parallel regions. It creates a graphical representation of the call graph from the analyzed meta information. The nodes in these call graphs are colored according to their relative position to the parallel regions. Figure 4.3 shows a sample of such a programmatically created call graph representation.

4.3 User Defined Files

The following files, depicted figure 4.1, are defined by the user:

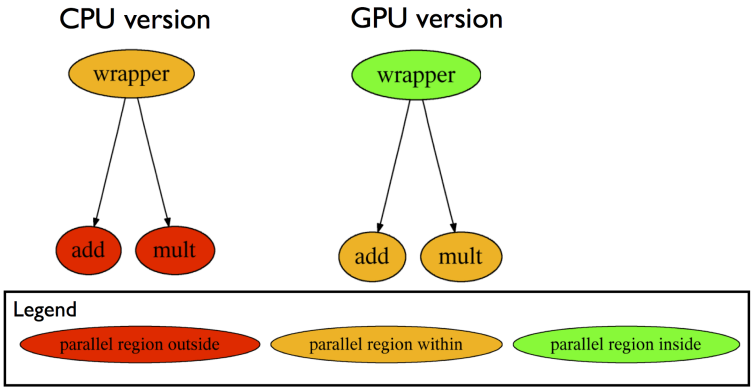


Figure 4.3: Condensed version of a simple callgraph, programmatically created by graphVizGraphWithAnalyzedCallGraph.py.

h90 Fortran sources A source directory that contains **Hybrid Fortran** files (h90 extension). It may also contain files with f90 or F90 extensions. The source directory is by default located at `path-to-project/source/*`.

Makefile Used to define module dependencies. The Makefile is by default located at `path-to-project/buildtools/Makefile`. Note: All source files are being copied into flat source folders before being compiled - the build system is therefore agnostic to the source directory structure implemented by the framework user.

storage_order.F90 This fortran file contains fortran preprocessor statements in order to define the storage order for both CPU and GPU implementation. This file is located at

`path-to-project/source/`
`hybrid_fortran_commons/storage_order.F90.`

4.4 Class Hierarchy

Figure 4.4 shows the classes created to implement the functionality described in section 4.2.

H90Parser parses **Hybrid Fortran** (h90) files. The parser uses a mixture of state machine and regular expression design patterns. More specifically: Each line is matched against a set of regular expressions. The set of regular expressions being used is determined by a state machine and the outcomes of the regular expression matches in turn determine the state transitions.

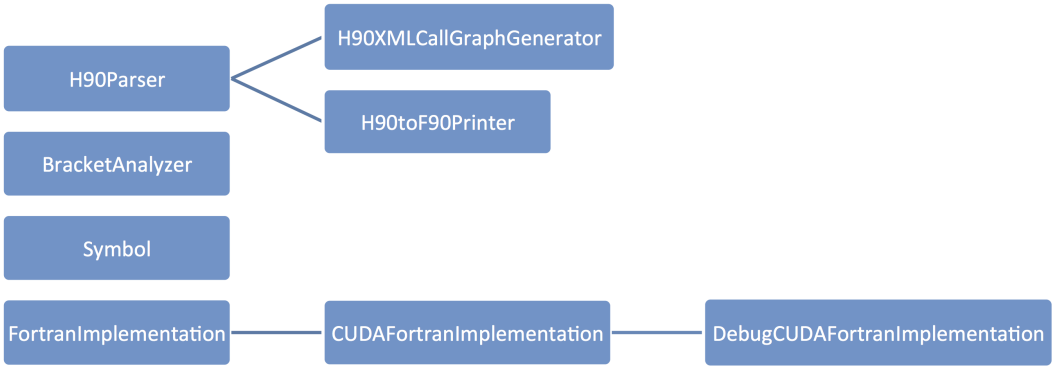


Figure 4.4: **Hybrid Fortran** Python Class Hierarchy.

See section 4.6 for a more detailed look at the **Hybrid Fortran** parser implementation.

H90XMLCallGraphGenerator (subclass of `H90Parser`) adds routine and call nodes to a new or existing call graph xml document. This functionality is used by `annotatedCallGraphFromH90SourceDir.py` as described in section 4.2.

H90toF90Printer (subclass of `H90Parser`) prints a fortran 90 file in F90 format (including preprocessor statements) to POSIX standard output. The configuration of this class includes

1. a **Hybrid Fortran** file as its main input (inherited from the parent class).
2. an xml callgraph including parsed **Hybrid Fortran** directives and the positions of parallel regions relative ot the routine nodes.
3. a FortranImplementation object which determines the parallel implementation.

`generateF90fromH90AndAnalyzedCallGraph.py` uses this functionality as described in section 4.2.

BracketAnalyzer is used to determine whether a Fortran line ends with an open bracket.

Symbol Stores array dimensions determined at the time of declaraton for later use and includes functionality to print adapted declaration and access statements.

FortranImplementation provides the concrete syntax for a standard Fortran 90 implementation of the **Hybrid Fortran** program.

CUDAFortranImplementation (subclass of `FortranImplementation`) provides the syntax for a CUDA Fortran implementation, thus handling

- the conversion of parallel region directives into CUDA kernels,
- the conversion of subroutines called by kernels into device subroutines,
- the copying of data from and to the device,
- the synchronization of threads after CUDA kernels have finished executing (asynchronous execution of kernels is currently not supported) and
- error handling.

DebugCUDAFortranImplementation (subclass of `CUDAFortranImplementation`) extends the CUDA Fortran implementation to include print statements to POSIX standard error output for all kernel parameters at a user defined data point after the execution of the kernel. This functionality enables debugging of device code since barebone CUDA Fortran currently does not offer printing or debugging for code executed on the GPU. (There is an emulation mode available which runs CUDA Fortran programs on the CPU, however it has been found to diverge too much from the device version).

4.5 Switching Implementations

Figure 4.5 shows the the most important class member functions of `FortranImplementation` classes and their role with respect to the example shown earlier in section 3.2.3. Each of these methods takes context information objects (for example a set of symbols that are referenced on this line, or a parallel region template containing the information users have passed with the directives) and returns strings that will be inserted at the indicated places into Fortran 90 files by the `H90toF90Printer` class. Introducing a new underlying technology such as OpenCL (for GPU implementations) or OpenMP (for CPU implementations) is as simple as writing a new `FortranImplementation` subclass containing these functions.


```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      use cudafor
5      real, intent(in) :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY, NZ))
6      real ,device :: a_d(DOM(NX, NY, NZ))
7      real ,device :: b_d(DOM(NX, NY, NZ))
8      real, intent(out) :: c(DOM(NX, NY, NZ)), d(DOM(NX, NY, NZ))
9      real ,device :: c_d(DOM(NX, NY, NZ))
10     real ,device :: d_d(DOM(NX, NY, NZ))
11
12     type(dim3) :: cugrid, cublock
13     integer(4) :: cuerror
14     a_d(:,:,:) = a(:,:,:)
15     c_d(:,:,:) = 0
16     b_d(:,:,:) = b(:,:,:)
17     d_d(:,:,:) = 0
18
19     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
20     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
21     call add <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:))
           , c_d(AT(:,:,:)))
22     cuerror = cudaThreadSynchronize()
23     if(cuerror .NE. cudaSuccess) then
24       write(6, *) 'CUDA error: ', cudaGetErrorString(cuerror)
25       stop 1
26     end if
27
28     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
29     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
30     call mult <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:)
           ), d_d(AT(:,:,:)))
31     cuerror = cudaThreadSynchronize()
32     if(cuerror .NE. cudaSuccess) then
33       write(6, *) 'CUDA error: ', cudaGetErrorString(cuerror)
34       stop 1
35     end if
36
37     c(:,:,:) = c_d(:,:,:)
38     d(:,:,:) = d_d(:,:,:)
39   end subroutine

41   attributes(global) subroutine add(a, b, c)
42     use cudafor
43     real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
           NZ))
44     real, intent(out) ,device :: c(DOM(NX, NY, NZ))
45     integer :: z
46     integer(4) :: x, y
47
48     x = (blockidx%x - 1) * blockDim%x + threadidx%x
49     y = (blockidx%y - 1) * blockDim%y + threadidx%y
50     do z=1,NZ
51       c(AT(x,y,z)) = a(AT(x,y,z)) + b(AT(x,y,z))
52     end do
53   end subroutine
```

Figure 4.5: Class member functions of "FortranImplementation" classes (Example shown with CUDAFortranImplementation).

4.6 Hybrid Fortran Parser

In order to interpret the directives introduced in cha 3 in the right context, it was necessary to create the parser program outlined in this section. This parser is used by the `annotatedCallGraphFromH90SourceDir` and `generateF90fromH90AndAnalyzedCallGraph.py` python scripts (as described in sec. 4.2) through subclasses.

This section gives a more detailed view of that parser. Figure 4.6 shows the state machine pattern that has been used for the parser implementation.

