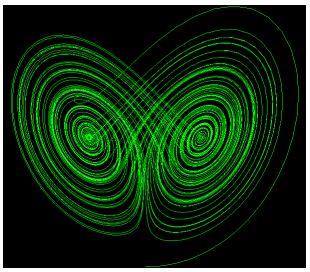
Departments of Mathematics, Physics and

MASTER THESIS

Modelling for Science and Engineering

Accelerating Atmospheric Models using GPUs

Christian Guzman Ruiz



September 2019

UAB
Universitat Autònoma de Barcelona

Abstract

Atmospheric models are widely used in meteorological institutions and investigation centers for weather forecast and climate prediction studies. The complexity of the equations used, the amount of data and the computational power needed to achieve a good accuracy on the simulation results demand a computational power only found in supercomputers. One of these models is The Multiscale Online Nonhydrostatic AtmospheRe CHemistry model (NMMB-MONARCH), a chemical weather prediction system used for dust forecast at centers like the Barcelona Dust Forecast Center under an initiative of the World Meteorological Organization. This model consumes thousands of hours of supercomputer every year.

For this reason, a prime focus is the evaluation of the computational performance of the model and, consequently, the improvement of the efficiency, reducing the model execution time and taking advantage of massive parallel supercomputers in an efficient way. In NMMB-MONARCH, the chemistry solver consumes most of the time. For this reason, we analyze this part and develop an adaptation of the code to profit from the advantages of new architectures based on GPUs, looking for an improvement on-time spent for this module and optimizing it for accelerators.

First, we focused on the most time-consuming functions of the chemistry solver. The functions that no come from any auxiliary library are very similar to each other, so we choose one of them to test the optimizations. We develop a GPU parallelization of the reactions calculated on the chemistry solver, obtaining a positive speedup for big mechanisms with a lot of reactions.

After this, we focused on the different points of the map that NMMB-MONARCH computes. The map data points, that we call cells, has no dependence from each other, so they can be parallelized without problems. Initially, the loop for all cells was located externally from the chemistry solver, calling it again for as many cells as we have. We move this loop from outside the solver to inside solver functions, having all the no dependent data present on the most time-consuming functions, and allowing the parallelization of this loop on GPU.

The results after this optimization were pretty good. Not only do we have a positive speedup using the GPU in front of the CPU with a low number of cells, but we also have a big speedup without using the GPU from the original version with the cell loop outside.

Acknowledgments

Special thanks to my tutor Mario Acosta; Matt Dawson, the developer of the chemistry module analyzed, and all the people involved from BSC.

Contents

Chapter 1

Introduction and Motivation

Computational models are simplified representations of an object or a process [1]. These models are considered useful because they can be instructive about the nature of real-world phenomena or systems [2], with a lot of applications on science and engineering [3]. For example, environmental models play an increasingly important role in understanding the potential implications of environmental changes [4].

There are a lot of types of models in the environmental sciences [5]. Commonly these models require a lot of computational resources and are involved around a lot of performance work [6]. For example, the model studied during this project is executed on the Top 500 supercomputer MareNostrum, from the Barcelona Supercomputing Center (BSC), specialized in high-performance computing (HPC) [7].

This model studied is used for various organizations like the Barcelona Dust Forecast Center under an initiative of the World Meteorological Organization [8], consuming thousands of hours of supercomputer every year only on Marenostrum supercomputer. However, The Multiscale Online Nonhydrostatic AtmospheRe CHemistry model (NMMB-MONARCH) is not using all the supercomputing resources that BSC has.

Marenostrum is divided into two supercomputers with different system architectures: Marenostrum IV (MN4) and CTE-POWER. The difference between these two architectures is that CTE-POWER is a GPU-Accelerated cluster, meanwhile, Marenostrum IV only uses the CPU for computation [9]. NMMB-MONARCH lacks any GPU code implementation, so it is only executed on Marenostrum IV. In fact, more of the environmental systems are only capable to execute on the CPU.

For many years, this state of the art was accepted by the scientific community since the GPU has low computational potential in front of the CPU, plus the added difficulty of programming for GPU. However, with the coming of new more powerful GPUs and his focus on Big Data [10], GPUs are earning advantage from the CPU on supercomputers. Many studies demonstrate a better efficiency of this GPU supercomputer in front of the old CPU based supercomputers [11], and more powerful GPU-Accelerated supercomputers are appearing [12].

For these reasons, we found interesting adapting the environmental model NMMB-MONARCH to the new and highly potential GPU architectures, in order to reduce

his execution time. However, translating code from CPU to GPU is not easy, and translating only one high time-consuming function of NMMB-MONARCH can take a lot of work. For this reason, we consider a heterogeneous implementation as the most optimal option, porting to GPUs one part of NMMB-MONARCH.

In consequence, we evaluate the different components which compound NMMB-MONARCH, searching only the most time-consuming parts. The result of this search is that the chemistry part is consuming a lot of the model execution time. Moreover, this chemistry model has a new, more adaptive and more potential comfortable option to solve (currently in development), *The Phlexible Module for Chemistry (phlex-chem)*. These elements make the *phlex-chem* module a very interesting option to study for GPU computation.

1.1 Objectives

In this master thesis we have two main objectives:

1. Reduce the execution time of NMMB-MONARCH.
2. Study the potential of GPU implementations on Earth Science models.

Both objectives are related to each other. We will study the potential of GPU by adapting the costly time execution code of NMMB-MONARCH to GPU, searching for a reduction of the total execution time. To achieve these objectives, we need to perform a series of tasks:

- Select the most time-consuming NMMB-MONARCH component by analyzing the model from a computational point of view.
- Study the efficiency of porting this component selected to GPU, in order to evaluate if we can improve the time execution using a GPU implementation.
- Propose possible GPU implementations and general optimizations for both GPU and CPU implementations.
- Apply these ideas and study the time results obtained, arguing the expected improvement and the one obtained.
- Evaluate the heterogeneous implementation, including transfers from CPU to GPU and GPU to CPU.

We divide this Master thesis into sections following these tasks described. In section 2, we have a brief description of NMMB-MONARCH and his components. In section 3 we explain the methodology used during the project and present the testing environment. In section 4 we present the impact on-time execution of NMMB-MONARCH modules. In section 5 we do an overview of the *phlex-chem* module. In section 6 we explain the differential equation solver CVODE. In section 7 we define some *phlex-chem* concepts using during the project, present

the *phlex-chem* workflow and the first measures of *phlex-chem*. In section 8 we do an analysis and optimizations for *phlex-chem* relevant functions. In section 9 we explain the most relevant optimization done in *phlex-chem*. Finally, section 10 contains the final conclusions and future work.

Chapter 2

An introduction to NMMB-MONARCH

The Multiscale Online Nonhydrostatic AtmospheRe CHemistry model (NMMB-MONARCH) is a fully online multiscale chemical weather prediction system for regional and global-scale applications developed at the Barcelona Supercomputing Center [13].

The model couples online a meteorological driver with the gas-phase and aerosol continuity equations to solve the atmospheric chemistry processes in detail. The model is designed to account for the feedbacks among gases, aerosol particles, and meteorology.

A description of the major components of NMMB-MONARCH follows.

2.1 Meteorology

The meteorological driver of the NMMB-MONARCH is the Nonhydrostatic Multiscale Model on the B-grid [14] developed at NCEP. The NMMB-MONARCH is conceived for short- and medium-range forecasting over a wide range of spatial and temporal scales, from large-eddy simulations (LES) to global simulations. Its unified nonhydrostatic dynamical core allows for running either regional or global simulations, both including embedded regional nests. The NMMB-MONARCH has been developed within the Earth System Modeling Framework (ESMF) [15].

2.1.1 Dynamics

The Dynamical core of the NMMB is described in [13]. The NMMB has been developed following the general modeling philosophy of the NCEP regional Weather Research and Forecasting (WRF) Nonhydrostatic Mesoscale Model [16]. The numerical schemes used in the model were designed following the principles presented in Janjic [16] [17].

2.1.2 Physics

The NMMB has several different physical packages, the operational configuration includes: (1) the Mellor-Yamada-Janjic (MYJ) level 2.5 turbulence closure for the treatment of turbulence in the Planetary Boundary Layer (PBL) and in the free atmosphere [18], (2) the surface layer scheme based on the Monin-Obukhov similarity theory [19] with introduced

viscous sublayer over land and water [20], (3) the NCEP NOAH [21] or the LISS land surface model [22] for the computation of the heat and moisture surface fluxes, (4) the GFDL or RRTMG long-wave and shortwave radiation package [23], (5) the Ferrier grid-scale clouds and microphysics [24], and (6) the Betts-Miller-Janjic convective parameterization [25]. Vertical diffusion is handled by the surface layer scheme and by the PBL scheme. Lateral diffusion is formulated following the Smagorinsky non-linear approach [26].

