TU WIEN Informatics

Analysis of the GPU Acceleration Potential of the FFT-Based Pressure Solver in the PALM-4U Model System

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Computational Science and Engineering

eingereicht von

Stefanie North, BSc BSc
Matrikelnummer 51803598

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Mitwirkung: Dipl.-Ing. Dr.techn. Daniel Cornel

Wien, 1. Jänner 2001

Stefanie North	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Analysis of the GPU Acceleration Potential of the FFT-Based Pressure Solver in the PALM-4U Model System

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Science and Engineering

by

Stefanie North, BSc BSc
Registration Number 51803598

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Daniel Cornel

Vienna, January 1, 2001

Stefanie North	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefanie North, BSc BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Jänner 2001

Stefanie North

Danksagung

Diese Arbeit wurde durch das Kompetenzzentrum VRVis ermöglicht. Die VRVis GmbH wird im Rahmen von COMET – Competence Centers for Excellent Technologies (911654) durch BMIMI, BMWET, Land Tirol, Land Vorarlberg und Wirtschaftsagentur Wien – Ein Fonds der Stadt Wien gefördert. Das Programm COMET wird durch die FFG abgewickelt.

Acknowledgements

This work was enabled by the Competence Centre VRVis. The VRVis GmbH is funded by BMK, BMAW, Tyrol, Vorarlberg and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (911654) which is managed by FFG.

Kurzfassung

Der Klimawandel führt zu einer Zunahme der Häufigkeit und Schwere extremer Wetterereignisse. Dadurch werden die Lebensgrundlagen von Menschen sowie wichtige Infrastrukturen gefährdet. Entscheidungshilfetools können die Entwicklung klimaresistenter Städte unterstützen, indem sie während des Planungsprozesses Informationen über die potenzielle Wirksamkeit bestimmter Maßnahmen liefern. Besonders wirkungsvoll sind dabei Systeme, die genaue Mikroklimamodelle integrieren. PALM-4U, ein wissenschaftlich validiertes städtisches Klimamodell, eignet sich dafür, ist jedoch außerhalb der wissenschaftlichen Gemeinschaft nur eingeschränkt nutzbar, da es für den Betrieb auf HPC-Clustern optimiert ist. Mit dem Aufkommen leistungsstarker GPUs wird die Nutzung auf einzelnen Workstations zunehmend möglich.

Diese Studie erforscht das Potential einer Laufzeitbeschleunigung der Druckberechnung des PALM-4U Models mit der Verwendung von einer Grafikkarte und der Änderung der Computerarchitektur. Die Leistungssteigerung wird anhand von drei Parametern gemessen: *Geschwindigkeit*, *Gültigkeit* (über NMSE, R und FB) und *Speichereffizienz*. Außerdem wird der Einfluss auf die Laufzeit der vollständigen Simulation gemessen und mögliche Bottlenecks identifiziert. Schließlich wird das vollständige Modell analysiert, um die allgemeine Machbarkeit der GPU-Optimierung zu bewerten und Erkenntnisse für die zukünftige Entwicklung zu gewinnen.

In der Druckberechnung wird mithilfe der Fast Fourier Transform die 3D-Poissongleichung umgeformt, die resultierenden 1D-Gleichungen werden mithilfe des Thomas Algorithmus gelöst. Die Codestruktur wird verändert, CUDA-Kernels implementiert und die cuFFT-Bibliothek integriert. Außerdem wird ein Mixed-Precision-Ansatz getestet.

Die Single-Core-GPU-Implementierung erreicht eine Beschleunigung von bis zu Faktor 65,5 bei Single Precision und bis zu Faktor 49,3 bei Double Precision für große Domängrößen. Die Stabilität des Systems bleibt durch den Mixed-Precision-Ansatz unbeeinträchtigt, und es werden keine signifikanten Abweichungen zwischen FP32- und FP64-Läufen beobachtet. Nach 45×10^3 Simulationsschritten zeigen NMSE (0,02), FB (-0,017) und R (0,96) eine stabile und genaue Berechnung. Darüber hinaus konnte der Speicherbedarf um bis zu 68% gesenkt werden. Diese Optimierungen verringern die Gesamtsimulationszeit um 15%. Damit wird das Potenzial zugänglicher, wissenschaftlich validierter Mikroklimamodelle durch eine GPU-Optimierung demonstriert.

Abstract

Due to climate change, the frequency and severity of extreme weather events is increasing, which endangers human livelihoods and key infrastructures. Decision support tools can guide the development of climate-resilient cities by providing information on the potential effectiveness of specific measures during the planning process. In urban environments, decision support tools that incorporate accurate micro-climate models are particularly effective. PALM-4U, a state-of-the-art, scientifically validated microclimate model, could offer this functionality, however it remains largely inaccessible outside the scientific community as it is optimised to run on HPC clusters. However, with the rise of high-performance GPUs, a shift towards single workstations is possible.

This study investigates the potential for performance increase of the PALM-4U's pressure solver, by utilising the GPU's acceleration potential in combination with a change in target architecture. Performance increase is measured using three parameters: *speed up*, *validity* (via NMSE, R, and FB), and *memory efficiency*. Also the effect on the runtime of the full simulation is measured and possible bottlenecks identified. Finally, the full model is analysed to assess the overall feasibility of GPU optimisation, providing insights to guide future development.

The pressure solver transforms the 3D Poisson equation using Fast Fourier Transform and solves the resulting 1D system via the Thomas algorithm. The code structure is optimised, CUDA-optimised kernels are implemented and the cuFFT library is integrated. In addition a mixed-precision approach is tested to evaluate its impact on performance and accuracy.

The single core GPU implementation achieves a speed up of up to 65.5 times in single precision and up to 49.3 times for double precision for large domain sizes. The stability of the system remains unaffected by the mixed-precision approach, and no significant variation is observed between FP32 and FP64 runs. After 45×10^3 simulation steps, NMSE (0.02), FB (-0.017) and R (0.96), demonstrate a stable and accurate performance consistent across precisions. Additionally, the memory requirement is reduced up to 68% compared to the baseline CPU solver. The optimisations leads to a runtime reduction of the full model by 15%, demonstrating the potential for accessible, scientifically validated microclimate models.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Research Structure . . . 4

2 Fundamental Background **5**
2.1 Large-Eddy Simulation . . . 5
2.2 PALM Model System . . . 7
2.3 Numerical Method . . . 10
2.4 Code Structure . . . 12
2.5 Graphics Processing Unit . . . 14
2.6 scenarify . . . 17

3 Related Work **19**
3.1 Urban Microclimate Modelling . . . 19
3.2 GPU-Accelerated Climate Modelling . . . 20
3.3 Advanced GPU Methods . . . 21
3.4 Discussion and Gaps . . . 23

4 Methods **25**
4.1 Research Design Overview . . . 25
4.2 Computational Environment Setup . . . 26
4.3 Experiment Design and Evaluation . . . 27

5 Implementation **33**
5.1 Setup . . . 33
5.2 Preliminary Implementation . . . 37
5.3 Data Types and Memory Optimization . . . 38
5.4 Algorithmic Optimization for GPU . . . 40

5.5 Resulting Structure and Limitations 46

6 Results **49**
6.1 Speed Up Analysis 49
6.2 Validity Evaluation 56
6.3 Memory Profiling 58
6.4 Structural Analysis of the Model 60

7 Discussion **63**
7.1 Performance Improvements 63
7.2 Runtime Impact on Full Simulation 65
7.3 Impact of Memory Management 65
7.4 Identified Bottlenecks 66
7.5 Feasibility of Full Model Optimization 66
7.6 Limitations of the Study 67
7.7 Implications 68

8 Conclusion **69**

Overview of Generative AI Tools Used **71**

List of Figures **73**

List of Tables **75**

List of Algorithms **77**

Bibliography **79**

CHAPTER 1

Introduction

Climate change is arguably one of the most complex challenge humanity has faced to date. As the planet warms, natural disasters such as extreme weather will continue to affect billions of people worldwide [1]. When looking at urban settings, climate change can lead to an increase in the intensity of heat extremes, which has a direct impact on human health, livelihoods and key infrastructures [1, 2, 3, 4]. As cities are home to half of humanity [5], it is important to consider these impacts and risks when designing and planning urban and rural settlements, as the consequences of climate change can be mitigated if suitable adaption measures are implemented [6, 7, 1]. This concept is known as Climate-Sensitive Urban Design and emphasises the importance of climate adaption in city planning, in addition to the focus on economic, spatial, functional and aesthetic aspects [8].

There have already been measured benefits from the implementation of adaption measures, but there is a wide gap between the actually implemented measures and the societally set goals. Research shows that climate-resilient development benefits form the availability of decision support tools to guide the active planning process [1] by providing information about the potential effectiveness of specific measures [9, 10]. To examine the impact of climate change on the urban environment, microscale modelling is used. An example is the PALM-4U urban climate model system, which simulates atmospheric processes in urban areas on a city- and building-resolving scale [11, 12]. Urban microclimate models such as PALM-4U can help identify options that enable mitigation and adaptation across a wide range of future scenarios, while remaining cost-effective and minimising potential risks [1].

The problem, however with high-accuracy simulations like PALM-4U is that they have been developed primarily for scientific research and run optimised on high-performance computing clusters (HPC). Although PALM-4U has the possibility to run on desktop computers in serial mode, the performance is slowed down drastically [13]. In some cases,

even scientific research is restricted by the limited computing power of the available HPC systems, which limits the spatial resolution that can be analysed in a study [14].

Consequently, accurate and scientifically validated urban microclimate models are not readily available outside the scientific community to people working on climate change adaptation, such as policy makers, urban planners, and disaster managers [15]. This is problematic, as decision-support tools could help close the adaptation gap in climate-resilient planning.

The aim of this work is to investigate the potential of utilisation the parallel computing capabilities of modern graphic processing units (GPUs) to speed up the calculation of the urban climate model PALM-4U. This will allow the model to be used interactively on a standard desktop computer, making it available outside the scientific community as a decision-support tool. This work is part of the COMET Module *ClimaSens - Climate-Sensitive Adaptive Planning for Shaping Resilient Cities* which aims to address the challenge of making high-accuracy urban climate models available to people working on climate change adaption without access to a high-performance computing cluster [16].

In a the past Knoop et al. [17] attempted to port the entire PALM Model System codebase to the GPU using OpenACC. They achieved an approximate 2.3 times speed up for the total model and a 1.5 times speed up for the FFT-based pressure solver when running the model with a minimum of four CPU cores, and saw a decrease in performance for an increase of cores. This work chose a different approach and researches the optimisation potential of the microclimate model by changing the target architecture to a single CPU/GPU workspace and using a low level GPU language. The focus is specifically put on the FFT pressure solver, which has been identified as a computationally intensive routine in the PALM model core. As PALM-4U is an extension of the PALM model designed for modelling urban settings, the two share the same computational core. Although the ultimate objective is to run PALM-4U efficiently, for the purposes of this analysis of the pressure solver, focusing on the PALM model is sufficient.

The following research questions will be answered:

1. What effect does transitioning an FFT-based pressure solver from a HPC cluster-optimised implementation to a GPU-accelerated version on a single workstation have on its performance?

2. Is it possible for the parallelisation of a single module to have a substantial impact on the runtime of an entire system?

3. How does memory management on GPUs affect the performance of the FFT-based pressure solver in the PALM-4U model?

4. Which bottlenecks or limitations can be identified in comparison to the GPU Cluster attempts?

5. Does this optimisation provide insight into the overall feasibility of the optimisations of the full model for the target architecture?

When implementing the optimised version, the main focus lies on analysing and adapting the structure of the code to fit the new target architecture. This also involves considering the GPU platform, particularly its defining features, such as memory architecture and thread hierarchy, which play an important role in efficiently organising and executing parallel tasks. For this, the NVIDIA API CUDA is used, as it provides an extension to the C/C++ language offering the possibility of specifying parallelism at thread level and also specifying GPU-specific operations [18]. Guiding the optimisation process, profiling tools such as Nsight Systems [19] and Nsight Computer [20] are used. Nsight Systems uses a unified timeline to visualise algorithms and identify optimisation opportunities. Nsight Compute provides detailed performance metrics on a kernel-by-kernel basis. Once the implementation is complete, the resulting code will be benchmarked against real urban topography, using Cologne. Different dimensions are chosen for the scalability analysis with the specific target application in mind. In addition to improving runtime speed, the required memory and margin of error compared to the existing CPU implementation will be analysed.

1.1 Contributions

This research presents several contributions to the field of microclimate modelling and high-performance computing, particularly in the context of adapting the PALM model system for single-node architectures using a GPU. The key contributions are:

- A structural analysis of the PALM codebase to identify components suitable for single-node GPU optimisation.
- Shift of the target architecture of the PALM model system from high-performance computing (HPC) clusters, to a single-node architecture, investigating the optimisation potential introduced by this change when optimising for a GPU execution.
- Implementation and evaluation of a GPU accelerated version of the FFT based pressure solver using the low-level programming language CUDA, showing an improved performance over the existing single-node desktop PC execution.
- Demonstration of the feasibility and performance benefits of using mixed-precision modelling in microclimate modelling for hybrid CPU-GPU execution.
- Proof-of-concept for running PALM on a single-node GPU, making microclimate modelling more accessible outside of HPC environments.
- Evaluation of GPU performance within the scenarify environment, using the prior PALM integration to provide realistic and relevant benchmarking.

1.2 Research Structure

Firstly, Chapter 2 presents relevant background information. Section 2.1 provides the theoretical basis for climate modelling. Section 2.2 provides a detailed description of the PALM-4U model and derives the Poisson pressure equation. Section 2.3 discusses the numerical method used to solve this equation. Section 2.4 provides detailed information about the relevant code section. Section 2.5 provides a basic introduction to the graphics processing unit (GPU), and Section 2.6 then describes the framework in which the optimised solver will be benchmarked. State-of-the-art research in this field is presented in Chapter 3, which initially focusses on current models used in microclimate modelling. Then common approaches, which are used during GPU acceleration attempts on climate models are presented. Finally, state-of-the-art GPU algorithms are introduced.

Chapter 4 outlines the methods used, starting with an in-depth analysis of the original codebase. It also presents the external libraries used in the implementation process. Starting from the baseline GPU implementation, Chapter 5 presents all performed implementation steps. Chapter 6 benchmarks the optimised GPU implementation, focusing on speed up and memory usage. Finally Chapter 7 discusses the presented results in the context of the presented research questions.

This work lays the foundation for further research which could expand the focus to analyse the optimisation potential of utilising the GPU in different parts of the urban climate model.

CHAPTER 2

Fundamental Background

The following chapter aims to lay the foundation for the rest of the research. Firstly the theoretical framework for modelling atmospheric flow is explained. Next, it provides a detailed explanation of the PALM model and how pressure calculations are performed. Then the numerical strategy for solving the resulting equation is presented and how this method is implemented in the PALM model. Finally an overview of the graphics processing unit and the scenarify framework is provided.

2.1 Large-Eddy Simulation

Naturally occurring flows, and therefore atmospheric flows, are turbulent. For this reason, researchers in the field of computational fluid dynamics (CFD) focus on flows where turbulence plays an important role. Although the theory is not fully understood, the developed methods enable the flow to be modelled with sufficient accuracy. To achieve that the three-dimensional computational grid must be large enough to capture large-scale effects, and the mesh size must be small enough to resolve the smallest dynamically significant length scale, known as the Kolmogorov micro-scale [21].

From this, different approaches were designed to simulate turbulence, an overview can be seen in Figure 2.1. The most accurate method is the direct numerical simulation (DNS), where the full Navier-Stokes equations are numerically solved, with corresponding domain size and mesh size to capture all scales making it computationally expensive [21]. As visualised in Figure 2.1, in the direct solver all scales, from large eddies to dissipating eddies, are resolved [22]. At the other end of the spectrum, where fewer computational resources are required, there is a solver that solves the Reynolds-averaged Navier–Stokes (RANS) equations. This method is useful if the focus is put on steady-state fluid flow rather than instantaneous flow. In these simulations, only averaged quantities are resolved and the turbulence is modelled separately leading to less accurate results [21].

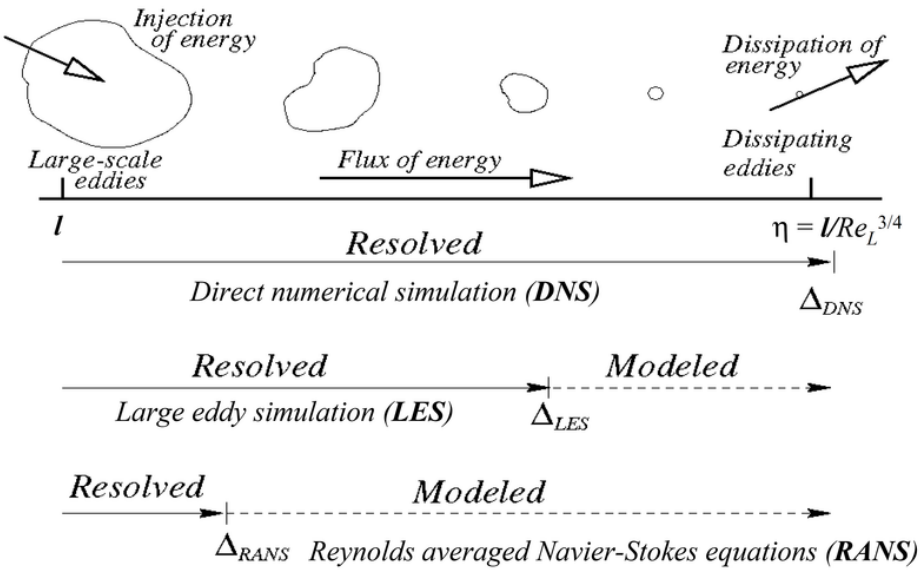


Figure 2.1: Overview of numerical modelling approaches of turbulent flow [22].

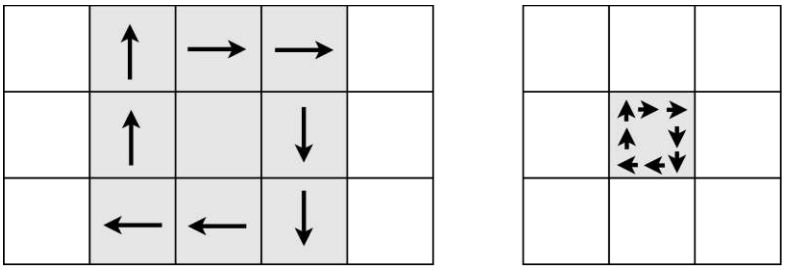


Figure 2.2: Comparison of large-scale resolved eddies (left) and parametrised sub-grid scale eddies (right).

As shown in Figure 2.1, the large-eddy simulation (LES) approach sits between DNS and RANS. Large-scale motions of turbulent flows are calculated directly, while turbulence at smaller scales is modelled. The difference between resolved and sub-grid scale (SGS) eddies can be seen in Figure 2.2, where the left figure shows a resolved eddy, while the right image shows sub-grid scale turbulence, which is not resolved. This approach achieves a higher level of accuracy than RANS because large eddies contain most of the turbulent energy, but is still less computationally expensive than a direct solver (DNS). It is therefore a widely used numerical tool for simulating realistic turbulent and transitional flows [21].

2.2 PALM Model System

This research is based on the PALM Model System 6.0, which has been used to study the atmospheric and oceanic boundary layer for around 20 years. In 2020, an extension to the climate model made it possible to simulate atmospheric processes in urban settings at a building-resolution scale. This is known as PALM-4U [12, 11]. As this research focuses on the pressure solver of the model core, which is shared by both PALM-4U and PALM, the focus for now is on the PALM model. However, the results can also be applied to the full PALM-4U model.

Figure 2.3 shows three exemplary results from a PALM-4U simulation of the Maria-Theresien-Platz in Vienna from initial test runs conducted to evaluate key output parameters relevant for subsequent analysis and future application [23]. Figure 2.3a shows the simulated wind speed in meters/second, while Figure 2.3b displays the calculated air temperature in degrees Celsius. Furthermore, the PALM-4U model can be used to simulate bioclimate indices, as shown in Figure 2.3c, which displays the universal thermal climate index (UTCI), which describes the physiological comfort of the human body [24].

In the default setting, which is what this study is using, PALM is a Large Eddy Resolving (LES) model. The model is using the finite difference approximation for model domain discretization in space with equidistant horizontal grid spacing. As it has been found to be well suited for fluid dynamics simulations, a staggered Cartesian grid is used [11]. This means that temperature and pressure are set at the centre of the grid cell. However, vector-valued variables such as velocities u, v, w are anchored at the boundary between two cells. This leads to the calculation of the velocity divergence at the centre of each cell, at the exact location of the pressure or temperature, which is beneficial for calculations at the boundary [25]. Figure 2.4 shows a visualisation of the grid and the positions of the variables within it.

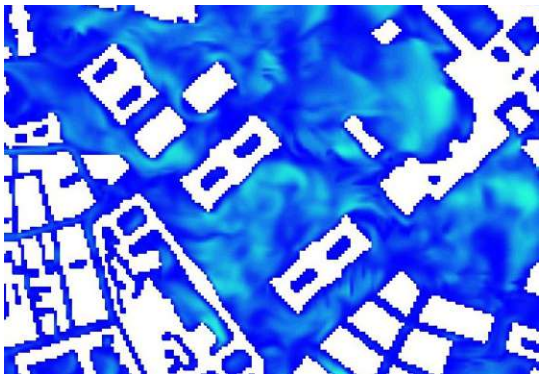
In order to achieve a more efficient calculation the model is optimised to run in parallel on HPC clusters. To enable this, the domain is decomposed into equally-sized 2D subdomains along the x and y directions, which are then divided up among the available cores. MPI communication is used to exchange information between compute cores [11].

