Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Language Extensibility and Configurability to Support Stencil Code Development

Nabeeh Jum'ah

Fakultät für Mathematik, Informatik und Naturwissenschaften

Fachbereich Informatik

an der Universität Hamburg eingereichte Dissertation

2020

Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Christopher Maynard
Betreuer:	Prof. Dr. Thomas Ludwig
Zweitbetreuer:	Dr. Julian Kunkel

Hamburg, 18-02-2021

Abstract

Stencil computations are essential for large-scale scientific computing, e.g., in earth system modeling. Such computations are normally time consuming. Execution time is a key concern to consider when developing code. For instance, it would be impractical to run a weather prediction model for one day while predictions should be generated multiple times per day. To minimize execution time, enormous efforts are dedicated to optimize stencil codes to exploit underlying hardware.

Scientific codes are usually developed using general-purpose languages, e.g., Fortran or C++. However, general-purpose languages lack the necessary semantics that allow to exploit some optimization possibilities as a result of the generality of such languages. To overcome this shortcoming, important code transformations that optimize code are done manually. This puts the burden on scientists, who must spend more time on optimization and learn architectural details of computer systems.

In addition to the challenge of understanding architectural details arise related challenges. One of those challenges is the pace of the architectural evolution (which takes place frequently to support HPC applications) in comparison to the life-time of models. This urges to port code to support architectural features that are introduced frequently. Another challenge is the diversity of architectures arising with heterogeneous computing on supercomputers, which complicates the situation even more.

Besides the architectural challenges, the wide range of algorithmic choices at the application level such as numerical methods diversity and grid types, form another factor of complexity for model development. Limitations of some methods and grid types push towards using new grids with different characteristics, e.g., icosahedral grids. Introducing new grids leads to different representations of stencils to apply numerical methods, e.g., triangular tessellations bring new forms of neighborhoods.

To overcome the challenges, this thesis lifts the semantical level of modeling languages to a higher level abstraction. We suggest reforming the software engineering of model development to maximize the use of application semantics to drive optimization by tools: Application requirements are analyzed to identify the grids and the stencils that comprise an application. The spatial relationships among the points forming the different stencils within the application are analyzed. Those spatial relationships are used to define new language extensions in addition to a set of basic extensions that we suggest. Thus, our suggestion is to use an application-adaptable set of language extensions to maximize the use of the application-enabled semantics to enable the optimization process.

In the suggested approach, we allow users to define language extensions and their role in the optimization process. Such details are provided through separate configuration files. This way, we keep the source code clean of optimization and remove the burden

of optimization from scientists, who are now able to write the scientific problem in the source code following an abstraction closer to their scientific concepts. The configuration files are prepared by scientific programmers who master the optimization for some target architecture. This enables a clear separation of concerns in the software engineering of models.

Important points we investigate in this thesis are the way to exploit the mentioned application-adaptable language extensions to drive the optimization process and the scalability over multiple nodes to support modern supercomputers. We also evaluate the impact of the new techniques on the quality of code and on development costs.

The **key contribution of this work** is developing an integrated approach with techniques to maximize the use of semantics in optimizing and scaling stencil computations to support modern supercomputers. This is accompanied with limiting scientists role to coding scientific problems, conforming to principle of separation of concerns, and improved code quality.

The effectiveness of the approach is validated by conducting experiments on various architectures: multi-core processors, GPUs, and vector engines. In order to be versatile, we demonstrate the achievable efficiency of the generated codes and the productivity for the scientists.

Analysis and experimental results show that we can achieve high percentages of the achievable performance on each architecture. We can minimize the number of field loads from memory to caches, and achieve about 80% of memory bandwidth, which is the limiting factor for performance of memory-bound computations. Those experimental results align with the theoretical expectations of achievable performance on the tested architectures. To evaluate performance portability, we use same source code (without any per-architecture changes or special code) on different architecture.

Using the suggested language extensions allows to reduce the code size to one third, and the development costs to less than one half. A key conclusion of this work is that the application-adaptable language extensions maximize code optimization through application-specific semantics while they can be tailored to the needs of specific applications or domains.

Acknowledgements

I would like in first place to express my gratitude and appreciation to my advisers: Prof. Thomas Ludwig and Dr. Julian Kunkel, for their great support, which made this work possible. Activities behind this work were very well coordinated among them and me. The meetings with whom have always lead to great ideas. I am glad for the great and flexible leadership of Prof. Ludwig, and to the extensive discussions with Dr. Kunkel, who offered the deep scientific and technical experiences, and emphasized the high commitment to the scientific and research ethics and quality, and widened the opportunities for collaborations and access to resources.

I am thankful to the support from the DFG (German Research Foundation), which supported the AIMES project I was employed in through the SPPEXA program (German Priority Programme 1648 Software for Exascale Computing). Thanks to the people behind the SPPEXA prgram and the colleagues in the AIMES project for all the feedback and discussions that enriched the work done in this thesis. My thanks also go to the members of the scientific computing group of Prof. Thomas Ludwig, who have always been supportive colleagues over the course of my work in the project.

During my research I needed to get access and run experiments on different systems. I would like to thank the German Climate Computing Center (DKRZ) lead by Prof. Thomas Ludwig, the Swiss National Supercomputing Center (CSCS), where I used their Piz Daint machine for GPU experiments, Erlangen regional computing center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), especially Prof. Gerhard Wellein and his group, where I used their tools to instrument code on Broadwell processors, NEC Deutschland, especially Dr. Erich Focht and his colleagues, who allowed to get access to a machine with NEC Aurora vector engines. Thanks to NVIDIA for the support through the PSG cluster.

I would like also to thank Prof. John Thuburn, University of Exeter, who helped to develop the shallow water equation solver.

Thanks to everybody who gave any hint or feedback, who discussed thoughts, and to everybody who gave this work any value.

Contents

1. Introduction

This section introduces the topic and motivates the work. We start with an overview of numerical simulations in Section 1.1 and stencil computations in Section 1.2. Architectural considerations are discussed briefly in Section 1.3. Problems and challenges conventional code development approaches face are illustrated in Section 1.4. Finally, we state the main questions that this thesis is entitled to provide answers for in Section 1.5.

1.1. Numerical Simulations

Due to applicability limitations of analytical solutions of many scientific problems, e.g., differential equations, numerical methods represent a viable alternative. Numerical solutions yield results with acceptable tolerances that allow them to replace exact mathematical solutions. Using computer machines to implement numerical solutions expanded gradually among many scientific fields. Therefore, many scientific advances depend on the advances of computing technologies.

Normally, numerical solutions comprise huge amounts of data and mathematical operations, e.g., the finite element method [Hre41, C+43] apply formulae at many points that discretize a problem domain. Executing many operations on huge amounts of data within time limitations classifies those solutions as high-performance computing (HPC) problems. Such problems need to give a special focus to the optimal use of hardware resources.

The optimal use of the hardware where a code runs is critical for the execution time of the code, and to the costs of running it. This is especially important for lengthy simulations, where the simulations could take hours or even days. Results of such simulations should be delivered within restricted time frame to be useful. Under such constraints, optimal use of resources is a key aspect in modeling.

Stencils are an important part of many of the numerical methods that are used in different scientific fields. Despite the mathematical differences among the different numerical methods, applying stencils represents a common feature of those methods.

1.2. Stencil Computations

Stencil computations [STDH03] represent an important family of scientific computations. A stencil computation computes the value of a field at one point in space using values of one or more fields at neighboring points. A stencil is defined by a set of neighborhood

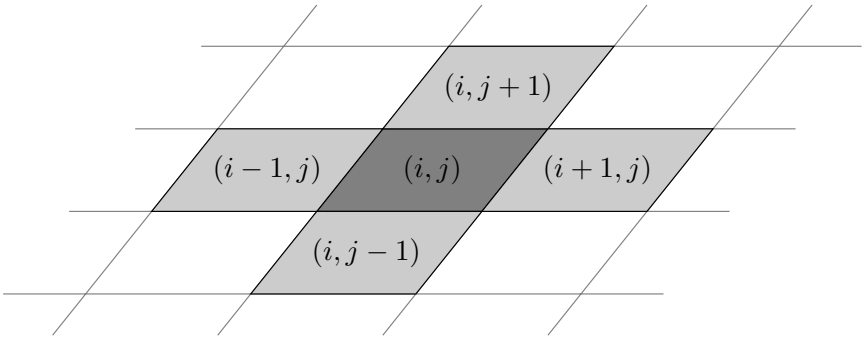


Figure 1.1.: A typical stencil on a rectangular 2D grid

relationships. An example of a simple stencil is given in Figure 1.1. The value of one field at the point marked with (i, j) is computed based on values at the points $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. Within a stencil computation, the same expressions and spatial relationships that form the stencil are applied repeatedly to compute field values at a specific set of points. So, the stencil that computes the value at (i, j) in Figure 1.1 is slid right to compute the value at $(i+1, j)$ based on its four neighbors, and so on with other points of the grid.

Stencil computations are important for many scientific fields, e.g., partial differential equation (PDE) solvers, computational fluid dynamics, image processing. Earth system modeling (ESM) [DMH+17] is one of the scientific domains where stencil computations are used extensively. ESM includes many scientific disciplines, e.g., physics, chemical processes, biological processes, numerical methods. The extensive use of stencils within the numerical methods in ESM makes it a good fit to our research regarding stencil computations. Hence, we take one discipline of ESM, i.e., stencil computations, which is the subject of this work, to investigate our approach and seek answers for the questions of this thesis.

Many different stencils (may be hundreds or thousands) are used to construct a simulation in an earth system model. Stencils of different shapes are used in a model and the various components within a model. Some stencils comprise points over vertical neighborhoods, e.g., physics computations, others comprise points over horizontal neighborhoods, e.g., dynamical cores. Furthermore, tessellation [GS87] (tiling a surface into non-overlapping tiles without gaps) techniques generate different neighborhoods, e.g., triangular, rectangular, hexagonal. Thus, different tessellations lead to different stencils. Besides, the localization of the fields with respect to the tessellation shapes, e.g., staggered grids [AL77], leads to additional stencil shapes. In general, grids [STDH03] represent the basis, over which stencils are defined.

Grids: An important aspect of stencil computations is the spatial relationships. Stencil computations provide solutions over a defined problem space. A problem space is discretized by a finite set of points, forming a grid.

Modern earth system modeling software tend to use higher-resolution grids (more points to discretize the problem space). Higher-resolution grids allow more accurate results. However, they demand more computational resources. To balance the resulting accuracy and the corresponding computational load, grid resolutions are decided based on the capabilities of the available technology and machines. This is essential to run simulations within realistic times.

Advances achieved in computing technologies are a determinant factor to the advances in modeling and hence for scientific progress for some sciences, e.g., earth system modeling. The need for performance represents an increasingly demanding driver for the hardware technologies to provide the necessary architectural improvements to support the needs of the scientific software.

1.3. Computer Evolution and Architecture Diversity

To handle the demanding HPC workloads and the arising computing problems, processor architectures are continuously evolving. Many ideas and techniques were developed to push the capabilities of the hardware. Addition of more cores, running mathematical operations over multiple data elements at once, and memory reorganization in hierarchies are some of the broad classes of architectural improvements. Hardware advances lead to fundamental classes of architectures: multi-core processors, graphics processing units (GPU) [Hwu11], vector engines, field-programmable gate arrays (FPGA) [BFRV12]. Different techniques were developed to allow those architectures to improve the amount of data processed per time unit, and the data movement between memories and processing units.

Multi-core processors: The rise of multi-core processors represents an evolution on the CPU (Central Processing Unit) technologies. Early processors reached performance improvements limits as a result of physical limitations. Frequency scaling could not be a solution any further. The next step was to evolve the processor technology in the number of cores comprising a processor. Multi-core processors [Rou13, GK06] include multiple processing units on a single chip, on which multiple threads of code can be executed concurrently.

The operating systems schedules running processes on the cores of a processor. The technologies introduced with multi-core processors are not specialized to HPC applications. Therefore, benefits are introduced to computing in general, where OS schedulers can schedule multiple processes concurrently.

HPC application benefit from multi-core processors as applications can be scaled over the cores of the processor. So, the workload of an HPC application can be divided among the cores, leading to reduced execution time. This way, counting for multiple cores while developing applications [HKM08] is an important point to consider to scale code through cores.

Many-core processors: The addition of multiple cores on a single chip inspired introducing new architectural concepts through the addition of more cores leading to many-core architectures [HBK06, JR13]. Many-core processors contain high numbers of cores ranging from dozens (e.g., Knights Landing [SGC+16]), to thousands (NVIDIA's Pascal GPUs [FD17]) of cores, which are much more in comparison to the cores on multi-core processors, which contain dozens of cores (Broadwell [NKD+15]). Higher throughput balances the lower performance per core and the latency on many-core architectures. Such features fit HPC applications, which comprise highly-parallel computations.

Many-core processors provide poor performance for non-parallelized codes, Therefore, parallelization of code to exploit the many-core processor [HKM08] capabilities needs effort to optimally find how to parallelize code over the processor resources. However, optimally-parallelized codes can achieve high performance improvements.

GPUs: GPUs are considered many-core processors. They comprise high number of cores, e.g., the P100 GPU comprises 3584 CUDA cores. GPUs contain streaming multiprocessors (SM) for highly-parallel computations.

SMs contain cores, which contain execution threads that execute the operations. Threads are organized in warps, which are groups of threads executing same operations concurrently. Efficient coding a computation to exploit the processing elements and maximize the benefit of the high memory bandwidth of a GPU needs special expertise.

Memory hierarchies: The different architectures use different configurations of memory hierarchies. Differences in caching levels and cache sizes need special care in order to maximize the use of the caches and reduce access to memory. This brings the challenge to tune code when targeting a specific processor.

1.4. Challenges with Conventional Modeling

Hardware solutions include improvements to existing architectures or even moving to new different architectures, e.g. GPUs, vector engines and FPGAs. To exploit the computational power of new hardware features and architectural improvements, software should be accordingly modified. Thus, source code must be rewritten to be aware of the underlying hardware features to use the hardware efficiently as illustrated in Figure 1.2. This figure illustrates the conventional code development with general-purpose languages (GPL). The stencils and the whole computation are written according to the rules of the modeling GPL.

1.4.1. Performance and Performance Portability Challenges

The semantical nature of GPLs lacks the necessary information to exploit some optimization opportunities. That is inherent in the design of those languages because they are intended to provide a way to generally solve programming problems. This shortcoming makes compilers generate codes with sub-optimal performance.

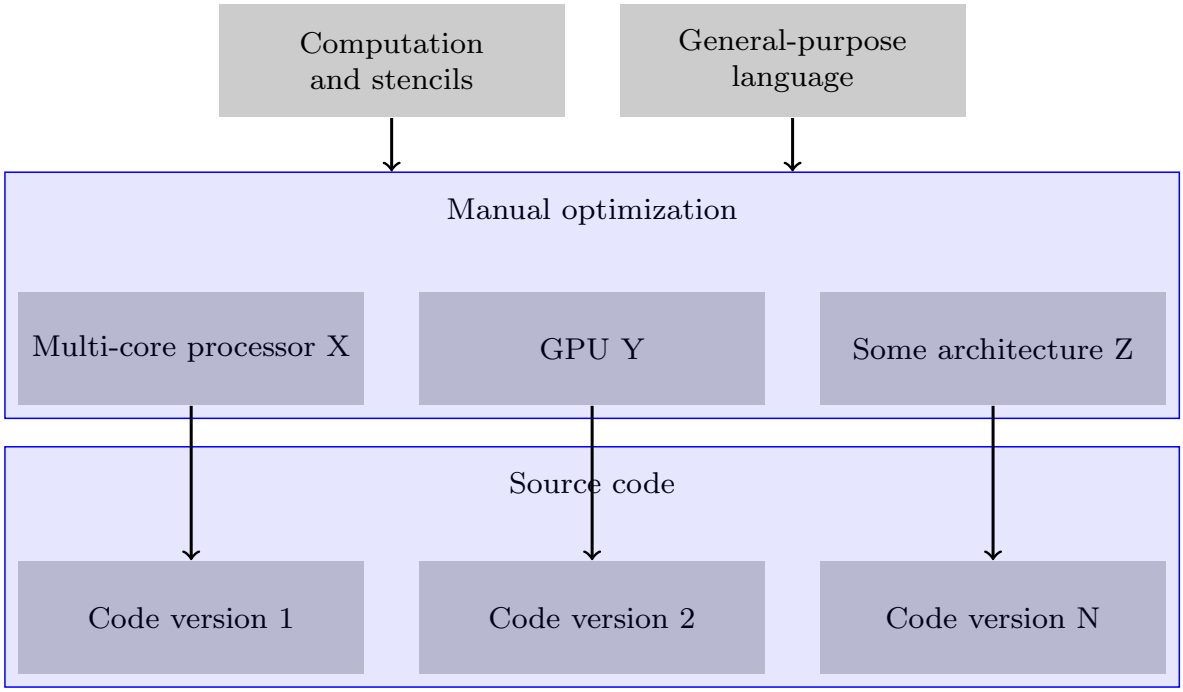


Figure 1.2.: A diagram showing conventional modeling with general-purpose languages

As shown in Figure 1.2, hardware features of underlying architecture should be considered in order to write optimized code. Therefore, optimization decisions must be made by developers (typically these are scientists who are expected to focus on scientific problem), and applied to the code manually.

Performance portability: There are multiple definitions of performance portability in the literature. One simple definition is "the ability of an application to achieve a similar high fraction of peak performance across target devices"[MSBCP14]. So, the performance portability of some code indicates its capability to use the underlying hardware efficiently on multiple target machines.

In fact, the move to the exascale computing era, the time when computers will be able to yield EFLOPS (10^{18} floating point operations/second), includes the use of heterogeneous systems. This means that software will need to make use of different architectures to exploit the available resources of a machine.

Writing and maintaining a software application that will run on different machines with different hardware resources is a tough task. Some part of the source code could target the features of the computational resources when targeting a particular machine. But also, the same part of the code could run on a different architecture when the same software is run on a different machine. Thus, optimization of source code to a specific machine or optimizing a specific part to a specific architecture allows the software to run efficiently on that machine, while losing performance when that code is run on a different machine [PMS17].

In fact, manual optimization of source code harms the performance portability of software. But also, it is necessary to optimize the source code to run the software efficiently.

1.4.2. Code Quality Challenges

As mentioned, the lower level semantics of general-purpose languages obligate many optimization decisions to be provided by software developers within source code. To develop an optimized code while constrained by the lower-level semantics of a modeling language, lower-level details will be included in that code.

Productivity and programmability: Scientists handle the optimization overhead when GPLs are used for modeling. Optimization needs detailed knowledge from the software developers about the target architecture, where the software will be executed. Also the expertise in the software development technologies and programming models that are needed to exploit the hardware features is essential for the developers. Thus, model development consumes longer time to look for hardware details and optimization techniques, and to find optimal code structure and apply transformations. This reduces the productivity of scientists and complicates their programming role.

Maintainability and readability: In addition to the challenges of optimizing code for a specific architecture, software would normally be run on different machines. This means that, the lower-level details in an optimized source code should be redundantly rewritten in the code to run efficiently on different machines. This is illustrated in Figure 1.2, where multiple code versions are written to support multiple architectures.

The discussed redundancy makes the maintainability of the code repository complex. Any later code improvement, debugging, or modification would be a tough mission. The readability of a redundant lower-level code is consequently harmed.

1.5. Contributions of this Work

In this thesis, we provide a solution to enable a stencil computation development methodology that withstands performance and performance portability challenges, and allows improved code quality. We investigate the extensibility of modeling languages by the user. The approach allows the user to tailor the modeling language to the needs of an application (see Figure 1.3). As the user defines the language extensions, s/he also specifies how the use of an extension affects the processing of the source code such as optimization and code generation. Thus, the user extends the language and configures the processing procedures of the added extensions. In this context, we study the transfer of optimization decisions from the source code to a user-controlled code transformation process through a set of semantically higher-level user-defined language extensions.

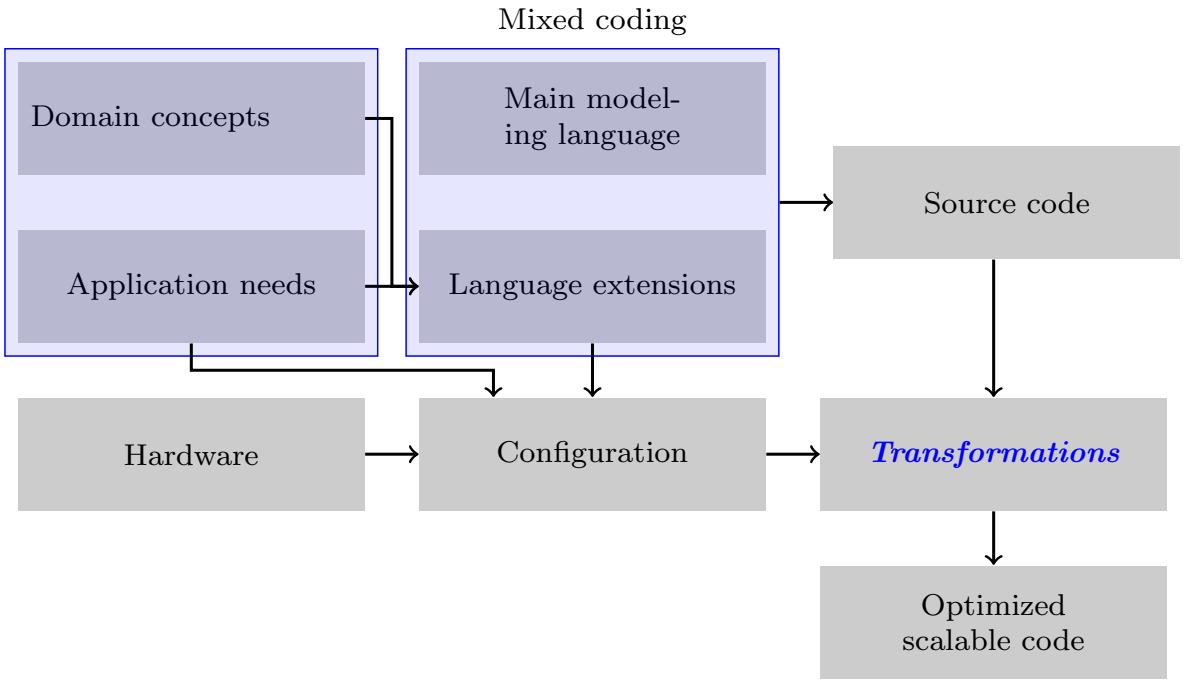


Figure 1.3.: An abstract diagram showing high-level concepts

Concrete questions: The questions that we answer in this thesis, which lead to understand how to achieve the high-level objectives of withstanding the mentioned challenges, and which make this work unique are:

Questions to address performance and performance portability challenges:

- How could user-defined application-adaptable language extensions (when mixed with a general-purpose language) convey the necessary information to drive the stencil code optimization process in a user-controlled code processing procedure?
- How could the semantics drive the scaling of code over multiple nodes?

Question to address code quality challenges:

- How does that affect the quality of the code?

Further contributions: Within the context of answering the questions, we develop an ***integrated approach for the software engineering of stencil computations***. We develop a modeling language solution and the means to translate that language. The solution that we develop serves mainly the quality of the code and its performance portability. In our approach, source code is separated from the optimization process, and the role of scientists is focused on the scientific activities. This role specialization makes our approach conforming to the principle of ***separation of concerns***. The optimization process in this context is applied using separate (configuration) files written by scientific programmers, who perform the other role besides the scientists role.

Software engineering reformation: The conventional software development process using GPLs as shown in Figure 1.2 depends on:

- Scientific problem (computation logic including stencils)
- Modeling language
- Hardware features and corresponding software tools and programming models

The scientific problem is coded conforming to modeling language rules/grammar and considering underlying hardware. Finding the optimal code to exploit hardware capabilities is necessary to write code. The process is reviewed or repeated (at least partially) when new hardware features or new architectures are introduced.

In comparison, the suggested software development process as shown in Figure 1.3 is reformed as follows:

- Scientific concepts are extracted from the application and the domain science
- Language extensions are formulated to extend the grammar of the modeling GPL
- Source code is written by scientists conforming to the extended grammar (GPL + extensions)
- Scientific programmers prepare configuration files based on hardware, once per targeted machine
- Source code is translated with a tool that transforms the code according to configurations

With this way, hardware changes are addressed with preparing configuration files corresponding to new features. Source code does not need to be modified. Furthermore, scientists don't have to care for the architecture-specific optimization, rather it is done as an added value in a value chain, where code is developed in one step, and optimization is applied to that product (source code) in a second step, which is driven by configuration files described separately from source code.

1.6. Structure of this Text

This thesis is structured as follows: In Chapter 2, background information is provided regarding concepts related to the work done in this thesis. Then, in Chapter 3, related work and the state of art of the research topic are discussed. In Chapter 4, we present our methodology to answer the questions of the thesis. To follow the described methodology we start with describing the development of the language extensions in Chapter 5. Applications developed with the developed language extensions are described in Chapter 6. In Chapter 7, the translation process is discussed, where we present and discuss the main contents regarding the translation techniques. Experimental results to validate the techniques we developed and our answers are presented in Chapter 8. Chapter 9 concludes this thesis.

2. Background

In this chapter, we review important concepts concerning background topics that are essential to understand the text. Readers who are familiar with these concepts can skip individual sections. First, earth system modeling for which we apply this work to is introduced in Section 2.1. Next, more about grids with different structure and arising challenges that scientists face when developing modern models are presented. Then, a short technical review of architectures that we use mostly in this work is provided in Section 2.3. Next, a set of optimization techniques that serve stencil computations is introduced in Section 2.4. Finally in Section 2.5, a short overview regarding domain-specific languages is provided.

2.1. Application Domain

The main track of this work is the study of the extensibility of a general-purpose language. An important aspect we investigate is the configurability of the code transformation using the semantics of the language extensions. The study is done in the field of earth system modeling, where stencils are extensively applied with a wide range of requirements.

As part of this section, the encoding of a typical simulation problem, the Shallow Water Equations is shown. Based on this example, the challenges for software development in the domain are discussed. Using the example problem we demonstrate concepts of model equations, problem domain, stencils, and kernels.

2.1.1. Earth System Modeling

Earth system modeling (ESM) [DMH+17] includes different families of models, e.g., climate modeling, which predicts long time-spans (decades), and numerical weather prediction (NWP), which predicts short time-spans (hours and days). Earth system models can be global, i.e, simulate the physical behavior of the whole earth, or local, i.e, cover specific local areas. Furthermore, recently there have been hybrid models that support global modeling with higher-resolution focus for specific regions [ZRRB15].

Processes: A combination of physical processes and their dynamics and their inter-dependencies and interactions are taken into account in modeling. Such processes include models for atmosphere, land, ocean and ice. Besides to the physical processes, the biological and the chemical processes also may be included in some models Figure 2.1. The dynamics which are included in a model are described by a set of equations – mainly PDEs.

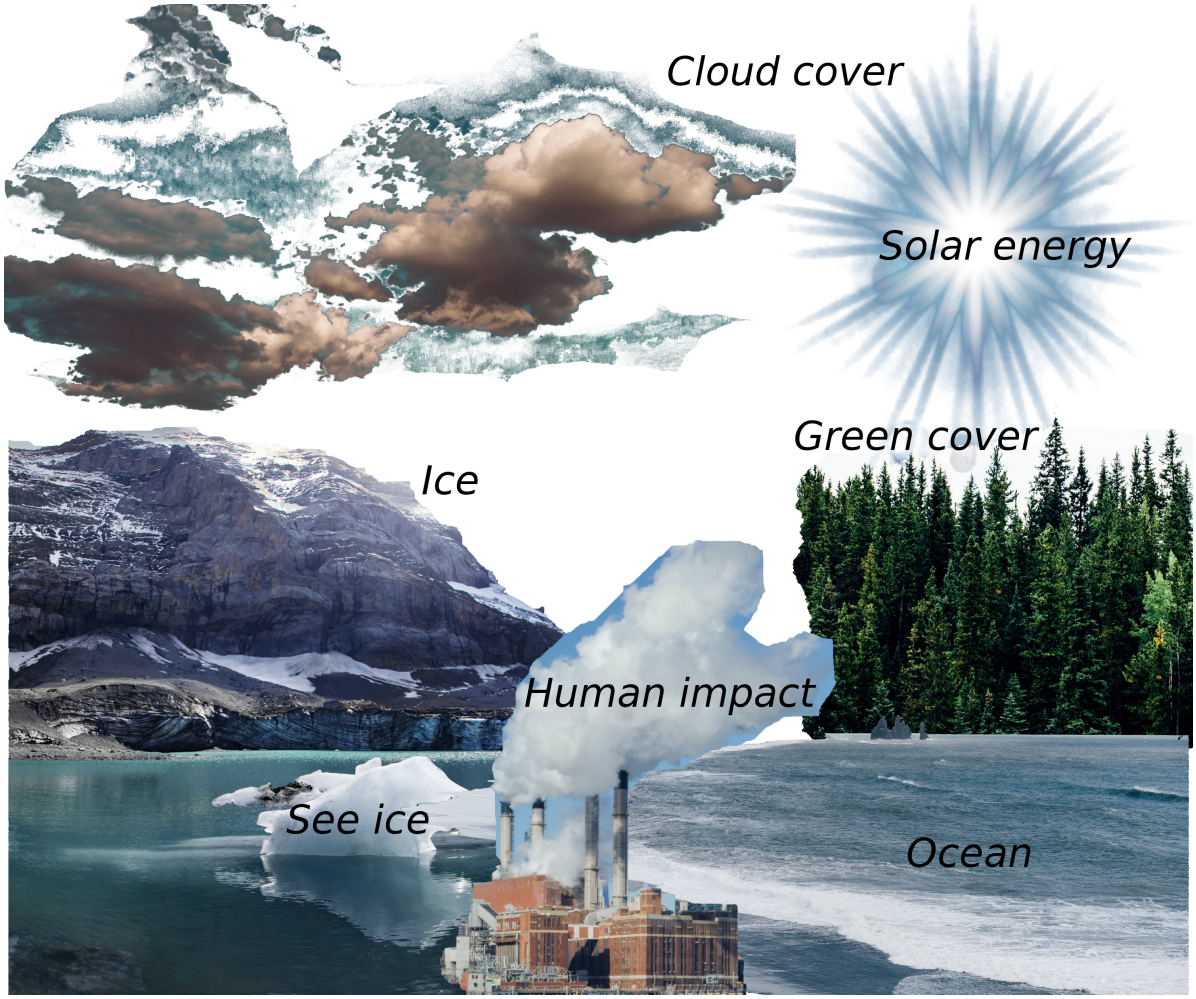


Figure 2.1.: Processes in earth system modeling

Models when run produce predictions of the state of the earth system over specific time periods. For example, a numerical weather prediction model produces predictions for the weather within the next days. The earth system models that include the biological and the chemical processes can be used to study the climate change and the impact of the human behavior on the system [Fla11]. The inclusion of more dynamics and processes allows for better models with more precise predictions. However, that increases the demand for computing power and more complex modeling. The demand for computational resources and the needed resolutions to include some dynamics within models are discussed further in [NDA+19]. As shown in [NDA+19], some dynamics, e.g, deep convection and surface drag, would need grid resolutions of 1 km and computational capabilities in the range of EFLOPS.

An example problem: *To demonstrate the concepts, let's take a compact shallow water equations (SWE) solver that was presented by [LO07]. The dynamics within the*

system that the model simulates are described with the equations

$$\frac{Dh}{Dt} = -h\nabla.\overrightarrow{u} \tag{2.1}$$

and

$$\frac{D\overrightarrow{u}}{Dt} = -g\nabla.h \tag{2.2}$$

which represent the conservation of mass and conservation of momentum respectively. The scalar field h is the water level, the vector field $\overrightarrow{u}$ is the velocity, and g is the gravity constant. The solvers purpose is to simulate the system behavior which is governed by the PDEs Equation (2.1) and Equation (2.2) to predict the state of the system over time.

2.1.2. Turning Equations into Computations

Earth system models simulate the dynamics of the earth system over a specific period of time. The system's state is defined by a set of variables each of which represents some quantity. Variables represent fields defined over the two-dimensional surface or three-dimensional space. Values of fields could be scalars or vectors.

The equations governing the dynamics of the system are formulated into algorithms to enable the simulation. This process depends on the numerical methods that the model uses to solve the equations. Simulations proceed in multiple time steps.

A simulation starts at an initial state of the system, where values of some fields are read from storage. Depending on the numerical methods used in a model, a set of modeling fields are updated during each time step.

Let's refer again to the SWE solver presented in [LO07]. So far, we have seen the equations (Equation (2.1) and Equation (2.2)) that govern the system. However, an algorithmic solution to solve the equations should be developed to simulate the system. Listing 2.1 shows the code that turns the equations into a simulation. The code is shown in this snippet, the details of kernels and stencils are discussed in the following text to demonstrate the concepts with this example.

Listing 2.1: **Part of simulation code to solve SWE as described in [LO07]**

```
1  const int DIV = 3;
2  const double SMOOTHING = 1.05;
3  const double HALFPKT = 0.5*(SMOOTHING/DIV);
4  const double LFTWALL = (1+HALFPKT)*DX+1e-12;
5  const double RGTWALL = N*DX - LFTWALL;
6
7  memset(newVels, 0, N*sizeof(double));
8  memset(newRho, 0, N*sizeof(double));
9  for(int i = 1; i < N-1; ++i)
10   for(int j = 0; j < DIV; ++j)
11   {
12     x = (i + (j+0.5)/DIV) * DX;
13     gx = x/DX - 0.5;
```

```
14      idx = int(gx);
15      fx = gx - idx;
16      u = (1-fx)*vels[idx] + fx*vels[idx+1];
17      nx = Clamp(x+DT*u, LFTWALL, RGTWALL);
18      left = nx/DX - HALFPKT;
19      right = nx/DX + HALFPKT;
20      li = int(left);
21      ri = int(right);
22      f1 = (ri - left) / SMOOTHING;
23      f2 = (right - ri) / SMOOTHING;
24      newRho[li] += rho[i] * f1;
25      newRho[ri] += rho[i] * f2;
26      newVels[li] += vels[i] * f1;
27      newVels[ri] += vels[i] * f2;
28    }
29  memcpy(vels, newVels, N*sizeof(double));
30  memcpy(rho, newRho, N*sizeof(double));
31
32  double s = CSQ * DT / DX * 0.5;
33  vels[1] -= (rho[2] - rho[1]) * s;
34  for(int i = 2; i < N-2; ++i)
35    vels[i] -= (rho[i+1] - rho[i-1]) * s;
36  vels[N-2] -= (rho[N-2] - rho[N-3]) * s;
```

Problem domain: To enable numerical computation of mathematical operators, the problem domain is discretized and the fields are defined over a finite set of points in that surface or space (the problem domain). The bigger the set of points discretizing a problem domain, the higher resolution has the simulation, which could be necessary for more precise predictions. On the other hand, the bigger this set is, the more time consuming is the simulation not only because there are more points to compute but because the time-step must be reduced to remain numerically stable. Thus, the compromise is to optimally use the existing hardware of a machine to run the highest possible resolution within an acceptable time.

[NDA+19] illustrates the needed computer resources to run higher resolution simulations. The paper shows the length of the simulated time (often called model time) when running the ICON model for one day over different numbers of compute nodes and different grid resolutions. The numbers in the paper show that the higher-resolution grids need using additional nodes or/and reducing simulated time.

To demonstrate the concept through the discussed SWE solver example, let's look again at Listing 2.1. The solver solves a one-dimensional version of the problem. The domain of the problem is discretized into N points. Therefore, the loops

```
3:  for(int i = 1; i < N-1; ++i)
```

and

```
27:  for(int i = 2; i < N-2; ++i)
```

traverse (part of) the set of points that discretize the problem domain. In fact, the outermost points of this single dimension space represent the boundaries of the problem domain.

Stencils: The nature of the numerical methods that are used in earth system modeling makes field values depend on spatially local values across time steps. Thus, an updated value of a variable at a specific point in space in one time step is computed based on a set of values of some variables (potentially including the variable itself) around that same point in space in the previous time step. Those patterns of computations define stencils.

To demonstrate the concept through the example solver shown in Listing 2.1, the statement

```
28:   vels[i] -= (rho[i+1] - rho[i-1]) * s;
```

represents a simple stencil. This stencil is a one-dimensional stencil updating the field vels at a point according to values of the field rho at the direct neighboring points, the points to the left and and to the right of the updated point.

Computational kernels: The mathematical operators that are computed in a model during a simulation are algorithms formulated as stencil operations applied over the problem's spatial domain. Those computations are coded within compute kernels. Those compute kernels are the most time-consuming parts of models. Therefore, they need a special care for optimization.

To demonstrate the concept of computational kernels through the example solver shown in Listing 2.1, the statement

```
27: for(int i = 2; i < N-2; ++i)
28:   vels[i] -= (rho[i+1] - rho[i-1]) * s;
```

represents a simple computational kernel applying a simple stencil. The kernel traverses points in a single dimension applying the stencil at each of the traversed set of points. In this kernel, the stencils are not applied at first two points as the first point has no left neighbor, and the second is computed with a special expression to count for the boundary conditions, i.e.,

```
26: vels[1] -= (rho[2] - rho[1]) * s;
```

Same is true for the last two points (the rightmost).

2.1.3. Space Discretization

The way a model space is discretized in a finite set of points is an important decision for the development of an earth system model. The shapes of the cells resulting from the tessellation process depend on the tessellation method.

Regular grids: The discretization of each dimension in the two- or three-dimensional Euclidean space have been one simple way to do discretize space. Such discretization method generates a simple rectangular tessellation of space forming a regular grid.

A direct mapping of the spatial coordinates of each point in the discretized space/plane to the array notation that is used in multi-dimensional arrays in most general-purpose languages, e.g, C/C++ and Fortran, simplifies addressing the variables data in memory. This makes such method viable and highly acceptable to be used in many models. However, some drawbacks of using this kind of space discretization pushed for the search for more complex discretization methods. For example, the difference between the areas covered by the rectangles near the pole and those near the equator and the rectangles in between obstructs the simulations in global models.

Icosahedral grids: One of the methods that have been adopted by recent models is to use an icosahedral grid. In an icosahedral grid, the spherical surface of the earth is mapped to an icosahedron. The icosahedron is a polyhedron with twenty triangles each of which is equal in area and shape. In fact, to map the sphere to the icosahedron, the triangles are actually spherical triangles.

To get higher resolution, the edges of the triangles are divided into a number of equal-length parts. This divides the triangles in smaller equal-area triangles. The grid resolution is increased by recursively dividing the edges of the triangles into two equal parts, where each triangle is divided into four smaller equal-area triangles (see Figure 2.2).

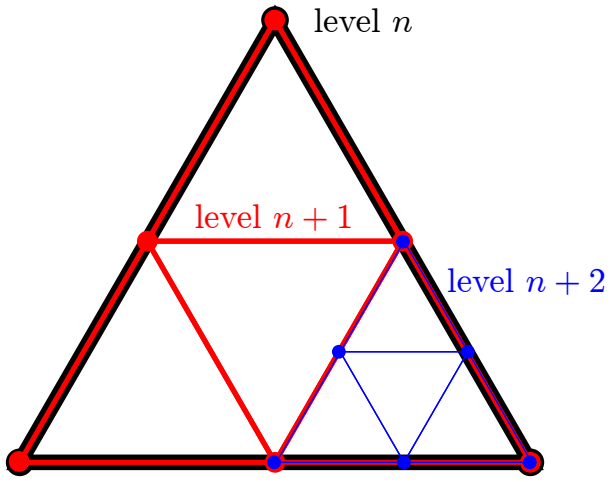


Figure 2.2.: The recursive division of the icosahedral grid of level n (one black triangle) into level n+1 (four red triangles) and finally level n+2 (illustrated for the rightmost red triangle)

The grid resolution refinement process allows this space discretization method to bypass the grid nesting drawback of the regular grid. A specific set of triangles that covers a specific local region on the earth surface can be further refined for obtaining a

higher resolution and hence, achieve a better accuracy. In Figure 2.2, for example the bottom right red cell is refined to *level n+2*. This way, the earth system models that use this space discretization method can support local modeling besides to global modeling.

In an icosahedral grid, the basic shape of the cell is the triangle. However, it is possible to synthesize hexagonal cells based on the triangular basic cells that result from the recursive refinement of the grid (see Figure 2.3 and Figure 2.4).

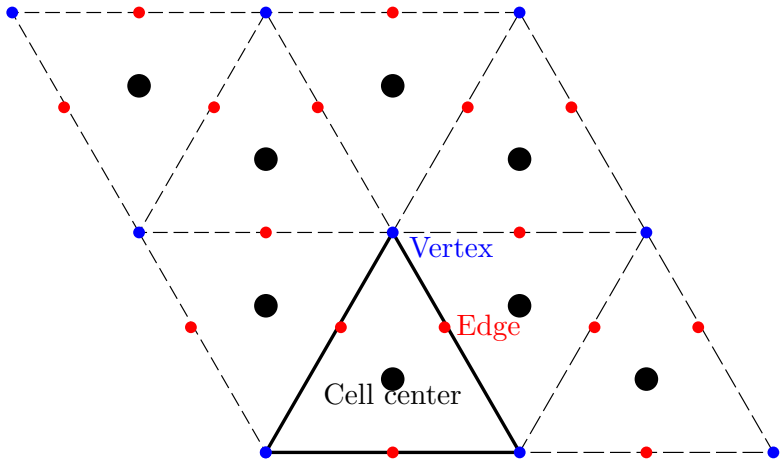


Figure 2.3.: Triangular icosahedral grid and field localization

Regardless of shape, the cells cover their own area and are logically separated by edges from neighboring cells. The vertices of those edges are also important in the structure of the grid in some solutions.

Field localization – collocated vs. staggered grids: The mathematical characteristics of the fields that are necessary to execute a computation with a numerical solution guide the model developers to discretize those fields over specific sets of grid points. For example, a field can be discretized over the set of points at the centers of the grid cells. Another field could be discretized on the edges of the grid' cells or at their vertices.

Decisions can differ between applications to use staggered or collocated grids (see Figure 2.3 and Figure 2.4). This can be affected by the numerical solution methods that applications use.

2.1.4. Data Structures

Handling the data of a field that is discretized over an icosahedral grid brings new challenges. That is true for the memory layout when accessing the data, and for storage and I/O in general. Although the space discretization method that is used in a model constrains the structure of the fields data, the memory layout is an important decision that heavily affects the performance of the model.

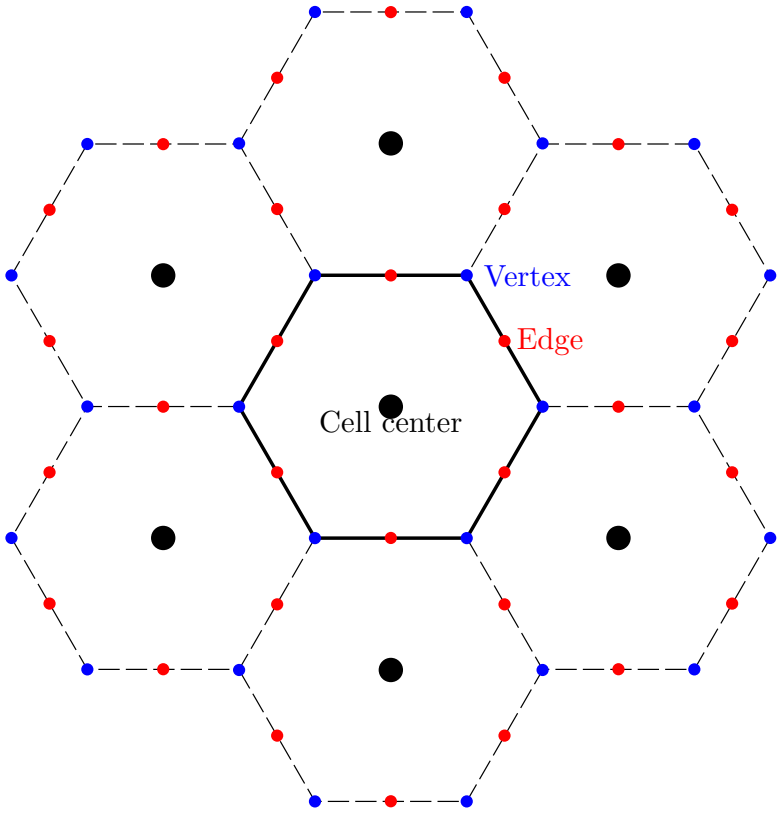


Figure 2.4.: Hexagonal icosahedral grid and field localization

Data locality in stencils: The locality of the data in the stencil operations is one factor that affects the memory layout decisions. This factor is inherent from the nature of the computations that the models execute, whatever hardware architecture is used to run the model. This inherent locality provides an opportunity to improve the performance of the model by using the memory bandwidth in a better way if the suitable memory layout of the fields was chosen.

Data layout optimization: The nature of the memory access in an architecture is a machine-dependent factor when deciding about the fields' memory layout. Stencil operations are memory-bound computations and the use of the memory bandwidth is very sensitive to the whole performance of the compute kernels. Therefore, the optimal memory layout of the fields' data is key to achieve good compute kernel performance. Many techniques can be used with the different space discretization methods to exploit the data locality. The use of the filling curves ([But68]) which count for locality, e.g, the Hilbert filling curve, generally improves the use of the memory bandwidth and can be also used by the models which use an unstructured grid.

2.1.5. Complexities Arising with Unstructured Grid Addressing

Along with the improvements that the use of the icosahedral grid brings it adds complexity to the code development. The memory layout and the addressing of field data in memory is one issue.

Grid structure and connectivity: In the contrary to the regular grids, the direct mapping of the Euclidean coordinates of the grid points to the multi-dimensional arrays does not fit generally to the unstructured grids. Thus, further grid-describing meta-data are needed to access fields data. Such meta-data may be either computed according to some formulae, or may be pre-computed and stored somehow –e.g, lookup tables– to allow access to a field's value at some point in space when needed later.

Filling curves: A space-filling curve [But68] may be used for the transformation of the two-dimensional surface into one-dimension to store the field values at a plane in a simple array. For the three-dimensional space in this example, a multi-dimensional array stores multiple planes where one array dimension represents the plane number (vertical), and another represents field values at the plane (horizontal).

Neighbor access: Meta-data helps to solve another issue that arises with the unstructured grids, which is the spatial relationships between the grid points. For example, the point to the left/right of a point in the regular grid can be addressed with the subtraction/addition of one to the corresponding horizontal dimension index in the multi-dimensional array notation. However, this is not the case for the the unstructured grids. In fact further information about the grid is then needed to access spatially-related grid points. This meta-data can be computed according to some formula, or can be pre-computed and stored in lookup tables to be ready for use later.

2.1.6. Common Concepts Across Grid Types

Whatever kind of grid a model uses or whatever differences exist in the grid structure details, the higher-level ideas are common and serve similar objectives. This commonality on the higher-level and the variability of implementation details for the same scientific concepts at some level simplifies and provides an opportunity to define a set of unified/common semantically higher-level constructs. Hiding the calculations that are necessary to address the data or to find the spatial relationships, or the corresponding pre-computed stored meta-data is possible by using a higher-level abstract interface. The same abstract interfaces, in the same way, hide the lower-level details for the regular grids likewise.

2.2. Architectural Aspects and Features

Processor architectures are continuously evolved to support computational workloads needed by HPC applications. Addition of more cores, running mathematical operations over multiple data elements at once, and memory reorganization in hierarchies are some of the broad classes of architectural improvements.

Core-level scaling: Parallel execution of application code is an important feature within the different architectures. Parallelization is provided in multiple levels. The coarse-grained level of parallelism includes adding more cores on a single chip.

Multi-core processors [GK06] provide more cores each able to run an independent computation at the same time. Operating systems can schedule codes of multiple applications to run on the different cores. Significantly more cores are provided by many-core architectures [HBK06] to execute larger computations like lengthy simulations.

Operations on multiple data elements: Compared to core-level parallelism, a finer-grained level of parallelism is provided by executing an arithmetic operation on multiple data elements at the same time. Single instruction multiple data (SIMD) [CTZ00] operations in multi-core processors and some many-core architectures, e.g, Xeon Phi [SGC+16], execute an arithmetic operation on a vector of elements. Vectors up to 8 double precision or 16 single precision elements are typical in today's processors. Vector engines, e.g, NEC's Aurora [KMI+18], executes operations with longer vectors (vector registers with 256 elements width, and FMA pipes allowing executing 32 double precision arithmetic operations per cycle). The concept of a single operation on multiple data elements is also used, but in a different way that is different from vectorization, on GPUs, were threads are grouped into warps.

Memory hierarchies: Peak performance of modern architectures allows to execute much more mathematical operations than the amount of the data that the memory can provide. The memory bandwidth bottleneck leads to reorganize the memory into a hierarchical structure [Prz90]. The different architectures implemented different caching systems to minimize the time to access data. Multi-core processors [LLK+17, SMN+19] use caches for data and code. Per-core and shared caches between the cores are also used. GPUs [Mit14, SMN+19] use L1/texture caches that enables the coalesced memory accesses, besides to caches that are shared between the streaming multiprocessors of the GPU.

2.3. Selected Processing Architectures

In this section we provide some technical description of three processors belonging to three different architectures: multi-core processors, GPUs, and vector engines.

Intel Broadwell multi-core processor: Multi-core processors are used even when accelerators are there (as host processors). Generally, computations are executed on those CPUs. Broadwell processors are currently used on many supercomputers, therefore, we discuss some technical details of Broadwell to illustrate architectural concepts of multi-core processors.

Broadwell [NKD+15] is a multi-core processor micro-architecture. Broadwell processors serve as CPUs to run computations generally, including running applications in configuration where nodes include accelerators, e.g, GPUs. Different Broadwell processors consist of 18 to 24 cores (36-48 threads). Along with the cores are 45 MB to 60 MB L3 caches. Data rates of Broadwell are 2133 to 2400 DDR.

Caches are organized in three levels, L1, L2, and L3. Specific processors include a fourth level. L1 caches are separated in a 32 KB 8-way set associative instruction cache, 64 Byte cache line, and a 32 KB 8-way set associative data cache, 64 Byte cache line per core. L2 caches are 256 KB 8-way set associative, 64 Byte cache line per core. Broadwell contains shared L3 caches (1.5 to 3 MB per core) 64 Byte cache line, 16-20 -way set associative. All caches work with write-back policy.

Vector units of Broadwell support AVX2 instruction set extensions, which apply SIMD operations on vectors of length 256 bits. FMA3 (fused multiply–add) operations are supported.

NVIDIA P100 GPU: GPUs got in the last few years growing attention in the field of HPC as a result of high memory bandwidth and support for highly-parallel computations. We discuss some technical details of Tesla P100 GPUs to illustrate architectural aspects of GPUs.

The Tesla P100 GPU [FD17] is an accelerator built with NVIDIA Pascal architecture, and supports GPGPU computations. It consists of 3584 cores. Supported floating point operations are double precision, single precision, and half precision. It can run up to 4.7 TFLOPS of double precision, 9.3 TFLOPS of single precision, and 18.7 TFLOPS of half precision. P100 contains HBM2 memories in two versions: 16 GB with 732 GB/s, or 12 GB with 549 GB/s.

P100 GPUs support CUDA, DirectCompute, OpenCL, OpenACC. Mainly it supports HPC and deep learning computations. An important feature of those GPUs is the unified memory and page migration. The page migration engine handles the data exchange between the host memory and the memory of the GPU.

NEC SX-Aurora TSUBASA: The third architecture is the vector engines, which are also used in HPC applications as a result of their wide-vector operations that support highly-parallel computations and the necessary supporting memory bandwidth. We discuss technical aspects of such vector engines through the SX-Aurora TSUBASA, which is one of the most modern vector engines.

The SX-Aurora TSUBASA [KMI+18] is a vector engine. It consists of 8 cores running with 1.6 GHz. Each core includes 3 FMA units, each of which handles 32 double precision floating point operations per cycle. The peak performance is 307.2 GFLOPS per core,

Table 2.1.: Memory access on Broadwell, P100, and SX-Aurora VE

Processor	Memory Interface	Theoretical Memory Bandwidth (GB/s)	LLC (MB)	#Cores
Broadwell	DDR4	76.8	45	18 (x2 threads)
P100 GPU	HBM2	549 or 732	4	3584 (SP)
SX-Aurora VE	HBM2	1200	16	8

2.45 TFLOPS for the vector engine.

Registers of the SX-Aurora TSUBASA hold 256 double precision floating point entry per register. The SX-Aurora TSUBASA contains shared 16 MB LLC. It is designed with HBM2 memory interface. Average memory bandwidth achievable is 150 GB/s per core, and 1.2 TB/s for the vector engine.

Comparison in terms of memory access: The importance of achieving optimal memory access in stencil computations leads to focus further on architectural features related to memory access. A summary in Table 2.1 shows that different memory interfaces are used in the different processors. The choice of the memory interface affects the data transfer rates. The accelerators, i.e., the P100 and the SX-Aurora, use HBM2 memory interfaces, which provides higher data rates. Clear differences are shown in the theoretical memory bandwidth between the three processors. An important point to consider to minimize memory access is to optimally use the caches. The differences between the three processors in terms of sizes of last level caches lead to different optimization needs.

In fact, not only cache sizes of LLC are important, but also the levels of cache and the sharing among processing resources (cores). We mentioned LLC so far as existence of data in this cache eliminates the need to access memory. However, distributing operations across processing resources (cores) should be done counting for closer, i.e., per core, caches in order to exploit those cache levels. In Table 2.1 we added a column to show the number of cores. The three processors use different numbers of cores, which affects how a computation should be parallelized over the cores to maximize the use of the caches.

2.4. Optimization Techniques

To exploit the features provided by hardware, software is normally written considering underlying hardware. A set of well-known techniques to use different hardware features are considered according to nature of software.

2.4.1. Parallelization

Modern processor architectures moved to scale computing through parallelization. Different architectures provide different techniques to parallelize computations. Scaling at the level of core count is one technique to provide parallel computing. Threads are organized to execute operations within cores in different ways according to architecture. GPUs, for example, organize threads within warps to execute operations in a way that is different from the role of threads of a multi-core processor.

Different programming models have been developed to allow software developers to parallelize code execution, e.g, pthreads [But97], OpenMP [Opec], and OpenACC [Opea]. Architectural details affect the design of the different programming models. Some programming models can be specialized for specific architectures, and some are planned to span different architectures. Developers should choose a programming model that fits the architecture they develop code for.

To demonstrate the impact of parallelizing code over multiple cores, we consider the work presented in [CBPP02]. From this work we cite the code Listing 2.2, which illustrates the parallelization of a simple stencil applied within a Jacobi [Jac45] computation. OpenMP is used to execute the iterations of the j loop over different execution threads.

Listing 2.2: **Parallelization of Jacobi using OpenMP ([CBPP02])**

```
1 !$OMP PARALLEL DO
2 do j = 1,n
3   do i = 1,n
4     A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
5   end do
6 end do
7 !$OMP END PARALLEL DO
```

This parallelization reduces the execution time of the code corresponding to the number of threads used to execute the code (see results in [CBPP02]).

2.4.2. Vectorization

Modern architectures support vector instructions. Vectorization allows executing instructions that apply operations to multiple data elements. Such capabilities are provided in different forms by the different architectures. Therefore, the concept is provided by GPU threads in a different way from that of multi-core processors.

Developers can exploit vectorization capabilities with different ways according to architectures. Compilers could do the vectorization automatically in some cases. Pragmas could guide compilers to make vectorization decisions. Writing some code parts with assembly or using compiler intrinsics allow developers to do the vectorization manually. In comparison, CUDA [CUD20] allows developers to assign computed elements to GPU threads.

To demonstrate the vectorization concept, let's see how the sum of two values stored

in two arrays can be vectorized. The code in Listing 2.3 shows a simple C loop to go over the values summing values from both arrays into a third one.

Listing 2.3: **A simple loop to add two values from two arrays into a third array**

```
int x[TESTSIZE];
int y[TESTSIZE];
int z[TESTSIZE];

for(int i=0; i < TESTSIZE; i++)
{
  z[i] = x[i] + y[i];
}
```

To simplify thinking about vectorization, assume we treat each four consecutive values in arrays as a group, then we can iterate over groups as shown in Listing 2.4. Actually those groups are treated as vectors when using SIMD capabilities of hardware, where vectors comprise four values. What we do is get a group, add each pair of inputs from the group (the vector), and repeat for the next group (four values).

