


ADVERTIMENT. L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons:  <https://creativecommons.org/licenses/?lang=ca>

ADVERTENCIA. El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons:  <https://creativecommons.org/licenses/?lang=es>

WARNING. The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license:  <https://creativecommons.org/licenses/?lang=en>

UAB

**Universitat Autònoma
de Barcelona**

Doctoral thesis

Research line: High performance computing

Porting of an Atmospheric Chemistry Solver to Parallel CPU-GPU Execution

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Department of Computer Architecture and Operating Systems

Author: Christian Guzman Ruiz

Director: Mario César Acosta Cobos

Co-Director: Eduardo Cesar Galobardes

Bellaterra (Cerdanyola del Vallès), September 12, 2024

Abstract

Earth System Models (ESMs) are crucial for understanding climate change, providing valuable insights into real-world phenomena and systems, with numerous applications in science and engineering. However, the scientific community requires greater computational power to reduce uncertainties in these models. Significant progress is being made by transitioning to heterogeneous computing systems that combine Central Processing Units (CPUs) and Graphics Processing Units (GPUs). These models can be accelerated notably by optimizing the most time-consuming components, such as chemistry, thereby reducing uncertainties and enhancing our understanding of climate dynamics.

The primary objective of this thesis is to improve the performance of chemistry solvers in atmospheric models by developing efficient solutions tailored to GPU-based heterogeneous architectures. Specifically, the focus is maximizing GPU performance, enhancing accuracy, ensuring portability, and minimizing development effort by minor modifications to the existing chemistry solver.

This thesis's critical contribution is developing a CPU-GPU version of the CAMP chemistry solver for the atmospheric model MONARCH, achieving significant acceleration. Furthermore, the developed chemical solver can be easily adapted to solve generic systems of Ordinary Differential Equations (ODEs) on the GPU due to its similarity to the CVODE library used in the CAMP CPU solver.

The open-source code has been released as an upgrade to CAMP to facilitate portability. The main contributions include a method for seamlessly integrating GPU code into chemistry solvers (Multi-cells), distributing computational loads across GPU threads (Block-cells), and an automatic load-balancing algorithm for concurrent CPU-GPU execution. Notably, the automatic load-balancing algorithm is designed to be simple, ensuring portability while delivering accelerated execution. Furthermore, a GPU linear solver has been developed following the Block-cells approach, enabling further exploration of Block-cells in iterative solvers and facilitating coupling with other systems—an important consideration given the widespread use of linear solvers in scientific and mathematical communities.

The results demonstrate a 250x speedup compared to the single-threaded CPU version when using four GPUs in a single node. In a node-to-node comparison, utilizing

four GPUs versus 80 CPU cores, the speedup achieved is 8.14x. When normalizing the cores used against other state-of-the-art solutions, the speedup is 16x. The substantial speedup achieved with minimal algorithmic changes underscores the significance of the new strategies developed in this thesis.

Resumen

Los modelos del sistema terrestre (ESMs) son cruciales para comprender el cambio climático, ya que brindan información valiosa sobre fenómenos y sistemas del mundo real, con numerosas aplicaciones en ciencia e ingeniería. Sin embargo, la comunidad científica requiere una mayor potencia computacional para reducir las incertidumbres en estos modelos. Se están logrando avances significativos mediante la transición a sistemas informáticos heterogéneos que combinan Unidades Centrales de Procesamiento (CPUs) y Unidades de Procesamiento Gráfico (GPUs). Estos modelos se pueden acelerar notablemente mediante la optimización de los componentes que consumen más tiempo, como la química, lo que reduce las incertidumbres y mejora nuestra comprensión de la dinámica climática.

El objetivo principal de esta tesis es mejorar el rendimiento de los solucionadores de química en modelos atmosféricos mediante el desarrollo de soluciones eficientes adaptadas a arquitecturas heterogéneas basadas en GPU. Específicamente, el enfoque se centra en maximizar el rendimiento de la GPU, mejorar la precisión, garantizar la portabilidad y minimizar el esfuerzo de desarrollo mediante la realización de modificaciones menores al solucionador de química existente.

La contribución fundamental de esta tesis es el desarrollo de una versión CPU-GPU del solucionador de química CAMP para el modelo atmosférico MONARCH, logrando una aceleración significativa. Además, el solver químico desarrollado se puede adaptar fácilmente para resolver sistemas genéricos de Ecuaciones Diferenciales Ordinarias (ODE) en la GPU, gracias a su similitud con la librería CVODE utilizada para el solver CPU de CAMP.

El código fuente abierto se ha publicado como una actualización de CAMP para facilitar la portabilidad. Las principales contribuciones incluyen un método para integrar fácilmente código GPU en los solucionadores de química (Multi-cells), distribuir las cargas computacionales entre los procesos de la GPU (Block-cells) y un algoritmo de equilibrio de carga automático para la ejecución concurrente de la CPU y la GPU. En particular, el algoritmo de equilibrio de carga automático está diseñado para ser simple, garantizando la portabilidad y ofreciendo una ejecución acelerada. Además, se ha desarrollado un solucionador lineal de GPU siguiendo el enfoque Block-cells, permitiendo una mayor exploración de Block-cells en solucionadores iterativos y facilitando el acoplamiento con otros sistemas, una consideración im-

portante dado el uso generalizado de solucionadores lineales en las comunidades científicas y matemáticas.

Los resultados demuestran una aceleración de 250x en comparación con la versión de CPU de un solo proceso utilizando cuatro GPU en un solo nodo. En una comparación de nodo a nodo, utilizando cuatro GPU frente a 80 núcleos de CPU, la aceleración lograda es de 8,14x. Al normalizar los núcleos utilizados frente a otras soluciones de última generación, la aceleración es de 16x. La aceleración sustancial lograda con cambios algorítmicos mínimos subraya la importancia de las nuevas estrategias desarrolladas en esta tesis.

Resum

Els Models del Sistema Terrestre (ESMs) són crucials per a comprendre el canvi climàtic, ja que brinden informació valuosa sobre fenòmens i sistemes del món real, amb nombroses aplicacions en ciència i enginyeria. No obstant això, la comunitat científica requereix una major potència computacional per a reduir les incerteses en aquests models. S'estan aconseguint avanços significatius mitjançant la transició a sistemes informàtics heterogenis que combinen Unitats Centrals de Processament (CPUs) i Unitats de Processament Gràfic (GPUs). Aquests models es poden accelerar notablement mitjançant l'optimització dels components que consumeixen més temps, com la química, així reduint les incerteses i millorant la nostra comprensió de la dinàmica climàtica.

L'objectiu principal d'aquesta tesi és millorar el rendiment dels solucionadors de química en models atmosfèrics mitjançant el desenvolupament de solucions eficients adaptades a arquitectures heterogènies basades en GPU. Específicament, la tesis es centra en maximitzar el rendiment de la GPU, millorar la precisió, garantir la portabilitat i minimitzar l'esforç de desenvolupament mitjançant la realització de modificacions menors al solucionador de química existent.

La contribució fonamental d'aquesta tesi és el desenvolupament d'una versió CPU-GPU del solucionador de química CAMP per al model atmosfèric MONARCH, aconseguint una acceleració significativa. A més, el solver químic desenvolupat es pot adaptar fàcilment per a resoldre sistemes genèrics d'Equacions Diferencials Ordinàries (ODE) a la GPU, gràcies a la seva similitud amb la llibreria CVODE utilitzada per al solver CPU de CAMP.

El codi font obert s'ha publicat com una actualització de CAMP per a facilitar la portabilitat. Les principals contribucions inclouen un mètode per integrar fàcilment codi GPU als solucionadors de química (Multi-cells), distribuir la càrrega computacional entre els processos de la GPU (Block-cells) i un algoritme d'equilibri de càrrega automàtic per a l'execució concurrent de la CPU i la GPU. Particularment, l'algoritme d'equilibri de càrrega automàtic està dissenyat per ésser simple, garantint la portabilitat i oferint una execució accelerada. A més, s'ha desenvolupat un solucionador lineal de GPU seguint l'enfocament Block-cells, la qual cosa permet una major exploració de Block-cells en solucionadors iteratius i facilita l'acoblament amb altres sistemes, una consideració important donat l'ús generalitzat de solu-

cionadors lineals en les comunitats científiques i matemàtiques.

Els resultats demostren una acceleració de 250x en comparació amb la versió de CPU d'un sol procés utilitzant quatre GPU en un sol node. En una comparació de node a node, utilitzant quatre GPU enfront de 80 nuclis de CPU, l'acceleració obtinguda és de 8,14x. Normalitzant els nuclis utilitzats enfront d'altres solucions d'última generació, l'acceleració és de 16x. L'acceleració substancial aconseguida amb canvis algorítmics mínims subratlla l'importància de les noves estratègies desenvolupades en aquesta tesi.

Acknowledgment

I express my deepest gratitude to Mario Acosta, Eduardo Cesar, and Oriol Jorba for their immense support throughout this thesis. Your suggestions and revisions were invaluable; this work would not have achieved its current quality without your guidance.

I would also like to thank Matthew Dawson and Guillermo Oyarzun for their strong early support in helping me understand CAMP and the data arrangements for GPUs.

Thank you to Camille Mouchel-Vallon and Hervé Petetin for their valuable revisions and suggestions during the final stages of this thesis.

To my supervisors, Kim Serradell and Carlos Perez, for their efforts in securing research funding.

Regarding funding sources, this work was partially supported by the Ministerio de Ciencia, Innovación y Universidades as part of the BROWNING project (RTI2018-099894-BI00 funded by MCIN AEI/10.13039/501100011033) and part of the CAROL project (PID2020-113614RBC21 funded by MCIN AEI/10.13039/501100011033), the Generalitat de Catalunya (2021-SGR00785, 2021-SGR-00574 and 2021-SGR-01550) and the AXA Research Fund through the AXA Chair on Sand and Dust Storms established at the Barcelona Supercomputing Center (BSC). We acknowledge the computer resources at Marenostrom 4 CTE-POWER and Marenostrom 5. This thesis expresses the opinions of the authors and not necessarily those of the funding commissions.

Contents

1	Introduction	1
1.1	Motivation and Context	1
1.2	State of the art	4
1.2.1	Parallel strategies to solve chemistry mechanisms	4
1.2.2	Parallel computing paradigms for GPU	5
1.3	Objectives	7
1.4	Contributions	8
1.5	Outline of the thesis	11
2	Methods	13
2.1	The atmospheric chemistry model MONARCH	13
2.1.1	Mathematical considerations	15
2.1.2	Computational implementation	21
2.2	The chemistry solver CAMP	22
2.2.1	Mathematical considerations	23
2.2.2	Computational implementation	24
2.3	Profiling tools	31
2.3.1	NVVP metrics	31
2.3.2	NCU metrics	32

3	Efficient data arrangement for GPU adaptation of ODE systems in atmospheric chemistry solvers as multiple cells	34
3.1	Introduction	34
3.2	Background	35
3.2.1	State of the art	35
3.2.2	Computational description	35
3.3	Implementations	38
3.4	Test environment	39
3.5	Results and discussion	40
3.6	Conclusions	41
4	Optimized thread-block arrangement in a GPU implementation of a linear solver for atmospheric chemistry solvers	43
4.1	Introduction	43
4.2	Background	44
4.2.1	State of the art	44
4.2.2	Computational description	45
4.3	Block-cells implementation	46
4.3.1	Coupling a GPU BCG linear solver in CAMP	46
4.3.2	Block-cells kernel configurations	47
4.4	Test environment	51
4.4.1	Hardware	51
4.4.2	Experimental setup	51
4.5	Results and discussion	53
4.5.1	NVVP profiling	53
4.5.2	Speedup	55
4.6	Conclusions	60

5	Extending the Block-Cells Approach for GPU-Accelerated ODE Solvers: Implementation, Testing, and Profiling in Box and 3D MONARCH Models	62
5.1	Introduction	62
5.2	Background	63
5.2.1	State of the art	63
5.2.2	Computational description: introduction to Block-cells	63
5.3	Implementations	64
5.3.1	Porting	64
5.4	Test environment	68
5.4.1	Box model runs	69
5.4.2	MONARCH run	70
5.5	Results and discussion	72
5.5.1	Box model runs	72
5.5.2	MONARCH model runs	73
5.6	Conclusions	78
6	Design and Performance Assessment of an Automatic Load Balancing CPU-GPU Algorithm for Atmospheric Chemistry Solvers across Marenostrom 4 and 5	80
6.1	Introduction	80
6.2	Background	81
6.2.1	State of the art	81
6.2.2	Computational description	83
6.3	Implementations	84
6.3.1	Porting	84
6.3.2	Load balance	87
6.4	Test environment	89
6.4.1	Box model experiments	90
6.4.2	MONARCH experiments	92
6.5	Results and discussion	93

CONTENTS

6.5.1	Box model runs	93
6.5.2	MONARCH model runs	94
6.6	Conclusions	98
7	Conclusions	101
7.1	Main findings	101
7.2	Future work	103

Chapter 1

Introduction

This chapter provides a general overview of the thesis, organized into the following subsections: the primary motivation driving this research and scientific context, the state-of-the-art in the field, the primary and subsequent objectives of the study, and its fundamental contributions. The chapter concludes with a subsection that outlines and summarizes the content of the thesis, offering a chapter-by-chapter breakdown of what is to come.

1.1 Motivation and Context

Climate change is a complex intergovernmental challenge, influencing various ecological, environmental, political, and economic disciplines [1]. Understanding climate change is vital for developing adaptation initiatives and preparing for extreme events like wildfires, floods, and droughts [2].

In the last decade, our understanding of climate change has significantly increased [3], alongside the demand for policy-relevant climate information. While there is great confidence that climate change is occurring, uncertainties remain [4], particularly regarding the amount of greenhouse gas and aerosol emissions and, even more significantly, the degree of warming and the likely impacts [5]. To reduce these uncertainties, enhancing the capability and comprehensiveness of Earth System Models (ESMs) is essential to represent new scenarios for our future climate with ever-increasing realism and detail [6].

Earth system models (ESM) aim to represent all relevant interactions of the Earth system components (i.e., atmosphere, ocean, sea ice, land surface, biosphere, ice sheets). They provide valuable information on the nature of real-world phenomena and systems [7], with many applications in science and engineering [8]. Remarkably, these models play an increasingly important role in understanding the potential implications of climate change [9].

Therefore, the scientific community necessitates higher model resolution, more experiments and ensembles to quantify uncertainty, increased complexity of ESMs, and more extended simulation periods compared to the current capabilities of climate models [6]. However, the enhancement of ESMs heavily depends on the availability of substantial computing power and data storage capacity [10].

High-Performance Computing (HPC) has evolved significantly over the past few years, transitioning from a supportive technology within the academic research community to a critical component of the numerical modeling framework [11], being essential for Earth System Modeling [12].

The advancements in HPC necessitate some degree of rewriting to fully leverage new and emerging architectures. As HPC technology evolves, the Earth Sciences community is shifting towards heterogeneous computing systems consisting of various processors. In these systems, different software and hardware components interact to enhance the computational power of HPCs [13]. Many applications face severe challenges in exploiting the unique characteristics of modern heterogeneous systems.

Current compute nodes typically combine a Central Processing Unit (CPU) with multiple Graphics Processing Units (GPUs). Integrating GPUs into Earth System Models (ESMs) requires refactoring the code to fit a different computing paradigm. This task is complex and time-consuming, especially for models with hundreds of thousands of lines of code, such as ESMs [14]. Hence, most optimization work on ESMs generally focuses on specific components of the model rather than the entire system. This approach preserves the CPU version for portability while leveraging the computational power of accelerators for particular parts of the code [15].

The atmospheric components of ESMs (i.e. atmospheric models) are a computer-coded representation of the atmosphere's dynamical, physical, chemical, and radiative processes [16]. Among ESM components, chemistry stands out for its significant computational burdens, typically accounting for between 50% and 95% of the total execution time [17] [18] [19].

Different levels of complexity are required in the chemistry representation of atmospheric components. Detailed gas-phase chemistry is necessary to model air quality and its impacts on health and greenhouse gas lifetimes. In addition, further detailed particulate matter modeling is required to understand its direct and indirect effects on climate and how it can affect health and ecosystems (terrestrial and water). Overall, the solution of a chemical mechanism (a set of chemical reactions describing the fate of dozens to thousands of chemical species with a wide range of lifetimes from seconds to weeks) can be the most expensive part of the problem. The resulting ODE system is mathematically *stiff* [20], and special care needs to be exercised in choosing the numerical integration scheme [17]. Other factors that can affect the computational cost of the solution of an ODE system are the desired accuracy and the hardware architecture used.

A key motivation behind this work is the under-utilization of GPUs in atmospheric

models despite the availability and power of these resources for several years. This thesis uses as a test bed the Multiscale Online Nonhydrostatic Atmosphere Chemistry (MONARCH) model [21] [22] [23] [24], a chemical weather prediction system developed and maintained by the Atmospheric Composition and Computational Earth Sciences groups at the Barcelona Supercomputing Center (BSC). The model provides operational regional mineral dust forecasts for the World Meteorological Organization (WMO; <https://dust.aemet.es/>) and participates in the WMO Sand and Dust Storm Warning Advisory and Assessment System for Northern Africa-Middle East-Europe (<http://sds-was.aemet.es/>). Since 2012, the system has contributed daily global aerosol forecasts to the multi-model ensemble of the International Cooperative for Aerosol Prediction (ICAP) initiative [25]. Since 2022, MONARCH is one of the members of the multi-model system of the Copernicus Atmosphere Monitoring Service (CAMS)—Air Quality Regional Production (<https://www.regional.atmosphere.copernicus.eu>).

The chemistry component in MONARCH serves as a prime example of being the most time-consuming element among the atmospheric components, consuming around 80% of the total execution time. This significant computational burden motivated our focus on optimizing this aspect of the model.

At the start of this thesis, MONARCH was integrating a new and promising chemistry component: the Chemistry Across Multiple Phases (CAMP) library [26]. CAMP is designed to streamline development by minimizing hard-coded elements within the code. Its library-based structure accelerates developments by enabling independent execution from the rest of the atmospheric model, avoiding unnecessary computations. CAMP can handle various types of chemistry, including gases and aerosols, using a unified solving procedure, in contrast to the default MONARCH and traditional approach that employs highly unrelated solvers for different components. Thus, enhancing CAMP effectively improves multiple solvers simultaneously, which would otherwise be much more complex. These advantages make CAMP an ideal host for our development. The contributions of this thesis further motivate the integration of CAMP into the broader atmospheric science community, solidifying its potential as a critical component for future advancements.

A key benefit of applying GPU computing to MONARCH and CAMP is the ability to effectively handle highly parallel computational workloads. An atmospheric domain typically contains millions of chemical concentrations. While a node may have access to tens of CPU processes, a GPU can manage millions of threads simultaneously. Implementing GPU computing thus offers a significant performance advantage due to the high degree of parallelization inherent in GPU architecture. Moreover, the simultaneous use of CPU and GPU architectures allows for an even greater computation acceleration by leveraging each of its strengths. Therefore, the primary motivation of this thesis is to enhance the performance of atmospheric models by exploiting the substantial computational resources provided by GPUs.

1.2 State of the art

The state-of-the-art discussion is divided into two categories: historical approaches to parallelized chemistry and available parallel computing paradigms for GPUs. The first category sets the background leading up to our contributions, highlighting the evolution of techniques and methodologies in parallelized chemistry. The second category presents the current GPU-related tools and paradigms for accelerating chemistry to enhance and build upon the established background, offering new opportunities for performance improvements in computational chemistry.

1.2.1 Parallel strategies to solve chemistry mechanisms

Historically, chemical solvers have been designed for single-thread execution. For example, this approach is employed by the Kinetic PreProcessor [27], which has been widely used to generate gas-phase mechanism code with several ODE solver options.

However, chemical systems in ESMs can benefit from massive parallelization. The initial implementations relied on domain decomposition across multiple interconnected CPUs, using MPI for communications between them [28]. This technique divides the computational domain (covering a global or regional geographical area) into smaller regions representing a fractional volume of the atmosphere, typically called grid cells in the community, which will be called cells from now on. The model assigns collections of cells to independent threads to solve the many physical and chemical processes in the atmosphere in parallel.

ESMs have gradually integrated more parallel programming interfaces such as OpenACC, OpenMP, and CUDA. MPI is the most used tool to distribute work across independent supercomputer nodes. MPI can further be used along another parallel approach to divide the load across individual CPU or GPU threads.

The typical alternatives for CPU parallelization are MPI-Only and OpenMP + MPI. Typically, MPI is preferred for simplicity in parallelizing most of the model, while OpenMP is used for small code sections. However, various studies on similar models reported that using MPI achieves the same efficiency as OpenMP [29] [30].

The GPU alternatives have reported in multiple studies positive results [31] [32] [33]. The main advantage is the combination of the MPI and GPU, leading to a CPU+GPU implementation. Both architectures can run simultaneously in this combination, leading to high accelerations.

However, GPU development in the chemistry field is at an early stage, where the usual approach is to choose between the two architectures. The GPU approach has shown performance improvements in several studies, even in this situation. A standard comparison metric is the speedup of a GPU against a single-thread execution

of the CPU version, which can differ considerably between studies. For example, a Rosenbrock solver in the CAM4-chem model achieves an $11.7\times$ speedup [34]. A chemistry module Kinetic PreProcessor (KPP) version in the climate model EMAC achieves $20.4\times$ speedup using 1 GPU against the single-thread CPU version [35]. A Runge-Kutta-Chebyshev (RKC) algorithm explicitly developed for GPU execution achieves up to a $59\times$ speedup [36]. The first and second studies solve gas-phase chemical kinetics, a time-expensive part of the chemistry in atmospheric models. However, the last example is only available for slightly stiff chemical kinetics and not for the complex chemistry of an atmospheric model.

The significant difference in speedup between studies highlights the difficulty and potential of porting the code to the GPU. Generally speaking, faithful translations of the original version ensure portability, while less faithful adaptations (for example, a different algorithm) can achieve more significant speedups and require more development effort.

Most current GPU solutions for chemistry solvers follow a similar parallelization strategy. Specifically, each computational process (i.e., GPU thread) solves the workload of a cell. A cell is the smallest unit obtained by discretizing the domain of study (e.g., the atmosphere in atmospheric models). Each cell describes the state of the atmosphere in that specific domain region (including variables such as chemical species concentrations, temperature, pressure, etc.). In each cell, the prognostic equations describing the evolution of state variables are solved through multiple algebraic operations over the concentrations array. During solving, many calculations related to specific species are independent. This means that they can be computed in parallel.

Recent works are adopting similar concepts. Specifically, a GPU implementation for calculating chemical rates in the MCIM chemistry solver achieves $66\times$ speedup without data transfer time [37] [38]. This approach ensures portability as the algorithm is untouched while achieving outstanding performance by adapting the routine specifically for GPUs. Although the ported code is tiny compared to the entire chemistry solver, the approach reflects the potential advantages of a GPU implementation.

1.2.2 Parallel computing paradigms for GPU

From the code languages used to work with the GPU, CUDA stands out as the most utilized in relevant state-of-the-art chemistry modules, such as KPP or CAM4-Chem [35] [34]. Nvidia has developed and extensively documented CUDA. Nvidia also provides robust profiling tools and libraries, such as Nsight and cuBLAS <https://developer.nvidia.com/cublas>, which are only available for CUDA. In addition, the CUDA lower-level interface with the GPU, which is very similar to the base CPU implementation in C, C++, or Fortran, adds flexibility to develop optimization strategies.

Languages based on compiler directives (or pragmas), such as OpenACC [39] or OpenMP [40], are adopted by various weather and climate modeling groups [41] [42] [43] [33]. Directive languages offer a higher-level interface than CUDA, aiming for more productivity and portability. However, the programmer loses flexibility compared to the lower-level CUDA language, which could make it challenging to find optimizations. Also, these directive-based approaches generate automated kernel code, which can obscure the underlying operations and make it challenging to analyze and address performance issues effectively.

An advantage of some directive languages is their support for multiple GPU vendors. For instance, OpenMP supports both Nvidia and AMD GPUs. This is an advantage over CUDA, which is specific to Nvidia GPUs. However, translating CUDA to AMD's HIP (Heterogeneous-Compute Interface for Portability) <https://github.com/ROCm/HIP> is relatively straightforward due to the similarities between the two languages. HIP is compatible with Nvidia GPUs but lacks the latest utilities from Nvidia's profiling tools, such as the Roofline model from Nsight. Therefore, while directive languages offer immediate portability across different GPUs, CUDA's detailed performance tools and mature ecosystem remain beneficial for in-depth optimization. In the future, adding support for other GPU vendors could be achieved without extensive effort, maintaining a balance between performance and portability.

Another solution is the use of source-to-source parsers. For example, the authors of the GPU EMAC-KPP study [35] wrote a parser to translate the Fortran code to CUDA [35], suggesting that parsers can facilitate the development work. However, this parser lacks portability and is only available for the KPP library, making it difficult to use in other chemistry solvers.

Nevertheless, the climate community is also developing parsers with portability in mind. For example, the CLAW translator aims to facilitate the transition from Fortran to OpenACC with minimal or no changes to the original code [44]. This approach is highly useful for physical parameterizations written in Fortran. However, CLAW may not be applicable for codes programmed in C, especially considering its focus on Fortran, as indicated by the examples available on its GitHub repository <https://github.com/claw-project/claw-language-specification>. This limitation makes using CLAW for C code challenging and may even result in missing routines necessary for accurate translation.

Another example of a source-to-source tool is LOKI <https://github.com/ecmwf-ifs/loki>, which offers more comprehensive documentation than CLAW. However, LOKI is also designed primarily for Fortran routines. Despite this, LOKI is in an early stage of development with frequent updates, suggesting it could become a valuable tool in the future. As it evolves, LOKI may offer greater flexibility and support for a broader range of programming languages, including C, making it a promising option to watch for future developments.

Another possibility for porting available solvers is utilizing GPU libraries, such as

cuBLAS or cuSOLVER <https://docs.nvidia.com/cuda/cusolver/index.htmls>, designed to maximize performance and continually updated. These libraries offer highly optimized routines for linear algebra operations and solvers, leveraging the total computational power of GPUs. However, they present a significant limitation: not all parts of the code can be run exclusively on the GPU. Specifically, the convergence condition of iterative algorithms cannot be evaluated on the GPU. This condition determines whether the algorithm should terminate or continue with more iterations. In other words, while the variable containing the error could theoretically reside on the GPU, the need for communication between the GPU and the CPU arises because these libraries are invoked from the CPU. Each iteration would necessitate transferring data between the CPU and GPU to check the convergence condition and then re-invoke the GPU for further computations. This frequent communication overhead can considerably slow down the performance, diminishing the benefits of using these high-performance libraries for the solver.

In summary, while GPU libraries like cuBLAS and cuSOLVER offer powerful capabilities, their reliance on CPU-GPU communication for convergence checking and other iterative controls would result in substantial time overhead.

In addition, various studies have reported optimizations to routines of the cuBLAS and cuSolver libraries. For instance, a Cholesky factorization approach outperforms cuSolver by 10% in single precision [45]. Similarly, another study evaluated and optimized BLAS operations on GPUs, achieving significant performance gains over standard implementations [46]. These findings suggest that while cuBLAS and cuSolver provide robust and high-performing solutions, there are instances where tailored optimizations can yield better performance.

This indicates that more effective optimizations may be available than these libraries offer. Custom optimization strategies tailored to the requirements and characteristics of a specific application or code might achieve superior performance, particularly by addressing specific bottlenecks and leveraging the unique aspects of the computational workload.

1.3 Objectives

The main objective of this thesis is to improve the computational performance of existing chemistry solvers used in atmospheric chemistry models, developing efficient solutions for GPU heterogeneous architectures. Our proposal is a novel data partitioning to increase the parallel workload, aiming for higher GPU performance, high accuracy, portability, and low development effort with minimal changes to the existing chemistry solver.

To achieve the main goal of the thesis, the following specific objectives are defined:

1. Lay the groundwork of a GPU implementation, defining how to adapt chemistry models for GPU execution, usually designed for the CPU. Adaptation must require an affordable development effort while maintaining accuracy.
2. Design a novel strategy to increase the parallel workload. Ensure higher performance than traditional approaches while keeping the accuracy and capabilities of the chemistry solver.
3. Extend the strategy to the whole chemistry solver and test the solution in a 3D complex atmospheric chemistry model, quantifying the accuracy and achieved performance.
4. Design and deploy an optimal GPU load-balancing implementation to maximize the utilization of CPU and GPU resources in heterogeneous runs.

An iterative and incremental methodology has been followed, in which each previously defined specific objective has been documented, tested, and analyzed. This includes exploring new optimization techniques, improving and standardizing the code, validating the code in an atmospheric model, and making final adjustments and optimizations. This approach ensures that each step builds on the previous one, resulting in a robust and efficient solution.

By focusing on custom optimizations and potentially integrating or even enhancing standard GPU libraries with additional tailored strategies, the thesis can aim to achieve significant performance improvements. This approach aligns with the iterative and incremental methodology of testing and analyzing each optimization step, ensuring that the final implementation is highly efficient and well-suited to the complex requirements of atmospheric chemistry modeling.

1.4 Contributions

The ultimate contribution of this thesis is developing a new solution that advances state-of-the-art performance and utilization of GPU and CPU resources in atmospheric chemistry models. The key contributions of this work can be summarized as follows:

1. Algorithmic contributions to chemistry solvers. These algorithms balance minimal development effort and optimal efficiency, representing an improvement over the state-of-the-art. The specific algorithms include:
 - (a) The Multi-cells approach to easily integrate GPU functions in chemistry solvers (details in chapter 3).
 - (b) The Block-cells distribution of the computational load for GPU computing (details in chapter 4).

- (c) An automatic load balancing algorithm for simultaneous CPU-GPU execution (details in chapter 6).
2. Release of our developments as an upgrade to CAMP. Our contributions include the CPU-GPU version with the load balance algorithm following the Block-cells implementation. The GPU code introduced corresponds to a new ODE solver for CAMP, replicating the CPU solver algorithm to port its capabilities. Consequently, our developments provide a GPU ODE solver for chemistry that can be easily adapted to solve generic ODE systems. This adaptability is particularly notable since the CAMP CPU solver corresponds to a slightly modified version of the CVODE library [47].
3. The release of a GPU linear solver following the Block-cells strategy [48] <https://github.com/cguzman95/BCG-CAMP-MONARCH>. This code is an isolated version of the linear solver used within the ODE solver, making it easier to understand the performance limitations of Block-cells on iterative solvers. Additionally, it enables coupling with other systems, which is significant given the widespread use of linear solvers in the scientific and mathematical communities.
4. Coupling our developments with MONARCH, enabling it to utilize CPU and GPU resources and benefit from an accelerated version of CAMP.
5. One manuscript, currently in progress and divided into Chapters 5 and 6, will be submitted for publication as soon as possible.
6. One peer-reviewed publication included as Chapter 4 and published in [49] C. Guzman Ruiz, M. Acosta, O. Jorba, E. Cesar Galobardes, M. Dawson, G. Oyarzun, C. Perez Garcia-Pando, and K. Serradell, "Optimized thread-block arrangement in a GPU implementation of a linear solver for atmospheric chemistry mechanisms," *Computer Physics Communications*, vol. 302, p. 109240, Sep. 2024
7. One proceeding publication included as Chapter 3 and published in [50] C. G. Ruiz, M. Dawson, M. C. Acosta, O. Jorba, E. C. Galobardes, C. P. Garcia Pando, and K. Serradell, "Adapting Atmospheric Chemistry Components for Efficient GPU Accelerators", in *Proceedings of Eighth International Congress on Information and Communication Technology*, X.-S. Yang, R. S. Sherratt, N. Dey, and A. Joshi, Eds. Singapore: Springer Nature Singapore, 2023, pp.129–138. *Proceedings of Eighth International Congress on Information and Communication Technology*, Springer Nature Singapore, 2023, pp.129-138.
8. Participation in conferences and workshops:
 - (a) [51] C. Guzman Ruiz, M. C. Acosta, O. Jorba, and C. P. García-Pando, "Novel approaches to accelerate chemistry for climate models,"

- in *Platform for Advanced Scientific Computing (PASC) 2023*, Congresscenter Davos, Switzerland, Jun. 2023. [Online]. Available: <https://pasc23.pasc-conference.org/presentation/>
- (b) [52] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “CAMP First GPU Solver: A Solution to Accelerate Chemistry in Atmospheric Models,” in *9th BSC Doctoral Symposium*, Universitat Politècnica de Catalunya, Spain, May 2022. [Online]. Available: <https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/9th-bsc-doctoral-symposium-2022>
- (c) [53] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “CAMP First GPU Solver: A Solution to Accelerate Chemistry in Atmospheric Models,” in *7th HPC Workshop of the European Network for Earth System Modelling*, Barcelona SuperComputing Center, Spain, May 2022. [Online]. Available: <https://portal.enes.org/hpc-workshops-detailed/#hpc7>
- (d) [54] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “Studying a new GPU treatment for chemical modules inside CAMP,” in *19th Workshop on HPC in Meteorology*, Online, Sep. 2021. [Online]. Available: <https://events.ecmwf.int/event/169/timetable/>
- (e) [55] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “Exploiting parallelism for CPU and GPU linear solvers on chemistry for atmospheric models,” in *8th BSC Doctoral Symposium*, Online, May 2021. [Online]. Available: <https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/8th-bsc-doctoral-symposium-2021>
- (f) [56] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “Accelerating Atmospheric Models using GPU,” in *The 2020 International Conference on High Performance Computing & Simulation (HPCS 2020)*, Mar. 2021. [Online]. Available: <https://hpcs2020.cisedu.info/>
- (g) [57] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “Accelerating Chemistry Modules in Atmospheric Models Using GPUs,” in *6th ENES Workshop on High Performance Computing for Climate and Weather*, Online, May 2020. [Online]. Available: <https://www.esiwave.eu/events/6th-hpc-workshop>
- (h) [58] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “Accelerating Chemistry Modules in Atmospheric Models Using GPUs,” in *NVIDIA GTC 2020 Spring*, Online, Mar. 2020. [Online]. Available: <https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s22005/>

1.5 Outline of the thesis

This thesis is organized into seven chapters, each addressing different contributions of the research. Chapter 2 introduces the numerical models employed in this thesis and the performance tools used to derive performance metrics on the GPU. The chapter briefly describes the atmospheric chemistry model MONARCH and the chemistry library CAMP. Both are the primary tools for solving reactive chemistry in the atmosphere and have been adopted as test beds for the new GPU solutions designed and developed in this thesis.

Chapter 3 corresponds to the first steps of adapting CAMP to a GPU computing paradigm. It presents the Multi-cells strategy, where multiple cells are solved as a single system. Originally, CAMP was designed to contain data from a single chemical system, and the parallelization, based on parallelizing the solving of chemical systems, was handled with MPI. Consequently, the call to the solving routine of CAMP from the atmospheric model contains a cell loop, calling the CAMP solving routine for each cell individually. Therefore, using any GPU routine in CAMP would result in a call to that routine for each cell. Considering that a typical experiment generates millions of cells, with a minimal load for each, the data transfer cost with the GPU would be very high many times. Ideally, a single call with the maximum load possible is desirable. The Multi-cells implementation achieves this.

The chapter also includes a GPU strategy for CAMP's most time-consuming function, including the time expended in data transfers between CPU and GPU.

Chapter 4 upgrades the Multi-cells strategy to a new one called Block-cells, comparing both approaches in a small linear solver routine to find the best configuration. The current state of the art for GPUs distributes the workload of solving a cell to a GPU thread. The Block-cells solution further divides the cell's load into smaller parallel computations, where each GPU thread solves a chemical concentration of a cell. This technique is already used to solve linear systems of equations in GPU, where each thread solves an equation [59]. As a disadvantage, translating the CPU code to CUDA requires more development effort due to handling synchronizations between threads, such as sharing an error between the threads of the same cell when a negative-signed concentration is found. As an advantage, it exploits the high parallelization capacity of the GPU, improving the performance.

The chapter also includes a performance assessment of Multi-cells and multiple Block-cells configurations and profiling metrics.

Chapter 5 validates the Block-Cells implementation in a MONARCH experiment, including its accuracy and performance evaluation. This involves extending Block-cells from a linear solver routine within CAMP to the ODE-solving procedure, translating nearly all the ODE solver code utilized in CAMP.

Chapter 6 updates the CAMP GPU version to a CPU-GPU heterogeneous computation model. This includes evaluating the optimal load distribution between the

1.5. OUTLINE OF THE THESIS

CPU and GPU. The validation is performed again on MONARCH, but the results correspond to Marenostrum 5 HPC facility since the previous Marenostrum 4 architecture was replaced for this new version. It also includes a discussion of the performance achieved on Marenostrum 5 against Marenostrum 4.

Finally, Chapter 7 summarizes the main findings of the research and discusses the perspectives for future works.

Chapter 2

Methods

This chapter extends on MONARCH and CAMP as the atmospheric and chemistry components used to test our developments. It also includes the profiling tools and metrics used to assess performance.

The first section provides an overview of MONARCH, including its relevance, components, and mathematical and computational description. The second section offers a mathematical and computational description of CAMP and the chemistry solved in this thesis. The final section discusses the profiling tools and metrics employed to evaluate performance.