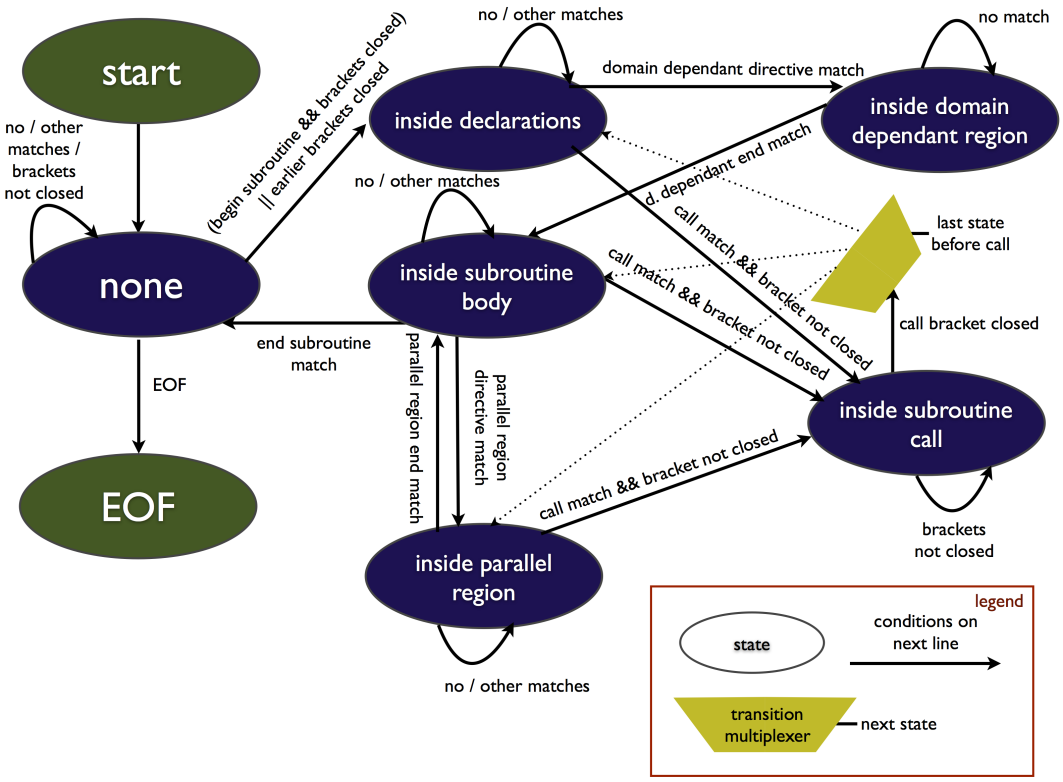


Figure 4.6: H90 Parser State Machine.

The state machine design pattern being used here resembles that of a Mealy machine. However, two changes have been applied to the Mealy machine properties:

1. The output is being detached from the machine. In other words the `H90Parser` class does not produce any output itself. Its subclasses (as described in sec. 4.4) are responsible for that task. This allows large parts of the parser code to be reused for both python programs dealing with h90 source files as described in sec. 4.2

2. A multiplexer is introduced as an additional element in order to reduce the number of states (which matches the way code is being reused in the actual implementation).

Chapter 5

Usability and Performance Validation

In this chapter the performance and usability of the **Hybrid Fortran** framework will be verified. For this reason, a sample implementation with a subset of the ASUCA physical core's functionality has been implemented. Sec. 5.1 will give an idea of the scope of this implementation. Sec. 5.2 will examine the usability of **Hybrid Fortran** compared to OpenACC when applied to the ASUCA physical process. Sec. 5.3 will offer an examination of the performance of the current CPU and GPU implementations that are being applied with **Hybrid Fortran** in comparison to the performance of OpenACC as well as code that has been optimized specifically for the CPU.

5.1 Scope of Sample ASUCA Implementation

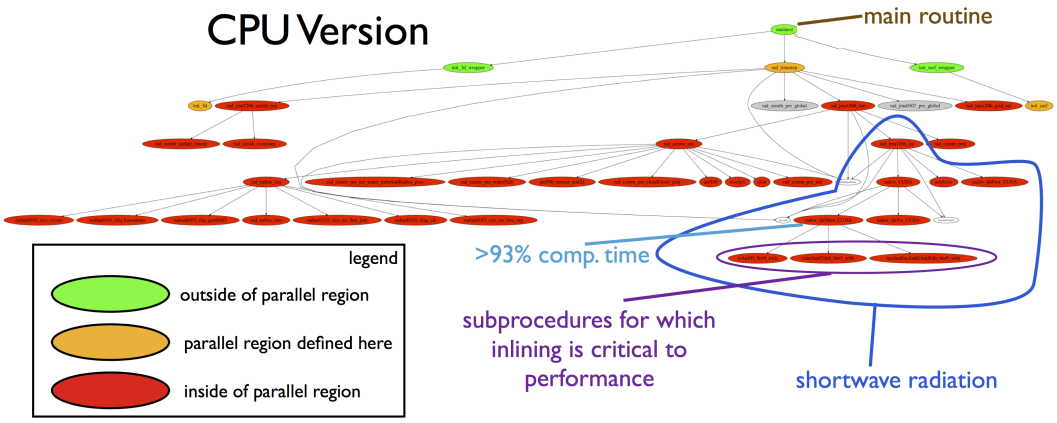


Figure 5.1: Overview CPU version of ASUCA sample implementation.

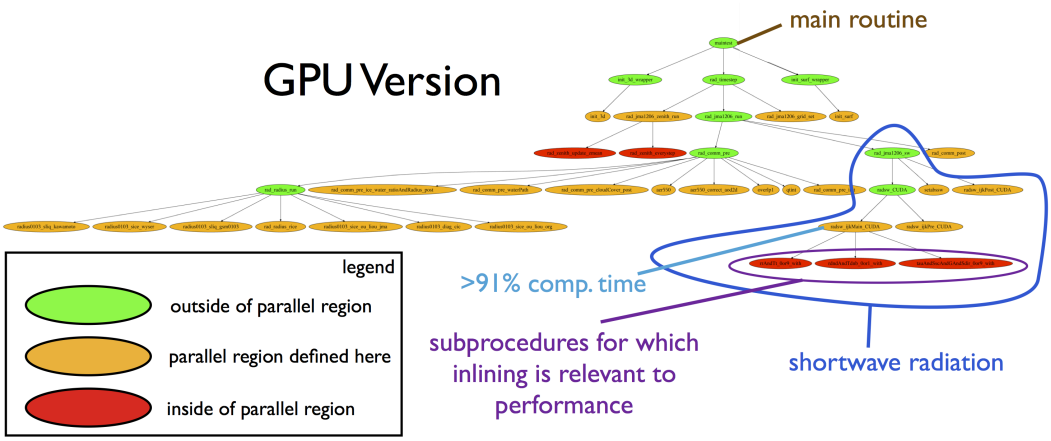


Figure 5.2: Overview GPU version of ASUCA sample implementation.

Fig. 5.1 and fig. 5.1 show the scope of the sample ASUCA implementation[1]. Essentially all routines necessary for the ASUCA radiation module have been implemented with the exception of longwave radiation. These figures allow the following observations:

1. The CPU "sees" three loops over the `IJ` domains (orange nodes) - two for initialisations and one for the actual timestep computation.
2. The GPU on the other hand sees 23 loops over the `IJ` domains, i.e. 23 CUDA kernels are being created.
3. One kernel, which is part of the shortwave radiation module, is responsible for over 93% of this implementation's computation time when executed on CPU and over 91% when executed on GPU.
4. The extent of the shortwave radiation module is indicated by the blue marking. This module has also been benchmarked using OpenACC as well as manual CUDA Fortran, as discussed in cha. 2, hence its performance will be analyzed more closely in sec. 5.3).
5. The violet marking shows three additional scalar subroutines (compared to the original code version) that have been introduced as a GPU optimization. The performance impact of their inlining will also be discussed in sec. 5.3.

[1]The automatical graphical callgraph representation has been used here. This can be created by running "make graphs" in the project directory.

5.2 Usability of Hybrid Fortran versus PGI OpenACC

In order to examine the usability differences between **Hybrid Fortran** and PGI OpenACC[2] we will examine the code changes necessary for the portation of an example subroutine. The `abssw` subroutine from the shortwave submodule has been chosen with the following criteria:

1. The data access patterns as well as the amount of computations per data array is representative for the ASUCA physical processes.
2. The subroutine is large enough to show typical code complexity, yet is small enough to be shown here.

Please note, however, that this subroutine's execution time is a small fraction of the radiation module's overall execution time.

We define two classes of source code modifications as follows:

Green Code Modification :

1. New code lines.
2. Code edits using a trivial find/replace operation over the entire routine (no regular expressions are necessary).

Yellow Code Modification :

1. Code edits inside existing code that cannot be implemented in one simple find/replace operation, i.e. there are dependencies with the existing code present.

The following listings are used to show the usability comparison:

- Lst. 5.1 shows the original `abssw` subroutine code for comparison.
- Lst. 5.2 shows the OpenACC version of the `abssw` subroutine with the necessary edits marked according to the definition above.
- Lst. 5.3 shows the **Hybrid Fortran** version of the `abssw` subroutine, the edits marked in the same way.

[2]Because of difficult usability in the tested version of HMPP OpenACC and given time constraints of this thesis, it has been decided not to pursue a HMPP implementation for the shortwave submodule. See also sec. 2.6.1.