In the dynamic core of the model, the prognostic equations are calculated at each time step, incrementing the designated variable by one time step. After this, the pressure solver is executed. In serial execution, this takes approximately 15% of each full time step. At the end of the time step, the desired output variables are calculated. A visualisation of the model core execution can be seen in Figure 2.5.

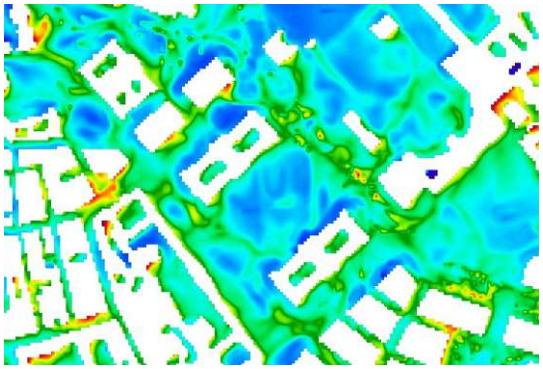
The PALM model solves the non-hydrostatic, filtered, incompressible Navier-Stokes equations in the Boussinesq-approximated form [26]. In this context filtered means the separation of the resolved scale from the subgrid scale. This allows the model to simulate the atmosphere from the urban microclimate scale to atmospheric phenomena that extend throughout the troposphere [11]. The Boussinesq approximation assumes that the variation in density can be ignored in the equations of motion except in the

buoyancy term [27].

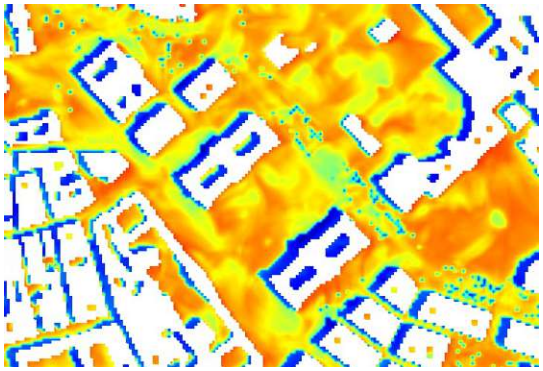
Since the focus of this analysis is on the pressure solver, This section presents the momentum equation of the PALM model core and how to derive the Poisson equation for pressure from it. The momentum equation consists of five forces: the advection force, the Coriolis force, the pressure force, the buoyancy force, and a stress term that includes the sub-grid scale which is not resolved by the model. In Equation 2.1 angular brackets represent the horizontal domain average of that variable, the variables with the subscript 0 symbolise a surface value, and the variables with a double prime denote sub-grid scale variables. $\pi^* = p^* + \frac{2}{3}\rho_0 e$ is the modified perturbation pressure, with p* being the perturbation pressure.



(a) wind speed ranging form $0m/s$(dark blue) to $4m/s$(light blue).



(b) air temperature ranging from 26.3řC(dark blue) to 32.4řC(red).



(c) The Universal Thermal Comfort Index (UTCI) ranges from 24 (dark blue), which indicates no heat stress, to 39.7 (red), which signifies strong heat stress [24].

Figure 2.3: PALM-4U simulation results of the Maria-Theresien-Platz in Vienna.

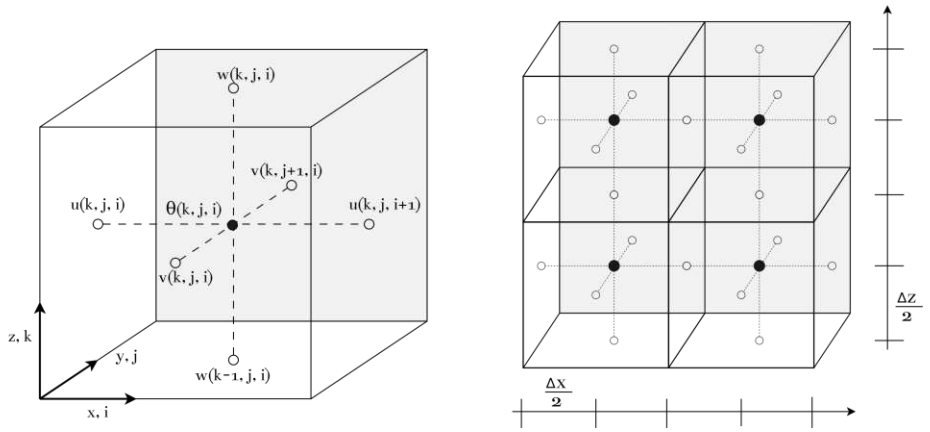


Figure 2.4: Visualisation of the computational grid.

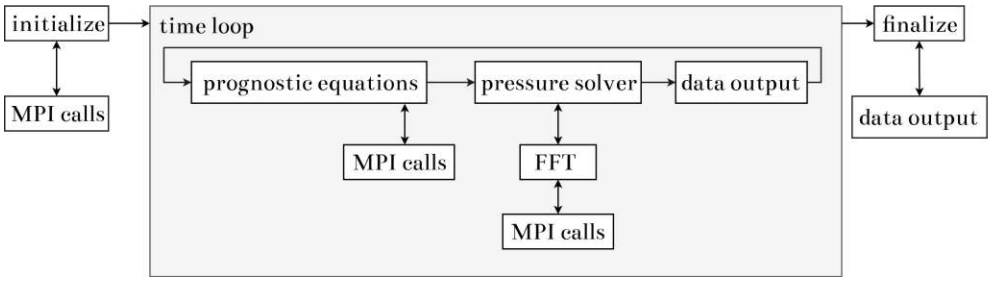


Figure 2.5: Visualisation of the time loop of the PALM model core [17].

$$\frac{\partial u_i}{\partial t} = -\frac{\partial u_i p_j}{\partial x_j} - \epsilon_{ijk} f_j u_k + \epsilon_{i3} f_3 u_{g,j} - \frac{1}{\rho_0}\frac{\partial \pi^*}{\partial x_i} + g\frac{\theta_v - \langle\theta_v\rangle}{\langle\theta_v\rangle}\delta_{i3} - \frac{\partial}{\partial x_j}\left(\overline{u_i'' p_j''} - \frac{2}{3} e \delta_{ij}\right) \tag{2.1}$$

Equation 2.2 shows the continuity equation which must also be fulfilled. Since the Boussinesq approximation is used in the governing equations, Equation 2.2 can be transformed into Equation 2.3, the continuity equation for incompressible flow.

$$\frac{\partial \rho}{\partial t} + \nabla(\rho u) = 0 \tag{2.2}$$

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

However, incompressibility is not provided as divergence of the flow field is produced as the result of the integration of the governing equations. To compensate, a predictor–corrector method is used, where a second equation is solved after each time step [25]. When

performing the time integration for the next time step, Equation 2.1 is split into two parts. In the first step, the pressure term is excluded from Equation 2.1 and a preliminary velocity $u_{i,pre}^{t+\Delta t}$ is calculated. The appearing divergence can then be attributed to the pressure term. In a second step, shown in Equation 2.4, the pressure term can be used to calculate the prognostic velocity.

$$u_i^{t+\Delta t} = u_{i,pre}^{t+\Delta t} - \Delta t \frac{1}{\rho_0} \frac{\partial \pi^{*t}}{\partial x_i} \tag{2.4}$$

To enforce incompressibility for $u_i^{t+\Delta t}$ Equation 2.5 can be written.

$$\frac{\partial}{\partial x_i} u_i^{t+\Delta t} = \frac{\partial}{\partial x_i} \left(u_{i,pre}^{t+\Delta t} - \Delta t \frac{1}{\rho_0} \frac{\partial \pi^{*t}}{\partial x_i} \right) \stackrel{!}{=} 0 \tag{2.5}$$

This results in a Poisson equation (Equation 2.6) for the modified perturbation pressure π^*.

$$\frac{\partial^2 \pi^{*t}}{\partial x_i^2} = \frac{\rho_0}{\Delta t} \frac{\partial u_i^{t+\Delta t,\mathrm{pre}}}{\partial x_i} \tag{2.6}$$

The ideal solution to this problem would result in a u that is free of divergence when inserted in Equation 2.4. However, in practice, a reduction in divergence of several orders of magnitude has been found to be sufficient [26].

2.3 Numerical Method

This section presents a solution method for the three-dimensional Poisson equation derived in the previous section. For better readability in this section, the modified perturbation pressure vector π^{*t} is written as u and the right-hand side of the equation as f and Equation 2.6 can be written as Equation 2.7.

$$\frac{\partial^2 u_i}{\partial x_i^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f \tag{2.7}$$

The finite difference method with equidistant grid spacing can be used to rewrite Equation 2.7, resulting in

$$\frac{1}{h^2} \begin{pmatrix} B & -I & & \\ -I & B & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & B \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix}$$

with

$$B = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \quad .$$

The j-th (block) row can be written as Equation 2.8.

$$-u_{j-1} + Bu_j - u_{j+1} = f_j \tag{2.8}$$

The Fourier method can be used to solve this equation, as it exploits knowledge of the eigenvalues and eigenvectors of the matrix B, where eigenvalues are

$$\lambda_j = 4 - 2cos\left(\frac{j\pi}{p+1}\right) = 4 - 2cos(\theta_j) \quad j = 1, ..., p$$

and eigenvectors are defined as

$$q_j = \sqrt{\frac{2}{p+1}} \times [sin(\theta_j), sin(2\theta_j), .., sin(p\theta_j),]^T \quad .$$

With $Q = [q_1, ..., q_p]$ being the eigenbasis, matrix B can be diagonalised to $Q^T BQ = \Lambda = diag(\lambda_i)$. A similar transformation can be used to rewrite Equation 2.8 as follows

$$-Q^T u_{j-1} + (Q^T BQ)Q^T u_j - Q^T u_{j+1} = Q^T f_j \quad .$$

By adapting the notation such that a bar denotes the variables in the Q-basis, the equation becomes

$$-\bar{u}_{j-1} + \Lambda\bar{u}_j - \bar{u}_{j+1} = \bar{f}_j \quad .$$

When the matrix equation is put back together, m independent tridiagonal systems emerge [28], as can be seen in Equation 2.9.

$$\begin{pmatrix} \lambda_i & -1 & & & \\ -1 & \lambda_i & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & \lambda_i & -1 \\ & & & -1 & \lambda_i \end{pmatrix} \begin{pmatrix} \bar{u}_{i1} \\ \bar{u}_{i2} \\ \vdots \\ \bar{u}_{ip-1} \\ \bar{u}_{ip} \end{pmatrix} = \begin{pmatrix} \bar{b}_{i1} \\ \bar{b}_{i2} \\ \vdots \\ \bar{b}_{ip-1} \\ \bar{b}_{ip} \end{pmatrix} \tag{2.9}$$

The full methods can therefore be separated into three stages [28, 29, 30].

1. Determine $\overline{f_j}$, $j = 1, 2, ..., n$ by applying a discrete 2D Fourier transform on the right-hand side of the original equation.

$$f(m+1, n+1, k) = \frac{1}{Mx * My} \sum_{i=0}^{Mx+1} \sum_{j=0}^{My} f(i, j, k) \omega_{Mx}^{-m(i-2)} \omega_{My}^{-n(j-1)} \tag{2.10}$$

   with:

$$\omega_M = exp(\frac{2\pi i}{M}) \tag{2.11}$$

   This can also be separated in two one-dimensional Fourier transforms, and executed with a transposition in between.

2. Solve the resulting one-dimensional tridiagonal linear system for $\bar{u}$ from Equation 2.9.

3. Compute the solution for u_i.

$$u(i, j, k) = \sum_{m=0}^{Mx-1} \sum_{n=0}^{My-1} \overline{u}(m+1, n+1, k) \quad \omega_{Mx}^{m(i-2)} \omega_{My}^{n(j-2)} \tag{2.12}$$

   with:

$$\omega_M = exp(\frac{2\pi i}{M}) \tag{2.13}$$

Taking the complex conjugate pairs formed by the complex Fourier modes into account can reduce the necessary computational work and storage. The transformation can be seen in Equation 2.14.

$$\overline{u}(m+1, n+1, k) = \overline{u}(Mx - m, Mx - n, k) \tag{2.14}$$

The benefits of this method for solving large three-dimensional systems lie in the fact that the three-dimensional coupled equation is transformed into a set of independent one-dimensional equations that can be solved in parallel.

2.4 Code Structure

This section describes how the numerical method of solving the pressure Poisson equation using the Fast Fourier Transform, described in Section 2.3, is embedded in the PALM model core. The complete PALM codebase consists of 551 Fortran 90 files, each file containing hundreds to thousands of lines of code. The project uses different subroutines and modules to create a framework, with each module managing a specific process.

The FFT pressure solver is spread out between seven files, Figure 2.6 shows how these files are connected. As the codebase grew over the years, more and more optimisations were added for different cases, this adds to the complexity of the code structure because all possible versions of the pressure solver can still be run if desired.

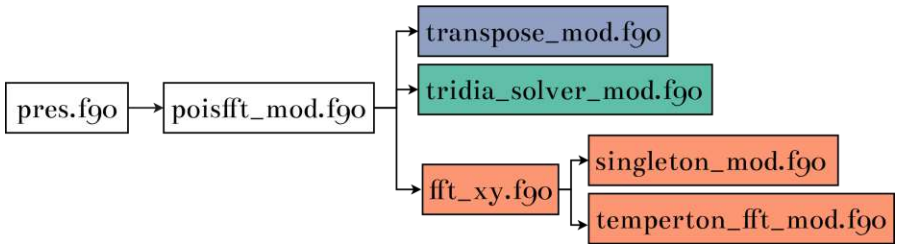


Figure 2.6: Visualisation of the file dependencies involved in the pressure solver. Arrows indicate the direction of function calls: a file at the tail of an arrow calls a file at the head of the arrow.

In the PALM model core, the pressure solver starts with the computation of the right hand side of the Poisson equation for the modified perturbation pressure from the preliminary velocities, as shown in Equation 2.6. This is executed in file *pres.f90*. After that, the solution method, in this case the FFT solver, is chosen and the designated file is called. File *poisfft_mod.f90* controls the solver and calls subroutines as needed. This file starts with the initialisation of the routines used. As the discrete Laplacian matrix of the Poisson equation is constant for each time step, the FFT and the coefficients of the Thomas algorithm are initialised once at the beginning of the simulation.

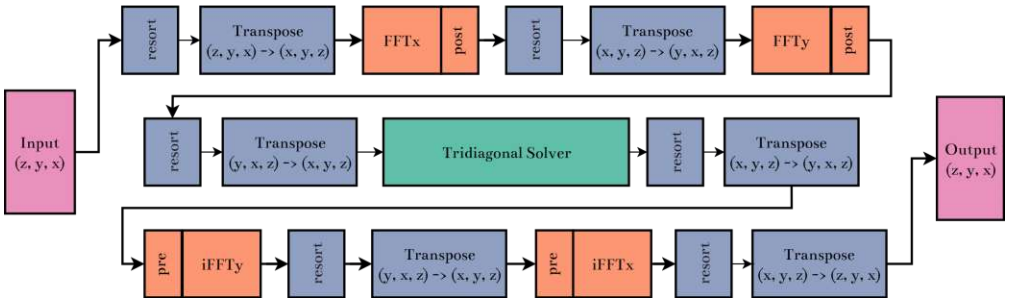


Figure 2.7: Breakdown of the FFT based pressure solver into 19 steps.

The code structure of the file *poisfft_mod.f90* can be separated into 19 steps that are executed each time step. These steps are visualised in Figure 2.7 and start after the allocation of the working memory needed. As the input array is spread across multiple CPU cores, it must be resorted to ensure the array is correctly arranged in memory for the MPI communication that is necessary during the execution of the three-dimensional transposition. The transposition is necessary, as the input array is stored in memory in a column-major order, where each node has a z-dimensional column of data. However, for the FFT in the x-direction, one full x-row needs to reside on one node. The same

process is repeated to shift the data for the FFT in the y-direction. As the Thomas coefficients are initialised and saved with x in its leading dimension, the data must be resorted and transposed again before the tridiagonal solver can be started. This solver is the central point of the method. Once completed, the solution array is transformed back to its original form. The method concludes with the deallocation of the temporary arrays and the return of the solution array.

As mentioned above, the functionality of the pressure solver is divided into different files. The file *poisfft_mod.f90* coordinates the FFT pressure solver, while the other functionality is wrapped up in subroutines in designated files. Table 5.1 lists and summarises all files connected to the FFT pressure solver.

Filename	Description
pres.f90	Driver for pressure calculation calculates the perturbation pressure (π^*) and does pre- and post-processing
poisfft_mod.f90	Driver for FFT calls all relevant modules
transpose_mod.f90	Computes 3D reorderings for MPI calls and 3D transposition of matrix
tridia_solver_mod.f90	Solved Tridiagonal System
fft_xy.f90	Pre- and post-processing and set up of 1D FFT calls and calls desired FFT methods
temperton_fft_mod.f90	1. FFT method
singleton_mod.f90	2. FFT method

Table 2.1: A summary of the description of each file.

2.5 Graphics Processing Unit

To optimise a module for the graphics processing unit (GPU), first the characteristics of this architecture need to be understood. A GPU differs from the standard CPU in that it offers higher memory bandwidth and instruction throughput. A CPU can execute a sequence of operations as quickly as possible, and to achieve parallelism, a few tens of these threads can be executed simultaneously. A GPU can execute thousands of threads in parallel, making it possible to offset the slower performance of individual threads and achieve greater throughput. This makes the GPU well suited for highly parallel computations [31].

These capabilities can be harnessed in several ways. One low-level option is CUDA C++, an extension to the C++ language that enables programmers to define C++ functions called *kernels*. At runtime, these kernels can be executed by hundreds of CUDA threads. To manage the large amount of threads the NVIDIA GPU has a unique architecture called SIMT (Single Instruction, Multiple Threads). Similarly to SIMD (Single Instruction,

Multiple Data), this architecture allows a single instruction to control multiple processing elements. However, unlike SIMD, SIMT enables the execution of both thread-level and data-parallel code, for independent and coordinated threads. When executing a kernel, threads are scheduled in groups of 32, known as warps, in which one common instruction is executed in parallel. This leads to full efficiency if all 32 threads have the same execution path. On a larger scale, threads are grouped together into structures called thread blocks. A user-set number of thread blocks is then organised into a grid, the size of which is dictated by the amount of data. Threads within a block reside on the same streaming multiprocessor core, enabling them to share resources. However, complete thread blocks are executed independently of each other, and the order in which they are executed is not fixed [31]. An overview can be seen in Figure 2.8.

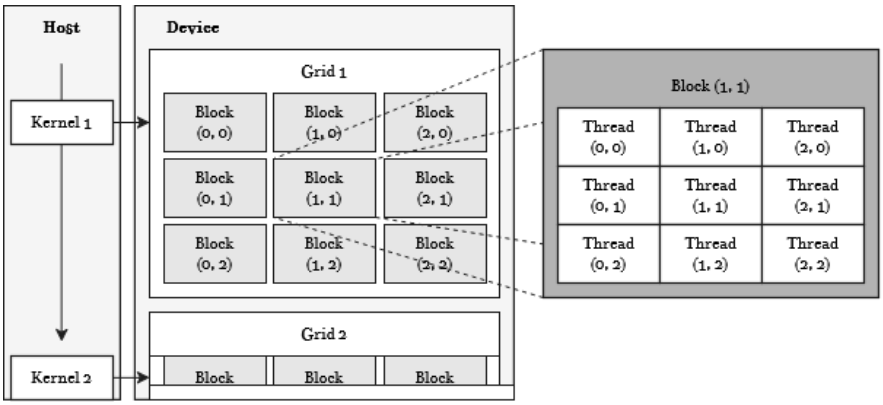


Figure 2.8: Visualisation of the CUDA Programming Model [31].

To optimise performance, CUDA uses various memory types. In general, memory can be divided into three groups according to memory access speed: host memory, device off-chip memory, and device on-chip memory. Figure 2.9 illustrates the different types of memory.

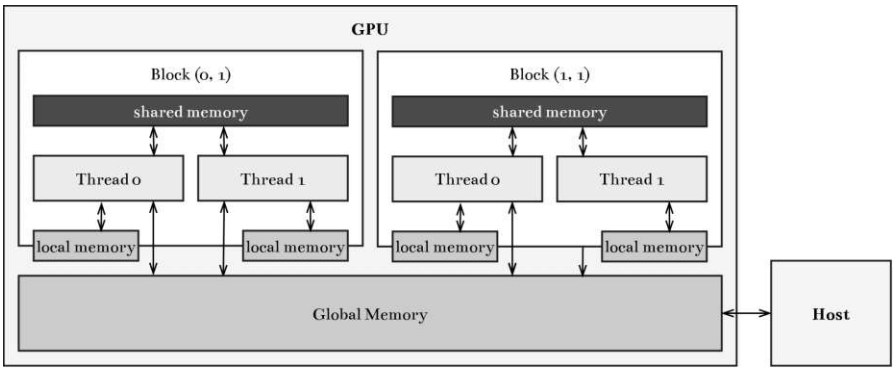


Figure 2.9: Visualisation of the CUDA Memory Model [31].