2.1 The atmospheric chemistry model MONARCH

An atmospheric model is a mathematical representation of the spatial and temporal distribution of state variables in the atmosphere. The computational domain is either global or regional, composed of cells representing a fractional volume of the atmosphere [60]. If chemistry processes are considered, such models are known as atmospheric chemistry models or chemical transport models [61]. The physico-chemical processes considered in such a model are the emissions of inert or reactive chemical species from anthropogenic or natural sources, the transport by advection in the direction of the wind and lateral and vertical diffusion, the photochemical transformations in the atmosphere, and the dry and wet deposition towards the surface.

Atmospheric chemistry models solve the mass balance equations:

$$y_t = \underbrace{-\nabla \cdot (\bar{u}y)}_{\text{advection}} + \underbrace{\nabla \cdot (\bar{K}\nabla y)}_{\text{turbulent diffusion}} + \underbrace{f(t, y)}_{\text{chemical kinetics}} + \underbrace{\sum_{i=1}^p r_i(t, y)}_{\text{other processes}} \quad (2.1)$$

The combined effects of advection, diffusion, chemical kinetics, and additional processes such as emission, deposition, aerosol thermodynamics, and interphase mass transfer govern the evolution of the vector field y representing chemical tracer concentrations. These processes are subject to appropriate initial and boundary conditions. The wind field vector \bar{u} and the turbulent diffusion tensor \bar{K} are typically pre-computed by a numerical weather prediction model and are constrained by observations through offline data assimilation [17].

In this thesis, the Multiscale Online Non-hydrostatic Atmosphere Chemistry model (MONARCH) is used as a testbed for the GPU solutions investigated to speed up the computation of the chemistry solver. MONARCH is a chemical weather modeling system that can be used at multiple spatial scales, ranging from regional scales at single-digit kilometer resolutions with explicit convection to coarse-resolution global scales with parameterized convection [21] [62]. MONARCH is continuously developed at the Barcelona Supercomputing Center (BSC) with a focus on mineral dust and other aerosols [21] [63] [64] [65], atmospheric chemistry [23] [24] [22] [24], emissions [66], data assimilation [67], workflow management [68], and operational forecasting [69] [25]. As introduced in Chapter 1, MONARCH contributes to several operational activities, from global aerosol forecasting (ICAP) to air quality over Europe under the Copernicus Programme (CAMS).

MONARCH consists of advanced chemistry and aerosol packages coupled online with the Non-hydrostatic Multiscale Model on the B-grid (NMMB) [70] [71]. MONARCH runs on both global and regional simulations. Different chemical processes are implemented following a modular operator-splitting approach to solving the advection, diffusion, chemistry, dry and wet deposition, and emission processes. NMMB is the model taking care of meteorology, making the meteorological state variables available at each internal time step of the model to solve the chemistry. To maintain consistency with the meteorological solver, the chemical species are advected and mixed at the corresponding time step of the meteorological tracers using the same numerical schemes implemented in the NMMB. The advection scheme is Eulerian, positive definite and monotone, maintaining consistent mass conservation of the chemical species within the study domain.

Table 2.1 summarizes the chemistry processes currently considered in the default version of MONARCH. The gas-phase chemistry solves an extended version of the Carbon Bond 2005 chemical mechanism (CB05) [72] [73]. The CB05 is well formulated for urban to remote tropospheric conditions, and it uses photolysis rates computed with the Fast-J photolysis model [74]. A mass-based aerosol module describes the life cycle of dust, sea salt, black carbon, organic matter (primary and secondary), sulfate (HSO_4^- , SO_4^{2-}), ammonium (NH_4^+), and nitrate (NO_3^-) aerosol components [65]. The resulting ODE system of the gas-phase chemistry is solved by default with an Eulerian backward iterative (EBI) solver. A modular coupling with the CAMP [26] library was recently introduced in MONARCH to allow a flexible chemistry configuration. CAMP is further described in the following section.

Table 2.1: Chemistry and aerosol processes available in MONARCH through configuration setup.

Process	Scheme
Tropospheric gas-phase chemistry	Carbon Bond 2005 (CB05) extended mechanism [72]
Aqueous sulfate formation	SO_2 oxidation by O_3 and H_2O_2 [65]
Inorganic Aerosol mechanism	EQSAM thermodynamic equilibrium model [75]
Organic Aerosol mechanism	Two-product scheme [76] or Simple non-volatile scheme [77]
Photolysis rates	Online Fast-J photolysis scheme [74]
Dry deposition of gas species	Wesley resistance approach [78]
Dry deposition of aerosols	Zhang scheme [79]
Wet deposition of gas species	Grid and sub-grid scale scavenging [80]
Wet deposition of aerosols	Adjustment scheme [21]
Biogenic emissions	Online MEGANv2.04 biogenic model [81]
Dust emissions	multiple mineral dust schemes [21] [82]
Sea salt emissions	multiple sea salt schemes [64] [83]
Pollen emissions	5 taxon emission scheme [84]
Dust Mineralogical composition	Explicit representation [85]

2.1.1 Mathematical considerations

This section focuses on the algorithmic and scientific approaches in the NMMB dynamics, covering aspects such as the primary model equations and discretization in time and space.

Model equations

Let s represent a generalized mass-based, terrain-following vertical coordinate that varies from 0 at the top of the model atmosphere to 1 at the surface [86]. Let π denote the hydrostatic pressure, and let π_{sfc} and π_T be the hydrostatic pressures at the surface and the top of the model atmosphere. The difference in hydrostatic pressure

2.1. THE ATMOSPHERIC CHEMISTRY MODEL MONARCH

between the surface and the top of the model column is given by $\mu = \pi_{\text{sfc}} - \pi_T$. π_T is a nonnegative constant, while π_{sfc} varies with time and horizontal position.

In this hybrid coordinate system, the hydrostatic pressure is computed using the formula:

$$\pi(x, y, s, t) = \pi_T + \sigma_1(s)\Pi + \sigma_2(s)\mu(x, y, t), \quad (2.2)$$

where Π is the constant depth of the hydrostatic pressure layer at the top of the model atmosphere, $\sigma_1(s)$ is zero at both the top and bottom of the atmosphere, and $\sigma_2(s)$ increases from 0 at the top to 1 at the bottom. The hypsometric equation,

$$\frac{\partial\Phi}{\partial\pi} = -\alpha, \quad (2.3)$$

relates the geopotential Φ to the hydrostatic pressure π . Assuming the atmosphere is dry, the specific volume α is related to the temperature T and pressure p by the ideal gas law:

$$\alpha = \frac{RT}{p}, \quad (2.4)$$

where R is the gas constant. The ideal gas law involves the actual pressure p , also called nonhydrostatic pressure, rather than the hydrostatic pressure π . Using the ideal gas law in equation 2.3, we obtain:

$$\frac{\partial\Phi}{\partial\pi} = -\frac{RT}{p}. \quad (2.5)$$

Integrating equation 2.5 from the surface, where the geopotential is denoted by Φ_{sfc} , to an arbitrary level s , we have:

$$\Phi(s) = \Phi_{\text{sfc}} + \int_s^1 \frac{RT}{p} \frac{\partial\pi}{\partial s'} ds'. \quad (2.6)$$

Using equation 2.3, the third equation of motion can be written as:

$$\frac{dw}{dt} = g \left(\frac{\partial p}{\partial \pi} - 1 \right). \quad (2.7)$$

Defining the ratio of the vertical acceleration to gravity g as:

$$\epsilon \equiv \frac{1}{g} \frac{dw}{dt}, \quad (2.8)$$

equation 2.7 can be rewritten as:

$$\frac{\partial p}{\partial \pi} = \epsilon + 1, \quad (2.9)$$

which defines the relationship between hydrostatic and nonhydrostatic pressures.

In the hydrostatic s coordinate system, the time derivative of a fluid property q , following the motion of an air parcel, can be expressed as:

$$\frac{dq}{dt} = \left(\frac{\partial q}{\partial t} \right)_s + \mathbf{v} \cdot \nabla_s q + \left(\dot{s} \frac{\partial \pi}{\partial s} \right) \frac{\partial q}{\partial \pi}, \quad (2.10)$$

where \dot{s} represents the vertical velocity, and the subscripts indicate the variables that are held constant during differentiation.

The nonhydrostatic continuity equation is given by:

$$w = \frac{1}{g} \left[\left(\frac{\partial \Phi}{\partial t} \right)_s + \mathbf{v} \cdot \nabla_s \Phi + \left(\dot{s} \frac{\partial \pi}{\partial s} \right) \frac{\partial \Phi}{\partial \pi} \right] + W(\lambda, \phi, t), \quad (2.11)$$

where w represents the vertical velocity, and W is an integration constant that may depend on horizontal coordinates (λ , ϕ) and time (t). For simplicity, we assume $W = 0$.

The hydrostatic mass continuity equation is given by:

$$\left[\frac{\partial}{\partial t} \left(\frac{\partial \pi}{\partial s} \right) \right]_s + \nabla_s \cdot \left(\mathbf{v} \frac{\partial \pi}{\partial s} \right) + \frac{\partial}{\partial s} \left(\dot{s} \frac{\partial \pi}{\partial s} \right) = 0, \quad (2.12)$$

which follows from the nonhydrostatic continuity equation.

The list of variables dealt with by the model dynamics is as follows:

- μ , π : hydrostatic pressure [Pa]
- p : nonhydrostatic pressure [Pa]
- T : temperature [K]
- q : specific humidity [kg/kg]
- c : total water condensate [kg/kg]
- u , v : wind components (velocities) [m/s]

Vertical coordinate

A hybrid pressure-sigma coordinate model [86] is used in the NMMB. The hydrostatic pressure is computed from Equation 2.2 with the hybrid coordinate. The transition to the hydrostatic pressure vertical coordinate occurs around 300 hPa. Over elevated terrain, the hybrid coordinate increases the vertical resolution, and the equations remain continuous without the need for computational internal boundary conditions that are typically required with steep mountains.

The Lorenz staggering of the variables is used in the vertical grid [87]. The geopotential and the nonhydrostatic pressure are defined at the interfaces of the layers, while all three velocity components and temperature are carried at the midpoints of the model layers.

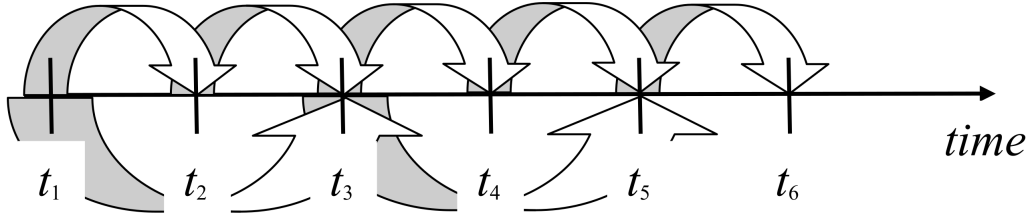


Figure 2.1: Time stepping process in the NMMB. Source: [71]

Temporal discretization

The NMMB employs the following types of time integration: Adams-Bashforth for horizontal advection, Crank Nicholson for vertical advection, and forward-backward scheme for adjustment terms [88] [89].

The Adams-Bashforth method is a linear multistep method. The next value, y^{n+1} , is calculated as a linear combination of the previous values y^n from earlier steps [90]. In NMMB, it is represented as:

$$\frac{y^{n+1} - y^n}{\Delta t} = 1.533f(y^n) - 0.533f(y^{n-1}). \quad (2.13)$$

The Crank-Nicholson scheme is represented as:

$$\frac{y^{n+1} - y^n}{\Delta t} = \frac{1}{2} [f(y^{n+1}) + f(y^n)], \quad (2.14)$$

Figure 2.1 demonstrates the time-stepping process in NMMB. The fundamental time step, $\Delta t = t_2 - t_1$, represents the interval used for the dynamical processes (indicated by the short, thin arrows). The time step for advection is typically twice as long, $2\Delta t$, as shown by the longer, wider arrows.

Spatial discretization

The horizontal grid corresponds to the Arakawa B-grid staggering. Figure 2.2 illustrates the staggering on the B-grid, where h represents mass points (such as temperature, pressure, height, or any mass or passive variable), \mathbf{v} represents the horizontal velocity vector, and Δx , Δy , and d are grid distances.

Consider the fluxes in the directions of the four coordinate axes connecting h points, as shown in Figure 2.3.

The following relationships define the velocity components in terms of the mass fluxes:

$$U\Delta y = \Delta\pi u\Delta y, \quad V\Delta x = \Delta\pi v\Delta x, \quad (2.15)$$

where Δx , Δy , and d are as defined in Figure 2.

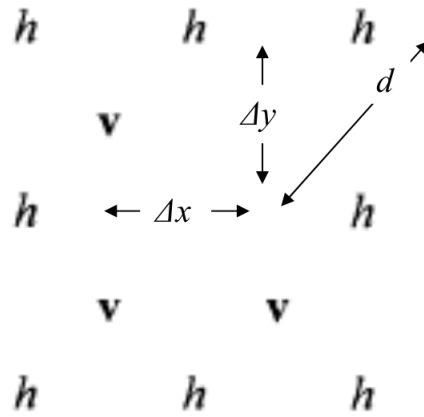


Figure 2.2: Representation of the B-grid staggering used in NMMB. Source: [71]

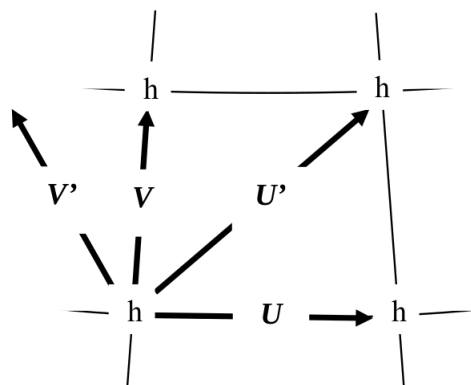


Figure 2.3: Representation of mass fluxes on the B grid. Source: [71]

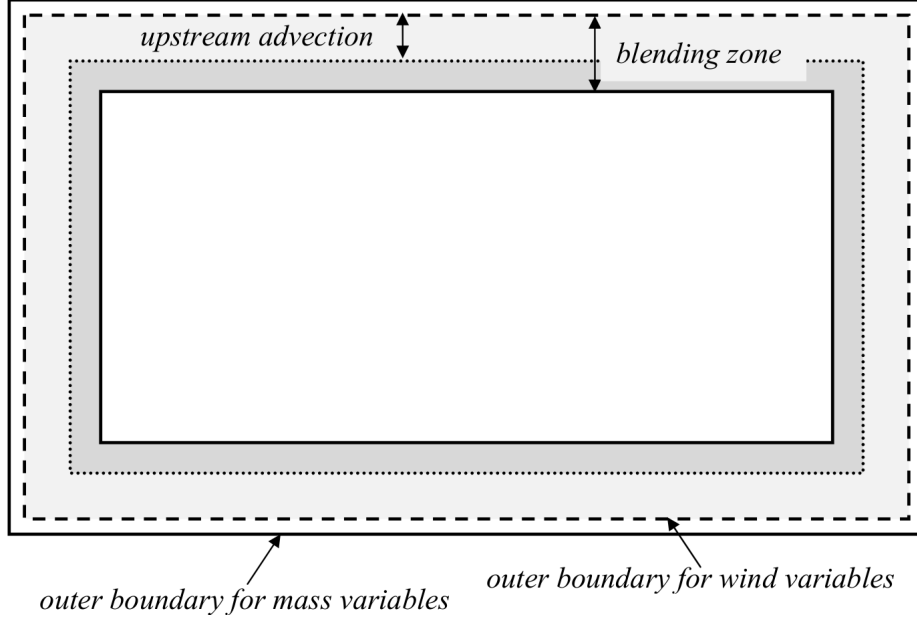


Figure 2.4: Lateral boundary conditions are used in NMMB. Source: [71]

Similarly, the mass fluxes in the diagonal directions are:

$$U'd = (u\Delta y + v\Delta x), \quad V'd = (-u\Delta y + v\Delta x). \quad (2.16)$$

The mass divergence term is then given by:

$$D = -\frac{1}{3} \left[\frac{2\Delta x(U\Delta y) + \Delta y(V\Delta x)}{\Delta x\Delta y} + \frac{\Delta x'(U'd) + \Delta y'(V'd)}{2\Delta x\Delta y} \right]. \quad (2.17)$$

The following approximation is used for horizontal discretization:

$$\nabla \cdot (\mathbf{v}\delta s\pi)_k \Delta s_k = \nabla \cdot (\mathbf{v}\Delta s\pi)_k = -D_k, \quad (2.18)$$

Boundary conditions

Figure 2.4 illustrates the lateral boundary conditions applied in the regional NMMB model. Velocity and mass variables are specified only on the outermost rows and columns, with the outer boundary passing through mass points. In the first three rows inside the domain, upstream differencing is used for advection. Additionally, a boundary blending zone is introduced. In this zone, the solution obtained by solving the model's equations gradually blends with the prescribed boundary conditions. The blending zone is five rows wide, with the weight of the prescribed boundary conditions decreasing linearly as the distance from the boundary increases [91].

The “across the pole” polar boundary conditions were implemented by introducing two ghost rows for the wind and mass variables. In this approach, the polar rows of points carry the mass variables. The mass points along the ghost lines are assigned values of mass variables from the opposite sides of the poles, aligned with the same meridians. Similarly, the wind points along the ghost lines carry the wind components from the other sides of the poles, but with reversed signs, because the coordinate axes change direction as the poles are crossed along a meridian.

The vertical boundary conditions are defined as follows:

$$\dot{s} = 0 \quad \text{and} \quad p = \pi \quad \text{at} \quad s = 0, \quad (2.19)$$

$$\dot{s} = 0 \quad \text{and} \quad \frac{\partial(p - p^*)}{\partial s} = 0 \quad \text{at} \quad s = 1, \quad (2.20)$$

where the pressure p^* is defined to satisfy the equation:

$$\frac{\partial p^*}{\partial s} = (1 + \epsilon_1) \left(\frac{\partial \pi}{\partial s'} \right)^{n+1}, \quad (2.21)$$

subject to the boundary condition:

$$p^* = \pi_T^* \quad \text{at} \quad s = 0. \quad (2.22)$$

2.1.2 Computational implementation

The computational implementation of MONARCH generally follows the next steps:

1. Read initial atmospheric data and MONARCH configuration, such as the number of MPI processes.
2. Solve transport by advection.
3. Solve turbulent diffusion and exchange wind components.
4. Solve microphysics.
5. Solve chemistry (using EBI or CAMP solver).
6. Exchange poles, East-West boundary, and chemistry.
7. Update time-step and output data. Repeat from step 2 until the last time step is reached.

The domain decomposition is applied horizontally across the atmospheric domain, segmenting it into regions, with each area being computed by an individual MPI process. The Earth System Modelling Framework <https://earthsystemmodeling.org/> [92] manages the MPI workflow. The output data is saved asynchronously to compute the next time step, while the remaining processes are dedicated to computation tasks. MONARCH supports saving output using 4 or 8 MPI processes, while any number of processes can be used for computation tasks.

The atmospheric input and output data of MONARCH is stored following the NetCDF (Network Common Data Form) format <https://www.unidata.ucar.edu/software/netcdf/> [93]. NetCDF was created by UCAR (University Corporation for Atmospheric Research) to store efficiently geolocalized data, mainly used in the Earth Sciences community, for applications using structured and unstructured data from models, satellites, or in-situ observation. Data in a netCDF file are stored in n-dimensional matrices that can be accessed through any subset without loading all the data. Its use to store the vast majority of the data generated by the community (from meteorology to decadal climate simulations) has led to the creation of an extensive ecosystem of analysis tools in numerous languages (Python, R, Matlab, Fortran, C++,...) and a joint agreement on standards to define the content of the files: variable names (short and standard), units, description, etc that are contained in the netCDF files and allow their immediate comparisons <https://cfconventions.org/https://github.com/PCMDI/cmip6-cmor-tables>.

2.2 The chemistry solver CAMP

We use the Chemistry Across Multiple Phases (CAMP) framework [26] as our test bed for solving chemical mechanisms. CAMP is a novel framework permitting runtime configuration of chemical mechanisms for mixed gas- and aerosol-phase chemical systems. In this thesis, we focus exclusively on gas-phase chemistry systems to maintain simplicity and effectively translate this time-intensive component of the chemistry process.

CAMP is designed to use external ODE solvers to solve the chemistry time-dependent equation ($y' = f(t, y)$). The default version of CAMP is coupled to the external CVODE solver of the SUNDIALS package using backward differentiation formulas (BDF) and Newton iteration <https://computing.llnl.gov/projects/sundials/cvode> [47] [94]. This algorithm is suitable for mathematically stiff systems. The variable-order, variable time-step CVODE solver with time-step error control provides accurate solutions. Thus, it was chosen as the initial solver option for CAMP [26]. The BDF algorithm requires the solution of a linear system at each integration step. CAMP is configured by default to use the KLU Sparse solver [95]. The sparse structure avoids storing zero values in the Jacobian matrix, typical for chemical systems [96]. Figure 2.5 summarizes the models used in this thesis and the primary process of interest, namely the gas-phase chemical mechanism solver.

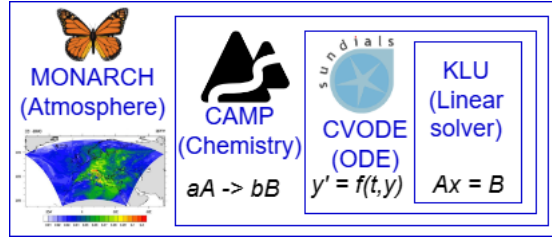


Figure 2.5: Modules encapsulated from the MONARCH atmospheric model to the KLU linear solver.

2.2.1 Mathematical considerations

The mass balance partial differential Equation 2.1 is typically discretized in atmospheric models using an operator split approach [97]. In this method, individual processes in the equation are solved sequentially over each time interval $[T, T + \Delta T_{\text{splitting}}]$, where $\Delta T_{\text{splitting}}$ represents the model time split step size. This step size is distinct from the integration step size, denoted as h , used by a chemical integrator. The integrator may take multiple steps of size h within each $\Delta T_{\text{splitting}}$, resulting in simplified problems for advection, diffusion, chemistry, and other processes. The solution of the chemical kinetic process, in particular, reduces to a system of ordinary differential equations (ODEs) within each cell of the model:

$$y' = f(t, y), \quad T \leq t \leq T + \Delta T_{\text{splitting}}, \quad y \in \mathbb{R}^d \quad (2.23)$$

, where y' is solved following the BDF implementation from the CVODE library, which reads as:

$$\sum_{i=0}^q \alpha_{n,i} y_{n-i} + h_n \beta_{n,i} y'_{n-i} = 0. \quad (2.24)$$

Here, the y_n are computed approximations to $y(t_n)$, $h_n = t_n - t_{n-1}$ is the step size, and the order q varies between 1 and 5. The coefficients α and β are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$ [98] [99].

A nonlinear system is solved (approximately) at each integration step, formulated as the root-finding problem:

$$F(y_n) \equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0, \quad (2.25)$$

where

$$a_n \equiv \sum_{i>0} (\alpha_{n,i} y_{n-i} + h_n \beta_{n,i} y'_{n-i}). \quad (2.26)$$

The root-finding problem is solved with a Newton iteration, which requires the solution of linear systems

$$M[y_n^{(m+1)} - y_n^{(m)}] = -F(y_n^{(m)}) \quad (2.27)$$

where

$$M \approx I - \gamma J, \quad J = \frac{\partial f}{\partial y}, \quad \text{and} \quad \gamma = h_n \beta_{n,0}. \quad (2.28)$$

in which I is the identity matrix and J is the Jacobian matrix. The Jacobian is held constant during the Newton iteration, resulting in a Modified Newton method.

CVODE employs a weighted root-mean-square norm in controlling errors at various levels, denoted $\|\cdot\|_{\text{WRMS}}$, for all error-like quantities. The multiplicative weights used in this norm are based on the current solution and the input of the user's relative and absolute tolerances.

$$W_i = \frac{1}{\text{rtol} \cdot |y_i| + \text{atol}_i} \quad (2.29)$$

Since $1/W_i$ represents tolerance for the component y_i , a vector with a norm of one is considered "small." We will generally omit the subscript WRMS on norms for brevity.

CVODE estimates the local error at each step and ensures it meets tolerance conditions; if the error test fails, the step size is reduced, and the step is recomputed. In addition to adjusting the step size to satisfy the local error test, CVODE periodically adjusts the integration order to maximize the step size. Integration begins at order one and dynamically changes after that. The core idea is to select an order q such that a polynomial of order q best fits the discrete data involved in the multistep method https://sundials.readthedocs.io/en/latest/cvode/Mathematics_link.html.

2.2.2 Computational implementation

CAMP is designed to decouple the specification of chemical mechanisms from the implementation of the solving procedure, facilitating the adaptation in host models like MONARCH. This approach contrasts with traditional implementations in several ways, as Figure 2.6 illustrates.

In the conventional approach (Figure 2.6a), the code for individual model components is usually adapted to facilitate interaction with the infrastructure of the atmospheric model. Often, the configuration for model components is hard-coded, and solvers are tightly linked with the representation of the chemical system. This rigidity complicates new configurations, such as adding new chemical species, tuning the solver tolerances, or preconditioning.

On the other hand, CAMP compiles separately from other model components and provides an interface based on configuration files to set a specific chemical system, update rates for processes such as emissions or photolysis (which are usually computed in separate modules), and solve the multi-phase chemical system at each time step (Figure 2.6b). This design means optimizing the CAMP solver, which is equivalent to optimizing multiple individual solvers in a traditional setup and streamlining development efforts. Additionally, CAMP is built to interface with various external solver packages, further separating the chemical system specification from the solver implementation.

CAMP has been designed with extensibility, accommodating various solver strategies, including GPU-based solvers, as developed in this thesis. Additionally, CAMP supports the scalability of chemical complexity by using a standardized JSON format for specifying multi-phase chemical systems at runtime. Figure 2.7 shows two examples of JSON configuration objects used by CAMP. The first example showcases a relatively simple Arrhenius-type reaction:

$$k = A \exp\left(-\frac{E_a}{k_b T}\right) \left(\frac{T}{D}\right)^B (1.0 + EP), \quad (2.30)$$

where A is a pre-exponential factor ($(\text{cm}^{-3})^{-(n-1)}\text{s}^{-1}$), k is referred to as the rate constant, E_a is the activation energy (J), k_b is the Boltzmann constant ($k_b = 1.38 \times 10^{-23} \text{ J/K}$), D (K), B (unitless) and E (Pa^{-1}) are reaction parameters, T is the temperature (K), and P is the pressure (Pa). Finlayson-Pitts and Pitts describe the first two terms [100]. The final term accommodates rate constants used by the solver scheme.

The second example illustrates a more complex configuration dataset, specifically the UNIFAC activity model [101]. This JSON configuration defines the components, their parameters, and the interaction parameters between different substances. This approach eliminates the need to hard-code these complex data sets into the model, which typically requires re-compilation and extensive modifications when adding new functional groups or interactions. Using JSON, CAMP allows runtime access to this data, making it easier for users to modify sub-models or parameterizations by simply updating the configuration files.

CAMP employs an object-oriented design to ensure scalability and extensibility. In contrast, atmospheric models often use a more straightforward approach with hard-coded parameters, which can limit flexibility. For example, the current default MONARCH model does not utilize an object-oriented framework, reflecting a more traditional, less adaptable design.

CAMP can run as an independent library without relying on an atmospheric model like MONARCH (i.e., in a box model configuration). It includes various validation tests with different chemistry configurations. This thesis employs two chemical mechanisms to validate our developments: (i) a simple chemical reaction where

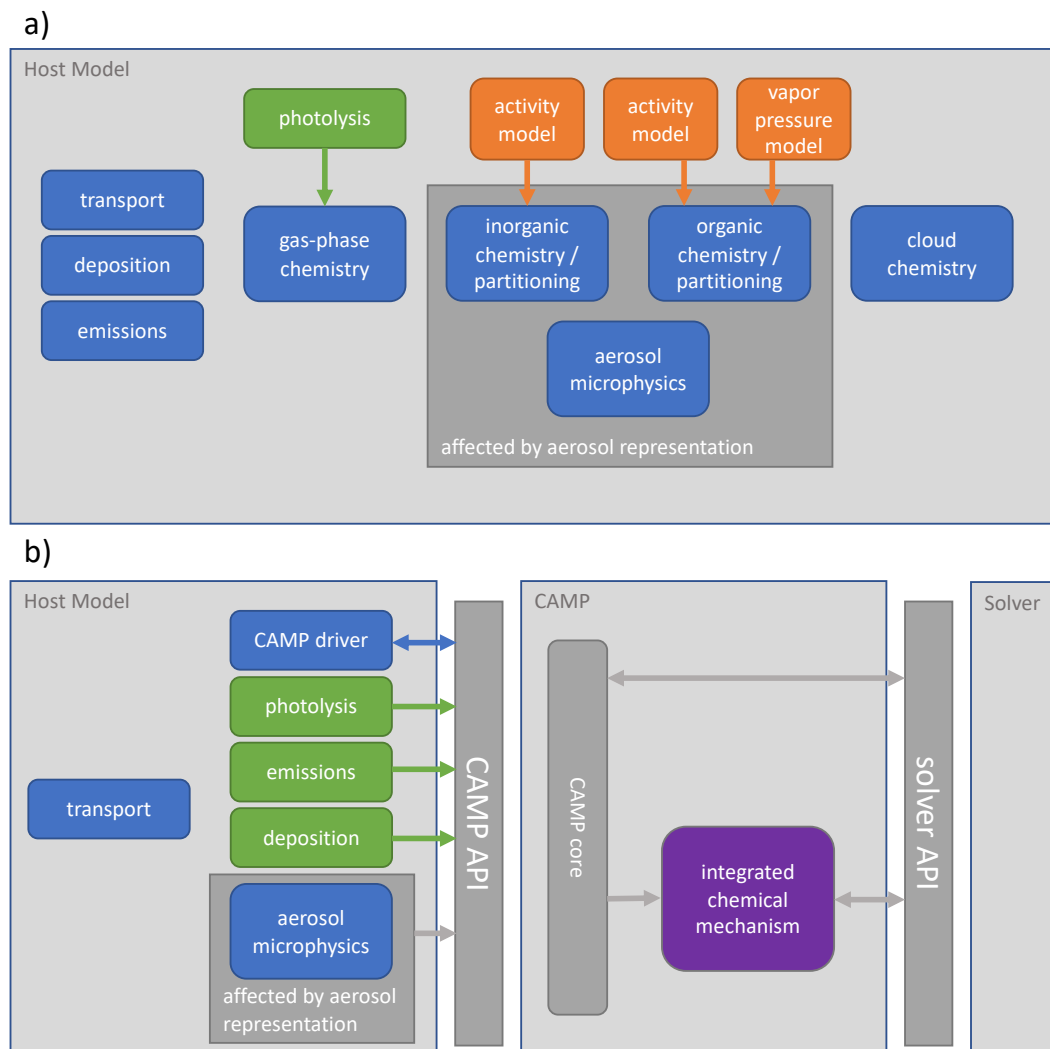


Figure 2.6: Interactions of chemistry and related modules in (a) a typical atmospheric model and (b) an atmospheric model using CAMP. Model components calculate rates or rate constants for physicochemical processes (green), calculate physical parameters (orange), or directly update the host model state (blue). Some modules that typically directly update the model state—deposition and emissions in (a)—now provide rates for these processes to CAMP (b). Parameter calculations—activity and vapor pressure models in (a)—are now integrated into the combined chemical mechanism (purple). Arrows indicate the primary flow of information among components. Source: [26]

```

{
  "reactants" : {
    "O" : {},
    "NO2" : {}
  },
  "products" : {
    "NO" : {}
  },
  "type" : "ARRHENIUS",
  "A" : 5.6E-12,
  "B" : 0.0E+00,
  "C" : 180.0
}

```

(a) Arrhenius reaction

```

{
  "name" : "n-butanol",
  "type" : "CHEM_SPEC",
  "UNIFAC groups" : {
    "OH" : 1,
    "CH2(-OH)" : 1,
    "CH2(hydrophobic tail)" : 2,
    "CH3(hydrophobic tail)" : 1
  }
  ...
},
{
  "name" : "n-butanol/water activity",
  "type" : "SUB_MODEL_UNIFAC",
  "phases" : [ "n-butanol/water mixture" ],
  "functional groups" : {
    "CH2(-OH)" : {
      "main group" : "CHn(-OH)",
      "volume param" : 0.6744,
      "surface param" : 0.540
    },
    "CH2(hydrophobic tail)" : {
      "main group" : "CHn(hydrophobic tail)",
      "volume param" : 0.6744,
      "surface param" : 0.540
    },
    ...
  },
  "main groups" : {
    "CHn(-OH)" : {
      "interactions with" : {
        "OH" : 986.5,
        "H2O" : 2314
      }
    },
    "OH" : {
      "interactions with" : {
        "CHn(-OH)" : 156.4,
        "CHn(hydrophobic tail)" : 156.4,
        "H2O" : 276.4
      }
    },
    ...
  }
}

```

(b) UNIFAC activity model

Figure 2.7: Two examples of CAMP configuration data in JSON format: an Arrhenius reaction (a), and a portion of a UNIFAC activity model configuration (b). Ellipses (...) indicate portions of the data omitted for brevity. Source: [26]

species A generates B and C , and (ii) the Carbon Bond 2005 (CB05) mechanism used in MONARCH [72]. The first configuration is utilized in Chapter 3, while the second is used in the remaining chapters. The latter configuration is a Box model that solves multiple cells to emulate a MONARCH experiment involving thousands of atmospheric cells.

The computational implementation of MONARCH-CAMP is detailed in Figure 2.8. The process begins with MONARCH reading input data and initializing the model based on atmospheric initial conditions and configuration. Following initialization, MONARCH predicts future atmospheric variables by solving meteorological processes, including advection, diffusion, and physics. The chemistry-solving process starts by updating the time-step size ($\Delta T_{\text{splitting}}$). The function $f(y)$ then solves the chemical equations for the current time step. The Jacobian matrix of the system is computed through the reaction rates ($J = \frac{\partial f}{\partial y}$), similarly to $f(y)$, and utilized in the linear solver as part of the Newton integration process. If the solution converges—meaning the accuracy error is within acceptable limits—the solver returns the updated chemical concentrations to MONARCH, which continues with the atmospheric processes. After completing the chemistry calculations, MONARCH performs post-chemistry operations and determines whether to proceed to the next integration time step or to conclude the simulation.

In computational terms, CAMP is a complex model with more than 40,000 code lines without considering validation tests.

CAMP has been adapted to follow the atmospheric model’s MPI (Message Passing Interface) approach. The atmospheric domain, which represents the numerous chemical concentrations within the atmosphere, is segmented into several hierarchical levels: Regions or sub-domains, Cells, and Chemical Concentrations.

- **Regions:** Each MPI process is assigned to compute a specific sub-domain region.
- **Cells:** Within each region, multiple cells exist, each representing the concentrations of a set of chemical species under study. The chemical mechanism describes how the different chemical species react in time. The cells can be interpreted as the smallest units in the atmosphere, similar to a point on a map.
- **Chemical Concentrations:** Within each cell, the chemical concentrations are treated as part of an ODE system derived from the chemical mechanism and solved accordingly.

Figure 2.9 illustrates this segmentation. Each ODE system in CAMP involves multiple chemical concentrations, which are solved using algebraic operations. For instance, vector multiplication is performed on concentration arrays, which are then used to update the system’s state.

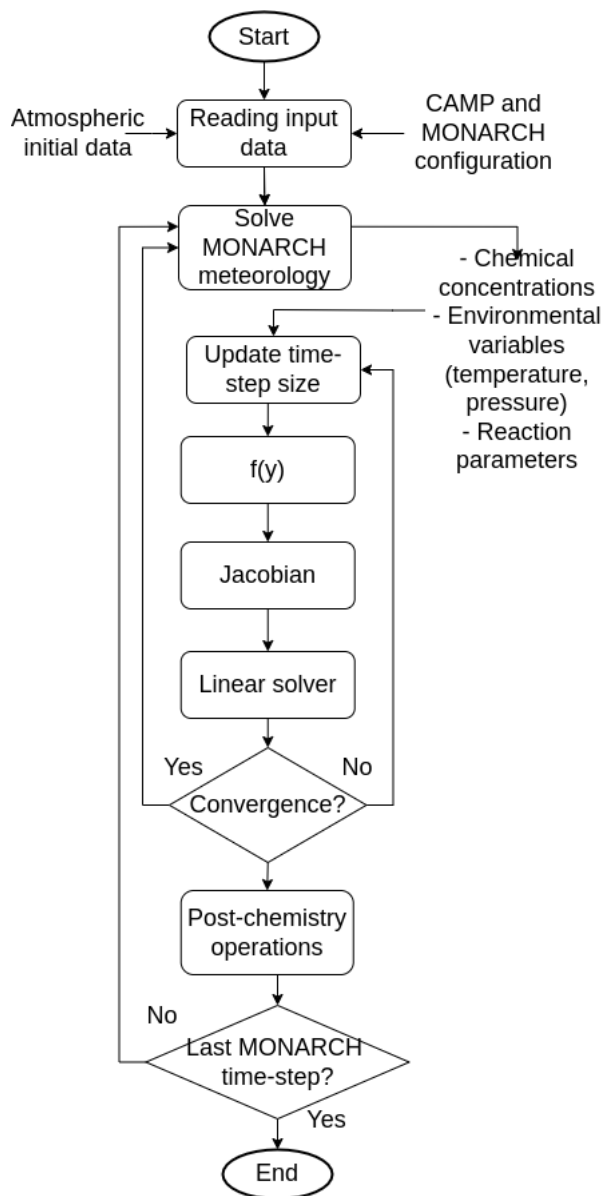


Figure 2.8: Workflow of CAMP coupled in MONARCH.