Listing 2.4: **Divide loop iterations to vectors of elements**

```
  for(int i=0; i < TESTSIZE ; i += 4)
  {
    z[i]   = x[i  ] + y[i  ];
    z[i+1] = x[i+1] + y[i+1];
    z[i+2] = x[i+2] + y[i+2];
    z[i+3] = x[i+3] + y[i+3];
  }
```

In fact, the code in Listing 2.4 we still apply operations on single values, unless automatic vectorization can transform code to use SIMD. However, we can do the actual vectorization based of the thoughts from Listing 2.4 through applying some changes as shown in Listing 2.5.

Listing 2.5: **Vectorized code of**

```
  __m128i *xx=(__m128i *)x;
  __m128i *yy=(__m128i *)y;
  __m128i *zz=(__m128i *)z;

  for(int i=0; i < TESTSIZE / 4; i++)
  {
    zz[i] = _mm_add_epi32 ( xx[i], yy[i] );
  }
```

In Listing 2.5 we iterate over each group of values (4-value vector) and apply SIMD operations. In this code we apply the intrinsic call $\_mm\_add\_epi32()$, which adds two vectors using SIMD capabilities of processor. A operation on four values is done in same time used by an operation on a single value as a result of using vector instead of scalar operations.

2.4.3. Data Layout and Loop Order

Memory is accessed extensively within loop nests. For example, the loop nest shown in Listing 2.2 accesses memory to read four values (16 bytes assuming single precision floating point) and write one value (4 bytes–assuming single precision) for each iteration, and repeats this for $n \times n$ iterations. Data locality of memory accesses within stencil computations represent an important opportunity to exploit caches. On the other hand, under access violating memory hierarchy, caches could be reducing performance.

Data should be stored in memory such that maximum number of elements moved to caches can be used, and data movement from/to the caches from/to memory should be minimized. This needs developers to find optimal data layout in memory. The layout of data is then a characteristic of the source code after such decision is made, as source code is written based on a specific assumed data layout.

Along with finding optimal data layout, data should be accessed in an order that keeps the use of caches optimal. This needs to write loop nests with an optimal loop order. Developers should write loops counting for the layout of data and how that data is accessed if a specific loop interchange is applied. Wrong loop orders can reduce performance substantially.

To demonstrate the impact of data layout and vectorization, we cite the work presented in [HSP+11]. The authors present experimental results of performance impact of applying vectorization automatically and using intrinsics besides data layout transformations. Different codes are tested on different processors. The numbers show viable results of applying data layout transformations and optimal use of vectorization capabilities.

2.4.4. Cache Blocking

Caches represent an important part of memory hierarchies in modern architectures. Optimal use of caches reduces access to farther memory levels, reducing access time for data, and hence improving performance of code. An important technique to increase the efficiency of cache usage is cache blocking. Cache blocking is applied by transforming loop nests by partitioning the loop traversal space, and hence, sub-setting the traversed data into partitions, and processing operations that need to access each partition while it resides in caches, before moving to traverse other partitions. This optimization technique allows significant performance improvement for stencil computations, which are sensitive to memory bandwidth. Applying this technique needs transforming loop nests, which is done normally by developers based on optimal blocking factors per machine. However, the performance gain of such transformations is worth the effort.

To demonstrate the use of blocking, we cite the work in [RT00] on tiling of 3D stencil computations. An example simple code of the 3D Jacobi before optimization is shown in Listing 2.6.

Listing 2.6: **3D Jakobi ([RT00])**

```
1 do K=2,N-1
```

```
2   do J=2,N-1
3     do I=2,N-1
4       A(I,J,K) = C*(B(I-1,J  ,K  ) + B(I+1,J  ,K  )+
5                     B(I  ,J-1,K  ) + B(I  ,J+1,K  )+
6                     B(I  ,J  ,K-1) + B(I  ,J  ,K+1))
```

After applying the tiling, the code is shown in Listing 2.7.

Listing 2.7: **Tiled 3D Jakobi ([RT00])**

```
1 do JJ=2,N-1,TJ
2   do II=2,N-1,TI
3     do K=2,N-1
4       do J=JJ,min(JJ+TJ-1,N-1)
5         do I=II,min(II+TI-1,N-1)
6           A(I,J,K) = C*(B(I-1,J  ,K  ) + B(I+1,J  ,K  )+
7                         B(I  ,J-1,K  ) + B(I  ,J+1,K  )+
8                         B(I  ,J  ,K-1) + B(I  ,J  ,K+1))
```

The code after optimization applies blocking to the I and J dimensions. The authors state that the technique lead to 17-121% performance improvements for key scientific kernels. However, it makes the code significantly more complex and places the burden onto the developer.

To demonstrate performance impact of blocking, we cite the work [KHO+05]. In this work, the authors apply the technique from [RT00] to optimize two applications. The authors state that the experiments show a minimum speedup of 10% and an average of 22%.

2.4.5. Loop Fusion

Besides optimizing loop nests according to different techniques, optimizing sequences of loops could bring additional performance improvements. Loop fusions reduce loop control instructions. However, this is not the main source of performance improvement for stencil codes. In comparison, data reuse across stencils brings much more performance improvement. Stencils access field data which could be accessed by other stencils. If data accessed within one stencil is used to compute other stencils while it is still in caches, this allows to reduce memory access.

To exploit such techniques, data dependencies and operation precedence should be analyzed to hold computation consistency under the applied transformations. This task needs developers to do an additional effort for the analysis and the code modifications. Identifying fusions, and the consequences of code transformations represent a challenge for developers to guarantee that fused kernels yield valid results.

To demonstrate the use of loop fusions, we cite the work in [MCT96]. An example simple code, in which stencils are executed within multiple loops, is shown in Listing 2.8.

Listing 2.8: **Stencils executed within multiple loops ([MCT96])**

```
1 DO I = 2, N
2   DO K = 1, N
3     X(I,K) = X(I,K) - X(I-1,K) * A(I,K)/B(I-1,K)
4   DO K = 1, N
5     B(I,K) = B(I,K) - A(I   ,K) * A(I,K)/B(I-1,K)
```

After applying the loop fusion, the code is shown in Listing 2.9.

Listing 2.9: **Applying loop fusion to code in Listing 2.8 ([MCT96])**

```
1 DO K = 1, N
2   DO I = 2, N
3     X(I,K) = X(I,K) - X(I-1,K) * A(I,K)/B(I-1,K)
4     B(I,K) = B(I,K) - A(I   ,K) * A(I,K)/B(I-1,K)
```

To demonstrate performance impact of loop fusion, we cite again the work [MCT96]. The authors show reduced execution time of an application (a PDE solver using ADI integration [BLT09] with 3D arrays) on different processors as a result of loop fusions.

2.4.6. Call Inlining

With call inlining, a call to a function/procedure is replaced with the code forming the body of that function/procedure. The body of the function/procedure is integrated within the context of the calling code. Call inlining introduces performance improvements. Basically call overhead is dismissed when inlining a call. However, this technique provides more valuable results when applied in stencil computations. Inlining a call opens the door for further optimization opportunities to inlined code. Mainly, calls that apply stencils could possibly allow loop fusions of loop nests when inlined.

2.5. Domain-specific Languages

General-purpose programming languages are usually used to develop software because they are designed to solve computational problems in general. On the contrary, there have been special-purpose languages that were designed to solve specific kinds of problems in a specific application domain.

A well-known special-purpose language is the structured-query language (SQL), which is specialized to handle relational databases. The example SQL statement

```
SELECT * FROM some_table;
```

allows to retrieve data from a table through the use of special keywords, e.g., SELECT, and usage rules.

Domain-specific languages (DSL) [Fow10] are designed with a specific domain in mind. This makes DSLs include a set of constructs that correspond to the concepts of the

domain, rather than counting for general computation needs. Such characteristic should make DSLs limited in scope, simpler to learn, and more expressive, although in practice this is not always the case.

In fact, well-designed DSLs can allow domain specialists exploit their deep knowledge in their field into building powerful applications. However, in many cases the compromise is between learning a new language and continuing using an already in use general-purpose language.

2.5.1. Challenges Facing the Use of DSLs

DSLs, as they serve specific domains, are used among smaller communities in comparison to GPLs (general-purpose languages). Therefore, they availability and support for tools is normally weaker than that of GPLs. Further, the number of DSLs that are developed in a domain within different teams limits the standardization of DSLs, which is not the case for GPLs.

Besides to standardization and tools, with smaller communities using a specific DSL it would not be easy to find many experts to refer to when problems arise. Also, documentation and sample code are not available in large amounts like in GPLs.

Another point to consider is code integration with other parts of software. Code that should be include DSL code, which should also work with other parts of software that are written in other languages represent an important point to consider when using DSLs. All mentioned considerations affect the decisions of using DSLs to develop applications.

2.5.2. Code Processing

Source code should be processed by some tools, e.g, compilers, as part of the application development process. Code that is developed with a DSL can be processed in either a special tool, or using GPL tools. In this regard, DSLs are classified in external and embedded DSLs. External DSLs need to have own tools to process source code while building applications. Embedded DSLs are designed and based on constructs of a host language. The tools/compiler of the host language are used to process the DSL code.

Chapter summary

In this chapter, we reviewed scientific concepts from the scientific domain of earth system modeling. We talked about models, stencils, kernels, and different kinds of grids. We also discussed the structure of data with those different grid kinds, and related concepts including filling curves and connectivity. An overview of Broadwell multi-core processors, P100 GPUs, and SX-Aurora TSUBASA vector engines was made briefly to provide technical background on the three architectures. We then introduced some optimization techniques which should be considered to optimize stencil computations. Lastly, we mentioned the basic concepts of DSLs, more details for practical relevant concepts are discussed in the next chapter.

In the next chapter, we provide a literature review related to the topic of this thesis.

The evolution of the solutions in the field and the state of art will be discussed in more detail.

3. Literature Review

In this chapter we review related work. We start with early solutions that used computers to solve scientific problems in Section 3.1, and the move to more performance-demanding numerical solutions. Next, in Section 3.2, we look at later techniques which arose as alternatives to manual optimization to exploit hardware resources. We discuss further the use of high-level coding in Section 3.3. In Section 3.4, we discuss in more detail concepts from two early related DSLs to get better understanding of DSL concepts in earlier work. In Section 3.5, we discuss the state of art of the research in the topic. We review in this section recent and ongoing closely related techniques, problems, and solutions. Finally, we provide a gap analysis in Section 3.6 to emphasize the research described in this text.

3.1. Historical Roots

In this section we look at the early history of the problem. We look at research done to apply software solutions to mathematical problems through machines, and how the problem turned into an HPC problem.

3.1.1. Numerical Solutions and Mathematics Software

Using computers to provide numerical solutions of partial differential equations has been used to simulate systems in different domains and applications for many decades. Among the early software packages that were developed to support numerical solutions is Ellpack [RB85], which was proposed to solve elliptical problems.

Besides to PDEs, linear algebra problems were subject to support with numerical solutions using computer machines. One of the first libraries that were developed to numerically solve linear algebra problems was Basic linear algebra subprograms (BLAS) [HKL73]. It was proposed to support Fortran programs with a set of suggested subprograms. BLAS was also implemented in assembly languages to support specific machines (IBM 360/67, the CDC 6600 and CDC 7600, and the Univac 1108) [LHKK79].

In the meantime, there have been early discussions of the portability [ABG77] of scientific software. The multiple versions of the software implementations (as we see in BLAS) brought development complexities and costs. The reprogramming of the existing software for new hardware was expensive. In later versions/extensions to the BLAS library [DDCHD90], providing portable and efficient implementations on high-performance computers got more attention.

To support solutions of mathematical problems on their processors, Intel developed their own library. In 2003 Intel released their initial version of MKL (Math Kernel

Library) [Int11] to support scientific, engineering, and financial applications. MKL provides manually-optimized implementations of a set of mathematical functions for different Intel processors. MKL was also optimized in different versions to optimally support the different processors.

3.1.2. HPC Considerations

In this section, we discuss efforts that shift the focus to efficiency and the optimization of the codes to exploit the computational power of the emerging computing technologies.

3.1.2.1. Parallelization

With the increasing need for computational resources and the accelerating advances in computing technologies, more care was given for optimizing the use of the hardware resources to run the software efficiently. Optimizing the use of the computational resources of multiple-computer/processor systems was an important point to consider to scale the software solutions.

The parallelization of large scientific applications over multiple computers/processors with high speed networks was made possible with both application-specific and system-specific codes [KSGK91].

High-performance Fortran (HPF) [Lov93] is one example of the efforts to extend a programming language, which is often used to develop scientific codes, to support parallel computing. HPF allowed work distribution among processors through data-parallelism. Some HPF features were difficult to implement, therefore, some compilers did not include the extensions. Furthermore, the rise of other viable solutions, e.g., OpenMP, limited the inclusion of HPF in compilers. However, HPF influenced the standardization of the later versions of Fortran.

Among the efforts to parallelize the numerical PDE solutions, [HRC+90] provided an Ellpack-based programming environment for parallel MIMD PDE solvers. Also, [Yao98] investigated the parallelization of the elliptic PDE numerical solutions on distributed memory systems. A study of the parallelization techniques of the PDE solutions and the parallelization algorithms was done in [HSV+98].

There have also been an effort to parallelize the BLAS library. The library LAPACK [CDO+95] was developed to provide a set of parallel basic linear algebra subprograms (PBLAS).

Domain decomposition was one of the points of research towards the parallelization of PDE solutions over multiple processors/computers. Examples of efforts include [CHH+90], [MF94], and [CMF95].

3.1.2.2. Blocking and Tiling for Memory Hierarchies

Besides to parallelization, there have been other efforts to optimally use other computational resources to efficiently run the computations. The efficient use of the memory

hierarchies minimizes the data access time, which reduces the time to run the computations. Blocking and tiling are techniques to improve the use of the caching memories.

The automatic restructuring of code was discussed in [CK89]. The aim of the restructuring is to benefit from caches with blocking. The memory hierarchy of the machine affects the impact of the blocking, thus tuning the code is necessary to optimally use the cache memories. [SD90] presented automatic blocking of nested loops to get rid of manual blocking and tuning for a specific machine.

Besides to the efforts to automatically block code, there are still efforts to use empirical tuning to optimize caching usage within scientific libraries. For example, empirical tuning of the blocking factor in Lapack to increase performance is used in [Wha08]. As mentioned earlier, Intel provides multiple optimized versions of the MKL library to support the different architectures. This is essential to allow a library to run with an optimal performance on an architecture.

Tiling of the iteration space to optimize the memory access in memory hierarchies was also discussed in [Wol87]. The author then discussed in [Wol89] the tiling and the parallelism of the tiled codes and the necessary code transformations to implement tiling on high performance parallel machines.

3.1.2.3. GPUs

Parallelization efforts lead further to use GPUs for general purpose computing. The authors of [BP07] implemented a 2D Euler solver on GPUs. [MV08] discusses the implementation of a computationally-intensive part of the WRF (Weather Research and Forecast) model on GPUs. The implementation achieved high speedup for the ported part, but also achieves an improved performance for the whole model.

More frequent data movement between the GPU's memory and the main memory of the host needs more time to alternate the computation between the CPU and the GPU. [LF14] discusses the potential to fully port an application to GPUs with OpenACC. Minimizing the need for data movement between the host and the GPU memories is a main concern that motivates the porting of the code fully to the GPU. The assessment was done with the COSMO climate and numerical weather prediction code. The results showed that GPUs achieved 3 to 7 times speedup on compared to the multi-core processors of the same generation.

Moving the data from and to the GPU is time consuming. The optimization of this data movement is essential to exploit the processing power of the GPUs. One effort to optimize the data movement from/to the GPU is MCL [Tuf17]. MCL is a library to support multiple GPUs programming. It provides efficient communication between the GPUs and the CPU.

Efforts to port and optimize libraries for GPUs and heterogeneous machines, e.g, BLASX [WWX+16], allow to develop scientific codes that use those machines efficiently.

3.2. Tool-Based Optimization

The manual optimization of the source code is time consuming and costly. Furthermore, deep knowledge in the architectural aspects and the running environment is needed to be able to optimally use a machine. Thus, there have been efforts to get around the manual optimization.

3.2.1. Automatic Tuning

The Automatically Tuned Linear Algebra Software (Atlas) [WPD01] presented a new technique called automated empirical optimization of software. The technique was intended to help manage the library to keep up with the accelerating hardware advances.

Tuning the software both for the running architecture and the user problem to run the software efficiently is discussed in [DDE+05]. The authors describe approaches to obtain tuned high-performance kernels, and the way to automatically choose the suitable algorithms. The approach is discussed mainly to generate sparse and dense BLAS kernels, and the selection of linear solver algorithms.

3.2.2. Code Generation for Numerical Simulations

The manual optimization of the code needs effort to optimally use the features of an architecture. Therefore, many research efforts used alternative techniques to avoid the manual optimization.

DEQSOL [Kon86] (Differential Equation Solver Language) provided a higher-level programming language for numerical solutions of differential equations. DEQSOL aimed at improving programming productivity with an architecture-independent language. This higher-level language programming improved the programming productivity with an order of magnitude compared to the Fortran programming in terms of lines of code. The higher-level code is translated into highly vectorizable Fortran code by a translator [KYS+87]. The generated code showed high vectorization ratios on the tested vector machines.

GENCRAY [WW92] provided a portable code generator that was written in the C language. The GENCRAY code generator accepts input that conforms to a LISP-style language. It generates Fortran 77 or Cray Fortran code. GENCRAY simplified the vectorization of the numeric codes and the parallelization on multiprocessor machines.

Later, FALCON [DRGG+95] used high level array language (MATLAB) as a source language, and provided static, dynamic, and interactive analysis and based on the analysis generated Fortran 90 code. The generated code included the parallelization directives. The paper showed that the generated code achieved 48 times the speed of the interpreted MATLAB code on a serial machine, and 140 times on a vector machine.

CTADEL [VEWC96] used higher-level language specifications for differential equations to generate code for serial, vector, shared virtual memory and distributed memory parallel computer architectures. This automated code generation solution was efficiently used to generate code for a limited area numerical weather prediction.

3.2.3. Performance Portability

With the emerging diversity of architectures, scientists realized the importance of the portability of the efficiency of code execution on different architectures. As we see with DEQSOL [Kon86], the efforts started shifting to focus on architecture-independent software development. The code generation techniques allowed to use architecture-independent high-level code, as the code is translated to the architecture-optimized code.

An approach of architecture-adaptive algorithms [KU93] was suggested to run algorithms on different parallel machines. Parallel algorithms in this approach adapt their behavior to the characteristics of the computing environment: the processors, memory hierarchies, and communication channels.

[Mar97] studied the user involvement with the code processing and optimization within FALCON [DRGG+95]. The user is involved interactively within the algorithm analysis, algorithm restructuring through transformation patterns, and within code generation. The results show that the interactive user intervention allowed to get more efficient code that automatic code generation.

3.2.4. Domain-Specific Libraries and Languages for PDE Solutions

Among the modern solutions that were developed to provide high-level interface for the PDE solution applications is DOLFIN [LWH12]. It was developed to support PDE solutions with finite element method in C++ and Python interfaces. It supports the unified form language (UFL) [ALØ+14] DSL, which is part of the same project (FEniCS), for the formulation of the PDE problems. UFL was developed to allow scientists to express PDEs in a form that is closer to the scientific representation. It was meant to provide separation of concerns between specifying finite element method as a method for the solution and the implementation of the method.

The work was extended with the Firedrake [RHM+17] library. Firedrake adopted UFL and used python run-time to solve PDEs with additional new automatic optimizations. Firedrake also introduces PyOP2[RMM+12] to separate between the mesh complexities and the parallel execution of the operators over the meshes.

3.3. High-Level Coding Considerations

The shift to use DSLs and higher-level coding represents an important move for high performance computing. Such solutions provide also a natural way to support performance portability.

3.3.1. Declarative DSLs for Stencil Computations

Some research efforts took the direction of code generation to develop new declarative DSLs to support stencil code development.

Ypnos [OBM10] is a declarative DSL to express stencil computations over structured grids. The static analysis of the declarative DSL code leads to generate an optimized parallel code.

Physis [MNSM11] provides a compiler-based programming framework that allows the development of stencil computations over structured grids on GPU-based clusters. The framework provides the users a declarative C-based DSL to represent the stencil computations. The framework translates the user's code into CUDA for the GPUs, and MPI code to handle the parallelization on multiple nodes.

3.3.2. Higher-Level Code Processing

The higher-level code can be processed in different ways. For example, [BRP07] discusses the development of stencil computations with Cray's Chapel language. The language was developed by Cray and supports parallel programming. The stencil codes make use of the language compiler to run efficiently on the hardware. The language constructs allow the programmer to pass information to the compiler to generate more efficient code.

The Pochoir [TCK+11] code can be used in alternative ways. Pochoir is an embedded domain-specific language in C++ to simplify stencil computation development with multi-dimensional grids. It provides a compiler to translate its code into an efficient parallel cache oblivious Cilk code. It also provides template library that can run the DSL code without translation.

Many solutions use the compilation of the higher-level code to generate optimized code for some target architecture.

3.3.3. Active Libraries

Active libraries like DSLs improve coding and performance portability. The application code uses the higher-level interfaces of the library, which are designed to allow for optimization through the domain concepts.

OP2 [MGR+12] is an active library that was developed to provide solutions for unstructured meshes. The application code uses the library API. The code translation uses the API to generate code for multi-core and many-core processors, and for GPUs.

The active library OPS [RMG+14] supports the development of multi-block structured grid computations. The provided API is used to translate the code and apply optimizations, e.g, cache blocking and parallelization. The translation process generates codes for different architectures.

3.3.4. GPU Back-ends

As we see with Physis [MNSM11], the higher-level code is translated into code that targets GPUs.

Programming environments to support the parallel execution over GPUs were developed, e.g, HMPP [DBB07]. HMPP used code annotation to mark codes for the execution

over GPUs. In fact, the annotation technique was used in many parallelization standards, as we discuss in the next section.

Specialized DSLs arose to target GPUs to run PDE solvers. Liszt [DJP+11] was developed to generate code for heterogeneous environments. Liszt provided a high-level DSL to formulate a PDE problem over an unstructured mesh. The DSL allowed the provided compiler to handle the parallelization, locality and synchronization.

GPUs were considered as targets for stencil code generators. In [HPS12] a scheme to generate stencil codes that run on GPUs efficiently is presented. The authors developed compiler algorithms to translate stencil operations which are described with a high-level form. The generated code is optimized for GPUs with the focus on maximizing computation and minimizing memory access. The compiler algorithms apply time-tiling to the generated GPU code.

STEPOCL [LBN13] is a code generator that was proposed to generate stencil codes for heterogeneous multi-device machines. The kernel descriptions are provided through XML input files. STEPOCL generates OpenCL code and provides the necessary data partitioning and exchange codes. It focuses on minimizing the data exchange to optimize performance on multiple devices.

Panda [SBC17] is a domain-specific compiler framework for stencil computations. The stencil computations are written with C code. Panda provides a set of directives to allow the programmers to guide the code translation process. The source code is translated by the framework to run on GPU-accelerated machines. The code is generated with CUDA,MPI, and OpenMP.

3.3.5. Code Annotation

As we see with Panda [SBC17], code annotation is used in some solutions to guide the optimization during the code translation process. In fact, many standard parallelization techniques, e.g, OpenMP [Opec], OpenACC [Opea] and OpenCL [Opeb], used code annotation to guide compilers to parallelize the annotated codes.

The Mint [UCB11] programming model allows the development of stencil computations for GPUs. The source code is written in C language, and the code is annotated with Mint pragmas. The translation process generates CUDA code.

Among the recent efforts to develop parallel software is MetaFork. MetaFork [Che17] is a high-level programming language that extends C/C++. It provides constructs to express concurrency within the structure of the source code. Parallelization pragmas allow the MetaFork compiler to provide concurrency to the generated code. The code can be translated into different concurrency models to run on multi-core processors.

3.4. Related Early DSLs

In this section we have a closer look at two DSLs to deeper understand the concepts of DSLs from both domains; the atmospheric modeling, and the stencil computations. We

discuss some details from an early DSL for atmospheric modeling, i.e., ATMOL, and another DSL for stencil codes, i.e., PATUS.

3.4.1. ATMOL

As an alternative to manual optimization of atmospheric models, ATMOL [AvE01] was introduced as a domain-specific language for the formulation and implementation of atmospheric models. It is one of the early efforts to use DSLs for the development of atmospheric models. It uses declarative constructs for the formulation of an atmospheric modeling problem.

3.4.1.1. Problem Coding

ATMOL allows the formulation of both the high-level and the low-level model details. ATMOL provides a way to specify the problem through a set of constructs. First, it provides the way to declare the independent variables of the space (and time when not dealing with steady state problems) which represents the domain for the model. Constructs to specify the dependent variables of the model are also provided. Scalar variables can be defined in ATMOL. Fields allow defining a dependent variable with respect to the grid.

ATMOL allows specifying the set of equations for the model PDEs. It provides a set of operators to specify an equation. It allows dealing with boundary conditions by conditional expression specification. Besides to the set of operators provided by ATMOL, it allows defining additional operators.

The following sample ATMOL computation specification (Listing 3.1) demonstrates the use of the mentioned concepts.

Listing 3.1: **A sample computation specification with ATMOL [AvE01]**

```
% Declare grid size variables n, m, and l:
n :: integer(1..infinity); m :: integer(1..infinity); l :: integer(2..infinity).
% For convenience, define macros for two grid domains spanning (i,j,k):
atmosphere := i=1..n by j=1..m by k=1..l; surface := i=1..n by j=1..m.
% Set coordinate system for symbolic derivation with chain-rule:
coordinates := [x, y]; coefficients := [h x, h y].
% Declare the model fields:
u ::float dim "m/s" field (x(half),y(grid),z(grid)) on atmosphere.
v ::float dim "m/s" field (x(grid),y(half),z(grid)) on atmosphere.
u_aux ::float dim "Pa m/s" field (x(half),y(grid),z(half)) on atmosphere.
v_aux ::float dim "Pa m/s" field (x(grid),y(half),z(half)) on atmosphere.
p ::float(0..107000) dim "Pa" field(x(grid),y(grid),z(grid)) monotonic k(+) on atmosphere.
p_s_t ::float dim "Pa/s" field (x(grid),y(grid)) on surface.
% Define macro for the horizontal wind velocity vector components:
V := [u_aux, v_aux].
% Equations:
p_s_t = -int(nabla .* V, z=1..l).
V = [u, v] * d p/d z.
```

The sample code illustrates defining problem domain with space and time dimensions, a set of fields, macros and equations. Three space dimensions besides to time are used to solve the subject problem. 3D and 2D grids are defined through those dimensions.

ATMOL-provided operators are used to specify the equations that govern the system. Those operators abstract higher-level mathematical operations.

Beyond the higher-level declarative constructs of ATMOL, it allows the specification of lower-level constructs for the numerical solution algorithms. User-defined PDE-based operations can be defined based on lower-level procedural codes.

Operators and expressions: ATMOL follows an operator precedence grammar for the expressions within the PDEs, the intermediate constructs, and the program code. To verify a model's specification, it uses basic types, unit types, and grid types for type checking. It uses type inference and coercion based on its types for dimensional analysis and the conversion of grids.

The aggregate operators in ATMOL are expressed with a uniform notation using a local scope of a variable in a construct. The scope of bindings of an expression is based on the binding of the variables in the expression. The bindings of the variables are identified from the definition constructs in the coordinates part. The free variables within the expression are processed by a substitution algorithm to simplify the expressions and for partial evaluation. For the storage of the result of an expression, ATMOL uses the local bindings and the free variables to identify the right type, dimensionality, and array bounds of the storing temporary array. To evaluate an expression, ATMOL analyses the set of free index variables which constitute the grid space for the problem. To handle array bounds, a domain inference and value range propagation algorithm is used. This guarantees that no references to grid points outside the problem domain occur.

3.4.1.2. Levels of Abstraction

Separation of concerns for the implementation of a model is exploited with five levels of abstraction: the meta level for the algebraic expressions symbolic manipulation, the model declarations, the coordinate-free scalar PDE problem, the numerical schemes, and the program code. The flexibility of ATMOL to allow specifying a model with a mix of the five abstraction levels (higher-level with lower-level details) is useful for some models which need to write lower-level non-PDE operators. It also allows bypassing the automatic higher-level code translation. The user can intervene with the ATMOL code processing at the different phases of the code translation process.

3.4.1.3. Code Generation

CTADEL [VEWC96] is used to translate ATMOL based model codes into efficient numerical codes. The role of CTADEL is to generate code from the higher-level specifications of a PDE-based model.

CTADEL uses a hierarchical classification of functionals and objects with their algebraic properties. This classification enhances the automatic simplification of the operators (based on its class). Many properties are done implicitly by CTADEL, e.g, associativity and commutativity, however they can be declared in ATMOL, and then CTADEL uses them for pattern matching and rewrite-rules application.

The higher-level ATMOL functional code of the operators is translated into a lower-level procedural pretty-printed Fortran code. The translation is done with template definitions of the operators. During the ATMOL code processing, most of the optimizations are done with the higher levels, before being processed in the lower-level code generation.

3.4.2. PATUS

PATUS [CSB11] is a framework for auto-tuning and code generation of stencil codes, which are parallelized and optimized for multi-core or many-core architectures. It takes the specification of the stencil operation plus the strategy that describes the parallelization and the optimization, and generates optimized code for the target architecture.

3.4.2.1. Problem Coding

A C-like domain-specific language is used to specify the stencil kernels. The stencil specification separates the higher-level description from the lower-level algorithmic implementation.

A stencil specification defines a rectangular stencil domain over which the stencil operation is applied. The stencil operation is defined with input and output grids given as arguments. The iteration of the stencil operation over the domain is not coded, instead, the code is generated based on the domain that is defined in the stencil specification. Therefore, the stencil operation is defined using a localized point-wise stencil expression. The number of time steps is defined within a stencil specification also.

The following code snippet Listing 3.2 demonstrates the specification of the Laplacian operator using PATUS.

Listing 3.2: **A sample PATUS stencil specification [CSB11]**

```
1 stencil laplacian
2 {
3   operation (double grid u,
4              double param alpha, double param beta)
5   {
6     u[x, y, z; t+1] = alpha * u[x, y, z; t] + beta *
7                     ( u[x-1, y, z; t] + u[x+1, y, z; t] +
8                       u[x, y-1, z; t] + u[x, y+1, z; t] +
9                       u[x, y, z-1; t] + u[x, y, z+1; t] );
10  }
11 }
```

An output field is updated based on a provided equation to compute the Laplacian with a 3D stencil. Explicit indices are used to specify the spatial relationships of the stencil elements. The indices besides to operators allow computing the output value for the next time-step.

3.4.2.2. Optimization

PATUS uses another DSL to write the code generation strategy. The strategy is independent of both the stencil code and the hardware.

A set of strategies are provided with the framework, however developers can build their own strategies. As the strategies are hardware independent, the auto-tuning is the way to provide performance portability. That is done by the selection of the suitable parameter configuration to fit the target architecture based on the stencil specification, the strategy, and the target hardware.

A strategy separates a stencil specification from its optimized implementation. Strategies allow defining loop structure –e.g, time step and blocking. Parallelization options are defined in strategies. Strategies depend on parameter values to control code generation. Some of those parameters are decided by the auto-tuner to optimize the generated code based on an architecture specification.

Listing 3.3 demonstrates the specification of PATUS strategy to apply cache blocking.

Listing 3.3: **A sample PATUS stragtegy specification [CSB11]**

```
1  strategy cacheblock (grid u, auto dim cb)
2  {
3    // iterate over time steps
4    for t = 1 .. stencil.t_max
5    {
6      // iterate over subdomain
7      for subgrid v(cb) in u[:, t] parallel
8      {
9        for plane pln in v[:, t]
10         for point pt in pln[:, t]
11           v[pt, t+1] = stencil (v[pt, t]);
12     }
13   }
14 }
```

The defined strategy applies a specified stencil by loops that divide the field update space into sub-spaces to optimize the use of the caches.

3.4.2.3. DSL processing and Code Generation

PATUS includes one parser for the stencil specifications and another for the strategies. As PATUS is written in Java, it uses Java classes provided by Cetus [DBM+09] for the internal representation of the strategies and the generated code. The code generator uses the abstract syntax tree of a strategy besides to the internal representation of the stencil specification to generate optimized code. Additional configuration information about the hardware characteristics and the architecture's programming model and the code generation back-end that should be used are provided to the code generator. The loops structure specified in a strategy that is used to generate code is used to generate C loops by the code generator. The parallelization of the C code is also decided based on the strategy specification. Further processing of loops like vectorization and unrolling could

be applied also to generated code. The strategy specification allows the code generator to determine the arrays and calculate the indices which it uses within the generated code.

The architecture specification and the code generation back-end are responsible for generating architecture-optimized code. Thus, modifying them allows support for further architectures besides to OpenMP and CUDA.

3.5. State of Art

In this section we describe the state of art of research in the field through looking on details of recent and ongoing efforts. A set of DSLs for performance portability, earth system modeling, and stencil computation development are discussed.

3.5.1. Kokkos

Kokkos [ESP+12] is a programming model to support productivity and performance portability for shared-memory systems. It is a C++ library and targets multi-core processors, Xeon Phi, and GPUs. Kokkos provides memory layout optimization through adapting multidimensional arrays storage according to the architecture.

3.5.1.1. Problem Coding

Kokkos defines 'parallel_for', 'parallel_reduce' and 'parallel_scan' patterns that the user uses in the source code to express the application computations. C++ functors or lambdas are used to convey the computational bodies that will be applied in parallel. The parallelization patterns accept the functor/lambda as a parameter. The execution space or the iteration count is also given to the parallelization pattern as a parameter.

Execution spaces, where Kokkos parallel patterns are executed, are defined by setting a default target. However, execution spaces can be passed as parameters when defining a parallelization pattern. The functor/lambda code defines the work unit.

3.5.1.2. Optimization

The Kokkos language constructs allow users to control data access within the source code. Data layout is controlled through those constructs.

Data access: Kokkos allows users to define 'view' objects to access the data. Views store pointers to the real data storage which resides on the real storage memory, e.g, host memory or GPU memory. They are multi-dimensional arrays, with numbers of dimensions known at compile time, while dimension ranges known either at compile time or at run time. The storage space for a view is specified through the view definition, and it should be known at compile time. The allocation of the data storage is done explicitly (no hidden/implicit allocation is done within Kokkos). The deallocation is handled by reference counting.

To access data from both host and GPUs, users can use 'MirrorView's or 'DualView's. 'MirrorView's are expensive as they cause deep copy between the host memory and the GPU memory. With 'DualView's the user should keep track of the data update and ask Kokkos to handle the copy.

The following snippet in Listing 3.4 shows a sample code using KOKKOS.

Listing 3.4: **A sample KOKKOS code (Based on code from [koka])**

```
typedef Kokkos::View<double*[3]> view_type;

...

    view_type a ("A", 10);

...

    Kokkos::parallel_for (10, KOKKOS_LAMBDA (const int i) {

      a(i,0) = 1.0*i;
      a(i,1) = 1.0*i*i;
      a(i,2) = 1.0*i*i*i;
    });
```

The snippet demonstrates the use of the *parallel_for* construct. A lambda is provided to be applied in parallel. Field data is accessed within the lambda through a view object.

Data layout: The data layout of the multi-dimensional array is defined at compile time. The user specifies the data layout through the definition of the view. By default, the 'LayoutLeft' is used for GPU memory storage, where stride 1 corresponds to the leftmost index, and the 'LayoutRight' is used for host memory storage, where stride 1 corresponds to the rightmost index. Users can choose other layouts, e.g, LayoutStride and LayoutTiled, or extend them.

The following snippet Listing 3.5 demonstrates the use of different memory layouts.

Listing 3.5: **A sample showing layouts in KOKKOS (Based on code from [kokb])**

```
typedef Kokkos::View<double**, Kokkos::LayoutLeft> left_type;
typedef Kokkos::View<double**, Kokkos::LayoutRight> right_type;
typedef Kokkos::View<double*> view_type;

...

template<class ViewType1, class ViewType2>
struct contraction {
  view_type a;
  typename ViewType1::const_type v1;
  typename ViewType2::const_type v2;
  contraction (view_type a_, ViewType1 v1_, ViewType2 v2_) :
    a (a_), v1 (v1_), v2 (v2_)
  {}

```

```
16   KOKKOS_INLINE_FUNCTION
17   void operator() (const view_type::size_type i) const {
18     for (view_type::size_type j = 0; j < v1.extent(1); ++j) {
19       a(i) = v1(i,j)*v2(j,i);
20     }
21   }
22 };
23
24 ...
25
26     int size = 10000;
27     view_type a("A",size);
28
29     // Define two views with LayoutLeft and LayoutRight.
30     left_type l("L",size,10000);
31     right_type r("R",size,10000);
32
33 ...
34
35     Kokkos::parallel_for(size,
36         contraction<left_type,right_type>(a,l,r));
```

The first few lines of the snippet show how the memory layout of a view is specified. The code later accesses the views based on the chosen layout, as we see in the operator definition, where the code should use the right indices to access data (notice $v1(i,j)$ and $v2(j,i)$) in Line 19.

3.5.2. YASK

Yet Another Stencil Kernel (YASK) [YTBD16] provides tools to generate optimized stencil code from higher-level stencil specifications. Stencils are applied over structured grids in codes that target Xeon and Xeon Phi processors. It is intended to be used to explore stencils and their performance on those processors.

3.5.2.1. Problem Coding and Optimization

Stencil specifications are written through C++ classes inheriting specific YASK-provided base classes. Grid definition is done within stencil specification code. During the build process, 'Fold Builder' converts the stencil into optimized C++ code. Optimization techniques like vector folding [You15] and cache blocking are used to optimize memory bandwidth usage.

Kernels that apply the stencil over the grid are generated throughout the build process. YASK provides Perl tools to generate optimized C++ code. This process is driven by a simple DSL. DSL code is given as an argument when calling the Perl tool to control the grid traversal code.

Stencil specification: YASK provides the class 'StencilBase' and its derived class 'StencilRadiusBase' as a basis to specify stencils. Abstract method 'define' should be

implemented in derived classes to determine stencil operation. Expression statements within the 'define' method are the tool's way to identify stencil computation formulas. YASK uses overloaded operators to build an abstract syntax tree (AST) – an AST is a tree structure that syntactically corresponds to the source code of a program. This AST is then used to generate optimized code making use of AVX vectorization. Grids are defined as members of stencil derived classes. Dimensions of the grids are determined in stencil class constructors.

The snippet in Listing 3.6 demonstrates the specification of a stencil within YASK. The snippet shows a specification of a 2D stencil with a specific radius. A class is derived to make use of a base class, where time and space dimensions are specified and grid is defined. Loops are executed to generate the access to the input data from the four stencil directions. Finally, the computed value is assigned to the next step.

Listing 3.6: **A sample YASK stencil specification [yas]**

```
class Test2dStencil : public StencilRadiusBase {

protected:

    // Indices & dimensions.
    MAKE_STEP_INDEX(t);           // step in time dim.
    MAKE_DOMAIN_INDEX(x);         // spatial dim.
    MAKE_DOMAIN_INDEX(y);         // spatial dim.

    // Vars.
    MAKE_GRID(A, t, x, y); // time-varying grid.

public:

    Test2dStencil(StencilList& stencils, int radius=2) :
        StencilRadiusBase("test_2d", stencils, radius) { }
    virtual ~Test2dStencil() { }

    // Define equation to apply to all points in 'A' grid.
    virtual void define() {

        // define the value at t+1 using asymmetric stencil.
        GridValue v = A(t, x, y) + 1.0;
        for (int r = 1; r <= _radius; r++)
            v += A(t, x + r, y);
        for (int r = 1; r <= _radius + 1; r++)
            v += A(t, x - r, y);
        for (int r = 1; r <= _radius + 2; r++)
            v += A(t, x, y + r);
        for (int r = 1; r <= _radius + 3; r++)
            v += A(t, x, y - r);
        A(t+1, x, y) EQUALS v;
    }
};

REGISTER_STENCIL(Test2dStencil);
```

Grid traversal: Perl tools take DSL commands to generate optimized C++ grid traversal code. The DSL is similar to C++ syntax. It drives the kernel loops generation. Grid traversal in kernels is divided in four looping structures to get higher performance. Multiple chips, multiple threads, caching are taken into account. The four looping structures are for ranks, regions, blocks, and halo handling. In its traversal generated code, YASK makes use of caches L1 and L2 with controlled prefetching. OpenMP is used to make use of processor multiple threads, by scheduling region loops. It also parallelizes halo and block loops. MPI is used to handle multiple ranks.

3.5.2.2. Code Processing

A Perl tool 'gen-loops.pl' takes DSL code as an argument and generates code for each of the looping structures. C++ header files are generated by this process. Provided source code uses the dynamically generated code in the generated headers to traverse grid. Grid update is done with this generated code plus code generated automatically by 'Fold Builder' to apply stencil operations at each grid point.

3.5.3. ExaStencils

ExaStencils [LAB+14] is an effort to support solving PDEs on structured grids using automatic generation of code. The solution is abstracted in four levels: continuous domain & continuous model, discrete domain and discrete model, algorithmic components and parameters, and complete program specification. The main idea is to allow the scientists to focus on research instead of programming skills, and to shorten the time to modify application after some parameter changes, where the code generator will generate the code again with the necessary changes, removing the burden on the scientists to modify the source code.

What the scientists need to do is provide a set of choices, e.g, solver, boundary conditions, stencil patterns. The tools will then generate the code for the chosen platform. ExaStencils provides the ExaSlang DSL [SKH+14] for programming numerical solvers.

3.5.3.1. Problem Coding and Levels of Abstraction

ExaSlang DSL is used for problem coding. ExaSlang is designed in four abstraction layers in order to support the different users and their needs in a separate level. The top layer supports domain scientists, and the last contains the most concrete form of multi-grid methods that fit a specific target platform. In between are layers for the mathematical and computational structures.

The following snippet Listing 3.7 illustrates a stencil specification in ExaSlang.

Listing 3.7: **A sample ExaSlang stencil specification [SKH+14]**

```
Val kappa : Real = /* ... */
Stencil OperatorStencil @finest {
    [ 0, 0] => (4.0 + kappa)
    [ 1, 0] => -1.0
    [-1, 0] => -1.0
    [ 0, 1] => -1.0
    [ 0,-1] => -1.0
}
```

The snippet shows a specification of a stencil on the finest grid. It gives a weight for each stencil point using relative position in two dimensions.

3.5.3.2. Optimization

Optimization strategy of the ExaStencils is to generate correct code version where further automatic optimization steps can be then applied [KL15]. This helps keep the correctness of the code while searching for a suitable set of optimization steps. ExaStencils code generator applies polyhedral and traditional optimizations. The polyhedral representation is extracted from the lower abstraction level of ExaSlang. Dependencies within the polyhedron are then analyzed. Parallelization and tiling optimizations are done based on analyzing the polyhedron. Kernel merging is possible through merging polyhedra of kernels. Other traditional optimization, e.g, vectorization and unrolling, are applicable besides to the polyhedral optimization.

At the lowest level of ExaSlang also memory layout optimizations are applied. In ExaSlang 4 level, the field accesses do not state the indices. By default, the indices are mapped to the loop iteration vector. However, ExaStencils provides an extension to the ExaSlang 4 to allow more complex memory layout transformations [KKKL18].

Final code is generated from the ExaSlang 4. It is normally C++ with OpenMP/CUDA/MPI.

The following snippet Listing 3.8 illustrates an ExaSlang optimization strategy.

Listing 3.8: **A sample ExaSlang strategy [SKH+14]**

```
var s = DefaultStrategy("example strategy")

// rename a certain function
s += Transformation("rename fct", { case x :
    FunctionStatement if(x.Name == "foo")
    => x.name = "bar"; x})

// evaluate additions
s += Transformation("eval adds", { case
    AdditionExpression(left :
    IntegerConstant, right : IntegerConstant
    ) => IntegerConstant(left + right) })

s.apply // execute transformations sequentially
```

When this strategy specification is applied it renames a function name and evaluates constant integer addition.

3.5.4. Stella and GridTools

Stella [GFO+14] is a library that offers an embedded DSL in C++ to support the development of stencil computations on structured grids. Template meta-programming is used to embed the DSL within C++. The Stella stencil code is translated into architecture-optimized code. The concepts from Stella were later reused in the development of the GridTools library [BCF+22]. Development of the GridTools library is continuing to support performance portability for stencil computations.

3.5.4.1. Memory Access

The GridTools library stores field data within multi-dimensional array-like objects, and provides access to the data via indices, i.e., (i,j,k). Data storage depends on the architecture, therefore, the user needs to specify the backend. When targeting GPUs, the backend is selected through the following statement

```
1 using backend_t = backend::cuda;
```

Alternatively, to target CPUs, the following statement is used

```
1 using backend_t = backend::mc;
```

The chosen backend is passed along with other information, e.g., field dimensions and data type, to let GridTools know what storage should be allocated for a field. The choice of the backend leads to decide the layout of the field data. After the storage of a field is defined, views can be created to read field data or write them. Also, synchronization can be called to guarantee up-to-date data values among views and data storage. The views facilitate the data access for CPUs when GPU backends are used.

3.5.4.2. Problem Coding and Optimization

The user provides the stencil update code through a functor. The loop logic is specified by the user through the DSL. Architecture independence is guaranteed through the design of the DSL as an abstract hardware model. The high-level code leads to generate code for multi-core processors with OpenMP, and for GPUs with CUDA.

The following snippet Listing 3.9 illustrates the definition of a GridTools operator.

Listing 3.9: **A sample showing GridTools stencil operator [BCF+22]**

```
1 struct lap_function {
2   typedef inout_accessor<0> out;
3   typedef in_accessor<1, extent<-1,1,-1,1> > in;
4   typedef arg_list = make_arg_list<out, in>;
5
6   template <typename Evaluator>
```

```
7   static void Do(Evaluator& eval) {
8     eval(out()) = eval(4*in() -
9                        in( 1, 0, 0) + in( 0, 1, 0) +
10                       in(-1, 0, 0) + in( 0,-1, 0));
11  }
12 };
```

The code shows the definition of the Laplacian operator. An output field is updated based on an expression that uses special indices to access field data. Features of C++ templates are used to enable access to neighbor elements.

Stencil specification: Users specify the stencil update formula through a functor. To allow this, the GridTools library uses a special method called 'apply' within a functor. The code within this 'apply' method allows GridTools to optimize the stencil update code at compile time. Besides the 'apply' method, 'accessor's are used within a functor to allow referring to the input and output fields that are involved in the computation of the stencil. Input accessors use 'extent's to specify the neighboring grid points that are used to compute the stencil. A stencil is specified using the accessors. The GridTools library provides the 'apply' method a context object to map the accessors to the data storage of the fields. This context object is called when using an accessor within the 'apply' method.

Problem domain and computation: The functor defines how to update a field based on input fields, however, additional code should be written to define and traverse the grid, and to apply the stencils at grid points that should be updated. Dimensions of the grid are described through special objects to specify the computational region and the boundary regions. Those objects are used to define an object that describes the grid.

The whole stencil computation is defined with the grid object and one or more 'multi_-stage' objects. The call to the stencils within 'multi_stage's depends on the dependencies among the stencils. Stencils with dependencies are added to different 'multi_stage's.

Stages: Stencil computations are defined through stencil stages. Stencil updates are applied within the stages of the computation. This improves the use of the data locality.

The user defines the functors that will be used for the updates. Those functors are used to define the stages of a computation. The DSL allows the user to define how the stages form the computation, i.e, the order and dependencies.

The following snippet Listing 3.10 demonstrates the specification of the stages that form a GridTools computation.

Listing 3.10: **A sample showing GridTools stages [BCF+22]**

```
1 auto horizontal_diffusion =
2   make_computation<BACKEND> (
3     make_multistage(
4       execute<forward>,
5       make_stage<lap_f>(lap(),  in()),
```

```
6         make_independent(
7           make_stage<flx_f>(flx(),in(),lap()),
8           make_stage<fly_f>(fly(),in(),lap())),
9         make_stage<comb_f>(out(),in(),flx(),fly())),
10      data_fields,  coords);
11
12  horizontal_diffusion->run();
```

The snippet shows a computation (horizontal diffusion) defined in multiple stages, where one stage is done, then two independent stages are executed, then a final stage is done after the two independent stages are finished.

3.5.5. Claw

Claw [CFF+18] is a code annotation technique to support performance portability of atmospheric modeling codes. The modeling code is written in Fortran by scientists based on single column abstractions. Claw directives are added to the code to guide applying the single column abstraction code horizontally. Other loops transformations are also guided with Claw directives.

3.5.5.1. Problem Coding and Optimization

The user annotates the single column code with 'parallelize' directives. Accompanying 'define dimension' directives guide the parallelization procedure to apply the column code repeatedly in the horizontal dimensions of the grid.

Along with the 'parallelize' directives, the user can add 'data' directives. Those directives guide the code transformations that will handle the movement of the data that is necessary to run the kernels.

Besides to parallelization of single column abstraction codes, Claw provides other low-level directives to guide the transformation of the loops. Directives for loop fusions and loop interchanges allow the user to ask the Claw compiler to apply those loop transformations in order to improve the use of data locality. Other low-level directives are provided to apply different transformations.

The following snippet Listing 3.11 demonstrates using CLAW to annotate code.

Listing 3.11: **A sample showing CLAW code [Cla]**

```
1  SUBROUTINE lw_solver(ngpt, nlay, tau, ...)
2    !$claw define dimension icol(1:ncol) &
3    !$claw parallelize
4    DO igpt = 1, ngpt
5      DO ilev = 1, nlay
6        tau_loc(ilev) = max(tau(ilev,igpt) ...
7        trans(ilev) = exp(-tau_loc(ilev))
8      END DO
9      DO ilev = nlay, 1, -1
10       radn_dn(ilev,igpt) = trans(ilev) *
11       radn_dn(ilev+1,igpt) + ...
```

```
12      END DO
13      DO ilev = 2, nlay + 1
14        radn_up(ilev,igpt) = trans(ilev-1) *
15        radn_up(ilev-1,igpt) + ...
16      END DO
17    END DO
18    radn_up(:,:) = 2._wp * pi * quad_wt *
19    radn_up(:,:)
20    radn_dn(:,:) = 2._wp * pi * quad_wt *
21    radn_dn(:,:)
22  END SUBROUTINE lw_solver
```

The snippet illustrates using dimension definition and parallelization directives. Those directives mark the loops that apply operations over columns to be applied in parallel for the set of columns in a problem domain.

3.5.5.2. Code Processing

Claw's compiler uses the information provided by the Claw directives to generate the output directives for the target architecture. It generates OpenMP directives to parallelize code on multi-core and many-core processors, and OpenACC directives for GPUs.

Claw's compiler is a source to source translation tool that uses the Omni compiler's Fortran front-end and back-end. It includes transformation objects that analyze the code for the applicability of a transformation, and then apply it if applicable. The Claw directives initiate the transformation procedures. Transformation procedures use the information about the target architecture and the output directives to apply the transformation.

3.5.6. Hybrid Fortran

Hybrid Fortran [MA17] was developed to support the ASUCA weather prediction model. It allows porting Fortran applications that were developed for CPUs to GPUs. It supports porting applications that works with structured grids.

3.5.6.1. Problem Coding

The goal of developing Hybrid Fortran was to allow the same existing source code of ASUCA, which was developed for CPUs, to be used for both CPU and GPU based machines. The Hybrid-Fortran-based code is intended to provide performance portability. An important requirement during the development of Hybrid Fortran was to keep the existing source code of ASUCA and to require the least possible code changes. This lead to develop a solution that supports Fortran code, in order to support the existing ASUCA model. Therefore, to provide the possibility to have a unified source code and to provide performance portability, Hybrid Fortran used directives to mark existing source code. The existing source code is used and marked with additional information that drives a transpiler to provide performance portability over CPUs and GPUs. The

transpiler translates the Fortran code that is marked with Hybrid Fortran markers into Fortran with OpenMP to support CPUs, or CUDA code to support GPUs. OpenACC is also supported as a back-end to support GPUs.

Hybrid Fortran makes use of the ideas of both stencil DSLs and directive based approach. To fulfill the requirements to keep existing source code, the kernel bodies are kept. However, the loop structure is abstracted and provided within the directives.

3.5.6.2. Optimization

Hybrid Fortran provides users special directives to guide the optimization process. Through directives, users control the data layout and the parallelization.

Data layout: One aspect to provide performance portability that Hybrid Fortran supports is the adaptation of the memory layout of the fields' data into memory. To support different memory layouts Hybrid Fortran allows the user through the directives to specify the index order that is intended to be applied within the final code. The Hybrid Fortran transpiler then generates macro wrappings to reorder array specifications and accesses according to the user provided directives. The index order of the original source code does not need to be changed. Instead, the indices are interchanged as specified through the directives. As a separate step, after the code is processed by the transpiler, the macros that Hybrid Fortran generates to handle index order are processed by the compiler to use the index order that the user provided through the directives.

Parallelization and granularity: Another optimization aspect that Hybrid Fortran applies is the parallelization. To support the parallelization of the kernels Hybrid Fortran allows the user to specify through the directives the target architecture- whether the code will be run on CPUs or GPUs. This information guides the transpiler through the code translation process to parallelize the code on the suitable granularity. The parallelization with a higher granularity is used when the user marks the code to run on CPUs, and a lower granularity is used for GPUs. The higher parallelization granularity on CPUs allows to use the cache better. However, the lower parallelization granularity on GPUs is necessary to deal with calls within the parallel regions. Along with the parallelization granularity guidance, the user provides further information to guide the transpiler to control the scope of the data structures used within the kernels. This allows the transpiler to guarantee the access of the threads or the GPU's streaming multiprocessors to the data that they need for their computations.

To optimize the data movement on GPUs, which is a critical issue for performance, Hybrid Fortran allows the user to tell the transpiler through the directives whether the data is present in the GPU memory or if it should be moved within the kernel.

The following snippet Listing 3.12 demonstrates the coarse-grained parallelization with HybridFortran directives.

Listing 3.12: **Coarse grained parallelization of physical processes [JKM+17]**

```
1 subroutine run_physics
```

```
2     use example_data_module, only: a, sum_a
3
4     real, intent(in), dimension(NZ, NX, NY) :: a
5     real, intent(out), dimension(NX, NY) :: sum_a
6
7     @domainDependant{attribute(autoDom, present)}
8     a, sum_a
9     @end domainDependant
10
11    @parallelRegion{appliesTo(CPU), domName(i,j), domSize(NX,NY)}
12
13    call sum_column(a, sum_a)
14    ! .. more calls to deep graphs of subroutines
15
16    @end parallelRegion
17 end subroutine
```

The code uses the *appliesTo*(*CPU*) within the *parallelRegion* to parallelize coarse-grained physics on processor cores.

The following snippet Listing 3.13 demonstrates the fine-grained parallelization with HybridFortran directives.

Listing 3.13: **Fine grained parallelization of physical processes [JKM+17]**

```
1 subroutine sum_column(a, sum_a)
2     real, intent(in), dimension(NZ) :: a
3     real, intent(out) :: sum_a
4     integer(4) :: k
5
6     @domainDependant(attribute(autoDom, present), domName(i,j), domSize(NX,NY)}
7     a, sum_a
8     @end domainDependant
9
10    @parallelRegion{appliesTo(GPU), domName(i,j), domSize(NX,NY)}
11
12    sum_a = 0.0
13    do k=1, NZ
14        sum_a = sum_a + a(k)
15    end do
16
17    @end parallelRegion
18 end subroutine
```

The code uses the *appliesTo*(*GPU*) within the *parallelRegion* to parallelize fine-grained physics on GPU threads.

Hybrid Fortran does not need to change the internal code of the kernels, however, the directives allow the transpiler to parallelize the code with the granularity that fits the target architecture. The same applies for the memory layout, where the original memory layout in the source code is kept in the source code, and changed in the generated code to fit the target architecture.