2.2 Chemistry

2.2.1 Emissions

The High-Elective Resolution Modelling Emission System version 3 [27] is used to pre-process the anthropogenic, biomass burning, soil, and ocean emissions for the NMMB-MONARCH model. HERMESv3 is an open-source, parallel and stand-alone multiscale atmospheric emission modeling framework that processes gaseous and aerosol emissions for use in atmospheric chemistry models. The user can flexibly define combinations of existing up-to-date global and regional emission inventories and apply country-specific scaling factors and masks. Each emission inventory is individually remapped onto the desired destination grid and processed using user-defined vertical, temporal and speciation profiles that allow obtaining emission outputs compatible with multiple chemical mechanisms. The selection and combination of emission inventories and databases are done through detailed configuration files.

The biogenic emissions are computed online within NMMB-MONARCH by using the Model of Emissions of Gases and Aerosols from Nature [28] version 2.04. MEGAN estimates the net emission rate of gases (more than 130 Non-Methane Volatile Organic Compounds (NMVOCs) and aerosols from terrestrial ecosystems into the above-canopy atmosphere. The model uses the 2-meter surface temperature and shortwave incoming radiation from the meteorological driver.

2.2.2 Photolysis

To compute the photolysis rates, the NMMB-MONARCH uses online the Fast-J [29] photolysis scheme. Fast-J has been coupled with the physics of each model layer (e.g., clouds and absorbers such as O3). The optical depths of grid-scale clouds from the atmospheric drivers are considered by using the fractional cloudiness based on relative humidity. The Fast-J scheme has been upgraded with CB05 photolytic reactions. The quantum yields and cross-section for the CB05 photolysis reactions have been revised and updated following the recommendations of Atkinson [30] and Sander [31].

2.2.3 Dry deposition

The dry deposition scheme follows the classical deposition velocity analogy, enabling the calculation of deposition fluxes from airborne concentrations. The aerodynamic resistance (Ra; depends only on atmospheric conditions) and the quasi laminar sublayer resistance (Rb; depends on friction velocity and molecular characteristics of gases) are computed following their common definition [32], while the canopy or surface resistance (Rc) is derived from the resistances of the surfaces of the soil and the plants. The properties of the plants are determined using land-use data (from the land use of the meteorological driver) and depending on the season. The surface resistance also depends on the diffusion coefficient, reactivity, and the water solubility of reactive trace gases.

The aerosol dry deposition is based on [33], which includes simplified empirical parameterizations for the deposition processes of Brownian diffusion, impaction, interception and gravitational settling detailed in [34]. Aerosol rebound at the surface is not taken into account due to limited knowledge of this process.

2.2.4 Wet deposition

The gas-phase cloud-chemistry processes are included in the system considering both sub-grid and grid-scale processes following [35]. The processes included are scavenging, vertical mixing and wet-deposition. Only in-cloud scavenging is considered currently.

Wet scavenging of aerosols by precipitation is computed separately for convective and grid-scale (stratiform) precipitation. It represents the most efficient process for the deposition of the smallest particles. The model includes parameterizations for in-cloud scavenging, and for below cloud scavenging. A detailed description of the schemes can be found in [36].

2.2.5 Chemistry mechanisms

Three different solutions are implemented in NMMB-MONARCH to solve a chemistry mechanism: (1) Fix code, (2) Partially run-time configured mechanism, and (3) Complete run-time configured mechanisms. The fix implementation uses the Carbon Bond 2005 chemical mechanism [37] extended with Toluene and Chlorine chemistry. The CB05 is well formulated for urban to remote tropospheric conditions and it considers 51 chemical species and solves 156 reactions.

1. **EBI:** An efficient and fast solver based on the Euler-Backward-Iterative (EBI) scheme.

2. **KPP:** A chemical mechanism and chemistry solver based on the *Kinetic PreProcessor* (KPP) package [38] with the main purpose of maintaining some flexibility when configuring the model.
3. ***Phlex-chem*** (provisional name)**:** A novel option of adding run-time configured mechanisms for gas- and aerosol-phase chemistry and mass transfer from the *Particle-resolved Monte Carlo* (PartMC) software library [39] [40]. This option also allows an abstract non-fixed representation of the aerosols that can be configured during run-time, describing the life cycle of mineral dust, sea-salt, black carbon, organic matter (both primary and secondary), sulfate, nitrate, and ammonium aerosols. *Phlex-chem* is also free code accessible at the chem_mod GitHub PartMC branch [41].

Chapter 3

Methodology

3.1 Test Environment

During this project, all the tests and executions were performed in two different HPC's [9]:

1. MareNostrum4 (MN4)
2. CTE-POWER

3.1.1 Marenostrum4

The MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors, Lenovo SD530 Compute Racks, a Linux Operating System and an Intel Omni-Path interconnection. See below a summary of the general-purpose cluster system:

Hardware

- Peak Performance of 11.15 Petaflops
- 384.75 TB of main memory
- 3,456 nodes:
    - 2x Intel Xeon Platinum 8160 24C at 2.1 GHz
    - 216 nodes with 12x32 GB DDR4-2667 DIMMS (8GB/core)
    - 3240 nodes with 12x8 GB DDR4-2667 DIMMS (2GB/core)
- Interconnection networks:
    - 100Gb Intel Omni-Path Full-Fat Tree
    - 10Gb Ethernet

Software

- Operating System: SUSE Linux Enterprise Server 12 SP2
- Compiler: ICC version 17.0.4

3.1.2 CTE-POWER

CTE-POWER cluster is a cluster-based on IBM Power9 processors, with a Linux Operating System and an Infiniband interconnection network. It's main characteristic is the availability of 4 GPUs per node, making it an ideal cluster for GPU accelerated applications.

Hardware

2 login node and 52 compute nodes, each of them:

- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node)
- 512GB of main memory distributed in 16 dimms x 32GB @ 2666MHz
- 2 x SSD 1.9TB as 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

Software

- Operating system: Red Hat Enterprise Linux Server 7.5 (Maipo).
- CPU Compiler: GCC version 6.4.0
- GPU Compiler: NVCC version 9.1

3.2 Work

This section explains the methodology used in this work. The sections of this project mostly follow the methodology explained.

First, since NMMB-MONARCH is divided into various independent components as we mentioned in section 2, we must select one to start. We investigate the time consumption of each NMMB-MONARCH component, in order to identify the most time-consuming module. We advance that chemistry module is the most time-consuming part, and from the different chemistry solver options, we select to work on *phlex-chem*. More details in section 4.

Second, since *phlex-chem* is not actually totally prepared to be executed on NMMB-MONARCH, we start to work with the tests defined on *phlex-chem*. Specifically, we tested *phlex-chem* execution using a test that simulates the default mechanism solved on

NMMB-MONARCH by the other solvers (CB05). This test also allows us to compare the time execution of KPP and EBI solvers.

In addition, we use the CB05 test to identify the most time-consuming functions of *phlex-chem*, following the same idea of NMMB-MONARCH components measurements. We identify that the functions Derivative and Jacobian consume a relevant time percentage of the model, and decide to start optimizing them. More details in section 7

Third, we develop a simpler version of the Derivative function, in order to search for optimizations and find out the advantages of computing it on the GPU. We adapt the GPU implementation and optimizations into the real *phlex-chem* Derivative function, obtaining a positive speedup for big mechanisms that includes a lot of reactions. More details in section 8.

Fourth, according to the first results, where the GPU implementation works better when more computation is done on the GPU side, develop a new optimization, that we call *N cells*. This optimization reduces the *phlex-chem* calls from NMMB-MONARCH by sending all the data that has no dependencies on each other. We measure the results of this optimization on a simple test that emulates the NMMB-MONARCH data structure and calls to *phlex-chem*. More details in section 9.

Finally, we draw the conclusions of this project and propose future work.

Chapter 4

Impact of NMMB-MONARCH modules on-time execution

To complete the objective of improving the performance of NMMB-MONARCH, we check first the time consumption of each module component. The chemistry solver option used by default is the EBI solver.

4.1 Configuration

To configure one run of NMMB-MONARCH we used the tool Autosubmit version 3.1 [42] developed at BSC. We almost used all the default configuration for a new NMMB-MONARCH experiment. We just changed a few parameters, especially we increase the number of days to let all the components work.

Parameter	Value
CELLS_WEST_EAST	351, 341, 201
CELLS_NORTH_SOUTH	271, 351, 201
VERTICAL_LEVELS	24
RESOLUTION	0.2
TIME_STEPS	45,20,6
NUM_DOMAINS	1
CHUNKSIZEUNIT	day
NUMCHUNKS	2
DO_PARTMC	false
MPI_PROCESSES	68

Table 1. Summary of the NMMB-MONARCH configuration

4.2 Measurements

To have an idea of the chemistry time consumption, in figure 1 we can see the percentage of time consumed for each module, using the faster chemistry solver (EBI) and CB05 mechanism. As we can see, the chemistry module is taking a lot of execution time in comparison with other modules, even using the faster solver.