Host Memory: The host memory is accessible by the CPU and is separate from the GPU.

When a GPU kernel wants to access a specific item that is allocated on the host memory, it must be copied from the host memory to the GPU via a PCI Expressbus. Since the default host memory is pageable, the operating system can move the memory without notice. This prevents the GPU from having direct memory access (DMA) via hardware. Therefore, during the copying process, the desired memory must first be fixed in a CPU staging memory buffer and then copied to the GPU in a second step [18, 31].

Global Memory: When data is copied from host memory to the GPU, it is stored in global memory, which is integrated directly into the GPU, but not on the SM. Compared to the PCI Express link (6GByte/s), access to global memory has a high peak bandwidth (140GByte/s), and data can be read and manipulated during kernel execution. For optimal performance, CUDA kernels should perform coalesced memory access operations. This uses the hardware optimally [18] as memory is always accessed in segments of 32, 64 or 128 bytes [31]. This means that in order to perform coalesced access, the element must be at least 32 bits in size, and the addresses of threads accessed within a warp must increase contiguously [18].

Local Memory: Local memory is part of off-chip global memory, but during a kernel execution each thread has its own stack which only it can access during kernel execution.

Shared Memory: Shared memory is physically located in every Streaming Multiprocessor (SM), which is a fundamental processing unit on the GPU. This memory can be used to exchange data between threads within a thread block during kernel execution. Access to shared memory is ten times faster than access to global memory. To optimise for 32-bit access, shared memory is structured in interleaved banks of memory, meaning threads read in an interleaved pattern all accessing a different bank at a time. If this access pattern is not followed and multiple threads try to interact with the same memory bank at the same time, a bank conflict occurs, and the hardware has to handle the requests sequentially, stalling the system [18].

As CPU and GPU threads are physically separate, they can execute code concurrently as the CUDA programming model assumes that they function on different memory spaces. This is called heterogeneous programming. However, this asynchronous behaviour is broken if memory needs to be exchanged between device and host because, as described above, host memory is not page-locked and could be moved by the CPU during the GPU copying process. To achieve asynchronous copying, where the CPU does not have to wait for the data transfer to and from the GPU to complete, the memory can be page-locked on the CPU. This means that the operating system loses the ability to move it unannounced. With the CUDA API page-fixed memory can be initialised as pinned memory which can be copied to the GPU asynchronously. In addition, the GPU can copy it in one step during a copy operation, achieving faster transfer performance [18, 31].

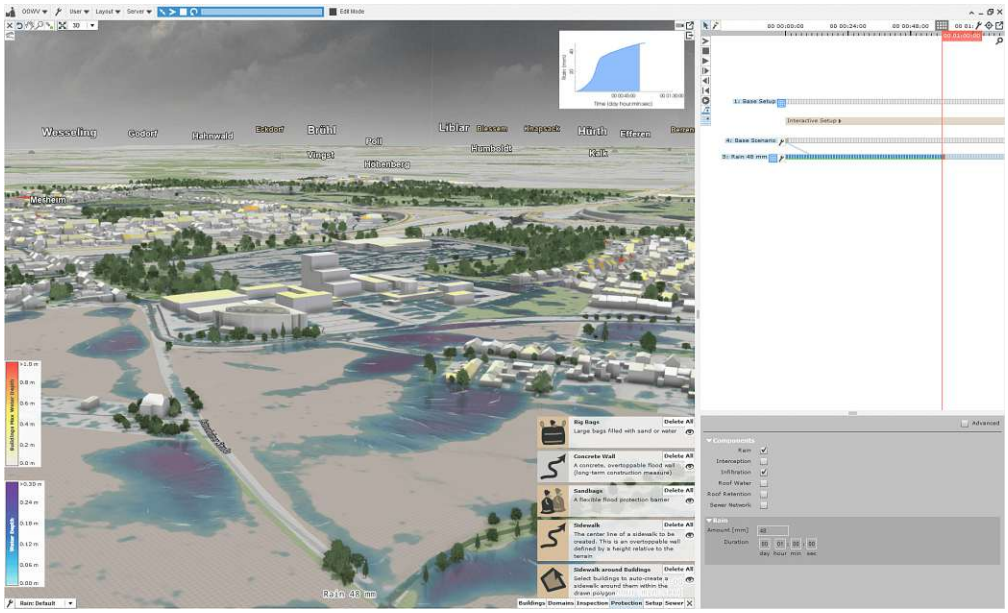


Figure 2.10: This figure shows an example of a scenarify simulation in the application [32].

2.6 scenarify

As described in the introduction, decision-support tools can be used to guide climate-sensitive urban design and help to close the adaptation gap in climate-resilient planning. One well-established decision-support tool for flood management is the scenarify system, an example of which can be seen in Figure 2.10. It integrates flood simulation, analysis and 3D visualisation tools, allowing information to be processed, filtered and extracted from time-dependent data within a single tool. scenarify is implemented using a wordlines concept, enabling scenarios to be created and switched between in order to evaluate the effect of different measures [32]. The resulting 3D visualisations are better suited to presentations and discussions, as they are more intuitive [33]. This streamlines the decision-making process.

scenarify follows the dataflow concept and consists of modular nodes that can be mixed and matched to create new modules and simulations, depending on the desired functionality. Extending functionality is therefore straightforward, as it does not require any alterations to the underlying architecture. This simplifies the integration of new models, such as PALM/PALM-4U, into the existing framework while enabling the use of the existing data preparation, visualisation and decision-support framework. For this reason, scenarify serves as a framework for the optimisations executed in this research.

CHAPTER 3

Related Work

This chapter outlines the related work to this research and is organised in three sections. The first section reviews state-of-the-art models used in urban climate studies. The second section presents current methods used when analysing the performance increase in climate models through the use of GPUs. It also gives an in-depth analysis of Knoop et al.'s paper. The final section focusses on GPU-accelerated algorithms that can be used to solve the Poisson equation via FFT on the GPU.

3.1 Urban Microclimate Modelling

To simulate the urban environment different modelling methods, with varying complexety and informational requirements exist. The radiation model SOLWEIG simulates three-dimensional radiation fluxes in complex urban domains, disregarding the effects of thermal mass or fluid flow through urban canyons [34]. SOLENE computes the sky irradiance (direct flux) and the sky vault (diffuse flux), as well as simulating their interaction with the environment's surfaces.[35]. The Rhinoceros 3D/Grasshopper native tool LADYBUG models solar radiation in combination with energy daylight simulations, but due to the computational complexity airflow and turbulence modelling is often not included, as is the interaction with vegetation [36]. VTUF-3D on the other hand does model the interaction of vegetation together with surface energy balance. However as with LADYBUG, airflow and turbulence are not considered [37].

In 2021 a review was conducted that examined 130 peer-reviewed papers published between 2006 and 2019 to investigate the use of urban climate models. The researchers found that during this period, most studies had used the microclimate model ENVI-met [38]. ENVI-met uses the dynamic coupling of heat transfer and vegetation with atmospheric flow to simulate microclimates in complex urban environments [39]. Among other things, temperature, wind speed and humidity as well as radiation exchange and thermal conductivity of surfaces are calculated [40]. Applications of this model include

studies on green roofs and facades in domains of size $40m \times 40m$ [41] and $180m \times 120m$ [42], simulated at a resolution of one meter. Other research has analysed urban heat mitigation potentials using a $400m \times 400m$ domain with a resolution of two meters [43]. In all ENVI-met simulations a domain height of double the maximum building height was chosen.

Due to limited computational power and memory, the maximum number of cells calculated with ENVI-met is approximately $250m \times 250m \times 30m$ [44] and the simulation time is limited to a 24 hour period. In addition, the model is only partially validated [40]. Furthermore, with ENVI-ment, it is possible to resolve the microscale, however, it is not possible to include atmospheric forcing data using a downscaling approach from mesoscale or macroscale simulations [45]. To make this possible, in 2019 the national research programme Urban Climate Under Change $[UC]^2$ extended the established PALM climate model to work at an urban scale, with the ability to downscale to capture larger effects [45]. In 2020, the resulting PALM-4U model was introduced [12]. Studies working with PALM-4U simulate typical domain sizes of $400m \times 256m$ [9] to $1km \times 1km$ with a resolution of one meter. The model is optimised for massively large parallel computers with CPUs only, but can also run on local workstations. However, as mentioned in the introduction, there it runs in serial mode with performance loss [14]. Typical simulation run times for Windcomfort simulations with PALM-4U can be as long as 5 hours for a domain size of $400m \times 450m \times 50m$ with a resolution of $1m$ running on an HPC (Levante) with 300 cores for a six hours simulation. For a single workstation with 52 cores, a smaller domain size of $320m \times 320m \times 80m$ and a resolution of $1m$, a runtime of 14h hours is suggested for three hours of simulation [46].

3.2 GPU-Accelerated Climate Modelling

GPUs have become an important solution for accelerating scientific applications. Two main approaches have been adopted when working with climate models. To achieve GPU runtime acceleration, firstly, the directive-based high-level programming paradigm OpenACC is used [47, 48, 49, 17]. Research using this method mainly focusses on porting an entire climate model to run on multiple CPUs and GPUs. The second method focusses on a specific subsection of the climate model, exploring acceleration potential using the low-level programming API CUDA to run mostly on one GPU [50, 51]. In a case study, Norman et al. compared the two approaches using an atmospheric climate kernel. They found that, although developing the OpenACC kernel is simpler, the CUDA kernel achieved a better speed up compared to the CPU version [52].

In 2016 during the GPU Hackathon at TU Dresden/Forschungszentrum Jülich, Knoop et al. [17] worked on a GPU-accelerated version of the PALM model core. Their goal was to port the entire codebase of the model to the GPU using OpenACC and MPI + OpenMP, increasing the computational power in small clusters. Their approach was similar to the first method described above. They added OpenACC directives to the hotspot subroutines throughout the codebase without changing the code structure, allowing the compiler to

optimise existing loops. To transfer data to the GPU, the code blocks are wrapped in data regions, with data automatically transferred at the start and end of each region. As this data movement can be a bottleneck in GPU code, they attempted to minimise it by merging data regions, but this was limited by the need for MPI communication to exchange data between the different nodes of the cluster. In some cases CUDA-aware MPI enabled them to transfer data directly between GPUs. However, they experienced severe performance loss when the data were non-contiguous in memory, forcing them to update the arrays on the CPU before sending them to the correct MPI rank. This was the biggest bottleneck, which reduced the final speed up they had expected [17].

When porting the FFT pressure solver they utilised the functionality of the NVIDIA C++ library cuFFT. As this library is not directly usable in Fortran they used the Fortran 2003 bind feature and the intrinsic module ISO C BINDING. To replace the original one-dimensional FFTs they created a *cuFFTPlan1D* and executed the forward and backward transformation with the *cuFFTExecD2Z* and *cuFFTExecZ2D* command [17].

With this method, they found a performance increase for up to ten CPU cores, with a runtime improvement ranging between 4.5 and 1.5 times for the prognostic equations as only a small amount of MPI calls are necessary in that section. However, they found that their method did not improve the performance of the pressure solver as much as expected. They only achieved a speed up of 1.5 times with a minimum of four cores, and with eight cores, the performance of their implemented version was slower than that of the CPU-based solver. They explained that this was due to the heavy optimisation for HPC clusters. Overall, they achieved performance improvements, with a maximum speed up of 2.3 times. The bottlenecks were MPI-heavy routines [17].

3.3 Advanced GPU Methods

As this study focusses on the pressure solver, This section provides information about the state-of-the-art algorithms required for this purpose. The numerical methods for solving the Poisson equation using Fast Fourier Transform (FFT) have long been developed as described in Section 2.3 [29, 53] and can be separated into three distinct parts: Transposition of a three-dimensional matrix, one-dimensional Fast Fourier Transforms and a tridiagonal matrix solver. Significant effort has been invested in optimising the algorithms involved, and with the rise of GPUs, the algorithms have also been adapted to this architecture.

Fast Fourier Transformation

The FFT is a fundamental building block that has been applied in a wide range of scientific and engineering disciplines. For that reason many different algorithms have been proposed like the Cooley-Turkey algorithm [54], Temperton algorithm [55], Vector radix FFT algorithm [56] and many more. Implementations and versions of these algorithms have

also been applied to the GPU to accelerate the computation. One such implementation is cuFFT, which is a state-of-the-art GPU-based FFT library provided by NVIDIA, with an algorithm based on the Cooley-Turkey algorithm [**nvidia_cuFFT_2025**]. It can be used to perform 1D, 2D and 3D FFTs and is optimised for GPUs. Several studies have been conducted to analyse its performance and potential setup optimisations when using cuFFT [57, 58, 59]. Recent studies have also attempted to utilise the tensor cores, special-purpose processing units of the GPU, for FFT computations. Pisha and Ligowski [60], Li et al. [61], and Durrani et al. [62] demonstrated tensor-core-based FFT methods and achieved better performance than the cuFFT library. Despite this, these methods have a number of drawbacks, such as limited accuracy [61] and supported vector size [60, 62]. Consequently, a more universal solution is to use standard GPU cores. When dealing with an array that exceeds the memory capacity of the GPU, two approaches have been developed: hybrid methods that distribute the workload between the CPU and GPU [63], and methods in which the GPU sequentially calculates 1D FFTs of sections of the array [64].

3D Transposition

Based on the optimised two-dimensional out-of-place matrix transposition on the GPU proposed by [65], which uses shared memory and padding to avoid bank conflicts, Jodra et al. developed a version that can work in three dimensions [66]. Additionally, a library named cuTT has been developed to handle tensor transposes, which can be applied for tensor ranks ranging from 2 to 12 [67] and has since been integrated into the Tensor Algebra Library TAL-SH [68]. If the GPU's memory is not large enough to fit the entire array, the transposition can be divided into memory chunks and calculated sequentially by the GPU [64].

Tridiagonal System Solver

The acceleration potential of solving the tridiagonal system using NVIDIA GPUs has been the focus of numerous studies. The state-of-the-art method for solving the tridiagonal system is the Thomas algorithm [69]. However, different approaches have been explored using other parallel algorithms such as Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) [70, 71, 72]. In the standard NVIDIA library cuSPARSE the function *cusparseSgtsv2StridedBatch* is provided, making batch computation of a high number of systems possible, using a combination of the CR and PCR algorithms. Compared to the Thomas algorithm, these algorithms need more operations. Pedro et al. developed an algorithm (*cuThomasBatch*) based on the Thomas algorithm [73] achieving better performance compared to *cusparseSgtsv2StridedBatch*, which has consequently been integrated into the cuSPARSE library as *cusparseSgtsvInterleavedBatch* [72].

3.4 Discussion and Gaps

A review of prior work in this field and state-of-the-art GPU methods suggests that the current approach to accelerate the PALM climate model prioritises a simple, high-level implementation of the entire code base when adapting for the GPU, potentially at the cost of performance. The approach to optimise models for the GPU using the low-level programming API CUDA, as describe in Section 3.2, has not yet been explored in the context of the PALM model. Switching to CUDA enables the use of state-of-the-art GPU techniques (3.3) with shared memory for example, which is inaccessible with OpenACC. Additionally, based on the identified bottleneck in GPU optimisation by Knoop et al. [17], this research changes the target architecture from a HPC cluster to a desktop workstation to eliminate MPI communication entirely.

CHAPTER 4

Methods

This chapter describes the methodology that will be used to investigate the performance gains of transitioning from the HPC-optimised pressure solver of the PALM model to a GPU-based, single workstation-optimised version. The methodology is structured to reflect the research questions set out in Chapter 1.

4.1 Research Design Overview

This study uses an experimental research design to explore the potential difference in performance between an optimised single workstation GPU implementation integrated into the scenarify framework and the baseline pressure solver of the PALM model. The performance measures used in this research are:

- **Speed Up**, which measures the runtime acceleration achieved by the optimisation, compared to the baseline implementation. It is defined as the ratio between median CPU wall clock time of the baseline and the minimum of the optimised version.
- **Validity**, which measures the numerical accuracy of the implemented solver compared to the baseline
- **Memory Efficiency**, which measured how much resources are need for the execution of the solver

Initially, the focus of the optimisation will be to identify structural changes made possible by the transition to a new target architecture. This will include an initial clean-up of MPI and cluster-specific code. Each section of the code will then be analysed and optimised for the GPU. Additionally, a mixed-precision approach will be tested in which the GPU code is executed in single precision. During this process, the speed up compared to the

original code will be calculated to identify effective optimisations. The profiling tools Nsight Compute and Nsight Systems will be used to guide the optimisation process.

Once the optimisation process is complete, the performance of the optimised pressure solver will be examined (RQ1). Then, the impact of the optimised module on the full simulation runtime will be analysed (RQ2). Thirdly, the runtime of the individual components of the pressure solver (such as the Fast Fourier Transform) will be evaluated in isolation, compared to the baseline. This will make it possible to identify any computational bottlenecks and analyse the effect of memory management (RQ3, RQ4).

As PALM is scientifically recognised and validated, the validity of the GPU execution must also be guaranteed. To ensure this, the numerical error between the original pressure solver and the GPU optimised version will be calculated as part of the performance evaluation, to make sure that the achieved speed up did not compromise the validity.

As this research is the first step towards analysing the optimisation potential of the full PALM-4U model for transitioning to a single workstation with a GPU, a high-level code analysis of the full model will be conducted to answer the final research question (RQ5).

Each research question will be addressed by a corresponding experiment. The details about the exact methodologies are described in Section 4.3.

4.2 Computational Environment Setup

The analysis will be performed on a desktop PC with 12 *AMD Ryzen 9 7900X3D* cores running at 4.4 GHz and 64 GB of RAM. The GPU is a *NVIDIA GeForce RTX 4070 Ti SUPER* with 8448 cores and 16 GB of memory. The theoretical peak performance of the GPU for FP32 is 44.19 TFLOPS and 689.0 GFLOPS for FP64, which corresponds to a ratio of 64:1. The code will be implemented using version 12.3 of the CUDA Toolkit with GPU driver version 571.96. The CUDA code is compiled using compiler flag *use_fast_math*.

As described in Section 3.3 the base algorithms used in the pressure solver have been known and studied for years, with that come state-of-the-art methods and libraries that can be used in any application. This section describes the libraries and external resources that will be used in the implementation.

cuFFT

In this research the state-of-the-art GPU FFT library *cuFFT* can be used to compute the FFTs needed for the solution of Equation 2.6. It is based on the Cooley–Tukey algorithm, which is an $\mathcal{O}(n \log n)$ algorithm. cuFFT can efficiently compute FFTs with various input sizes, but the library is optimised for input sizes of the form $2^a \times 3^b \times 5^c \times 7^d$, and generally performs better with smaller prime factors. Half, single, and double precision are supported, but lower precision generally achieves higher performance. The numerical error increases with $log_2(N)$, where N is the input size. In order to perform an

FFT, a plan must be created to allocate space for the computations. In the worst case, $8 \times batch \times n_0..n_{rank-1}$ elements are required, in the best case, $1 \times batch \times n_0..n_{rank-1}$ elements are required. Here, n_{rank-1} represents the number of elements in each dimension (rank) [74].

NVIDIA Nsight Suite

The NVIDIA Nsight Suite is a collection of developer tools that can be used to build, debug, and profile applications. In this study, two tools from this suite are used: *Nsight Systems* is a system-wide performance analysis tool [19]. It will be used throughout the implementation process to visualise the application's algorithms on a timeline, thereby guiding the optimisation process. *Nsight Compute* will also be used during the implementation, as it provides detailed performance metrics on a kernel-by-kernel basis and generates a detailed report on the optimisation potential of each profiled kernel [20].

4.3 Experiment Design and Evaluation

This section describes the methods that will be used to evaluate the implemented pressure solver in order to answer the set of research questions. Research question RQ1 focuses on the effect of the transition to an optimised single workstation PC with GPU on the performance. For that all three of the selected performance measures described in Section 4.1 need to be addressed. The next three subsections (Subsection 4.3.1, Subsection 4.3.2, Subsection 4.3.3) focus on each performance measure individually to describe the methods that are going to be used to analyse the performance in order to answer the formulated research question. To answer the research questions RQ2, RQ3 and RQ4, Subsection 4.3.1 explains how the speed up performance measure will be used to identify the effect of the single module optimisation on the overall runtime of the model (RQ2), the effects of memory management (RQ3) and possible bottlenecks (RQ4). The methodology chosen to answer the final research question (RQ5) is described in Subsection 4.3.4.

4.3.1 Speed Up Evaluation Methodology

The first of the three performance measures is *speed up* as it quantifies the difference between the runtime of the optimised GPU module and the original PALM module. A scalability analysis will be performed to gain a comprehensive understanding of how the achieved runtime scales with increasing workload. For this reason, a variety of realistic domain sizes will be analysed for the final application. These domain sizes range from a base area of $50m \times 50m$ to $1024m \times 1024m$ and are chosen according to microclimate studies described in Section 3.1. The scales can be grouped into three categories: Building, Site/Block and Neighbourhood, according to definitions in Urban Design [75, 76]. As the vertical extent should be at least twice the height of the tallest building, as outlined in Section 3.1, a fixed vertical domain size of $128m$ is set. Since the GPU (Section 2.5) and cuFFT (Section 4.2) achieve optimal performance when the computational domain sizes

are powers of two (i.e. 2^x), the chosen domain sizes are a combination of domains that meet this condition and those that do not. A grid spacing of $1m$ is chosen in all directions for all domain sizes. Table 4.1 gives an overview of all the domain sizes selected for analysis. The measurements are going to be conducted using the real-world topography of Cologne in the scenarify framework, with a suitable location chosen for each domain size.

