

論文 / 著書情報
Article / Book Information

題目(和文)	現業気象予報モデルをソースコード変換によりGPU計算に適用させる 高生産性フレームワーク
Title(English)	A Framework for High Productivity GPU Acceleration Through Code Transformation Applied to Operational Weather Prediction Model in Japan
著者(和文)	ミュラー ミシェル
Author(English)	Michel Mueller
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第10851号, 授与年月日:2018年3月26日, 学位の種別:課程博士, 審査員:青木 尊之,奥野 喜裕,末包 哲也,肖 鋒,長崎 孝夫
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第10851号, Conferred date:2018/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Doctoral Thesis

**A Framework for
High Productivity GPU Acceleration
Through Code Transformation
Applied to Operational Weather Prediction Model
in Japan**

Michel Müller

Department of Energy Sciences
Tokyo Institute of Technology

February 27, 2018

Abstract

Numerical weather prediction is steadily advancing towards increasingly high resolutions in order to allow accurate modelling of cloud forming processes. This advancement is accompanied with an increasing demand for memory bandwidth. GPUs are very promising in this regard - their internal memory is highly performant compared to conventional approaches and today's GPU programming models have matured for scientific use cases in an ongoing process since more than ten years. On the other hand, numerical weather prediction is a very demanding environment with regards to operational requirements for the codebases in use - maintainability and ease-of-use for domain scientists is paramount for a successful deployment of new hardware architectures. While the performance of GPUs for earth system software has been studied extensively in related work, there is a gap in studies and frameworks where productivity plays a central role. In this thesis we therefore want to close this gap.

We introduce "Hybrid Fortran", a new approach that allows high performance GPGPU porting for data parallel Fortran codes. This technique requires only minimal changes for a CPU targeted codebase, a significant advancement in terms of productivity. It has been successfully applied to both dynamical core and physical processes of ASUCA, a Japanese mesoscale weather prediction model with more than 150k lines of code. By using extensive code transformation techniques, more than 85% of the original codebase was left unchanged and the overall code size has been extended by less than 4%. This shows very promising productivity, maintainability and ease-of-adoption in an operational setting.

By means of a minimal weather application that resembles ASUCA's code structure, Hybrid Fortran is compared to both a performance model as well as today's commonly used method, OpenACC. As a result, the Hybrid Fortran implementation is shown to deliver the same or better performance than OpenACC and its performance agrees with the model both on CPU and GPU. In a full scale production run, using an ASUCA grid with 1581 x 1301 x 58 cells and real world weather data in 2km resolution, 24 NVIDIA Tesla P100 running the Hybrid Fortran based GPU port are shown to replace more than 50 18-core Intel Xeon Broadwell E5-2695 v4 running the reference implementation - an achievement comparable to more invasive GPGPU rewrites of other weather models.

We want to broaden the perspective regarding software tooling efforts in the field of weather- and climate modelling. A review of relevant hybrid GPU- and CPU implementation methods is provided, including a characterization and

comparison by means of a productivity model. We conclude that Hybrid Fortran is a highly promising and unique approach for single-grid data-parallel Fortran applications, i.e. we see potential use for many applications - both legacy and new.

Contents

Abstract	iii
1 Introduction	1
1.1 Important Terms and Concepts	2
1.2 GPU Computing	6
1.2.1 Processor Design and Current Performance Characteristics in Contrast to CPUs	6
1.2.2 Programming Model	9
1.2.3 Concluding Remarks	14
1.3 Numerical Weather Prediction	14
1.3.1 Fundamental Theory and Numerical Methods	15
1.3.2 Microscale Phenomena	18
1.3.3 Cloud Resolution	19
1.3.4 GPUs Applied to Numerical Weather Prediction	20
1.3.5 ASUCA NWP Model	21
1.3.6 Key Challenges for GPU Computing in NWP	22
1.4 Related Work	24
1.4.1 ASUCA GPU Port by Shimokawabe et al.	25
1.4.2 COSMO	25
1.4.3 Weather Research and Forecasting Model (WRF)	25
1.4.4 Non-hydrostatic Icosahedral Model (NIM)	26
1.4.5 Accelerated Model for Climate and Energy (ACME)	26
2 A Method for High Productivity GPU Porting	29
2.1 Motivation	29
2.2 Parallel Loop Abstraction	30
2.3 Compile-time Defined Memory Layout and Device Data Region	33
2.3.1 Storage Order	33

2.3.2	Device Data Region	34
2.3.3	Dimensionality Changes	35
2.4	Transformed Code	36
2.5	Code Transformation Method	39
2.5.1	Framework Components	39
2.5.2	Overview of Transformation Process	41
2.5.3	Procedural Shell	44
2.5.4	Parser State Machine	46
2.5.5	Application Model	47
2.5.6	Transformation of Application Model to Fortran Source	49
2.6	Concluding Remarks	52
3	Application to Numerical Weather Prediction	53
3.1	Hybrid ASUCA Implementation	54
3.1.1	Design Goals	54
3.1.2	Data Structures- and Layout	57
3.1.3	Parallelization	57
3.1.4	Communication between Host and Device	61
3.1.5	GPU Implementation of the ASUCA Model	64
3.2	Verification	70
3.3	Productivity	71
3.4	Comparison to Previous Work	73
3.5	Applicability to Further Models	74
3.6	Concluding Remarks	76
4	Performance Analysis	79
4.1	Feasibility for High Throughput Co-processors	79
4.2	Reduced Weather Application	82
4.2.1	Application Model	82
4.2.2	Performance Model	84
4.2.3	Performance Results	86
4.3	Hybrid ASUCA	89
4.3.1	Hybrid ASUCA Overall Kernel Performance	90

CONTENTS	vi
4.3.2 Hybrid ASUCA Strong Scaling GPU Results	90
4.4 Concluding Remarks	93
5 Productivity Review of Hybrid Implementation Methods	95
5.1 Classification and Outline of Methods	96
5.1.1 Domain-Specific Languages	96
5.1.2 Directive-Based Porting Methods	97
5.1.3 Granularity Optimization Methods	98
5.1.4 Memory Layout Transforming Methods	98
5.2 Productivity Characterization	99
5.2.1 Evaluation Criteria: Data Structures	99
5.2.2 Evaluation Criteria: Control Structures	100
5.2.3 Evaluation	101
5.3 Concluding Remarks	105
6 Conclusion and Future Work	107
Appendices	109
A Hardware Performance	109
B Performance Test Configurations	111
B.1 Reduced Weather Configuration for Performance Tests	111
B.2 ASUCA Configuration for Tests with 301 x 301 x 58 Grid	112
B.3 ASUCA Configuration for Tests with 1581 x 1301 x 58 Grid on Reedbush-H	114
B.4 ASUCA Configuration for Tests with 1581 x 1301 x 58 Grid on TSUBAME 3.0	115
B.5 ASUCA IJ domain decomposition	115
C Sources	117
C.1 Reduced Weather: Diffusion	117
C.2 Reduced Weather: Code Generated by Hybrid Fortran for Diffusion	118
C.3 Reduced Weather: Physical Processes	119
C.4 Reduced Weather: Radiation	119

CONTENTS	vii
C.5 Reduced Weather: Heat Exchange	120
C.6 Reduced Weather: Time Integration	120
Acknowledgements	123
Bibliography	125

List of Figures

1.1	GPU execution model.	9
1.2	Interrelations of atmospheric models with respect to their approximations.	17
1.3	ASUCA's model simulation boundaries.	21
2.1	Simplified code structure of ASUCA.	32
2.2	Simplified code structure of ASUCA using Hybrid Fortran.	33
2.3	Hybrid Fortran software components and build workflow.	40
2.4	Steps involved in the code transformation method.	41
2.5	Callgraph coloring on CPU.	43
2.6	Callgraph coloring on GPU.	43
2.7	Inheritance graph of procedural classes involved in the Hybrid Fortran code transformation.	45
2.8	Parser state machine.	46
2.9	Example of an intermediate global application model.	48
2.10	Application model class hierarchy and composition.	49
2.11	Example of an intermediate global application model after routine splitting has been applied.	50
2.12	Implementation class hierarchy.	51
3.1	Overview of simulated processes in Hybrid ASUCA when run on GPUs.	56
3.2	Horizontal parallelization, CPU vs. GPU.	58
3.3	Granularity transformation of physical processes call tree for GPU.	67
3.4	Memory footprint optimization for radiation code.	68
3.5	Granularity transformation of precipitation module with column-wise CFL convergence.	69
3.6	Total cloud cover result with ASUCA using a 2km resolution grid with real initial data.	71

3.7	Flow of code from reference implementation to Hybrid ASUCA by number of lines of code.	72
3.8	New code required for Hybrid ASUCA versus estimate of equivalent OpenACC implementation.	72
4.1	Output at $j = 100$	82
4.2	Code structure of the reduced weather application.	83
4.3	Comparing performance with the reduced weather application for handwritten vs. Hybrid Fortran generated vs. model on 256^3 Grid.	89
4.4	Execution time measurements for 75 long time steps of ASUCA executed in four different configurations.	90
4.5	Strong scaling speedup on $1581 \times 1301 \times 58$ ASUCA Grid.	91
4.6	Impact of communication for strong scaling on $1581 \times 1301 \times 58$ ASUCA Grid.	91
4.7	Impact of modules for strong scaling on $1581 \times 1301 \times 58$ ASUCA Grid.	92
5.1	Interactive tool for productivity examination.	106
B.1	Overview of simulated processes in Hybrid ASUCA when run on GPUs.	113

List of Tables

1.1	Xeon 8180 Skylake-SP versus NVIDIA Tesla P100.	8
1.2	Performance metrics for Xeon 8180 Skylake-SP versus NVIDIA Tesla P100.	8
3.1	Comparison of Hybrid ASUCA to GPU implementation by Shimokawabe et al.	74
4.1	Model Output Cycle Execution Time for TSUBAME 2.5.	86
4.2	Influence of Storage Order on Performance.	87
4.3	Execution Time with “Naive” Parallelization.	87
4.4	Execution Time with Data Region.	88
4.5	Block Size Impact on Execution Time.	88
5.1	Data structure characterization for hybrid implementation methods.	100
5.2	Control structure characterization for hybrid implementation methods.	101
5.3	Scoring matrix $A_{ij}(data)$	103
5.4	Scoring matrix $A_{ij}(control)$	104
5.5	Productivity score $P(r)$ for different use cases r	105
A.1	Hardware performance metrics.	110
B.1	MPI IxJ domain decomposition by number of processes.	116

Introduction

With the growth in single threaded performance on CPUs slowing down [1], HPC applications today are increasingly required to make effective use of on-chip parallelism. In 2005 this was phrased “the free lunch is over”: rather than relying on a steady and exponential increase of computational performance to automatically benefit already existing software, it has become necessary for applications to specifically target on-chip parallelism [2].

As a consequence of this paradigm shift, co-processors commonly referred to as “accelerators” have become increasingly common in supercomputers, since these chip architectures are generally designed to offer a higher computational throughput for a given number of transistors compared to CPUs. Top500 currently lists six of the top ten systems to make use either NVIDIA Tesla or Intel Xeon Phi based accelerators [3]. Offloading computationally intensive tasks to co-processors has thus become an increasingly common task in the field of HPC. Such co-processors generally require a different programming paradigm to achieve high efficiency however.

This need for architecture specific programming has created a divide between domain scientists who mainly care about application logic, and the supercomputers their applications run on. A high rate of change on both hardware- as well as application side exacerbate this issue. To cite an instance, Argonne National Laboratory’s Norman et al. write about a recent accelerator implementation of the U.S. DOE’s “Accelerated Model for Climate and Energy”: “The original CUDA Fortran¹ refactoring gave good performance, but it turned out that the codebase was still somewhat in flux. This meant that changes needed to be propagated into the CUDA code, which was quite difficult to support” [4].

The work presented in this thesis has initially been motivated by an evaluation of operative GPU support for the Japanese production weather prediction

¹CUDA Fortran is a low level programming interface for Nvidia co-processors available in Fortran

model “ASUCA”. As the main mesoscale weather prediction model used in operation and developed at the Japan Meteorological Agency it is used to generate nine-hour-forecasts every hour. ASUCA uses a structured grid and is implemented in Fortran[5][6]. In order to gain accelerator support, the aforementioned maintainability issues are a main point of contention. The underlying model and its implementation is constantly being pushed further by the application owners, and it is thus of fundamental importance to maintain ease-of-use and high productivity for the domain scientists when working with the code, both on CPU and GPU. A single codebase for both architectures is paramount. We therefore investigate methods to re-target structured grid applications in Fortran with low programming effort for an efficient GPU implementation while at the same time retaining CPU compatibility.

This thesis is structured as follows: Chapter 1 introduces the concepts of parallel- and GPU computing in contrast to more conventional programming methods. Numerical weather prediction and its use case for GPUs is discussed as well related work in this field of research. In Chapter 2 we propose a new high productivity method for implementing applications with both multi-core CPU as well as GPU support. The application of that method to ASUCA is discussed in Chapter 3. A performance analysis of the method and its current applications is provided in 4. In Chapter 5 a review of available methods in this field is provided and the contribution of our method is discussed in this context. Finally, Chapter 6 concludes this thesis and we discuss potential future work based on the contributions presented here.

1.1 Important Terms and Concepts

In this Section we define technical terms that are essential to the understanding of this thesis.

High Performance Computing With transistor counts still roughly doubling every two years (i.e. scaling according to Moore’s law), computing hardware is still under rapid development, creating difficulties to quantify what application or system falls within the field of “High Performance Computing” (commonly abbreviated to “HPC”). Our attempt to give such a definition in a forward-compatible way goes as follows:

1. On the systems side, when the utilized hardware in a computing system would not fit in a typical six-tatami-sized Japanese bedroom, either for spatial- or (more commonly) thermal reasons, we call it a HPC system.
2. On the applications side we call a HPC application when the underlying use case demands for a HPC system in order to achieve the requirements

in either computational performance (typically floating point operations per second, FLOPS), working memory bandwidth (bytes per second, B/s), working memory size (bytes, B), or storage and/or cluster-internal network performance (bytes per second and bits per second, respectively), often referred to using the shorthand "I/O" (input/output). Typically excluded from the HPC term are applications where by far the largest share of the workload arrives from a high number of computational requests coming from the external network - we call these "web applications" instead, on which this thesis will not focus.

Parallel Computing A high computational-, memory- or I/O performance can be achieved in two ways:

1. speeding up all utilized components. For computational performance this entails either increasing the clock rate or decreasing the clock cycles required for each operation. For memory- and network performance it means increasing the number of physical connections (lanes and/or wires), increasing the clock rate or decreasing the latency by shortening the path taken by electromagnetic waves.
2. increasing the number of operations that are executed in parallel. This refers to the number of values computed for a single instruction (called "vector size"), the number of independently operating threads ("core" count per processor), the number of processors on one mainboard (also called "sockets" due to how processors are connected to the rest of the system) and, finally, the number of mainboards ("nodes") working together in a system.

Over time, the importance of the first approach to speeding up HPC systems has gradually decreased due to various physical limitations (feature size miniaturization [7], leakage current and the resulting thermal effects [8] as well as information travel time being bound to light speed and the resulting effects on electric pathway length and thus node size). Instead, increasing the number of parallel operations has gained more and more significance in order to scale up performance. This has resulted in new problems for application developers however: While applications are sped up from the first kind of performance increase (also called "single threaded performance") without any change required for the software, parallel performance gains tend to require a significant reengineering of software. The overarching theme of this thesis is thus to find ways to lower the hurdles that this shift towards parallel computing creates for application developers.

Latency In the context of computing, the term "latency" is used to describe the total time to solution when executing a task.

Granularity In parallel computing, granularity is defined as the amount of work performed by a task [9]. Large and small workloads per task are described as coarse-grained and fine-grained, respectively. In the context of this thesis we use this term to describe the amount of work performed per thread in a single shared-memory system (as opposed to MPI-processes that are used in conjunction for multi-node parallelization).

GPGPU Programming In this work we make extensive use of a specific type of co-processors, commonly called “Graphics Processing Unit” (GPU), although we consider this term outdated. While graphics was the initial and only purpose of these devices, today it is only one of their use cases. GPUs are now used for many computational tasks in mobile-, desktop- and supercomputing. The main commonality of these tasks is that they are highly parallelizeable and require a high throughput. “General-purpose computing on graphics processing units” (GPGPU) is the term commonly applied to these use cases[10]. We thus use the terms “GPU” and “GPGPU” in this work to distinguish from the usage of CPUs, even though this thesis is not focused on graphics.

Kernel In the context of this thesis, “kernel” refers to an inner loop using a data-parallel algorithm that is compiled for parallel execution. The term has become common in GPGPU programming, however we use it also for code compiled for CPU.

Stencil Code Stencil codes are a class of kernel whereby array elements are updated based on a fixed pattern involving neighboring data points. Said pattern is called a stencil. A typical example are 7-point Von-Neumann-style stencils where an update involves the centering data point as well as six neighbors in three dimensions.

Occupancy The percentage of a processor’s function units (either floating-point or integer, whichever dominates) that are occupied on average during the runtime of a kernel. Occupancy is related to a number of algorithmic aspects - generally it is a combination of the exposed parallelism in an algorithm and the usage of computing resources alongside the function units, such as memory bandwidth, on-chip caches, registers and schedulers. If any of these resources are used at a higher ratio (compared to function units) than the chip architecture is designed for, occupancy is lowered.

Host In the context of GPUs the term “host” refers to the system which hosts said graphics co-processors. The host system is controlled by one or multiple CPUs.

Memory Layout We define memory layout as the order in which application data is stored in the non-persistent off-chip memory system that is closest to the computational units, also called “working memory”. This refers to main- and device memory for host computing and GPGPU, respectively.

Cache A cache is a temporary store of data used to shorten access times for subsequent uses. In the context of this thesis, caches are always data storage units that are located on-chip, i.e. they operate on the same chip as the computational units that process said data. Cache optimization is thus a very common technique for HPC software, i.e. the optimization of data access patterns to maximize “cache hits”.

Library A library in programming is a set of functions, often provided by a third party.

Language Extension Language extensions are additions to the grammar of a programming language.

Domain-Specific Language A domain-specific language (DSL) is a programming language that is specialized to accommodate a specific application domain.

Directive Directives are a way to enhance otherwise semantically complete code with additional information that instructs the compiler on how to implement the application for specific architectures. A common example are directives added to loops, instructing the compiler to parallelize the loop in a specific way. Directive-based programming thus stands in contrast to DSLs, language extensions and libraries, where the semantics of the code changes in order to accommodate the implementation.

CUDA NVIDIA’s GPGPU programming interface in the form of a runtime library as well as a language extension for C, C++ and Fortran.

OpenACC A higher level directive-based GPGPU programming interface. OpenACC is an industry standard, its main vendor supporters are NVIDIA, Cray and AMD.

OpenMP A higher level directive-based programming interface for parallel computing. OpenMP is a widely supported industry standard. Recently OpenMP has introduced GPU support, however the implementations with regards to GPU are currently in a less mature state compared to OpenACC.

Code Divergence Any computer code at its essence operates on two levels:

1. it defines the application logic,
2. it instructs specific computing infrastructure how to operate in order to implement the application logic.

As the latter is dependent on the underlying architecture, code divergence is a measure of the fraction of code that requires reimplementation when switching or upgrading the computing architecture. In order to reduce code divergence we therefore need to reduce this secondary role of computer code - this can be done by moving the responsibility to the compiler (i.e. by making use of compiler optimizations), by abstracting it through the means of library, language extensions or DSLs, and/or by centralizing the divergent characteristics into a configuration (either acting at compile-time or run-time).

Hybrid Code In the context of this thesis a code is called “hybrid” if its runtime is both suitable for CPU-only as well as GPU-assisted systems.

1.2 GPU Computing

This Section outlines the relevancy of GPUs in HPC and their differences compared to CPU, especially with respect to the context of this thesis.

1.2.1 Processor Design and Current Performance Characteristics in Contrast to CPUs

Arguably the most important concept for the understanding of GPU computing is the contrast between throughput optimization and latency optimization in processor design. Central Processing Units (CPUs) are, at their core, designed to decrease latency in terms of the time spent to finish any given task, independent of the amount of data processed. This is a very important characteristic for tasks like the displaying of graphical user interfaces (where a click or a finger tap should result in immediate feedback to the user), for various system housekeeping jobs required on modern personal computing operating systems, as well as to allow for communication between various processes and system components with minimal delay. When fundamental lower limits on latency due to thermal processes started to become apparent, throughput optimizing techniques - designed to accelerate tasks with larger amounts of data - have been added to existing CPU designs as an added value in order to keep marketing new processors to buyers. For instance, in the late 1990s Intel began releasing consumer processors (“Pentium”) with the “MMX” instruction set - new operations that, among

other benefits, allowed the computation of two 32-bit floating point values simultaneously (thus having a vector size of two). Over time, the throughput of new processors has kept growing faster than latency has decreased, through increasing use of vector instructions (also called “single instruction, multiple data” or SIMD) and independent threads on the same microchip (“core” count) - under the condition that software is designed to make use of these techniques. Due to the pressure of existing applications however, CPU designers were for the most part not allowed to increase latency between generations in order to achieve even higher throughput (i.e. by reducing the complexity of cores, allowing for an even higher number of cores on the same die size) [11].

In contrast, GPUs are a type of microchip designed for high throughput from the beginning, usually sacrificing low latency. Even though modern microchips have transistor counts on the order of 10^9 to 10^{10} , it is important to recognize that every transistor still has a thermal cost and occupies space on the chip die, thus needs to be budgeted carefully towards a specific goal. When designing a Von-Neumann-type computing system towards throughput, this essentially means allocating as much die space as possible to computational units while keeping enough memory lanes, caches, registers and work schedulers in order to balance the system for workloads that the computing market currently demands of such a throughput maximising co-processor. Such workloads today typically include computer graphics, cryptographic hash function solvers (e.g. Bitcoin mining) and numerical solvers (e.g. grid based methods, particle methods or neural network based methods).

Due to the NVIDIA corporation’s current market dominance in GPUs used for HPC in general, and their prominence within the computing services at Tokyo Institute of Technology (i.e. in the TSUBAME supercomputer series) in particular, this thesis will focus on NVIDIA GPUs - however many of the concepts introduced here apply to other GPU brands as well, albeit often with different terminology and programming interfaces.

Table 1.1 shows a basic hardware performance comparison of the latest CPUs and GPUs as of 2017. We concentrate on double precision (64-bit) floating point (FP) performance here, since this is the most important computational type for atmospheric simulations.

The following formula gives the theoretical peak double precision FP performance on these two systems:

$$P_{peak} = \frac{N_{cores} \cdot l_{vector} \cdot f_{clock}}{CPI_{min}}, \quad (1.1)$$

with N_{cores} being the number of cores, l_{vector} the vector length, f_{clock} the clock frequency and CPI_{min} the minimum cycles per instruction.

Table 1.1: Comparison of the latest generation Intel CPU (“Xeon 8180 Skylake-SP”) and NVIDIA GPU (“Tesla P100”) intended for HPC purposes. “SM” stands for “streaming multiprocessor”.

Characteristic	CPU	GPU
Vector length (double precision)	16	32
Core- or SM count	28	60
Clock frequency	2.5 GHz	1.3 GHz
Memory bandwidth	119 GB/s	720 GB/s
Thermal design power	205W	300W
Die size	$\sim 700 \text{ mm}^2$	610 mm^2
List price	\$10,009	\$7374 (including 16 GB HBM2 Memory)

Substituting 2 as the maximum number of FP operations per instruction (fused multiply-add), one arrives at performance metrics as listed in Table 1.2.

Table 1.2: Performance metrics for the latest generation Intel CPU (“Xeon 8180 Skylake-SP”) and NVIDIA GPUs (“Tesla P100”, PCI-E version) intended for HPC purposes.

Metric	CPU	GPU
Peak GFLOP/s (double precision)	2240	5004
GFLOP/s per Watt	10.9	16.7
GFLOP/s per Dollar	0.22	0.68
Memory bandwidth per Dollar	11.89 MB/s	97.64 MB/s
FLOP/Byte system balance	18	6

In addition, one of the earlier differentiating factors of GPUs was the decreased significance of caches and thus increase of die space allocated to computational floating-point units (FPU). A merging between the design spaces of GPUs and CPUs (as shown in the above comparison with recent Skylake-SP releases) should also show a consolidation of these differences.

Taking these metrics into consideration one can observe the following:

1. After years of competition between the two architectural philosophies in the space of HPC, on first sight they now look similar. Today’s available CPUs have caught up to GPU significantly in theoretical throughput within a factor of two, for comparable die sizes.
2. In the GFLOP/s per Watt metric, the most efficient CPUs have caught up even more, at around two thirds of current GPUs.
3. The main differentiators today are cost and memory bandwidth. In the above comparison, computational performance on CPU costs more than

three times as much as on GPU, more so if we include the cost of working memory in the calculation (which is included in the listed GPU price but excluded on CPU). In terms of bandwidth, NVIDIA's flagship GPUs have a six-fold advantage in absolute terms and more than an 8-fold advantage per dollar spent over the fastest current Intel CPU.

4. Due to a relative stagnation of memory bandwidth coupled with the quickly increasing computational throughput, the system balance for FLOP per Byte is now three times higher on CPU vs. GPU. To achieve system balance, a Xeon 8180 requires 144 (or equivalent) 64-bit floating point operations per 64-bit value read or stored on the working memory - a rare occurrence in most algorithms, resulting in most applications being constrained by the memory bandwidth on this particular CPU, thus lowering the achievable performance according to the Roofline model.

1.2.2 Programming Model

In Section 1.2.1 we have concluded that, in theory, CPU and GPU are today similar at least in their theoretical computational throughput. In terms of programming models, significant differences can be observed from the two architectures however. In this section, the GPU programming model is outlined in contrast to its equivalent on CPU, i.e. CPU code that is written towards use in a HPC context.

Execution Model

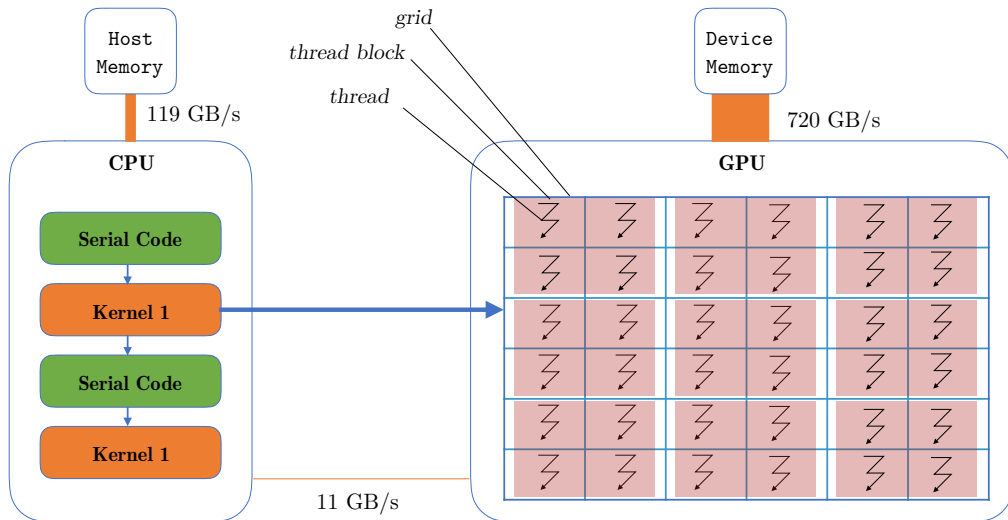


Figure 1.1: GPU execution model.

Figure 1.1 outlines the execution model on GPU. Serial parts of an application reside on the CPU while parallel parts are offloaded to the graphics co-processor by mapping the data to be operated on to a CUDA grid, consisting of thread block. The memory bandwidth between the involved systems is important to consider - the PCI-express bus between CPU and GPU is very slow compared to the memory bus available to the processors. Due to inefficiencies of automatic approaches the standard in HPC is thus to handle the location and movement of data between the memory systems manually.

Parallelization

Consider the following kernel, which is inspired by code modelling physical processes, as found in atmospheric simulations.

Listing 1.1: Sample kernel, illustrating the differences in the CPU and GPU programming models.

```
do j = 1, ny
  do i = 1, nx
    if (b(i,j)) then
      do k = nz - 1, 1, -1
        a(k,i,j) = a(k+1,i,j) * exp(- sqrt(gamma(k,i,j)) * tau)
      end do
    else
      do k = 1, nz - 1
        a(k,i,j) = 0.0d0
      end do
    end if
  end do
end do
```

Listing 1.1 shows a calculation that is both computationally intensive (square root and exponential calculation per point in a three dimensional grid) as well as expensive in terms of memory bandwidth (two loads and one store per grid point). It is safe to assume that due to the expensive computations, this code would benefit from vectorization. In the horizontal (I-J domain) this code is clearly data-parallel, while in the vertical (K domain) it has a loop-carried dependency and thus needs to be executed sequentially.

Vectorization of this code for CPU is highly dependant on chip model-, generation and compiler, mainly due to the sequential loop and the data dependant branch present. Branches, backward jumps and complex data dependencies can easily break vectorization, e.g. resulting in up to 32x slowdown (single precision) compared to the peak performance for the CPU cited in Table 1.1. Thus, for consistent vectorization across compilers and CPU architectures, a loop reordering will generally be required, with one of the data-parallel dimensions (I or J) becoming the innermost loop. A storage reordering is thus also required in order for the innermost loop to iterate over data that is contiguous in memory (which

is an important caching optimization, necessary to make efficient use of the available bandwidth). Multi-threading parallelization on the other hand is trivial for this example - a single OpenMP directive (here `!$omp parallel do`) applied to the outermost loop is sufficient for parallelization, assuming $ny > n_{cores}$.

In case of modern NVIDIA GPUs the CPU's vector length corresponds to the number of "CUDA cores" per "streaming multiprocessor", while the core count corresponds to the number of streaming multiprocessors per chip. From the programmer's perspective however, these two levels of parallelisms are treated in a unified way, resulting in significant simplification compared to vectorization-aware programming on CPU. A CUDA core, while less independent than a CPU core, allows arbitrary branches, backwards jumps (inner loops) and data dependencies, with the exception of loop carried dependencies in the domain over which the code is parallelized (here the I-J domain). Each CUDA core can thus be thought as executing its own thread (in contrast to computational units on CPU that are part of a vector and always need to be instructed as a whole group). GPU threads are grouped into so-called "warps" - a group of thread that is executed in lock-step. Branches are handled in the warp scheduler by letting threads wait that do not participate in the active branch. Thus, the worst case performance loss for a branch inside the kernel code is the warp size (32 in case of single precision) in case that particular branch is distributed "sparsely" across the data set in a very specific way where exactly one CUDA core per warp is ever active. In grid-based applications however, such as the thesis-motivating ASUCA weather prediction model, branches tend not to be sparse in such a way.

As a result of this unification of vectorization parallelization across instruction schedulers, the following GPU implementation, here using OpenACC directives, will allow the usage of all available CUDA cores, assuming $ny \cdot nx \gg n_{cores}$:

Listing 1.2: Sample kernel on GPU using OpenACC.

```
!$acc kernels
do j = 1, ny
  do i = 1, nx
    if (b(i,j)) then
      do k = nz - 1, 1, -1
        a(k,i,j) = a(k+1,i,j) * exp(- sqrt(gamma(k,i,j)) * tau)
      end do
    else
      do k = 1, nz - 1
        a(k,i,j) = 0.0d0
      end do
    end if
  end do
end do
```

The `kernels` directive is declarative and instructs the compiler to generate code where the I-J loop body (which the compiler can pick up automati-

cally as parallelizable, or it can be instructed to do so through the use of additional `!$acc loop independent` directives) is scattered across all available CUDA cores. This will create a grid of $ny \cdot nx$ threads, subdivided into thread blocks, the smallest unit of threads that is loaded in ensemble onto the streaming multiprocessors. Choosing the blocking dimensions manually is a common way to optimize the generated code further, which can be done through the use of `vector` clauses applied through `!$acc loop` directives.

Implementing the code shown in Listing 1.1 using the lower level CUDA, entails separating it into device code (the I-J loop body, including branch and inner K-loop, with an additional mapping of block- and thread indices to i- and j indices), resulting in a so-called “kernel”, and host code (setup of a CUDA grid, including blocking, as well as a kernel scatter call to parallelize the kernel across the CUDA grid). The resulting code is significantly more verbose, however it also corresponds more closely to the operations executed in hardware. Especially the semantic difference in parallel- and sequential loops becomes more apparent in this version of the code.

Data Movement

While the GPGPU programming model allows for a straightforward implementation of data parallel code when compared to a vector-aware CPU version, there is one significant complication in order to create efficient GPU applications: The discrete GPUs that are relevant to this work are equipped with their own built-in working memory, in addition to the host system’s memory, in order to allow for a high bandwidth as shown in Table 1.1. While GPU vendors have been working on application code agnostic software- and hardware solutions to the problem of data synchronization between the two memory systems, it is generally still the case that GPU applications need to be aware of the memory state in order to achieve the highest performance. Consider a “naive” implementation, where the above kernel resides in a routine that is unaware of GPU memory, as shown in Listing 1.3:

Listing 1.3: “Naive” routine specification for the sample kernel

```
real(8), intent(inout), dimension(nz,nx,ny) :: a
real(8), intent(in), dimension(nz,nx,ny) :: b, gamma
real(8), intent(in) :: tau
integer(4) :: i, j, k

!$acc kernels
do j = 1, ny
  do i = 1, nx
    ...
```

In this case each call of the kernel will lead to the data (`a`, `b`, `gamma`) being copied from- and to the host memory, using a bus that is much slower than

the bandwidth available between the GPU and its working memory, thus giving up on a major advantage of the GPU. Modern GPUs however have a working memory that is large enough to hold a considerable problem size - 16GB in case of the P100 outline in Table 1.1. It is thus very beneficial to keep- and reuse the application’s data on the GPU’s fast working memory. Doing so manually with OpenACC can be done by adding device state information to the `!$acc kernels` directive and managing the data movement separately using a `!$acc data` directive that is applied to a larger region of the problem - in case of regular grid simulation software this typically wraps the simulation’s time integration loop. In the example of Listing 1.3 it entails adding the clause `present(a,b,gamma)` to the `kernels` directive, as well as `copy(a,b,gamma)` to the simulation’s data region (i.e. the code region encompassed by an `!$acc data` directive). In addition, `update` directives are used to update the data on either device or host in case of intermediary input- and output operations, such as storage and network transmission of data between cluster nodes.