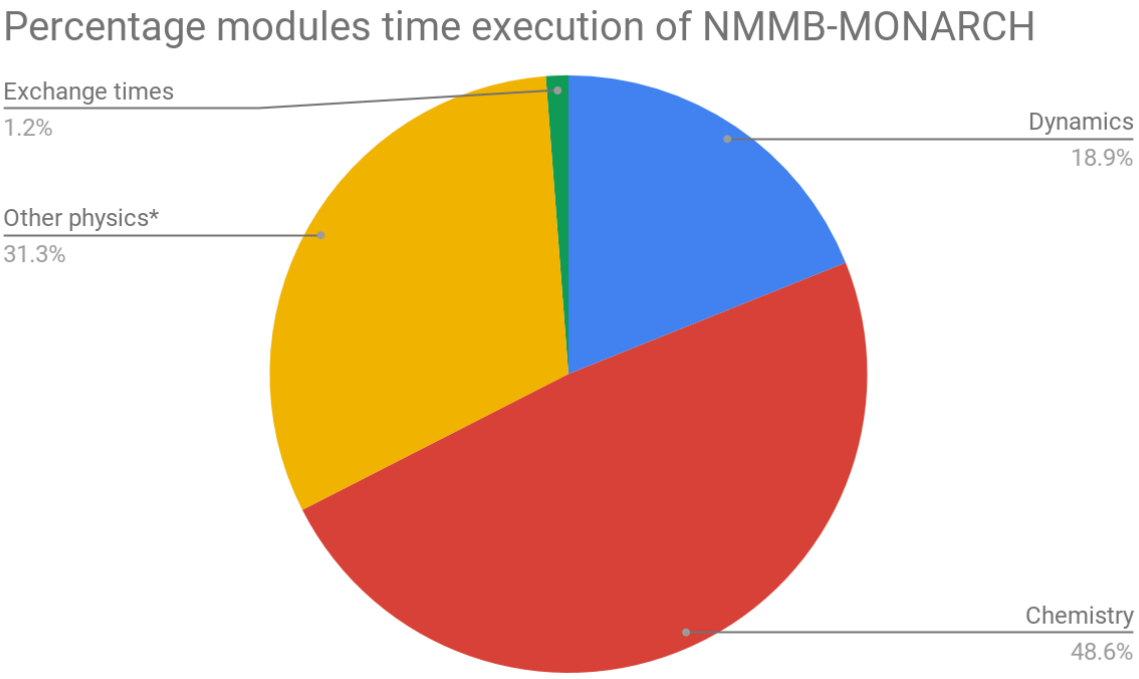


*Figure 1. Comparison of modules time percentages of NMMB-MONARCH. *Other physics includes the physics modules except for chemistry (emissions, photolysis, dry deposition, and wet deposition).*

So in order to improve the performance of NMMB-MONARCH, a nice option is to focus on the chemistry solver. If we consider that this measure comes from the faster solver, we have even more space to improve the performance on the other solvers.

Moreover, *phlex-chem* is the most promising and comfortable solver to develop due to his flexibility, in contrast with the other two options that have a lot of hard code. For these reasons, in this project choose to optimize the slower and most promising solver: *phlex-chem*

Chapter 5

Phlex-chem overview

The Phlexible Module for Chemistry (phlex-chem) is a module within PartMC designed to provide a flexible framework for incorporating chemical mechanisms into PartMC or another host model. A key feature of this novel framework is an abstracted aerosol representation that allows the same chemical mechanism to be solved on models with different aerosol representations (e.g., binned, modal, or particle-resolved)

The flexibility of the model is achieved through a combination of JSON input files and run-time model configuration. The JSON format is widely used for structured data, and allows entire gas- and aerosol-phase chemical mechanisms to be described in a human-readable format, with as much complexity as is required to describe the system.

Defining it as simple as possible, all chemistry models try to predict the concentrations of some chemical reactants (for example,O_3or H_2O) by solving some set of equations. The chemical reactants are also called *species*.

The equations to solve are composed of the concentrations of the reactants $[X_1 X_2 \dots X_n]$and the rate constants that affect these concentrations $[r_1 r_2 \dots r_n]$. As an example, a chemical equation can have the following form:

$$r_1 X_1 + r_2 X_2 \; -> \; r_3 X_3$$

As we can see, the unknowns of the equations are the rate constants. These rate constants are calculated by different equations called reactions. The set of all system reactions is called a *mechanism*. These concepts are defined more fully below from a *phlex-chem* perspective.

5.1 Mechanism and reactions

A reaction represents a transformation of the model state due to a physical or chemical process that occurs within a phase (gas or aerosol) or across the interface between two phases. In the *phlex-chem* model, reactions are grouped into mechanisms, which are solved over time-steps specified by the host model.

The reactions types included on *phlex-chem* at the moment are:

- Aqueous-phase equilibrium
- Arrhenius
- CMAQ special reaction type for 2HO2 (+ H2O) -> H2O2 (1)
- CMAQ special reaction type for OH + HNO3 -> NO3 + H2O (2)
- Condensed-chase Arrhenius
- Emission
- First-order loss
- Henry's Law phase transfer
- PD-FiTE activity
- Photolysis
- SIMPOL phase transfer
- Troe (fall-off)
- Wet deposition
- ZSR aerosol water

In general, a reaction can be viewed as one or more equations to solve during time-step iteration processes. In this project, we work only with the most common reaction types, composed by one equation, since the others are not used frequently and are harder to translate implementations.

The only reactions tested on this project are the ones used for the computation of the CB05 mechanism: Arrhenius, CMAQ (1), CMAQ (2), photolysis, and troe. As an example, we will explain an Arrhenius-like reaction type.

5.2 Arrhenius reaction

Arrhenius-like reaction equations are calculated as follows:

$$Ae^{(\frac{-E_a}{k_b T})}(\frac{T}{D})^B(1.0 + E * P)$$

where A is the pre-exponential factor ($(\#cm^{-3})^{-(n-1)}s^{-1}$), n is the number of reactants, E_a is the activation energy (J), k_b is the Boltzmann constant (J/K), D (K), B (unitless) and E (Pa^{-1}) are reaction parameters, T is the temperature (K), and P is the pressure (Pa). The first two terms are described by Finlayson-Pitts and Pitts [43]. The final term is included to accommodate EBI solver type 7 rate constants.

Input data for Arrhenius equations has the following JSON format:

```
{
  "type" : "ARRHENIUS",
  "A" : 123.45,
  "Ea" : 123.45,
  "B"  : 1.3,
  "D"  : 300.0,
  "E"  : 0.6E-5,
  "time unit" : "MIN",
  "reactants" : {
    "spec1" : {},
    "spec2" : { "qty" : 2 },
    ...
  },
  "products" : {
    "spec3" : {},
    "spec4" : { "yield" : 0.65 },
    ...
  }
}
```

The key-value pairs reactants, and products are required. Reactants without a *qty* value are assumed to appear once in the reaction equation. Products without a specified yield are assumed to have a yield of 1.0. Note that yield and qty will affect the species concentration, being also reaction parameters.

Chapter 6

Differential equation solver: CVODE

To predict future concentrations we need to solve a time-dependant equation [*y' = f(t,y)*] by adding a time-step to our equation. *Phlex-chem* uses the library CVODE from the SUNDIALS package for this purpose [44].

CVODE is a package written in C for solving initial value problems for ordinary differential equations (ODE). CVODE solves both stiff and nonstiff systems, using variable-coefficient *Adams* and *BDF* (Backward Differentiation Formula) methods [45] [46]. In the highly modular organization of CVODE, the core integrator module is independent of the linear system solvers, and all operations on N-vectors are isolated in a module of vector kernels.

All concepts mentioned below are fully defined in the CVODE and SUITESPARSE documentation [47] [48]. We briefly define the most important concepts and describe the ones used on this project.

6.1 Mathematical considerations

CVODE solves ODE initial value problems (IVPs) in real *N*-space, which we write in the abstract form

$$y' = f(t, y), \;\; y(t_0) = y_0,$$

where $y \in R^N$. Here we use y' to denote dy/dt. While we use *t* to denote the independent variable, and usually this is time, it certainly need not be. CVODE solves both stiff and nonstiff systems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself.

The methods used in cvode are variable-order, variable-step multistep methods, based on formulas of the form

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} y'^{n-i} = 0.$$

Here the y^{n} are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately one of two multistep methods. For nonstiff problems, CVODE includes the Adams-Moulton formulas, characterized by $K_1 = q$ and $K_2 = 0$ above, where the order q varies between 1 and 12. For stiff problems, cvode includes the Backward Differentiation Formulas (BDF) in the so-called fixed-leading coefficient (FLC) form, given by $K_1 = q$ and$K_2 = 0$with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$[49] [50].

For either choice of formula, a nonlinear system must be solved (approximately) at each integration step. This nonlinear system can be formulated as either a root-finding problem

$$F(y^{n}) \equiv y^{n} - h_n \beta_{n,0} f(t_n, y^{n}) - a_n = 0,$$

or as a fixed-point problem

$$G(y^{n}) \equiv h_n \beta_{n.0} f(t_n, y^{n}) + a_n = y^{n}.$$

Where

$$a_n \equiv \sum_{i>0} (\alpha_{n,i} y^{n-1} + h_n \beta_{n,i} y'^{n-1}).$$

By default, CVODE solves it with a Newton iteration which requires the solution of linear systems

$$M[y^{n(m+1)} - y^{n(m)}] = -F(y^{n(m)}),$$

in which

$$M \cong I - \gamma J,\ J = \partial f/\partial y,\ and\ \gamma = h_n \beta_{n,0}.$$

The exact variation of the Newton iteration depends on the choice of the linear solver. The linear solver modules distributed with SUNDIALS are organized in two families, a *direct* family comprising direct linear solvers for dense, banded, or sparse matrices, and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [51] [49] or the thread-enabled SuperLU_MT sparse solver library [53] [54] (serial or threaded vector modules only),
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver,
- PBCGS, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- PCG, a scaled preconditioned CG (Conjugate Gradient method) solver.