Group		Nx [m]	Ny [m]	Nz [m]	N [m^3]
Building		50	50	128	
		64	64	128	2^{19}
Site/Block		100	100	128	
		128	128	128	2^{21}
Neighbourhood	*small*	200	200	128	
		256	256	128	2^{23}
Neighbourhood	*medium*	512	512	128	2^{25}
		600	600	128	
Neighbourhood	*large*	1024	1024	128	2^{27}

Table 4.1: Table of the selected domain sizes for the speed up performance analysis.

To quantify the speed up, the wall clock time for both versions is going to be measured using the PALM internal measurement system, which uses the Fortran intrinsic *SYSTEM_CLOCK* timing function. The measurements for the CPU and GPU versions are both going to be taken inside the original Fortran code, such that the full execution time of the module is measured. For the optimised GPU version this includes the calls to the C bind-in functions and the copy of the data to the GPU. When timing CUDA programs with a CPU measurement technique, it is important to ensure that the CPU and GPU are synchronised before taking the measurement.

To get representative measurements in order to answer RQ1, the simulation will be run five times for 10 seconds each in each domain, with a step size of 1 second. However, as the CFL condition is used to adjust the internal time step size, the actual internal calculated steps are expected to vary depending on domain size and topography. Due to computational limitations, for the largest domain size of $1024m \times 1024m \times 128m$, only 5 seconds are going to be calculated.

In order to answer RQ2, the impact of the optimisations on the overall runtime of a full simulation step will be measured using the same timing framework as described above. In the original sequentially executed implementation, the solving of the Poisson equation accounts for 15% of a simulation step. The aim is to measure whether this percentage can be reduced and what effect this would have on the overall runtime. This will provide

insight into whether optimising one module can affect the overall simulation time. For this analysis, a domain size of $512m \times 512m \times 128m$ is chosen.

Different operations are expected to be affected by GPU optimisations in different ways, particularly communication operations between the GPU and CPU and computation operations performed on the GPU. To isolate the effect of memory movement (RQ3) and to identify possible bottlenecks (RQ4), the code is divided into four parts, which are categorised based on the analysis in Section 2.4. The speed up will be analysed for each operation category. Additionally, the percentage of execution time attributed to each section will be compared between the original Fortran code and the optimised code. The CPU measurement method described above will be used to measure the execution times of the Fortran code, and CUDA events will be used to accurately measure the wall clock of individual sections of the GPU code. This analysis will be conducted for the representative domain size of $512m \times 512m \times 128m$.

4.3.2 Methodology for Validity Assessment

As the aim of this research is to accelerate the runtime of the serial execution of the scientifically validated PALM model without compromising accuracy, the numerical error will be calculated to determine whether the implemented algorithm produces the same results as the original Fortran-based version. For this purpose, the relative L^2 norm error is chosen. This error is calculated using Equation 4.1, where $\mathbf{u}$ is the result of the Fortran code and $\tilde{\mathbf{u}}$ the result of the optimised C++ and CUDA code.

$$\frac{\|\mathbf{u} - \tilde{\mathbf{u}}\|_2}{\|\mathbf{u}\|_2} \quad , \quad \|\mathbf{u}\|_2 = \sqrt{\sum_{k=1}^{N} |u_k|^2}. \tag{4.1}$$

The error will be measured for the modified perturbation pressure at the end of the Poisson solver, in order to analyse the numerical accuracy of the method for different domain sizes and will be compared to the machine epsilon at the given precision. The values for the machine epsilon can be seen in Table 4.2 [77].

Data Type	Machine epsilon value
single precision	1.192093×10^{-7}
double precision	2.220446×10^{-16}

Table 4.2: Table of the used machine epsilon for comparison [77].

To quantify the immediate effect of the error in the pressure calculation on the velocity output, the L^2 error and the maximum pointwise difference will be calculated for wind velocities $\mathbf{u}, \mathbf{v}, \mathbf{w}$ after 10 seconds of simulation time. As the velocity output is given in single precision, only differences visible at that level of precision will be measured.

To ensure that the GPU-based pressure solver does not cause instability during longer model runs, a ten hour simulation will be executed on a domain of size $128 \times 128m \times 128m$ due to computational limitations when running the non-optimised full model. If an error in the pressure calculation results in an explosion in velocity, the CFL condition, which ensures numerical stability, would reduce the size of PALM's internal time step. For this reason, the number of internal simulation steps will be counted and compared. The resulting wind speed, as calculated with Equation 4.2, and the wind speed components $\mathbf{u}, \mathbf{v}, \mathbf{w}$, will then be compared to that of a model run with the original pressure solver. Three statistical measures, taken from a paper by Resler et al. [78] in which PALM-4U was validated against real-world observations, will be used as a reference when analysing the effect of optimisations on model outputs: Normalized Mean Squared Error (NMSE), Fractional Bias (FB), Pearson Correlation Coefficient (R).

$$\mathrm{X} = \sqrt{u^2 + v^2 + w^2} \tag{4.2}$$

$$\mathrm{FB}_X = 2 \cdot \frac{\overline{X_{\mathrm{GPU32}} - X_{\mathrm{original}}}}{\overline{X_{\mathrm{GPU32}} + X_{\mathrm{original}}}} \tag{4.3}$$

$$\mathrm{NMSE}_X = \frac{\overline{(X_{\mathrm{GPU32}} - X_{\mathrm{original}})^2}}{\overline{X_{\mathrm{GPU32}}} \cdot \overline{X_{\mathrm{original}}}} \tag{4.4}$$

$$R = \frac{\mathrm{cov}(X_{\mathrm{GPU32}}, X_{\mathrm{original}})}{\sigma_{X_{\mathrm{GPU32}}} \cdot \sigma_{X_{\mathrm{original}}}} \tag{4.5}$$

Here, X_{GPU32} denotes the optimised pressure solver run on the GPU in single precision, while X_{original} denotes the use of the original PALM pressure solver. $\bar{X}$ indicates the arithmetic mean.

4.3.3 Memory Profiling Methodology

Since GPUs have limited memory capacity, the memory footprint of the derived methods is going to be analysed and compared to that of the original algorithm as part of the performance comparison to answer research question RQ1.

As three-dimensional arrays set on the computational grid account for most of the required memory, to estimate the memory footprint, the number of three-dimensional arrays initialised and used in each method will be counted to calculate the memory required per computational cell. As external libraries will be used in the GPU version, the memory reserved for each function call is also taken into account. To put the memory required by the Poisson solver into perspective, the amount of memory required by the full PALM model will be calculated. To find the number of arrays, regex in combination with Python will be used to identify all allocated unique three- or four- dimensional arrays.

4.3.4 Approach to Assessing Full Model Optimization Feasibility

In order to answer research question RQ5, the feasibility of full model optimisation will be analysed. For that the characteristics of the pressure solver that led to its selection for this research will be analysed for the full model and evaluated in the context of the optimisation carried out.

The characteristics that will be analysed here can be divided into three categories:

- *Connectivity*
- *Memory requirement*
- *HPC optimisations*

The first category to be analysed is the dependency of the selected module on others with regard to variables and methods. High connectivity means that, if not all computation is performed on the GPU, a significant number of communication steps between the GPU and CPU are necessary. In order to understand how the files are connected and which modules use global arrays, all Fortran files in the codebase that are called during time stepping will be analysed using Python in order to count the loaded modules. Particular focus will be given to the 3D arrays module, as it includes arrays on the computational grid, which are updated with each time step.

The next parameters to be analysed are the number of three- and four-dimensional arrays used in each module. This analysis will provide insight into the memory footprint of each module, since the arrays defined on the computational grid are the main contributor to the required memory. These arrays can be categorised as either allocated inside the module or loaded from the outside. This distinction will also help identify how many arrays need to be copied to and from the GPU at each time step if the rest of the model remains on the CPU, and how many temporary arrays the module requires for its calculations.

The PALM codebase was originally designed to operate on HPC systems, using MPI (Message Passing Interface) for inter-node communication. During integration into a precompiled library for a single CPU system, these MPI calls will become obsolete and can therefore be removed. Despite the removal of these MPI routines, the code's overall structure remains tailored to the HPC cluster, adding optimisation potential when changing the target architecture. To identify routines involving MPI communication, the original PALM source code will be analysed to provide an overview of MPI-heavy routines by counting the number of times that *CALL MPI* is written in the code, as well as the number of calls to the exchange boundaries module, for each file. As *MPI barriers* can stall execution, they will be counted separately.

CHAPTER 5

Implementation

In this chapter the implementation details and the integration into scenarify framework are given. First, the setup is explained, then, the optimisations made to the original code to fit the new target architecture are described. Next, the preliminary implementation is optimised step by step. As NVIDIA Nsight Compute was used to guide the optimisations, the profiler results are presented to visualise the effect of the optimisations.

5.1 Setup

The setup can be separated into two sections, first it must be decided which parts of the pressure solver are suitable for optimisation at this stage. Then the interface to combine the optimised C++/CUDA code with the original FORTRAN code needs to be setup.

5.1.1 Intersection with Original Code

As the transfer between GPU and CPU is expensive, the goal is to adapt or replace as much code as possible to enable an optimised GPU single desktop PC method for the pressure solver. As the rest of the codebase will still reside on the CPU for now, it is important to choose a fitting entry point for the optimisation of the pressure solver.

As described in Subsection 4.3.4, specific characteristics can be used to identify the optimal intersection point for integration into the original code. To achieve this, the files connected to the pressure solver are analysed. As for the FFT algorithm, cuFFT will be used, the two files that compute the FFT with different algorithms are ignored here. The results can be seen in Table 5.1.

From this analysis it can be seen that only file *pre.f90* needs the globally allocated arrays defined on the computational grid at each time step. File *poisfft_mod.f90* is not connected to the global 3D arrays and only needs one external array from *pres.f90*. The

Filename	connected modules	uses global arrays	# external arrays	# temporary arrays	# MPI calls	# MPI barriers
pres.f90	14	Yes	6	0	10	4
poisfft mod	9	No	1	7	1	0
transpose mod f90	3	No	2	0	8	8
fftx mod f90	8	No	5	0	0	0
tridia solver mod	7	at init	3	2	0	0

Table 5.1: Files connected to the FFT pressure solver, showing for each: the number of loaded modules, whether the 3D Arrays module is included, the number of external three or four dimensional arrays, the number of temporary arrays allocated and the number of inter-node communication.

MPI communication steps are evenly distributed throughout the execution of the pressure solver, creating regular barriers in the process indicating a code section that was heavily optimised for a distributed computing system.

The pressure calculation starts as described in file *pres.f90*. However as this file has a relatively high number of connected modules and needs multiple global arrays for the calculation of the perturbation pressure, in the context of this research, this calculation is kept on the CPU. Only the resulting array is copied to the GPU and handed to the subroutine *poisfft*, which using `transpos_mod.f90`, `fftx_mod.f90` and `trifia_solver_mod.f9` solves the Poisson equation. This means that only one array needs to be copied to and from the GPU at each time step. After solving the Poisson equation, only one array is returned to the initial pressure file to apply the boundary condition and continue with time stepping.

5.1.2 FORTRAN - C++ Integration

Since PALM is written in FORTRAN 90 and the objective is to optimise and adapt it using CUDA C++, it was necessary to establish an interface that bridges the gap between FORTRAN and CUDA C++. This work focusses on CUDA C++ and not CUDA FORTRAN to explore optimisation opportunities, particularity because the target architecture is Windows-based and the HPK-SDK Cuda FORTRAN Compiler is only available for Linux-based systems. Additionally, the target application is a C++ codebase. For this reason PALM is used as a pre-compiled Fortran library to allow for steering from a C++ environment. The optimisations implemented are also written in C++ and CUDA.

To achieve this interaction, multiple steps were performed. On the C++ side, a standard function with the desired functionality was implemented. In the Algorithm 1, this function is called `FunctionInCpp`. Additionally, a function pointer (`setFunction`)

was created to hold the address of the C++ function. As the subroutine is going to be called from Fortran, a Fortran procedure pointer with the corresponding subroutine was created with C binding to ensure C-compatible linkage. This can be seen in Algorithm 2 as `native` in lines 1–7. A second Fortran subroutine was created that can be called from a C++ environment, accepting a C function pointer as input and assigning it to the Fortran procedure pointer (`SetCallback`), using `c_f_procpointer`. To integrate C++ and Fortran, the Fortran setter routine was loaded and called in C++, passing the C++ function pointer. These steps are split into functions `loadFunctionPointer` and `setFunctionPointer` in the Algorithm 1 [79]. This made it possible to test different versions of the GPU code during the implementation process by simply setting the function pointer to the desired version at runtime.

Algorithm 1 C++ Function Pointer Setup and Usage

Require: C++ function pointer `setFunction`
Require: Fortran binding function pointer `externSetFunction`
Require: C++ callback function `FunctionInCpp`
Require: Shared library handle `lib`

```
1: procedure LOADFUNCTIONPOINTER
2:     setFunction ← reinterpret_cast<GetProcAddress(lib,
   "externSetFunction")>
3: end procedure

4: procedure SETFUNCTIONPOINTER
5:     setFunction(FunctionInCpp)
6: end procedure
```

The C binding allows C-style pointers, which are initialised during PALM execution, to be passed to C++ as input and output. However, variables that only need to exist in the C++ context do not need to be exchanged and can be initialised in the C++ environment. The same applies to CUDA arrays. This decoupling is necessary because the compiler used to build the pre-compiled PALM library is not CUDA-compatible. A global namespace has been created in which all C++-only arrays are initialised, eliminating the need to allocate and deallocate CUDA arrays with each C++ function call. A visualisation can be seen in Algorithm Figure 5.1.

With that, the full integration between Fortran and C++ was achieved, allowing for the adaptation and optimisation of the selected section.

Algorithm 2 Fortran Procedure Setup and Invocation

Require: Callback procedure `native(array)`
Require: Target array `array(:,:,:)`

```
1:  PROCEDURE(native), POINTER :: callback
2:  procedure NATIVE(array)
3:      USE, INTRINSIC :: iso_c_binding
4:      USE kinds
5:      IMPLICIT NONE
6:      REAL(wp), INTENT(INOUT), DIMENSION(*) :: array
7:  end procedure

8:  procedure SETCALLBACK(new_callback)
9:      bind(C, name="externSetFunction")
10:     callback ← c_f_procpointer(new_callback)
11: end procedure

12: procedure FUNCTION(array)
13:     call callback(array)
14: end procedure
```

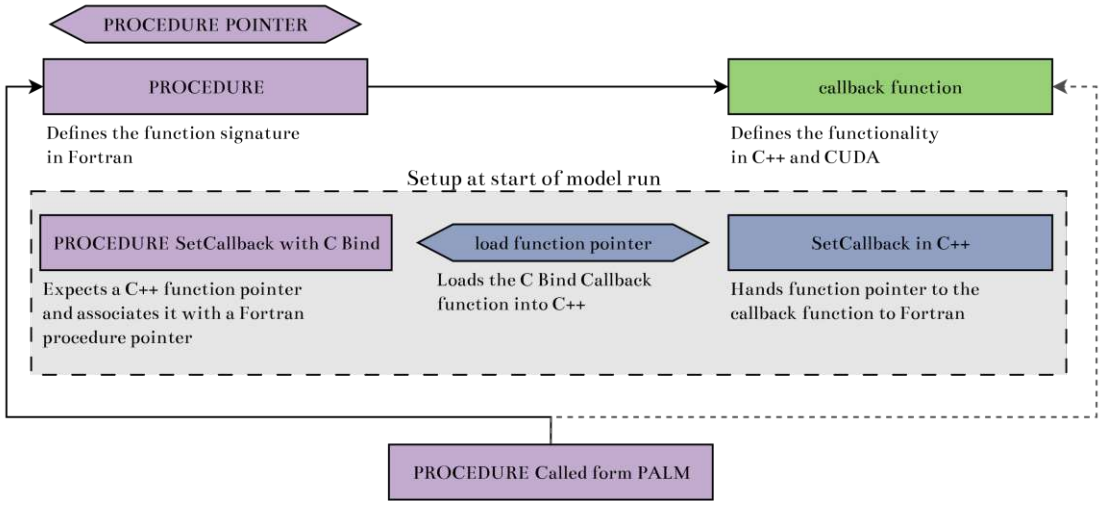


Figure 5.1: Visualisation of the Fortran and C++ Integration process.

5.2 Preliminary Implementation

This analysis shifts the target architecture of PALM from high-performance computing clusters to single desktop PC and GPU. The initial optimisations presented here focus on parts of the code that have become redundant due to this shift in architecture, with any function calls that were solely required for HPC execution having been removed.

When running the solver on an HPC cluster, the order of the array elements is important for the efficiency of the MPI communication. For that reason the array elements originally had to be rearranged before each transposition. However, as the full array now resides on one node, these code sections are no longer necessary. Figure 5.2 shows the new code structure. The crosses indicate the six dropped function calls used to rearrange the data, as opposed to the structure shown in Figure 2.7.

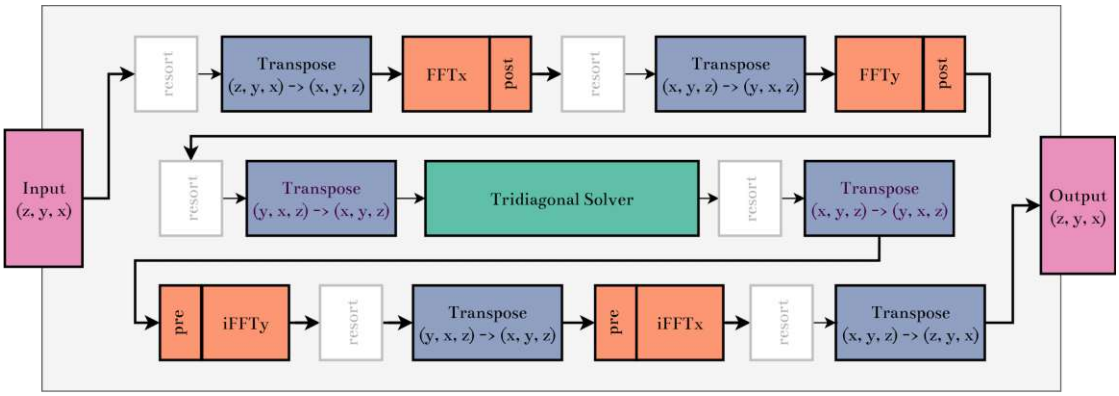


Figure 5.2: The timeline shows the structure of the Poisson FFT solver. The greyed-out blocks indicate the dropped function calls used to rearrange the data, as opposed to the structure shown in Figure 2.7.

Following the initial modifications, a preliminary version was implemented by converting several lines of Fortran code into C++ and CUDA. As this implementation only served as a baseline, the original code was not optimised for the GPU. Except for the above-described changes, its structure remains the same as that of the original Fortran implementation.

The right-hand side of Equation 2.6 is calculated in the Fortran code using the preliminary velocities and copied to the GPU at the start and end of the method. The cuFFT library, similar to the implementation of Knoop et al. [17], was used for the Fast Fourier Transform. This involved creating a one-dimensional (1D) batched plan for each Fast Fourier Transform (FFT) dimension (x and y) and direction (forward and inverse). All the necessary information about the chosen transform is saved in each plan, such as the domain size and batch size. These plans are executed at each time step, as shown in Figure 5.2. After each forward FFT, the resulting complex array must be converted to real numbers, and before each inverse FFT, it must be converted to complex numbers. A custom CUDA kernel was implemented for this purpose, translating the original loop

into CUDA code. As Jodra et al. concluded that although the cuFFT library has the option to perform FFTs on non-contiguous array elements, it is more efficient to use batched one-dimensional FFTs on contiguous data with optimised three-dimensional transpositions in-between, this approach was chosen [80]. The necessary transpositions were implemented as CUDA kernels, however in the initial implementation not GPU optimised. Finally the tridiagonal solver was translated to GPU code.

5.3 Data Types and Memory Optimization

The GPU architecture has been optimised for specific data types, and the device memory is limited. This section details the optimisations made to the initial implementation in order to account for this.

5.3.1 Enabling Float Support

The PALM model system, as it is used in the C++ interface, can only be run in double precision without substantial changes throughout the model system, which is outside the scope of this study. However, as described in Section 2.5 and Section 4.2, the GPU hardware and with it cuFFT is much better optimised for single precision. For this reason, the first optimisation involved enabling single precision calculations for the Poisson solver on the GPU. This was achieved by casting the array elements from double to single precision at each time step, and then back again after the GPU section had finished and the array was handed back to PALM. All functions that are executed on the GPU are templated so that they are compatible with both types. Each subsequent implementation was run and analysed in both single and double precision.

To convert from double to float, the input array is cast on the CPU using OpenMP to enable parallel computation. Although a CUDA kernel could perform naive casting with higher parallelism, this approach was chosen because it reduces the number of bytes transferred between the host and the device at each time step, halving the required transfer time. Additionally, if the calculation is performed using single-precision arithmetic, less memory needs to be allocated to the GPU, since no additional array of doubles is required on the device, aside from the single-precision array.