5.2.1 Original CPU Optimized Version of Example Subroutine

Lst. 5.1 shows the original CPU optimized code version of the example subroutine.

```
1  subroutine setabssw(czeta, pmlv, gdp, qlev, qmlv, ozvt, &
2    & sh2o, so3, tco2, to2n, to2s)
3    use pp_vardef
4    use pp_phys_const, only: grav
5    use rad_grid, only: kmax, kmp1
6    use rad_const, only: eps, eps_2
7    use rad_bnd_tbl, only: sblco2, sblo2n, sblo2s
8    implicit none
9
10   real(8), intent(in) :: czeta
11   real(8), intent(in) :: pmlv(kmp1)
12   real(8), intent(in) :: gdp(kmax)
13   real(8), intent(in) :: qlev(kmax)
14   real(8), intent(in) :: qmlv(kmp1)
15   real(8), intent(in) :: ozvt(kmax) ! amount of ozone (cm-STP)
16   real(8), intent(out) :: sh2o(kmp1) ! amount of water vapor (g/cm
         **2)
17   real(8), intent(out) :: so3(kmp1) !amount of ozone (g/cm**2)
18   real(8), intent(out) :: tco2(kmax) ! effective optical depth of
         CO2
19   real(8), intent(out) :: to2n(kmax) ! effective optical depth of
         O2 (NIR)
20   real(8), intent(out) :: to2s(kmax) ! effective optical depth of
         O2 (SR)
21
22   !===== local variables =====
23   integer(4) :: k
24   integer(4) :: ip
25   integer(4) :: iz
26   real(8) :: wrk
27   real(8) :: wktr(kmp1)
28   real(8) :: pp
29   real(8) :: dp
30   real(8) :: zz
31   real(8) :: dz
32   real(8) :: tr00
33   real(8) :: tr01
34   real(8) :: wcmu
35   real(8), parameter :: eps_t = 1.d0 + eps_2
36   real(8), parameter :: pzx1 = 81.d0 - eps_2
37   real(8), parameter :: rho3stp = 2.142d-3
38
39   integer(4), save :: initial = 1
40   real(8), save :: gi05
41
42   if (initial == 1) then
43     gi05 = 5.d0 / grav
44     initial = 0
45   end if
```

```
46
47    wktr(kmp1) = 1.d0  ! work for co2 abs. in NIR region
48    sh2o(kmp1) = 1.d0  ! work for  o2 abs. in NIR
49    so3(kmp1) = 1.d0   ! work for  o2 abs. in S-R band
50
51    if (cmu <= 0.d0) then
52      wcmu = eps
53    else
54      wcmu = czeta
55    end if
56    wrk = 81.d0 + 40.d0 * log10(wcmu)
57
58    do k = 1, kmax
59      pp = 21.d0 + 20.d0 * log10(pmlv(k))
60      if (pp < eps_t) pp = eps_t
61      if (pp > pzx1) pp = pzx1
62      ip = int(pp)
63      dp = pp - ip
64      zz = wrk
65      if (zz < eps_t) zz = eps_t
66      if (zz > pzx1) zz = pzx1
67      iz = int(zz)
68      dz = zz - iz
69      !(co2,NIR)
70      tr00 = (1.d0 - dz) * sblco2(ip, iz) + dz * sblco2(ip, iz + 1)
71      tr01 = (1.d0 - dz) * sblco2(ip + 1, iz) + dz * sblco2(ip + 1,
            iz + 1)
72      wktr(k) = (1.d0 - dp) * tr00 + dp * tr01
73      !(o2,NIR)
74      tr00 = (1.d0 - dz) * sblo2n(ip, iz) + dz * sblo2n(ip, iz + 1)
75      tr01 = (1.d0 - dz) * sblo2n(ip + 1, iz) + dz * sblo2n(ip + 1,
            iz + 1)
76      sh2o(k) = (1.d0 - dp) * tr00 + dp * tr01
77      !(o2,Schuman-Runge band)
78      tr00 = (1.d0 - dz) * sblo2s(ip, iz) + dz * sblo2s(ip, iz + 1)
79      tr01 = (1.d0 - dz) * sblo2s(ip + 1, iz) + dz * sblo2s(ip + 1,
            iz + 1)
80      so3(k) = (1.d0 - dp) * tr00 + dp * tr01
81    end do
82
83    !((CO2 and O2 ;optical depth))
84    do k = 1, kmax
85      tco2(k) = - czeta * log(wktr(k) / wktr(k + 1))
86      to2n(k) = - czeta * log(sh2o(k) / sh2o(k + 1))
87      to2s(k) = - czeta * log(so3(k) / so3(k + 1))
88    end do
89
90    !((O3 ,unscaled))
91    do k = 1, kmax
92      so3(k) = rho3stp * ozvt(k)
93    end do
94
95    !((H2O ,unscaled))
96    do k = 1, kmax
```

```
97      sh2o(k) = gi05 * gdp(k) * (qlev(k) + 0.5d0 * (qmlv(k) + &
98      & qmlv(k + 1)))
99    end do
100
101   return
102 end subroutine setabssw
```

Listing 5.1: Example ASUCA subroutine (original CPU optimized version).

5.2.2 OpenACC Version of Example Subroutine

Lst. 5.2 shows the OpenACC code version of the example subroutine.

Notes:

1. Since GPU implementation requires the parallelizable loops to be close to the computations, the `IJ` loops have been introduced here. In the original code version these loops are implemented in the `main` subroutine.

2. Many small edits need to be introduced for array declarations and accesses with dependencies in the `IJ` domain. These edits have been classified as "yellow", since they require a certain amount of care. Experience has shown that the safest way to implement these changes is a find/replace operation using regular expressions or a series of simple find/replace operations for every `IJ` dependant array for every `k` access pattern (i.e. `k`, `k-1`, `k+1`).

3. `DOM` and `AT` are preprocessor macros that define the storage order, i.e. the output of these macros equals the input strings, comma separated and reordered according to the definitions in `storage_order.F90`.

```
1  subroutine setabssw(nx, ny, czeta, pmlv, gdp, qlev, qmlv, ozvt, &
2    & sh2o, so3, tco2, to2n, to2s)
3    use pp_vardef
4    use pp_phys_const, only: grav
5    use rad_grid, only: kmax, kmp1
6    use rad_const, only: eps, eps_2
7    use rad_bnd_tbl, only: sblco2, sblo2n, sblo2s
8    implicit none
9
10   integer(4), intent(in) :: nx, ny
11
12   real(8), intent(in) :: czeta(nx, ny)
13   real(8), intent(in) :: pmlv(DOM(nx, ny, kmp1))
14   real(8), intent(in) :: gdp(DOM(nx, ny, kmax))
15   real(8), intent(in) :: qlev(kmax)
16   real(8), intent(in) :: qmlv(kmp1)
17   real(8), intent(in) :: ozvt(kmax) ! amount of ozone (cm-STP)
18   real(8), intent(out) :: sh2o(DOM(nx, ny, kmp1)) ! amount of water
         vapor (g/cm**2)
```

```
19   real(8), intent(out) :: so3(DOM(nx, ny, kmp1)) ! amount of ozone
         (g/cm**2)
20   real(8), intent(out) :: tco2(DOM(nx, ny, kmax)) ! effective
         optical depth of CO2
21   real(8), intent(out) :: to2n(DOM(nx, ny, kmax)) ! effective
         optical depth of O2 (NIR)
22   real(8), intent(out) :: to2s(DOM(nx, ny, kmax)) ! effective
         optical depth of O2 (SR)
23
24   !===== local variables =====
25   integer(4) :: i, j
26   integer(4) :: k
27   integer(4) :: ip
28   integer(4) :: iz
29   real(8) :: wrk
30   real(8) :: wktr(DOM(nx, ny, kmp1))
31   real(8) :: pp
32   real(8) :: dp
33   real(8) :: zz
34   real(8) :: dz
35   real(8) :: tr00
36   real(8) :: tr01
37   real(8) :: wcmu
38   real(8), parameter :: eps_t = 1.d0 + eps_2
39   real(8), parameter :: pzx1 = 81.d0 - eps_2
40   real(8), parameter :: rho3stp = 2.142d-3
41
42   !Note: Initialisation of static variables has been moved to init
         phase.
43
44   !===== implementation =====
45   !$acc data &
46   !$acc    copyin(sblco2, sblo2n, sblo2s) &
47   !$acc    copyin(pmlv, qlev, qmlv, ozvt) &
48   !$acc    present(sh2o, so3, tco2, to2n, to2s, czeta, gdp) &
49   !$acc    create(wktr)
50   !$acc kernels
51   !$acc loop
52   do j = 1, ny
53     !$acc loop
54     do i = 1, nx
55       wktr(AT(i,j,kmp1)) = 1.d0  ! work for co2 abs. in NIR region
56       sh2o(AT(i,j,kmp1)) = 1.d0  ! work for  o2 abs. in NIR
57       so3(AT(i,j,kmp1)) = 1.d0   ! work for  o2 abs. in S-R band
58
59       if (czeta(i, j) <= 0.d0) then
60         wcmu = 1.d-6
61       else
62         wcmu = czeta(i, j)
63       end if
64       wrk = 81.d0 + 40.d0 * log10(wcmu)
65
66       do k = 1, kmax
67         pp = 21.d0 + 20.d0 * log10(pmlv(AT(i,j,k)))
```

```
68          if (pp < eps_t) pp = eps_t
69          if (pp > pzx1) pp = pzx1
70          ip = int(pp)
71          dp = pp - ip
72          zz = wrk
73          if (zz < eps_t) zz = eps_t
74          if (zz > pzx1) zz = pzx1
75          iz = int(zz)
76          dz = zz - iz
77          !(co2,NIR)
78          tr00 = (1.d0 - dz) * sblco2(ip, iz) + dz * sblco2(ip, iz +
                1)
79          tr01 = (1.d0 - dz) * sblco2(ip + 1, iz) + dz * sblco2(ip +
                1, iz + 1)
80          wktr(AT(i,j,k)) = (1.d0 - dp) * tr00 + dp * tr01
81          !(o2,NIR)
82          tr00 = (1.d0 - dz) * sblo2n(ip, iz) + dz * sblo2n(ip, iz +
                1)
83          tr01 = (1.d0 - dz) * sblo2n(ip + 1, iz) + dz * sblo2n(ip +
                1, iz + 1)
84          sh2o(AT(i,j,k)) = (1.d0 - dp) * tr00 + dp * tr01
85          !(o2,Schuman-Runge band)
86          tr00 = (1.d0 - dz) * sblo2s(ip, iz) + dz * sblo2s(ip, iz +
                1)
87          tr01 = (1.d0 - dz) * sblo2s(ip + 1, iz) + dz * sblo2s(ip +
                1, iz + 1)
88          so3(AT(i,j,k)) = (1.d0 - dp) * tr00 + dp * tr01
89        end do
90
91        !((CO2 and O2 ;optical depth))
92        do k = 1, kmax
93          tco2(AT(i,j,k)) = - czeta(i, j) * log(wktr(AT(i,j,k)) /
                wktr(AT(i,j,k + 1)))
94          to2n(AT(i,j,k)) = - czeta(i, j) * log(sh2o(AT(i,j,k)) /
                sh2o(AT(i,j,k + 1)))
95          to2s(AT(i,j,k)) = - czeta(i, j) * log(so3(AT(i,j,k)) / so3(
                AT(i,j,k + 1)))
96        end do
97
98        !((O3 ,unscaled))
99        do k = 1, kmax
100         so3(AT(i,j,k)) = rho3stp * ozvt(k)
101       end do
102
103       !((H2O ,unscaled))
104       do k = 1, kmax
105 sh2o(AT(i,j,k)) = gi05 * gdp(AT(i,j,k)) * (qlev(k) + 0.5d0 * (
      qmlv(k) + &
106         & qmlv(k + 1)))
107       end do
108     end do
109   end do
110   !$acc end kernels
111   !$acc end data
```

```
112
113   return
114 end subroutine setabssw
```

Listing 5.2: Example ASUCA kernel subroutine in OpenACC

5.2.3 Hybrid Fortran Version of Example Subroutine

Lst. 5.3 shows the **Hybrid Fortran** code version of the example subroutine.