Using the lower level CUDA, the same can be achieved by explicitly specifying the data to reside on the GPU through the attribute `device` added to the specification part in each routine, and managing the data copying explicitly using CUDA’s library functions. A newer method that can be efficient, at least in theory, since the release of “Pascal” generation NVIDIA GPUs (2016), is the use of “managed” data, also called “unified memory” in vendor terminology. In this case the GPU drivers and hardware will manage the data movement in a way that is more efficient than the naive implementation shown in Listing 1.3 - for each memory page requested either on device or host, the driver will determine whether a copy operation is necessary, thus allowing for implicit data regions. To access this functionality the data is declared using the `managed` attribute in the specification part of each routine.

Memory Coalescence

An important aspect to consider for any modern computer system is the method in which data is loaded from working memory to the processor’s internal cache. Since off-chip memory access is always a high latency operation compared to computational instructions, such operations are always throughput optimized, i.e. the chunks of data loaded for each operation are made large enough to occupy the available bandwidth for a given memory bus. When a single 32- or 64 bit value is requested by an application, the processor typically exchanges an entire cache block with data that is adjacent to the requested value. Such caching architectures are thus optimized for the access of values that are adjacent to each other in working memory. Memory access patterns in applications are then of major importance - so-called “random access” offers only a fraction of the bandwidth available to “sequential access”.

On GPUs, the memory architecture is optimized for a particular mode of operation: So-called “coalesced” memory access occurs when threads inside a warp access data in the same order as their indexing sequence within the warp. In practical terms this means that the first thread index should always be mapped to the dimension in which data is stored with unit stride in working memory, i.e. a unit incrementation of the first thread index corresponds to a unit incrementation of the array index in case of a “flat” array (an array which is stored as a single contiguous memory region, regardless of the number of array dimensions, which is the case for all built-in arrays in Fortran). Reconsidering Listing 1.2, this is not the case - the *k*-index is accessed with unit stride, while the *i*-index is mapped to the first thread index. This code is thus very bandwidth inefficient. To correct this, since the *k*-loop cannot be executed in parallel, the arrays are required to be reordered to I-J-K order to allow for coalesced memory access.

1.2.3 Concluding Remarks

Even with CPUs closing the gap in computational performance per unit of expended energy, there are distinct advantages that remain in favor of GPU computing: The methods with which data parallel codes require optimization to allow for a significant fraction of peak performance is more straightforward and the outcome is more predictable - largely thanks to a significantly higher memory bandwidth and a unification of vectorization and multi-threading in the programming model.

As is always the case with specialized co-processors, there are types of applications that cannot benefit however, such as applications that are dominated by latency due to a low grade of parallelism, as well as applications with memory access patterns that can be cached on CPU, but cannot be coalesced on GPU.

1.3 Numerical Weather Prediction

Numerical weather prediction (NWP) is among the most demanding computational tasks of humankind. Charney, Fjörtoft and Von Neumann pioneered the field in 1950 with their landmark paper on how to successfully apply then novel computing technology in order to predict weather numerically [12]. With a multitude of non-linear effects to account for in many orders of magnitude of scale in both space and time, NWP has remained a “grand challenge for computing”, staying at the forefront of adopting cutting edge hardware technology [13]. Predicting future weather in an increasingly precise fashion has thus occupied many great minds, as doing so successfully is of immediate value for humankind, increasing safety and efficiency of the many human activities that are exposed to the elements of our home planet.

As part of our work we were given the opportunity to improve a weather model of national importance to Japan, a highly motivating proposition. In this section we start by giving an introduction to the fundamentals of weather prediction and its numerical treatment, followed by a discussion of the motivation for using GPU in NWP, an introduction to ASUCA, the Japanese weather model whose GPU implementation has motivated this thesis, as well as key problems that arise in this context.

1.3.1 Fundamental Theory and Numerical Methods

The problem of weather forecasting was first formulated comprehensively by Vilhelm Bjerknes in 1904, as the following set of equations:

1. the three hydrodynamic equations of motion, giving differential relations among the three velocity components, density and air pressure,
2. the continuity equations for air and water, i.e. conservation of mass,
3. the equation of state for ideal gases,
4. the first law of thermodynamics, i.e. conservation of energy.

Bjerknes postulated that this problem, as a set of seven equations with seven unknowns combined with suitable boundary conditions for top and bottom of the atmosphere, is solvable [14]. Due to the complexity of the initial condition as well as the non-linearity of the equations, analytic solutions are not practical however. Weather prediction is therefore assisted by iterative numerical methods, i.e. through discretization of space and time. Lewis Fry Richardson pioneered this approach during the First World War by using human computers (unsuccessfully however due to numerical instabilities [15]), his seminal paper on NWP was later published in 1922 [16]. Courant, Friedrichs and Lewy's work on numerical stability in 1928 [17] was an important breakthrough to get around the problems that plagued Richardson, preparing Charney, Fjörtoft and Von Neumann to develop the first operational NWP by using automatic computers after the Second World War [12].

Today's commonly used numerical solver methods for operational NWP are the finite-difference method (WRF [18], COSMO [19]) and the finite-volume method applied on a regular grid, i.e. using cells with identical geometry and volume (NASA-NCAR fvGCM & CAM [20], ASUCA [6]). Due to stability, forward Euler schemes and lower order Runge-Kutta iteration methods are problematic - third-order Runge-Kutta is thus the most commonly used scheme.

Computational time restrictions due to the nature of a prediction (i.e. computational time must be significantly lower than simulated time) are fundamental

to the way in which the NWP problem is discretized and solved. The hydrodynamic equations of motion are formulated as Navier-Stokes equations in different forms, trading motion approximations with gains in computational time [21]. In “free atmosphere”, i.e. atmospheric regions with sufficient distance to boundary layer effects, advection, diffusion as well as sound-, gravity- and Rossby waves (Coriolis induced) are the main phenomena of interest. While sound waves do not have any meteorological relevance, they become relevant for the stability of solutions for large scale, high-Mach-number atmospheric flows such as typhoons and jet streams [21]. We differentiate between the following compressibility models:

- In fully incompressible systems, the density of air is assumed to remain constant for isothermal pressure changes - only expansion and contraction with diabatic heating and cooling is accounted for. This eliminates both sound- and gravity waves and thus allows for the largest time steps - accepting large errors however.
- Boussinesq systems allow for compression only due to acceleration by gravity, thus allowing for gravity waves while still eliminating sound waves.
- Anelastic systems are derived under the assumption that potential temperature changes are small and that vertical oscillations are restricted in a statically stable environment (i.e. the timescale is set by the buoyancy frequency). Ogura first formulated this model in 1962 [22]. This approximation however introduces intrinsic errors and, in their original formulations, these systems fail to conserve energy. Various improvements, especially regarding energy conservation, have been achieved by Durran [23], Lilly [24], Bannon [25] as well as Arakawa and Konor [26] among others. Sound waves are still filtered under these assumptions.
- Hydrostatic models are horizontally compressible but vertically incompressible, i.e. vertically propagating sound waves are eliminated [27].
- A fully compressible system is the most computationally intensive as it requires either very short timesteps or intricate implicit iteration schemes to resolve sound waves. WRF was the first fully compressible non-hydrostatic NWP model and subsequently has had wide-ranging effects on the NWP community - since it is Open Source, its algorithms have been incorporated and extended in many other weather models [28].

Due to the different propagation speeds of the mentioned phenomena and the resulting requirements on time step - and thus computational requirements - NWP models use numerical techniques to optimize for lower computational costs while achieving stability requirements by treating the mentioned phenomena separately. The most commonly used approximations and techniques in this family,

besides the compressibility approximations mentioned above, are the following [29] [27]:

- Spherical-geopotential approximation assumes gravity to have no horizontal component.
- The depth of the atmosphere is shallow compared to the horizontal extent of meteorological phenomena, thus vertical motion is assumed to be much smaller than horizontal motion. The grid used for spatial discretization is therefore chosen with a finer vertical and a coarser horizontal resolution - the ratio of horizontal spacing to vertical spacing between grid cells is called “aspect ratio” in this context. Gravitational effects are assumed to be constant with distance from the earth’s surface.
- Due to smaller timesteps required to resolve vertical sound- and gravity waves explicitly, vertical propagation tends to be solved implicitly instead. A common technique for models solving the compressible Navier-Stokes equations is thus to employ horizontally explicit, vertically implicit iteration schemes.
- Advection- and Coriolis effects being slower, the timestep with which relevant variables are updated is further slowed down by employing time-splitting schemes, i.e. having a separate shorter timestep with which sound- and gravity waves are solved.

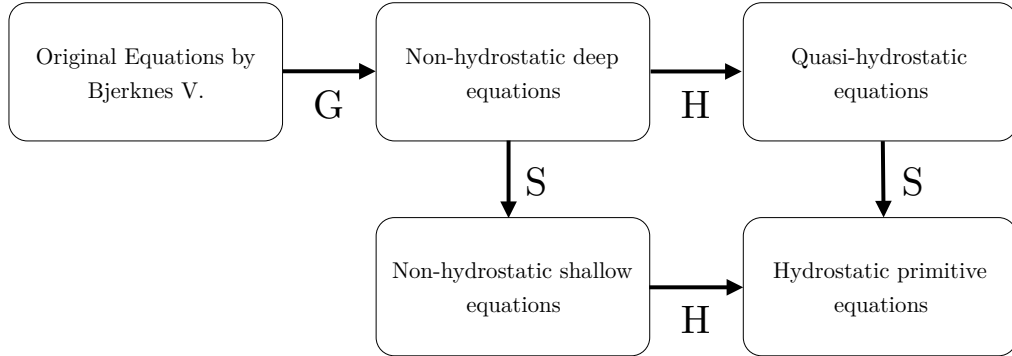


Figure 1.2: Interrelations of atmospheric models with respect to their approximations.

White et al. 2005 gives an overview of the different approximations with regards to motion, shown in Figure 1.2 [30]. The transitions G, H and S represent spherical-geopotential, hydrostatic- and shallow-atmosphere approximations, respectively. Which of these approximations are used, depends mostly on the scale of the model. Generally, the more computational performance is available, and

the smaller the scale, the less approximations are used. Regional NWP models have reached the limits of hydrostatic approximation around a decade ago [31] and for global weather prediction the shift towards non-hydrostatic models is now on-going [32]. Most models still use the spherical-geopotential as well as the shallow-atmosphere approximations however [27]. Davis et al. from the UK Met Office deserve mentioning here, as they have introduced a fully compressible deep-atmosphere non-hydrostatic weather model already in 2005 [33]. We anticipate that due to growing stability issues with higher grid resolutions and thanks to the steadily increasing computational performance, NWP models in the coming years will increasingly resemble the original vision of Vilhelm Bjerknes.

1.3.2 Microscale Phenomena

Today’s weather prediction models have achieved resolutions in the range of one to two kilometers between grid cells, allowing the simulation of mesoscale phenomena as part of the dynamical solvers (commonly referred to as “dynamical core”). Smaller scale phenomena (from molecular scale to hundreds of meters, referred to as microscale) are estimated through various models called “physical processes”, in a method called “parametrization”: physical processes generate tendency values that are used in order to account for microscale phenomena in the next grid cell average.

Irrespective of the achieved grid resolution, more fine-grained air motion is always present in atmosphere, making the solution of prognostic equations for such turbulence necessary to increase accuracy. Incorporating these turbulence effects creates unclosed equations, as found by Keller and Friedman in 1924 [34] that must be dealt with by decoupling turbulence terms from the rest of the system as a parametrization. Eddy viscosity models treat this part of the problem [35]. Other microscale phenomena requiring specific treatment are boundary layer effects due to terrain proximity, solar radiation, light absorption- and reflection as well as formation and movement of water substances. This section outlines the modelling of these effects.

Cloud Microphysics and Precipitation Pioneering work in numerically simulating the life cycle of water substances in clouds has been achieved by Berry, Ogura and Orville, among others, throughout the 1960s [22] [36]. Berry’s method of “binning” the particle spectrum into sizes or masses was replaced with bulk schemes where particles are categorized according to function, resulting in predictions for characteristics such as mass (mixing ratio) and concentration of particles [37]. Kessler was able to simplify warm cloud and warm rain microphysics (i.e. processes for clouds remaining below freezing altitude) using this method by reducing the effects into a series of parametrized relationships in 1967

[38] [36]. Ogura and Takahashi extended this work in 1971 by building a model for the life cycle of thunderstorms that combines a dynamical solver for vertical motion (coupled with mass- and energy continuity) with a parametrization for the condensation of water vapor to produce cloud droplets, the conversion of such droplets to raindrops, glaciation, sublimation of water vapor, melting of ice crystals as well as evaporation of these substances, thus creating a cold cloud model. Precipitation from such cold clouds, i.e. rain, snow and graupel, was modelled later, e.g. by Murakami in 1990 [39].

Planetary Boundary Layer and Surface Ground surface effects influence the atmosphere up to an altitude of one to two kilometers. This atmospheric layer is called the “planetary boundary layer” (PBL). It is divided into two parts: the surface boundary layer, up 20 to 50 meters high, and the Ekman layer above that. The surface boundary layer is generally modelled in two dimensions - its atmospheric variables are set equal to those on the ground. Incorporating turbulence in the planetary boundary layer into a weather model brings the same closure problem (Keller and Friedman 1924 [34]) as described above. For the surface boundary, the mixing length theory developed by Prandtl [40] is sufficient to solve the closure problem, while in the Ekman layer additional modelling is required, such as Mellor-Yamada type PBL models [41].

Radiation Heating of the atmosphere, apart from ground effects, is a two-part problem: Incoming ultraviolet, visible light and infrared energy from the sun power the earth’s atmosphere on one hand, reflective and absorptive effects from larger aerosols interact with the involved EM waves on the other. These two families of phenomena are often distinguished as “shortwave”- and “longwave” radiation in weather models, respectively. Various parametrization schemes are used for NWP, such as Briegleb [42] and NASA Goddard [43]. The most important aerosol to consider is water - various parametrization schemes thus treat absorption and reflection of longwave radiation in clouds, such as Goody 1952 [44].

1.3.3 Cloud Resolution

Clouds are a meteorological phenomenon that stretch multiple orders of magnitude, from tens of meters to hundreds of kilometers in scale. Simulating clouds and cloud systems in the context of weather prediction is thus a very computationally demanding task. On one hand the horizontal grid resolution is required to be high enough to simulate the evolution of significant cloud formations, on the other hand the horizontal extent of such grids needs to be large enough to account for mesoscale cloud systems [45]. Finally, in order to predict weather in an operational setting, the time-to-solution is required to be at least one order

of magnitude lower than the simulated physical time, imposing strict limits on the amount of computing resources that are allowed to be spent per grid cell and time step.

In order to achieve and improve cloud resolving capabilities, weather prediction models have therefore been driven towards increasingly high horizontal resolution, i.e. lower grid aspect ratio. A cloud resolving model (CRM) is considered to have a resolution of 4 kilometers or less in each horizontal dimension [46]. The goal is to acquire space- and time varying fields for temperature, water vapour, cloud water, precipitation as well as horizontal- and vertical wind motion variables on scales that allow the resolution of deep convective clouds, e.g. cumulonimbus clouds due to their high impact on human activity as well as overall weather conditions [45]. Integrating CRM capabilities into operative numerical weather models can therefore be considered one of the most important goals for meteorological agencies over the last twelve to fifteen years.

In older models, since the atmosphere is thin compared to the horizontal extent, the hydrostatic approximation was used in order to significantly reduce computational complexity. This approximation assumes the vertical pressure gradient to be the product of density times the gravitational acceleration. This is valid for high aspect ratios where vertical momentum from non-gravitational effects can be neglected. With increasing horizontal resolution this approximation has become invalid however, i.e. non-hydrostatic models are required for cloud resolution [47].

1.3.4 GPUs Applied to Numerical Weather Prediction

With weather prediction being one of the grand challenges for computing, the rise of GPUs in HPC has been accompanied with researchers starting to apply these co-processors to NWP. In 2004, Fan et al. published a landmark paper for the use of a GPU cluster as a Lattice Boltzmann type solver, thus demonstrating GPUs as an instrument capable to be applied to computational fluid dynamics, with implications for adjacent fields such as NWP [48]. In 2006 NVIDIA released CUDA, the first vendor supported general purpose GPU programming interface [49]. In 2008, Michalakes et al. reported their use of CUDA based GPU programming for the WRF Single Moment 5-tracer (WSM5) microphysics [50], thus showing an early application in NWP. In 2011 Shimokawabe et al. demonstrated a near-production weather model running completely on 3990 GPUs [51].

Given the underlying numerical problem described in Section 1.3.1 and the pressure towards implementing CRM as described in Section 1.3.3, modern numerical weather prediction has some commonalities across models:

- With horizontally-explicit vertically-implicit time discretization, the hori-

zontal domain is always parallelizeable. The vertical domain can be partially parallelized for those equations that are solved explicitly in the vertical.

- Due to the push towards horizontal resolution increase, dynamical cores tend to dominate heavily in terms of the budgeted computational time. Since finite-difference- and finite-volume based solver methods tend to have a low arithmetic intensity, i.e. the number of FLOP per byte loaded from memory is low, NWP tends to be bounded by the memory bandwidth of the underlying compute architecture.
- Memory access patterns in finite-difference and finite-volume methods are highly regular, i.e. it is comparatively simple to find mappings that result in sequential- and coalesced memory access.

Given the strengths of GPUs discussed in Section 1.2, these characteristics all point towards NWP being a “GPU-friendly” problem - a highly parallel problem bounded by memory bandwidth, using regular memory access patterns.

1.3.5 ASUCA NWP Model

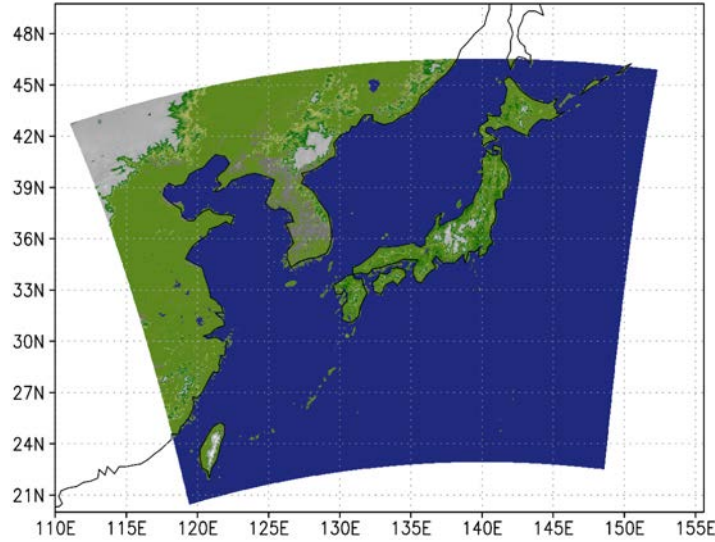


Figure 1.3: ASUCA’s model simulation boundaries.

The regional scale numerical weather prediction model “ASUCA” (“Asuca is a System based on a Unified Concept for Atmosphere”) is one of the main operational forecast models in Japan [6]. It is developed by the Japan Meteorological Agency (JMA) and used in production since 2014, covering all of Japan as well as relevant ocean areas and surrounding landmasses in East Asia (depicted

in Figure 1.3) on a rectangular grid with two kilometer resolution, generating nine-hour-forecasts every hour [5].

ASUCA is a fully compressible, non-hydrostatic model. For spatial discretization ASUCA employs a terrain-following, Arakawa-C-type rectangular grid. The model is based on general coordinates, allowing transformations for both latitude / longitude- as well as Lambert conformal conic projections. By convention, its horizontal dimensions are named I and J, with K being the vertical dimension [5] [52].

Dynamical processes are solved by using the finite-volume method. Since various physical processes require a higher vertical resolution, a non-homogeneous vertical cell separation order of 10^3m is applied, in contrast to a 2-kilometer-resolution applied in the horizontal domain. Since in an explicit method the smallest spatial dimension determines the time step, a horizontally-explicit vertically-implicit (HEVI) time stepping method is employed. Atmospheric waves, advection as well as the Coriolis acceleration are solved using a time splitting method as proposed by Wicker and Skamarock in 2002, combining a third-order Runge-Kutta time integration scheme using a long time step for slower processes (advection, Coriolis) with a secondary third-order Runge-Kutta iteration for sound- and gravity waves [53]. Vertical advection of water substances (i.e. precipitation) is solved using a separate time step for each column, based on the Courant-Friedrichs-Lewy convergence condition [6] [17]. ASUCA is structured as a dynamical core interfacing with physical processes through tendency variables, as is common in NWP (see also the discussion in Section 1.3.2).

ASUCA's dynamical core is bounded by memory bandwidth, as is common for finite-volume and finite-difference based spatial discretizations for dynamical systems. It also constitutes a significant majority of the runtime in operational settings [54] [55]. As described in Section 1.3.4, GPUs are thus an attractive target architecture, with a memory bandwidth that is typically 5 to 7 times higher than Intel Xeon architectures of a similar generation.

ASUCA is a protected asset of the Japanese government and can not be published at this time - in this thesis we thus refer to code snippets to discuss the implementation relevant to this thesis.

1.3.6 Key Challenges for GPU Computing in NWP

Due to the employed spatial discretizations being too coarse grained for many important microscale phenomena, NWP involves applying physical parametrizations resulting in tendency values, as described in Section 1.3.2. In order to increase accuracy this part of the code tends to be in constant flux - i.e. the

treatment of physical processes is a very active field. For this reason, the implementation of NWP models typically cannot be separated cleanly into tasks for domain scientists and tasks for computer scientists. It is of high importance that domain scientists are able to keep full code ownership, which has implications for ease-of-use and the required familiarity with the underlying frameworks and libraries. Since GPU computing requires some far-reaching changes to the software, as discussed in Section 1.2.2, keeping these aspects in mind when designing a solution is crucial for a successful deployment of this technology.

This implies the following tangible characteristics to strive for when applying GPU computing to existing NWP code:

1. The same code is to be applicable to all relevant hardware architectures, i.e. code divergence is to be minimized.
2. Revised codes are to be as familiar as possible to the domain scientists that have been working with the previous codes for many years. Since NWP codes are written almost exclusively in Fortran today, Fortran language based solutions are preferred.

In this Section we discuss specific problems that typically arise for GPU ports of regular grid NWP models when given such requirements.

Parallelization Granularity

Since GPU programming models each vector component as its own thread, it supports a very high number of parallel threads while having low thread switching overhead, accepting the trade-off of a high latency for completing individual tasks, as discussed in Section 1.2. This high latency, as well as the way in which the streaming processor schedulers work, results in a very high number of threads being optimal, preferably a multiple of the available computational function units (i.e. tens of thousands of threads or more). Register pressure is thus a major factor for the optimization of GPU code, since the number of utilized registers scales linearly with the number of active threads, i.e. the less registers a code uses, the more threads can be loaded simultaneously, which increases the code's efficiency up to a certain saturation point. Furthermore, GPU threads until now do not support real function calls involving context switches. Instead, function calls inside kernels are inlined, thus the registers occupied with their scope is never freed during the lifetime of a thread.

These characteristics point to an important distinction in the GPGPU programming model: A fine-grained parallelization is strongly preferred or even necessary in order to limit the register pressure present in each kernel. Code written with CPUs in mind is often too coarse-grained in order to be applied to GPU

as-is. In the context of NWP this is especially common for physical processes, i.e. code with relatively complex computations but without dynamically taking horizontally neighboring data into account, thus allowing CPU-oriented code to minimize thread switching overhead and optimize for caching by maximizing the amount of work done for a vertical column of data.

Offering a solution to allow for differing task granularities on the same code is thus an important feature for hybrid CPU/GPU programming frameworks.

Memory Layout

There are two major aspects in which the memory layout on GPU differs from that on CPU:

Stride-1 Access On CPU, the memory layout is generally chosen such that the fastest varying dimension is mapped to stride-1 access, in order to increase cache hits. GPUs on the other hand require the first parallel dimension to be mapped to stride-1 in order to coalesce the memory operations, as discussed in Section 1.2.2. In case of NWP these are not necessarily the same dimensions. Taking ASUCA as an example, the vertical dimension K is the fastest varying one, even though it is executed sequentially in most kernels. The original codebase thus employs KIJ memory order, with I and J being the two horizontal dimensions. In Section 4.2.3 it will be shown that for a reduced weather application, KIJ order leads to a 7.7x slowdown on GPU (versus IJK) while IJK ordering leads to a 35% slowdown on CPU (versus KIJ).

Privatization To create a more fine-grained parallelization for GPU (as discussed earlier), kernel fission is to be employed. This in turn requires thread-local data structures and passed-in data slices to be extended in the parallel domain (IJ). Many scalars thus become 2D-arrays. Many 1D- become 3D-arrays.

Thus, in order to make performance portability possible, the memory layout is required to be changeable and extendable, depending on the target architecture and the parallelization granularity.

1.4 Related Work

In this we discuss previous or ongoing work that is related to this thesis from the application perspective. For a more in-depth treatise on the framework perspective and how our own solution compares to available code hybridization methods, please refer to Chapter 5.

1.4.1 ASUCA GPU Port by Shimokawabe et al.

In 2010, Shimokawabe et al. completed a GPU-only CUDA C port of the dynamical core and part of the physical core of ASUCA. A performance model for the five most relevant kernels was established as part of this work. 14 TFLOP/s were achieved in single precision using 528 Tesla S1070 GPUs [54].

In 2011, 24 GFLOP/s was achieved in double precision for ASUCA dynamical core on a single NVIDIA Fermi based Tesla M2050 GPU on TSUBAME 2.0, a 6-fold speedup when compared to the system's Xeon X5670 CPUs running the original Fortran code. This work also included extensive work regarding the optimization inter-node communication and an overlapping method in order to run communication and computation in parallel. By applying weak scaling (i.e. increasing the grid size together with the number of GPUs), up to 145 TFLOP/s were achieved in single precision on 3990 Tesla M2050 GPUs [51].

In 2014, the previous work was generalized by employing a stencil library and DSL based on C++ templates that generates either CUDA kernels or CPU code. Employing previously researched optimization methods to this more generalized method, up to 209.6 TFLOP/s were achieved for ASUCA in single precision on 4108 Tesla K20x GPUs [52].

1.4.2 COSMO

In 2014, MeteoSwiss' Fuhrer et al. proposed a mixed approach for COSMO, another structured grid regional weather and climate model used in operation in several European nations. In this mixed approach, a C++ stencil library and DSL called "STELLA" is used for the dynamical core, while a "less disruptive" OpenACC based method is applied to the physical processes in order to retain most of the original Fortran codebase [56]. In COSMO's case, the physical processes already expose fine grained parallelism and kernel fusion was employed in some instances in order to optimize for less kernel launches by Lapillonne et al. [57]. Thanks to the code refactoring a speedup of 1.5x was achieved on CPU. By employing GPU code generation an additional speedup of 2.9x was achieved (comparing Tesla K20x to 8-core Xeon E5-2670 on Piz Daint) [56]. STELLA has subsequently been generalized for use cases beyond COSMO and rectangular grids under the new name "GridTools" [58].

1.4.3 Weather Research and Forecasting Model (WRF)

The U.S. National Center for Atmospheric Research's "Weather Research and Forecasting model" (WRF) is the most widely used weather model worldwide [59]. It too uses a structured grid. WRF consists of multiple hundreds of thousands of lines of Fortran code. GPU ports known to us have been only partial,

leading to heavy pressure on the memory bus between CPU and GPU and thus a lower speedup compared to full GPU ports of other weather models [60]. The following partial GPU ports have been achieved among others:

- in 2008, Michalakes and Vachharajani ported the “WSM5” cloud micro-physics to CUDA C, achieving an overall speedup of 1.25x for a benchmark workload [59],
- in 2010, Ruetsch et al. ported the longwave radiation physics to CUDA Fortran [61],
- in 2012, Mielikainen et al. ported the Goddard shortwave radiation scheme to CUDA C [62],
- in 2013, Vanderbauwhede and Takemi ported the advection scheme to OpenCL and integrated it together with the Fortran based WRF. Doing so, an overall speedup of 2x was achieved [60].

1.4.4 Non-hydrostatic Icosahedral Model (NIM)

In 2010 the dynamical core of the Non-hydrostatic Icosahedral Model (NIM) developed at the U.S. National Oceanic and Atmospheric Administration (NOAA) was ported to GPU by Govett et al. by using a Fortran-to-CUDA-C transpiler called “F2C-ACC” [63]. In 2014 this work was compared to the industry standard “OpenACC” on Titan using Tesla K20x GPUs, showing 77% higher performance with F2C-ACC compared to Cray’s OpenACC Fortran implementation and 220% higher performance compared to PGI’s OpenACC. Overall however the GPU implementation was not able to achieve a speedup compared to CPU due to the missing physical processes and the thus high communication overhead [64]. Since then, performance of both hardware and software have improved significantly however, resulting in a speedup of 2.5x when comparing NVIDIA GPUs to CPUs and 2.0x when comparing Xeon Phi to the same CPU in 2017. Govett et al. have also reported that OpenACC’s performance issues and bugs have been fixed and F2C-ACC has thus been fully replaced with the industry standard [65].

1.4.5 Accelerated Model for Climate and Energy (ACME)

Norman et al. have investigated methods to create a hybrid GPU/CPU codebase for the U.S. Department of Energy’s “Accelerated Model for Climate and Energy” (ACME). A previous CUDA Fortran based port was dropped due to issues with having to maintain two separate codebases for CPU and GPU. For this reason, OpenACC was evaluated as a replacement, but limitations with resource pressure and inlining when porting large kernels to GPU were found. It was

concluded that such large kernels need to be split up to be feasible on GPU with OpenACC, which aligns with our findings about ASUCA's physical processes (see also Chapter 3). Since CPUs perform better with fused loops for caching reasons, for a pure OpenACC based solution it was found that two separate code versions are required for CPU and GPU, resulting in similar maintainability issues as with a pure CUDA Fortran port [4].

A Method for High Productivity GPU Porting

In Chapter 1 GPU computing has been introduced, as well as its advantages and challenges when applying it to numerical weather prediction. In this chapter a solution will be proposed that meets these challenges when applying it to legacy numerical applications for structured grids written in Fortran. This solution will from here on be called “Hybrid Fortran” - a Fortran language extension and code transformation framework that enables a low code divergence, hybrid GPU/CPU implementation of structured grid applications through flexible parallelization granularity, a memory layout transformation- and a kernel generation scheme.

In Section 2.1 the motivation for introducing a new framework is summarized, while in Sections 2.2 and 2.3, Hybrid Fortran’s approach to parallelization (granularity) as well as memory layout and movement are described, respectively, using the NWP model ASUCA as a usage example. In Section 2.5 a treatise is given on the code transformation scheme that is applied. Finally, concluding remarks are given in Section 2.6.

2.1 Motivation

Problems with GPU computing, when applied to NWP, have been outlined in Section 1.3.6: Memory layout and parallelization granularity are common to diverge between CPU and GPU. In this section a brief overview of the solution space with regards to existing methods is given. For a more in-depth treatise of existing methods, as well as Hybrid Fortran’s uniqueness and contribution to the overall field, please refer to Chapter 5.

In terms of the port productivity for porting existing applications, directive based methods have obvious advantages over stencil DSLs: existing codes do not need to be rewritten (e.g. refer to Listing 1.2 in Section 1.2.2). The performance

of a “naive” GPU port with directives, even taking data regions into account, may be unsatisfying however, due to the discussed memory layout- and granularity problems. Libraries that are based on a stencil DSL for their user-code solve at least one of these issues: Memory layout is abstracted and can thus be transformed for different hardware architectures - sacrificing the reuse of legacy code however.

Granularity issues are more difficult to solve for the general case - transforming the amount of work per thread as defined in code involves a transformation of the call tree, awareness of inputs and outputs of each code region, as well as the ability to privatize or globalize thread-local data. In some existing methods this transformation is assisted one-way, transforming automatically from fine-grained to coarse-grained kernels (called “kernel fusion”), while in other methods the task of optimizing for code granularity is entirely delegated to the programmer, thus imposing limits on code reuse and productivity, as well as requiring code divergence. A third possibility is to ignore coarse-grained parts of the code and only port the portion that already has a suitable code structure for GPU - an approach that leads to significant decreases in performance due to host-to-device communication.

Hybrid Fortran is designed to allow for flexible high-throughput implementations with none of the aforementioned drawbacks. More tangibly, it allows

1. to delegate parallelization “boilerplate”¹ to the framework,
2. to transform the memory layout and
3. to define multiple parallelization granularities for the same code,

thus achieving an efficient GPU implementation without rewriting major parts of the computational code and keeping the code compatible to CPU.

2.2 Parallel Loop Abstraction

Consider the following kernel from JMA’s ASUCA reference implementation. As part of the dynamical core it is executed within a second order Runge-Kutta scheme, applied to atmospheric waves with high time resolution. It applies lateral and upper damping to ASUCA’s grid point values.