To accelerate memory transfer further, the array is cast onto a page-locked, pinned memory array. As described in Section 2.5, copying memory from a page-locked position is faster compared to pageable memory and the transfer can be executed asynchronously, enabling the CPU computation and the copying process to overlap. This effect is visualised in Figure 5.3. The first line shows a synchronous execution, where the light grey area represents the casting operation on the CPU and the dark block represents the memory transfer, which is divided into the transfer to the page-locked memory buffer (black) and the transfer to the device (pink). In asynchronous execution, the copy to the buffer as part of the copying process can be skipped, as the data are already cast in pinned

memory and with that, cast and copy can overlap. 3 shows dummy code that explains the process.

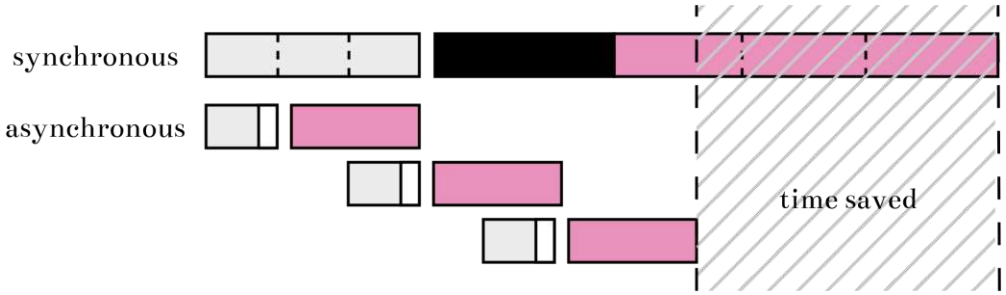


Figure 5.3: A comparison of synchronous and asynchronous execution. In the synchronous case (top), the light grey block represents the casting operation on the CPU, followed by a memory transfer to a page-locked (pinned) memory buffer (black) and then to the device (pink). In the asynchronous case (bottom), the casting operation occurs directly in the pinned memory. This allows the casting and memory transfer to overlap, eliminating the intermediate copy step.

Algorithm 3 Cast and Copy to GPU

Require: Nx, Ny, Nz
Ensure: Processed data on *deviceSamples*

$N \leftarrow Nx \times Ny \times Nz$
$sectionSize \leftarrow \left\lceil \frac{N}{numSections} \right\rceil$

for $s = 0$ **to** *numSections* - 1 **do**
    $offset \leftarrow s \times sectionSize$
    $correctSize \leftarrow \min(sectionSize, N - offset)$
    Convert samples[$globalIdx + j$] for $j = 0$ to $correctSize - 1$
    Asynch. copy samplesFP32[offset] to deviceSamples[offset + correctSize]
end for

5.3.2 Memory Optimisations

The first implementation revealed two primary contributors to the memory consumption of the method. Methods to address and minimise their needed memory are discussed in this section.

Firstly, following the architectural shift, extensive temporary memory allocations were no longer necessary. The extra memory was only required when the array was split between different nodes in an HPC cluster, with each node processing arrays of different sizes depending on the coordinate permutation. However, as the aim was to optimise the solver for a single node, the entire three-dimensional array is now stored as a single, flattened

one-dimensional array. This means that the memory requirement does not change with each transposition and that a single temporary array can be used for all out-of-place transpositions, or whenever a temporary array is needed in the new implementation.

Secondly, in the initial implementation, a cuFFT plan is created for each version of the Fast Fourier Transform (FFT) that needs to be executed. At the initiation of the plan, each cuFFT plan allocates the memory space required for that specific Fast Fourier Transform during execution. This memory space can range from one to eight times the size of the input array, as described in Section 4.2, but it is only used during the computation of that specific FFT plan. To reduce the memory footprint of the FFT calculation, it is possible to point all four required cuFFT plans to the same workspace, thus reducing the allocated memory by a factor of four. This is possible because the plans are only ever called in sequence. The optimisation is implemented using the cuFFT API, as shown in 4 [74].

Algorithm 4 Optimized cuFFT Plan Initialization

Require: N_x, N_y, N_z

```
// Create plans and disable auto allocation
for all plan in plans do
    cuFFTCreate(plan)
    cuFFTSetAutoAllocation(*plan, 0)
    cuFFTMakePlan(plan, ..., size)
end for
// Find maximum workspace size and allocate once
workSize ← max(size for all plans)
cudaMalloc(&workArea, workSize)
// Assign the same pointer to all plans
for all plan in plans do
    cuFFTSetWorkArea(*plan, &workArea)
end for
```

5.4 Algorithmic Optimization for GPU

This section provides an overview of the optimisation of individual operations. It starts with the optimisation of the pre- and post-processing kernel, where a change in parallelism granularity was performed to increase performance. Next, the optimised transposition algorithm, implemented based on methods described in the literature, is presented. The final subsection outlines the optimisation process of the tridiagonal matrix solver.

5.4.1 Pre-Processing and Post-Processing Acceleration

After each forward Fast Fourier Transform (FFT) and before each inverse FFT, the array must be converted from real to complex, or vice versa. In the initial implementation, the

original OpenMP-accelerated sorting loop was converted to CUDA, without adapting for that architecture. This means each thread sorted one full row. However, this approach restricts the number of CUDA threads that can be utilised to the size of the batch, significantly limiting parallelisability and, consequently, the achievable performance.

To optimise this kernel, a new method was developed where each thread moves a single array element, rather than a full row, from its original position to its new position in the array. This technique utilises parallelisation capabilities of the GPU by creating an equal number of threads to the elements of the input array, which then work in parallel. The algorithm, shown in 5, illustrates how real values are assigned to the correct locations in the complex array. Figure 5.4 shows a visualisation of the variables used. First, the thread index is calculated. From this and the size of each block, the index of the element in the block and the index of the block itself can be calculated. These can then be used to determine the correct location to which the current element needs to be moved. As the values are complex conjugate pairs, the complex part of the array is sorted from the end of each batch.

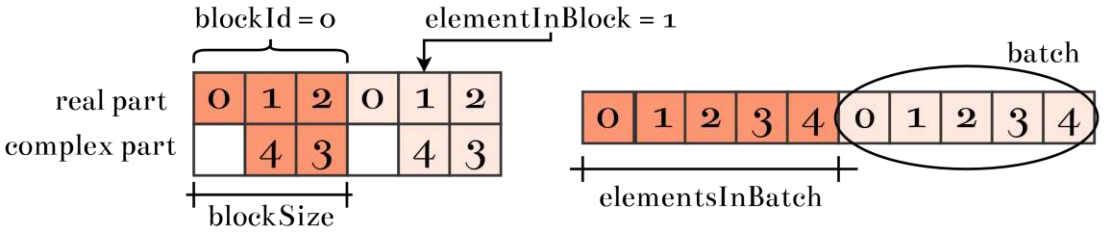


Figure 5.4: Visualisation of the variable definitions used in 5. The figure on the left shows the complex number resulting from a forward fast Fourier transform (FFT), or the pre-processed input for an inverse FFT. The figure on the right shows the real number after post-processing, or before pre-processing.

Algorithm 5 createComplexData

Require: $complexDevice, realDevice, batches, elementsInBatch$

```
idx ← threadIdx.x + blockIdx.x × blockDim.x
blockSize ← elementsInBatch/2 + 1
// if idx in the first section = real values
if idx < batches × blockSize then
    elementInBlock ← ⌊idx/elementsInBatch⌋
    blockId ← ⌊idx/elementsInBatch⌋
    if elementInBlock < blockSize then
        idxReal ← elementInBlock + blockSize × blockId
        complexDevice[idxReal].x ← realDevice[idx]
    else
        idxComplex ← elementsInBatch − elementInBlock + blockSize × blockId
        complexDevice[idxComplex].y ← realDevice[idx]
    end if
end if
```

Nsight Compute was used to profile the initial and the optimised kernel to compare their performance. The results of the runtime comparison are shown in Table 5.2. As the primary function of the kernel is to rearrange data, this results in high memory traffic and relatively low arithmetic intensity, which leads to both versions of the kernel being memory-bound. However, the new kernel achieves a higher compute throughput. Additionally, depending on domain size, the new version allows for an approximately 99% increase in the number of threads launched. Overall, the new version delivers a clear performance increase, achieving a maximum speed up of up to 6.4 times.

Function	Initial [ms]	New [ms]	speed up (×)
Postprocessing			
FP32	2.02	0.39	5.2
FP64	2.62±0.13	1.26±0.2	2.1
Preprocessing			
FP32	2.61±0.4	0.41	6.4
FP64	2.05±0.1	0.84±0.01	2.4

Table 5.2: Comparison of performance metrics between old and new implementations for FP32 and FP64 precision.

5.4.2 Transpose Optimization

In the preliminary implementation described in Section 5.2, the transposition kernel was implemented without taking the GPU architecture into account. This section describes

how it was adapted for GPU implementation. Previous works by Jodra et al. [66] and Ruetsch and Micikevicius [81] were used as a guide for a custom reimplementation.

Transpositions are defined as permutations of the x, y and z axes of the array. There are six possible permutations for a 3D array, which can be grouped into three categories. The trivial identity permutation ($T(A) = A$), three involution transpositions ($T_{xyz}(T_{yxz/xzy/zyx}(A)) = A$) and two rotation transpositions ($T_{xyz}(T_{zxy}(T_{yzx}(A))) = A$, $T_{xyz}(T_{yzx}(T_{zxy}(A))) = A$) [66]. Two types of transposition, both falling into the category of involution transpositions, are required for the pressure solver of PALM, they can be seen in Figure 5.5.

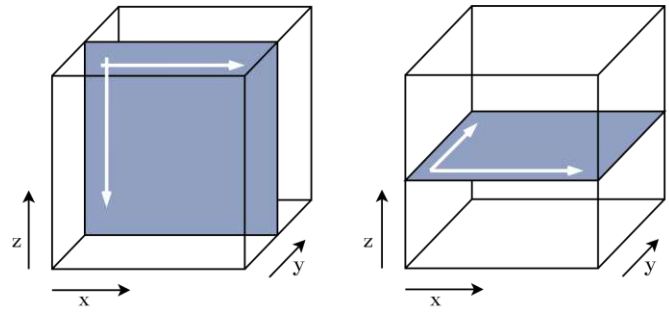


Figure 5.5: Figure of the two needed involution transpositions of the three-dimensional array.

Ruetsch and Micikevicius describe an optimised transposition method for two-dimensional matrices, the principles of which can be applied to three-dimensional arrays [81]. This proposed optimised 2D matrix transposition method exploits the knowledge of the GPU architecture to optimise the kernels for coalesced global memory access. More specifically, the method decomposes the matrix into square shared memory tiles that are used as temporary memory within a thread block. Threads read coalesced from global memory and write into the rows of the shared memory tile. When a thread block has completed this first sweep, threads read columns from shared memory to write them back to the new location in global memory, again in a coalesced manner. However, with square tiles this leads to memory bank conflicts when a number of threads want to access a column of the shared memory at the same time. To avoid this, padding the memory tile was proposed and the dimension of the tile changed to $32 \times (32 + 1)$ [81].

Since three-dimensional involution transpositions can also be seen as transpositions of a set of two-dimensional planes, the principles of Ruetsch and Micikevicius's two-dimensional matrix transposition can be applied [66]. The algorithm can be seen in pseudocode in 6. Figure 5.6 visualizes how each plane can be placed inside the three-dimensional array. The difference between this execution and the previous work described is that compared to works which only benchmarked cubic matrices, this work deals with mainly cuboids matrices as the vertical dimension is mostly smaller than the two horizontal dimensions.

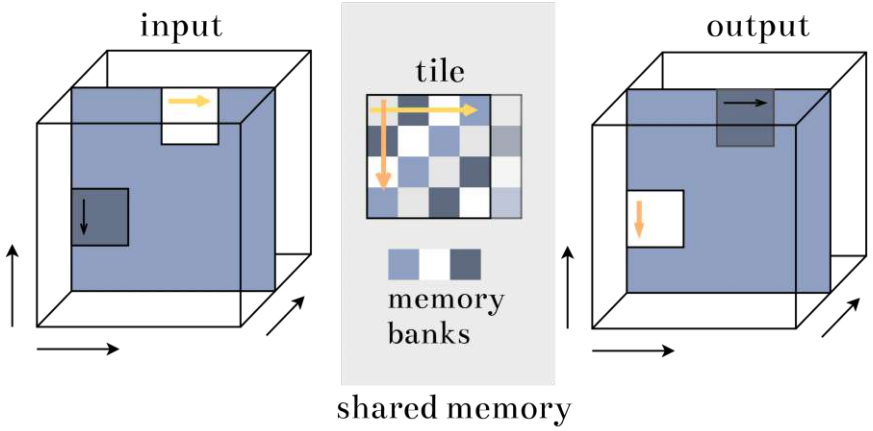


Figure 5.6: Figure visualising the three-dimensional transposition algorithm, using a shared memory tile with padding.

Algorithm 6 Transpose_xyz2yxz

Require: *idata*: 3D input array (x, y, z)
Ensure: *odata*: 3D output array (y, x, z)

```
1: Allocate tile[TILE_DIM][TILE_DIM+1] in shared memory
2: Compute index_in, index_out from thread and block indices
3:
4: val ← 0
5: if access is valid then
6:     val ← idata[index_in]
7: end if
8: tile[threadIdx.y][threadIdx.x] ← val
9:
10: ___syncthreads()
11:
12: if access is valid then
13:     odata[index_out] ← tile[threadIdx.x][threadIdx.y]
14: end if
```

To see if the optimisation has had any effect Nsight Compute was used, the resulting calculated speed up is shown in Table 5.3. Although the new version has an increased amount of shared memory, which limits the 'occupancy' of the kernels, it performs better due to improvements in memory access and the absence of bank conflicts. As the implemented transposition kernel is optimised for memory access in single precision, the transposition time for both required involution transpositions could be nearly halved.

During the implementation process the cuTT library, which is briefly mentioned in

Function	Initial [ms]	New [ms]	speed up (×)
xyz2yxz			
FP32	0.84	0.47	1.8
FP64	0.84±0.01	0.83±0.01	1.01
xyz2zyx			
FP32	0.83±0.1	0.49	1.7
FP64	1±0.1	0.84±0.01	1.2

Table 5.3: Comparison of performance between old and new implementations of the transposition kernels for FP32 and FP64 precision.

Section 3.3, was tested, however using NVIDIA Nsight it was found that before each call the kernel called a preprocessor which degrades performance.

5.4.3 Tridiagonal Solver GPU Adaptation

In PALM to solve the tridiagonal matrix the Thomas algorithm was chosen. In the preliminary implementation the standard Thomas algorithm which pre-computes coefficients was implemented for the GPU. To make sure that this was the optimal solver different implementations were tested. As described in Section 3.3 the NVIDIA linear algebra library for sparse matrices called cuSPARSE includes an optimized Thomas algorithm for GPUs called cuThomas first presented by Valero-Lara et al. [73] as well as an algorithm using Cyclic Reduction and Parallel Cyclic Reduction [72].

Both algorithms were tested in batch mode. As cuThomas needs fewer internal steps it performed better than the CR/PCR in a comparison of single batch execution. Yet this algorithm overwrites part of the tridiagonal matrix with each function call which must be reset during each time step. Consequently, it is not suitable for repeated usage with a fixed tridiagonal matrix, such as in the Poisson solver implemented in this research. Additionally, as neither algorithm was optimised to work with constant upper and lower diagonals for each equation, but rather with different tridiagonal matrices, both had a larger memory footprint than the initially implemented kernel. Furthermore, both versions performed the same as or worse than the custom kernel.

For this reason, the self-implemented Thomas algorithm kernel, which is derived from the original PALM Fortran function was chosen for the GPU-optimised version. However as the array now resides entirely on the GPU, the structure around the Thomas algorithm could be adapted to exploit this. In the original implementation, the coefficients of the Thomas algorithm are initialised with x in the leading dimension. This means that the Fourier-transformed array must be transposed before the tridiagonal solver is applied, and transposed back afterwards. These transpositions could be eliminated by adapting the Thomas algorithm to work with y in the leading dimension. Based on the

initial implementation, this structural change made it possible to reduce the number of transposition operations from six to four.

5.5 Resulting Structure and Limitations

The final pipeline of the optimised FFT-based Poisson pressure solver can be seen in Figure 5.7. The greyed-out blocks indicate code elements that could be eliminated during the transition from the HPC-optimised code to an optimised version for a single workstation combined with a GPU. More than 40% of the original number of steps could be removed, ignoring the initial and final memory transactions, this translates to only 53% of the steps remaining in the optimised version.

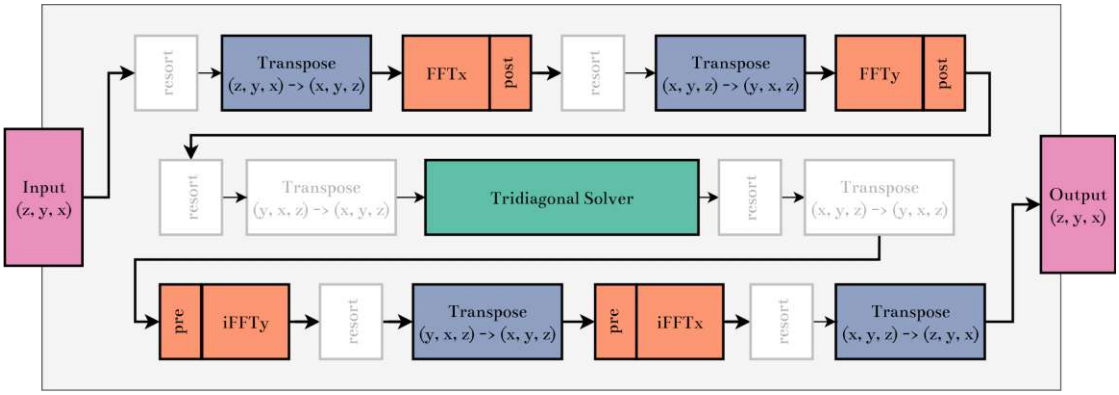


Figure 5.7: The solver structure of the GPU and standard workstation implementation after the changes for the new architecture.

To see how this optimisation fits into the otherwise unchanged PALM model code, Table 5.4 shows all the files connected to the FFT pressure solver and the extent to which they were altered by the optimisations presented in this research.

Filename	Not Changed	Adapted	Replaced
pres.f90	X		
poisfft_mod.f90		X	
transpose_mod.f90			X
tridia_solver_mod.f90			X
fft_xy.f90			X
temperton_fft_mod.f90			X
singleton_mod.f90			X

Table 5.4: This Table summarises the extent to which the files required for the FFT-based pressure calculation during time stepping were altered during optimisation.

As can be seen in Table 5.4, none of the files called by the Poisson FFT steering file (*poisfft_mod.f90*) are used in the new implementation. The Poisson FFT steering file itself has been adapted to call only C++ and CUDA functions. Only the *pres.f90* file was not altered. As described in Subsection 5.1.1, this intersection point was chosen because many external 3D arrays are required in this file for the calculation of the wind velocities from the perturbation pressure using Equation 2.4. If more of the PALM model would reside on the GPU, optimising and replacing this file would be the next step.

CHAPTER 6

Results

This chapter presents the benchmarking and the evaluation results of the implemented solver, based on the methodologies described in Section 4.3. It begins with a scalability analysis, examining the speed up achieved with this new method in three stages. This is followed by a validation of the results. Then, the memory footprint of the optimised version is compared to the original implementation. Finally the results of the full model analysis are presented.

As described in Chapter 4, a range of domain sizes and locations in the city of Cologne was selected for a representative analysis. These variations enabled the solver to be tested under different conditions. Figure 6.1 shows exemplary simulation results for different domain sizes and their locations, visualising the resulting wind velocities.

6.1 Speed Up Analysis

This section presents the results of the performance analysis, it is split into three sections. First, the results of the scalability analysis of the achieved speed up are presented. Next, the impact of the optimisations on various sections of the code is examined. Finally, the impact of the pressure solver speed up on the full model run is analysed.

6.1.1 Scalability Analysis

Figure 6.2 shows the performance of the FFT pressure solver, which has been optimised for a single desktop PC and GPU, compared to the original serial PALM execution. The y-axis displays the speed up of the measured wall clock time relative to the baseline execution. A speed up of one is indicated by a dashed line, showing an execution time identical to that of the original PALM version. The x-axis shows different domain sizes spanning multiple orders of magnitude. At the bottom, the number of cells in the complete domain is shown. As only square domains were chosen for this analysis, the

(a) Quartier am Gustav Heinemann Ufer Cologne - Domain 256 m × 256 m.

(b) Mainzer Street to "Die Bottmühle" Cologne - Domain 512 m × 512 m.

Figure 6.1: Two exemplary simulation domains are shown, where colours indicate wind velocity ranging from nearly zero, shown in green, over yellow and orange to blue, which indicates 50 per cent of the maximum simulated velocity. Velocities that exceed this threshold are not displayed.

x-axis at the top of the Figure shows the number of cells in the horizontal dimension, which spans from 50m to 1024m. The maximum domain size for the interactive simulation in the ClimaSense module is $1024m \times 1024m$, and larger sizes are deemed irrelevant. The exact dimensions chosen are summarised in Subsection 4.3.1. The colours indicate the calculation precision of the optimised version. Blue represents the speed up of the single precision GPU calculation, including casting and copying. Orange indicates the speed up when running the GPU version in double precision. Again, the wall clock time was measured, which includes the time taken to copy the needed memory to the GPU. The line style shows the type of domain size analysed: solid lines show domain sizes chosen to optimally fit the memory architecture of the GPU and dashed lines show the results of the model run on domain sizes not subject to this restriction. Thus, four different scaling benchmarks are displayed.