Notes:

1. Like in the OpenACC version, the `IJ` loops are introduced here, however in an abstracted way such that they are only applied to the GPU implementation, using the `@parallelRegion{appliesTo(GPU),..}` directive.
2. The small "yellow" edits that need to be done manually in the OpenACC version are applied automatically by the framework here.
3. Compared to the OpenACC version, a higher number of "green" edits are required here. 14 of these edits can be performed using only two find-/replace operations (`kmax` to `KMAX_CONST` and `kmp1` to `KMP1_CONST`). The reason for these changes is that external module symbols cannot be accessed directly in kernel subroutines with **Hybrid Fortran** in its current state. They would have to be passed as input parameters. In this case it was chosen to use preprocessor constants instead in order to potentially save registers on the GPU.
4. The only place where "yellow" edits are still needed is in the input parameter definition at the beginning of the subroutine. In other words, the number of "yellow" edits is constant here, while in the OpenACC implementation this grows with the size of the subroutine.
5. The application of the storage order macros `DOM` and `AT` has been abstracted by passing them to the `@domainDependant` directives.

```
1  subroutine setabssw(nx, ny, sblco2, sblo2n, sblo2s, czeta, pmlv,
       gdp, qlev, qmlv, ozvt, &
2    & sh2o, so3, tco2, to2n, to2s)
3    use pp_vardef
4    implicit none
5    integer(4), intent(in) :: nx, ny
6    real(8), intent(in), dimension(81, 81) :: sblco2, &
7      & sblo2n, sblo2s
8    real(8), intent(in) :: czeta
9    real(8), intent(in) :: pmlv(KMP1_CONST)
10   real(8), intent(in) :: gdp(KMAX_CONST)
11   real(8), intent(in) :: qlev(KMAX_CONST)
```

```
12  real(8), intent(in) :: qmlv(KMP1_CONST)
13  real(8), intent(in) :: ozvt(KMAX_CONST) ! amount of ozone (cm-STP
        )
14  real(8), intent(out) :: sh2o(KMP1_CONST) ! amount of water vapor
        (g/cm**2)
15  real(8), intent(out) :: so3 (KMP1_CONST) ! amount of ozone (g/cm
        **2)
16  real(8), intent(out) :: tco2(KMAX_CONST) ! effective optical
        depth of CO2
17  real(8), intent(out) :: to2n(KMAX_CONST) ! effective optical
        depth of O2 (NIR)
18  real(8), intent(out) :: to2s(KMAX_CONST) ! effective optical
        depth of O2 (SR)
19
20  !===== local variables =====
21  integer(4) :: k
22  integer(4) :: ip
23  integer(4) :: iz
24  real(8) :: wrk
25  real(8) :: pp
26  real(8) :: dp
27  real(8) :: zz
28  real(8) :: dz
29  real(8) :: tr00
30  real(8) :: tr01
31  real(8) :: wcmu
32  real(8), parameter :: eps_t = 1.d0 + eps_2
33  real(8), parameter :: pzx1 = 81.d0 - eps_2
34  real(8), parameter :: rho3stp = 2.142d-3
35
36  @domainDependant {}
37  nx, ny
38  @end domainDependant
39
40  @domainDependant {declarationPrefix(real(kind = r_size))}
41  grav, eps, eps_2, gi05
42  @end domainDependant
43
44  @domainDependant {domName(i, j), domSize(nx, ny), attribute(
        autoDom)}
45  czeta
46  gdp, qlev, ozvt, tco2, to2n, to2s
47  pmlv, qmlv, sh2o, so3, wktr
48  sblco2, sblo2n, sblo2s
49  @end domainDependant
50
51  !===== implementation =====
52  @parallelRegion{appliesTo(GPU), domName(i, j), domSize(nx, ny)}
53
54  !Note: Initialisation of static variables has been moved to init
        phase.
55
56  wktr(KMP1_CONST) = 1.d0  ! work for co2 abs. in NIR region
57  sh2o(KMP1_CONST) = 1.d0  ! work for  o2 abs. in NIR
```

```
58   so3(KMP1_CONST) = 1.d0   ! work for  o2 abs. in S-R band
59
60   if (czeta <= 0.d0) then
61     wcmu = 1.d-6
62   else
63     wcmu = czeta
64   end if
65   wrk = 81.d0 + 40.d0 * log10(wcmu)
66
67   do k = 1, KMAX_CONST
68     pp = 21.d0 + 20.d0 * log10(pmlv(k))
69     if (pp < eps_t) pp = eps_t
70     if (pp > pzx1) pp = pzx1
71     ip = int(pp)
72     dp = pp - ip
73     zz = wrk
74     if (zz < eps_t) zz = eps_t
75     if (zz > pzx1) zz = pzx1
76     iz = int(zz)
77     dz = zz - iz
78     !(co2,NIR)
79     tr00 = (1.d0 - dz) * sblco2(ip, iz) + dz * sblco2(ip, iz + 1)
80     tr01 = (1.d0 - dz) * sblco2(ip + 1, iz) + dz * sblco2(ip + 1,
           iz + 1)
81     wktr(k) = (1.d0 - dp) * tr00 + dp * tr01
82     !(o2,NIR)
83     tr00 = (1.d0 - dz) * sblo2n(ip, iz) + dz * sblo2n(ip, iz + 1)
84     tr01 = (1.d0 - dz) * sblo2n(ip + 1, iz) + dz * sblo2n(ip + 1,
           iz + 1)
85     sh2o(k) = (1.d0 - dp) * tr00 + dp * tr01
86     !(o2,Schuman-Runge band)
87     tr00 = (1.d0 - dz) * sblo2s(ip, iz) + dz * sblo2s(ip, iz + 1)
88     tr01 = (1.d0 - dz) * sblo2s(ip + 1, iz) + dz * sblo2s(ip + 1,
           iz + 1)
89     so3(k) = (1.d0 - dp) * tr00 + dp * tr01
90   end do
91
92   !((CO2 and O2 ;optical depth))
93   do k = 1, KMAX_CONST
94     tco2(k) = - czeta * log(wktr(k) / wktr(k + 1))
95     to2n(k) = - czeta * log(sh2o(k) / sh2o(k + 1))
96     to2s(k) = - czeta * log(so3(k) / so3(k + 1))
97   end do
98
99   !((O3 ,unscaled))
100  do k = 1, KMAX_CONST
101    so3(k) = rho3stp * ozvt(k)
102  end do
103
104  !((H2O ,unscaled))
105  do k = 1, KMAX_CONST
106    sh2o(k) = gi05 * gdp(k) * (qlev(k) + 0.5d0 * (qmlv(k) + &
107    & qmlv(k + 1)))
108  end do
```

```
109   @end parallelRegion
110
111   return
112 end subroutine setabssw
```

Listing 5.3: Example ASUCA kernel subroutine in Hybrid Fortran

5.2.4 Usability Results

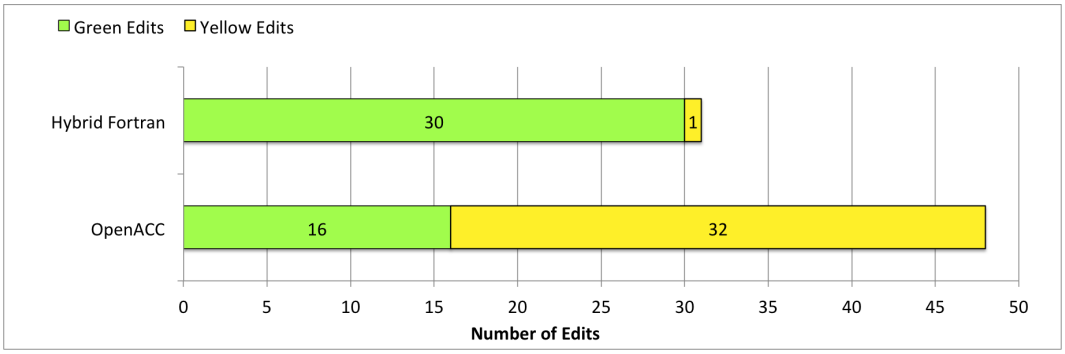


Figure 5.3: Class and number of edits compared to original CPU code for "setabssw" kernel.

Fig. 5.3 shows the results for the usability examination of sec. 5.2. Even though counting two find/replace operations as 14 edits, the **Hybrid Fortran** version still has a lower total number of edits than the OpenACC version. We can also see that the majority of code modifications are of the non problematic "green" kind in the **Hybrid Fortran** case, while in the OpenACC version the majority of edits need to be applied more carefully.

We conclude that **Hybrid Fortran** is more easily applicable to the ASUCA physical core and it bears less potential for errors than OpenACC - an important aspect to keep in mind when doing GPU portations, since debugging on the GPU is not as straight forward as on the CPU, i.e. debugging for GPU has a higher cost in developer time.

5.3 Performance of Hybrid Fortran

In this section the performance of **Hybrid Fortran** implemented code will be examined. The shortwave radiation module has been reused for that purpose, since benchmark implementations were already done using pure CUDA Fortran as well as OpenACC (see sec. 2.5.3). In sec. 5.3.1 we will examine the CPU performance, while sec. 5.3.2 will show the GPU performance. All performance measurements have been repeated five times, the results shown here are averaged over those five runs. Again, only computation time is shown here, no host-to-device data copy time is being evaluated since the host-to-device bus performance is not expected to be relevant for the ASUCA GPU implementation.

5.3.1 CPU Performance Comparison for Shortwave Radiation

In this section we compare the CPU performance of the following implementations of the shortwave radiation (see also sec. 2.7.4):

1. Original CPU optimized code.
2. OpenACC implementation compiled for CPU execution.
3. Preprocessed CUDA Fortran implementation compiled for CPU execution.
4. **Hybrid Fortran** implementation compiled for CPU execution.