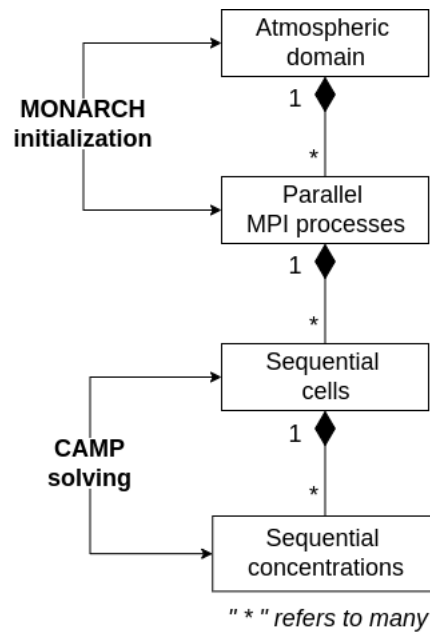


Figure 2.9: Segmentation of the atmospheric domain into chemical ODEs. During MONARCH initialization, the atmospheric domain is segmented into regions, each assigned to an MPI process for computation. CAMP then sequentially processes the cells containing chemistry data within these regions. Each cell represents a set of chemical species concentrations (state) that CAMP solves as a system of Ordinary Differential Equations (ODEs) to determine the updated concentration state. The solving process involves sequentially calculating these concentrations through algebraic operations, such as vector multiplications.

Solving the numerous cells within a region is computed sequentially due to the limited number of CPU processors compared to the number of chemical concentrations typically present in a domain. For example, while a node may have tens of CPU processes available, a domain may contain millions of chemical concentrations.

In contrast, GPUs are not constrained by such limitations. They can execute millions of threads simultaneously, making them highly suitable for parallel computation. Consequently, this thesis represents a transition from sequential models to a highly parallel paradigm, leveraging GPU capabilities to handle the extensive computational demands of atmospheric chemistry more efficiently.

2.3 Profiling tools

This work employed two profiling tools to assess GPU performance: Nvidia Visual Profiler (NVVP) and Nvidia Nsight Compute (NCU). NCU is the updated tool and has recently replaced NVVP, which has been deprecated. NVVP was used in the earlier stages of this thesis (as detailed in Chapter 4), while NCU was utilized for the latter parts.

Both tools provide performance and resource utilization metrics, albeit with different terminologies. Specifically, we focus on metrics related to Memory and Arithmetic resources. In NVVP and NCU, Arithmetic resources are called Computation Intensity and SM, respectively.

Beyond these, the tools offer different metrics, which we describe below.

2.3.1 NVVP metrics

We utilize the percentage of time consumed by different types of instructions, such as memory dependence or synchronization, to identify areas for potential improvement. For example, if a lot of time is spent on memory operations, this suggests focusing on optimizations like vectorization to improve performance. This metric is only available in NVVP, as we have not found an equivalent for NCU.

To evaluate the efficiency of our implementation and determine how closely it approaches the ideal case, we consider the following metrics:

- *Global load efficiency* is the ratio of requested global memory load throughput to required global memory load throughput expressed as a percentage. This metric is used to measure memory efficiency.
- *Warp execution efficiency* is the ratio of the average active threads per warp to the maximum number of threads per warp expressed as a percentage. This metric is used to measure computational efficiency.

- *Occupancy* is the ratio of the average active warps per active cycle to the maximum number of warps supported (a warp in CUDA is a group of 32 threads). This metric is used to measure computational efficiency by accounting for synchronization overhead. When combined with "Warp execution efficiency," it can provide an approximate quantification of the time spent on synchronization activities.

2.3.2 NCU metrics

To evaluate the efficiency of our implementation and determine its proximity to the ideal performance, we employ the Roofline model [102] [103] as provided by Nvidia Nsight Compute (NCU). Figure 2.10 illustrates a typical Roofline plot. The X-axis represents the Arithmetic Intensity, the ratio of floating-point operations (FLOPs) to the number of bytes processed. This metric gives insight into the memory efficiency of our application by indicating how many operations are performed per byte of data moved.

The Y-axis represents Performance, quantified as the number of FLOPs per second, reflecting the arithmetic throughput of our application. The graph's blue lines indicate the performance bound, with the left and right squares representing the optimal values for single and double-precision floating-point operations, respectively. These points indicate how close our implementation is to the theoretical maximum performance.

In the Roofline model, the left and right points correspond to single-precision and double-precision floating-point operations, where single precision uses 4 bytes and double precision uses 8 bytes to represent floating-point numbers. Since CAMP primarily handles critical variables, such as chemical concentrations, using double precision, this thesis emphasizes the evaluation and optimization of double-precision operations.

The Roofline model also highlights application bottlenecks. If the application's performance point is located to the left of the optimal square, it indicates that the application is memory-bound, meaning the performance is limited by memory bandwidth. Conversely, suppose the point is closer to the right and nearer to the blue performance bound line. In that case, the application is compute-bound, meaning the performance is constrained by the computational power of the processor rather than memory throughput. This distinction is crucial for guiding optimization efforts, as it helps identify whether improving memory access patterns or enhancing computational efficiency should be prioritized.

Another essential metric is the cache hit rate, which indicates the percentage of memory requests successfully retrieved from the cache rather than slower memory. A high cache hit rate suggests efficient memory distribution and access patterns,

2.3. PROFILING TOOLS

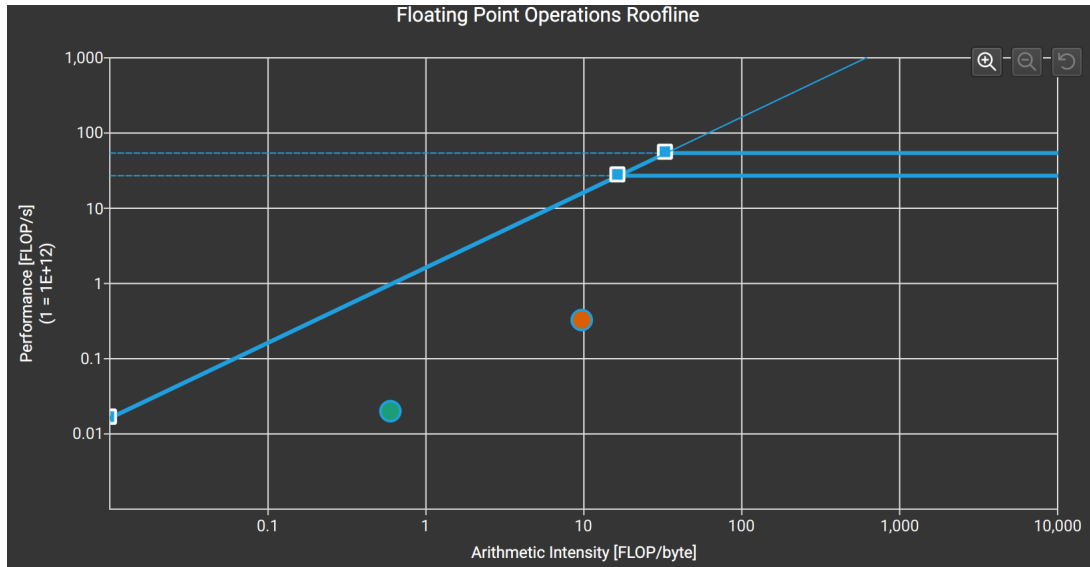


Figure 2.10: Example of a Roofline model from NCU.

meaning that most required data is readily available in the cache. This metric is beneficial for identifying inefficient implementations.

For example, if a matrix is accessed in reverse order or non-contiguous, the program might access memory locations that are far apart, leading to poor cache utilization. In such scenarios, the cache hit rate would be notably low, signaling the need for optimizations to improve memory access patterns. Increasing the cache hit rate can reduce memory latency and improve overall performance.

Chapter 3

Efficient data arrangement for GPU adaptation of ODE systems in atmospheric chemistry solvers as multiple cells

3.1 Introduction

Atmospheric chemistry models are parallelized following a domain decomposition approach. This method simplifies parallelization without requiring extensive development associated with more complex alternatives. Each parallel process solves thousands of cells, as the number of cells within the domain significantly exceeds the number of parallel processes. As these models have been historically developed for CPU-based architectures, MPI and OpenMP parallelization paradigms are commonly used. However, computation can be further parallelized using other parallel architectures, such as GPUs, in combination with MPI parallelization.

This chapter presents a new implementation to simultaneously solve the chemical mechanism integration of multiple cells in a single-thread execution. This adaptation facilitates the integration of GPU functions capable of simultaneously solving the entire domain of interest. Otherwise, utilizing the GPU solution would entail solving a single cell, resulting in a workload that is too insignificant to justify launching a GPU kernel. This approach requires less development effort to test GPU implementations than translating the entire solver to GPU. We refer to this implementation as Multi-cells.

Also, we present a GPU implementation that divides the workload between chemistry equations instead of the classical domain division used in CPU execution. In the classical implementation, the chemistry of each grid cell within a sub-domain

is solved sequentially through a loop. On the other hand, the GPU approach allows for greater parallelization as each grid cell has multiple chemical reactions to solve. This approach is tested on the most time-consuming function of the chemical module, the partial derivatives of y to time $f(y)$, and is responsible for solving the chemistry reactions without advancing the time step.

This chapter is organized as follows. In section 3.2, we briefly describe CAMP, plus present the most time-consuming function of CAMP. In section 3.3, we present the GPU implementation of this function, an optimization to reduce GPU accesses, and the Multi-cells implementation for the whole CAMP module. In section 3.4, we define the software configuration. Section 3.5 shows the result of the implementations presented. Finally, section 3.6 concludes the work and overviews possible future work.

3.2 Background

This section describes the state-of-the-art and computational description as the starting point before our developments.

3.2.1 State of the art

The state-of-the-art related to GPU chemistry modules is explained previously in Section 1.2. This section details current approaches to implementing GPU computing in chemistry modules and the available GPU tools. For example, it explains why CUDA is used over other parallel languages such as OpenMP or OpenACC.

Section 1.1 defines the motivation behind using CAMP instead of other chemistry modules. Section 2.2 provides more details about the benefits of CAMP and includes a computational description of CAMP in Section 2.2.2. This description serves as the starting point for the implementation presented in this chapter.

3.2.2 Computational description

In this section, we extend the description of CAMP provided in Section 2.2.2 to include more details about specific elements related to our implementation. Specifically, this section describes data structures and workflow associated with the most time-consuming functions.

The chemical reactions in CAMP can include integer parameters (e.g., array indices, stoichiometric coefficients, ionic charge, etc.) and floating-point parameters (e.g., conversion factors, rate parameters, etc.). The set of chemical species concentrations

3.2. BACKGROUND

(y) is named the *state* array, and the set of partial Derivatives of these species to time ($f(y)$) is called the *deriv* array.

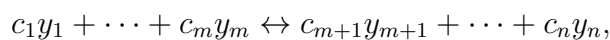
CAMP predicts future concentrations after the data is read using the external ODE solver CVODE <https://computing.llnl.gov/projects/sundials/cvode> [47] [94]. CVODE solves the time-dependent equation ($y' = f(t, y)$) using the CAMP-provided set of Derivatives ($f(y)$) stored in the *deriv* array. CVODE also uses a Jacobian matrix provided by CAMP. From the matrix structure options that CVODE offers, we choose the SPARSE structure [104] to store the Jacobian, as this is a good choice for Jacobian structures with few non-zero elements, as is the case for many chemical mechanisms.

Derivative and Jacobian functions have similar input and output, following the same structure. The only difference is the structure where we store the data (an array for the Derivative and a sparse matrix for the Jacobian) and some extra linear operations. So, we only need to analyze one of them since we can extrapolate the optimization ideas and techniques.

Inside MONARCH, CAMP is required to solve chemistry multiple times—one time for each MONARCH time-step and cell. A cell represents a volume of the atmosphere; the collection of all the cells composes a 3-dimensional domain representing the region under study. The number of cells depends on the user-selected MONARCH configuration. MONARCH typically computes many cells in a large geographical area with high precision. Each cell has its own *state*, which, in terms of chemical processes, is independent of other cell *state* values during the chemistry time integration. In Figure 3.1, we summarize the flow described in a diagram.

The CAMP functions executed during the solving take a considerable execution time. We configured a box model experiment in CAMP (without MONARCH) to measure this impact with a CB05 chemical mechanism. The experiment results show that CVODE occupies 70% of the total execution time, and Derivative and Jacobian are around 30%. Despite being small functions compared to the whole ODE solver, the Derivative and Jacobian have a relevant impact on general performance, with the Derivative generally more time-expensive than the Jacobian. So, in a similar way to selecting the chemistry component of MONARCH, we choose to work around the Derivative to analyze our GPU implementation and search for a relevant reduction of the model execution time.

In general, chemistry models try to predict future concentrations of a set of chemical species by solving ordinary differential equations that represent the reactions that compose a chemical mechanism. Reactions take the general form:



where species y_i participates in the reaction with stoichiometric coefficient c_i . The rate of change for each participating species y_i to reaction j is given by

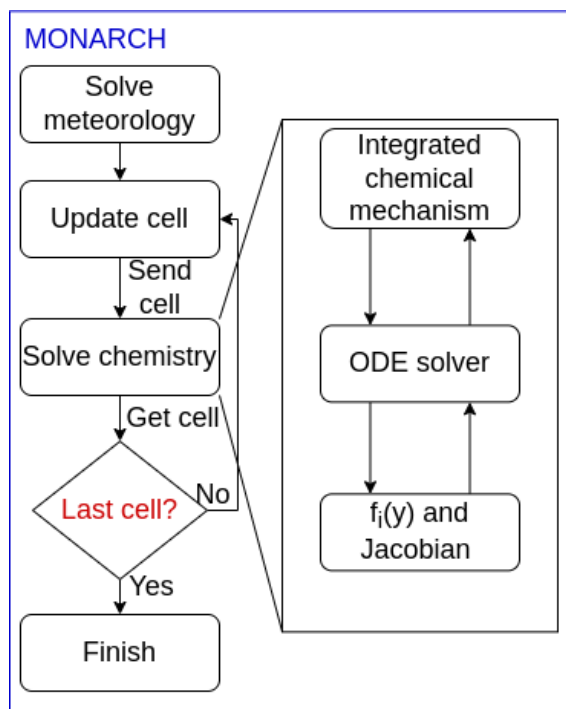


Figure 3.1: MONARCH overall flow diagram with CAMP as chemistry solver.

$$\left(\frac{dy_i}{dt}\right)_j = \begin{cases} -c_i r_j(\mathbf{y}, T, P, \dots) & \text{for } i \leq m \\ c_i r_j(\mathbf{y}, T, P, \dots) & \text{for } m < i \leq n \end{cases},$$

where the rate r_j of reaction j is an often complex function of the entire model state (including species concentrations \mathbf{y} , environmental conditions, such as temperature, T , and pressure, P , physical aerosol properties, such as surface area density and number concentration, etc.). The overall rate of change for each species y_i at any given time is, thus,

$$f_i \equiv \frac{dy_i}{dt} = \sum_j \left(\frac{dy_i}{dt}\right)_j,$$

where \mathbf{f} is referred to as the derivative of the system throughout this chapter.

Then, in the Derivative function, we multiply the rate constants saved on the reaction parameters array with the corresponding concentrations on the *state* array, filling the following concentration array (*deriv*). This operation is done for each reaction, adding all the results obtained from the reactions in the corresponding place of the *deriv* array. So, we can say that each reaction contributes to the *state* concentrations, increasing or decreasing the value.

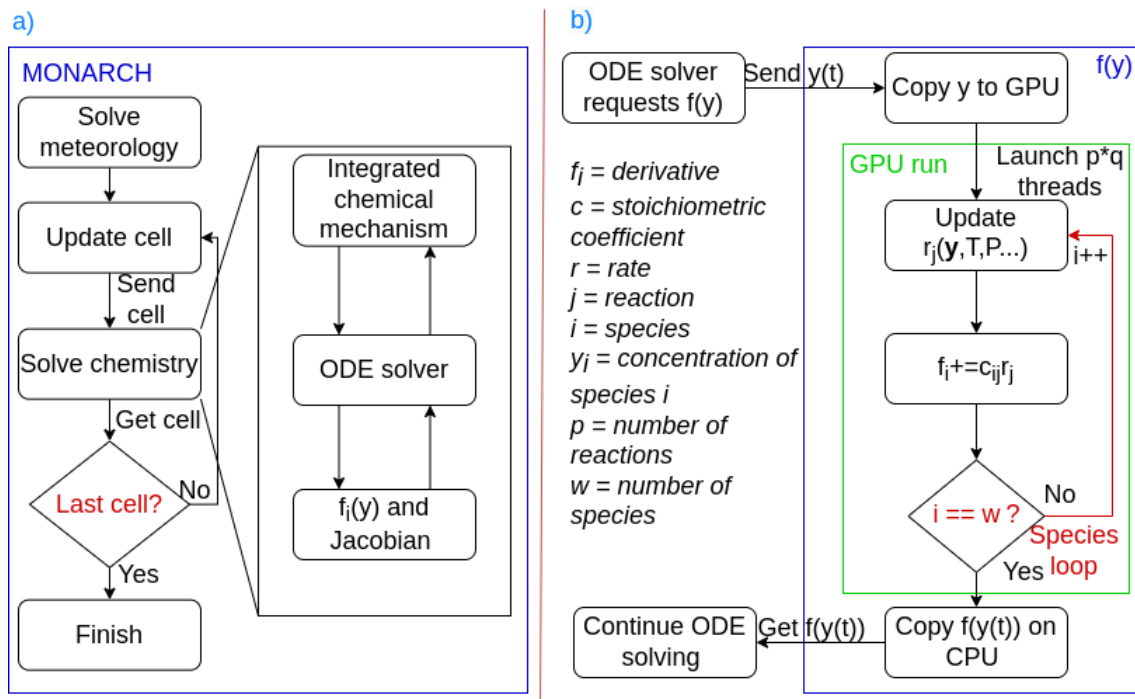


Figure 3.2: Figure 3.2a: Comparison of original and Multi-cells overall workflows from the MONARCH point of view. Figure 3.2b: Derivative workflow diagram for GPU execution.

3.3 Implementations

The Multi-cells implementation groups the input data from each cell into a single data structure to be computed. The MONARCH workflow described in figure 3.1 is updated to figure 3.2a. The cell loop disappears inside the solving internal functions, avoiding the process of updating the input data from cells and re-initializing the ODE solver. As an example, the Derivative equation is updated as follows:

$$f_i \equiv \frac{dy_{ik}}{dt} = \sum_j \left(\frac{dy_{ik}}{dt} \right)_j$$

Where y_{ik} refers to the species y_i from cell k .

Our GPU strategy is the parallelization of each reaction data packet. Figure 3.2b shows the resultant GPU-based Derivative flow diagram.

We compute the sum of contributions to f using the CUDA operation *atomicAdd*. This function avoids a possible thread overlapping when updating the same variable. Reactions between common species can produce this interference.

Reaction data is allocated on global memory at the initialization of the program.

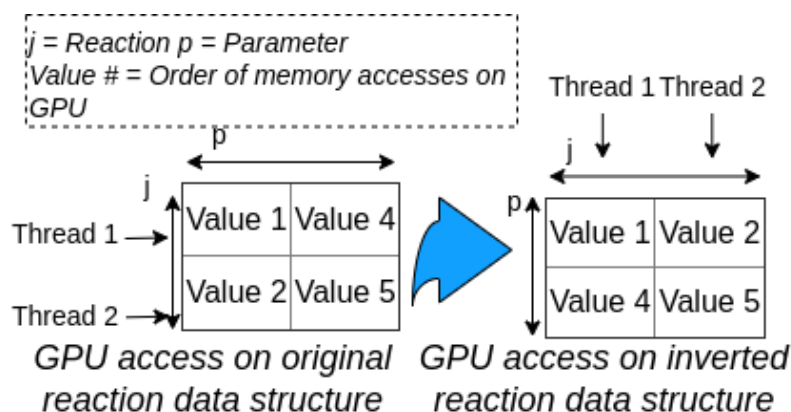


Figure 3.3: Data structure inversion for GPU Derivative. “Value” numbers represent the GPU memory arrangement and access order, “ j ” is the number of reactions, and “ p ” is the number of parameters.

To send and receive the rest of the data (state array) from the GPU, we first check the size of this array. If it contains few data variables, *state* is passed as a function parameter, taking advantage of the constant memory. Otherwise, the data is copied directly into global memory.

The number of GPU threads initialized is equal to the number of reactions. Another relevant GPU parameter, the number of blocks per thread, is configured to the maximum available for the GPU used (1024 threads/block). Lower configurations of threads/blocks don’t show performance improvement in our tests. Due to the possibility of using a GPU with less capacity in the future, we add a run-time checking of GPU hardware specifications to ensure the correct execution of the program regardless of the GPU used (for example, avoid demanding more threads than the GPU limit).

In the still CPU-based implementation, all the reaction data packets are initially stored consecutively in memory. Then, the parallelization by reactions results in each thread accessing no-consecutive values of the reaction data structure. We reordered this structure to follow a sequential reading of the data in the GPU. The first reaction parameters accessed are stored consecutively in the reaction data structure, and so on. Figure 3.3 illustrates the changes in the data packet structure, simulating the structure as a matrix where initially, the rows are the data packets and columns are the parameter values.

3.4 Test environment

All the tests were performed on the CTE-POWER cluster provided by the Barcelona Supercomputing Center [105]. The detailed hardware specifications of each node are described below.

- Operating system: Red Hat Enterprise Linux Server 7.5 (Maipo).
- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and four threads/core, total 40 physical cores per node and 160 virtual threads using hyper-threading)
- 512GB of main memory distributed in 16 dimms × 32GB @ 2666MHz
- 2 x SSD 1.9TB local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.
- Single Port Mellanox EDR
- GPFS via one fiber link 10 GBit
- Compilers: GCC version 7.3.0 and NVCC version 10.1.105 for CPU and GPU code, respectively.

We work around a basic chemical mechanism of 3 species, where species A generates B and C through 2 Arrhenius reactions. A is initialized at 1.0, while B and C are set to zero. Each cell has a small offset of 0.1 on the initial concentrations to generate different results. For example, at the first concentration value, we sum a 0.1 offset value, at the second 0.2, and so on till Multi-cells species. The rest of the variables, like temperature, pressure, or reaction data parameters, are initialized to the same values for all the cells.

3.5 Results and discussion

In figure 3.4a, we can see how Multi-cells speedups CAMP a factor of $8\times$ for multiple numbers of cells. Most of this speed-up is produced by the reduction of solving iterations. In the One-cell case, the number of iterations scales linearly with the number of cells factor, while in the Multi-cells case, the number of iterations is independent of the number of cells computed, keeping almost the same number of iterations for the number of cells. For example, One-cell takes around $6e^6$ iterations to solve 10,000 cells (an average of 600 iterations per cell). In contrast, Multi-cells take around 700 to solve all cells independently of the number of cells.

The GPU implementation speeds up the Derivative function for many cells. In figure 3.4b, we can see how, for 10,000 cells, the GPU version achieves $1.2\times$ speedup. On the other hand, a lower number of cells slows down the function, shown as a speedup below $1\times$ for less than 10,000 cells. We can also see that optimizing memory access improves the overall speedup by $1.3\times$ approximately for all numbers of cells.

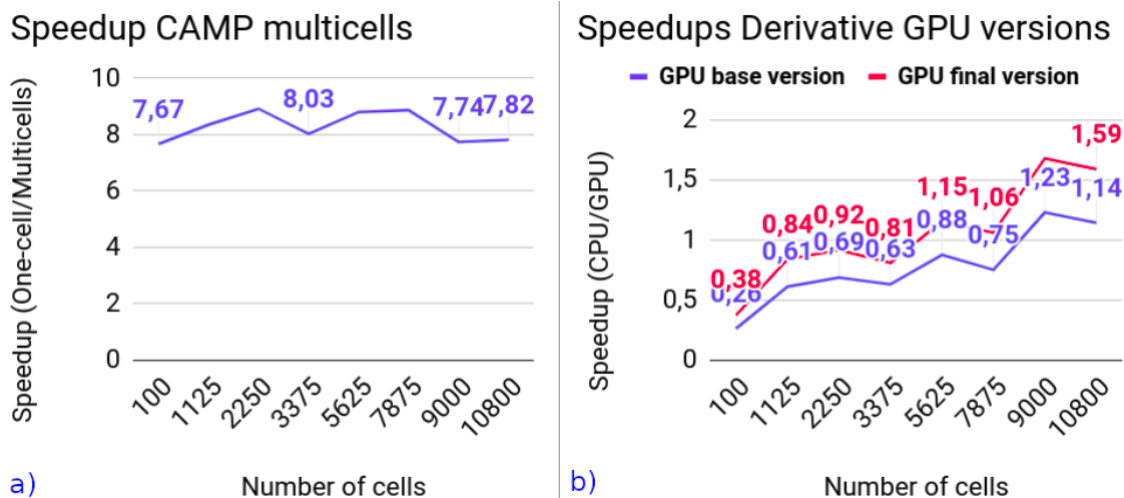


Figure 3.4: On the left (figure *a*)): CAMP speedup using Multi-cells optimization in front of the original One-cell version. On the right (figure *b*)): Speedup of base and final single-GPU versions compared to single-thread CPU versions. The final version applies the optimization on GPU memory access to the base version.

We also compare the final GPU version against a CPU case parallelized with MPI, emulating the parallelization used in MONARCH. The number of MPI processes is configured to follow the proportion of GPUs used for available GPUs. So, we use 40 MPI threads from the 160 available, like the GPU experiments presented, and 1 GPU from the four available. We obtained that the GPU execution is three times slower than the MPI, but only because the time of data movements between CPU and GPU takes nearly 90% of the GPU execution time. The GPU computation time is $3.5\times$ times faster than the MPI time (0.04s for GPU and 0.14s for MPI). This data movement is produced by updating the species concentrations on each call to Derivative. We can conclude that the GPU Derivative function has a small computation load for data movement produced (reaction data, concentration values, etc.)

3.6 Conclusions

In this chapter, we focused on improving the performance of CAMP for execution in an atmospheric model environment like MONARCH. MONARCH simulations perform one CAMP simulation for each grid cell of the geographic simulation region for each MPI thread. These cells have no inter-dependencies during the chemistry solving; thus, they have the potential to be parallelized by the GPU. However, the classical MONARCH implementation calls the CAMP-solving process for each grid cell. The CAMP solving library (CVODE) needs to reinitialize its internal solving variables for each cell iteration.

3.6. CONCLUSIONS

Furthermore, to implement a GPU implementation over the cells, it would be necessary to translate the complete solving code into GPU format, which can be exhaustive work. The first implementation presented in this chapter aims to solve these issues. This strategy is relatively novel in the atmospheric community and can be used as an example to speed up the model. In the paper, we refer to this implementation using the name Multi-cells.

The Multi-cells strategy groups the data for each cell into a single structure to be solved. MONARCH's cell loop is moved into CAMP's internal solving functions. The results show a considerable reduction in the calls to the Derivative function. The solving module uses approximately the same number of iterations to solve all the cells than to solve a single cell. Concerning the improvement in execution time, the Multi-cells implementation achieves nearly $8\times$ speedup for all the cells tested, up to $9\times$ speedup.

Next, we developed a CUDA version of the Derivative function by parallelizing its reaction loop among GPU threads. The new version obtains nearly $1.2\times$ speedup for approximately 10,000 cells. The CPU version performs better than the GPU for fewer cells. The third implementation reorders the reaction data structure to improve its access in the GPU Derivative version, increasing the GPU speedup by $1.3\times$ for all the cells tested.

Finally, we inspect the time execution consumed on moving data between GPU and CPU. For 10,800 cells, data movement takes 90% of the total time execution. Comparing the results with a 40 MPI process execution, the computation time for the GPU version is $3.5\times$ faster. Thus, the next chapter will focus on reducing GPU data movement by translating more CPU functions to the GPU, for example, the Jacobian or tasks from the ODE solving and overlapping some CPU and GPU work. This should increase the computation performed on the GPUs and reduce data movement by transferring data only at the start and the end of the solving, reducing data movement during solver iterations. This can be done by parallelizing the following solver functions executed after or before the Derivative calculation until all the solvers are executed in GPU. In future chapters, we expect to evaluate the GPU-based chemistry solving in MONARCH, checking the impact for various atmospheric experiments with an MPI implementation alongside the GPU-CUDA chemistry. Lastly, we expect to explore load balancing the CPU and GPU using overlapping and asynchronous communication since the CPU is not currently performing any work during GPU execution.

Chapter 4

Optimized thread-block arrangement in a GPU implementation of a linear solver for atmospheric chemistry solvers

4.1 Introduction

In this chapter, we present a new strategy to improve the computational load distribution of a GPU chemical solver. The current solutions assign to each thread the workload of a cell [34] [35] [36]. This workload can be further divided into solving each species concentration of a cell since many of these calculations are independent. In this way, we increase the distribution or parallelization of the load beyond the work assigned to one cell per thread to a total number of cells times the number of chemical species. This leads to significantly improved exploitation of the GPU's high bandwidth capacity. Applying this change requires extra development work to efficiently communicate data dependencies between species in the same cell and transform concentration array loops into parallel tasks. However, we demonstrate the rewards of this effort in terms of improved performance and present a novel approach that should encourage the community to consider seriously porting complex chemical solvers to accelerators. We name this approach Block-cells.

The Block-cells solution also offers functionality unavailable in efficient CUDA libraries such as cuBLAS or cuSolver. For context, some algebraic operations require communication between GPU blocks, such as finding the minimum value of an array. Inter-block communication can consume over 50% of the total execution time [106]. In the context of atmospheric chemistry, the cells to solve are typically small enough to fit within individual GPU blocks, obviating the need for inter-block communication. Ideally, having an option within these libraries to indicate this scenario would

be beneficial, allowing for more efficient handling.

However, such an option is not available in current CUDA libraries. Consequently, sending the cells as a single data structure results in excessive and unnecessary communication overhead. Alternatively, invoking these libraries for each cell individually would lead to millions of calls, introducing a significant overhead due to the initialization of each GPU call.

To circumvent these inefficiencies, we manually implemented the Block-cells approach, which avoids the limitations of existing libraries and optimizes performance by eliminating unnecessary communication and minimizing the overhead of GPU calls. This manual implementation allows for efficient parallelization of the ODE solver while maintaining the flexibility to handle the specific requirements of atmospheric chemistry simulations. By doing so, we achieve a more tailored and high-performing solution that leverages the strengths of GPU computing without being constrained by the limitations of existing CUDA libraries.

We have extended the work from the previous Chapter 3 with the implementation in CAMP of a linear solver optimized for use on GPUs: a Biconjugate Gradient (BCG) linear solver [59]. We compare the default CPU-Based KLU linear solver [95] available in CAMP with the new GPU-based linear solver to evaluate the Block-cells approach.

The application context used in this chapter is described in Section 4.2. Section 4.3 introduces the new Block-cells approach. In Section 4.4, we present the software configuration for the tests performed. Results are discussed in Section 4.5. Finally, Section 4.6 presents concluding remarks and future work.

4.2 Background

This section describes the state-of-the-art and computational description as the starting point before our developments.

4.2.1 State of the art

The state-of-the-art related to GPU chemistry modules is explained previously in Section 1.2. This section details current approaches to implementing GPU computing in chemistry modules and the available GPU tools. For example, it explains why CUDA is used over other parallel languages such as OpenMP or OpenACC.

Section 1.1 defines the motivation behind using CAMP instead of other chemistry modules. Section 2.2 provides more details about the benefits of CAMP and includes a computational description of CAMP in Section 2.2.2.

Our starting point is defined in the previous Chapter 3. In that chapter, the Multi-cells strategy is presented to easily integrate GPU functions into chemistry modules. The chapter includes a GPU version of the most time-consuming function, following the Multi-cells strategy. This chapter extends this work by adding a GPU implementation of another time-consuming function, the linear solver. The linear solver was chosen because it shares many operations with the ODE solver, such as two-vector multiplication, facilitating the future development of a GPU version of the entire ODE solver. However, translating the whole ODE solver involves a significant amount of code. Therefore, the approach followed in this chapter aims to be as efficient as possible to minimize future development efforts related to code re-writing.

4.2.2 Computational description

This section defines the base implementation of CAMP before this thesis and the implementation presented in the previous Chapter 3.

CAMP can easily be implemented in an atmospheric host model following one of two approaches: One-cell or Multi-cells [50]. Figure 4.1 shows a schematic workflow of the One-cell or Multi-cells configurations for CAMP. One-cell corresponds to most models' classical implementation of chemical mechanism solvers. In this configuration, CAMP is employed to advance the concentrations of the chemical species in each model grid cell sequentially in time. The overall rate of change for each species y_i and reaction j at any given time is, thus,

$$f_i \equiv \frac{dy_i}{dt} = \sum_j \left(\frac{dy_i}{dt} \right)_j,$$

In contrast, the Multi-cells approach takes advantage of the flexibility of CAMP by grouping multiple cells in batches and solving the chemical mechanism for each batch of cells simultaneously. As an example, the equation is updated as follows:

$$f_i \equiv \frac{dy_{ik}}{dt} = \sum_j \left(\frac{dy_{ik}}{dt} \right)_j$$

where y_{ik} refers to the species y_i from cell k .

With this approach, there is no need to loop over cells or re-initialize ODE solver parameters and data structures for each cell, and solver iterations could be reduced by a factor of 10^4 , resulting in up to a $14\times$ speedup over the One-cell approach [50]. Moreover, the Multi-cells approach maximizes the amount of information passed to the solver simultaneously, making it possible to explore massive parallelism. However, Multi-cells could lead to less accuracy since it solves a single enormous structure with all the ODE equations coming from each cell instead of solving them

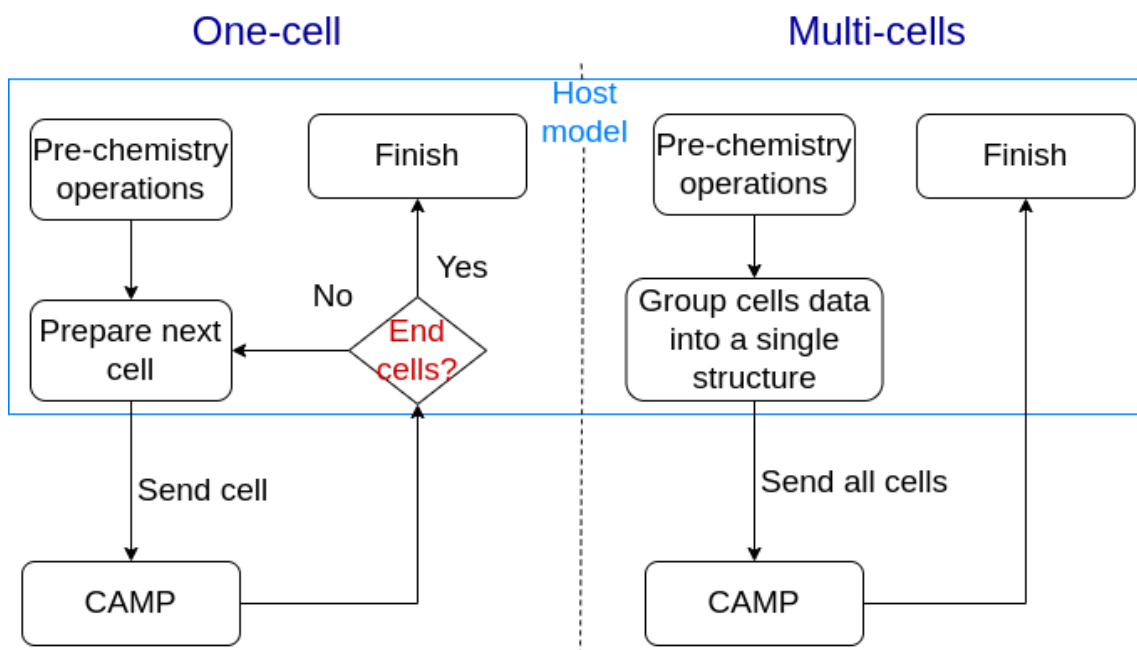


Figure 4.1: Workflow diagram of the One-cell (left) and Multi-cells (right) solving strategies.

separately. For this reason, we present a different approach, named Block-cells, explained in Section 4.3.

4.3 Block-cells implementation

In this Section, we present our methodology to optimize the computational load of a linear solver using GPUs, namely the Block-cells approach. As a first step, we have coupled a GPU-based linear solver in CAMP to analyze the performance of different GPU-based parallelization strategies: One-cell, Multi-cells and Block-cells. As a second step, we evaluated different kernel configurations of the Block-cells approach.

