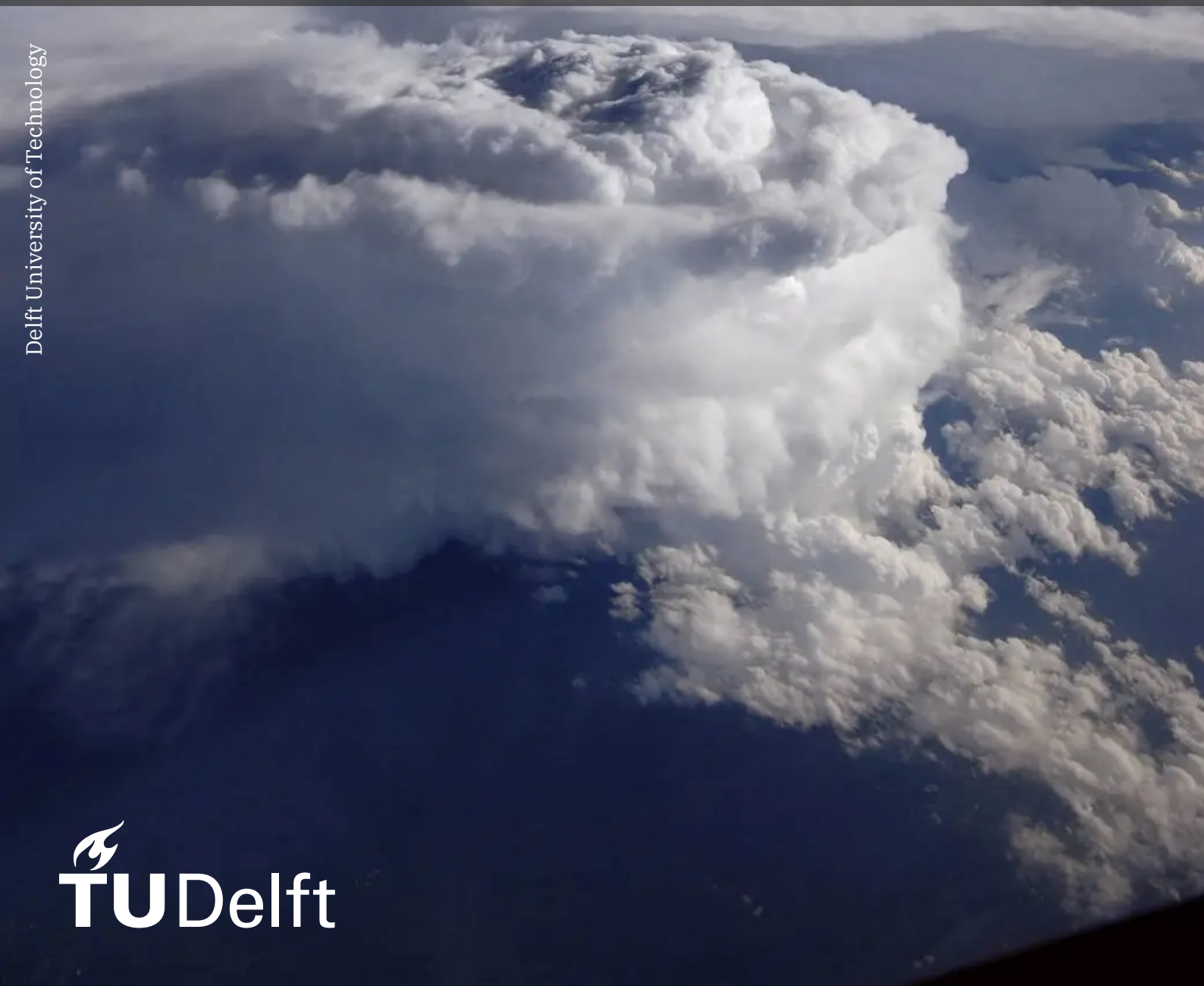
GPU-Accelerated Atmospheric Large Eddy Simulation

Preparing the Dutch Atmospheric Large Eddy Simulation model for the Exascale Era

C.A.A. Jungbacker



Delft University of Technology

TU Delft

GPU-Accelerated Atmospheric Large Eddy Simulation

Preparing the Dutch Atmospheric Large Eddy Simulation model for the Exascale Era

by

C.A.A. Jungbacker

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday, March 28 at 10:30 AM.

Student number:	4911067	
Project duration:	May, 2023 – March, 2024	
Thesis committee:	Dr. S. R. de Roode,	TU Delft, chair
	Dr. F. R. Jansson,	TU Delft, supervisor
	Dr. P. Simões Costa,	TU Delft, supervisor
	Prof. dr. A. P. Siebesma,	TU Delft

Cover: Storm Cell Over the Southern Appalachian Mountains by NASA (Modified)

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

With the completion of this thesis, my time as a student at TU Delft has come to an end. During my BSc in civil engineering, I developed a passion for fluid dynamics, numerical mathematics and scientific programming. Naturally, when the time came to choose an MSc program, I was searching for something in which I could combine the three. After I attended an information session on the Geoscience and Remote Sensing track and was introduced to the field of atmospheric modeling, I knew my destiny. Needless to say, this thesis topic really was the cherry on top of the cake.

This thesis would not have been possible without my supervisors. First and foremost, I would like to thank Fredrik, my daily supervisor, for his support during this project. Besides providing me with excellent guidance and valuable feedback, you have made me feel part of the research group by inviting me to various section meetings, introducing me to new people and showing off the cool projects you were using DALES for. I look forward to working with you in the future. Pedro, thank you for your great advice on OpenACC programming and for helping me out with the infamous Poisson solver. Last but not least, thank you Stephan for setting me up with this topic in the first place and for the enthusiasm with which you have received my work over the past months.

I would also like to thank Niels Jansen for keeping Suske and Wiske (the GPU workstations, named after the famous cartoon characters) running during my thesis work, and SURF (`www.surf.nl`) for the support in using the National Supercomputer Snellius. The computing time was provided by the Ruisdael Observatory (`www.ruisdael-observatory.nl`).

Finally, a word of thanks to my friends and family for their support over the past eleven months, and to my fellow students in “Het Afstudeerhok” on the third floor. I truly cherished the coffee breaks and the trips to PSOR, the latter of which seemed to happen increasingly more often near the end of our theses.

Caspar Jungbacker
Delft, March 2024

Abstract

Large Eddy Simulation (LES) is a mathematical technique for performing simulations of turbulent flows, such as those found in the Earth's atmosphere. Compared to traditional numerical weather and climate models, LES is more accurate in representing turbulent processes and cloud dynamics. The computational burden of LES, however, has historically limited its application to relatively small domain sizes. In this work, part of the DALES atmospheric LES model was ported to Graphics Processing Units (GPUs) using the OpenACC programming model. GPUs, originally designed for accelerating computations related to 3D computer graphics, excel at parallel computations, which are abundant in LES models. The performance of the GPU port of DALES was measured on an NVIDIA RTX 3090 in a desktop workstation and an NVIDIA A100 in the Snellius supercomputer and compared to the existing CPU implementation. For the BOMEX intercomparison case, a speedup of 11.6 was achieved versus 8 CPU cores on the desktop system, while on Snellius a speedup of 3.9 was observed compared to 128 CPU cores. Furthermore, the existing MPI parallelization of DALES was adapted such that multiple GPUs can be used simultaneously. The multi-GPU implementation was tested on up to 64 NVIDIA A100 GPUs. This thesis represents a step towards the enhancement of the scalability of DALES, enabling simulations on larger domains at higher resolutions. While a substantial acceleration of DALES was achieved, further efforts are needed to port more components of the model to the GPU to facilitate the simulation of increasingly realistic meteorological phenomena.

Contents

1 Introduction

Large Eddy Simulation (LES) is a technique used to accurately simulate complex fluid flows, of which the Earth's atmosphere is an example. Specifically, LES explicitly partly resolves turbulence, as opposed to traditional numerical weather and climate models. The latter use parameterizations for turbulent processes, leading to significant uncertainties (Schalkwijk, Jonker, Siebesma, & Van Meijgaard, 2015). However, due to resolution requirements, LES is computationally too expensive to apply on large domains.

Since it was discovered that Graphics Processing Units (GPUs) can be used to accelerate Artificial Intelligence (AI) models, the complexity of these models has been increasing at an ever-increasing rate (Mittal & Vaishay, 2019). This has sparked the onset of a feedback loop, where successes in AI lead to improvements in GPU technology, and vice versa. AI models consist of a large number of parallel computations, an area where the GPU excels. Fortunately, LES models are also comprised of such computations. Therefore, GPUs can help accelerate LES models and enable simulations on larger domains.

This thesis describes the process of GPU-accelerating DALES, an LES code tailored for atmospheric flows, by using the OpenACC programming model. The efforts are limited to the dynamical core and moist thermodynamic routines of DALES. Furthermore, the model is going to be adapted such that multiple GPUs can be used simultaneously. Validation of the accelerated model will be done through an approach based on ensemble statistics. A study of the performance of the model will be done on computer systems on two different scales: a desktop system with consumer hardware, and the Snellius supercomputer, hosted at SURF[1] on the Amsterdam Science Park. For both systems, the speedup of the accelerated model is going to be examined and on Snellius, the scaling to multiple GPUs will be tested.

The outline of this report is as follows: in chapter 2 the theory behind the modeling of turbulent flows is discussed and a technical description of DALES is given. Chapter 3 will explain GPU technology, and how GPUs can be used to accelerate fluid simulation codes. Then in chapter 4, the implementation of OpenACC in DALES will be described. Chapter 5 will present the results of the validation and performance studies. Finally, the conclusions and recommendations for future work are to be found in chapter 6.

[1] https://www.surf.nl/

2

Atmospheric Large Eddy Simulation

This chapter focuses on the science of computational fluid dynamics. First, a general description of turbulent flows is given. Second, different techniques for performing computer simulations of turbulent flows are explained and discussed in the context of numerical weather and climate modeling. Finally, the subject model of this thesis, the Dutch Atmospheric Large Eddy Simulation (DALES) model, will be discussed in more detail.

2.1. Turbulent flows

Under the right conditions, a fluid (or gas) in motion exhibits circular patterns that vary in size (Figure 2.1). A flow that features such patterns is described as a *turbulent flow*, and in turbulence theory, the circular patterns are called *eddies* (Pope, 2000). We, as mankind, are surrounded by these kinds of flows in our daily lives. Examples include the flow of water through pipes, airflow around a moving car, waterfalls and the Earth's atmosphere. Because turbulent flows are so predominantly present in our living environment, they are studied intensively by the scientific community and the industry. With these insights, accurate computer models are developed that attempt to predict the effects of turbulence.

To characterize turbulent flows, several scaling laws and non-dimensional numbers have been derived. The *Reynolds number* Re is defined as the ratio between inertial forces and viscous forces in a fluid:

$$\mathrm{Re} = \frac{\text{Inertial forces}}{\text{Viscous forces}} = \frac{u\mathscr{L}}{\nu} \tag{2.1}$$

in which u is the flow speed, $\mathscr{L}$ is a characteristic length scale of the flow, usually determined by the geometry of the flow, and ν is the kinematic viscosity of the flow. In turbulent flows, the inertial forces dominate and the Reynolds number is therefore high (> 4000 (Pope, 2000)).

In turbulent flows, the larger an eddy, the more unstable it is. Therefore, large eddies tend to break up into slightly smaller eddies. These smaller eddies are once again unstable and break up into even smaller eddies. This phenomenon is repeated until the eddies are small enough such that the viscous forces in the fluid are able to convert the remaining kinetic energy into heat. The length scale at which the conversion of kinetic energy into heat happens is called the *Kolmogorov length scale*, and is given by:

$$\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{1/4} \tag{2.2}$$

where ε is the rate of dissipation of kinetic energy.

2.2. Turbulence modeling

The motion of fluids is mathematically described by the famous Navier-Stokes equations, of which various forms exist. Here, the form that applies to *incompressible* fluids is used. An incompressible fluid is a fluid whose density does not change as a result of pressure changes (Pope, 2000). Using Einstein's summation convention, the Navier-Stokes equations for an incompressible fluid are given by:

$$\frac{\partial u_i}{\partial x_i} = 0, \tag{2.3}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j^2} + g_i, \tag{2.4}$$

where u_i denotes the velocity ($i = 1,2,3$, corresponding to x, y and z directions respectively), ρ is the density, p is the pressure, g_i denotes the body forces and ν is the kinematic viscosity. If one desires to study the evolution of a flow field in time, one would have to find a solution to Equations 2.3 and 2.4. However, no analytical solution to the general form of these equations has been found. To find an approximate solution to the Navier-Stokes equations, numerical methods are often used.

The objective of solving Equations 2.3 and 2.4 such that eddies of all scales are represented imposes two requirements on the computational mesh. First, the mesh has to span a large enough area such that the largest scales can be captured. Second, the mesh spacing must be small enough to be able to represent the Kolmogorov scale. If both requirements are met, all turbulent motions can be resolved and no parameterizations are needed. This technique is called Direct Numerical Simulation (DNS) (Pope, 2000). While DNS is very accurate, it is computationally *very* expensive. Even for moderate domain sizes, DNS is often unfeasible due to the resolution requirements.

Smagorinsky (1963) proposed the idea of explicitly resolving only the largest, most energetic eddies and using a model for the smaller eddies. This idea was further worked out by Lilly (1967) and Deardorff (1974). To arrive at the governing equations of LES, a filter is applied to Equations 2.3 and 2.4. This filter can be thought of as a low-pass filter; small-scale, high-frequency motions are filtered out and large-scale motions remain (Leonard, 1975). After the filtering operation, the filtered Navier-Stokes equations are obtained:

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0, \tag{2.5}$$

$$\frac{\partial \overline{u}_i}{\partial t} + \frac{\partial \overline{u}_i \overline{u}_j}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{u}_i}{\partial x_j^2} + f_i - \frac{\partial \tau_{ij}}{\partial x_j}. \tag{2.6}$$

In Equation 2.6, it can be seen that the term $\partial \tau_{ij}/\partial x_j$ arises, which is called the *subfilter-scale stress tensor*. The evaluation of τ_{ij} requires a model, called the *subfilter-scale model*. Some examples of subfilter-scale models include the Smagorinsky model (Smagorinsky, 1963), the Bardina model (Bardina, Ferziger, & Reynolds, 1980) or the Vreman model (Vreman, 2004). These models are turbulent viscosity models, which assume that the subfilter-scale motions similarly dissipate kinetic energy as the viscous forces in the fluid. Because the small-scale turbulent motions are modeled in an LES, the required resolution of the computational mesh is significantly lower compared to DNS, making it a suitable technique for a wider variety of problems, specifically problems that require larger domain sizes. Still, a high enough resolution must be used such that enough of the energy-carrying eddies are resolved. For this reason, LES is not considered as computationally cheap.

Instead of attempting to (partly) *resolve* the small-scale turbulent motions, as done with DNS and LES, one could also choose to *model* the effect of turbulence on the mean flow field. Models that follow this line of thought are referred to as Reynolds-Averaged Navier-Stokes models (Pope, 2000). The governing equations of a RANS model follow from applying the Reynolds averaging rules to the Navier-Stokes equations (Equations 2.3 and 2.4). To this end, the velocity field u_i is decomposed into a mean part $\overline{u_i}$, and a part u_i' that represents fluctuations due to turbulence, such that:

$$u_i = \overline{u_i} + u_i'. \tag{2.7}$$

This act is called *Reynolds decomposition*. This decomposition operation is applied to Equations 2.3 and 2.4, yielding:

$$\frac{\partial \left(\overline{u_i} + u_i'\right)}{\partial x_i} = 0, \tag{2.8}$$

$$\frac{\partial \left(\overline{u_i} + u_i'\right)}{\partial t} + \left(\overline{u_j} + u_j'\right) \frac{\partial \left(\overline{u_i} + u_i'\right)}{\partial x_j} = -\frac{1}{\left(\overline{\rho} + \rho'\right)} \frac{\partial \left(\overline{p} + p'\right)}{\partial x_i} + \nu \frac{\partial^2 \left(\overline{u_i} + u_i'\right)}{\partial x_j^2} + \overline{g_i} + g_i', \tag{2.9}$$

Finally, all terms in Equations 2.8 and 2.9 are averaged in time, yielding the RANS equations (Pope, 2000, Chapter 4):

$$\frac{\partial \overline{u_i}}{\partial x_i} = 0, \tag{2.10}$$

$$\frac{\partial \overline{u_i}}{\partial t} + \overline{u_j} \frac{\partial \overline{u_i}}{\partial x_j} = -\frac{1}{\overline{\rho}} \frac{\partial \overline{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{u_i}}{\partial x_j^2} + \overline{g_i} + \frac{\partial \overline{u_i' u_j'}}{\partial x_i}. \tag{2.11}$$

The term $\overline{u_i' u_j'}$ in Equation 2.11 is called the Reynolds stress. A RANS model only solves Equations 2.10 and 2.11 for the mean quantities $\overline{u_i}$ and $\overline{p}$, meaning that the Reynolds stress term needs to be modeled. Often used models include turbulent viscosity models (similar to LES) or Reynolds stress models, which use transport equations to evaluate the Reynolds stress (Pope, 2000). More recently, the use of machine learning methods has been explored for modeling the Reynolds stress (Brunton, Noack, & Koumoutsakos, 2020).