¹“Boilerplate” is an expression used in programming to describe code that is purely infrastructure-related (i.e. related to utilized libraries and programming languages or targeted hardware architectures) - in contrast to code that directly reflects logic or computations required to create the desired output.

Listing 2.1: Lateral and upper damping kernel applied to grid point values.

```

!$OMP PARALLEL DO
  do j = ny_mn, ny_mx
  do i = nx_mn, nx_mx
  do k = nz_mn, nz_mx
    dens_ptb_damp(k,i,j) = &
      & mratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,1) ) &
      & + tratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,2) ) &
      & - dens_ref_f(k,i,j)
  end do
  end do
  end do
!$OMP END PARALLEL DO

```

Using Hybrid Fortran we replace the OpenMP directives as well as the loop instructions to be parallelized for the finest granularity architecture (GPU) with our parallelization DSL:

Listing 2.2: Lateral and upper damping kernel, modified with Hybrid Fortran.

```

@parallelRegion{ &
  & domName(i,j), domSize(nx_mn:nx_mx,ny_mn:ny_mx), &
  & startAt(nx_mn,ny_mn), endAt(nx_mx,ny_mx), template(TIGHT_STENCIL) &
  & }
  do k = nz_mn, nz_mx
    dens_ptb_damp(k,i,j) = &
      & mratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,1) ) &
      & + tratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,2) ) &
      & - dens_ref_f(k,i,j)
  end do
@end parallelRegion

```

This shows an explicit distinction between loops that are treated as parallelizable (and are thus restricted in their access patterns, i.e. loop carried dependencies are not supported) and loops that are always executed sequentially. The attributes `domName` and `domSize` specify the relevant domain iterators and the domain size relevant to data objects accessed within the parallel region (this relevancy will later be discussed in more detail in Section 2.3). The attributes `startAt` and `endAt` explicitly state the region boundaries, which can be a subset of the domain size (however, if omitted, the domain size is also assumed as the region boundary).

For CPU targets, Hybrid Fortran generates an OpenMP code version very similar to the reference code shown in Listing 2.1², with multi-core parallelization applied to the outermost loop. For GPU targets it defaults to CUDA Fortran kernels (thus generating all the necessary host- and device code boilerplate and data copy operations) with an option to use OpenACC kernels with CUDA

²Privatization is the main difference: Hybrid Fortran generated OpenMP code uses “first-private” as the default policy with an explicit “shared” clause for all arrays used in the kernel.

compatible data structures (device pointers)³. The attribute `template` specifies a macro suffix used for the generated block size parameters - this allows a central configuration for the block sizes used in an application, rather than leaking this architecture-specific optimization to the user code in each kernel. If omitted, configurable default block sizes are used.

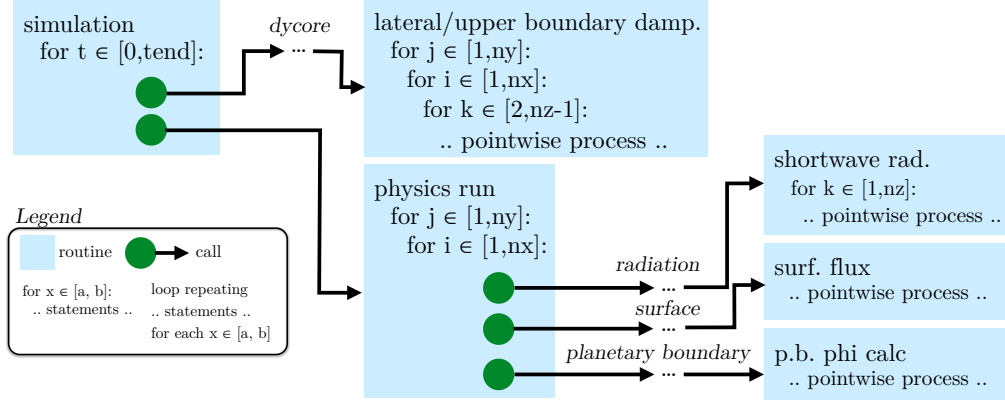


Figure 2.1: Simplified code structure of ASUCA.

The main advantage of this parallelization DSL is the following: replacing parallelizable loops with the `@parallelRegion` construct allows the user to specify multiple granularities in the same code. Consider ASUCA’s code structure, shown in simplified form in Figure 2.1. It shows two selected kernels and their embedding in the call graph - the lateral and upper boundary damping already discussed in this section, as well as the physics kernel. Many physical processes are called within this single kernel (of which three sample processes are depicted here). This code therefore has a very coarse granularity, which is problematic on GPU as discussed in Section 1.3.6.

With Hybrid Fortran one can solve this problem as follows: An additional `appliesTo` attribute in the `@parallelRegion` statement allows the user to selectively apply parallel regions to either CPU or GPU. Applying the parallelization at different granularities therefore becomes possible. Figure 2.2 shows the resulting code structure, with the physics run being split into many kernels for GPU while remaining a single coarse grained kernel for CPU.

³OpenACC is mainly used for reduction support - Hybrid Fortran does not automatically generate reduction kernels, however it supports the “reduce” clause, which is forwarded to the generated OpenMP or OpenACC kernels.

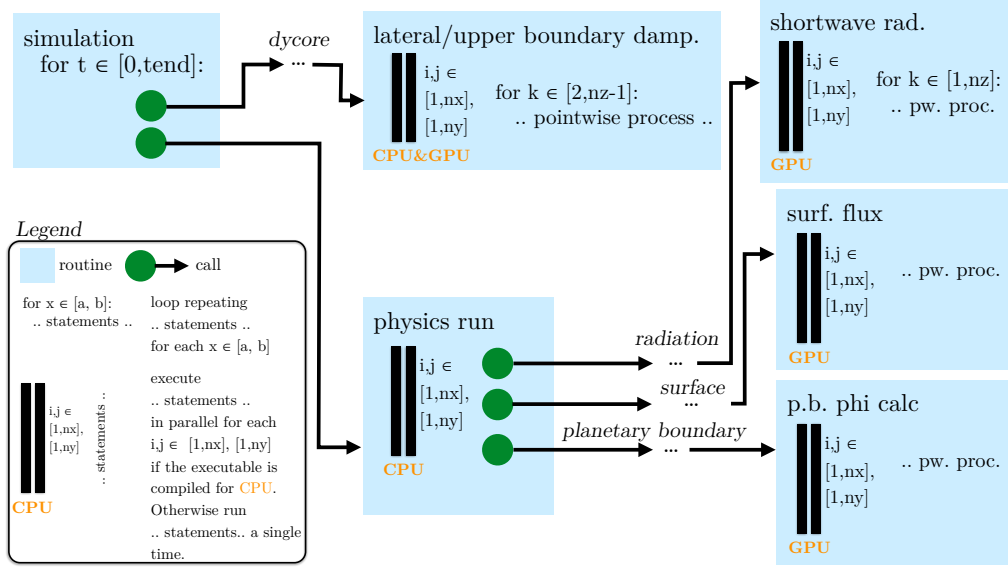


Figure 2.2: Simplified code structure of ASUCA using Hybrid Fortran.

2.3 Compile-time Defined Memory Layout and Device Data Region

As discussed in Section 2.1, it is necessary to consider two major aspects for implementing the memory layout: storage order on one hand and the compile-time defined granularity requiring a varying dimensionalities of data objects on the other. Hybrid Fortran uses an additional DSL extension to Hybrid Fortran, the `@domainDependant` construct, as a declarative way for the user to specify additional required information concerning data objects. This concerns memory layout as well as device memory operations, which will be discussed in this section.

2.3.1 Storage Order

Revisiting the code sample from Section 2.2, the following Listing shows the specification of the routine implementing the discussed lateral and upper damping kernel:

Listing 2.3: Routine implementing the lateral and upper damping kernel with Hybrid Fortran.

```
subroutine lateral_and_upper_damping()
  use ref, only : dens_ref_f
  use svar, only : dens_ptb_damp
  ! ... further imports omitted
  implicit none
```

```

@domainDependant{ &
    & attribute(autoDom, present), &
    & accPP(AT_TIGHT_STENCIL), domPP(DOM_TIGHT_STENCIL) &
    & }
dens_ref_f, dens_ptb_damp
@end domainDependant

@domainDependant{ &
    & attribute(autoDom, present), &
    & accPP(AT4_TIGHT_STENCIL), domPP(DOM4_TIGHT_STENCIL) &
    & }
dens_ptb_bnd
@end domainDependant

! ... initialisation of tratio_bnd and mtratio_bnd omitted
! ... kernel omitted (already shown in listing 1.2)
end subroutine

```

This shows the specification of the local module data object `dens_ptb_bnd` (density perturbation in the boundary layer) as well as the external module data objects `dens_ref_f` (reference density) and `dens_ptb_damp` (density perturbation in ASUCA grid).

The `autoDom` attribute is used to delegate the dimensions setup to the data object specification parser (which gathers this information in a separate pass from the source modules, here `ref` and `svar`), rather than having the user specify the dimensions explicitly again in the `@domainDependant` construct. The attributes `accPP` and `domPP` are employed to specify the macro names used to implement the dimension ordering for accesses and specification parts, respectively. These macros wrap all dimension lists in access expressions and specifications of respective data objects in the generated code. When `accPP` and `domPP` attributes are omitted, default macro names are used. In case of Listing 2.3 we use explicit macro names for the dynamical core since the default macros are already used with different assumptions for the physical processes (see the paragraph on “Dimensionality Changes” below).

2.3.2 Device Data Region

Similar to OpenACC, in Hybrid Fortran we implement data regions by adding state attributes to data objects. The `present` attribute, shown in Listing 2.3, indicates that the respective objects are located on the device in case of GPU compilation. Analogous `transferHere` attributes are used in the main simulation routine in order to instruct Hybrid Fortran to implement the memory copy operations to- and from the device, once at the beginning and end of the simulation. For dummy variables with specified `intent`, Hybrid Fortran will use the Fortran intent information to determine the correct copy operation⁴, which

⁴Simple examples of this feature can be found in <https://github.com/muellermichel/Hybrid-Fortran/blob/v1.00rc10/examples/demo/source/example.h90>.

minimizes the potential for bugs in comparison to OpenACC’s explicit `copyIn`, `copyOut` and `copy` clauses. Halo region updates, required for every timestep, are implemented explicitly in code sections guarded from CPU compilation.

2.3.3 Dimensionality Changes

Due to the compile-time defined parallelization granularity, discussed in Section 2.2, it is necessary to modify the dimensionality of data objects in certain cases in the source generation. This requires hints from the framework user. Consider the following surface flux code snippet:

Listing 2.4: Surface flux code snippet.

```
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run( &
    ! ... inputs and further tile variables omitted
    & tau_x_tile_ex(lt), tau_y_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(tau_x_tile_ex(lt) ** 2 + tau_y_tile_ex(lt) ** 2))
else
  tau_x_tile_ex(lt) = 0.0_r_size
  tau_y_tile_ex(lt) = 0.0_r_size
  ! ... further tile variables omitted
end if
! ... sea tiles code and variable summing omitted
```

Since this process is defined inside the call graph of the physics kernel, as shown in Figure 2.1, the relevant 2D- and 3D grid point values are already sliced and passed in as scalars or 1D-arrays, that is, data parallelism is not exposed at this level. Hybrid Fortran allows implementing this as a fine-grained kernel (as outlined in Figure 2.2) without modifying the computational user code, as demonstrated in the following snippet:

Listing 2.5: Surface flux code snippet with Hybrid Fortran.

```
@domainDependant{domName(i,j), domSize(nx,ny), attribute(autoDom,
  ↪ present)}}
tlcvr, tau_x_tile_ex, tau_y_tile_ex, u_f
@end domainDependant

@parallelRegion{appliesTo(GPU), domName(i,j), domSize(nx,ny)}
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run( &
    ! ... inputs and further tile variables omitted
    & tau_x_tile_ex(lt), tau_y_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(tau_x_tile_ex(lt) ** 2 + tau_y_tile_ex(lt) ** 2))
else
  tau_x_tile_ex(lt) = 0.0_r_size
  tau_y_tile_ex(lt) = 0.0_r_size
```

```

! ... further tile variables omitted
end if
! ... sea tiles code and variable summing omitted
@end parallelRegion

```

Using our parallelization DSL to provide additional dimensionality information, Hybrid Fortran is able to rewrite this code into a 2D kernel. Dimensions missing from the user code are inserted at the beginning of the dimension lists in access expressions and data object specifications. As an example, the expression `u_f(lt)` is converted to `u_f(AT(i,j,lt))`, employing the default ordering macro already mentioned in the paragraph “Storage Order”. Dimensions are extended whenever there is a match found for `domName` or `domSize` information between data objects and parallel regions within the same routine *or* in routines called within the call graph of the same routine. It is therefore necessary for Hybrid Fortran to gather global information about the application before implementing each routine.

2.4 Transformed Code

Revisiting Listing 2.5, the following code is generated when applying Hybrid Fortran with the OpenACC backend:

Listing 2.6: Surface flux code snippet after conversion with OpenACC backend.

```

!$acc kernels deviceptr(taux_tile_ex) deviceptr(tauy_tile_ex) &
!$acc& deviceptr(tlcvr) deviceptr(u_f)
!$acc loop independent vector(CUDA_BLOCKSIZE_Y)
outerParallelLoop0: do j=1,ny
!$acc loop independent vector(CUDA_BLOCKSIZE_X)
  do i=1,nx
    ! *** loop body *** :
    lt = tile_land
    if (tlcvr( AT(i,j,lt) ) > 0.0_r_size) then
      call sf_slab_flg_land_run(&
        ! ... inputs and further tile variables omitted
        & taux_tile_ex( AT(i,j,lt) ), tauy_tile_ex( AT(i,j,lt) )
        &
        & )
      u_f( AT(i,j,lt) ) = sqrt(sqrt(taux_tile_ex( AT(i,j,lt) )** 2 + &
        & tauy_tile_ex( AT(i,j,lt) )** 2))
    else
      taux_tile_ex( AT(i,j,lt) ) = 0.0_r_size
      tauy_tile_ex( AT(i,j,lt) ) = 0.0_r_size
      ! ... further tile variables omitted
    end if
    ! ... sea tiles code and variable summing omitted
  end do
end do outerParallelLoop0
!$acc end kernels

```

Device data is interoperable with the CUDA Fortran backend, thus device pointers are used instead of passing the management to OpenACC. OpenACC

directives together with this data type can thus be directly used in the user code as well, i.e. it remains interoperable with device code generated by Hybrid Fortran.

As discussed in Section 2.3, storage ordering macros (here `AT()`) are applied to all array access statements. For the thread block setup, the configurable default sizes `CUDA_BLOCKSIZE_X/Y` are used since no template is specified for the parallel region at hand. Parallel region loops (here for indices `i` and `j` are set up explicitly to parallelize. Other loops, such as the loop over `k`, use a `!$acc loop seq` directive to explicitly avoid parallelization and give the framework user full expressiveness over the desired granularity.

Applying CUDA Fortran backend to the same user code produces the following host code (here shown together with the routine header and footer):

Listing 2.7: Surface flux host code snippet after conversion with CUDA Fortran backend.

```
subroutine hfd_sf_slab_flx_tile_run( &
  ! ... inputs omitted
& )
  use cudafor
  type(dim3) :: cugrid, cublock
  integer(4) :: cugridSizeX, cugridSizeY, cugridSizeZ, &
    & cuerror, cuErrorMemcpy
  ! ... other imports and specifications omitted

  cuerror = cudaFuncSetCacheConfig( &
    & hfd_sf_slab_flx_tile_run, cudaFuncCachePreferL1)
  cuerror = cudaGetLastError()
  if(cuerror .NE. cudaSuccess) then
    ! error logging omitted
    stop 1
  end if
  cugridSizeX = ceiling(real(nx) / real(CUDA_BLOCKSIZE_X))
  cugridSizeY = ceiling(real(ny) / real(CUDA_BLOCKSIZE_Y))
  cugridSizeZ = 1
  cugrid = dim3(cugridSizeX, cugridSizeY, cugridSizeZ)
  cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
  call hfd_sf_slab_flx_tile_run <<< cugrid, cublock >>>( &
    ! ... inputs and further tile variables omitted
    & nx, ny, tile_land, u_f, tlcvr & ! required data objects are
    & tau_x_tile_ex, tau_y_tile_ex & ! automatically passed to kernel
    & )
  cuerror = cudaThreadSynchronize()
  cuerror = cudaGetLastError()
  if(cuerror .NE. cudaSuccess) then
    ! error logging omitted
    stop 1
  end if
end subroutine
```

The prefix `hfd_` is added to host routines that use device data. Hybrid Fortran also duplicates the code for a pure host version of these routines (without a name change in order to remain interoperable with code that is not passed

through Hybrid Fortran). In contexts where the data is not residing on the device, such as the setup part of an application, Hybrid Fortran automatically chooses the host version when generating the call statements at compile-time. Code residing within parallel regions is moved within a separated kernel routine (using prefix `hfki_` with *i* representing the kernel number). In case of the surface flux sample shown here, the kernel routine is generated as follows:

Listing 2.8: Surface flux device code snippet after conversion with CUDA Fortran backend.

```
attributes(global) subroutine hfk0_sf_slab_flx_tile_run(&
  ! ... inputs and further tile variables omitted
  &, nx, ny, tile_land, u_f, tlcvr &
  &, tau_x_tile_ex, tau_y_tile_ex &
  & )
  use cudafor
  use pp_vardef ! defines r_size
  implicit none
  real(r_size), device :: u_f(DOM(nx,ny,ntlm))
  real(r_size), device :: tlcvr(DOM(nx,ny,ntlm))
  real(r_size), device :: tau_x_tile_ex(DOM(nx,ny,ntlm))
  real(r_size), device :: tau_y_tile_ex(DOM(nx,ny,ntlm))
  integer(4), value :: lt
  integer(4), value :: nx
  integer(4), value :: ny
  integer(4), value :: tile_land
  ! ... other imports and specifications omitted

  i = (blockidx%x - 1) * blockDim%x + threadidx%x + 1 - 1
  j = (blockidx%y - 1) * blockDim%y + threadidx%y + 1 - 1
  if (i .GT. nx .OR. j .GT. ny) then
    return
  end if
  ! *** loop body *** :
  lt = tile_land
  if (tlcvr( AT(i,j,lt) ) > 0.0_r_size) then
    call hfd_sf_slab_flx_land_run( &
      ! ... inputs and further tile variables omitted
      tau_x_tile_ex( AT(i,j,lt) ), tau_y_tile_ex( AT(i,j,lt) ) &
      & )
    ! ... rest of loop body already shown in listing 1.6
  end subroutine
```

The specification part of these kernels is automatically generated, applying device state information and converting input scalars to pass-by-value⁵, among other transformations.

It is notable that CUDA Fortran requires a fairly large amount of boiler plate code for grid setup, iterator setup, host- and device code separation as well as memory- and error handling - Hybrid Fortran allows the user to pass on the responsibility for that to the framework. Compared with the code generated by OpenACC however (assembly-like CUDA C or NVVM intermediate repre-

⁵reduction kernels are thus not supported with this backend - we use the OpenACC backend for this purpose.

sensation), the Hybrid Fortran generated CUDA Fortran code remains easily readable to programmers experienced with CUDA. Experience shows that this is a productivity boost, especially in the debugging and manual performance optimization phase of a project.

Regarding the OpenMP backend, since the surface flux example is parallelized at a much more coarse-grained level for CPU, the generated CPU code for the sample at hand is a one-to-one copy of the user code shown earlier in Listing 2.4. The parallelization is generated at a higher level in the call graph (by use of a parallel region construct with `appliesTo(CPU)` clause) as follows:

Listing 2.9: Parallelization of physical processes on CPU.

```
!$OMP PARALLEL DO DEFAULT(firstprivate)
!$OMP$ SHARED( ... inputs and outputs omitted ... )
  outerParallelLoop0: do j=1,ny
    do i=1,nx
      call physics_main(i, j, &
        ! ... inputs and outputs omitted
        & )
    end do
  end do outerParallelLoop0
!$OMP END PARALLEL DO
```

2.5 Code Transformation Method

The following section is a treatise on the method employed to transform the user code (as presented in Sections 2.2 and 2.3) to the code passed to the compiler (see Section 2.4). This process is applied transparently to the user, i.e. it is applied automatically by the means of a provided common Makefile. All components of this transformation are available as Open Source⁶.

2.5.1 Framework Components

In this Section a description of the components, involved in building an application using the Hybrid Fortran framework, is provided. These components are also outlined in Figure 2.3, allowing an overview for the interactions between these components, as well as providing a distinction between user facing, framework facing and machine facing components (from left to right).

Hybrid Sources Hybrid Fortran differentiates between the following user defined source file types and applies a distinct compilation path to them:

⁶Accessible, including sample applications, under <https://github.com/muellermichel/Hybrid-Fortran>.

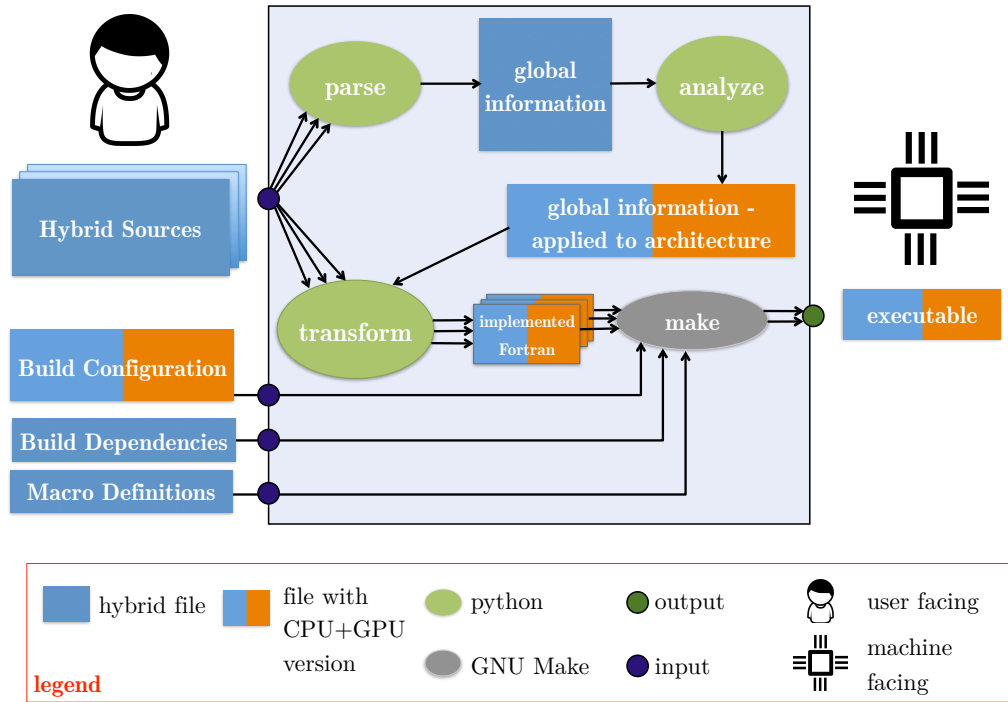


Figure 2.3: Hybrid Fortran software components and build workflow.

- .f90** Standard Fortran source files without any preprocessing step necessary. Such files are directly passed to the specified compiler (see also “Build Configuration”).
- .F90** Standard Fortran source files containing macro definitions. Files of this type are passed to a `gcc -E` based preprocessor, with some additional changes applied in order to allow for multi-line macros within Fortran files.
- .h90** Hybrid Fortran source files - i.e. Fortran source files that also contain the Hybrid Fortran language extensions. These files do not contain any other macros.
- .H90** Fortran source files that contain both Hybrid Fortran language extensions as well as macros. This distinction gives greater flexibility to the user by allowing Hybrid Fortran kernels to be generated first before being passed to the code transformation process.

Build Configuration The build configuration is repeated for each of the target architectures. In its most basic form it consists of a file with GNU Makefile syntax, defining the compiler and linker, as well as their flags, involved in applying the final compilation step in order to build executables.

Build Dependencies The source dependency tree, defined in Makefile syntax. Hybrid Fortran offers an interface to plug in an automated dependency parser, as is often used in larger software projects.

Macro Definitions Macros that are applied across all hybrid files are defined in a special .F90-file. Some of the transformations described in Section 2.5.2 involve a final step that is parametrized through the use of such macros.

Python Framework Scripts A series of python scripts, based on a common Hybrid Fortran python library, is applied to all hybrid sources of a project, in order to extract global information and generate sources that are compatible with a Fortran compiler. For a more detailed description of this process refer to Section 2.5.2.

Implemented Fortran The result of the python based transformation process is one Fortran source file per hybrid source file and architecture. It contains standard Fortran syntax with OpenMP-, OpenACC- or CUDA Fortran directives and/or library calls, as well as macro calls in order to parametrize the storage order and GPU thread blocking setup.

Executable Using the aforementioned build configuration for each implemented Fortran file, all code defined within hybrid sources is added to a static library. In the default case this library is linked directly to an executable. It is also possible to plug in a framework Makefile at this point and hand over executable creation to that Makefile. This facilitates a hybridization of large projects in small steps, i.e. individual smaller application components can be ported to Hybrid Fortran and easily integrated within an existing build infrastructure.

2.5.2 Overview of Transformation Process

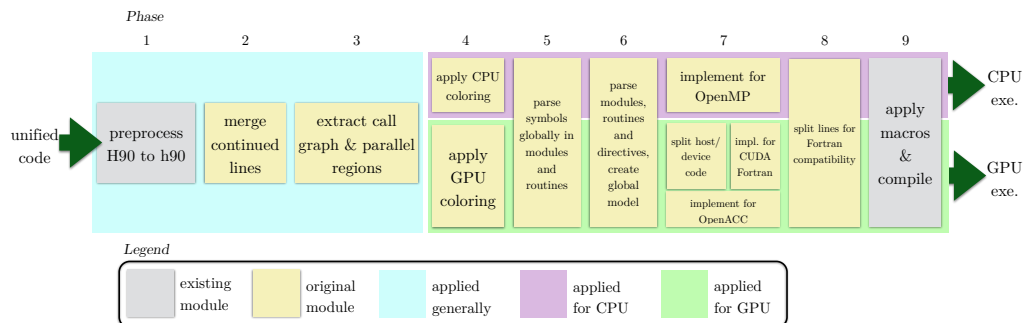


Figure 2.4: Steps involved in the code transformation method.

Figure 2.4 shows the Hybrid Fortran code transformation process in nine distinct phases. Some of the components are shared between all hardware architecture implementations, while some process code is applied only to one of the architectures - as indicated by the coloring in the figure. Each of the numbered paragraphs in this section corresponds to one transformation phase. All of the processes described in this Section are compile-time only, i.e. this method has zero runtime performance overhead.

Phase 1: Macro Preprocessing In case of .H90 hybrid source files, an initial macro preprocessing step is applied in order to facilitate the generation of kernel code through macros. This is especially useful in order to reduce code repetition when a multitude of kernels only has parametric differences that can be handled through a single user code, without the need for runtime branches and without relying on compilers to optimize such code branches. The operating system's GNU compiler tool-chain is used for this purpose (`gcc -E`).

Phase 2: Merging of Continued Lines The Fortran language standard enforces strict line length limits. Line continuations are thus widely used in Fortran code. To facilitate a simplified parsing in later stages, such line continuations are merged in this phase.

Phase 3: Call Graph and Parallel Region Parsing Facilitating phase 4, the application's call graph and parallel region directives are parsed globally.

Phase 4: Call Graph Coloring In order to be able to determine the correct implementation of code with respect to varying parallelization granularity, the transformation process requires positioning information for each routine towards relevant parallel regions. Each routine either directly contains a parallel region, has parallel regions within its call graph, or is itself called within a parallel region. This information can be visualized as a coloring of the call graph, with a separate coloring for each target architecture. Parallel region `appliesTo` clauses are the deciding factor for this coloring.

Revisiting the code example from Section 2.2, Figures 2.5 and 2.6 show the graph colorings created in this phase for CPU- and GPU, respectively. This global positioning information is required in Phase 7 in order to implement the correct data dimensionality, as discussed in Section 2.3, as well as the correct type of subroutine in case of GPUs being the target, i.e. host-, kernel- or device routines.

Phase 5: Global Symbol Parsing As shown earlier in the discussion for Listing 2.3, Hybrid Fortran requires symbol information for symbols imported from other modules, in order to implement the correct data transformation- and movement operations. In this phase, such information is thus parsed and stored in a global symbol model, as well as linked to the various routine scopes.

Phase 6: Global Model Generation The information gathered in phases 3 and 4 is compiled into a global application model, with model classes representing the modules, routines and code regions. Code regions are modelled using subclasses for specification-, sequential-, call- and parallel regions. This model can be regarded as an intermediate representation of the code structured as gathered by the parser. For each code line it contains a list of the data objects used within the line and their known meta information (gathered previously from local module scope, imported scope, routine scope as well as added information from Hybrid Fortran directives). Each routine model is also assigned an implementation class, facilitating phase 7.

Phase 7: Implementation In this phase the global model prepared in previous phases is used to generate implemented Fortran code. Hybrid Fortran can be regarded as a higher level tool-chain that is reusable for various hardware- and software architectures. In order to facilitate the decoupling of framework components from these underlying architectures, only phase 7 is dependent on the backend implementation. For this purpose, a Python implementation class hierarchy is used, enabling maximum code reuse for a set of implementation methods. For more detail on this transformation phase, refer to Section 2.5.6.

Phase 8: Line Splitting In order to generate code that is compatible with standard Fortran compilers, the generated lines are split at appropriate positions containing white space until the standard maximum line length is satisfied. In order to facilitate this phase, the code generation in phase 7 uses space characters where Fortran allows.

Phase 9: Final Macro Processing and Compilation Preprocessor (`gcc -E`) is applied one more time in order to process generated macros for the parametrized storage order and GPU thread blocking setup. Subsequently, a user specified compiler and linker is used in order to create the CPU and GPU executables.

2.5.3 Procedural Shell

Phases 3 through 7 are implemented as a Python script library. There are three main scripts involved in this process: Two programs to implement phases 3 and

4, respectively, as well as a further program to implement phases 5, 6 and 7.

Figure 2.7 provides an overview of the procedural class structure underlying these Python programs.

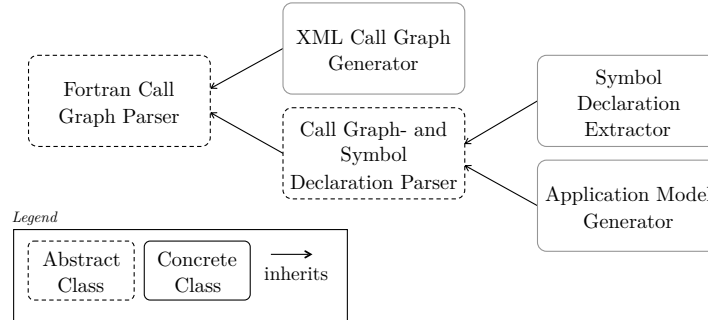


Figure 2.7: Inheritance graph of procedural classes involved in the Hybrid Fortran code transformation.

Fortran Call Graph Parser Abstract class⁷ that provides base functionality to parse control structures of Fortran code that are relevant to the transformation, as well as language extensions used in Hybrid Fortran. Facilitating Fortran’s comparably simple structure per line of code it is based on a finite state machine with a state assigned to each code line (described in more detail in Section 2.5.3). Per state- and state transition a procedural python function is called that is implemented by subclasses where behavior is required for the represented code feature.

XML Call Graph Generator Based on the call graph parser this class generates an XML file⁸ containing the call graph as well as meta-information given by the user by means of the Hybrid Fortran language extension (e.g. @domainDependant and @parallelRegion statements). This facilitates phase 3 of the transformation process.

Call Graph- and Symbol Declaration Parser Abstract class that extends the call graph parser to also parse symbol declarations in the specification part of modules and routines.

Symbol Declaration Extractor Based on the symbol declaration parser, this class generates a python in-memory data structure that represents symbolic

⁷“Abstract” classes are a concept in Object Oriented Programming. Such classes are implemented not to be used directly, but as a means to share code within their inheritance graph. In contrast, “concrete” classes provide full functionality, i.e. functionality that can be used directly by the consumers of such classes in other parts of a program.

⁸XML, short for Extensible Markup Language, is a format to store information for computer programs.

Application Model Generator In phase 6, based on the call graph- and symbol declaration parser, the application model generator is used to generate a global application model. An implementation is assigned to each routine, based on user specification (implementation classes can be changed from the default, either globally through options given to the `./configure` script or by wrapping routines in an `@scheme / @end scheme` structure that allows specifying a scheme name, which allows the assignment of scheme names to implementation classes in the build configuration). Subsequently, in phase 7, the generated model used to convert each line of code to standard Fortran (as described in Section 2.5.6).

The diagram is a state transition graph for an LR(0) parser. It consists of the following states and transitions:

- start** (red oval) transitions to **none** on the empty string.
- none** (grey oval) has a self-loop on `else` and transitions to **End of File** (red oval) on `EOF`.
- none** transitions to **inside declarations** on `(begin subroutine && brackets closed)`.
- inside declarations** (grey oval) has a self-loop on `else` and transitions to **inside domain dependant region** on `domain dependant directive match`.
- inside declarations** transitions to **inside subroutine body** on `(begin subroutine && brackets closed)`.
- inside declarations** transitions to **inside parallel region** on `parallel region end match`.
- inside domain dependant region** (grey oval) has a self-loop on `else` and transitions to **inside declarations** on `d. dependant end match`.
- inside domain dependant region** transitions to **inside subroutine call** on `call match && bracket not closed`.
- inside domain dependant region** transitions to a **transition multiplexer** on `call bracket closed`.
- inside subroutine body** (grey oval) has a self-loop on `else` and transitions to **inside declarations** on `end subroutine match`.
- inside subroutine body** transitions to **inside parallel region** on `parallel region directive match`.
- inside parallel region** (grey oval) has a self-loop on `else` and transitions to **inside declarations** on `parallel region end match`.
- inside parallel region** transitions to **inside subroutine call** on `call match && bracket not closed`.
- inside subroutine call** (grey oval) has a self-loop on `else` and transitions to **inside domain dependant region** on `d. dependant end match`.
- inside subroutine call** transitions to **transition multiplexer** on `call bracket closed`.
- The **transition multiplexer** (grey trapezoid) outputs to **last state before call** (text label).