These implementations have been compiled using `ifort -fast -openmp`[3]. The OpenACC CPU implementation has additionally been compiled using `pgf90`, since it has been found to perform poorly using the OpenACC agnostic `ifort` compiler. The `pgf90` compiler has for that reason been used with the following parameters[4]:

-Mconcur for enabling OpenMP multicore support.

-Mvect=sse for enabling SSE vectorization support.

-Minline=levels:5,reshape for enabling in-module-inlining for up to 5 levels including array parameter reshaping.

-O4 for enabling the most aggressive compiler optimizations.

[3]The Intel compiler tends to be very aggressive using the "fast" setting. This already enables the highest optimization settings as well as SSE vectorization. In general this has been found to perform well on the Intel hardware architecture used on TSUBAME 2.0.

[4]These settings have been carefully tested to perform well. Compiling the test module simply using "pgf90 -fast -O4" performs around 30% worse than the settings shown here.

-r8 for treating `real` variables using double precision by default.

A few notes on the test settings:

1. Concerning the multicore measurements: Since the parallel loops over `IJ` are placed outside the radiation module for the original code version as well as the **Hybrid Fortran** CPU implementation, the execution time measurement for one submodule becomes non trivial in that case (thread synchronization would have to be used for the counters, which could have negative impact on the execution time itself). For that reason we have used the following approximation to measure the execution time of the shortwave radiation module in that case:
   (a) The ratio of shortwave radiation module execution time versus the overall test program execution time has been measured for all domain sizes in single core execution. These ratios have been found to be between 92% and 95% for all domain sizes for all test versions.
   (b) These ratios have then been applied to the overall execution time in case of multicore execution.
2. `KIJ` storage order has been used for CPU execution for all test cases.
3. Double precision arithmetics have been used for the computations.

Fig. 5.4 shows the single core execution time results on CPU. The following performance characteristics can be examined from these results:

1. As expected the execution time is proportional to the area of the `IJ` domain.
2. The shortwave radiation module performance has been found to be heavily dependant on the inlining method being used. As a GPU optimization some scalar subroutines have been introduced (see fig. 5.1 in sec. 5.1) for the **Hybrid Fortran** version. Trying to inline these with the `ifort` compiler has proven to be difficult, hence a manually inlined version was implemented as well, essentially recreating the same code as in the original version. Doing so lead to an improvement of factor 1.8 in CPU performance, giving the **Hybrid Fortran** version almost the same performance characteristics as the original CPU version.
3. Curiously, `ifort` compiled OpenACC code performs more than 200% worse than the `pgf90` compiled version. One could suspect that `ifort` is more dependant on the loop structure being optimized for CPU. In all preliminary tests for the original code version as well as **Hybrid Fortran** CPU implementations, `ifort` has performed at least 30% better than `pgf90`. For the following tests we therefore concentrated on `pgf90` for the OpenACC CPU implementations while keeping `ifort` for all other CPU implementations.

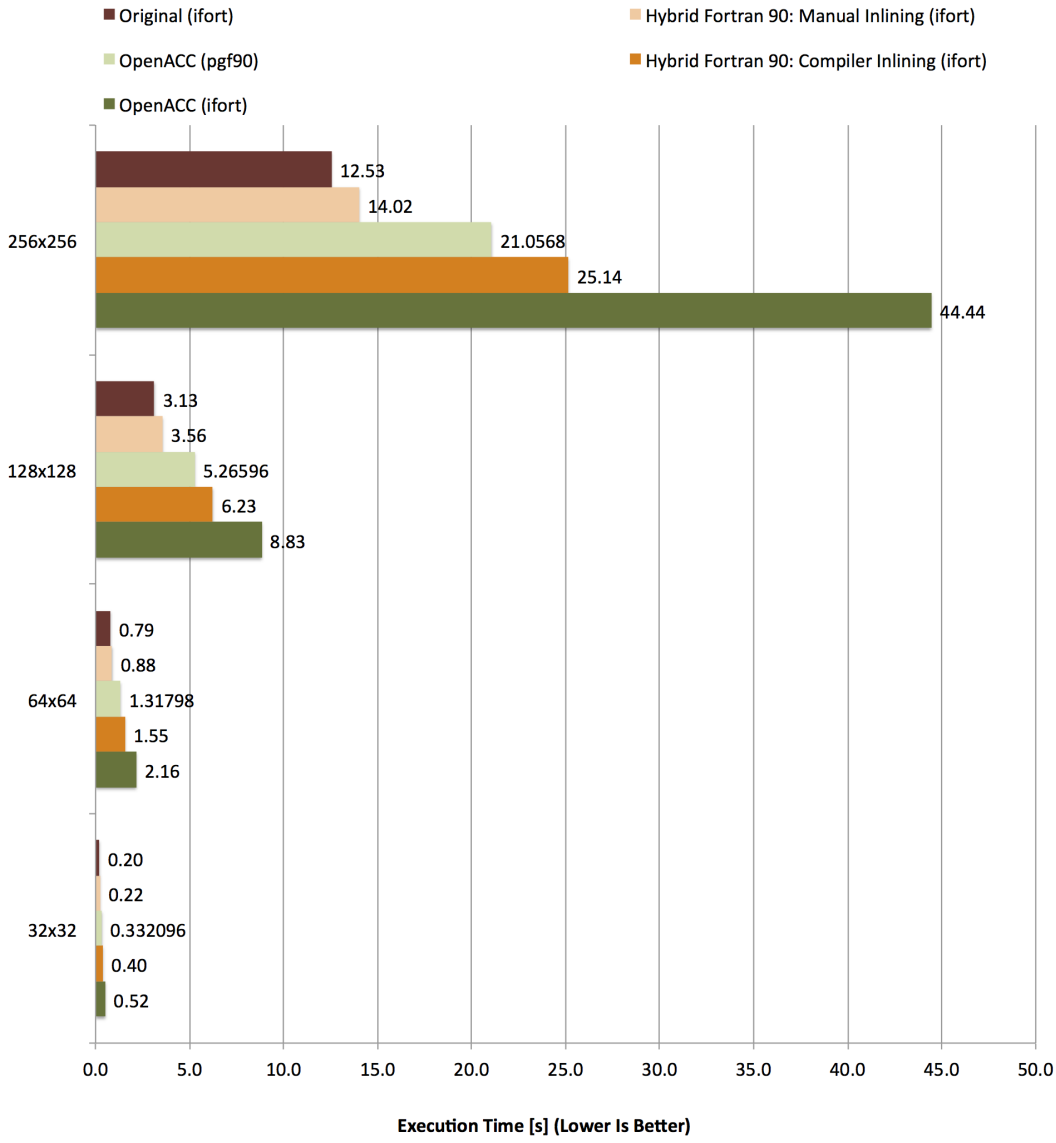


Figure 5.4: Single core CPU Execution time results for the "radsw" subprocedure grouped by grid sizes.

4. The OpenACC version has also continually been optimized, both through compiler settings and preprocessor directives for the reordering of the inner loops of the `radsw` subroutine (i.e. the order of the spectral loop and the parallel `IJ` loops has been optimized for CPU as well as GPU). Also, the OpenACC version is already fully inlined, which has been found to be optimal for CPU execution. Nevertheless we were unable to come close to the original CPU execution times using the OpenACC program structure. This was expected, however, because of the loop structure target conflict explained in sec. 2.5.3.

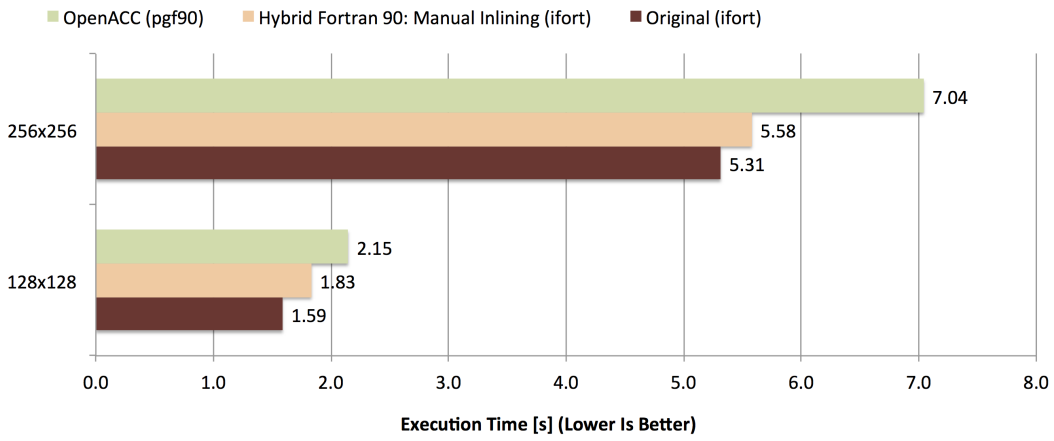


Figure 5.5: Six core CPU Execution time results for the "radsw" subprocedure grouped by grid sizes.

Fig. 5.5 shows the six core execution time results on CPU. For the multicore tests we have used the best performing version of both **Hybrid Fortran** and OpenACC. In case of OpenACC, the OpenMP directives for multicore execution have been added at the same place as the OpenACC accelerator directives (which are omitted for CPU compilation). We can see that this leads to very similar results as the single core case shown before - however the differences are less pronounced. This is to be expected since the shortwave radiation submodule is less computationally bound on multicore execution in comparison to single core execution, which hides computational inefficiencies (see also sec. 2.5.3). The performance loss graphs in fig. 5.6 and fig. 5.7 also show this characteristic.

Overall it becomes clear that complex OpenACC code portations can at best be expected to loose between 30% and 60% when executed on CPU, even after optimizations. The **Hybrid Fortran** version that has been automatically adjusted for the CPU, remarkably only shows 5% loss at best, 16% at the most. In other words, using **Hybrid Fortran** instead of OpenACC has been found to result in an improvement from 26% up to 50% in CPU execution time - a remarkable achievement.