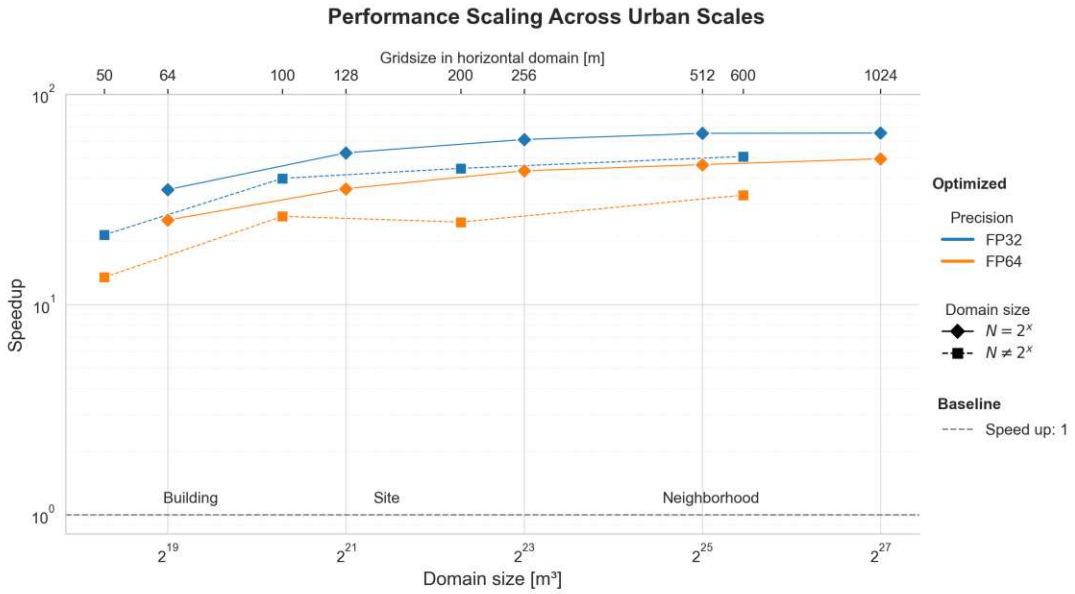


Figure 6.2: This Figure shows the results of the scalability analysis. Colours indicate the used date type and line style the domain type.

As can be seen in Figure 6.2, for all domain sizes and data types, the optimised version for the target architecture and GPU achieves a speed up compared to the baseline PALM implementation. For all domain sizes, the single precision calculation performed better than the double precision calculation. Additionally, for all tested domain sizes and both data types, simulations with the optimised domain size performed better than simulations on a non-GPU-optimised grid. Furthermore, for all tested domain sizes, simulations using single precision with edge lengths not divisible by 32 performed better than those using restricted, optimised domain sizes in double precision. The speed up is smallest for the smallest tested domain size and increases afterwards for all combinations.

Table 6.1 shows the maximum speed up over the baseline implementation achieved for

Precision	Domain Type	Max. speed up (×)
FP32	$N = 2^x$	65.4
FP32	$N \neq 2^x$	50.6
FP64	$N = 2^x$	49.3
FP64	$N \neq 2^x$	33.1

Table 6.1: The maximum achieved speed up compared to the original PALM implementation is shown for all combinations of precision and domain type.

each combination. The best overall performance was achieved with single precision runs and domain sizes that optimally fit the GPU. With this combination, a maximum speed up of 65.4 times was measured compared to the original sequential PALM execution. Next is the single precision execution with $N \neq 2^x$, achieving a maximum speed up of 50.6 times. The simulations in double precision then follow, with speed ups of 49.3 times and 33.1 times respectively.

6.1.2 Performance Comparison of Key Operations

As described in the Section 2.4 the operation of the Poisson FFT pressure solver can be categorised into four parts: memory operation including casting, operations connected to the three-dimensional matrix transposition, operations connected to the FFT and finally the tridiagonal matrix solver. In this section the effect of the optimisations on the different categories are presented. This analysis was done on the representative domain size of $512m \times 512m \times 128m$.

Figure 6.3 illustrates the percentage of each category of operations in the simulation for the original PALM solver and the single and double precision optimised versions. In the original version, 67.5% of the execution time can be attributed to operations connected to the transposition operations. The next largest section, accounting for 24%, is the execution of the FFT. The Thomas algorithm accounts for 7.5% of the execution time. Memory operations and other set-up tasks, such as the allocation and deallocation of temporary arrays, make up the smallest section at 1%.

Even though only one array needs to be copied to and from the GPU for the optimised execution, this section accounts for the largest proportion of the optimised version in both single and double precision. It was found that 67.2% of the single precision execution was taken up by the memory copy to the GPU and the casting from double to floats. The copying process takes 55.9% of the double precision execution. The FFT is second most expensive operation for both data types, however the percentage is much smaller for single precision than double precision. Transpositions, which account for more than half of the original execution time, only run for 11.7% and 7.7% of the execution time in the optimised cases. This was expected, as during optimisation for the single desktop PC, many of the function calls connected to transposition could be removed. The smallest

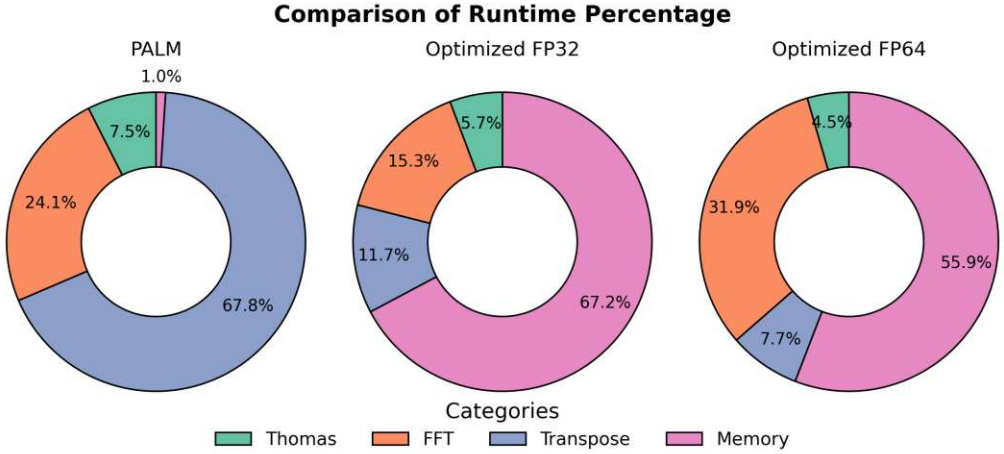


Figure 6.3: Visualisation showing the proportion of execution time for each of the four operation categories of the solver.

section, accounting for only 5.7% or 4.5%, is attributed to the Thomas algorithm.

Operation	speed up (×)		Relative speed up
	FP32	FP64	FP64/FP32
Transpose	546	324	1.7
Fast Fourier Transform	148	27	5.4
Thomas algorithm	115	57	2
Memory operations	1.7	0.67	2.1

Table 6.2: Speed up of the execution time for each of the four operation categories of the solver.

Table 6.2 shows the calculated speed up for each category. As could be expected from Figure 6.3, the biggest speed up was found for the transpositions, where a speed up of up to 546 times was measured. The biggest difference in speed up between single and double precision was in the FFT category, where the speed up was measured to be 148 times for single precision and 27 times for double precision. This was to be expected, given that the library used is optimised to work with floats. The GPU version of the Thomas algorithm shows a speed up of 115 times for single precision and 57 times for double precision. The smallest difference between the original and the GPU version was found in the memory operations category, with a speed up of 1.7 times for the single precision and a speed up of 0.67 for double precision. This demonstrates on the one hand that the the setup and deallocation of the six necessary temporary arrays that happen each time step in the initial implementation is slower than the copying process of one array to the GPU in single precision. On the other hand it shows that despite the need

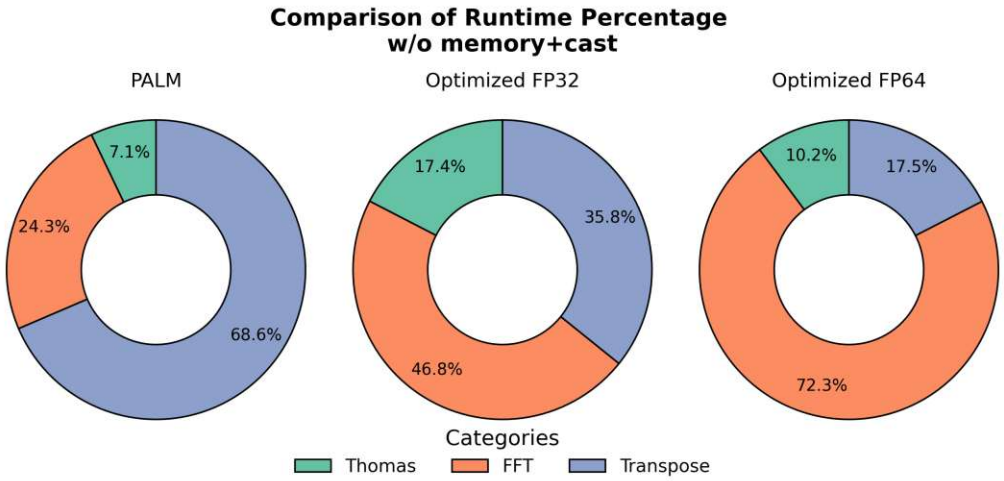


Figure 6.4: Visualisation of the proportion of execution time for the operation categories of the solver, without taking memory management into account.

for an additional casting step to calculate in single precision, the entire copying process is still faster than the double precision copying process.

Comparing the execution times of the solver ignoring the CPU/GPU interactions and precision changes can provide insight into the potential speed up between Fortran-based CPU and GPU execution. This assumes that the entire simulation would run on the GPU, with no memory copying necessary, and that there would be no limitations on GPU memory space. As memory was only a minor factor in the original PALM execution, Figure 6.4 shows that this chart stayed similar to the one in Figure 6.3. Excluding memory operations, 46.8% of the GPU execution time can now be attributed to the FFT category in the single precision case and 72.3% in the double precision case. Table 6.3 compares the speed up calculated using wall clock time with that calculated using only the time taken for mathematical operations. In the single precision case, the time taken for memory copying and casting between floats and doubles cannot be separated. Therefore, the shown speed up of 191.7 times compares the original double precision execution on a CPU with the single precision execution on a GPU. However, as the difference in performance between single and double precision on CPUs is much smaller than on GPUs, this can still provide valuable insight. As the ratio of FP32 to FP64 performance on CPUs is typically 2:1 [82, 83] compared to 64:1 on GPUs, the speed up compared to the original PALM execution using floating point can be calculated to be at least 95 times, if a perfect 2:1 ratio compared to the double precision time would occur in the original PALM execution. Ignoring memory operations, the speed up for the double precision execution would be 91.9 times.

Precision	Domain type	speed up (×)	speed up w/o cast+memcopy (×)
single	$N = 2^x$	65.2	190.1
double	$N = 2^x$	46.3	91.9

Table 6.3: Comparison of the wall clock time speed up to the theoretical speed up if memory transactions were excluded.

6.1.3 Evaluating the Global Effect

The effect of the time improvement of the pressure solver in context of the overall simulation is shown in Figure 6.5. To quantify the relative change of the execution time the x-axis shows the simulation time in percent of the original PALM execution. For this analysis the representative domain size of $512m \times 512m \times 128$m was chosen.

As described by Amdahl's law, the speedup is limited by the fraction of the code that can be parallelised, as can be seen in Equation 6.1. Here, s is the fraction of the code that is run in serial mode, and p is the fraction that is run in parallel on n processors [84].

$$Speedup = \frac{1}{s + \frac{p}{N}} \quad with \quad s + p = 1 \tag{6.1}$$

As stated in Section 6.4, in the original PALM implementation run in serial mode during each time step 15.1% of the runtime are spent on solving the pressure Poisson equation, this percentage could be reduced to 0.3% for FP32 and 0.4% for FP64. The simulation time was reduced by around 15%, meaning the single precision calculation with the optimised version took 85.2% of the original time, while the double precision version took 85.3%.

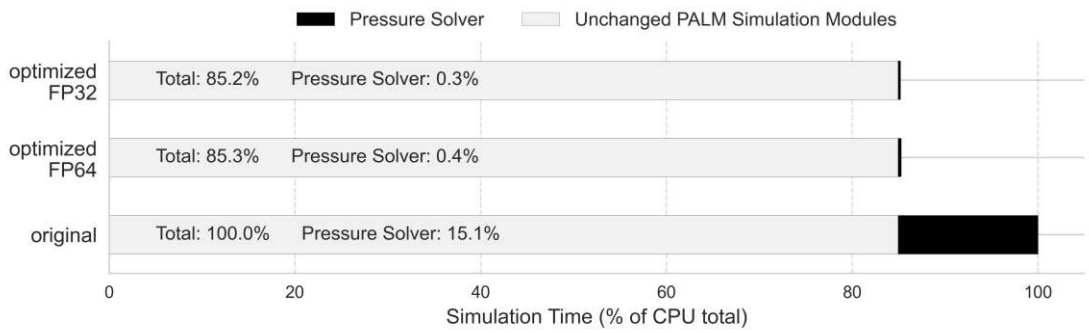


Figure 6.5: Figure showing the global effect in simulation time with the optimisations made.

6.2 Validity Evaluation

This section presents the results of the validity evaluation described in Subsection 4.3.2. First, the numerical accuracy of the implemented algorithm is presented, then the effect of the new implementation on the resulting velocities is shown.

6.2.1 Numerical Error of Pressure Solver

Figure 6.6 shows the different domain sizes on the x-axis. The same optimised domain sizes chosen for the performance evaluation were used here. The y-axis shows the relative L^2 norm error. The colour of the line corresponds to the data precision used during execution. The dashed line shows the corresponding machine precision for single and double precision.

It can be seen that the error depends on the precision used in the calculation. If the set-up of the solver is used from the original model the results of the double and single precision implementations are the same as the original implementation up to the corresponding machine precision. When the discrete Laplacian matrix is set up in double precision using CUDA, the double-precision pressure calculation shows a normalised L^2 error of 10^{-11}, while the single-precision calculation is minimally affected.

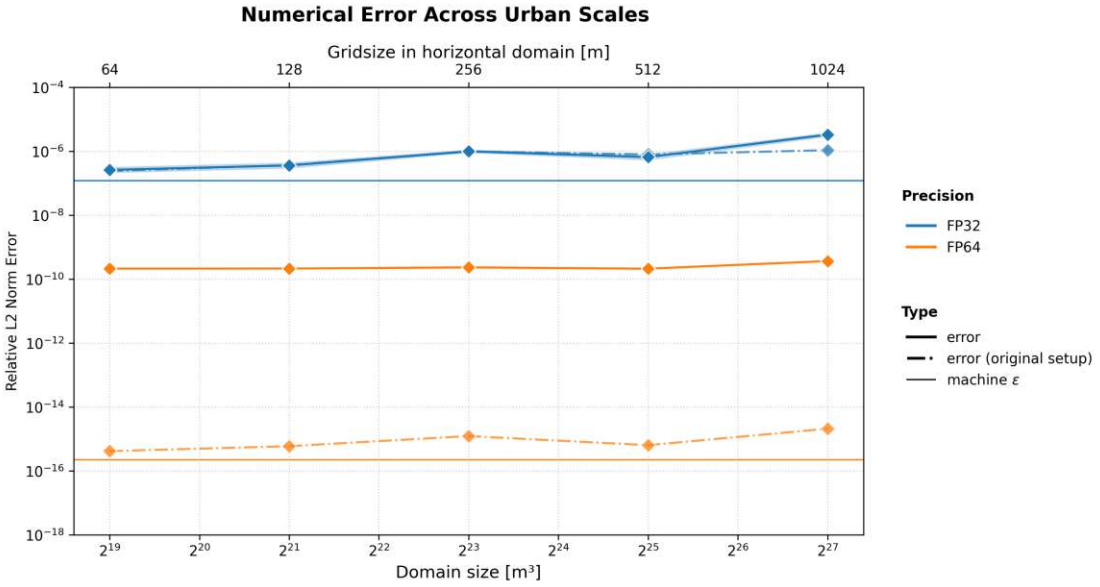


Figure 6.6: Numerical accuracy of the implemented Poisson solver, executed in single and double precision. The dashed lines correspond to the machine epsilon.

6.2.2 Impact on Wind Speed

To evaluate the immediate effect of the numerical error in the pressure calculation on the resulting wind speed, as described in Subsection 4.3.2 the relative L^2 norm error and the

maximum pointwise difference between two coordinate points were calculated between the wind speed output of a ten second simulation with the optimised pressure solver and with the original solver. The results are shown in Table 6.4. A dash indicates that no measurable difference was found. Overall, it can be concluded that keeping the optimised pressure calculation in double precision has no effect on the final result. However, if the pressure calculation is performed using the optimised single precision, the relative L^2 error is in the order of 10^{-8}, with a maximum difference in the last digit. The relative L^2 error and the pointwise difference appeared to increase in proportion to the domain size. However a different topography was used for each domain size which could have an influenced on the accuracy.

Domain Size		Relative L^2 Norm Error		Max. Difference	
Nx	N	FP32	FP64	FP32	FP64
64	2^{19}	3.02e-08	-	1.00e-06	-
128	2^{21}	8.85e-08	-	2.00e-06	-
256	2^{23}	8.33e-08	-	5.00e-06	-
512	2^{25}	8.67e-08	-	7.60e-05	-

Table 6.4: Comparison of the effect of error in the pressure calculation in the single precision output of the velocities $(\mathbf{u}, \mathbf{v}, \mathbf{w})$.

6.2.3 Error Propagation and Simulation Stability

To evaluate the effect of the numerical error resulting from the mixed-precision approach on the wind speed a ten hour simulation was evaluated. Since the CFL condition can reduce the step size when instabilities occur, the number of internal simulation steps provides an initial indication of the stability of the model. It was found that, in the ten-hour simulation, the number of internal time steps adjusted according to the CFL condition differed by less than 1% from one another. The exact numbers can be seen in Table 6.5.

	Internal Steps
PALM	45 760
PALM + optimised pressure solver (FP32)	45 672
PALM + optimised pressure solver (FP64)	45 345

Table 6.5: Resulting number of internal steps after a 10 hours simulation on a $128m \times 128m \times 128m$ domain.

As described in Subsection 4.3.2 three error measures were calculated for the wind speed and the individual wind speed components and can be seen in Table 6.6. For comparison,

the same measures from a validity study of PALM-4U wind speed at a location in Prague with a domain size of $1400m \times 1400m \times 256m$ and a resolution of $2m$ [78] can be seen next to it. After ten hours of simulation time (over 45 thousand steps), the optimised pressure solver produced wind speeds for which the mean squared difference between the baseline and the optimised versions are less than 5% of the product of their means. Furthermore, the wind speeds showed a near-zero bias as a result of the optimised pressure solver. The R coefficient also indicates a good pattern match between the model runs. The simulation was run using both the FORTRAN and GPU setups. However, the error does not depend on whether the setup was executed in the original code or on the GPU in double precision.

Metric	Wind speed		Wind speed components		Reference
	FP32	FP64	FP32	FP64	Value
NMSE	0.01	0.01 | 0.01	0.03	0.03 | 0.05	1.0
FB	-0.01	-0.02 | -0.02	-0.01	-0.02 | -0.01	0.5
R	0.93	0.94 | 0.95	0.98	0.98 | 0.98	0.5

Table 6.6: Statistical performance metrics comparing wind speeds after a 10-hour simulation of the original PALM model with those resulting from the optimised pressure solver: Normalized Mean Squared Error (NMSE), Fractional Bias (FB), Pearson Correlation Coefficient (R). The second value in the FP64 columns show the simulation results when using the FORTRAN initialised discrete Laplacian matrix. The reference value indicates the difference between PALM-4U and observations in a validation study [78].

6.3 Memory Profiling

This section presents the results of the memory analysis. As described in Subsection 4.3.3, to approximate the required memory, the number of arrays defined on the computational grid were counted, as these are responsible for the majority of the used memory. The requirements of the original double precision pressure solver are compared to those of the single- and double precision optimised solvers presented in this research. For GPU versions, the original CPU array copied to the GPU is counted as one array because it represents the same array in two memory locations: CPU and GPU memory. In a fully GPU-based implementation, only one of these memories would be required. The results are summarised in Table 6.7.

In the original solver, eleven arrays are necessary, more than half of which can be assigned to temporary arrays for transposition. To compute and save the coefficients for the Thomas algorithm, three arrays are necessary, making use of the fact that the upper and lower diagonal values are the same for every equation. An additional temporary array is allocated to run the Thomas algorithm. The final arrays required are the input and

output arrays. As described in Chapter 5, the number of temporary arrays required has been reduced in the new versions. In both optimised options, one real-valued and one complex temporary array are needed, and these are used throughout each method. As the generic temporary array can also be reused in the tridiagonal solver, the memory requirement for this section could also be reduced compared to that for the original solver. In the single precision version, a second input array is required on the CPU to convert the incoming doubles to floats, this array differs from the original input array and must be taken into account. With the described memory optimisations for cuFFT, the library only needs one array as a workspace.