Legend:

- state**: represented by an oval.
- transition multiplexer**: represented by a trapezoid.
- conditions on next line**: represented by an arrow.
- next state**: indicated by a line from the transition multiplexer.

Figure 2.8 documents the state machine underlying the procedural classes described above. The applied state machine design 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. The parser class, being abstract as described above, does not produce any output itself - instead the output definition is delegated to its subclasses.
2. A multiplexer is introduced as an additional element in order to reduce the number of states necessary.

State transitions are triggered based upon regular expression pattern matching within the code line. As an example, lines within the “subroutine body state” are matched against the regular expression `^\s*do\W.*` in order to determine whether they represent a loop header (beginning of a line `^`, followed by an arbitrary number of white space characters `\s*`, followed by the keyword “do”, followed by a non-word-character `\W`, followed by any characters on the same line `.*`).

2.5.5 Application Model

In phase 6 a global application model is constructed by the application model generator class. In this Section a description of this model is given. Consider the following sample code for a module containing two routines:

Listing 2.10: User code example illustrating the application model generated in Hybrid Fortran.

```

module module_1
contains
  subroutine routine_1(a, b, my_flag, c)
    implicit none
    real, dimension(NX, NY), intent(in) :: a, b
    logical, intent(in) :: my_flag
    real, dimension(NX, NY), intent(out) :: c
    real, dimension(NX, NY) :: a_2, b_2

    if (my_flag = .false.) then
      return
    end if

    @parallelRegion{domName(x,y), domSize(NX, NY)}
    call called_routine1(a_2, a)
    call called_routine2(b_2, b)
    @end parallelRegion

    @parallelRegion{domName(x,y), domSize(NX, NY)}
    c(x,y) = a_2(x,y) + b_2(x,y)
    @end parallelRegion
  end subroutine

  subroutine routine_2(a, b, c)
    ...

```

!<- specification

!<- generic c. region

!<- early exit

!<- generic c. region

!<- parallel region 1

!<- call region 1

!<- call region 2

!<- parallel region 2

!<- generic c. region

```

end subroutine
end module

```

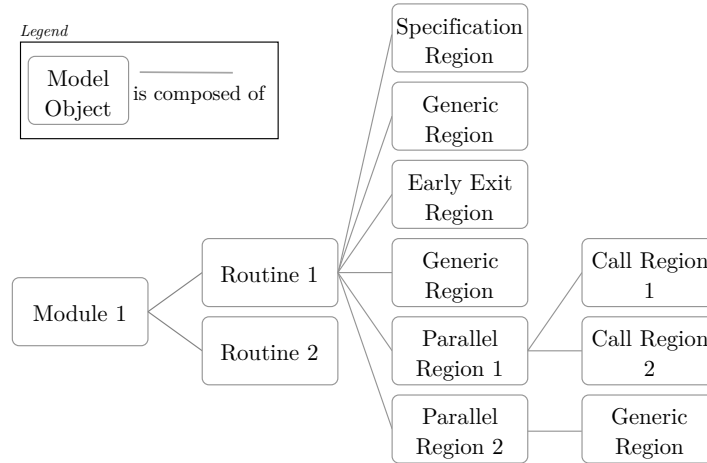


Figure 2.9: Example of an intermediate global application model.

Figure 2.9 illustrates the object structure for the in-memory model that is generated by the “application model generator” class (as described in Section 2.5.3) for this specific sample code. Code lines inside routines are separated into regions of specific types, according to the underlying code structure - in this example, “routine 1” consists of a specification part (as nearly all Fortran routines), an “early exit region” surrounded by two “generic code regions” (the branch structure surrounding the `return` statement), as well as two parallel regions. Parallel regions themselves are composed of a list of regions - in the given example the first parallel region consists of two call regions, while the second contains a generic code region. In general, parallel regions may be composed of generic regions, early exit regions and call regions - recursion with further parallel regions are not allowed, i.e. Hybrid Fortran at this point does not support dynamic parallelism.

The class inheritance graph to model any such code structure is thus given by Figure 2.10. Modules are composed of routines, which are composed of regions, some of which (of type parallel region) consist of further regions. In the example given by Listing 2.10, “generic code regions” are modeled by the “Region” base class, while other regions are given some more specific behaviour through the means of subclassing as illustrated. The generic region base class, finally, is composed of the list of code lines associated with it (modelled as standard Python strings, thus not shown in the class hierarchy), as well as, for each code line, a list of symbol objects.

These model objects (i.e. instances of the “Symbol” class) contain all infor-

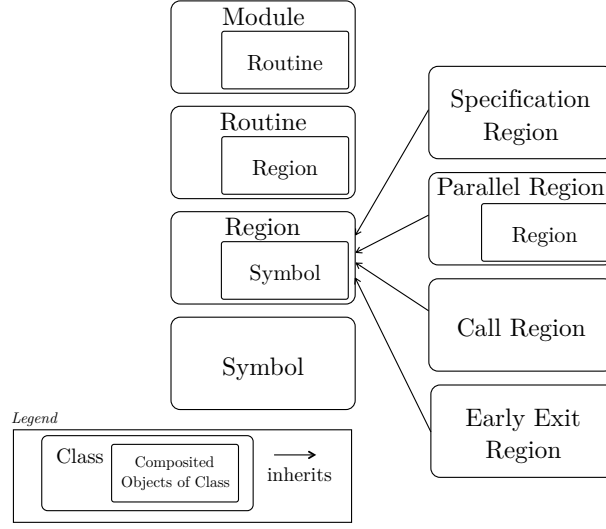


Figure 2.10: Application model class hierarchy and composition.

mation about the symbols' specifications in terms of their data domains, intent (`in`, `out` or `inout`), type, as well as the storage ordering macros to be applied (given through the `@domainDependant` specification as described in Section 2.3)

2.5.6 Transformation of Application Model to Fortran Source

In this section, transformation phase 7 is described, i.e. the final phase concerning the python based script library.

Each of the model classes (as described in Section 2.5.5) have an implementation function at their disposal, i.e. a function that converts the class instance's respective partial model into a Fortran source string. Region class objects assemble their code lines after applying further transformations (described below), routine objects call the implementation function of each region and assemble their output, and, finally, module objects call the implementation- and assemble the output of each of their routines, producing a string that is a valid Fortran source.

The following transformations are applied during phase 7, during the assembly step as described above:

1. In the setup part of an application there are often singular non-performance-critical calls to code that is also shared with the inner simulation (where it is performance critical, since it is called at least once per timestep). In order to avoid data type incompatibilities between device data routines

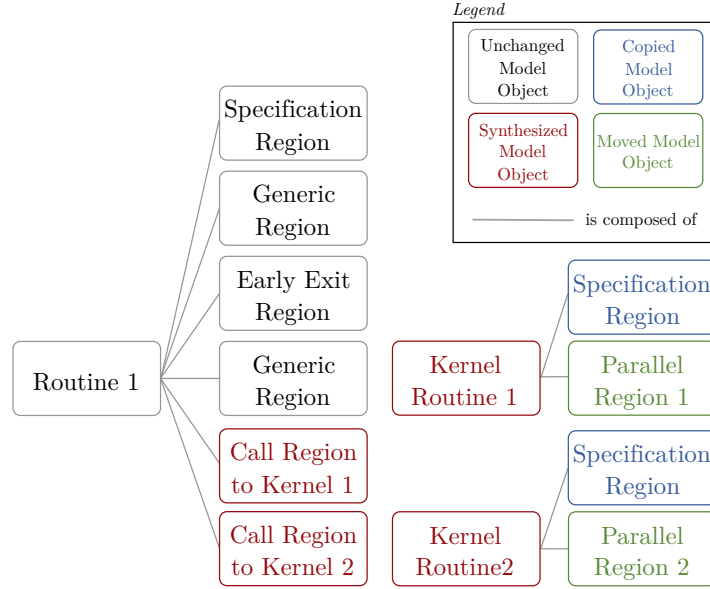


Figure 2.11: Example of an intermediate global application model after routine splitting has been applied.

and host routines outside of the device data region, each routine model is copied, including all its member regions, and assigned a host-only Fortran implementation class (see also the implementation class hierarchy described below). All routines defined in hybrid sources thus remain compatible with host-only callers. In order to distinguish between the parallelized versions and the host-only version of a routine, the former are renamed with a prefix in the routine name, including all their respective calls in the application model.

2. Other than Hybrid Fortran and OpenACC, CUDA Fortran distinguishes between different types of routines. GPU kernels are implemented as **global** subroutines, routines called within kernels as **device** subroutines (see the earlier Listing 2.8 as an example). In order to map from user code routines to CUDA Fortran **global** and **device** routines (as well as the host code setting up and calling each kernel), and to support multiple kernels per routine, Hybrid Fortran employs a routine splitting method. Reconsidering the sample application model given in Section 2.5.5, Figure 2.11 illustrates the model composition graph for “Routine 1” after it has been split for a CUDA Fortran based implementation. Kernel routines are synthesized and given a copy of the original specification region. Parallel regions are moved to the kernel regions and replaced with synthesized call regions for the respective kernels in the original host routine. The newly generated routine’s positioning towards kernels (as determined in phase 4)

is updated to reflect the new call tree after the split.

3. Given the parent routine’s positioning towards kernels as well as the information available to the symbol model objects, each symbol’s data domains are recalculated. As an example, for routines called inside parallel regions where the parallel domain is sliced for each call, symbols depending on the parallel domains have those data domains removed. After matching the symbol object to its reference in the code line, its reference in the code is transformed to use the configured storage order macros as well as the updated domains.
4. Any syntax that is specific to OpenMP, OpenACC or CUDA Fortran is inserted using their respective implementation classes assigned to each routine model object, as described below.

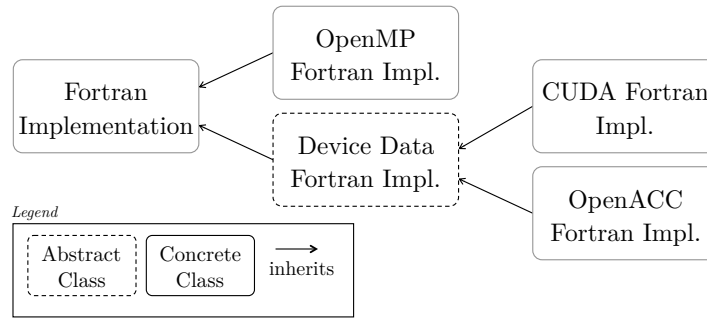


Figure 2.12: Implementation class hierarchy.

In order to separate the concerns of general application modelling and analysis from back-end implementation specific code generation, Hybrid Fortran employs a generic interface for code features triggered during the assembly process described in this Section. As an example, a routine object’s implementation function calls a routine header prefix method - in case of the “Fortran Implementation” class, this method returns an empty string, in case of the “CUDA Fortran Implementation” equivalent it returns `attributes(global)` (compare with Listing 2.8). Since the routine splitting method described above is implementation specific as well, it is also implemented within the “CUDA Fortran Implementation” class - for all other implementation types the respective interface method performs a no-op instead.

Figure 2.12 illustrates the implementation classes supported by Hybrid Fortran at this time. Further implementations, such as OpenCL or OpenMP for GPU, can be implemented with low effort by extending from this work. The following describes the current implementations:

Fortran Base class, implements a sequential CPU-only Fortran version of the code.

OpenMP Fortran Extends the base class by adding parallelization through the means of OpenMP directives.

Device Data Fortran Abstract implementation class that allocates temporary data on the device and transfers inputs to- and outputs from device memory, depending on the instructions given by the program. CUDA device pointers are used as the underlying data structure, allowing interoperability between the OpenACC- and CUDA Fortran backends.

OpenACC Fortran Based on the “device data Fortran” implementation class, employs OpenACC directives to implement kernels and data regions.

CUDA Fortran Based on the “device data Fortran” implementation class, employs CUDA Fortran library calls and intrinsics to implement kernels and data regions.

2.6 Concluding Remarks

In this chapter the design and implementation of a new code transformation method and Fortran language extension, called Hybrid Fortran, has been described. This method, geared at highly data parallel HPC applications written in Fortran, offers an abstraction to solve code granularity issues common for physical processes implemented on GPU. It does so by transforming call tree and memory layout of the application. Furthermore it implements parallelization for both CPU and GPU by generating the required boilerplate code for OpenMP, OpenACC or CUDA Fortran. Granularity- and layout transformations are performed at compile-time, thus having zero runtime overhead, an important characteristic in tooling geared for HPC purposes.

Through the means of routine splitting as well as facilities that allow mixing different back-end implementations in the same executable, Hybrid Fortran is a flexible solution that can be applied with low effort to both the dynamical solver (usually consisting of fine-grained kernels) as well as physical processes (often containing coarse-grained tasks). Several applications have been successfully ported with this method, including an operational mesoscale weather prediction model of national importance to Japan.

Application to Numerical Weather Prediction

In Chapter 1, GPU computing has been introduced, its application to Numerical Weather Prediction (NWP) has been motivated and key challenges regarding said application have been outlined. Large parts of current day NWP models demand frequent adjustments by domain scientists - maintainability and code ownership are therefore important aspects of any solution that is geared towards operational NWP. GPU computing has certain requirements that work against this goal: Many typical processes require a splitting of code into smaller tasks in order to allow efficient execution within the GPU's massively parallel processing paradigm. Due to differences in the hardware architecture compared to CPUs this has negative implications on code divergence for hybrid CPU/GPU codebases if solved using conventional methods.

In order to meet these challenges, a new method called “Hybrid Fortran” has been introduced in Chapter 2. It allows a more streamlined approach to porting massively data parallel Fortran application to GPU, solving granularity-, memory layout- and parallelization boilerplate divergence issues by means of an automated source-to-source transformation. Its user code is based in Fortran language (a widely used standard in NWP), extended with some additional instructions in order to facilitate the transformation process.

By applying Hybrid Fortran to ASUCA, Japan's operational national mesoscale model, a major NWP model has been ported to GPU while keeping CPU compatibility, using a single coherent framework. This chapter provides a discussion on “Hybrid ASUCA”, the name given to this new implementation.

The ASUCA model has been described earlier in Section 1.3.5. Its new GPU-capable implementation is discussed in Section 3.1, with the verification process outlined in Section 3.2. Productivity results for ASUCA are provided in Section 3.3. A comparison with a previous GPU implementation of ASUCA and a discussion of the applicability of our method to further models is provided

in sections 3.4 and 3.5, respectively. Performance results follow in Chapter 4, including a performance analysis concerning the applied methodology.

3.1 Hybrid ASUCA Implementation

In this section, the hybrid implementation of ASUCA is discussed from an engineering perspective. Hybrid ASUCA has been developed based on branch of ASUCA from late 2013 (henceforward called the “reference version”) that represents its production state at the start of this GPU porting project. It is aiming at a highly (but not bitwise-) accurate port of the existing model, i.e. the model itself is to remain identical to the reference version given by JMA.

3.1.1 Design Goals

Consider the simulation process overview, provided in Figure 3.1. The green shading in the figure indicates how often a simulation component is iterated per time step - i.e. a darker shade of green generally has a higher importance in terms of performance. The dotted line represents the boundaries between host and device, i.e. a data bus such as PCI Express on Intel based host systems. This bus is slow compared to the memory bandwidth available on the device, a major aspect to take into account when implementing any application on GPU. The first design goal of any GPGPU application is thus typically to minimize the amount of data that needs to be transferred on said host bus. For Hybrid ASUCA specifically this requires that all processes accessing and/or updating grid values during the main simulation are to be ported to GPU such that all simulation data can reside on the fast device memory. There are only two use cases where the PCI Express bus is required during the main simulation time advancement: In order to output data to files as well as to exchange halo data with neighboring nodes.

When porting large applications to new architectures - e.g. ASUCA’s reference version consists of 155 thousand lines of code - code divergence between the implementations targeted at different hardware architectures is a common issue. This issue is amplified in NWP models as such software is often adapted by domain scientists, especially concerning the physical parametrization as already discussed in Section 1.3.6. ASUCA’s operations will remain on CPU systems for the immediate future, even if GPU adoption is being researched and prepared. In Hybrid ASUCA’s design this is anticipated by allowing support for both CPU- and GPU based architectures.

Acceptance and migration costs are further factors to consider in the design. A switch of operations to the hybrid model entails either porting the latest production version or porting the changes made to the production version since

the reference was branched off. A solution that is as similar as possible to the reference is thus beneficial in terms of acceptance (minimization of the learning required to become familiar with the adapted model) as well as to minimize friction for a migration of operations.

To summarize, Hybrid ASUCA requires a GPU port of all functions called within the simulation, its user code is to be applicable both on CPU and GPU architectures and it is to remain as similar as possible to the original code. The only method known to the author that matches these requirements is “Hybrid Fortran”, introduced in Chapter 2, as it has been developed specifically to meet these goals. In this Section, its application to ASUCA is discussed. For a performance analysis of the method and its application to ASUCA we refer to Chapter 4. For a treatise on other available methods, and a comparison of their characteristics, see also Chapter 5.

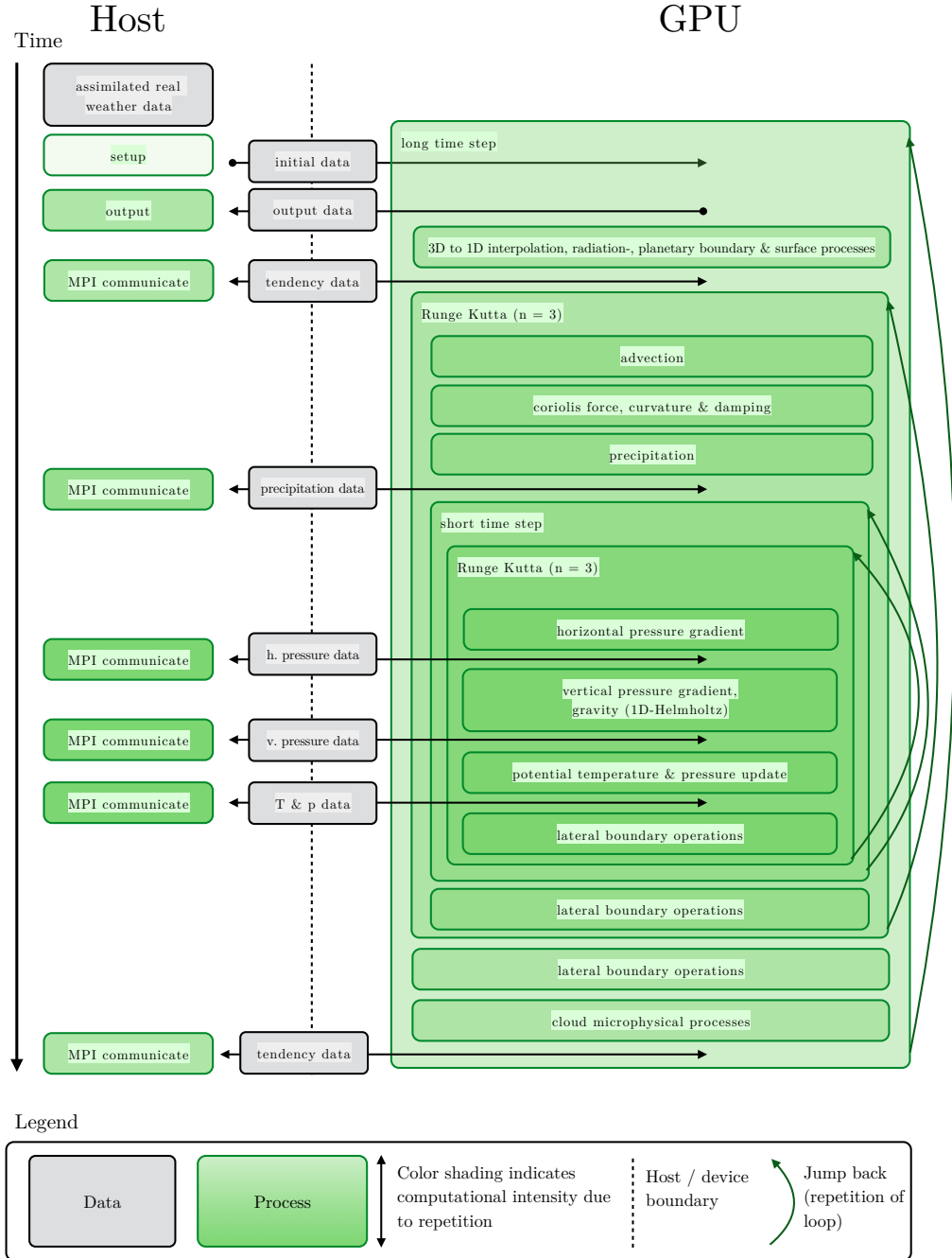


Figure 3.1: Overview of simulated processes in Hybrid ASUCA when run on GPUs.

3.1.2 Data Structures- and Layout

This section gives an outline on how ASUCA's grid is stored in memory. The methods used to achieve the characteristics described here have been introduced in detail in Section 2.3.

ASUCA uses Fortran's flat multidimensional arrays as its basic data structure for the values of its rectangular grid. The simplicity of this approach is its advantage - a flat array can be implemented in an efficient way on any computing architecture with sufficient working memory. The trade-off is that it lacks a parametrization for the underlying memory layout, i.e. the layout is hard coded in the application. The storage order chosen in the reference implementation is KIJ, i.e. the vertical k -direction is accessed with stride-1. To match this order in the code, the vertical domain is the fastest varying, i.e. loops over the vertical domain are the innermost in each process. Since the vertical dimension has a limited number of values (58 in the reference configuration used for validation), this allows ASUCA to retain the currently iterated k -column to remain in cache in many of its processes.

For a GPU implementation of ASUCA the KIJ ordering is problematic. The vertical domain, being implicitly solved, requires sequential iteration. In order to coalesce memory accesses (and thus benefit from the high memory throughput available), GPU code requires the stride 1 dimension to be parallelizeable, thus demanding I-first or J-first ordering. For a more detailed discussion of memory coalescence we refer to Section 1.2.2. The performance impact of memory layout is also shown in Section 4.2.3.

In Hybrid ASUCA this issue is solved by making use of Hybrid Fortran's memory layout transformation. While the user code remains in KIJ order, it is reordered through code-to-code transformation to IJK for the GPU case. The order is parametrized by defining ordering macros based on the hardware architecture the application is being compiled for.

3.1.3 Parallelization

As discussed earlier in Section 1.1, finding an efficient parallelization can be regarded as the most fundamental task when applying a model to HPC systems. This section describes how this is achieved for the ASUCA NWP model on heterogenous systems involving multi-node, multi-core CPU and many-core GPU hardware architectures. Hybrid ASUCA supports either multi-core- or GPU parallelization, coupled with concurrent execution on multi-node. GPU and multi-core parallelization is combined for different parts of the application, however work sharing between the two types of shared memory parallelizations is not supported.

We distinguish between the different parallelizations by the method in which working memory is accessed. If each task only has its own memory available it follows that work, intermediate value updates as well as results need to be communicated by ways of a message passing scheme. Such a parallelization method is called “distributed memory parallelization”. In contrast, if tasks have access to a shared memory to exchange their data, we call it a “shared memory parallelization”.

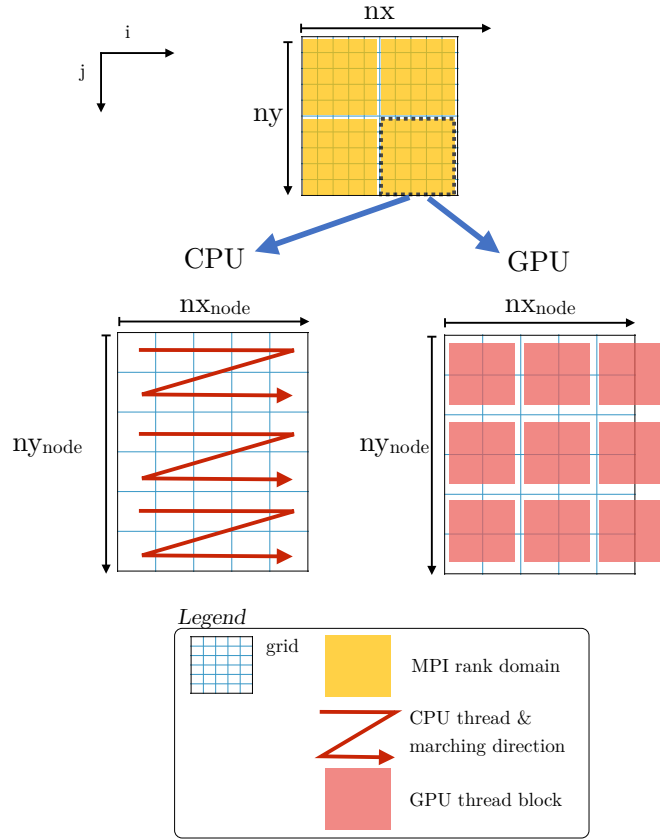


Figure 3.2: Horizontal parallelization, CPU vs. GPU.

Since the vertical domain is treated sequentially in the HEVI method, parallelization is applied in the horizontal IJ domain.

Distributed Memory Parallelization

In order to scale to multiple nodes (and thus a distributed-memory cluster), ASUCA’s data grid is decomposed into blocks in the horizontal domain that are scattered to individual ranks, using the MPI standard for communication, as outlined in Figure 3.2. Hybrid ASUCA is compatible and has been tested with

both the OpenMPI- and the MVAPICH2 implementation of the MPI standard.

Figure 3.1 shows the types of data that require communication during time integration due to horizontal dependencies of neighboring grid cells - this includes tendency-, precipitation-, pressure- and temperature data. Since the entire simulation is ported to GPU, its internal data resides on GPU. Full updates of the data on host is only required for file output, which is executed for a subset of time steps in operation - we refer also to Section 3.1.4 regarding the optimization of data communication. Kernels with horizontal data dependencies require communication of the outermost layers to neighboring nodes- or GPUs. Such data regions are called “halos” due to their ring shape. Since halos grow with the surface area of a block while the computations per node grow with the volume, the most basic optimization of communication is to minimize the block surface area by reducing the number of nodes involved in the computation while satisfying the overall performance required given time budgets in an operational weather prediction setting. GPUs allow a higher density in computational performance, i.e. a lower number of GPU rich nodes are able to replace existing CPU nodes, thus relaxing potential throughput- or latency bottlenecks in the inter-node network.

Shared Memory Parallelization

Given the trend towards increasing parallel throughput on each compute node, shared-memory parallelization has continuously gained in importance over the last two decades. Parallel tasks executed on the same node tend to be implemented through the use of shared memory in bandwidth-constrained applications as this avoids unnecessary memory copying operations. Hybrid ASUCA therefore applies the paradigm of parallel threads with access to a shared memory to both multi-core and GPU parallelization.

On CPU, the slowest varying dimension J is split into chunks, equal to the number of cores available on the system. Affinity of threads is set such that neighboring threads are assigned to cores on the same socket in order to maximize L2 cache usage on multi-socket systems. Each thread is then assigned $n_z \cdot n_x \cdot chunksize_y$ grid values to compute per step for three-dimensional processes (all processes shown in Figure 3.1 with the exception of lateral-, and surface boundary operations, which are two-dimensional processes). On CPU the parallelization is thus coarse-grained in order to reduce thread switching overhead. Figure 3.2 shows this marching order with three CPU threads on a 6x6 grid - all dimensions have been scaled down for illustrative purposes.

In contrast, thread context switching on GPU is computationally inexpensive, with caching- and register usage optimized for tasks with comparatively low complexity, thus encouraging a very fine-grained parallelization - as has been

discussed in more detail in Section 1.3.6. Each column in the IJ domain is thus assigned a separate thread, forming a CUDA grid. The CUDA grid is therefore a two-dimensional projection of the three-dimensional ASUCA grid in the IJ plane. Rectangular collections of threads are grouped into thread blocks - the smallest grouping of threads that is loaded as an ensemble onto a GPU streaming multiprocessor, as discussed in Section 1.2.2.

Figure 3.2 shows the 6x6 grid example subdivided into nine thread blocks, each with 2x2 threads - again down-scaled for illustrative purposes. Each thread in a two-dimensional CUDA grid gets assigned a tuple of four indices: two for the x-y-coordinates of the thread within a block, and two for the coordinates of the block within the CUDA grid. In CUDA programming it is typically the first task of the kernel program to map from the thread index tuple to the data indices in a structured grid application. Since thread blocks are indivisible during initialization, if the data grid is not evenly divisible by the block size in each direction there are threads with no data to process, as indicated for the rightmost column of threads in Figure 3.2. This is solved by letting threads outside the data domain exit early, in case of CUDA through explicit coding.

Parallelization schemes, even a comparatively simple one as presented here, involves programming “boilerplate” (an expression used in programming to describe code that is purely infrastructure-related). For data-parallel applications using multi-threading, the OpenMP standard is designed to minimize such boilerplate and thus simplify parallelization for the application programmer. Such directives are already employed in ASUCA’s reference implementation in order to use CPU-based multi-threading. On GPU however, OpenMP has only slowly been gaining maturity and was unavailable at the outset of the project underlying this thesis [66]. The OpenACC standard has been developed to close this gap, abstracting the scattering of threads as well as the mapping of their own indices to loop indices for data access, thus offering a similar interface as OpenMP [65] [67]. Combining OpenMP with OpenACC directives is therefore a potentially viable approach to hybridize ASUCA for the discussed hardware architecture, even though it involves combining two separate sets of directives for every kernel.

The viability of such an approach is however limited to kernels with code granularity that is fitting both GPU and CPU. Neither OpenMP nor OpenACC have built-in methods to split or combine kernels that are mismatched in granularity for a particular hardware architecture, thus resulting in code divergence [4]. For most of ASUCA’s physical processes, and thus significant parts of its codebase, granularity of the kernels is too coarse for GPU, as will be discussed in Section 3.1.5. Hybrid Fortran has been specifically developed to provide facilities that solve this granularity problem. Since it also allows a unification of the boilerplate required for both OpenACC and OpenMP, Hybrid Fortran’s parallelization method has been applied to all of ASUCA’s kernels called within the

main simulation, including the dynamical core. This approach is described in detail in the earlier Section 2.2.

3.1.4 Communication between Host and Device

In this section a reduced performance model is created for ASUCA in order to estimate the impact of communication between host and device on the overall performance and to outline the solution design for an efficient GPU implementation.

According to Amdahl, the speedup of computations for a fixed problem size is limited by the serial parts of the problem as follows:

$$S_{Amdahl}(f, S) = \frac{1}{(1 - f) + \frac{f}{S}}, \quad (3.1)$$

with S_{Amdahl} being the theoretical speedup in latency for finishing the whole task, f being the fraction of the runtime being parallel and S being the parallel speedup of that fraction [68] [69].

Assuming a data-parallel memory bandwidth bounded application running on a single GPU with zero kernel launch overhead (by having a large enough problem size), we can substitute

$$f = \frac{t_{comp}}{t_{comp} + t_{comm}}, \quad (3.2)$$

with t_{comm} and t_{comp} being the communication- and computation time, respectively. Subsequently, for a bandwidth bounded application these terms can be defined as follows:

$$\begin{aligned} t_{comp} &= n_i \frac{m \cdot b}{BW_D}, \\ t_{comm} &= \frac{m_{comm} \cdot b}{BW_{HtoD}}, \end{aligned} \quad (3.3)$$

where n_i is the number of iterations over the data, m and m_{comm} the number memory transfers between device processor and device memory and GPU and host memory, respectively, b the byte length of each value, and finally, BW_D and BW_{HtoD} the memory bandwidth available on device and between host and device, respectively.

The parallel runtime fraction f thus becomes

$$f = \frac{BW_{HtoD}}{BW_{HtoD} + \frac{m_{comm}}{n_i m} \cdot BW_D}. \quad (3.4)$$

We note that f is agnostic of byte length b . It can be determined using two ratios: The hardware determined ratio of device memory bandwidth (BW_D) to PCI Express bandwidth (BW_{HtoD}) as well as the algorithmically determined ratio of memory operations in device memory ($n_i m$) to memory operations across the PCI Express bus (m_{comm}). On Pascal generation NVIDIA Tesla GPUs (P100), the device-to-PCI-Express bandwidth ratio is approximately 46 to 1 (refer to Appendix A).

Consider again the ASUCA process overview as shown in Figure 3.1. Most of the code involved with implementing physical processes is executed once per long time step. Given the low ratio of computational time to code size, one could argue for a solution where physical processes are executed on CPU, while processes iterated within the two-step third order Runge-Kutta scheme is implemented on GPU. The performance analysis in this section can be used to estimate performance limitations that would arise from such an implementation strategy. For such an evaluation, one must first determine n_i , which depends on the number of short time steps.