4.3.1 Coupling a GPU BCG linear solver in CAMP

Originally, CAMP's only linear solving option was the CPU-based KLU sparse linear solver. However, to leverage the benefits of GPU architecture, we have coupled CAMP to a sparse CUDA version of the Biconjugate Gradient (BCG) algorithm developed at the Barcelona Supercomputing Center [59]. The BCG algorithm was chosen because multiple studies have found it performs better than other Conjugate

4.3. BLOCK-CELLS IMPLEMENTATION

Gradient (CG) methods [107]. Moreover, it is designed specifically for use on GPUs, making it a better candidate for this application than simply translating a CPU-focused algorithm like KLU.

We applied the BCG solver to the One-cell and Multi-cells implementations. In the One-cell version, each call made to the GPU passes the state of a single cell. As a cell comprises hundreds of species, this under-utilizes the millions of threads available on modern GPUs. The Multi-cells approach, in contrast, gathers all the cells into a single data structure before any GPU computation, preparing all the data for parallel execution, resulting in a promising solution to the problem of adequately exploiting the capacity of the GPU. In this approach, the GPU can simultaneously compute the states of thousands of cells.

However, the Multi-cells implementation requires an extra reduction operation on the CPU. This operation involves summing each element of an array to obtain a final single value, which determines if the BCG performs another iteration or finishes. Figure 4.2 illustrates how, in the Multi-cells approach, data is transferred to the CPU during each solving iteration to perform the reduction and convergence checking. This data reduction can account for more than 50% of the total execution time [106].

This extra reduction is necessary with the Multi-cells approach, as convergence must be evaluated for the entire system comprising all cells. However, the whole system can also be treated as multiple independent systems of cells as in the One-cell implementation and form the basis for a new parallel implementation. The computational load for the system’s cells can be distributed across blocks, following a CUDA thread block distribution [108]. This way, the CPU reduction operation can be avoided, encapsulating the BCG operations in a single kernel call. This implementation is called the Block-cells approach, illustrated in Figure 4.3.

However, calculating chemical species concentrations should be equal to or less than the maximum block size to avoid communication between GPU blocks. In modern GPUs, this block size is usually 1024 threads. Only in rare cases do ESMs use mechanisms with more than 1024 species. Often, chemical mechanisms comprise less than two hundred species. Therefore, each block can accommodate one or more cells. For example, a mechanism with 100 species can be solved with up to 10 cells in a block. In this chapter, we evaluate the performance of various block size configurations.

4.3.2 Block-cells kernel configurations

In the following, we refer to the number of species in a cell as the “cell size.”

In Block-cells (1), the number of threads per block corresponds to the number of species concentrations in a cell. The number of blocks corresponds directly to the number of cells.

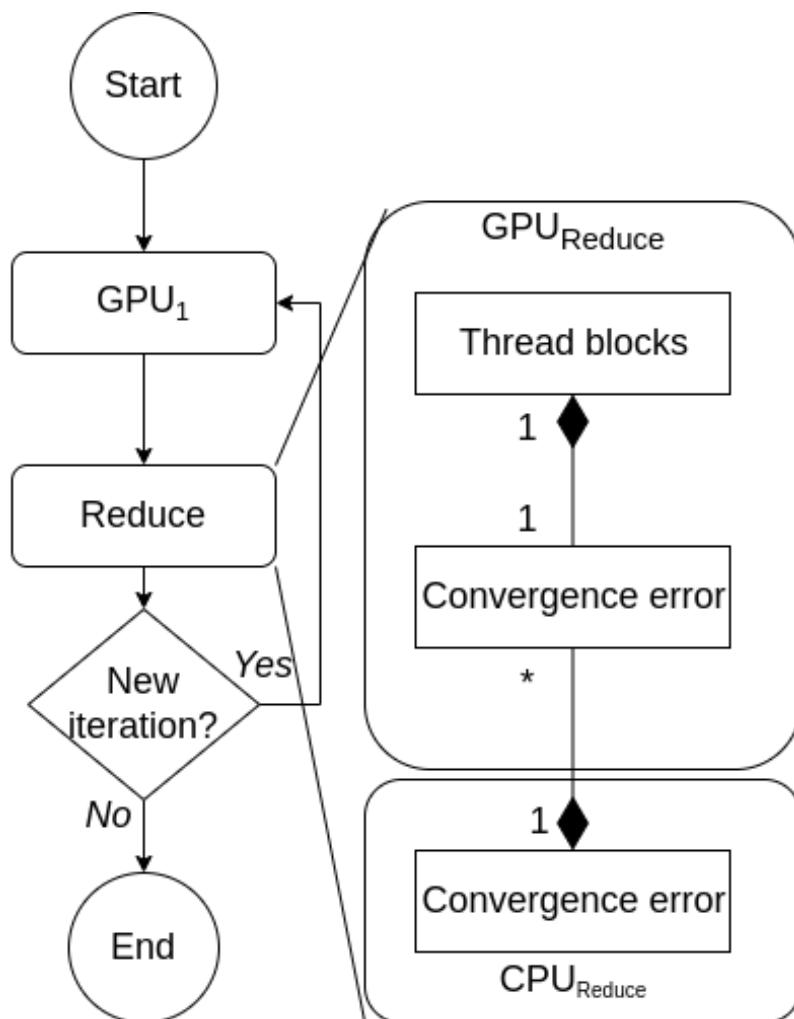


Figure 4.2: Diagram of BCG Multi-cells interactions between CPU and GPU (excluding data transfers). GPU_1 contains all the BCG operations in the GPU except for *Reduce* (e.g., functions like dot vector, SPMV, etc.). The *Reduce* computation is divided between CPU and GPU. Each thread block performs a reduction operation on the GPU, resulting in a convergence error for each block. Then, another reduction is performed over these errors on the CPU side. The final value of the reduction is used to check for convergence, i.e., if the algorithm needs to iterate again or finish.

In Block-cells (N), we calculate the maximum number of cells per block without partitioning any cell. It is calculated by dividing the block size by the cell size and rounding down the result. The residual of the division is calculated in a separate kernel. For example, in an experiment with 11 cells, 100 chemical species, and 1024 threads per block, the GPU would run a kernel with 1000 threads corresponding to 10 cells and another kernel of 100 threads for the last cell.

In Block-cells (2) and Block-cells (3), we test 2 and 3 cells per block, respectively.

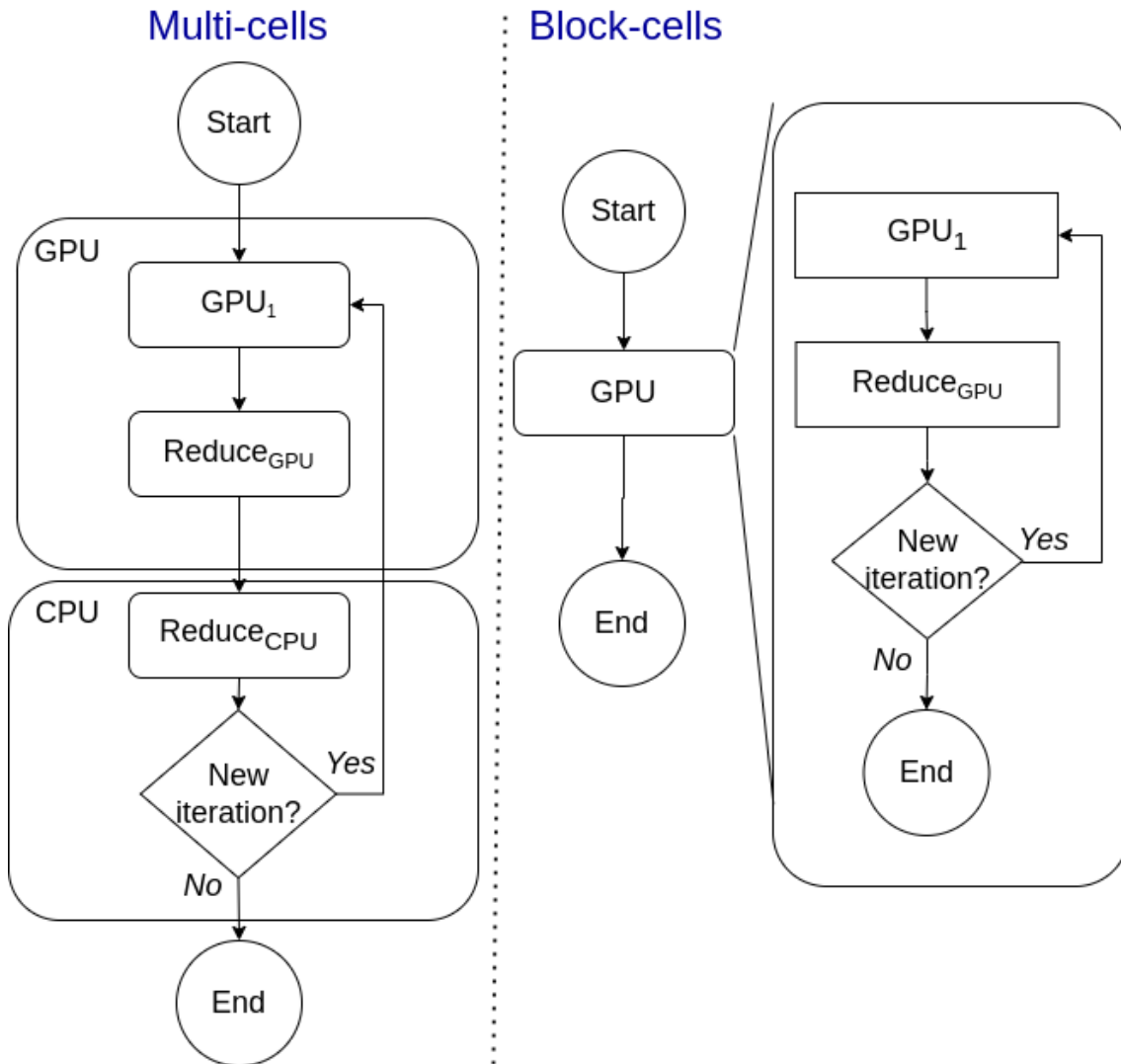


Figure 4.3: Diagram of Multi-cells and Block-cells interactions between CPU and GPU (excluding data transfers).

This range is in the middle of Block-cells (1) and Block-cells (N), allowing us to see how the performance varies with block size and to find the optimal value. Table 4.1 summarizes the cells-per-block and threads-per-block configurations.

The reduction operation in the GPU uses a shared memory array, whose length is always set to a power of two, following recommendations from NVIDIA developers intended to improve the efficiency of reduction operations [109]. Therefore, if the number of threads per block is not a power of two, the shared memory is set to the next power of two (e.g., for 100 threads per block, the next power of two is 128). Table 4.2 lists the shared memory configurations for the Multi-cells and both Block-cells (1) and (N) implementations.

4.3. BLOCK-CELLS IMPLEMENTATION

Table 4.1: Cells per block and threads per block of Multi-cells, Block-cells (1), and Block-cells (N). $float(Cells/block)$ indicates that part of a cell can be computed in a block while another block would compute the rest. $int(Cells/block)$ indicates the opposite; each cell should be computed by a single block. Chemical *Species* depends on the chemical configuration used.

Case	Cells/block	Threads/block
Multi-cells	$float(Cells/block)$	Maximum
Block-cells (1)	1	Species
Block-cells (2)	2	Species
Block-cells (3)	3	Species
Block-cells (N)	$int(Cells/block)$	$int(Cells/block)*Species$

Table 4.2: Shared memory of Multi-cells, Block-cells (1), and Block-cells (N).

Case	Shared memory
Multi-cells	Maximum
Block-cells (1 to N)	$NextPowerOfTwo(Threads/block)$

Block-cells (N) and Multi-cells solve a system composed of multiple cells. These require fewer solver iterations than Block-cells (1), as is the case for the Multi-cells and One-cell configurations of the CPU solver [50].

The number of solver iterations is measured differently between the CPU and GPU Block-cells cases. The iterations for the CPU One-cell case correspond to the sum of iterations for all cells, as they are solved sequentially. However, for the GPU Block-cells cases, the cells are solved simultaneously across multiple threads. Therefore, the effective solver iterations correspond to those performed on the last thread block to finish the algorithm.

We expect Block-cells (1) to require fewer iterations than Block-cells (N) because solving a system of a single cell should be less complex than solving a system with multiple cells.

Nevertheless, the Block-cells (N) approach still has the advantage of reducing intermediate variables. For example, instead of storing an error of convergence for each cell, a single variable can be used for multiple cells. In addition, this approach can reduce the number of idle threads. For example, consider a system of 10 cells and 100 threads per block. By design, threads in CUDA are always launched in groups of 32, called warps. Thus, for Block-cells (1), each cell launches 128 threads, resulting in 28 idle threads per cell and 280 idle threads per block. In contrast, for Block-cells (N), the block is composed of 1000 threads, resulting in 24 idle threads, 256 less than Block-cells (1). In other words, grouping cells can be considered a

trade-off between a higher utilization of the GPU resources (specifically, memory storage and threads) and more solver iterations.

4.4 Test environment

4.4.1 Hardware

All the tests were performed on the CTE-POWER cluster provided by the Barcelona Supercomputing Center [105]. The detailed hardware specifications of each node are described below.

- Operating system: Red Hat Enterprise Linux Server 7.5 (Maipo).
- CPU Compiler: GCC version 7.3.0
- GPU Compiler: NVCC version 10.1.105
- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and four threads/core, total 40 physical cores per node and 160 virtual threads using hyper-threading)
- 512GB of main memory distributed in 16 dimms \times 32GB @ 2666MHz
- 2 x SSD 1.9TB local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.
- Single Port Mellanox EDR
- GPFS via one fiber link 10 GBit

In addition, we use the NVIDIA Visual Profiler (NVVP) from the CUDA toolkit v11.5.1 to visualize the profiling data of the GPU experiments and assess the performance of the tests under analysis.

4.4.2 Experimental setup

CAMP allows the solving of chemical mechanisms of a wide range of complexity and can treat a combination of gas and aerosol reactions. Here, we select an intermediate complexity gas and aerosol problem used in [26]. The gas phase chemistry is the Carbon Bond 2005 (CB05) mechanism [72] with fixed photolysis reaction rate constants during the integration. The mechanism is extended with secondary aerosol

4.4. TEST ENVIRONMENT

production from isoprene using a two-product model approximation. In addition, we add emissions at each time step that shift the species concentrations away from equilibrium. The reader is referred to [26] for further details of the chemical system solved here.

The relative tolerance of the CVODE solver is set to $1.0e^{-4}$, while the absolute tolerance ranges between 1.0 and $1.0e^{-3}$ depending on the chemical species (the specific configuration corresponds to the CAMP-CB05 mechanism [72] [26] and can be found in the JSON file https://github.com/open-atmos/camp/blob/main/test/chemistry/cb05c1_ae5/cb05c1_ae5_abs_tol.json. Any error of accuracy below this level of tolerance is considered negligible. The KLU solver uses the same tolerance. The tolerance of the BCG linear solver is set to $1.0e^{-30}$. This tolerance corresponds to the lowest level of accepted tolerance in CAMP. CAMP uses this low tolerance level to keep chemistry systems positive-definitive, avoiding negative concentrations produced during the CVODE solving. Any negative value greater than $-1.0e^{-30}$ produces an extra iteration in the CVODE solving algorithm. Thus, this tolerance avoids any possible extra iterations the BCG algorithm produces.

We use CAMP as a box model where the number of cells to be solved can be configured from 1 to 10,000 cells. The domain decomposition of some atmospheric models allocates around 40,000 cells per MPI process (distributed as $20 \times 20 \times 100$ for axes x, y, and z). The 10,000 cells correspond to levels where the performance results are stabilized, and any greater number of cells gives similar results.

The timing results are averaged over 720 time steps, with a time-step size of 2 min, representing 24 simulation hours. The profiling metrics obtained through NVVP are from the first time-step.

In addition, we evaluate the performance of Multi-cells with various initial conditions as our implementation depends on using CAMP with Multi-cells. Thus, we design two configurations with different initial conditions among cells. One configuration uses the same initial values for all cells. This is referred to as the *ideal* case. The other uses different initial conditions for each cell, referred to as the *realistic* case.

Specifically, the *realistic* case tries to emulate an atmospheric environment. Each cell is considered to be located at a different altitude in the atmosphere, resulting in different initial conditions. First, the pressure is configured to scale linearly with the number of cells from 1000 to 100 hPa. Second, the chemical reactions of type *emissions* also scale linearly with a rate from 1 to 0. In this way, a cell at 100 hPa has 0 emissions, while a cell at 1000 hPa has the maximum emissions value. Finally, the temperature is calculated from the pressure for dry adiabatic conditions [110].

Table 4.3 shows the various configurations of threads-per-block and shared memory analyzed for the GPU implementations. The Multi-cells case solves 6.6 cells per block, which means that part of a cell is computed in another block. Block-cells (1) solves the same threads-per-block as the number of species per cell. Finally, Block-cells (N) truncates the 6.6 cells of Multi-cells to 6 cells, resulting in 924 threads per

4.5. RESULTS AND DISCUSSION

Table 4.3: GPU kernel configuration of the implementation presented in Section 4.3 and tested in this chapter

Case	Cells/block	Threads/block	Shared memory
Multi-cells	6.6	1024	1024
Block-cells (1)	1	156	256
Block-cells (2)	2	312	512
Block-cells (3)	3	468	512
Block-cells (N)	6	924	1024

block. The shared memory length is always configured to a power of two.

4.5 Results and discussion

This Section presents and discusses the accuracy, performance metrics, and speedup of the BCG implementations for GPU compared to the default CAMP version based on the KLU solver and One-cell approach. The relative error between the species concentrations using the BCG implementations is below the CVODE tolerance of 0.01%. Hence, results from the new BCG linear solver implementation are not significantly different from those of the base implementation.

4.5.1 NVVP profiling

The NVVP profiling shows that both Block-cells configurations, (1) to (N), spend a similar amount of time executing various types of instructions (memory dependence, synchronization, etc.). Memory operations account for 50% of the Block-cells execution time, while for Multi-cells this value is 89%. Thus, the memory dependence bottleneck is reduced by 39% going from the Multi-cells to the Block-cells configuration as the synchronization is moved to the GPU. Block-cells spends 35% of the time performing synchronization tasks, as the synchronizations are now performed on the GPU instead of the CPU. Overall, these metrics indicate a performance improvement from Multi-cells to Block-cells.

Table 4.4 shows that all Block-cells configurations have similar computation intensity, which is greater than for Multi-cells. This is because Multi-cells applies multiple kernel calls, one for each vector operation—such as a matrix multiplication, a vector by another vector, or a reduction kernel. Thus, the scheduler concatenates operations between kernels, increasing the kernel synchronization overhead. Memory utilization and bandwidth are higher than Block-cells. However, these metrics decrease linearly from Block-cells (1) to (N), indicating that grouping cells reduces the

4.5. RESULTS AND DISCUSSION

Table 4.4: Device memory bandwidth and utilization of computation intensity and Memory metrics from the NVVP for 10,000 cells run.

Case	Computation intensity	Memory	Bandwidth(GB/s)
Multi-cells	7%	75%	715
Block-cells (1)	11%	65%	597
Block-cells (2)	11%	65%	568
Block-cells (3)	11%	55%	474
Block-cells (N)	11%	45%	445

Table 4.5: Efficiency and occupancy from the Properties view of NVVP for 10,000 cells run.

Case	Global Load Eff.	Warp Execution Eff.	Occupancy
Multi-cells	24.2%	35.1%	68.6%
Block-cells (1)	36.5%	75.4%	61.5%
Block-cells (2)	36.3%	75.3%	61.5%
Block-cells (3)	37%	72%	46.7%
Block-cells (N)	36.8%	75.5%	45.3%

efficiency of processing memory operations.

Next, we evaluate the NVVP metrics of *Global load efficiency*, *Occupancy* and *Warp execution efficiency*, which are defined as follows [111]:

- *Global load efficiency* is the ratio of requested global memory load throughput to required global memory load throughput expressed as a percentage;
- *Occupancy* is the ratio of the average active warps per active cycle to the maximum number of warps supported (a warp in CUDA is a group of 32 threads);
- *Warp execution efficiency* is the ratio of the average active threads per warp to the maximum number of threads per warp expressed as a percentage.

Table 4.5 demonstrates that both global load and warp efficiencies improve across all Block-cells configurations compared with the Multi-cells approach. However, Block-cells (1) and (2) exhibit higher occupancy compared with (3) and (N). This discrepancy arises because Block-cells (1) and (2) are powers of two, whereas the others are not. Consequently, the GPU architecture, optimized for powers of two, organizes resources more efficiently in these cases.

We conducted memory requirement estimations by tallying the memory-allocated arrays of the various methods. The memory required is as follows:

- KLU One-cell: 18KB
- KLU Multi-cells: 18 KB per cell
- BCG Multi-cells and Block-cells (1) to (N): 29KB per cell

It is worth noting that the Multi-cell and Block-cell approaches utilize identical arrays; the only distinction lies in how they organize the data. At most, the Multi-cells approach requires two additional variables, each with a length equal to the number of cells, for computing the reduction on the CPU. The increased memory requirements of BCG compared with KLU stem from the BCG requirement of nine additional auxiliary arrays.

4.5.2 Speedup

The One-cell CPU-based KLU solver is $2\times$ faster than the GPU-based One-cell BCG solver. This is a consequence of the GPU One-cell approach, which launches a kernel and transfers data between CPU and GPU for each cell, resulting in significant overhead. This overhead is sharply reduced in the Multi-cells approach, which performs these operations only once for all cells.

We tested a GPU Multi-cells configuration with the reduction operation on the CPU to quantify the impact of the data transfers. The original configuration transfers one variable from the GPU to the CPU for each GPU block, corresponding to the convergence error explained in Figure 4.2. The modified configuration transfers the full array of data concentrations. As the original configuration uses 1024 threads per block, the new configuration transfers $1024\times$ more data. Consequently, the new configuration is $12\times$ slower for 10,000 cells, indicating that the data transfers are very expensive.

Figure 4.4 shows that Block-cells (1) iterates less than Block-cells (N) ($1.7\times$ fewer iterations for 10,000 cells and *realistic* conditions). This confirms our expectation that the Block-cells (N) approach generates a more complex system, which takes longer to solve than the slowest cell in Block-cells (1).

Results also indicate that varying initial conditions between the cells increase the performance improvement of Block-cells (1) compared with Block-cells (N), as more iterations are reduced under *realistic* than *ideal* conditions. Moreover, these variations of the initial conditions increase the standard deviation. The plots show a low standard deviation (0.1 for a $1.8\times$ reduction). However, the standard deviation and reduced iterations could be relatively greater when large initial differences exist among the cells. Thus, the computational cost could be higher and more volatile (due to a higher standard deviation) in complex scenarios, such as in actual atmospheric simulations.

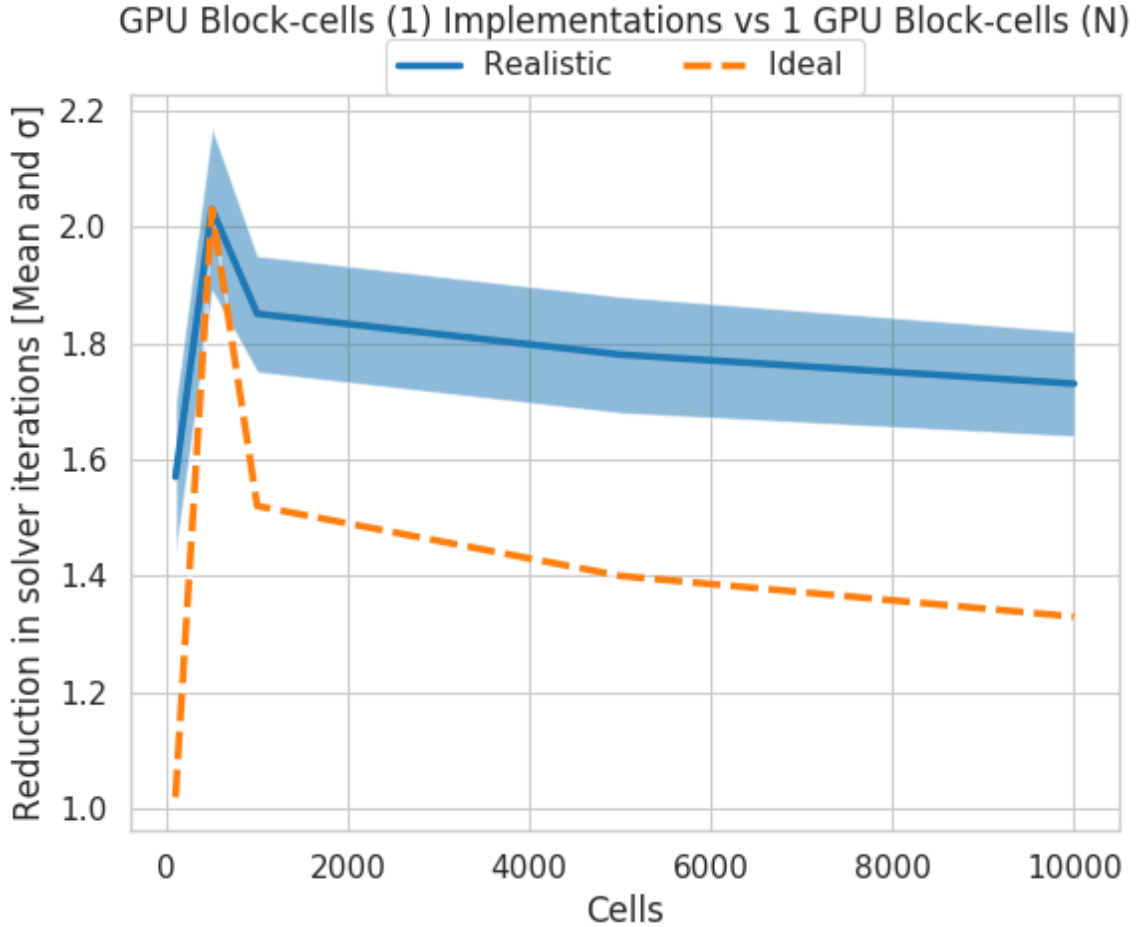


Figure 4.4: Reduction in the number of solving iterations of Block-cells (1) against Block-cells (N) using *ideal* (orange dashed line) or *realistic* (blue line) initial conditions. This reduction is calculated by dividing the solving iterations of Block-cells (N) by Block-cells (1) and corresponds to the iterations of the last thread block to finish the algorithm. The reduction is averaged over 720 time steps. The blue shade indicates the standard deviation of the reduced iterations for all time steps using *realistic* conditions.

We use the MPI library in the one-core CPU base case experiments instead of disabling it because it is simpler to configure, and the differences are negligible. Specifically, in our tests, disabling the MPI library only reduces the execution time by 0.2%. This is because MPI is only used for initialization and measuring the execution time.

Figure 4.5 shows that the speedup decreases linearly as the block size increases, with Block-cells (1) demonstrating the highest performance. This suggests that solving cells individually is faster compared with grouping them. However, despite achieving a reduction of approximately 70% in the number of solver iterations, the speedup of Block-cells (1) is not as high as expected. This discrepancy implies that there are

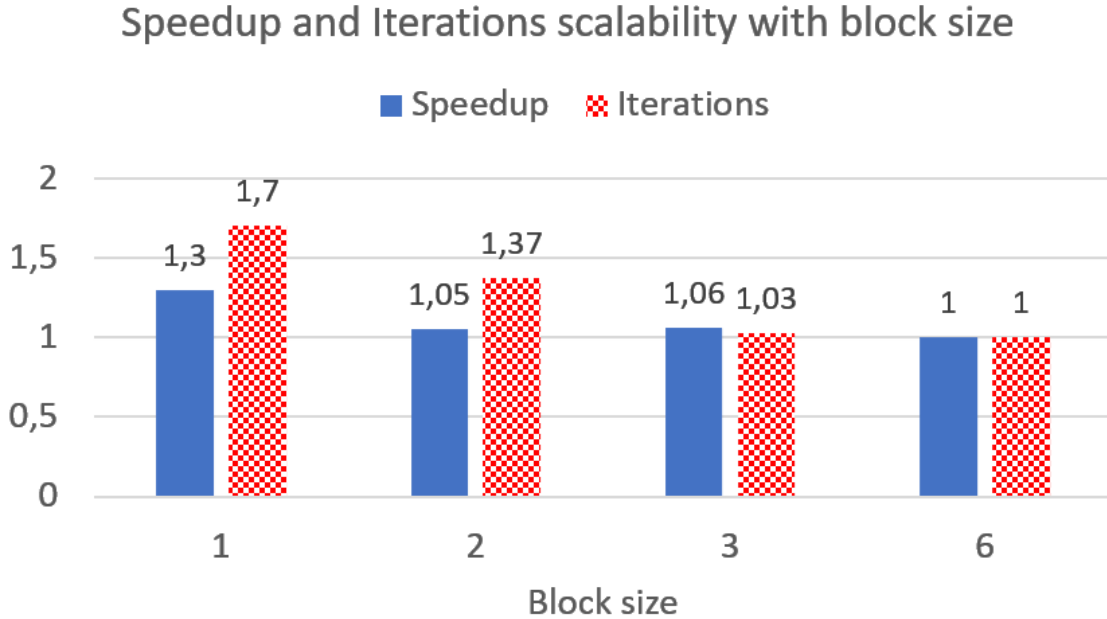


Figure 4.5: Speedups and iterations of GPU linear solver configurations. The speedup and iterations are normalized over the worst case, which corresponds to Block-cells (N) and Block-cells (3), respectively. The Block-cells (1), (2),(3), and (N) implementations correspond to block sizes 1,2,3 and 6. The results are averaged over 720 time steps and are configured with 10,000 cells and *realistic* conditions.

additional benefits associated with other configurations. We attribute this to using the same solver parameters across multiple cells, which reduces data requirements and subsequently lowers memory usage compared with Block-cells (1).

The figure also reveals a linear decrease in the number of solver iterations with increasing block size, as anticipated due to the increased complexity of the system with more cells involved. Similarly, the speedup exhibits a comparable trend, except for block size 2, where the speedup closely mirrors that of block size 3, despite a significant reduction in iterations between the two cases. This discrepancy suggests another metric influences the speedup besides the iterations, although we have not identified a clear metric in the NVVP report. Further investigation into this behavior may be warranted in the future.

Figure 4.6 illustrates the speedups of Multi-cells, Block-cells (N), and Block-cells (1) against the CPU-based One-cell implementation using a single core. The standard deviation depicted in Figure 4.6 arises from the inherent differences between the BCG and KLU linear solvers. For instance, BCG is an iterative solver, and the sequence of floating point operations differs between the CPU and GPU, introducing variability into the comparison. The Multi-cells approach exhibits the smallest standard deviation, whereas those for Block-cells (N) and Block-cells (1) are larger. This discrepancy stems from configuring the remainder of the ODE solver code fol-

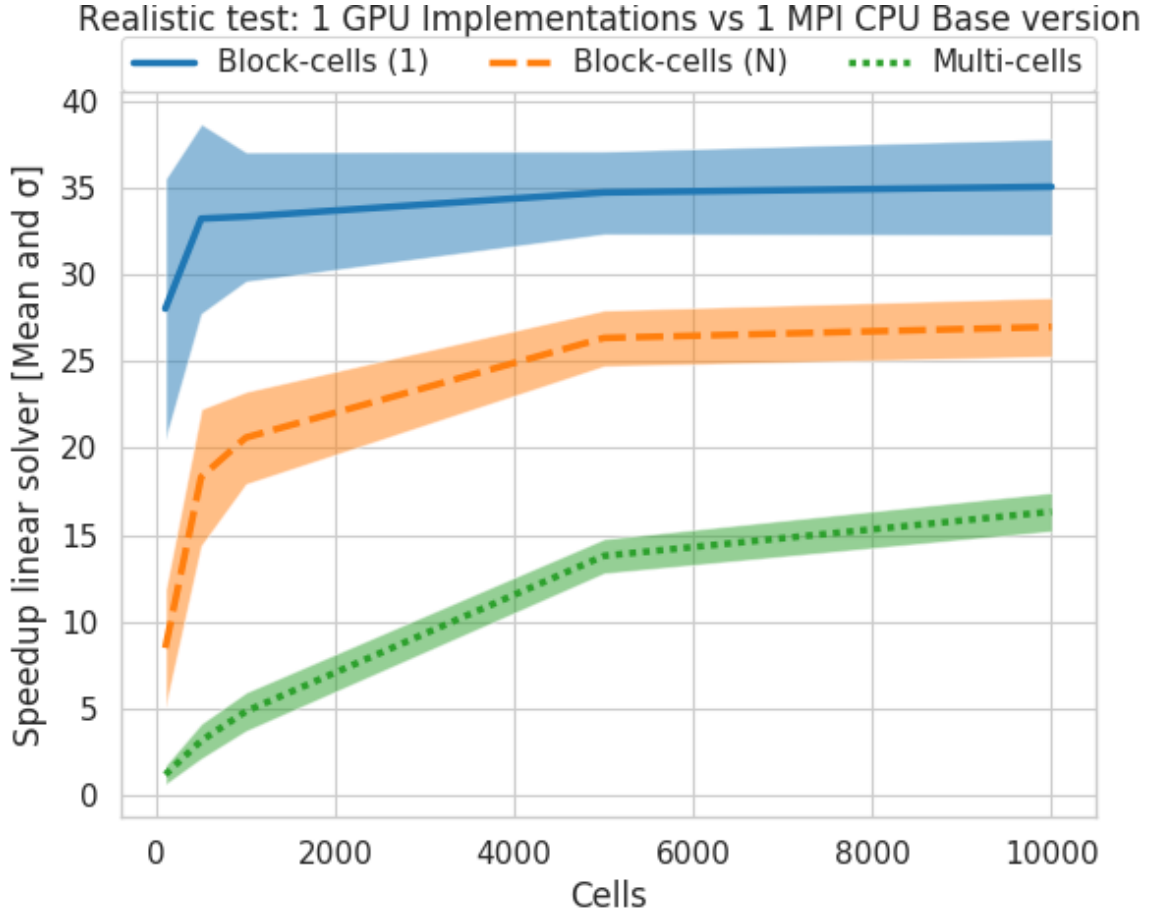


Figure 4.6: Speedups of GPU linear solver implementations compared to One-cell version executed by one CPU core. The implementations presented are Multi-cells (dotted line), Block-cells (N) (dashed line), Block-cells (1) (continuous line). The speedups are averaged over 720 time steps (as explained in Section 4.4). The area covering the speedups is the standard deviation.

lowing the Multi-cells approach. Specifically, Block-cells (1) and Block-cells (N) solve the system differently from the ODE solver they are embedded in, resulting in divergent results and increased variance. Block-cells (1) displays more variability than Block-cells (N) because they diverge more significantly from Multi-cells, as the Block-cells (N) approach solves some cells as a single system.

All the analyzed implementations result in speedups over the single-core CPU implementation. Specifically, Block-cells (N) achieves a $27\times$ speedup, compared with a $17\times$ speedup from Multi-cells, indicating an improvement in computational time associated with data transfers between GPUs. Block-cells (1) is the fastest implementation with a $35\times$ speedup.