6.2 CVODE organization

The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for this project.

The overall organization of the cvode package is shown in figure 2. The central integration module, implemented in the files *cvode.h*, *cvode impl.h*, and *cvode.c*, deals with the evaluation of integration coefficients, estimation of local error, selection of step size and order, and interpolation to user output points, among other issues.

CVODE utilizes generic linear and nonlinear solver modules defined from SUNDIALS package by the SUNLINSOL API and SUNNONLINSOL API respectively [55]. As such, CVODE has no knowledge of the method being used to solve the linear and nonlinear systems that arise. For any given user problem, there exists a single nonlinear solver interface and, if necessary, one of the linear system solver interfaces are specified, and invoked as needed during the integration.

At present, the package includes two linear solver interfaces. The primary linear solver interface, CVLS, supports both direct and iterative linear solvers built using the generic SUNLINSOL API. These solvers may utilize a SUNMATRIX object for storing Jacobian information, or they may be matrix-free. Since CVODE can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to CVODE will expand as new SUNLINSOL modules are developed.

CVODE's linear solver interface consists of four primary phases, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, and only as required to achieve convergence.

All state information used by cvode to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the cvode package, and so, in this respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the cvode memory structure. The reentrancy of cvode was motivated by the anticipated multicomputer extension but is also essential in a uniprocessor setting where two or more problems are solved by intermixed calls to the package from within a single user program.

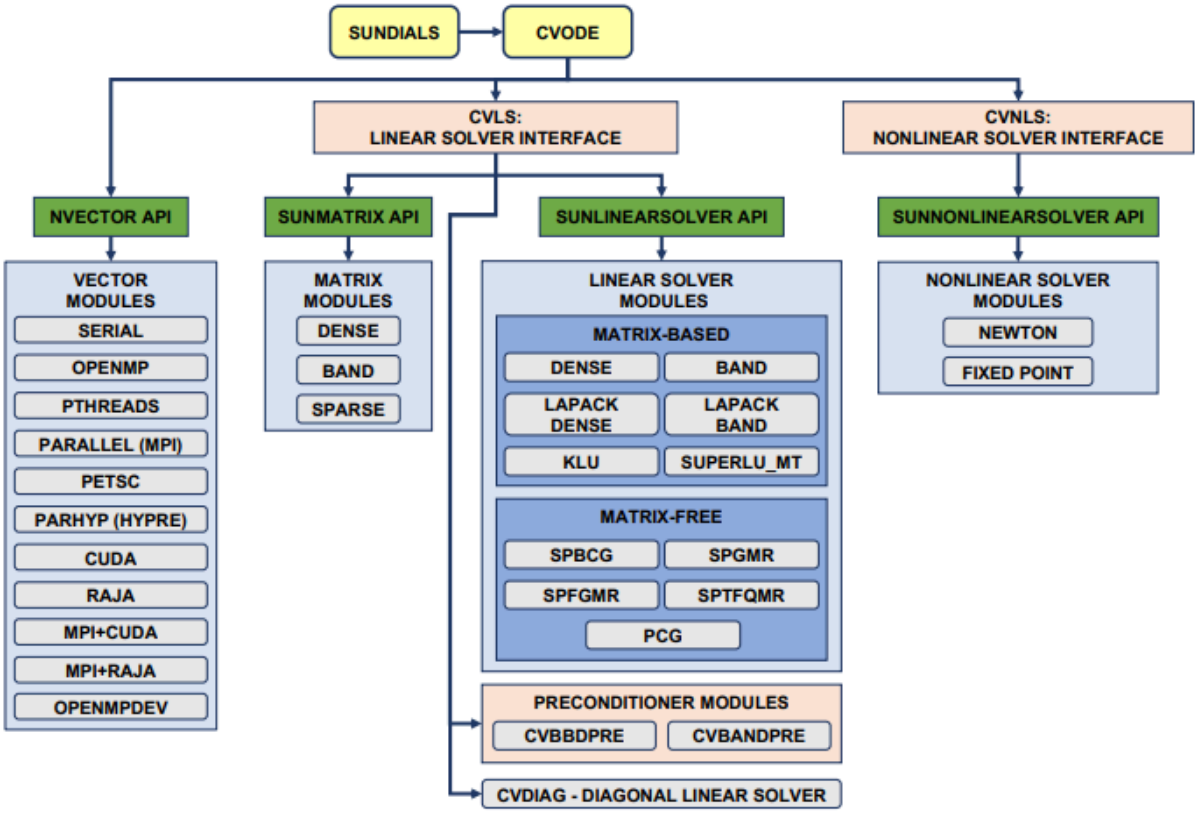


Figure 2: Overall structure diagram of the cvode package. Modules specific to cvode begin with "CV" (CVLS, CVDIAG, CVBBDPRE, CVBBDPRE, CVBANDPRE, and CVNLS), all other items correspond to the generic solver and auxiliary modules. Note also that the LAPACK, KLU and SUPERLU_MT support is through interfaces to external packages. This package is downloaded and compiled independently.

6.3 CVODE *phlex-chem* implementation: KLU SPARSE

From all the options available in figure 2, the ones chosen are the following: Serial, SPARSE, and KLU. These options were chosen at the start of the *phlex-chem* developers, and we will continue the study following the same options. The reason is that these options are interesting to *phlex-chem* since they are optimized to solve systems of equations that vary a lot on the number of parameters (a consequence of the fact that on *phlex-chem* the user can define the number of parameters without limit).

SERIAL option means that there is no parallelization during the solving. In this section, we explain the SPARSE structure and the KLU solving method. Notice that this KLU method is a matrix-based method, meaning that we need to calculate the Jacobian of the system ($J = \partial f / \partial y$) each time-step iteration, unlike other methods that do not need it.

6.3.1 SPARSE structure

A sparse matrix is defined as one that has few nonzeros in it. The quantification of the adjective 'few' is not specified. The decision of using this structure or not depends on the fill-in properties of the matrices (decomposition of the model domain and the number of neighbors).

However, sparse matrices typically have O(n) nonzero entries. Dense matrices are typically represented by a two-dimensional array. The zeros of a sparse matrix should not be stored if we want to save memory. This fact makes a two-dimensional array unsuitable for representing sparse matrices. Sparse matrices are represented with a different kind of data structure. They can be represented in two different data structures: column compressed form or row compressed form.

A column compressed form consists of three vectors Ap, Ai, and Ax. Ap consists of column pointers. It is of length n + 1. The start of column k of the input matrix is given by Ap [k].

Ai consists of row indices of the elements. This is a zero-based data structure with row indices in the interval [0,n). Ax consists of the actual numerical values of the elements.

Thus the elements of a column k of the matrix are held in Ax [Ap [k]...Ap [k+1]). The corresponding row indices are held in Ai [Ap [k]...Ap[k+1]). Equivalently, a row compressed format stores a row pointer vector Ap, a column indices vector Ai and a value vector Ax. For example, the matrix:

$$\begin{bmatrix} 5 & 0 & 0 \\ 4 & 2 & 0 \\ 3 & 1 & 8 \end{bmatrix}$$

when represented in column compressed format will be

Ap: 0 3 5 6
Ai: 0 1 2 1 2 2
Ax: 5 4 3 2 1 8

and when represented in row compressed format will be

Ap: 0 1 3 6
Ai: 0 0 1 0 1 2
Ax: 5 4 2 3 1 8

Let *nnz* represent the number of nonzeros in a matrix of dimension $n * n$. Then in a dense matrix representation, we will need n^2 memory to represent the matrix. In sparse matrix representation, we reduce it to $O(n + nnz)$ and typically $nnz << n^2$.

6.3.2 KLU method

This section describes the SUNLINSOL implementation for solving sparse linear systems with KLU. The SUNLINSOL_KLU module is designed to be used with the corresponding KLU sparse matrix type.

The SUNLINSOL_KLU module is a SUNLINSOL wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [51] [52]. Additionally, this wrapper only supports double-precision calculations.

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLINSOL_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse pivot information. This routine also returns diagnostic

information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_KLU module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.

- On subsequent calls to the “setup” routine, it calls the appropriate KLU “refactor” routine, followed by estimates of the numerical conditioning using the relevant “rcond”, and if necessary “condest”, routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.

- The module includes the routine SUNKLUReInit, that can be called by the user to force a full or partial refactorization at the next “setup” call.

- The “solve” call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

Chapter 7

Phlex-chem on NMMB-MONARCH: Highlighted concepts

Before starting to enter in a *phlex-chem* analysis, we consider appropriate remember the most relevant concepts from before and make some definitions.