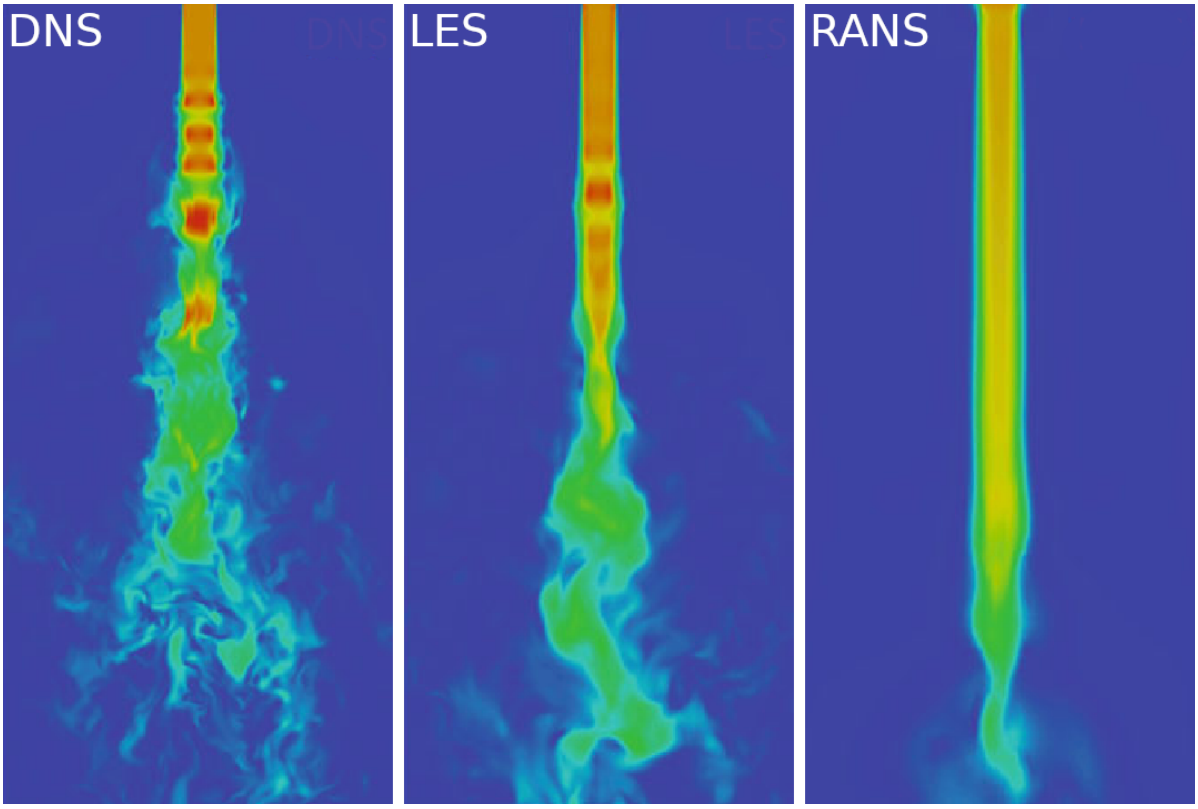


Figure 2.1: Visualization of the velocity distribution of a jet flow, as simulated by the three turbulence modeling techniques. From left to right: Direct Numerical Simulation (DNS), Large Eddy Simulation (LES) and Reynolds-Averaged Navier-Stokes (RANS). Adapted from Rodriguez (2019).

Because turbulence is not explicitly resolved by RANS models, the required resolution is the least out of the three simulation techniques. For this reason, RANS is computationally cheap but is also the least accurate.

A visual comparison between the three techniques can be found in Figure 2.1. Following the theory, it can be seen that compared to the full-resolved DNS flow field, the LES simulation is able to reproduce most of the large-scale turbulent features. Especially at a greater distance from the entry of the main jet (at the top of the image), LES loses the small-scale details compared to DNS. The RANS flow field shows little instabilities at all as a consequence of the time averaging.

2.3. Towards high-resolution weather and climate models

Due to limitations of computational capacity, global numerical weather and climate models often use grids with low horizontal resolution in combination with the RANS formulation. Due to the low resolution, small-scale phenomena, such as cloud formation, have to be parameterized, which introduces significant uncertainties in weather forecasts and climate projections (Slingo & Palmer, 2011). At higher resolutions (in the order of 1-2 kilometers), these small-scale processes are still not fully resolved, but their influence on the resolved flow field can be captured partly (Schär et al., 2020). Therefore, global weather and climate models are continuously developed and updated to support increasing resolutions. One of the problems that arise at these high resolutions is that the aforementioned parameterizations do not perform well when convective processes are partly resolved, and therefore have to be adapted to this fact (Wyngaard, 2004). The range

of resolutions at which this problem persists is also referred to as the *terra incognita*, or the *grey zone* (Schalkwijk et al., 2015; Wyngaard, 2004). Furthermore, Schär et al. (2020) identify the exploitation of modern Graphics Processing Unit technology as another key challenge towards high-resolution weather and climate modeling. GPU technology will be explained more thoroughly in chapter 3, but for now it suffices to say that a GPU is a hardware device that can accelerate specific calculations of a weather or climate model. To allow a numerical weather or climate model to make use of this specialized hardware, significant changes to the source code are needed.

On the other end of the resolution spectrum, the LES technique has been used to develop atmospheric models that explicitly resolve turbulent motions and do not require the aforementioned parameterizations, therefore also lacking the associated uncertainties (Schalkwijk et al., 2015). Hence, LES can potentially act as a replacement for RANS-based models for high-resolution atmospheric modeling. However, as discussed before, the computational burden associated with LES is quite high, which limits the application of LES over very large domain sizes. Again, GPUs can help speed up calculations, making large-scale LES possible. The use of GPUs in LES has been demonstrated by Schalkwijk et al. (2015), who were able to do a weather forecast-like simulation over the entire Netherlands using an LES model.

2.4. DALES

The Dutch Atmospheric Large Eddy Simulation (DALES) model is a large-eddy simulation model designed for high-resolution modeling of the atmosphere and its processes, like cloud formation and precipitation (Heus et al., 2010; Ouwersloot, Moene, Attema, & De Arellano, 2017).

2.4.1. Prognostic equations

DALES uses five prognostic variables to define the state of the atmosphere: three velocity components u, v and w, the liquid water potential temperature θ_l and the total water specific humidity q_t. The general form of the momentum equation (Equation 2.4) for the atmosphere is given by (Stull, 1988):

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} - g\delta_{i3} + f_i. \tag{2.12}$$

Compared to Equation 2.4, Equation 2.12 lacks the viscous term and introduces the gravity g. In atmospheric flows, the Reynolds number (Equation 2.1) is large, which is why the viscous term can be safely neglected. DALES makes use of the *anelastic approximation*, meaning that density differences are neglected except for the vertical direction (Böing, 2014). To this end, a time-independent base density profile ρ_0 is introduced, which only varies in the vertical direction. Equation 2.12 then can be written as:

$$\frac{\partial u_i}{\partial t} + \frac{1}{\rho_0}\frac{\partial \rho_0 u_i u_j}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} - g\delta_{i3} + f_i. \tag{2.13}$$

The pressure gradient $\partial p/\partial x_i/\rho$ can be expressed as a function of the virtual potential temperature (Böing, 2014):

$$\frac{1}{\rho}\frac{\partial p}{\partial x_i} \approx -g\delta_{i3} - g\delta_{i3}\frac{\theta_v - \theta_{v,e}}{\theta_{v,e}} + \frac{\partial}{\partial x_i}\frac{p'}{\rho_e}, \tag{2.14}$$

in which the subscript e denotes an environmental state that only varies in the vertical direction and in time. p' is the pressure fluctuation from the environmental pressure p_e. In turn, θ_v can be evaluated from the prognostic variables:

$$\theta_v = \left(\theta_l + \frac{L_v}{c_{pd}\Pi_e}q_c\right)\left(1 - \left(1 - \frac{R_v}{R_d}q_t\right) - \frac{R_v}{R_d}q_c\right), \tag{2.15}$$

where L_v, c_{pd}, R_v and R_d are constants (their meaning and values are omitted here for brevity but can be found in e.g. Stull (1988)), $\Pi_e = (p_e/p_0)^{R_d/c_{pd}}$ in which $p_0 = 10^5$ is a reference pressure and q_c is the cloud condensate, which can be diagnosed. After combining Equations 2.13 and 2.14 and applying the LES filtering operation, we arrive at the governing momentum equation of DALES:

$$\frac{\partial \overline{u}_i}{\partial t} = -\frac{1}{\rho_0}\frac{\partial \rho_0 \overline{u}_i \overline{u}_j}{\partial x_j} - \frac{\partial \pi}{\partial x_i} + g\delta_{i3}\frac{\overline{\theta_v} - \theta_{v,e}}{\theta_{v,e}} + f_i - \frac{\partial \tau_{ij}}{\partial x_j} \tag{2.16}$$

where π denotes the modified pressure given by $\pi = p'/\rho_e + 2e/3$, where e is the turbulence kinetic energy. Furthermore, f_i contains the body forces (e.g., the Coriolis force) and τ_{ij} is the subfilter-scale stress tensor. Similarly, the momentum equation for a scalar φ, where $\varphi \in \{\theta_l, q_t\}$, is given by:

$$\frac{\partial \overline{\varphi}}{\partial t} = -\frac{1}{\rho_0}\frac{\partial \rho_0 \overline{u}_j \overline{\varphi}}{\partial x_j} - \frac{\partial R_{u_j,\varphi}}{\partial x_j} + S_\varphi, \tag{2.17}$$

in which $R_{u_j,\varphi}$ is a sub-filter scale flux and S_φ is a source term.

2.4.2. Subfilter-scale model

DALES uses an eddy-viscosity model to evaluate the subfilter-scale fluxes. Following this model, the subfilter-scale stress tensor in Equation 2.6 is given by:

$$\tau_{ij} = -K_m\left(\frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i}\right), \tag{2.18}$$

in which K_m is the eddy viscosity for momentum. For scalars, the subfilter-scale flux $R_{u_j,\varphi}$ is given by:

$$R_{u_j,\varphi} = -K_h\frac{\partial \overline{\varphi}}{\partial x_j}, \tag{2.19}$$

where K_h is the eddy diffusivity for thermodynamic scalars. K_m and K_h can be calculated in two ways: as a function of the turbulence kinetic energy e as formulated by Deardorff (1980) or using the Smagorinsky model. Following Deardorff (1980), K_m and K_h are modeled as:

$$K_m = c_m \lambda e^{1/2}, \quad K_h = c_h \lambda e^{1/2}, \tag{2.20}$$

where c_m and c_h are constants and λ is a length scale. This formulation introduces a new prognostic variable, namely the turbulence kinetic energy e. The prognostic equation for e is given by:

$$\frac{\partial e}{\partial t} = -\frac{\partial \overline{u}_j e}{\partial x_j} - \tau_{ij} \frac{\partial \overline{u}_i}{\partial x_j} + \frac{g}{\theta_0} R_{w,\theta_v} - \frac{\partial R_{u_j,e}}{\partial x_j} - \frac{1}{\rho_0} \frac{\partial R_{u_j,\pi}}{\partial x_j} - \varepsilon, \tag{2.21}$$

of which the first two terms can be calculated from the resolved velocity field. The terms containing subfilter-scale fluxes ($R_{u_j,\varphi}$) and the dissipation rate ε have to be parameterized. The third term, representing the subfilter-scale production of TKE due to buoyancy effects, is parameterized as:

$$\frac{g}{\theta_0} R_{w,\theta_v} = \frac{g}{\theta_0} \left(A R_{w,\theta_l} + B R_{w,q_t} \right), \tag{2.22}$$

in which A and B are coefficients whose value depends on moisture and temperature. Next, the fourth and fifth terms of Equation 2.21, which represent the turbulent transport of TKE, are combined into a single term:

$$-\frac{\partial}{\partial x_j} \left(R_{u_j,e} + \frac{1}{\rho_0} R_{u_j,\pi} \right) = \frac{\partial}{\partial x_j} \left(2 K_m \frac{\partial e}{\partial x_j} \right). \tag{2.23}$$

Finally, the dissipation rate ε is modeled as:

$$\varepsilon = \frac{c_\varepsilon e^{3/2}}{\lambda}, \quad c_\varepsilon = 0.19 + 0.51 \frac{\lambda}{\Delta}, \tag{2.24}$$

where Δ is the LES filter width. Equations 2.21, 2.22, 2.23 and 2.24 are combined into a prognostic equation for the *square root* of e:

$$\begin{aligned} \frac{\partial e^{1/2}}{\partial t} = &- \overline{u}_j \frac{\partial e^{1/2}}{\partial x_j} + \frac{1}{2e^{1/2}} \left(K_m \left(\frac{\partial \overline{u}_j}{\partial x_i} + \frac{\partial \overline{u}_i}{\partial x_j} \right) \frac{\partial \overline{u}_i}{\partial x_j} - K_h \frac{g}{\theta_0} \frac{\partial}{\partial z} \left(A \overline{\theta}_l + B \overline{q}_t \right) \right) \\ &+ \frac{\partial}{\partial x_j} \left(2 K_m \frac{\partial e^{1/2}}{\partial x_j} \right) - \frac{c_\varepsilon e}{2\lambda} \end{aligned} \tag{2.25}$$

2.4.3. Mass conservation

Following the anelastic approximation, the expression for mass conservation reads:

$$\frac{\partial \rho_0 \overline{u}_i}{\partial x_i} = 0. \tag{2.26}$$

Equation 2.26 states that the divergence of the velocity field u_i must be equal to zero. This condition can be enforced by using Chorin's projection method (Chorin, 1967). Per Chorin's method, the time integration of $\overline{u}_i$ is split into two parts. The first step consists

of calculating an intermediate velocity field that is not divergence-free, by evaluating all right-hand terms of Equation 2.16, except for the pressure term:

$$\frac{\partial \overline{u}_i^*}{\partial t} = -\frac{1}{\rho_0}\frac{\partial \rho_0 \overline{u}_i \overline{u}_j}{\partial x_j} + g\delta_{i3}\frac{\overline{\theta_v} - \theta_{v,e}}{\theta_{v,e}} + f_i - \frac{\partial \tau_{ij}}{\partial x_j}, \tag{2.27}$$

where $\overline{u}_i^*$ denotes the velocity field that is not divergence-free. From Equation 2.27, it follows that:

$$\frac{\partial \overline{u}_i}{\partial t} = \frac{\partial \overline{u}_i^*}{\partial t} - \frac{\partial \pi}{\partial x_i}. \tag{2.28}$$

To obtain an explicit equation for the pressure π, the divergence of Equation 2.28 can be taken. After rearranging terms, this leads to the following expression:

$$\frac{\partial^2 \pi}{\partial x_i^2} = \frac{\partial}{\partial x_i}\frac{\partial \overline{u}_i^*}{\partial t}. \tag{2.29}$$

Despite its simplicity at first sight, Equation 2.29 is often the most costly part of a CFD simulation (Costa, 2018). DALES offers the option to solve this equation with the Fast Fourier Transform (FFT) algorithm, which can offer significantly better performance compared to iterative solvers that are often used (Hockney, 1965). After Equation 2.29 is solved, the velocity field is corrected according to Equation 2.28, which concludes the projection method.