3.5.7. Atlas

Atlas [DBD+17][1] is a software library that was developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) to support numerical weather prediction. It supports the necessary data structures for modeling over massively parallel configurations. It is supposed to provide the necessary framework for NWP and climate modeling exascale high-performance simulations on heterogeneous hardware environments.

3.5.7.1. Problem Coding and Optimization

Atlas is developed originally in the C++ language. The library is implemented with object oriented features, hence classes represent the different objects within the library. However, it can work with C language, and supports GPUs with CUDA. Programming with Fortran is also possible through C-Fortran bindings. Fortran derived types are used to mirror the C++ classes through C, and subroutines within those derived types delegate the calls to the member functions of the C++ classes.

Grids and meshes: Atlas provides the data structures that are necessary for the simulations over the whole earth surface or parts of it. Both structured and unstructured grids are supported in Atlas. It provides the data structures that are necessary to represent the grids, the meshes which define the connectivity within the defined grids, the fields which have values defined over those grids, and the function spaces that define the discretization spaces of of the fields on the grids.

Grids in Atlas are the sets of points underlying the modeling discretization. Grid points have spatial coordinates that can be fetched without the need for connectivity information. Within some computations, the field values are fetched depending on the order imposed by the grid definition alone without the need for meshes when connectivity information are not necessary. However, for more complex computation where the connectivity information is necessary, meshes need to be defined over the grids. A mesh object in Atlas defines the relationships between the grid points and the resulting edges and nodes.

Atlas provides a variety of grid types, for both global and regional modeling, in order to support different NWP and climate models. Regular, unstructured, and reduced grids can be used with Atlas. Objects that carry information about the required grid are passed to the library to create a grid of a specific type. Other classes help dealing with grids like projection objects which map grids to geographical coordinates. To support regional models, 'domain' objects are used in Atlas to limit the models to limited regions. Grid classes are organized by polymorphism to deal with grids through a grid class while providing support for different types of grids.

Parallelization: Partitioner objects partition the grids and create 'distribution' objects that define to which partition each grid point belongs. Different partitioner types

[1] Notice that this solution is different from the previously mentioned Atlas (Automatically Tuned Linear Algebra Software [WPD01]

are provided; checkerboard partitioners, equal-region partitioner, and matching-mesh partitioner. Besides to the function of providing connectivity information on the grids, the mesh objects allow the decomposition of the simulation domains over multiple nodes and allows the execution of the resulting partitions on different MPI processes. Halo exchange is identified based on the meshes to allow executing stencil computations with multiple node configurations.

Meshes can be read from files by mesh-reader objects, or generated for grids by mesh-generator objects. Different types of connectivity are supported in Atlas meshes: block connectivity, irregular connectivity, and multi-block connectivity. The choice of the connectivity depends on the structure of the grids within the model, however, the performance varies according to the connectivity type.

As the grid objects do not contain any information about domain decomposition, the parallelization over the MPI processes can be done by generating the mesh and partitioning on one MPI process and distributing this information to the other processes. Another way that Atlas implements to parallelize structured grids over multiple MPI processes is to use parallel mesh generation. To handle halo regions, besides to the global indices of the elements, the partition number and element index on the remote partition are stored locally. Gather-scatter operations and halo exchange are supported over the different MPI processes.

Fields: Fields in Atlas are containers for the values that a variable carries over the discretized space. The spatial discretization with respect to the grid structure is defined through the function space. Thus, the function space implements the communication of the halo to synchronize the variables data between the different MPI tasks on the different nodes in multiple node configurations. Atlas provides a set of function space classes: node columns, edge columns, structured columns, and spectral function space. However, additional classes can be defined according to the object oriented paradigm that is adopted by the library. The parallelization is supported by 'gather scatter' and 'halo exchange' objects for node column and edge column function spaces. However, a 'trans' object is used to support the parallelization with structured columns and spectral function spaces.

The field object contains the variable values which could be either scalar, vector, or tensor fields. The storage of the field data is contiguous in memory, and can be mapped to some arbitrary indexing mechanism that defines the memory layout of it. The memory layout is defined by the associated function space that also makes the parallelization possible when accessing the field data. Beside to a field's data and the associated function space, Metadata, e.g, field name and unit, are also stored along with a field object.

Operators and computations: Atlas simplifies the use and definition of mathematical operations by providing implementations of some operations. To help the users to use the finite volume method, Atlas provides a class that supports writing the necessary operators based on the fields and the function spaces.

3.5.7.2. Targeting GPUs and Memory Access

GPUs are supported by the Atlas library besides to the CPU support. To handle the separate GPU memory from the host main memory, Atlas makes use of the a GridTools module as a storage layer besides to its native storage. This allows Atlas to make use of the functionality provided by GridTools to manage memory access on GPUs. To support Fortran operators porting to GPUs, Atlas uses OpenACC directives. The following Listing 3.14 is an example code to demonstrate the use of OpenACC to port the kernel to GPUs

Listing 3.14: **A sample code from Atlas using OpenACC [Dec19]**

```
type ( atlas_Field ) :: field1
type ( atlas_Field ) :: field2
real (8) , pointer :: v1 (: ,:)
real (8) , pointer :: v2 (: ,:)

field1 = atlas_Field ( kind = atlas_real (8) , shape =[ n , n ])
field2 = atlas_Field ( kind = atlas_real (8) , shape =[ n , n ])

call field1 % clone_to_device ()
call field2 % clone_to_device ()

call field % device_data ( v1 )
call field % host_data ( v2 )

! acc data present ( v1 ) copyin ( v2 )
! $acc kernels
do j =1 , n
  do i =1 , n
    v1 (i , j ) = v2 (i , j ) + 42.
  enddo
enddo
! $acc end kernels
! $acc end data
```

The snippet shows the use of the OpenACC directives to move the data to the device memory on the GPU and to run the kernel on the GPU.

3.5.8. PSyclone

PSyclone is a code generator that is being developed (gets operational 2022) for the use in finite element, finite difference, and finite volume applications. It is developed to support the finite element dynamical core in the GungHo project [FGH+13], which includes the development of the next generation software of the MetOffice, which should scale up on machines that have millions of cores. The development also spans two benchmarks that use finite difference method in ocean modeling. Different APIs are developed to support the differences within the different applications; the dynamical core and the benchmarks.

3.5.8.1. Levels of Abstraction

The separation of concerns in the development of the dynamical core is a main point in PSyclone. The software development is done with three layers: the algorithm layer, the parallelization system layer, and the kernel layer. The algorithm layer represents the code that applies the scientific solutions over the problem domain by calling the kernels and the infrastructure routines. The parallelization layer (PSy) allows the execution of the code on the processor resources that exist on one node in parallel. Those resources could be multi-core, many core, and GPU processing resources. The parallelization layer can be optimized to any type of those architectures or some combinations of them. The kernel layer provides the implementation of the code kernels as subroutines that operate on local fields. The parallelization layer does need need to be manually written by the user, as a code generator is developed within the GungHo project. This code generator helps the user optimize the parallelization code, or even generates this layer's code automatically.

Kernel layer: A kernel code within the Kernel layer is defined by a subroutine that applies to a set of elements in the problem domain, e.g, a column. The code within such a subroutine should be serial. The kernel developers should provide some kernel metadata along with each kernel to allow PSyclone to generate the optimal parallelization code for the kernel. As this metadata is necessary to guide PSyclone for the code generation of the PSy code, the developer can neglect writing this kernel metadata when writing the PSy layer code manually. The format of the metadata is designed to fit the API that the kernel uses, so the kernels of the dynamical core have different metadata format of that of ocean benchmarks kernels. The following code Listing 3.15 demonstrates the concept of the kernel and the metadata

Listing 3.15: **A sample PSyclone code**

```
1  module integrate_one_module
2    use kernel_mod
3    implicit none
4    private
5    public integrate_one_kernel
6    public integrate_one_code
7    type, extends(kernel_type) :: integrate_one_kernel
8      type(arg) :: meta_args(2) = (/&
9      arg(READ, (CG(1)*CG(1))**3, FE), &
10     arg(SUM, R, FE)/)
11     integer :: ITERATES_OVER = CELLS
12     contains
13     procedure, nopass :: code => integrate_one_code
14   end type integrate_one_kernel
15   contains
16   subroutine integrate_one_code(layers, p1dofm, X, R)
17     integer, intent(in) :: layers
18     integer, intent(in) :: p1dofm(6)
19     real(dp), intent(in) :: X(3,*)
```

```
20      real(dp), intent(inout) :: R
21    end subroutine integrate_one_code
22 end module integrate_one_module
```

Algorithm layer: In the algorithm layer the scientists specify what should be executed. This is done through calls to the kernel-layer kernels through calling 'invoke' with the necessary kernel information, i.e, the metadata type name that refers to the kernel, and call arguments. Parallelization also is not allowed for the code of the algorithm layer. Multiple kernels can be called within one 'invoke' call. An 'invoke' call with multiple kernels allows the PSy layer to optimize the calls to the called kernels. The 'invoke' calls are translated into calls to the kernel-layer kernels in a way that aligns with the PSy layer. When the PSy layer is written manually, the 'invoke' calls are replaced with manually written kernel-layer kernel calls.

PSy layer: The PSy layer includes the code that links the algorithm layer with the kernels that it calls in the kernel layer. The PSy includes codes that apply the kernels, which apply to parts of the problem space, to the whole problem space. It handles the arguments passed within the 'invoke' calls to fit the real generated kernel calls. The necessary distributed memory operations, e.g, halo exchange, are also carried out within the generated PSy layer code. The PSy layer also allows an optimization expert to make optimizations like distributed memory operations, shared memory optimization and single node optimization.

The PSyclone tool depends on the order of the kernels within an 'invoke' call to generate the right calls. It analyzes the arguments passed to each kernel. The matching of the types of the arguments passed within the algorithm layer with the expected arguments that a kernel-layer kernel should receive is essential for the generation of the PSy code. The provided metadata which is written along the code helps the PSyclone tool to no further information that is necessary to generate the PSy-layer code. The parsing of the metadata is done in a special API-specific parsing stage within PSyclone.

3.5.8.2. Code Generation

The PSy-layer code generation functionality is organized in a hierarchy of objects. The top level of this hierarchy is the 'PSy' object, which generates the PSy layer code for one algorithm layer file. The PSy object depends on input from the parser, which provides invoke information through special objects. Under the 'PSy' comes the 'invokes' object, which includes a set of 'invoke' objects, each of which represents one invoke. Each 'invoke' object includes a 'schedule' object, which is a tree of objects that represent the parts of the code that will be generated for the PSy layer. An initial schedule object is generated initially without optimization, however, transformations on the structure of a schedule object allow to optimize the code for the different architectures. The transformations include inlining calls for kernels, loop fusions, and OpenMP parallelization. The user can apply the transformations either interactively or by writing a Python script.

Communication: To support distributed memory, PSyclone generates the necessary communication code within the PSy layer code. The generation of the halo exchange code is done automatically by the tool. PSyclone adds the necessary halo exchange objects to the 'schedule' objects. Modifying the schedule object then allows to apply some optimizations like overlapping communication with computation. This technique of adding halo exchange objects to a schedule object allows to handle communication based on invoke calls, however, to keep track of any needed communication outside the invoke calls PSyclone uses flags to check when the data should be updated.

3.6. Gap Analysis

So far, we have discussed the challenges facing model developers, and provided in this chapter a review of the evolution of the solutions. We discussed the rise of performance challenge as a result of need to performance for models. Performance portability was also discussed as it appeared as a result of the architectural diversity. Besides, the impact on code quality was also discussed. We looked at details of DSLs and solutions to understand concepts and approaches that have been researched so far.

To clear the scope of research we are doing, we define a set of criteria to describe what has been done so far, and what opportunities can be exploited:

- **Data layout:** This criterion is defined by the level of flexibility and simplicity to control data placement.

  Data layout, which is an important factor in achieving optimal performance, is taken into account in existing solutions. The choice among a set of alternative index orders or a set of target architectures allows to switch between a set of alternative layouts.

  Opportunity: Existing techniques can be replaced by a more flexible technique that allows to control the placement of data elements in memory through formulae that represent mathematical transformations. Faster and less-costly exploration of layouts and achievable performance (without the pain of code rewriting) serves porting code –with performance– to newly-introduced architectures and features.

  Exploiting this possibility contributes to our objective of overcoming performance and performance portability challenges.

- **Performance portability:** This criterion is defined by the effort to support newly-introduced architectures and features.

  Performance portability is already considered in existing solutions, e.g., many solutions already support executing kernels on GPUs. However, solutions run a set of optimization procedures to target a specific target among a few architectures that a solution supports. Mostly, solutions support multi-core processors and GPUs.

  Opportunity: If optimization procedures can be controlled through a configuration information per machine, machine features and architectural improvements

(which are introduced frequently in comparison to code lifetime) can be exploited with configuration changes rather than re-engineering DSLs and their tools. Not only source code does not need rewriting, but also DSLs and tools.

Exploiting this possibility contributes to our objective of overcoming performance and performance portability challenges.

- **Multi-node scaling:** This criterion is defined by the effort to scale code to multiple nodes and the flexibility to support communication technology changes.

  Scaling code to multiple nodes is supported in some solutions, where MPI is used to communicate halo data. Other solutions do optimization for a single node, while the user needs to care about scaling code to multiple nodes.

  Opportunity: Scaling could be done with tools where needed information to drive the scaling are based on semantics extracted from source code. Code in this case would be unaware of single vs. multiple node runs, hence, no source code rewriting would be needed. Furthermore, configurable code scaling process allows to replace and fit communication code to underlying machine, e.g., use an alternative communication library, even without the need to change the DSL tools.

  Exploiting this possibility contributes to our objective of overcoming performance and performance portability (scaling to multi-node machines) challenges.

- **Code integration:** This criterion is defined by the effort to fit stencils within other code.

  In existing solutions, a code module or code to specify stencil computations are written and processed by tools, and other modeling code is written to interface to generated code. Alternatively, some solutions allow using model code, while DSL is applied through marking code with directives.

  Opportunity: The code could be written completely with one sequence of ideas, where a stencil is written within code as if it is completely written with GPL. This could be done using additional language constructs with lifted semantics. Neither separate modules are needed, nor separate translation of DSL modules and writing other code to use translation results and then compiling/linking code. Rather, code is treated as a (homogeneous) program that is written in one language.

  Exploiting this possibility contributes to our objective of overcoming code quality challenges.

- **Application-specific constructs:** This criterion is defined by possibility to adapt language to application.

  Existing solutions provide domain-specific approaches, but do not provide flexible ways to exploit application-specific information.

  Opportunity: Exploiting application-specific information allows to maximize the information to optimize performance. Adapting language constructs simplifies

referring to stencil components, hence, the coding of stencils. Code reduction in terms of code size and reduced complexity of code maintainability are facilitated.

Exploiting this possibility contributes to our objective of overcoming code quality challenges, but also is important to overcome performance and performance portability challenges.

- **Optimization overhead in source code:**

  In existing solutions, some information regarding architecture and optimization is provided through source code or directives.

  Opportunity: A clean code could be developed to reflect scientific problems without hardware information. This improves code quality in all aspects including developers effort, development time, code maintainability, and code size.

  Exploiting this possibility contributes to our objective of overcoming code quality challenges.

This gap analysis is summarized in Table 3.1. The table shows the criteria, approaches in existing solutions, and the new features that we target adding.

To support incremental porting of existing models besides supporting development of new models, we investigate using mixed languages for model development. The general-purpose language that is used to implement an existing model is used besides to a set of language extensions. The same GPL code of the model is still used, while parts of it are rewritten partially with the language extensions. In an existing model that is developed with Fortran for example, this mixing allows replacing a nested 'do' loop, for example, by a DSL iterator. However, the same 'Fortran' body of the loop is still used. Within this 'Fortran' body, some minor changes are done like using DSL indices instead of normal 'Fortran' array indices. The iterator is still surrounded with the original code that was around the original Fortran nested 'do' loop before porting. When processing this mixed code by a tool, the higher-level semantics are then extracted from the source code and used by the tool to transform the code and apply different optimization procedures, including generating code to handle halo exchange. The set of language extensions that we investigate are defined by the user to support the application-specific needs besides to general domain knowledge.

To the best of our knowledge, the use of user-defined language extensions that the user adapts to the application-specific needs, and the transfer of the high-level semantics from such extensions to a user-controlled code transformation process to drive the optimization process has not been investigated before. Also, the automatic generation of code that handles halo exchange based on automatic identification of halo patterns through such semantics has also not been investigated.

Chapter summary

In this chapter we provided a literature review of related work. We reviewed the use of computers to enable numerical simulations and the move towards HPC. Techniques to

Table 3.1.: Existing solutions and new features

	Techniques used so far (with example solutions)	New features
Performance & performance portability challenges		
Data layout	A set of alternatives known to provide performance on supported architectures: e.g, HybridFortran allows to choose index order	• Flexible controlled data location • User-provided formulae
Performance portability	Support multiple targets: e.g, Kokkos, GridTools, CLAW, HybridFortran, Atlas, PSyclone support GPUs	Dynamic support for architectural changes
Multi-node scaling	• MPI support: e.g, Atlas, PSyclone • User coded: e.g, Kokkos	• Unaware code, tool applied • Dynamic to support alternative communication infrastructure
Code quality challenges		
Code integration	• DSL code modules: e.g, GridTools stencil computation specifications • In-source dirctives: e.g, CLAW	• Lift semantics using new constructs • Stencils inlined within host language
Application-specific constructs	Not supported	Maximize semantic value per application
Optimization overhead in source code	Guided optimization: e.g, CLAW directives, stencil precedence in GridTools	Clean source code: scientific problem not optimization

optimize code beyond manual efforts were then discussed. High-level coding was given further emphasis as it is related to the techniques discussed in this thesis. Two early related DSLs were considered with more detail to understand ESM and stencil-computation DSLs concepts better. Details regarding the state of art of problems and solutions are covered. We finalized the chapter with a gap analysis.

In the next chapter, we present the design and the the methodology that we use throughout this work.

4. Design and Methodology

In this chapter, we describe the high-level design of the model development using our techniques, and the methodology we use to answer the questions of the thesis. We start Section 4.1 with a quick introduction to the design. Then we discuss some important concepts and principles related to the approach. Next, we discuss how the design allows to overcome the challenges. Finally, we describe the methodology that we follow to answer the questions of this thesis in Section 4.2. A high-level description is provided first on the phases of the work, then each phase is discussed in more detail.

4.1. Design

Application development under the techniques we investigate starts with the scientific problem and the mathematical methods that are used to solve it numerically (see Figure 4.1). Models apply (many) stencils to compute various fields numerically. The choice of the mathematical method leads to decide applicable grids and then the stencils to apply to execute the simulations. Stencils include a specific set of grid points with respect to the point being computed. The analysis of this set of points in the stencils allows to identify the spatial relationships that an application needs. Spatial relationships and sets of grid points used in an application are used to define a set of language extensions that will be used to develop the application.

Beside to application-specific needs, we consider a set of domain-specific concepts, which are not specific to the application. Those concepts represent an important basis for the development of the language extensions. Iterators, for example, traverse a set of grid points, whatever grid is being used. Thus an iterator statement is generally applicable to different applications in the domain.

To allow tools to process the language extensions, configurations are used. Language extensions are a main input to write configurations, but also they include information regarding sets of grid points that are used in an application. To transform source code into a target-optimized code, the architectural features of a target machine are considered when writing configurations.

Our high-level design separates the scientific activities (blue-shaded group on left hand side in the figure) from the optimization activities (red-shaded group on right hand side). The scientific problem behind an application allows to shape the language extensions and use them to write the source code. The features of the hardware are used by scientific programmers beside to language extensions and grid point sets to write configurations. Translation tools are then used to apply the knowledge from the configuration files to transform the source code into target-optimized code.

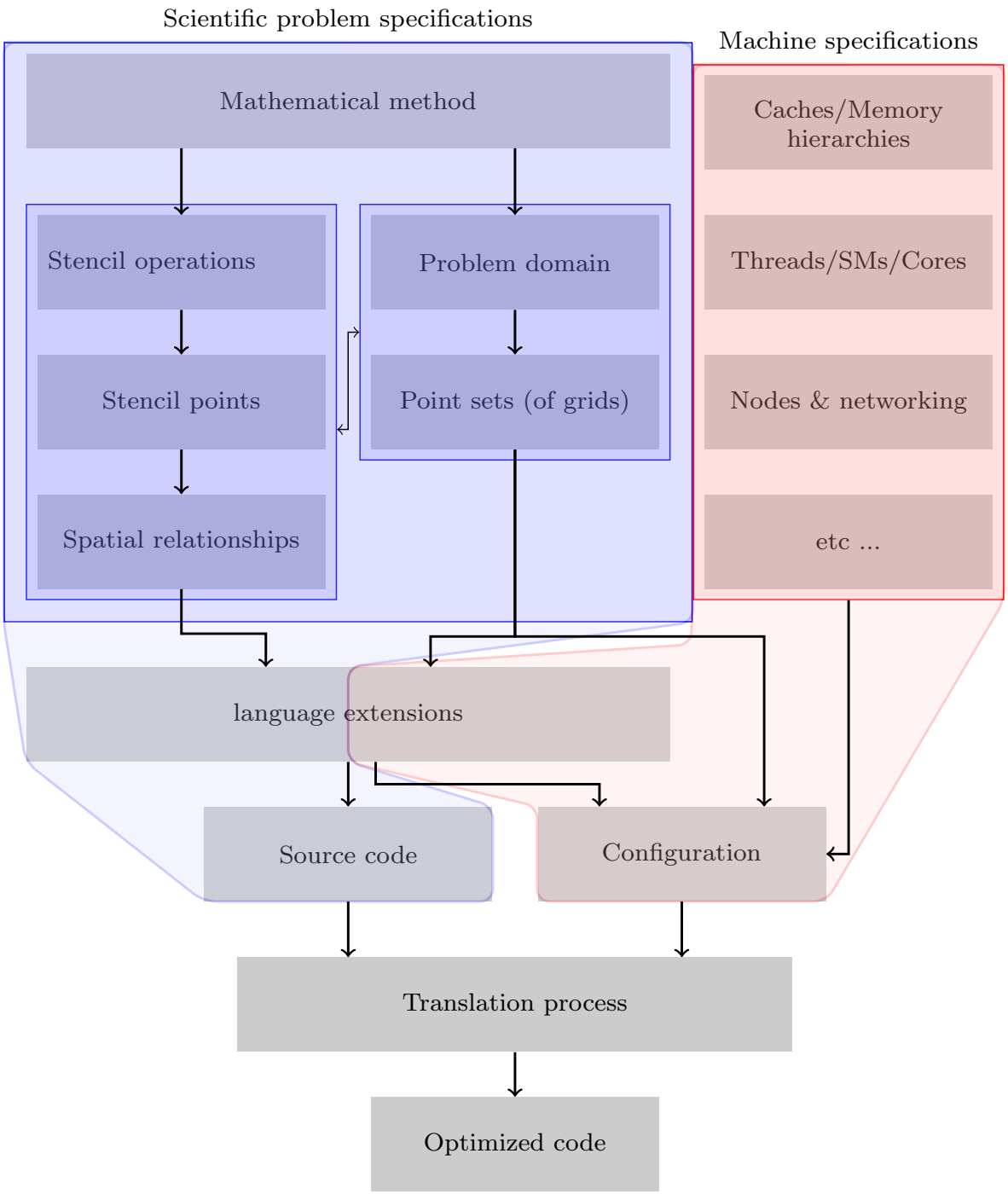


Figure 4.1.: Language extensions development and use with configurations

4.1.1. Key Concepts and Principles

Our suggested approach considers the shortcomings of modeling with conventional methods. It is developed to overcome the coding obstacles that affect the quality and the performance portability of code. A set of principles are adopted within this approach. In this section we discuss those principles and the key concepts that we consider to provide our solution.

Domain-specific abstraction and improved semantics: Rather than thinking and coding with machine concepts, e.g., arrays and loops, we suggest using scientific concepts. In this work, we raise the abstraction level through providing language constructs that developers need to write stencil codes. In case of designing a language to support development for a specific domain of problems, the generality assumptions (that is considered when designing general-purpose languages) can be reduced. This allows us to define assumptions that fit the specific domain we deal with, which is stencil computations.

Separation of concerns: With non-machine semantics, code is developed with alternative (domain/scientific) language components. Those language components reflect scientific concepts, but do not impose any machine-specific details on how to execute the code on an underlying machine. This gives code processing tools the flexibility to implement the high-level code on target machines.

In such an approach, we separate the roles of problem coding from optimization. Scientists write the source code that solves a specific problem in terms of scientific concepts. Code optimization is applied during further code processing.

User-controlled optimization: To optimize source code that is written with high-level language components, tools are guided by users to optimize the code. Users provide rules for optimization that are then executed repeatedly throughout the code over the kernels that comprise a model. This is an important aspect that we offer in our approach to make our solution support the speed of the technology evolution in HPC architectures. It is also important to support the wide spectrum of application-specific needs, e.g. supporting different kinds of grids. Besides, allowing users to control the optimization and the language processing empowers the scientists to ship the code with the tools instead of relying on external tools, e.g., compiler infrastructures installed on particular machines.

Mixed-language programming: Designing and developing a new language with the necessary semantical features is not acceptable by scientists. Scientists tend to use specific modeling languages and prefer to stick to use those languages for modeling. Therefore, the optimal approach is to introduce language components with the necessary semantical features, which can be used besides to the modeling language that scientists prefer. This mixed language approach allows scientists to use features of their main

modeling language, but use the additional components when needed within the same code. This reduces the effort to learn a new language and code porting.

Memory-Oblivious Data Access (MODA): We introduce Memory-Oblivious Data Access (MODA) technique to replace local-memory-bound explicit data access. Our translation techniques extract the necessary semantics from the source code to automatically generate code to handle scaling of stencil computations on multiple nodes, in addition to shared-memory parallelization and node-level optimization. This allows to overcome fixed data layouts, which limits use of memory bandwidth across architectures. It also allows tools to track data location both across and on nodes instead of developers doing that within source code.

Alternative indices that are not aware of underlying hardware and memory are used in MODA. Spatial relationships are used to refer to grid points. Using this hardware-detached design, MODA allows using the same source code on different run configurations including shared and distributed memory, and different architectures. This allows our solution to support performance portability and fit different architectures.

To cope with different application needs, e.g., collocated vs. staggered, regular vs. icosahedral grids, triangular vs. hexagonal vs. rectangular cells, our approach allows users to define language extensions to specify indices. To demonstrate the concept with simple rectangular grids, instead of the array notation in Figure 1.1, we can replace the i and j indices with spatial relationships as shown in Figure 4.2. The names in the figure are just suggested names that users define according to the used grid, other names can be defined for other grid structures. Index adaptability to application needs allows to bypass data access problems in the source code, including memory layout and scalability.

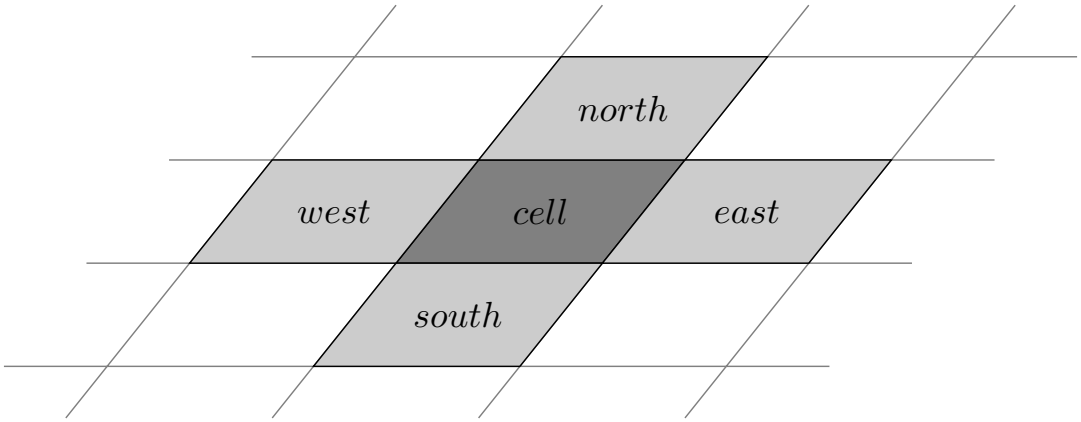


Figure 4.2.: Alternative MODA indices

4.1.2. Considering Challenges

In our design we consider clearly the challenges discussed in Section 1.4. Table 4.1 shows briefly how we suggest genuine concepts to serve that purpose.

4.1.2.1. Performance and performance portability challenges

The design cares for performance and performance portability challenges through considering the following aspects:

Data layouts: Committing to the principles discussed in the previous section, mainly the "Memory-Oblivious Data Access (MODA)", leads to adapt the data layouts to target architectures. We use the principle "Domain-specific abstraction and improved semantics" to replace memory-bound indices with scientific concepts. Conforming to the principle "User-controlled optimization", we use user-provided configuration information to transform the higher-level indices into array notation.

Performance portability: Committing to the principles "Domain-specific abstraction and improved semantics" and "Memory-Oblivious Data Access (MODA)" allows to write high-level code that is unaware of underlying architecture. The principle "Separation of concerns" enables the use of high-level coding, where the code is separated from the optimization process, which is performed according to the principle "User-controlled optimization". Under those principles, another party, i.e., scientific programmers, cares for optimization aspects while scientists write source code.

Multi-node scaling: The principles "Memory-Oblivious Data Access (MODA)" enables scaling high-level code to multiple nodes. Committing to this principle makes code unaware of multi-node parallelism, and allows tools along with the principle "Domain-specific abstraction and improved semantics" to identify necessary communication. Tools through conforming to the principle "User-controlled optimization" handle communication-related activities.

4.1.2.2. Code quality challenges

The design counts for code quality challenges through considering the following aspects:

Code integration: The principle "Mixed-language programming" enables the use of GPL together with DSL. Statements to apply stencils are written within the remainder of the code just as any other statements. This provides an advantage for use of existing model codes. Incremental porting of existing codes, which are written with GPL, is doable by replacing some GPL statements with DSL statements.

Application-specific constructs: The principle "Memory-Oblivious Data Access (MODA)" allows to define language extensions that conform to the principle "Domain-specific abstraction and improved semantics" which serve better the needs of a subject application. Such extensions fit the application and allow more direct coding. This improves the code quality besides semantical impact.

Optimization overhead in source code: The principle "Separation of concerns" enables clean coding. Under this principle, optimization is done through separate files rather that within source code.

4.2. Methodology

In order to answer the questions of this thesis we consider two important points in the questions:

- Using user-defined application-adaptable language extensions to develop stencil codes, mixed within some modeling general-purpose programming language
- Using a user-controlled translation procedure to transform high-level code into usable code that can be further processed by other tools, e.g. compilers

The points imply that high-level language extensions should be developed to extend the grammar of the general-purpose language, while still using that GPL. Translation procedures should be designed also to translate the mixed high-level code, which can not be processed by compilers in its original form. The specifications of the language extensions should be able to semantically provide some information to the translation procedure. The design of the translation procedure should use this semantical information in code transformation. Developing and analyzing the algorithms that use the semantical attributes of different extensions to transform code will answer the questions of the thesis. A high-level description of the work done in this thesis is shown in Figure 4.3.

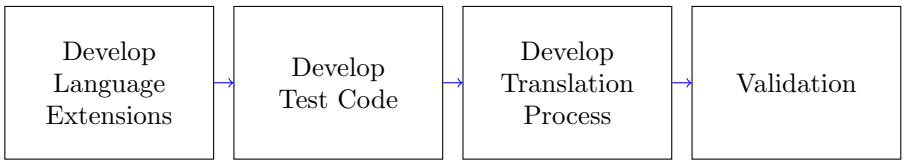


Figure 4.3.: Methodology

Each part of the methodology as described in the figure is further discussed in more detail in the following sections.

Performance & performance portability challenges	
Data layout	• Use scientific concepts to refer to data (spatial relationships) rather than explicit array notation • Transforming high-level data references into array notation is user-controlled • User-provided formulae provide a flexible way to control data location in memory
Performance portability	• Exactly one code version is written for a model, regardless of architecture-specific details • Translation process applies optimizations corresponding to a target architecture • User-provided information match application and hardware features to guide code translation • Dynamic support for the frequent architectural changes is realized through this configurability
Multi-node scaling	• Source code is unaware of single vs. multiple node runs • Use spatial relationships to identify necessary communication • Tools handle communication supported with user guidance • Support communication infrastructure changes
Code quality challenges	
Code integration	• Keep preferred modeling GPL • Lift semantics using new constructs (language extensions) • Stencils inlined within other code using mixed GPL & DSL • Incremental porting of existing code that uses GPL only
Application-specific constructs	• Consider application besides general domain concepts • Use extensions closer to application to maximize semantic value of the extensions per application • Use of application-specific constructs allows simpler (direct and short) coding of operations
Optimization overhead in source code	• Developers do not care about underlying hardware • Source code is unaware of underlying machine, no optimization pragmas or codes and no redundant per-machine codes

Table 4.1.: Design principles to overcome challenges

4.2.1. Language Extensions Development

An important feature of the language extensions in the assumed context is to provide the necessary semantics to drive the code transformation process to exploit hardware features, which differ from one architecture to another. This necessitates first to define a set of extensions that fit the purpose of answering the thesis questions. A high-level description of the activities to develop the language extensions is shown in Figure 4.4 followed with more details on the activities in the following text.

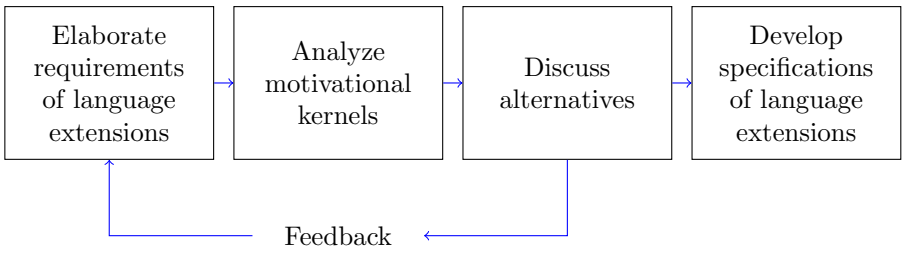


Figure 4.4.: Language extensions development

Elaborate requirements of language extensions: As mentioned in the questions, we assume that the language extensions should support reuse of existing GPL code. Besides, we assume that they are adaptable to support the needs of a subject application, which needs then to give the user the capability to define the language extensions. Those assumptions represent only a starting point towards eliciting the requirements to define the set of language extensions. However, we need to do more for the elaboration of effective requirements for the purpose of answering the questions of this thesis, and provide a beneficial solution for practical use.

Analyze motivational kernels: Motivating sample codes are analyzed to identify the representative coding complexities and optimization-critical parts. Different kernels including different stencils, cases, problems are analyzed. Kernels using different grid structures are included to make sure that our answers cover the adaptability aspects to support applications that use different grids.

Discuss alternatives: Based on the key factors that complicate the code development, as identified from the mentioned sample kernels, suggested alternatives are stated. Continuous discussions are a key to find an acceptable set of language extensions. Acceptance by the scientists who develop models is necessary to provide a useful set of extensions that can be used in practice for production purposes. Choosing an alternative that is acceptable by scientists while providing optimal semantical values helps achieving optimal performance and better programming experience. As a result of the discussions concerning evaluating the alternatives, a specification of a set of language extensions is developed.

Develop specifications of language extensions: The specification of the set of language extensions should fulfill the assumed requirements. This specification should define the necessary language keywords and constructs that allow specifying stencils. For the sake of providing comprehensive answers, the resulting specification should be sufficient to investigate the semantics transfer in terms of a set of optimizations. Besides, conveying the necessary semantics to identify the necessary communication and to generate the code to serve this purpose should also be considered.

4.2.2. Test Code Development

After the specification of a set of language extensions is developed, test code should be developed. A set of stencils are developed with the specified language extensions. The developed code demonstrates the use of the language extensions, besides testing/validation purposes. Different well-known operators are represented with the language extensions within the developed test code as well as a full application to demonstrate the concepts of using the language extensions for modeling.

4.2.3. Translation Process Development

After the language extensions are specified and test codes are prepared using the language extensions, the high-level code should be prepared to be processed by tools to be run on the target architectures. A source-to-source translation process is developed to handle this code preparation for the further processing by other tools, e.g. compilers. The activities to develop the translation process are shown in Figure 4.5 followed by text to discuss the details of those activities.

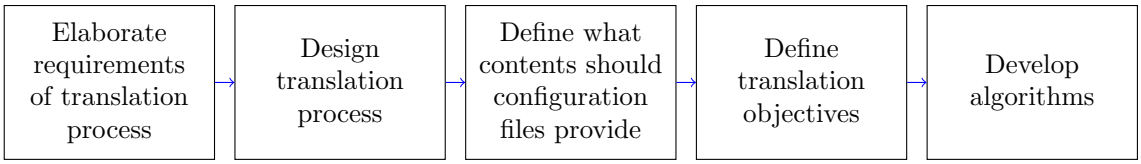


Figure 4.5.: Translation process development

Elaborate requirements of translation process: Requirements should be collected and analyzed to design a translation process that is sufficient to provide answers to the thesis questions. Collection of the requirements is necessary to identify a viable design of the translation process. This is essential as the translation process plays a key role within the total approach that we present.

Design translation process: Adaptability of the language extensions leads to allow users to define some extensions. This means that the user-defined language extensions behavior should be defined by the user. Providing users the flexibility to define the

language components complicates the code processing. Non-static grammars with changing language rules do not allow using usual tools, e.g. a parser, that provide built-in functionality to deal with the subject language grammar. Rather, we need to develop some alternative techniques to deal with the dynamical nature of language grammar.

To handle the dynamical nature of the grammar, we allow the users to define a set of extensions and configure the behavior of the defined extensions during the code processing. Thus, both the syntax and the behavior of the language extensions are defined together by the users.

When developers identify the needs of an application, the necessary language extensions that enable the code development are clear. Each of those extensions can then be defined; how it would be used in the source code, and what should the tool do when it faces an occurrence of that extension within source code. To provide this, we let the users specify all this information and provide it to a translation tool through a configuration file. A configuration file defines how the tool will translate the source code into a new version that targets a specific machine/configuration. Surely, the generated version will be further processed by other tools, e.g. a compiler, to be eventually run on the target machine.

Define what contents should configuration files provide: Designing the translation process includes specifying what should be provided through configuration files and what functionality should be provided by the tools. This defines the contents of the configuration files.

Define translation objectives: How the information that the user provides through the configuration files is used to transform the code during the translation process is a key point to answer the thesis questions. We start with a set of high-level representations of inputs from source code and a corresponding set of input configuration information. At the source code side, we develop a set of expected extensions for each rule that extends the grammar of the general-purpose language. At the configuration side, we develop a set of possible input configuration information. The product of the developed sets forms tuples that can be mapped to optimized target codes through transformations which represent different optimization aspects/procedures. Those mappings represent a higher representation of what the transformation does (mapping language extensions and configuration information to transformation objectives).

Develop algorithms: Developing algorithms to implement those mappings defines how the different transformations are applied. This set of algorithms answers how the extensions convey the necessary semantics to the translation process which uses user-provided configuration information to optimized code and handle communication.

4.2.4. Validation

After the language extensions have been developed and the test codes are prepared using those language extensions, and the translation techniques have been designed, we need to develop an implementation and validate the suggested concepts, techniques and algorithms. The activities to validate the developed techniques and concepts are shown in Figure 4.6 followed by details to discuss those activities.

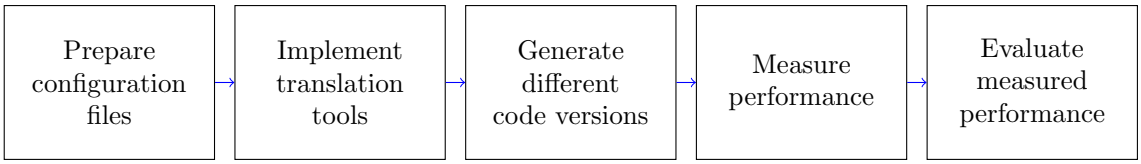


Figure 4.6.: Validation

Prepare configuration files: Based on the designed structure of the configuration files, we develop a set of configuration files. As mentioned, the algorithms map tuples of language extensions (within source code) and information from configuration files to optimization objectives, thus, using alternative information within configuration files allows to apply and investigate different optimization procedures/possibilities. Preparing the necessary configuration files allows to validate the various optimization procedures, under different architectures, and under different configurations, e.g., single vs. multiple nodes.

Implement translation tools: Using code with different stencils and complexities, along with different configuration inputs–including different architectures, we can generate different versions as target codes. To do that we implement the algorithms in a set of scripts that apply the designed source-to-source translation process.

Generate different code versions: Using the developed tools, we translate the test code. Different code versions are generated according to different transformation procedures corresponding to each configuration input. This allows to test the optimized code and validate the transformation/optimization procedures.

Measure performance: Next step is to validate the transformations. We check the generated code for the different expected optimization aspects. We run the different code versions on the corresponding hardware configurations and check the improvements that can be achieved with the different aspects of optimization, based on the different configuration inputs.

Evaluate measured performance: We support the measurements with collecting metrics through hardware performance counters or performance measurement tools and evaluating the measured performance according to expected performance under different optimization aspects. Comparison of measured to expected performance allows to check whether the optimization procedures are effective to generate efficient code on target hardware. This way we also evaluate the support of performance portability by our suggested approach.

The described methodology includes the development of a set of language extensions that fits the goals of this thesis. Developing codes using those language extensions proves in practice the usability of them, and allows to empirically examine and validate the approach. The introduced translation process adheres to the achievement of the thesis goals besides to the language extensions. Our methodology puts all together in an empirical approach to validate the solution, where the translation process is applied to translate test codes on real systems with different configurations. This allows to judge the performance and performance portability across different machines, architectures, and configurations of the solution besides to code quality improvements through evaluating developed test codes.

Chapter summary

In this chapter, we presented the design and the methodology that we use in this work. We discussed important concepts and principles considered in our design, and how we consider the challenges in the suggested design. We concluded the chapter with a discussion of the methodology of this work.

In the next chapter we discuss the development of the set of language extensions.

5. Extension Set Development

In this chapter, we discuss the development of the language extensions. First, the requirements are enlisted in Section 5.1 as a starting point to develop a representative extension set that will serve real applications and answer our research questions. Then, we discuss some development considerations regarding stencil computations through code samples in Section 5.2. These samples represent key code sections taken from different models, in order to allow the extension set to fulfill the needs of those applications. Finally, we present in Section 5.3 the set of language extensions that we use in this work.

5.1. Extension Development Constraints

In the first phase of the development of the language extensions a list of requirements that make the extensions generally useful for modeling were developed. In this section, the list of requirements are introduced, these are grouped into high-level requirements and requirements increasing the abstraction level.

5.1.1. High-Level Requirements

The list of the high-level requirements that guide the specification of the language extensions is:

RNCC [1] **Commitment to existing models' source code**

Starting the development of the DSL with existing models' code allows to develop language extensions that fit better the needs of those models. When this is done with a set of models, this allows to generalize the solution to the needs of the domain. With the principle of mixed-language programming (see page 63), the use of the GPL allows to keep most of the existing code of a subject model. Consequently, porting an existing model can be done incrementally.

RNCS **DSL code readability and maintainability**

Modeling code that is developed with the language extensions should be easy to read and understand by scientists. Scientists should be able to maintain their models with less efforts in comparison to the scenario of using a GPL for modeling. Abstraction based on scientific concepts and removing the machine abstractions

[1] Requirements are assigned short names, e.g., RNCC, for later reference in text

makes the language more direct to represent computations with concepts that scientists prefer.

RNWO **Write-Once Use-Many**

The language extensions should facilitate writing code once for the different target machines. Code should not include hardware details, so that rewriting an algorithm repeatedly for different platforms is not needed. In particular, there should not be any repeated "pattern" inside the code that could be extracted and abstracted further. This requirement achieves implicitly the following:

Single Code: Code should be written once, in one place, and not repeated in any other places for any other platforms.

Code Maintainability: Once code is written, it can be modified easily, one time, as it is written only in one place.

RNPP **Performance portability**

One of our goals is to enable performance-portable coding. Performance portable code uses efficiently the resources of underlying hardware on the different target architectures. The higher-level code design should support performance portability of applications, and the specification of the language extensions should count for that.

RNPR **Productivity**

The time that scientists would spend on model development, testing, and maintenance should be reduced in comparison to the scenario of using a GPL for modeling.

RNMC **Mixed coding within a host modeling language**

Instead of a new language with new grammar, new rules should be added to existing languages. New keywords can be introduced in addition to new kinds of statements and expressions that can be used in place of the counterparts from the host language.

RNEX **Language extensions adaptability**

The language extensions should include a basic set of extensions that reflect the common concepts of the domain science in general. But also, additional language extensions (or dialects) should be supported to address the specific needs of the applications.

5.1.2. Abstraction

Besides to the high-level requirements, the following requirements guide the abstraction within the language extensions:

RNDA **Domain abstraction**

During the DSL development, the concepts and operations of the domain science should be abstracted. Abstraction is a key for a successful DSL representation of a domain, and performance portability. The right abstractions make programming easy for scientists based on the rationale of their domain. It also allows them to focus on concepts and operations (algorithms), without being lost with the burden of hardware details and optimization.

For example, data representation instead of arrays is described in a level of abstraction that allows performance portability. They are described such that, after compilation, the real layout of data is chosen to be suitable for the target hardware.

RFSD **Allow specifying field dimensions**

Allow specifying the dimensionality of fields, at least whether a field is two or three dimensional.

RFSL **Allow specifying field localization with respect to tessellations**

Allow specifying the location of field data, at least whether a field is located at the cell centers, on the edges that separate the cells, or at the vertices at the crossings of the edges.

RNFC **Higher-level constraints/guidelines on defining field characteristics**

The language extensions provide guidelines on defining how field characteristics are specified. However, the actual set of extensions that developers use to specify field characteristics are not completely defined in the specifications. A basic set of extensions should be defined, with the possibility to be modified or expanded by users. Keep modification and expansion attainability in mind to support the adaptability to the application-specific needs.

RNDT **Data types flexibility**

Using the language extensions should be flexible enough to allow using the different necessary data types, including both primitive data types and data structures.

RFGT **Allow traversing specific sets of grid points to apply stencils**

New constructs should provide such functionality to allow writing kernels in higher-level code.

RFAN **Allow access to neighboring grid points regardless of used grid type**

While traversing a set of grid points within an operator, stencil needs access to neighbors. This should be supported whatever type of grid is used.

5.2. Motivational Coding Cases

In this section, we introduce a set of code examples taken from different earth system models. Those models have different computational approaches with different grid structures.

These cases are representative for challenging kernels and comprise a wide range of use cases and forms of stencils. This selection allows to establish a set of extensions that serves wider range of applications, but also examine our concept of application adaptability of the language extensions.

The three models are DYNAMICO [DDT+15], ICON [ZRRB15], and NICAM [STY+14].

- **The icosahedral hydrostatic dynamical core (DYNAMICO)**

  DYNAMICO uses an icosahedral grid with hexagonal cells. The diamonds of the icosahedron, which covers the problem domain, are divided into patches which allow distribution of the computation over multiple nodes. This division forms a semi-structured grid. One index is used to access the two-dimensional surface of the grid.

- **The ICOsahedral Non-hydrostatic modeling framework of DWD and MPI-M (ICON)**

  The grid used in ICON is an unstructured grid. Triangular tessellation is used to discretize the surface. Multiple processes comprise the model including atmosphere and ocean subsystems. Helper data structures are used to indirectly describe connectivity and allow access to neighboring grid points.

- **The Non-hydrostatic Icosahedral Atmospheric Model (NICAM)**

  NICAM is a global atmospheric model that uses an icosahedral grid. Recursive division of the edges of the icosahedron allow to gain fine resolution grids. An Arakawa-A grid is used then to localize variables at the vertices of the grid triangles. Hexagonal cells are generally formed by those triangular shapes. However, there are still 12 points resulting from the icosahedron structure that lead to pentagonal cells.

  The grid is divided into regions to allow execution on multiple nodes. Multiple regions can be assigned to a single process. The problem domain is defined over the three dimensional space with two indices; one to address the horizontal, and another for the vertical.

The following cases have been extracted:

- **Indirect addressing**

  This set of cases uses indirect addressing to access field data (see Page 17). Managing the access to grid cells, edges, and vertices manually is a tough mission for developers (scientists).

  – Accessing edges of a cell

  - Indirect addressing to access cells sharing an edge
  - Indirect addressing of vertical dimension
  - Referring indirectly to both vertices and cells around an edge
  - Stencils with direct neighboring (vertical) elements
  - Mixed dimensionalities
  - Different temporary data structures to optimize for different architectures

- **Offset-based addressing**

  This set of cases uses offsets along with array indices to access field data. This approach does not need additional lookup tables, however scientists still need to care for computing the locations of grid elements manually.

  - Addressing edges of a cell
  - Horizontal and vertical neighbors
  - Vector fields
  - Scalar Weight Variables
  - Vector Weight Variables
  - Region boundaries
  - Computing scalar fields in separate steps based on vector components

5.2.1. Indirect Addressing

The problems considered here use indirect addressing to refer to neighboring grid elements. Accessing the memory locations of data values is a typical access pattern that must be considered. Lookup tables are used to refer to neighboring grid elements. The following code examples execute parts of different operators and show how they use indirect accessing to read and write elements on cells, edges, and vertices.

5.2.1.1. Accessing Edges of a Cell

First, let's have a look at a sample code (Listing 5.1) from the main solver routine for nonhydrostatic dynamical core of ICON model. This code interpolates contravariant correction to cell centers based on values located at edges.

In this code, to access the data of the array *z_w_concorr_me*, indirect indices are used, where helper lookup arrays store the indices. Using indirect indices is common in unstructured grids (see Line 19 and the indices in the yellow box). Lookup arrays are used to refer to the edges of the grid cells, two arrays are used to store edge block numbers (*ieblk*) and edge numbers (*ieidx*) (The underlined indices within the yellow box). Besides to cell block number and cell number, those lookup arrays include a special dimension to refer to the number of the edge, where each cell has three edges. Therefore, the stencil includes a repeated access to same field, but on different (three) edges.

Listing 5.1: **Manual loop optimization and indirect indexing to access edges of a cell**

```
1  !$OMP DO PRIVATE(jb,i_startidx,i_endidx,jk,jc,z_w_concorr_mc)
       ↪ ICON_OMP_DEFAULT_SCHEDULE
2  DO jb = i_startblk, i_endblk
3
4    CALL get_indices_c(p_patch, jb, i_startblk, i_endblk, &
5                       i_startidx, i_endidx, rl_start, rl_end)
6
7   ! Interpolate contravariant correction to cell centers...
8  #ifdef __LOOP_EXCHANGE
9    DO jc = i_startidx, i_endidx
10 !DIR$ IVDEP
11   DO jk = nflatlev(jg), nlev
12 #else
13   DO jk = nflatlev(jg), nlev
14   DO jc = i_startidx, i_endidx
15 #endif
16
17       z_w_concorr_mc(jc,jk) =                                    &
18         p_int%e_bln_c_s(jc,1,jb) *                               &
19         z_w_concorr_me( ieidx(jc,jb,1) , jk, ieblk(jc,jb,1) ) + &
20         p_int%e_bln_c_s(jc,2,jb) *                               &
21         z_w_concorr_me(ieidx(jc,jb,2),jk,ieblk(jc,jb,2)) + &
22         p_int%e_bln_c_s(jc,3,jb) *                               &
23         z_w_concorr_me(ieidx(jc,jb,3),jk,ieblk(jc,jb,3))
```

Besides the data access problem, notice in the example code the amount of optimization that the scientists do through the source code.

- An OpenMP directive is used to parallelize the do loop (Line 1).
- The scientists apply blocking to the loops through the *do* loop in Line 2, such that the inner loops traverse blocks of cells.
- The second and third (inner) loops are replicated to optimize the access on different architectures. A loop interchange is applied within redundant code sections in Line 8 and Line 12.
- A directive is used within one version (see Line 10) of the redundant code to guide vectorization.

5.2.1.2. Indirect Addressing to Access Cells Sharing an Edge

In addition to using lookup arrays to retrieve edge block numbers and edge numbers of edges that bound a cell, lookup arrays are used to access other neighboring grid elements, e.g, cells sharing an edge. The code snippet in Listing 5.2 demonstrates the concept. This code is extracted from the horizontal gradient of the Exner pressure within the dynamical core of ICON.

This code snippet traverses the edges of the grid and updates a field (*z_gradh_exner*) that is located at the edges based on values of other fields, e.g, *z_exner_ex_pr*, which are located at the centers of the cells. Values of the cell-located variable are accessed on cells on both sides of a shared edge. The code in the yellow box in Line 3 shows the use of the lookup tables *icidx*, *ikidx*, and *icblk* to access cells sharing an edge.

Listing 5.2: **Indirect indexing of cells sharing an edge**

```
1   z_gradh_exner(je,jk,jb) =                                    &
2     p_patch%edges%inv_dual_edge_length(je,jb)              * &
3    (z_exner_ex_pr(icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)) + &
4     p_nh%metrics%zdiff_gradp(2,je,jk,jb)                   * &
5    (z_dexner_dz_c(1,icidx(je,jb,2),                         &
6                   ikidx(2,je,jk,jb),icblk(je,jb,2))     + &
7     p_nh%metrics%zdiff_gradp(2,je,jk,jb)                   * &
8     z_dexner_dz_c(2,icidx(je,jb,2),                         &
9                   ikidx(2,je,jk,jb),icblk(je,jb,2))) -   &
10   (z_exner_ex_pr(icidx(je,jb,1),                           &
11                  ikidx(1,je,jk,jb),icblk(je,jb,1))     + &
12    p_nh%metrics%zdiff_gradp(1,je,jk,jb)                   * &
13   (z_dexner_dz_c(1,icidx(je,jb,1),                         &
14                  ikidx(1,je,jk,jb),icblk(je,jb,1))     + &
15    p_nh%metrics%zdiff_gradp(1,je,jk,jb)                   * &
16    z_dexner_dz_c(2,icidx(je,jb,1),                         &
17                  ikidx(1,je,jk,jb),icblk(je,jb,1)))))
```

5.2.1.3. Indirect Addressing of Vertical Dimension

In the same previous code snippet (Listing 5.2), stencils update the edge-located field which is a 3D field. The same technique of lookup arrays that is used to access horizontal neighboring grid elements is used also to access vertical dimension. The lookup array *ikidx* (see the yellow box in Line 3) is used to store the vertical dimension for indirect addressing.

5.2.1.4. Referring Indirectly to Both Vertices and Cells Around an Edge

In applications using staggered grids, some stencils include access to cells and their edges, but also in some applications vertices are also used. In the following code snippet (Listing 5.3), the values of a variable at the vertices of an edge and on the cells sharing the edge are accessed indirectly using lookup arrays. The code is extracted from upwind-biased values computation for the density *rho* within the dynamical core of the ICON model. The *rho* values on the edges are computed based on values of *rho* at the vertices bounding an edge and the cells sharing it.

In this code snippet, the edge-located field *z_rho_e* is updated while traversing the grid edges. Different values are read in the stencil to update that field. The field *z_rho_v*, which is located at the vertices, is read at the two vertices of the edge using the *ivblk* and *ividx* lookup arrays to read the block number and vertex number respectively

(see the yellow box in Line 17). The third index of those arrays, which takes the values 1 and 2, is used to refer to the number of the vertex of the edge, where each edge is bound by two vertices.

In the same stencil, the values of the *rho* variable, which is located at the cell centers, are read using the *icblk* and *icidx* lookup arrays (see the yellow box in Line 21), which store the cell block number and cell number respectively.