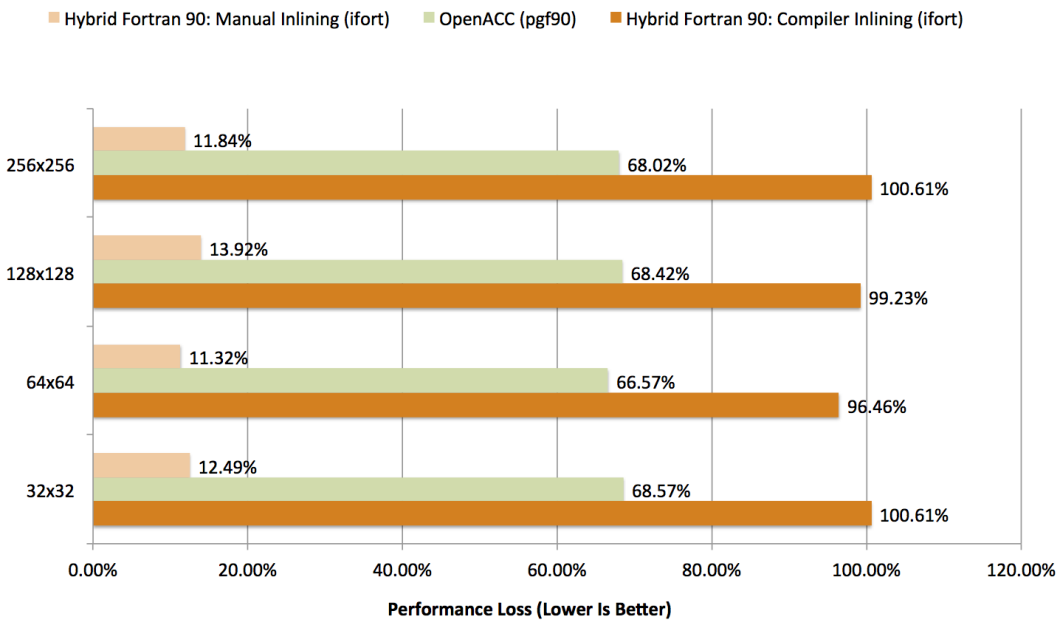


Figure 5.6: Performance loss for the "radsw" kernel on single core CPU (compared to original CPU code compiled with "ifort") grouped by grid sizes.

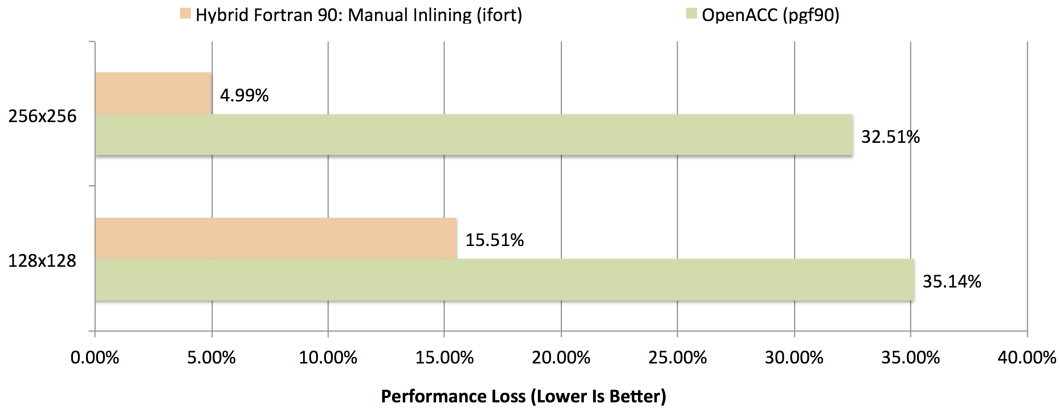


Figure 5.7: Performance loss for the "radsw" kernel on six core CPU (compared to original CPU code compiled with "ifort") grouped by grid sizes.

5.3.2 GPU Performance Comparison for Shortwave Radiation

In terms of GPU performance we compare the **Hybrid Fortran** version with the PGI OpenACC version as well as a preliminarily implemented CUDA Fortran version referred to as "Preprocessed CUDA Fortran", see also sec. 2.7.4.

Some notes on the settings:

1. `IJK` storage order has been used throughout here, which is optimal for GPU execution.
2. The same compiler settings as introduced in sec. 2.2.1 have been used for the OpenACC version.
3. The same compiler settings as introduced in sec. 2.2.3 have been used for the CUDA Fortran and **Hybrid Fortran** GPU versions.
4. As all other tests with shortwave radiation, this comparison has been done using double precision arithmetics.

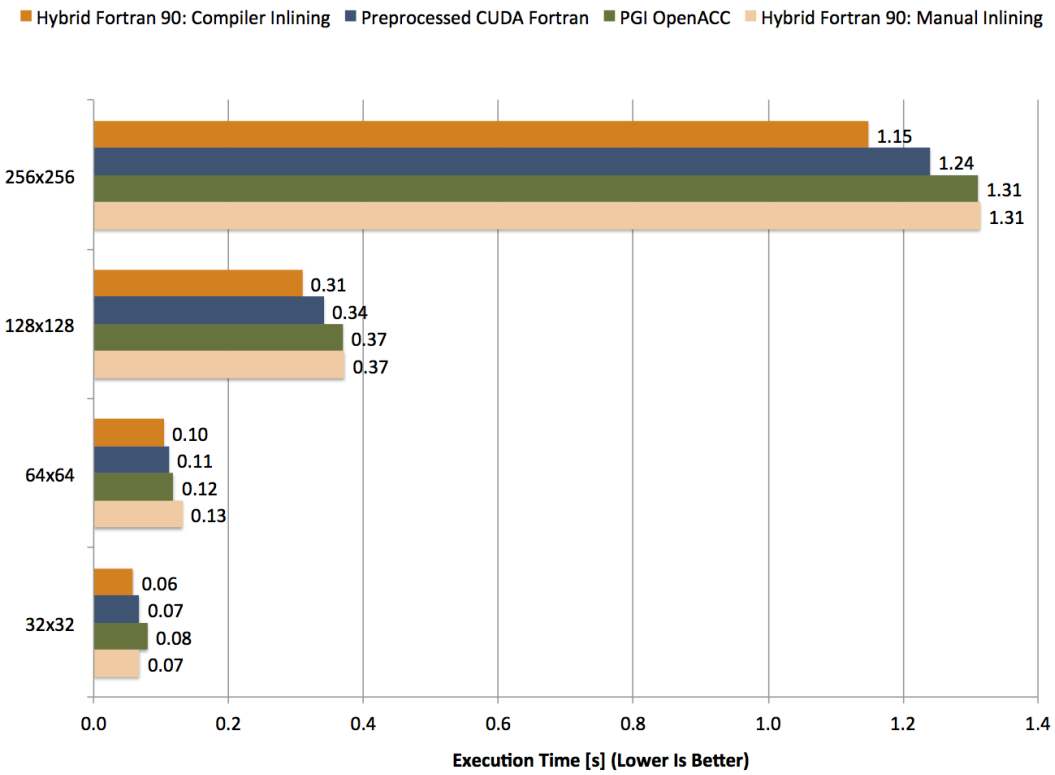


Figure 5.8: GPU Execution time results for the "radsw" kernel grouped by grid sizes.

Fig. 5.8 shows the execution time results on GPU. We can make the following observations:

1. From 32×32 to 128×128 `IJ` domain sizes the execution time scales under proportional to the `IJ` area. This is an expected behaviour on NVIDIA GPUs as their scheduler can make better use of the available bandwidth as well as computational ressources with a higher number of threads. Between 128×128 and 256×256 the performance appears to saturate. Fig. 5.9 shows this behaviour more clearly in terms of speedup.

2. **Hybrid Fortran** with manual inlining shows the same GPU performance as the PGI OpenACC version - an interesting result, since those two versions are very similar in their code structure in terms of loop positioning. This shows that PGI OpenACC performs well on the GPU for bandwidth limited problems (which holds true for shortwave radiation executed on GPU, see also sec. 2.5.3).

3. Using the new scalar subroutines inside the `radsw` kernel and letting them be inlined by `nvcc` (which is at the base of the CUDA Fortran compiler being used), the **Hybrid Fortran** version is able to gain another 14%. We expect different register allocation behaviour on the GPU depending on code inlining to be the reason for this characteristic. We have therefore introduced preprocessor directives to switch between manual inlining and compiler inlining depending on whether the code is compiled for GPU or CPU - we consider this to be still a fair comparison however, since the OpenACC code has been optimized using preprocessor directives as well.

4. Compared to the hand-implemented CUDA Fortran version, the **Hybrid Fortran** gains 8% in speed. We expect a reduction in branches inside the `radsw` kernel to be the reason - this optimization has been found while implementing the **Hybrid Fortran** version thanks to increased familiarity with the codebase.

5. In sec. 2.4.2 a speedup of 11.1× versus single core and 5.3× versus six core CPU execution was estimated for memory bandwidth limited problems. The best results (by the **Hybrid Fortran** GPU version) shown in fig. 5.9 and fig. 5.10 are all within 15% of those values for the saturated case, even within 1.5% in the speedup versus single core execution. This shows that

   (a) The hardware model introduced in sec. 2.4.2 is very reasonable for memory bandwidth limited problems.

   (b) The **Hybrid Fortran** GPU implementation for shortwave radiation is close to the optimum performance that can be achieved, validating the capabilities of this framework.

Overall the GPU performance of the compared approaches is relatively similar in this bandwidth limited case. CUDA Fortran implementations (both implemented by hand and implemented automatically by using the new **Hybrid Fortran** framework) do perform between 5% and 15% better than PGI OpenACC - a similar result has been found for the simple 3D Diffusion test shown in sec. 2.7.2 where PGI OpenACC performed on par with CUDA C as well. For computationally bound problems however we would expect a much more significant advantage for CUDA code versions based on the results of the Particle Push, shown in sec. 2.7.3, where CUDA C was found to perform 2.3× faster than PGI OpenACC.