2.4.4. Discretization

For the discretization of the governing equations, DALES makes use of an Arakawa C-grid (Arakawa & Lamb, 1977). This means that the velocity components are defined at the faces of the grid cells, while pressure, turbulence kinetic energy and scalars are defined at the centers of the grid cells. DALES features several numerical schemes to evaluate advective terms, ranging from a simple but less accurate second-order central difference scheme to accurate fifth and sixth-order schemes. The κ advection scheme, as described by Hundsdorfer, Koren, vanLoon, and Verwer (1995), is also available. The latter scheme ensures that the advected quantity remains positive.

For time integration of prognostic variables, DALES uses a third-order Runge-Kutta scheme. The maximum time step size Δt for which numerical stability is guaranteed is determined by the Courant-Friedrichs-Lewy criterion (CFL) and the diffusion number d:

$$\mathrm{CFL} = \max\left(\left|\frac{\overline{u}_i \Delta t}{\Delta x_i}\right|\right), \quad d = \max\left(\sum_{i=1}^{3}\frac{K_m \Delta t}{\Delta x_i^2}\right). \tag{2.30}$$

For the CFL and d maximum values are prescribed, from which the maximum time step Δt is determined.

2.4.5. Parallelization

To speed up computations, DALES is parallelized using the Message Passing Interface (MPI). MPI is a protocol and programming interface for managing data communication between processors. When multiple processors are used for running a DALES simulation,

the computational domain is split up into sub-domains, and each sub-domain is assigned to a processor. MPI is then used whenever a processor needs access to data that is located on a different sub-domain. The sub-domains can have various shapes, as illustrated in Figure 2.2. The computational efficiency of each decomposition depends, among other factors, on the size and shape of the full domain and the number of processors used.

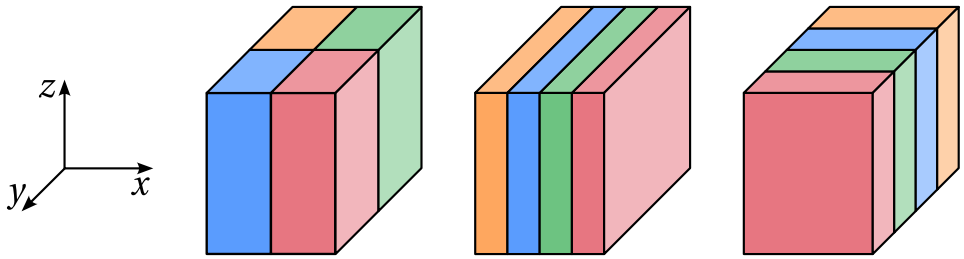


Figure 2.2: Possible domain decompositions in DALES. The full computational domain is illustrated by the cube and sub-domains are indicated by the different colors. From left to right: z-aligned pencils, y-aligned slabs and x-aligned slabs.

2.4.6. Applications

DALES has been tested in a variety of model intercomparison studies. Examples include the BOMEX case on shallow cumulus convection (Siebesma et al., 2003), the GABLS case on stable boundary layers (Beare et al., 2006) and the ASTEX case on stratocumulus transition (Van Der Dussen et al., 2013). DALES also features an interactive chemistry module, which has been used by Vilà-Guerau De Arellano et al. (2011) to study the behavior of atmospheric chemical reactants over the Amazon rainforest. De Bruine, Krol, Vilà-Guerau De Arellano, and Röckmann (2019) expanded DALES with an explicit aerosol scheme to study aerosol-cloud interactions. Furthermore, a Python interface has been developed for DALES by Van Den Oord et al. (2020). Using this interface, the model can be coupled to other (global) weather and climate models. This functionality has been exploited by Jansson et al. (2019) to perform *superparameterization* experiments; a DALES instance is nested in each column of a selected region of a global atmospheric model such that processes related to clouds and convection are solved explicitly.

3

Graphics Processing Units

This chapter starts with an introduction to the technology behind Graphics Processing Units (GPUs) and how it compares to Central Processing Units (CPUs). Next, different methods of GPU programming are presented. Third, some examples of how GPUs are applied to accelerate CFD applications are given. Finally, some trends in the field of GPU computing are discussed.

3.1. CPU and GPU architecture

The main component of a CPU is the *core*, which performs arithmetic operations. Nowadays, CPUs often consist of multiple cores, from which the term *multicore processor* arises (Rauber & Rünger, 2023). These cores each have private access to a small amount of fast memory called *cache memory*. Cores are connected via an *interconnection network* to form a CPU. The main memory is not part of the CPU and is physically located somewhere else in the computer. A schematic of a multicore CPU can be found in Figure 3.1. In a CPU, each core can be controlled individually, which makes the CPU a Multiple Instructions, Multiple Data (MIMD) system; each core can fetch instructions by itself at any time and it does not have to wait until all other cores in the CPU have finished executing (Flynn, 1966). In practice, the MIMD design of CPUs means that each core can be occupied with a different application. A modern high-end CPU can have in the order of tens of cores.

The GPU was originally invented to accelerate computations revolving around 3D computer graphics (Aamondt, Wai Lun Fung, & Rogers, 2018). In the early 2000s, it was discovered that GPUs could also be used for general calculations. Specifically, GPUs were found to offer enormous performance benefits over the traditional CPU for highly parallel computations, such as the matrix-matrix product (Larsen & McAllister, 2001).

GPUs are made up of Streaming Multiprocessors (SMs), and each SM contains multiple cores that can perform calculations (Rauber & Rünger, 2023). A schematic representation of an SM can be found in Figure 3.2. Unlike the cores inside a CPU, the cores in an SM cannot function independently; each core has to execute the same instruction, and a new instruction can only be executed once all cores have finished executing the previous instruction. This execution model is called Single Instruction, Multiple Data (SIMD) (Flynn, 1966). The NVIDIA A100, a modern high-end data center GPU, consists of 108 SMs, each containing 64 single-precision cores for a total of 6,912 cores and 32 double-precision cores for a total of 3,456 cores (NVIDIA, 2020). Despite their massive amount

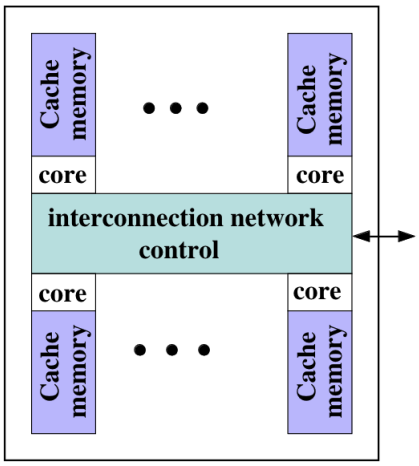


Figure 3.1: Diagram of a modern multicore CPU (Rauber & Rünger, 2023). The arrow indicates a connection to other components of the computer, such as the main memory.

of cores, GPUs do not outperform CPUs in every workload. To fully utilize the potential performance of a GPU, an algorithm has to be suited for SIMD execution. I.e., an algorithm has to consist of a (very) large amount of independent calculations that can be executed in parallel. This is the reason why GPUs can never fully replace CPUs. The majority of the tasks related to the operating system of a computer are sequential by nature, for which the CPU is much better suited.

Another area where GPU design is different from traditional CPUs is the memory configuration. GPUs generally do not share memory with CPUs. Rather, GPUs store data in memory that is located on the GPU itself. This memory is optimized for high throughput, meaning that a large amount of data can flow from or to memory in a given time interval (Aamondt et al., 2018). CPU memory, on the other hand, is located somewhere else in the computer and is optimized for providing low latency. This means that the time between the CPU requesting data from memory and receiving the requested data is low. The high memory bandwidth makes the GPU also a suitable device for memory-intensive applications.

The last important component in a GPU system is the interconnection between CPU and GPU. This connection is often quite slow compared to the memory bandwidth of both CPUs and GPUs. Because the CPU and GPU do not share memory, data transfers between CPU and GPU are needed during runtime, which can significantly slow down an application.

Figure 3.2: Diagram of a Streaming Multiprocessor (SM). The boxes indicated by “INT32”, “FP32” and “FP64” are the computational cores for integers, single-precision floating-point and double-precision floating-point calculations, respectively. (Rauber & Rünger, 2023)

3.2. Programming GPUs

To exploit the capabilities of GPUs, an application has to be written using a specialized programming model. There are numerous programming models available, each with different characteristics, which can be categorized as follows:

Native kernel-based models

Kernel-based methods involve the writing of special functions (also referred to as kernels) that describe the computations that have to be done by a single thread of the GPU. At runtime, this kernel is then *launched* across multiple threads, which all perform the kernel on their own piece of data. The word *native* in this context means that the model is designed for use with a specific family of GPUs. Perhaps the most well-known example of a native kernel-based model is NVIDIA's Compute Unified Device Architecture (CUDA) (Dally, Keckler, & Kirk, 2021). Native kernel-based programming models provide very fine-grained control over the GPU, thereby allowing for a great degree of optimization and consequently, the best performance. A drawback, however, is that these methods require (partial) rewriting of existing code. If a CPU version of the program is required in addition to a GPU version, two separate versions of the code have to be maintained, which is expensive and can introduce bugs.

Portable kernel-based models

As opposed to native kernel-based models, portable kernel-based models are designed to work on hardware from a variety of vendors. The program still consists of kernels, but these kernels can be launched with different backends of different accelerator vendors. The great benefit is that a program can run on different hardware without the need to rewrite the code, hence the name *portable*. When the application is compiled, the target hardware is selected and the kernels are translated into code that can run natively on the selected hardware. Examples of portable kernel-based programming models include Kokkos (Trott et al., 2022) and SYCL (Khronos Group, 2023).

Directive-based models

A fundamentally different approach to GPU programming is the use of compiler directives. Compiler directives are instructions to the compiler to handle a section of code differently. These compiler directives are placed near computationally expensive sections of the code that can benefit from GPU acceleration. When the code is compiled, the annotated sections of code are compiled into GPU-compatible code. Because directive-based methods do not require the writing of kernels, an application can be ported to GPUs with relatively low effort. An additional benefit of compiler directives is that they are added to the original code, meaning that only one version of the source code has to be maintained. When the application has to be built for CPUs, the directives are simply ignored by the compiler. Two popular directive-based programming models are OpenACC (Farber, 2017) and OpenMP (Antao et al., 2016).

Standard language parallelism

A more recent trend has been the implementation of parallel programming features directly in programming languages. An example is the Fortran `do concurrent` construct. Its usage is similar to directive-based models: parallelizing loops. With the right compiler, `do concurrent` loops can be executed on GPUs (Kedward et al., 2022). A great benefit of standard language parallelism is that it is part of the syntax of the language, thus offering good usability for developers already familiar with the language.

3.3. Usage of GPUs for computational fluid dynamics (CFD)

Niemeyer and Sung (2014) have examined the status of GPU computing in the field of CFD. To demonstrate the potential benefits of using GPUs for CFD, two case studies were performed: a 2D Laplace equation solver, resembling the Poisson equation that is often found in CFD codes, and a lid-driven cavity flow. Four implementations were tested: single-core CPU, multi-core CPU with OpenMP, GPU with CUDA, and GPU with OpenACC. For mesh sizes up to 512^2, the wall-clock time for the GPU implementations exceeded that of the multi-core GPU implementation. While the authors do not explicitly explore the possible causes of this behavior, it can be argued that the increase in wall-clock time is due to data transfers between the CPU and GPU. For larger mesh sizes, the GPU implementations outperformed the CPU implementations. Specifically, for the Laplace equation, the GPU solver showed a speedup of about 4.6, while for the lid-driven cavity flow, the speedup was about 2.8. Remarkably, Niemeyer and Sung (2014) showed that as the mesh size increases, the wall-clock time of the OpenACC implementation converges to that of the CUDA implementation, indicating that the benefits of the low-level optimizations that CUDA offers are not as important for large problem sizes.

Costa (2018) has developed a tool for DNS of turbulent flows, called CaNS. The dynamical core of CaNS is very similar to that of DALES: both use finite-difference discretization on a structured, staggered grid, an FFT-based solver for the pressure, and third-order Runge Kutta time integration. Parallelization of CaNS is achieved through domain decomposition with MPI, with further fine-grained (i.e., at the level of loops) parallelization via OpenMP. CaNS was later adapted for GPUs using CUDA Fortran (Costa, Phillips, Brandt, & Fatica, 2021), but this was later switched out in favor of OpenACC. NVIDIA's cuFFT library was used to perform the FFT calculations on GPUs. Performance analysis was done on two systems: an NVIDIA DGX Station, a system comparable in size to a modern desktop PC and containing 4 Tesla V100 GPUs, and an NVIDIA DGX-2, a system that is more comparable to something that one would find in a supercomputer. The NVIDIA DGX-2 contains 16 of the same Tesla V100 GPUs. Costa et al. (2021) found that for a constant problem size, one would need about 6100 to 11200 CPU cores to match the wall-clock time per time step of the 16 Tesla V100s in the NVIDIA DGX-2. This is still a conservative estimate, as linear scaling was assumed for the CPU code, whereas in reality, performance often scales sublinearly for a given problem size due to overhead introduced by communication between CPUs.

DALES itself has been ported to GPUs before by Schalkwijk, Griffith, Post, and Jonker (2012). To this end, the original Fortran code of DALES was translated to C++, and calculations were moved to the GPU using CUDA, resulting in the GPU-resident Atmospheric Large-Eddy Simulation (GALES) model. Schalkwijk et al. (2012) found that GALES was able to reduce the wall-clock time per time step by a factor of 2 compared to DALES. Since then, the company Whiffle has adopted GALES and further developed it into the GPU-Resident Atmospheric Simulation Platform (GRASP). GRASP is often used for very accurate simulations of windfarms (Verzijlbergh, 2021).

Another LES model that shares characteristics with DALES is MicroHH, as described by Van Heerwaarden et al. (2017). In terms of the physical assumptions, DALES and MicroHH are almost identical, but MicroHH also features the option to perform DNS of atmospheric flows. MicroHH is written in C++ and simulations on the GPU are made possible through CUDA. Again, significant performance benefits were found when running the model on the GPU. A limitation of MicroHH, however, is that it cannot utilize more than one GPU at a time.

3.4. Trends in GPU computing

As can be seen in Figure 3.3, the number of supercomputers with hardware accelerators (of which GPUs are an example) installed has grown steadily over the last decade. With the launch of Frontier in 2022, supercomputers have reached a significant milestone: the ability to perform 1×10^{18} floating-point operations per second (Choi, 2022). To reach this level of performance, Frontier too makes heavy use of GPUs. This trend can be partially attributed to the rapid advances in the field of Artificial Intelligence (AI), whose algorithms greatly benefit from the large number of cores offered by GPUs (Mittal & Vaishay, 2019). As the development of AI algorithms does not show any sign of slowing down (Xu et al., 2021), GPU technology will continue to improve for the foreseeable future from which the CFD community can profit too.

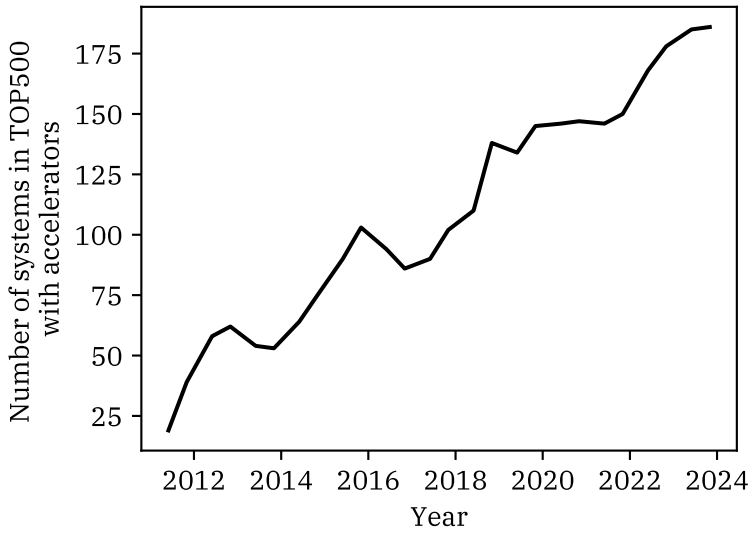


Figure 3.3: Total number of systems in the TOP500 (list of the 500 most powerful supercomputers worldwide) that feature hardware accelerators over time. Data from TOP500 (2023)

4

Implementation

In this chapter, the technical details of the implementation of OpenACC in DALES are discussed. First, an explanation is given on why OpenACC was chosen for DALES. Thereafter, it is explained how OpenACC was used to offload computationally expensive parts of the code to the GPU, where special attention is given to the Poisson solver. Finally, some further optimizations are covered, as well as the extension to multiple GPUs.