Listing 5.3: **Indirect addressing of both vertices and cells around an edge**

```
1  #else
2      DO jk = 1, nlev
3        DO je = i_startidx, i_endidx
4  #endif
5
6          z_rho_e(je,jk,jb) =                                    &
7              p_int%c_lin_e(je,1,jb)                           * &
8              p_nh%prog(nnow)%rho(icidx(je,jb,1),                &
9                                  jk,icblk(je,jb,1))           + &
10             p_int%c_lin_e(je,2,jb)                           * &
11             p_nh%prog(nnow)%rho(icidx(je,jb,2),                &
12                                 jk,icblk(je,jb,2))           - &
13             dtime * (p_nh%prog(nnow)%vn(je,jk,jb)            * &
14             p_patch%edges%inv_dual_edge_length(je,jb)        * &
15            (p_nh%prog(nnow)%rho(icidx(je,jb,2),                &
16                                 jk,icblk(je,jb,2))           - &
17             p_nh%prog(nnow)%rho (icidx(je,jb,1)jk,icblk(je,jb,1)) ) + &
18             p_nh%diag%vt(je,jk,jb)                           * &
19             p_patch%edges%inv_primal_edge_length(je,jb)      * &
20             p_patch%edges%tangent_orientation(je,jb)         * &
21            (z_rho_v (ividx(je,jb,2),jk,ivblk(je,jb,2))     - &
22             z_rho_v(ividx(je,jb,1),jk,ivblk(je,jb,1))))
```

5.2.1.5. Stencils with Direct Neighboring (Vertical) Elements

Although horizontal stencils are most frequently used, e.g, in dynamics modeling, vertical neighbors are also important, e.g, physics. Besides to using lookup arrays for indirect addressing in the vertical direction, direct addressing of vertical neighbors is used in some stencils .

Computed vertical neighbors (Listing 5.4) are possible when the vertical dimension is accessed directly without lookup arrays. This case is extracted from temporal averaging of the density *rho* and the virtual potential temperature *theta* within the dynamical core of the ICON model. The neighbors are addressed relative to the vertical index with normal + and − operators.

In this code snippet, the values of the variables *rho* and *theta_v* are read from two neighboring vertical levels. The values are accessed by using the indices jk and $jk-1$ (as seen within the yellow box in Line 7 and Line 16) to access the values at the current level and the level directly below.

Listing 5.4: **Stencils including vertical neighbors**

```
1     DO jk = 2, nlev
2       DO jc = i_startidx, i_endidx
3
4         ...
5
6         z_rho_tavg_m1    = wgt_nnow_rth                    * &
7                            p_nh%prog(nnow)%rho(jc,jk-1,jb)    + &
8                            wgt_nnew_rth                      * &
9                            p_nh%prog(nvar)%rho(jc,jk-1,jb)
10        z_theta_tavg_m1 = wgt_nnow_rth                     * &
11                           p_nh%prog(nnow)%theta_v(jc,jk-1,jb)+ &
12                           wgt_nnew_rth                      * &
13                           p_nh%prog(nvar)%theta_v(jc,jk-1,jb)
14
15        z_rho_tavg       = wgt_nnow_rth                    * &
16                           p_nh%prog(nnow)%rho(jc,jk,jb)      + &
17                           wgt_nnew_rth                      * &
18                           p_nh%prog(nvar)%rho(jc,jk,jb)
19        z_theta_tavg     = wgt_nnow_rth                    * &
20                           p_nh%prog(nnow)%theta_v(jc,jk,jb)  + &
21                           wgt_nnew_rth                      * &
22                           p_nh%prog(nvar)%theta_v(jc,jk,jb)
```

5.2.1.6. Mixed Dimensionalities

A case of vertical integration is shown in the code snippet in Listing 5.5. This code is extracted from code that computes the contribution of thermal expansion to vertical wind within the dynamical core of the ICON model. A two-dimensional field is updated based on the values of another three-dimensional variable.

In this code snippet, the values of the three variables *ddt_exner_phy*, *exner*, and *inv_ddqz_z_full*, all of which is located at the cell centers of the 3D grid (see the three indices in the yellow box in Line 13), are used to calculate some expression. The value of the calculated expression is summed over each column and used to update the 2D field *z_thermal_exp* (see the two indices in the yellow box in Line 11). Thus, values on the 2D surface are sums of (3D) columns over that surface.

Listing 5.5: **Vertical integration**

```
1 DO jb = i_startblk, i_endblk
2
3   CALL get_indices_c(p_patch, jb, i_startblk, i_endblk,   &
4     i_startidx, i_endidx, rl_start, rl_end)
5
6 ...
7
8       z_thermal_exp(:,jb) = 0._wp
9       DO jk = 1, nlev
```

```
10          DO jc = i_startidx, i_endidx
11            z_thermal_exp(jc,jb) =
12                        z_thermal_exp(jc,jb) + cvd_o_rd        &
13                      * p_nh%diag%ddt_exner_phy(jc,jk,jb)      &
14                      /(p_nh%prog(nnow)%exner(jc,jk,jb)        &
15                      * p_nh%metrics%inv_ddqz_z_full(jc,jk,jb))
16          ENDDO
17        ENDDO
18
19 ...
20 ENDDO
```

5.2.1.7. Different Temporary Data Structures to Optimize for Different Architectures

The optimal way to move data between different parts of the code depends on the architecture. The following code snippet (Listing 5.6) shows a case that was written with sections to account for different architecture when passing data. Scalar variables in one section are rewritten as arrays in the other section to transfer temporary calculation results between loops. This code is extracted from calculating the values of *rho* and *theta_v* on the grid edges within the dynamical core of the ICON model.

In the shown code snippet, which is reduced with deleting much of the expressions because of the length of the original code to demonstrate the main problem, data movement is handled in two alternative methods:

- Within the first alternative code, which targets cache-based machines, doing computations within fused loops allows better performance. Thus, the calculated data is moved simply using the simple variables *distv_bary_1* (computed within the yellow box in Line 8 and used within the yellow box in Line 15) and *distv_bary_2* within the loop.

- In the second alternative, the computation is done through two separate loops (no fusion). To transfer calculated data between the two loops, the array *z_distv_bary* is used, where the calculated data is written to this array in the first loop (marked by the yellow box in Line 33) and read again in the other loop (marked by the yellow box in Line 50).

Listing 5.6: **Different data movement techniques on different architectures**

```
1 #ifdef __LOOP_EXCHANGE
2   DO je = i_startidx, i_endidx
3     DO jk = 1, nlev
4
5       z_ntdistv_bary_1 =  ...
6       z_ntdistv_bary_2 =  ...
7
```

```
8         distv_bary_1 = z_ntdistv_bary_1 * ...                    &
9                      + z_ntdistv_bary_2 * ...
10
11        distv_bary_2 = z_ntdistv_bary_1 * ...                     &
12                     + z_ntdistv_bary_2 * ...
13
14        z_rho_e(je,jk,jb) = ...                                   &
15          + distv_bary_1 * ...                                    &
16          + distv_bary_2 * ...
17
18        z_theta_v_e(je,jk,jb) = ...                               &
19          + distv_bary_1 * ...                                    &
20          + distv_bary_2 * ...
21
22      ENDDO ! loop over edges
23    ENDDO   ! loop over vertical levels
24
25  #else
26    DO jk = slev, elev
27      DO je = i_startidx, i_endidx
28
29        z_ntdistv_bary(1) =  ...
30
31        z_ntdistv_bary(2) =  ...
32
33        z_distv_bary(je,jk,jb,1) =                                &
34              z_ntdistv_bary(1) * ...                             &
35            + z_ntdistv_bary(2) * ...
36
37        z_distv_bary(je,jk,jb,2) =                                &
38              z_ntdistv_bary(1) * ...                             &
39            + z_ntdistv_bary(2) * ...
40
41      ENDDO ! loop over edges
42    ENDDO   ! loop over vertical levels
43
44          DO jk = 1, nlev
45            DO je = i_startidx, i_endidx
46
47              ...
48
49              z_rho_e(je,jk,jb) = ...                             &
50                + z_distv_bary(je,jk,jb,1) * ...                  &
51                + z_distv_bary(je,jk,jb,2) * ...
52
53              z_theta_v_e(je,jk,jb) = ...                         &
54                + z_distv_bary(je,jk,jb,1) * ...                  &
55                + z_distv_bary(je,jk,jb,2) * ...
56
57            ENDDO ! loop over edges
58          ENDDO   ! loop over vertical levels
```

```
59 #endif
```

5.2.2. Offset-Based Addressing

Compared to applications that use unstructured grids, where special data structures are used to describe the structure/connectivity of the grid, some models, e.g., NICAM and DYNAMICO, use grids in which connectivity is described with some simple formulae, which simply allows access from an element to other elements with simple arithmetic operations, e.g, addition or subtraction.

5.2.2.1. Addressing Edges of a Cell

The following code snippet (Listing 5.7) is a kernel taken from the DYNAMICO model to compute the divergence operator of the horizontal flux. The stencil updates the cell-located field based on edge-located flux field.

The outer loop (in Line 7) traverses the vertical dimension of the grid, and the inner loop (in Line 9) is used to traverse the horizontal dimensions. The horizontal dimensions are represented with one array dimension, and hence are accessed with one index, i.e, ij. A directive is used at the inner loop to guide SIMD vectorization (see Line 8).

The stencil is applied at each cell of the grid, and updates the cell-located field *convm* based on the values of the edge-located field $hflux$. The horizontal indices to access the edges of the cell ij are computed with adding offsets (u_right (see the yellow box in Line 13), u_rup, u_lup, u_left, u_ldown, and u_rdown) to the cell index ij. Six values are read within the stencil as the application uses a hexagonal grid, where each cell is bound with six edges.

Listing 5.7: **Horizontal divergence accessing edges of a cell using offsets**

```
1  ! hflux in kg/s
2  REAL(rstd),INTENT(IN)  :: hflux(3*iim*jjm,llm)
3  ! mass flux convergence
4  REAL(rstd),INTENT(OUT) :: convm(iim*jjm,llm)
5  INTEGER :: ij,l
6
7  DO l=ll_begin, ll_end
8     !DIR$ SIMD
9     DO ij=ij_begin,ij_end
10       ! convm = -div(mass flux), sign convention
11       ! as in Ringler et al. 2012, eq. 21
12       convm(ij,l)= -1./Ai(ij) *(                &
13             ne_right*hflux( ij+u_right ,l)    +   &
14             ne_rup*hflux(ij+u_rup,l)          +   &
15             ne_lup*hflux(ij+u_lup,l)          +   &
16             ne_left*hflux(ij+u_left,l)        +   &
17             ne_ldown*hflux(ij+u_ldown,l)      +   &
18             ne_rdown*hflux(ij+u_rdown,l))
19    END DO ! ij
```

```
20 END DO ! llm
```

5.2.2.2. Horizontal and Vertical Neighbors

Stencils may include vertically neighboring components besides to horizontal components. In the following code snippet (Listing 5.8), which is extracted from code that computes the Bernouilli term out of kinetic energy and geopotential within the DYNAMICO model, both kinds of neighborhoods are used.

The field *berni* is updated based on values from the geopotential field *geopot* and the fields *le*, *de*, and *u*. The last three fields are accessed at six horizontally-neighboring points. However, the average of the geopotential field *geopot* is computed from the current cell and the one directly above in the vertical dimension as shown in Line 5. The horizontal neighboring values of the fields *le*, *de*, and *u* are accessed using the offsets *u_right*, *u_rup*, *u_lup*, *u_left*, *u_ldown*, and *u_rdown* relative to the horizontal index *ij*

Listing 5.8: **Horizotal and vertical neighbors**

```
1  DO l=ll_begin,ll_end
2  !DIR$ SIMD
3    DO ij=ij_begin,ij_end
4
5      berni(ij,l) = .5*(geopot(ij,l)+geopot(ij,l+1))          &
6       + 1/(4*Ai(ij))*(                                       &
7          le(ij+u_right)*de(ij+u_right)*u(ij+u_right,l)**2 + &
8          le(ij+u_rup)  *de(ij+u_rup)  *u(ij+u_rup,l)**2   + &
9          le(ij+u_lup)  *de(ij+u_lup)  *u(ij+u_lup,l)**2   + &
10         le(ij+u_left) *de(ij+u_left) *u(ij+u_left,l)**2  + &
11         le(ij+u_ldown)*de(ij+u_ldown)*u(ij+u_ldown,l)**2 + &
12         le(ij+u_rdown)*de(ij+u_rdown)*u(ij+u_rdown,l)**2 )
13   ENDDO
14 ENDDO
```

5.2.2.3. Vector Fields

Besides to simple scalar fields, some computations include vector fields, where multiple values corresponding to different spatial components are needed. The following sample code snippet (Listing 5.9) is extracted based on computing the divergence operator of the horizontal velocity within the NICAM model, where divergence is computed taking into account three components, i.e, x, y, and z, of the velocity.

The code traverses all the regions in the l loop in Line 24. In each region, both the vertical and the horizontal are traversed through the k (in Line 25) and the g (in Line 26) loops respectively. The horizontal component (g) is again represented with one array as with ij in DYNAMICO model.

The velocity is represented with the three components Vx, Vy, and Vz. Those three vector components are accessed within the stencil via computing predefined constant

offsets (additions and subtractions) with respect to the current horizontal grid point g. Thus, we see the expressions $g+1$ (see the yellow box in Line 28), $g+iall+1$, $g+iall$, $g-1$, $g-iall-1$, and $g-iall$ used as horizontal indices . One dimension of the coefficients array $coef\_div$ is used to refer to the vector component, which can be accessed with values $XDIR$, $YDIR$, or $ZDIR$.

Listing 5.9: **Horizontal divergence using vector fields (NICAM)**

```
1     integer :: iall ! num. of the horizontal grid for i-axis
2     integer :: gall = iall * iall
3     integer :: kall ! num. of the vertical layer
4     integer :: lall ! num. of the region for this process
5
6     ! scalar
7     real(8) :: scl      (gall,kall,lall)
8     ! horizontal velocity V (x-component)
9     real(8) :: vx       (gall,kall,lall)
10    ! horizontal velocity V (x-component)
11    real(8) :: vy       (gall,kall,lall)
12    ! horizontal velocity V (x-component)
13    real(8) :: vz       (gall,kall,lall)
14    ! constant vector coefficient
15    real(8) :: coef_div(gall,0:6,3,lall)
16
17    integer :: XDIR=1, YDIR=2, ZDIR=3
18    integer :: gmin, gmax
19    integer :: g, k, l
20
21    gmin = 1    + iall + 1 ! start point of the inner grid
22    gmax = gall - iall - 1 ! end   point of the inner grid
23
24    do l = 1, lall
25    do k = 1, kall
26    do g = gmin, gmax
27       scl(g,k,l) = coef_div(g,0,XDIR,l) * Vx(g        ,k,l) &
28                  + coef_div(g,1,XDIR,l) * Vx(g+1      ,k,l) &
29                  + coef_div(g,2,XDIR,l) * Vx(g+iall+1,k,l) &
30                  + coef_div(g,3,XDIR,l) * Vx(g+iall  ,k,l) &
31                  + coef_div(g,4,XDIR,l) * Vx(g-1      ,k,l) &
32                  + coef_div(g,5,XDIR,l) * Vx(g-iall-1,k,l) &
33                  + coef_div(g,6,XDIR,l) * Vx(g-iall  ,k,l) &
34                  + coef_div(g,0,YDIR,l) * Vy(g        ,k,l) &
35                  + coef_div(g,1,YDIR,l) * Vy(g+1      ,k,l) &
36                  + coef_div(g,2,YDIR,l) * Vy(g+iall+1,k,l) &
37                  + coef_div(g,3,YDIR,l) * Vy(g+iall  ,k,l) &
38                  + coef_div(g,4,YDIR,l) * Vy(g-1      ,k,l) &
39                  + coef_div(g,5,YDIR,l) * Vy(g-iall-1,k,l) &
40                  + coef_div(g,6,YDIR,l) * Vy(g-iall  ,k,l) &
41                  + coef_div(g,0,ZDIR,l) * Vz(g        ,k,l) &
42                  + coef_div(g,1,ZDIR,l) * Vz(g+1      ,k,l) &
43                  + coef_div(g,2,ZDIR,l) * Vz(g+iall+1,k,l) &
44                  + coef_div(g,3,ZDIR,l) * Vz(g+iall  ,k,l) &
```

```
45                     + coef_div(g,4,ZDIR,l) * Vz(g-1      ,k,l) &
46                     + coef_div(g,5,ZDIR,l) * Vz(g-iall-1,k,l) &
47                     + coef_div(g,6,ZDIR,l) * Vz(g-iall   ,k,l)
48       enddo
49       enddo
50       enddo
```

5.2.2.4. Scalar Weight Variables

Variables that are used as weights for stencils may need to carry different values for different stencil components. The following code snippet (Listing 5.10) shows an example. This code is extracted from NICAM model code to compute the Laplacian operator.

In this code snippet, the 2D variable *clap* weighs the *scl* 3D field in the applied stencil. However, the seven components of the stencil, i.e, the center of the stencil and the six surroundings, are weighed with different values. This is done by defining this variable with an additional array index (see the yellow box in Line 13, notice the underlined additional index). The additional index refers to the neighboring point.

Listing 5.10: **Multi-valued weight fields**

```
1    do l = 1, ADM_lall
2  !OCL PARALLEL
3       do k = 1, ADM_kall
4          do n = OPRT_nstart, OPRT_nend
5             ij      = n
6             ip1j    = n + 1
7             ijp1    = n     + ADM_gall_1d
8             ip1jp1  = n + 1 + ADM_gall_1d
9             im1j    = n - 1
10            ijm1    = n     - ADM_gall_1d
11            im1jm1  = n - 1 - ADM_gall_1d
12
13            dscl(n,k,l) = clap(n,l,0) * scl(ij     ,k,l) &
14                        + clap(n,l,1) * scl(ip1j   ,k,l) &
15                        + clap(n,l,2) * scl(ip1jp1 ,k,l) &
16                        + clap(n,l,3) * scl(ijp1   ,k,l) &
17                        + clap(n,l,4) * scl(im1j   ,k,l) &
18                        + clap(n,l,5) * scl(im1jm1 ,k,l) &
19                        + clap(n,l,6) * scl(ijm1   ,k,l)
20         enddo
```

5.2.2.5. Vector Weight Variables

Instead of defining weight variables in separate variables per vector dimension, they are defined in one variable with additional array dimensions. This is similar to adding a dimension to support scalar weight variables weighing different neighbors in Listing 5.10, but with an additional index. An example is shown in Listing 5.11, which is extracted from the NICAM model code to compute the gradient operator.

In this code snippet, the field *cgrad* is defined with three vector component values. This is done using an additional (the last) index as shown in the yellow box in Line 14, notice the underlined index d which refers to one of the X, Y, or Z vector components. Also, those vectors are defined not only at the centers of the cells, but also for the six neighbors.

Listing 5.11: **Single-variable vector fields**

```
1  do d = 1, ADM_nxyz
2    do l = 1, ADM_lall
3  !OCL PARALLEL
4      do k = 1, ADM_kall
5         do n = OPRT_nstart, OPRT_nend
6            ij      = n
7            ip1j    = n + 1
8            ijp1    = n      + ADM_gall_1d
9            ip1jp1  = n + 1  + ADM_gall_1d
10           im1j    = n - 1
11           ijm1    = n      - ADM_gall_1d
12           im1jm1  = n - 1  - ADM_gall_1d
13
14           grad(n,k,l,d) = cgrad(n,l,0,d) * scl(ij     ,k,l) &
15                         + cgrad(n,l,1,d) * scl(ip1j   ,k,l) &
16                         + cgrad(n,l,2,d) * scl(ip1jp1 ,k,l) &
17                         + cgrad(n,l,3,d) * scl(ijp1   ,k,l) &
18                         + cgrad(n,l,4,d) * scl(im1j   ,k,l) &
19                         + cgrad(n,l,5,d) * scl(im1jm1 ,k,l) &
20                         + cgrad(n,l,6,d) * scl(ijm1   ,k,l)
21        enddo
```

5.2.2.6. Region Boundaries

Dividing the problem domain into regions, e.g, for parallelization reasons, requires special handling of some sets of grid points. The following example code snippet (Listing 5.12), which is extracted from NICAM model code to compute the gradient operator, calculates some expression for inner points of a region, and assigns zero to the outer grid points of the region.

The field *grad* is computed based on some stencil in the area that is horizontally limited between $OPRT\_nstart$ and $OPRT\_nend$. However, the other grid points which lie horizontally before and after those points are set to zero, as shown in code marked with the yellow box in Line 21.

Listing 5.12: **Region boundaries**

```
1      do d = 1, ADM_nxyz
2      do l = 1, ADM_lall
3      do k = 1, ADM_kall
4         do n = OPRT_nstart, OPRT_nend
5            ij      = n
```

```
6            ip1j    = n + 1
7            ijp1    = n      + ADM_gall_1d
8            ip1jp1  = n + 1  + ADM_gall_1d
9            im1j    = n - 1
10           ijm1    = n      - ADM_gall_1d
11           im1jm1  = n - 1  - ADM_gall_1d
12
13           grad(n,k,l,d) = cgrad(n,l,0,d) * scl(ij     ,k,l) &
14                         + cgrad(n,l,1,d) * scl(ip1j   ,k,l) &
15                         + cgrad(n,l,2,d) * scl(ip1jp1 ,k,l) &
16                         + cgrad(n,l,3,d) * scl(ijp1   ,k,l) &
17                         + cgrad(n,l,4,d) * scl(im1j   ,k,l) &
18                         + cgrad(n,l,5,d) * scl(im1jm1 ,k,l) &
19                         + cgrad(n,l,6,d) * scl(ijm1   ,k,l)
20       enddo
21       grad( 1:OPRT_nstart-1,k,l,d) = 0.0_RP
22       grad(OPRT_nend+1:ADM_gall      ,k,l,d) = 0.0_RP
23     enddo
24     enddo
25     enddo
```

5.2.2.7. Computing Scalar Fields in Separate Steps Based on Vector Components

Computations may include updating some scalar field based on vector components. Each component in this case is computed separately, and the final scalar value is computed based on the different components. An example is shown in Listing 5.13, where three separate components are computed based on different vector components, and a final scalar is then computed. The code is extracted from NICAM model code to compute the divergence operator.

The three vector components, i.e, x, y, and z, are used in three loop constructs to calculate three separate results $sclx$, $scly$, and $sclz$ as shown in the yellow boxes in the Lines 20, 40, and 60 respectively. The results are stored temporarily in three arrays. Finally, the scalar variable scl is computed based on the three temporary computations as marked with the yellow box in Line 84.

Listing 5.13: **Computing scalars in separate components**

```
1   gall     = ADM_gall
2   gall_1d  = ADM_gall_1d
3   kall     = ADM_kall
4
5   do l = 1, ADM_lall
6    !$omp parallel default(none), private(n, k, ij, ip1j, ip1jp1,
       ↪ ijp1, im1j, ijm1, im1jm1), &
7    !$omp shared(OPRT_nstart, OPRT_nend, gall, gall_1d, kall, l, scl,
       ↪  sclx, scly, sclz, cdiv, vx, vy, vz)
8      do k = 1, kall
9
10       !$omp do
```

```
11          do n = OPRT_nstart, OPRT_nend
12             ij      = n
13             ip1j    = n + 1
14             ijp1    = n      + gall_1d
15             ip1jp1  = n + 1  + gall_1d
16             im1j    = n - 1
17             ijm1    = n      - gall_1d
18             im1jm1  = n - 1  - gall_1d
19
20             sclx(n) = cdiv(n,l,0,1) * vx(ij     ,k,l) &
21                     + cdiv(n,l,1,1) * vx(ip1j   ,k,l) &
22                     + cdiv(n,l,2,1) * vx(ip1jp1 ,k,l) &
23                     + cdiv(n,l,3,1) * vx(ijp1   ,k,l) &
24                     + cdiv(n,l,4,1) * vx(im1j   ,k,l) &
25                     + cdiv(n,l,5,1) * vx(im1jm1 ,k,l) &
26                     + cdiv(n,l,6,1) * vx(ijm1   ,k,l)
27          enddo
28          !$omp end do nowait
29
30          !$omp do
31          do n = OPRT_nstart, OPRT_nend
32             ij      = n
33             ip1j    = n + 1
34             ijp1    = n      + gall_1d
35             ip1jp1  = n + 1  + gall_1d
36             im1j    = n - 1
37             ijm1    = n      - gall_1d
38             im1jm1  = n - 1  - gall_1d
39
40             scly(n) = cdiv(n,l,0,2) * vy(ij     ,k,l) &
41                     + cdiv(n,l,1,2) * vy(ip1j   ,k,l) &
42                     + cdiv(n,l,2,2) * vy(ip1jp1 ,k,l) &
43                     + cdiv(n,l,3,2) * vy(ijp1   ,k,l) &
44                     + cdiv(n,l,4,2) * vy(im1j   ,k,l) &
45                     + cdiv(n,l,5,2) * vy(im1jm1 ,k,l) &
46                     + cdiv(n,l,6,2) * vy(ijm1   ,k,l)
47          enddo
48          !$omp end do nowait
49
50          !$omp do
51          do n = OPRT_nstart, OPRT_nend
52             ij      = n
53             ip1j    = n + 1
54             ijp1    = n      + gall_1d
55             ip1jp1  = n + 1  + gall_1d
56             im1j    = n - 1
57             ijm1    = n      - gall_1d
58             im1jm1  = n - 1  - gall_1d
59
60             sclz(n) = cdiv(n,l,0,3) * vz(ij     ,k,l) &
61                     + cdiv(n,l,1,3) * vz(ip1j   ,k,l) &
```

```
62                      + cdiv(n,l,2,3) * vz(ip1jp1,k,l) &
63                      + cdiv(n,l,3,3) * vz(ijp1   ,k,l) &
64                      + cdiv(n,l,4,3) * vz(im1j   ,k,l) &
65                      + cdiv(n,l,5,3) * vz(im1jm1,k,l) &
66                      + cdiv(n,l,6,3) * vz(ijm1   ,k,l)
67          enddo
68          !$omp end do nowait
69
70          !$omp do
71          do n = 1, OPRT_nstart-1
72             scl(n,k,l) = 0.0_RP
73          enddo
74          !$omp end do nowait
75
76          !$omp do
77          do n = OPRT_nend+1, gall
78             scl(n,k,l) = 0.0_RP
79          enddo
80          !$omp end do
81
82          !$omp do
83          do n = OPRT_nstart, OPRT_nend
84             scl(n,k,l) = sclx(n) + scly(n) + sclz(n)
85          enddo
86          !$omp end do
87
88       enddo
89       !$omp end parallel
90    enddo
```

5.3. The Specifications of the Language Extensions

In this part, we define the specifications of the language extensions that we add to the general-purpose language, and discuss the development of the specifications based on the nominated codes from the three models and the discussed requirements. We formed the extensions considering each model, and kept in mind that maximizing the commonality of the extensions between the three models leads to define domain-specific extensions. We call this set of language extensions **GGDML**, which stands for **G**eneral **G**rid **D**efinition and **M**anipulation **L**anguage.

In the following text, we discuss the extensions that help to

- declare variables on the grid
- specify grids' definition extensions, which are used to define special sets of cells, edges or vertices of the grid
- reference the grid variables

- iterate a grid or subsets of it
- use a notion of reduction for the stencil codes in which an operation is applied over multiple neighbors

5.3.1. Declarations

A first core concept, where we should start, is the grid concept. Many concepts are then built on this abstraction. Grids represent discretizations of space, where field values are located.

Different methods are used to generate the tessellations. The simplest way is to divide the space (the domain where the problem is to be solved) using orthogonal lines. This method divides a 2D surface into rectangles, forming a regular grid. Another example is to divide the surface of a sphere into twenty spherical triangles of equal area, forming a spherical icosahedron. Resolution refinements are achieved by dividing the arcs of the triangles into equal lengths and connecting the resulting points. Some grids are formed by synthesizing hexagons from the triangles.

5.3.1.1. Declaration Specifiers

Whatever method is used to generate the tessellations, applications have a set of options to decide regarding the location of field values with respect to the primitive grid shape. For example, an application can choose the centers of the cells, e.g, rectangles, to localize all field values. Another application can also use the edges that bound the cells, or the vertices shared between the edges to store some fields [AL77].

Normally, applications deal with 2D or 3D spaces. The discussed concepts regarding the tessellation methods and field value locations, which consider 2D surfaces, is also applied to 3D spaces. The surface is tessellated as mentioned and the vertical dimension is also divided into a set of elevations.

General-purpose programming languages in general define their own declaration specifiers. To allow developers to declare fields with specific characteristics, we add additional declaration specifiers besides the standard set of declaration specifiers that the host programming language defines.

We define our declaration specifiers into groups, each of which includes a set of alternative specifiers. Our basic set of declaration specifiers includes a group of specifiers to specify the dimensionality of the field, and another to specify the localization of the field with respect to the grid's primitive shapes.

Field localization To distinguish where the field values are located with respect to the primitive grid shapes, the localization declaration specifiers group provides the declaration specifiers CELL, EDGE, and VERT. This group allows our extension set to conform to the requirement **RFSL** (Section 5.1).

The normal syntax that would be used to declare the variable in the general-purpose language is used, however, an additional specifier is added in the declaration to specify

the localization of the field. For example, to declare a variable to hold a single precision floating point value of the *temperature* field at the cell centers, and two other fields localized on edges and at vertices, in Fortran we write the following declaration:

Listing 5.14: **Example Fortran declarations with field localization specifiers**

```
1 real , CELL :: temperature
2 real , EDGE :: vel_c1
3 real , VERT :: pro_v1
```

In C we declare the variables as follows:

Listing 5.15: **Example C declarations with field localization specifiers**

```
1 float CELL temperature;
2 float EDGE vel_c1;
3 float VERT pro_v1;
```

Field dimensionality As discussed, stencil applications normally use 2D and/or 3D grids as problem domains. To let developers specify whether a field is discretized over a 2D or a 3D grid, we provide the dimensionality declaration specifier group. This group contains the basic declaration specifiers 2D and 3D. This group allows our extension set to conform to the requirement **RFSD** (Section 5.1).

The declaration specifiers in the dimensionality group are used besides the other declaration specifiers; standard general-purpose language and extensions, e.g, localization group. For example, to complete the declaration of the *temperature* and the *vel_c*1 fields over a 3D grid, and the *pro_v*1 fields over a 2D surface grid, in Fortran we write the following declaration:

Listing 5.16: **Example Fortran declarations with field dimensionality specifiers**

```
1 real , CELL , 3D :: temperature
2 real , EDGE , 3D :: vel_c1
3 real , VERT , 2D :: pro_v1
```

In C we declare the variables as follows:

Listing 5.17: **Example C declarations with field dimensionality specifiers**

```
1 float CELL 3D temperature;
2 float EDGE 3D vel_c1;
3 float VERT 2D pro_v1;
```

Additional localizations and dimensionalities The declaration specifiers that are presented in the previous paragraphs allow to achieve the requirements **RFSL** and **RFSD** (Section 5.1). Basic declaration specifiers are provided to support the basic localization and dimensionality field characteristics that are mentioned in the two requirements. However, according to the requirement **RNFC**, the language extensions specification

should provide higher-level guidelines to define declaration specifiers rather than a static set of fixed specifiers, in order to support the adaptability of the language extensions to the application-specific needs. Therefore, our specifications for the declaration specifiers allows adding new groups and allows modifying the mentioned declaration specifiers or adding new specifiers to the localization and/or dimensionality groups.

As an example to such application-specific needs, some models use vertically-staggered grids, in which half levels in the vertical dimensions are used. To support such an application, a new declaration specifier can be added to a corresponding declaration specifier group, e.g., use HL for half levels.

5.3.1.2. Data Types

Developers use declarations as in the pure host general-purpose language. The additional declaration specifiers are used from the syntax point of view exactly as those of the host language. Thus, the data types that are used in the host language are still used. This gives the developers high flexibility to use whatever data to represent a field over a grid. Besides to primitive data types, more structured data types can be used, e.g, arrays or structures.

The following snippet shows declaration of fields with different data types. The first field in the code snippet, i.e., *single_prec_var*, stores a single-precision floating point value for each edge in the 3D grid. The field *double_prec_var* stores double precision floating point values per edge. And the field *some_type_var* stores values of some defined type, e.g., for fixed point, at each edge of the 3D grid.

Listing 5.18: **Example C declarations with different data types**

```
1 float           EDGE  3D  single_prec_var;
2 double          EDGE  3D  double_prec_var;
3 typedef         ...       SOMETYPE_t;
4 SOMETYPE_t      EDGE  3D  some_type_var;
```

As the example shows, the types 'float', 'double', and a defined type from the host language are applicable to the declarations. This flexibility of using the data types that the host language provides allows our extension set to conform to the requirement **RNDT** (Section 5.1).

5.3.2. Iterator

Performance-demanding stencil computations comprise a number of stencils. Stencils are applied within loop structures that traverse a specific domain. Those parts consume most of the running time of such family of applications. Focusing on those parts to optimally use the machine resources leads to achieve nearly-optimal runs of the applications.

To allow our language extensions to support performance-demanding stencil computations, we provide a special iterator to apply stencils. This iterator is an additional construct besides to the loop constructs that the host language provides. However, this

suggested addition carries different semantics in comparison to loops of general-purpose languages.

Our iterator extension implies that the body statement of the iterator, where stencils occur, should be applied at each one of a specific set of grid points. In comparison to the notions of counters or conditions in loops, the iterator extension uses the notion of grid and stencils, which is a natural fit for parallel execution. The semantics that limit the automatic parallelization of loops are superseded with different semantics, based on which, both code developers and tool developers can build on the assumption of parallelism.

Within the structure of the iterator, the developers specify the set of grid points on which the body statement will be applied. The body statement also is provided. The first part of the iterator structure is written using added (non-host language) keywords and extensions to specify the domain, where the body will be applied. However, the body statement is a host language statement. This can be any kind of host language statements, e.g, conditional, loop, expression, compound statements.

An iterator is a statement that can be written in place of host language statements. Speaking from the grammar's point of view, the iterator is a new kind of statement that is added to the standard set of statement types of the host language. So, the code can be written using the general-purpose host language, and the iterator statement is just a statement within the code. This, besides to using host language statements as a body of the iterator allows to conform to the requirement **RNMC** besides to fulfilling the requirement **RFGT** (Section 5.1).

To write an iterator statement, developers use the keyword *foreach*. Next, developers choose some id/name for an index that will be used within the body statement to refer to the current point on the grid. Then, the keyword *in* is added. Then, an expression is provided to specify the set of grid points where the body statement will be applied. The last part is the body statement. Te following snippet shows an example simple iterator statement to assign the value zero to each value of the field *temperature*, where the iterator applies the assignment statement to all values of the field.

Listing 5.19: **An example iterator used within Fortran code**

```
1 real , CELL , 3D :: temperature
2
3 FOREACH cell IN grid
4   temperature(cell) = 0.0
5 END FOREACH
```

The C version is as follows:

Listing 5.20: **An example iterator used within C code**

```
1 float CELL 3D temperature;
2
3 foreach cell in grid
4 {
5   temperature[cell] = 0.0;
6 }
```

To show how the code is used in the context of the host language, let's take the following example:

Listing 5.21: **An example use of an iterator within a block of C code**

```
1  int some_condition = 0;
2  float CELL 3D temperature;
3  float CELL 3D one_field;
4  float CELL 3D second_field;
5
6  do_something_with(some_condition);
7
8  if(some_condition == 0)
9    foreach cell in grid
10   {
11     temperature[cell] = 0.0;
12   }
13 else
14   foreach cell in grid
15   {
16     temperature[cell] = one_field[cell]    * 0.4 +
17                         second_field[cell] * 0.6 ;
18   }
```

This code checks some condition and assigns field values differently according to the condition. The iterator statements are written as parts of the C language if statement. The iterator body statement in the second iterator statement is a C-language statement, in which C mathematical operators $+$ and $*$ are used to compute a C-language expression. This is just a simple example, however, other C-language statements can be written within the iterator body statement, e.g, conditional statements.

5.3.2.1. Grid Specification

Iterator statements should specify which set of grid points to be traversed to apply the body statement, and hence apply the necessary stencils. A special expression extension is added to the standard set of expressions of the host language to allow developers to specify this set of grid points. From a grammar point of view, this is adding a new rule to the host language grammar to allow a new kind of expressions.

Grid specification expressions range from simple expressions which allows selecting a predefined grid as is, to more complicated expressions that apply modifiers to choose a subset or superset of grid points. Such modifiers are operators that modify an initial definition of a predefined grid to choose a specific subset of the points in that grid, or add new points to it temporarily for the current iterator traversal purpose. Uses of such modifiers include selecting specific levels in the vertical dimension (subsets), where some stencils should not be applied to all levels, and adding columns based on initial definitions of surface grids (supersets).

Simple grid specification expressions Definitions of grids that support an application are not mainly defined through expressions within iterators in source code. Rather, main grids are defined globally in an application configuration. Tools then can use the necessary information from the definitions to generate only the necessary code in the target code version.

Grid definitions allow inferring information regarding fields. Declarations of fields use declaration specifiers that indicate field characteristics, e.g, whether the field is a 3D one. This information allow the tools to identify the grid points where the field values are located. Tools keep this information about the grid points where field data are located in some tables (symbol tables) for later use.

Field declarations are one advantage of using global definitions of grids through application configurations. Iterator statements can be written with reference to those grid definitions. In this case an iterator statement can use the identifier of an intended grid as defined in an application configuration file, and the tools will use the definition of the grid from the configuration to identify which set of grid points to traverse when applying the iterator body statement.

Away from the tools and implementation, and to focus on the extensions' specifications, the grid specification expression in this case is simply an identifier. The following snippet Listing 5.22 demonstrates the use of grid identifiers.

Listing 5.22: **Example iterators with simple expressions to specify grids**

```
1 foreach cell in CELL2Dgrid
2 ...
3
4 foreach cell in CELL3Dgrid
5 ...
6
7 foreach edge in EDGE3Dgrid
8 ...
```

Each of the example iterator statements uses a different identity as an expression to specify which grid points will the body statement be applied to. Those identities should be names of defined grids within application configuration files.

Besides to the specification of the language extensions, we recommend that tool implementations provide an additional feature to simplify coding of iterator statements: Rather than keeping in developer's mind to refer to the different grids by their identities, the tools have the flexibility to use the special identity **(grid)** and identify from the index identity, that the developers use, to automatically identify the intended grid. A tool implementation can define a set of possible indices to infer specific grids, which can be used by the developers as in Listing 5.23. Further on implementation of this feature is discussed later in Page 127.

Listing 5.23: **Example iterators with standard indices to identify grids automatically**

```
1 //traverse cells of 3D grid
2 foreach cell in grid
3 ...
```

```
4
5 //traverse cells of 2D grid
6 foreach cc in grid
7 ...
8
9 //traverse edges of 3D grid
10 foreach edge in grid
11 ...
```

In this example, the tools can allow the assumption that the *cell* index means the $CELL3Dgrid$ grid is intended, and then automatically use the definition of that grid. Same is true for *edge* which leads to use the $EDGE3Dgrid$ grid definition, and *cc* which leads to use the $CELL2Dgrid$ grid definition. Using such special indices depends on how the tools support them. However, our language extensions specifications do not need to go to this level of details.

Grid specification expressions using modifier operators In addition to the introduced simple expressions to specify the grid points that an iterator statement traverses to apply some statement, expressions using operators can be used. Operators allow to start with a predefined grid definition, and define a new version temporarily for the use of that specific iterator statement, where the expression is being used.

To specify a set of grid points using such expressions, an identity of a grid that is defined in a translation configuration file is used. Identities of dimensions are also used. Dimensions can be either already defined dimensions that are used to define the grid, or new dimensions. Referring to already defined dimensions is done for the purpose of removing those dimensions, or modifying their boundaries. Both cases, removing and modifying dimensions, lead to specify a subset of the points that comprise the original grid. To the contrary, new dimensions are added to specify a superset of the points that form the original grid. Operators are used to apply the modifications to the grid based on the dimension expressions. Dimension expressions can be either a simple identity of an already defined dimension, and this can be used to reduce the dimensionality by removing that dimension, or using an identity with further information regarding the boundaries of the dimension, which can be used to add a new dimension or modify an existing one.

To summarize the operators that can be used to specify grid points in an iterator statement, our language extensions use the three operators

- $*$ to add a new dimension
- / to remove a defined dimension
- | to modify a defined dimension

To add a new dimension, the expression $dim\_name\{lower\_boundary..upper\_boundary\}$ is used after the $*$ operator. To drop an existing dimension, the identity of the dimension is added after the / operator. Modifying an existing dimension can be done in four forms:

- $|\{new\_lower_boundary..\}$
- $|\{..new\_upper\_boundary\}$
- $|\{new\_lower\_boundary..new\_upper\_boundary\}$
- $|\{one\_value\_dimension\}$

The first two cases allow to redefine one of the boundaries of the dimension. The third allows to modify both boundaries. And the last allows to reduce the whole dimension to one value. Notice that this reduction is not equal to removing the dimension which reduces the dimensionality of the traversed grid points. The reduction here is useful for some cases, e.g, traversing a specific layer (vertical level) of the 3D grid.

Different example expressions are shown in Listing 5.24.

Listing 5.24: **Example iterators with grid expressions using operators**

```
1 foreach cell in CELL2Dgrid * height {0 .. max_height}
2 ...
3
4 foreach cell in CELL3Dgrid / height
5 ...
6
7 foreach cell in CELL3Dgrid | height {5}
8 ...
9
10 foreach cell in CELL3Dgrid | height {2 .. max_height-2}
11 ...
```

In the first iterator statement, an expression is used with a predefined two dimensional grid, to which a new dimension is added, forming a three dimensional grid. In the second iterator, the contrary happens that the iterator traverses the projection of the 3D grid onto a surface grid. The third iterator statement traverses exactly one level (at vertical level # 5) of the 3D grid. The last traverses an inner part of the 3D grid (ignoring most lower and most upper two levels).

5.3.2.2. Access Operators

Iterators apply body statements at a set of specified grid points. To refer to the current grid point where a body statement is being applied, iterator statement includes an ID that developers provide. This ID can be used within the body statement to refer to the field values which the computation includes. Simply, using the ID as an index with any field, means that the value of that field that corresponds to the current grid point is to be accessed.

Referring to current grid point is not sufficient to write stencil computations. Stencils include spatially related points. Therefore, an expression that represents a stencil includes access not only to the current grid point of the iterator, but also to other neighboring points. In fact, those neighboring points can even in some applications be from other

sets of points, which are not included in the set of grid points that the iterator statement traverses, e.g, accessing edges of the cell in an iterator statement that traverses grid cells.

To access field data given those considerations, spatial relationships are represented by special language extensions; access operators. Those language extensions allow access to grid points with respect to current iterator point. Thus, they are modifiers applicable to iterator indices, on which they apply specific operations to access their neighborhood.

The variability of grid usage, which includes a wide range of possibilities, makes specifying a fixed set of such access operators a restriction that limits supporting different applications in the domain science. Therefore, in our specifications we provide rules that allow developers to define the access operators that their applications need. The syntax used when using an access operator in the source code should conform to the following:

- access operator name
- followed by parentheses, which can either be
  - empty, or
  - include one or more comma-separated additional operands

Specifying rules to use access operators allows to deal with different types of grids in a way that fits the needs of subject applications. This allows to fulfill the requirement **RFAN** (Section 5.1). So far, we mentioned the syntax guidelines that govern the use of access operators. With those syntax guidelines, only rules to use access operators are specified. The actual specifications of the access operators are defined as follows:

- Application developers decide which grids will be used in their solution
- Developers also decide what stencils will be used in code
- Analyzing the stencils, the spatial relationships of the grid points that form the stencil with respect to the stencil's computation point can be identified
- An access operator extension is added corresponding to each spatial relationship
- A suitable name is given to each access operator to reflect the spatial relationship
- The spatial distribution of points that have the same relationship to the stencil's computation point leads to specify the set of operands, e.g, a cell has multiple neighboring cells and we need to distinguish each neighboring cell.

Note that specifying names and operands are flexible, that different alternatives can be used even for the same grid type. To clear this flexibility let's take an example: The four neighboring cells of a cell in a rectangular grid can be accessed with four access operators without operands; *east*, *north*, *west*, and *south*. But also, an access operator with an operand that represents the number of the neighbor from 1 to 4 represents an alternative. Also, an access operator with two operands that represent offsets in X and Y directions from the central cell is another alternative.

The following snippet (Listing 5.25) shows a set of expressions using different access operators:

Listing 5.25: **Example access operators**

```
// empty paranthesis
C:       cell.east()
Fortran: cell%east()

// one further operand, where neighbor cells are numbered
c:       cell.neighbor(1)
Fortran: cell%neighbor(1)

// two further operands, to use offsets in both dimensions
c:       cell.XYoffset(1,1)
Fortran: cell%XYoffset(1,1)

// refer to a point outside the scope of the iterator
// grid point set
c:       cell.east_edge()
Fortran: cell%east_edge()
```

In the first expression, an access operator with empty parentheses is used o refer to a neighboring cell, which is another point on the same grid. The second expression shows an access operator with an operand, where a set of neighboring points on the same grid are numbered. This can be used to refer to neighbors in different types of grids, e.g, triangular, rectangular, hexagonal tessellations. The third expression shows another example with two operands within the parenthesis, which are used as offsets to the current X and Y coordinates in a rectangular grid. The last example expression shows an operator that is used to access an edge of the current cell. This example demonstrates the case when an iterator traverses the set of points at the cell centers, in which the access operator allows to access a point that does not belong to that set, where we refer to the edge from the current cell.

5.3.2.3. Reduction Expression

Stencils in general refer to multiple surrounding points around the current grid point. Normally those surroundings have a common spatial relationship to that point, with a slight difference. Such references can be done using one access operator, but with different values for an operand, e.g, multiple neighboring cells can be accessed using $cell.neighbor(1)$, $cell.neighbor(2)$ ... To support coding a stencil expression that includes a set of such expressions that are repeatedly used with the operand changing, we provide a language extension. This extension is a new type of expression that extends the set of expression types which the host language provides. However, it is equivalent to a compound expression that includes a set of simpler expressions. But also, this kind of expression can itself be nested. The syntax of this expression is shown in Listing 5.26.

Listing 5.26: **The syntax of the REDUCE expression**

```
REDUCE(operator,range={from..to},sub_expression)
```

It starts with the added keyword **REDUCE**, with parentheses, where three parts are provided. First part is an operator that will be applied to operate on the repetition of the *sub_expression*. Second part is the repetition range, e.g, neighboring cells are numbered 1 to 3 in a triangular grid. Last part is the *sub_expression* that will be repeated, which will be written using the repetition range identifier.

An example code is provided in Listing 5.27 to demonstrate the use of the **REDUCE** expression.

Listing 5.27: **An example REDUCE expression**

```
foreach cell in grid
{
  new_value[cell] =
      0.4 * old_value[cell] +
      0.6 * REDUCE( + , N={0..2},
                    old_value[cell.neighbor(N)]);
}
```

The **REDUCE** expression uses the + operator to return a summation value. The N range identifier ranges from 0 to 2. The *sub_expression* is the value of a field (*old_value*) at the cell's neighbor $\#N$. This means that the value of that field is read at the three neighbors and the sum will be returned as the value of the **REDUCE** expression. The specifications of the language extensions do not impose restrictions on the actual generated code corresponding to the REDUCE expression. However, computed results should be equal to those of explicit expressions, e.g., the three terms summed with the + operator explicitly.

Chapter summary

We discussed the requirements that accompanied the development and specification of the language extensions. Then, we discussed a set of sample stencils from different models to better understand the needs of the applications. The choice of the stencils allows to work with different kinds of problems to achieve a more comprehensive solution. We concluded with the specifications of the language extensions, which conform to the listed requirements.

In the next chapter, we demonstrate the use of our language extensions to develop a set of mathematical operators and a solver application.

6. Coding with GGDML

In this chapter, we present two applications that are developed using the GGDML language extensions. The two applications are intended to study different stencils and grids. This allows to show the use of our language extensions under different application needs, and hence, to investigate the extensions adaptability to the application needs for the purpose of answering the questions of this thesis. The first application (Section 6.1) is a prototype that comprises a set of mathematical operators. It demonstrates the use of GGDML to write such operators. The second application (Section 6.2) is a complete application implementing a solver for the shallow water equations. It demonstrates the use of GGDML to write a full model. Later, these codes are used conduct the performance experiments and analysis.

6.1. Basic Operators on an Icosahedral Grid

This prototype application is developed based on concepts from the ICON model, which uses an unstructured icosahedral grid. Triangular tessellation is used to discretize the surface. With our language extensions, the kernels do not reflect such grid details. However, this code allows to investigate using those high-level kernels with existing grid structures and connectivity data structures of those unstructured grids. Such environment allows to understand porting existing application code incrementally to use the language extensions.

6.1.1. Structure

The code is run in time steps. The main loop advances the simulation one time step per iteration, during which the components of the model are called to execute their parts of the simulation as shown in Listing 6.1.

Listing 6.1: **Main time-stepping loop**

```
1     uint64_t ts;          // time steps
2     double dt = 0;        // time spent
3     double st = mtime(); // start time
4     timestep(ts=0;
5              ((ts < timesteps) || (dt < mintime)) && dt < maxtime;
6              ts++){
7         // Let each component (com1-com5) do its
8         // computations for the current time step
9         com1.compute(g);
10        com2.compute(g);
```

```
11          com3.compute(g);
12          com4.compute(g);
13          com5.compute(g);
14
15          dt = mtime() - st;
16      }
```

The modeling code is developed within a set of components, each of which represents simulating some natural process. A data structure is used to encapsulate the description of the component (Listing 6.2). This data structure includes the functions that initialize the component, do the computations that the component contributes to the simulation, handle the necessary I/O operations, and cleanup after the simulation is finished. Other functions allow to return the size of the memory allocated for the variables that the component allocates, and the number of the floating point operations that the component executes in each call to its computation function. Also a validation function is included within the component structure for testing purposes.

Listing 6.2: **Component data structure**

```
1  typedef struct {
2      int loaded;
3      void (*init)(GRID*);
4      void (*compute)(GRID*);
5      void (*io)(GRID*);
6      double (*flops)(GRID*);
7      double (*memory)(GRID*);
8      uint64_t (*checksum)(GRID*);
9      void (*cleanup)(GRID*);
10 } MODEL_COMPONENT;
```

6.1.1.1. Component Initialization

When the application starts up, the set of the components that we intend to run during the simulation are initialized by calling the initialization function of each of those components. In this step, each loaded component allocates and loads the initial values of its variables.

The fields are declared within the components through the DSL specifiers. Listing 6.3 illustrates field declarations from one of the components.

Listing 6.3: **Field declaration**

```
1 GVAL CELL 3D gv_temp;
2 GVAL EDGE 3D gv_grad;
3 GVAL CELL 3D gv_dvg;
```

The allocation of a field is done within the containing component initialization function. DSL constructs allow to express within the source code that a field is to be allocated. Tools that process the language extensions will use the information carried by the DSL constructs to generate the allocation code for that field.

The initialization of the variables is done through NetCDF within the component initialization phase. The DSL allows to use higher-level constructs for the I/O operations. Translation tools can also generate data structures for the variables to expose the attributes of the variables to give the programmers more flexibility within the source code. An example component initialization function is shown in Listing 6.4.

Listing 6.4: **An example component initialization**

```
1  void com1_init(GRID* g)
2  {
3      com1.loaded = 1;
4      ALLOC gv_temp;
5      ALLOC gv_grad;
6      ALLOC gv_dvg;
7
8      io_read_register(g, "gv_temp", (GVAL *)gv_temp, FLOAT32,
9                       FLOAT32, GRID_POS_CELL, GRID_DIM_3D);
10     io_write_define(g, "gv_temp", (GVAL *)gv_temp, FLOAT32,
11                     GRID_POS_CELL, GRID_DIM_3D, &io_gv_temp);
12     io_write_define(g, "gv_grad", (GVAL *)gv_grad, FLOAT32,
13                     GRID_POS_EDGE, GRID_DIM_3D, &io_gv_grad);
14     io_write_define(g, "gv_dvg",  (GVAL *)gv_dvg,  FLOAT32,
15                     GRID_POS_CELL, GRID_DIM_3D, &io_gv_dvg);
16 }
```

6.1.1.2. Component Computations

Within the main time step loop, the *compute* functions of the components that the user wants to run within the simulation are called. In a component's *compute* function, the component calls the set of kernels that it should execute.

The kernels are provided in their own files. They are written using the DSL language extensions besides to the general-purpose language C. An example *compute* function is shown in Listing 6.5.

Listing 6.5: **An exmple component compute function**

```
1  void com1_compute(GRID* g)
2  {
3      grad(g);
4      dvg(g);
5      step(g);
6  }
```

6.1.1.3. I/O Operations

The test application uses NetCDF to store and read field data. During the component initialization (within application initialization phase), the initialization functions allow reading field initial values and registering fields for output operations. At specific time

steps, according to developers' decisions, the main loop of the application calls the I/O handler of the components that the user wants to write out their output. When the I/O function of a component is called, the component calls the NetCDF API to write the data of the fields that are registered for output. An example function to handle I/O operations within a component is shown in Listing 6.6.

Listing 6.6: **Example component I/O operations**

```
1 void com1_io(GRID* g){
2     io_write_announce(g, &io_gv_grad);
3     io_write_announce(g, &io_gv_dvg);
4 }
```

6.1.2. Operators

A set of kernels are developed to implement the functionality of the component set that comprises the application. The kernels are developed with C and the GGDML language extensions.

The set of kernels covers a set of representing modeling cases. Both two- and three-dimensional grids are used. Different kernels that traverse the grid's cells and another set of kernels that traverse the grid's edges are provided. Accessing the neighboring grid components in different cases is also demonstrated. Vertical and horizontal neighborhoods are demonstrated, e.g, above cell, cell's neighboring cells, cells sharing an edge ...

6.1.2.1. A Vertical Integration Kernel

In the next code snippet in Listing 6.7 we show a kernel that performs vertical integration. The code is written using the GGDML iterator. The iterator traverses the cells of the three dimensional grid.

The *gv_vi* field is a two-dimensional field that covers the horizontal grid of the surface. The field *gv_temp* is located at the centers of the cells of the three-dimensional grid. The kernel sums the the values of the *gv_temp* field over a whole column into a value of the output horizontal field *gv_vi*, which represents a projection of the 3D grid on the 2D surface.

Listing 6.7: **Vertical integration**

```
1     foreach cell in grid
2     {
3         gv_vi[cell] += gv_temp[cell];
4     }
```

Translation tools will know from the declaration of the variables whether they are located at the cells of the whole 3D grid or on the 2D surface. This allows the tools to generate the right code automatically.

6.1.2.2. A Divergence Kernel

The following code snippet in Listing 6.8 shows a kernel to compute the divergence of a field. The *foreach* iterator traverses the cells of the three dimensional grid. The *gv_grad* field is located on the edges of the grid. It represents the gradient of some field (*gv_temp*), the values of which are updated in other kernels. The *gv_dvg* field is the divergence value and is located at the centers of the grid cells. The values of the divergence field are calculated based on the recorded gradient field values. The language extensions (*cell.edge*(*n*)) allow to simplify the access to the edges of the traversed cells.

Listing 6.8: **Divergence**

```
1     foreach cell in grid
2     {
3         gv_dvg[cell] = REDUCE( + , N={0..2},
4                         gv_grad[cell.edge(N)] * div_coef[N][cell]
5                         );
6         }
7     }
```

6.1.2.3. A Gradient Kernel

In the following code in Listing 6.9, we compute the gradient of the *gv_temp* field. The iterator traverses the edges of the three dimensional grid. On each edge, the value of the gradient field *gv_grad* is computed based on the values of the *gv_temp* field at the centers of the cells that share the edge. The language extensions (*edge.cell*(*n*)) simplify the access and abstract the reference to the cells that share and edge.

Listing 6.9: **Gradient**

```
1     foreach edge in grid
2     {
3             gv_grad[edge] = gv_temp[edge.cell(0)] -
4                             gv_temp[edge.cell(1)] ;
5     }
```

6.1.2.4. A Laplacian Kernel

The kernel in Listing 6.10 computes the Laplacian of the *gv_temp* field. The iterator traverses the cells of the three dimensional grid. At the center of each cell, a new value of the *gv_temp* field is computed based on the values of the *gv_temp* field at the centers of the neighboring cells. The language extensions (*edge.neighbor*(*n*)) simplify the access and abstract the reference to the neighboring cells.