Introducing a preconditioner to the linear solver could potentially alter the associated speedups. If the preconditioner effectively reduces the number of iterations re-

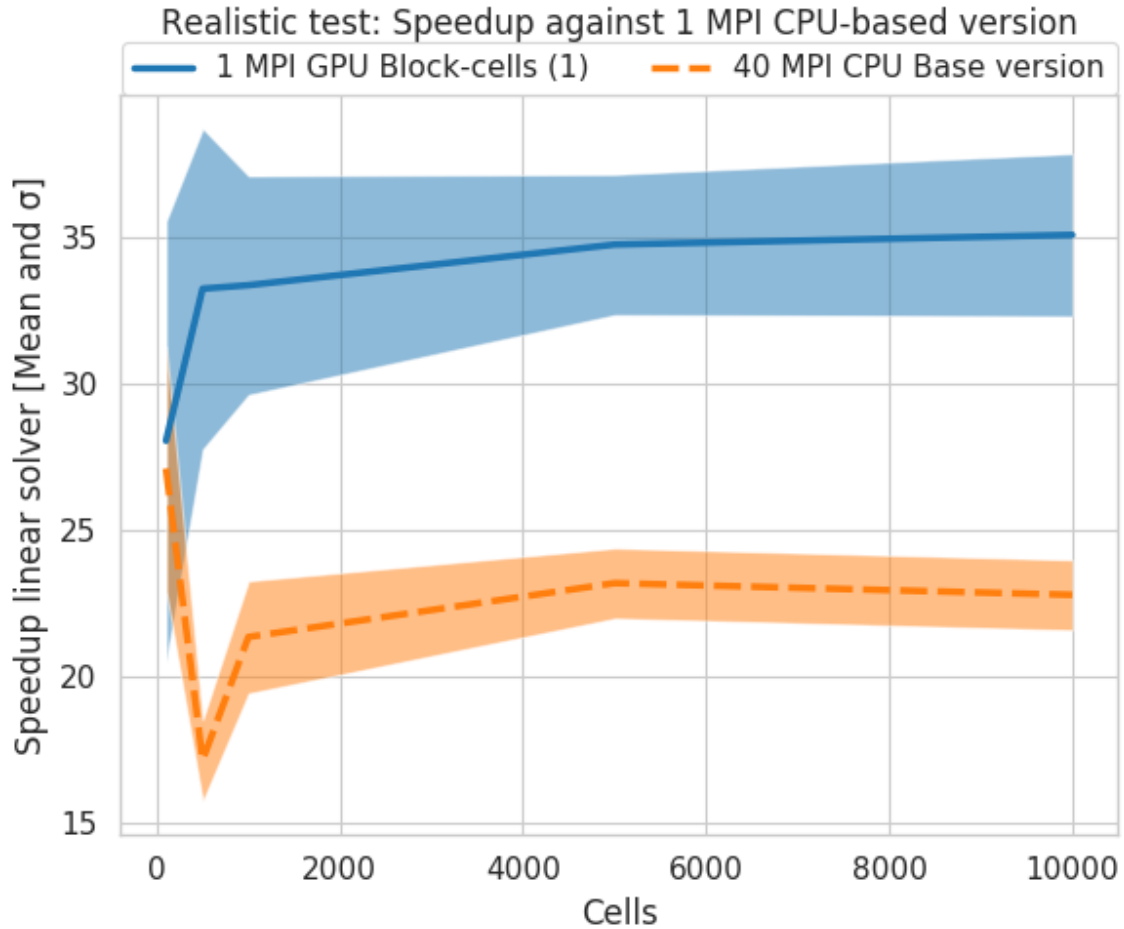


Figure 4.7: Speedups of GPU linear solver using 1 MPI core (continuous line) and CPU solver using 40 MPI cores (dashed line), both against the base CPU version using 1 MPI core. The speedups are averaged over 720 time steps. The area covering the speedups is the standard deviation.

quired, this reduces more iterations on implementations with higher iteration counts. Consequently, the speedup of Block-cells (N) may approach that of Block-cells (1), given that Block-cells (N) typically entails more iterations than Block-cells (1), as demonstrated in Figure 4.4. However, a good scaling is not compromised independently of the use or not of a preconditioner. In any case, pursuing an effective preconditioner represents a promising avenue for optimization in future research endeavors.

Figure 4.7 shows that Block-cells (1) achieves a greater speedup than the 40 cores MPI implementation ($35\times$ vs. $23\times$). This highlights the performance improvements made possible through the efficient use of the GPU. Moreover, these results can be expected to improve by a factor of 4 when using the 4 GPUs available in the node since no communication is required among the GPUs. We will explore this possibility in future work.

Finally, we inspect the time execution consumed on moving data between GPU and CPU of the best configuration, Block-cells (1). For 10,000 cells, this time on data movement takes 36% of the total time execution, which is significant. Data transfers cannot be reduced as the implementation already transfers a minimal amount of information, that is, the input and output concentrations. Alternatively, we can augment computational operations by incorporating more code from CVODE into the GPU kernel. However, this approach may require additional data transfers because CVODE uses numerous auxiliary variables. Thus, to offset these transfers, we must translate a substantial portion of the computational workload to the GPU approach, such as developing a GPU-based Block-cells (1) version of CVODE. This option requires further exploration in future investigations.

4.6 Conclusions

The main goal of this chapter has been to achieve an optimized distribution of the computational load of a chemical solver for use on GPUs. We used the CAMP framework to benchmark our developments. The standard approach in CAMP, as in most atmospheric packages, is to compute the chemical state of each cell separately. We refer to this implementation as One-cell. As in the non-parallelized CPU-based solver, these cells are computed one-by-one sequentially. In contrast, the Multi-cells approach is based on grouping the cells of the model into a single system and solving them simultaneously.

Compared with the single-threaded (sequential; not parallelized) solver used by CAMP, Multi-cells has the advantage of requiring significantly fewer solver iterations, as the total number of iterations required for the One-cell approach is the sum of the individual number of iterations needed for each cell [50]. This chapter shows that the Multi-cells performance strongly depends on the initial conditions. The speedup can be as low as $1\times$ under idealized conditions where all cells have the same initial conditions (i.e., same initial concentrations, temperature, pressure, etc). Nevertheless, the speedup can be as much as $6\times$ when the initial conditions vary among cells. Therefore, we expect Multi-cells to outperform the single-threaded One-cell approach in CAMP under realistic conditions.

We also evaluated the GPU implementation of the BCG linear solver for the One-cell and Multi-cells cases against the CPU-based KLU solver using the One-cell approach. For the BCG One-cell configuration, the speedup is less than $1\times$, as there are CPU-executed instructions between the execution of each kernel, preventing the cells from being computed in parallel. The Multi-cells approach solves this problem by computing cells in parallel, resulting in up to a $17\times$ speedup for 10,000 cells. However, the Multi-cells implementation allows the load of an individual cell to be divided between two thread blocks, requiring communication between blocks that can account for more than 50% of the execution time.

To improve the performance of the Multi-cells approach, we proposed the novel Block-cells strategy to avoid communication across thread blocks. The Block-cells approach ensures that each cell is computed within a single block, avoiding communication and synchronization across thread blocks. Block-cells can be configured differently depending on the number of cells assigned per block. This paper compares configuring one cell per block versus using the maximum number of cells per block. We call these configurations Block-cells (1) and Block-cells (N), respectively. Configurations with multiple cells per block but less than the maximum possible result in performance between these two extremes.

We find that both configurations increase the overall memory efficiency by 12% relative to the Multi-cells approach, indicating improved use of hardware resources. We also show that Block-cells (1) has 15% more kernel occupancy than Block-cells (N), meaning it has more active threads per warp. Moreover, we show that Block-cells (1) requires $\sim 80\%$ fewer solver iterations than Block-cells (N). This leads to different performance improvements for Block-cells (N) and Block-cells (1), which attain up to $27\times$ and $35\times$ speedups, respectively, relative to the CPU-based One-cell configuration.

We also compare the speedup obtained over an equivalent MPI implementation, which uses the maximum number of physical cores available on a node (40). In this MPI implementation, we emulate an actual ESM experiment and solve the number of cells in each process equal to the total cells divided by the number of processes. Block-cells (1) is up to 50% faster than this MPI implementation. This highlights the advantage of the GPU-based Block-cells approach over traditional CPU-based approaches for the specifications used in this study.

In summary, the new Block-cells strategy improves upon the previously developed GPU-based Multi-cells approach and a traditional CPU-based parallel implementation using MPI. Moreover, we present evidence that the Block-cells approach can be an excellent alternative to other GPU-based deployments, in which the workload of a cell is handled by a single thread, not by a thread block. It should also be noted that the Block-cells approach can be applied to the rest of the CAMP ODE solver algorithm, which now represents $\sim 95\%$ of the total solving time after using the Block-cells strategy to the linear solver.

Thus, in the next chapter, we will apply the Block-cells strategy to the rest of the BDF algorithm. We expect this to improve the performance relative to other GPU-based chemical solvers, as the complete algorithm will be ported to GPUs. More specifically, we hope to obtain a speedup similar to that found for the linear solver, as the BDF algorithm is conceptually identical to the BCG algorithm: both are iterative algorithms based on similar algebraic operations (i.e., vector multiplication or vector reduction to a variable). However, the BDF algorithm is more complex than the linear solver and may present unique challenges. We also expect to evaluate the Block-cells GPU-based chemistry solver in MONARCH. In future chapters, we can use the CPU and GPU solvers in a hybrid implementation.

Chapter 5

Extending the Block-Cells Approach for GPU-Accelerated ODE Solvers: Implementation, Testing, and Profiling in Box and 3D MONARCH Models

5.1 Introduction

This chapter presents a GPU version of the ODE-solving procedure of the chemistry module CAMP. The method used to adapt the ODE solver for GPU computing is introduced in the previous Chapter 4. This chapter includes results using the CAMP GPU version within the atmospheric model MONARCH to validate our approach in a more realistic case scenario. The primary goal of this work is to accelerate MONARCH while maintaining high accuracy. This chapter details the significant changes made during the porting process and presents the performance and accuracy results of the MONARCH-CAMP GPU simulations.

This chapter is organized as follows: Section 4.2 provides the application context relevant to this chapter. Section 5.3 explains the porting of the chemistry solver to Block-Cells. Section 5.4 presents the software configuration for the tests performed. Accuracy losses, acceleration, and performance are discussed in Section 5.5. Finally, Section 5.6 presents concluding remarks and future work.

5.2 Background

This section describes the state-of-the-art and computational description as the starting point before our developments.

5.2.1 State of the art

The state-of-the-art related to GPU chemistry modules is explained previously in Section 1.2. This section details current approaches to implementing GPU computing in chemistry modules and the available GPU tools. For example, it explains why CUDA is used over other parallel languages such as OpenMP or OpenACC.

Section 1.1 defines the motivation behind using MONARCH and CAMP instead of other atmospheric and chemistry modules. Section 2 provides more details about the benefits of MONARCH and CAMP, including a computational description of CAMP in Section 2.2.2.

Our starting point is defined in the previous Chapter 4. In that chapter, the Block-cells strategy is presented to accelerate a linear solver deployed for atmospheric chemistry [49], achieving a 35x speedup compared to the single-thread CPU version. This strategy was designed to be implemented keeping the same applicability as the base algorithm, avoiding changes to the solving algorithm. It can be applied to the rest of the chemistry solver, which shares many operations with the linear solver, making the porting process more straightforward.

The linear solver shares many operations with the ODE solver, such as two-vector multiplication, facilitating the future development of a GPU version of the entire ODE solver. This chapter extends this work by applying the Block-cells strategy to most of the code related to the ODE-solving function.

5.2.2 Computational description: introduction to Block-cells

The starting point is the Block-cells strategy applied to the linear solver routine, which we call BCG since it follows the Biconjugate Gradient algorithm described in Chapter 4 [49]. The Block-cells approach differs from the usual implementation of GPU chemistry modules. In the traditional implementation, each CPU process solves an atmospheric grid cell, a method referred to in this thesis as One-cell and sometimes as `One-cell-per-thread`. Each cell contains a chemical system that can be further parallelized to solve each chemical concentration, leveraging GPUs' high parallel computation capacity. This concept forms the basis of the Block-cells strategy, where each thread solves a chemical concentration, and each cell is grouped in a CUDA thread block, utilizing the shared memory of the block to transfer data rapidly between threads. Figure 5.1 illustrates the difference between the Block-cells and traditional implementations.

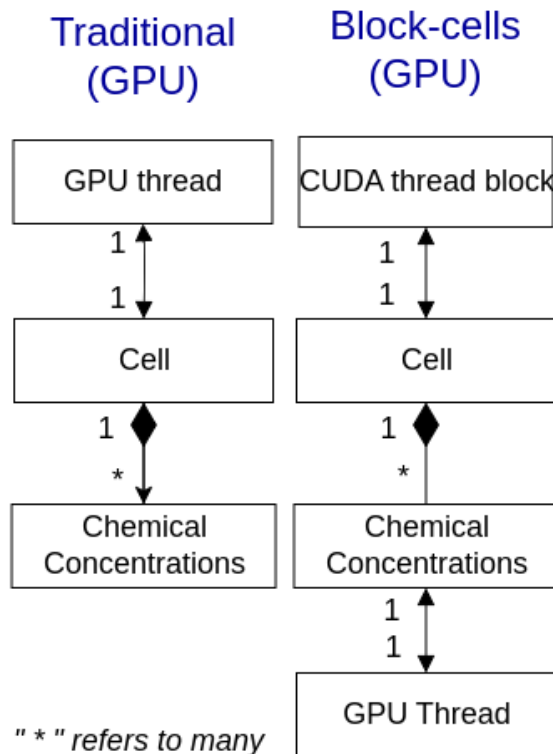


Figure 5.1: Differences between the traditional parallelization approach for CPU and GPU and our Block-cells strategy. The relations are one-to-one and one-to-many, so, for example, an MPI process contains multiple cells.

5.3 Implementations

5.3.1 Porting

When porting the CAMP solver to the Block-cells approach, it is noted that BCG (Bi-Conjugate Gradient method) accounts for approximately 20% of the total code lines from the CAMP solver. The translation of the remaining code could be a considerable task. Fortunately, many CVODE operations, such as vector multiplication and vector-by-matrix operations, are already utilized in the linear solver, simplifying the development process. The operations not included in the linear solver are introduced below, including the approach to adapt them to the GPU.

One operation is illustrated in Figure 5.2. The central concept is handling the synchronization between thread blocks to ensure they follow the same path. For instance, if a thread detects an invalid value, such as a negative concentration, this information should be shared with the rest of the threads in the same block. A similar operation is already implemented in the linear solver, known as the Reduce operation. In the Reduce operation, each thread holds a value from an array; these values are then summed up to a single value, which is shared among all threads to,

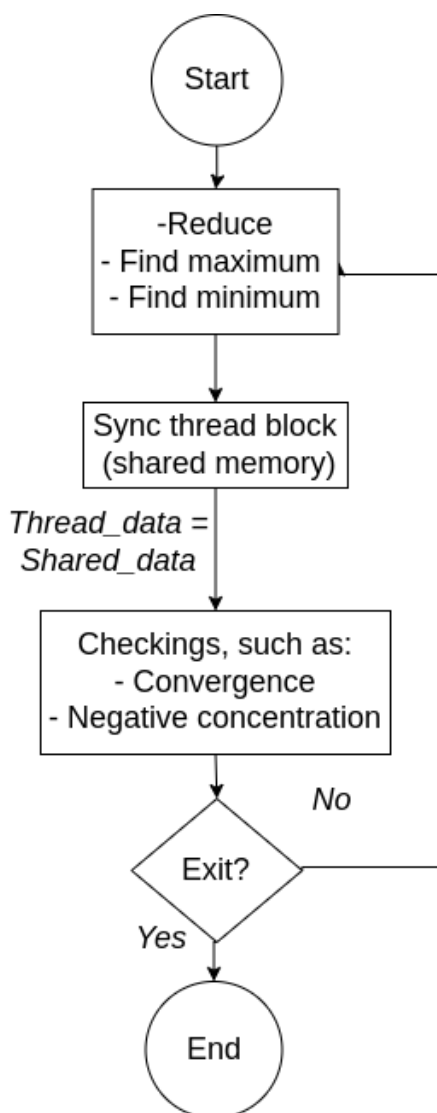


Figure 5.2: Handling of data communications between the same thread block.

for example, check for convergence. This operation can be extrapolated to find an array's maximum or minimum value.

Another new concept not present in the linear solver is calculating the $f(y)$ and Jacobian matrix. In these functions, the chemical rates of the species are multiplied by the current time-step concentration. For example, a chemical reaction can increase the oxygen atom concentration by A , and another can reduce it by B , and so on. The code is organized as follows: First, a reaction is selected, then a rate is multiplied and added to the output concentration array. Thus, there is a rate loop inside a reaction. Often, chemical mechanisms comprise hundreds of reactions, while the concentrations updated in a reaction are around ten. Consequently, it is better to parallelize the reaction loop than the rate loop.

5.3. IMPLEMENTATIONS

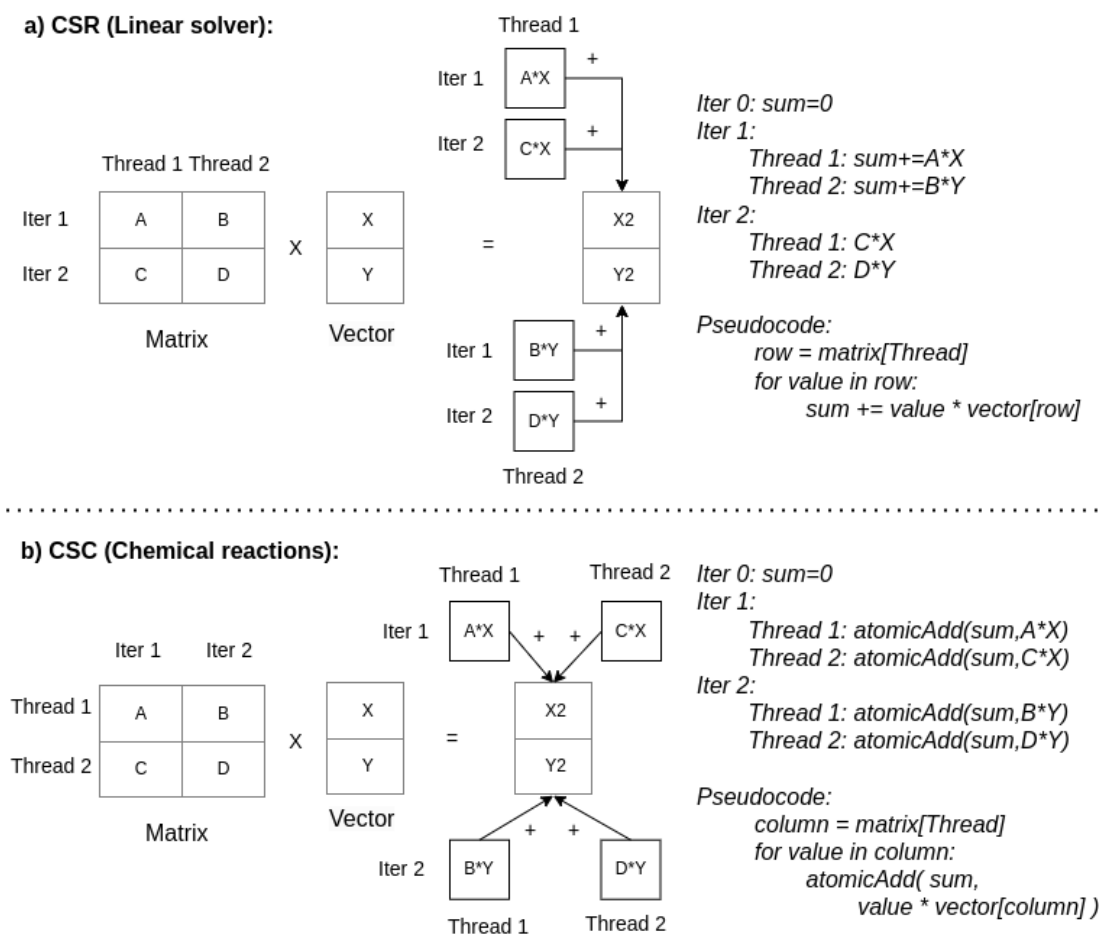


Figure 5.3: Matrix multiplication in linear solver and reactions functions.

Interestingly, this operation is equivalent to matrix-vector multiplication, where multiple factors are multiplied by a vector and summed. This operation is already performed in the linear solver, with the matrix stored in the Compressed Sparse Row (CSR) format. However, since parallelized reactions can update the same concentration simultaneously, this operation must be atomic to prevent data overlap, while the CSR format does not require atomic operations. Due to the atomic requirement, the code is much closer to the Compressed Sparse Column (CSC) format. We use the function *atomicAdd* from the CUDA library to perform the atomic operation. Figure 5.3 summarizes the main differences between these formats.

This atomic operation is usually notably time-expensive. Thus, adapting the code to follow the CSR format could save execution time. However, it would require isolating the matrix multiplication from other operations performed during the reaction calculation, such as checking for negative concentrations. This could be an extensive work that may be investigated in the future.

Figure 5.4 illustrates the components that remain on the CPU, those translated to

5.3. IMPLEMENTATIONS

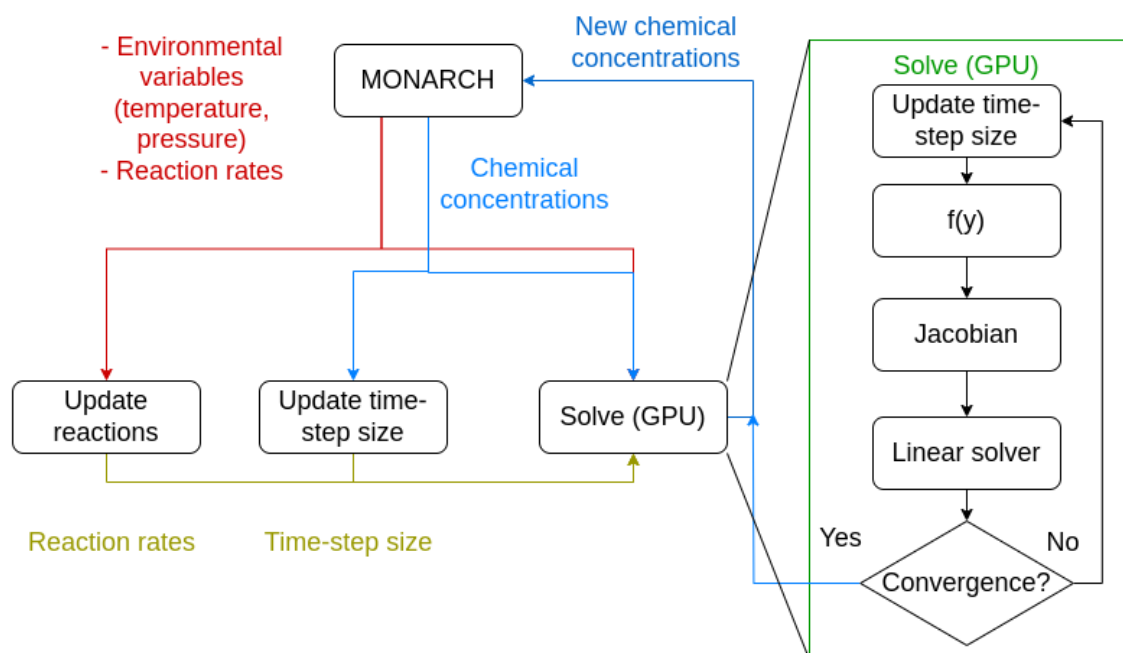


Figure 5.4: Components on CPU and GPU code. $f(y)$, Jacobian, and Linear solver are relevant elements of the ODE solver.

the GPU, and the data exchanged with the host model MONARCH. For instance, the *Update reactions* function calculates the reaction rates using environmental variables from the model or those passed by the model. The *Update time-step size* involves initialization steps performed by CVODE before entering the main solving loop, which includes checks and calculating the time-step size. These functions are executed on the CPU because they represent a low computational load and involve extensive code, particularly for the time-step size calculation.

Concerning the MONARCH coupling, we followed the Multi-cells implementation for running GPU code, as explained in Chapter 5. Specifically, in the CPU base version, single-cell data is sent to the CAMP-solving routine, which is encapsulated inside a loop that iterates over all cells. In the GPU adaptation, this loop is removed, and the CAMP-solving routine is called with data from all cells, enabling the parallelization of these cells.

Grouping the cell data involves identifying and extracting this data from the interface between MONARCH and CAMP. The data is compressed by multiple values, such as temperature, pressure, and reaction rates, which require a minor modification to the code. Ultimately, the primary challenge was understanding and managing the complex and extensive interface between MONARCH and CAMP, comprising thousands of code lines.

5.4 Test environment

All the tests were performed on the CTE-POWER cluster provided by the Barcelona Supercomputing Center [105]. The detailed hardware specifications of each node are described below.

- Operating system: Red Hat Enterprise Linux Server 7.5 (Maipo).
- CPU Compiler: GCC version 7.3.0
- GPU Compiler: NVCC version 10.1.105
- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and four threads/core, total 40 physical cores per node and 160 virtual threads using hyper-threading)
- 512GB of main memory distributed in 16 dimms × 32GB @ 2666MHz
- 2 x SSD 1.9TB local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.
- Single Port Mellanox EDR
- GPFS via one fiber link 10 GBit

In addition, we use the profiler Nsight Compute Version 2020.1.0 to visualize the profiling data of the GPU experiments and assess the performance of the tests under analysis. The chemistry time steps are set to one when profiling. These metrics are from the first time step. This is because the performance remains consistent between time steps. Different initial conditions lead to varying numbers of iterations in the ODE solver. Despite the difference in iterations, these iterations execute the same operations, such as the linear solver. We observed this behavior in the Box model simulations, validating this reasoning.

To measure accuracy between the CPU and GPU versions, we use the Normalized Root Mean Square Error (NRMSE), represented as a percentage and calculated as

$$\text{NRMSE} = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - y_i)^2}}{x_{\max} - x_{\min}}$$

Where x is the concentration of a chemical species from cell i , obtained from the last time-step of the model execution and the Base version of CAMP, y is the equivalent

of x for the optimized version, n is the number of cells, and x_{max} and x_{min} are the maximum and minimum value from x .

The absolute tolerance of the CVODE solver is set to $1.0e^{-4}$. Any error of accuracy below this level of tolerance is considered negligible. The KLU solver uses the same tolerance. The tolerance of the BCG linear solver is set to $1.0e^{-30}$. Any negative value greater than $-1.0e^{-30}$ produces an extra iteration in the CVODE solving algorithm.

5.4.1 Box model runs

During the development process, we used CAMP as a simplified box model with a trimmed configuration. This model runs the gas-phase CAMP chemical setup, specifically the Carbon Bond 5 (CB05) mechanism used in MONARCH, and enables us to scale the number of cells.

We believe it is valuable to compare the results of this test with a MONARCH run. If the results are very similar, this test can be reliably used for future developments with greater confidence. The details of this configuration are outlined in Chapter 4 on the linear solver. The main elements of the configuration are:

- Time-step size: 120 seconds
- Number of time-steps: 720
- Threads per block: 86
- MPI processors of the CPU version: 1 to 40, corresponding to a single thread and a node execution. This is used to calculate the speedup of the GPU implementation against the CPU version, providing a comparison against the single-thread and full node-to-node cases.
- Cells: 1,000 to 100,000. This range of cells is used because the GPU performance increases with the amount of parallel computational workload. Specifically, launching the GPU introduces a significant time overhead compared to the CPU case, making the GPU surpass the CPU when enough data is computed. For example, the previous work reported that launching the GPU with a single cell is 27 times faster than the CPU single-thread version while launching 10,000 cells is 35 times faster (see Chapter 4, [49]).
- GPUs: 1 to 4. Four GPUs are used to measure the speedup scaling with the number of cells, while 1 to 4 GPUs are used to measure the speedup scaling with the number of GPUs. This ensures that the speedup scales linearly with the number of GPUs. This is the expected outcome because there are no communications between GPUs, and the computational load is distributed evenly among them.

5.4. TEST ENVIRONMENT

- Pressure: Scales linearly with the number of cells from 1000 to 100 hPa. Emissions are also scaled linearly from 1 to 0. In this way, a cell at 100 hPa has 0 emissions, while a cell at 1000 hPa has the maximum emissions value.
- Temperature: Follows dry adiabatic conditions [110].

5.4.2 MONARCH run

MONARCH was configured very similarly to the CAMP paper [112], with the difference of disabling the aerosol chemistry as the CAMP GPU porting has focused only on the gas-phase chemical reactions. This configuration is detailed below.

The regional domain covers Europe and northern Africa. We used a rotated latitude–longitude projection with a regular horizontal grid spacing of 0.2 degrees. The top of the atmosphere was set at 50 hPa with 48 vertical layers. The computational resources required are two nodes (80 cores). A smaller domain was configured, scaling the cores and the region to run in 20 cores. The 20-core and 80-core (2-node) configurations allow us to evaluate the scalability of the GPU speedup with the number of GPUs. In addition, the 20-core configuration facilitates development thanks to lower execution times. In the 20-core case, only 2 GPUs are used; for the 80-core configuration, 4 GPUs are used for each node. This scalability should align with the expected linear scalability and the results from the box model experiments. Figure 5.5 displays the domains of study.

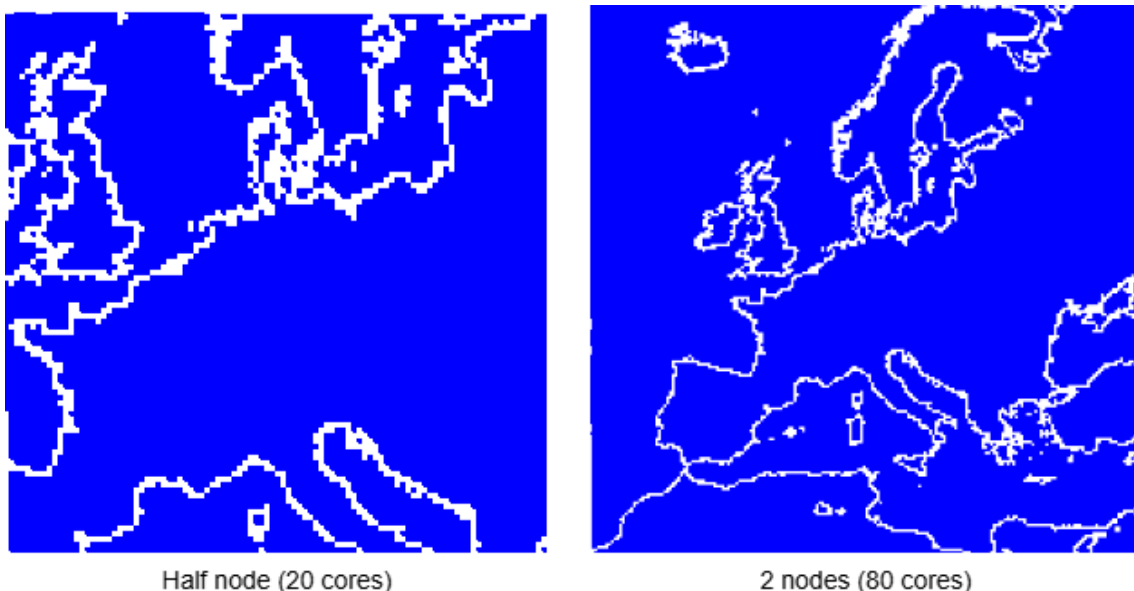


Figure 5.5: The domains of study extracted from Ncview version 2.1.10.

Meteorological initial and boundary conditions were obtained from the ECMWF global model forecasts at 0.125 degrees [113] and chemical boundary conditions

from the CAMS global model forecasts at 0.4 degrees [114]. The applied anthropogenic emissions are based on the CAMS-REG-APv3.1 database [115] [116], and the biomass burning emissions (forest, grassland, and agricultural waste fires) are from the GFASv1.2 analysis [117]. Both datasets were processed using the HERMESv3 system, an open-source, stand-alone multi-scale atmospheric emission modeling framework developed at the BSC that computes gaseous and aerosol emissions for use in atmospheric chemistry models [118] [119]. The HERMESv3 system was used to remap the original datasets and to derive hourly and speciated emissions. Aggregated annual emissions were broken down into hourly resolutions using the emission temporal profiles reported by [120]. The emissions of non-methane volatile organic compounds were speciated using the split factors reported by [115]. The Autosubmit workflow manager efficiently executed the MONARCH modeling chain [68].

The selected chemistry configuration is designed to compute the gas-phase chemistry of the mechanism CB05, neglecting aerosol processes. Therefore, MONARCH is configured as shown in Table 5.1.

The simulation lasts one day, corresponding to 480 chemistry time steps with a time step size of 45 seconds. While validation simulations in the community typically extend longer than one day—up to two years, as in the KPP GPU study [35]—our hardware transition from Marenostrom 4 to Marenostrom 5 limited our metrics to those from Marenostrom 4. We decided to present these metrics here and plan to conduct future work on Marenostrom 5, comparing the performance of both systems.

We also tested 20 steps to evaluate the variation of the NRMSE with the number of time steps. This evaluation is crucial because, in atmospheric models, an initial accuracy error can propagate and increase over subsequent time steps. However, we use the same algorithm as the CPU version with only minor changes, the most significant being using a different linear solving algorithm. Thus, we expect some slight deviations, but overall, the accuracy error should be acceptable and remain constant throughout the time steps.

We use 4 GPUs, while the CPU cores vary based on the domain size: 20 for the small domain and 80 for the large domain. The 20-core experiment is a quick test, while the 80-core experiment shows if the results change when scaling the domain. The speedup against the 4 GPUs in the 20-core setup allows for comparison with the Box model, while the 80-core experiment provides a fairer comparison since a node has 80 cores and 4 GPUs available.

Regarding the GPU configuration, the number of threads per block increases from the Box case (86 to 127) to accommodate species specific to MONARCH but absent in the CAMP chemical mechanism. These additional threads are initialized with data arrays but do not involve computations within CAMP. Therefore, the performance impact of this increase is negligible.

After configuring the domain, the number of cells is approximately 10,000 per CPU

Table 5.1: Configured chemistry schemes in MONARCH for CAMP experiments.

Process	Scheme	Comments
Gas-phase mechanism	CB05 [72]	solved using CAMP [26]
Aqueous sulfate formation	Described by Spada [65]	Deactivated
Inorganic Aerosol mechanism	EQSAM [75]	Deactivated
Organic Aerosol mechanism	Two-product or Simple scheme [77]	Deactivated
Photolysis rates	Fast-J [74]	Activated
Dry deposition of gas species	Resistance approach [78]	Activated
Dry deposition of aerosols	[79]	Deactivated
Wet deposition of gas species	Grid and sub-grid scale [80]	Activated
Wet deposition of aerosols	[21]	Deactivated
Biogenic emissions	MEGANv2.04 [81]	Activated
Dust emissions	[21] [82]	Activated but not involved in chemistry
Sea salt emissions	[64] [83]	Activated but not involved in chemistry
Pollen emissions	[84]	Deactivated
Dust Mineralogical composition	[85]	Deactivated

core. This quantity should be sufficient to saturate the speedup since in the previous Chapter 4, the saturation is around 10,000 cells for the whole node, corresponding to 250 cells per CPU core. To confirm this, in Section 5.5, we will present a scalability plot of the box model with varying numbers of cells.

5.5 Results and discussion

5.5.1 Box model runs

For all the Box model experiments increasing the number of cells, the NRMSE is below 0.02%. Table 5.2 shows that the speedup stops scaling beyond 100,000 cells. It's worth noting that the speedup remains positive from 5,000 cells onwards, indicating that even smaller configurations can benefit from GPU acceleration. The

speedup scales with the number of cells because launching the computation (also considering the overhead of data transfers) introduces a significant time overhead. The GPU must process a substantial amount of data to overcome this overhead. The table shows that the speedup stops scaling at 100,000 cells. A typical MONARCH execution uses 400,000 cells per node (or even more), corresponding to 100,000 cells per GPU since a node contains 4 GPUs. Therefore, we are processing enough data on the GPU to minimize the overhead of launching the computation.

N^o of Cells	Speedup vs CPU node
1000	0.4
5000	1.7
10 000	4.7
50 000	5.5
100 000	5.9

Table 5.2: Speedup scaling the number of cells, using 4 GPUs against all the node cores (40).

Table 5.3 shows that the speedup scales linearly with the number of GPUs, which is expected since there is no communication between the GPUs. Additionally, the speedup for a single GPU is very similar to the linear solver speedup achieved in the previous Chapter 4 (31x and 35x), which is a consequence of being composed by many similar operations to the BCG.

N^o of GPUs	Speedup vs 1 CPU Core
1	31
2	69
3	103
4	137

Table 5.3: Speedup scaling the number of GPUs. Speedup using 1 to 4 GPUs against 1 CPU core and 100,000 cells.

5.5.2 MONARCH model runs

The results for MONARCH are divided into two parts. The first presents the speedup and accuracy results, while the second covers the profiling results and performance evaluation.

Speedup and accuracy

We first show the results of MONARCH using the 20 cores MPI setup. Figure 5.6 presents the box plot of the six gas-phase species of CB05 with the highest NRMSE, showing the quantiles and median of the relative error ($100 * \left(\left| 1 - \frac{x}{y} \right| \right)$). The median error is considerably low, below 0.5%. The 75th percentile is 2%, indicating that some species are less accurate, but the median below 0.5% shows that the errors are acceptable. Additionally, Table 5.4 indicates that the highest NRMSE is below 1%, which is low enough to be considered acceptable.

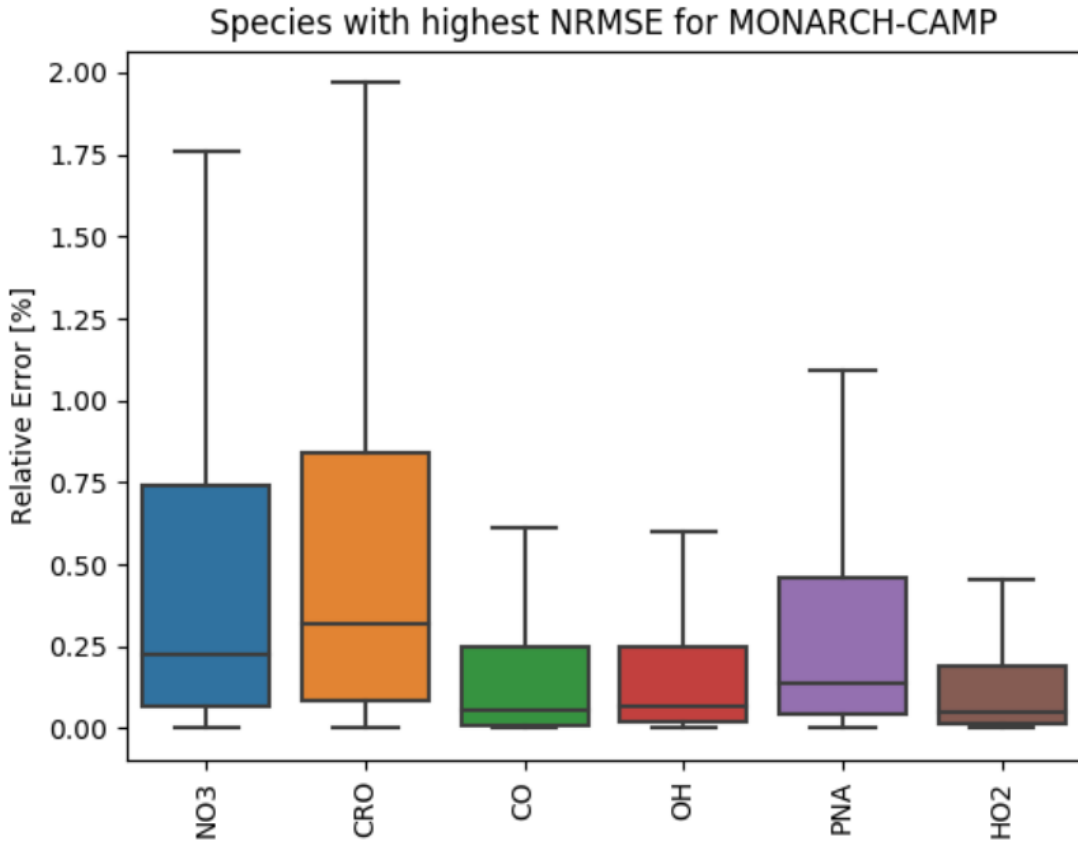


Figure 5.6: Relative error of the species with the highest NRMSE in MONARCH-CAMP for 20 CPU cores and 480 time steps.