4.1. OpenACC

As discussed in section 3.2, there are numerous programming models available for targetting accelerators (GPUs). For this work, the directive-based approach with OpenACC was chosen. This choice was based on several observations. For once, DALES features a wide palette of physical schemes for the simulation of processes related to clouds, precipitation, land surface, buildings, et cetera. Consequently, the sheer volume of the DALES source code is also quite large. Because it can be added to the existing code, the usage of OpenACC prevents the code base from growing even more. Second, DALES is constantly being updated and extended by developers with various backgrounds, ranging from students to researchers. Often, these developers do not possess intimate knowledge of GPU architecture and GPU programming. In this regard, OpenACC is a good choice because it removes the need for low-level optimization of kernels. Finally, DALES is deployed on various computer systems, ranging from modest desktop workstations to supercomputers like Snellius and Fugaku (Jansson et al., 2023). In this regard, OpenACC is a good choice because it can be compiled for a variety of computing devices.

To compile OpenACC code, one has to use a compatible compiler. Throughout this work, the NVFortran compiler has been used. This compiler is included in the NVIDIA HPC SDK, a collection of compilers and libraries for the development of GPU applications, such as an MPI library.

4.2. Offloading loops

The first step towards a GPU port of DALES was to offload computationally expensive parts of the code. These parts generally consist of (nested) loops that perform some calculation local to each grid point. This offloading was done by adding *compute constructs* to the code. In OpenACC terms, a compute construct instructs the compiler that the annotated code should be compiled for the selected device. Here, the term *device* refers to the accelerator.

Following this terminology, the CPU is referred to as the *host* OpenACC features three compute constructs: `kernels`, `parallel` and `serial`. In this work, the `kernels` and `parallel` constructs were used, which will be discussed in the following section.

4.2.1. The `kernels` construct

When a loop or structured block is annotated with a `kernels` construct, the compiler is given the freedom to parallelize the code by itself. To do so, the compiler first has to determine if the annotated code can be parallelized *at all*. This means that the generated device kernels are guaranteed to produce correct results, but it may also lead to unnecessary serial execution of parallelizable code because the compiler has to make a conservative decision.

In this work, the `kernels` construct was mainly used to offload sections that contained Fortran array syntax. Array syntax is a special syntax that allows for compact notation of operations on (multi-dimensional) arrays. For example, instead of constructing a (nested) loop to add two arrays to each other, one can simply write `C = A + B`. The `kernels` construct can be used to efficiently offload these sections of code. An example of a section of Fortran array syntax that is offloaded with the `kernels` construct can be found in Listing 4.1.

Listing 4.1: Fortran array syntax offloaded using the OpenACC `kernels` construct. This snippet comes from the module `tstep` of DALES and resets the arrays for the prognostic variables between time steps.

```
1 !$acc kernels
2 up = 0.
3 vp = 0.
4 wp = 0.
5 thlp = 0.
6 qtp = 0.
7 svp = 0.
8 e12p = 0.
9 !$acc end kernels
```

4.2.2. The `parallel` construct

Similar to the `kernels` construct, the `parallel` construct also lets the compiler generate parallel device code. The difference between the two is that the `parallel` construct lets the programmer decide how a section of code should be parallelized, while the `kernels` construct transfers this responsibility to the compiler.

By default, code that is annotated with a `parallel` construct will be executed by all available cores on the device redundantly. In other words, all cores will execute all loop iterations at the same time. It should be clear that this will not result in any speedup, since the parallelism offered by the accelerator is not being exploited. Therefore, by itself, the `parallel` is not useful. To distribute loop iterations over the available device threads, the `loop` directive can be added to a `parallel` construct. An example of the usage of the `parallel loop` construct can be found in Listing 4.2. In this example, the loop contains 1000 iterations. When the program encounters this loop, 1000 threads will be spawned on the device. Each core will get a private copy of the loop index `i` and will do the calculation for this value of `i`.

In DALES, loops are often nested. These nested loops can be offloaded by nesting `loop` directives. This is demonstrated in the left pane of Listing 4.3. Again, when the outermost loop is encountered, 1000 threads will be generated. Then, each thread will generate 1000 more threads for the next loop. This procedure is repeated until the innermost loop

is reached. Another option is to add a `collapse` clause to a `parallel loop` directive. The `collapse` clause will combine all nested loops into one large loop. This approach is often more efficient than nesting `loop` directives as all threads can be generated at once.

It should be noted that for a loop to be parallelizable, all iterations should be independent of one another. In contrast to the `kernels` construct, the compiler will not check for such dependencies if the `parallel loop` construct is used. Dependencies between loop iterations can lead to incorrect results or performance bottlenecks if one does not take appropriate action.

Listing 4.2: Example of a simple loop that is parallelized using the `parallel loop` constuct.

```
!$acc parallel loop
do i = 1, 1000
  a(i) = b(i) + 2
end do
```

Listing 4.3: Two ways to offload nested loops: in the left pane by nesting `loop` directives, and in the right pane by using the `collapse` clause. This snippet comes from the `tstep` module of DALES and integrates the velocity field u in time.

```
!$acc parallel loop
do k = 1, k1
  !$acc loop
  do j = 2, j1
    !$acc loop
    do i = 2, i1
      u0(i,j,k) = um(i,j,k) + rk3coeff
          * up(i,j,k)
    end do
  end do
end do
```

```
!$acc parallel loop collapse(3)
do k = 1, k1
  do j = 2, j1
    do i = 2, i1
      u0(i,j,k) = um(i,j,k) + rk3coeff
          * up(i,j,k)
    end do
  end do
end do
```

4.3. The Poisson solver

As discussed in subsection 2.4.3, DALES uses a pressure correction method to ensure a divergence-free velocity field. In the current version of DALES, the Poisson equation can be solved in two ways: with Fast Fourier Transforms (FFTs) or an iterative solver. In this work, the solver based on FFTs has been extended to make use of GPUs.

The Poisson solver consists of five steps :

1. Evaluate the right-hand side of Equation 2.29;
2. Perform a 2D FFT at every vertical level;
3. Solve a tridiagonal system;
4. Perform a 2D inverse FFT at every vertical level;
5. Correct the velocity field with the gradient of the calculated pressure to make it divergence-free.

Steps (1), (3) and (5) mainly consist of nested loops that have been parallelized using the `parallel loop` and `kernels` constructs. In DALES version 4.4, the FFTs of steps (2) and (4) can be evaluated either by using the FFTPACK library (Swarztrauber, 1982), which is included with the source code of DALES, or the more recent Fastest Fourier Transform in the West (FFTW) library (Frigo & Johnson, 1998). Neither one of these libraries can

utilize GPUs. Therefore, DALES had to be coupled to a new FFT library. Currently, there are multiple FFT libraries available that support execution on GPUs. NVIDIA offers cuFFT, which is designed for NVIDIA GPUs (Nvidia, n.d.). cuFFT is written in C++/CUDA, but Fortran bindings are included. rocFFT is a similar library developed by AMD, for AMD GPUs (AMD, 2023b). Additionally, AMD offers the hipFFT library (AMD, 2023a). hipFFT functions as a wrapper library around cuFFT and rocFFT, allowing for execution on both Nvidia GPUs and AMD GPUs. Another promising new library is VkFFT, developed by Tolmachev (2023). VkFFT supports multiple backends allowing for execution on all major vendors of data center GPUs (Nvidia, AMD, Intel). In addition, VkFFT supports discrete cosine transforms, which are useful for simulations with Dirichlet or Neumann boundary conditions [1] (Schumann & Sweet, 1988). For this work, the cuFFT library was chosen because of the availability of Fortran bindings and its tight integration within the NVIDIA HPC SDK.

The evaluation of 2D Fourier transforms in a pencil decomposition is a non-trivial task. In this setting, each sub-domain has access to the full vertical direction, but only parts of the horizontal direction (see Figure 2.2). To be able to evaluate a Fourier transform in a horizontal direction, the sub-domains have to be transposed such that each device has full access to that direction. This transposition is further complicated by the fact that communication between processes (GPUs) needs to take place. The complete forward FFT algorithm is as follows:

1. The domain is transposed from z-aligned to x-aligned;
2. 1D Fourier transforms are evaluated in the x-direction;
3. The domain is transposed from x-aligned to y-aligned;
4. 1D Fourier transforms are evaluated in the y-direction;
5. The domain is transposed from y-aligned to z-aligned.

After the final transposition, a tridiagonal system is solved for the Fourier coefficients of the pressure $\hat{p}$. Then, to convert the Fourier coefficients back to pressure in physical space, the algorithm above is executed in reverse order. The full routine for solving the Poisson equation in a pencil decomposition is visualized in Figure 4.1.

[1] DALES version 4.4 does not support Dirichlet or Neumann boundary conditions. However, work is currently being done to include these in an updated version of DALES (see Liqui Lung, Jakob, Siebesma, and Jansson (2023))

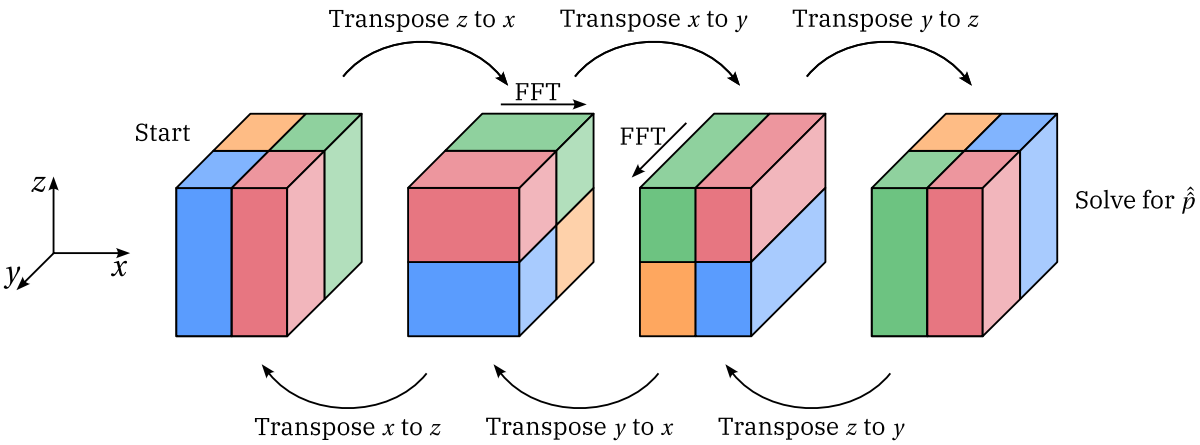


Figure 4.1: Impression of transpositions required for a multi-dimensional FFT on a pencil-decomposed domain.

4.4. Optimizations

4.4.1. Optimizing data locality

In most computing systems, the host and device both have their own memory space. Therefore, if the device has to perform calculations on some data, this data has to be copied from the host memory to the device memory. With OpenACC, the programmer has the option to leave data management to the compiler or to manage it themselves explicitly. In most cases, letting the compiler decide when to move data will be detrimental to the performance of an application.

Data movement by the compiler will be illustrated with Listing 4.4 as an example. Upon reaching the first loop, the array `a` is not present in device memory yet. Hence, the compiler adds an instruction right before the loop to copy array `a` from host to device memory. When the copy has finished, the loop is executed on the device. After the loop is done executing, `a` is copied back to the host again and the device memory is freed. Next, the program reaches the second loop. Once again, the compiler adds instructions to copy `a` to the device before the loop and copy it back to the host after the loop. One may notice that there is no need to perform any data movement between these loops; data can be copied to the device before the start of the first loop, and only has to be copied back to the host after the second loop has finished executing. Since data transfers are a major bottleneck in device computing, optimizing the transfers manually is an essential task to improve performance.

OpenACC offers multiple directives to manage data transfers manually. The `enter data` directive has been used the most in DALES. This directive can be used with a `create` clause, which takes a list of variables as input and allocates device memory for these variables, or a `copyin` clause, which does the same but additionally initializes the device copies of the variables with the values that are in host memory. Data that is moved by the `enter data` directive stays on the device until the program has finished executing, or an `exit data` directive is encountered. The latter deallocates device memory after (optionally) copying the data back to the host. This method of data management is applied to the algorithm of Listing 4.4 in Listing 4.5. In this example, array `a` is allocated before the first loop. Between the two loops, no data movement is done. After the second loop is done, `a` is copied back to the host memory and the device memory that was occupied is made available again.

In Listing 4.5, one can also notice the use of the `default` clause. This clause can be used on a compute construct and tells the compiler what the default assumption on the locality of the data accessed in a loop should be. By providing the value `present`, the compiler assumes that the data is already present on the device and no data movement is needed. If the data is not present, the program will crash. The `default(present)` clause was added to all compute constructs in DALES to ensure that all data resides on the device.

Listing 4.4: Example algorithm without explicit data management.

```
!$acc parallel loop
do i = 1, i1
  a(i) = a(i) + 2
end do

!$acc parallel loop
do i = 1, i1
  a(i) = a(i) / 4
end do
```

Listing 4.5: Example algorithm with explicit data management by including `enter data` and `exit data` directives.

```
!$acc enter data copyin(a)

!$acc parallel loop default(present)
do i = 1, i1
  a(i) = a(i) + 2
end do

!$acc parallel loop default(present)
do i = 1, i1
  a(i) = a(i) / 4
end do

!$acc exit data copyout(a)
```

4.4.2. Memory allocation

In DALES, most of the memory required for storing variables is allocated during the startup procedure and is deallocated only after the simulation has finished. Some subroutines, however, allocate and deallocate memory once they are called, which can be every time step. As memory allocations on the GPU are quite costly, this workflow can introduce a significant performance penalty. This was fixed by moving arrays from the subroutine scope to the module scope. The arrays are then allocated once during the startup procedure and are kept alive throughout the time loop. The implementation of this optimization is illustrated in Listing 4.6.

Listing 4.6: Improving memory allocation. On the left, the array `profile` belongs to the scope of the subroutine `calculate_profile`. On the right, `profile` moved to the scope of the module and allocated in the subroutine `init_profile`, which is called once in the startup procedure of the program.

```
1  module statistics
2
3  contains
4
5    subroutine calculate_profile
6      implicit none
7      real, allocatable :: profile(:)
8      allocate(profile(kmax))
9      !$acc enter data copyin(profile)
10     ! Perform calculation on profile
11     !$acc exit data copyout(profile)
12   end subroutine
13 end module
```

```
1  module statistics
2
3  real, allocatable :: profile(:)
4
5  contains
6    subroutine init_profile
7      implicit none
8      allocate(profile(kmax))
9      !$acc enter data copyin(profile)
10   end subroutine
11
12   subroutine calculate_profile
13     implicit none
14     ! Perform calculation on profile
15   end subroutine
16
17   subroutine exit_profile
18     implicit none
19     !$acc exit data delete(profile)
20     deallocate(profile)
21   end subroutine
22 end module
```

4.5. Asynchronous kernels

OpenACC allows for asynchronous execution of device kernels. This means that when the host sends a task to the device, it does not wait until the device has finished executing this task. The host is allowed to continue running the program while the device is busy processing. New tasks can even be sent to the device while it is still occupied with the previous task. This is possible because the device manages tasks through *queues*. When the host sends a task to the device, the task is put in a queue. The device then executes the enqueued tasks in order. Additionally, tasks can be executed concurrently by placing them in different queues.