Listing 6.10: **Laplacian**

```
1     foreach cell in grid
2     {
```

```
3           gv_temp_alt[cell] = gv_temp[cell] * lap_coef[3][cell] +
4                   gv_temp[cell.neighbor(0)] * lap_coef[0][cell] +
5                   gv_temp[cell.neighbor(1)] * lap_coef[1][cell] +
6                   gv_temp[cell.neighbor(2)] * lap_coef[2][cell] ;
7       }
```

6.1.2.5. A 3D Variable Weighted by a Horizontal Parameter

In the code in Listing 6.11 the declarations show that the field *gv_temp*, which is used within the computation of the kernel, is defined at the cell centers of the three-dimensional grid. The *gv_outvar* field is defined over the edges of the three-dimensional grid. A parameter variable *gv_ind2Dparam* is defined over the surface cell centers. The kernel iterates the cells of the three-dimensional grid. It computes the output field *gv_outvar* on the grid edges based on the *gv_temp* values at the cells that share the edge. The kernel uses the parameter values, which are the same for all the cells in each column, as weights for the computation.

Listing 6.11: **Horizontally weighted fields**

```
1  extern GVAL CELL 3D gv_temp;
2  extern GVAL CELL 2D gv_ind2Dparam;
3  extern GVAL EDGE 3D gv_outvar;
4
5  ...
6
7       foreach edge in grid
8       {
9               gv_outvar[edge] = gv_ind2Dparam[edge.cell(0)] *
10                                gv_temp[edge.cell(0)]       -
11                                gv_ind2Dparam[edge.cell(1)] *
12                                gv_temp[edge.cell(1)]       ;
13      }
```

6.1.2.6. Vertical and Horizontal Neighbors and Different Kinds of Parameters

The following code in Listing 6.12 includes two kernels, one updates the values of the field *gv_o8var* over the whole grid except two vertical levels, and the other updates the values of the same field for the last vertical level only.

The kernels access two fields: *gv_grad* and *gv_o8var*. The field *gv_grad* is located at the edges of the 3D grid, and the field *gv_o8var* is located at the cell centers of the 3D grid.

Two parameters are used within the kernels: *gv_o8param* and *gv_o8par2*. Every cell on the surface has three values for the parameter *gv_o8param*. Those values are used as weights for the edges of the cell. The values are used allover the column cells in the 3D grid. The other parameter (*gv_o8par2*) has one value per cell over all the 3D grid.

In both kernels we use the DSL 'REDUCE' expression to represent the sum over the edges of a cell. We also used the $cell.edge(n).below()$ and $cell.edge(n).below(m)$ to allow access from a cell to the edges in the vertical levels below the cell.

Listing 6.12: **Computations using 2D and 3D neighbors**

```
extern GVAL EDGE 3D gv_grad;
extern GVAL CELL 2D gv_o8param[3];
extern GVAL CELL 3D gv_o8par2;
extern GVAL CELL 3D gv_o8var;

...

    freach cell in grid|height{1..(g->height-1)}
    {
        GVAL v0 = REDUCE( +,N={0..2},
                          gv_o8param[N][cell]    *
                          gv_grad[cell.edge(N)]
                        );

        GVAL v1 = REDUCE( +,N={0..2},
                          gv_o8param[N][cell]     *
                          gv_grad[cell.edge(N).below()]
                        );

        gv_o8var[cell] = gv_o8par2[cell] * v0    +
                         (1-gv_o8par2[cell]) * v1;
    }

    foreach cell in grid|height{(g->height-1)..(g->height-1)}
    {
        GVAL v0 = REDUCE( +,N={0..2},
                          gv_o8param[N][cell]    *
                          gv_grad[cell.edge(N)]
                        );

        GVAL v1 = REDUCE( +,N={0..2},
                          gv_o8param[N][cell]    *
                          gv_grad[cell.edge(N).below()]
                        );

        GVAL v2 = REDUCE( +,N={0..2},
                          gv_o8param[N][cell]    *
                          gv_grad[cell.edge(N).below(2)]
                        );

        gv_o8var[cell] = 0.4 * v0 + 0.3 * v1 + 0.3 * v2;
    }
```

6.2. Shallow Water Equation Solver

The second application[1] solves the shallow water equations (SWE) on a 2D regular grid. The application uses an explicit time stepping scheme in which all eight fields are updated once in each time step.

In this modularized code, every kernel includes the necessary mathematical operations and expressions to update exactly one field. This code is easy to understand and maintain, and includes eight kernels updating:

- the two components of the flux: (the kernels are) $flux1$ and $flux2$,
- the tendencies of the two components of the velocity: *compute_u_tendency* and *compute_v_tendency*,
- the tendency of the surface level: *compute_h_tendency*,
- the two components of the velocity: *update_u* and *update_v*,
- and the surface level: *update_h*.

6.2.1. Structure

The source code of this application includes declaration of the fields using basic GGDML declaration specifiers. The code includes allocation and deallocation of those fields, however, again this is done through GGDML constructs. Eight kernels apply stencils to update some of the fields using GGDML language extensions.

6.2.1.1. fields

Shallow water equations describe fluid velocities and surface level over time under a set of assumptions. A set of fields are necessary to represent the different quantities that describe the system. Mainly, fields to represent the fluid surface level and velocities in both X and Y directions are needed. Other fields are necessary to measure the depth of the average surface level which is constant over time, and fluxes and tendencies of the surface level and velocities which are computed in each time step to help computing the main fields.

All the fields are measured using single precision floating point. The problem is a two-dimensional problem, thus we use the basic 2D GGDML dimensionality declaration specifier to declare all fields. Our solution localizes surface level measurement at the centers of the grid cells. Velocities and fluxes are localized on the edges that separate the grid cells. Other fields follow those localizations as needed. Localization is also described by the basic GGDML localization declaration specifiers, i.e, $CELL$ and $EDGE$. Field declarations are shown in Listing 6.13.

[1] Full code is available at `https://github.com/aimes-project/ShallowWaterEquations` and in Appendix A

Listing 6.13: **Field declaration**

```
float CELL 2D f_H;
float CELL 2D f_HT;
float EDGE 2D f_U;
float EDGE 2D f_UT;
float EDGE 2D f_V;
float EDGE 2D f_VT;
float CELL 2D f_B;
float EDGE 2D f_F;
float EDGE 2D f_G;
```

The declarations specify the different application fields. However, the declarations do not do the real memory allocations for the fields data. Declaration specifiers, from both C and GGDML, convey information regarding the data types and spatial characteristics of the fields. This information allow tools to further deal with those variables. One of the further steps to deal with the fields is allocating memory to store their data.

GGDML language extensions provide a construct that allows developers to decide when to allocate each field. After the application finishes the simulation, memory allocated for different purpose are usually deallocated. GGDML also allows developers to decide in code when to deallocate memory that was allocated for the different fields in the application. Those *ALLOC* and *DEALLOC* constructs mark the real scope of use of the fields within the application. Listing 6.14 shows the allocations and deallocations of fields (one field is shown for simplicity).

Listing 6.14: **Field allocation/deallocation**

```
    // Allocate memory before simulation starts
    ALLOC f_H;
    ...

...
    // Deallocate memory after simulation finishes
    DEALLOC f_H;
    ...
```

6.2.1.2. Time-Stepping Loop

The main code that runs the simulation is a time-stepping loop. After the fields are allocated and can be used and are initialized with initial values, the different simulation kernels can be executed. Deallocation can then be done somewhere after the time-stepping loop, when the simulation is finished, but also after all operations regarding the fields are done, e.g, writing to storage.

In this application we run the simulation for a specific number of time steps. In each time step, we compute the two flux components by calling a function *compute_flux*. We then call the function *update_values* which calls the different functions to compute the tendencies and update the velocities and surface level. Listing 6.15 shows the time-stepping loop of the application. The *timestep* is just a loop construct, which is similar to

for loop in C language. However, it allows to do further processing, e.g, instrumentation markup for profiling tools.

Listing 6.15: **Time-stepping loop**

```
1    int time_step = 0;
2     double run_time = time_sec();
3
4     //time-stepping loop
5     timestep(time_step = 0;
6              time_step < TIMESTEPS;
7              time_step++) {
8
9         // Compute flux
10        compute_flux();
11
12        // Compute tendencies and update values
13        update_values();
14    }
```

6.2.2. Kernels

The simulation is executed in eight kernels which are written using the GGDML iterator language extension. Different access operators are defined to specify the different stencils, as a result of using different neighboring grid elements. Among those access operators are operators to access cells around edges, and different operators to access edges around cells.

Three kernels update fields that represent the main simulation quantities, i.e, surface level and velocities. The other kernels compute helping fields, which are essential to compute the main simulation quantities.

6.2.2.1. Flux

The first function that the time-stepping loop calls computes the flux, both X and Y components (refer to Listing 6.16). The X-dimension component is represented by the field f_F. This field is localized on the edges of the grid. A kernel computes this field based on the values of the surface level, f_H, at the centers of the two cells around the edge. Two access operators, i.e, *east_cell* and *west_cell*, are used to refer to those two cells that share the edge.

Same is true for the Y-dimension component of the flux, f_G, which is computed in its own kernel. Two other access operators, *north_cell* and *south_cell*, are defined to access the cells sharing the edge. Access operators differ to support the different grid elements. The X-component is computed based on averaging two cells in the X-direction, while the Y-component is computed by averaging two cells in the Y-direction.

The other input field for the flux computation besides to the surface level, f_H, is the value of the velocity. X-dimension velocity is used to compute the X-component of the

flux, and Y-dimension velocity is used to compute the Y-component of the flux. Velocity fields are localized on the edges where we compute also the flux, therefore we access them directly with the iterator index without the need to use any access operators.

Listing 6.16: **compute_flux**

```
void compute_flux()
{
    // Compute the flux component in the X dimension
    foreach e IN grid {

        // Use GGDML access operators east_cell & west_cell
        //  to refer to the cells sharing the edge
        f_F[e] = f_U[e] * (f_H[e.east_cell()] +
                           f_H[e.west_cell()]) / 2.0;
    }

    // compute the flux component in the Y dimension
    foreach e in grid {

        // Use GGDML access operators north_cell & south_cell
        // to refer to the cells sharing the edge
        f_G[e] = f_V[e] * (f_H[e.north_cell()] +
                           f_H[e.south_cell()]) / 2.0;
    }
}
```

6.2.2.2. Velocity Tendencies

To update the velocities, we compute first the tendencies in both dimensions. This is done in two kernels, each of which is coded in its own function. Both functions are called by the *update_values* function which is called in the main time-stepping loop.

In the first function, *compute_U_tendency* (Listing 6.17), a GGDML iterator updates the tendency of the X-direction velocity. In the iterator, local variables and C-language operators are used to compute different expressions that contribute values to update the velocity tendency. The difference is the use of GGDML indices using user-defined access operators.

Listing 6.17: **compute_U_tendency**

```
void compute_U_tendency()
{
    // Compute different terms of tendency
    // Use GGDML iterator to traverse the edges where
    //  the U-Tendency is located
    foreach e in grid {

        // Use GGDML access operators edge_????_neighbor
        // to refer to the neighboring U edges in X direction
        float udux = f_U[e] * (f_U[e.edge_east_neighbor()] -
```

```
11                                  f_U[e.edge_west_neighbor()])
12                        / (2.0 * dx);
13
14          // Use GGDML access operators edge_??_neighbor
15          // to refer to the neighboring V edges
16          float vbar = (f_V[e.edge_ne_neighbor()] +
17                        f_V[e.edge_nw_neighbor()] +
18                        f_V[e.edge_se_neighbor()] +
19                        f_V[e.edge_sw_neighbor()]) / 4.0;
20
21          // Use GGDML access operators edge_v?????_neighbor
22          // to refer to the neighboring U edges in Y direction
23          float vduy = vbar * (f_U[e.edge_vnorth_neighbor()] -
24                               f_U[e.edge_vsouth_neighbor()])
25                        / (2.0 * dy);
26
27          // Use GGDML access operators east_cell & west_cell
28          //  to refer to the cells sharing the edge
29          float gdhbx = g * (f_H[e.east_cell()] +
30                             f_B[e.east_cell()] -
31                             f_H[e.west_cell()] -
32                             f_B[e.west_cell()]) / dx;
33
34          float fvbar = f * vbar;
35          f_UT[e] = fvbar - udux - vduy - gdhbx;
36      }
37  }
```

The term *udux* is computed on U-edges, based on differences of values of the X-direction velocity field *f_U* between the neighboring U-edges in X direction. For this purpose we define the access operators *edge_east_neighbor* and *edge_west_neighbor*.

The term *vbar* is computed on U-edges by averaging the Y-direction velocity field *f_V* on the four surrounding V-edges. To refer to those four neighboring edges we define the four access operators *edge_ne_neighbor*, *edge_nw_neighbor*, *edge_se_neighbor*, and *edge_sw_neighbor*.

The term *vduy* is computed on the U-edges based on differences of the X-direction velocity field *f_U* on the neighboring U-edges in the Y direction. Therefore, we define the access operators *edge_vnorth_neighbor* and *edge_vsouth_neighbor*.

The term *gdhbx* is computed on the U-edges based on the actual surface level at the centers of the two cells sharing the U-edge. Hence, we define the access operators *east_cell*, and *west_cell*.

Finally, all those terms are used together in C-language expressions to compute the velocity tendency in the X direction. The computed value is stored in the *f_UT* field.

The values *dx* and *dy* are application-level constant values that define the discretization of the space in both X and Y dimensions. They are C-language floating point variables initialized with values to control the finite number of grid points that correspond to the problem domain.

Another function, *compute_V_tendency*, is used to compute the tendency of the Y

direction velocity (refer to Listing 6.18).

Listing 6.18: **compute_V_tendency**

```
void compute_V_tendency()
{
    // Compute different terms of tendency
    // Use GGDML iterator to traverse the edges where
    //  the V-Tendency is located
    foreach e in grid {
        // Use GGDML access operators edge_?????_neighbor
        // to refer to the neighboring V edges in Y direction
        float vdvy = f_V[e] * (f_V[e.edge_north_neighbor()] -
                     f_V[e.edge_south_neighbor()])
                     / (2.0 * dy);

        // Use GGDML access operators edge_??_neighbor
        // to refer to the neighboring U edges
        float ubar = (f_U[e.edge_en_neighbor()] +
                      f_U[e.edge_es_neighbor()] +
                      f_U[e.edge_wn_neighbor()] +
                      f_U[e.edge_ws_neighbor()]) / 4.0;

        // Use GGDML access operators edge_h????_neighbor
        // to refer to the neighboring V edges in X direction
        float udvx = ubar * (f_V[e.edge_heast_neighbor()] -
                             f_V[e.edge_hwest_neighbor()])
                     / (2.0 * dx);

        // Use GGDML access operators north_cell & south_cell
        //  to refer to the cells sharing the edge
        float gdhby = g * (f_H[e.north_cell()] +
                           f_B[e.north_cell()] -
                           f_H[e.south_cell()] -
                           f_B[e.south_cell()]) / dy;

        float fubar = f * ubar;
        f_VT[e] = 0.0 - vdvy - udvx - gdhby - fubar;
    }
}
```

The term *vdvy* is computed on V-edges, based on differences of values of the Y-direction velocity field *f_V* between the neighboring V edges in Y direction. For this purpose we define the access operators *edge_north_neighbor* and *edge_south_neighbor*.

The term *ubar* is computed on V-edges by averaging the X-direction velocity field *f_U* on the four surrounding U-edges. To refer to those four neighboring edges we define the four access operators *edge_en_neighbor*, *edge_es_neighbor*, *edge_wn_neighbor*, and *edge_ws_neighbor*.

The term *udvx* is computed on the V-edges based on differences of the Y-direction velocity field *f_V* on the neighboring V-edges in the X direction. Therefore we define the access operators *edge_heast_neighbor* and *edge_hwest_neighbor*.

The term *gdhby* is computed on the V-edges based on the actual surface level at the centers of the two cells sharing the V-edge. Hence we define the access operators *north_cell*, and *south_cell*.

Finally, all those terms are used together in C-language expressions to compute the velocity tendency in the Y direction. The computed value is stored in the f_VT field.

6.2.2.3. Velocities

After the tendencies of the velocities are computed, the velocities can be computed based on the updated values of velocity tendencies. Two functions, *update_U* and *update_V*, update the velocity in both directions. They are called by the *update_values* function which is called in the main time-stepping loop.

The first function, *update_U*, updates the X-direction velocity(Listing 6.19).

Listing 6.19: **update_U**

```
1 void update_U()
2 {
3     foreach e in grid {
4         f_U[e] = f_U[e] + f_UT[e] * dt;
5     }
6 }
```

The current value of the X-direction velocity, i.e, f_U, is modified by adding the tendency in the X-direction velocity f_UT multiplied by the time increment dt. The time increment is an application-level constant value that is defined as a C-language floating point variable.

The velocity in the Y direction is similarly computed in the function *update_V* (refer to Listing 6.20).

Listing 6.20: **update_V**

```
1 void update_V()
2 {
3     foreach e in grid {
4         f_V[e] = f_V[e] + f_VT[e] * dt;
5     }
6 }
```

Similarly, the current value of the Y-direction velocity, i.e, f_V, is modified by adding the tendency in the Y-direction velocity f_VT multiplied by the time increment dt.

6.2.2.4. Surface Level Tendency

To compute the surface level, f_H, in each time step, we first compute the tendency of the surface level (f_HT). A function, *compute_H_tendency*, executes a kernel to compute this tendency field (refer to Listing 6.21). This function is called by the *update_values* function which is called in the main time-stepping loop. The kernel traverses the cells of the grid.

Listing 6.21: **compute_H_tendency**

```
void compute_H_tendency()
{
    // Compute the two terms of tendency
    // Use GGDML iterator to traverse the grid cells
    foreach c in grid {

        // Use GGDML access operators east_edge & west_edge
        //  to refer to the U edges of the cell
        float df = (f_F[c.east_edge()] -
                    f_F[c.west_edge()]) / dx;

        // Use GGDML access operators north_edge & south_edge
        //  to refer to the V edges of the cell
        float dg = (f_G[c.north_edge()] -
                    f_G[c.south_edge()]) / dy;

        f_HT[c] = df + dg;
    }
}
```

Two terms, df and dg, are computed within the kernels to find the contribution from both X and Y components of the flux. The term df is computed at the cell centers based on the difference between the X-component of the flux on the two U-edges surrounding the cell. Likewise, the term dg is computed at the cell centers based on the difference between the Y-component of the flux on the two V-edges surrounding the cell. The sum of the two contributions is stored in the field f_HT.

6.2.2.5. Surface Level

After the surface level tendency is computed, the surface level , f_H, can be updated. A function, $update\_H$, executes a kernel to compute this field (refer to Listing 6.22). The function $update\_H$ is called by the $update\_values$ function which is called in the main time-stepping loop.

Listing 6.22: **update_H**

```
void update_H()
{
    // Update the surface level
    // Use GGDML iterator to traverse the grid cells
    foreach c in grid {
        f_H[c] = f_H[c] - dt * f_HT[c];
    }
}
```

The kernel traverses the cells of the grid, updating the value of the surface level f_H by adding the surface level tendency f_HT multiplied by the time increment dt.

Chapter summary

In this chapter, we demonstrated the use of GGDML to develop well-known mathematical operators in a prototype application. We also demonstrated the use of GGDML through writing a full model (a shallow water equation solver). We illustrated the flexibility of modeling by adapting the extensions to support different applications with different needs. We did this through two applications using different grids, and different stencil shapes and neighborhoods. For each of the two applications, different extensions that best fit the grids and stencils were developed.

The first prototype application includes a set of frequently-needed mathematical operators. The gradient, divergence, and Laplacian operators happen so frequently in simulation code. Coding those operators using GGDML reflects the applicability of the language extensions in real application. Using the GGDML language extensions illustrates the reduced effort and the clearness of codes that compute basic operators, which form a large part of real modeling software. The use of the user-defined access operators that reflect the spatial relationships directly as needed by the applications, i.e., fitting the grids that are used within the application, simplifies much for scientists coding the stencils that compute the operators. Simplifying coding of stencils reduces the effort and time spent by scientists on coding the mathematical operators that their models execute.

The shallow water equation solver code demonstrates the use of GGDML language extensions to develop a complete application. Full models with different processes comprise hundreds of thousands of lines of code. Time restrictions to write so long codes limited us to shorter codes. Furthermore, for demonstration purposes, long codes would not be a suitable tool to let readers understand the code and comprehend the concepts behind coding with our language extensions. Therefore, we chose this problem as it is a small problem with few fields, but sufficient to demonstrate concepts. The code demonstrates the use of a variety of stencils including fields at cell centers and fields on edges between cells, applying different operations. The whole solver code with GGDML is less than 300 lines of code.

In the next chapter, we discuss the translation process of high-level code.

7. Code Translation

In this chapter, we discuss the translation of high-level code that is developed with GGDML. We start with describing the design drivers that guide the design of the translation process in Section 7.2. Next, we discuss the high-level design of this process in Section 7.3. We give a closer look at configuration files in Section 7.4.
In Section 7.5 we discuss and formulate the relationship between information extracted from source code and information fetched from configuration files, and the optimizations that use such inputs to allow improved performance. This in fact describes the 'what' question regarding the matching of inputs to optimize code, but the details on 'how' to apply the information to transform code into optimized scalable code are discussed in Section 7.6.

7.1. Translation Process

Higher-level code that is written using GGDML mixed with a host programming language must be translated into code that can be further processed by other tools, e.g, compilers. An important aspect of the translation process is the optimization to a specific target machine. GGDML language extensions hold an important role in transferring the necessary semantics of high-level code for the translation tools. The extensions enable and drive (partially) the optimization process by the tools.

Discussing the transfer of the semantical information that allows optimization, and how to exploit such information within tools is an important part of answering the main questions of this thesis. Important details and core concepts of our approach regarding the optimization process are covered in this chapter.

The top-level theme and research objectives of this work are concerned with new compiler concepts rather than conventional compiler techniques. Mainly, the thesis questions focus on application-adaptable user-defined language extensions. Such extensions allow more precise semantical content by application focus besides domain knowledge. However, designing techniques and tools to handle the processing of such language extensions demands more effort. Language flexibility requires that additional activities should be involved besides standard compilation procedures, e.g., parsing, optimization and code generation.

To cover clearly our methodological work on the development of the translation process

- We start with an overview of the main design requirements guiding the translation techniques and tools development. Those requirements represent a guarantee to achieve research goals.

- After knowing the main design guidelines of the translation process, we discuss the design itself at the high-level. Main phases of the whole translation process are introduced. This allows to have an image of the translation process before going further in the details of how configuration information are used and how optimization procedures are applied.
- Information that is fetched from configuration files play a key role in the translation process. Therefore, discussing the contents of configuration files before proceeding to their use introduces those contents.
- Now that the language extensions and their use in the source code are already known (from previous chapters) and the configuration files are discussed, we relate both inputs to the optimization process. In this step, we state the necessary semantical inputs from source code and inputs from configuration files that are necessary to apply each optimization aspect.
- After identifying the inputs necessary to apply each optimization procedure, the optimization procedures are explained. A set of algorithms describe how the inputs from source code and configuration files are used to apply the different optimization aspects.

Each point in this methodological coverage is done in own section in this chapter.

With the methodological work shown in this chapter, important aspects are made clear. The discussions show 1) how we can make use of application-specific semantics to optimize the application to exploit hardware features; 2) the adaptability of the language extensions and how to support such language flexibility; 3) how our techniques allow the users to define language extensions; 4) how tools support such dynamic language processing of added rules to language grammar; 5) how can tools use such additions for optimization purposes.

7.2. Design Drivers

In this section, we review the requirements that guide the development of the translation process and tools.

RFPD **Parse extended language**

Within source to source translation, parse source code that uses extended language to be further processed.

RFRS **Read source input files**

Read input source code from the models code repository to be parsed and processed during source-to-source translation.

RFRC **Read configuration files**

Read configuration files to handle source-to-source translation process.

RFGC **Generate target code**

Generate code for various targets based on the configuration files, which drive the translation process.

RFOP **Optimize code**

Apply the relevant optimization procedures and rewrite code in a way to get optimal performance on target machines.

RFWS **Write processed code trees**

After source code tree contents are processed the solution should write the processed form into an output code tree.

RNPP **Performance portability**

Support performance portability through configurable code transformations that optimize code per target machine.

RNCT **Compilation time**

Runtime of the tool must be in the order of the regular compile time.

RNFL **Flexibility**

Code translation should be flexible enough to allow adapting DSL components to application needs. Supporting user-defined language extensions leads to flexible tools that allow modifying language grammar. Rather than hard-coded functionality, configurable grammar processing should be supported.

RNTS **Tools simplicity**

The translation tools should be lightweight, compact, and simple to deal with. Scientists should not be concerned about using it. Despite its complex functionality, it should simply integrate into build systems, e.g, make.

RNTM **Tools maintainability**

Maintainability of the tools is an important issue for scientists. So, the translation tool should be designed as a file that ships with code repositories just like a script or a makefile.

7.3. High-Level Design

Higher-level code follows the grammar of a host general-purpose language and a set of additional rules corresponding to the language extensions. To translate higher-level code, a translation tool uses the rules of the grammars to parse the source code and builds AST structures. Different modules handle the standard host language rules and the additional rules. AST structures are the basis for optimization and code generation.

When the translation tool is called, it is given the source code tree, the modules to handle the language grammar, and a configuration file to guide the code translation process. The approach is illustrated in Figure 7.1:

- A first step that the main module of the tool makes is to load the language modules, which are used to parse source code according to the grammars of the DSL and the GPL.
- The configuration file is then loaded, and the language modules configure themselves according to the contents of that file.
- Next, the code files within the source code tree are read and parsed, and AST structures are generated.
- Optimizations are applied and code is generated into an output directory tree.

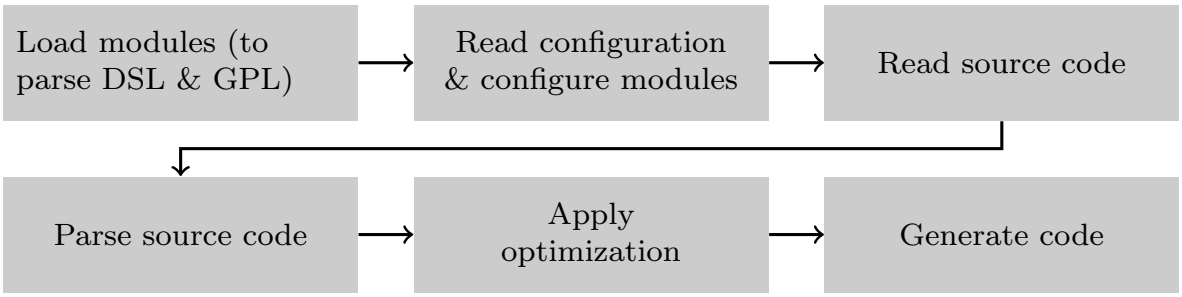


Figure 7.1.: Translation Process

7.3.1. Modular Structure

To handle the different rules, i.e, the original standard set of rules of the host language and the added extension rules, we utilize a translation process based on a modular structure. The main component of the translation tool calls specialized modules that process the different grammar rules. One module processes the rules of the host language. Another specialized module handles the additional rules that correspond to the language extensions.

The modular structure of the tool allows to separate the code of the tool according to the functionality. This simplifies the task of further development and maintainability of the tool. Supporting different host languages or DSL handlers can be easier.

Users use the translation tool by calling the main module of the tool. Language modules are passed with the call to the main module. The main module then refers to the language handler modules to process a given source code tree.

7.3.2. Configuring Code Translation

The dynamic nature of the grammar, which allows modifying the set of added rules, requires that the tool is able to process the added rules. Built-in hard-coded processing of the source code, which is used by conventional tools, cannot support such requirement.

To give the tools the capability to process added rules, a user (typically the scientific programmer) provides some information through configuration files. Those files are read at code translation time, when the translation tool is called. Any added rules are built within the tool at translation time based on this configuration information. This way, we can handle user-defined application-adaptable language extensions.

When a source code tree is to be translated, users pass a configuration file in the call to the tool's main module. The configuration information is passed then to the module that handles the added rules. Next, this module parses the contents of the configuration file and builds its internal data structures that will be used to handle the language extensions.

7.3.3. Source Code Trees

Users pass the source code that should be translated by the tool when calling the translation tool. A single file or alternatively the root of the folder that contains a source code tree can be passed. Folders under the source code tree are traversed recursively to process code files. Non-code files are copied as is, i.e, passed through, to the output code tree. The code files are processed and their processed versions are written to the output code tree. Thus, the output code tree is identical in its structure to the source code tree, with the difference of the code in code files. The generated trees are then ready to be further processed with conventional tools, e.g., built with 'make' to generate binaries.

7.3.4. Handling Grammars

Each code file that is loaded by the translation tool's main module is passed to the language handler modules. The code from a file is parsed by either the host language module, or the module that handles the language extensions. The host language handler processes code conforming to the standard rules of the host language, therefore, this module handles a constant well known set of rules. On the contrary, the extension handler must read configuration files to know the rules of the additional extensions.

During a call to the translation tool, after the language modules are loaded and a configuration file is chosen, the extension handler parses the different sections of the configuration. Throughout this phase, some sections define the syntax of the extensions.

Both grammar handler modules need to collaborate in order to support mixed coding. This is essential as our design of the language allows using extensions within host language constructs, and vise versa. That is how we allowed the user to write an iterator's body with the host language within the GGDML iterator, and allowed using GGDML iterator in place of a host language statement.

7.3.5. Optimization and Code Generation

Parsing a configuration file does not only allow defining the syntax of the language extensions, but also allows to define the behavior of the translation process. This allows the translation tool to apply a variety of optimization procedures based on the language extensions. Parallelization of operations within an iterator, for example, is applied based on information from a configuration file when a GGDML iterator is met within the code.

After applying the different described optimization procedures to the AST, the code should be written to the output code version. The optimization procedures transform the code into host language grammar compatible constructs, and some annotations that can be further processed by other tools, e.g, compilers. The ASTs of the different code files within the code tree are used by the code generator to write the target code into output files.

The translation process does not impose or need any special order for parsing of the source code files. The same is true for optimization and code generation. Therefore, users do not need to provide such information for the tools.

7.4. Configuration Files

Adaptability of the language extensions is supported by modification and addition of new extensions. Conventional tools, in which grammar processing is coded in tool, are not sufficient to support this flexibility. Users define those extensions and how they affect the translation of the source code. Therefore, configuration files are used to allow configuring the translation process.

Configuration files include a variety of sections to allow users to define different extensions, and to control how the translation tools can transform the code when extensions are found in source code. Some sections are essential to let the tools translate the code, e.g, grids definitions. But also, there are non-obligatory sections, which can simply be ignored and not added to the configuration files, e.g, the cache blocking. A configuration file is illustrated in Listing 7.1. The shown contents are taken from a configuration file that we use to translate the shallow water equation solver code. The full contents of the original file are shown in Appendix B, and available online under `https://github.com/aimes-project/ShallowWaterEquations`. It demonstrates the different sections of a configuration file, which are then discussed in more detail in the remainder of this section.

Listing 7.1: **Contents of a configuration file**

```
1 SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=2D)
2 ...
3 ALLOCATIONS:
4 ...
5   CASE loc=CELL:
6   {
7     // Code template
8   }
```

```
9   ...
10
11 DEALLOCATIONS:
12   CASE loc=CELL:
13   {
14     // Code template
15   }
16   ...
17
18 GLOBALDOMAIN:
19   COMPONENT(CELL2D):
20     RANGE OF YD= 0 TO GRIDY
21     RANGE OF XD= 0 TO GRIDX
22   ...
23   DEFAULT=CELL2D[CELL2D:cell,ce,c][EDGE2D:edge,ed,e]
24
25 INDEXOPERATORS:
26   east_cell(): XD=$XD
27   west_cell(): XD=$XD-1
28   ...
29
30 ANNOTATIONS:
31  LEVEL 0:pragma omp parallel for
32
33 CBLOCKING:
34   XD=20000
35
36 MEMORYLAYOUTS:
37   LAYOUT(2):
38     INDEX=$0
39     INDEX=$1
40
41 LOOPINTERCHANGE:
42   0
43   ...
44
45 DOMAINDECOMPOSITION:
46   nodes=...
47   ...
48   INITIALIZATION:
49   {
50     // Code template
51
52   FINALIZATION:
53   {
54
55 ...
56
57 COMMUNICATION:
58   COMMINITIALIZATION:
59   {
60     // Code template
```

```
61
62   COMMCODE:
63     SECTION (east_cell()) 2D READ:
64     {
65       // Code template for halo pattern
```

7.4.1. Declaration Specifiers

Declaration specifiers, as discussed in Section 5.3.1, are not a constant set of specifiers as in conventional language specifications. Rather, GGDML provides guidelines and concepts for how the GGDML declaration specifiers should be used, and users define the actual specifiers. Declaration specifiers are defined in groups, in each of which a set of alternatives is provided. As those groups and specifiers are not constant, they are defined through configuration files.

Configuration files include a section where all declaration specifiers are added. An example definition is shown in Listing 7.2.

Listing 7.2: **Defining declaration specifiers**

```
1 SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=3D|2D)
```

This example includes defining two groups of specifiers. The first is named *loc* denoting the localization of the fields with respect to the primitive shape of the grid tessellation. Under this group are two alternatives; $CELL$ or $EDGE$. One of them can be exclusively used when declaring a field to reside on the cell center or on the edges around the cells respectively. The other group is named *dim* denoting the dimensionality of the grid on which the field is defined. Under this group are the two options $3D$ and $2D$. Either of those options can be used to specify if the grid is three dimensional or two dimensional.

The section is started with section name, i.e, $SPECIFIERS$, as is the case with all sections of configuration files. For each group, a $SPECIFIER$ clause is written, which includes the name of the group and the alternative specifiers separated by the | character. The count of the alternative specifiers in the group is not constrained. For example, the *loc* group could include an additional $VERTEX$ option if the application needs to support localizing fields at the vertices of the cells. In the same way, the *dim* group could include an additional $1D$ specifier if one dimensional grids are needed in the application.

Defining the declaration specifiers in the described way provides the necessary flexibility to allow users define the language extensions that fits the needs of the application in hand. This is an essential point to examine the application-adaptable language extensions.

7.4.2. Problem Domain

Specifying the space over which to solve a problem is an important part of stencil computations. As space is discretized in grids, describing the grid points used to compute a solution specifies the problem domain.

Generally, grid points are arranged and located corresponding to some rule or description. In all grids, there are dimensions describing the grid points, at least one. There is also a definite number of points in the grid and in each of its dimensions. Thus, defining the grid dimensions and the ranges of those dimensions would be sufficient to describe the problem domain.

GGDML code, as described in Section 5.3.2, does not specify the grid details in the source code. Rather, grids are defined and named globally to the application through configuration files. Grid names can be used within applications source code to specify the points to traverse in an iterator. Modifier operators are provided by GGDML to modify the set of points to traverse for a particular iterator instance.

Global definitions of grids are provided through configuration files in a specific section. An example definition is shown in Listing 7.3.

Listing 7.3: **Defining global domain**

```
1 GLOBALDOMAIN:
2  COMPONENT(CELL3D):
3   RANGE OF ZD= 0 TO Z_count
4   RANGE OF YD= 0 TO Y_count
5   RANGE OF XD= 0 TO X_count
6  ENDCOMPONENT
7
8  DEFAULT=CELL3D[CELL3D:cell,ce,c][EDGE3D:edge,ed,e]
9 ENDGLOBALDOMAIN
```

The *GLOBALDOMAIN* section includes definitions of grids defining the problem domain. The name is chosen as the definitions specify the domain of the whole problem regardless of any domain decomposition under multiple-node configurations. One *COMPONENT* is shown in the example to allow traversal of the grid points at the cell centers of the 3D regular grid. The name *CELL3D* is given to this set, and can be used in iterators to traverse this set of grid points. The dimensions of this set of points and the ranges of each dimension are provided, each dimension in one line. Each line starts with *RANGE OF* and the name of the dimension, and then after the 'equal' character the boundaries of the dimension are given separated with the word 'TO'. Normally, the lower boundary is 0, and the upper is the count of the grid points in that dimension. The boundaries can be expressions allowed by the host language. So, variables that are declared within the application code can be used as well as constants. Multiple components can be added to define different sets of grid points. Multiple grids can be defined using the different components of the global domain.

To simplify the use of the grids in source code, we allow using the special grid identifier "**grid**" (as mentioned in Page 97) along with special iterator index names, the use of which can tell which set of grid points should the iterator traverse. This is specified through the *DEFAULT* part of the global domain definition. The first occurrence of *CELL3D*, the one outside the square brackets, means that any index name with the special grid name *grid* refers to the points in the *CELL3D* set of points. The other square-bracket-surrounded parts define special indices to be used along with the special

grid name *grid*. In the example definition, *cell*, *ce*, or *c* refer to the $CELL3D$ set of grid points, and *edge*, *ed*, or *e* refer to the $EDGE3D$ set of grid points. Allowing multiple alternative indices for each grid gives more flexibility to use shorter names, e.g., *c*, or more expressive names, e.g., *cell*. The entries in the brackets override the default grid that is outside of the brackets if any of those special iterator index names is used.

Using the example definition of a problem domain and the special grid name *grid* and the special indices, simpler iterators can be written as demonstrated in Listing 7.4.

Listing 7.4: **An iterator using the problem domain definition in Listing 7.3**

```
1 foreach cell in grid
2 ...
```

In this example, *cell* is used as an iterator index along with the *grid* as an identity of the set of grid point that the iterator should traverse. This leads the tools according to the example problem domain configuration (Listing 7.3) to choose the grid point set named $CELL3D$ to iterate in this iterator.

7.4.3. Access Operators

As discussed, grid points are arranged and located corresponding to some rule or description. In structured collocated rectangular grids each point is surrounded with exactly two points in each dimension, one from each side. Therefore, in 2D rectangular grids, each point has four neighboring points. This can be more complicated in unstructured grids, where some description of the grid points relationships should be explicitly provided. Even with unstructured grids, rules could be partially used to describe the grid points relationships, e.g, regular vertical levels.

Knowing the rule or the description of the grid is necessary to specify stencils, as multiple neighboring grid points are accessed in a stencil instance. GGDML-based Source code does not include such grid structure information. Access operators, as discussed in Section 5.3.2, are used as modifiers to GGDML indices to allow access to neighboring grid points. GGDML does not provide a static set of such access operators, however, they are defined by users to support the needs of a particular application. GGDML access operators reflect the spatial relationships of the grid points, thus, they reflect the rules or descriptions which define the grid. They can be used in the source code, but they are defined in configuration files. An example is illustrated in Listing 7.5.

Listing 7.5: **An example acces operator definition**

```
1 INDEXOPERATORS:
2  east_neighbor(): XD=$XD+1
```

The section $INDEXOPERATORS$ holds the definitions of the access operators. An access operator allows to compute the indices through formulae. Each formula is written on a separate line. The name of the access operator is written along with a colon before each formula. The formula $XD = \$XD + 1$ in the example takes the X-dimension

value of the grid point and adds 1 to compute the X-dimension of the location of the neighboring grid point.

The example demonstrates the idea of defining an access operator with a simple formula. Formulae can include any expressions that the host language supports. Different expressions to compute or fetch indices provide the flexibility to supporting different kinds of grids, e.g, using lookup tables to support unstructured grids.

7.4.4. Memory Layout

Stencil computations are characterized by low arithmetic intensity. Thus, performance of those computations depends on the optimal use of the memory bandwidth. Using the right memory layout to store the data of the fields in memory is a key optimization aspect.

The nature of the stencils applied throughout an application is necessary to decide the optimal data layout. But also, the underlying architecture should be considered to understand how data in memory is accessed to optimize that access. Application-specific and architecture-specific information are used to specify the layout of the fields. This is provided through a section in configuration files. The fields are declared in source code with GGDML declaration specifiers, and have no explicit specification of data layout. Field data access is also handled with spatial GGDML indices, which carry no information of the data layout.

Field data layout is controlled by actual field allocation code and the transformations applied to the GGDML indices that define the actual data access. Actual data indices are computed based on dimensions comprising the grids where the fields are defined. Each actual array index (that allows to address data in memory) is defined based on a formula within the data layout section in a configuration file. An example data layout configuration section is demonstrated in Listing 7.6.

Listing 7.6: **An example memory layout transformation configuration**

```
1 MEMORYLAYOUTS:
2  LAYOUT(2):
3   INDEX=$0
4   INDEX=$1
5  ENDLAYOUT
6 ENDMEMORYLAYOUTS
```

This simple configuration assigns one array index for each grid dimension of a two-dimensional grid. A two-dimensional array is used to store the field and the grid components are used directly as array indices. If the indices are to be swapped, the configuration is then specified as in Listing 7.7.

Listing 7.7: **Memory layout transformation swapping grid dimensions within arrays**

```
1 MEMORYLAYOUTS:
2  LAYOUT(2):
3   INDEX=$1
```

```
4    INDEX=$0
5   ENDLAYOUT
6 ENDMEMORYLAYOUTS
```

What we do is just defining the first index using the second grid dimension, and the second array index using the first grid dimension. More complex layouts can be defined using such formulae. As an example, a field defined on a grid with some dimensionality can be stored in an array with different dimensionality. Transformations specified by the formulae allow to transform the grid indices into array indices. An example is shown in Listing 7.8.

Listing 7.8: **Memory layout transformation with different grid & array dimensionalities**

```
1  LAYOUT(2):
2   INDEX=($0+1)*(GRIDX+3) + $1 + 1
3  ENDLAYOUT
```

In this example, the indices of the 2D grid are transformed into a single array index, i.e., the 2D grid fields are stored in a 1D array. Different expressions that the host language accepts can be used within formulae to find array indices. Therefore, simply any transformation can be applied including filling curves and mathematical transformations. Constants allowed by the host language and variables declared within the source code can be used to form the expressions that are used to define indices.

7.4.5. Annotations

Annotating code allows using well-known compiler pragmas to guide some optimization procedures. Parallelization using OpenMP or OpenACC pragmas is one step in optimizing code, which enables the use of multiple cores or threads. To use such capabilities, we allow users to annotate the code that is generated by the translation tool. Mainly, this is used to annotate loops that traverse grids in order to control and guide the traversal process and optimally use the hardware resources.

A section ($ANNOTATIONS$) in configuration files is used to specify annotations. This section includes specifying different annotations under different labels each of which is specified in its own line. To annotate different grid traversal levels in a loop nest, the word $LEVEL$ followed by a space and the nesting level are used as a label. One way to make use of this structure is to assign work at a specific loop nest level to specific set of resources, e.g, assign top level loop to OpenMP threads. Detailed guidance, e.g, assigning distributing work in one grid dimension to GPU gangs and another dimension to vectors, is possible with the use of the annotation labels.

An example is shown in Listing 7.9

Listing 7.9: **Example annotation configuration**

```
1 ANNOTATIONS:
2  LEVEL 0:pragma omp parallel for
3  LEVEL -1:pragma omp simd
```

```
4 ENDANNOTATIONS
```

In this example, an OpenMP pragma will be generated to parallelize the rows of a grid on a multi-core processor. Also, the innermost loop will be annotated for vectorization. Labels with negative numbers are used by the tool to refer to reversed loop order starting from an innermost loop (the smallest/fastest step in the nest). The provided annotations represent a recommended parallelization strategy for the grid traversal, however, tools implementation could extract information from this recommended strategy and analyze kernels to generate the final annotations.

In addition to annotation of loop nests, which allows control over grid traversal, the section accepts other labels. Such additional labels allow to control annotation of blocking loops if blocking is to be generated, and time stepping of the simulation.

7.4.6. Communication

Scaling code to allow execution on multiple nodes is an important optimization aspect, where single-node runs are becoming insufficient for modern models which use higher-resolution grids and more fields and processes. Model code that is written with GGDML is not aware of this detail as GGDML hides hardware details. Therefore, necessary information that enable scaling over multiple nodes are provided through configuration files.

Scaling code over multiple nodes needs decomposing the problem domain into subdomains. Domain decomposition can be done either automatically by the translation tool or can be guided by the configuration files. Domain decomposition results in distributing the work and the data of the problem over the used nodes. The nature of stencil operations, where neighboring points comprise a stencil operation, include accessing data which could reside on another subdomain/node. This leads to the need to communicate data between the nodes to guarantee access to the necessary data for the computations on each node. Synchronization is also an important point that should be taken into account to guarantee the consistency of the computations. To handle such details, we use different sections within configuration files to let users guide the scaling process.

Handling communication in simulations is normally done using some library, e.g, MPI. Configuration files include necessary information to initialize the communication library that is selected to be used for halo exchange, and to let the library finalize at the end of the simulation. Other information are also specified including

- the necessary files that should be included to use the communication library
- the amount of processes that are being used, which can be a number or even a name of a variable, which allows using different amounts of nodes between different runs of the same code
- a variable name that holds the number of the process with respect to the set of running processes, which allows identifying each process

Listing 7.10 illustrates the structure of this configuration information.

Listing 7.10: **Example configuration using a communication library**

```
DOMAINDECOMPOSITION:
 nodes=...
 processID=...
 INCLUDE: ...
 INITIALIZATION:
 {
  ...
 }
 ENDINITIALIZATION
 FINALIZATION:
 {
  ...
 }
 ENDFINALIZATION
ENDDOMAINDECOMPOSITION
```

This configuration structure allows using alternative libraries to handle halo exchange. Selection flexibility allows users to decide which libraries to use for their applications. Even changes in library versions are addressed using this technique when major updates to the APIs are introduced.

Communication to handle the exchange of the halo data between the nodes is done by calling the communication API that the selected communication library exposes. Users control the use of the library APIs to handle the necessary communications in the application. To make this possible, the structure of the grid itself and the stencils that the application applies are used to identify halo patterns. With GGDML, this is straight forward as GGDML indices are used to access field data. GGDML indices reflect neighborhood relationships, and hence, halo regions. Using GGDML indices allows users to provide code templates to execute the communication of the halo for each halo pattern.

A section (*COMMUNICATION*) to configure communication is used in configuration files. One part of this section is *COMMCODE* which contains the template codes for the different halo patterns. One *SECTION* is used to provide code for each halo pattern. An example snippet is shown in Listing 7.11

Listing 7.11: **Example halo exchange configuration**

```
COMMUNICATION:
...
COMMCODE:
SECTION (north_edge()) 2D READ:
{
  if(mpi_world_size>1){
    comm_tag++;
    int pp = mpi_rank!=0 ? mpi_rank - 1 : mpi_world_size - 1;
    int np = mpi_rank!=mpi_world_size - 1 ? mpi_rank + 1 : 0;
    MPI_Isend($var_name[0],GRIDX+1,
           MPI_FLOAT,pp,comm_tag,MPI_COMM_WORLD,&mpi_requests[0]);
    MPI_Irecv($var_name[local_Y_Eregion],GRIDX+1,
           MPI_FLOAT,np,comm_tag,MPI_COMM_WORLD,&mpi_requests[1]);
```

```
14      MPI_Waitall(2,mpi_requests,MPI_STATUSES_IGNORE);
15   }
16 }
17 ENDSECTION
18 ...
```

This example demonstrates using a compound C statement to handle communication. To start the definition of code template, the word *SECTION* is used. Then, the extension *north_edge* is specified in between the parentheses to tell the tool that this section corresponds to this kind of spatial relationships. This example tells that this code is to be used to communicate necessary halo data to allow access to the edge that bounds the current cell at the north side. In a stencil, in which the north edge is used to compute the value at cell center, tools infer that some cells need to access edges that reside on other nodes, and use the provided code template to handle the data communication. After the parenthesis other options are specified to tell more information about the use of this code section, e.g, if this code should be used when this halo region is to be accessed for read or write. Within the code, placeholders can be used, which can be processed by the translation tool to fit the code where it should be used. For example, the $*var_name* placeholder would be substituted by the name of the field, the data of which is to be communicated. An important point to notice from this example is that, calls to synchronization API, e.g., $MPI\_Waitall()$, should be used more carefully to decrease synchronization overhead. But we used it in our experiments for code simplicity and because the communication time (including the synchronization) is negligible.

The information provided within the different sections of the communication code are sufficient to handle communication, with full control from the users. Analysis of both the source code and the different sections of the communication code are done by the translation tool to generate the actual communication. Details of this analysis will be discussed later in this chapter.

7.4.7. Cache Blocking

Data reuse is an important aspect of optimization of stencil computations, which need optimal use of memory bandwidth and caching, as a result of their low arithmetic intensity. A well known technique to improve data reuse is cache blocking. Stencils usually access data of some field at multiple neighboring points. Non-optimized codes may need to access the same data multiple times in memory. This is expensive to the computation performance. Optimal use of the caches allows fetching data once from memory to caches and reusing that data further while it is still in caches.

Code that is written using GGDML is unaware of caching and such hardware details, however, the translation tool can transform GGDML iterators applying cache blocking throughout the translation process. To control whether cache blocking is to be applied or not, and the cache blocking factor and on which grid dimensions to be applied if users select to apply it, information is provided in a simple section within configuration files. An example is shown in Listing 7.12.

Listing 7.12: **Example cache blocking configuration**

```
1 CBLOCKING:
2 XD=20000
3 ENDCBLOCKING
```

This simple section starts with the word *CBLOCKING*. Inside this section is the name of the grid dimension which will be the basis for cache blocking. The number represents the blocking factor that the user recommends to be used be the translation tool. The absence of the *CBLOCKING* section from a configuration file implies that the translation tool should not apply cache blocking.

7.4.8. Loop Interchange

The order of the dimension visits in a loop nest is decisive for the performance of that loop. A GGDML iterator applies its body statement to a set of grid points. Grid definitions from configuration files are used to generate the necessary loops that traverse the set of points in the iterator. User can specify the order of the loops, which grid dimensions should be visited faster (inner loops) and which slower (out loops), through configuration files. The absence of such information in configuration files leads the tool to order the loops as they are in the definition of the grid. This default behavior generally allows optimal data access as inner loops (fastest) access unit stride data. However, if the user defines some field allocation layout that is better accessed with a different loop order, then loop interchange can be applied.

Loop interchange is specified within configuration files through a corresponding section. In this section, which is marked with *LOOPINTERCHANGE*, a single line should be used to specify a loop nest level. Thus, an N-dimensional grid needs N lines to tell the order of the loops. An example loop interchange specification is shown in Listing 7.13

Listing 7.13: **Example loop interchange configuration**

```
1 LOOPINTERCHANGE:
2 1
3 0
4 2
5 ENDLOOPINTERCHANGE
```

In this example, the the inner loop (#2) iterates the dimension #2, the unit stride, so closest data are visited in closer time periods. However, the most outer loop and the middle loop are interchanged with respect to array order. Assuming an array $array[z][y][x]$, the loops are applied as

Listing 7.14: **Resultant loop interchange**

```
1 for (y ...
2   for(z ...
3     for(X ...
4       array[z][y][x] = ...
```

Loop interchange configuration section is one of the sections that could be improved with tool implementation that could do further analysis of kernels besides per-grid interchange recommendations from configuration files. This could maximize the impact of loop interchange, and improve the support for the use of multiple grids.

7.5. Mappings: *Optimization=Semantics×Configurations*

We have so far discussed the language extensions that users use to write source code and the configuration files that guide the code translation. In this section, we discuss matching different language extensions with corresponding configuration information to enable different optimization aspects. The higher-level "**what**" question is answered, as we identify what objectives are achievable and driven by which inputs. Understanding those mappings leads to understand how the underlying information extractions from source code and configurations could be used to drive the optimization process. The actual algorithms that use the extracted information to optimize the code are discussed in the next section.

7.5.1. Field Allocation

Declarations of the fields within the source code use declaration specifiers that are provided as language extensions. Those extensions (declaration specifiers) carry information regarding the fields, e.g, grid dimensionality. No further information regarding the allocation of field data is needed within the source code. However, the definition of the declaration specifiers within the configuration files provides the necessary information to let the tools know how to actually do the memory allocation.

To demonstrate matching a declaration of a field, which is declared with GGDML declaration specifiers, with the corresponding configuration sections, we discuss the following example. The field f_H is declared using the GGDML declaration specifiers $CELL$ and $2D$ in Listing 7.15.

Listing 7.15: **An example declaration with GGDML specifiers**

```
1 float CELL 2D f_H;
```

When the code is parsed, the declaration specifiers are enough to tell the translation tool that the field is defined at the centers of the cells of the 2D grid. No more information need to be provided regarding the allocation of the memory space and the data access.

The declaration specifiers are defined in our example as follows (Listing 7.16) in the configuration file.

Listing 7.16: **Configuration section defining declaration specifiers**

```
1 SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=3D|2D)
```

This section allows the tool to know that the application can use the defined groups of declaration specifiers. However, it does not tell how to do the allocation of memory for the fields.

The user control over memory allocation of field data is provided through the memory allocation section in the configuration file. In our example, we use the following code template (Listing 7.17) to allocate memory for the field data.

Listing 7.17: **User-provided template to allocate a field**

```
CASE loc=CELL & dim=2D:
{
      int num_Y_rows = 2 + local_Y_Cregion;
      int num_X_rows = 2 + GRIDX;

      $var_name = malloc(
              num_Y_rows * num_X_rows * sizeof($data_type) +
              num_Y_rows * sizeof(char*));

      char* pos = (char*)$var_name + num_Y_rows * sizeof(char*);
      for(int j=0; j < num_Y_rows; j++){
          $var_name[j] = ($data_type*)pos;
          pos+= num_X_rows * sizeof($data_type);
              for(int i=0; i < num_X_rows; i++){
                  $var_name[j][i] = ($data_type) 0;
              }
      }
      for(int j=0; j < num_Y_rows-1; j++){
          $var_name[j] += 1;
      }
      $var_name += 1;
}
ENDCASE
```

This code template is used to allocate memory for fields that are declared with $CELL$ as a localization declaration specifier and $2D$ for dimensionality.

Matching the declaration specifiers used in Listing 7.15 and Listing 7.17, the translation tool outputs the following Listing 7.18 code to allocate the memory for the field f_H.

Listing 7.18: **Generated code to allocate the field declared in Listing 7.15**

```
    {
        int num_Y_rows = 2 + local_Y_Cregion;
        int num_X_rows = 2 + GRIDX;
        f_H = malloc(num_Y_rows * num_X_rows * sizeof(float) +
                     num_Y_rows * sizeof(char *));
        char *pos = (char *) f_H + num_Y_rows * sizeof(char *);
        for (int j = 0; j < num_Y_rows; j++) {
            f_H[j] = (float *) pos;
            pos += num_X_rows * sizeof(float);
            for (int i = 0; i < num_X_rows; i++) {
                f_H[j][i] = (float) 0;
            }
```

```
13          }
14          for (int j = 0; j < num_Y_rows - 1; j++) {
15              f_H[j] += 1;
16          }
17          f_H += 1;
18      }
```

The tool processes some expressions within the code template, e.g, $*var_name* and $*data_type*, and generates the code where the variable should be allocated in the code.

7.5.2. Parallelization

Iterators within the source code tell that the body is to be applied for a specific set of grid points. This set of grid points is specified through the grid expression part of the iterator. Therefore, the source code implies no information regarding how to parallelize the operations over the processor resources, e.g, cores. However, the grid expression tells that multiple instances of the stencil/body are to be repeatedly executed. To control how to apply the stencil operations in parallel over the processor resources, users can annotate loops using some programming model, e.g, OpenMP or OpenACC. Annotations are configured using a section within configuration files. The translation tool can match information from the source code, i.e, on which grid points to instantiate the iterator body, and the information from the annotation section. The annotation section guides annotation of loops based on the definition of the problem domain as provided in the configuration file. Thus, domain definition is also included in the process to decide how to parallelize the application of the stencil/body over the grid points specified in the iterator statement.

To demonstrate this matching of the information from the source code and the information from the configuration file, an example iterator statement is shown in Listing 7.19.

Listing 7.19: **An example iterator traversing cells of a 2D grid**

```
1 foreach c in grid
2 {
3 ...
```

This iterator traverses the grid points at the cell centers of the 2D grid. What the tool extracts from this iterator statement is to apply the iterator body to all the mentioned grid points. To know what is this set of points, the tool should refer to the definition of the problem domain. Let us take the example definition of a set of grid points specified in Listing 7.20.