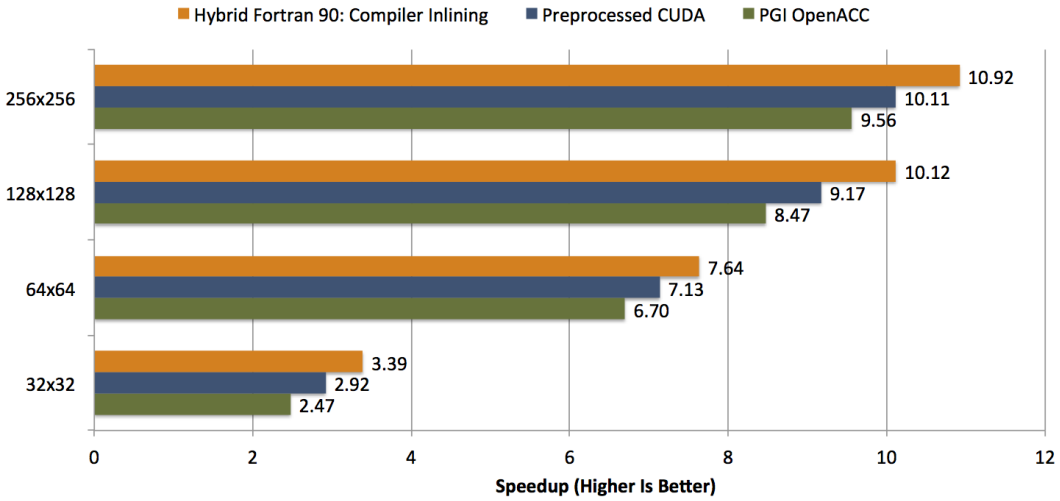


Figure 5.9: GPU Speedup results for the "radsw" kernel compared to single core CPU (ifort) grouped by grid sizes.

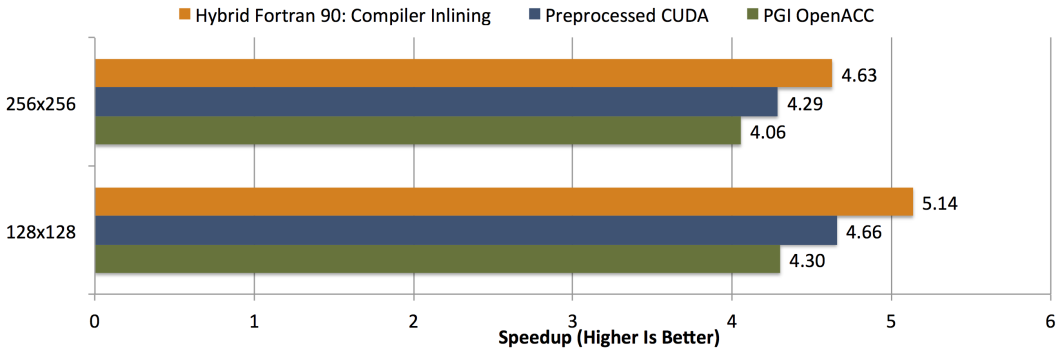


Figure 5.10: GPU Speedup results for the "radsw" kernel compared to six core CPU (ifort) grouped by grid sizes.

Chapter 6

Achievements and Future Work

In this chapter the achievements of this thesis will be laid out (sec. 6.1) and possible future improvements will be listed (sec. 6.2).

6.1 Achievements

By implementing the **Hybrid Fortran** framework as well as validating its usability and performance properties through a number of the ASUCA physical core's submodules, the goals of this thesis, as presented in sec. 1.3, have been achieved. Namely, using this new framework, the physical core of ASUCA can be ported to GPU while

1. keeping a familiar development for the JMA researchers, i.e. Fortran 90.
2. using a well manageable amount of code modifications.
3. keeping the code executable on the CPU.
4. having the potential for optimal GPU performance.
5. keeping the CPU performance on par with the original CPU optimized code base.

We have also shown that **Hybrid Fortran** is the superior choice for the ASUCA physical processes compared to OpenACC in its current stage, in terms of usability (sec. 5.2), performance (sec. 5.3) and flexibility (sec. 3.5).

6.2 Future Work

There are three major areas where **Hybrid Fortran** can be improved in the future:

General Stencil Compatibility While the framework in its current form is suitable for JMA's needs when it comes to the physical core of ASUCA, it will have to be extended for general stencil computation compatibility, i.e. it will have to support stencil accesses with offsets in the parallel domains. For this matter a third directive will have to be introduced in order to define the offsets per array access. As an example, `myArray(i+1,j,k)` will then become `myArray@{i+1,j}(k)` in order to keep the `IJ` domain dependencies abstracted from the user code. For the CPU version, the "vertical"[1] arrays at offset positions could be passed as additional, automatically introduced input parameters, thus enabling automatic optimization for CPU caches through outside loops even with more complex memory access patterns.

Automatic Array Allocation and Parameter Mapping Currently the **Hybrid Fortran** framework shares many code restrictions with CUDA Fortran for the kernel and inside kernel subroutines, most prominently the inability to allocate local arrays. Lifting this restriction by implementing automatic array allocation in the wrapper routine and passing these arrays to the kernel, would reduce the portation cost to **Hybrid Fortran** even further.

Support for More Implementations The flexibility of this framework will become even clearer, when OpenCL is supported as an alternative GPU implementation, thus giving the option to change accelerator hardware vendors. We expect OpenCL support to be achievable with low implementation cost because of the abstracted nature of the implementation, shown in sec. 4.4. In sec. 4.5 a blueprint for such an adaptation has already been layed out.

[1] as in orthogonal to the parallel domains, in this example the "K" dimension.

Bibliography

[1] J. Preshing, "A look back at single-threaded CPU performance." http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance, 2012. [Online; accessed October 4, 2012].

[2] TOP500, "TSUBAME 2.0 ranking." http://i.top500.org/system/10588, 2012. [Online; accessed October 4, 2012].

[3] M. Z. Jacobson, *Fundamentals of atmospheric modeling*. Cambridge University Press, 2005.

[4] K. Kawano, J. Ishida, and C. Muroi, "Development of a new nonhydrostatic model ASUCA at JMA," Talk Held at the 12th Internation Specialist Meeting on The Next Generation Models on Climate Change and Sustainability for Advanced High Performance Computing Facilities, 2010. [http://www.prime-pco.com/climate12/pdf/kohei_kawano.pdf; accessed October 4, 2012].

[5] J. Ishida, C. Muroi, K. Kawano, and Y. Kitamura, "Development of a new nonhydrostatic model ASUCA at JMA," *CAS/JSC WGNE Research Activities in Atmospheric and Oceanic Modelling*, 2010.

[6] M. Baldauf, "COSMO / CLM training course." http://www.clm-community.eu/index.php?menuid=1&downloadid=952&reporeid=256, 2011. [Online; accessed October 4, 2012].

[7] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model ASUCA production code," in *Proceedings of ACM/IEEE 2010 International Conference for High Performance Computing, Networking, Storage and Analysis*, nov. 2010.

[8] T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Muroi, "145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction.," *Procedia CS*, vol. 4, pp. 1535–1544, 2011.

[9] TOP500, "TOP500 list - June 2012." http://www.top500.org/list/2012/06/100, 2012. [Online; accessed October 5, 2012].

[10] NVIDIA, "NVIDIAs next generation CUDA compute architecture: Fermi." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. [Online; accessed June 3, 2011].

[11] J. C. B. et al., "OpenMP for accelerators," Talk Held at the International Workshop on OpenMP, 2011. [http://www.ncsa.illinois.edu/Conferences/IWOMP11/program/presentations/beyer.pdf; accessed September 27, 2012].

[12] NVIDIA, "NVIDIA developer zone: OpenACC." http://developer.nvidia.com/cuda/openacc, 2012. [Online; accessed September 27, 2012].

[13] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), ACM, 2011.

[14] Tobias Gysi, *Supercomputing Systems AG*, "A stencil library for the new dynamic core of COSMO," Talk Held at 2012 GPU Technology Conference, San Jose, 2012. [http://nvidia.fullviewmedia.com/gtc2012/0517-N-S0256.html; accessed October 5, 2012].

[15] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," 2008.

[16] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 7:1–7:11, ACM, 2009.

[17] S. Kronig and M. Müller, "Feasibility and performance of weather computations using GPGPU programming," *Semester Thesis Supervised by Prof. Tröster G., Electronics Laboratory (IfE), ETH Zurich*, 2011.

[18] *TSUBAME 2.0 User's Guide.*

[19] G. W. et al., "Node-level performance of the lattice Boltzmann method on recent multicore CPUs," Talk Held at ParCFD 2011, 2011. [http://www.skalb.de/all_pdf/ParCFD2011%20Minisymposium%20LBM-CPUs.pdf; accessed September 28, 2012].

[20] I. Corporation, "Specifications Intel Xeon processor X5670." http://ark.intel.com/products/47920/Intel-Xeon-Processor-X5670-%2812M-Cache-2_93-GHz-6_40-GTs-Intel-QPI%29, 2010. [Online; accessed September 26, 2012].

[21] A. Vladimirov, "Arithmetics on Intel's Sandy Bridge and Westmere CPUs," 2012. [http://research.colfaxinternational.com/file.axd?file=2012%2F4%2FColfax_FLOPS.pdf; accessed September 28, 2012].

[22] NVIDIA, "Specifications NVIDIA Tesla C2050 GPU computing processor." http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf, 2010. [Online; accessed September 26, 2012].

[23] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[24] "Simple SSE and SSE2 optimized sin, cos, log and exp." http://gruntthepeon.free.fr/ssemath, 2011. [Online; accessed October 1, 2012].

[25] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor," 1999.

[26] The Portland Group, *CUDA Fortran Programming Guide and Reference*, 2012.

Appendix A

Usage of the Hybrid Fortran Framework

This appendix is intended to give the informations necessary for using the **Hybrid Fortran** framework.