7.1 Important definitions

In this section we will remember and name some concepts used on *phlex-chem* to make easy reference them later:

- **Species**: Chemical reactant presented on the system. Examples: H20, O3, NO2…
- **State**[X_1… [X_N]: Array containing the species concentrations of the system.
- **Reaction:** Equation defined by the user, read as JSON input.
- **Mechanism**: Group of all the reactions of the system.
- **Rates** [r_N]**:** Results of the reactions defined. Each rate can affect one or more species concentration. These rates can be affected by environmental variables like temperature and pressure.
- **F(y)**: The chemical equations of the species relations, following the structure $r_1X_1 + r_2X_2 \; -> \; r_3X_3$.
- **Derivative**: Solution of the equation F(y). CVODE needs these results to apply the time factor to calculate y' = f(t,y).
- **Jacobian:** `Jacobian matrix (∂f /∂y). CVODE reads the results to apply the time factor to calculate y' = f(t,y).`
- **Cell:** Set of all system variables (state, rates, etc). A cell represents an area of the world map, usually very small, depending on the precision that the NMMB-MONARCH user configured. NMMB-MONARCH always computes a large number of cells, computing a bigger map area with high precision. The number of cells to compute is divided into N cores on the cluster, letting each core to process typically 10,800 cells. Each cell has its own values, and are independent of each other.

7.2 Workflow

NMMB-MONARCH is divided into two main modules: Meteorology and Chemistry. Also, the NMMB-MONARCH model runs over a number of time-step iterations trying to predict the future concentration of chemical species.

Each NMMB-MONARCH iteration, it runs the chemistry solver main of our interest: *Phlex-chem*. This module connects the chemical concentrations with the equations related to them (that can be defined by the user) and calls the external solver CVODE to solve the system for the next time-step. CVODE then initialize his own time-step and starts solving the system.

`CVODE gets the results from chemical calculations ( y' = f(t,y) or derivative function) and the Jacobian (∂f /∂y), try to solve the system, check if convergence, and if not, repeat the process until convergence.`

In figure 3 we summarize the flow described in a diagram.

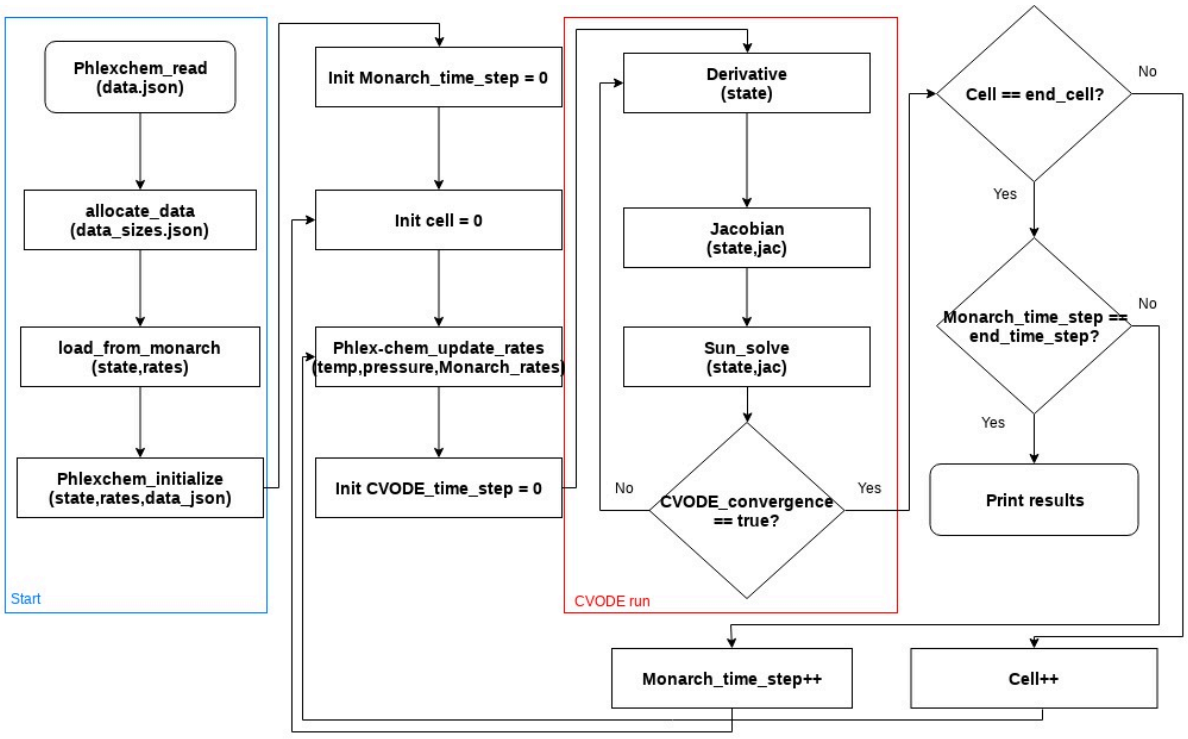


Figure 3. Phlex-chem flow diagram on NMMB-MONARCH. The function Sun_solve includes all the CVODE functions executed between the Jacobian and the check for convergence.

7.3 General measurements

In this section, we present some first measurements done on *phlex-chem* to give a general performance overview of the module. These results are measured on a test that executes *phlex-chem* independently from NMMB-MONARCH. The test uses the CB05 mechanism as input data for the JSON files. The time-step is set at the start (unlike in NMMB-MONARCH) to 100. The number of cells is set to 100. Taking account time-step and number of cells, we have a total of 10,000 solver calls.

Moreover, this CB05 test also includes the execution of KPP and EBI solver. In figure 4 we present the time execution of this solvers and *phlex-chem* in order to evaluate more accurately the time difference between them.

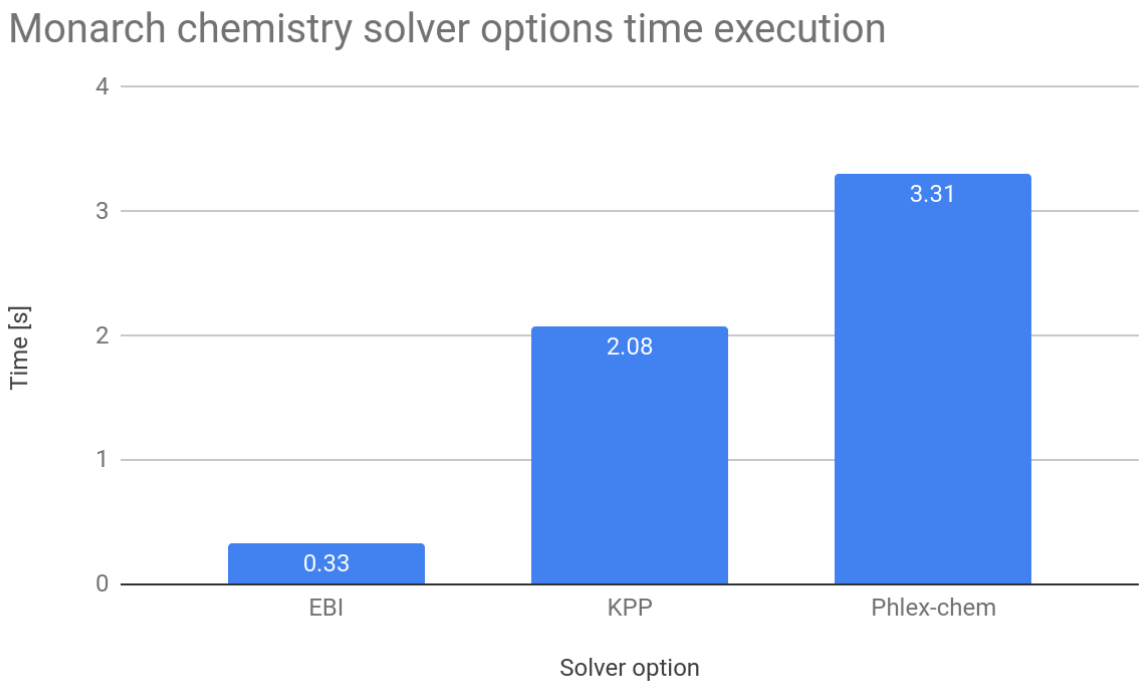


Figure 4. Time execution of NMMB-MONARCH chemistry solver options. These measurements are done in the CTE-POWER cluster with the GCC compiler.

As we can see, EBI is the faster option, followed by KPP and last *phlex-chem*. However, *phlex-chem* is the most flexible solver, and EBI the least one, so for the moment seems more flexibility can mean more time execution, or at least is related to this solvers. Moreover, KPP and *phlex-chem* times seem not so much unrelated (KPP has only 1.59x speedup in front *phlex-chem*), but EBI is much faster than the others (10 times more from *phlex-chem*).

This means that there is a lot of possible time improvement since to solve the CB05 mechanism we only need the 10% time of actual *phlex-chem* time execution. Moreover, we are not far away to overcome the KPP time-execution.