Listing 7.20: **An example problem domain definition**

```
1 GLOBALDOMAIN:
2 ...
3  COMPONENT(CELL2D):
4   RANGE OF YD= 0 TO GRIDY
```

```
5   RANGE OF XD= 0 TO GRIDX
6  ENDCOMPONENT
7 ...
8 ENDGLOBALDOMAIN
```

This configuration section tells the tool that this set of grid points is defined over two dimensions with the specified ranges. Now the tool knows the actual set of points where to apply the iterator body. However, the distribution of the computation over the processor resources, e.g, cores of the multi-core processor, is not yet defined. To decide how to parallelize the execution, annotation information are taken from the annotation section from the configuration file. An example annotation section is provided in Listing 7.21.

Listing 7.21: **An example section to guide annotation**

```
1 ANNOTATIONS:
2  LEVEL 0:pragma omp parallel for
3 ENDANNOTATIONS
```

What the tool gets from this section is to parallelize the outermost for loop that iterates the first dimension of the grid, i.e, YD, using the *omp* pragma.

The result of matching the information from the iterator statement in the source code with the information from the configuration file regarding the problem definition and the annotation is shown in Listing 7.22.

Listing 7.22: **Generated OpenMP parallel loop from iterator in Listing 7.19**

```
1 #pragma omp parallel for
2             for (size_t YD_index = (0);
3                  YD_index < (local_Y_Eregion);
4                  YD_index++) {
5                 for (size_t XD_index = blk_start;
6                      XD_index < blk_end;
7                      XD_index++) {
```

7.5.3. Cache Blocking

Source code that is written with GGDML is written with high-level iterators, which apply operations repeatedly at a set of grid points. The code does not include any details regarding how the dimensions are iterated. Thus cache blocking is not the concern of the model code developers. To exploit caching optimization with cache blocking technique, our translation tool applies transformations to the code. Those transformations are done based on matching information from the iterator statement from the source code, with corresponding information from configuration files.

The grid expression part of the iterator tells the tool which grid points to traverse to apply the iterator body. The tool uses the definition of the set of grid points from the configuration file and checks the cache blocking section (in configuration file) to check which dimension of the grid should be used for blocking and with which blocking factor.

Matching all those information together allows the tool to generate the necessary loop nest with cache blocking.

To demonstrate the idea, let us consider the iterator statement in the code snippet in Listing 7.19. This statement within the source code tells the translation tool that the body of the iterator is to be applied to the cell centers of the 2D grid. Only the set of points to traverse are specified, but not how to do the blocking. Then the tool refers to the definition of this set of grid points within the example configuration shown in Listing 7.20.

The translation tool does not yet know whether the user wants to apply cache blocking. Thus, the tool refers to the other section of the configuration (Listing 7.23) where desired cache blocking is described.

Listing 7.23: **An example blocking configuration**

```
1 CBLOCKING:
2 XD=20000
3 ENDCBLOCKING
```

This information allows the tool to apply cache blocking to the XD dimension of the grid point set. Thus, it divides the XD dimension into blocks of 20000 and generates the loops to traverse the grid points with this division. The code in Listing 7.19 is then transformed into the code in Listing 7.24.

Listing 7.24: **Applying blocking to code from Listing 7.19**

```
1              for (size_t blk_start = (0); blk_start < (GRIDX);
2                   blk_start += 20000) {
3                  size_t blk_end = GRIDX;
4                  if ((blk_end - blk_start) > 20000)
5                      blk_end = blk_start + 20000;
6 ...
7                  for (size_t YD_index = (0);
8                       YD_index < (local_Y_Cregion);
9                       YD_index++) {
10 ...
11                     for (size_t XD_index = blk_start;
12                          XD_index < blk_end;
13                          XD_index++) {
14 ...
```

7.5.4. Loop Order

Instead of explicit nested loops, stencils are applied with GGDML through iterators, which carry no information about in which order to traverse the grid dimensions. The GGDML iterator tells the translation tool that the operation is to be applied for a specified set of grid points. This information is not enough to know how to traverse the grid points. A configuration file provides a definition for the set of grid points. This definition indicates what loops to use to traverse those points. However, again

this information does not define the structure of the loop nest (the order of the loops), although it is still possible to use a default order. The preferred order of the loops is taken from the loop interchange section within the configuration file. Matching all these information allows the tool to generate the code of the nested loop using the user preferences.

To demonstrate the idea, we use the example iterator in Listing 7.19. Let us assume that the definition of the set of grid points to traverse by the iterator in Listing 7.19 is the one shown in Listing 7.20. From the definition, the tool knows that it should iterate the body over the two dimensions with the given ranges. The tool then uses the user-preferred loop order defined in Listing 7.25.

Listing 7.25: **An example loop order/interchange configuration**

```
1 LOOPINTERCHANGE:
2 1
3 0
4 ENDLOOPINTERCHANGE
```

The preferred order lets the tool generate the code with the loop order, where the dimension XD is iterated in the outer loop and the dimension YD is iterated in the inner loop. The resulting generated code from the discussed example iterator and configuration information is shown in Listing 7.26.

Listing 7.26: **Loop interchange applied to code in Listing 7.19**

```
1 ...
2                   for (size_t XD_index = blk_start;
3                        XD_index < blk_end;
4                        XD_index++) {
5 ...
6                       for (size_t YD_index = (0);
7                            YD_index < (local_Y_Cregion);
8                            YD_index++) {
9 ...
```

7.5.5. Memory Layout

GGDML-based code uses GGDML indices to access field data. GGDML indices refer to grid points and correspond to spatial relationships (in terms of problem space), where relationships between grid points are represented with those indices. With this kind of indices, actual location of data in memory is not specified in the source code. The translation tool matches field access indices with the definition of the grid point set and with a specified memory layout to generate the array indices that address the actual data in memory. Whenever a field access is found, the tool processes the index including resolving access operators, and creates a basic set of array indices that refers to data in an abstract default structure. Data layout transformation information, which is provided through a section in configuration files, is used to apply transformation formulae on the

default structure to generate the actual array indices.

To demonstrate the idea, a simple field access without any access operators is shown in Listing 7.27.

Listing 7.27: **An example field access**

```
1 foreach c in grid
2 {
3   ...
4   f_HT[c] = df + dg;
```

When the tool finds the expression which accesses the field f_HT, it finds that the field is accessed using the index c, which refers to a cell (a grid point) among the set of grid points at the cell centers of the 2D grid. With this information, the tool refers to the definition of this set of grid points (as shown in Listing 7.20).

Based on the definition of the grid point set, the tool temporarily represents the access with indices that access data in an abstract structure, which is an imaginary two-dimensional array for the two dimensions YD and XD. The state of the expression $f\_HT[c]$ is currently equivalent to the C expression $f\_HT[YD\_index][XD\_index]$. However, this data structure is just an imaginary one. To generate the actual array indices that address data elements in the actual data structure, the process should further proceed using memory layout transformation formulae. Using the formula in Listing 7.28, the tool generates the actual field access expression.

Listing 7.28: **An example memory layout transformation configuration**

```
1 MEMORYLAYOUTS:
2 ...
3  LAYOUT(2):
4   INDEX=($0+1)*(GRIDX+3) + $1 + 1
5  ENDLAYOUT
6 ENDMEMORYLAYOUTS
```

The formula leads the tool to generate one index to access one-dimensional array that holds the data of the field. The tool applies the transformation according to the formula and generates the code shown in Listing 7.29.

Listing 7.29: **Generated code corresponding to code from Listing 7.27**

```
1 ...
2 #pragma omp for
3     for (size_t YD_index = (0);
4          YD_index < (local_Y_Cregion);
5          YD_index++) {
6 #pragma omp simd
7         for (size_t XD_index = blk_start;
8              XD_index < blk_end;
9              XD_index++) {
10 ...
11             f_HT[(YD_index + 1) * (GRIDX + 3) + XD_index + 1] =
12                                                       df + dg;
```

7.5.6. Scalable Multi-node Parallelization

Source code that is developed with GGDML is unaware of the underlying hardware. It does not include any information or assumptions whether it will be run on a single node or on multiple nodes. Developers use spatial relationships to express stencils. The tools can scale the code to multiple nodes using further information from configuration files when translating the high-level code. What the tool needs from the source code is spatial relationships. This can be extracted from the GGDML indices that are used to access fields data. Spatial relationships allow the tool to identify data elements that reside on other nodes. The tool can analyze the different field accesses and indices to identify the necessary data communications. GGDML indices are enabled by access operators, which are defined in a section in configuration files. When the tool identifies the needed communications, it uses information from another section from configuration files to handle the communication. Matching the use of the indices from the source code with the definitions of the access operators (and hence the spatial relationships) and with the information on how to handle the communication of the different halo patterns allows the tool to generate scalable code that runs on multiple nodes.

To demonstrate the idea, take the example kernel shown in Listing 7.30.

Listing 7.30: **An example iterator with GGDML access operators**

```
1  foreach c in grid{
2          ...
3          float dg = (f_G[c.north_edge()] -
4                      f_G[c.south_edge()]) / dy;
```

In this kernel, the edges bounding a cell at the north and south sides are accessed. The access is written with the spatial relationships *north_edge* and *south_edge*. These are the only information that the source code needs to provide.

The definitions of those access operators are provided within the configuration file as in Listing 7.31.

Listing 7.31: **Access operator definition (in configuration file)**

```
1  north_edge(): YD=$YD+1
```

Definitions of access operators indicate whether the data reside on the same node or on another node. In the example *north_edge* definition, which is taken from a row-based domain decomposition, it is clear that the north-most cell row on one node needs to access the south-most edge row on the neighboring node which handles the rows north of the current node. When the tool knows such communication needs, it needs to know how to communicate this halo pattern. To do that, different halo patterns are mapped to code templates in the configuration file. An example halo pattern is shown in Listing 7.11.

Using the template that is provided to handle the halo pattern, the tool can generate the necessary code to handle the communication that needs to be done for the stencil to

guarantee the access to the data which resides on other nodes. Matching all the previous information leads the tool to generate the communication code in Listing 7.32 during the translation of the kernel in Listing 7.30.

Listing 7.32: **Part of generated communication code corresponding to Listing 7.30**

```
{
  if (f_G_dirty_flag[24] == 1) {
    if (mpi_world_size > 1) {
        comm_tag++;
        int pp = mpi_rank != 0 ? mpi_rank - 1 : mpi_world_size - 1;
        int np = mpi_rank != mpi_world_size - 1 ? mpi_rank + 1 : 0;

        MPI_Isend(f_G[0], GRIDX + 1, MPI_FLOAT, pp, comm_tag,
                  MPI_COMM_WORLD, &mpi_requests[0]);

        MPI_Irecv(f_G[local_Y_Eregion], GRIDX + 1, MPI_FLOAT,
                  np, comm_tag, MPI_COMM_WORLD, &mpi_requests[1]);

        MPI_Waitall(2, mpi_requests, MPI_STATUSES_IGNORE);
    }
  }
  f_G_dirty_flag[24] = 0;
}
```

7.6. Algorithms: *Implementing the Mappings*

In the previous section, we discussed mapping source code components and configurations to objectives that should be achieved with the translation process, at a coarse level. We discussed "**what**" objectives are achieved through which source code components and which configuration information. In this section we discuss "**how**" the translation process uses those two input sources to achieve objectives. To do this, we developed a set of algorithms. Those algorithms are implemented in the translation tool. The complete details of the translation process and the optimization procedures are lengthy. Therefore, we discuss the algorithms with simplified pseudo-code.

7.6.1. GGDML Kernel Processing

We start with the main algorithm that is used to translate GGDML kernels (Algorithm 1), which also executes other algorithms to transform kernels. After a GGDML kernel is parsed and a corresponding AST is built, this AST is where transformations occur. This algorithm gets the AST node of the grid expression and uses it to identify the set of grid points to which the kernel body is to be applied. The set of grid points is used to build an initial (default) loop nest descriptor. This initial descriptor is passed further to other transformations.

Algorithm 1: Kernel translation algorithm

```
input  : DSL_kernel_AST
output : processed_kernel_AST

k_AST ← create_empty_AST();
grid_expression ← get_grid_expression(DSL_kernel_AST);
initial_body_AST ← get_kernel_body(DSL_kernel_AST);
grid_point_set ← parse_grid_point_set(grid_expression);
default_loop_nest_descriptor ← initialize_loop_nest_descriptor(grid_point_set);
processed_body_AST, field_analysis_data_structure ← process_body(initial_body_AST);
handle_needed_communication(field_analysis_data_structure, k_AST);
if f_annotate_for_likwid then
  | annotate_instrumentation_start(k_AST);
end
if f_blocking then
  | annotate_block_loop(k_AST);
  | build_block_loop(k_AST);
end
if f_loop_interchange then
  | loop_nest_descriptor ← interchange_loops(default_loop_nest_descriptor,
  |   loop_interchange_data_structure);
end
handle_domain_decomposition(k_AST, loop_nest_descriptor);
foreach dimension in loop_nest_descriptor do
  | if range(dimension)==1 then
  |   | handle_single_layer(k_AST, dimension);
  | else
  |   | annotate_dimension(k_AST, dimension);
  |   | build_dimension_loop(blocking_data_structure, k_AST, dimension);
  | end
end
add_kernel_body_to_AST(k_AST, processed_body_AST);
if f_annotate_for_likwid then
  | annotate_instrumentation_close(k_AST);
end
handle_needed_communication(field_analysis_data_structure, k_AST);
if f_use_dirty_flags then
  | set_dirty_flags_for_updated_halo_regions(field_analysis_data_structure);
end
```

Code profiling and instrumentation: This algorithm checks if the users want the translation process to make their code ready for instrumentation with Likwid tools. If that is the case, the algorithm modifies the AST of the new code accordingly. Starting and closing markups are added.

Apply blocking: Blocking is also enabled and configured by users. If blocking is enabled, the tool applies necessary annotation and builds the blocking loop on the new AST.

Apply loop interchange: If a loop interchange is specified in configuration files, the tool will transform the initial (default) loop nest descriptor according to the loop interchange

configuration information. The transformation generates a new loop nest descriptor that conforms to the user-provided loop interchanges.

Domain decomposition: Domain decomposition is applied to the AST using the loop nest descriptor. This is necessary to enable code scalability over multiple nodes. Other related functions are also done within the algorithm. Communication of halo regions is handled. The tool allows optionally using dirty flags to track updated halo regions, which allows to communicate only updated regions. If this feature is enabled, the algorithm sets and resets those flags to communicate updated regions.

Build loop nest: To build the loop nest, the algorithm uses the loop nest descriptor. Each dimension in the descriptor is checked to either build a loop AST node with necessary annotation, or an alternative node for single layer dimensions.

Process kernel body: The AST node of the kernel body is passed for further body processing before being added to the resulting kernel AST. The body processing is necessary for field analysis and GGDML index translation. The resulting body is added to the resulting kernel AST. The extracted information through field analysis is used further, e.g, identify necessary communication.

7.6.2. Kernel Body Processing

As discussed so far, the main algorithm (Algorithm 1) that handles the translation of kernels calls another part of the code to process kernel bodies. The algorithm to process a kernel body is shown in Algorithm 2. While translating a kernel, transformations are applied to the AST of a kernel body based on the high-level extensions that are used within the source code of a kernel. During this translation process, the access operators are applied to eventually generate the addresses of data elements in memory. Definitions of access operators are provided through configuration files. Definitions are done via formulae which describe how indices are transformed to find memory addresses. Further transformations before finding the actual addresses of field data in memory are memory layout transformations, which will be described in own algorithms later.

Stencil structure analysis: An important function of this algorithm is analyzing a kernel body while it is being traversed for transformations. This analysis allows the tool to understand the structure of the stencils that are executed within a kernel body. Fields that are updated in each stencil are identified, and fields that should be read to compute new field values are also identified. The analysis allows to keep aware of stencil' shapes through the indices that are used to access the fields. This analysis is essential for further later procedures, e.g, identifying needed communication.

The algorithm works on an input kernel body AST, and returns a processed AST and a data structure that describes the stencils that exist in the kernel. The algorithm traverses the input AST and rebuilds a processed copy of it on the output AST, which is

Algorithm 2: process_body

```
input  : initial_body_AST
output : processed_body_AST, field_analysis_data_structure

process_body(initial_body_AST);
field_analysis_data_structure ← new_field_analysis_structure();
foreach foreach node in initial_body_AST do
    if update_expression(node) then
        update_entry ← create_empty_entry();
        uf ← get_updated_field(node);
        add_to_entry(update_entry, uf);
        lao ← get_list_of_access_operators(uf);
        add_to_entry(update_entry, lao);
        foreach accessed_field in node do
            add_to_entry(update_entry, accessed_field);
            lao ← get_list_of_access_operators(accessed_field);
            add_to_entry(update_entry, lao);
        end
        add_entry(field_analysis_data_structure, update_entry);
    end
    if field_access(node) then
                                                  /* e.g., reading array element */
        grid_point_set ← get_field_domain(node, symbol_table);
        index_list ← create_empty_list();
        foreach dimension in grid_point_set do
            add_dimension(index_list, dimension);
        end
        foreach access_operator in node do
            assert_valid_indices(access_operator);
            foreach placeholder in definition(access_operator) do
                evaluate_and_replace(placeholder);
            end
            transform(access_operator, index_list);
        end
        node ← transform_memory_layout(node, index_list, layout_conf_struct);
    end
    add_node(processed_body_AST, node);
end
return processed_body_AST, field_analysis_data_structure;
```

an empty tree at the beginning of the algorithm. The current AST node being traversed is checked if it represents a field update. If that is the case, the updated field (and the access operators if any, which does not often happen) is added to the field access data structure. The update includes stencil operation which needs to access some fields at a set of neighboring grid points. Each field access that is part of the stencil operation is added to the field access data structure along with the list of access operators. Keeping accessed field and a list of access operators for each access within the data structure allows further analysis by the translation tool. This analysis does not comprise any change to the AST.

Index transformations: Throughout the AST traversal, the nodes are also checked for index transformations. Any node that represents a field access, which uses GGDML indices to refer to grid points, is copied and processed before being added to the output AST. The node is checked to know at which set of grid points is the field defined. This information is used along with the definition of the grid point set in the configuration file to identify the dimensions of the domain. Identified dimensions represent the basis for the set of indices that will be used to access field data.

Each access operator that is used to access the field is processed. First, the definition of the access operator is fetched from the access operator definition data structure (based on configuration file contents). Then, the formula is processed to know how to transform the current indices. This includes substituting placeholders, e.g, reference to current value (before transformation) of an index. Finally, the formula is used to transform the indices. After this phase, the indices are matched to the indices that the iterator uses to access the different fields (using host language indices/array notation).

If memory layout transformations are to be applied, are then applied to the current set of indices at this point. The resulting indices are used to build a node on the output body AST.

7.6.3. Declaration and Field Allocation

To handle fields, we develop different algorithms. Some are used to handle declaration specifiers and others are related to memory allocation. Algorithms show how to parse the configuration files to build data structures, and how to use those data structures to transform source code later.

Parsing GGDML specifier configurations: To handle the GGDML declaration specifiers, the definitions of the specifiers and the groups to which they belong are extracted from configuration files into corresponding data structures. A suitable algorithm is shown in Algorithm 3. Configuration files comprise a section where groups of specifiers are defined. The algorithm reads that section from a configuration file and builds a data structure that holds a list of specifier groups. In this structure, each group is composed of a list of specifiers.

Parsing memory allocation configurations: Besides the specifiers definitions is another part of configuration files that is related to field declaration, that is the allocation of field data in memory. Field allocation is defined in configuration files via code templates that can be used along with criteria that decide when to use those templates. Declaration specifiers that are used to declare a field are checked against the criteria accompanying the template codes later to identify the allocation template to use for the allocation of memory for field data. An algorithm is shown in Algorithm 4 to build a data structure to hold field allocation information.

The algorithm reads the section which defines field allocation from a configuration file. An entry is added to the data structure corresponding to each allocation template.

Algorithm 3: parse_specifiers_conf

```
input  : conf_file
output : spec_data_structure

spec_data_structure ← new_spec_structure();
spec_section ← get_specifier_section(conf_file);
foreach group in spec_section do
    new_entry ← create_spec_entry();
    add_group_name(new_entry, group);
    foreach specifier in group do
        add_specifier(new_entry, specifier);
    end
    add_group(new_entry, spec_data_structure);
end
return spec_data_structure;
```

Each entry contains the code template and a set of usage criteria based on declaration specifiers. The criteria tell according to the declaration specifiers used to declare a field whether the template should be used for the allocation.

Algorithm 4: parse_allocation_conf

```
input  : conf_file
output : allocation_conf_data_structure

allocation_conf_data_structure ← new_alloc_structure();
alloc_section ← get_alloc_section(conf_file);
foreach alloc_template in alloc_section do
    new_entry ← create_alloc_entry();
    add_usage_criteria(new_entry, alloc_template);
    add_template_code(new_entry, alloc_template);
    add_alloc_entry(new_entry, allocation_conf_data_structure);
end
return allocation_conf_data_structure;
```

GGDML specifier detection in source code: Among the uses of an extracted data structure that describes specifiers is in parsing source code. During parsing source code and building an AST, the tokens should be checked according to the grammars of the host language and the GGDML extensions. Grammar rules govern the use of GGDML declaration specifiers to declare fields. When a rule is checked, code tokens are checked against expected token types. To handle this, an algorithm is shown in Algorithm 5.

The algorithm checks the next token from the source code whether it is a GGDML specifier. It traverses the specifier groups, and the specifiers in each group. If the token is the current specifier being traversed, the specifier is returned. If all the groups are searched and the token is not found, then a null string is returned, indicating that the token is not a GGDML specifier.

Algorithm 5: is_DSL_specifier

```
input  : next_token, spec_data_structure
output : specifier|""

foreach group in spec_data_structure do
    foreach specifier in group do
        if next_token == specifier then
            return specifier;
        end
    end
end
return "";
```

Declaration transformations: Field declarations should be translated in order to allow host language compilers to understand them. Translation of field declarations is described in Algorithm 6.

Algorithm 6: translate_declaration

```
input  : decl_AST
output : translated_AST

var_name ← get_var_name(decl_AST);
data_type ← get_data_type(decl_AST);
DSL_specifier_list ← get_DSL_specifiers(decl_AST);
translated_AST ← replace_DSL_specifiers(decl_AST);
grid_point_set ← identify_grid_point_set(DSL_specifier_list);
add_to_symbol_table(var_name, data_type, grid_point_set, DSL_specifier_list, decl_AST);
return translated_AST;
```

The field name is extracted from the input declaration AST, in addition to data type and declaration specifiers. Declaration specifiers are used further to identify the set of grid points on which the field is defined. All this information are stored into a record in the symbol table for further use. The input declaration AST is copied and processed to build a new AST. Replacements of GGDML declaration specifiers are applied to the new AST to make it compatible with the grammar of the host language.

Generating memory allocation code: Generating code that allocates memory for field data is done based on data structures that describe field allocation. Those structures are built using Algorithm 4. An algorithm is shown in Algorithm 7 where those data structures are used to generate field allocation code.

Field name is used against the symbol table to retrieve data type and declaration specifiers. The declaration specifiers of the field are used to check against the criteria for the allocation templates in the data structure that describes field allocation. This leads to choose the template code that should be used to allocate memory. Template code includes placeholders, e.g, field name, that should be processed. The algorithm substitutes all placeholders in the allocation code and uses it to build an AST node. The created AST node contains the code that will allocate memory for the field.

Algorithm 7: allocate_field

```
input  : variable_name, symbol_table, allocation_conf_data_structure
output : AST_node

data_type ← get_data_type(symbol_table, variable_name);
specifiers ← get_field_decl_specifiers(symbol_table, variable_name);
alloc_code ← get_template_code(specifiers, allocation_conf_data_structure);
foreach placeholder in alloc_code do
  | substitute(placeholder, alloc_code, variable_name, data_type);
end
AST_node ← create_node(alloc_code);
return AST_node;
```

7.6.4. Annotation

To apply annotations we develop a set of algorithms. Those algorithms show how to parse the configuration sections that drive annotation into corresponding data structures, and how to use those data structures to annotate code.

Parsing annotation configuration: Code annotation is driven by configuration files. To extract information from a configuration file, an algorithm (Algorithm 8) reads the section in the file and builds a corresponding data structure.

Algorithm 8: parse_annotation_configuration

```
input  : conf_file
output : annotation_conf_data_structure

annotation_conf_data_structure ← new_annotation_structure();
annotation_section ← get_annotation_section(conf_file);
foreach line in annotation_section do
  | new_entry ← create_annotation_entry();
  | add_annotation_criteria(new_entry, line); /* blocking, nested loop level, ... */
  | add_annotation_description(new_entry, line);
  | add_annotation_entry(new_entry, annotation_conf_data_structure);
end
return annotation_conf_data_structure;
```

This algorithm reads the annotation section from an input file and iterates over all the lines in it. Each line in an annotation section describes the annotation that is applied under a specific criterion. Different criteria are valid, including the different levels of a loop nest. For each line, a new record is built that holds the criterion and the annotation description. The different records are added to a data structure that describes annotation configuration. This data structure is used later by other algorithms to annotate code.

Annotating blocked loops: A simple algorithm is shown in Algorithm 9 to annotate a blocking loop when cache blocking is enabled. This algorithm looks for a record in the annotation configuration data structure which corresponds to 'block' loop. If such

record exists means that block loops should be annotated. In this case, the annotation description is fetched from the record and used to add the necessary contents to the kernel AST.

Algorithm 9: annotate_block_loop

```
input  : k_AST, annotation_conf_data_structure
output : k_AST

annotation_descr ← get_annotation_description(annotation_conf_data_structure, 'block');
if annotation_descr then
  | generate_annotation(k_AST, annotation_descr);
end
```

Annotating loop nests: Another algorithm (Algorithm 10) is used to annotate different loops within a loop nest. The algorithm uses the nesting depth of the dimension and uses it to check the annotation criteria in the annotation configuration data structure. If a direct record exists then it is used to retrieve the annotation description and apply it to the AST. Otherwise, a negative index is tried, e.g, depth level of (-1) corresponds to innermost loop. Again, if a record is there, the annotation description is retrieved and applied to the AST. If neither of the cases holds, then a default annotation record is searched. Default records are intended to annotate loops which have no corresponding records. If no default annotation record exists, the loop is not annotated.

Algorithm 10: annotate_dimension

```
input  : k_AST, dimension, annotation_conf_data_structure
output : k_AST

annotation_descr ← get_annotation_description(annotation_conf_data_structure, dimension);
if annotation_descr then
  | generate_annotation(k_AST, annotation_descr);
else
  | annotation_descr ←
  |   get_annotation_description_in_negative_index(annotation_conf_data_structure,
  |   dimension);
  | if annotation_descr then
  |   | generate_annotation(k_AST, annotation_descr);
  | else
  |   | annotation_descr ← get_annotation_description(annotation_conf_data_structure,
  |   |   'default');
  |   | if annotation_descr then
  |   |   | generate_annotation(k_AST, annotation_descr);
  |   | end
  | end
end
```

7.6.5. Cache Blocking

We develop algorithms to handle cache blocking. Those algorithms describe parsing the corresponding configuration section from a configuration file into a data structure, and applying blocking to the corresponding loops.

Parsing blocking configuration: To apply blocking to kernels, information are extracted from configuration files. Algorithm 11 describes how the information are used to build a simple data structure that can be used by other algorithms to apply cache blocking while translating kernels. The cache blocking section is read from a configuration file. Each line in that section contains a dimension to which blocking should be applied and a blocking factor to that dimension. One entry is created from each line with this information. All those entries are held by the blocking data structure.

Algorithm 11: parse_blocking_conf

```
input  : conf_file
output : blocking_data_structure

blocking_data_structure ← new_blocking_structure();
blocking_section ← get_blocking_section(conf_file);
foreach line in blocking_section do
    new_entry ← create_blocking_entry();
    add_blocking_dimension(new_entry, line);
    add_blocking_factor(new_entry, line);
    add_blocking_entry(new_entry, blocking_data_structure);
end
return blocking_data_structure;
```

Applying blocking: The kernel translation algorithm shown in Algorithm 1 checks if cache blocking is enabled, and if that is the case it executes the *build_block_loop* algorithm (Algorithm 12) to apply cache blocking. Algorithm 12 uses information from the configuration file via the blocking data structure to build a blocking loop. Dimension and blocking factor are fetched from the data structure. Initial information about the dimension boundaries are used along with the blocking factor to build the different expressions that comprise the loop's AST node.

7.6.6. Loop Interchange

We develop algorithms to handle loop interchange. Those algorithms parse the loop interchange configuration sections into corresponding data structures, and describe how to use those data structures to apply loop interchanges.

Parsing loop interchange configuration: Information from configuration files are extracted to build a loop interchange data structure that guides the loop interchange process in Algorithm 13. This algorithm reads the corresponding section from a configuration file

Algorithm 12: build_block_loop

```
input  : blocking_data_structure, k_AST
output : k_AST

blocked_dimension ← get_blocking_dimension(blocking_data_structure);
blocking_factor ← get_blocking_factor(blocking_data_structure);
loop_index ← build_index_name(blocked_dimension);
loop_lower_bound ← get_lower_bound(blocked_dimension);
loop_upper_bound ← get_upper_bound(blocked_dimension);
lower_bound_expression ← build_lower_bound_expression(loop_lower_bound);
upper_bound_expression ← process_and_build_upper_bound(loop_upper_bound);
loop_stepping_expression ← build_loop_stepping_expression(loop_upper_bound,
  blocking_factor);
AST_node ← build_loop_node(loop_index, lower_bound_expression,
  upper_bound_expression, loop_stepping_expression);
add_AST_node(AST_node, k_AST);
```

and adds an entry per line to the data structure. The lines (and corresponding entries) describe which dimension should be traversed, defining the loop order that should be applied.

Algorithm 13: parse_loop_interchange_conf

```
input  : conf_file
output : loop_interchange_data_structure

loop_interchange_data_structure ← new_loop_interchange_structure();
loop_interchange_section ← get_loop_interchange_section(conf_file);
foreach line in loop_interchange_section do
  |  add_dimension_order(loop_interchange_data_structure, line);
end
return loop_interchange_data_structure;
```

Applying loop interchanges: The data structure that is built using Algorithm 13 is used by Algorithm 14 to apply the described loop interchange. This algorithm is executed by Algorithm 1 after making sure that a loop interchanged is to be applied. An input loop nest descriptor is provided to this algorithm with an initial loop order, based on which the algorithm generates a new processed copy of that loop nest descriptor. The algorithm copies the needed dimensions from the initial descriptor to the new one in the order that is described by the loop interchange data structure.

7.6.7. Memory Layout

High-level code is written using GGDML high-level indices, which carry no information regarding the actual location of data elements in memory. To transform source code into a form that uses a particular data layout in memory, transformations are guided by descriptions through configuration files.

Algorithm 14: interchange_loops

```
input  : default_loop_nest_descriptor, loop_interchange_data_structure
output : loop_nest_descriptor

loop_nest_descriptor ← new_loop_nest_descriptor();
foreach interchange in loop_interchange_data_structure do
  | copy_reorder_loop(default_loop_nest_descriptor, loop_nest_descriptor, interchange);
end
return loop_nest_descriptor;
```

Parsing memory-layout transformation configuration: The algorithm (Algorithm 15) describes how the configuration information are used to build a data structure to allow memory layout transformation when translating field access indices. In this algorithm, the memory layout section is read from a configuration file. The different layouts are described in subsections. Each memory layout comprises one or more indices, each of which is described by a formula in one line. Those indices will be the actual indices that the code will use to access data in arrays. What the algorithm does is building a data structure that contains the data layouts, and the list of indices in each layout. This data structure can later be used to apply the formulae to construct the indices through the memory layout transformation algorithm (Algorithm 16).

Algorithm 15: parse_memlayout_conf

```
input  : conf_file
output : layout_conf_struct

layout_conf_struct ← create_empty_struct();
foreach layout in conf_file do
  | index_list ← new_empty_list();
  | foreach index in layout do
  |   | add(index_list, index);
  | end
  | add(layout, layout_conf_struct);
end
return layout_conf_struct;
```

Applying memory layout transformations: Actual use of memory layout transformation is applied when a field is accessed. The *process_body* algorithm (Algorithm 2) calls the layout transformation algorithm (Algorithm 16).

The layout transformation algorithm starts with an existing AST node for the field access, a list of current indices (with default/initial memory layout), and a memory layout data structure. The algorithm applies the transformation that is described via a configuration file to the node which comes from the source code, and applies the optimization. A new empty list of indices is created to be filled according to the current indices and the transformation data structure. Field name is used to fetch field information from the symbol table, where the corresponding declaration specifiers allow to identify the memory layout that will be applied to the field. After choosing the layout

Algorithm 16: transform_memory_layout

```
input  : field_node, index_list, layout_conf_struct, symbol_table
output : new_node

layout ← find_corresponding_layout(layout_conf_struct, field_node, symbol_table);
new_index_list ← create_empty_list();
foreach index in layout do
    index_expression ← extract_formula(index);
    foreach placeholder in index_expression do
        substitute(placeholders, index_expression, index_list);
    end
    add(index_expression, new_index_list);
end
new_node ← apply_indices (field_node, new_index_list);
return new_node;
```

from the layout data structure, each index in the layout is processed. Processing the indices is done by fetching the formula that describes the index, substituting placeholders, and adding a new index to the list of the generated indices based on the formula. After processing all the indices in the layout, the new list of indices is used to build a new version of the field access node. The new field access node is built with a set of indices that conforms to the definition of the memory layout transformation.

7.6.8. Inlining and Loop Fusion

To exploit inter-kernel optimization opportunities, our translation process comprises a special optimizer. This optimizer exposes the functionality to analyze possibilities to apply call inlinings and loop fusions, and provides the interfaces which if called makes the necessary transformations to apply a call inlining or a loop fusion. This optimizer is activated or deactivated by users during code translation.

During code translation, the tools parse the different code files into AST structures. Inlining possibilities are checked by the inter-kernel optimizer, if activated, by analysis of calls and function bodies. A call to a function, the body of which is defined even in a different code file, could be a candidate for inlining. Close loops traversing same ranges are also analyzed for loop fusion possibilities. This analysis includes all data dependencies within loops, and possibilities to move code that resides between loops. If the loop fusion analysis is found to keep consistency of code, the fusion is listed as a candidate fusion. Inlining and fusion candidates are listed for the user to choose what to apply. According to user choice, the tool automatically uses analysis information to apply necessary transformations, including handling necessary variables, moving code around, transforming loops etc. Algorithm 17 and Algorithm 18 describe the analysis to detect possible inlinings and fusions.

Algorithm 17: check_inlining

```
input  : AST_list
output : possible_inlinings

function_definition_list ← create_empty_list();
foreach AST in AST_list do
    foreach function_definition in AST do
        add_function_definition(function_definition, function_definition_list);
    end
end
foreach AST in AST_list do
    foreach function_call in AST do
        if f_inline(function_call, function_definition_list) then
            add_possible_inlining_entry(function_call, possible_inlinings);
        end
    end
end
return possible_inlinings;
```

Algorithm 18: check_fusions

```
input  : AST_list
output : possible_fusions

foreach AST in AST_list do
    foreach loop in AST do
        neighboring_loop ← find_neighboring_loop(loop, AST);
        if loop_range(loop) is same as loop_range(neighboring_loop) then
            consistent ← check_consistency(loop, neighboring_loop, AST);    /* parameters
             are used for analysis of loop bodies and statements in between */
            if consistent then
                add_possible_fusion(loop, neighboring_loop, possible_fusions);
            end
        end
    end
end
return possible_fusions;
```

7.6.9. Scaling on Multiple Nodes

The developers responsibility to track data location, to communicate data between nodes, and to use the right memory indices to access data locally is shifted to the tools through the semantics of the GGDML extensions. Depending on the domain decomposition method, an access operator leads to identify the needed communication if any. For example, *north_edge* is sufficient to let a tool know that the data of the edges should be communicated when the edges of a set of cells reside on a different node when dividing the surface into sub-domains. To do this, we develop (Algorithm 19) to infer some information from the AST and use this information to generate the necessary code to handle the communication.

Algorithm 19: Necessary communication detection algorithm - *[JK19b]*

```
/* traverse the iterator AST                                              */
foreach AST_node in iterator_subtree do
    /* if the node is an expression to access a field data                 */
    if AST_node is a field_access_expression then
        /* get field name, list of indices, and access type               */
        field_name ← get_field_name(AST_node);
        access_type ← get_access_type(AST_node);          /* e.g., read */
        index_node_list ← get_index_node_list(AST_node);
        /* iterate over the access indices                                */
        foreach index_node in index_node_list do
            /* use indices to identify necessary communication            */
            if is_GGDML_index(index_node) then
                /* build a list of access operators                       */
                AO_list ← fetch_access_operator_list(index_node);
                /* check all access operators if they require halo exchange */
                foreach AO in AO_list do
                    if is_access_operator_a_probable_halo_exchange_reason(AO) then
                        add_entry_to_needed_halo_exchange_list(AO, field_name,
                          access_type);
                    end
                end
            end
        end
    end
end
/* check redundancies and dependencies                                    */
analyze_and_rebuild_needed_halo_exchange_list();
/* generate code to handle communication                                  */
generate_code_halo_pattern_communication_code();
```

Identifying and handling needed communication: In this algorithm, we look for data access expressions and process all the access operators used to access data. This processing includes checking if the access operator corresponds to a halo pattern. Information is logged in a list about the variable, e.g., whether we need to read some halo region from a different node. This list is further processed to analyze dependencies and redundancies to optimize communication. Finally, code is generated to handle the communication. The generated code includes the necessary data preparations and calls to communication library routines, e.g, MPI_Isend or MPI_Irecv.

To demonstrate the work of the algorithm, lets take a look at the example code shown in Listing 7.33. Assume in an application we need to use a staggered grid to compute the divergence at the centers of the grid cells based on flux values which reside on the edges between the grid cells. In this case, we can define a set of access operators to support this application, e.g., *east_edge*, *north_edge*, *west_edge*, and *south_edge*. Using those access operators, the kernel can be written as shown in Listing 7.33. The new access operators define new spatial relationships that allowed access to the cell edges.

Listing 7.33: **Example GGDML code using access operators in a staggered grid**

```
1  // Traverse the cells of the grid
2  foreach c in grid{
3        // Use GGDML access operators east_edge & west_edge
4        //  to refer to the U edges of the cell
5        float df = (f_F[c.east_edge()] -
6                    f_F[c.west_edge()]) / dx;
7
8        // Use GGDML access operators north_edge & south_edge
9        //  to refer to the V edges of the cell
10       float dg = (f_G[c.north_edge()] -
11                   f_G[c.south_edge()]) / dy;
12
13       f_HT[c] = df + dg;
14 }
```

Assume applying a domain decomposition of the Y dimension, where a set of consecutive X-rows is stored on a node and processed on it. Based on this domain decomposition and the relationships between the cells and their edges, the expression $f\_G[c.north\_edge()]$ means an X-row of edges (the halo/south-most row) should be communicated from the node that is responsible for the north neighborhood. The translation process generates the necessary MPI code (as shown in Listing 7.32) to handle the needed communication.

Some data access expressions imply the need to access halo data which resides on the same node, which does not need MPI communication. In this case, a normal data copy can be done. For example, the access operator *east_edge* in the expression $f\_F[c.east\_edge()]$ and the mentioned domain decomposition case means the cells at the rightmost column needs to access their right edges. In this application, we use periodic boundaries in which the rightmost edge of a row is itself the leftmost one. This means, copying those edges allows the rightmost cells to access edges using the same computational kernel. Again the translation process generates the following code (Listing 7.34) to copy the data of those halo edges.

Listing 7.34: **Generated data copy from example code in Listing 7.33**

```
1        for (int j = 0; j < local_Y_Eregion; j++) {
2            f_F[j][GRIDX] = f_F[j][0];
3        }
```

After the necessary data is ready in memory on the processing node to execute the computation, the compute kernel can be run. To improve this in lengthy communication cases, the communication code time can be overlapped with the computation time, given that inner regions do not depend on the data that should be communicated. In this case, the computation of the outer region (which depends on halo data) should start after the communication is finished. The idea is demonstrated in Figure 7.2. In this figure, all the neighbors of the point $P2$ (any point in the inner region) are on the same node. The south neighbor of the point $P1$ (and the other points in the outer region) is stored on another node, therefore it should be communicated first. So, the computation

of the inner region can be started, the communication of the halo is done concurrently, and the computation of the outer region is started after the communication is finished. If the communication time is shorter than the computation of the inner region, the communication time will be hidden.

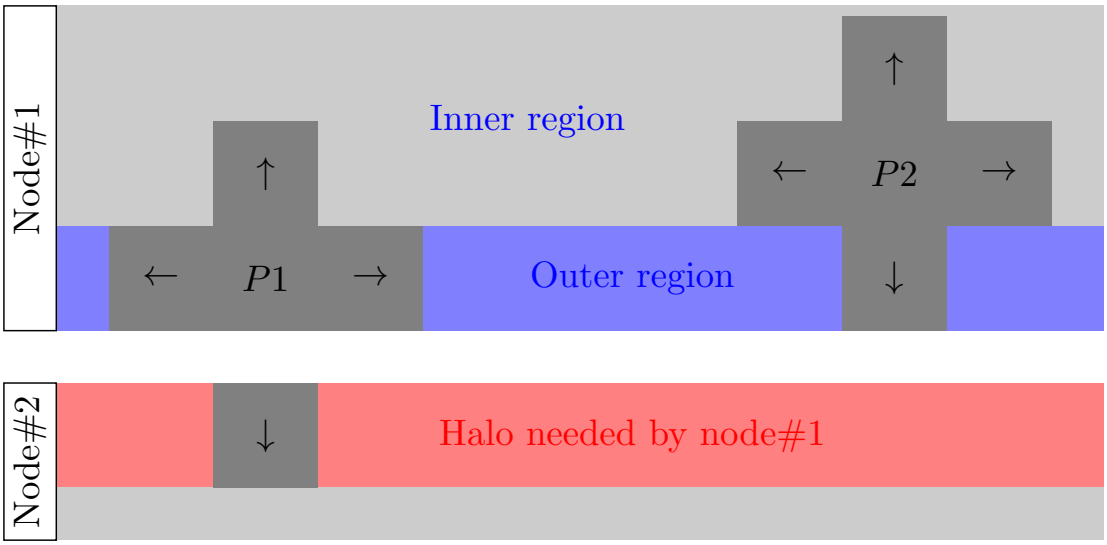


Figure 7.2.: Communication-computation overlapping

The computation kernel that is generated from the example code in Listing 7.33 is shown in Listing 7.35.

Listing 7.35: **Generated computing code from example code in Listing 7.33**

```
for (size_t blk_start = (0);
     blk_start < (GRIDX);
     blk_start += 20000) {
    size_t blk_end = GRIDX;
    if ((blk_end - blk_start) > 20000) blk_end = blk_start + 20000;
    #pragma omp parallel for
    for (size_t YD_index = (0);
         YD_index < local_Y_Cregion;
         YD_index++) {
        #pragma omp simd
        for (size_t XD_index= blk_start;
             XD_index < blk_end;
             XD_index++){
            {
                float df = (f_F[YD_index][XD_index + 1] -
                            f_F[YD_index][XD_index]) / dx;
                float dg = (f_G[YD_index + 1][XD_index] -
                            f_G[YD_index][XD_index]) / dy;
                f_HT[YD_index][XD_index] = df + dg;
            }
```

```
21             }
22         }
23 }
```

Chapter summary

In this chapter, we discussed the design drivers behind the design of the translation process. We also discussed the high-level design of the translation process. Next, we discussed the contents of configuration files that guide the translation process. Then we formulated the relationships between information extracted from source code and information fetched from configuration files and the optimization aspects. We concluded the chapter with details of how to match the inputs to transform code to exploit target hardware and to support scalability over multiple nodes.

The translation process we introduced in this chapter is a code transformation process rather than a code generation framework. Rather than providing a new language or rules to specify stencils, scientists use their preferred modeling language. Source code is written with host language expressions and operators as usual. Advantages are provided via language extensions which simplify coding for scientists, e.g., indices to access neighborhoods, and drive the code transformation, e.g., iterators to apply blocking to loop nests. Source code is parsed within the translation process as compilers do, with the exception that additional rules are used to understand the language extensions. Occurrences of the language extensions guide the tools to apply the transformations to the ASTs. After the transformations are applied, the ASTs could be unparsed into a new version of the code that consists of the modeling language without language extensions.

In the next chapter we validate our techniques through experimental work and theoretical analysis.

8. Validation

In this chapter, we discuss work done to validate our approach and developed techniques and designs. We start with describing our validation plan in Section 8.1. Then, we evaluate the impact of using our high-level language extensions on quality of code and estimated impact on development costs in Section 8.2. Next, we discuss some experiments that we executed to evaluate performance of code that is written using GGDML and processed with the discussed techniques in Section 8.3. We conclude the chapter in Section 8.4 with a discussion of the performance portability of code using our techniques.

8.1. Validation Plan

To judge the validity of our work, techniques, and answers, the objectives of the work are referenced to find out how well could we achieve them. Our work targets mainly to enable performance-portable coding, and scalable code to support modern multi-node machines, while improving the quality of code. Code quality includes reducing the effort that the scientists need to write and maintain code, and hence the cost of model development and maintainability.

Evaluating what we have done compared to the objectives is done with

- evaluating the impact of the using the language extensions on the quality of code and projections of development costs
- evaluating the achievable performance of the high-level code on different configurations and architectures, including multi-node machines
- evaluating the achievable performance of same high-level code on different architecture with respect to expected performance of code optimized for each architecture to evaluate performance portability.

First we start with evaluating code quality aspects. Under code quality evaluation we compare codes developed with our language extensions with same codes developed with original modeling language. We also carry out some analysis of projected development costs using the Constructive Cost Model (COCOMO) [B+81].

To evaluate performance, we measure performance of a single operator and a complete application on different systems. A set of key optimization techniques is used. The impact of applying each of those optimization techniques is evaluated by applying the technique through changing the configuration files while using the same source code. The optimization techniques are chosen based on the nature of stencil computations, which are memory bound. Therefore, we found that viable techniques to evaluate are

- cache blocking, optimal data layout, loop order and vectorization at stencil level
- inter-kernel optimization at application level
- and scalability across nodes.

Efficient use of memory bandwidth within each stencil and exploiting the data reuse across stencils allows optimal use of node resources. Using the same source code to run on multiple nodes with optimized used of node resources allows to optimal use of underlying machine. Therefore, we believe that evaluating the impact of those optimization techniques reflects the success to achieve performance through our techniques. Throughout the performance evaluation, empirical results are recorded and are analyzed in comparison to expected performance according to theoretical analysis.

To evaluate performance portability we use two performance portability metrics. One metric reflects achieved performance portability across a set of architectures. The other metric compares performance portability of code on multiple nodes on two different machines.

8.2. Impact on Code Quality and Development Costs

Quality of code is an important factor to validate when evaluating the developed language extensions, because of the development and maintenance impact. Lines of code (LOC) is a simple metric to measure the impact of using the DSL on the quality of code.

To evaluate the projected impact of using our language extensions on code quality and costs of model development, at an early stage of our work we prepared some initial estimations. We used LOC of original code from existing models in comparison to same codes that were rewrote using our language extensions for that purpose. We took two relevant kernels from each of the three icosahedral models, DYNAMICO, ICON, and NICAM, and analyzed the achieved code reduction in terms of LOC ([JKZ+17]). We rewrote the kernels, which were originally written in Fortran, using GGDML + Fortran. LOC comparison results are shown in Figure 8.1.

The average reduction in terms of LOC is 70%, i.e, LOC in GGDML+Fortran is 30% of that of the original Fortran code. More reduction is noticed in some stencils, e.g, NICAM example No.2, reduced to 12%.

Influence on development costs: In addition to quality of code, it is useful to estimate the benefits resulting from the code reductions when using GGDML for model development in terms of development costs. To estimate budget benefits, we use the Constructive Cost Model (COCOMO) [B+81] as a model to estimate complexity of development effort

Much of the contents here under the title "Impact on Code Quality and Development Costs" are published in [JKZ+17].

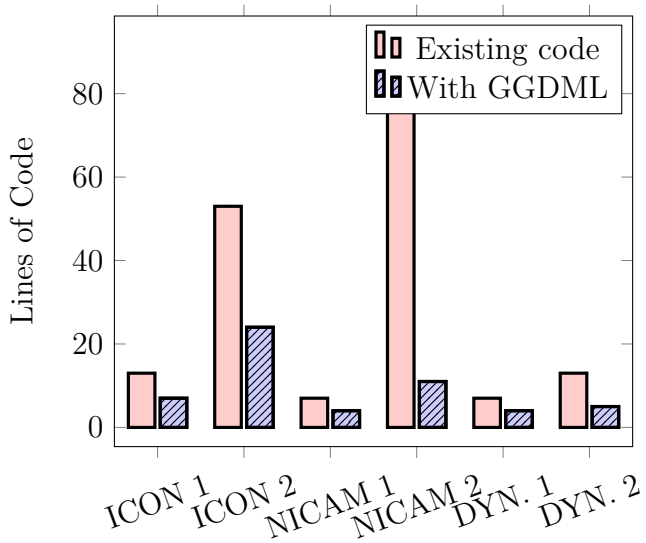


Figure 8.1.: GGDML impact on the LOC on several scientific kernels [JKZ+17]

and costs. We make our estimations based on characteristics of a model with comparable structure of the ICON model.

We apply the formulae

$$E = a \cdot KLOC^b \tag{8.1}$$

$$D = c \cdot E^d \tag{8.2}$$

$$P = \frac{E}{D} \tag{8.3}$$

where E is the applied effort in person months, D is the estimated development time in months, P is the number of people required, and the model values a, b, c, and d are constants with the values shown in Table 8.1.

	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 8.1.: Basic COCOMO model constants [AQQ13, KD16]

Our estimations are shown in Table 8.2. The table shows the effort in person month, development time and average number of people (rounded) for three development modes: the embedded model is typically for large project teams working on a big and complex code base, the organic model for small code and the semi-detached mode for in-between. We assume the semi-detached model is appropriate, but as COCOMO was developed for industry projects, we don't want to restrict the development model. The estimations are

based on a code with 400 KLOC, where 300 KLOC of the code are the scientific portion that allows for code reduction while 100 KLOC are infrastructure.

From the predicted developed effort, it is apparent that the code reductions would be leading to a significant effort and cost reduction that would justify the development and investment in DSL concepts and tools.

Development Style	**Code-base**	**Effort applied (person month)**	**Dev. Time (months)**	**People required**	**Dev. costs (M€)**
Embedded	Fortran	4773	37.6	127	23.9
	DSL	1133	28.8	72	10.4
Semi-detached	Fortran	2462	38.5	64	12.3
	DSL	1133	29.3	39	5.7
Organic	Fortran	1295	38.1	34	6.5
	DSL	625	28.9	22	3.1

Table 8.2.: COCOMO cost estimates [JKZ+17]

Source code LOC comparison with optimized code generated for different targets: In addition to the initial code size predictions at an early stage in our work, we made later another comparison of a complete prototype code which is worth being shown. The purpose is to show the ratio of source code to generated code versions for different architectures or configurations. Table 8.3 show a comparison of the sizes of the different code versions.

Source code in this application is written using GGDML and C. Generated codes are in C language. Two kernels (denoted Kernel1 and Kernel2) of different sizes are considered in the comparison besides to the application itself.

Code	GGDML	MPI	GASPI	GPU(multiple nodes)	VE(single node)
Kernel1	5	20	20	19	14
Kernel2	10	148	218	180	21
Application	161	779	946	748	531

Table 8.3.: LOC of the GGDML application code vs. the generated GPL code for different target platforms

8.3. Performance Evaluation

Among the points we investigate in this work is how to deliver semantics to allow tools to optimize code. Therefore it is important to evaluate the success to provide performance. In this section we discuss some experiments and measurements to evaluate performance under different configurations and architectures.

8.3.1. Test Codes

We used the code of a Laplacian solver, and a shallow water equation solver to do our experiments. The Laplacian represents a solver of a mathematical operator, within a component of a model. The shallow water equation solver, which is a well-known small full model among the simplest earth system modeling problems, allows to evaluate a complete model.

Laplacian Solver: The first code is written with an unstructured triangular grid covering the surface of the globe. The application was used in the experiments to apply the Laplacian operator of a field at the cell centers based on field values at neighboring cells. Generally, this code includes fields that are localized at the cell centers, and on the edges of the cells. The horizontal grid of the globe surface is mapped to a one dimensional array using Hilbert space-filling-curve. We used 1,048,576 grid points (and more points over multiple-node runs) to discretize the surface of the globe. The code is written with 64 vertical levels. The surface is divided into blocks.

Shallow water equation solver: The other code is the shallow water equation solver. It is developed with a structured grid. Structured grids are also important to study for icosahedral modeling, as some icosahedral grids can be structured. Fields are located at centers of cells and on edges between cells. This solver uses the finite difference method. The source code of this application is available online on Github[1] and in Appendix A. As part of the testing, we investigate performance and performance portability of code developed using the DSL.

8.3.2. Test Systems

The experiments were executed in different times during the course of the research work and used different machines based on availability and architectural features.

- Mistral
  The German Climate Computing Center (DKRZ) provides nodes with Intel(R) Xeon(R) E5-2695 v4 (Broadwell) @ 2.1GHz Broadwell processors.
- A test machine at Erlangen regional computing center (RRZE)
  At Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), where we used Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz Broadwell processors.
- PSG cluster
  From NVIDIA, equipped with Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz Haswell processors and different GPU types (we used P100 and V100 GPUs).

[1] `https://github.com/aimes-project/ShallowWaterEquations`

- Piz Daint
  The Swiss supercomputer provides nodes equipped with two Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz processors and NVIDIA(R) Tesla(R) P100 GPUs.

- A test system from NEC Deutschland
  Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz Broadwell processors with SX-Aurora TSUBASA vector engines.

8.3.3. Cache Blocking

To evaluate the impact of applying the cache blocking optimization procedure while translating the GGDML code, we carried out a set of experiments. The experiments that we show are executed on the shallow water equation solver. All the configurations that we prepared for the prototype application in which the Laplacian operator is included were prepared with blocking, therefore we did not evaluate comparisons of code with and without blocking for the Laplacian operator.

8.3.3.1. Multi-core processors

First, we executed the cache blocking experiments on Broadwell processors. In the first experiment we varied the width of the grid and measured the performance[2] of the application. The measured performance before and after blocking (with a blocking factor of 20K) are shown in Figure 8.2.

Before blocking, the measurements show that the performance decreased with wider grids. This is explained by loading data multiple times with wider grids, which can be avoided with applying cache blocking using a suitable blocking factor.

The processor has 2.5 MB L3 cache per core. Using a blocking factor of 20K, with kernels accessing 5 fields at most, where field data is stored in single precision floating point format, a grid row needs 0.38 MB of cache memory. The choice of the blocking factor guarantees that more than two grid rows exist in L3 cache while it is still needed, i.e, all stencil components are guaranteed to be still in cache. This reduces the access to main memory and allows to minimize the running time of each stencil, and hence, the application. The impact of using the cache blocking is clear in the figure, where we could keep the performance from dropping over wider grids.

Blocking factors: To understand the impact of using different blocking factors on the performance, we fixed the grid width and varied the blocking factor. The measured performance of the application is shown in Figure 8.3.

The measurements show that small blocking factors, smaller than 1K, harm the performance of the application. Similarly, large factors, larger than 32K, lead to lower performance. The calculations made for the 20K do not hold for large factors, e.g, 64K.

[2]The performance measurements within the various experiments mentioned in this thesis were computed within code and measured with tools using performance counters, e.g., Likwid, and both measurements were comparable.