Implementation Variant	Number of 3D Arrays	Numerical Precision	Memory per Cell [bytes]	Relative Memory Usage [%]
PALM Pressure Solver	11	double	80	100
Input	1			
Temporaries	6			
Thomas algorithm	4			
Optimized Pressure Solver	6	double	48	54
Input	1			
Temporary	2			
Thomas algorithm	2			
FFT	1			
Optimized Pressure Solver	7	single	28	31
Input	1 + 1			
Temporary	2			
Thomas algorithm	2			
FFT	1			

Table 6.7: Comparison of memory usage between the original PALM pressure solver and two optimised variants: The number of three-dimensional arrays used and the numerical precision are shown. From these, the memory required per computational cell and the memory usage relative to the baseline were calculated.

A comparative analysis of the relative memory usage of the optimised solver and the original double precision PALM pressure solver, shows that the memory of the adapted double precision solver requires 54% of memory. It was anticipated that executing the code in single precision would result in a memory reduction of at least 50%. However, in the single precision implementation presented here, it was demonstrated that the adapted solver requires only 31% of the original memory, taking the array necessary for casting into account.

As stated in the introduction, this analysis and adaptation of the pressure solver to the GPU was the first step in optimising the full PALM model for a single workspace CPU and GPU. Finally, the memory required by the pressure solver was compared to that needed by the entire simulation. All code files connected to the solver were checked, and the number of unique allocated arrays on the computational grid were counted. The found 448 arrays can possibly be reduced depending on the activated module and the selected implementation version of the specific sections of the code. It should also be noted that PALM allocates and deallocates arrays as needed, so it is not possible to determine which of these arrays need to be allocated at the same time. However, compared to the full model, the pressure solver requires a small amount of memory: all versions of the pressure solver require less than 2.5%. The optimised single precision version requires only 0.7%.

Implementation	Numerical Precision	Number of 3D Arrays	Relative Memory Usage [%]
Full PALM Model	FP64	448	100
PALM Pressure Solver	FP64	11	2.5
Optimized Pressure Solver	FP64	6	1.3
Optimized Pressure Solver	FP32	7	0.7

Table 6.8: Comparison of the memory usage of the different pressure solvers with that required by the full PALM model.

6.4 Structural Analysis of the Model

As described in Subsection 4.3.4 to answer the final research question the full model was analysed with the same parameters as the files connected to the pressure solver. Table 6.9 shows the top five files ranked with respect to the number of external modules loaded. The file *time_integration.f90* coordinates the calculation of the time steps and is clearly at the top of the list with 44 external modules loaded. All five of the listed files use the global arrays module. As described, particular focus is given to the 3D arrays module, as this signifies arrays defined on the computational grid that are updated by different routines during the simulation process. This means that the module depends on the current iteration of the arrays. However, since this analysis only counts the number of loaded modules, it is not clear to what extent these arrays and modules are actually used.

Table 6.10 shows the top five modules when sorted with regard to the total number of three- or four-dimensional arrays, separated into external and temporary arrays. Comparing Table 6.9 and Table 6.10 it is notable that files *prognostic_equations_mod.f90* and *radiation_model_mod.f90* are listed on both, indicating many dependences on external modules.

Filename	# Modules loaded	included 3D arrays
time_integration.f90	44	Yes
chemistry_model_mod.f90	37	Yes
prognostic_equations_mod.f90	29	Yes
ocean_mod.f90	24	Yes
radiation_model_mod.f90	24	Yes

Table 6.9: Top five files called during a time step, ranked by the number of modules loaded. The second column indicates whether the 3D array module was included among the loaded modules.

Filename	# external arrays	# temporary arrays
salsa_mod.f90	55	26
surface_mod.f90	62	0
radiation_model_mod.f90	44	6
bulk_cloud_model_mod.f90	33	0
prognostic_equations.f90	33	0

Table 6.10: Top five files called during a time step, ranked by the number of three or four dimensional external arrays used. The second column indicates the number of arrays allocated inside the file.

Table 6.11 shows five files that were found to have the highest number of MPI calls and barriers, indicating code that is optimised for a HPC system.

Filename	# MPI calls	# MPI barrier
radiation model mod .f90	129	5
lagrangian partical model .f90	97	16
bulk cloud model mod.f90	32	0
multi_agent_system_mod .f90	26	5
dynamics_mod .f90	22	0

Table 6.11: Top five files called during a time step, ranked by the number inter-node communications.

CHAPTER 7

Discussion

Building on the results presented in the previous chapter, this chapter discusses the main findings in the context of the research questions set at the beginning. The results are interpreted in the context of existing literature and their implications.

7.1 Performance Improvements

Transitioning the pressure solver from a HPC cluster-optimised implementation to a GPU-accelerated version on a single workstation makes it possible to adapt the code structure for the new target architecture, as running a HPC cluster-optimised code in serial mode is not efficient. Compared to HPC execution, where the heaviest routine of the FFT-based pressure solver is the computation of the Fast Fourier Transform itself [17], the highest percentage of computation time in serial execution mode was detected in functions associated with transposition, as can be seen in Figure 6.3. Therefore, in order to transition the codebase to the new target architecture optimally, it was necessary to remove obsolete functionality. This mainly affected the transposition section, as six function calls to rearrange the data before each transposition and two full transpositions could be omitted during the optimisation process. This meant that the transposition section was reduced to a much smaller part of the final execution. These changes including the addition of GPU specific code bring performance improvements. Performance as defined in Section 4.1 can be split up into three sections: *Speed up*, *Validity* and *Memory Efficiency*.

The runtime analysis, of Section 6.1, shows a speed up of up to 65 times. For comparison, Knoop et al. [17], accelerated the original model with OpenACC, achieved a maximum speed up of 1.5 times when running with a minimum of four CPU cores in a HPC cluster. The present study indicates that performance acceleration with a GPU appears to be more successful for single workstation executions. Optimisations especially for serial execution and the use of the low-level GPU programming language CUDA also contribute

to this. This outcome was anticipated to some extent, as it was concluded by Knoop et al. that the biggest performance bottleneck in their attempt was the MPI communication, which is a fundamental component of any HPC execution.

The speed up achieved by the presented optimisations depends on the size of the domain and the data type used. As expected, the single precision simulation achieved a higher speed up than the double precision simulation, even when the casting process was included in the timings. The effect of domain size on performance can be categorised in two ways: the number of cells and the edge length of the domain. Firstly, as an increasing number of cells need to be calculated, the GPU's parallel capabilities are utilised more and more. For this reason, the smallest domain size tested of $50m \times 50m \times 128m$ achieves the smallest speed up, while the largest domain size of $1024m \times 1024m \times 128m$ achieves the largest speed up, with the measured speed up ranging from 13.5 times to 65 times. The second factor is the edge length of the domain. It is preferable that the edge lengths of the domain are divisible by 32, as this reflects the fact that the GPU is optimised to work in batches of 32. This effect can be seen in the results: domains that followed this recommendation exhibited higher performance than those that did not. Depending on whether double or single precision was used, the change in domain size resulted in a speed up difference of approximately 20–30%.

The validity analysis shows that the optimised implementation presented for the pressure solver are numerically accurate. This means that the optimised GPU approach can be integrated into the existing PALM model in both double and single precision as the mixed-precision approach did not affect the stability of the system. The error induced by the discrete Laplacian matrix is likely caused by floating-point rounding effects that are amplified by local operations, however the accuracy of the discrete Laplacian matrix does not impact the simulation output. The difference between the three PALM simulation versions is negligible after a ten hour simulation. Furthermore, it was found that the magnitude of the error was the same, regardless of whether the pressure solver was calculated in single- or double precision. Even though only the error after a ten hour simulation was analysed explicitly here, many studies have shown that single precision is suitable for short-term simulations and long-term climate modelling [85, 86]. Currently, the PALM model core can be run in single precision as a test feature [87], however this was not tested during this research as it would have required substantial changes throughout the model system.

From a working memory perspective, Table 6.7 indicates that a successful adaptation to the new architecture was achieved, as the number of temporary arrays required for transposition could be reduced. This drastically reduced the method's required workspace, as the temporary arrays contributed the most to the memory footprint, as presented in Section 6.3. For double precision execution, the memory required is only 54% of the original, and if the calculation were single precision, this could be further reduced to 31% of the original. The biggest impact on memory efficiency was reducing temporary arrays, which was possible due to the change in target architecture.

7.2 Runtime Impact on Full Simulation

To analyse the possibility if the optimisation of one module, here the Poisson pressure solver, can have an substantial effect on the overall runtime, the runtime of a simulation step with the original pressure solver was compared to a simulation step with the optimised version. The results can be seen in Figure 6.5.

Of a full simulation step approximately 15% can be assigned to the pressure solver, with the optimised version of this solver this could be reduced to $0.3 - 0.4\%$. This leads to runtime reduction of approximately 15% and a overall runtime speed up of 1.17 times. This tells us that the parallelisation and optimisation of a single module can have an substantial impact on the runtime of an entire model, however the impact is bounded by the percentage of the runtime of the original module following Amdahl's law.

7.3 Impact of Memory Management

The effect of the optimisations on the different operation categories was analysed to determine the extent to which memory transactions affect overall performance. Subsection 6.1.2 showed that more than half of the GPU execution time is spent on memory interactions between the GPU and CPU, even though only one array needs to be exchanged between the two. Therefore it is important to find the best intersection point in hybrid CPU and GPU methods to limit the necessary memory transfers. One could also try to hide the copying cost by overlapping the copying operation with computation on a part of the array [31]. However in the analysed pressure solver this was not applied as a full overlap was not possible, due to the different array permutations needed during execution. The FFTs need a full row to execute, where for the Thomas algorithm a full column must be accessible at a time. An option would be to transpose the matrix while putting it into pinned memory to overlap the asynchronous memory copy with the batched FFT executions [64], but it would be necessary to synchronise the application before the start of the matrix solver and only start overlapping again after this is finished, consuming the time initially saved. In addition the chosen FFT library (cuFFT) is optimised to run on big batched FFTs and the implemented transposition operation is also optimised for the GPU. Due to these reasons concurrent copy and GPU computation was not implemented for this method.

If in the final implementation the full PALM model would run in C++ and not Fortran, pinned memory could be used for all arrays that would need to be exchanged between CPU and GPU to accelerate the copying process. In the case of mixed-precision approach that was tested to leverage the GPUs optimised performance in single precision this was already implemented. The double precision array elements were cast onto a single precision page-locked pinned memory location, to accelerate the memory transfer. However with double precision in the current version this would not be applicable as the input array for the GPU is already allocated in the Fortran code as pageable. This made it possible to accelerate memory transfer, resulting in single precision casting with pinned memory

copying being faster than double precision pageable copying.

7.4 Identified Bottlenecks

The main bottleneck in this implementation is regrading memory, on the one hand the performance loss due to the copying to the GPU and the amount of memory needed for the computation and the restrictions that this imposes on the possible domain sizes. In contrast to the implementation by Knoop et al. [17], which used a distributed memory system to manage massive domain sizes, this implementation is localised to one CPU and GPU, and is therefore designed for domain sizes up to 1 km².

The current implementation assumes that the full array fits on the GPU. If that is not possible this method would not bring the performance speed up as was achieved here. It may be possible to avoid having the entire array on the GPU at once, instead copying sections of the array back and forth between the CPU and GPU, overlapping computation and communication. However, as described above, the FFT-based pressure solver requires the array in different permutations throughout the process. If the array could not be placed on the GPU in its entirety, additional memory transactions would be necessary before and after the tridiagonal solver to ensure that the required slice of the array was present on the GPU. This would slow down the optimised routines and result in a smaller speed up.

However, Table 6.8 shows that, although the memory footprint could be reduced to about 30% of the original pressure solver, the pressure solver is not a memory-heavy routine when compared to the full model. Therefore, if the full model should be optimised for the new target architecture and GPU the high memory requirements could turn out to be a bottleneck, ultimately limiting the maximum extent of the simulation domain.

7.5 Feasibility of Full Model Optimization

The aim of this research was to explore the acceleration potential of the pressure solver as part of the PALM-4U model system. The pressure solver in the model core was ideally suited to GPU optimisation, while the rest of the simulation ran on the CPU. As the intersection point was chosen carefully, only one array needed to be copied from the GPU at each time step. This is important, as the runtime benchmarking showed that this memory movement was a limiting factor in the speed up of the method.

The analysis showed that, compared to the pressure solver, the other modules are much more closely coupled with other parts of the model. For example, although the *radiation module* appears to be highly optimised for HPC due to its large number of MPI calls, it is also highly dependent on other modules and variables. The same is true of the *time integration* and *prognostic equation* modules, which are a central part of the model core and require high connectivity. However, as file lengths and the number of modules per file vary, this analysis can only provide a high-level overview.

Despite this, the PALM-4U model's modular structure makes it highly suitable for a module-by-module optimisation process for a single workstation PC with a GPU. If any other section is chosen for optimisation in future research, it must be examined in more detail to identify potential issues and determine the optimal intersection point. After all, even though the entry pressure solver module is highly dependent on other arrays, this did not limit the final optimisation thanks to the clever choice of intersection point. In addition, efficient lossless compression of floating-point data could be implemented to reduce the data transfer bottleneck.

7.6 Limitations of the Study

While this study demonstrates the potential for accelerating a part of the model core of the microclimate PALM-4U, the results cannot be generalised to the entire model, especially as different parts of the model have different computational requirements and would benefit differently form a GPU architecture.

Furthermore, the reported speed up should not be interpreted as an improvement over the original PALM model running on a high-performance computing architecture for which it was designed and optimised. Benchmarking was performed against the models serial execution on a standard desktop workstation.

In addition, a detailed investigation into the underlying cause of the numerical inaccuracy in the discrete Laplacian matrix is not currently prioritized, as it does not affect the overall simulation results and because the machine-precision accurate single-precision calculation is the primary focus of this study due to its performance advantages. However, if needed, this issue will be examined more closely in future studies.

Finally, this work did not explore the long-term effects of mixed-precision simulations on the PALM climate model. This research involved performing short-term (seconds) and medium-term (ten hours) simulations. The simulation time was limited by the serialised runtime of the full model, on a domain of $128m \times 128m \times 128m$, the full runtime with the optimised solver is three times the simulation time. For this reason, the long-term influence on accuracy and stability would need to be explored in future studies. Nevertheless, the ten hour simulation with over 45,000 internal steps shows that the mixed-precision approach is promising.

7.7 Implications

This study suggests that it is possible to successfully accelerate a microclimate model such as PALM-4U using GPUs. However, as the comparison with the study by Knoop et al. [17] shows, this acceleration has more potential if it is accompanied by a change in computing architecture from an HPC cluster to a single workstation. As with a shift away from high-performance computing clusters, the main bottleneck identified by Knoop et al. can be avoided. Furthermore, the findings indicate that changing to single precision for computations on the GPU is promising. These findings open up new ways to make urban microclimate modelling more accessible and efficient.

CHAPTER 8

Conclusion

This research investigated the potential for accelerating the PALM urban microclimate model, focussing on a key component of the core of the model. Restructuring the method to optimise for serial execution and utilising the GPU's parallel capabilities achieved a speed up of up to 65 times for the pressure solver. Although only part of the model core was optimised, this resulted in a 15% reduction in the runtime of the full model. These findings demonstrate the value of GPU acceleration and optimisation for specific target architectures in climate modelling.

Although the scope of this work was limited to the Poisson FFT pressure solver, the results suggest that applying the approach to other sections, or even the full original model codebase, could further enhance performance. However, memory constraints could affect the scalability and effectiveness of optimising the full model. Therefore, careful memory management must be a key consideration in any future study.

Ultimately, this work highlights the importance of modern hardware architecture in enabling complex climate models to be transferred from high-performance computing clusters to standard desktop computers. This could make the models available to people outside the scientific community as decision-support tools for climate-resilient planning.

Overview of Generative AI Tools Used

This thesis used the DeepL tool (www.deepl.com) to refine the grammar and fluency of the text without altering its meaning.

List of Figures

2.1 Overview of numerical modelling approaches of turbulent flow [22]. 6
2.2 Comparison of large-scale resolved eddies (left) and parametrised sub-grid scale eddies (right). 6
2.3 PALM-4U simulation results of the Maria-Theresien-Platz in Vienna. . . . 8
2.4 Visualisation of the computational grid. 9
2.5 Visualisation of the time loop of the PALM model core [17]. 9
2.6 Visualisation of the file dependencies involved in the pressure solver. Arrows indicate the direction of function calls: a file at the tail of an arrow calls a file at the head of the arrow. 13
2.7 Breakdown of the FFT based pressure solver into 19 steps. 13
2.8 Visualisation of the CUDA Programming Model [31]. 15
2.9 Visualisation of the CUDA Memory Model [31]. 15
2.10 This figure shows an example of a scenarify simulation in the application [32]. 17

5.1 Visualisation of the Fortran and C++ Integration process. 36
5.2 The timeline shows the structure of the Poisson FFT solver. The greyed-out blocks indicate the dropped function calls used to rearrange the data, as opposed to the structure shown in Figure 2.7. 37
5.3 A comparison of synchronous and asynchronous execution. In the synchronous case (top), the light grey block represents the casting operation on the CPU, followed by a memory transfer to a page-locked (pinned) memory buffer (black) and then to the device (pink). In the asynchronous case (bottom), the casting operation occurs directly in the pinned memory. This allows the casting and memory transfer to overlap, eliminating the intermediate copy step. . . . 39
5.4 Visualisation of the variable definitions used in 5. The figure on the left shows the complex number resulting from a forward fast Fourier transform (FFT), or the pre-processed input for an inverse FFT. The figure on the right shows the real number after post-processing, or before pre-processing. 41
5.5 Figure of the two needed involution transpositions of the three-dimensional array. 43
5.6 Figure visualising the three-dimensional transposition algorithm, using a shared memory tile with padding. 44
5.7 The solver structure of the GPU and standard workstation implementation after the changes for the new architecture. 46

6.1 Two exemplary simulation domains are shown, where colours indicate wind velocity ranging from nearly zero, shown in green, over yellow and orange to blue, which indicates 50 per cent of the maximum simulated velocity. Velocities that exceed this threshold are not displayed. 50
6.2 This Figure shows the results of the scalability analysis. Colours indicate the used date type and line style the domain type. 51
6.3 Visualisation showing the proportion of execution time for each of the four operation categories of the solver. 53
6.4 Visualisation of the proportion of execution time for the operation categories of the solver, without taking memory management into account. 54
6.5 Figure showing the global effect in simulation time with the optimisations made. 55
6.6 Numerical accuracy of the implemented Poisson solver, executed in single and double precision. The dashed lines correspond to the machine epsilon. . . 56

List of Tables

2.1 A summary of the description of each file. 14

4.1 Table of the selected domain sizes for the speed up performance analysis. 28
4.2 Table of the used machine epsilon for comparison [77]. 29

5.1 Files connected to the FFT pressure solver, showing for each: the number of loaded modules, whether the 3D Arrays module is included, the number of external three or four dimensional arrays, the number of temporary arrays allocated and the number of inter-node communication. 34
5.2 Comparison of performance metrics between old and new implementations for FP32 and FP64 precision. 42
5.3 Comparison of performance between old and new implementations of the transposition kernels for FP32 and FP64 precision. 45
5.4 This Table summarises the extent to which the files required for the FFT-based pressure calculation during time stepping were altered during optimisation. 46

6.1 The maximum achieved speed up compared to the original PALM implementation is shown for all combinations of precision and domain type. 52
6.2 Speed up of the execution time for each of the four operation categories of the solver. 53
6.3 Comparison of the wall clock time speed up to the theoretical speed up if memory transactions were excluded. 55
6.4 Comparison of the effect of error in the pressure calculation in the single precision output of the velocities $(\mathbf{u}, \mathbf{v}, \mathbf{w})$. 57
6.5 Resulting number of internal steps after a 10 hours simulation on a $128m \times 128m \times 128m$ domain. 57
6.6 Statistical performance metrics comparing wind speeds after a 10-hour simulation of the original PALM model with those resulting from the optimised pressure solver: Normalized Mean Squared Error (NMSE), Fractional Bias (FB), Pearson Correlation Coefficient (R). The second value in the FP64 columns show the simulation results when using the FORTRAN initialised discrete Laplacian matrix. The reference value indicates the difference between PALM-4U and observations in a validation study [78]. 58

6.7 Comparison of memory usage between the original PALM pressure solver and two optimised variants: The number of three-dimensional arrays used and the numerical precision are shown. From these, the memory required per computational cell and the memory usage relative to the baseline were calculated. 59
6.8 Comparison of the memory usage of the different pressure solvers with that required by the full PALM model. 60
6.9 Top five files called during a time step, ranked by the number of modules loaded. The second column indicates whether the 3D array module was included among the loaded modules. 61
6.10 Top five files called during a time step, ranked by the number of three or four dimensional external arrays used. The second column indicates the number of arrays allocated inside the file. 61
6.11 Top five files called during a time step, ranked by the number inter-node communications. 61

List of Algorithms

1 C++ Function Pointer Setup and Usage 35
2 Fortran Procedure Setup and Invocation 36
3 Cast and Copy to GPU 39
4 Optimized cuFFT Plan Initialization 40
5 createComplexData 42
6 Transpose_xyz2yxz 44

Bibliography

[1] Intergovernmental Panel On Climate Change (Ipcc). *Climate Change 2022 – Impacts, Adaptation and Vulnerability: Working Group II Contribution to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change*. 1st ed. Cambridge University Press, June 2023. ISBN: 978-1-009-32584-4. DOI: 10.1017/9781009325844.