	NO3	CRO	CO	OH	PNA	HO2
NRMSE [%]	0.68	0.2	0.17	0.16	0.15	0.13

Table 5.4: Species with the highest NRMSE in MONARCH-CAMP for 20 CPU cores.

Table 5.5 shows the error and speedup metrics for the Box and MONARCH tests using the 20-core domain configuration. The error is more significant in MONARCH,

which is reasonable given MONARCH’s more diverse initial conditions. Despite this, the error remains acceptable as it is below 1%. The difference in speedup is 13% (16.6 and 14.7). These low differences arise because the MONARCH and Box tests are similar but not completely identical. Specifically, both share the same chemical configuration, but the initial concentration and environmental variables differ. For instance, the Box model emulates cells of different altitudes with different temperatures and pressures, similar to the MONARCH configuration. However, MONARCH also contains cells at the same altitude and pressure levels, which vary from time step to time step due to factors like the day and night cycle. Nevertheless, the speedups are similar enough to indicate that the Box test reliably assesses MONARCH’s performance.

Test	NRMSE Max. [%]	Speedup
Box	0.02	16.6
MONARCH	0.68	14.7

Table 5.5: Speedup and accuracy error using 4 GPUs against 20 CPU cores.

Next, we evaluate the configuration results for the 80-core (2-node) setup. This evaluation aims to validate our assumptions and the results from the Box model, where the speedup is expected to scale linearly with the number of GPUs and MPI processors. Additionally, we hope that the accuracy should also be very similar. The NRMSE for 1-hour and 24-hour simulation (20 and 480 chemistry time-steps) is 0.42% and 0.45%, respectively. The decay and the low variation suggest that the NRMSE maintains stability with the time steps.

Figure 5.7 shows the six species with the highest NRMSE as shown in Figure 5.6 but for the 80-core case. Results show a 75 quantile of 1.4, lower than the two presented for the 20 cores experiment. These differences are likely due to the varying initial conditions between the two cases.

Table 5.6 presents the error and speedup metrics for the Box and MONARCH tests using the 80-core domain configuration. The accuracy error decreases from 0.68% to 0.42%, indicating that increasing the experiment data still maintains an acceptable NRMSE below 1%. Additionally, the table shows the theoretical speedup normalized to 20 cores. Remarkably, the actual speedup achieved is higher than expected, increasing from 7x to 9.8x.

Upon investigation, we discovered that the CPU version frequently reaches the maximum number of solving iterations, resulting in more execution time and, consequently, enhanced speedup of the GPU version. This behavior is produced by an update between the initial CAMP paper [26] and this study, related to the function *guess_helper* detailed below.

```
/* Code extracted from the function "guess_helper" */
```

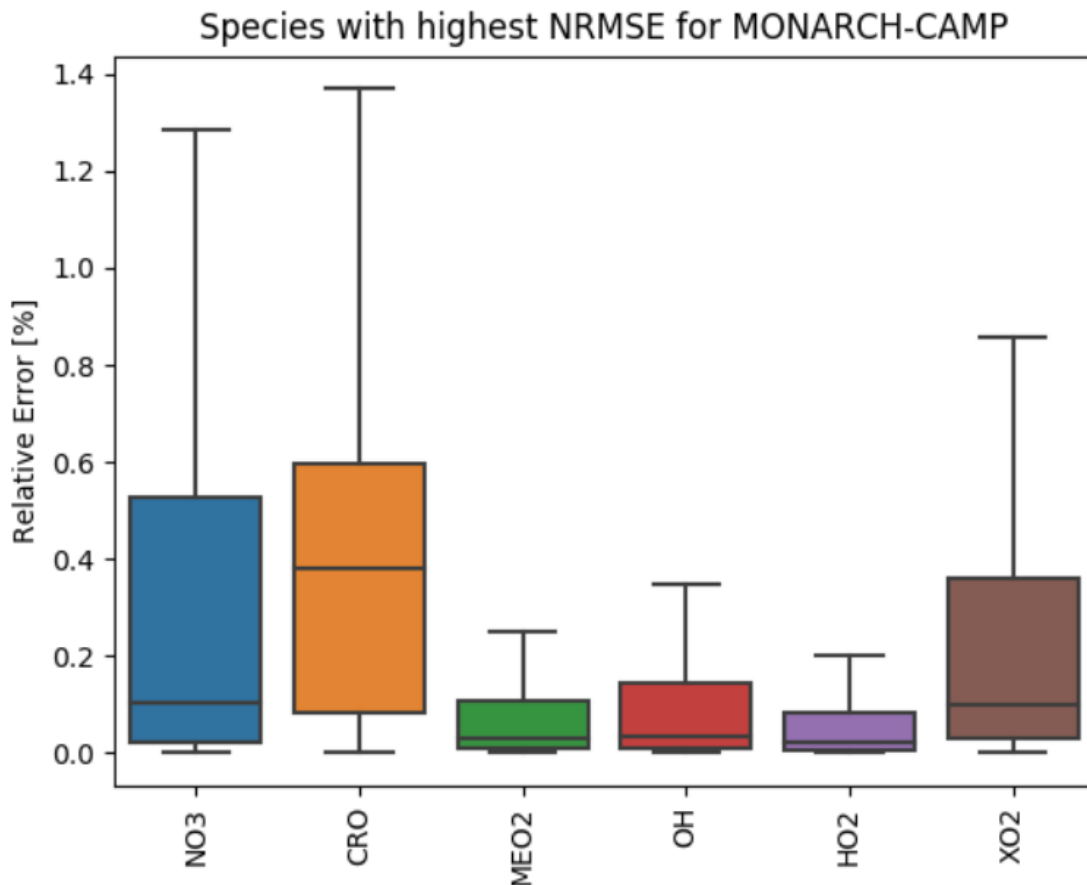



Figure 5.7: Relative error of the species with the highest NRMSE in MONARCH-CAMP for 80 CPU cores (2 nodes) and 480 time-steps.

Test	Accuracy error [%]	Speedup
Box	0.02	8x
MONARCH	0.42	9.8x
MONARCH_20 (Normalized from 20 cores)	0.68	7x

Table 5.6: Speedup and accuracy error using 8 GPUs against 40 CPU cores normalized and not normalized.

```
h_j *= 0.95 + 0.1 * rand() / RAND_MAX; // Previous version
h_j *= 0.95 + 0.1 * iter / GUESS_MAX_ITER; // Current version
```

This update was initially introduced to reduce solving iterations but was not tested in MONARCH. Interestingly, the GPU version does not encounter the issue of solver failures, likely because it uses a different linear solving algorithm, which is the most significant difference apart from the compiler. This hidden improvement in the speedup is produced because the CPU performs slower than expected. Since this

work focuses primarily on GPU advancements, investigating this CPU issue will be deferred to future work.

Profiling

The Nsight profiler reports that arithmetic resources (SM) utilization is 12%, and memory is 24%. The profiler indicates low utilization, which should be at least 60% to avoid this warning. The previous linear solver reported utilization of 11% for SM and 65% for memory. Thus, this new version worsens memory utilization from 65% to 24%. This likely results from the numerous synchronizations of the added *Reduce* or *Atomic* operations. This indicates room for improvement, but removing the atomic operation requires a significant development effort due to the many data structures involved, as we state in section 5.3.1. This work limits our focus to a GPU adaptation with low development efforts; therefore, we will aim to improve this utilization in future work.

Figure 5.8 shows the performance represented as a Roofline model from Nsight. The red point represents the double-level floating point operations. The arithmetic intensity is 4 FLOPs/byte, close to the ideal value of 7. Being less than the perfect value indicates that our application is memory-bound. However, being close to the ideal is positive, suggesting that the bottleneck is likely in other areas.

Additionally, the cache L1 and L2 hit rates are 82% and 99%, respectively, indicating that the memory accesses are highly efficient. This is because most of the operations in the GPU ODE solver involve sequential access. For instance, typical operations include the sum or multiplication of two vectors of the same length. The only exceptions to this pattern are the synchronization operations between thread blocks (such as *Atomic*, *Reduce*, and using shared memory as described in Figure 5.2) and matrix-vector multiplication. This suggests the CPU algorithm was already efficient, as our GPU implementation replicates it. Thus, we leveraged this performance benefit of the CPU algorithm in our GPU implementation.

On the other hand, the performance in terms of FLOPs/s is 67 times less than the ideal case (6.7 Tera-FLOPs/s against 0.1 Tera-FLOPs/s). Thus, while our memory accesses are almost optimal, our arithmetic throughput and utilization of SM and memory are low. A possible explanation for this behavior is that many threads remain idle for a considerable time, indicating a synchronization problem. While waiting for synchronization, the throughput is halted, although memory accesses can continue with a high hit rate. This synchronization issue likely belongs to the *atomicAdd* operation. In future work, we aim to improve this synchronization issue.

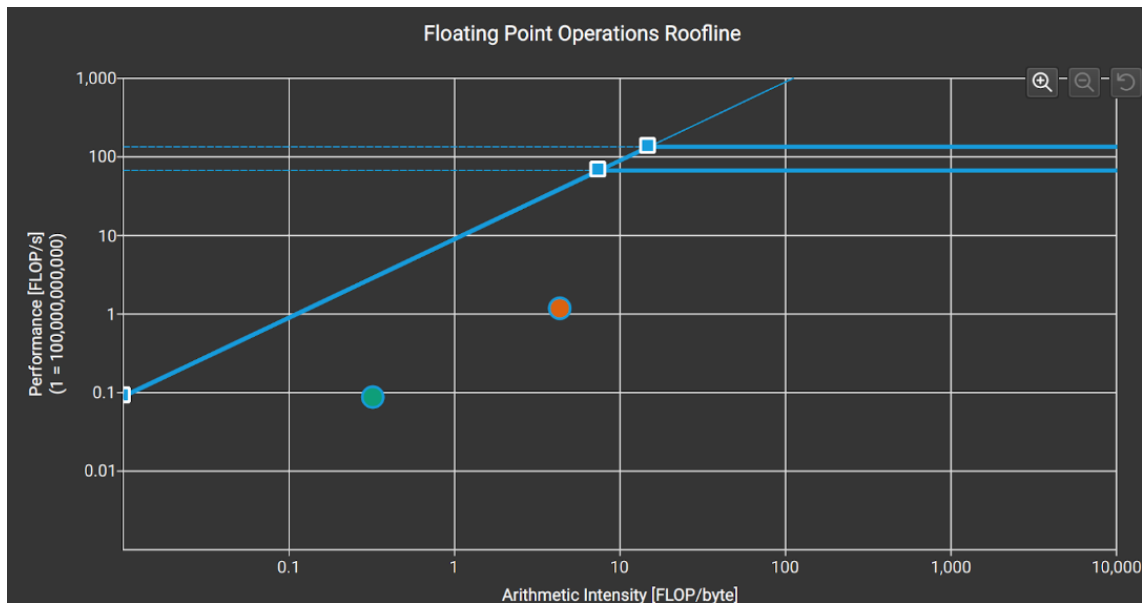


Figure 5.8: Roofline model for first time-step, GPU 0, and process 0 of MONARCH-CAMP. The performance bound is represented as the blue curve. The green and red points represent the performance for single- and double-level floating point operations, respectively.

5.6 Conclusions

This chapter presented an enhanced GPU version of an atmospheric chemistry solver. We utilized the CVODE library for the ODE solver, the CAMP chemistry framework, and the MONARCH atmospheric model.

We extended the Block-cells strategy from the BCG linear solver to the BDF ODE solver, which involves approximately five times more lines of code. The linear solver has already implemented many operations, so we focused on detailing the new additions. These include handling data communications between threads and implementing the *atomicAdd* operation to solve the chemical reactions during the $f(y)$ function.

Our initial results focus on a Box model emulating MONARCH input, facilitating development. The Normalized Root Mean Square Error (NRMSE) is below 0.02%. The speedup achieved in a node-to-node comparison between the GPU and CPU versions is up to 5.9x. The speedup against 1 CPU core is 31x for 1 GPU, similar to the 35x speedup of just the linear solver. This indicates that the new additions slightly reduced the speedup and that the similarities between the linear and ODE solvers are significant. Scaling the test to 4 GPUs shows a linear increase in speedup, reaching an outstanding 137x speedup against 1 CPU core.

We coupled our CAMP GPU version into MONARCH. Adapting from the CPU to the GPU version required only slight code modifications, but the interface between

5.6. CONCLUSIONS

CAMP and MONARCH involved thousands of lines of code, requiring careful adjustments. In our initial experiments with a small domain of 20 cores, we observed an NRMSE of 0.68% and a speedup of 14.7x against 4 GPUs. The NRMSE was notably higher than the Box model for the same configuration, with an NRMSE of 0.02%. However, this error is still acceptable as it remains below 1%. In terms of performance, the results are very similar to the Box model, with a speedup of 14.7x compared to 16.6x. This difference in speedups is due to the increased complexity of the MONARCH simulation compared to the Box model, incorporating factors like the day and night cycle. However, this similarity indicates that the Box model reliably assesses performance without needing to run a complex system like MONARCH.

The NRMSE of the two nodes experiment at 20 and 480-time steps is 0.45%, and 0.42%, respectively. This small variation in NRMSE and the slight decay from 20 to 480 time steps suggests that the NRMSE remains stable as the number of time steps increases.

Additionally, the experiment with the 80-core domain shows an NRMSE of 0.42%, lower than the 0.68% of the 20-core experiment. This confirms that scaling the domain retains a low NRMSE. The expected speedup, after normalizing the 20-core version, is 7x. The node-to-node comparison yielded a 9.8x speedup, higher than expected due to slower CPU performance—a factor we aim to address in future versions.

We want to highlight this 9.8x speedup since it is notably higher than other state-of-the-art modules. For instance, the KPP GPU study reports a 1.75x speedup, and the CAM4-Chem reports 1.95x [35] [36]. This remarks the benefits of the Block-cells implementation.

About profiling, the memory performance, represented as arithmetic intensity, is 4 FLOPs/byte, close to the ideal value of 7. Furthermore, the cache L1 and L2 hit rates are 82% and 99%, respectively, close to the ideal.

In the next chapter, we will show a heterogeneous CPU-GPU version primarily because it can be implemented without significant changes, such as altering the algorithm. Moreover, our GPU version operates under a single kernel call, facilitating the asynchronous execution of CPU code. This implementation was tested on the new Marenstrum 5, which features significant hardware upgrades, such as updating the GPUs from those released in 2017 (Nvidia Volta) to those released in 2022 (Nvidia Hopper).

Chapter 6

Design and Performance Assessment of an Automatic Load Balancing CPU-GPU Algorithm for Atmospheric Chemistry Solvers across Marenostrom 4 and 5

6.1 Introduction

This chapter presents a heterogeneous CPU-GPU implementation where both resources solve atmospheric chemistry simultaneously. The objective is to develop efficient code with minimal changes. To accomplish this objective, this chapter introduces two different load balancing strategies: a Fixed case, where the computational load on the GPU is fixed during execution, and an Automatic implementation, where the computational load between the CPU and GPU is adjusted at runtime. The second strategy is designed to balance load differences during runtime due to the many atmospheric processes involved, such as the day and night cycle. This chapter also presents a load balance metric to quantify the degree of load balance achieved.

Additionally, we provide metrics on the Marenostrom 5 cluster, the next generation of the machine previously used in Chapter 5. Comparing old and new metrics will quantify the advantages of this new architecture for our code, shedding light on which new architectural features could help accelerate this type of computation.

This chapter is organized as follows: Section 6.2 provides the application context relevant to this chapter. Section 6.3 describes the code modifications implemented

for the heterogeneous CPU-GPU computation strategy. Section 6.4 presents our experiments' software configuration and testing environment. Accuracy losses, acceleration, and performance results are discussed in Section 6.5. Finally, Section 6.6 provides concluding remarks.

6.2 Background

This section describes the state-of-the-art and computational description as the starting point before our developments.

6.2.1 State of the art

The state-of-the-art related to GPU chemistry modules is explained in a previous chapter, specifically in Section 1.2. This section details current approaches to implementing GPU computing in chemistry modules and the available GPU tools. For example, it explains why CUDA is used over other parallel languages such as OpenMP or OpenACC.

Section 1.1 defines the motivation behind using MONARCH and CAMP instead of other atmospheric and chemistry modules. Section 2 provides more details about the benefits of MONARCH and CAMP, including a computational description of CAMP in Section 2.2.2.

This chapter continues the development of a GPU ODE solver for atmospheric chemistry, which, in the previous Chapter 5, was deployed in the MONARCH atmospheric model, achieving a speedup of 9.8x in a node-to-node comparison against the CPU version. This was accomplished with minor modifications to the code by following a parallelization strategy called Block-cells [49], where each GPU thread predicts the concentration of a chemical species.

This work aims to continue that effort by implementing an efficient heterogeneous computation strategy, upgrading from a solely GPU-based version to one that runs on both CPU and GPU architectures. The GPU code was designed with this objective, specifically by coding all the GPU processes within a single kernel call. This facilitates the development, continuing the methodology of developing an efficient code with minor changes.

One requisite for efficient implementation is the simultaneous high utilization of CPU and GPU resources. This is complex to achieve mainly because the computational power of both resources is uneven. Additionally, the computational load can substantially differ depending on the experiment configuration, such as the atmospheric region or the chemistry mechanism used in the simulation. Moreover, the load can vary during runtime due to the many changing atmospheric conditions

involved, such as the day and night cycle. Therefore, an efficient implementation requires a runtime configuration that dynamically balances the load.

We aim to achieve an efficient CPU-GPU implementation by balancing the computational load such that the execution times between the CPU and GPU are nearly equal. We explored existing CPU-GPU load balancing algorithms to integrate with our model. Still, the available solutions fall into two extremes: too simplistic to achieve effective load balancing or too complex and challenging to port. A study presents different load-balancing strategies highlighting this dichotomy [121]. The simpler algorithms are tailored for specific cases and lack general applicability, while the more complex solutions require extensive machine learning and OpenCL knowledge and are not open source.

An example of a simple algorithm studied is the Alternate Assignment [122]. In this scheduler, all jobs are added randomly to the job pool. The jobs are then assigned alternately between the CPU and GPU. This policy demonstrates that assigning each processor an almost equal number of jobs without considering the jobs' device suitability results in sub-optimal execution [121].

An example of a complex load-balancing method is the Troodon machine-learning-based approach [121]. The model is trained through multiple benchmark suites <http://developer.amd.com/tools-and-sdks/opencv-zone/amd-8> (a link unavailable as of today), [123] [124] <http://impact.crhc.illinois.edu/parboil/parboil.aspx>. In this scheduler, the jobs are mapped according to their suitability with the CPU and GPU. In our case, it is complex to calculate this suitability due to the many variables involved. In addition, the coupling of this wellspring method is considerably complex because the code is unavailable, as far as we know. However, on average, the method achieves up to 2.82x speedup for a single-node heterogeneous system. This high speedup highlights the potential of a well-balanced load. Thus, the complex code complicates the method in our case, but the high speedup encourages us to aim for an efficient load-balancing strategy.

As a middle point, a study presents a load-balancing strategy where the computational workload is organized into a grid, similar to the grid cells in our implementation [33]. This study introduces a metric to balance the load across multiple MPI processors and to enable CPU-GPU co-execution of different tasks. However, the focus of this algorithm is limited to balancing CPU resources without addressing GPU load balancing. Additionally, the algorithm is not open source and relies on complex concepts like *weighted linear regression*, which are only briefly defined. Due to these complexities and limitations, we find it more practical to develop our load-balancing algorithm rather than attempting to port this method.

In our context, we can speculate on the impact of manually distributing the load. Given that the chemical mechanism remains consistent across time steps, the computational load should also remain similar. The primary factors influencing load variability are environmental variables such as temperature and reaction rates, which

fluctuate significantly from day to night. Consequently, some variation in load distribution is expected over a daily cycle.

To address these dynamic changes, we propose an automatic load-balancing algorithm that adjusts in run time. Our objective for this algorithm is to ensure ease of portability across different modules while reserving more complex optimizations for future work. Nonetheless, we expect this algorithm to outperform static load distributions. Furthermore, this algorithm will benefit experiments involving different domain regions and sizes, as the optimal fixed load distribution may vary. Our goal is for the algorithm to identify the optimal load distribution within a few time steps.

6.2.2 Computational description

Our base code is the GPU Block-cells version of CAMP from Chapter 5. This version utilizes a single kernel to execute the entire GPU-solving code. This approach simplifies development compared to using multiple kernels, such as those calling cuBLAS routines or small OpenACC kernels mixed with CPU code. We streamline the development process and reduce complexity by isolating the GPU code from the CPU code.

Figure 6.1 summarizes the concept of running both architectures in parallel. However, we must define the *data* distribution between the GPU and CPU.

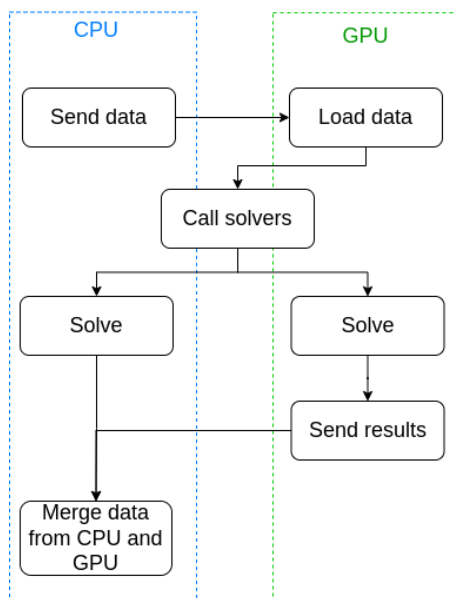


Figure 6.1: Workflow of the CPU-GPU implementation. *Data* refers to the atmospheric data containing chemical concentrations, temperature, pressure, and reaction rates. *Results* refer to the solved concentrations of the chemical species.

The *data* we refer to are atmospheric cells to solve. These independent cells allow us to divide them freely between the GPU and CPU. For example, we can distribute 70% of the cells to the GPU and 30% to the CPU. For simplicity, we will refer to the cells as a load when referring to the percentage of cells that a GPU or CPU computes.

6.3 Implementations

6.3.1 Porting

The CPU code differs slightly from the base CPU version. In the base version, the host model calls the CAMP-solving routine once for each cell. In the CPU-GPU case, the host model calls the CAMP-solving routine once, sending a pointer with all the cells. Then, the cells are computed sequentially inside the CAMP-solving routine just after the call to the GPU kernel. Below, we show a pseudo-code of the resulting implementation:

```
factor_of_cells_to_gpu=0.7; //70%
solve(cells){
/* Solve GPU asynchronously*/
n_cells_gpu=n_cells*factor_of_cells_to_gpu;
send_cells_to_gpu(cells_pointer,n_cells_gpu);
solve_gpu(cells_pointer,n_cells_gpu);

/* Solve CPU */
n_cells_cpu=n_cells*(1-factor_of_cells_to_gpu);
cells_pointer_cpu=update_cells_pointer_to_first_cell_cpu(
    cells_pointer,n_cells_cpu);
for (i_cell = n_cells_gpu; i_cell < n_cells; i_cell++){
    solve_CVODE(cells_pointer_cpu)
    update_cells_pointer_to_next_cell_cpu(cells_pointer_cpu)
}

/* Merge GPU and CPU data */
receive_data_from_GPU(cells_pointer,n_cells_gpu);
wait_for_GPU_to_finish();
}
```

As a technical note, *cells* consist of multiple arrays, including temperature, pressure, reaction rates, and concentrations. Therefore, updating *cells_pointer* refers to updating the pointers of these arrays.

In the updated implementation, we have decoupled the time-step size calculation from the CVODE library. Previously, CVODE was used to create structures that grouped data from all cells. For the CPU-GPU version, we separated these structures so the GPU handles data for all cells, while the CPU uses CVODE to manage data for a single cell. This simplifies the code and facilitates a possible deployment as an independent library of the GPU solver.

To ensure consistent performance between the CPU-only and the CPU part of the CPU-GPU version, we utilized the Extrae profiler (version 4.2.1) <https://tools.bsc.es/extrae> [125]. Specifically, we extracted the IPC (Instructions Per Cycle) metric, referred to by Extrae as *Useful IPC*, which measures the execution of instructions, excluding those related to waiting or data transfer, such as MPI_WAIT. The experiments were conducted with multiple MPI processors, and since the load was evenly distributed, the IPC remained consistent across all processors. The IPC was averaged over the ODE-solving execution, which corresponds to the function that calls the external CVODE solver <https://computing.llnl.gov/projects/sundials/cvode> [47]. The results showed that the IPC of the CPU part of the CPU-GPU version was identical to that of the CPU-only version, with a value of 4, indicating good performance. This consistency is expected since the CVODE routine remains unchanged between versions. Future work will focus on further detailing and analyzing the performance metrics of the CPU solver to enhance its efficiency.

We also ported part of the solving routine from the CPU to the GPU. Figure 6.2 illustrates the workflow differences between the previous and current implementations, while the diagram representing the state before this thesis work is shown in Figure 2.8. In the last work, most of the ODE solver from CVODE was translated to run on the GPU, while the remaining CPU parts corresponded to calculating the reaction rates and the time-step size. The reaction rates are part of the CAMP library, already prepared for CPU-GPU implementation. Precisely, the reaction rates are calculated for all the cells on the CPU side before any call to the solver.

In the current work, the time-step size update was moved to the GPU, whereas in the CPU code, this update remains unchanged. In the previous work, we deferred the porting of this routine for two reasons: it involves hundreds of lines of code, which seemed to require significant development effort, and its execution time was relatively low compared to the other components of the ODE solver.

Notably, this development was less complex than anticipated because most of the time-step size calculation code was tied to conditionals that were never triggered in the CAMP configuration. For example, most of the code was only executed after the first iteration of the ODE solver. However, CAMP resets the solver state at each iteration, making each time step akin to the first call of the solver. Thus, this part of the code was not integrated into the GPU code, simplifying the porting.

Porting this code to the GPU reduces the data transfer between the CPU and GPU by 4x. Previously, 13 arrays were transferred, each with a size equal to the

6.3. IMPLEMENTATIONS

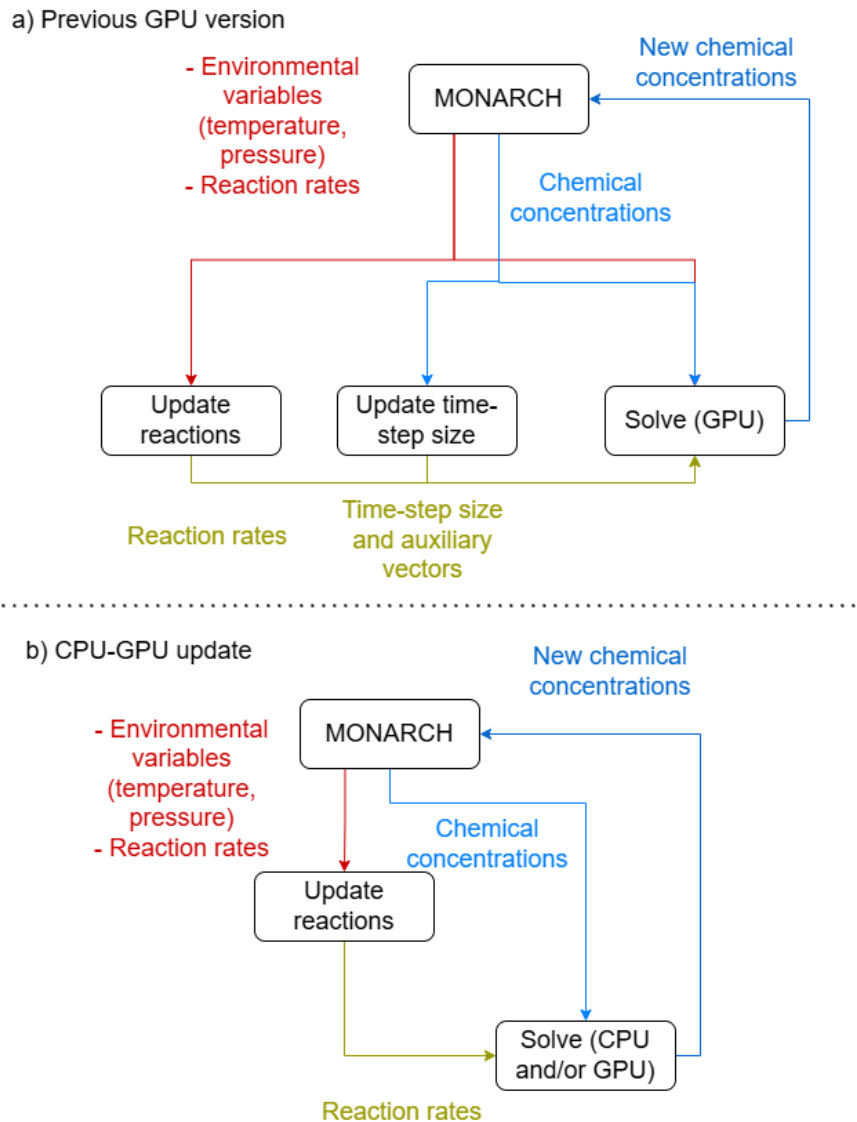


Figure 6.2: Workflows between MONARCH and the CAMP solver for the previous GPU version (a) and the new CPU-GPU implementation (b).

concentrations to solve. Ten of these arrays correspond to auxiliary variables of the ODE solving. Specifically, the input data required for the ODE solving is two arrays corresponding to concentration and environmental variables. The output data is the updated array of concentrations. However, during the routine of updating the time-step size, ten arrays were updated with the values of the initial concentration array. Thus, moving the time-step size calculation to the GPU eliminates the need to transfer these ten arrays, simplifying the implementation and enhancing performance by reducing data transfer overhead between the CPU and GPU.

We also changed the CSC (Compressed Sparse Column) Sparse structure for a CSR

(Compressed Sparse Row) structure. This gives more stable concentrations due to avoiding atomic operations, which have an element of randomness. Also, the execution time was around 4% better.

6.3.2 Load balance

We define the load balance as:

$$LoadBalance[\%] = Avg(100 * \frac{min(timeGPU, timeCPU)}{max(timeGPU, timeCPU)})$$

Where $timeCPU$ corresponds to the time taken by the CPU-related code, starting from the completion of the GPU solver call to the point where the CPU waits for the GPU to finish. $timeGPU$ corresponds to the time of the data transfers between CPU and GPU, kernel, and waiting time for the GPU to finish. As a side note, two timers are required to measure $timeGPU$. This is because the CPU code is located between the kernel call and the data transfers from the GPU to the CPU. The Avg represents the average load balance across different time steps.

We define the percentage of computational load handled by the GPU as $LoadGPU$. Similarly, the percentage of computational load managed by the CPU is denoted as $LoadCPU$, and it is calculated using the formula.

$$LoadCPU = 100 - LoadGPU$$

We define the metric $ShortGPU$ to determine whether the execution time on the GPU is shorter than the CPU time. $ShortGPU$ is defined as

$$ShortGPU = \begin{cases} True, & \text{if } timeGPU < timeCPU \\ False, & \text{otherwise} \end{cases}$$

Automatic load balance

To explain the automatic load balance, we start with a simple example. Suppose we set $LoadGPU$ to 95%, resulting in a $LoadBalance$ of 20% and $ShortGPU$ being $True$. This indicates that we are 80% away from the ideal load balance of 100%. We can define this distance from the ideal case as:

$$RemainingLB = 100 - LoadGPU$$

We want to reduce the $LoadGPU$ to increase $LoadBalance$ closer to the ideal 100%. Reducing $LoadGPU$ by another 5% (from 95% to 90%) would increase the $LoadBalance$ by at least 20%. However, we want to reach 100% as fast as possible.

6.3. IMPLEMENTATIONS

Thus, we reduce *LoadGPU* by more than 5%, for example, 7.5%. We can define the new *LoadGPU* as:

$$LoadGPU_{t+1} = LoadGPU_t + IncreaseLG$$

Where t represents the current time step, $t + 1$ is the next step, and *IncreaseLG* is the amount by which *LoadGPU* is reduced, 7.5% in this example.

Suppose the current *LoadBalance* rises to 80% after advancing a time step. Thus, the *LoadBalance* has increased by 60% (from 20% to 80%). We can define this increase as:

$$DiffLB = LoadGPU_t - LoadGPU_{t+1}$$

Then, the *RemainingLB* is 20%. Since the previous *IncreaseLG* increased the *LoadBalance* by 60%, we want to reduce that. For example, we can use 7.5% instead of 3.25%. Therefore, we must define how to identify whether to increase or decrease *IncreaseLG*. This distinction proceeds as follows:

$$IncreaseLG_{t+1} = \begin{cases} IncreaseLG_t * 1.5, & RemainingLB > DiffLB \\ IncreaseLG_t/2, & otherwise \end{cases}$$

The multiplication factor *IncreaseLG* must be smaller than the division factor to ensure stable convergence of the load balance adjustments. If these factors were equal, the load balance could oscillate between two values, leading to a perpetual adjustment cycle without achieving convergence. For example, if both factors are equal, the *LoadBalance* might increase by 2 in one iteration and decrease by 2 in the next, resulting in no net progress towards a stable load balance. By setting *IncreaseLG* to be smaller than the division factor, the adjustments are made more gradually, allowing the load balance to move smoothly and progressively toward an optimal state, thus ensuring a stable convergence process.

If the variable *ShortGPU* changes from *True* to *False*, indicating that the CPU is now faster than the GPU, the calculation of *DiffLB* should account for the change in the computational load balance. The formula for *DiffLB* in this situation is:

$$DiffLB = 100 - LoadGPU_{t-1} + 100 - LoadGPU_t$$

In addition, the *IncreaseLG* must change the sign since we want to increase the *LoadGPU* instead of decreasing it more.

We implement constraints in our adjustment algorithm to prevent the GPU load percentage (*LoadGPU*) from exceeding practical limits, such as 100% or dropping to 0%.

Here is the full algorithm in pseudo-code:

6.4. TEST ENVIRONMENT

```
/* Set if GPU time is less than CPU */
shortGPU=0;
if(timeGPU<timeCPU) shortGPU=1;

/* Set how much the load balance has increased */
DiffLB=LoadBalance-LastLoadBalance;
if(shortGPU != LastShortGPU){
    DiffLB=100-LastLoadBalance+100-LoadBalance;

/* Change the increase sign because we surpass the limit of 100%,
swapping from one architecture being short in time to the other */
    IncreaseLG*=-1;
}

/* Set the remaining load balance to reach the ideal case of 100% */
RemainingLB=100-LoadBalance;

/* Set the Increase in Load GPU */
if(RemainingLB > DiffLB) IncreaseLG*=1.5;
else IncreaseLG/=2;

/* Update values for next iteration */
LastShortGPU=shortGPU;
LastLoadBalance=LoadBalance;
LastLoadGPU=LoadGPU;
if(LoadBalance!=100) LoadGPU+=IncreaseLG;

/* Avoid the GPU percentage reaching or exceeding 100\% and 0\% */
if(sd->load_gpu>99) sd->load_gpu=99;
if(sd->load_gpu<1) sd->load_gpu=1;

/* Set the amount of load to GPU */
nCellsGpu=nCells*LoadGPU/100;
```

6.4 Test environment

All the tests were performed on the cluster Marenstrum 5 ACC (Accelerated partition) provided by the Barcelona Supercomputing Center [126]. The detailed hardware specifications of each node are described below.

- 2x Intel Sapphire Rapids 8460Y+ at 2.3Ghz and 32c each (64 cores node)
- 4x Nvidia Hopper GPUs with 64 HBM2 memory

6.4. TEST ENVIRONMENT

- 512 GB of Main memory, using DDR5
- 4x NDR200 (BW per node 800Gb/s)

In addition, we use the profiler Nsight Compute Version 2023.2.1.0 to visualize the profiling data of the GPU experiments and assess the performance of the tests under analysis. When profiling, the chemistry time steps are set to one. The remaining time steps perform very similarly to the Box model simulations, which is expected since the operations carried out by the ODE solver are similar between time steps.