In ASUCA, the number of short timesteps n_{sound} (using the nomenclature of Wicker and Skamarock [53]) is determined separately for each outer Runge-Kutta step:

$$\begin{aligned} n_{sound}(1) &= \left\lceil \frac{\Delta t/3}{\Delta x/c_s} \right\rceil, \\ n_{sound}(2) &= \left\lceil \frac{\Delta t/2}{\Delta x/c_s} \right\rceil, \\ n_{sound}(3) &= \left\lceil \frac{\Delta t}{\Delta x/c_s} \right\rceil. \end{aligned} \quad (3.5)$$

With a horizontal grid resolution (Δt) of two kilometres, a long time step (Δt) of 16.66 seconds and c_s using the value 400 (empirically determined using stability- and efficiency criteria), we arrive at n_{sound} values of 2, 2 and 4, giving n_i as follows:

$$n_i = 3 \cdot 3 \cdot \sum_{i=1}^3 n_{sound}(i) = 72, \quad (3.6)$$

3 · 3 being the Runge-Kutta iterations according to Wicker and Skamarock’s nested RK3 scheme. Plugging this into Equation 3.1 allows us to determine S_{Amdahl} as a function of the ratio of memory operations on device memory m to communicated memory operations m_{comm} for each of the innermost Runge-Kutta iterations (i.e. already taking the factor $n_i = 72$ into account). The theoretical speedup S is assumed to be 6.5x - the memory bandwidth ratio between GPU and CPU on the test cluster “Reedbush-H”, according to its hardware metrics listed in Appendix A.

ASUCA’s dynamical core uses a low number of time-invariant variables that are variant on the grid position (reference points for pressure, density, potential temperature and Exner function) compared to a large number of time-variant grid variables. To keep a simple model for this section’s assessment, given the large number of kernels involved in the dynamical core, we hereby assume that limiting the GPU implementation to the dynamical core would require a full update once per long timestep of each grid variable that is accessed on GPU, that is, $m = m_{comm}$. Amdahl’s law therefore shows that such an implementation would limit the speedup from CPU to GPU on the test cluster to approximately 2x. In contrast, porting all physical processes to GPU allows to keep all simulation data on the fast GPU device memory, disregarding halo updates and file output - in theory with a full port, a speedup of 6.5x is achievable on Reedbush-H. Hybrid ASUCA thus defines device state information for grid variables, using the method outlined in Section 2.3.2 by defining the long timestep integration loop as boundaries for the device data region. This entails the usage of a transfer specification in the specification part of the main simulation routine as follows:

Listing 3.1: Data transfer implementation in Hybrid Fortran.

```
@domainDependant{attribute(autoDom, transferHere)}
!... list of data object names ...
@end domainDependant
```

For a total of 255 two-dimensional and three-dimensional arrays containing grid values, this method is used to transfer data at the simulation boundaries. Scalar data is automatically converted and passed in to kernels as compatible data types by Hybrid Fortran through compile-time transformation. Arrays with grid values require additional specifications as follows, in routines called within the simulation where they are in scope, in order to avoid unnecessary communication:

Listing 3.2: Data device state implementation in Hybrid Fortran.

```
@domainDependant{attribute(autoDom, present)}
!... list of data object names ...
@end domainDependant
```

Implementing the halo exchange as described in Section 3.1.3 is achieved by running a kernel (parallelized for either GPU or CPU) for initializing the MPI buffers and subsequently transferring said buffers to host memory in the GPU case before executing MPI communication with other nodes. File output on the other hand requires a full update of the data on the host system, which is implemented manually through Fortran-style array copy operations from device array to host array, only executed for the GPU code version.

3.1.5 GPU Implementation of the ASUCA Model

For a majority of kernels, as Section 3.3 will show, GPU implementation using the Hybrid Fortran method is a low effort transition. In this section we focus on parts of the model that employ non-trivial changes for GPU compatibility, while providing an overview of all the processes involved as already outlined in Figure 3.1.

Dynamical Core

In recent years, CPUs have developed towards highly parallel, superscalar architectures, as shown in Section 1.2.1 using Intel’s recent flagship Xeon processor as an example. ASUCA’s dynamical core has been developed as an answer to this shift towards node-level parallelism [70]. Dynamical modelling of physical interactions in the atmosphere generally involves a tight coupling of variables. As ASUCA’s model is fully compressible and non-hydrostatic, coupling is tight in all spatial dimensions. Irrespective of the dimensions chosen for parallelization, numerical solutions involve several value updates across independent memory systems of concurrently operating processors. These characteristics show that fine-grained parallelization is a necessity in dynamical cores, commonly expressed as “tight loops”. Since such fine-grained parallelism is already suitable for GPUs, the control structures used in the code do not need any changes at a fundamental level.

Hybrid Fortran has therefore been developed to accommodate codes with tight loops in order to minimize the changes required for ASUCA’s dynamical core. Routine splitting is employed during source-to-source transformation in order to synthesize CUDA Fortran kernels, as described in Section 2.5.6, allowing for a multitude of tight parallel regions defined per Fortran routine. Porting ASUCA’s dynamical core to Hybrid Fortran subsequently has been a simple replacement of OpenMP parallelized `do`-loops with `@parallelRegion` constructs, using the parallelization scheme for CPU and GPU already described in Section 3.1.3.

Physical Processes Model

In NWP, the term “physical processes” is used to describe parametrizations that require a higher resolution than is possible to compute directly in the dynamical core of the model. Grid variables such as pressure, wind velocity and potential temperature are used as an input to such physical process models, resulting in tendencies employed to correct the dynamical solution for physical effects.

For Hybrid ASUCA the physical processes library from the model “MSM0705” (as referred by JMA) has been ported to GPU while retaining support for multi-threaded scalar hardware architectures [71].

Radiation is modelled using a scheme developed by Kitagawa in 2000 [72] [73], based on the 18-band model by Briegleb, 1992 [72] [42]. For optical absorption by clouds, a statistical model by Goody from 1952 is combined with a correction for optically thin clouds in higher atmospheric levels as proposed by Kiehl and Zehnder in 1995 [72] [74] [44]. Absorption by dust particles is estimated by computing a transmission function look-up table (level k_1 to level k_2 for band n_x) based on an empirical model developed at NASA Goddard in 2001 [43] [75]. Aerosol- and water continuum absorption is treated as proposed by Zhong and Haigh, 1995 [76].

To account for turbulences due to proximity to terrain, a Mellor-Yamada based model improved by Nakanishi and Niino is used [77] [41]. Wind momentum-, sensible heat- and latent heat surface fluxes are simulated (Beljaars and Holtlag, 1991 [78]), coupled with the aforementioned turbulence model.

A Kessler-type warm rain model is implemented as a parametrization for cloud microphysics, simulating water vapor, cloud water and rain [79] [80]. By means of a secondary time splitting method, vertical advection of water substances with high terminal velocity is treated [6].

Adaptive Parallelization Granularity for Physical Processes

Physical processes in ASUCA are horizontally independent models as outlined above. Since no horizontal communication is necessary for such processes, they are executed independently on CPU for each column without any thread synchronization in order to optimize for caching [70] [81] [82]. The parallelization of these processes is thus very coarse-grained and unsuitable for a direct port to GPU, as discussed in Section 1.3.6. In Hybrid ASUCA this is solved by employing the granularity transformation method described in Section 2.2. Figure 3.3 illustrates the extent as to which the call physical process call tree is transformed between CPU and GPU implementation with this method for radiation-, planetary boundary layer- and surface processes, by splitting a single CPU parallelization to many GPU kernels. The tree structure is automatically generated

based on the global information gathered from the application, with the dots representing routines and arrows representing calls. The same scheme is also employed to implement water substance terminal velocity processes (“sedimentation”), which is not included in the figure.

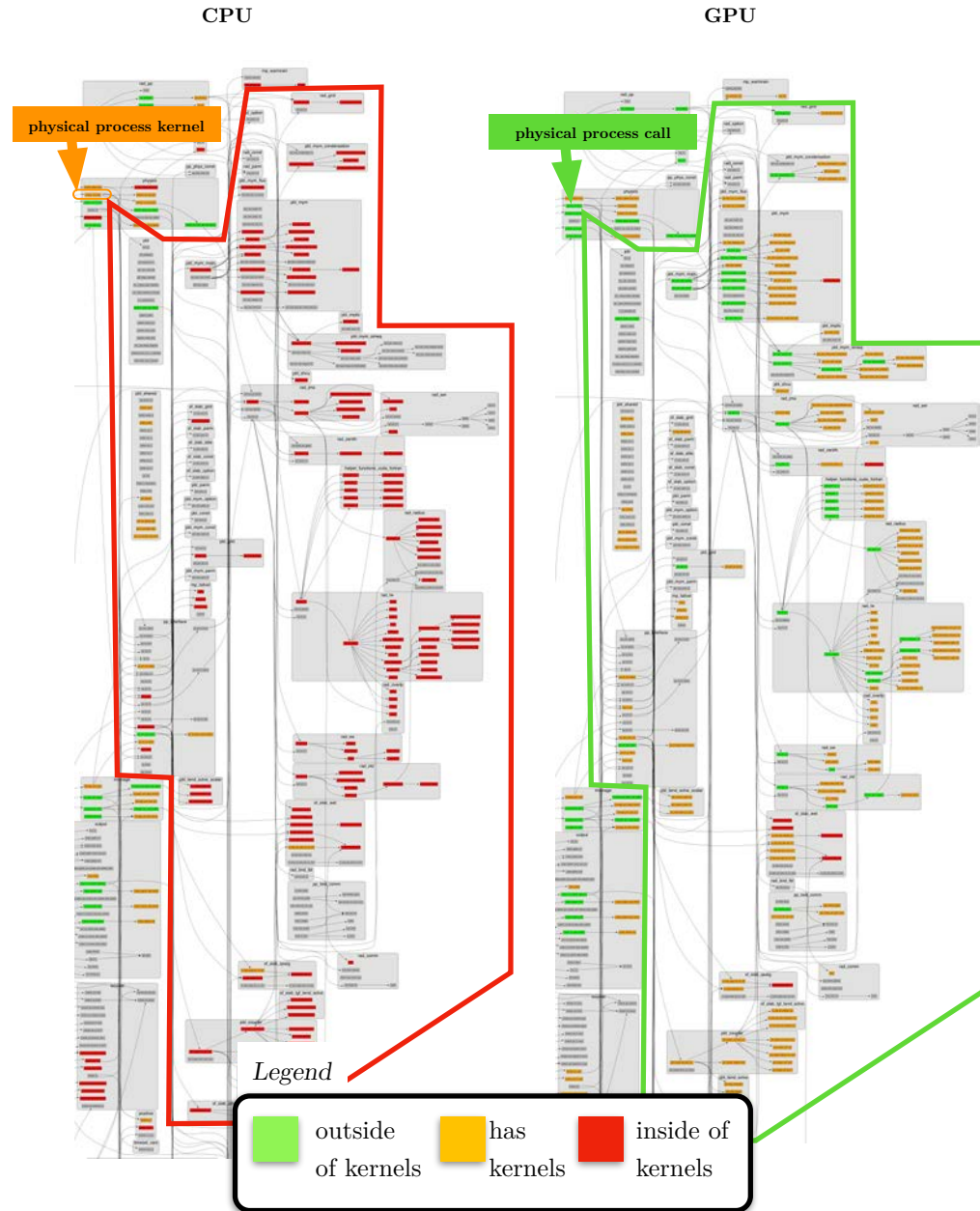


Figure 3.3: Granularity transformation of physical processes call tree for GPU.

Radiation Memory Footprint Optimization

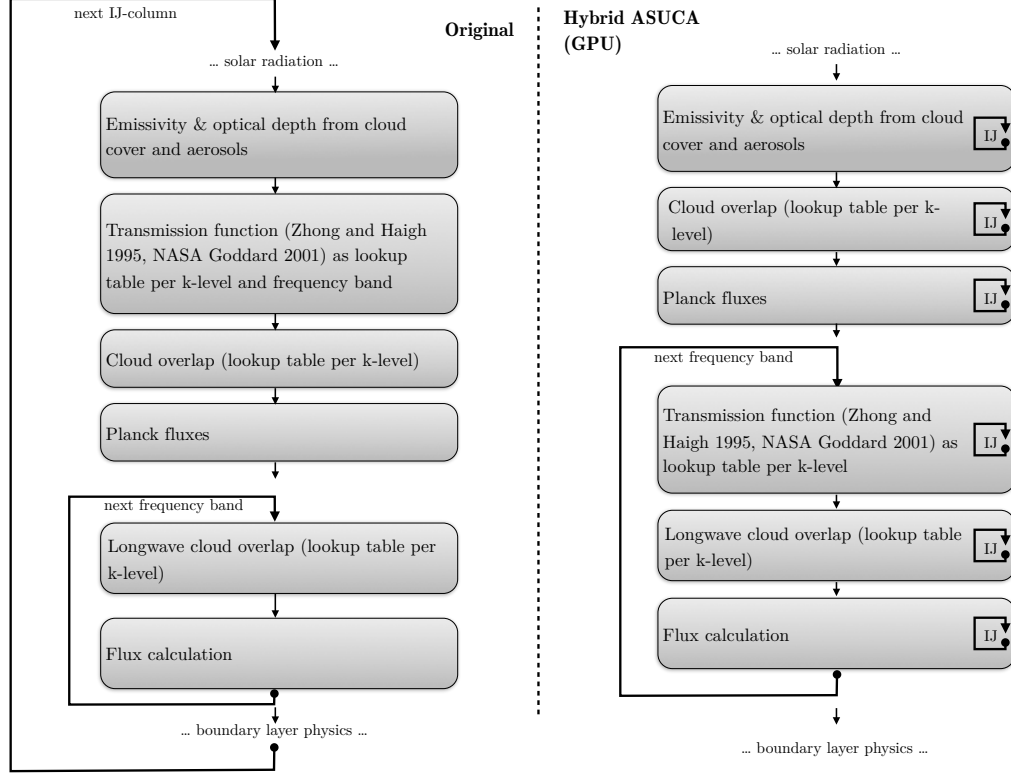


Figure 3.4: Memory footprint optimization for radiation code.

Figure 3.4 outlines the changes made to the original radiation code from the JMA MSM0705 model. In the original version the transmission functions for aerosol-, water continuum absorption and dust particles are computed once per vertical column, creating a lookup table for every k -level and each of the ten frequency bands that are treated separately. Due to the high number of threads required to make efficient use of the GPU cores, the amount of memory used by this original approach is problematic, i.e. it significantly reduces the simultaneous number of threads. For Hybrid ASUCA we have thus optimized this approach by computing the aforementioned transmission functions separately per frequency band, which reduces the required amount of thread-local memory by a factor of approximately 10x.

Column-wise Courant-Friedrichs-Lewy Convergence

The time splitting method employed for the water substance sedimentation process requires some specific treatment in order to achieve a fine-grained parallelization for GPU, due to data-dependent per-column branching based on the

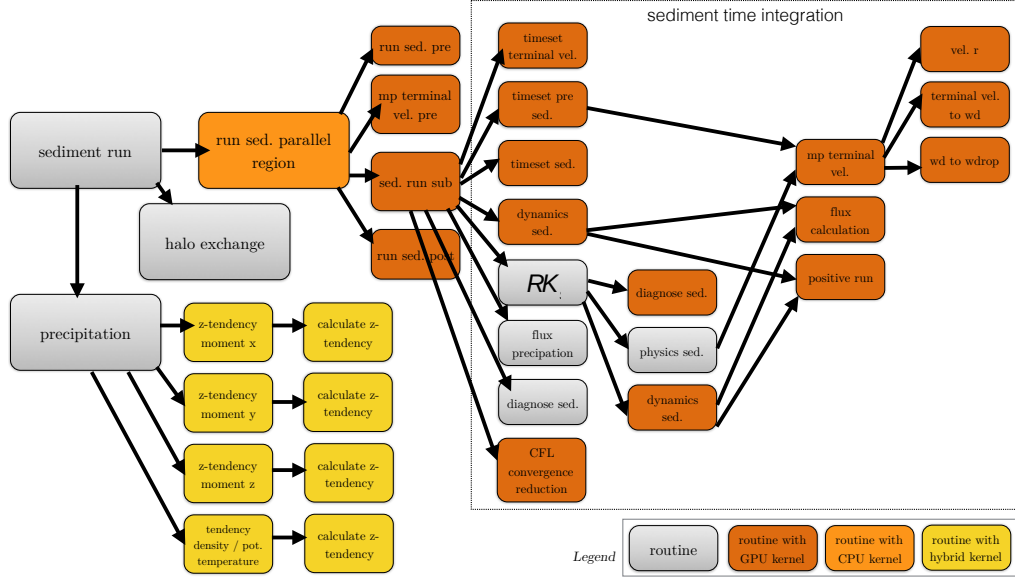


Figure 3.5: Granularity transformation of precipitation module with column-wise CFL convergence.

Courant-Friedrichs-Lewy convergence condition. Figure 3.5 shows the call tree and kernel structure of the sedimentation code in Hybrid ASUCA. As in other physical processes, a single CPU parallelization is split into multiple GPU kernels. The following listing shows the timestep loop used to port per-column time splitting to GPU. A reduction kernel is employed to evaluate the values of the array `cfl_reached`, which is privatized in IJ for GPU using the method described in Section 2.3.3. In case of all columns converging, the time integration is exited in order to avoid unnecessary kernel launches. In addition, `cfl_reached` is passed down to all kernels called within the time integration for a fine-grained breaking of GPU threads in case only a subset of columns has converged.

Listing 3.3: Sedimentation timestep with CFL convergence condition.

```
@domainDependant{
    attribute(autoDom, present), domName(i,j), domSize(nx,ny)
}
cfl_reached, dt_rk_rest, ...
@end domainDependant

! ... initialization of variables

timestep_sed: do
    ! ... Runge-Kutta based iterative solution to sedimentation

    @parallelRegion{appliesTo(GPU), domName(i, j), domSize(nx, ny)}
    if ( dt_rk_rest < dt_rk_rest_epsln ) then
        cfl_reached = .true.
    end if
    @end parallelRegion
```

```

call all_true_for_xy_plane(cfl_reached, all_cfl_reached)

if (all_cfl_reached) then
  exit timestep_sed
end if
end do timestep_sed

```

3.2 Verification

Correctness as well as completeness of Hybrid ASUCA for both hardware architectures has been continuously verified throughout the development process. In this Section we outline the way in which correctness of the model has been ensured.

Hybrid ASUCA uses 64bit precision floating point arithmetic for all computations and on all architectures. A series of test cases with idealized data have been used in order to verify Hybrid ASUCA’s output against the reference implementation, ensuring a root mean square error lower than 10^{-9} for pressure, moment and temperature variables. This includes a radiation module test, a physical process ensemble test for radiation, planetary boundary region and surface (“GABLS3” [70]) as well as a two-dimensional (IK) “warm bubble” test for the dynamical core in isolation. In addition, tests with the same error metric have been conducted for a series of module configurations of ASUCA together with real weather input data on multiple grid sizes - 51x61x58, 301x301x58 as well as 1581x1301x58, the latter being a full sized operational mesoscale grid with two-kilometer resolution for Japan and its surrounding region. A subset of these tests (available for single node execution with time to solution on the order of minutes) has been used as an automatic verification system after each new build of Hybrid ASUCA, such that we were able to test each newly ported module without significant development overhead.

Figure 3.6 shows the resulting cloud cover for a test with the full sized operational grid, with all physical process simulations enabled as discussed in Section 3.1.5. The test has been conducted on the Tokyo University cluster “Reedbush H” [83]. This cluster offers two 18-core Xeon E5-2695 v4 CPUs per node as well as two NVIDIA Tesla P100 GPUs per node. At least seven nodes (14 CPU sockets) or 24 GPUs were required due to the problem’s memory requirements. The visualization has been created by use of the “Grid Analysis and Display System” (GrADS), a two-dimensional visualization tool commonly used for weather analysis and prediction [84].

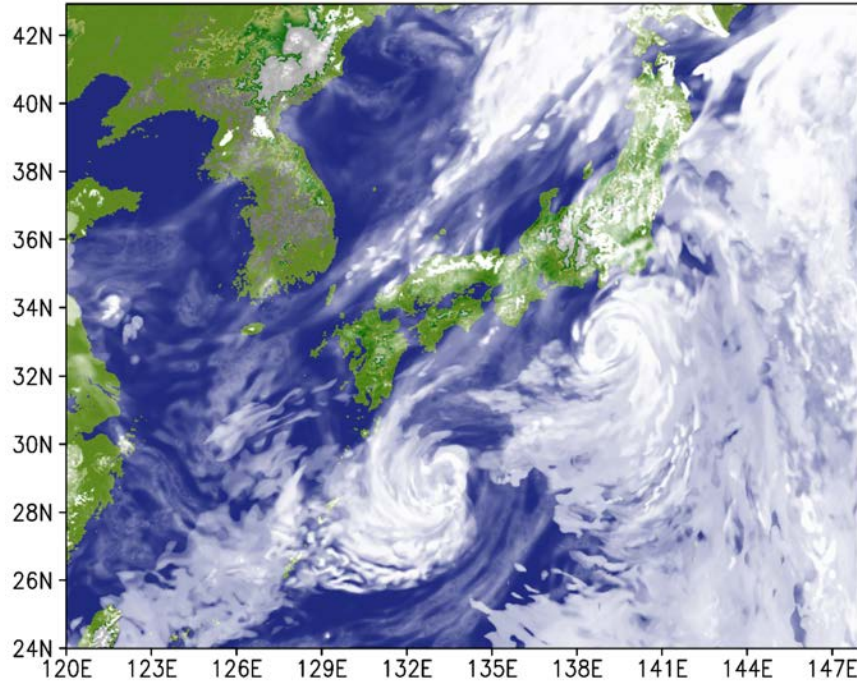


Figure 3.6: Total cloud cover result with ASUCA using a 2km resolution grid with real initial data.

3.3 Productivity

To examine the productivity of this solution we have analyzed the code and compared it against the reference implementation. The high-level results of this analysis is shown in Figure 3.7. In order to gain GPU support in addition to the already existing multi-core and multi-node parallelization, the code has grown by less than 4% in total, from 155k lines of code to 161k. Sanitizing the two code versions (removing white space, comments and merging continued lines), the code has grown by 12%, from 91k to 102k lines of code. 95% of the sanitized reference code is used as-is in the new implementation, while 5% or approximately 5k lines of code is replaced with approximately 15k new code lines.

Code changes and additions have the largest impact in terms of productivity. We we have analyzed the additional 15k lines of code in more detail. Figure 3.8 shows a breakdown of these changes and compares them to an estimate of what would be required with an OpenACC based implementation. The following methodology has been used for this analysis:

1. For the parallelization- and data layout DSL line count we have used global application model information gathered by Hybrid Fortran. As an estimate

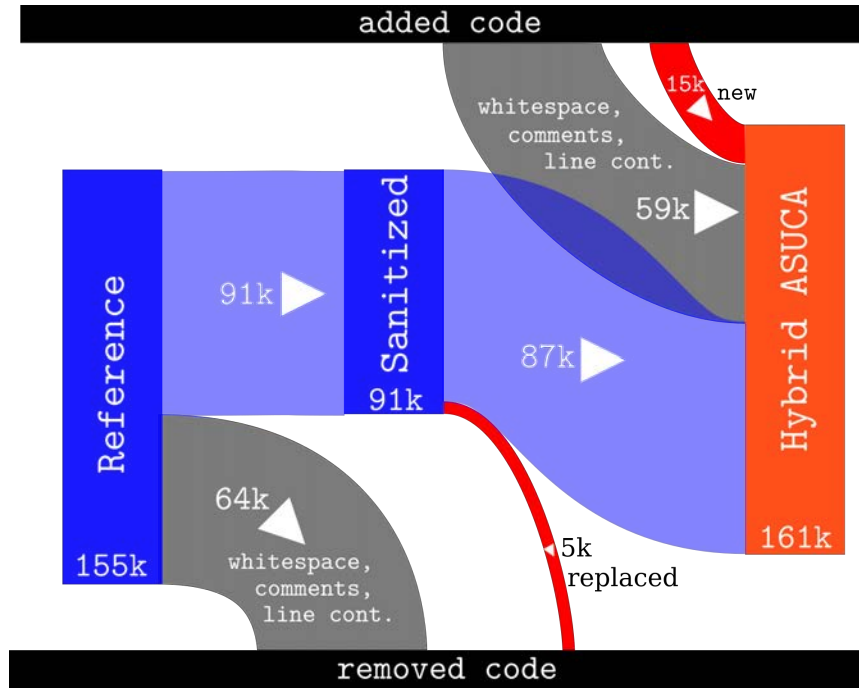


Figure 3.7: Flow of code from reference implementation to Hybrid ASUCA by number of lines of code.

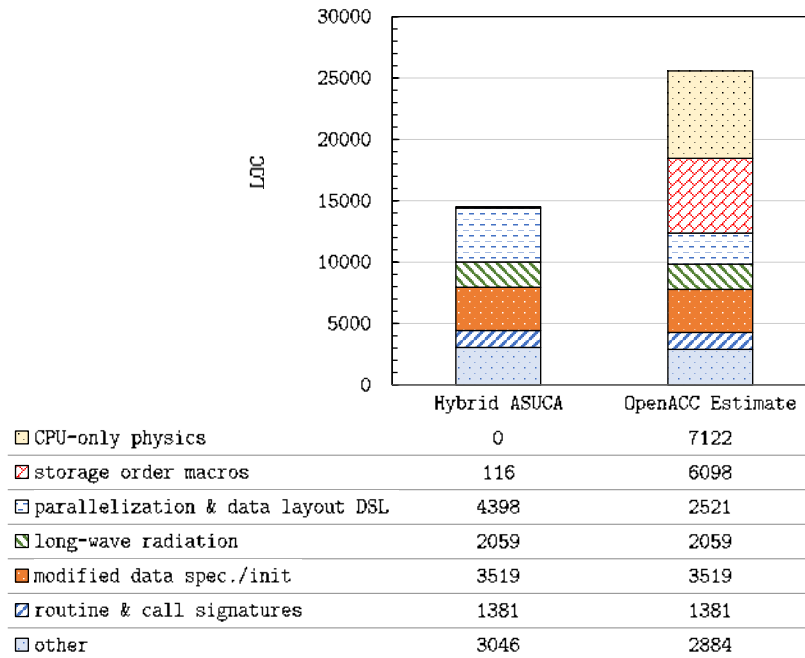


Figure 3.8: New code required for Hybrid ASUCA versus estimate of equivalent OpenACC implementation.

for OpenACC, the OpenACC version generated by one of the backend implementations has been used.

2. Since OpenACC does not offer a granularity abstraction we have used the global application model to arrive at a set of routines that require a kernel positioning change - the resulting code lines require duplication for the CPU in an equivalent OpenACC implementation (shown as “CPU-only physics” in figure 3.8).
3. We have used the OpenACC backend in Hybrid Fortran to count the lines of code where storage order macros are introduced in order to achieve a compile-time defined data layout.

“Parallelization & data layout DSL” refers to the number of code lines for `@parallelRegion` and `@domainDependant` directives in case of Hybrid Fortran, as well as `OpenACC !$acc` directives in case of OpenACC. Hybrid Fortran replaces the requirement for code changes to implement a varying data layout (“storage order macros”, 6098 LOC) as well as a code duplication for multiple parallelizations with varying granularities (“CPU-only physics”, 7122 LOC) with a higher number of DSL code lines compared to OpenACC (4398 vs. 2521 LOC). “Modified data specifications / initializations” (3519 LOC) as well as “routine & call signature” (1381 LOC) refers to changes applied to the setup of data and call parameter lists, respectively. These changes are necessary due to device code limitations and optimizations and are largely required for both the Hybrid Fortran version as well as a potential OpenACC solution, so we use the result from Hybrid ASUCA as an estimate for what would be required with OpenACC. Finally, one physics module concerning long-wave radiation (2059 LOC), has been replaced with a version that uses less local memory per thread (a factor of 10 improvement in that regard) to make it more GPU-friendly. We again estimate that an OpenACC version would have approximately the same code size.

This result shows that an equivalent OpenACC implementation of ASUCA is estimated to require approximately 11k LOC in additional changes compared to the Hybrid Fortran based implementation. When comparing to the sanitized reference codebase, an OpenACC user code would require approximately 28% of code lines to be changed or added, while Hybrid Fortran currently requires 16%.

3.4 Comparison to Previous Work

In the introduction chapter, Section 1.4.1, Dr. Shimokawabe’s work on applying GPUs to ASUCA was briefly discussed. In this section we compare that implementation to Hybrid ASUCA.

Table 3.1: Comparison of Hybrid ASUCA to GPU implementation by Shimokawabe et al.

Characteristic	Hybrid ASUCA	Shimokawabe
Main focus	Productivity	Efficiency at large scale (# of GPUs)
User language	Fortran	C++
Parallelization	DSL	DSL
Memory layout	Transformation	DSL
Dynamical core	Port of Ishida et al. 2010 [6]	Port of Ishida et al. 2010 [6]
Cloud microphysics	Warm rain (Kessler-type)	Warm rain (Kessler-type)
Radiation	MSM0705 based	n/a
Boundary layer physics	MSM0705 based	n/a
Strong scaling (# of GPUs)	42 P100 (equiv. to 159 K20x)	512 K20x
Strong scaling (# of GPUs)	n/a	4108 K20x

Table 3.1 shows the differences in focus between Hybrid ASUCA and Shimokawabe et al. [52]. Dr. Shimokawabe’s work was targeted at achieving computations on the largest scale available, i.e. a near-full utilization of TSUBAME 2.5 for a weak scaling result as well as using 512 GPUs for strong scaling. For comparison, Hybrid ASUCA was only scaled using strong scaling, reaching a performance peak at 42 GPUs, using a device model that is equivalent in theoretical performance to 159 GPUs used by Shimokawabe (P100 vs. K20x). On the other hand, with Hybrid ASUCA we were first able to port a complete model including operation-ready physical processes using a unified framework and we have demonstrated that only a minor rewrite is necessary for a previously CPU-targeted code to achieve full GPU compatibility (while Shimokawabe et al. relied on a full rewrite using a different user language and did not demonstrate radiation- as well as boundary layer physics).

3.5 Applicability to Further Models

In this section we use the model developed by the Consortium for Small Scale Modelling (COSMO) as a further example to discuss the applicability of the methodology and software tools developed in this thesis in the wider field of NWP. In order to further corroborate our thesis of general applicability, citations from leading experts are provided about the importance of the underlying problems that have been treated in this work.

The COSMO weather model is based on an initial development by Germany. A later formed consortium by Germany, Greece, Italy, Poland, Romania, Russia, and Switzerland has been pooling their software development resources to further enhance the model in various aspects and to adapt it to their local

requirements. Structurally it resembles ASUCA quite closely - these models share many commonalities, such as fully compressible, non-hydrostatic model equations, an Arakawa-C-type horizontal grid, a Lorenz-type vertical grid staggering using terrain-following coordinates, the use of generalized coordinates as well as a time integration scheme as proposed by Wicker and Skamarock. The main differences are in the spatial discretization methods (finite-difference versus finite-volume) and the physical parametrizations used. As ASUCA, COSMO is using Fortran language with a similar module structure [19] [85].

COSMO has been developed with NEC vector processors in mind. This includes manual granularity optimizations where one of the horizontal data dimensions is passed in to subroutines in order to provide enough fine-grained parallelism for vectorization. These optimizations come with a penalty in performance portability however [86]. For the GPU-based adaptation by the Swiss National Supercomputing Center (CSCS), the C++ template-based stencil library “STELLA” was applied to the dynamical core, combined with OpenACC used for the physical parametrization. For single-node performance, a speedup of 2.8x has been reported for the dynamical core comparing 8 cores of an AMD Opteron 6200 CPU (Interlagos) to one Tesla K20x GPU. For the physical processes, a speedup of 1.3x has been reported between the same systems, however using a flat MPI parallelization with 32 processes in order to utilize a full Cray XE6 node with two 16-core Interlagos CPUs. To achieve this performance for physical processes, the loop structure and parallelization granularity had to be adapted manually, creating a finer grained code structure while fusing some of vertical loops. Furthermore the storage order was adapted manually from KIJ to IJK for these parametrizations [87].

We expect that applying Hybrid Fortran to COSMO’s physical processes would be a boost in productivity. As this part of the code is “regularly updated and extended by many members of the user community” (Cumming et al. 2013), using the updated versions of the code in Switzerland currently entails merging the community updates with the aforementioned modifications required for GPU. Hybrid Fortran streamlines the process of adapting CPU-targeted code for a different memory layout and it provides an additional degree of freedom for granularity in the transformation step, as it has been demonstrated for ASUCA in this chapter. Since COSMO has been written for CPU vectorization, which ASUCA has not, Hybrid Fortran would need some slight improvements to fully accommodate COSMO’s original code: while Hybrid Fortran currently supports two levels of granularity (CPU and GPU), COSMO would need three (CPU-multi-core, CPU-vector and GPU).

Looking at the wider field of NWP and climate modelling, productivity of high-throughput ports for legacy applications has been problematic. Given the current literature we expect that mismatched code granularity has been the main

challenge in this regard. To quote Lawrence et al. in a review of many previous projects applying high-throughput architectures to NWP and climate models [88]:

“All these approaches were effectively addressing fine-grained parallelism in some way or other without addressing coarser grained concurrency, and all involved various levels of ‘intrusion’ into code, from adding / changing codes, to complete rewrites or translations.”

Norman et al. write about the reason the DOE’s ACME model cannot share a single source core for CPU and GPU due to register pressure [4]:

“The only remedy for this at present is to break the kernel up into multiple kernels. (...) On the CPU one would want to keep an element loop fused together for caching reasons.”

These examples show that a more streamlined and flexible approach to code granularity, such as demonstrated in this work, could have a strong impact on productivity in HPC going forward.

3.6 Concluding Remarks

In this chapter we have demonstrated the use of our method “Hybrid Fortran” as a valid GPU porting strategy for large data-parallel applications by showing the successful completion of GPU support for the operational weather model “ASUCA” with national importance to Japan.