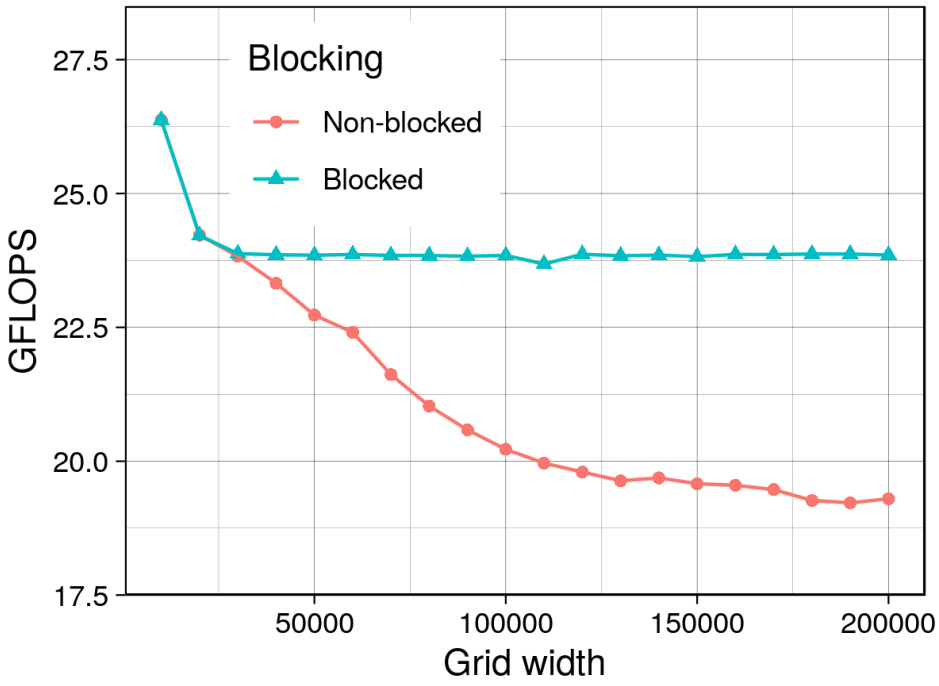


Figure 8.2.: Variable grid width with and without blocking on Broadwell

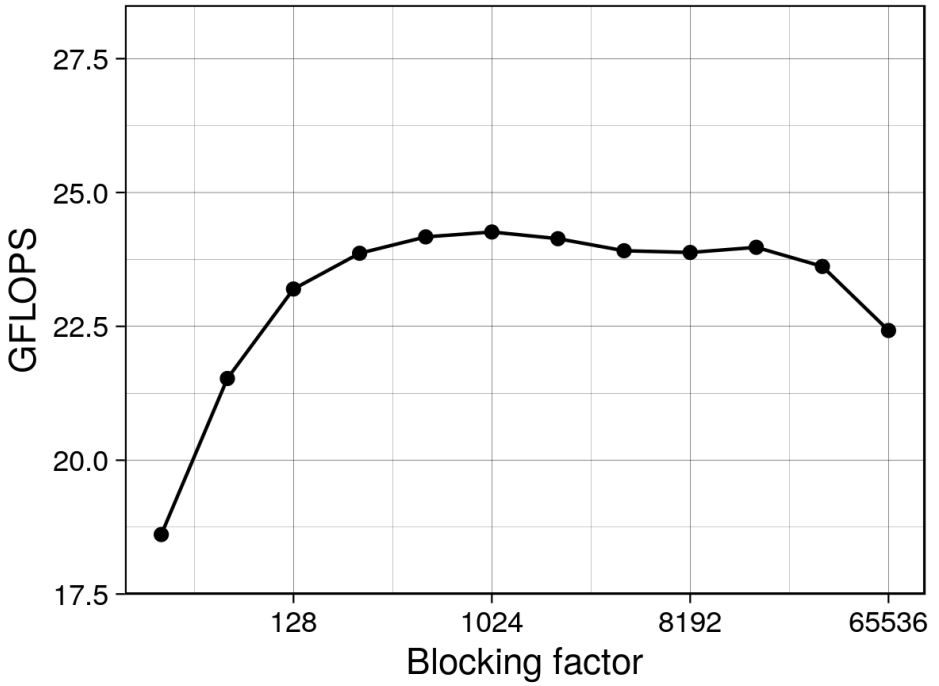


Figure 8.3.: Different blocking factors on Broadwell

Such factors will lead to exceed the L3 cache capacity, hence we see the performance drop.

8.3.3.2. GPUs

Again, we used our tools to apply cache blocking while generating code for GPUs and executed experiments to measure performance. We executed experiments on the Tesla P100 with 16 GB memory and PCIe interconnect to the host, using the PGI (17.7.0) C compiler. In the first experiment we varied the grid width and measured performance of code both before and after blocking (20K blocking factor). The measurements are shown in Figure 8.4.

Before blocking, the performance was dropping under wider grids. This is because of the limitations of cache capacity, which causes loading field data multiple times. After blocking, performance was retained over wider grids. Blocking the code with a factor of 20K, with kernels accessing 5 fields at most, where field data is stored in single precision floating point format, a grid row needs 0.38 MB of cache memory. With the 4 MB L2 cache of the P100 GPU, it is possible to store multiple grid rows. This means when a stencil is computed, all its points are stored in cache, and no access to device memory is needed, decreasing the time to run the kernels and hence the application. The figure also shows the bad performance when blocking grids with smaller widths. This is a result from loading data from other blocks multiple times, which leads to load more data from device memory each time we increase the block count (decrease block size).

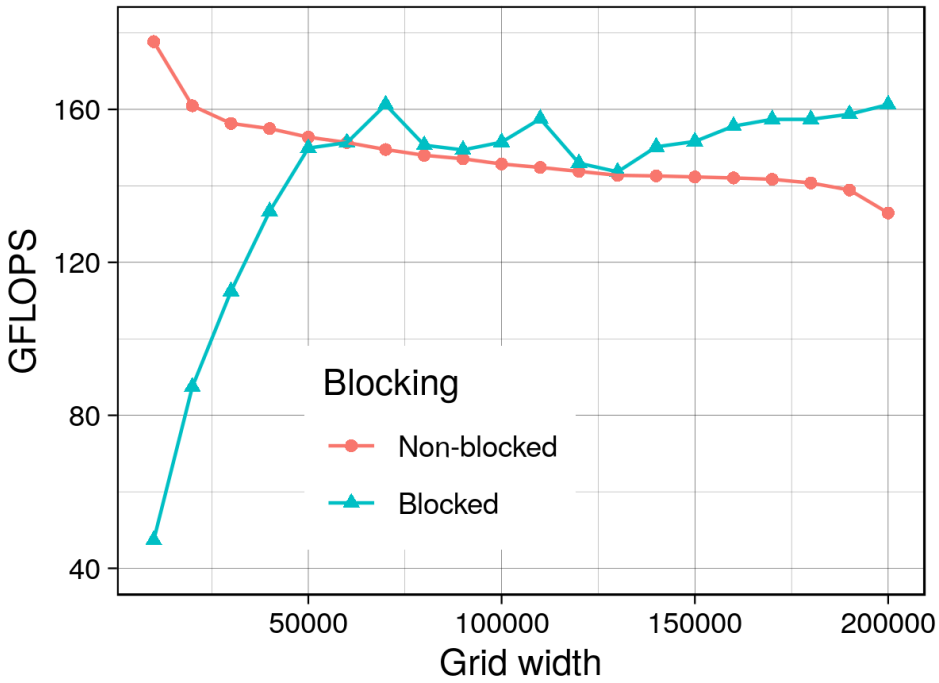


Figure 8.4.: Different grid widths with and without blocking on P100 GPU

Blocking factors: After testing different grid widths, we fixed the grid width and varied the blocking factor to understand how changing the blocking factor affects the performance. We executed the experiments and recorded measured performance (shown in Figure 8.5). The measurements show a narrower range of acceptable blocking factors in comparison to that we measured on Broadwell processors. This difference stems from the dynamic scheduling according to data availability to warps on GPUs, which generally gives the advantage to GPUs to beat the data access latency. Smaller blocks limit the dynamic scheduling to process stencils efficiently, decreasing performance. An important point we deduce from this is that GPUs are more sensitive to the choice of the blocking factor than multi-core processors, and even to apply blocking at all (for smaller grid widths).

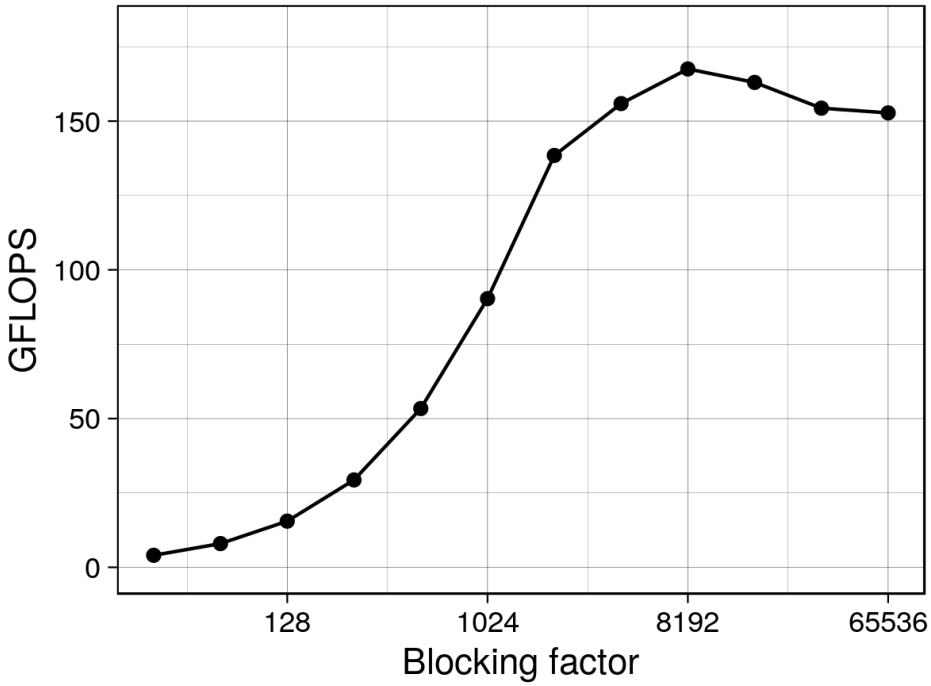


Figure 8.5.: Different blocking factors on P100 GPU

8.3.4. Inter-kernel Optimization

To further optimize the use of the memory bandwidth, inter-kernel optimization improves the application-level performance. To validate the techniques we apply through our translation process, we executed a set of experiments on the shallow water equation solver. We investigated the techniques on Broadwell multi-core processors, P100 GPUs, and on SX-Aurora vector engines.

The contents here under the title "Inter-kernel Optimization" are published in [JK20].

8.3.4.1. Multi-core Processors

The multi-core processor experiments were run on the Broadwell processors, using the Intel (ICC 17.0.5) C compiler. First, we evaluate the code generated for Broadwell with different grid widths, investigating kernel merging impact both with and without cache blocking. The results before and after blocking (block size of 20K) are shown in Figure 8.6.

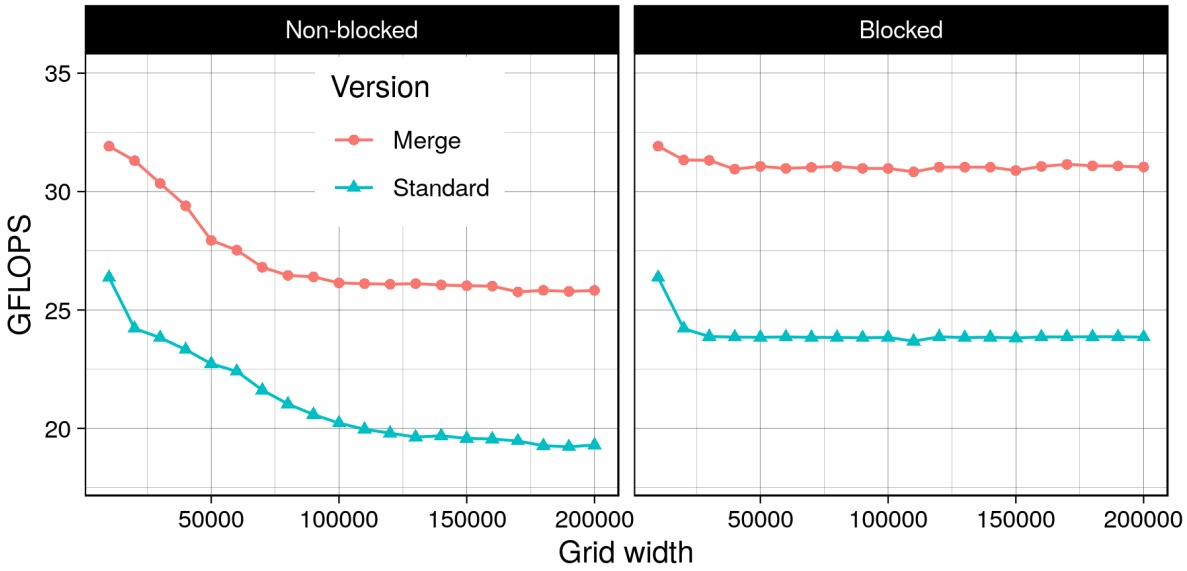


Figure 8.6.: Variable grid width with and without blocking/merging on Broadwell

Merging the kernels results in the expected code optimization reducing the necessary memory traffic over all grid widths. Without blocking, the results of the measurements show that the performance decreased with wider grids since the capacity of the caches is exhausted. Appropriate blocking eliminates performance loss. Given that the data are stored as single precision floating point, and that the maximum number of fields to access within a kernel is eight (when merging kernels), the 20K block width means the cache holds 0.61 MB per grid row. The processor has 2.5 MB L3 cache per core. Therefore, the 20K blocking factor guarantees that more than two grid rows, and hence all the elements of the stencil (both in X and Y dimensions) are still in the L3 caches.

Varying blocking factors: To better understand kernel merging along with blocking relationship, we varied the block sizes. We fixed the grid width to 100k cells in the X dimension. Results are shown in Figure 8.7. Kernel merging provided performance improvement over all the tested blocking factors.

Theoretical analysis: To understand the data movement between the cores and the main memory we instrumented the code with 'Likwid'. The measured metrics and values for the different kernels are shown in Table 8.4.

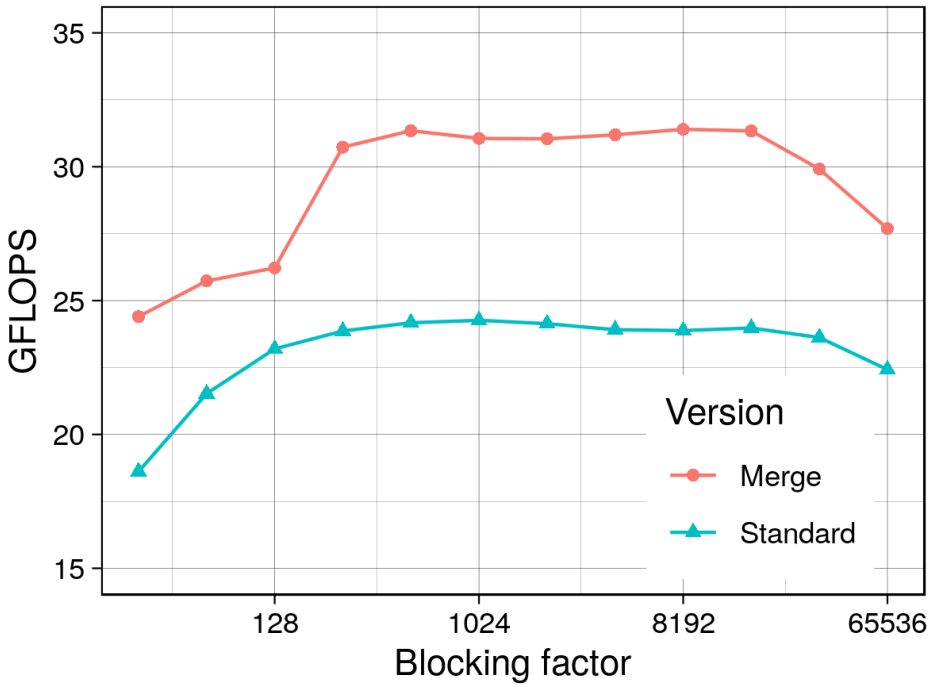


Figure 8.7.: Different blocking factors on Broadwell with and without merging

Kernel	Time (s)	GFLOPS	Memory Bandwidth (GB/s)
flux1	26.9	11.2	59.8
flux2	26.6	11.3	62.8
compute_U_tendency	41.3	41.2	62.3
update_U	19.5	10.3	62.8
compute_V_tendency	46.4	36.7	61.8
update_V	19.3	10.3	63.3
compute_H_tendency	26.6	11.3	62.9
update_H	19.8	10.1	62.4
Standard_code	226.3	23.8	62.2
flux_and_tendencies	96.9	41.3	59.5
velocities	39.6	10.1	61.3
compute_surface	40.6	12.3	60.7
Merged_code	177.0	31.0	60.2

Table 8.4.: Likwid instrumentation on Broadwell for kernels with and without merging

The kernels are bound by the memory bandwidth. Theoretical maximum memory bandwidth of the Broadwell processor is 76.8 GB/s[3]. The kernels are optimized to read each variable only once from memory. For example the kernel $flux1$ accesses the memory to read two fields –reused more than once– and update one field. Multiplying the number of bytes accessed per grid cell by the grid dimensions and the time steps, this kernel needs to access 1491 GB during an application run. To compare with the measured values, if we multiply the kernel's runtime (26.86 s) by the measured memory bandwidth (61.22 GB/s), we find that the kernel accessed 1605 GB which is close to the theoretical calculations.

The achieved memory throughput of the code is close to the optimum. As long as we access the minimum amount of data in the memory with a high percentage of max memory bandwidth, the only way to optimize the code further is to decrease number of memory accesses for the application level.

In the standard code version, we need 33 accesses to the main memory for each grid cell in each time step. The arithmetic intensity of the code is 0.45 FLOP/Byte. Given the peak processor performance (2.3 GHz · 18 cores · 16 Single FP/core · 2(fused multiply–add)) and the memory bandwidth (76.8 GB/s), the threshold arithmetic intensity to achieve the peak performance is 17.25 FLOP/Byte. The arithmetic intensity of the code is far from this threshold intensity, which explains why the achieved performance is far from the peak performance of the processor. Optimizations must increase the arithmetic intensity to increase the performance of the application.

What we gain in the merged code is reusing the values of some fields while they are still in the caches or the processor registers instead of reading them from the memory. This reduces the number of accesses to the main memory from 33 accesses to 24 accesses for each grid cell in each time step. This way, we can increase the intensity of the code to 0.63 FLOP/Byte. This is an increase by about 37% which explains the performance gain we can observe in the diagrams.

8.3.4.2. GPUs

To understand data movement between the GPU threads and the device memory when applying kernel merging, we prepared experiments for the P100 GPU. The GPU experiments are run on the Tesla P100 GPUs, using the PGI (17.7.0) C compiler. We first measure the performance impact of the kernel merging with different grid widths (see Figure 8.8) both with and without cache blocking. Without blocking, the performance decreases over the tested grid widths regardless of applying kernel merging. However, merged code performance degrades faster after the grid width of 110k. Performance drops beyond the standard code around the grid width of 140k. This is a result of the cache limitation on the GPU as a merged kernel accesses more variables per grid cell. A kernel that accesses 8 fields on a grid that is 140k wide, where each field needs 4 bytes per cell, needs 4.27 MB, which exceeds the 4 MB L2 cache of the P100 GPU.

[3] The streaming benchmark 'stream_sp_mem_avx' from the 'Likwid' tools measured 67 GBytes/s on the processor.

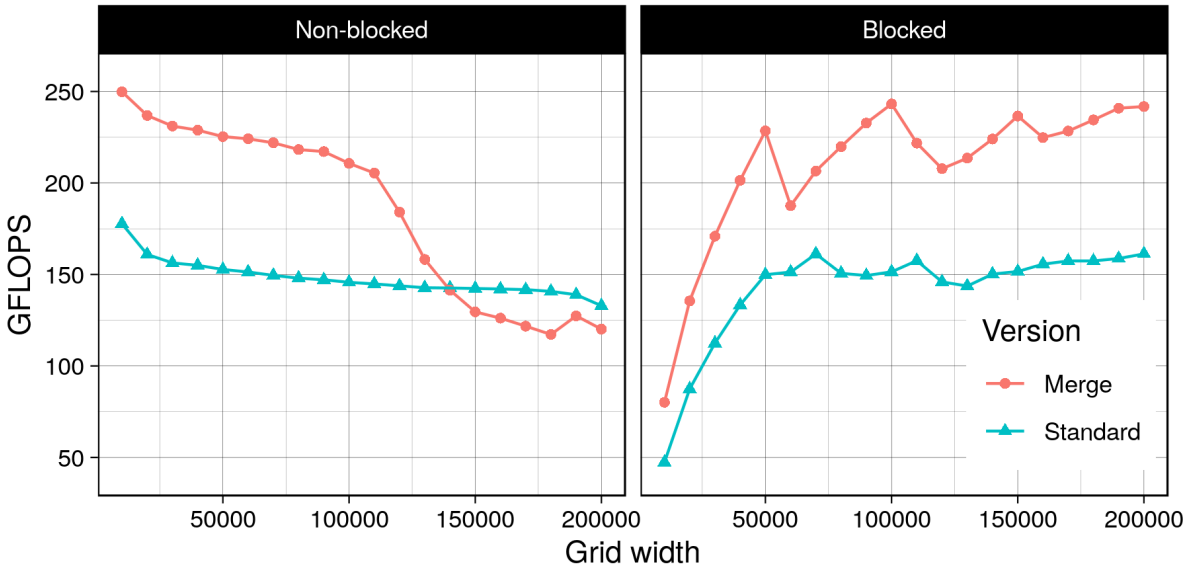


Figure 8.8.: Different grid widths on P100 GPU with and without blocking/merging

The blocking version (20k block size) does not exhibit the sharp drop over wider grids, and the merged code is better over the tested grid widths. This is a result of fitting the necessary data within the caches (remember that the 20k row in a block needs 0.61 MB for a kernel that accesses 8 fields).

Varying blocking factors: To investigate further the impact of the kernel merging along with blocking, we test different block sizes again (see Figure 8.9). In general, kernel merging improves performance with all the tested block sizes. Optimal block sizes are around 10k. Smaller (and larger) block sizes harm the performance for both code versions.

Theoretical analysis: To gain a deeper understanding the 'nvprof' tool is used to collect different metrics. Table 8.5 shows the kernels measured memory throughput and accessed data volumes. Execution times and GFLOPS are also shown.

The measured data volumes that kernels access show data reuse at warp level. For example, the $flux1$ kernel accesses the device memory to read two fields – reused within the kernel – and updates one field. The memory access is coalesced, thus, the theoretical estimation of the data volume that the threads should access during the runtime of the kernel should be 12 bytes multiplied by the grid size and by the count of the time steps, which gives 1117 GB. In comparison, the computed value based on the 'nvprof' measurements is 1175 GB as shown in the table which is close to our expectation.

All kernels are memory bound. The measured memory throughput of the P100 on the test nodes was measured with a CUDA STREAM benchmark yielding about 498 GB/s. The memory throughput that was measured for the kernels shows high percentages

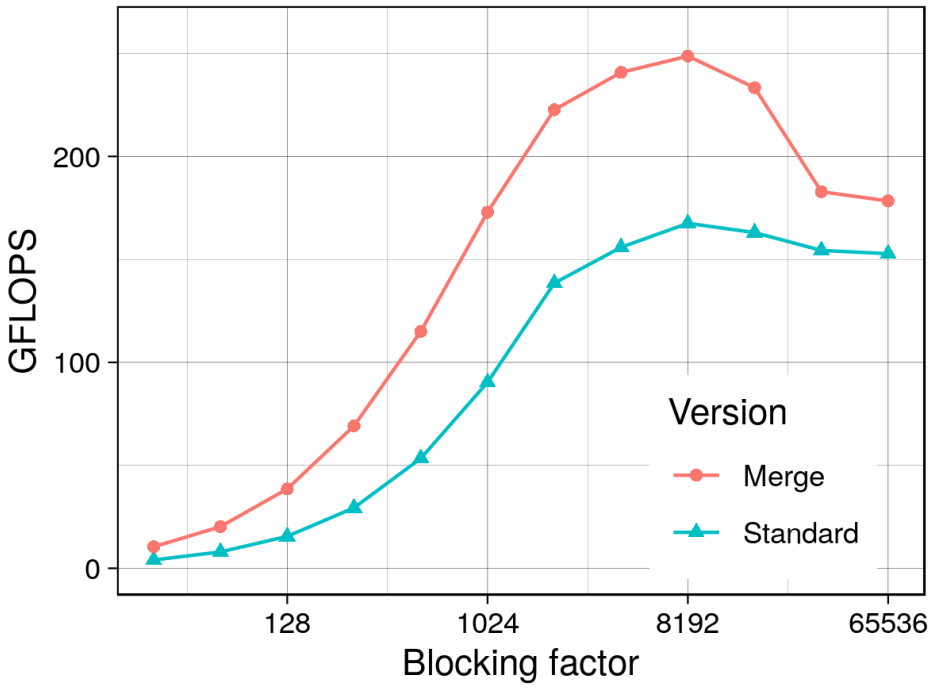


Figure 8.9.: Different blocking factors on P100 GPU with and without merging

Kernel	Memory Throughput (GB/s)	Data Volume (GB)	Kernel Time (s)	GFLOPS
flux1	447	1,175	2.63	114
flux2	478	1,570	3.29	91
compute_u_tendency	358	3,338	9.33	225
update_u	376	1,126	2.99	67
compute_v_tendency	374	4,195	11.22	196
update_v	376	1,126	3.00	67
compute_h_tendency	333	1,588	4.77	105
update_h	387	1,126	2.91	69
Standard_code	380	15,244	40.13	149
flux_and_tendencies	396	5,970	15.08	325
velocities	360	2,268	6.31	63
compute_surface	403	2,303	5.71	123
Merged_code	389	10,542	27.11	221

Table 8.5.: Metric measurements of kernels on P100 GPU with and without merging

(67%-96%) of the streaming memory throughput. Reducing device memory access leads to focus on the application-level optimization.

The data access is coalesced in all the kernels, before and after merging. With data reuse, the standard kernels access the device memory 38 times × grid cells × time steps

in total. However, the merged kernels reduce the accesses to 26. The numbers of the accesses look different from those of the Broadwell because the scheduling of the work on GPU threads is different, and hence the caching of the data is different. The access reduction explains the performance improvement between the two code versions (221 GFLOPS : 149 GFLOPS) as the arithmetic intensity is shifted from 0.39 to 0.58 through merging.

8.3.4.3. Vector Engines

The vector engine experiments were run on SX-Aurora TSUBASA vector engine using the NCC (1.3.0) C compiler. On Aurora vector engine, we vary the grid width from 10k to 100k and measure the performance (see Figure 8.10). Merging improved performance over all the grid widths. Performance is not dropping without blocking (at least at the chosen grid widths).

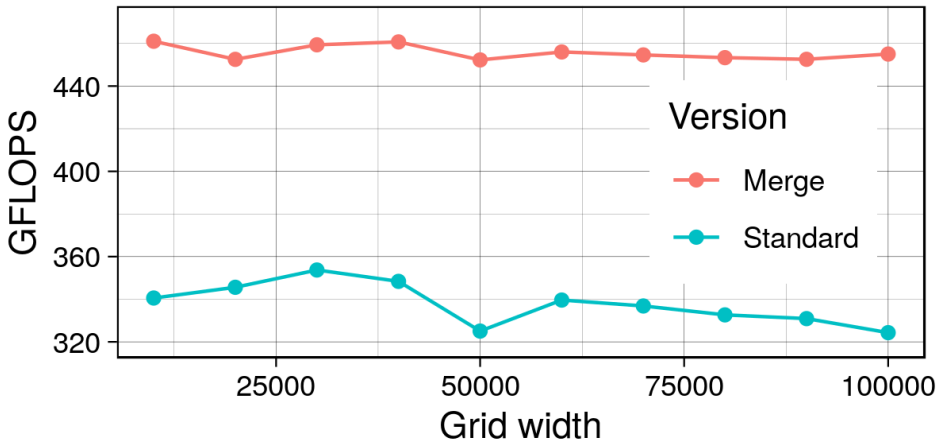


Figure 8.10.: Different grid widths with and without merging on NEC Aurora

To understand the performance, NEC's 'ftrace' tool is used (see Table 8.6). The theoretical memory bandwidth of the vector engine is 1.2 TB/s. Based on the 'ftrace' measurements, the computed values of the memory throughput show that all the kernels run with a high percentage of the memory bandwidth (80%) before and after the kernel merging.

The performance ratio before and after the kernel merging is 453 GFLOPS : 322 GFLOPS. This result is roughly the ratio of the arithmetic intensities which we discussed in the multi-core processor results (0.63 : 0.45).

8.3.5. Memory Layout, Loop Order, and Vectorization

The optimal exchange of data between memory and processors needs matching the memory layout of the data and using the right loop orders to exploit data locality. Choosing an optimal data layout and the right access order allows to efficiently use

Kernel	Time (s)	GFLOPS	Memory Throughput (GB/s)
flux1	1.30	230	858
flux2	1.51	199	989
compute_U_tendency	5.29	359	986
update_U	1.21	166	927
compute_V_tendency	5.22	384	1,001
update_V	1.21	165	924
compute_H_tendency	1.52	330	984
update_H	1.20	167	934
Standard_code	18.63	322	961
flux_and_tendencies	8.40	500	911
velocities	2.43	165	922
compute_surface	2.31	303	940
Merged_code	13.25	453	911

Table 8.6.: Kernel measurements of both code versions on the NEC Aurora

the memory bandwidth, and is necessary to exploit the vectorization capabilities of the underlying hardware. Because of the connections between those optimization aspects, we conducted experiments to understand their impact together in one section.

At an early stage of the work we started studying the impact of applying memory layout transformations using our techniques using the Laplacian operator. An experiment of those is discussed first in this section, then we discuss more experiments using the shallow water equation solver.

8.3.5.1. Early Memory Layout Experiments Using Laplacian Operator on GPUs

The experiment discussed here was done using the Laplacian operator on an unstructured grid with 1048576 surface points × 64 vertical levels. The simulations run 100 time steps (explicit time stepping scheme). The experiment was executed on P100 and V100 GPUs on the PSG cluster. We used the PGI compiler version 17.10 to compile the code.

In this experiment we study the impact of applying the memory layout transformation that we develop to represent 3D unstructured grid into a single dimension array. Two code versions were generated from the same source code, each with a different data layout

- **3D**: a three-dimensional addressing with three-dimensional array
- **3D-1D**: a transformed addressing that maps the original three-index addresses into a 1D index.

The contents here under the title "Early Memory Layout Experiments Using Laplacian Operator on GPUs" are published in [JK18]

The experiment shows performance impact of transforming the data layout on two kinds of GPUs; P100 and V100. The results are summarized in Table 8.7. The results show

	Performance (GFLOPS)		
	Serial	P100	V100
3D	1.97	220.38	854.86
3D-1D	1.99	408.15	1240.19

Table 8.7.: Data layout transformation on P100 & V100 GPUs

that applying the data layout transformation to optimize the data access on both GPU versions was successful and allowed to achieved improved performance.

8.3.5.2. Shallow Water Equation Solver Experiments

Besides to the early experiments on the PSG cluster, we developed later the shallow water equation solver and have done deeper experiments to evaluate the impact of memory layout transformations along with loop order and vectorization. Those experiments were executed on Broadwell multi-core processors and SX-Aurora vector engines.

Multi-core processors: The multi-core processor experiments are run on Broadwell processors, using the Intel C compiler (ICC 17.0.5 20170817). First, we generated the code for the multi-core processors, in three code versions: scattered access, constant short distance between array elements, and contiguous array elements. We profiled the three code versions with 'Likwid' ([THW10]). The measurements for the different kernels are shown in Table 8.8.

The first eight rows in the table show the measurements for the different kernels. The runtime and the measured GFLOPS performance of the AVX instructions are shown in the three code versions. The last row summarizes the runtime and the measured GFLOPS of the whole application (total application-level measured GFLOPS and not only AVX).

For the code with contiguous array elements, if we multiply the time by the GFLOPS we find that the first eight kernels executed around $4.9 \cdot 10^{12}$ FLOP on the AVX vector units. This is close to the measured total GFLOPS of the application. In the code with constant short distance separating its elements, some kernels (but not all) were vectorized. Performance of this code version is nearly half that of the contiguous array version. However, for the code with scattered access, the executed operations on the AVX vector units are 0, and the performance was only 12% of the performance that the unit stride code achieves.

In fact, the vectorized arithmetic operations are not the only factor of those results, but also the vectorized data movement and memory access patterns. The memory access

The contents here under the title "Shallow Water Equation Solver Experiments" are published in [JK19a].

to the non-contiguous arrays degrades the role of the caches, and hence the use of the memory bandwidth. This explains the ratio of the performance of the array with constant short distance between elements to the performance of the contiguous array code. In the code with elements separated by constant short distance, we use 4 bytes to store a single precision value and there are 4 bytes separating the values, the ratio of the needed data is 4:8. Thus, the efficiency of using the memory bandwidth is half that of contiguous array, and performance is also the same ratio. For stencil computations it is well known that the optimal use of the memory bandwidth is critical to achieve an optimal performance. This is because stencil computations are memory bound.

Given the arithmetic intensity of 0.45 FLOP/Byte of the application, and the measured memory throughputs around 68 GB/s[4] (theoretical bandwidth 76.8 GB/s), the code with contiguous arrays is nearly optimal achieving 80% of the theoretical memory bandwidth. The code generation generated vectorizable code applying a single pattern across several operations.

Kernel	Scattered		Constant short distance		Contiguous	
	Time (s)	AVX GFLOPS	Time (s)	AVX GFLOPS	Time (s)	AVX GFLOPS
flux1	250	0	52	0	27	11
flux2	248	0	54	0	27	11
compute_U_tendency	431	0	80	21	41	41
update_U	158	0	39	0	20	10
compute_V_tendency	432	0	94	18	47	37
update_V	158	0	40	0	20	10
compute_H_tendency	251	0	55	0	28	11
update_H	158	0	40	0	20	10
Application Level	2,103	3	466	13	244	25

Table 8.8.: Performance measurements for different layouts on Broadwell

Vector engines: The vector engine experiments are run on the SX-Aurora TSUBASA vector engine using the NCC (1.3.0) C compiler. We generated again three code versions of the same source code for the Aurora vector engine: scattered access, constant short distance between array elements, and contiguous array elements. 'Ftrace' was used for the measurements, in which we record performance metrics for the different kernels. Results are shown in Table 8.9.

The first eight rows show the measurements for the different kernels. The runtime and the measured GFLOPS performance are shown for both the three code versions. The last row summarizes the runtime and the measured GFLOPS of the whole application.

The arithmetic operations of all codes are executed by the vector units. However, the efficiency of using the vector units differs, where the contiguous array code is nearly twice the performance of the code with constant short distance between array elements, and four times faster than the code with scattered access. Again, the vectorization of arithmetic operations is not the only factor of this result, but also memory access

[4]According to Likwid's stream_sp_mem_avx benchmark

patterns. As with multi-core processors, the memory access to non-contiguous arrays degrades the role of the caches, and hence the use of the memory bandwidth.

As mentioned before, the arithmetic intensities of the application level are 0.45 FLOP/Byte. The theoretical memory bandwidth of the used vector engine is 1.2 TB/s. Based on the numbers, the code with contiguous array elements is nearly optimal running with a high percentage (80%) of the theoretical memory bandwidth.

	Scattered		Constant short distance		Contiguous	
Kernel	Time (s)	GFLOPS	Time (s)	GFLOPS	Time (s)	GFLOPS
flux1	5.37	56	3.96	76	1.30	230
flux2	5.36	56	4.08	74	1.51	199
compute_U_tendency	20.67	92	8.26	230	5.29	359
update_U	3.82	52	2.44	82	1.21	166
compute_V_tendency	20.66	97	9.12	220	5.22	384
update_V	3.82	52	2.43	82	1.21	165
compute_H_tendency	6.88	73	4.26	117	1.52	330
update_H	3.82	52	2.44	82	1.20	167
Application level	70.40	80	37.17	161	18.63	322

Table 8.9.: Performance measurements of different layouts on the NEC Aurora

Results show the impact of generating unit stride code, which provides twice the performance of codes with constant short distance separating array elements, and about eight times the performance of scattered access on the Broadwell processors. On the Aurora vector engine, the unit stride code achieves twice the performance of the code with constant short distance between array elements, and four times the performance of the code with scattered access.

The experiments spotlight the importance of matching data layout and memory access patterns for yielding vectorizable code and efficient use of memory bandwidth. The performance impact of changing the array stride and data layout and access in the generated codes is of great importance. Our tools and techniques prove to handle those optimization aspects well, while making use of the high-level GGDML language extensions. The use of the GGDML language extensions allowed to productively write scientific codes in a single source code, and the use of the translation technique provided performance-portability to get nearly optimal code on the two platforms.

8.3.6. Scalability

One point we investigate in this work is the transfer of the semantics to drive parallelization of code over multiple nodes. Scalability is a key point to support modeling for the recent supercomputers and exascale computing era. In this section we present experiments, in which we use high-level code using GGDML, and translate it using different configurations to support multiple nodes. We executed experiments both on machines with multi-core processors and on machines equipped with GPUs. First, we discuss early experiments, in which we tested scaling the Laplacian on a few GPUs on the PSG cluster, and on Mistral. Then, we discuss later experiments, in which we executed the full shallow water equation solver application on Piz Daint, where we could have access to many more GPUs, and on Mistral.

8.3.6.1. Laplacian Operator Experiments

To evaluate the scalability of the Laplacian operator on multiple nodes with GPUs, we translated the code for GPU-accelerated machines using MPI (besides OpenACC) and we executed it on 1-4 nodes. Figure 8.11 shows the performance of the application when it is run on the P100 nodes. The figure shows the performance achieved in both cases when measuring the strong and the weak scalability. The performance has been measured to find the maximum achievable performance when no halo exchange is performed, and to find the performance of an optimized code with halo exchange. The performance gap reflects the cost of the data movement from and into the GPU's memory as limited by the PCIe3 bus and along the network using Infiniband. This gap differs according to the data placement of the elements that need to be communicated to other nodes. Thus, putting the elements in an order in which halo elements are closer to each other in memory reduces the time for the data exchange from and into the GPU's device memory. The scalability (both strong and weak) measurements are described in Table 8.10. The table shows how the performance improves with the nodes. Also, it shows the ratio that is achieved when running the code with respect to the maximum performance gain (that is achieved without halo exchange). The computing time spent each time step for the whole grid (1024x1024x64 elements) is measured to be 8.34ms.The communication times spent during each time step are shown in Table 8.11.

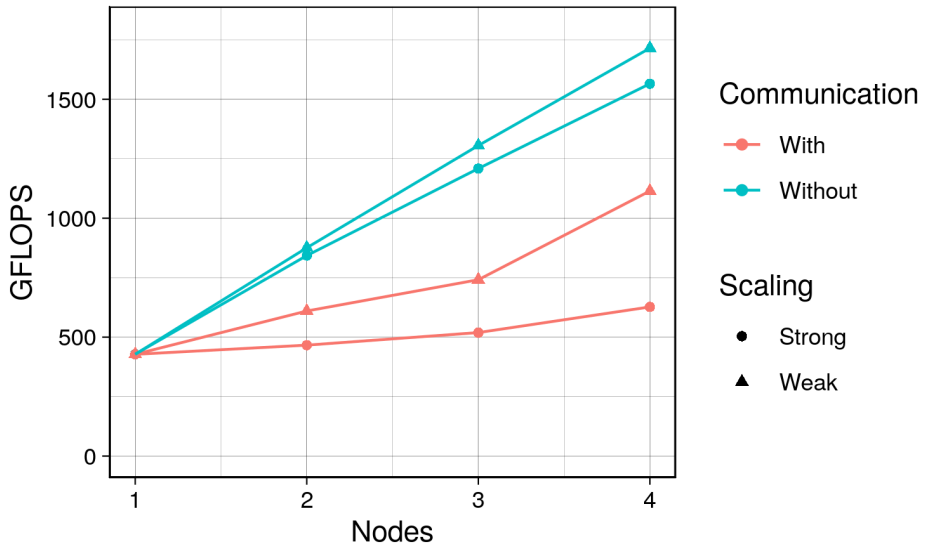


Figure 8.11.: Performance Scalability on nodes with P100 GPUs

The communication times between different numbers of MPI processes running in different mappings over nodes are recorded, Table 8.11 shows the measured values on the PSG cluster. We executed the application in 2,4,8,16,32,64, and 128 processes over

The contents here under the title "Laplacian Operator Experiments" are published in [JK18]

Number of nodes	Strong scaling			Weak scaling		
	Without communication	With communication	Ratio	Without communication	With communication	Ratio
2	1.97	1.09	55%	2.07	1.43	70%
3	2.82	1.21	43%	3.05	1.73	58%
4	3.65	1.47	40%	4.01	2.60	65%

Table 8.10.: Performance Scalability on nodes with P100 GPUs

1,2, and 4 nodes. For multiple nodes, we mapped the MPI processes to the nodes in three ways: cyclic, blocked with balanced numbers of processes on each node, and in blocks where the processes subsequently fill the nodes. The time was measured over 1000 time steps in each case. The measured times show that optimizing the communication time is essential to achieve better performance, and that optimizing the data movement from/into the GPU's memory is essential to minimize the halo exchange time.

# processes	1	2 nodes			4 nodes		
		Cyclic	Block (balanced)	Block (unbalanced)	Cyclic	Block (balanced)	Block (unbalanced)
2	1.21	1.18	1.11	1.21			
4	1.03	0.93	0.86	1.18	0.88	0.90	1.24
8	1.00	0.84	0.77	1.52	0.77	0.75	1.58
16	0.80	0.83	0.56	1.59	0.69	0.54	1.60
32	1.29	0.77	0.64	1.26	0.69	0.51	1.24
64		1.33	0.82	0.78	0.84	0.52	0.77
128					1.48	1.32	1.23

Table 8.11.: Communication time per time step (in ms) on PSG cluster

Broadwell experiments: To evaluate the scalability of the generated code with multiple MPI processes on CPU nodes, we executed it on 1,4,8,12,16,20,24,28,32,36,40, and 48 nodes. The performance is shown in Figure 8.12.

Both the strong and the weak scalability efficiency are calculated according to the equations

$$Efficiency_{strong} = T_1/(N \cdot T_N) \cdot 100\% \tag{8.4}$$

$$Efficiency_{weak} = T_1/T_N \cdot 100\% \tag{8.5}$$

where N is the number of processes, T_1 is the execution time on one process, and T_N is the execution time on N processes. The results are shown in Figure 8.13. The efficiency is about 100% up to 48 MPI processes for the weak scaling measurements. The Strong scaling measurements decrease from 100% at one process to about 70% at 48 processes in a linear trend.

The performance of the generated code that uses OpenMP with the MPI is also evaluated. We generated code with OpenMP and MPI and executed it with multiple

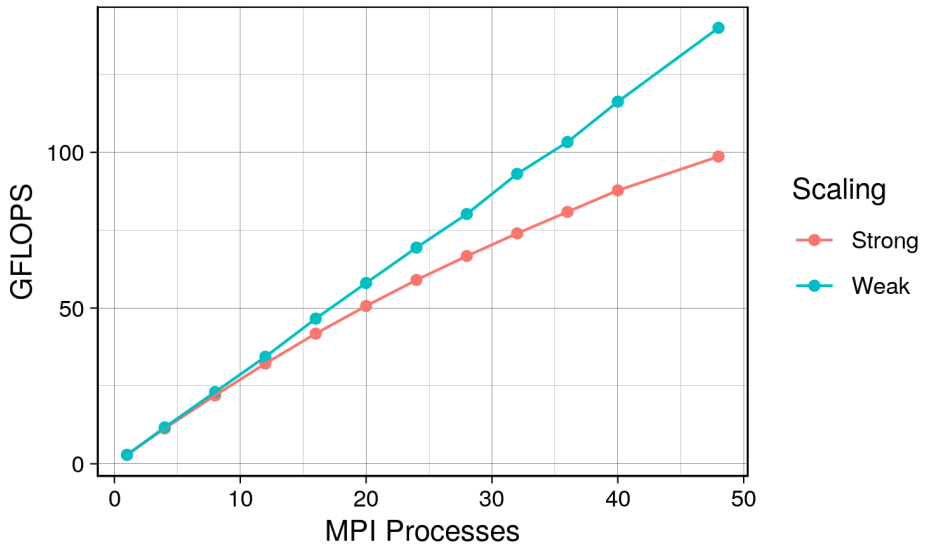
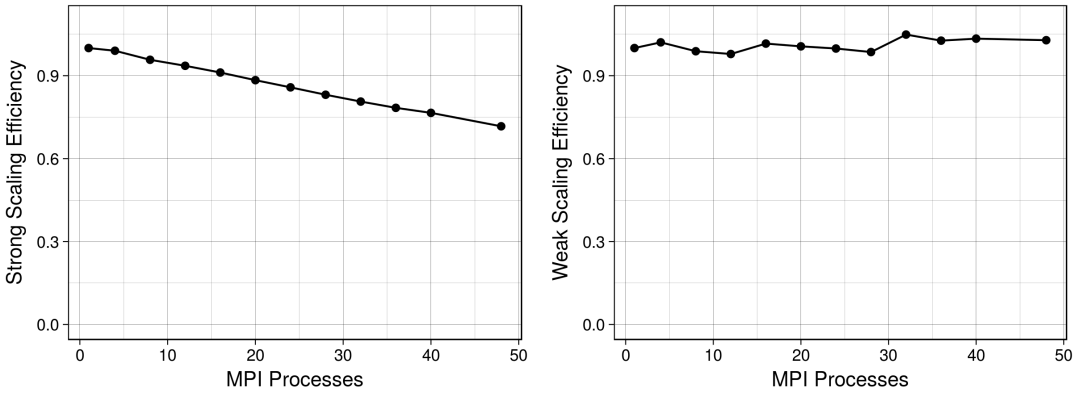


Figure 8.12.: MPI process scalability of Laplacian on Mistral



(a) Strong Scaling Efficiency (b) Weak Scaling Efficiency

Figure 8.13.: Scaling Efficiency

numbers of nodes, using different numbers of cores on each node. We executed the code on 1,4,8,12,16,20,24,28,32,36,40 and 48 nodes and 1,2,4,8,16,32, and 36 cores per node. The measurements are shown in Figure 8.14.

8.3.6.2. Shallow Water Equation Solver Experiments

To evaluate the scalability of the shallow water equation solver application, we started with the GGDML + C code of the solver. We started with configuration files to

The contents here under the title "Shallow Water Equation Solver Experiments" (except GASPI experiments, which were done later) are published in [JK19b].

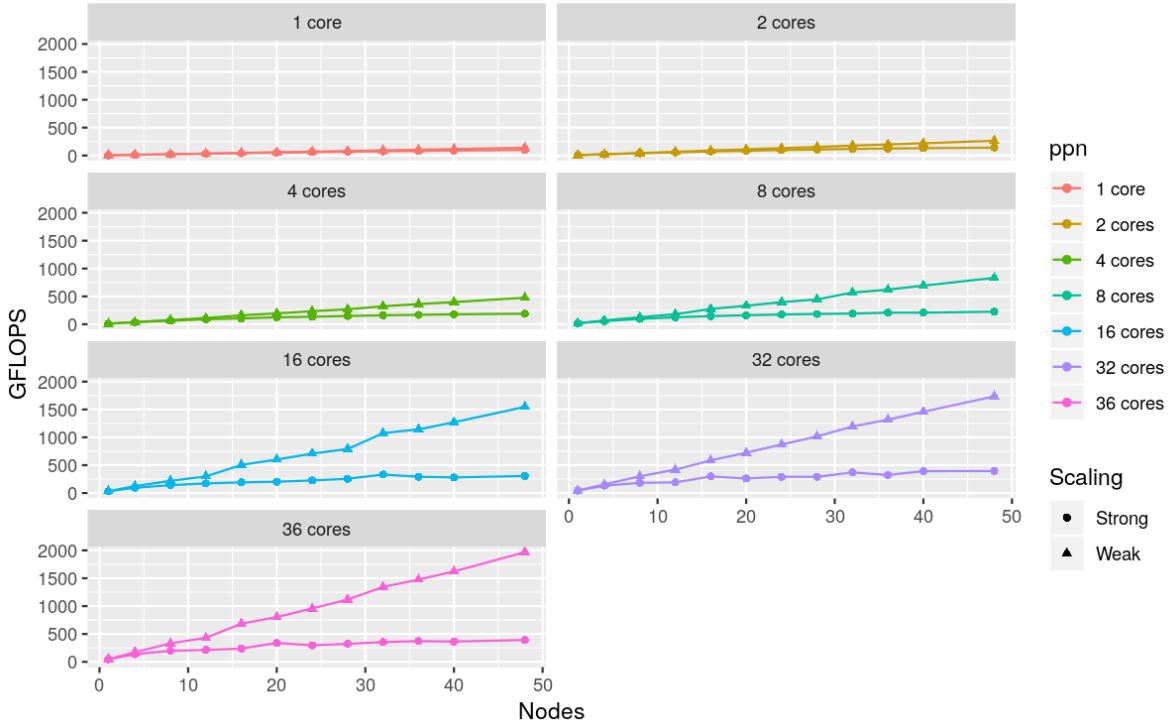


Figure 8.14.: MPI+OpenMP scalability

guide the code translation into C with OpenMP for multi-core processors, and C with OpenACC for GPUs. Optimization procedures were applied during the translation process, e.g., blocking, to exploit the features, e.g., caching, of the processing units (per node). Parallelization on the node resources, i.e., the cores of the multi-core processors and the threads and SMs on GPUs, was applied using OpenMP and OpenACC.

Testing environments: We used the shallow water equation solver code to run the main scalability experiments. The multi-core processor experiments are executed on the Mistral machine (with Broadwell processors) using the Intel (18.0.2) C compiler and the IntelMPI (2018.1.163) library. The GPU experiments are run on the nodes on the machine 'Piz Daint' at the Swiss National Supercomputing Center (CSCS). The GPUs are Tesla P100 with 16 GB memory and PCIe interconnect to the host. We used the PGI (17.7.0) C compiler and the MPICH (7.6.0) library.

Domain decomposition: To distribute the work between the running resources, both on multi-core processors and on GPUs, the problem domain is decomposed into local domains that reside on each node. Contiguous lines of the grid are given to each local domain. While the domain decomposition strategy maximizes load balance between nodes, other on-node considerations are taken into account. Data reuse, and distribution over cores/threads were maximized with blocking and on-node parallelization.

Mistral experiments: Translating the source code for the Broadwell and running it on a single node shows near optimal use of the processor. The application (and the kernels) runs with around 80% of the processor's memory bandwidth (measurement with the 'stream_sp_mem_avx' benchmark from the 'Likwid' tools measured 67 GBytes/s). This code uses caches optimally, where minimal data movement between memory and processor is needed. Minimizing the movement of the data in a memory-bound code means the code runs with about an optimal performance.

Next we modified the configuration files to count for parallelization over multiple nodes. We used the defined access operators, as discussed in Chapter 7, to generate communication code and scale the same source code on multiple nodes. Using configuration files that count for multiple nodes, we generated the necessary MPI code to handle halo exchange. We executed the generated code on the target machines. In the experiments we use multiples of ten, up to hundred nodes (1, 10, 20 ... 100). The results are shown in Figure 8.15.

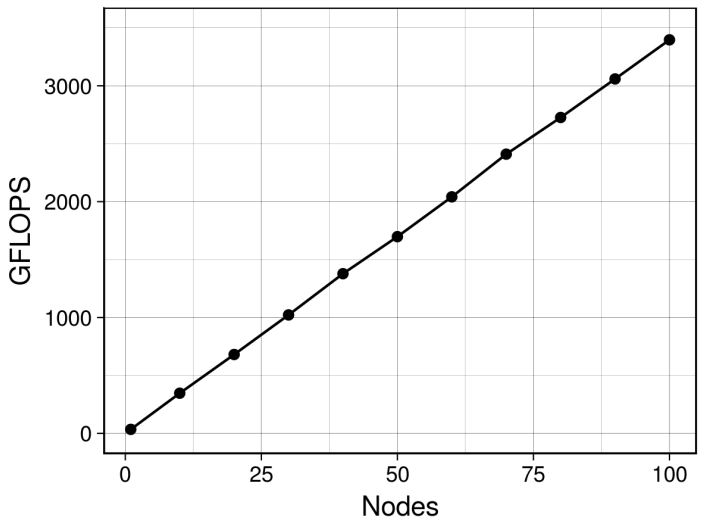


Figure 8.15.: Scaling SWE solver on multiple Broadwell nodes

Piz Daint experiments: Translating the source code for the P100 GPU and running it on a single node shows near optimal use of the GPU. The application (and the kernels) runs with around 80% of the GPU's memory bandwidth (measurement with a CUDA STREAM benchmark yielded about 498 GB/s). This code uses caches and warps nearly optimally, where minimal data movement between the device memory and the executing GPU threads is done. This means the code runs with about an optimal performance.

Using the access operators again we generated the application code that includes the necessary communication code, which allowed to run the same source code on multiple nodes with GPUs. We generated the necessary MPI code to handle halo exchange, besides to the OpenACC code. The application scaled to multiple nodes with GPUs. Again we use multiples of ten, up to hundred nodes (1, 10, 20 ... 100). The results are

shown in Figure 8.16.

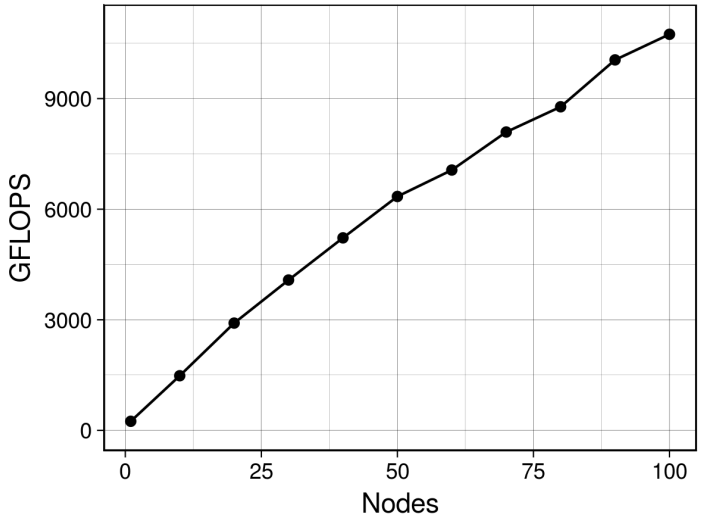


Figure 8.16.: Scaling SWE solver on multiple nodes with P100 GPUs

GASPI initial experiments: Our translation technique is flexible, allowing to replace the library that is used for multi-node parallelization and communication without the need to modify the translation tool. This is enabled by using the desired library to write the necessary sections of configuration files. To test this flexibility, we executed experiments with GASPI library as an alternative to MPI library. We could install the library on the Mistral machine and measure performance. We prepared configurations to use GASPI and generated code for the target machine. The measured performance of running code on multiple nodes (10, 20, 30 ... 100) is shown in Figure 8.17.

The results show scaling the same source code on multiple nodes successfully. Replacing the MPI with GASPI was possible with changing some contents of related configuration files. Performance could be better improved with optimizing the use of the library features. However, for time reasons, we restricted our experiments to prove the scalability of the same code using an alternative library, which proved to work successfully.

8.4. Performance Portability

An important aspect of the solution that we developed is performance portability. The scientists develop a single source code. The source does not need to be modified when targeting a new machine. To support performance portability, we use different configuration files to target different hardware configurations. Our experiments were done on multi-core processors, GPUs, and vector engines. Using the same source code and getting high performance ratio to the expected performance on each architecture shows that the technique is successful to provide performance portability.

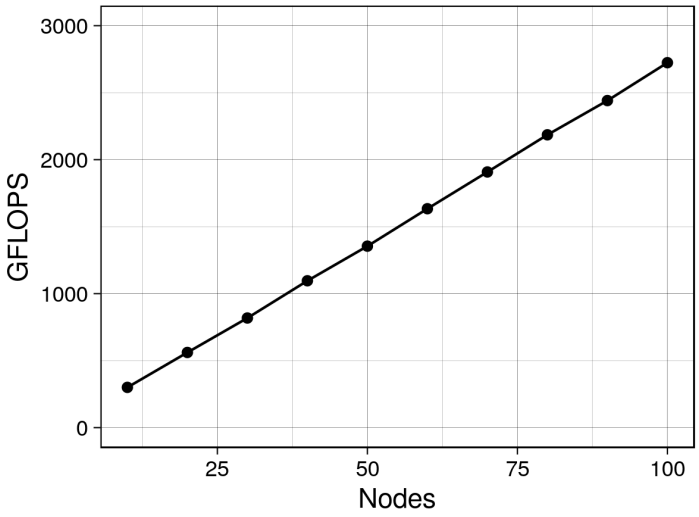


Figure 8.17.: SWE solver scalability with GASPI library

There have been efforts to define and measure performance portability as it is becoming an important aspect of software development in the field of HPC. Among those efforts we discuss two metrics, and evaluate the use of our techniques according to their suggested formulae.