Asynchronous kernel execution can significantly benefit performance. This performance increase can especially be noticed for relatively small loops that, for example, only iterate over one or two dimensions. In these cases, the cost of sending the kernel to the device can be relatively high compared to the actual cost of executing the kernel. Most of the offloaded loops in DALES have been made asynchronous by adding an `async` clause to the `parallel loop` or `kernels` directives. The usage of the `async` clause is demonstrated in Listing 4.7. Here, the calculations for `thvf` and `thv0` are fully independent and are therefore put in two different queues (indicated by the argument provided to the `async` clause). The expression for `rhof` depends on `thvf` and is therefore put in the same queue as the latter. Once this section of code is reached during execution, all loops are sent to the device without waiting for each to complete before sending the next one. At line 21 in Listing 4.7, a `wait` directive is encountered, where the program will wait until the tasks in all queues have been executed.

Listing 4.7: Example of the usage of the `async` clause in the thermodynamics module of DALES.

```
!$acc parallel loop default(present) async(1)
do k = 1, k1
  thvf(k) = thvf(k) / ijtot
end do

!$acc parallel loop collapse(3) default(present) async(2)
do k = 1, k1
  do j = 2, j1
    do i = 2, i1
      thv0(i,j,k) = (thl0(i,j,k) + rlv * ql0(i,j,k) / (cp * exnf(k))) &
                    * (1 + (rv / rd - 1) * qt0(i,j,k) - rv / rd * ql0(i,j,k))
    end do
  end do
end do

!$acc parallel loop default(present) async(1)
do k = 1, k1
  rhof(k) = presf(k) / (rd * thvf(k) * exnf(k))
end do

!$acc wait
```

4.6. Extension to multiple GPUs

With OpenACC, it is possible to utilize multiple GPUs (Farber, 2017). However, OpenACC can only make use of devices that live on the same node. In practice, this means that one is limited to using only a few GPUs (for example, the Snellius supercomputer has 4 GPUs per node (SURF, n.d.-a)). Another option is to adapt the current MPI parallelization described in subsection 2.4.5, which was done in this work. Following this approach, the full computational domain is decomposed into sub-domains and distributed across CPUs. Each CPU then has access to a GPU. This approach is very flexible as multiple nodes can be used, allowing DALES to scale to an arbitrary number of GPUs.

As mentioned before, data transfers from GPU to CPU are expensive. To circumvent the need for CPU-GPU data transfers when doing MPI communications, special implementations of the MPI standard have been developed, often referred to as GPU-aware MPI (Potluri, Hamidouche, Venkatesh, Bureddy, & Panda, 2013). As the name suggests, a GPU-aware MPI implementation can handle data that is located in GPU memory. When a call to the MPI library is made, the GPU-aware MPI backend checks if the provided data is located in CPU or GPU memory. If the data is indeed located in GPU memory, the MPI backend will make sure that data is transferred directly from GPU to GPU and does not go through the CPU first.

To make the location of the data in GPU memory available to the MPI backend, the OpenACC `host_data` directive can be used in combination with the `use_device` clause. This is illustrated in Listing 4.8. In this example, the reference to `array` in the `MPI_Allreduce` call will point to device memory.

Listing 4.8: Providing a device pointer to an MPI call with a `host_data` directive.

```
!$acc host_data use_device(array)
call MPI_Allreduce(MPI_IN_PLACE, array, sz, MPI_REAL, MPI_SUM, comm, ierror)
!$acc end host_data
```

5

Validation and Benchmarking

This chapter delves into the validation and performance analysis of DALES on the GPU. First, the test case used for validation is introduced and the results of the validation study are presented. Next, the systems and performance metrics used for examining the performance of the model are explained. Finally, the model performance is discussed.

5.1. BOMEX

The Barbados Oceanographic and Meteorological Experiment (BOMEX) case has been used for benchmarking and validation of the GPU-accelerated version of DALES. This case originates from the field experiment carried out by Holland and Rasmusson (1973). During this experiment, observations of horizontal wind components, temperature and specific humidity were made every 90 minutes over a 500×500 km area in the Atlantic Ocean, east of the island of Barbados. The meteorological conditions during the observed period of five days were relatively constant and gave rise to the development of shallow cumulus convection without the presence of precipitation. Because of these steady-state conditions, the BOMEX observations formed a good base for the LES intercomparison study carried out by Siebesma et al. (2003). One of the main goals of this study was to compare the ability of different LES models, including DALES, to produce shallow cumulus clouds. Since the original publication, the BOMEX case has been widely used as a benchmark for new LES models (e.g. Van Heerwaarden et al. (2017)). The setup of the BOMEX case is described by Siebesma and Cuijpers (1995). Because of the nature of the case, BOMEX only stresses a relatively limited portion of the components of DALES: the dynamical core, which is occupied with solving the horizontal and vertical transport of momentum and scalars, and the moist thermodynamics scheme, responsible for cloud formation. Therefore, BOMEX is a good starting point for validation, which can later be expanded on as more components of DALES are accelerated.

5.2. Model validation

Modifying the source code of an application carries the risk of introducing bugs. Therefore, the updated source code was validated against the original CPU implementation to ensure that the application logic did not change as a result of the addition of OpenACC directives and associated optimizations. However, validation of an atmospheric code like DALES is complicated by the fact that weather is a chaotic phenomenon. As described by Lorenz (1963), a small change in the initial conditions of a simulation can lead to vastly different

outcomes. Furthermore, computers cannot work with infinitely long decimal numbers, giving rise to intermediate round-off errors during the simulation. These round-off errors can accumulate and influence the simulation. Therefore, the output of two model runs, both started from the same initial conditions, will generally not be identical. In this work, a statistical approach was used to validate the model instead. First, the original CPU version of DALES was used to create an ensemble data set. For this data set, fifty model runs (also called *ensemble members*) were used, each initialized with a random perturbation applied to the prognostic fields. Next, the ensemble mean and standard deviation were calculated. Similarly, a fifty-member ensemble was generated using the OpenACC version of DALES. In Figure 5.1, the OpenACC ensemble mean and standard deviation are compared to the statistics of the original ensemble for the liquid water potential temperature θ_l, total water specific humidity q_t (both prognostic variables), total liquid water potential temperature flux $w'\theta_l'$ and the total moisture flux $w'q_t'$ (both diagnostic variables). The plotted quantities were calculated during the last hour of the case. It can be seen that the statistics of the OpenACC version show good correspondence with the original version. Hence, there is little reason to believe that the model physics have changed as a result of the offloading.

5.3. Performance metrics

Wall clock time and speedup

Wall clock time is the total time it takes for an application to finish executing. In other words, wall clock time is the real-world time that passes while the application is running. When speaking of the *speedup* of an application, what is usually meant is that the wall clock time of that application has decreased. Speedup is often defined as the performance of the improved application relative to the old application for a constant problem size. Mathematically, this is equal to:

$$\text{Speedup} = \frac{t_{\text{old}}}{t_{\text{new}}}, \tag{5.1}$$

in which t_{old} and t_{new} are the wall clock time needed to solve the same problem by the old application and improved application respectively. In the context of DALES, a constant problem size means that the size and resolution of the computational grid stay constant and that the same physical schemes are used.

Strong scaling

A parallel application may still have sections that have to be executed sequentially and therefore do not benefit from the utilization of more processors. Hence, the speedup that an application obtains from parallelization is ultimately limited by the work that has to be executed serially. This limitation was first quantified by Amdahl (1967) in what is now known as Amdahl's law:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}} \tag{5.2}$$

in which f denotes the fraction of the application that can be run in parallel, and N is the number of processors. To illustrate the significance of Amdahl's law, assume an application that can be fully parallelized. In this case, the fraction f is equal to 1. Notice that under the assumption that $f = 1$, Equation 5.2 reduces to: Speedup = N . In other

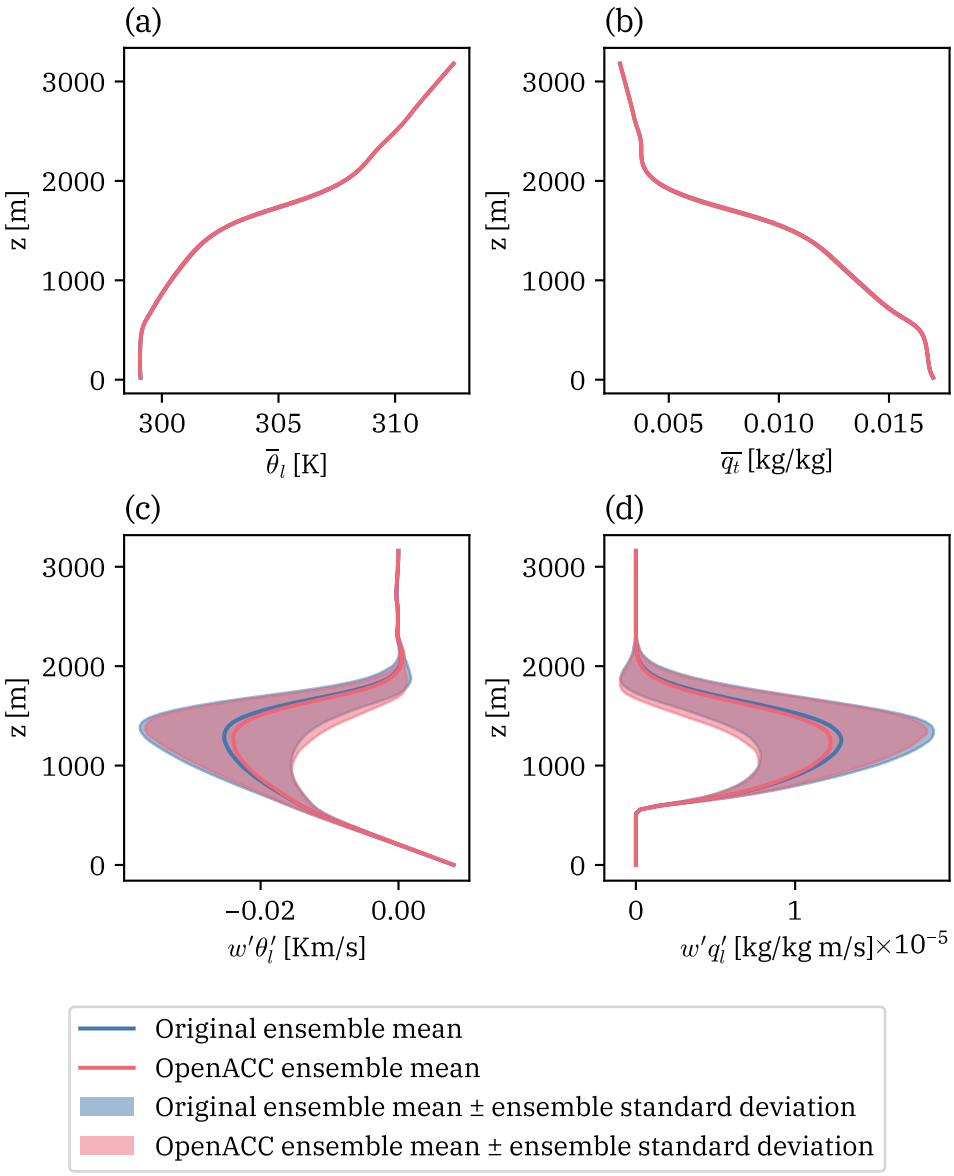


Figure 5.1: Ensemble mean vertical profiles and standard deviation of (a) liquid water potential temperature, (b) total water specific humidity, (c) total liquid water potential temperature flux and (d) total moisture flux.

words: if the application is executed on, for example, 512 processors, it theoretically runs 512 times faster compared to a single processor. Now, assume that only 99% of the application can be parallelized. Using $f = 0.99$, it follows from Equation 5.2 that the application only runs 83.8 times faster when using 512 processors. Therefore, to create an efficient parallel application, the sequential portion of the application should be kept as small as possible. The speedup of an application as a function of the number of processors is also called *strong scaling*. To measure the strong scaling performance of an application, the problem size (i.e., the number of grid points) is kept constant and the number of processors is increased. The strong scaling performance helps to determine the optimal amount of resources to use for an application.

Weak scaling

Gustafson (1988) voiced some skepticism regarding Amdahl's law. He stated that it is not realistic to assume that the problem size stays constant when more processors are available for the task. Instead, it should be assumed that the problem size tends to increase with the number of processors. Gustafson proposed that the speedup should be calculated as:

$$\text{Speedup} = (1 - f) + f \times N. \tag{5.3}$$

Now, for the same parallel fraction as above ($f = 0.99$), using 512 processors would result in a speedup of 506.89. This is a much more promising number than the upper bound of 83.3 obtained from Equation 5.2. The performance of a parallel application in the context of Gustafson's law is called *weak scaling*. Weak scaling performance is a useful metric to assess whether an application can efficiently tackle larger problem sizes if more computational resources are available.

5.4. System configuration

Performance testing of DALES was done on three systems: a high-end desktop workstation, and the Snellius supercomputer. On Snellius, DALES was tested on both a CPU-only node and a node containing GPUs. The specifications of all tested systems can be found in Table 5.1 and an impression of a CPU and GPU node of Snellius can be found in Figures 5.2 and 5.3, respectively. On Snellius, instead of comparing a single GPU to a single (multicore) CPU, a more fair comparison is to compare one GPU to a full CPU node which contains two (multicore) CPUs. The reason for this is that on supercomputers like Snellius, accounting of the used resources happens through System Billing Units (SBUs). When a proposal for a certain project is granted computational resources, an amount of SBUs is obtained. As computing resources are being used, SBUs are subtracted from the granted budget depending on the type and amount of systems and the run time of the application. A single A100 GPU and a full CPU node both cost 128 SBU per hour (SURF, n.d.-b).

Table 5.1: Specification of the computer systems used for performance testing of the CPU and GPU versions of DALES.

	Desktop system	Snellius CPU node	Snellius GPU node
CPU	Intel Core i9 11900K	AMD EPYC 7H12	Intel Xeon Platinum 8360Y
CPU clock speed	3.50 GHz (5.40 GHz Turbo)	2.60 GHz (3.30 GHz Turbo)	2.40 GHz (3.50 GHz Turbo)
CPU cores	8	64 per socket (2 sockets per node)	36 per socket (2 sockets per node)
Memory	64 GB	256 GB	512 GB
GPU	NVIDIA RTX 3090	-	NVIDIA A100 SXM (4 GPUs per node)
GPU memory	24 GB	-	40 GB

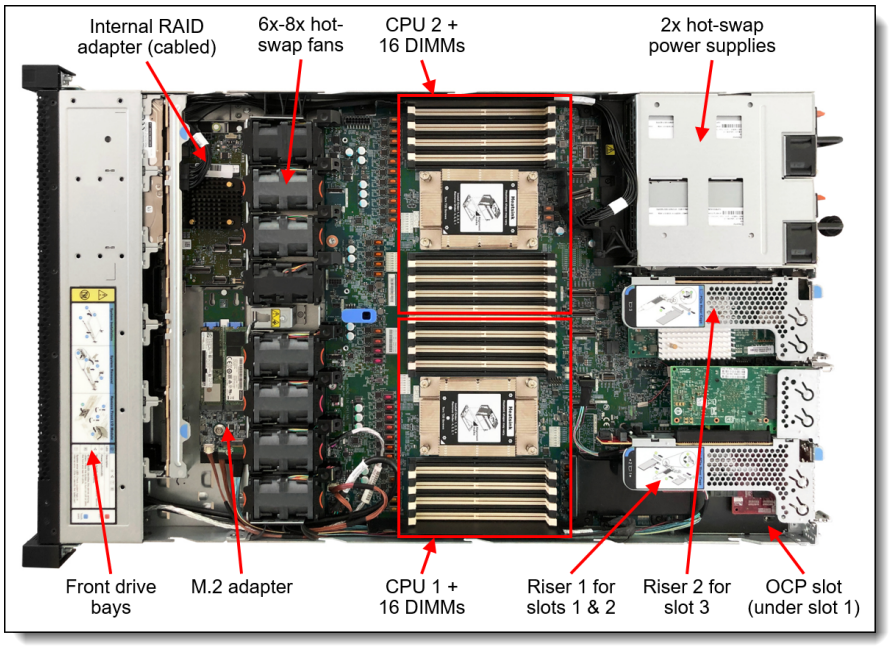


Figure 5.2: Snellius CPU node (Watts, 2024b).