A.1 Framework Dependencies

Hybrid Fortran requires the following software components:

1. Any Fortran 90 CPU compiler.
2. For GPU compilation: PGI (`pgf90`) with CUDA Fortran support.
3. Python v2.6 or compatible.
4. GNU Make.
5. A POSIX compatible operating system.
6. For the graphical callgraph representation using `make graphs`: "pydot" python library[1] as well as the "Graphviz" program package[2].

A.2 User Defined Components

The following files displayed in figure 4.1 are defined by the user:

h90 Fortran sources A source directory that contains Hybrid Fortran files (h90 extension). It may also contain files with f90 or F90 extensions. The source directory is by default located at `path-to-project/source/*`.

[1] http://code.google.com/p/pydot/
[2] http://www.graphviz.org/Download..php

Makefile Used to define module dependencies. The Makefile is by default located at `path-to-project/buildtools/Makefile`. Note: All source files are being copied into flat source folders before being compiled - the build system is therefore source directory structure agnostic, i.e. files can be placed into arbitrary subdirectores below the source directory.

MakesettingsCPU CPU compiler settings are specified in `MakesettingsCPU`, located at `path-to-project/buildtools/`.

MakesettingsGPU GPU compiler settings are specified in `MakesettingsGPU`, located at `path-to-project/buildtools/`.

storage_order.F90 This fortran file contains fortran preprocessor statements in order to define the storage order for both CPU and GPU implementation.

A.3 Build Interface

make builds both cpu and gpu versions of the codebase situated in `path-to-project/source/*`.

make cpu builds the cpu version of the codebase situated in `path-to-project/source/*`.

make gpu builds the gpu version of the codebase situated in `path-to-project/source/*`.

make install builds both cpu and gpu versions of the codebase situated in `path-to-project/source/*` and installs the executables into the test folder defined in `path-to-project/buildtools/MakesettingsGeneral`.

make install_cpu Like `make install`, but it performs these steps only for the cpu version.

make install_gpu Like `make install`, but it performs these steps only for the gpu version.

make TARGETS DEBUG=1 builds TARGETS in debug mode (use any of the targets defined above). Uses the `DebugCUDAFortranImplementation` in case of GPU compilation, which prints predefined data points for every kernel parameter after every kernel execution.

make graphs creates the graphical callgraph representations in the `path-to-project/build/callgraphs/` directory.

A.4 Test Interface

The following files are part of the sample test interface provided with **Hybrid Fortran**:

accuracy.py Compares a Fortran 90 `.dat` file with a reference `.dat` file. Endianness and number of bytes per floating point value can be specified using command line parameters, see `--help` for usage.

allAccuracy.sh Compares all Fortran 90 `.dat` files in the `./out` directory with a specified reference directory.

runTestWithArchitecture.sh Use `cpu` or `gpu` as command line parameter for running tests over multiple domain sizes for either the CPU or GPU version. This script is intended to be edited in order to suit the specific use case.

runTests.sh Edit this file in order to call the `runTestWithArchitecture.sh` with the architectures you would like to test as well as appropriate command line definitions for logging the results.

A.5 Migration to Hybrid Fortran

Assuming that the starting point is a Fortran 90 source code, use the following guidance in order to port your codebase to **Hybrid Fortran**:

1. Make a renamed copy of the example directory.
2. Make sure it compiles on your system by typing `make install`. Otherwise it is likely that some dependencies are missing, please reconsider section A.1.
3. Delete `source/main.h90` and copy in your sourcecode. Please note that all loops that are to be run in parallel and their entire callgraph should be visible to the compiler in the source subdirectory. Your sourcecode may be in an arbitrary subdirectory structure below the `source` directory and may consist of `f90` and `F90` files. The buildsystem will find all your sourcefiles recursively and copy them into the respective build directory in a flat hierarchy.
4. Adjust `buildtools/MakesettingsGeneral` - the configuration options should be self explanatory.
5. Adjust `buildtools/Makefile` by adding your dependencies. See the `asuca-pp`'s Makefile as an example. If your previous build system was already built on `GNU Make` it should be possible to copy in your dependency definitions without change. If you would like to exclude some files or folders in your source directory from compilation (in order to integrate modules one by one), you can do this by editing the `EXCEPTIONS` variable in `MakesettingsGeneral`.

6. Adjust the `FFLAGS` and `LDFLAGS` variables in `buildtools/MakesettingsGPU` and `buildtools/MakesettingsCPU` to reflect the compiler and linker options that are needed to compile your codebase. Please note that currently only CUDA Fortran with the Portland Group compiler `pgf90` is supported and tested as the GPU implementation.

7. Run `make clean; make install` and run your program in order to test whether the integration of your sourcecode into the **Hybrid Fortran** build system has been successful. It should create the cpu and gpu executable versions of your program in the test directory, however the gpu version will not run on the GPU yet, since no directives have been defined so far.

8. Integrate a test system. You can use the test scripts that have been provided with this framework (see sec. A.4) or use any other test system. If you would like to integrate the provided system, please copy in the four test scripts in the locations described in sec. A.4 and do the following:

   (a) Edit `runTestWithArchitecture.sh` in order to reflect the runtime parameters of your executable.

   (b) Have a look at `runTests.sh` for choosing the architectures you would like to run the tests for (cpu and gpu or only one of those architectures). For the moment it is best to comment out the gpu test, since the code will first be adapted for cpu architecture only.

   (c) Add calls to the `helper_functions` module procedures `write2DToFile` and `write3DToFile` to your program to be tested in order to write your data to the `.dat` files in the `test\out` folder. Have a look at the `asuca-pp/source/test/rad_pp/main_pp.h90` file as an example. You may choose any filename for the `.dat` files, the `allAccuracy.py` script will find them automatically in the `test\out` folder.

   (d) Create the `test/out` folder.

   (e) Compile (`make install` in project directory) and run your program again to make sure the files are being created. Make sure that they actually contain data, for example by checking the file size.

   (f) Repeat steps 8c, 8d, 8e in your reference source code.

   (g) Copy the reference data created by the last step into a `test/ref` directory and make sure it is referenced correctly within `runTestWithArchitecture.sh`.

   (h) Run "`runTestWithArchitecture.sh cpu`" and check whether the two versions match by checking the output.

   (i) From now on you can run the `runTests.sh` skript every time you have modified and recompiled your program.

9. Define the parallel regions that are to be accelerated by GPU[3]. Rename all files that (a) contain such regions or (b) contain subroutines that are part of the call hierarchy inside those regions from `*.f90` or `*.F90` to `*.h90`.

10. Make sure your program still compiles and runs correctly on CPU by executing `make clean;make install_cpu; ./runTests.sh`.

11. Replace the loops of your parallel region with a `@parallelRegion{appliesTo(CPU), ...}` directive. See sec. 3.2.2 for details.

12. Run `make graphs`. You should now have a graphical representation of your call hierarchy as "seen" from your parallel regions upwards and downwards in the call tree.

13. Define the subprocedures within that call hierarchy that are to be ported as GPU kernels. These subprocedures should have the following properties:

    (a) They only call subprocedures from the same module.

    (b) They only call one more level of subprocedures (rule of thumb).

    (c) The set of these subprocedures is self enclosed for all data with depencies in your parallel domains, i.e. your data is only directly read or written to inside these to-be kernels and their callees. If this is not the case it will be necessary to restructure your codebase, e.g. put pre- and postprocessing tasks that are defined inline within higher level subprocedures into subprocedures and put them into your set of kernels.

14. Analyse for all kernels which data structures they require and which of those structures are dependant on your parallel domain. Hint: Data structures that are local to kernels are often part of these parallel domain dependants. Define `@domainDependant` directives for those data structures within all kernels, kernel subprocedures and all intermediate subprocedures between your kernels and your CPU parallel region. See sec. 3.2 for details.

15. If your kernels use local arrays, declare them in the respective kernel caller using automatic arrays and pass them to the kernel subroutine using parameters. Use appropriate `@domainDependant` directives.

16. If your kernels use scalars from other modules, pass them from the kernel caller using parameters. Use appropriate `@domainDependant` directives.

17. If your kernels use scalars or arrays from their own module, pass them from the kernel caller using parameters. Use appropriate `@domainDependant` directives.

[3]For example "do" loops that are already executed on multicore CPU using OpenMP statements.

18. If your kernels use arrays from other modules, pass them from the kernel caller using parameters. In the current framework version, these arrays also need to be input parameters of the kernel caller itself for appropriate automatic device data handling. That is, they need to be passed down to the kernels on two levels. Use appropriate `@domainDependant` directives.

19. Wrap the implementation sections of all your kernels with `@parallelRegion{appliesTo(GPU), ...}` / `@end parallelRegion` directives. See sec. 3.2.2 for details.

20. Test and debug on CPU by executing `make clean;make install_cpu DEBUG=1; ./runTests.sh`.

21. Switch to GPU tests in `runTests.sh` and test and debug on GPU using `make clean;make install_gpu DEBUG=1; ./runTests.sh`.

22. Congratulations, you have just completed a CPU/GPU hybrid portation using **Hybrid Fortran**.

Appendix B

Contents of the Attachments

Measurements Measurements for graphs shown throughout this thesis.

ThesisWeatherOnCUDA.pdf Previous Semester Thesis by Stefan Kronig, Michel Müller in field of Weather Prediction on GPU.

Example Hybrid Fortran Directory Simple example codebase implemented using **Hybrid Fortran** as shown in cha. 3.

ASUCA Physical Process Sample Implementation Sample ASUCA implementation using **Hybrid Fortran** as shown in cha. 5.

Test Scripts Scripts to use as a sample test interface for your **Hybrid Fortran** implementations as referenced in appendix A.4.

Particle Push Implementation Directory Particle push implementation using CUDA C and OpenACC as shown in cha. 2.

3D Diffusion Implementation Directory 3D diffusion implementation using CUDA C and OpenACC as shown in cha. 2.

Shortwave Radiation Implementation Directory Shortwave radiation implementation using CUDA Fortran and OpenACC as shown in cha. 2.

Legal Disclaimer ASUCA Code Legal disclaimer concerning the attached ASUCA source code.