Pennycook et al. metric A metric to measure the performance portability of an application solving a problem on a set of platforms was presented in [PSL16]. The authors formulated their metric to measure the performance portability Φ of an application a solving a problem p on a set of platforms H as:

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{8.6}$$

where $e_i(a,p)$ is the performance efficiency of running the application a to solve problem p on platform i. Two performance efficiency metrics are considered by Pennycook et al. in [PSL16]:

- Architectural efficiency: where efficiency is measured to theoretical maximum performance by architecture, which reflects efficiency of application to use the underlying hardware.
- Application efficiency: where efficiency is measured in comparison to best measured performance, which allows to compare code to best performance achieved by applications.

To evaluate the use of our techniques with this performance portability metric, we measured performance efficiency using architectural efficiency. The lack of information

on running same problem with other applications on same hardware architectures limits the evaluation in terms of application efficiency. With architectural efficiency, the performance portability metric indicates the efficiency of the application to use the underlying hardware while being executed on the targeted machines.

To evaluate the maximum achievable performance of the underlying hardware, we measure achieved memory throughput of the code vs. the memory bandwidth of the architecture. We use memory bandwidth efficiency because the code is memory bound. The measurements are shown in Table 8.12. Based on the efficiencies shown

Architecture	Measured throughput (GB/s)	Memory bandwidth (GB/s)	Efficiency
Broadwell	60	77	0.784
P100	389	499	0.780
SX-Aurora VE	911	1,200	0.759

Table 8.12.: Architectural efficiency in terms of memory bandwidth

in Table 8.12 and the formulae of the performance portability (Equation (8.6)), the metric for the code we developed is 77.4%. This measurement is done for the set of architectures $H = \{$Broadwell, P100, SX-Aurora Tsubasa$\}$. The calculated performance portability metric indicates ***an average efficiency of using the different hardware architectures when running the same application code***.

Zhu et al. metric Another metric was suggested by Zhu et al. in [ZNG07] to measure performance portability when parallelizing code over multiple nodes. This metric indicates the performance portability of an application running on a number of nodes when executed on a machine in comparison to running on same number of nodes on another machine. The formula to express the performance portability P_n of an application running on n nodes on a target machine T in comparison to a base machine B is:

$$P_n = \frac{S_n^T}{S_n^B} \times 100\% \tag{8.7}$$

where S_n^T and S_n^B are the absolute speedups on both the target and base machines respectively.

To evaluate the use of our techniques using this metric, we measured the speedup on different number of nodes on two machines: one with Broadwell multi-core processors, and another with GPUs. The measured runtime of the sequential execution of the code on a broadwell processor is 1288 seconds. The measured runtime and the speedup on different numbers of nodes on both machines are shown in Table 8.13. Table 8.13 shows also the calculated performance portability metric (according to Equation (8.7)) on different numbers of nodes. The performance portability metric shows the percentage of performance achieved on the machine with P100 GPUs with respect to the machine with

Nodes	Time (s)		Speedup		Performance portability (%)
	Base system	Target system	Base system	Target system	
10	174	40	7	32	430
20	177	41	7	31	429
30	174	44	7	29	394
40	175	46	7	28	381
50	177	47	7	27	375
60	177	51	7	25	347
70	177	52	7	25	342
80	176	55	7	24	323
90	177	54	7	24	329
100	177	56	7	23	317

Table 8.13.: Performance portability according to Zhu et al. metric

Broadwell processors. The column in the table shown this comparison for 10,20,30...100 nodes.

Chapter summary

In this chapter, we presented experiments work and provided theoretical analysis to validate our techniques. We discussed the impact of using our language extensions on the quality of code, and hence on development costs and productivity of scientists and maintainability of code. We then discussed experiments to evaluate performance of code developed and processed with our language extensions and techniques. We concluded the chapter with a discussion of the performance portability of code developed using our techniques.

In the next chapter, we conclude this text with conclusions from a scientific perspective and from a software engineering perspective, and with an outlook on future work to continue the achievements of this work.

9. Conclusion

In this chapter, we conclude the thesis. We summarize the activities in our work in Section 9.1 including launching the work from the research questions and objectives, to the review of literature and gap analysis, to design and research methodology, to the application of this methodology, where language extensions and high-level codes were developed, translation process was designed and developed, and approach was validated. In this section, we provide a summary on the impact of using our techniques based on measured and estimated impact on code quality and performance and performance portability. Finally, we provide in Section 9.2 an outlook on future work to build on what has been achieved so far.

9.1. Summary

In this thesis, we study the use of domain-specific language extensions to transfer the necessary semantics from source code to processing tools to drive the optimization process and scaling code to multiple nodes. Different domain-specific languages have been developed to support development of stencil computations, however, in this work we investigate application-adaptable language extensions. The extensions that we study are defined by users. This feature allows to convey precise semantics that serve the application based on its special needs.

9.1.1. Research Questions and Work Objectives

Domain-specific solutions have been a topic for research so far. However, the use of flexible language constructs to increase the semantical capabilities of modeling languages represented a viable point to study for us. The huge costs of running models and maintaining them (and developing them) make scientific contributions to optimize modeling of high value. We expected the impact of fitting a modeling language to subject applications on the semantical capabilities to improve optimization possibilities and code quality.

Our first step was to list and formalize a set of research questions and objectives. We designated the following questions as targets to seek answers for in this research:

- How could user-defined application-adaptable language extensions (when mixed with a general-purpose language) convey the necessary information to drive stencil code optimization process in a user-controlled code processing procedure?
- How could the semantics drive the scaling of code over multiple nodes?

- How does that affect the quality of the code?

The questions serve our objectives regarding code quality, performance and performance portability.

9.1.2. Design and Methodology

To seek answers for the questions, the next step was to check out the research that has been done in the field to understand what has been done so far. We reviewed and analyzed related efforts to understand the gap between existing solutions and the objectives that we target. To address the gap between existing solutions and research objectives, we developed our approach that depends on language extensibility and configurability.

Language extensibility: Our approach offers an integrated solution that supports model development process, with new software engineering concepts. With this solution, analysis of an application's requirements leads to identify which grids to use and what stencils comprise the application. The identified grids and the stencils are used to define a set of language extensions. Using spatial relationships between points which form a stencil on a specific kind of grid does not carry any machine-level semantics or computer scientific concepts. It rather allows to apply scientific concepts to code development. Therefore, an application's source code, according to our approach, is written using high-level semantics.

Language configurability: To consume the semantical value of the language extensions, that value is passed from the source code to a code processing tool, where optimization procedures are applied. Since a set of the language extensions are defined per application, there should be a way to tell the tools how to deal with those extensions. Such information are passed through configuration files. A configuration file allows to tell the tools how to use that extracted information from source code to apply different optimization procedures.

Methodology: We planned our research methodology based on the language extensibility and configurability concepts. The plan included

- developing a set of language extensions,
- developing code to study and evaluate our approach and demonstrate the use of those extensions,
- designing the translation process, including matching source code and configuration file inputs to optimization procedures, and developing the ways to apply the optimization and scaling of code,
- and finally, validating the approach.

Surely there have been other alternatives which we dismissed for their shortcomings. Example alternatives are using language features, e.g., C++ templates, or using compiler frameworks. However, this would have restricted our research as a result of

- limitations to language syntax
- sticking to a specific language
- dependence on external solutions (frameworks)

In our approach, we decided rather to extend a modeling language, where new rules could be added to existing grammars. This allows applicability to the different languages in general.

We considered and decided this research methodology because it empirically leads to evaluate and prove or disprove the feasibility of the approach to improve code quality and performance portability. Besides, language extensions and tools are actually developed within the research work, so if the approach proves to be feasible, the language extensions and the tools would have already been developed, and may be usable for modeling and further research.

Developing the language extensions and the high-level codes: According to our research method, we developed a set of language extensions. We got in contact with experienced scientists working on different models and got a set of performance-sensitive codes. We suggested language extensions and used them to rewrite the code samples and discussed the resulting code iteratively to reach an acceptable set of language extensions.

Using the developed language extensions, we developed two applications:

- an application including a set of frequently-used operators, e.g., divergence,
- and a full application to solve a specific problem, which is the shallow water equations.

Those codes demonstrate coding with the language extensions, allowing people in the domain to understand the concepts. But also, they serve testing and evaluation purposes.

Developing the translation process: Throughout the development of the code translation process, we defined related inputs from source code in terms of language extensions along with corresponding information from configuration files, and mapped such combinations to different optimization aspects. We then developed techniques to implement the different transformations, which match inputs from source code with configuration information to yield optimized code.

Through the developed translation process, we studied the applicability of multiple optimization techniques within the translation tools to achieve performance close to the theoretical performance that should be expected based on the nature of the code. We listed a set of optimization techniques that lead to optimized memory access including cache blocking, loop fusions, optimal data layout in memory. Besides to memory access,

we targeted parallel processing capabilities both at the coarse level, e.g, by multiple/many cores and streaming multiprocessors, and at the finer level, e.g, by vector units.

In addition to node-level parallelism, we considered code scalability over multiple nodes, which represents an important challenge to consider on modern supercomputer machines and in preparation for exascale computing era. An important feature of our approach is that high-level code is developed independently of targeting a single node or multiple nodes. Application scalability through the techniques that we developed is driven by the semantics of the language extensions, which reflect the spatial relationships within stencils. No code should be written in source code to handle domain decomposition and communication of halo data. All such details are generated by the tools based on the high-level field access indices.

9.1.3. Objectives Achievement and Approach Validity

Evaluating our techniques experimentally shows the success to achieve the objectives that we designated to support model development through improved code quality and performance portability. Results show success to extract the necessary semantics from the source code and the use of the configuration information to apply code transformation to exploit underlying hardware.

Code quality and development costs: The complexities that arise from code optimization in conventional modeling, especially for multiple architectures, represent an obstacle facing code maintainability. Code redundancies to target different architectural features complicate code fixes and further development. The language extensions that we suggested allow to avoid the need for code redundancies.

The reduced code complexity and the lack of optimization work at the code development level improve the productivity of scientists while developing model codes. Formulating scientific problems in terms of scientific concepts eases the model development from scientists perspective, and allows focus on the scientific problem in hand.

Coding simplification leads to reduce code size and development costs considerably. Our estimations show that code size is reduced to less than one third. An estimation of development costs of a model with even less stencils indicates a saving of more than half the model's development costs.

Performance impact: We developed techniques to support developing stencil computations, which are among the most performance-demanding applications. Thus, performance of applications that are developed according to our model development concepts is an essential aspect to consider when evaluating those techniques. We carried out multiple experiments to evaluate different optimization aspects. Experiments spanned evaluating a single mathematical operator, and a full application comprising multiple kernels.

Experimental results show that our techniques to apply cache blocking transformations allowed to optimize the use of the caches on the different architectures. We could retain performance under different scenarios thanks to exploiting the caching technologies.

In addition to cache blocking, we improved memory access with techniques to control the data layout in memory. We developed a genuine technique that allows to provide formulae to specify the location of a data element in memory. This data location flexibility, in addition to other techniques to control the temporal aspect of data access, e.g, loop ordering in loop nests for which we developed a transformation procedure, shows to maximize the data reuse while being in cache memories.

To exploit optimization opportunities across stencils and kernels, we developed inter-kernel optimization procedures. To evaluate our inter-kernel optimizer we carried out detailed experiments to check the impact under different circumstances. Generally, we found that this application level optimization improved the performance between one third to a half of the performance of the application with optimized stencils (application-level optimized code–with inter-kernel optimization achieved 130% to 150% of stencil-level optimization).

Other developed techniques which allowed to apply different optimization aspects, e.g, vectorization and core/thread parallelization, allowed to increase performance. Vectorization experiments show that we could generate vectorized code on multi-core processors and vector engines. We could also exploit the warps on GPUs to maximize the fine-level parallelism. Workloads were balanced among cores on the tested architectures through the coarse-level parallelization, i.e, cores and streaming multiprocessors.

Scalability is an important part of our work, as a result of the evolution of HPC towards multiple-node solutions. Our techniques allow to transform the same high-level code to exploit multiple node configurations. To evaluate our across-node parallelization, we executed multiple experiments. Our techniques prove to generate scaling code, where we ran experiments up to one hundred nodes. Scalability experiments show success on both host multi-core processors, and on GPU-equipped machines. We also evaluated the flexibility of our techniques to replace underlying parallelization and communication libraries, through using GASPI as an alternative to MPI. We could successfully generate code with only changing some sections in configuration files, without introducing any changes to the translation tools. Again, the results show the success to scale code using GASPI, which we tested up to one hundred nodes. In both MPI and GASPI experiments we used also OpenMP to exploit core-level parallelism.

Performance portability: Performance portability is a nightmare for model developers as a result of the diversity of architectures arising in the field of HPC. In our techniques we could bypass this shortcoming, by allowing scientists to write exactly one version of code that is unaware of underlying hardware, and hence any optimization considerations.

We considered three different architectures in our experiments: multi-core processors, GPUs, and vector engines. We have used the same source codes across all our experiments without modifying any single line of code or including any architecture-specific line of code. Experimentally, we could achieve a near optimal performance, where we could run with a high memory throughput (80% of maximum memory throughput on the different architectures). We could judge our techniques to support performance portability after proving to have achieved that high percentage of maximum achievable performance

across all tested architectures.

Research method viability: We developed a set of language extensions that can be adapted to the requirements of a subject application. Besides, we formulated mappings that relate language extensions from source code and configuration information from configuration files to optimization aspects. Then we moved from the 'what' level to the 'how' level, where we described techniques that we develop to use the mappings to apply code transformations that lead to optimized and scalable code. Through this research path, we could answer the questions as we defined how to convey the necessary semantics from the application code to transform the code into optimized scalable code.

In addition to the semantics transfer, we show also the impact of the use of this kind of language extensions on the code quality. As we discussed, our language extensions reduced code size and development costs, and allowed to eliminate the need for code redundancies, leading to improve code maintainability significantly.

It is clear that application-adaptable language extensions could convey ***more precise semantics*** *from application code to drive optimization procedures. The application-specific spatial relationships could allow to specify stencils in source code, and allow to* ***identify the necessary communication****, which is a key consideration for scalability across nodes. Those relationships could allow memory-oblivious data access, where source code is unaware of the actual location of data elements. This lead to a* ***flexible technique that allows users to control the location of the data elements*** *(the memory layout). In addition to application-specific spatial relationships, the other application-adaptable extensions allowed to control the optimization according to application features, e.g, more control on how the operations on the grids that the application uses are parallelized, blocked, and traversed. This allowed to* ***maximize the performance impact per application****.*

9.2. Future Work

During the research that we have been doing, we found opportunities to further push the science and the software engineering of model development. In this section we mention some of those opportunities.

The genuine flexible techniques that allows unprecedented control of data location in memory, which we developed in this work, allowed to explore performance of different alternative data layouts with a small effort, where source code does not need to be modified. Among those alternatives, we could imitate data structured in arrays of structures (AoS), with minor changes in configuration files. Thus, we could evaluate performance impact of different layouts with minor costs. Still we see opportunities to use this advantage to explore optimal data layouts for coupled earth system models, where multiple components access data in different patterns, e.g, more frequent column operations in a component and more frequent horizontal stencils in another component. Exploring optimal data layouts is very expensive if different code versions need to

be written manually with different memory layouts. However, using our data layout transformation techniques to explore alternatives saves time and efforts making such research feasible.

Another point that is worth being studied is the separation of data access from data management. Using our MODA techniques we enabled code to be memory oblivious, where a data element is accessed using simple array notation. Source code is unaware of the real location in memory or whether the data is stored in host memory or a GPU's device memory, or whether that data is on the node at all or on another node. Investigating the generalization of this technique is promising for exascale computing, where data tracking and communication across nodes and synchronization are essential to distribute workloads over multiple nodes. With separating data access from data management, referring to a field with $field[index]$ is enough in source code, while tools should be able to track communication and synchronization to guarantee computation consistency. Our experiments proved success of the techniques under domain-specific assumptions, however, how wider can that be generalized is worth being investigated.

Bibliography

[ABG77] Thomas J Aird, Edward L Battiste, and Walton C Gregory. Portability of mathematical software coded in fortran. *ACM Transactions on Mathematical Software (TOMS)*, 3(2):113–127, 1977.

[AL77] Akio Arakawa and Vivian R Lamb. Computational design of the basic dynamical processes of the ucla general circulation model. *General circulation models of the atmosphere*, 17(Supplement C):173–265, 1977.

[ALØ+14] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9, 2014.

[AQQ13] Mohammed Mugahed Al Qmase and M Rizwan Jameel Qureshi. Evaluation of the cost estimation models: case study of task manager application. *International Journal of Modern Education and Computer Science*, 5(8):1, 2013.

[AvE01] Robert A van Engelen. Atmol: A domain-specific language for atmospheric modeling. *CIT. Journal of computing and information technology*, 9(4):289–303, 2001.

[B+81] Boehm Barry et al. Software engineering economics. *New York*, 197, 1981.

[BCF+22] Mauro Bianco, Paolo Crosetto, Oliver Fuhrer, Stefan Moosbrugger, Carlos Osuna, Hannes Vogt, and Thomas Schulthess. CSCS GridTools. `https://pasc17.pasc-conference.org/fileadmin/user_upload/pasc17/program/post144s2.pdf`, 2017 (Accessed: 2017-12-22).

[BFRV12] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[BLT09] Peter Benner, Ren-Cang Li, and Ninoslav Truhar. On the adi method for sylvester equations. *Journal of Computational and Applied Mathematics*, 233(4):1035–1045, 2009.

[BP07] Tobias Brandvik and Graham Pullan. Acceleration of a two-dimensional euler flow solver using commodity graphics hardware. *Proceedings of the*

Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 221(12):1745–1748, 2007.

[BRP07] Richard F Barrett, Philip C Roth, and Stephen W Poole. Finite difference stencils implemented using chapel. *Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122*, 2007.

[But68] Arthur R Butz. Space filling curves and mathematical programming. *Information and Control*, 12(4):314–330, 1968.

[But97] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[C+43] Richard Courant et al. *Variational methods for the solution of problems of equilibrium and vibrations*. Verlag nicht ermittelbar, 1943.

[CBPP02] B Chapman, Fréderic Bregier, Amit Patil, and Achal Prabhakar. Achieving performance under openmp on ccnuma and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, 2002.

[CDO+95] J Choiy, J Dongarraz, S Ostrouchovx, A Petitetx, D Walker, and RC Whaleyx. Lapack working note 100 a proposal for a set of parallel basic linear algebra subprograms. *University of Tennessee, Knoxville*, 1995.

[CFF+18] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. The claw dsl: Abstractions for performance portable weather and climate models. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '18, pages 2:1–2:10, New York, NY, USA, 2018. ACM.

[Che17] Xiaohui Chen. Metafork: A compilation framework for concurrency models targeting hardware accelerators. 2017.

[CHH+90] NP Chrisochoides, CE Houstis, Elias N Houstis, PN Papachiou, and SK Kortesis. Domain decomposer: a software tool for mapping pde computations to parallel architectures. 1990.

[CK89] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *PPSC*, pages 400–405, 1989.

[Cla] CSCS Claw. `http://www.xcalablemp.org/download/workshop/4th/Valentin.pdf`. Accessed: 2017-12-22.

[CMF95] Nikos Chrisochoides, Nashat Mansour, and Geoffrey Fox. A comparison of data mapping algorithms for parallel iterative pde solvers. *Concurrency: Practice Experience*, 1995.

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[CTZ00] Gianni Conte, Stefano Tommesani, and Francesco Zanichelli. The long and winding road to high-performance image processing with mmx/sse. In *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 302–310. IEEE, 2000.

[CUD20] CUDA. `https://developer.nvidia.com/cuda-zone`, 2019 (Accessed: 2019-08-20).

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[DBD+17] Willem Deconinck, Peter Bauer, Michail Diamantakis, Mats Hamrud, Christian Kühnlein, Pedro Maciel, Gianmarco Mengaldo, Tiago Quintino, Baudouin Raoult, Piotr K. Smolarkiewicz, and Nils P. Wedi. Atlas : A library for numerical weather prediction and climate modelling. *Computer Physics Communications*, 220:188 – 204, 2017.

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12), 2009.

[DDCHD90] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[DDE+05] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R Clint Whaley, and Katherine Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[DDT+15] Thomas Dubos, Sarvesh Dubey, Marine Tort, Rashmi Mittal, Yann Meurdesoif, and Frédéric Hourdin. Dynamico, an icosahedral hydrostatic dynamical core designed for consistency and versatility. *Geoscientific Model Development Discussions*, 8(2), 2015.

[Dec19] Willem Deconinck. Development of atlas, a flexible data structure framework. *arXiv preprint arXiv:1908.06091*, 2019.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik

Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[DMH+17] R. Döscher, H. Martins, C. Hewitt, F. Whiffin, van den Hurk, and B. (Eds.). *European Earth System Modelling for Climate Services*, volume 1 of *Climateurope Publication Series*. 2017.

[DRGG+95] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, B Marsolf, and D Padua. Falcon: A matlab interactive restructuring compiler. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288. Springer, 1995.

[ESP+12] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming*, 20(2):89–114, 2012.

[FD17] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[FGH+13] R Ford, MJ Glover, DA Ham, CM Maynard, SM Pickles, G Riley, and N Wood. Gung ho: A code design for weather and climate prediction on exascale machines. In *Proceedings of the Exascale Applications and Software Conference*, 2013.

[Fla11] Gregory M Flato. Earth system models: an overview. *Wiley Interdisciplinary Reviews: Climate Change*, 2(6):783–800, 2011.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[GFO+14] Tobias Gysi, Oliver Fuhrer, Carlos Osuna, Benjamin Cumming, and Thomas Schulthess. Stella: A domain-specific embedded language for stencil codes on structured grids. In *EGU General Assembly Conference Abstracts*, volume 16, 2014.

[GK06] Pawel Gepner and Michal Filip Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13. IEEE, 2006.

[GS87] Branko Grünbaum and Geoffrey Colin Shephard. *Tilings and patterns*. Freeman, 1987.

[HBK06] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. *white paper, Intel*, 2006.

[HKL73] Richard J Hanson, Fred T Krogh, and CL Lawson. A proposal for standard linear algebra subprograms. 1973.

[HKM08] Wen-mei Hwu, Kurt Keutzer, and Timothy G Mattson. The concurrency challenge. *IEEE Design & Test of Computers*, 25(4):312–320, 2008.

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[HRC+90] Elias N Houstis, John R Rice, NP Chrisochoides, HC Karathanasis, PN Papachiou, MK Samartzis, EA Vavalis, Ko Yang Wang, and Sanjiva Weerawarana. //ellpack: A numerical simulation programming environment for parallel mimd machines. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 96–107. ACM, 1990.

[Hre41] Alexander Hrennikoff. Solution of problems of elasticity by the framework method. *J. appl. Mech.*, 1941.

[HSP+11] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *International Conference on Compiler Construction*, pages 225–245. Springer, 2011.

[HSV+98] EN Houstis, A Sameh, E Vavalis, E Gallopoulos, and TS Papatheodorou. Parallel numerical algorithms and software. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1998.

[Hwu11] Wen-Mei W Hwu. *GPU computing gems emerald edition*. Elsevier, 2011.

[Int11] R Intel. Intel math kernel library reference manual. Technical report, Tech. Rep. 630813-051US, 2012.[Online]. Available: `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf`, 2011.

[Jac45] Carl GJ Jacobi. Ueber eine neue auflösungsart der bei der methode der kleinsten quadrate vorkommenden lineären gleichungen. *Astronomische Nachrichten*, 22(20):297–306, 1845.

[JK18] Nabeeh Jumah and Julian Kunkel. Performance Portability of Earth System Models with User-Controlled GGDML code Translation. In *High Performance Computing*, number 11203 in Lecture Notes in Computer Science. Springer, 2018.

[JK19a] Nabeeh Jumah and Julian Kunkel. Automatic vectorization of stencil codes with the ggdml language extensions. In *Proceedings of the 5th Workshop*

on Programming Models for SIMD/Vector Processing, WPMVP'19, pages 2:1–2:7, New York, NY, USA, 2019. ACM.

[JK19b] Nabeeh Jumah and Julian Kunkel. Scalable parallelization of stencils using moda. In Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, pages 142–154, Cham, 2019. Springer International Publishing.

[JK20] Nabeeh Jumah and Julian Kunkel. Optimizing Memory Bandwidth Efficiency with User-Preferred Kernel Merge. In *Euro-Par 2019: Parallel Processing Workshops*, volume 11997 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2020.

[JKM+17] Nabeeh Jumah, Julian M Kunkel, Michel Müller, Hisashi Yashiro, Thomas Dubos, and John Thuburn. D1.1 model-specific dialect formulations. 2017.

[JKZ+17] Nabeeh Jumah, Julian M Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Thomas Meurdesoif. Ggdml: icosahedral models language extensions. *Journal of Computer Science Technology Updates*, 4(1):1–10, 2017.

[JR13] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[KD16] BARDSIRI VAHID KHATIBI and Mahboubeh Dorosti. An improved cocomo based model to estimate the effort of software projects. 2016.

[KHO+05] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*, pages 36–43. ACM, 2005.

[KKKL18] Stefan Kronawitter, Sebastian Kuckuk, Harald Köstler, and Christian Lengauer. Automatic data layout transformations in the exastencils code generator. *Modern Physics Letters A*, 28(03):1850009, 2018.

[KL15] Stefan Kronawitter and Christian Lengauer. Optimizations applied by the exastencils code generator. *Faculty Inform. Math., Univ. Passau, Passau, Germany, Tech. Rep. MIP-1502*, pages 1–10, 2015.

[KMI+18] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. Performance evaluation of a vector supercomputer sx-aurora tsubasa. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 685–696. IEEE, 2018.

[koka] https://github.com/kokkos/kokkos/blob/master/example/tutorial/03_simple_view_lambda/simple_view_lambda.cpp. Accessed: 2019-12-17.

[kokb] https://github.com/kokkos/kokkos/blob/master/example/tutorial/Advanced_Views/01_data_layouts/data_layouts.cpp. Accessed: 2019-12-17.

[Kon86] Chisato Konno. A high level programming language for numerical simulation: Deqsol. *IEEE Denshi Tokyo*, 25:50–53, 1986.

[KSGK91] HT Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. *Parallelizing a new class of large applications over high-speed networks*, volume 26. ACM, 1991.

[KU93] Arnold R Krommer and Christoph W Ueberhuber. Architecture adaptive algorithms. *Parallel computing*, 19(4):409–435, 1993.

[KYS+87] Chisato Konno, McHIRU Yamabe, M Saji, N Sagawa, Y Umetani, H Hirayama, and T Ohta. Automatic code generation method of deqsol. *Journal of Information Processing, Information Processing Society of Japan*, 11(1):15–21, 1987.

[LAB+14] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

[LBN13] Pei Li, Elisabeth Brunet, and Raymond Namyst. High performance code generation for stencil computation on heterogeneous multi-device architectures. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1512–1518. IEEE, 2013.

[LF14] Xavier Lapillonne and Oliver Fuhrer. Using compiler directives to port large scientific applications to gpus: An example from atmospheric science. *Parallel Processing Letters*, 24(01):1450003, 2014.

[LHKK79] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[LLK+17] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific

kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26. ACM, 2017.

[LO07] Richard Lee and Carol O'Sullivan. A fast and compact solver for the shallow water equations. In *VRIPHYS*, pages 51–57, 2007.

[Lov93] David B Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[LWH12] Anders Logg, Garth N Wells, and Johan Hake. Dolfin: A c++/python finite element library. In *Automated Solution of Differential Equations by the Finite Element Method*, pages 173–225. Springer, 2012.

[MA17] Michel Müller and Takayuki Aoki. Hybrid fortran: High productivity gpu porting framework applied to japanese weather prediction model. *arXiv preprint arXiv:1710.08616*, 2017.

[Mar97] Bret Andrew Marsolf. *Techniques for the interactive development of numerical linear algebra libraries for scientific computation*. PhD thesis, Citeseer, 1997.

[MCT96] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[MF94] Nashat Mansour and Geoffrey C Fox. Parallel physical optimization algorithms for allocating data to multicomputer nodes. *The Journal of Supercomputing*, 8(1):53–80, 1994.

[MGR+12] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PHJ Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[Mit14] Sparsh Mittal. A survey of techniques for managing and leveraging caches in gpus. *Journal of Circuits, Systems, and Computers*, 23(08):1430002, 2014.

[MNSM11] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2011.

[MSBCP14] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. On the performance portability of structured grid codes on many-core computer architectures. In *International Supercomputing Conference*, pages 53–75. Springer, 2014.

[MV08] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[NDA+19] Philipp Neumann, Peter Düben, Panagiotis Adamidis, Peter Bauer, Matthias Brück, Luis Kornblueh, Daniel Klocke, Bjorn Stevens, Nils Wedi, and Joachim Biercamp. Assessing the scales in numerical weather and climate predictions: will exascale be the rescue? *Philosophical Transactions of the Royal Society A*, 377(2142):20180148, 2019.

[NKD+15] Ankireddy Nalamalpu, Nasser Kurd, Anant Deval, Chris Mozak, Jonathan Douglas, Ashish Khanna, Fabrice Paillet, Gerhard Schrom, and Boyd Phelps. Broadwell: A family of ia 14nm processors. In *2015 Symposium on VLSI Circuits (VLSI Circuits)*, pages C314–C315. IEEE, 2015.

[OBM10] Dominic A Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24. ACM, 2010.

[Opea] OpenACC. `https://www.openacc.org/`.

[Opeb] OpenCL. `https://www.khronos.org/opencl/`.

[Opec] OpenMP. `https://www.openmp.org/`.

[PMS17] James Price and Simon McIntosh-Smith. Exploiting auto-tuning to analyze and improve performance portability on many-core architectures. In *International Conference on High Performance Computing*, pages 538–556. Springer, 2017.

[Prz90] Steven A Przybylski. *Cache and memory hierarchy design: a performance directed approach*. Morgan Kaufmann, 1990.

[PSL16] Simon J Pennycook, Jason D Sewall, and Victor W Lee. A metric for performance portability. *arXiv preprint arXiv:1611.07409*, 2016.

[RB85] John R Rice and Ronald F Boisvert. Solving elliptic problems using ellpack. *Springer Series in Computational Mathematics, New York: Springer, 1985*, 1985.

[RHM+17] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):24, 2017.

[RMG+14] István Z Reguly, Gihan R Mudalige, Michael B Giles, Dan Curran, and Simon McIntosh-Smith. The ops domain specific abstraction for multi-block structured grid computations. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*, pages 58–67. IEEE, 2014.

[RMM+12] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.

[Rou13] Margaret Rouse. Definition: multi-core processor. *TechTarget. Retrieved March*, 6, 2013.

[RT00] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 32–32. IEEE, 2000.

[SBC17] Mohammed Sourouri, Scott B Baden, and Xing Cai. Panda: A compiler framework for concurrent cpu

+

+ gpu execution of 3d stencil computations on gpu-accelerated supercomputers. *International Journal of Parallel Programming*, 45(3):711–729, 2017.

[SD90] Robert Schreiber and Jack J Dongarra. Automatic blocking of nested loops. 1990.

[SGC+16] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[SKH+14] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 42–51. IEEE, 2014.

[SMN+19] Matheus S Serpa, Francis B Moreira, Philippe OA Navaux, Eduardo HM Cruz, Matthias Diener, Dalvan Griebler, and Luiz Gustavo Fernandes. Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE, 2019.

[STDH03] P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra. *Computational Science - ICCS 2002: International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings*. Number pt. 1 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.

[STY+14] Masaki Satoh, Hirofumi Tomita, Hisashi Yashiro, Hiroaki Miura, Chihiro Kodama, Tatsuya Seiki, Akira T Noda, Yohei Yamada, Daisuke Goto, Masahiro Sawada, et al. The non-hydrostatic icosahedral atmospheric model: Description and development. *Progress in Earth and Planetary Science*, 1(1):18, 2014.

[TCK+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[Tuf17] Finn-Haakon Ellingsrud Tuft. Mcl: A library for supporting multi-gpgpu programming. Master's thesis, 2017.

[UCB11] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[VEWC96] Robert Van Engelen, Lex Wolters, and Gerard Cats. Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications. In *Proceedings of the 10th international conference on Supercomputing*, pages 86–93. ACM, 1996.

[Wha08] R Clint Whaley. Empirically tuning lapack's blocking factor for increased performance. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 303–310. IEEE, 2008.

[Wol87] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.

[Wol89] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[WPD01] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[WW92] Sanjiva Weerawarana and Paul S Wang. A portable code generator for cray fortran. *ACM Transactions on Mathematical Software (TOMS)*, 18(3):241–255, 1992.

[WWX+16] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.

[Yao98] Aixiang I Song Yao. *An Efficient Parallel Three-Level Preconditioner for Linear Partial Differential Equations*. PhD thesis, Virginia Tech, 1998.

[yas] `https://github.com/intel/yask/blob/master/src/stencils/SimpleTestStencils.cpp`. Accessed: 2018-05-20.

[You15] Charles Yount. Vector folding: improving stencil performance via multi-dimensional simd-vector representation. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 865–870. IEEE, 2015.

[YTBD16] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2016 Sixth International Workshop on*, pages 30–39. IEEE, 2016.

[ZNG07] Weirong Zhu, Yanwei Niu, and Guang R Gao. Performance portability on earth: a case study across several parallel architectures. *Cluster Computing*, 10(2):115–126, 2007.

[ZRRB15] Günther Zängl, Daniel Reinert, Pilar Rípodas, and Michael Baldauf. The icon (icosahedral non-hydrostatic) modelling framework of dwd and mpi-m: Description of the non-hydrostatic dynamical core. *Quarterly Journal of the Royal Meteorological Society*, 141(687):563–579, 2015.

Appendices

A. Shallow Water Equation (SWE) Solver - GGDML Code

```
/*****************************************************************\
  SWESol.c:
  Solve Shallow Water Equations using GGDML language Extensions
  Nabeeh Jum'ah & Prof. John Thuburn
\*****************************************************************/

#include <sys/time.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

// Fields defined using GGDML specifiers
float CELL 2D f_H;       /* Surface level and tendency  */
float CELL 2D f_HT;
float CELL 2D f_B;
float EDGE 2D f_U;       /* Velocity and tendencies */
float EDGE 2D f_UT;
float EDGE 2D f_V;
float EDGE 2D f_VT;
float EDGE 2D f_F;       /* Flux */
float EDGE 2D f_G;

// Scalars
const float dx = 1.0;    /* Space and time intervals */
const float dy = 1.0;
const float dt = 0.001;
const float g  = 9.8;    /* Gravity */
const float f  = 0.1;    /* Coriolis */

/*****************************************************************\
  Compute flux:
  Compute both X ynd Y components of the flux
\*****************************************************************/

void compute_flux()
{
    // Compute the flux component in the X dimension
    foreach e IN grid {

        // Use GGDML access operators east_cell & west_cell
        //  to refer to the cells sharing the edge
```

```
42          f_F[e] = f_U[e] * (f_H[e.east_cell()] +
43                             f_H[e.west_cell()]) / 2.0;
44      }
45
46      // compute the flux component in the Y dimension
47      foreach e in grid {
48
49          // Use GGDML access operators north_cell & south_cell
50          // to refer to the cells sharing the edge
51          f_G[e] = f_V[e] * (f_H[e.north_cell()] +
52                             f_H[e.south_cell()]) / 2.0;
53      }
54  }
55
56  /*****************************************************************\
57    compute_U_tendency:
58    Compute tendency of the velocity in the X dimension
59  \*****************************************************************/
60
61  void compute_U_tendency()
62  {
63      // Compute different terms of tendency
64      // Use GGDML iterator to traverse the edges where the U-Tendency
65      // is located
66      foreach e in grid {
67
68          // Use GGDML access operators edge_????_neighbor
69          // to refer to the neiboring U edges in X direction
70          float udux = f_U[e] * (f_U[e.edge_east_neighbor()] -
71                                 f_U[e.edge_west_neighbor()])
72                               / (2.0 * dx);
73
74          // Use GGDML access operators edge_??_neighbor
75          // to refer to the neiboring V edges
76          float vbar = (f_V[e.edge_ne_neighbor()] +
77                        f_V[e.edge_nw_neighbor()] +
78                        f_V[e.edge_se_neighbor()] +
79                        f_V[e.edge_sw_neighbor()]) / 4.0;
80
81          // Use GGDML access operators edge_v?????_neighbor
82          // to refer to the neiboring U edges in Y direction
83          float vduy = vbar * (f_U[e.edge_vnorth_neighbor()] -
84                               f_U[e.edge_vsouth_neighbor()])
85                             / (2.0 * dy);
86
87          // Use GGDML access operators east_cell & west_cell
88          //  to refer to the cells sharing the edge
89          float gdhbx = g * (f_H[e.east_cell()] +
90                             f_B[e.east_cell()] -
91                             f_H[e.west_cell()] -
92                             f_B[e.west_cell()]) / dx;
93
```

```
94          float fvbar = f * vbar;
95          f_UT[e] = fvbar - udux - vduy - gdhbx;
96      }
97  }
98
99  /*****************************************************************\
100   update_U:
101   Update the velocity in the X dimension
102 \*****************************************************************/
103
104 void update_U()
105 {
106     foreach e in grid {
107         f_U[e] = f_U[e] + f_UT[e] * dt;
108     }
109 }
110
111 /*****************************************************************\
112   compute_V_tendency:
113   Compute tendency of the velocity in the Y dimension
114 \*****************************************************************/
115
116 void compute_V_tendency()
117 {
118     // Compute different terms of tendency
119     // Use GGDML iterator to traverse the edges where the V-Tendency
120     // is located
121     foreach e in grid {
122         // Use GGDML access operators edge_?????_neighbor
123         // to refer to the neiboring V edges in Y direction
124         float vdvy = f_V[e] * (f_V[e.edge_north_neighbor()] -
125                                f_V[e.edge_south_neighbor()])
126                             / (2.0 * dy);
127
128         // Use GGDML access operators edge_??_neighbor
129         // to refer to the neiboring U edges
130         float ubar = (f_U[e.edge_en_neighbor()] +
131                       f_U[e.edge_es_neighbor()] +
132                       f_U[e.edge_wn_neighbor()] +
133                       f_U[e.edge_ws_neighbor()]) / 4.0;
134
135         // Use GGDML access operators edge_h????_neighbor
136         // to refer to the neiboring V edges in X direction
137         float udvx = ubar * (f_V[e.edge_heast_neighbor()] -
138                              f_V[e.edge_hwest_neighbor()])
139                           / (2.0 * dx);
140
141         // Use GGDML access operators north_cell & south_cell
142         //  to refer to the cells sharing the edge
143         float gdhby = g * (f_H[e.north_cell()] +
144                            f_B[e.north_cell()] -
145                            f_H[e.south_cell()] -
```

```
                                f_B[e.south_cell()]) / dy;

        float fubar = f * ubar;
        f_VT[e] = 0.0 - vdvy - udvx - gdhby - fubar;
    }
}

/*****************************************************************\
  update_V:
  Update the velocity in the Y dimension
\*****************************************************************/

void update_V()
{
    foreach e in grid {
        f_V[e] = f_V[e] + f_VT[e] * dt;
    }
}

/*****************************************************************\
  compute_H_tendency:
  Compute tendency of the surface level
\*****************************************************************/

void compute_H_tendency()
{
    // Compute the two terms of tendency
    // Use GGDML iterator to traverse the grid cells
    foreach c in grid {

        // Use GGDML access operators east_edge & west_edge
        //  to refer to the U edges of the cell
        float df = (f_F[c.east_edge()] -
                    f_F[c.west_edge()]) / dx;

        // Use GGDML access operators north_edge & south_edge
        //  to refer to the V edges of the cell
        float dg = (f_G[c.north_edge()] -
                    f_G[c.south_edge()]) / dy;

        f_HT[c] = df + dg;
    }
}

/*****************************************************************\
  update_H:
  Update the surface level
\*****************************************************************/

void update_H()
{
    // Update the surface level
```

```
198     // Use GGDML iterator to traverse the grid cells
199     foreach c in grid {
200         f_H[c] = f_H[c] - dt * f_HT[c];
201     }
202 }
203
204 /*****************************************************************\
205   update_values:
206   Call the tendecies computations and the update kernels
207 \*****************************************************************/
208
209 void update_values()
210 {
211     compute_U_tendency();
212     update_U();
213     compute_V_tendency();
214     update_V();
215     compute_H_tendency();
216     update_H();
217 }
218
219 /*****************************************************************\
220   time_sec:
221   A helper function to measure time
222   It returns a floating point value
223   Differnce between two calls allows measuring code execution time
224 \*****************************************************************/
225
226 double time_sec()
227 {
228     struct timeval tv;
229     gettimeofday(&tv, NULL);
230     return (double) tv.tv_sec + (double) tv.tv_usec / 1000000.0;
231 }
232
233 /*****************************************************************\
234   main:
235   The main entry point for the code
236   It allocates and deallocates the memory for the fields, and runs
237   the time-step loop
238 \*****************************************************************/
239
240 #define TIMESTEPS 1000
241
242 int main(int argc, char **argv)
243 {
244     // Initialize necessary libraries
245     INITCOMMLIB;
246     INITCOMM;
247
248     // Allocate necessary memory for the fields
249     ALLOC f_H;
```

```
250     ALLOC f_HT;
251     ALLOC f_U;
252     ALLOC f_UT;
253     ALLOC f_V;
254     ALLOC f_VT;
255     ALLOC f_B;
256     ALLOC f_F;
257     ALLOC f_G;
258 
259     int time_step = 0;
260     double run_time = time_sec();
261 
262     //time stepping loop
263     timestep(time_step = 0;
264              time_step < TIMESTEPS;
265              time_step++) {
266 
267         // Compute flux
268         compute_flux();
269 
270         // Compute tendencies and update values
271         update_values();
272     }
273     run_time = time_sec() - run_time;
274     printf("%f,%f\n",
275            run_time,
276            60.0 * GRIDX * GRIDY * TIMESTEPS / run_time / 1000000000
277           );
278 
279     // Deallocate memory
280     DEALLOC f_H;
281     DEALLOC f_HT;
282     DEALLOC f_U;
283     DEALLOC f_UT;
284     DEALLOC f_V;
285     DEALLOC f_VT;
286     DEALLOC f_B;
287     DEALLOC f_F;
288     DEALLOC f_G;
289 
290     // Finalize necessary libraries
291     FINCOMMLIB;
292 
293     return 0;
294 }
```

B. A Sample Configuration File for the SWE Solver Code

```
EXTERN: GVAL uint64_t size_t

INCLUDEPATHS:
.
ENDINCLUDEPATHS

SPECIFIERS: SPECIFIER(loc=CELL|EDGE) SPECIFIER(dim=2D)

DECLARATIONS:
 SUBSTITUTE CELL WITH NOTHING
 SUBSTITUTE EDGE WITH NOTHING
 SUBSTITUTE 2D WITH *restrict *restrict
ENDDECLARATIONS

ALLOCATIONS:
GLOBALVARS:
 int local_Y_Cregion;
 int local_Y_Eregion;
ENDGLOBALVARS
 CASE loc=CELL:
 {
            int num_Y_rows = 2 + local_Y_Cregion;
            int num_X_rows = 2 + GRIDX;
            $var_name = malloc(
                          num_Y_rows*num_X_rows*sizeof($data_type)+
                          num_Y_rows*sizeof(char*));
            char* pos = (char*)$var_name + num_Y_rows*sizeof(char*);
            for(int j=0;j<num_Y_rows;j++){
                $var_name[j] = ($data_type*)pos;
                pos+=num_X_rows*sizeof($data_type);
                    for(int i=0;i<num_X_rows;i++){
                        $var_name[j][i] = ($data_type)0;
                    }
            }
            for(int j=0;j<num_Y_rows-1;j++){
                $var_name[j] += 1;
            }
            $var_name += 1;
 }
 ENDCASE
 CASE loc=EDGE:
```

```
42  {
43              int num_Y_rows = 2 + local_Y_Eregion;
44              int num_X_rows = 2 + GRIDX + 1;
45              $var_name = malloc(
46                            num_Y_rows*num_X_rows*sizeof($data_type)+
47                            num_Y_rows*sizeof(char*));
48              char* pos = (char*)$var_name + num_Y_rows*sizeof(char*);
49              for(int j=0;j<num_Y_rows;j++){
50                  $var_name[j] = ($data_type*)pos;
51                  pos+=num_X_rows*sizeof($data_type);
52                      for(int i=0;i<num_X_rows;i++){
53                          $var_name[j][i] = ($data_type)0;
54                      }
55              }
56              for(int j=0;j<num_Y_rows-1;j++){
57                  $var_name[j] += 1;
58              }
59              $var_name += 1;
60  }
61   ENDCASE
62 ENDALLOCATIONS
63
64 DEALLOCATIONS:
65   CASE loc=CELL:
66    {
67     free((void*)&$var_name[-1]);
68    }
69   ENDCASE
70   CASE loc=EDGE:
71    {
72     free((void*)&$var_name[-1]);
73    }
74   ENDCASE
75 ENDDEALLOCATIONS
76
77 GLOBALDOMAIN:
78   COMPONENT(CELL2D):
79    RANGE OF YD= 0 TO GRIDY
80    RANGE OF XD= 0 TO GRIDX
81   ENDCOMPONENT
82   COMPONENT(EDGE2D):
83    RANGE OF YD= 0 TO GRIDY+1
84    RANGE OF XD= 0 TO GRIDX+1
85   ENDCOMPONENT
86   DEFAULT=CELL2D[CELL2D:cell,ce,c][EDGE2D:edge,ed,e]
87 ENDGLOBALDOMAIN
88
89 INDEXOPERATORS:
90 east_cell(): XD=$XD
91 west_cell(): XD=$XD-1
92 north_cell(): YD=$YD
93 south_cell(): YD=$YD-1
```

```
edge_east_neighbor(): XD=$XD+1
edge_west_neighbor(): XD=$XD-1
edge_north_neighbor(): YD=$YD+1
edge_south_neighbor(): YD=$YD-1
edge_ne_neighbor(): YD=$YD+1
edge_nw_neighbor(): YD=$YD+1
edge_nw_neighbor(): XD=$XD-1
edge_se_neighbor(): XD=$XD
edge_sw_neighbor(): XD=$XD-1
edge_vnorth_neighbor(): YD=$YD+1
edge_vsouth_neighbor(): YD=$YD-1
edge_en_neighbor(): XD=$XD+1
edge_es_neighbor(): XD=$XD+1
edge_es_neighbor(): YD=$YD-1
edge_wn_neighbor(): XD=$XD
edge_ws_neighbor(): YD=$YD-1
edge_heast_neighbor(): XD=$XD+1
edge_hwest_neighbor(): XD=$XD-1
east_edge(): XD=$XD+1
west_edge(): XD=$XD
north_edge(): YD=$YD+1
south_edge(): YD=$YD
ENDINDEXOPERATORS

ANNOTATIONS:
 LEVEL 0:pragma omp parallel for
ENDANNOTATIONS

CBLOCKING:
XD=20000
ENDCBLOCKING

DOMAINDECOMPOSITION:
 nodes=1
 processID=0
 INCLUDE:
 INITIALIZATION:
 {
 }
 ENDINITIALIZATION
 FINALIZATION:
 {
 }
 ENDFINALIZATION
ENDDOMAINDECOMPOSITION

LOCALDOMAIN:
 COMPONENT(CELL2D):
  RANGE OF YD= 0 TO local_Y_Cregion
 ENDCOMPONENT
 COMPONENT(EDGE2D):
  RANGE OF YD= 0 TO local_Y_Eregion
```

```
146  ENDCOMPONENT
147 ENDLOCALDOMAIN
148
149 COMMUNICATION:
150 COMMINITIALIZATION:
151 {
152  local_Y_Cregion = GRIDY;
153  local_Y_Eregion = GRIDY+1;
154 }
155 ENDCOMMINITIALIZATION
156 ENDCOMMUNICATION
```

List of Figures

List of Listings

List of Tables

Papers Published From This Thesis

1. [JKZ+17] Nabeeh Jumah, Julian M Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Thomas Meurdesoif. **GGDML: Icosahedral Models Language Extensions**. Journal of Computer Science Technology Updates, 4(1):1–10, 2017

2. [JK18] Nabeeh Jumah and Julian Kunkel. **Performance Portability of Earth System Models with User-Controlled GGDML Code Translation**. In High Performance Computing, number 11203 in Lecture Notes in Computer Science. Springer, 2018

3. [JK19a] Nabeeh Jumah and Julian Kunkel. **Automatic Vectorization of Stencil Codes With the GGDML Language Extensions**. In Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing, WPMVP'19, pages 2:1–2:7, New York, NY, USA, 2019. ACM

4. [JK19b] Nabeeh Jumah and Julian Kunkel. **Scalable Parallelization of Stencils Using MODA**. In High Performance Computing, pages 142–154, Cham, 2019. Springer International Publishing

5. [JK20] Nabeeh Jumah and Julian Kunkel. **Optimizing Memory Bandwidth Efficiency With User-Preferred Kernel Merge**. In Euro-Par 2019: Parallel Processing Workshops, volume 11997 of Lecture Notes in Computer Science (LNCS). Springer, 2020

Zusammenfassung

Stencil-Berechnungen sind für große wissenschaftliche Berechnungen wesentlich, zum Beispiel bei der Modellierung von Erdsystemen. Diese Berechnungen sind normalerweise zeitintensiv. Eine geringe Ausführungsdauer ist jedoch ein wichtiger Aspekt bei wissenschaftlichen Berechnungen. Zum Beispiel wäre es unpraktisch, wenn ein Modell zur Wettervorhersage von morgen einen Tag lang rechnen würde. Um die Ausführungsdauer zu minimieren, werden große Anstrengungen unternommen die Hardware gut auszunutzen.

Wissenschaftliche Anwendungen werden normalerweise unter Verwendung von Allzwecksprachen entwickelt, z.B. Fortran oder C++. Allzwecksprachen fehlt jedoch aufgrund ihrer Allgemeinheit semantische Information, die es ermöglicht bestimmte Optimierungsmöglichkeiten zu nutzen. Um diesen Mangel zu beheben, werden zumeist wichtige Code-Transformationen, die der Codeoptimierung dienen, manuell durchgeführt. Dies belastet die Wissenschaftler, die Zeit für die Optimierung aufwenden und Details über die Architekturen der Computersystemen lernen müssen.

Hierbei ergeben sich weitere damit verbundene Herausforderungen. Ein Wissenschaftler muss für eine Optimierung viele Details der Architekturen verstehen. Weiterhin ist das Tempo der Architekturentwicklung im Vergleich zur Lebensdauer von Modellen sehr hoch. Zur bestmöglichen Unterstützung einer Architektur und um neu eingeführte Besonderheiten zu unterstützen, ist eine Portierung des Codes notwendig. Eine weitere Herausforderung ist die bestehende große Vielfalt der Architekturen von heterogenen Supercomputern.

Neben den Herausforderungen, die aus Architektur stammen, führen die auf Anwendungsebene vielfältigen Auswahlmöglichkeiten der Algorithmen, also die Vielfalt der numerischen Methoden und die Gittertypen, einen weiteren Komplexitätsfaktor zur Modellentwicklung ein. Einschränkungen bestehender Methoden und Gittertypen führen dazu, dass neue Gitter mit unterschiedlichen Eigenschaften verwendet werden, z.B. ikosaedrische Gitter. Das Einführen neuer Gitter führt zu unterschiedlichen Formen von Stencils, um numerische Methoden anzuwenden, z.B. bringen dreieckige Tessellationen neue Formen von Nachbarschaften.

Um die Herausforderungen zu bewältigen, hebt diese Arbeit die semantische Ebene der Modellierungssprachen auf eine höhere Abstraktionsebene. Der Ansatz basiert auf einem anwendungsanpassbaren Satz von Spracherweiterungen, um die Verwendung der anwendungsbezogenen Semantik zu maximieren und Optimierungen zu ermöglichen. Dies erfordert ein Umdenken im Software-Engineering Prozess bei der Modellentwicklung, um die Ausnutzung der Anwendungssemantik zu maximieren, damit die Optimierung durch Werkzeuge ermöglicht wird: Zunächst müssen die Anwendungsanforderungen analysiert

werden, um die Gitter und Stencils, aus denen eine Anwendung besteht, zu identifizieren. Dann werden die räumlichen Beziehungen zwischen den Punkten, die die verschiedenen Stencils innerhalb der Anwendung bilden, analysiert. Diese räumlichen Beziehungen werden dann verwendet, um Spracherweiterungen zu definieren.

Im vorgeschlagenen Ansatz ermöglichen wir den Benutzern, Spracherweiterungen und deren Rolle im Optimierungsprozess zu definieren. Diese Informationen werden durch separate Konfigurationsdateien bereitgestellt. Auf diese Weise halten wir den Quellcode frei von Optimierungen und entlasten Wissenschaftler von der Optimierungsaufgabe. Dies ermöglicht es ihnen, das wissenschaftliche Problem gemäß einer Abstraktion, die zu ihren wissenschaftlichen Konzepten passt, auszuprogrammieren. Der Ansatz ermöglicht auch eine bessere Trennung von Rollen bei der Softwareentwicklung von Modellen. Wissenschaftler können sich auf Ihr Model konzentrieren, die Konfigurationsdateien können von wissenschaftlichen Programmierern erstellt werden, die die Optimierung für eine bestimmte Zielarchitektur beherrschen.

Wichtige Punkte, die wir in dieser Arbeit untersuchen, sind die Möglichkeit, die genannten anwendungsanpassbaren Spracherweiterungen zu nutzen, um den Optimierungsprozess voranzutreiben, und die Skalierbarkeit über mehrere Knoten hinweg zu verbessern. Wir bewerten auch die Auswirkungen der neuen Techniken auf die Codequalität und die Entwicklungskosten.

Der Hauptbeitrag dieser Arbeit ist die Entwicklung eines integrierten Ansatzes mit Techniken zur Maximierung des Einsatzes von Semantik bei der Optimierung und Skalierung von Stencil-Berechnungen zur Verwendung mit modernen Supercomputer. Dies geht einher mit einer Beschränkung der Rolle der Wissenschaftler auf die Kodierung wissenschaftlicher Probleme nach dem Prinzip der Trennung von Rollen zugunsten einer verbesserten Codequalität.

Die Wirksamkeit des Ansatzes wird durch Experimente mit verschiedenen Architekturen bestätigt: Mehrkernprozessoren, GPUs und Vektor-Engines. Die Vielseitigkeit des Ansatzes wird anhand der erreichbare Effizienz der generierten Codes und der Produktivität für die Wissenschaftler bewertet.

Analysen und experimentelle Ergebnisse zeigen, dass wir mit jeder Architektur einen hohen Prozentsatz der nominellen Leistung erzielen können. Wir können die Anzahl der Datenbewegungen vom Speicher in die Caches minimieren und etwa 80Diese experimentellen Ergebnisse stimmen mit den theoretischen Erwartungen an die erreichbare Leistung der getesteten Architekturen überein. Um die Portabilität der Leistung zu bewerten, verwenden wir denselben Quellcode (ohne Änderungen für ausgewählte Architekturen oder sonstigen speziellen Code) für unterschiedliche Architekturen.

Durch die Verwendung der vorgeschlagenen Spracherweiterungen können die Codegröße auf ein Drittel und die Entwicklungskosten auf weniger als die Hälfte reduziert werden. Eine wichtige Schlussfolgerung dieser Arbeit ist es, dass die anwendungsanpassbaren Spracherweiterungen die Codeoptimierung durch anwendungsspezifische Semantik maximieren, wobei sie an die Anforderungen bestimmter Anwendungen oder Domänen angepasst werden können.

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum	Unterschrift