By allowing ASUCA to retain 95% of its original codebase and limiting overall code growth to 4%, a hybrid version with CPU multi-core and GPU many-core support has been created with high potential for acceptance by the Japan Meteorological Agency. For a majority of the usage patterns within ASUCA our method allows a low-effort implementation. Granularity- and memory layout issues are solved by automated transformations rather than code divergence or changes of user code.

The efficiency gain enabled by a complete GPU port, including both dynamical core and physical processes, as achieved by Hybrid ASUCA, has been demonstrated by means of a performance model. For a performance analysis of implementation and method we refer to the following Chapter 4.

Through careful output data verification, correctness of the implementation has been ensured, with the root mean square error compared to the reference

version remaining lower than 10^{-9} , including for tests with real weather input data.

In a discussion of the applicability of our method to other models in the field of climate- and weather modelling we have identified a potentially wide-ranging impact of this work.

Performance Analysis

In this chapter the performance aspects of our work are discussed. In Section 4.1 a performance model is introduced for high throughput co-processors with the purpose of showing general speedup feasibility based on the communication- and workload characteristics of an application. Section 4.2 presents a reduced weather application, its performance model, the impact of certain optimization techniques as well as a comparison between OpenACC and the method proposed in this thesis. Performance results for Hybrid ASUCA as well as our concluding remarks are discussed in Sections 4.3 and 4.4, respectively.

4.1 Feasibility for High Throughput Co-processors

We call an application feasible for offloading to a co-processor when a speedup is attainable over execution on the host system, that is, when the total latency with offloading is smaller than without. Therefore inequality

$$t_H > t_D + t_{comm} \quad (4.1)$$

must hold, where t_H is the computation time for the host implementation, t_D is the computation time on the device and t_{comm} is the time spent for communication between host and device.

In this section we propose a simple performance model that can be used as a feasibility check. Intermediate communication exchanges and occupancy effects such as caching, register usage and scheduling are disregarded in this model. Under these assumptions the performance scales linearly with the grid size in a structured grid application, i.e. inequality 4.1 can be analyzed independent from grid size. For memory bandwidth bounded algorithms with sequential memory access we can express the time spent on each grid point as

$$\begin{aligned} t_{H_{bb}} &= n_i \frac{m \cdot b}{BW_H}, \\ t_{D_{bb}} &= n_i \frac{m \cdot b}{BW_D}, \end{aligned} \quad (4.2)$$

where n_i is the number of iterations over the data, m is the number of values to read and write from/to memory per point update, b is the byte length of each value and BW_H , BW_D are the memory bandwidths available to host and device.

For algorithms with random memory access this becomes

$$\begin{aligned} t_{H_{ra}} &= n_i \frac{m}{RA_H}, \\ t_{D_{ra}} &= n_i \frac{m}{RA_D}, \end{aligned} \quad (4.3)$$

with RA_H and RA_D denoting the number of random memory accesses per second (measured in GUP/s) [89].

For floating point computationally bounded algorithms these terms become

$$\begin{aligned} t_{H_{cb}} &= n_i \frac{c}{P_H}, \\ t_{D_{cb}} &= n_i \frac{c}{P_D}, \end{aligned} \quad (4.4)$$

where c is the number of floating point cycles spent per point update and P_H , P_D are the floating point operations per second available on host and device. Finally, t_{comm} can be expressed as

$$t_{comm} = \frac{m_{HtoD} \cdot b}{BW_{HtoD}}, \quad (4.5)$$

where m_{HtoD} is the number of values that need to be transferred from/to the device after n_i iterations and BW_{HtoD} is the bandwidth available to these transfers. Substituting these terms in inequality 4.1 and rearranging the terms such that algorithmic variables can be expressed in terms of constants given by hardware we arrive at

$$n_i \frac{m}{m_{HtoD}} > \frac{\frac{BW_H}{BW_{HtoD}}}{\left(1 - \frac{BW_H}{BW_D}\right)} \quad (4.6)$$

for sequential access memory bandwidth bounded,

$$n_i \frac{m}{m_{HtoD}} > \frac{\frac{bRA_H}{BW_{HtoD}}}{\left(1 - \frac{RA_H}{RA_D}\right)} \quad (4.7)$$

for random memory access bounded and

$$n_i \frac{c}{b \cdot m_{HtoD}} \left[\frac{\text{FLOP}}{B} \right] > \frac{P_H P_D}{BW_{HtoD}(P_D - P_H)} \left[\frac{\frac{\text{FLOP}^2}{s^2}}{\frac{B}{s} \frac{\text{FLOP}}{s}} \right] \quad (4.8)$$

for computationally bounded algorithms. Note that the term

$$\frac{c}{b \cdot m_{HtoD}} \quad (4.9)$$

resembles algorithmic intensity as defined in the Roofline model [90] - here we only use it for the memory transferred between host and device however.

Appendix A lists the relevant performance metrics that are used for the performance models and efficiency discussions in this work. As an example, when evaluating the inequalities 4.6, 4.7 and 4.8 for the TSUBAME 2.5 cluster[91] (Tesla K20x and 6-core Westmere Xeon X5670 CPU) we arrive at

$$n_i \frac{m}{m_{HtoD}} > 5.39 \quad (4.10)$$

for memory bandwidth bounded,

$$n_i \frac{m}{m_{HtoD}} > 0.26 \quad (4.11)$$

for random memory access bounded and

$$n_i \frac{c}{m_{HtoD}} > 111.74 \quad (4.12)$$

for computationally bounded algorithms run in double precision, i.e. at least 111.74 double precision floating-point operations or 5.39 GPU sequential memory operations are required per memory transfer from- and to the device (assuming the runtime is dominated by sequential memory access) to achieve a speedup when porting from TSUBAME 2.5 CPU to GPU.

4.2 Reduced Weather Application

As ASUCA is a fairly large and complex application, a reduced weather application is hereby presented as model that allows closer inspection, including full access to the underlying source code. It is constructed with ASUCA's code properties in mind in order to illustrate some of its engineering characteristics.

4.2.1 Application Model

The reduced weather application has the following domains: horizontal spatial domain IJ , vertical spatial domain K and time domain τ . Equation 4.13 lists the main equation governing this application: heat diffusion with T being the temperature defined in IJK -space, κ the thermal diffusivity and f the rate of heat input through radiation.

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T + f \quad (4.13)$$

This equation is solved numerically by using the explicit Euler method in three dimensions. In addition, the $k = 0$ and $k = nz$ boundaries are modeled as infinite heat wells with temperatures 330K and 200K respectively. Radiation f is assumed to be uniform and statically applied across the spatial domain. The KI and KJ boundaries are modeled cyclically. We initialize the temperature function to 0K with a 300K box from one quarter to three quarters of each spatial dimension and set a delta of 0.1s between time steps. Figure 4.1 shows slices at $j = 100$ for the resulting data with time steps at $t = 0s$, $t = 30s$ and $t = 90s$. This shows the heat diffusion, the radiation effect and the vertical boundary heat exchange (330K and 200K respectively).

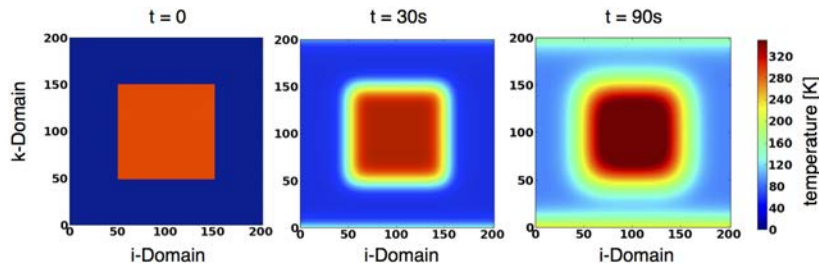


Figure 4.1: Output at $j = 100$.

In order to adhere to ASUCA's code structure the application is split into a dynamical core (which here only consists of the diffusion $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$) and physical processes with no interaction in the horizontal domain (radiation, vertical

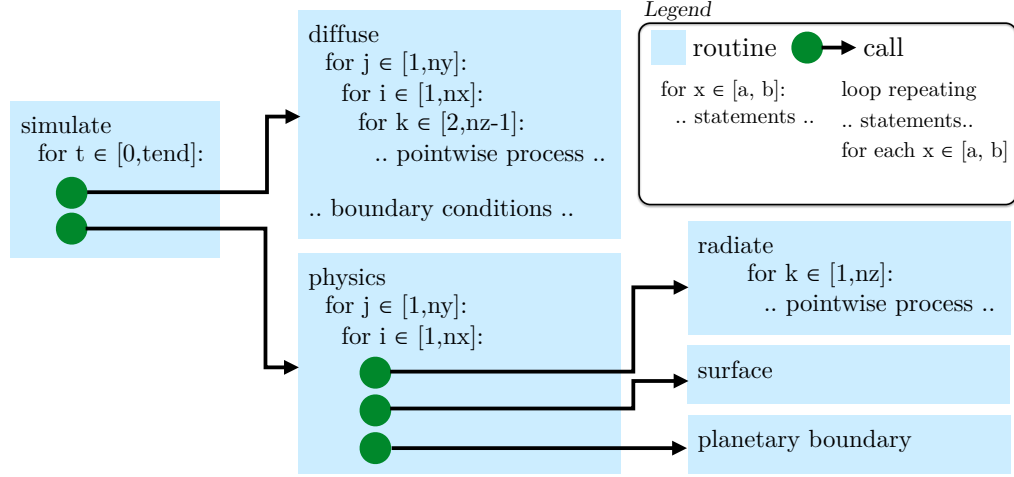


Figure 4.2: Code structure of the reduced weather application.

boundaries). The dynamical core consists of four tight stencil kernels - the inner IJK region as well as the boundary regions for the IJ, KI and KJ boundaries. Physical processes on the other hand employ (comparatively) coarse-grained parallelization, shown in Listing 4.1, with each parallel thread executing all physical processes for a single K-column. Figure 4.2 shows this code structure including the computationally relevant call graph, loops and data domains. The entire code is listed in Appendix C.

Listing 4.1: Physical processes.

```

subroutine run_physics(&
  & energy, energy_surf, energy_pbl)
  ! ..more specifications..
  real(8), intent(inout):: energy(0:nx+1,0:ny+1,nz)
  real(8), intent(in):: energy_surf(0:nx+1,0:ny+1)
  real(8), intent(in):: energy_pbl(0:nx+1,0:ny+1)

  do j = 0,ny+1
    do i = 0,nx+1
      call radiate(energy(i,j,:))
      call exchange_heat_with_boundary( &
        & energy(i,j,:), energy_surf(i,j), 1)
      call exchange_heat_with_boundary( &
        & energy(i,j,:), energy_pbl(i,j), nz)
    end do
  end do
end subroutine

```

The radiation process is modelled as a vector addition for columns in k , while the heat boundary exchange process calculates the heat exchange for scalar points.

Finally, the loop over time is applied in the top level subroutine `simulate` as

shown in the following listing:

Listing 4.2: Main time integration loop.

```

module simple_weather
  real(8), pointer :: energy(:,:,:)
  real(8), pointer :: energy_u(:,:,:)

  ! .. initialization, cleanup and output code ..

  subroutine simulate( &
    & start_time, end_time, timestep, output_timestep)
    ! ..more specifications..
    real(8), pointer :: energy_temp(:,:,:)

    time = start_time
    do while (.true.)
      if (modulo(time + 0.001d0, output_timestep) &
        & < 0.01d0) then
        call write_data(energy, "energy", time)
      end if
      call run_physics( &
        & energy, energy_surf, energy_pbl)
      call diffuse(energy_u, energy)

      energy_temp => energy_u
      energy_u => energy
      energy => energy_temp

      time = time + timestep
      if (time > end_time) then
        return
      end if
    end do
  end subroutine
end module

```

The application is tested by initializing the grid to zero Kelvin with a 300K box from one quarter to three quarters of each spatial domain. Figure 4.1 shows slices at $j = 100$ for the resulting data with timesteps at $t = 0$, $t = 30$ s and $t = 90$ s and one timestep executed for every 0.1s. This shows the energy diffusion, the radiation effect and the surface and planetary boundary heat exchange.

4.2.2 Performance Model

In this section a performance model is constructed for the reduced weather application, using the following assumptions:

1. the bandwidth bounded model for sequentially accessed memory can be applied to the inner region of **diffusion** as well as **radiation**, thus assuming an optimized storage order,
2. cached memory accesses do not affect execution time, i.e. such accesses are assumed to be hidden by host- or device memory accesses in the pipeline,

3. through sufficiently large spatial dimensions, one-dimensional and two-dimensional parametrizations can be omitted from the model, i.e. their runtime is negligible,
4. operations that can be hoisted out of loops are optimized by the compiler.

Diffusion requires a maximum of eight memory accesses per point update as well as six add- and two multiply instructions, being a Von-Neumann-type seven point stencil code. We define and calculate “arithmetic intensity” as

$$\frac{c}{b \cdot m} = \frac{8}{8 \cdot 8} \left[\frac{FLOP}{B} \right] = 0.125 \left[\frac{FLOP}{B} \right], \quad (4.14)$$

where m is the number of values to read and write from/to memory per point update, b is the byte length of each value and c is the number of floating point cycles spent per point update. In contrast to the roofline model we use floating point cycles rather than operations here in order to have a more generalized model that can be applied to multi-cycle floating point operations such as division or exponentiation [90].

The minimum arithmetic intensity for compute boundedness is $5.9 \left[\frac{FLOP}{B} \right]$ and $7.8 \left[\frac{FLOP}{B} \right]$ for Tesla K20x and P100, respectively, using the performance metrics from Appendix A). Thus, **diffuse** is clearly memory bandwidth bounded. This holds true even if six of the eight memory accesses are cached. Due to the lower random access memory bandwidth, the boundary region JK is clearly also memory bandwidth bounded. The same holds for the radiation process since it requires two memory accesses and one FLOP per point¹. We therefore conclude that the reduced weather application is memory bandwidth bounded for all processes.

The reduced weather application is therefore highly sensitive to the cache performance of the target architecture. Applying the above parameters to the speedup condition (equation 4.6), a speedup on GPU is feasible if the output is not required at every time step.

With some simplification in the domain boundaries (by assuming sufficiently large domains for halos to be neglectable) the model GPU execution time can be determined as

$$t_D = \frac{\Delta t_{output}}{\Delta t_{timestep}} (n_x \cdot n_y \cdot n_z \cdot \left(\frac{b \cdot m_{sa}}{BW_D} + \frac{b \cdot m_{HtoD}}{BW_{HtoD}} \right) + n_y \cdot n_z \cdot \frac{m_{ra}}{RAD}), \quad (4.15)$$

¹One should note however that this does not necessarily reflect typical physical processes in atmospheric models - such processes tend to consist of a mix of compute bounded- and memory bandwidth bounded algorithms.

and the model CPU execution becomes

$$t_H = \frac{\Delta t_{output}}{\Delta t_{timestep}} (n_x \cdot n_y \cdot n_z \cdot \frac{b \cdot m_{sa}}{BW_H} + n_y \cdot n_z \cdot \frac{m_{ra}}{RA_H}), \quad (4.16)$$

where b is the byte length of each value, m_{sa} is denoting the number of sequential memory accesses in the inner region of **diffuse** and **radiation** and m_{ra} is denoting the number of random access updates in the boundary region JK, n_x , n_y and n_z are denoting the grid sizes in x-, y- and z-direction, respectively, BW_H , BW_D and BW_{HtoD} are the bandwidths available on host, device and between host and device, respectively, and RA_H and RA_D are the number of random memory accesses per second (measured in GUP/s) [89]. It holds that $m_{sa} = 10$ without caching, $m_{sa} = 4$ with caching and $m_{ra} = 4$. See also Section 4.2.3 for a comparison of the measured performance with this model.

Table 4.1 lists the resulting model execution times for one output cycle where $n_x = n_y = n_z = 512$, $n_i = \frac{\Delta t_{output}}{\Delta t_{timestep}} = 100$ for single core, six core and GPU execution on TSUBAME 2.5, each with and without caching.

Table 4.1: Model Output Cycle Execution Time for TSUBAME 2.5.

	with cache	without cache
1 Core	5.48s	13.70s
1 Socket	2.62s	6.55s
1 GPU	0.35s	0.82s

With $\frac{t_{output}}{t_{timestep}} = 100$, $m = 4$ the expected speedup becomes

$$\frac{100 \cdot \frac{4}{BW_H}}{100 \cdot \frac{4}{BW_D} + \frac{1}{BW_{HtoD}}} = 7.53 \quad (4.17)$$

For large values of $n_i = \frac{\Delta t_{output}}{\Delta t_{timestep}}$ the model speedup asymptotically approaches the bandwidth speedup between CPU and GPU. e.g. 8.26 on TSUBAME 2.5.

4.2.3 Performance Results

This section facilitates the previously discussed reduced weather application in order to compare Hybrid Fortran with OpenACC as well as the performance model constructed in Section 4.1. For the performance measurements in this section the configuration as listed in Appendix B.1 has been used.

Performance Portable Storage Order

Table 4.2 shows the impact of storage order on execution time. For the reduced weather application, choosing a sub-optimal storage order impacts CPU execution time negatively by 35%, while on GPU the slowdown is 7.7x. This verifies the necessity of a flexible storage order for applications with similar data structures as ASUCA, as discussed in Section 3.1.2.

Table 4.2: Influence of Storage Order on Execution Time, $nx = ny = nz = 128$.

	IJK Order	KIJ Order
CPU Single Core	1.73s	1.28s
GPU (OpenACC) (Fastest Implementation)	0.10s	0.77s

“Naive” Parallelization

Table 4.3: Execution Time with “Naive” Parallelization.

$nx \cdot ny \cdot nz$	128^3	256^3
CPU Single Core Measurement	1.28s	8.20s
CPU Single Core Model w/ cache	0.74s	5.70s
CPU Single Core Model w/o cache	1.77s	13.91s
CPU 6 Core Measurement	0.40s	4.25s
CPU 6 Core Model w/ cache	0.38	2.84s
CPU 6 Core Model w/o cache	0.87	6.77s
GPU Measurement	163.13s	n/a

In order to establish a baseline performance we apply a basic parallelization to the reduced weather application with OpenACC and OpenMP directives. Storage order is made variable across the whole application by using macros for accessing and specifying multi-dimensional arrays. I-J-K order is used for GPU and K-I-J order for the CPU implementation. Table 4.3 shows the resulting CPU performance from this parallelization.

We conclude that the measured CPU performance is already well within the models with perfect and no cache. The GPU performance is however very slow - more than 400x slower than the six core CPU version, which is not in agreement with the models we have constructed. This is to be expected however, since no data region has yet been defined, without which the data is being copied over the slow PCI express bus for every kernel invocation. Further discussion for the reduced weather application therefore focuses on GPU performance.

Table 4.4: Execution Time with Data Region.

$nx \cdot ny \cdot nz$	128^3	256^3
GPU Measurement	0.095s	0.63s
GPU Model w/ cache	0.027s	0.19s
GPU Model w/o cache	0.087s	0.66s

Data Region

The amount of I/O to and from the GPU can be greatly reduced by using a data region in order to avoid host-to-device data copies at every kernel invocation. Table 4.4 shows the resulting performance with OpenACC.

Block Size

Table 4.5: Block Size Impact on Execution Time.

$nx \cdot ny \cdot nz$	128^3	256^3	$256^2 \cdot 512$
GPU Automatic Block size	0.095s	0.63s	2.99s
GPU 32 x 16 Block size	0.10s	0.55s	2.16s
GPU Model w/ cache	0.027s	0.19s	0.69s
GPU Model w/o cache	0.087s	0.66	2.60s

As Table 4.5 shows, up to 38% performance improvement is possible over the OpenACC version discussed in the previous paragraph by changing the default block size using vector clauses for each parallel loop. PGI accelerator uses a 128×1 block size for its OpenACC implementation by default while 32×16 has been experimentally determined to be the best block size for this application except for a small degradation for the smallest measured grid size.

Performance Comparison of Implementation Methods

Figure 4.3 shows the performance results for the reduced weather application with a Hybrid Fortran based implementation in comparison with the models introduced in Section 4.2.2 as well as the OpenMP/OpenACC based approach discussed above. For each of the implementations the highest performing version (32 x 16 block size with data region) has been selected for this comparison. This shows that measured CPU performance aligns well in between the two models (with and without cache). For the Kepler based GPU architecture on TSUBAME 2.5, L1 cache appears to be less effective for this application and the measured performance is closer to the model without cache. It is noteworthy that the code generated by Hybrid Fortran performs as well or better than the equivalent OpenACC code for both target architectures in this example.

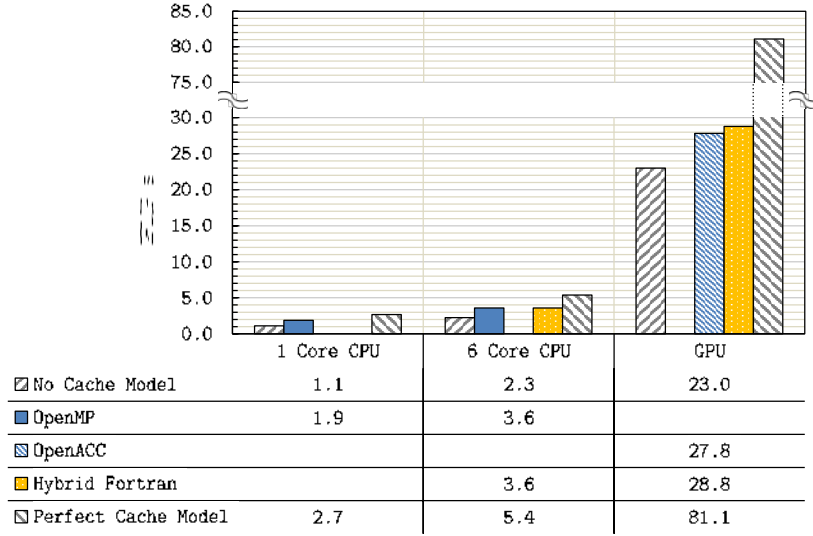


Figure 4.3: Comparing performance with the reduced weather application for handwritten vs. Hybrid Fortran generated vs. model on 256^3 Grid.

Overall we observe a speedup of 8x for the Hybrid Fortran version vs. 6-core CPU, compared to the speedup of 7.7x for the OpenACC version. The Hybrid Fortran implementation’s speedup is thus very close to the bandwidth increase of 8.2x between the two architectures (see Appendix A). On CPU the Hybrid Fortran implementation performs equivalently to the manually coded OpenMP implementation, which is unsurprising given that the CPU code generated by Hybrid Fortran is largely identical.

By defining

$$\text{efficiency} = \frac{\text{speedup}}{\frac{BW_D}{BW_H}} \quad (4.18)$$

we arrive at an efficiency of 97% for the Hybrid Fortran implementation versus 93% of the OpenACC implementation.

4.3 Hybrid ASUCA

In this Section we evaluate the performance of the Hybrid ASUCA application introduced in Chapter 3.

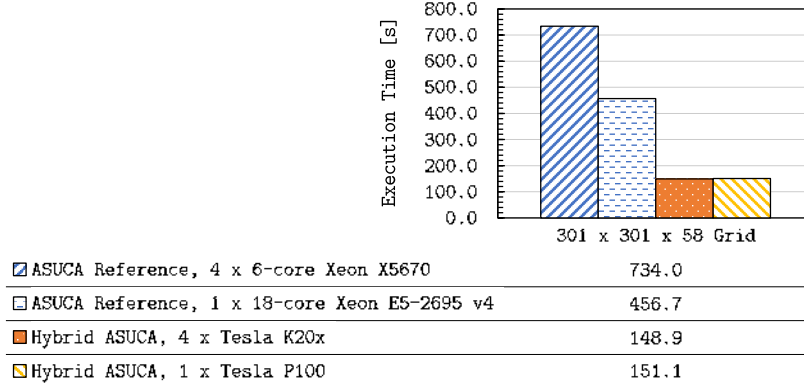


Figure 4.4: Execution time measurements for 75 long time steps of ASUCA executed in four different configurations.

4.3.1 Hybrid ASUCA Overall Kernel Performance

In this section, performance results for the Hybrid ASUCA implementation are discussed for a $301 \times 301 \times 58$ grid that is small enough for single GPU or single socket execution with the latest architecture (Tesla P100 on Reedbush-H), yet still allows a useful performance analysis in terms of occupancy. This allows to draw conclusions for the kernel performance as opposed to the communication overhead (which impacts performance more strongly, the more nodes are used for the same grid size, i.e. when applying strong scaling as will be shown in Section 4.3.2). Additionally, an older system (TSUBAME 2.5 with Tesla K20x) is used for comparison purposes. Since the two compared GPUs differ in their available device memory (16 GB for P100 vs. 6 GB for K20x) we compare four K20x GPUs to one P100. See Appendix B.2 for the software configuration used for this test.

Figure 4.4 shows the execution times for this configuration on four hardware/software configurations as listed. A speedup of 4.9x has been achieved by the port on Kepler GPU vs. Westmere 6-core Xeon X5670. On newer hardware however (Pascal vs. Broadwell) the speedup has so far been a more modest 3.1x.

4.3.2 Hybrid ASUCA Strong Scaling GPU Results

Using a full scale production grid, Hybrid ASUCA has been tested on the new Tokyo University cluster “Reedbush H” [83]. This cluster has two 18-core Xeon E5-2695 v4 CPUs per node as well as two NVIDIA Tesla P100 GPUs per node. At least seven nodes (14 sockets) or 24 GPUs are required for this test due to the grid’s memory requirements.

Figure 4.5 shows that a speedup of up to 3.5x on GPU has been achieved for

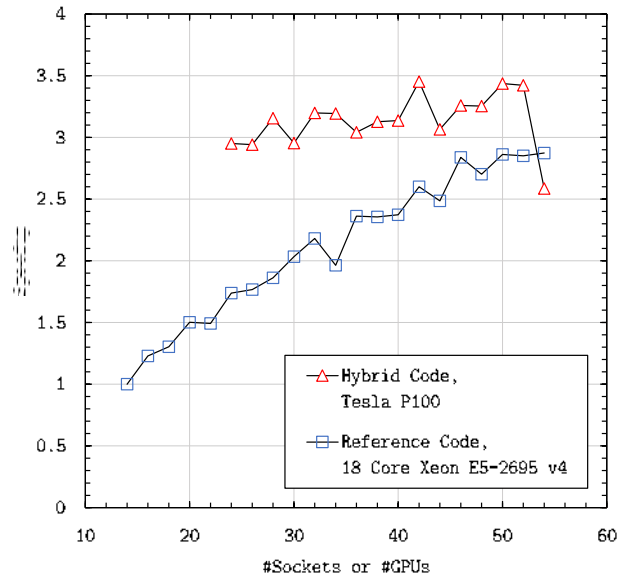


Figure 4.5: Strong scaling speedup on 1581 x 1301 x 58 ASUCA Grid.

this production run, on a 1581 x 1301 x 58 grid with 2km horizontal resolution using real world sample input data, when comparing the fastest time to solution on GPU with the minimum number of CPU sockets required to complete the simulation due to host memory requirements. For a given performance it shows that 24 GPUs can replace more than 50 18-core CPU sockets. When comparing the same number of GPUs and CPU sockets, the GPU is up to 76% faster. See Appendix B.3 for the software configuration used in this test.

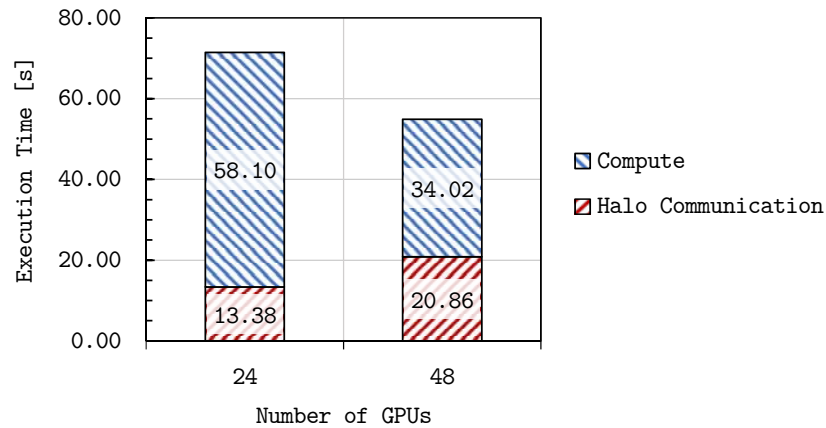


Figure 4.6: Impact of communication for strong scaling on 1581 x 1301 x 58 ASUCA Grid.

The following factors influence scalability and will need to be improved in

order to achieve better strong scaling:

1. MPI communications code has been used as-is, with no further optimizations applied with respect to the targeted cluster. When testing the impact of communication on performance on TSUBAME 3.0 using the minimally required 24 GPUs, as shown in Figure 4.6, communication requires approximately 13.4s or 18.7% of the overall runtime of 71.5s (to compute a 600 seconds simulation of the full regional grid in 2km resolution). When doubling the number of GPUs this increases to 20.9s while the compute time decreases from 58.1s to 34.0s, thus the communication then takes 38% of the runtime. Overlapping communication and computations has been shown to be effective in enabling better scaling by Shimokawabe et al. [54], thus this approach will be the first step to improve performance at larger scales.
2. Since GPUs require a large enough problem size per chip in order to have a sufficient number of threads to fill all schedulers, strong scaling is limited when the problem size per GPU becomes too small.

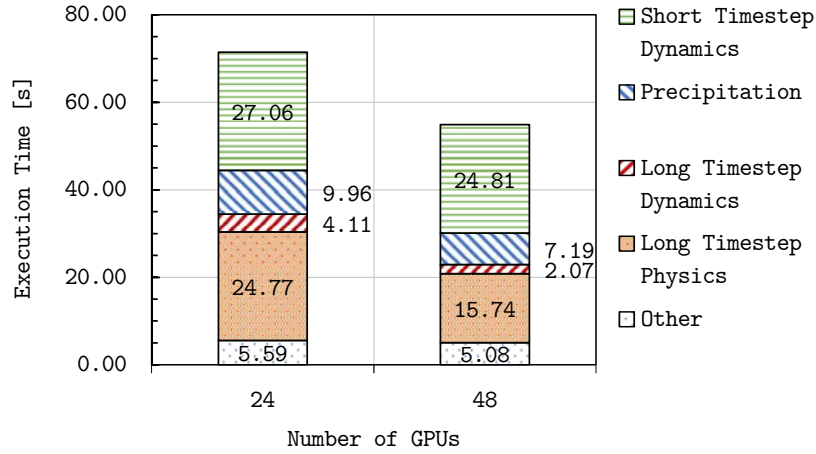


Figure 4.7: Impact of modules for strong scaling on 1581 x 1301 x 58 ASUCA Grid.

Figure 4.7 categorizes the performance impact of the different modules of ASUCA on performance, including communication. The simulation of fast moving sound- and gravity waves has the highest impact, followed by radiation- and boundary layer physics. Since the physics calculations do not require communication, the impact of fast moving dynamics increases with the number of nodes, rendering it the most important optimization target for larger scale simulations. For a listing of the configurations used to gather the data for figures 4.6 and 4.7, please refer to Appendix B.4.

4.4 Concluding Remarks

Efficiency in terms of performance for Hybrid Fortran as a GPU porting method has been demonstrated in this chapter by first constructing a reduced weather application along with a performance model, and subsequently applying and comparing various GPU porting techniques. With Hybrid Fortran a speedup of 8x is observed on TSUBAME 2.5 for GPU vs. a full CPU socket, a 4% improvement over the OpenACC version. On CPU the Hybrid Fortran implemented version performs equivalently to a manually coded OpenMP version.

The performance of Hybrid ASUCA has been evaluated on TSUBAME 2.5 and Reedbush-H. For a 301x301x58 grid, speedups of 4.9x and 3.1x are observed for K20x versus Westmere Xeon as well as P100 versus Broadwell Xeon, respectively. When scaling up to 24 Tesla P100, less than half the number of GPUs is required compared to Broadwell CPU sockets to achieve the same result.

Productivity Review of Hybrid Implementation Methods

In Chapter 2 we have presented a new method for targeting CPUs and GPUs with the same code in an efficient and productive manner. In this chapter we investigate the merit of this method when comparing it to other approaches available that allow targeting multiple processor architectures, including GPU. Application to legacy code is of special interest to us, as it adds additional dimensions to the problem in terms of the assumptions about the target architecture that have been made when originally writing these codes, and how this relates to productivity when adapting them to GPU. We regard establishing GPU compatibility for a vast body of highly useful legacy applications as an important problem in HPC.

We limit the discussion to methods that are suitable for HPC purposes, which precludes methods with a significant runtime overhead. In terms of applications we focus on weather and climate, which does not preclude such methods to be valuable outside those fields however. Performance aspects are therefore considered as given, i.e. we assume that users intend to and are able to use these tools to achieve compelling speedups on GPU - instead we focus on productivity aspects in order to achieve such implementations. In terms of programming languages we concentrate on frameworks with interfaces to C, C++ and Fortran, as these languages cover a vast majority of today's HPC applications.

This chapter is structured as follows: Section 5.1 provides an overview of methods that we analyze in this context. In Section 5.2 a characterization is given for these methods, including a productivity model and a scoring based on a series of use cases. Finally we conclude the chapter in Section 5.3 by presenting an interactive tool that has been created based on the knowledge gathered for this review.

5.1 Classification and Outline of Methods

In this section we provide an outline of notable hybrid implementation methods used in weather- and climate models, as well as methods that are similar to Hybrid Fortran in terms of memory layout- or code granularity optimizations.