Following the same idea of measure NMMB-MONARCH components, we also measured time executions of the *phlex-chem* functions, especially to know the relevance of Derivative and Jacobian functions into the time execution (the ones that no belongs to the library solver). In figure 5, we can see the percentage of time consumed from the most time-consuming functions:

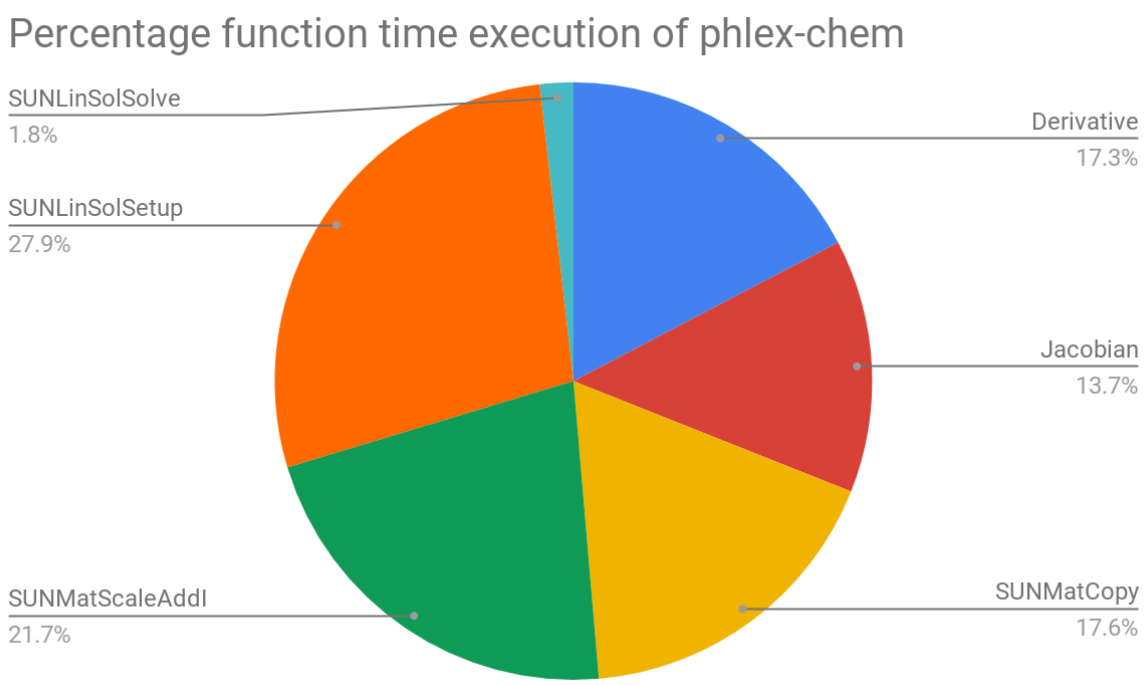


Figure 5. Comparison of function time percentages of phlex-chem with cb05 mechanism. The name functions that start with SUN prefix means that are functions from the CVODE library.

As we can see, the profile is flat among all functions. Moreover, Derivative and Jacobian functions occupy 30% of the program time execution, more than any of the other CVODE functions. So, optimize Derivative and Jacobian time will have a relevant impact on general performance.

However, we must consider the case where this high percentage usage comes from a lot of solver calls instead of our code on Jacobian and Derivative. Executing *phlex-chem* 10,000 times from the cells and time-step loops, we measured the total number of iterations for each function. In figure 6 we can resolve that the Derivative and Jacobian are computing 3-4 and 1-2 iterations per phlex-chem call respectively. Moreover, the number of iterations of all

functions is similar. As a result, we can secure that the high percentage usage does not come from a large number of calls from CVODE.

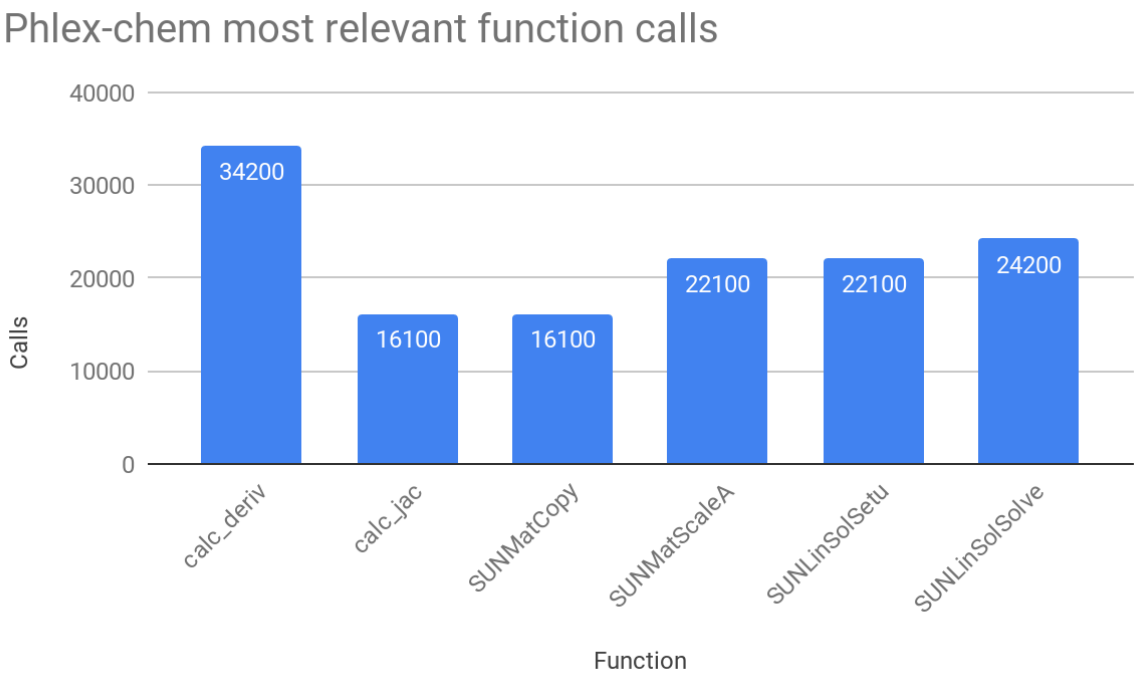


*Figure 6. Most relevant function calls of phlex-chem. Tested with 100*100 cell and time-step loops respectively (10.000 total phlex-chem calls).*

In addition, the time for only one execution of these functions is very low, in the order of µs. In figure 7 we can see this time-executions for one function call.

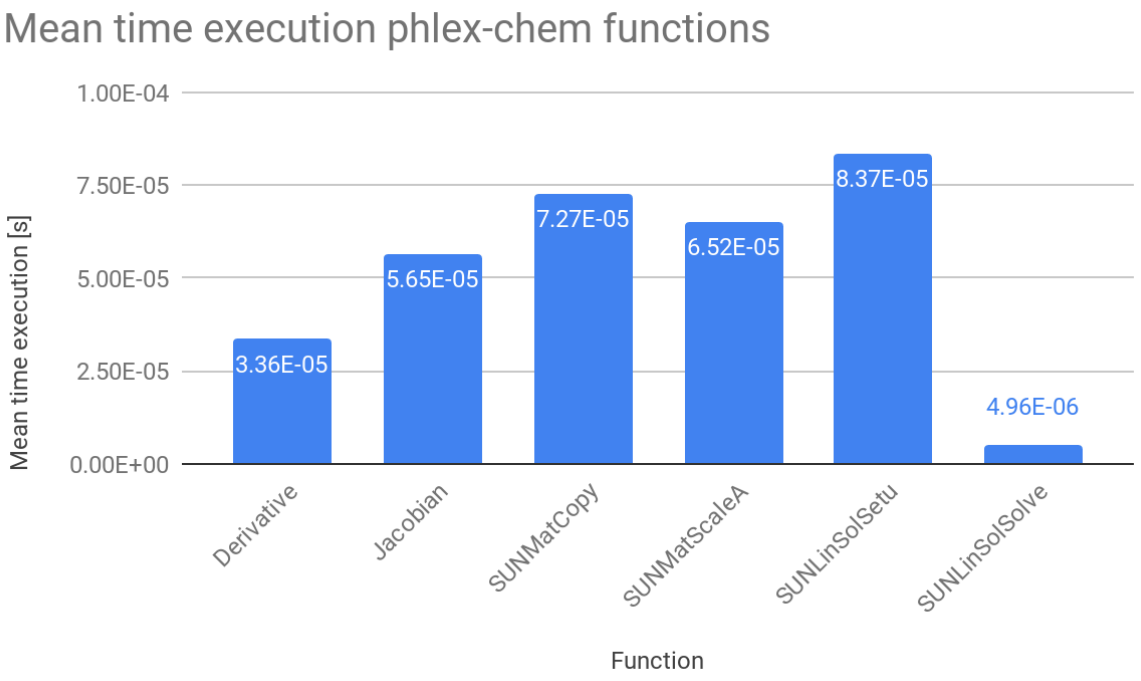


Figure 7. Meantime execution per iteration of most relevant phlex-chem functions.

We also have to mention that either derivative and Jacobian functions have very 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 KLU sparse matrix for the Jacobian) and some extra linear operation. So, we only need to analyze one of them since we can extrapolate the optimization ideas and techniques.

Chapter 8

Phlex-chem: Analysis and optimizations for relevant functions

In this section, we will focus on the derivative function executed on the *phlex-chem* module. We work around the CB05 *phlex-chem* test mentioned in section 7.3.