We compare the performance metrics with the results from the previous GPU-only version presented in Chapter 5, which was executed on the GPU partition of the Marenostrom 4 cluster, MN4 CTE-POWER <https://www.bsc.es/marenostrom/marenostrom/technical-information>. For simplicity, we refer to this cluster as Marenostrom 4, which is consistent with the updated GPU version named Marenostrom 5. This comparison allows us to evaluate the performance effects resulting from the different architectures.

To measure accuracy between the CPU and GPU versions, we use the Normalized Root Mean Square Error (NRMSE), represented as a percentage and calculated as

$$\text{NRMSE of a chemical specie} = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - y_i)^2}}{x_{\max} - x_{\min}}$$

where x is the concentration of a chemical species from cell i , obtained from the last time-step of the model execution and the Base version of CAMP, y is the equivalent of x for the optimized version, n is the number of cells, and x_{\max} and x_{\min} are the maximum and minimum value from x . In addition, while we refer to the Error of an experiment, we refer to the maximum NRMSE between the chemical species, that is:

$$\text{Error} = \max\{NRMSE_{s_1}, \dots, NRMSE_{s_N}\}$$

$NRMSE_{s_1}$ refers to the NRMSE of the first chemical species s_1 , and s_N refers to the last chemical species.

The absolute tolerance of the CVODE solver is set to $1.0e^{-4}$. Any error of accuracy below this level of tolerance is considered negligible. The KLU solver uses the same tolerance. The tolerance of the BCG linear solver is set to $1.0e^{-30}$. Any negative value greater than $-1.0e^{-30}$ produces an extra iteration in the CVODE-solving algorithm.

6.4.1 Box model experiments

During the porting development, we employed a simplified configuration called the "Box model," which emulates the input from MONARCH. This approach was also

utilized in the previous GPU version discussed in Chapter 5. The Box model yielded performance results comparable to MONARCH, with an accuracy error approximately 0.5% smaller. This configuration allows us to work with smaller setups, such as a single CPU process or one cell, which facilitates development and testing. This is advantageous compared to MONARCH, which does not support such granular configurations.

Here, we configure the test to get the speedup against the single-thread CPU core. This is useful to compare other state-of-the-art studies that do not report a node-to-node comparison, such as RKC [36].

- Time-step size: 120 seconds
- Number of time steps: 720
- Number of cells: 100,000.
- Threads per block: 86
- Pressure: Scales linearly with the number of cells from 1000 to 100 hPa. Emissions are also scaled linearly from 1 to 0. In this way, a cell at 100 hPa has 0 emissions, while a cell at 1000 hPa has the maximum emissions value.
- Temperature: Follows dry adiabatic conditions [110].

Configuration for comparison with Marenostrom 4

We find it interesting to compare the GPU-only version of Marenostrom 5 to assess the impact of the newer architecture. This version utilizes the GPU code from Marenostrom 4, with updated compilation scripts to run on Marenostrom 5 <https://github.com/open-atmos/camp/tree/GPUonly>. This configuration sets the number of cells to 10,000, with time-steps configured to 1 and MPI processors to 1 since we only do profiling. Using 1 MPI processor instead of a full node is faster to profile, while increasing the number of processors shows less than a 2% difference in performance metrics. The Box model configuration demonstrates performance and speedup comparable to MONARCH as shown in Chapter 5. Consequently, the results from this Box model on Marenostrom 5 are expected to be similar to those from the MONARCH case on Marenostrom 4.

We also included the CPU-GPU version in the comparison. However, many changes in the porting process could affect the performance, such as incorporating additional code on the GPU, cleaning up variables, or using the CSR Sparse structure instead of CSC. A detailed investigation of these results will be deferred to future work, mainly because these aspects offer potential avenues for optimization.

6.4.2 MONARCH experiments

The MONARCH experiment is configured similarly to the setup described in Chapter 5, covering the regional domain of Europe and northern Africa. In the previous chapter, we evaluated two configurations of this domain: one representing the whole domain and a smaller version of the domain. For this chapter, the smaller configuration is used to facilitate development.

Additionally, we employed both configurations to assess the linear scalability of speedup concerning MPI processes and GPUs. The results indicated that speedup approached linearity, although a gap remained due to the CPU solver’s unexpected behavior, which required more iterations for larger configurations, thereby slowing simulations. With this factor accounted for, we anticipate that speedup should be linear, as demonstrated in the Box model configuration.

Additionally, scaling the number of nodes should proportionally scale CPU and GPU resources. Consequently, a single-node configuration should exhibit similar speedup characteristics to a two-node configuration for CPU and CPU-GPU versions. This approach also streamlines the profiling task and allows us to utilize the faster queue of Marenostrom 5, thereby accelerating development. Furthermore, it simplifies the comparison by focusing on a single-node setup, making it easier for readers to understand node-to-node performance. Therefore, in this chapter, we have replaced the 20-core and two-node configurations with a single node featuring 80 cores. This adjustment simplifies the analysis by concentrating on a single configuration, expecting that scaling to larger runs with additional nodes will follow a linear trend from the single-node results.

We tested different numbers of chemistry time steps: 6 and 480. The 6-time-step configuration detects and resolves early issues and determines the optimal value for *LoadGPU*. Once this value is established, we use the 480-time-step configuration to evaluate fixed and automatic load balancing. This extended configuration assesses whether the results are consistent over multiple time steps. In the previous approach of Chapter 5, the accuracy error remains acceptably low for the 480-time-step configuration. Therefore, we expect a similar result here. However, the performance of the automatic and fixed configurations may vary due to run-time processes, such as the day and night cycle, which affect solving complexity and result in different computational loads.

Load balance configuration

To obtain the execution time for the fixed load balance, we exclude the measurements of *timeCPU* and *timeGPU* from the execution. This approach provides a more accurate comparison against the automatic load balance, which relies on these metrics. Additionally, we performed an extra run with these timers to calculate the *LoadBalance* metric.

For the automatic load balance, we explore starting values of *LoadGPU* ranging from 84% to 98%. This range helps identify the optimal value and observe how the speedup varies with different *LoadGPU* settings. The best value found within this range is then used for extended runs.

6.5 Results and discussion

6.5.1 Box model runs

Speedup against 1 CPU core

Table 6.1 shows that in Marenostrom 5, we have doubled the speedup of a single GPU compared to the Marenostrom 4 version (from 31x to 70x). The speedup scales linearly with the number of GPUs. In a node-to-node comparison, the CPU-GPU version achieves a speedup of 250x, compared to 137x for the GPU-only Marenostrom 4 version. We consider this a significant improvement.

Version	N ^o of GPUs	Speedup vs 1 CPU Core
Marenostrom 5	1	70
Marenostrom 5	4	250
Marenostrom 4	1	31
Marenostrom 4	4	137

Table 6.1: Speedup using 1 and 4 GPUs against 1 CPU core. The 1 GPU case uses 20 cores for the CPU execution, while the 4 GPU case uses 80. The GPU computes 95% of the load with automatic load balance, which is the best value between other simulations with values from 84% to 9%.

Profiling

Figure 6.3 shows that Marenostrom 5 increases the utilization of SMs and Memory by 5% and 28%, respectively, compared to Marenostrom 4. We attribute this improvement to the higher capabilities of the Marenostrom 5 GPU, which has more SMs and Memory available to handle the high computational load. The figure also indicates that memory utilization decreases by 10% when comparing the GPU-only version to the CPU-GPU version.

Figure 6.4 shows that Marenostrom 5 exhibits approximately 9x times more Performance and 6x higher Arithmetic Intensity than Marenostrom 4, underscoring the advantages of the newer GPU. However, the Arithmetic Intensity decreases by 3x

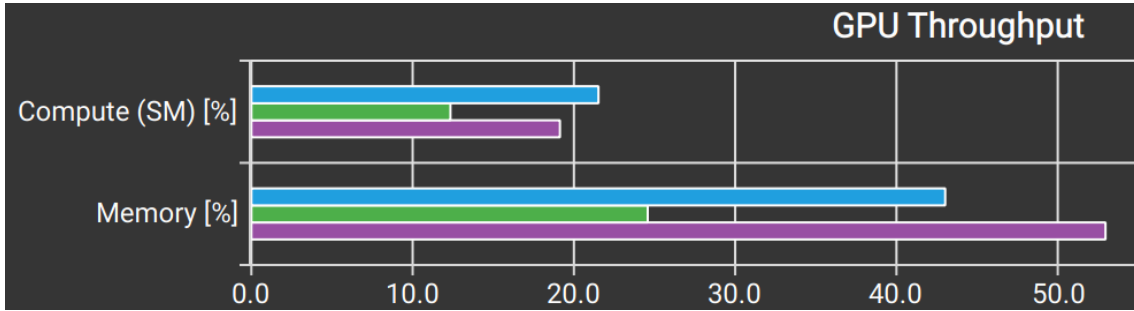


Figure 6.3: Utilization of GPU resources. The first (Blue), second (Green), and third (Purple) bars correspond to the versions CPU-GPU Marenostrom 5, GPU-Only Marenostrom 4, and GPU-only Marenostrom 5.

in the CPU-GPU version compared to the GPU-only version. This indicates that there is potential for further improvement. Identifying the cause of this reduction amid all the code changes would require a detailed investigation. As this chapter focuses on developing a CPU-GPU version with manageable effort, we defer this investigation and additional optimizations to future work. The cache L1 and L2 hit rates improved from 82% and 99% in the GPU-Only version to 92% and 99%, respectively. This enhancement is attributed to using the CSR sparse structure instead of the CSC structure in the GPU-only version. The transition to CSR was primarily made to achieve more stable results by avoiding *atomicAdd* operations. These improved cache hit rates further support the use of CSR.

6.5.2 MONARCH model runs

Speedup and load balance in MONARCH

Table 6.2 shows that *LoadBalance* varies slightly after the first time-step, with a difference of 10% between the best and worst values. Therefore, the load balance should remain consistent over a longer run, barring significant changes in reaction rates, such as the transition from day to night. Even with these changes, the load balance should stabilize rapidly. The table reveals that the automatic configuration achieves nearly three times better load balance than the fixed case. This suggests that the optimal load balance is not well represented by the fixed settings tested, and the automatic configuration is closer to this optimal value.

Table 6.3 shows that the optimal configurations for both fixed and automatic load balancing are achieved at 85% and 95% GPU load, resulting in speedups of 5.13 and 4.71, respectively. In short runs, such as those with six-time steps, the fixed load balance yields a speedup that is 1.09 times faster compared to the automatic load balance. Additionally, the figures demonstrate that the error decreases as the load balance improves. This is expected because a more significant portion of the computation is offloaded to the GPU, similar to the original CPU-only version. The

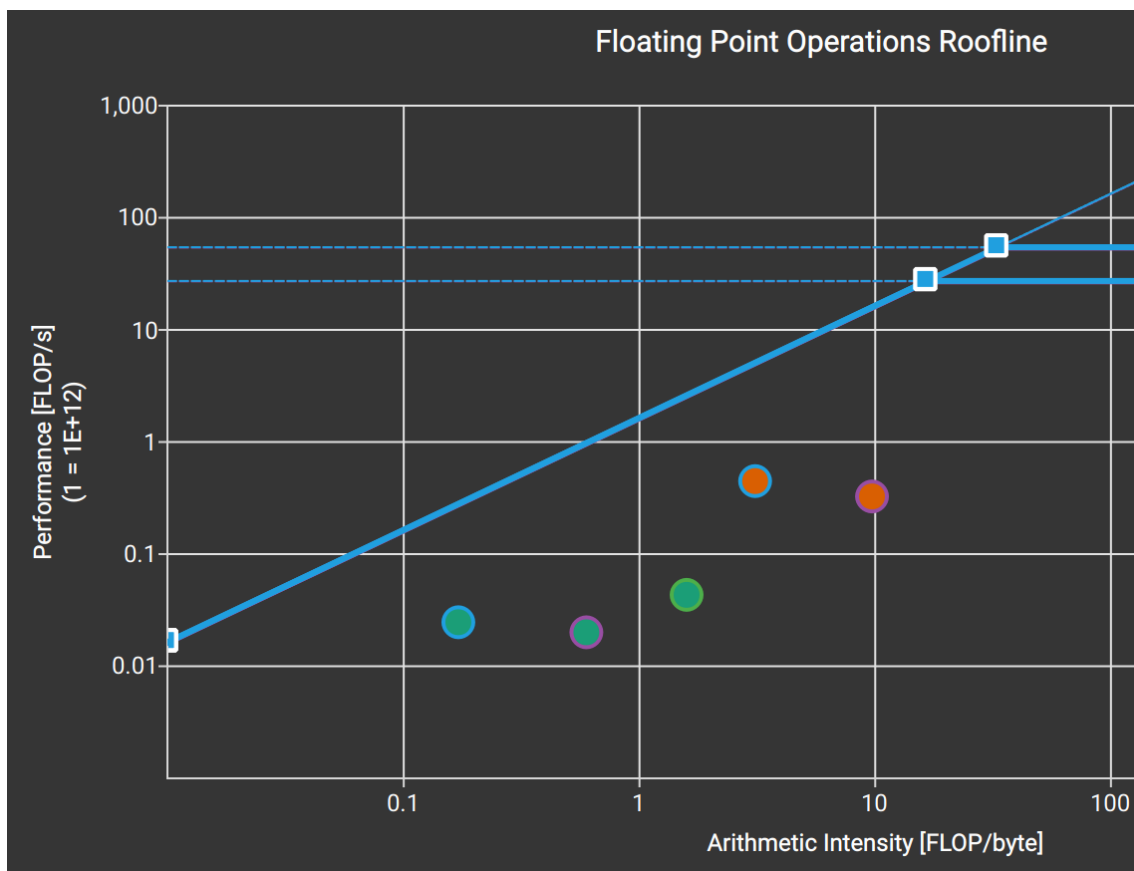


Figure 6.4: Roofline model. The performance bound is represented as the blue curve. Starting from the left, the first and second points are single floating point operations, while the rest are double. The third, fourth, and fifth points correspond to versions GPU-Only Marenostrom 4, CPU-GPU Marenostrom 5, and GPU-Only Marenostrom 5. We are interested in doubles since chemical concentrations are represented in double.

Time-step	Load balance [%]	
	(not averaged) - Fixed 95%	(not averaged) - Automatic 95%
1	16	15
2	30	87
3	27	84
4	28	86
5	28	85
6	29	84

Table 6.2: Load balance evolution between time steps.

automatic load balance configuration also improves the load balance by a factor of 1.12 compared to the fixed load balance. Thus, while a fixed load balance is

preferable for short runs due to its higher speedup, it is worth noting that these short runs are primarily for debugging and performance estimation purposes. In the long run, the performance and load balance might differ.

Load to GPU [%]	Speedup	Error	Load balance [%]
100	4.29	0.5	0
98	4.44	0.49	10
95	4.59	0.48	28
90	4.84	0.47	56
85	5.13	0.46	87
84	5.07	0.45	87
Automatic (90)	4.37	0.47	73
Automatic (95)	4.71	0.48	77
Automatic (98)	4.7	0.49	60

Table 6.3: Speedup, error, and load balance with different fixed and automatic loads to GPU. The time steps are set to 6. The number after "Automatic" refers to the starting *LoadGPU* value.

As a side note, the speedup for the Fixed 85% configuration when measuring the load balance is 6.22x, slightly lower than the 6.51x speedup observed without these measurements. This indicates an overhead of approximately 5% due to the load-balancing calculations.

Table 6.4 shows that the automatic load balance achieves a speedup of 8.14x, 1.25 times faster than the fixed configuration. Interestingly, despite the higher speedup, the load balance in the automatic case is 3% lower compared to the fixed load case. This suggests that the average load balance does not directly correlate with long-run speedup. Consequently, exploring alternative metrics for performance evaluation in future work would be worthwhile.

Load to GPU [%]	Speedup	Error	Load balance [%]
Fixed	6.51	0.56	61
Automatic	8.14	0.59	58

Table 6.4: Speedup, error, and load balance for GPU's best fixed and automatic loads. The fixed *LoadGPU* is set to 85% for the Fixed case, while the Automatic case is initially set to 95%. The time steps are set to 480.

In the previous GPU-only Marenstrum 4 version, the node-to-node speedup was measured at 9.8x; see Chapter 5. However, this result was partly influenced by the CPU version experiencing a slowdown due to numerous convergence failures. In the current experiment, no convergence failures were observed. Consequently, a fairer comparison is with the previous normalized speedup of 8x from the 20-core experiment, where no convergence failures occurred.

Thus, the speedup of 8.14x is 1.16 times greater than the previous GPU-Only Marenostrom 4 version, Chapter 5. This improvement may appear modest compared to the 2x speedup observed against a single core in the new configuration from Table 6.1. The discrepancy arises because Marenostrom 5 nodes have doubled the number of MPI processes compared to Marenostrom 4 while maintaining the same number of GPUs. This doubling of MPI processes results in a 2x acceleration for the CPU version due to the increased parallelism in computation. When normalizing for the number of GPUs, the speedup relative to the GPU-Only Marenostrom 4 version is 2.32x. This enhancement reflects the improvements in GPU performance and the effectiveness of our CPU-GPU implementation. Consequently, we can consider our objective of reducing execution time through the CPU-GPU load balance a success.

Profiling MONARCH

Figure 6.5 illustrates that the CPU-GPU version on Marenostrom 5 achieves 5% more utilization of SM and Memory than the GPU-only version on Marenostrom 4. However, it is notable that memory usage decreases by 23% from the same CPU-GPU version on the Box model, as shown in Figure 6.3. We attribute this reduction to the increase in threads, from 86 to 127. Even though these additional threads are idle, the compiler allocates memory registers. Adjusting this distribution could involve significant effort due to the complexity of the CAMP-MONARCH structure, which comprises thousands of lines of code. Consequently, we plan to explore this optimization in future work.

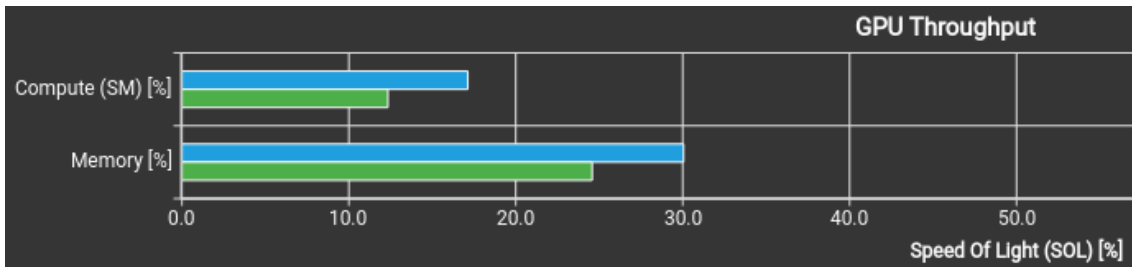


Figure 6.5: Utilization of GPU resources. The upper (Blue) and lower (Green) bars correspond to the versions CPU-GPU Marenostrom 5 and GPU-Only Marenostrom 4.

Figure 6.6 demonstrates that the CPU-GPU version on Marenostrom 5 achieves approximately 9x times higher performance and 2x times greater Arithmetic Intensity than the GPU-only version on Marenostrom 4. Notably, these results for the CPU-GPU configuration are consistent with the Box model results shown in Figure 6.4. This consistency suggests that the Box model effectively represents the GPU performance of MONARCH.

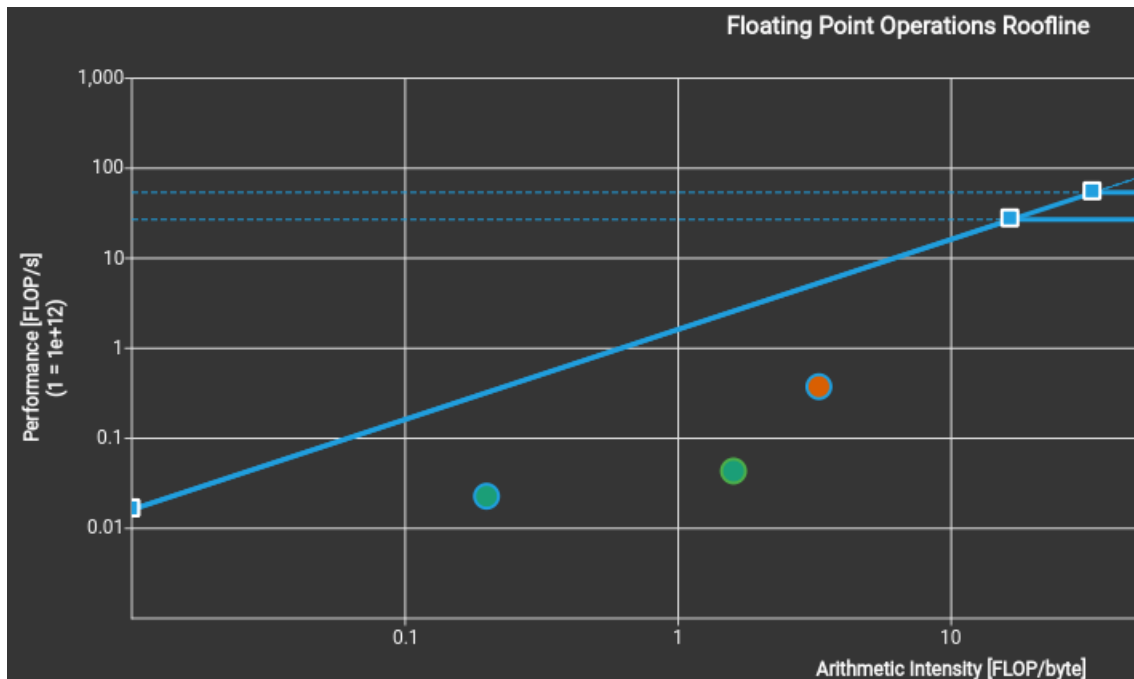


Figure 6.6: Roofline model. The performance bound is represented as the blue curve. Starting from the left, the first point represents single floating point operations, while the rest are double. The second and third points correspond to GPU-only Marenostrom 4 and CPU-GPU Marenostrom 5 versions. We are interested in doubles since chemical concentrations are represented in double.

The cache L1 and L2 hit rates match those observed in the Box model case, showing improvements from 82% and 99% in the previous version to 92% and 93%, respectively.

In summary, the utilization, performance, and cache hit rates have all improved compared to the previous version, highlighting the advantages of the newer architecture and our CPU-GPU implementation. However, there is still room for improvement in the CPU-GPU version. Despite high cache hit rates, the low utilization suggests potential synchronization issues, possibly due to atomic operations. Future work will focus on enhancing utilization and performance.

6.6 Conclusions

This chapter presented a heterogeneous CPU-GPU atmospheric chemistry solver and an automatic load-balancing algorithm. The main goal has been to achieve an easy-to-port and efficient implementation. We utilized the GPU and CPU solvers of the CAMP chemistry framework and integrated them into the MONARCH atmospheric model.

We merged the GPU and CPU solvers into an asynchronous execution, allowing the GPU and CPU to compute the workload concurrently. During this merge, we completely separated the CVODE library from the GPU execution, thereby reducing the amount of code and simplifying the GPU library’s porting process for future developers.

Moreover, we ported additional CPU code to the GPU, specifically the time step size calculations from the solving algorithm. We removed unused variables and reduced the amount of data transferred between CPU and GPU by 4x times. Additionally, we changed the sparse storage structure from CSC to CSR. These changes collectively reduce the execution time.

Then, we presented an automatic load-balancing algorithm, which adjusts the load for each chemistry time step based on the distribution from the previous time step. The algorithm is designed to stabilize at the optimal load balance, where the CPU and GPU times are nearly equal. The speed of achieving stabilization depends on the initial load value assigned to the GPU. The algorithm’s code is open source and compressed into less than a hundred lines, facilitating future porting to other software.

We configured experiments to evaluate the performance of our load-balancing algorithm against a fixed load value for all the time steps (Fixed and Automatic implementations). The hardware used is the recent Marenostrom 5 cluster. This cluster represents a significant update from previous results, where a GPU-Only version was presented on Marenostrom 4 (see Chapter 5). Therefore, we also set up a GPU-only version on Marenostrom 5 to assess the performance impact of this newer cluster by comparing the GPU-only version on Marenostrom 5 with the previous results on Marenostrom 4.

Our initial results focus on a Box model emulating MONARCH input. The best case is the automatic load balance, starting at 95% of the load to the GPU. This case reports a 70x speedup against the single-thread CPU version using a single GPU. Increasing the GPUs to 4, the speedup scales to 250x. This speedup doubles the previous results from the GPU-Only Marenostrom 4 version.

Additionally, the Marenostrom 5 GPU-Only version shows a 5% and 28% increase in SM and Memory utilization, respectively, compared to the Marenostrom 4 version. It also exhibits a 9x and 6x improvement in Performance and Arithmetic Intensity metrics. These results underscore the advantages of the newer architecture.

Compared to the GPU-Only Marenostrom 5 version, the CPU-GPU version shows a 10% decrease in memory utilization and a 3x reduction in Arithmetic Intensity. The cause of this decline is attributed to the numerous code changes implemented, which will be investigated further in future work to enhance utilization. However, improvements are seen in the cache L1 and L2 hit rates, which increased from 82% and 99% to 92% and 99%, respectively, due to the change from the CSC to the CSR structure.

6.6. CONCLUSIONS

The MONARCH results reveal a stable load balance (with only a 10% deviation) after the first time step for the subsequent five steps. This stability suggests that the fixed load balance would likely outperform the automatic load balance in shorter runs. Specifically, the best-fixed configuration, with 85% of the load assigned to the GPU, achieves a 5.13x speedup. In comparison, the best automatic configuration, with 95% load to the GPU, results in a 4.71x speedup. Thus, the fixed load balance is 9% faster than the automatic case for very short runs.

The 1-day MONARCH results indicate that the automatic load balance outperforms the fixed load balance by 25%, achieving a total speedup of 8.14x. Interestingly, despite this increased speedup, the load balance metric is 3% lower than that of the fixed case, suggesting that the load balance may not directly correlate with improved performance. This discrepancy highlights the need to explore alternative metrics to understand performance dynamics better. Future work will focus on identifying and analyzing these metrics to optimize the load-balancing approach further.

The presented CPU-GPU version of MONARCH demonstrates a 2.32x improvement in speedup over the previous GPU-Only Marenostrom 4 version. This notable enhancement underscores the advantages of the heterogeneous implementation, particularly with optimizations such as reduced CPU-GPU data transfers. The high speedup achieved with our streamlined algorithm reflects our success in meeting the objectives of an efficient and straightforward implementation.

Chapter 7

Conclusions

This thesis presented the porting of an atmospheric chemistry solver to parallel execution on CPUs and GPUs. We used the CAMP chemistry solver and the MONARCH atmospheric model to assess the effectiveness of our adaptations. The main conclusions and recommendations for future research are summarized in Sections 7.1 and 7.2.

7.1 Main findings

In Chapter 3, we introduced Multi-cells, a data arrangement designed to facilitate the integration of GPU functions into a chemistry solver. In this approach, atmospheric cells are grouped and treated as a single system for solving. We tested this method using a simple chemical mechanism involving three species, where species *A* generates species *B* and *C*, with minor variations in initial concentrations across the cells. This approach reduced the number of iterations required to solve all cells to match the number needed to solve a single cell. This optimization resulted in an 8× speedup compared to the base version, which solves cells independently and sequentially in a loop.

Additionally, we developed a CUDA version of the most time-consuming function, the $f(y)$ function, by parallelizing its reaction loop across GPU threads. We enhanced data access by reorganizing the reaction data structure, resulting in a 1.3x acceleration of the GPU function. Overall, the optimized GPU function achieved a 1.6x speedup compared to the single-threaded CPU version.

In Chapter 4, we introduced Block-cells, a novel approach to distributing the computational load in a GPU-based chemical solver. In this method, each GPU thread computes the concentration of a species within a cell, thereby increasing the level of parallelization beyond traditional implementations, which typically assign one thread per cell. This approach significantly enhances the utilization of the GPU's

7.1. MAIN FINDINGS

high bandwidth capacity. We tested Block-cells using the CB05 chemical mechanism, commonly used in atmospheric models, under varying and identical initial conditions across cells. Notably, these initial conditions also influenced Multi-cells' speedup, which ranged from 1x to 6x depending on whether the initial conditions were uniform or varied between cells.

We evaluated the Block-cells strategy by integrating a GPU-accelerated Biconjugate Gradient (BCG) linear solver into CAMP. The performance was compared to the base implementation, which utilizes the KLU linear solver on the CPU. When using the GPU BCG function within the base version of CAMP, without Multi-cells, the speedup was less than 1x. However, incorporating Multi-cells resulted in a 17x speedup. This outcome highlights the crucial role of approaches like Multi-cells in effectively integrating GPU functions.

We tested various configurations of the Block-cells strategy by varying the number of cells per block from 1 to 6. Using one cell per block resulted in a 35x speedup, while increasing the number of cells per block reduced the speedup to 27x. Notably, the highest speedup was 50% faster than the node-to-node comparison using MPI, underscoring the superiority of the GPU-based Block-cells approach over traditional CPU-based methods.

In chapter 5 we extended the Block-cells strategy from the BCG linear solver to the whole ODE solver of CAMP, which involves approximately five times more lines of code. Our initial results focus on a Box model emulating MONARCH input, facilitating development. The error, represented as Normalized Root Mean Square Error (NRMSE), is below 0.02%. The speedup achieved in a node-to-node comparison between the GPU and CPU versions is up to 5.9x. The speedup against 1 CPU core is 31x, similar to the 35x speedup of just the linear solver. This indicates that the new additions slightly reduced the speedup and that the similarities between the linear and ODE solvers are significant. Scaling the test to 4 GPUs shows a linear increase in speedup, reaching an outstanding 137x speedup against 1 CPU core.

We integrated our CAMP GPU version into MONARCH and compared the results against the CPU version in a node-to-node comparison. The error is stabilized at 0.42% for a 24-hour simulation (20 and 480 chemistry time steps), indicating that the NRMSE remains stable for short runs as the number of time steps increases. The speedup in this node-to-node comparison was 9.8x, significantly surpassing other state-of-the-art modules. For example, the KPP GPU study reported a 1.75x speedup, and CAM4-Chem achieved a 1.95x speedup. These results highlight the advantages of the Block-cells implementation.

In Chapter 6, we introduced a parallel CPU-GPU implementation and a simple and effective automatic load-balancing algorithm. This approach integrates the GPU and CPU solvers into an asynchronous execution model, enabling concurrent computation of the workload by both processors. The load-balancing algorithm is designed to achieve optimal performance by stabilizing when the CPU and GPU

times are nearly equal. Additionally, we implemented multiple optimizations, such as reduced CPU-GPU data transfers.

Our initial results focus on a Box model emulating MONARCH input. We assessed the previous GPU-only version on the Marenostrom 4 and 5 clusters. Comparing the execution on Marenostrom 5 against Marenostrom 4, it exhibits a 9x and 6x improvement in Performance and Arithmetic Intensity metrics. These results underscore the high capabilities of the new architecture. We compared our automatic load-balancing algorithm (Automatic case) to a fixed computational load method applied across all time steps (Fixed case). The best case is the automatic load balance, starting at 95% of the load to the GPU. This case reports a 70x speedup against the single-thread CPU version using a single GPU. Increasing the GPUs to 4, the speedup scales to 250x. This speedup doubles the previous results from the GPU-Only Marenostrom 4 version.

The 1-day MONARCH results indicate that the automatic load balance outperforms the fixed load balance by 25%, achieving a total speedup of 8.14x in a node-to-node comparison. This speedup is similar to the previous version, while the number of cores was doubled in the new architecture Marenostrom 5. Thus, the normalized speedup against the GPU-Only version is 2.32x. This notable enhancement underscores the advantages of the heterogeneous implementation, particularly with optimizations such as reduced CPU-GPU data transfers.

Throughout this thesis, we focused on accelerating atmospheric chemistry computations using GPUs. Our results demonstrate a significant performance improvement compared to other state-of-the-art chemistry modules. For instance, the KPP and CAM4-chem achieve speedups of 1.75x and 1.95x in node-to-node comparisons, respectively, while our approach achieves a speedup of 8.14x, over four times greater. This advantage is particularly notable considering that the Marenostrom 5 cluster uses 20 CPU cores per GPU, compared to the 5 CPU cores per GPU used in KPP GPU experiments. When normalizing for CPU cores, our implementation is 16 times faster. Additionally, even when compared to the RCK study, which reports a 59x speedup in a more complex scenario not available on atmospheric models, our implementation achieves a remarkable 250x speedup. Thus, our implementation is 4.2 times faster than other complex developments that require extensive code modifications.

The high speedup against other state-of-the-art solutions is achieved with minimal changes to the algorithm, thanks to the Block-cells strategy, resulting in a solution that is both low in development effort and high in performance.

7.2 Future work

The work developed in this thesis opens several avenues for future research. The immediate future work will involve publishing a paper that combines the content

from Chapters 5 and 6.

In the short term, further optimizations can be pursued with a focus on the two key priorities of this thesis: performance and simplicity. One area of improvement is the load-balancing algorithm. This could involve studying how load balance evolves between time steps and refining the algorithm to better align with this evolution by fine-tuning the parameters involved. Additionally, alternatives to the averaged load balance, such as using the median, could be explored to better correlate with the observed acceleration. This possible improvement in load balancing could ultimately result in even greater acceleration.

Another research direction involves analyzing the CPU and GPU solvers' performance metrics to identify areas for improvement. This includes understanding the underlying causes of the low-performance metrics highlighted in this thesis and exploring optimizations, such as replacing atomic operations to improve thread synchronization.

For future optimizations, one approach is to examine the changes made between the GPU-Only and CPU-GPU versions, such as transitioning from the CSC to the CSR structure. Additionally, MONARCH's chemical mechanism configuration modifications are worth exploring to resemble the Box model case closely. Specifically, aligning the 86 threads per block used in the Box configuration with the 127 threads used in MONARCH could improve memory usage by eliminating idle threads that consume memory registers, thereby accelerating execution. These efforts aim to refine the GPU implementation, ensuring it operates at peak efficiency while maintaining or improving the accuracy of atmospheric modeling results.

In the long term, a potential research direction is to explore integrating the methods presented in this thesis with other chemistry solvers. While the CVODE solver is designed to handle simple and complex chemistry mechanisms, other solvers, such as KPP, may be better suited for specific chemical reactions. Since the CAMP framework and our methods are designed with portability in mind, this approach should be a viable way to accelerate execution further.

Notably, the portability of the implementations presented in this thesis is primarily limited by the GPU's block size. Therefore, these methods can be applied to any solver with fewer unknowns than the maximum threads per block, typically 1024 threads, which is a considerable number. This opens the door to exploring the efficiency of our solvers in other fields, such as Computational Fluid Dynamics (CFD). We anticipate that our methods will generally enhance the state-of-the-art use of GPUs, leading to higher performance on HPC.

In my opinion, investing time in designing efficient strategies is crucial for enhancing performance. While many optimizations focus on extracting the last bit of performance, this often results in complex applications that are difficult to port and optimize further. The Earth Science Model is a clear example, which has gradually incorporated optimizations tailored for CPU architecture. When GPU computing

7.2. FUTURE WORK

emerged as a viable alternative, these CPU-specific optimizations usually complicated the transition to GPUs. Therefore, while all optimizations have value, simplicity is essential to facilitate new developments and provide portability across various applications.