5.1.1 Domain-Specific Languages

Domain-specific languages applied to stencil algorithms have been one method to abstract parallelization boiler-plate and memory layout for hybrid GPU/CPU code. For this matter, direct data accesses with loop- or thread indices are abstracted in the point-wise stencil code. This generally requires a complete rewrite of existing code.

Shimokawabe et al. Shimokawabe et al. (2014) have completed an implementation of the ASUCA dynamical core and a portion of its physical processes using their own C++ library. Memory access patterns are abstracted using a stencil DSL. Kernels are defined as C++ functors, passed to the library for parallelization. Shimokawabe et al. use a combination of MPI processes for multi-node parallelization, OpenMP for multi-GPU or multi-core CPU as well as CUDA for the GPU thread level parallelization. The CPU-only implementation appears to be sub-optimal since it requires boundary data exchange even within a single process that would have access to shared memory. Extensive work was done regarding inter-node and inter-GPU communication schemes as well as overlapping in order to optimize communication time or hide it behind computations. To facilitate this, a higher level abstraction is provided over MPI communication code, achieving a productive and efficient method to communicate the data structures required for this library with neighboring nodes. The grid structure is assumed to be regular in this framework. [52].

STELLA & GridTools Fuhrer et al. (2014) have implemented the dynamical core of COSMO for operational use at MeteoSwiss, using a purpose-built C++ stencil DSL library for structured grid applications (“STELLA”). STELLA shares many of its characteristics with the efforts of Shimokawabe et al.: memory access is abstracted using a stencil DSL, kernels are defined as functors, communicators are provided for regular structured grids [56]. The successor project under which this approach is being enhanced is called “GridTools” [58].

GGDML Jumah et al. have proposed a general grid definition and manipulation language (GGDML), an extension to Fortran with applicability to other languages, based on the requirements for three existing models: DYNAMICO, ICON and NICAM. This shares some of characteristics of GridTools as well

as the work by Shimokawabe et al., however with some important distinctions: firstly, GGDML is designed from the beginning to support icosahedral grids, and secondly, GGDML extends Fortran instead of relying on C++ user code, which facilitates migration from mostly Fortran based climate- and weather models. This work is part of the AIMES project [92]. Since it is a work-in-progress, currently not yet suitable for use in production, we exclude it from further scoring in subsequent sections.

Atlas The Atlas library by Deconinck et al. (2017) goes a step beyond Shimokawabe et al. and Fuhrer et al. in terms of the abstraction chosen for data structures. Rather than letting the user directly specify stencil offsets, it separates the concerns between function space and the underlying data model, including the chosen grid structure. This allows the user code to be written on a higher level, specifying mathematical operators such as nabla, gradient, laplacian and curl, and subsequently synthesizing the stencil operations required to numerically approximate these operators. The most compelling characteristic of this design is that function space code can be reused for multiple grid types, such as regular and icosahedral grids [93].

5.1.2 Directive-Based Porting Methods

Directives are used to steer compilers on how to optimize or parallelize already existing code for a specific hardware architecture. Privatization of thread-local data is generally supported in directive-based methods, while storage ordering is not. To switch between multiple parallelization granularities, code duplication is necessary. We are aware of the following implementations of atmospheric models with this method:

- Lapillonne et al. have implemented the relevant physical processes of COSMO for operational use at MeteoSwiss using OpenACC in Fortran [57].
- Govett et al. have ported the dynamical core of the Fortran-based Non-hydrostatic Icosahedral Model (NIM) to GPU, first using their own directive-based transformation tool “F2C-ACC” and later using OpenACC (after critical performance issues and bugs were fixed by the compiler vendors). At the time of this writing we are not aware of a GPU implementation of the NIM physical processes however, an issue that has reduced the potential speedup due to the communication overhead caused by running the physical processes on CPU [64][65].
- Norman et al. have implemented the “Accelerated Model for Climate and Energy” (ACME) for the U.S. DOE using OpenACC. As with ASUCA,

ACME’s physical processes are problematic for GPU due to their coarse-grained parallelization. GPU-specific code duplication was the only solution found when using OpenACC [4].

5.1.3 Granularity Optimization Methods

Kernel fusion has been the main approach to granularity optimization applied to GPGPU programming we are aware of. We take note of the following related work:

- Gysi and Hoefler have implemented kernel fusion as well as loop fusion for the aforementioned STELLA framework [94].
- Clement et al. are applying kernel fusion and an OpenACC/OpenMP targeted transformation to Fortran code in an ongoing effort as part of the C2SM project (as of yet unpublished¹).
- The granularity optimization technique used in Hybrid Fortran has been discussed in Section 2.2.

We are not aware of previous or ongoing work regarding kernel fission (and thus support for coarse-grained parallel programming) applied to GPUs.

5.1.4 Memory Layout Transforming Methods

While DSLs abstract the memory layout, they also require a full rewrite of the point-wise code. The following work allows for the reuse of existing code while keeping the performance portability gains of an abstracted memory layout.

Kokkos With the C++ library “Kokkos” (part of the Trilinos project), Edwards et al. have demonstrated that existing point-wise code can be reused even when the underlying data structures are converted to an abstracted memory layout. OpenMP, Pthreads as well as CUDA are provided as backends to the user code for this library. Granularity optimizations are not supported at the time of this writing, neither is Fortran user code [95].

ICON With a DSL created for the climate model “ICON”, Torres et al. have shown that the Fortran syntax can be extended to allow for an abstracted memory layout. A code transformation based on the ROSE compiler[96] is employed towards that goal [97]. Sawyer et al. have subsequently built on this abstraction

¹These open-sourced efforts can be found at <https://github.com/C2SM-RCM/claw-compiler>.

to port the dynamical core of ICON to GPU using OpenACC directives [98]. We are not aware of any granularity optimizations supported in the ICON DSL.

Hybrid Fortran Hybrid Fortran transforms memory access patterns for existing codes as discussed in Section 2.3.

5.2 Productivity Characterization

In this section, productivity of the various presented methods is characterized. For this purpose we present a productivity model, which is then applied to the methods for different use cases.

5.2.1 Evaluation Criteria: Data Structures

Data structures used in computer programming have machine-facing and application-facing characteristics - both of which are highly important in the context of HPC. In this section we characterize the methods described earlier in this chapter with respect to these aspects.

Memory Layout In methods with a fixed memory layout the user code is bound to a specific storage layout being executed on hardware. Changing the layout thus generally requires a rewrite of all access- and specification patterns of data structures. Using multiple layouts in the same code requires either duplicating code, or (more likely) the use of macros or C++ templates for reordering the layout at compile-time, i.e. the user of a method with fixed memory layout needs to add additional techniques to make it variable. In memory layout abstracting methods the user is given an abstraction that allows the framework to implement various layouts in the backend, with the same user-facing code. Abstractions however imply that legacy codes need to be rewritten in order to make use of the method. Transforming methods on the other hand take existing user code and reorder the memory layout, making adoption for legacy codes significantly easier.

Grid Support In terms of grid support we distinguish between tools with fixed and variable support. Fixed, again, means that a grid change requires a rewrite of the user code, while variable grid support allows the ability to change grids or use multiple grids with the same code.

Characterization of Methods Table 5.1 shows the data structure evaluation criteria, as discussed above, applied to the methods outlined in Section 5.1.

Table 5.1: Data structure characterization for hybrid implementation methods.

method	memory layout	grid
Shimokawabe et al.	abstracted	fixed
STELLA & GridTools	abstracted	fixed
Atlas	abstracted	variable
OpenACC & OpenMP	fixed	fixed
CLAW plus OpenACC	fixed	fixed
Kokkos	transforming	fixed
ICON	transforming	fixed
Hybrid Fortran	transforming	fixed

5.2.2 Evaluation Criteria: Control Structures

In this section we provide a characterization of the methods described in Section 5.1 with respect to their control structures.

Parallelization Schemes We call a parallelization scheme fixed if it is specific to one hardware architecture. CUDA is the most commonly used example for such a scheme. Since this chapter is limited to hybrid implementation methods, none of the listed have a fixed parallelization scheme. With an abstracted parallelization, a higher level abstraction is introduced, such as a library call that is passed kernel functors or a parallelization DSL. Transforming parallelizations can be applied to parallelize existing loops (assuming no loop carried dependencies), i.e. when omitting parallelization the application still produces the same results on a single thread.

Granularity The need for granularity transformations has been discussed in Section 1.3.6. As with memory layout we distinguish between methods with fixed-, abstracted- and transforming granularity. Fixed methods require rewrites for making the granularity finer or coarser. Granularity abstraction allows the user to explicitly specify the granularity for different architectures. Transforming methods, finally, apply granularity changes automatically to existing codes. For transformations we differentiate between methods allowing only one direction (either kernel fusion or fission) or both. Abstracted granularity also implied abstracted parallelization.

Communication Schemes Certain frameworks facilitate easier inter-node and inter-GPU communication by using data structure abstractions to synthesize the required halo regions needed for communications. We therefore distinguish between fixed- and abstracted communication schemes.

Programming Language Changing the underlying programming language of a legacy application carries significant productivity penalties. Support for an application’s original language is therefore an important characteristic to consider. We distinguish between methods for C++, Fortran and methods supporting both.

Characterization of Methods Table 5.2 shows the control structure evaluation criteria, as discussed above, applied to the methods outlined in Section 5.1.

Table 5.2: Control structure characterization for hybrid implementation methods.

method	parallelization	granularity	communication	language
Shimokawabe et al.	abstracted	fixed	abstracted	C++
STELLA & GridTools	abstracted	transforming (kernel fusion)	abstracted	C++
Atlas	abstracted	fixed	abstracted	C++ and Fortran
OpenACC & OpenMP	transforming	fixed	fixed	C++ and Fortran
CLAW plus OpenACC	transforming	transforming (kernel fusion)	fixed	Fortran
Kokkos	abstracted	fixed	fixed	C++
ICON plus OpenMP	transforming	fixed	fixed	Fortran
Hybrid Fortran	abstracted	abstracted	fixed	Fortran

5.2.3 Evaluation

For a productivity evaluation we analyze how the data- and control structure and characteristics (outlined in the Tables 5.1 and 5.2) manifest themselves with respect to specific use cases. For this purpose we quantify the productivity impact by estimating the importance of these characteristics relative to each other. The evaluation formula developed in this section is designed to allow for adjustments by the user of this evaluation, depending on which aspects match up with the requirements of an application.

We start by defining an application requirement as a binary vector of aspects r_α , separately for data- and control structures, with α being one of $[data, control]$.

Regarding data structures, each application either does or does not require (1) the ability to change the memory layout (e.g. due to GPUs having a different optimal layout compared to CPUs, as discussed in Section 1.3.6), and (2) the ability to use multiple grids with the same code. An application requiring the

former but not the latter aspect can be represented with r_{data} being $\begin{bmatrix} 1 & 0 \end{bmatrix}$, for an application requiring both it would be $\begin{bmatrix} 1 & 1 \end{bmatrix}$.

For control structures we define $r_{control}$ as a binary vector representing requirements for (1) the parallelization of fine-grained code [F2F], (2) the coarsening of fine-grained parallelism [F2C], (3) the refining of coarse parallelism [C2F], (4) the reimplementing of inter-node or inter-GPU communication [COMM], (5) the applicability to C/C++ code [CXX] and (6) the applicability to Fortran code [FOR], resulting in a vector with six components.

We then introduce two scoring matrices $A_{ij}(\alpha)$ with the i dimension being the methods outlined in Section 5.1 and the j dimension being the requirement criteria as outlined above for r_α . Higher scores in these matrices correspond to a higher investment required for each individual coding aspect, i.e. 0 in each column is ideal. The scoring within each of the two α -categories is done independently, i.e. the individual scores in $A_{ij}(\alpha)$ are to be regarded relative to each other, but separately for $\alpha \in [data, control]$. Characteristics are transformed to individual criteria scores for $A_{ij}(data)$ as follows:

- transforming capabilities for memory layout as well as variable grid capabilities correspond to an investment score of 0 in the respective columns,
- abstracted memory layout corresponds to an investment score of 1, signifying a complete rewrite,
- fixed memory layout- and grid correspond to an investment score of 3, signifying a complete rewrite as well as code duplication being necessary. We thus use a penalty factor of 3 for code duplication compared to rewrites, anticipating significant maintainability issues in future version - code duplication in complex earth system codes is almost always highly undesirable.

For $A_{ij}(control)$, the transformation from characteristics to criteria scores is chosen as follows, with 1 signifying minor rewrites of particular code aspects, 3 being minor rewrites as well as code duplication of these aspects, 5 signifying complete rewrites and 15 standing for complete rewrites as well as code duplication. We therefore retain the penalty factor of 3 between rewrite and code duplication, however we introduce a finer grained scoring for less invasive rewrites, estimating 20% of code to be rewritten for these aspects. Individual columns are scored based on the characterization table 5.2 as follows:

- Transforming parallelization corresponds to an investment score of 0 for [F2F], i.e. we consider necessary changes to user codes, such as the addition of directives, to be negligible compared to other techniques.

- Abstracted parallelization corresponds to an [F2F] score of 1, signifying a minor investment as only the loop headers need to be rewritten for this category,
- The [F2C] and [C2F] column highly depends on granularity transformation capabilities of the utilized framework. If none are available, a rewrite as well as code duplication is necessary to support different architectures efficiently, giving an investment score of 15. If the opposite direction is supported, e.g. kernel fusion where kernel fission is needed, it corresponds to a complete rewrite, i.e. a score of 5, as users need to split up kernels manually. A full support of the granularity transformation corresponds either to a score of 0 or 1, depending on the parallelization type as in [F2F].
- The communication part of an application is treated as a “minor” part, i.e. corresponding to 20% of the application. The [COMM] column therefore gives an investment score of 1 to frameworks with abstracted communication and a score of 3 to frameworks with fixed communication.
- The language columns [CXX] and [FOR] give scores of 0 or 5, depending on whether a framework supports the respective language.

Tables 5.3 and 5.4 show the resulting scoring matrices.

Table 5.3: Scoring matrix $A_{ij}(data)$.

method	memory layout change	grid change
Shimokawabe et al.	1	3
STELLA & GridTools	1	3
Atlas	1	1
OpenACC & OpenMP	3	3
CLAW plus OpenACC	3	3
Kokkos	0	3
ICON	0	3
Hybrid Fortran	0	3

As a further refinement of the scoring system we introduce weighting vectors w_α . The goal is to correct for some over- and under-representation that we would have for certain columns when applying uniform weighting. The [F2F], [F2C] and [C2F] granularity transformations are counted three times, even though a specific kernel can only require one of these. Similarly, a language change investment is underrepresented since applications typically contain a considerable amount of support code besides kernels that needs to be ported as well. We furthermore give grid changes a 50% higher weight compared to memory layout changes, since we regard it as a more fundamental change in the underlying code. The resulting weighting vectors are listed as follows:

Table 5.4: Scoring matrix $A_{ij}(\text{control})$.

method	F2F	F2C	C2F	COMM	CXX	FOR
Shimokawabe et al.	1	15	15	1	0	5
STELLA & GridTools	1	1	5	1	0	5
Atlas	1	15	15	1	0	0
OpenACC & OpenMP	0	15	15	3	0	0
CLAW plus OpenACC	0	0	5	3	5	0
Kokkos	1	15	15	3	0	5
ICON plus OpenMP	0	15	15	3	5	0
Hybrid Fortran	1	1	1	3	5	0

$$w_{data} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, w_{control} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 3 \\ 5 \\ 5 \end{bmatrix}. \quad (5.1)$$

In order to normalize scores we calculate the maximum investment score I_{max} (i.e. the worst case) using the following formula, with $A_{jmax}(\alpha)$ being the theoretical worst case investment for each column:

$$\begin{aligned} A_{jmax}(data) &= \begin{bmatrix} 3 & 3 \end{bmatrix} \\ A_{jmax}(control) &= \begin{bmatrix} 15 & 15 & 15 & 3 & 5 & 5 \end{bmatrix} \\ I_{max}(\alpha, r_\alpha) &= w_\alpha \cdot \text{diag}(r_\alpha) \cdot A_{jmax}(\alpha)^T \end{aligned} \quad (5.2)$$

Finally, the productivity score $P(r_\alpha)$ for each method is given as:

$$\begin{aligned} P_\alpha(r_\alpha) &= 1 - A_{ij}(\alpha) \cdot (w_\alpha \cdot \text{diag}(r_\alpha))^T \\ P(r) &= \text{mean}(P_{data}(r_{data}), P_{control}(r_{control})) \end{aligned} \quad (5.3)$$

For the scoring results we differentiate between a “Christmas tree use case”, i.e. a use case that requires all features given by r as a vector-of-ones, sec-

Table 5.5: Productivity score $P(r)$ for different use cases r .

method	Xmas tree	phys.	dyn. (Fortran)	dyn. (C++)
Shimokawabe et al.	0.35	0.46	0.54	0.79
STELLA & GridTools	0.47	0.55	0.54	0.79
Atlas	0.67	0.69	0.79	0.79
OpenACC & OpenMP	0.31	0.36	0.41	0.41
CLAW plus OpenACC	0.31	0.45	0.41	0.15
Kokkos	0.39	0.63	0.64	0.90
ICON	0.39	0.86	0.91	0.65
Hybrid Fortran	0.52	0.98	0.90	0.64

only a typical use case for earth system physics packages where coarse-grained Fortran code is to be ported to GPU (with $r_{control}$ being $[1 \ 0 \ 1 \ 0 \ 0 \ 1]$, r_{data} being $[1 \ 0]$), thirdly a typical use case for earth system dynamics where fine-grained Fortran code is ported (with $r_{control}$ being $[1 \ 0 \ 0 \ 1 \ 0 \ 1]$, r_{data} being $[1 \ 0]$) and lastly the same dynamics use case repeated for C++ codes. Table 5.5 shows the results from this analysis.

5.3 Concluding Remarks

There is no one-size-fits-all solution (yet) available to solve all the problems we have described in an efficient and productive manner. Generally, the more variability there is in the ideal runtime characteristics of an application on the chosen hardware platforms, the more useful abstractions become in dealing with this variability. At this point we want to advocate for a careful examination of these abstractions for each new use case, both on the application developer- as well as the framework developer side. For earth system software where the aforementioned variability is high, the Atlas library appears to deliver be the most promising approach. For physical packages and dynamical cores written in Fortran on a single grid, Hybrid Fortran has very promising productivity characteristics.

In addition to the productivity model presented in this chapter, an interactive spreadsheet based tool has been developed, allowing users to define their use case r_α , as well as adjusting the weights w_α . The result is shown as a heat map for the various characteristics on a flattened plane. The tool is publicly accessible².

²https://docs.google.com/spreadsheets/d/1zf7SBq_C6zQy-FuPZPPXaeVgsL26D6oSt8y7VhANTog

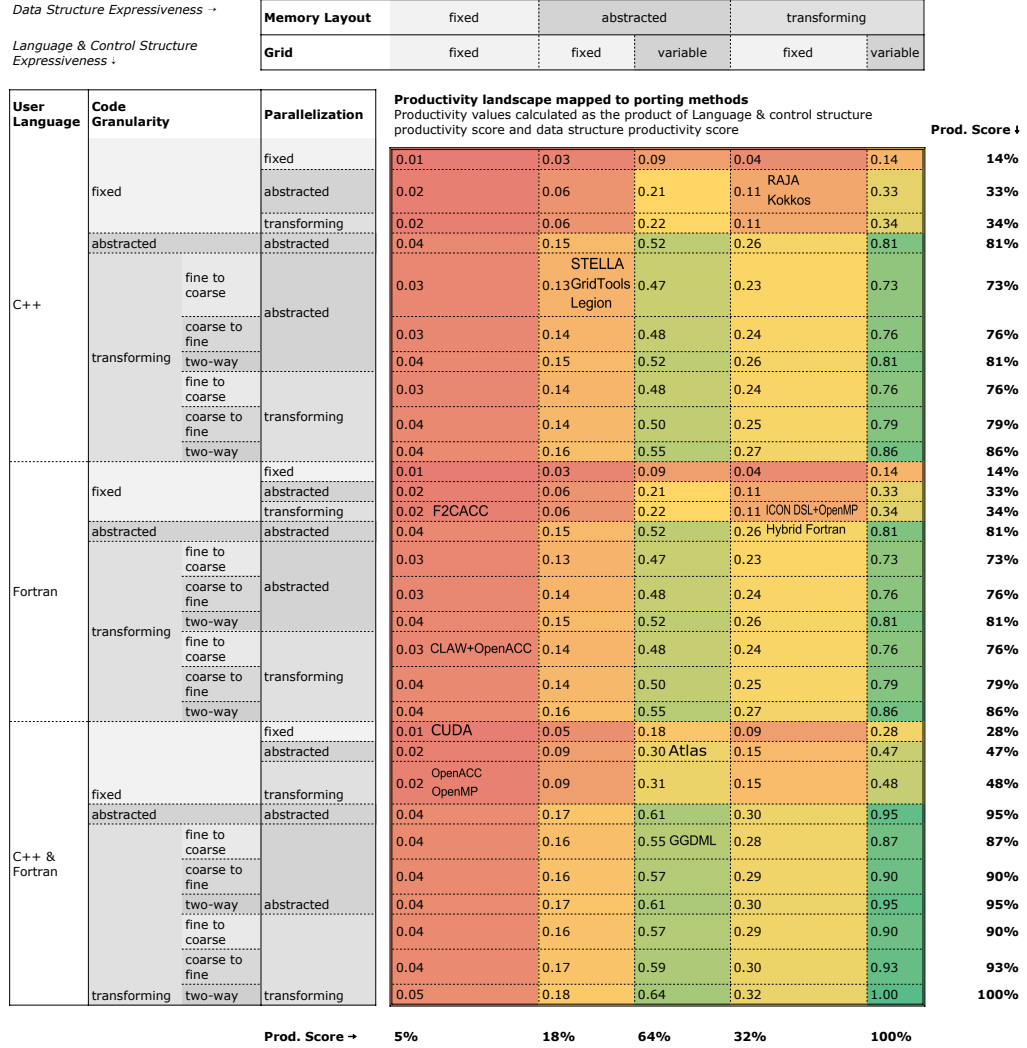


Figure 5.1: Interactive tool for productivity examination.

Conclusion and Future Work

Numerical weather prediction is one of the most demanding computing applications known to humankind. Not only is the demand for computational performance virtually insatiable, but the model approximations of the underlying physics is in constant flux, with researchers continuing to look for more exact, more stable and more efficient schemes in order to push the boundaries of accurate and meaningful weather prediction.

With silicon based microchip technology nearing its physical boundaries, this drive for change on the application side is today getting coupled with an accelerated development of hardware architecture, and thus programming paradigms, towards superscalar- and manycore processors. The combination of these two shifting environments creates friction, leading to a lowering of productivity of the programmers, maintainability of the applications as well as efficiency compared to the theoretical performance of the hardware.

In this work we have therefore demonstrated a method to significantly reduce this friction while modernizing large data parallel applications such as NWP. “Hybrid Fortran” is a method that allows bridging the gap between applications written with a particular hardware in mind, to allow a more seamless deployment of GPUs, a major trend in recent years in HPC. Hybrid Fortran provides abstractions over architecture specific programming while allowing existing CPU targeted user code to be reused as much as possible. This approach allows using a mixture of OpenMP, OpenACC, and CUDA Fortran programming interfaces in the backend, combined in a unified programming interface. A global application model is extracted automatically, facilitating the transformation process. Storage order and parallelization granularity use transformation schemes that adapt to the targeted hardware architecture, i.e. existing code written for CPU is transformed to match the architecture specific granularity and memory layout.

ASUCA, a regional weather prediction model of national importance to Japan, has been re-implemented by applying Hybrid Fortran to both dynamical core and physical processes. To our knowledge this is the first time a complete production

weather prediction model has been ported to GPU using a single unified programming paradigm. The new implementation shows a speedup of up to 4.9x on single GPU compared to a single Xeon socket. When scaling up to 24 Tesla P100, less than half the number of GPUs is required compared to contemporary Xeon CPU sockets to achieve the same result. Approximately 95% of the existing codebase has been reused for this implementation and our implementation has grown by less than 4% in total, even though it is now supporting GPU as well as CPU. A speedup of up to 3.5x on GPU was achieved for a full scale production run (1581 x 1301 x 58 grid with 2km horizontal resolution using real world sample input data).

Hybrid Fortran compares favourably in terms of productivity to other approaches common in this field, as is shown by the means of a productivity model. It is however important to note that there is no one-size-fits-all solution (yet) available to solve all common problems facing today's and tomorrow's earth system software. Our productivity model shows that choosing the right abstraction for each use case is paramount for productivity and maintainability of the codebases. Hybrid Fortran overall shows to be very promising for dynamical cores and physical processes written in Fortran, with the exception of multi-grid applications where code is to be reused on various grid types. Going forward towards exascale computing, leading experts in the field appear to be in agreement with us that code granularity is a major issue to solve in order to increase the productivity to transition to high-throughput architectures.

Considering that much of the changes still required in the user code are “mechanical” in nature, we expect additional productivity gains to be possible from further automation. We strive to achieve a solution where a Hybrid Fortran based transformation can be applied to large structured grid applications wholesale with minimal input required by the user. Extending the transformations based on the now available global application model will allow us to come significantly closer to this goal. More specifically, replacing the device data state attributes per routine and per data object with a generalized data region facility (and keeping track of all the involved data objects automatically) will greatly reduce the work needed to port to Hybrid Fortran. Among other applications this will make Hybrid Fortran viable for a full GPGPU port of the WRF model, an application that is one order of magnitude larger in code size compared to ASUCA and has global impact. Using COSMO as an example, generalizing the granularity abstraction to an arbitrary number of granularity levels and parallelization domains will benefit Hybrid Fortran's applicability.

Hardware Performance

Table A.1 lists the relevant performance metrics that are used for the performance models and efficiency discussions in this work. For a definition of the terms, please see Section 4.2.2. For the random access memory performance, the test included with the HPC Challenge[89] benchmark suite has been used (employing the unit “Giga Updates Per Second”). For the GPU, a CUDA port¹ has been used while for the CPU test an open source version by Underwood et al. has been employed[99] (using 2^{20} table size and 1000 update sets). The GPU bandwidth is specified as reported by the NVIDIA provided CUDA bandwidth test.

¹https://github.com/muellermichel/cuda_randomaccess

Table A.1: Hardware performance metrics.

Term	Hardware	Sustained Performance	Unit	System	Comment
P_{H1C}	Xeon X5670 (1 Core)	9.50	GFLOP/s	TSUBAME 2.5	In double precision [100]
P_H	Xeon X5670 (1 Socket)	57	GFLOP/s	TSUBAME 2.5	In double precision [100]
P_D	NVIDIA Tesla K20x	1030	GFLOP/s	TSUBAME 2.5	Linpack DP perf. [101]
P_D	NVIDIA Tesla P100	3900	GFLOP/s	Reedbush-H, Piz Daint	Linpack DP perf. [102]
BW_{H1C}	Xeon X5670 (1 Core)	9.80	GB/s	TSUBAME 2.5	according to Intel specifications[100]
BW_H	Xeon X5670 (1 Socket)	20.5	GB/s	TSUBAME 2.5	according to Intel specifications [100], confirmed by STREAM benchmark
BW_H	Xeon E5-2670 (1 Socket)	51.2	GB/s	Piz Daint	according to Intel specifications[103]
BW_H	Xeon E5-2695 v4 (1 Socket)	76.8	GB/s	Reedbush-H	according to Intel specifications [104], confirmed by STREAM benchmark
BW_D	NVIDIA Tesla M2050	108.6	GB/s	TSUBAME 2.0	Reported by CUDA bandwidth test
BW_D	NVIDIA Tesla K20x	169.4	GB/s	TSUBAME 2.5, Piz Daint,	Reported by CUDA bandwidth test
BW_D	NVIDIA Tesla P100	499.4	GB/s	Reedbush-H, Piz Daint, TSUBAME 3.0	Reported by CUDA bandwidth test on Reedbush-H
BW_{HtoD}	PCI Express 2.x	4.32	GB/s	TSUBAME 2.5	Reported by CUDA bandwidth test
BW_{HtoD}	PCI Express 3.x	10.96	GB/s	Reedbush-H, Piz Daint	Reported by CUDA bandwidth test on Reedbush-H
RA_H	Xeon X5670 (1 Socket)	0.12	GUP/s	TSUBAME 2.5	Reported by RandomAccess benchmark
RA_D	NVIDIA Tesla K20x	0.88	GUP/s	TSUBAME 2.5	Reported by RandomAccess benchmark

Performance Test Configurations

B.1 Reduced Weather Configuration for Performance Tests

The following configuration has been used for the results discussed in Section 4.2.3:

1. TSUBAME 2.5 has been used, i.e. Tesla K20x GPUs and Xeon X5670 CPUs.
2. For the CPU performance measurements, `ifort` and `icc` with `-fast` option has been used.
3. Thread affinity has been set to `compact` in order to ensure that the comparison is being done to exactly one of the two CPU socket.
4. For the GPU performance measurements, PGI Accelerator version 15.04 with `-fast` option and compute capability 3.x has been used.
5. GPU results include memory copy time from and to device unless stated otherwise.
6. Hybrid Fortran uses the OpenACC based backend implementation.
7. Hybrid Fortran uses the L1-cache-preferred setting for GPUs.
8. The number of time steps `NT` has been set to 100. Output to disk is only performed once at the end of the simulation.
9. All code measured has been open-sourced in the Hybrid Fortran GitHub repository¹.

¹Hybrid Fortran: <https://github.com/muellermichel/Hybrid-Fortran/tree/0.95>

B.2 ASUCA Configuration for Tests with 301 x 301 x 58 Grid

The following configuration has been used for the results discussed in Section 4.3.1:

1. 75 time steps have been executed, with $\Delta T = 8.0s$, thus having a simulation of 10 minutes physical time.
2. All ported modules enabled as depicted in Figure B.1
3. Real model input data
4. All floating point computations done in double precision
5. Intel Fortran compiler with `-fast -no-ipo` settings has been used for the reference code running on CPU. Versions 14.0.2 and 17.0.2 have been used for Westmere Xeon X5670 and Broadwell Xeon E5-2695 v4 respectively (inter-procedural optimization has resulted in compiler errors).
6. PGI accelerator has been used for the GPU measurements, with optimizations disabled (`-O0`) due to accuracy problems; Versions 16.9 and 16.10 have been used for K20x and P100 respectively.
7. Memory error correction (ECC) was enabled on GPU.

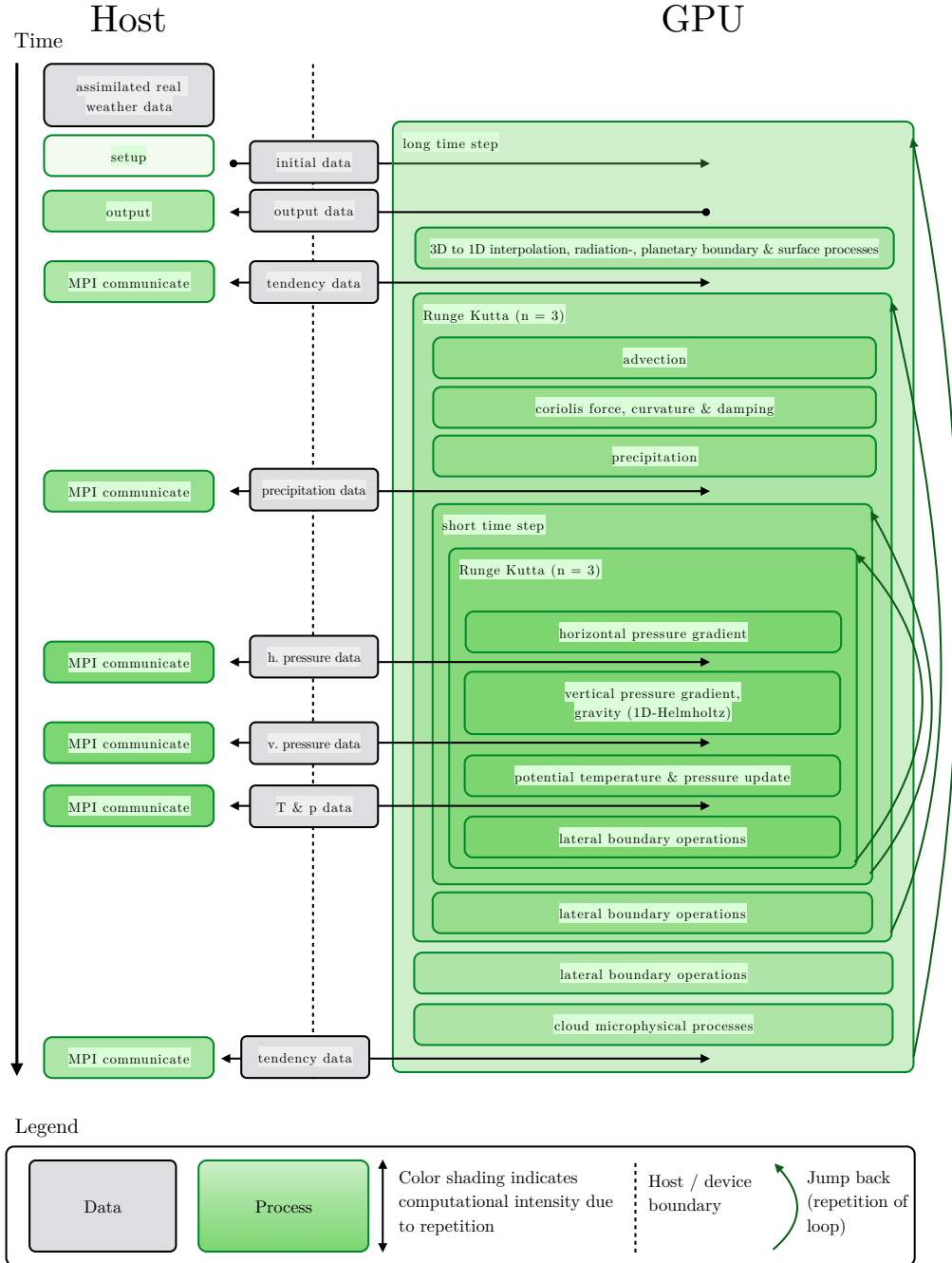


Figure B.1: Overview of simulated processes in Hybrid ASUCA when run on GPUs.