8.1 Concepts

First, we will describe the variables used in the *phlex-chem* to compute the Derivative function. This includes the data structure used to store the data. All these concepts mentioned are the same for the Jacobian, except for the output name that is called *Jac* instead of *Deriv*:

- **N_CALLS**: Number of Derivative calls.
- **N_SPEC:** Number of state variables (species).
- **N_CTS**: Number of total reaction parameters involved in reactions (mentioned in section 5.2). In NMMB-MONARCH it has a value between the range [3-30].
- **N_RXN:** Number of reactions. There are mechanisms from less than 100 reactions until 10,800 reactions.
- **Header**: Three integer values. The first value represents the type of reaction (Arrhenius, troe, etc.). The second value is the random size of *Deriv_indices*. The third value is the random size of *React_params*.
- **Deriv_indices:** Integer type array, random size between N_CTS range. This value points to random indices of the *deriv* array, simulating the link between the reaction parameters and the species concentrations affected.

- **React_params:** Double type array, random size between N_CTS range. Reaction parameters of the system, initialized at unknown values, depending on the mechanism. They can be yields, rate constants, etc.
- **RXN**: Condensed data array structure composed by the structures *Header*, *Deriv_indexes*, and *React_params* for each reaction. The size is *[N_RXN]* per *[Header+Deriv_Indices+React_params]*, where *Deriv_indices* and *React_params* size has different random value on each reaction.
- **State**: Double type array of species concentrations. Set at unknown values, on each Derivative iteration, depending on the CVODE calculations. Size of N_SPEC.
- **Deriv**: Double type array of species concentrations of the next time step. The output of each iteration. Size of N_SPEC.

In the next figure we can see RXN data structure for one reaction (we will also call it a *data packet* or *row)*:

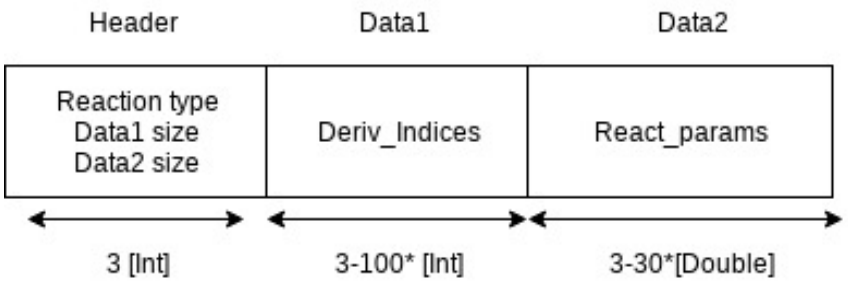


*Figure 8. Data packet structure. *Range of size values, dependent on reaction type.*

An important concept to remark is the equation solved in the Derivative function. This equation is already mentioned in section 5, and is shown again here as a reminder:

$$r_1 X_1 + r_2 X_2 \; -> \; r_3 X_3$$

Putting in context with the data explained, on Derivative function we multiply the rate constants saved on the *React_params* array with the corresponding concentration on state array, filling the next 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 adds a contribution to the state concentration, increasing or decreasing the value.

In summary, the operation calculated is highly studied for the GPU computation, known as a SAXPY (Single-precision real Alpha X plus Y) operation. The formula for this SAXPY operation is the following, considering that X represents the state,α the *React_param*s and variable *i* represents a reaction:

$$y_{i+1} <- \alpha x + y_i,\ i = 0, 1 \ldots N\_RXN$$

We have summed up the data flow of the Derivative function in figure 9.

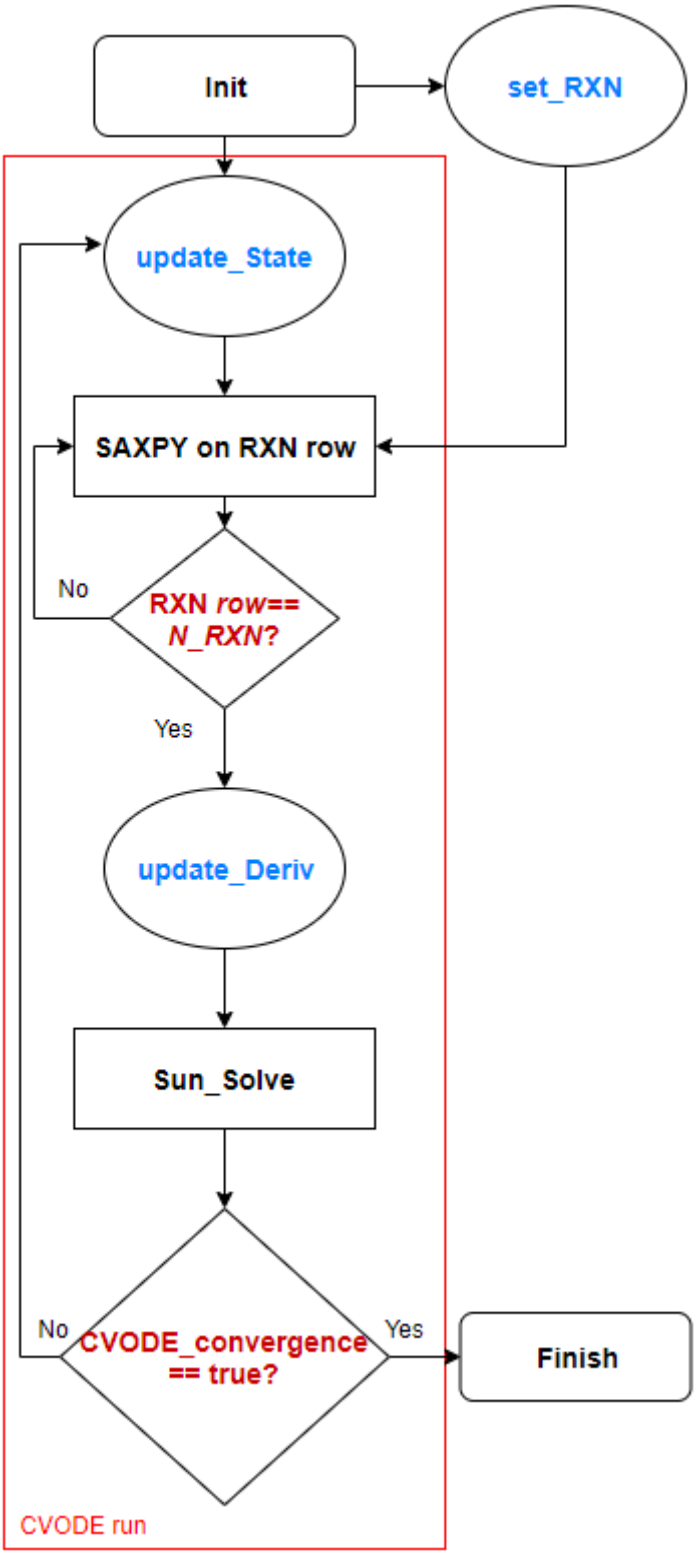


Figure 9. Derivative data flow diagram. Blue names refer to data variables defined in section 7.1. Sun_Solve and CVODE convergence come from the CVODE library.

8.2 GPU implementation strategy

It is important to say that each reaction is independent of each other. Thus, our GPU strategy will parallelize each reaction, or also said, each *row* of the RXN structure. In consequence, the total number of threads will be equal to the number of reactions. In figure 10, we can see the resulting flow diagram.

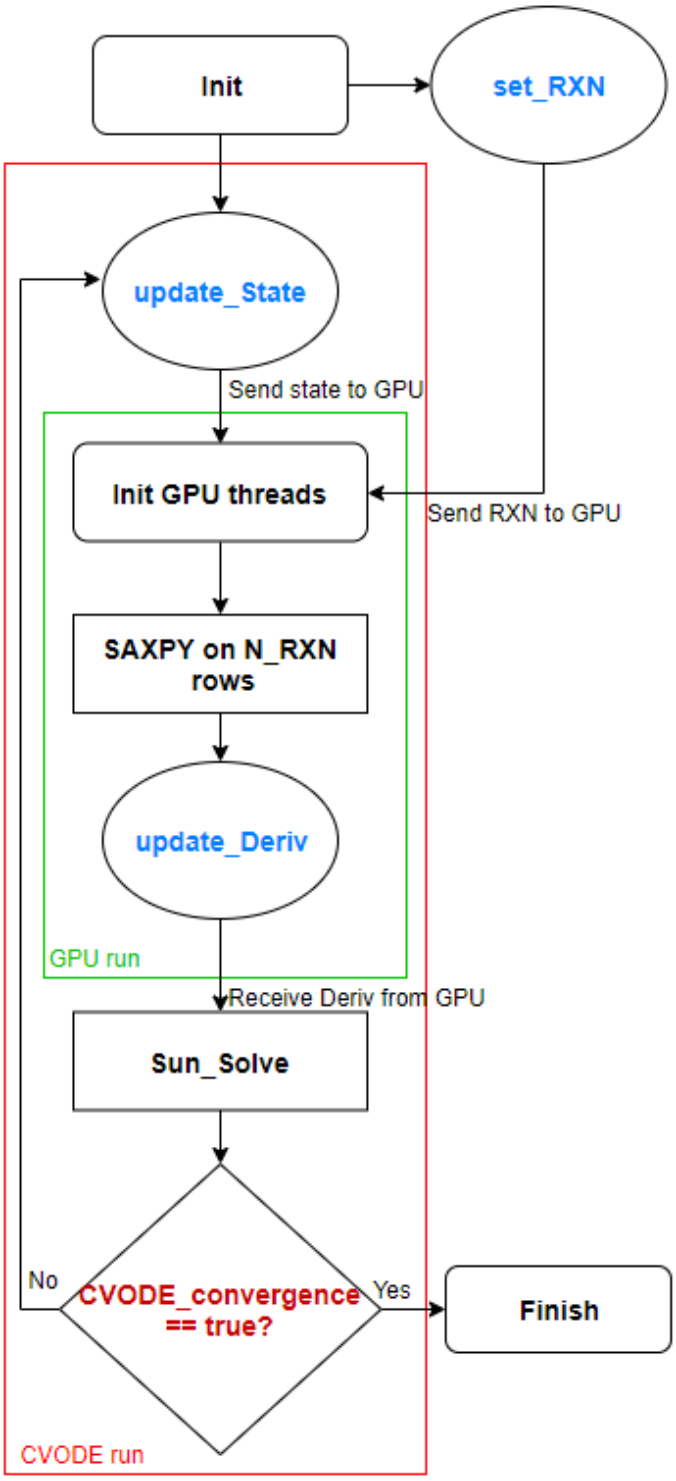


Figure 10. Derivative data flow diagram on the GPU. Blue names refer to data variables defined in section 7.1. Sun_Solve and CVODE convergence come from the CVODE library.