5.5. Speedup

Wall-clock time and speedup of the GPU-accelerated DALES were measured on both the desktop system and the Snellius supercomputer. This was done for multiple horizontal grid configurations, starting from 128 × 128 grid points up to 1024 × 512 on the desktop system and 1024 × 1024 grid points on Snellius. Each run was limited to 100 time steps with a fixed time step size, and only the wall-clock time of the time loop itself (i.e., excluding the startup phase of the model) was measured.

Figure 5.4 shows the wall-clock time and speedup as measured on the desktop system. Speedup was calculated according to Equation 5.1, where t_{old} denotes the CPU run and t_{new} denotes the GPU run, and was only considered for the multi-CPU case. Speedup was found to increase with domain size until the maximum domain size of 1024 × 512 grid points was reached, where it reached a value of 11.6. The trend indicates that the overhead related to GPU execution (sending data, sending instructions) is relatively large for small problem sizes, but decreases with problem size.

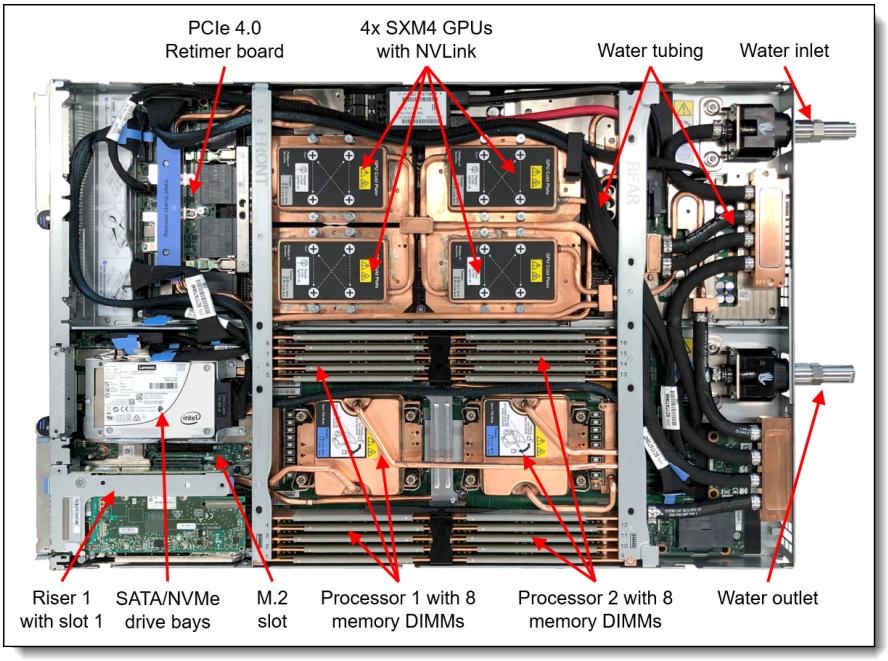


Figure 5.3: Snellius GPU node (Watts, 2024a).

As discussed in section 5.4, on Snellius, a single GPU was compared to a full CPU node, which hosts two 64-core CPUs (Table 5.1). Because these configurations cost the same amount of SBUs per hour, "speedup" can also be interpreted as "cost savings" in this case. Similar to the desktop system, speedup increased with domain size. The maximum speedup was about 3.9. The absolute value of the speedup was lower compared to the desktop system, as the Snellius CPU node features more CPU cores in total.

Comparing Figure 5.4 to Figure 5.5, it can be seen that the A100 GPU in the Snellius system only marginally outperformed the RTX 3090 GPU. This is an interesting observation, as the A100 offers more than 17 times the performance in double-precision calculations compared to the RTX 3090 (NVIDIA, 2020, 2021). This indicates that the main factor preventing speedup is not the amount of cores in a GPU. Jansson et al. (2023) mention that DALES was found to be limited by memory bandwidth. In other words, DALES performs relatively few arithmetic operations to the amount of data it has to read from and write to memory, a measure which is also referred to as *arithmetic intensity*. The memory bandwidth of the A100 is only about 1.7 times higher than that of the RTX 3090, which explains why it is only marginally faster.

Interestingly, a single desktop CPU significantly outperformed a single CPU of a Snellius CPU node. This can be explained by the fact that the Core i9 processor in the desktop system has a higher turbo clock speed. CPU clock speed denotes how many instructions a CPU can process in a second (Rauber & Rünger, 2023). In the context of scientific computing, a higher clock speed generally leads to more computations per second but also increases power usage and heat generation by the CPU, which is why the turbo clock speed cannot be sustained for long periods. Overheating is generally not a problem when only a single core of the CPU is performing at the turbo clock speed, however, from which the desktop CPU can benefit in this test.

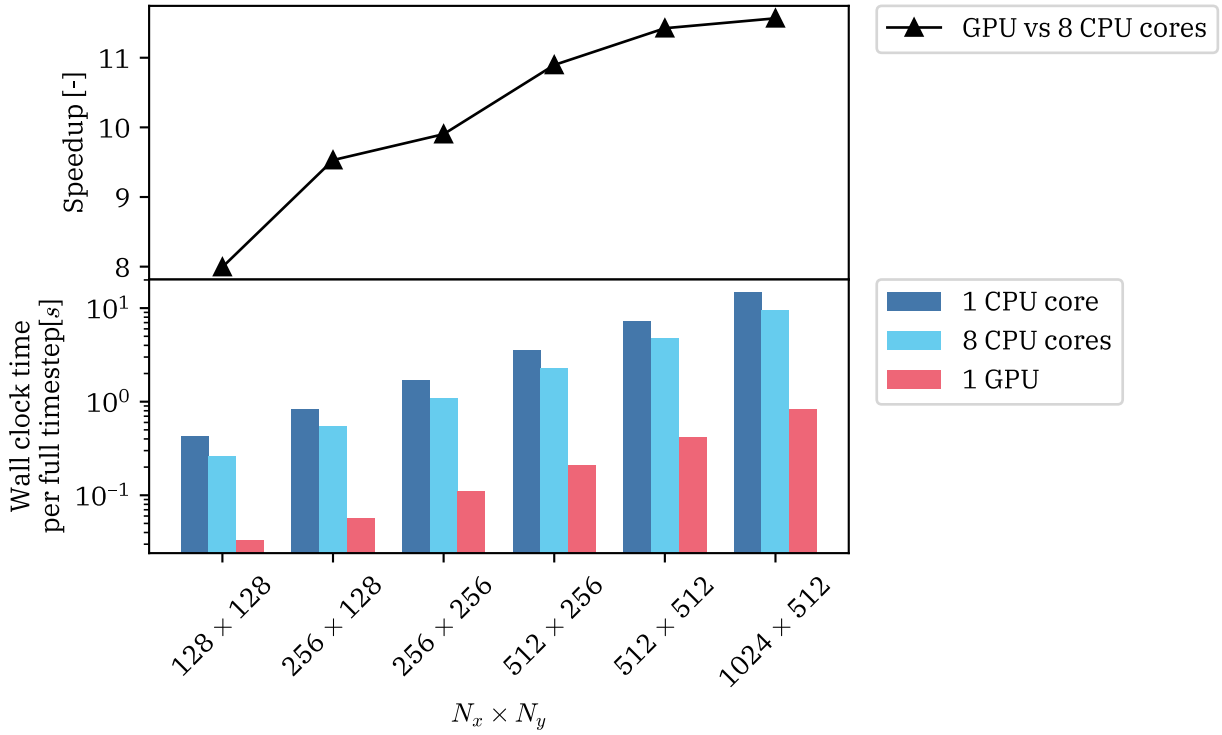


Figure 5.4: Speedup for different horizontal grid sizes on the desktop system.

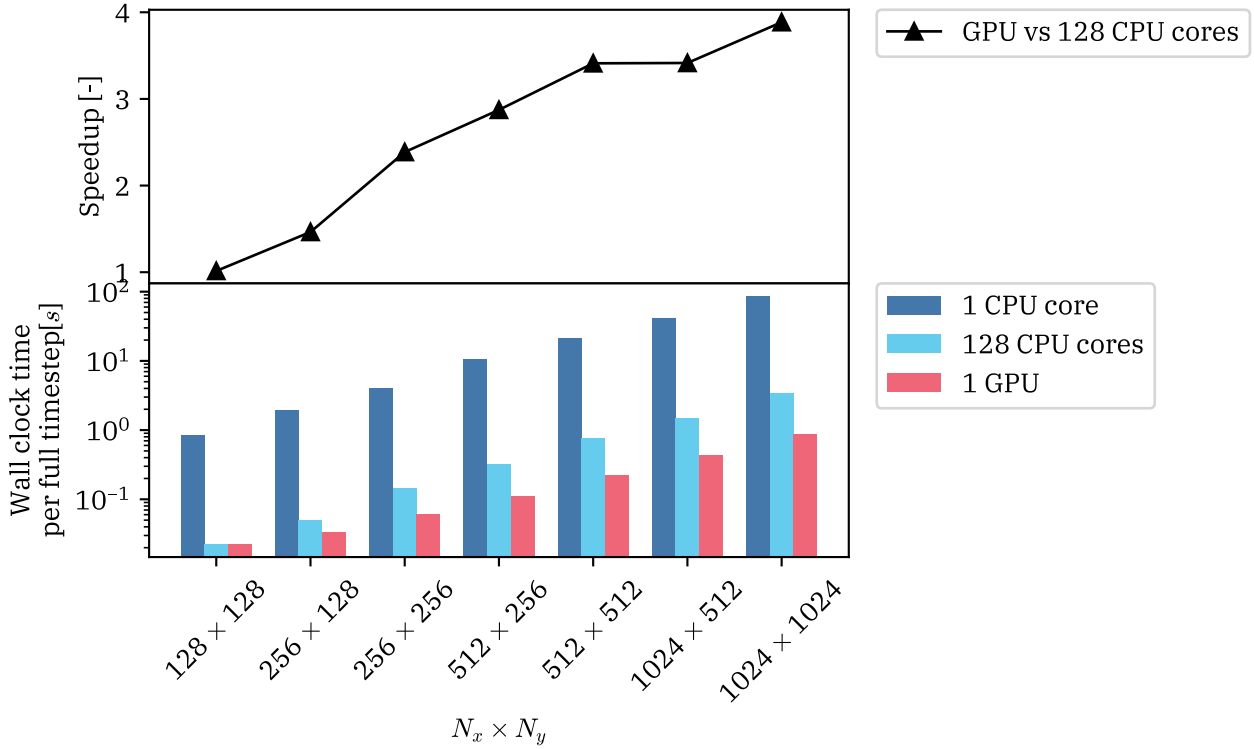


Figure 5.5: Speedup for different horizontal grid sizes on Snellius.

5.6. Scaling

Strong and weak scaling tests were performed exclusively on Snellius, as the desktop system only features a single GPU. For strong scaling, a horizontal grid size of 1024 × 1024 points was used, which nearly fully utilized the available 40Gb of memory of a single NVIDIA A100 GPU. Then, while keeping the domain size constant, the number of GPUs was doubled for each subsequent simulation until a maximum of 64 GPUs was reached. The task of choosing a suitable domain decomposition was left to MPI, leading to a slab decomposition along the y-direction in all cases. The wall-clock time of the time loop itself (i.e., excluding the start-up phase of the model) was measured during the first 100 time steps and averaged. Figure 5.6 shows the average wall-clock time per timestep as a function of the number of GPUs used. It can be seen that up to four GPUs, DALES scales nearly linear, meaning that at four GPUs the simulation is almost four times as fast than when using a single GPU. At eight GPUs, the wall-clock time is greater than when using four GPUs. This decrease in performance is expected, as the eight GPUs are spread over two nodes and data therefore has to traverse the inter-node interconnection network. This network is significantly slower than the NVLink interconnection between the four GPUs *within* a node.

To measure weak scaling performance, a horizontal domain size of 1024 × 1024 grid points with a grid spacing of 100 m in both directions was used per GPU. Next, the number of GPUs was doubled per simulation, therefore also doubling the total domain size. The domain was extended alternating in the x and y direction. Again, the wall-clock time of the time loop was averaged over the first 100 times steps. For this test, both a slab decomposition along the y-direction and a pencil decomposition of the domain were tested. In Figure 5.7, the ratio of the wall-clock time measured for multi-GPU runs to the single-GPU run is plotted. A wall-clock time ratio greater than one indicates a slowdown. In the ideal case, indicated by the horizontal black line, the runtime would not increase with the number of GPUs as the workload per GPU stays constant. In reality, however, inter-GPU communication introduces a performance penalty that can grow quite significant. From Figure 5.7, it can be seen that the wall-clock time remained near constant up to four GPUs, likely due to the availability of the fast NVLink connection between GPUs. From eight GPUs onward, the wall-clock time increases with the number of GPUs. This is expected, as the amount of data that has to be communicated also increases with the number of GPU. Finally, the pencil decomposition showed strikingly worse performance than the slab decomposition, which also worsened with the number of GPUs. This could be explained by the fact that in the setting of a slab decomposition, each GPU contains the full x and z directions of the domain, thus requiring two fewer collective inter-GPU communications in the Poisson solver.

Figure 5.8 provides further evidence for the hypothesis that communications hinder performance in simulations with multiple nodes. Here, the speedup of using 2 nodes (8 GPUs) over 1 node (4 GPUs) for different computationally expensive modules of DALES is plotted. The grid size was equal for both simulations. It can be seen that the advection, thermodynamics and subgrid modules experienced a speedup that was close to the ideal value of 2. The Poisson equation solver, which contains the global transpose operations as described in section 4.3 and therefore heavily relies on communication between GPUs, showed a significant *slowdown* compared to the single-GPU case. The bar labeled “Other” contains, among other things, the routines that manage the exchange of data at the boundaries of the MPI sub-domains.

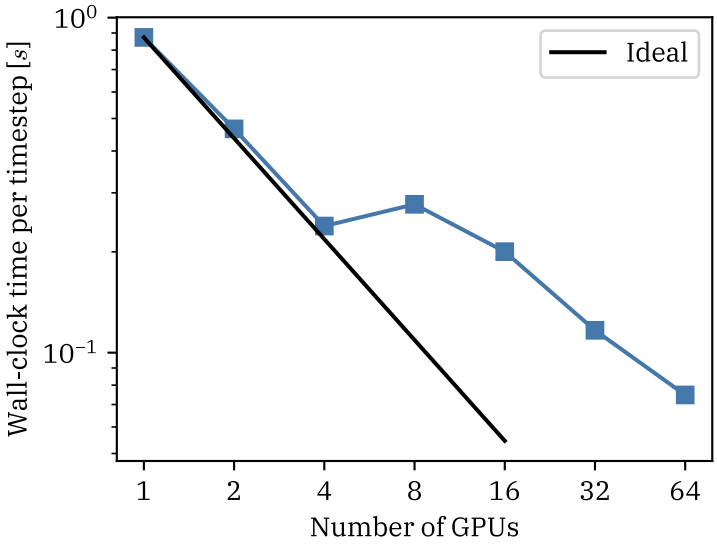


Figure 5.6: Strong scaling on Snellius for a horizontal domain size of 1024 × 1024 grid points.