B.3 ASUCA Configuration for Tests with 1581 x 1301 x 58 Grid on Reedbush-H

The following configuration has been used for the strong scaling results discussed in Section 4.3.2:

1. All ported modules have been enabled as depicted in Figure B.1.
2. 36 time steps have been executed, with $\Delta T = 16.\overline{6}s$, thus having a simulation of 10 minutes physical time.
3. OpenMPI version 1.10.7 has been used.
4. The GPU implementation uses two MPI processes per node.
5. The CPU implementation uses one MPI process per node, with OpenMP core affinity settings to keep the threads in a compact manner on each socket (in order to optimize cache locality).
6. The horizontal domain (IJ) has been decomposed according to Appendix B.5.
7. Output to file has been limited to the beginning and the end of the simulation.
8. A single MPI process has been used only for problem setup and output on a separate node. This process is not included in the number of processes shown in the x-axis of the result as well as Appendix B.5.
9. Setup time is not included in the measured computation time and thus the speedup shown in Figure 4.5.
10. For the CPU implementation, Intel compiler version 17.0.2 has been used with `-fast` and `-no-ipo` flags (inter-procedural optimization has resulted in compiler errors).
11. For the GPU implementation, PGI Accelerator version 16.10 has been used with compiler optimizations disabled (an unresolved issue with incorrect computations has been identified in conjunction with compiler optimizations).
12. Memory error correction (ECC) was enabled on GPU.

B.4 ASUCA Configuration for Tests with 1581 x 1301 x 58 Grid on TSUBAME 3.0

The following configuration has been used for the module- and communication impact results discussed in Section 4.3.2:

1. All ported modules have been enabled as depicted in Figure B.1.
2. 36 time steps have been executed, with $\Delta T = 16.\overline{6}s$, thus having a simulation of 10 minutes physical time.
3. OpenMPI version 2.1.2 has been used.
4. The GPU implementation uses two MPI processes (i.e. GPUs) per node.
5. The horizontal domain (IJ) has been decomposed according to Appendix B.5.
6. A single MPI process has been used only for problem setup and output on a separate node. This process is not included in the number of processes shown in the x-axis of the result as well as Appendix B.5.
7. Both setup- and file output time is not included in the measured computation time and thus the speedup shown in the figures.
8. PGI Accelerator version 17.10 has been used with -O2 optimizations.
9. Memory error correction (ECC) was enabled on GPU.

B.5 ASUCA IJ domain decomposition

Table B.1 shows the composition of the horizontal domain into blocks for the 1581 x 1301 x 58 ASUCA grid runs.

Table B.1: MPI IxJ domain decomposition by number of processes.

# processes	IxJ decomposition (CPU and GPU)
7	1 x 7
8	2 x 4
9	1 x 9
10	2 x 5
11	1 x 11
12	3 x 4
13	1 x 13
14	7 x 2
15	3 x 5
16	4 x 4
17	1 x 17
18	3 x 6
19	1 x 19
20	2 x 10
21	3 x 7
22	2 x 11
23	23 x 1
24	2 x 12
25	5 x 5
26	2 x 13
27	3 x 9

# processes	IxJ decomposition (GPU only)
28	2 x 14
30	3 x 10
32	2 x 16
34	2 x 17
36	3 x 12
38	2 x 19
40	4 x 10
42	2 x 21
44	4 x 11
46	2 x 23
48	4 x 12
50	2 x 25
52	4 x 13

APPENDIX C

Sources

This section presents the sources for the reduced weather application¹ used for performance analysis in Chapter 4. Please note that Hybrid Fortran itself is completely Open Source².

C.1 Reduced Weather: Diffusion

Listing C.1: Diffusion.

```
subroutine diffuse( &
  & energy_u, energy)
  ! ..more specifications..
  real(8),intent(out):: energy_u(0:nx+1,0:ny+1,nz)
  real(8),intent(in):: energy(0:nx+1,0:ny+1,nz)

  ! ----- inner region IJK: -----
  do j = 1,ny
    do i = 1,nx
      do k = 2, nz-1
        energy_u(i,j,k) = energy(i,j,k) * &
          & (1.0d0 - 6 * diffusion_velocity) &
          & + diffusion_velocity * ( &
            & energy(i-1,j,k) + energy(i+1,j,k) + &
            & energy(i,j-1,k) + energy(i,j+1,k) + &
            & energy(i,j,k-1) + energy(i,j,k+1) &
            & )
      end do
    end do
  end do

  ! -- boundary regions IJ, IK, JK: ----
  do j = 1,ny
    do i = 1,nx
      energy_updated(i,j,1) = (1.0d0 - 5 * diffusion_velocity) &
```

¹The complete source code for the reduced weather application can also be compiled and run from the Hybrid Fortran GitHub repository: https://github.com/muellermichel/Hybrid-Fortran/tree/0.95/examples/simple_weather

²Accessible, including sample applications, under <https://github.com/muellermichel/Hybrid-Fortran>.

```

& * energy(i,j,1) + diffusion_velocity * ( &
&   energy(i-1,j,1) + energy(i+1,j,1) + &
&   energy(i,j-1,1) + energy(i,j+1,1) + &
&   energy(i,j,2) &
& )
energy_updated(i,j,nz) = (1.0d0 - 5 * diffusion_velocity) * &
&   energy(i,j,nz) + diffusion_velocity * ( &
&   energy(i-1,j,nz) + energy(i+1,j,nz) + &
&   energy(i,j-1,nz) + energy(i,j+1,nz) + &
&   energy(i,j,nz-1) &
& )
end do
end do

do i = 0,nx+1
do k = 1,nz
energy_updated(i,0,k) = (1.0d0 - 2 * diffusion_velocity) &
& * energy(i,0,k) + diffusion_velocity * ( &
&   energy(i,1,k) + energy(i,nx+1,k) &
& )
energy_updated(i,nx+1,k) = (1.0d0 - 2 * diffusion_velocity) &
& * energy(i,nx+1,k) + diffusion_velocity * ( &
&   energy(i,nx,k) + energy(i,0,k) &
& )
end do
end do

do j = 0,ny+1
do k = 1,nz
energy_updated(0,j,k) = (1.0d0 - 2 * diffusion_velocity) &
& * energy(0,j,k) + diffusion_velocity * ( &
&   energy(1,j,k) + energy(nx+1,j,k) &
& )
energy_updated(nx+1,j,k) = (1.0d0 - 2 * diffusion_velocity) &
& * energy(nx+1,j,k) + diffusion_velocity * ( &
&   energy(nx,j,k) + energy(0,j,k) &
& )
end do
end do
end subroutine

```

C.2 Reduced Weather: Code Generated by Hybrid Fortran for Diffusion

Listing C.2: Diffusion code generated for CUDA Fortran

```

attributes(global) subroutine hfk0_diffuse(...)
! ... use statements ...
real(8), device :: energy_u(0:nx+1,0:ny+1,nz)
real(8), device :: energy(0:nx+1,0:ny+1,nz)
! ... further specifications and initializations
i = (blockidx%x - 1) * blockDim%x + threadidx%x + 1 - 1
j = (blockidx%y - 1) * blockDim%y + threadidx%y + 1 - 1
if (i .GT. nx .OR. j .GT. ny) then
return
end if
! .... computational code within parallel region ...
end subroutine

```

```

subroutine diffuse(thermal_energy_updated, thermal_energy)
  ! ... use statements ...
  real(8), intent(out) :: energy_u(0:nx+1,0:ny+1,nz)
  real(8), device :: energy_u_hfdev(0:nx+1,0:ny+1,nz)
  real(8), intent(in) :: energy(0:nx+1,0:ny+1,nz)
  real(8), device :: energy_hfdev(0:nx+1,0:ny+1,nz)
  ! ... further specifications ...
  energy_u_hfdev(:,:,:) = 0
  if ( nx+1 - 0 + 1 .gt. 0 .and. ny+1 - 0 + 1 .gt. 0 .and. nz - 0 .
      ↪ gt. 0 ) then
    energy_hfdev(:,:,:) = energy(:,:,:)
  end if
  ! ... further initializations ...
  cugridSizeX = ceiling(real(nx) / real(32))
  cugridSizeY = ceiling(real(ny) / real(16))
  cugridSizeZ = 1
  cugrid = dim3(cugridSizeX, cugridSizeY, cugridSizeZ)
  cublock = dim3(32, 16, 1)
  call hfk0_diffuse <<< cugrid, cublock >>>( &
    ! ... passing in required scalars
    energy_u_hfdev, energy_hfdev &
    & )
  ! ... error handling code ...
  ! ... repeat the above for all boundary regions ...
  if ( nx+1 - 0 + 1 .gt. 0 .and. ny+1 - 0 + 1 .gt. 0 .and. nz - 0 .
      ↪ gt. 0 ) then
    energy_u(:,:,:) = energy_u_hfdev(:,:,:)
  end if
end subroutine

```

C.3 Reduced Weather: Physical Processes

Listing C.3: Physical processes.

```

subroutine run_physics(&
  & energy, energy_surf, energy_pbl)
  ! ..more specifications..
  real(8), intent(inout) :: energy(0:nx+1,0:ny+1,nz)
  real(8), intent(in) :: energy_surf(0:nx+1,0:ny+1)
  real(8), intent(in) :: energy_pbl(0:nx+1,0:ny+1)

  do j = 0,ny+1
    do i = 0,nx+1
      call radiate(energy(i,j,:))
      call exchange_heat_with_boundary( &
        & energy(i,j,:), energy_surf(i,j), 1)
      call exchange_heat_with_boundary( &
        & energy(i,j,:), energy_pbl(i,j), nz)
    end do
  end do
end subroutine

```

C.4 Reduced Weather: Radiation

Listing C.4: Radiation process.

```

subroutine radiate(energy)

```

```

! ..more specifications..
real(8), intent(inout), dimension(nz) :: energy

radiation_intensity = 0.1d0
do k=1,nz
    energy(k) = energy(k) + radiation_intensity
end do
end subroutine

```

C.5 Reduced Weather: Heat Exchange

Listing C.5: Surface / planetary heat exchange.

```

subroutine exchange_heat_with_boundary( &
    & energy, boundary_energy, boundary_level)
! ..more specifications..
real(8), intent(inout), dimension(nz) :: energy
real(8), intent(in) :: boundary_energy
integer(4), intent(in) :: boundary_level

transfer_velocity = 0.01d0
energy_transfer_to_boundary = &
    & transfer_velocity * &
    & (energy(boundary_level) - boundary_energy)
energy(boundary_level) = energy(boundary_level) &
    & - energy_transfer_to_boundary
end subroutine

```

C.6 Reduced Weather: Time Integration

Listing C.6: Main time integration loop.

```

module simple_weather
real(8), pointer :: energy(:,:,:)
real(8), pointer :: energy_u(:,:,:)

! .. initialization, cleanup and output code ..

subroutine simulate( &
    & start_time, end_time, timestep, output_timestep)
! ..more specifications..
real(8), pointer :: energy_temp(:,:,:)

time = start_time
do while (.true.)
    if (modulo(time + 0.001d0, output_timestep) &
        & < 0.01d0) then
        call write_data(energy, "energy", time)
    end if
    call run_physics( &
    & energy, energy_surf, energy_pbl)
    call diffuse(energy_u, energy)

    energy_temp => energy_u
    energy_u => energy
    energy => energy_temp
end do
end module

```

```
        time = time + timestep
        if (time > end_time) then
            return
        end if
    end do
end subroutine
end module
```


Acknowledgements

I would like to express my deepest gratitude to Prof. Takayuki Aoki, not just for his inexhaustible support and guidance during my doctoral course at the Tokyo Institute of Technology, but also for believing in me and letting me start my research at his laboratory.

I am deeply grateful to Dr. Kohei Aranami and Dr. Tabito Hara at the Japan Meteorological Agency, not just for allowing me to use the original ASUCA code, but also for providing expert support when I was lost, and last but not least, for preparing real initial data.

I would like to thank Dr. Takashi Shimokawabe. Without his support and inspiration I would not have been able to pursue this work. Never will I forget the moment where he challenged me when I was still a master student - could this problem be solved in an easier, more user friendly way? Attempting to answer this question was what started the work underlying this thesis.

I would like to express my gratitude to Prof. Naoya Maruyama for allowing me to pursue further research in his group at the RIKEN Advanced Institute for Computational Science in Kobe, Japan.

Special thanks go to Dr. Martin Schlueter and Dr. Johan Hysing, for very valuable discussions and for providing further validating use cases to my method.

I cannot think of a way to express enough gratitude to Dr. Mohamed Wahib Attia. His encouragement was always there when I needed it most, as was his never-ending support and helpful tips on how to navigate the academic world.

My deepest appreciation I would like to extend to Dr. Chester Wong, Dr. Christian Conti, Ms. Ania Brown, Mr. Michael Gestrich and all members of Aoki Laboratory for their encouragement and support, for providing an always friendly atmosphere and lots of good laughs. Special thanks are extended to Dr. Conti for his extensive proofreading and valuable and critical input to my work.

To the world's most helpful secretary, Ms. Chieko Takahashi, I would like to express my deepest gratitude - her assistance has gone above and beyond.

I would like to thank the staff members of the TSUBAME- and Reedbush supercomputers - their help was always very timely and much appreciated.

Finally, to my family - I would like to thank my two-year-old son Leo for always being cheerful even when I was too busy for playing, my wife Mihoko for her endless love and support and my parents Anita and Erwin for always having my back even though I have been in a place far away for several years.

Bibliography

- [1] J. Preshing, “A look back at single-threaded CPU performance,” 2012. Retrieved 2017-12-22: <http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance>.
- [2] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobbs’s journal*, vol. 30, no. 3, pp. 202–210, 2005.
- [3] TOP500, “Top500 list november 2017,” 2017. Retrieved 2017-12-24 from Top500.org: <https://www.top500.org/lists/2017/11>.
- [4] M. R. Norman, A. Mametjanov, and M. Taylor, “Exascale programming approaches for the accelerated model for climate and energy,” 2017.
- [5] M. Sakamoto, J. Ishida, K. Kawano, K. Matsubayashi, K. Aranami, T. Hara, H. Kusabiraki, C. Muroi, and Y. Kitamura, “Development of yin-yang grid global model using a new dynamical core ASUCA,” 2014.
- [6] 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*, vol. 40, pp. 0511–0512, 2010.
- [7] R. W. Keyes, “Fundamental limits of silicon technology,” *Proceedings of the IEEE*, vol. 89, pp. 227–239, Mar 2001.
- [8] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *ACM SIGARCH Computer Architecture News*, vol. 39.3, pp. 365–376, ACM, 2011.
- [9] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Higher Education, 1st ed., 1992.
- [10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in *Computer graphics forum*, vol. 26.1, pp. 80–113, Wiley Online Library, 2007.
- [11] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *Commun. ACM*, vol. 53, pp. 58–66, Nov. 2010.
- [12] J. G. Charney, R. Fjörtoft, and J. v. Neumann, “Numerical integration of the barotropic vorticity equation,” *Tellus*, vol. 2, no. 4, pp. 237–254, 1950.

- [13] P. Lynch, “The ENIAC forecasts: A re-creation,” *Bulletin of the American Meteorological Society*, vol. 89, no. 1, pp. 45–55, 2008.
- [14] V. Bjerknes, *Das Problem der Wettervorhersage: betrachtet vom Standpunkte der Mechanik und der Physik*. Meteorologische Zeitschrift, 1904.
- [15] P. Lynch, “Richardson’s forecast: What went wrong?,” in *NWP50*, NOAA, 2004.
- [16] E. W. Woolard, “L. F. Richardson on weather prediction by numerical process,” *Monthly Weather Review*, vol. 50, no. 2, pp. 72–74, 1922.
- [17] R. Courant, K. Friedrichs, and H. Lewy, “Über die partiellen Differenzengleichungen der mathematischen Physik,” *Mathematische annalen*, vol. 100, no. 1, pp. 32–74, 1928.
- [18] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock, “Development of a next generation regional weather research and forecast model,” in *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the use of high performance computing in meteorology*, vol. 1, pp. 269–276, World Scientific, 2001.
- [19] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, and T. Reinhardt, “Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities,” *Monthly Weather Review*, vol. 139, no. 12, pp. 3887–3905, 2011.
- [20] S.-J. Lin, “A “vertically lagrangian” finite-volume dynamical core for global models,” *Monthly Weather Review*, vol. 132, no. 10, pp. 2293–2307, 2004.
- [21] M. J. Kurowski, W. W. Grabowski, and P. K. Smolarkiewicz, “Anelastic and compressible simulation of moist deep convection,” *Journal of the Atmospheric Sciences*, vol. 71, no. 10, pp. 3767–3787, 2014.
- [22] Y. Ogura and N. A. Phillips, “Scale analysis of deep and shallow convection in the atmosphere,” *Journal of the Atmospheric Sciences*, vol. 19, no. 2, pp. 173–179, 1962.
- [23] D. R. Durran, “Improving the anelastic approximation,” *Journal of the atmospheric sciences*, vol. 46, no. 11, pp. 1453–1461, 1989.
- [24] D. Lilly, “A comparison of incompressible, anelastic and boussinesq dynamics,” *Atmospheric research*, vol. 40, no. 2-4, pp. 143–151, 1996.
- [25] P. R. Bannon, “On the anelastic approximation for a compressible atmosphere,” *Journal of the Atmospheric Sciences*, vol. 53, no. 23, pp. 3618–3628, 1996.

- [26] A. Arakawa and C. S. Konor, “Unification of the anelastic and quasi-hydrostatic systems of equations,” *Monthly Weather Review*, vol. 137, no. 2, pp. 710–726, 2009.
- [27] A. Staniforth and N. Wood, “Recent research for dynamical cores of non-hydrostatic, deep-atmosphere, unified models,” in *ECMWF Seminar Proceedings: Recent Developments in Numerical Methods for Atmosphere and Ocean Modelling*, ECMWF, Reading, pp. 15–26, 2004.
- [28] W. C. Skamarock and J. B. Klemp, “A time-split nonhydrostatic atmospheric model for weather research and forecasting applications,” *Journal of Computational Physics*, vol. 227, no. 7, pp. 3465 – 3485, 2008. Predicting weather, climate and extreme events.
- [29] A. Bátkai, P. Csomós, I. Faragó, A. Horányi, and G. Szépszó, *Mathematical Problems in Meteorological Modelling*. Springer, 2016.
- [30] A. A. White, B. J. Hoskins, I. Roulstone, and A. Staniforth, “Consistent approximate models of the global atmosphere: shallow, deep, hydrostatic, quasi-hydrostatic and non-hydrostatic,” *Quarterly Journal of the Royal Meteorological Society*, vol. 131, no. 609, pp. 2081–2107, 2005.
- [31] Z. Janjic, “Recent advances in global nonhydrostatic modeling at ncep,” in *Proceedings of the ECMWF Workshop on Non-hydrostatic Modeling*, pp. 95–107, 2010.
- [32] S.-J. Choi and S.-Y. Hong, “A global non-hydrostatic dynamical core using the spectral element method on a cubed-sphere grid,” *Asia-Pacific Journal of Atmospheric Sciences*, vol. 52, no. 3, pp. 291–307, 2016.
- [33] T. Davies, M. J. Cullen, A. J. Malcolm, M. Mawson, A. Staniforth, A. White, and N. Wood, “A new dynamical core for the met office’s global and regional modelling of the atmosphere,” *Quarterly Journal of the Royal Meteorological Society*, vol. 131, no. 608, pp. 1759–1782, 2005.
- [34] L. Keller and A. Friedmann, “Differentialgleichungen für die turbulente Bewegung einer kompressiblen Flüssigkeit,” in *Proc. First. Int. Congr. Appl. Mech.*, pp. 395–405, 1924.
- [35] M. Germano, U. Piomelli, P. Moin, and W. H. Cabot, “A dynamic subgrid-scale eddy viscosity model,” *Physics of Fluids A: Fluid Dynamics*, vol. 3, no. 7, pp. 1760–1765, 1991.
- [36] Y. Ogura and T. Takahashi, “Numerical simulation of the life cycle of a thunderstorm cell,” *Monthly Weather Review*, vol. 99, no. 12, pp. 895–911, 1971.

- [37] J. M. Straka and E. R. Mansell, “A bulk microphysics parameterization with multiple ice precipitation categories,” *Journal of Applied Meteorology*, vol. 44, no. 4, pp. 445–466, 2005.
- [38] E. Kessler, “On the continuity of water substance,” *ESSA Technical Memorandum*, vol. IERTM-NSSL 33, 1967.
- [39] M. Murakami, “Numerical modeling of dynamical and microphysical evolution of an isolated convective cloud,” *Journal of the Meteorological Society of Japan. Ser. II*, vol. 68, no. 2, pp. 107–128, 1990.
- [40] L. Prandtl, “Bericht über Untersuchungen zur ausgebildeten Turbulenz,” *Zeitschrift für angew. Math. u. Mechanik*, vol. 5, no. 2, pp. 136–139, 1925.
- [41] G. L. Mellor and T. Yamada, “Development of a turbulence closure model for geophysical fluid problems,” *Reviews of Geophysics*, vol. 20, no. 4, pp. 851–875, 1982.
- [42] B. P. Briegleb, “Delta-eddington approximation for solar radiation in the near community climate model,” *Journal of Geophysical Research: Atmospheres*, vol. 97, no. D7, pp. 7603–7612, 1992.
- [43] Y. J. Kaufman, D. Tanré, O. Dubovik, A. Karnieli, and L. A. Remer, “Absorption of sunlight by dust as inferred from satellite and ground-based remote sensing,” *Geophysical Research Letters*, vol. 28, no. 8, pp. 1479–1482, 2001.
- [44] R. M. Goody, “A statistical model for water-vapour absorption,” *Quarterly Journal of the Royal Meteorological Society*, vol. 78, no. 336, pp. 165–169, 1952.
- [45] F. Guichard and F. Couvreux, “A short review of numerical cloud-resolving models,” *Tellus A: Dynamic Meteorology and Oceanography*, vol. 69, no. 1, p. 1373578, 2017.
- [46] W. Y. Y. Cheng and W. R. Cotton, “Sensitivity of a cloud-resolving simulation of the genesis of a mesoscale convective system to horizontal heterogeneities in soil moisture initialization,” *Journal of Hydrometeorology*, vol. 5, no. 5, pp. 934–958, 2004.
- [47] K. Saito, J.-I. Ishida, K. Aranami, T. Hara, T. Segawa, M. Narita, and Y. Honda, “Nonhydrostatic atmospheric models and operational development at JMA,” *Journal of the Meteorological Society of Japan. Ser. II*, vol. 85, pp. 271–304, 2007.
- [48] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU cluster for high performance computing,” in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pp. 47–47, IEEE, 2004.

- [49] M. Harris, “Optimizing CUDA,” *SC07: High Performance Computing With CUDA*, 2007.
- [50] J. Michalakes and M. Vachharajani, “GPU acceleration of numerical weather prediction,” *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.
- [51] 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 Computer Science*, vol. 4, pp. 1535–1544, 2011.
- [52] T. Shimokawabe, T. Aoki, and N. Onodera, “High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’14, (Piscataway, NJ, USA), pp. 251–261, IEEE Press, 2014.
- [53] L. J. Wicker and W. C. Skamarock, “Time-splitting methods for elastic models using forward time schemes,” *Monthly Weather Review*, vol. 130, no. 8, pp. 2088–2097, 2002.
- [54] 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 the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.
- [55] H. Dursun, K.-i. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta, “In-core optimization of high-order stencil computations,” in *PDPTA*, pp. 533–538, 2009.
- [56] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. C. Schulthess, “Towards a performance portable, architecture agnostic implementation strategy for weather and climate models,” *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 45–62, 2014.
- [57] X. Lapillonne and O. Fuhrer, “Using compiler directives to port large scientific applications to GPUs: An example from atmospheric science,” *Parallel Processing Letters*, vol. 24, no. 01, 2014.
- [58] O. Fuhrer, “Grid Tools: Towards a library for hardware oblivious implementation of stencil based codes.” Retrieved 2017-12-22 from <http://www.pasc-ch.org/projects/2013-2016/grid-tools>, 2014.
- [59] J. Michalakes and M. Vachharajani, “GPU acceleration of numerical weather prediction,” in *IPDPS*, pp. 1–7, IEEE, 2008.

- [60] W. Vanderbauwhede and T. Takemi, “An investigation into the feasibility and benefits of GPU/multicore acceleration of the weather research and forecasting model,” in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 482–489, IEEE, 2013.
- [61] G. Ruetsch, E. Phillips, and M. Fatica, “GPU acceleration of the long-wave rapid radiative transfer model in WRF using CUDA Fortran,” in *Many-Core and Reconfigurable Supercomputing Conference*, 2010.
- [62] J. Mielikainen, B. Huang, H.-L. A. Huang, and M. D. Goldberg, “GPU acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (WRF) model,” *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 2, pp. 555–562, 2012.
- [63] M. Govett, J. Middlecoff, and T. Henderson, “Running the NIM next-generation weather model on GPUs,” in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 792–796, IEEE Computer Society, 2010.
- [64] M. Govett, J. Middlecoff, and T. Henderson, “Directive-based parallelization of the NIM weather model for GPUs,” in *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on*, pp. 55–61, IEEE, 2014.
- [65] M. Govett, J. Rosinski, J. Middlecoff, T. Henderson, J. Lee, A. MacDonald, N. Wang, P. Madden, J. Schramm, and A. Duarte, “Parallelization and performance of the NIM weather model on CPU, GPU and MIC processors,” *Bulletin of the American Meteorological Society*, 2017.
- [66] N. I. James Beyer, “Targeting GPUs with OpenMP 4.5 device directives,” 2016. Retrieved 2017-12-22 from NVIDIA: <http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf>.
- [67] OpenACC, “The OpenACC application programming interface version 2.5,” 2015. Retrieved 2017-12-22 from OpenACC.org: http://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf.
- [68] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era,” *Computer*, vol. 41, pp. 33–38, July 2008.
- [69] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS ’67 (Spring)*, (New York, NY, USA), pp. 483–485, ACM, 1967.

- [70] T. Hara, K. Kawano, K. Aranami, Y. Kitamura, M. Sakamoto, H. Kusabiraki, C. Muroi, and J. Ishida, "Development of the physics library and its application to ASUCA," *CAS/JSC WGNE Res. Activ. Atmos. Oceanic Modell*, vol. 42, pp. 0505–0506, 2012.
- [71] T. Hara, "Upgrade of the operational JMA mesoscale model and implementation of improved Mellor-Yamada level 3 scheme," in *22nd Conference on Weather Analysis and Forecasting/18th Conference on Numerical Weather Prediction*, 2007.
- [72] R. Nagasawa, "Improvement of a radiation process for the non-hydrostatic model," in *Proc. of 12th Conference on Atmospheric Radiation, American Meteorological Society, P2-10*, <http://ams.confex.com/ams/pdfpapers/113356.pdf>, 2006.
- [73] H. Kitagawa, "Radiation scheme (in Japanese)," *Report of Numerical Prediction Division*, vol. 46, pp. 16–31, 2000.
- [74] J. T. Kiehl and C. S. Zender, "A prognostic ice water scheme for anvil clouds," *World Climate Programme Research, WCRP-90*, no. 713, pp. 167–188, 1995.
- [75] S. Yabu, S. Murai, and H. Kitagawa, "Clear-sky radiation scheme (in Japanese)," *Report of Numerical Prediction Division*, vol. 51, pp. 53–64, 2005.
- [76] W. Zhong and J. D. Haigh, "Improved broadband emissivity parameterization for water vapor cooling rate calculations," *Journal of the Atmospheric Sciences*, vol. 52, no. 1, pp. 124–138, 1995.
- [77] M. Nakanishi and H. Niino, "An improved Mellor–Yamada level-3 model with condensation physics: Its design and verification," *Boundary-Layer Meteorology*, vol. 112, pp. 1–31, Jul 2004.
- [78] A. C. M. Beljaars and A. A. M. Holtslag, "Flux parameterization over land surfaces for atmospheric models," *Journal of Applied Meteorology*, vol. 30, no. 3, pp. 327–341, 1991.
- [79] M. Ikawa and K. Saito, "Description of a non-hydrostatic model developed at the forecast research department of the MRI," *Technical Reports of the Meteorological Research Institute*, vol. 28, p. 238pp, 12 1991.
- [80] E. Kessler, *On the Distribution and Continuity of Water Substance in Atmospheric Circulations*, pp. 1–84. Boston, MA: American Meteorological Society, 1969.
- [81] J. Kwiatkowski, "Evaluation of parallel programs by measurement of its granularity," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 145–153, Springer, 2001.

- [82] C. C. Douglas, J. Hu, M. Kowarschik, U. Rde, and C. Wei, “Cache optimization for structured and unstructured grid multigrid,” *Electronic Transactions on Numerical Analysis*, vol. 10, pp. 21–40, 2000.
- [83] T. University, “Reedbush introduction (in Japanese),” 2017. Retrieved 2017-12-22 from Tokyo University: http://www.cc.u-tokyo.ac.jp/system/reedbush/reedbush_intro.html.
- [84] B. Doty, “The grid analysis and display system,” *GRADS Manual*, vol. 10, p. 148, 1995.
- [85] U. Schttler, G. Doms, and C. Schraff, “A description of the nonhydrostatic regional COSMO-model part VII: user’s guide,” *Deutscher Wetterdienst Rep. COSMO-Model*, vol. 4, p. 142, 2008.
- [86] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, “STELLA: a domain-specific tool for structured grid methods in weather and climate models,” in *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov 2015.
- [87] B. Cumming, C. Osuna, T. Gysi, M. Bianco, X. Lapillonne, O. Fuhrer, and T. C. Schulthess, “A review of the challenges and results of refactoring the community climate code COSMO for hybrid Cray HPC systems,” *Proceedings of Cray User Group*, 2013.
- [88] B. N. Lawrence, M. Rezný, R. Budich, P. Bauer, J. Behrens, M. Carter, W. Deconinck, R. Ford, C. Maynard, S. Mullerworth, *et al.*, “Crossing the chasm: How to develop weather and climate models for next generation computers?,” *under review for Geosci. Model Dev.*, 2017.
- [89] P. R. Luszczyk, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, “The HPC challenge (HPCC) benchmark suite,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 213, 2006.
- [90] 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.
- [91] TOP500, “TSUBAME 2.5 ranking,” 2013. Retrieved 2017-12-22 from Top500.org: <http://www.top500.org/system/178249>.
- [92] N. Jumah, J. Kunkel, G. Zngl, H. Yashiro, T. Dubos, and Y. Meurdesoif, “GGDML: Icosahedral models language extensions,” 2017.
- [93] W. Deconinck, P. Bauer, M. Diamantakis, M. Hamrud, C. Khnlein, P. Maciel, G. Mengaldo, T. Quintino, B. Raoult, P. K. Smolarkiewicz, and N. P. Wedi, “Atlas : A library for numerical weather prediction and

- climate modelling,” *Computer Physics Communications*, vol. 220, no. Supplement C, pp. 188 – 204, 2017.
- [94] T. Gysi and T. Hoefer, “Integrating STELLA & MODESTO: Definition and optimization of complex stencil programs,” 2017.
- [95] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling many-core performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-specific languages and high-level frameworks for high-performance computing.
- [96] D. Quinlan, “ROSE: Compiler support for object-oriented frameworks,” *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.
- [97] R. Torres, L. Linardakis, J. Kunkel, and T. Ludwig, “ICON DSL: A domain-specific language for climate modeling,” in *International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO*, 2013.
- [98] W. Sawyer, G. Zaengl, and L. Linardakis, “Towards a multi-node OpenACC implementation of the ICON model,” in *EGU General Assembly Conference Abstracts*, vol. 16, 2014.
- [99] K. D. Underwood, S. J. Plimpton, R. B. Brightwell, C. T. Vaughan, and M. Davis, “A simple synchronous distributed-memory algorithm for the HPC RandomAccess benchmark,” tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States); Sandia National Laboratories, Albuquerque, NM, 2006.
- [100] Intel, “Intel Xeon processor X5670,” 2010. Retrieved 2017-12-22 from Intel: http://ark.intel.com/products/47920/Intel-Xeon-Processor-X5670-12M-Cache-2_93-GHz-6_40-GTs-Intel-QPI.
- [101] T. Prickett Morgan, “NVIDIA launches not one but two Kepler2 GPU coprocessors,” 2012. Retrieved 2017-12-22 from The Register: http://www.theregister.co.uk/2012/11/12/nvidia_tesla_k20_k20x_gpu_coprocessors/?page=2.
- [102] R. Xu, F. Han, and N. Dandapanthu, “Application performance on P100-PCIe GPUs,” 2017. Retrieved 2017-12-22 from Dell: http://en.community.dell.com/techcenter/high-performance-computing/b/general_hpc/archive/2017/03/14/application-performance-on-p100-pcie-gpus.

- [103] Intel, “Intel Xeon processor E5-2670,” 2012. Retrieved 2017-12-22 from Intel: http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI.
- [104] Intel, “Intel Xeon processor E5-2695 v4,” 2016. Retrieved 2017-12-22 from Intel: http://ark.intel.com/products/91316/Intel-Xeon-Processor-E5-2695-v4-45M-Cache-2_10-GHz.