[2] T. R. Oke. "The energetic basis of the urban heat island". en. In: *Quarterly Journal of the Royal Meteorological Society* 108.455 (Jan. 1982), pp. 1–24. ISSN: 0035-9009, 1477-870X. DOI: 10.1002/qj.49710845502.

[3] A. Haines et al. "Climate change and human health: Impacts, vulnerability and public health". en. In: *Public Health* 120.7 (July 2006), pp. 585–596. ISSN: 00333506. DOI: 10.1016/j.puhe.2006.01.002.

[4] Jonathan A. Patz et al. "Impact of regional climate change on human health". en. In: *Nature* 438.7066 (Nov. 2005), pp. 310–317. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature04188.

[5] Ashish Sharma et al. "The Need for Urban-Resolving Climate Modeling Across Scales". en. In: *AGU Advances* 2.1 (2021), e2020AV000271. ISSN: 2576-604X. DOI: 10.1029/2020AV000271.

[6] S.E Gill et al. "Adapting Cities for Climate Change: The Role of the Green Infrastructure". en. In: *Built Environment* 33.1 (Mar. 2007), pp. 115–133. ISSN: 02637960. DOI: 10.2148/benv.33.1.115.

[7] Alistair Hunt and Paul Watkiss. "Climate change impacts and adaptation in cities: a review of the literature". en. In: *Climatic Change* 104.1 (Jan. 2011), pp. 13–49. ISSN: 0165-0009, 1573-1480. DOI: 10.1007/s10584-010-9975-6.

[8] Elisa Palazzo and Wan Nurul Mardiah Wan Mohd Rani. "Regenerating Urban Areas Through Climate Sensitive Urban Design". In: *Advanced Science Letters* 23.7 (July 2017), pp. 6394–6398. DOI: 10.1166/asl.2017.9277.

[9] Michal Belda et al. *Sensitivity analysis of the PALM model system 6.0 in the urban environment*. Aug. 2020. DOI: 10.5194/gmd-2020-126.

[10] E Scott Krayenhoff et al. “Cooling hot cities: a systematic and critical review of the numerical modelling literature”. en. In: *Environmental Research Letters* 16.5 (May 2021). Publisher: IOP Publishing, p. 053007. ISSN: 1748-9326. DOI: 10.1088/1748-9326/abdcf1.

[11] Björn Maronga et al. “Overview of the PALM model system 6.0”. en. In: *Geoscientific Model Development* 13.3 (Mar. 2020), pp. 1335–1372. ISSN: 1991-9603. DOI: 10.5194/gmd-13-1335-2020.

[12] Mohamed Hefny Salim et al. “Introducing the Urban Climate Model PALM System 6.0”. en. In: *International Journal of Applied Energy Systems* 2.1 (Jan. 2020), pp. 15–18. ISSN: 2636-3720. DOI: 10.21608/ijaes.2020.169937.

[13] Björn Maronga et al. “Development of a new urban climate model based on the model PALM – Project overview, planned work, and first achievements”. en. In: *Meteorologische Zeitschrift* (June 2019). Publisher: Schweizerbart’sche Verlagsbuchhandlung, pp. 105–119. ISSN: , DOI: 10.1127/metz/2019/0909.

[14] Maja Žuvela-Aloise et al. “Evaluation of city-scale PALM model simulations and intra-urban thermal variability in Vienna, Austria using operational and crowdsourced data”. In: *Urban Climate* 59 (Feb. 2025), p. 102245. ISSN: 2212-0955. DOI: 10.1016/j.uclim.2024.102245.

[15] Antonina Kriuger et al. *Innovative urban climate model PALM-4U as a support tool for municipal climate adaptation strategies.* en. Tech. rep. EGU21-12563. Conference Name: EGU21. Copernicus Meetings, Mar. 2021. DOI: 10.5194/egusphere-egu21-12563.

[16] VRVis GmbH. *ClimaSens. Climate-sensitive Adaptive Planning for Shaping Resilient Cities.* 2024. URL: https://projekte.ffg.at/projekt/4856674 (visited on 03/12/2025).

[17] Helge Knoop et al. “Porting the MPI Parallelized LES Model PALM to Multi-GPU Systems - An Experience Report”. en. In: Jan. 2016. URL: https://openreview.net/forum?id=CokxUKSqwU (visited on 03/10/2025).

[18] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* en. Google-Books-ID: KUxsAQAAQBAJ. Addison-Wesley, 2013. ISBN: 978-0-321-80946-9.

[19] NVIDIA. *Nsight Systems.* en. 2025. URL: https://developer.nvidia.com/nsight-systems (visited on 05/18/2025).

[20] NVIDIA. *Nsight Compute.* en. 2025. URL: https://developer.nvidia.com/nsight-compute (visited on 05/18/2025).

[21] Yang Zhiyin. “Large-eddy simulation: Past, present and the future”. In: *Chinese Journal of Aeronautics* 28.1 (Feb. 2015), pp. 11–24. ISSN: 1000-9361. DOI: 10.1016/j.cja.2014.12.007.

[22] Jurij Sodja. “Turbulence models in CFD”. In: *University of Ljubljana* (Jan. 2007), pp. 1–18.

[23] *palm4u – PALM*. URL: https://palm.muk.uni-hannover.de/trac/wiki/palm4u (visited on 07/23/2025).

[24] Peter Bröde et al. "Deriving the operational procedure for the Universal Thermal Climate Index (UTCI)". en. In: *International Journal of Biometeorology* 56.3 (May 2012), pp. 481–494. ISSN: 0020-7128, 1432-1254. DOI: 10.1007/s00484-011-0454-1.

[25] Aristides A. N. Patrinos and Alan L. Kistler. "A numerical study of the Chicago lake breeze". In: *Boundary-Layer Meteorology* 12.1 (Aug. 1977), pp. 93–123. ISSN: 1573-1472. DOI: 10.1007/BF00116400.

[26] B. Maronga et al. "The Parallelized Large-Eddy Simulation Model (PALM) version 4.0 for atmospheric and oceanic flows: model formulation, recent developments, and future perspectives". English. In: *Geoscientific Model Development* 8.8 (Aug. 2015). Publisher: Copernicus GmbH, pp. 2515–2551. ISSN: 1991-959X. DOI: 10.5194/gmd-8-2515-2015.

[27] Radyadour Kh. Zeytounian. "Joseph Boussinesq and his approximation: a contemporary view". en. In: *Comptes Rendus. Mécanique* 331.8 (July 2003), pp. 575–586. ISSN: 1873-7234. DOI: 10.1016/S1631-0721(03)00120-7.

[28] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Minnesota: Univeristy of Minnesota - Society for Industrial and Applied Mathematics, 2003. URL: https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[29] Ulrich Schumann and Roland Sweet. "Fast Fourier Transforms for Direct Solution of Poisson's Equation with Staggered Boundary Conditions". In: *Journal of Computational Physics* 75 (Mar. 1988), pp. 123–137. DOI: 10.1016/0021-9991(88)90102-7.

[30] Helmut Schmidt. *Three-dimensional, direct and vectorized elliptic solvers for various boundary conditions*. ger. Als Ms. gedr.. Mitteilung / Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt. Köln: Wiss. Berichtswesen der DFVLR, 1984.

[31] NVIDIA. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 05/18/2025).

[32] *scenarify ist eine Visualisierungs- und Simulationssoftware für Hochwasser und Starkregen*. de. URL: https://www.vrvis.at/produkte-loesungen/produkte-lizenzen/scenarify (visited on 07/23/2025).

[33] Sebastian Grottel et al. "Real-time visualization of urban flood simulation data for non-professionals". In: *Workshop on Visualisation in Environmental Sciences (EnvirVis)* Eurographics Association (2015), pp. 37–41.

[34] Fredrik Lindberg et al. "SOLWEIG 1.0 – Modelling spatial variations of 3D radiant fluxes and mean radiant temperature in complex urban settings". en. In: *International Journal of Biometeorology* 52.7 (Sept. 2008), pp. 697–713. ISSN: 0020-7128, 1432-1254. DOI: 10.1007/s00484-008-0162-7.

[35] Francis Miguet. “A FURTHER STEP IN ENVIRONMENT AND BIOCLIMATIC ANALYSIS: THE SOFTWARE TOOL SOLENE”. en. In: *Building Simulation* (2007).

[36] Mostapha Sadeghipour Roudsari et al. “Ladybug: A Parametric Environmental Plugin For Grasshopper To Help Designers Create An Environmentally-conscious Design”. en. In: *Building Simulation Conference Proceedings*. ISSN: 2522-2708. IBPSA, Aug. 2013. DOI: 10.26868/25222708.2013.2499.

[37] Kerry A. Nice et al. “Development of the VTUF-3D v1.0 urban micro-climate model to support assessment of urban vegetation influences on human thermal comfort”. In: *Urban Climate* 24 (June 2018), pp. 1052–1076. ISSN: 2212-0955. DOI: 10.1016/j.uclim.2017.12.008.

[38] Cho Kwong Charlie Lam et al. “A review on the significance and perspective of the numerical simulations of outdoor thermal environment”. In: *Sustainable Cities and Society* 71 (Aug. 2021), p. 102971. ISSN: 2210-6707. DOI: 10.1016/j.scs.2021.102971.

[39] Sebastian Huttner and Michael Bruse. “Numerical modeling of the urban climate – a preview on ENVI-met 4.0”. en. In: *The seventh International Conference on Urban Climate* (2009).

[40] S. Tsoka et al. “Analyzing the ENVI-met microclimate model’s performance and assessing cool materials and urban vegetation applications–A review”. en. In: *Sustainable Cities and Society* 43 (Nov. 2018), pp. 55–76. ISSN: 22106707. DOI: 10.1016/j.scs.2018.08.009.

[41] Zhixin Liu et al. “Modeling microclimatic effects of trees and green roofs/façades in ENVI-met: Sensitivity tests and proposed model library”. In: *Building and Environment* 244 (Oct. 2023), p. 110759. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2023.110759.

[42] Kristian Fabbri et al. “Effect of facade reflectance on outdoor microclimate: An Italian case study”. In: *Sustainable Cities and Society* 54 (Mar. 2020), p. 101984. ISSN: 2210-6707. DOI: 10.1016/j.scs.2019.101984.

[43] Nils Eingrüber et al. “Investigation of the ENVI-met model sensitivity to different wind direction forcing data in a heterogeneous urban environment”. English. In: *Advances in Science and Research*. Vol. 20. ISSN: 1992-0628. Copernicus GmbH, July 2023, pp. 65–71. DOI: 10.5194/asr-20-65-2023.

[44] Simon Helge. “Modeling urban microclimate : development, implementation and evaluation of new and improved calculation methods for the urban microclimate model ENVI-met”. eng. Dissertation. Johannes Gutenberg-Universit at Mainz, 2016. URL: https://openscience.ub.uni-mainz.de/handle/20.500.12030/4044 (visited on 05/23/2025).

[45] Dieter Scherer et al. “Urban Climate Under Change [UC]2 – A National Research Programme for Developing a Building-Resolving Atmospheric Model for Entire City Regions”. en. In: *Meteorologische Zeitschrift* (June 2019). Publisher: Schweizerbart’sche Verlagsbuchhandlung, pp. 95–104. ISSN: , DOI: 10.1127/metz/2019/0913.

[46] Antonina Krueger. “PALM-4U Handbuch für die Praxis”. de. In: (2023). URL: https://www.uc2-propolis.de/imperia/md/assets/propolis/images/propolis_handbuch_fuer_die_praxis_final_31-08-2023.pdf.

[47] Jinrong Jiang et al. “Porting LASG/ IAP Climate System Ocean Model to Gpus Using OpenAcc”. In: *IEEE Access* 7 (2019), pp. 154490–154501. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2932443.

[48] Michail Alvanos and Theodoros Christoudias. “GPU-accelerated atmospheric chemical kinetics in the ECHAM/MESSy (EMAC) Earth system model (version 2.52)”. en. In: *Geoscientific Model Development* 10.10 (Oct. 2017), pp. 3679–3693. ISSN: 1991-9603. DOI: 10.5194/gmd-10-3679-2017.

[49] Mark Govett et al. “Parallelization and Performance of the NIM Weather Model on CPU, GPU, and MIC Processors”. In: *Bulletin of the American Meteorological Society* 98.10 (Oct. 2017), pp. 2201–2213. ISSN: 0003-0007, 1520-0477. DOI: 10.1175/BAMS-D-15-00278.1.

[50] Yuzhu Wang et al. “Using a GPU to Accelerate a Longwave Radiative Transfer Model with Efficient CUDA-Based Methods”. en. In: *Applied Sciences* 9.19 (Sept. 2019), p. 4039. ISSN: 2076-3417. DOI: 10.3390/app9194039.

[51] Jarno Mielikainen et al. “GPU Compute Unified Device Architecture (CUDA)-based Parallelization of the RRTMG Shortwave Rapid Radiative Transfer Model”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 9.2 (Feb. 2016). Conference Name: IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, pp. 921–931. ISSN: 2151-1535. DOI: 10.1109/JSTARS.2015.2427652.

[52] Matthew Norman et al. “A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel”. In: *Journal of Computational Science*. Computational Science at the Gates of Nature 9 (July 2015), pp. 1–6. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2015.04.022.

[53] Gunilla Sköllermo. “A Fourier method for the numerical solution of Poisson’s equation”. en. In: *Mathematics of Computation* 29.131 (1975), pp. 697–711. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1975-0371096-4.

[54] James W. Cooley and John W. Tukey. “An algorithm for the machine calculation of complex Fourier series”. en. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1965-0178586-1.

[55] Clive Temperton. “Implementation of a self-sorting in-place prime factor FFT algorithm”. en. In: *Journal of Computational Physics* 58.3 (May 1985), pp. 283–299. ISSN: 00219991. DOI: 10.1016/0021-9991(85)90164-0.

[56] D. Harris et al. “Vector radix fast Fourier transform”. In: *ICASSP '77. IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 2. May 1977, pp. 548–551. DOI: 10.1109/ICASSP.1977.1170349.

[57] David Střelák and Jiří Filipovič. “Performance analysis and autotuning setup of the cuFFT library”. en. In: *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. Limassol Cyprus: ACM, Nov. 2018, pp. 1–6. ISBN: 978-1-4503-6591-8. DOI: 10.1145/3295816.3295817.

[58] Fan Zhang et al. *A GPU Based Memory Optimized Parallel Method For FFT Implementation*. en. arXiv:1707.07263 [cs]. July 2017. DOI: 10.48550/arXiv.1707.07263.

[59] Akira Nukada and Satoshi Matsuoka. “Auto-tuning 3-D FFT library for CUDA GPUs”. en. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Portland Oregon: ACM, Nov. 2009, pp. 1–10. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654090.

[60] Louis Pisha and Łukasz Ligowski. “Accelerating non-power-of-2 size Fourier transforms with GPU Tensor Cores”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. May 2021, pp. 507–516. DOI: 10.1109/IPDPS49936.2021.00059.

[61] Binrui Li et al. “tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. ISSN: 2168-9253. Sept. 2021, pp. 1–11. DOI: 10.1109/Cluster48925.2021.00035.

[62] Sultan Durrani et al. “Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles”. en. In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Atlanta, GA, USA: IEEE, Sept. 2021, pp. 345–355. ISBN: 978-1-6654-4278-7. DOI: 10.1109/PACT52795.2021.00032.

[63] Shuo Chen and Xiaoming Li. “A hybrid GPU/CPU FFT library for large FFT problems”. In: *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*. ISSN: 2374-9628. Dec. 2013, pp. 1–10. DOI: 10.1109/PCCC.2013.6742796.

[64] Jaehong Lee and Duksu Kim. “Large-scale 3D fast Fourier transform computation on a GPU”. In: *ETRI Journal* 45.6 (Dec. 2023). Publisher: John Wiley & Sons, Ltd, pp. 1035–1045. ISSN: 1225-6463. DOI: 10.4218/etrij.2022-0297.

[65] Liang Gu et al. “An empirically tuned 2D and 3D FFT library on CUDA GPU”. In: June 2010, pp. 305–314. DOI: 10.1145/1810085.1810127.

[66] Jose L. Jodra et al. “Efficient 3D Transpositions in Graphics Processing Units”. en. In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 876–891. ISSN: 1573-7640. DOI: 10.1007/s10766-015-0366-5.

[67] Antti-Pekka Hynninen and Dmitry I. Lyakh. *cuTT: A High-Performance Tensor Transpose Library for CUDA Compatible GPUs*. arXiv:1705.01598 [cs]. May 2017. DOI: 10.48550/arXiv.1705.01598.

[68] Dmitry I. Lyakh. “An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU”. In: *Computer Physics Communications* 189 (Apr. 2015), pp. 84–91. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2014.12.013.

[69] Llewellyn Hilleth Thomas. “Elliptic problems in linear difference equations over a network”. In: *Watson Sci. Comput. Lab. Rept., Columbia University, New York* 1 (1949), p. 71.

[70] Yao Zhang et al. “Fast tridiagonal solvers on the GPU”. en. In: ().

[71] Hee-Seok Kim et al. “A Scalable Tridiagonal Solver for GPUs”. en. In: *2011 International Conference on Parallel Processing*. Taipei, Taiwan: IEEE, Sept. 2011, pp. 444–453. ISBN: 978-1-4577-1336-1. DOI: 10.1109/ICPP.2011.41.

[72] NVIDIA. *cuSPARSE - CUDA Toolkit Documentation*. 2025. URL: https://docs.nvidia.com/cuda/cusparse/ (visited on 04/08/2025).

[73] Pedro Valero-Lara et al. “cuThomasBatch & cuThomasVBatch, CUDA Routines to Compute Batch of Tridiagonal Systems on NVIDIA GPUs”. en. In: (2018).

[74] NVIDIA. *cuFFT - CUDA Toolkit Documentation*. 2025. URL: https://docs.nvidia.com/cuda/cufft/ (visited on 03/10/2025).

[75] Denise Hertwig et al. “Urban signals in high-resolution weather and climate simulations: role of urban land-surface characterisation”. In: *Theoretical and Applied Climatology* 142 (Oct. 2020). DOI: 10.1007/s00704-020-03294-1.

[76] Jeffrey Raven et al. “Urban Planning and Urban Design”. In: *Climate Change and Cities: Second Assessment Report of the Urban Climate Change Research Network*. Ed. by Cynthia Rosenzweig et al. Cambridge: Cambridge University Press, 2018, pp. 139–172. ISBN: 978-1-316-60333-8. DOI: 10.1017/9781316563878.012.

[77] De La Fraga and Luis Gerardo. “Differential Evolution under Fixed Point Arithmetic and FP16 Numbers”. en. In: *Mathematical and Computational Applications* 26.1 (Mar. 2021). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 13. ISSN: 2297-8747. DOI: 10.3390/mca26010013.

[78] Jaroslav Resler et al. “Validation of the PALM model system 6.0 in a real urban environment: a case study in Dejvice, Prague, the Czech Republic”. English. In: *Geoscientific Model Development* 14.8 (Aug. 2021). Publisher: Copernicus GmbH, pp. 4797–4842. ISSN: 1991-959X. DOI: 10.5194/gmd-14-4797-2021.

[79] Paul Norvig. *Interoperability of Fortran with C/C++: coding tutorial (2024)*. en. Jan. 2024. URL: https://www.paulnorvig.com/guides/interoperability-of-fortran-with-cc.html (visited on 07/02/2025).

[80] Jose Luis Jodra et al. “A Study of Memory Consumption and Execution Performance of the cuFFT Library”. In: *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. Nov. 2015, pp. 323–327. DOI: 10.1109/3PGCIC.2015.66.

[81] Greg Ruetsch and Paulius Micikevicius. “Optimizing Matrix Transpose in CUDA”. en. In: (2009). URL: https://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf.

[82] Wikipedia contributor. *Floating point operations per second*. en. Page Version ID: 1297964186. June 2025. URL: https://en.wikipedia.org/w/index.php?title=Floating_point_operations_per_second&oldid=1297964186 (visited on 07/11/2025).

[83] Karl Rupp. *CPU, GPU and MIC Hardware Characteristics over Time | Karl Rupp*. en-US. June 2013. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (visited on 07/11/2025).

[84] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[85] E. Adam Paxton et al. “Climate Modelling in Low-Precision: Effects of both Deterministic & Stochastic Rounding”. en. In: *Journal of Climate* 35.4 (Feb. 2022). arXiv:2104.15076 [physics], pp. 1215–1229. ISSN: 0894-8755, 1520-0442. DOI: 10.1175/JCLI-D-21-0343.1.

[86] Siyuan Chen et al. “Mixed-precision computing in the GRIST dynamical core for weather and climate modelling”. en. In: *Geoscientific Model Development* 17.16 (Aug. 2024). Publisher: Copernicus GmbH, pp. 6301–6318. ISSN: 1991-9603. DOI: 10.5194/gmd-17-6301-2024.

[87] Leibniz University Hannover. *PALM preprocessor options*. 2021. URL: https://palm.muk.uni-hannover.de/trac/wiki/doc/app/cpp_options (visited on 07/11/2025).