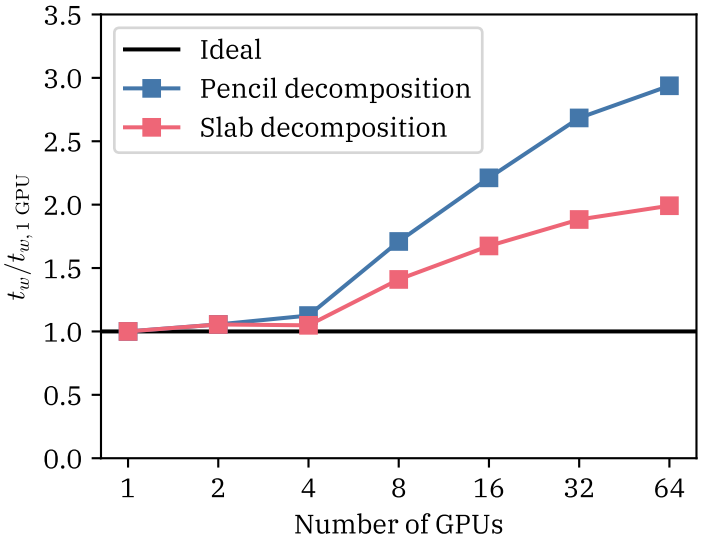


Figure 5.7: Weak scaling on Snellius for a horizontal domain size of 1024 × 1024 grid points per GPU

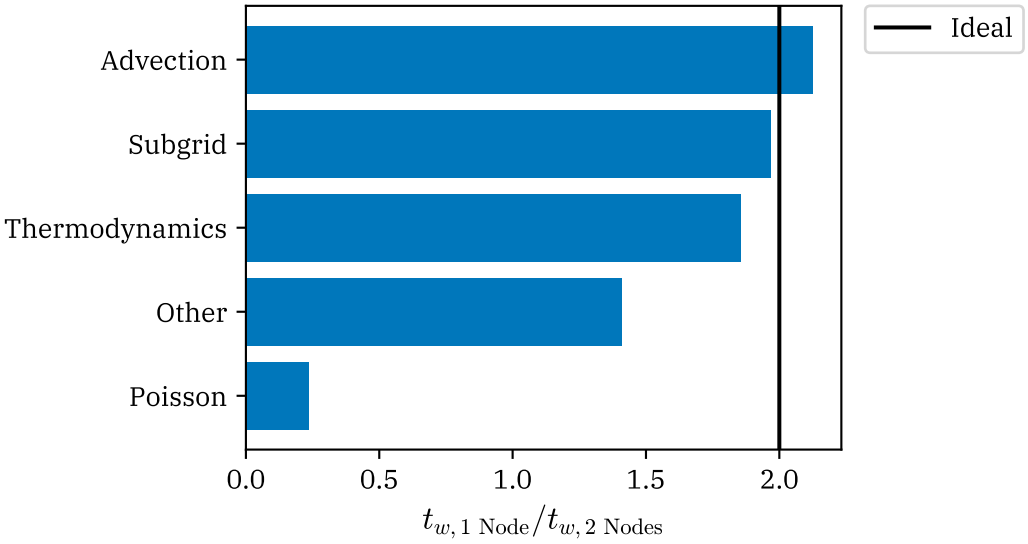


Figure 5.8: Speedup of the computationally most expensive modules of DALES when using 2 nodes (8 GPUs) versus 1 node (4 GPUs) GPUs on a horizontal domain of 1024 × 1024 grid points.

5.7. Single precision calculations

DALES offers the option to store the prognostic variables as single-precision floating point numbers (Jansson et al., 2023). As a single-precision floating point number consists of half as many bits as a double-precision floating point number (the default precision in DALES), it can be read from and written to memory much quicker, offering possibly a very significant performance enhancement for a memory bandwidth-bound application like DALES. Moreover, GPUs, specifically GPUs that are produced for the consumer market, often contain significantly more cores for single-precision calculations than for double-precision calculations. For example, the NVIDIA RTX 3090 GPU present in the tested desktop system (Table 5.1) features just two double-precision cores per SM for a total of 164 cores, versus 128 single-precision cores per SM for a total of 10,496 cores (NVIDIA, 2021). Therefore, switching from double-precision calculations to single-precision calculations should, in theory, lead to a 64-fold speedup on the RTX 3090.

To assess whether or not the usage of single-precision for the prognostic variables leads to any significant performance increase, DALES was compiled with single-precision enabled and the simulations described in section 5.5 were repeated on the GPU of the desktop system. The speedup of the single-precision simulations over the double-precision simulations can be found in Figure 5.9 for different horizontal grid configurations. Interestingly enough, single-precision simulations were only found to be slightly faster than double-precision simulations.

Why did the usage of single-precision floating-point numbers for the prognostic variables not lead to a speedup of 64 compared to double-precision? First, such a speedup can only arise in the absence of any memory considerations. As has been mentioned several times, DALES is a memory-bound application and a speedup of the order of 64 was therefore already highly unlikely. A closer inspection of the source code of DALES revealed another issue. Most Fortran compilers, including the GFortran and NVFortran compilers used in this work, use single-precision floating-point numbers for any real-valued variable (e.g. velocity, temperature, grid spacing, et cetera). For compatibility reasons, however, DALES

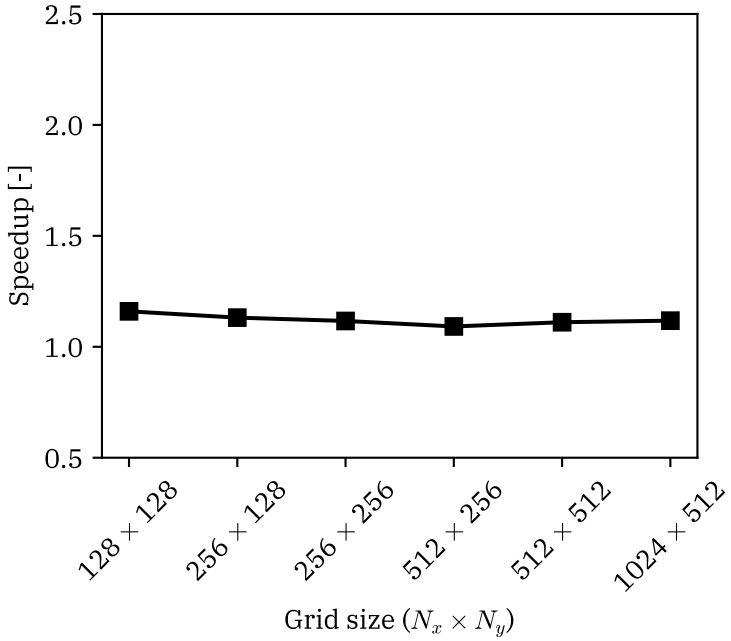


Figure 5.9: Obtained speedup when using single-precision floating-point numbers for the prognostic variables.

has to be compiled with the option `-fdefaultreal8` (GFortran) or `-Mr8` (NVFortran), which increases the default precision of real values to double precision. This means that all calculations in DALES are done in double precision. When the single-precision option for prognostic variables as described by Jansson et al. (2023) is used, only the prognostic variables are stored in single-precision format. All other real-valued variables and parameters are nevertheless compiled with double precision. In Fortran, when a single-precision variable encounters a double-precision variable in a calculation, it has to be temporarily converted to a double-precision floating-point number for the computer to be able to execute the calculation. This means that many of the calculations in DALES are executed in double precision, regardless of the precision used for the prognostic variables. For this reason, the single-precision cores of the RTX 3090 GPU were still not utilized in this test. The observed speedup in Figure 5.9 can therefore be attributed to savings in memory bandwidth by the reduced precision of the prognostic variables.

6

Conclusions and Recommendations

6.1. Conclusions

The objective of this thesis was to accelerate the DALES model by using GPUs. This has been achieved through the use of the OpenACC GPU programming model. The scope of the research was limited to the BOMEX model-intercomparison case which uses a limited subset of the physical schemes offered by DALES. Validation of the offloaded code was done using ensembles. For both the unmodified and GPU-accelerated versions of DALES, fifty-member ensembles were constructed and compared. The ensemble mean and standard deviation of the GPU-accelerated version of DALES did not show any significant deviations from the original version.

On a high-end desktop workstation, featuring an 8-core CPU and an NVIDIA RTX 3090 GPU, DALES was found to be up to 11.6 times faster when running on the GPU compared to the 8 CPU cores. On the Snellius supercomputer, the wall-clock time of DALES running on a single NVIDIA A100 GPU was compared to that on a full CPU node, because these two configurations cost the same number of SBUs per hour. The GPU was up to 3.9 times faster than the CPU node.

One of the goals of this thesis was to enable DALES to make use of multiple GPUs for a single simulation, which was achieved through the use of GPU-aware MPI. The implementation was tested on the Snellius supercomputer. Strong scaling tests showed near-linear strong scaling up to 4 GPUs. From 8 GPUs onward, diminishing returns were observed. This performance drop is most likely caused by the relatively slow interconnection between nodes. Weak scaling tests show that multiple GPUs can be used relatively efficiently for large domain simulations. Up to 4 GPUs, near-linear weak scaling was observed. Similar to the strong scaling tests, a performance decrease was observed going from a single node to multiple nodes.

In general, the performance of DALES seemed to be limited by memory bandwidth. This is supported by the observation that the NVIDIA A100 GPU, as present in Snellius GPU nodes, only slightly outperformed the NVIDIA RTX 3090 GPU of the desktop workstation, despite the A100 offering 17 times the theoretical computational performance. The use of single-precision floating-point numbers for the prognostic variables did not yield a significant speedup either, because most of the calculations in DALES are still executed in double precision.

6.2. Recommendations

6.2.1. Accelerating more components

Since only a limited portion of DALES was accelerated in this work, a clear direction for future work is to add OpenACC directives to other components of DALES. For example, the Cloud Botany case as described by Jansson et al. (2023) requires, in addition to the components used by the BOMEX case, a radiation scheme and a precipitation scheme. Offloading more components of the model to the GPU will accelerate these more realistic cases, and can, through longer simulations, bigger domains and/or bigger ensembles, help with the development of new scientific insights.

6.2.2. Performance tuning

For improving performance when using a large number of GPUs ($\mathcal{O}(10^2)$ to $\mathcal{O}(10^3)$), the use of a specialized domain decomposition library could be explored. Romero, Costa, and Fatica (2022) developed such a library, called cuDecomp. cuDecomp automatically chooses the most efficient domain decomposition and inter-GPU communication backend at runtime. This is done through the same transposition algorithm as encountered in the Poisson solver of DALES. While the slab decomposition outperformed the pencil decomposition in all cases of the weak scaling benchmarks of DALES, Romero et al. (2022) found, for an application similar to DALES, that the slab decomposition is not necessarily the most efficient option when hundreds of GPUs are used. Moreover, Romero et al. (2022) found that MPI is not always the most efficient backend for inter-GPU communication between high numbers of GPUs.

As DALES appears to be severely memory bandwidth bound, the model's performance could be improved by attempting to reduce memory usage. In practice, this can be done by storing fewer variables and computing more "on the fly". This strategy will, however, only yield a performance benefit in cases where the additional computations save on memory operations (loading and storing data).

GPUs are well suited for calculations in single precision, especially consumer-grade GPUs targeted for computer gaming like the RTX 3090 used in this work. Since the current GPU port of DALES is not yet fully configured for using single-precision floating-point numbers, performance can still be gained in this regard. Full single-precision support can be achieved by converting more variables to variable precision, as is the case for the prognostic variables, such that no conversion to double-precision has to be done at runtime. During this process, care has to be taken to make sure that the loss of precision does not result in any numerical instabilities.

6.2.3. Exploring portability

The current GPU-implementation of DALES has been tested exclusively on NVIDIA GPUs. However, newer supercomputers are starting to feature accelerators made by different vendors. For example, the Frontier and LUMI supercomputers both use GPUs made by AMD, while the GPUs of the upcoming Aurora supercomputer are supplied by Intel (Trader, 2021). Therefore, it would be worthwhile to examine how DALES performs on GPUs other than NVIDIA. While OpenACC code can be compiled for AMD GPUs, it cannot for Intel GPUs. One approach to reaching portability is to translate the OpenACC directives to OpenMP. While this strategy appears to be very involved, tools exist that can perform this translation automatically (Servat, 2022). With the use of such tools, the only task left to the developer is to validate the translated code.

Another component that requires extra care when exploring execution on GPUs from

other vendors is the FFT library. In this work, the cuFFT library has been used, which only supports execution on NVIDIA GPUs. As discussed in section 4.3, multiple other FFT libraries exist that support execution on AMD and/or Intel GPUs. Out of these libraries, rocFFT supports both NVIDIA and AMD GPUs and features bindings for the Fortran programming language. Furthermore, rocFFT has a similar interface as cuFFT and would therefore be relatively straightforward to implement in DALES. Alternatively, vkFFT could be used. vkFFT offers multiple benefits over rocFFT: it supports NVIDIA, AMD and Intel GPUs and features discrete cosine transforms for non-periodic boundary conditions.

OpenACC can also be compiled for multi-core CPUs. While DALES can already make use of multiple CPU cores via the existing MPI parallelization, switching to OpenACC may improve efficiency for multi-core CPU runs. The reasoning behind this line of thought is that OpenACC is a *shared memory* programming model, meaning that all CPU cores have access to the same memory space. In practice, this means that no domain decomposition is needed (unlike with MPI) to make use of multiple CPU cores, which also removes the need for communication between cores and thus can speed up simulations. For simulations with multiple CPUs and/or multiple nodes in a supercomputer, MPI would still be needed as only the cores *inside* a CPU can share memory. Once again, the FFT library requires some care, as it would also need to operate in a shared memory configuration. The performance of DALES on multi-core CPUs using OpenACC is particularly interesting for supercomputers that rely on massive amounts of CPUs for their computational power, such as the Fugaku supercomputer (Sato et al., 2020).

References

Aamondt, T. M., Wai Lun Fung, W., & Rogers, T. G. (2018). *General-Purpose Graphics Processing Architecture* (No. 44). Morgan & Claypool.

AMD. (2023a). *hipFFT.* Retrieved from `https://hipfft.readthedocs.io/en/rocm-5.7.0/`

AMD. (2023b). *rocFFT.* Retrieved from `https://rocfft.readthedocs.io/en/rocm-5.7.0/index.html`

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (p. 483). Atlantic City, New Jersey: ACM Press. doi: 10.1145/1465482.1465560

Antao, S. F., Bataev, A., Jacob, A. C., Bercea, G.-T., Eichenberger, A. E., Rokos, G., … O'Brien, K. (2016, November). Offloading Support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)* (pp. 1–11). Salt Lake City, UT, USA: IEEE. doi: 10.1109/LLVM-HPC.2016.006

Arakawa, A., & Lamb, V. R. (1977). Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. In *Methods in Computational Physics: Advances in Research and Applications* (Vol. 17, pp. 173–265). Elsevier. doi: 10.1016/B978-0-12-460817-7.50009-4

Bardina, J., Ferziger, J., & Reynolds, W. (1980, July). Improved subgrid-scale models for large-eddy simulation. In *13th Fluid and PlasmaDynamics Conference.* Snowmass,CO,U.S.A.: American Institute of Aeronautics and Astronautics. doi: 10.2514/6.1980-1357

Beare, R. J., Macvean, M. K., Holtslag, A. A. M., Cuxart, J., Esau, I., Golaz, J.-C., … Sullivan, P. (2006, February). An Intercomparison of Large-Eddy Simulations of the Stable Boundary Layer. *Boundary-Layer Meteorology*, *118*(2), 247–272. doi: 10.1007/s10546-004-2820-6

Brunton, S. L., Noack, B. R., & Koumoutsakos, P. (2020, January). Machine Learning for Fluid Mechanics. *Annual Review of Fluid Mechanics*, *52*(1), 477–508. doi: 10.1146/annurev-fluid-010719-060214

Böing, S. (2014). *The interaction between deep convective clouds and their environment* (Doctoral dissertation, [object Object]). doi: 10.4233/UUID:AA9E6037-B9CB-4EA0-9EB0-A47BF1DFC940

Choi, C. Q. (2022, June). *The Beating Hart of the World's First Exascale Supercomputer.* Retrieved from `https://spectrum.ieee.org/frontier-exascale-supercomputer`

Chorin, A. J. (1967). The numerical solution of the Navier-Stokes equations for an incompressible fluid. *Bulletin of the American Mathematical Society*, *73*(6), 928–931. doi: 10.1090/S0002-9904-1967-11853-6

Costa, P. (2018, October). A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows. *Computers & Mathematics with Applications*, *76*(8), 1853–1862. doi: 10.1016/j.camwa.2018.07.034

Costa, P., Phillips, E., Brandt, L., & Fatica, M. (2021, January). GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows. *Computers & Mathematics with Applications*, *81*, 502–511. doi: 10.1016/j.camwa.2020.01.002

Dally, W. J., Keckler, S. W., & Kirk, D. B. (2021, November). Evolution of the Graphics Processing Unit (GPU). *IEEE Micro*, *41*(6), 42–51. doi: 10.1109/MM.2021.3113475

Deardorff, J. W. (1974, August). Three-dimensional numerical study of the height and mean structure of a heated planetary boundary layer. *Boundary-Layer Meteorology*, *7*(1), 81–106. doi: 10.1007/BF00224974

Deardorff, J. W. (1980, June). Stratocumulus-capped mixed layers derived from a three-dimensional model. *Boundary-Layer Meteorology*, *18*(4), 495–527. doi: 10.1007/BF00119502

De Bruine, M., Krol, M., Vilà-Guerau De Arellano, J., & Röckmann, T. (2019, December). Explicit aerosol–cloud interactions in the Dutch Atmospheric Large-Eddy Simulation model DALES4.1-M7. *Geoscientific Model Development*, *12*(12), 5177–5196. doi: 10.5194/gmd-12-5177-2019

Farber, R. (2017). *Parallel Programming with OpenACC*. Elsevier.