Bibliography

- [1] K. Abbass, M. Z. Qasim, H. Song, M. Murshed, H. Mahmood, and I. Younis, “A review of the global climate change impacts, adaptation, and sustainable mitigation measures,” *Environmental Science and Pollution Research*, vol. 29, no. 28, pp. 42 539–42 559, Jun. 2022. [Online]. Available: <https://doi.org/10.1007/s11356-022-19718-6>
- [2] W. Leal Filho, U. M. Azeiteiro, A.-L. Balogun, A. F. F. Setti, S. A. R. Mucova, D. Ayal, E. Totin, A. M. Lydia, F. K. Kalaba, and N. O. Oguge, “The influence of ecosystems services depletion to climate change adaptation efforts in Africa,” *Science of The Total Environment*, vol. 779, p. 146414, Jul. 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0048969721014820>
- [3] C. J. Merchant, F. Paul, T. Popp, M. Ablain, S. Bontemps, P. Defourny, R. Hollmann, T. Lavergne, A. Laeng, G. de Leeuw, J. Mittaz, C. Poulsen, A. C. Povey, M. Reuter, S. Sathyendranath, S. Sandven, V. F. Sofieva, and W. Wagner, “Uncertainty information in climate data records from Earth observation,” *Earth System Science Data*, vol. 9, no. 2, pp. 511–527, Jul. 2017, publisher: Copernicus GmbH. [Online]. Available: <https://essd.copernicus.org/articles/9/511/2017/>
- [4] F. J. Doblas-Reyes, J. García-Serrano, F. Lienert, A. P. Biescas, and L. R. L. Rodrigues, “Seasonal climate predictability and forecasting: status and prospects,” *WIREs Climate Change*, vol. 4, no. 4, pp. 245–268, 2013, _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcc.217>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/wcc.217>
- [5] L. R. Mudryk, P. J. Kushner, C. Derksen, and C. Thackeray, “Snow cover response to temperature in observational and climate model ensembles,” *Geophysical Research Letters*, vol. 44, no. 2, pp. 919–926, 2017, _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/2016GL071789>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/2016GL071789>
- [6] T. Palmer, “Climate forecasting: Build high-resolution global climate models,” *Nature*, vol. 515, no. 7527, pp. 338–339, Nov. 2014, publisher:

- Nature Publishing Group. [Online]. Available: <https://www.nature.com/articles/515338a>
- [7] J. D. Sterman, “All models are wrong: reflections on becoming a systems scientist,” *System Dynamics Review*, vol. 18, no. 4, pp. 501–531, 2002. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sdr.261>
- [8] M. Poznic, “Models in Science and Engineering: Imagining, Designing and Evaluating Representations,” PhD Thesis, Delft University of Technology, 2017.
- [9] M. Carmen Lemos, C. Kirchhoff, and V. Ramprasad, “Narrowing the Climate Information Usability Gap,” *Nature Climate Change*, vol. 2, Oct. 2012.
- [10] N. D Bennett, B. Croke, A. Jakeman, L. T H Newham, and J. P Norton, “Performance evaluation of environmental models,” in *Conference: iEMSs2010 at Ottawa, Canada*, Jul. 2010.
- [11] H. Casanova, A. Legrand, and Y. Robert, *Parallel Algorithms*. New York: Chapman and Hall/CRC, Oct. 2011.
- [12] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, “Putting Polyhedral Loop Transformations to Work,” in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Berlin, Heidelberg: Springer, 2004, pp. 209–225.
- [13] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures,” *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, Feb. 2014, publisher: Cambridge University Press.
- [14] K. Alexander and S. M. Easterbrook, “The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations,” *Geoscientific Model Development*, vol. 8, no. 4, pp. 1221–1232, Apr. 2015, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/8/1221/2015/>
- [15] R. Kelly, “GPU Computing for Atmospheric Modeling,” *Computing in Science & Engineering*, vol. 12, pp. 26–33, Sep. 2010.
- [16] M. Z. Jacobson, *Fundamentals of Atmospheric Modeling*, 2nd ed. Cambridge: Cambridge University Press, 2005. [Online]. Available: <https://www.cambridge.org/core/books/fundamentals-of-atmospheric-modeling/A6B866737D682B17EE46F8449F76FB2C>
- [17] H. Zhang, J. C. Linford, A. Sandu, and R. Sander, “Chemical Mechanism Solvers in Air Quality Models,” *Atmosphere*, vol. 2, no. 3, pp. 510–532, Sep. 2011, number: 3 Publisher: Molecular Diversity Preservation International. [Online]. Available: <https://www.mdpi.com/2073-4433/2/3/510>

- [18] M. Christou, T. Christoudias, J. Morillo, D. Alvarez, and H. Merx, “Earth system modelling on system-level heterogeneous architectures: EMAC (version 2.42) on the Dynamical Exascale Entry Platform (DEEP),” *Geoscientific Model Development*, vol. 9, no. 9, pp. 3483–3491, Sep. 2016, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/9/3483/2016/>
- [19] T. Christoudias, T. Kirfel, A. Kerkweg, D. Taraborrelli, G.-E. Moulard, E. Raffin, V. Azizi, G. v. d. Oord, and B. v. Werkhoven, “GPU Optimizations for Atmospheric Chemical Kinetics,” in *The International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 136–138. [Online]. Available: <https://doi.org/10.1145/3432261.3439863>
- [20] L. F. Shampine and C. W. Gear, “A User’s View of Solving Stiff Ordinary Differential Equations,” *SIAM Review*, vol. 21, no. 1, pp. 1–17, Jan. 1979, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1021001>
- [21] C. Pérez, K. Haustein, Z. Janjic, O. Jorba, N. Huneus, J. M. Baldasano, T. Black, S. Basart, S. Nickovic, R. L. Miller, J. P. Perlwitz, M. Schulz, and M. Thomson, “Atmospheric dust modeling from meso to global scales with the online NMMB/BSC-Dust model – Part 1: Model description, annual simulations and evaluation,” *Atmospheric Chemistry and Physics*, vol. 11, pp. 13 001–13 027, 2011. [Online]. Available: <https://doi.org/10.5194/acp-11-13001-2011>
- [22] A. Badia, O. Jorba, A. Voulgarakis, D. Dabdub, C. Pérez García-Pando, A. Hilboll, M. Gonçalves, and Z. Janjic, “Description and evaluation of the multiscale online nonhydrostatic atmosphere chemistry model (nmmb-monarch) version 1.0: gas-phase chemistry at global scale,” *Geoscientific Model Development*, vol. 10, no. 2, pp. 609–638, 2017. [Online]. Available: <https://gmd.copernicus.org/articles/10/609/2017/>
- [23] O. Jorba, D. Dabdub, C. Blaszcak-Boxe, C. Pérez, Z. Janjic, J. M. Baldasano, M. Spada, A. Badia, and M. Gonçalves, “Potential significance of photoexcited NO₂ on global air quality with the NMMB/BSC chemical transport model,” *Journal of Geophysical Research: Atmospheres*, vol. 117, no. D13, 2012, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1029/2012JD017730>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1029/2012JD017730>
- [24] A. Badia and O. Jorba, “Gas-phase evaluation of the online nmmb/bsc-ctm model over europe for 2010 in the framework of the aqmeii-phase2 project,” *Atmospheric Environment*, vol. 115, pp. 657 – 669, 2015. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1352231014004026>

- [25] P. Xian, J. S. Reid, E. J. Hyer, C. R. Sampson, J. I. Rubin, M. Ades, N. Asencio, S. Basart, A. Benedetti, P. S. Bhattacharjee, M. E. Brooks, P. R. Colarco, A. M. da Silva, T. F. Eck, J. Guth, O. Jorba, R. Kouznetsov, Z. Kipling, M. Sofiev, C. Perez Garcia-Pando, Y. Pradhan, T. Tanaka, J. Wang, D. L. Westphal, K. Yumimoto, and J. Zhang, “Current state of the global operational aerosol multi-model ensemble: An update from the International Cooperative for Aerosol Prediction (ICAP),” *Quarterly Journal of the Royal Meteorological Society*, vol. 145, no. S1, pp. 176–209, 2019, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3497>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/qj.3497>
- [26] M. L. Dawson, C. Guzman, J. H. Curtis, M. Acosta, S. Zhu, D. Dabdub, A. Conley, M. West, N. Riemer, and O. Jorba, “Chemistry Across Multiple Phases (CAMP) version 1.0: an integrated multiphase chemistry model,” *Geoscientific Model Development*, vol. 15, no. 9, pp. 3663–3689, May 2022, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/15/3663/2022/>
- [27] V. Damian, A. Sandu, M. Damian, F. Potra, and G. Carmichael, “The kinetic preprocessor KPP—A software environment for solving chemical kinetics,” *Computers & Chemical Engineering*, vol. 26, pp. 1567–1579, Nov. 2002.
- [28] O. Tintó, M. Acosta, M. Castrillo, A. Cortés, A. Sanchez, K. Serradell, and F. J. Doblas-Reyes, “Optimizing domain decomposition in an ocean model: the case of NEMO,” *Procedia Computer Science*, vol. 108, pp. 776–785, Jan. 2017.
- [29] L. Szustak, R. Wyrzykowski, K. Halbiniak, and P. Bratek, “Toward Heterogeneous MPI+MPI Programming: Comparison of OpenMP and MPI Shared Memory Models,” in *Euro-Par 2019: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, and S. L. Scott, Eds. Cham: Springer International Publishing, 2020, pp. 270–281.
- [30] B. Armstrong, S. W. Kim, and R. Eigenmann, “Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite,” in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds. Berlin, Heidelberg: Springer, 2000, pp. 482–493.
- [31] T. Yamagishi and Y. Matsumura, “GPU Acceleration of a Non-hydrostatic Ocean Model with a Multigrid Poisson/Helmholtz solver,” *Procedia Computer Science*, vol. 80, pp. 1658–1669, Jan. 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050916309899>

- [32] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, “Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers,” *International Journal of Computational Fluid Dynamics*, vol. 31, pp. 1–16, Oct. 2017.
- [33] R. Borrell, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, O. Lehmkuhl, V. Mehta, H. Owen, M. Vázquez, and G. Oyarzun, “Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics,” *Future Generation Computer Systems*, vol. 107, pp. 31–48, Jun. 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X1930994X>
- [34] J. Sun, J. S. Fu, J. B. Drake, Q. Zhu, A. Haidar, M. Gates, S. Tomov, and J. Dongarra, “Computational Benefit of GPU Optimization for the Atmospheric Chemistry Modeling,” *J. Adv. Model. Earth Syst.*, vol. 10, no. 8, pp. 1952–1969, Aug. 2018. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1029/2018MS001276>
- [35] M. Alvanos and T. Christoudias, “Accelerating Atmospheric Chemical Kinetics for Climate Simulations,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2396–2407, Nov. 2019, conference Name: IEEE Transactions on Parallel and Distributed Systems.
- [36] K. E. Niemeyer and C.-J. Sung, “Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs,” *Journal of Computational Physics*, vol. 256, pp. 854–871, Jan. 2014, arXiv: 1309.2710. [Online]. Available: <http://arxiv.org/abs/1309.2710>
- [37] Q. Tan, J. Sun, J. Dennis, M. Dawson, and S. Project, “GPU Enablement of MICM Chemistry Solver,” SIParCS Project 9, Tech. Rep., Aug. 2023.
- [38] M. Dawson, J. Sun, K. Shores, D. Fillmore, Q. Tan, C. Craig, J. Gim, M. Waxmonsky, F. Vitt, A. Conley, and A. Karsenti, “Model Independent Chemistry Model (MICM),” Aug. 2024. [Online]. Available: <https://github.com/NCAR/micm>
- [39] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Boston: Addison-Wesley Professional, Sep. 2017.
- [40] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998, conference Name: IEEE Computational Science and Engineering. [Online]. Available: <https://ieeexplore.ieee.org/document/660313>
- [41] S. Ubbiali, C. Kühnlein, C. Schär, L. Schlemmer, T. C. Schulthess, M. Staneker, and H. Wernli, “Exploring a high-level programming model for

- the NWP domain using ECMWF microphysics schemes,” *Geoscientific Model Development Discussions*, pp. 1–30, Jun. 2024, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/preprints/gmd-2024-92/>
- [42] X. Lapillonne, K. Osterried, and O. Fuhrer, “Chapter 13 - Using OpenACC to port large legacy climate and weather modeling code to GPUs,” in *Parallel Programming with OpenACC*, R. Farber, Ed. Boston: Morgan Kaufmann, Jan. 2017, pp. 267–290. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780124103979000135>
- [43] X. Lapillonne, W. Sawyer, P. Marti, V. Clement, R. Dietlicher, L. Kornblueh, S. Rast, R. Schnur, M. Esch, M. Giorgetta, D. Alexeev, and R. Pincus, “Global climate simulations at 2.8 km on GPU with the ICON model,” Copernicus Meetings, Tech. Rep. EGU2020-10306, Mar. 2020, conference Name: EGU2020. [Online]. Available: <https://meetingorganizer.copernicus.org/EGU2020/EGU2020-10306.html>
- [44] V. Clement, P. Marti, X. Lapillonne, O. Fuhrer, and W. Sawyer, “Automatic Port to OpenACC/OpenMP for Physical Parameterization in Climate and Weather Code Using the CLAW Compiler,” *Supercomputing Frontiers and Innovations*, vol. 6, no. 3, pp. 51–63, Sep. 2019, number: 3. [Online]. Available: <https://superfri.org/index.php/superfri/article/view/285>
- [45] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, “High-performance Cholesky factorization for GPU-only execution,” *Proceedings of the General Purpose GPUs*, pp. 42–52, Feb. 2017, conference Name: PPOPP ’17: 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming ISBN: 9781450349154 Place: Austin TX USA Publisher: ACM. [Online]. Available: <https://dl.acm.org/doi/10.1145/3038228.3038237>
- [46] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, “Evaluation and tuning of the Level 3 CUBLAS for graphics processors,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008, pp. 1–8, iSSN: 1530-2075. [Online]. Available: <https://ieeexplore.ieee.org/document/4536485>
- [47] R. Serban and A. C. Hindmarsh, “CVODES: The Sensitivity-Enabled ODE Solver in SUNDIALS,” in *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, Jun. 2008, pp. 257–269. [Online]. Available: <https://dx.doi.org/10.1115/DETC2005-85597>
- [48] C. Guzman Ruiz, M. Dawson, M. Acosta, O. Jorba, E. Cesar, C. Perez Garcia-Pando, and K. Serradell, “CAMP - version GPU Linear solver,” *Mendeley Data*, vol. V1, May 2022. [Online]. Available: <https://data.mendeley.com/datasets/5gkgnv3skg>

- [49] C. Guzman Ruiz, M. Acosta, O. Jorba, E. Cesar Galobardes, M. Dawson, G. Oyarzun, C. Pérez García-Pando, and K. Serradell, “Optimized thread-block arrangement in a GPU implementation of a linear solver for atmospheric chemistry mechanisms,” *Computer Physics Communications*, vol. 302, p. 109240, Sep. 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010465524001632>
- [50] C. G. Ruiz, M. Dawson, M. C. Acosta, O. Jorba, E. C. Galobardes, C. P. García-Pando, and K. Serradell, “Adapting Atmospheric Chemistry Components for Efficient GPU Accelerators,” in *Proceedings of Eighth International Congress on Information and Communication Technology*, X.-S. Yang, R. S. Sherratt, N. Dey, and A. Joshi, Eds. Singapore: Springer Nature Singapore, 2023, pp. 129–138.
- [51] C. Guzman Ruiz, M. C. Acosta, O. Jorba, and C. P. García-Pando, “Novel approaches to accelerate chemistry for climate models,” in *Platform for Advanced Scientific Computing (PASC) 2023*, Congresscenter Davos, Switzerland, Jun. 2023. [Online]. Available: <https://pasc23.pasc-conference.org/presentation/>
- [52] C. Guzman Ruiz, M. C. Acosta, M. Dawson, O. Jorba, C. P. García-Pando, and S. Kim, “CAMP First GPU Solver: A Solution to Accelerate Chemistry in Atmospheric Models,” in *9th BSC Doctoral Symposium*, Universitat Politècnica de Catalunya, Spain, May 2022. [Online]. Available: <https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/9th-bsc-doctoral-symposium-2022>
- [53] —, “CAMP First GPU Solver: A Solution to Accelerate Chemistry in Atmospheric Models,” in *7th HPC workshop of the European Network for Earth System modelling*, Barcelona SuperComputing Center, Spain, May 2022. [Online]. Available: <https://portal.enes.org/hpc-workshops-detailed/#hpc7>
- [54] —, “Studying a new GPU treatment for chemical modules inside CAMP,” in *19th workshop on HPC in meteorology*, Online, Sep. 2021. [Online]. Available: <https://events.ecmwf.int/event/169/timetable/>
- [55] —, “Exploiting parallelism for CPU and GPU linear solvers on chemistry for atmospheric models,” in *8th BSC Doctoral Symposium*, Online, May 2021. [Online]. Available: <https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/8th-bsc-doctoral-symposium-2021>
- [56] —, “Accelerating Atmospheric Models using GPU,” in *The 2020 International Conference on High Performance Computing & Simulation (HPCS 2020)*, Mar. 2021. [Online]. Available: <https://hpcs2020.cisedu.info/>
- [57] —, “Accelerating Chemistry Modules in Atmospheric Models Using GPUs,” in *6th ENES workshop on High Performance Computing*

- for Climate and Weather*, Online, May 2020. [Online]. Available: <https://www.esiwace.eu/events/6th-hpc-workshop>
- [58] —, “Accelerating Chemistry Modules in Atmospheric Models Using GPUs,” in *NVIDIA GTC 2020 Spring*, Online, Mar. 2020. [Online]. Available: <https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s22005/>
- [59] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, “MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner,” *Computers & Fluids*, vol. 92, pp. 244–252, Mar. 2014.
- [60] D. Steyn, P. Builtjes, M. Schaap, and G. Yarwood, “Regional air quality modeling: North American and European perspectives,” *EM: Air and Waste Management Association’s Magazine for Environmental Managers*, pp. 6–10, Jul. 2012.
- [61] J. H. Seinfeld and S. N. Pandis, *Atmospheric chemistry and physics: from air pollution to climate change*. Wiley, 1998, google-Books-ID: IK8PAQAAMAAJ.
- [62] A. Badia, O. Jorba, A. Voulgarakis, D. Dabdub, C. Pérez García-Pando, A. Hilboll, M. Gonçalves, and Z. Janjic, “Description and evaluation of the Multiscale Online Nonhydrostatic Atmosphere Chemistry model (NMMB-MONARCH) version 1.0: gas-phase chemistry at global scale,” *Geoscientific Model Development*, vol. 10, no. 2, pp. 609–638, Feb. 2017, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/10/609/2017/>
- [63] K. Haustein, C. Pérez, J. M. Baldasano, O. Jorba, S. Basart, R. L. Miller, Z. Janjic, T. Black, S. Nickovic, M. C. Todd, R. Washington, D. Müller, M. Tesche, B. Weinzierl, M. Esselborn, and A. Schladitz, “Atmospheric dust modeling from meso to global scales with the online NMMB/BSC-Dust model – Part 2: Experimental campaigns in Northern Africa,” *Atmospheric Chemistry and Physics*, vol. 12, no. 6, pp. 2933–2958, Mar. 2012, publisher: Copernicus GmbH. [Online]. Available: <https://acp.copernicus.org/articles/12/2933/2012/>
- [64] M. Spada, O. Jorba, C. Pérez García-Pando, Z. Janjic, and J. M. Baldasano, “Modeling and evaluation of the global sea-salt aerosol distribution: sensitivity to size-resolved and sea-surface temperature dependent emission schemes,” *Atmospheric Chemistry and Physics*, vol. 13, no. 23, pp. 11 735–11 755, Dec. 2013, publisher: Copernicus GmbH. [Online]. Available: <https://acp.copernicus.org/articles/13/11735/2013/>
- [65] M. Spada, “Development and evaluation of an atmospheric aerosol module implemented within the NMMB/BSC-CTM,” PhD Thesis, Universitat

- Politecnica de Catalunya, 2015. [Online]. Available: <http://hdl.handle.net/2117/95991>
- [66] H. Navarro-Barboza, M. Pandolfi, M. Guevara, S. Enciso, C. Tena, M. Via, J. Yus-Díez, C. Reche, N. Pérez, A. Alastuey, X. Querol, and O. Jorba, “Uncertainties in source allocation of carbonaceous aerosols in a Mediterranean region,” *Environment International*, vol. 183, p. 108252, Jan. 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0160412023005251>
- [67] E. Di Tomaso, N. A. J. Schutgens, O. Jorba, and C. Pérez García-Pando, “Assimilation of MODIS Dark Target and Deep Blue observations in the dust aerosol component of NMMB-MONARCH version 1.0,” *Geoscientific Model Development*, vol. 10, no. 3, pp. 1107–1129, Mar. 2017, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/10/1107/2017/>
- [68] D. Manubens-Gil, J. Vegas-Regidor, C. Prodhomme, O. Mula-Valls, and F. J. Doblas-Reyes, “Seamless management of ensemble climate prediction experiments on HPC platforms,” in *2016 International Conference on High Performance Computing & Simulation (HPCS)*. Innsbruck, Austria: IEEE, Jul. 2016, pp. 895–900. [Online]. Available: <http://ieeexplore.ieee.org/document/7568429/>
- [69] S. Basart, J. Benavides, D. Bowdalo, M. Dawson, E. Tomaso, M. Gonçalves Ageitos, M. Guevara, M. Klose, F. Macchia, V. Obiso, M. Olid, M. Pay, M. Porquet, K. Serradell, C. Tena Medina, and C. Pérez García-Pando, “Predicción de la calidad del aire multiescala con el modelo MONARCH en el Centro Nacional de Supercomputación,” in *Sexto Simposio Nacional de Predicción ”Memorial Antonio Mestre”*. AEMET, Jan. 2019, pp. 405–406.
- [70] Z. I. Janjic, J. P. Gerrity, and S. Nickovic, “An Alternative Approach to Nonhydrostatic Modeling,” *Monthly Weather Review*, vol. 129, no. 5, pp. 1164 – 1178, 2001, place: Boston MA, USA Publisher: American Meteorological Society. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/129/5/1520-0493_2001_129_1164_aaatnm_2.0.co_2.xml
- [71] Z. Janjic and L. Gall, “Scientific documentation of the NCEP nonhydrostatic multiscale model on the B grid (NMMB). Part 1 Dynamics,” *University Corporation for Atmospheric Research*, 2012. [Online]. Available: <https://opensky.ucar.edu/islandora/object/technotes%3A502/>
- [72] G. Yarwood, S. Rao, M. Yocke, and G. Whitten, “Updates to the Carbon Bond Chemical Mechanism: CB05. Final Report to the US EPA, RT-0400675,” 2005.

- [73] G. Sarwar, H. Simon, P. Bhave, and G. Yarwood, "Examining the impact of heterogeneous nitryl chloride production on air quality across the United States," *Atmospheric Chemistry and Physics*, vol. 12, no. 14, pp. 6455–6473, Jul. 2012, publisher: Copernicus GmbH. [Online]. Available: <https://acp.copernicus.org/articles/12/6455/2012/>
- [74] O. Wild, X. Zhu, and M. J. Prather, "Fast-J: Accurate Simulation of In- and Below-Cloud Photolysis in Tropospheric Chemical Models," *Journal of Atmospheric Chemistry*, vol. 37, pp. 245–282, 2000.
- [75] S. Metzger, F. Dentener, S. Pandis, and J. Lelieveld, "Gas/aerosol partitioning: 1. A computationally efficient model," *Journal of Geophysical Research: Atmospheres*, vol. 107, no. D16, pp. ACH 16–1–ACH 16–24, 2002, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1029/2001JD001102>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1029/2001JD001102>
- [76] K. Tsigaridis and M. Kanakidou, "Global modelling of secondary organic aerosol in the troposphere: a sensitivity analysis," *Atmospheric Chemistry and Physics*, 2003.
- [77] S. J. Pai, C. L. Heald, J. R. Pierce, S. C. Farina, E. A. Marais, J. L. Jimenez, P. Campuzano-Jost, B. A. Nault, A. M. Middlebrook, H. Coe, J. E. Shilling, R. Bahreini, J. H. Dingle, and K. Vu, "An evaluation of global organic aerosol schemes using airborne observations," *Atmospheric Chemistry and Physics*, vol. 20, no. 5, pp. 2637–2665, Mar. 2020, publisher: Copernicus GmbH. [Online]. Available: <https://acp.copernicus.org/articles/20/2637/2020/>
- [78] M. L. Wesely, "Parameterization of surface resistances to gaseous dry deposition in regional-scale numerical models," *Atmospheric Environment (1967)*, vol. 23, no. 6, pp. 1293–1304, Jan. 1989. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0004698189901534>
- [79] L. Zhang, S. Gong, J. Padro, and L. Barrie, "A size-segregated particle dry deposition scheme for an atmospheric aerosol module," *Atmospheric Environment*, vol. 35, pp. 549–560, 2001.
- [80] K. M. Foley, S. J. Roselle, K. W. Appel, P. V. Bhave, J. E. Pleim, T. L. Otte, R. Mathur, G. Sarwar, J. O. Young, R. C. Gilliam, C. G. Nolte, J. T. Kelly, A. B. Gilliland, and J. O. Bash, "Incremental testing of the Community Multiscale Air Quality (CMAQ) modeling system version 4.7," *Geoscientific Model Development*, vol. 3, no. 1, pp. 205–226, Mar. 2010, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/3/205/2010/>
- [81] A. Guenther, T. Karl, P. Harley, C. Wiedinmyer, P. I. Palmer, and C. Geron, "Estimates of global terrestrial isoprene emissions using MEGAN (Model

- of Emissions of Gases and Aerosols from Nature),” *Atmospheric Chemistry and Physics*, vol. 6, no. 11, pp. 3181–3210, Aug. 2006. [Online]. Available: <https://www.atmos-chem-phys.net/6/3181/2006/>
- [82] M. Klose, O. Jorba, M. Gonçalves Ageitos, J. Escribano, M. L. Dawson, V. Obiso, E. Di Tomaso, S. Basart, G. Montané Pinto, F. Macchia, P. Ginoux, J. Guerschman, C. Prigent, Y. Huang, J. F. Kok, R. L. Miller, and C. Pérez García-Pando, “Mineral dust cycle in the Multiscale Online Nonhydrostatic Atmosphere Chemistry model (MONARCH) Version 2.0,” *Geoscientific Model Development*, vol. 14, no. 10, pp. 6403–6444, Oct. 2021, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/14/6403/2021/>
- [83] L. Jaeglé, P. K. Quinn, T. S. Bates, B. Alexander, and J. Lin, “Global distribution of sea salt aerosols: new constraints from in situ and remote sensing observations,” *Atmospheric Chemistry and Physics*, vol. 11, pp. 3137–3157, 2011.
- [84] M. Sofiev, P. Siljamo, H. Ranta, T. Linkosalo, S. Jaeger, A. Rasmussen, A. Rantio-Lehtimäki, E. Severova, and J. Kukkonen, “A numerical model of birch pollen emission and dispersion in the atmosphere. Description of the emission module,” *International Journal of Biometeorology*, vol. 57, no. 1, pp. 45–58, Jan. 2013. [Online]. Available: <https://doi.org/10.1007/s00484-012-0532-z>
- [85] M. Gonçalves Ageitos, V. Obiso, R. L. Miller, O. Jorba, M. Klose, M. Dawson, Y. Balkanski, J. Perlwitz, S. Basart, E. Di Tomaso, J. Escribano, F. Macchia, G. Montané, N. M. Mahowald, R. O. Green, D. R. Thompson, and C. Pérez García-Pando, “Modeling dust mineralogical composition: sensitivity to soil mineralogy atlases and their expected climate impacts,” *Atmospheric Chemistry and Physics*, vol. 23, no. 15, pp. 8623–8657, Aug. 2023, publisher: Copernicus GmbH. [Online]. Available: <https://acp.copernicus.org/articles/23/8623/2023/>
- [86] S. Eckermann, “Hybrid $\sigma - p$ Coordinate Choices for a Global Model,” *Monthly Weather Review*, vol. 137, Jan. 2009.
- [87] E. N. Lorenz, “Energy and Numerical Weather Prediction,” *Tellus*, vol. 12, no. 4, pp. 364–373, 1960, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2153-3490.1960.tb01323.x>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2153-3490.1960.tb01323.x>
- [88] W. F. Ames, *Numerical methods for partial differential equations*. New York, Barnes & Noble, 1969. [Online]. Available: http://archive.org/details/numericalmethods0000unse_j2x9

- [89] F. Mesinger, Z. Janjic, S. Nickovic, D. Zupanski, and D. Deaven, “The Step-Mountain Coordinate: Model Description and Performance for Cases of Alpine Lee Cyclogenesis and for a Case of an Appalachian Redevelopment,” *Monthly Weather Review - MON WEATHER REV*, vol. 116, Jul. 1988.
- [90] A. Pletinckx, D. Fiß, and A. Kratzsch, “Developing and Implementing Two-Step Adams-Bashforth-Moulton Method with Variable Stepsize for the Simulation Tool DynStar,” *ACC Journal*, vol. 23, pp. 51–61, Jun. 2017.
- [91] Z. Janjic, R. Gall, and E. Pyle, “Scientific Documentation for the NMM Solver,” *National Center for Atmospheric Research University Corporation for Atmospheric Research*, 2010. [Online]. Available: <https://opensky.ucar.edu/islandora/object/technotes%3A490/>
- [92] C. Hill, C. Deluca, V. Balaji, M. Suarez, and A. Da Silva, “Architecture of the Earth System Modeling Framework,” *Computing in Science & Engineering*, vol. 6, pp. 18–28, Feb. 2004.
- [93] R. Rew and G. Davis, “NetCDF: an interface for scientific data access,” *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, Jul. 1990, conference Name: IEEE Computer Graphics and Applications. [Online]. Available: <https://ieeexplore.ieee.org/document/56302>
- [94] S. D. Cohen, A. C. Hindmarsh, and P. F. Dubois, “CVODE, A Stiff/Nonstiff ODE Solver in C,” *Computers in Physics*, vol. 10, no. 2, p. 138, 1996. [Online]. Available: <http://scitation.aip.org/content/aip/journal/cip/10/2/10.1063/1.4822377>
- [95] T. Davis and E. Natarajan, “Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems.” *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 1–17, Jan. 2010.
- [96] C. Xu, Y. Gao, Z. Ren, and T. Lu, “A sparse stiff chemistry solver based on dynamic adaptive integration for efficient combustion simulations,” *Combustion and Flame*, vol. 172, pp. 183–193, Oct. 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010218016301651>
- [97] N. N. Yanenko, “Uniform Schemes,” in *The Method of Fractional Steps: The Solution of Problems of Mathematical Physics in Several Variables*, N. N. Yanenko and M. Holt, Eds. Berlin, Heidelberg: Springer, 1971, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-3-642-65108-3_1
- [98] G. D. Byrne and A. C. Hindmarsh, “A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations,” *ACM Trans. Math. Softw.*, vol. 1, no. 1, pp. 71–96, Mar. 1975. [Online]. Available: <https://dl.acm.org/doi/10.1145/355626.355636>

BIBLIOGRAPHY

- [99] K. R. Jackson and R. Sacks-Davis, “An Alternative Implementation of Variable Step-Size Multistep Formulas for Stiff ODEs,” *ACM Trans. Math. Softw.*, vol. 6, no. 3, pp. 295–318, Sep. 1980. [Online]. Available: <https://dl.acm.org/doi/10.1145/355900.355903>
- [100] B. J. Finlayson-Pitts and J. N. Pitts, *Chemistry of the Upper and Lower Atmosphere: Theory, Experiments, and Applications*. Academic Press, 2000, google-Books-ID: tU5NnwEACAAJ.
- [101] A. Fredenslund, R. L. Jones, and J. M. Prausnitz, “Group-contribution estimation of activity coefficients in nonideal liquid mixtures,” *AIChE Journal*, vol. 21, no. 6, pp. 1086–1099, 1975.
- [102] S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.
- [103] J. Says, “Accelerating HPC Applications with NVIDIA Nsight Compute Roofline Analysis,” Nov. 2020. [Online]. Available: <https://developer.nvidia.com/blog/accelerating-hpc-applications-with-nsight-compute-roofline-analysis/>
- [104] R. D. Skeel, “Construction of variable-stepsize multistep formulas,” in *Mathematics of Computation Vol. 47, No. 176 (Oct., 1986)*, published by American Mathematical Society, 1986, pp. 503–510.
- [105] BSC CNS, “Support Knowledge Center @ BSC-CNS,” 2022. [Online]. Available: <https://www.bsc.es/user-support/power.php>
- [106] S. Xiao and W. Feng, “Inter-block GPU communication via fast barrier synchronization,” in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–12, iSSN: 1530-2075.
- [107] B. Babaoğlu, “Application of biconjugate gradient stabilized method with spectral acceleration for propagation over terrain profiles,” Thesis, Bilkent University, 2003, accepted: 2016-07-01T10:59:31Z. [Online]. Available: <http://repository.bilkent.edu.tr/handle/11693/29420>
- [108] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, “GPGPU Processing in CUDA Architecture,” *Advanced Computing: An International Journal*, vol. 3, no. 1, pp. 105–120, Jan. 2012, arXiv:1202.4347 [cs]. [Online]. Available: <http://arxiv.org/abs/1202.4347>
- [109] M. Harris, “Optimizing Parallel Reduction in CUDA,” 2024. [Online]. Available: <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>

BIBLIOGRAPHY

- [110] Weather classes, “The SkewT and related diagnostic tools.” [Online]. Available: https://www.weatherclasses.com/uploads/1/3/1/3/131359169/dry_adiabatic_overview.pdf
- [111] NVIDIA, “Profiler User’s Guide.” [Online]. Available: <http://docs.nvidia.com/cuda/profiler-users-guide/index.html>
- [112] M. L. Dawson, C. Guzman, J. H. Curtis, M. Acosta, S. Zhu, D. Dabdub, A. Conley, M. West, N. Riemer, and O. Jorba, “Chemistry Across Multiple Phases (CAMP) version 1.0: An integrated multi-phase chemistry model,” *arXiv:2111.07436 [physics]*, Nov. 2021, arXiv: 2111.07436. [Online]. Available: <http://arxiv.org/abs/2111.07436>
- [113] ECMWF, “Documentation of the Integrated Forecasting System,” Tech. rep., ECMWF, Tech. Rep., 2020. [Online]. Available: <https://www.ecmwf.int/en/publications/ifs-documentation>
- [114] J. Flemming, V. Huijnen, J. Arteta, P. Bechtold, A. Beljaars, A.-M. Blechschmidt, M. Diamantakis, R. J. Engelen, A. Gaudel, A. Inness, L. Jones, B. Josse, E. Katragkou, V. Marecal, V.-H. Peuch, A. Richter, M. G. Schultz, O. Stein, and A. Tsikerdekis, “Tropospheric chemistry in the integrated forecasting system of ecmwf,” *Geoscientific Model Development*, vol. 8, no. 4, pp. 975–1003, 2015. [Online]. Available: <https://gmd.copernicus.org/articles/8/975/2015/>
- [115] J. J. P. Kuenen, A. J. H. Visschedijk, M. Jozwicka, and H. A. C. Denier van der Gon, “Tno-macc ii emission inventory; a multi-year (2003–2009) consistent high-resolution european emission inventory for air quality modelling,” *Atmospheric Chemistry and Physics*, vol. 14, no. 20, pp. 10 963–10 976, 2014. [Online]. Available: <https://www.atmos-chem-phys.net/14/10963/2014/>
- [116] C. Granier, S. Darras, H. A. C. Denier van der Gon, J. Doubalova, N. Elguindi, B. Galle, M. Gauss, M. Guevara, J.-P. Jalkanen, J. Kuenen, C. Liousse, B. Quack, D. Simpson, , and K. Sindelarova, “The copernicus atmosphere monitoring service global and regional emissions (april 2019 version),” Copernicus Atmosphere Monitoring Service (CAMS) report, Tech. Rep., 2019. [Online]. Available: <https://doi.org/10.24380/d0bn-kx16>
- [117] J. W. Kaiser, A. Heil, M. O. Andreae, A. Benedetti, N. Chubarova, L. Jones, J.-J. Morcrette, M. Razinger, M. G. Schultz, M. Suttie, and G. R. van der Werf, “Biomass burning emissions estimated with a global fire assimilation system based on observed fire radiative power,” *Biogeosciences*, vol. 9, no. 1, pp. 527–554, 2012. [Online]. Available: <https://www.biogeosciences.net/9/527/2012/>
- [118] M. Guevara, C. Tena, M. Porquet, O. Jorba, and C. Pérez García-Pando, “HERMESv3, a stand-alone multi-scale atmospheric emission modelling

BIBLIOGRAPHY

- framework – Part 1: global and regional module,” *Geoscientific Model Development*, vol. 12, no. 5, pp. 1885–1907, May 2019. [Online]. Available: <https://www.geosci-model-dev.net/12/1885/2019/>
- [119] —, “HERMESv3, a stand-alone multi-scale atmospheric emission modelling framework – Part 2: The bottom-up module,” *Geoscientific Model Development*, vol. 13, no. 3, pp. 873–903, Mar. 2020, publisher: Copernicus GmbH. [Online]. Available: <https://gmd.copernicus.org/articles/13/873/2020/>
- [120] D. H. A. C. van der Gon, C. Hendriks, J. Kuenen, A. Segers, and A. J. H. Visschedijk, “Description of current temporal emission patterns and sensitivity of predicted AQ for temporal emission patterns ,” 2011.
- [121] U. Ahmed, Y. Khalid, M. Aleem, M. Iqbal, and A. Islam, “Troodon A machine-learning based load-balancing application scheduler for CPU–GPU system,” *Journal of Parallel and Distributed Computing*, May 2019.
- [122] H. Choi, D. Son, S. Kang, J. Kim, H.-H. Lee, and C.-H. Kim, “An efficient scheduling scheme using estimated execution time for heterogeneous computing systems,” *The Journal of Supercomputing*, vol. 65, Aug. 2013.
- [123] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasonmayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to GPU codes,” in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10. [Online]. Available: <https://ieeexplore.ieee.org/document/6339595>
- [124] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54. [Online]. Available: <https://ieeexplore.ieee.org/document/5306797>
- [125] A. Munera, S. Royuela, G. Llort, E. Mercadal, F. Wartel, and E. Quiñones, “Experiences on the characterization of parallel applications in embedded systems with Extrae/Paraver,” in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–11. [Online]. Available: <https://doi.org/10.1145/3404397.3404440>
- [126] Barcelona SuperComputing Center, “MareNostrum 5,” Apr. 2024. [Online]. Available: <https://www.bsc.es/ca/marenostrum/marenostrum-5>