Flynn, M. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, *54*(12), 1901–1909. doi: 10.1109/PROC.1966.5273

Frigo, M., & Johnson, S. (1998). FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)* (Vol. 3, pp. 1381–1384). Seattle, WA, USA: IEEE. doi: 10.1109/ICASSP.1998.681704

Gustafson, J. L. (1988, May). Reevaluating Amdahl's law. *Communications of the ACM*, *31*(5), 532–533. doi: 10.1145/42411.42415

Heus, T., Van Heerwaarden, C. C., Jonker, H. J. J., Pier Siebesma, A., Axelsen, S., Van Den Dries, K., ... Vilà -Guerau De Arellano, J. (2010, September). Formulation of the Dutch Atmospheric Large-Eddy Simulation (DALES) and overview of its applications. *Geoscientific Model Development*, *3*(2), 415–444. doi: 10.5194/gmd-3-415-2010

Hockney, R. W. (1965, January). A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the ACM*, *12*(1), 95–113. doi: 10.1145/321250.321259

Holland, J. Z., & Rasmusson, E. M. (1973, January). Measurements of the Atmospheric Mass, Energy, and Momentum Budgets Over a 500-Kilometer Square of Tropical Ocean. *Monthly Weather Review*, *101*, 44. doi: 10.1175/1520-0493(1973)101<0044:MOTAME>2.3.CO;2

Hundsdorfer, W., Koren, B., vanLoon, M., & Verwer, J. (1995, March). A Positive Finite-Difference Advection Scheme. *Journal of Computational Physics*, *117*(1), 35–46. doi: 10.1006/jcph.1995.1042

Jansson, F., Janssens, M., Grönqvist, J. H., Siebesma, A. P., Glassmeier, F., Attema, J., ... Kölling, T. (2023, November). Cloud Botany: Shallow Cumulus Clouds in an Ensemble of Idealized Large-Domain Large-Eddy Simulations of the Trades. *Journal of Advances in Modeling Earth Systems*, *15*(11), e2023MS003796. doi: 10.1029/2023MS003796

Jansson, F., van den Oord, G., Pelupessy, I., Grönqvist, J. H., Siebesma, A. P., & Crommelin, D. (2019, September). Regional Superparameterization in a Global Circulation Model Using Large Eddy Simulations. *Journal of Advances in Modeling Earth Systems*, *11*(9), 2958–2979. doi: 10.1029/2018MS001600

Kedward, L. J., Aradi, B., Certik, O., Curcic, M., Ehlert, S., Engel, P., ... Vandenplas, J. (2022, March). The State of Fortran. *Computing in Science & Engineering*, *24*(2), 63–72. doi: 10.1109/MCSE.2022.3159862

Khronos Group. (2023, October). *SYCL 2020 Specification (revision 8)* (Tech. Rep.). Retrieved from `https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf`

Larsen, E. S., & McAllister, D. (2001, November). Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (pp. 55–55). Denver Colorado: ACM. doi: 10.1145/582034.582089

Leonard, A. (1975). Energy Cascade in Large-Eddy Simulations of Turbulent Fluid Flows. In *Advances in Geophysics* (Vol. 18, pp. 237–248). Elsevier. doi: 10.1016/S0065-2687(08)60464-1

Lilly, D. K. (1967). The representation of small-scale turbulence in numerical simulation experiments. In *Proc. IBM sci. Comput. Symp. on environmental science* (pp. 195–210).

Liqui Lung, F. P. A., Jakob, C., Siebesma, A. P., & Jansson, F. R. (2023, November). *Open Boundary Conditions for Atmospheric Large Eddy Simulations and the Implementation in DALES4.4* (Preprint). Atmospheric sciences. doi: 10.5194/gmd-2023-196

Lorenz, E. N. (1963, January). Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, *20*(2), 130–141. doi: 10.1175/1520-0469(1963)020%3C0130:DNF%3E2.0.CO;2

Mittal, S., & Vaishay, S. (2019, October). A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture*, *99*, 101635. doi: 10.1016/j.sysarc.2019.101635

Niemeyer, K. E., & Sung, C.-J. (2014, February). Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing*, *67*(2), 528–564. doi: 10.1007/s11227-013-1015-7

Nvidia. (n.d.). *cuFFT.* Retrieved from `https://docs.nvidia.com/cuda/cufft/index.html`

NVIDIA. (2020). *NVIDIA A100 Tensor Core GPU Overview* (Tech. Rep.). Retrieved from `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`

NVIDIA. (2021). *NVIDIA Ampere GA102 GPU Architecture* (Tech. Rep.). Retrieved from `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf`

Ouwersloot, H. G., Moene, A. F., Attema, J. J., & De Arellano, J. V.-G. (2017, January). Large-Eddy Simulation Comparison of Neutral Flow Over a Canopy: Sensitivities to Physical and Numerical Conditions, and Similarity to Other Representations. *Boundary-Layer Meteorology*, *162*(1), 71–89. doi: 10.1007/s10546-016-0182-5

Pope, S. B. (2000). *Turbulent flows*. Cambridge ; New York: Cambridge University Press.

Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., & Panda, D. K. (2013, October). Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing* (pp. 80–89). Lyon, France: IEEE. doi: 10.1109/ICPP.2013.17

Rauber, T., & Rünger, G. (2023). *Parallel programming: For multicore and cluster systems* (Third edition ed.). Cham: Springer.

Rodriguez, S. (2019). LES and DNS turbulence modeling. In *Applied computational fluid dynamics and turbulence modeling: Practical tools, tips and techniques* (pp. 197–223). Cham: Springer International Publishing. doi: 10.1007/978-3-030-28691-0_5

Romero, J., Costa, P., & Fatica, M. (2022, July). Distributed-memory simulations of turbulent flows on modern GPU systems using an adaptive pencil decomposition library. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (pp. 1–11). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3539781.3539797

Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., ... Shimizu, T. (2020, November). Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: Inter-*

national Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1–15). Atlanta, GA, USA: IEEE. doi: 10.1109/SC41405.2020.00051

Schalkwijk, J., Griffith, E. J., Post, F. H., & Jonker, H. J. J. (2012, March). High-Performance Simulations of Turbulent Clouds on a Desktop PC: Exploiting the GPU. *Bulletin of the American Meteorological Society*, *93*(3), 307–314. doi: 10.1175/BAMS-D-11-00059.1

Schalkwijk, J., Jonker, H. J. J., Siebesma, A. P., & Van Meijgaard, E. (2015, May). Weather Forecasting Using GPU-Based Large-Eddy Simulations. *Bulletin of the American Meteorological Society*, *96*(5), 715–723. doi: 10.1175/BAMS-D-14-00114.1

Schär, C., Fuhrer, O., Arteaga, A., Ban, N., Charpilloz, C., Di Girolamo, S., ... Wernli, H. (2020, May). Kilometer-Scale Climate Models: Prospects and Challenges . *Bulletin of the American Meteorological Society*, *101*(5), E567-E587. doi: 10.1175/BAMS-D-18-0167.1

Schumann, U., & Sweet, R. A. (1988, March). Fast Fourier transforms for direct solution of poisson's equation with staggered boundary conditions. *Journal of Computational Physics*, *75*(1), 123–137. doi: 10.1016/0021-9991(88)90102-7

Servat, H. (2022). *Intel Application Migration Tool for OpenACC* to OpenMP * API.* Retrieved from `https://github.com/intel/intel-application-migration-tool-for-openacc-to-openmp`

Siebesma, A. P., Bretherton, C. S., Brown, A., Chlond, A., Cuxart, J., Duynkerke, P. G., ... Stevens, D. E. (2003, May). A Large Eddy Simulation Intercomparison Study of Shallow Cumulus Convection. *Journal of the Atmospheric Sciences*, *60*(10), 1201–1219. doi: 10.1175/1520-0469(2003)60<1201:ALESIS>2.0.CO;2

Siebesma, A. P., & Cuijpers, J. W. M. (1995, March). Evaluation of Parametric Assumptions for Shallow Cumulus Convection. *Journal of the Atmospheric Sciences*, *52*(6), 650–666. doi: 10.1175/1520-0469(1995)052%3C0650:EOPAFS%3E2.0.CO;2

Slingo, J., & Palmer, T. (2011, December). Uncertainty in weather and climate prediction. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *369*(1956), 4751–4767. doi: 10.1098/rsta.2011.0161

Smagorinsky, J. (1963, January). General Circulation Experiments with the Primitive Equations . *Monthly Weather Review*, *91*, 99–164. doi: 10.1175/1520-0493(1963)091%3C0099:GCEWTP%3E2.3.CO;2

Stull, R. B. (1988). *An Introduction to Boundary Layer Meteorology*. Dordrecht: Kluwer Academic Publishers.

SURF. (n.d.-a). *Snellius hardware and file systems.* Retrieved from `https://servicedesk.surf.nl/wiki/display/WIKI/Snellius+hardware+and+file+systems`

SURF. (n.d.-b). *Snellius partitions and accounting.* Retrieved from `https://servicedesk.surf.nl/wiki/display/WIKI/Snellius+partitions+and+accounting`

Swarztrauber, P. N. (1982). Vectorizing the FFTs. In *Parallel Computations* (pp. 51–83). Elsevier. doi: 10.1016/B978-0-12-592101-5.50007-5

Tolmachev, D. (2023). VkFFT-A Performant, Cross-Platform and Open-Source GPU FFT Library. *IEEE Access*, *11*, 12039–12058. doi: 10.1109/ACCESS.2023.3242240

TOP500. (2023). *TOP500 Lists.* Retrieved from `https://www.top500.org/lists/top500/`

Trader, T. (2021, September). *How Argonne is Preparing for Exascale in 2022.* Retrieved from `https://www.hpcwire.com/2021/09/08/how-argonne-is-preparing-for-exascale-in-2022/`

Trott, C. R., Lebrun-Grandie, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., ... Wilke, J. (2022, April). Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 805–817. doi: 10.1109/TPDS.2021.3097283

Van Den Oord, G., Jansson, F., Pelupessy, I., Chertova, M., Grönqvist, J. H., Siebesma, P., & Crommelin, D. (2020, July). A Python interface to the Dutch Atmospheric Large-Eddy Simulation. *SoftwareX*, *12*, 100608. doi: 10.1016/j.softx.2020.100608

Van Der Dussen, J. J., De Roode, S. R., Ackerman, A. S., Blossey, P. N., Bretherton, C. S., Kurowski, M. J., ... Siebesma, A. P. (2013, July). The GASS/EUCLIPSE model intercomparison of the stratocumulus transition as observed during ASTEX: LES results. *Journal of Advances in Modeling Earth Systems*, *5*(3), 483–499. doi: 10.1002/jame.20033

Van Heerwaarden, C. C., Van Stratum, B. J. H., Heus, T., Gibbs, J. A., Fedorovich, E., & Mellado, J. P. (2017, August). MicroHH 1.0: A computational fluid dynamics code for direct numerical simulation and large-eddy simulation of atmospheric boundary layer flows. *Geoscientific Model Development*, *10*(8), 3145–3165. doi: 10.5194/gmd-10-3145-2017

Verzijlbergh, R. (2021). Atmospheric flows in large wind farms. *Europhysics News*, *52*(5), 20–23. doi: 10.1051/epn/2021502

Vilà-Guerau De Arellano, J., Patton, E. G., Karl, T., Van Den Dries, K., Barth, M. C., & Orlando, J. J. (2011, April). The role of boundary layer dynamics on the diurnal evolution of isoprene and the hydroxyl radical over tropical forests. *Journal of Geophysical Research*, *116*(D7), D07304. doi: 10.1029/2010JD014857

Vreman, A. W. (2004, October). An eddy-viscosity subgrid-scale model for turbulent shear flow: Algebraic theory and applications. *Physics of Fluids*, *16*(10), 3670–3681. doi: 10.1063/1.1785131

Watts, D. (2024a, January). *Lenovo ThinkSystem SD650-N V2 Server* (Tech. Rep.). Lenovo. Retrieved from https://lenovopress.lenovo.com/lp1396-thinksystem-sd650-n-v2-server

Watts, D. (2024b, March). *Lenovo ThinkSystem SR645 Server* (Tech. Rep.). Retrieved from https://lenovopress.lenovo.com/lp1280-thinksystem-sr645-server

Wyngaard, J. C. (2004, January). Toward Numerical Modeling in the "Terra Incognita". *Journal of the Atmospheric Sciences*, *61*(14), 1816–1826. doi: 10.1175/1520-0469(2004)061%3C1816:TNMITT%3E2.0.CO;2

Xu, Y., Liu, X., Cao, X., Huang, C., Liu, E., Qian, S., ... Zhang, J. (2021, November). Artificial intelligence: A powerful paradigm for scientific research. *The Innovation*, *2*(4), 100179. doi: 10.1016/j.xinn.2021.100179

A

Compiling and Running DALES on GPUs

A.1. Setting up the NVIDIA HPC SDK

First, an OpenACC compatible compiler is needed. As discussed, the NVIDIA HPC SDK was used in this work. The HPC SDK is bundled with the `nvfortran` compiler, the cuFFT library required for the Poisson solver and a GPU-aware MPI library. In In this work, version 23.3 was used, which can be installed as follows:

```
$ wget https://developer.download.nvidia.com/hpc-sdk/23.3/
    nvhpc_2023_233_Linux_x86_64_cuda_multi.tar.gz
$ tar xpzf nvhpc_2023_233_Linux_x86_64_cuda_multi.tar.gz
$ nvhpc_2023_233_Linux_x86_64_cuda_multi/install
```

This will install the SDK under the directory `/opt/nvidia/hpc_sdk`. The compilers and other libraries can be added to the `PATH` with the following script:

```
#!/bin/bash
YEAR=2023
NVHPC_INSTALL_DIR=/opt/nvidia/hpc_sdk
NVARCH=`uname -s`_`uname -m`; export NVARCH
NVCOMPILERS=$NVHPC_INSTALL_DIR; export NVCOMPILERS
MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/$YEAR/compilers/man; export MANPATH
PATH=$NVCOMPILERS/$NVARCH/$YEAR/compilers/bin:$PATH; export PATH
export PATH=$NVCOMPILERS/$NVARCH/$YEAR/comm_libs/mpi/bin:$PATH
export MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/$YEAR/comm_libs/mpi/man
export PATH=$NVCOMPILERS/$NVARCH/$YEAR/profilers/Nsight_Systems/bin:$PATH
export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/compilers/lib:
    $LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/comm_libs/mpi/lib:
    $LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$NVHPC_INSTALL_DIR/$NVARCH/$YEAR/math_libs/lib64:
    $LD_LIBRARY_PATH
```

A.2. Obtaining the DALES source code

The most up-to-date version of the OpenACC version of DALES can be found on GitHub (`https://github.com/dalesteam/dales`) under the branch `OpenACC`. Downloading is easily done through a Git Clone:

```
$ git clone -b OpenACC https://github.com/dalesteam/dales.git
```

For building DALES, NetCDF-Fortran and CMake are required, in addition to the NVIDIA compilers and libraries. Assuming that all requirements are met, DALES can be built as follows:

```
$ mkdir dales/build
$ cd dales/build
$ export SYST=NV-OpenACC
$ cmake ..
$ make -j
```

A.3. Running DALES on the GPU

The input files for the BOMEX case as used in this thesis are included in the DALES source code and can be found under `/cases/bomex`. To run the case on the GPU, two modifications to `namoptions.001` are needed: 1) under the section `&PHYSICS`, add the line `lfast_thermo=.true.` and 2) under the section `&SOLVER`, add the line `solver_id=200`.

Finally, DALES can be run using `mpirun`:

```
$ cd <path to dales>/cases/bomex/
$ mpirun -np N <path to dales>/build/src/dales4.4 namoptions.001
```

where `N` is the number of GPUs available on the system.