To do this parallelization, we extract the data sizes from the RXN *Header*. On CPU is not necessary since we are accessing RXN sequentially, reading at the same time the data to process and the pointer for the next reaction to the process. But on GPU all threads access to

his own reaction, so we need to assign a different pointer to each thread before starting to work with the RXN.

However, GPU does not work fine with the mixed data type used in a *row*. Mixing integer and doubles (data types that occupy 4 and 8 bytes/unit respectively) raises a miss-aligned error on the GPU when *Deriv_indices* size is not divisible by double size. This forces us to separate each *row* in integer and double arrays, or put another way, in an array with *Header* plus *Deriv_indices* and another array with *React_params*.

Also, remember from section 8.1 that the SAXPY operation done by each reaction adds his own contribution to the corresponding place on the *deriv* array, but we can have more reactions that state concentrations. In consequence, some threads will try to add his contribution to the same array value. To solve this, we compute the SAXPY operation atomically.

RXN structure is allocated on global memory at the initialization of the program. The rest of the input data are passed as a function parameter, taking advantage of the constant memory [56] since it has few data. Output data (*deriv* array) is initialized at zero at the start and transferred from GPU to CPU at the end.

We also added portability for the GPU. During run-time, we consult the GPU specifications, such as the number of threads and blocks, in order to execute correctly the program regardless of the GPU used.

8.3 Measurements & analysis

8.3.1 Basic version

First, we tested the GPU strategy by developing a basic version of the Derivative function that emulates the workflow and functionality of the Derivative on *phlex-chem*. The basic Derivative is a simple script executed without *phlex-chem* or CVODE, by initializing the input data at similar values to CB05 Arrhenius reactions. This version has implementations for CPU and GPU, allowing us to compare the efficiency of GPU between CPU and take a first look at GPU potential.

The configuration for this version is the following:

- **N_CALLS**: Set to 1000.
- **N_SPEC:** Set to 100.
- **N_RXN:** Set to 5000.
- **N_CTS**: Set at random value in the NMMB-MONARCH range [3-30].

The execution times of this basic Derivative function for GPU and CPU on CTE-POWER. We also measured the time for CPU on MN4 in order to search the best time-execution for CPU and compare if we still have a positive speedup. In figure 11 we can see the results.

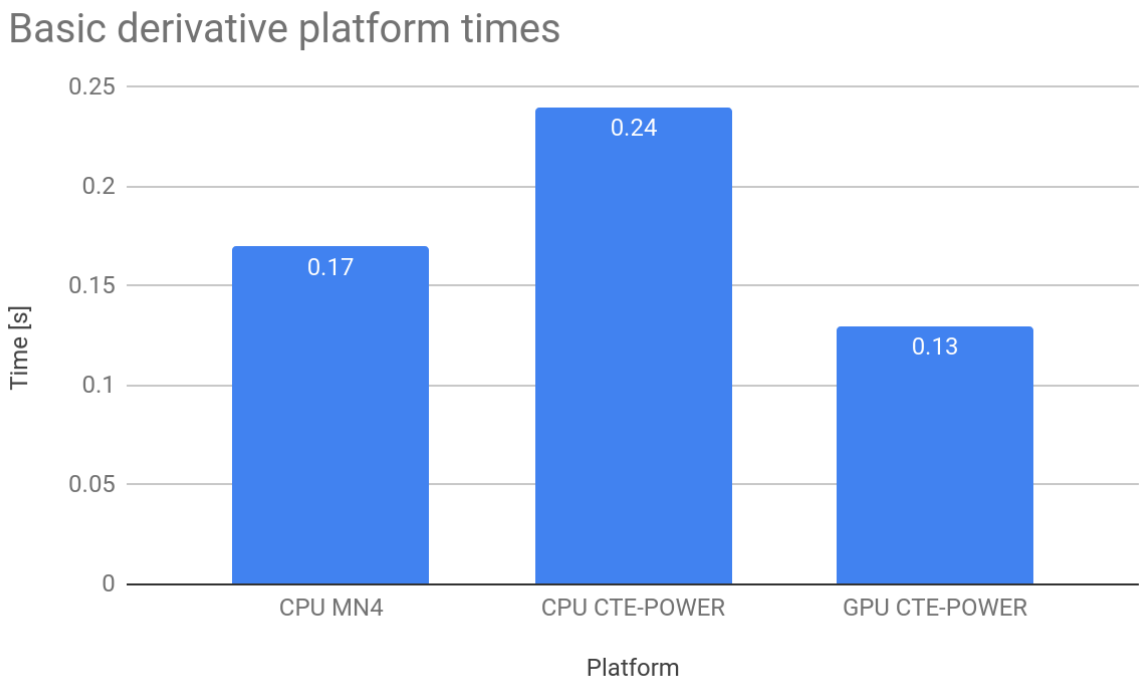


Figure 11. Basic Derivative time executions for different platforms. CPU MN4 execution is compiled with ICC compiler, meanwhile, CPU CTE-POWER is compiled with GCC and GPU CTE-POWER with NVCC.

Note that the GPU version is 1.3 times better from the best CPU execution (CPU MN4). In addition, on the same cluster (CTE-POWER) we have a better speedup of 1.85. This means we always have a positive speedup for GPU, but we should consider for the next measures that the practical speedup is worst than the measured speedup on CTE-POWER.

Deeping into GPU metrics, with the CUDA profiler Nsight [57] we can see in the next figures that memory is the most used on the system. This implies that memory accesses are the bottleneck of the system.

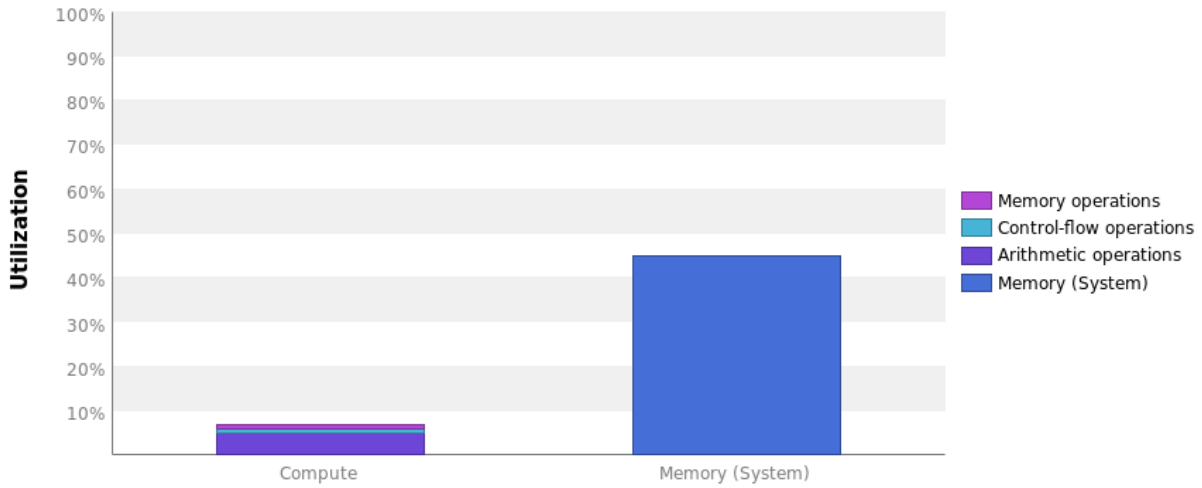


Figure 12. Percentage of utilization for compute and memory operations on GPU for basic Derivative. The utilization is measured considering the maximum GPU capacity.

8.3.2 *Phlex-chem* case: CB05 mechanism

We translate the GPU strategy tested before on the basic version to the Derivative code on *phlex-chem*. The workflow will be the same as the *phlex-chem* flow diagram described in section 7.2. As a reminder, in figure 13 we can see only the part of the CVODE solving, including all the CVODE relevant functions:

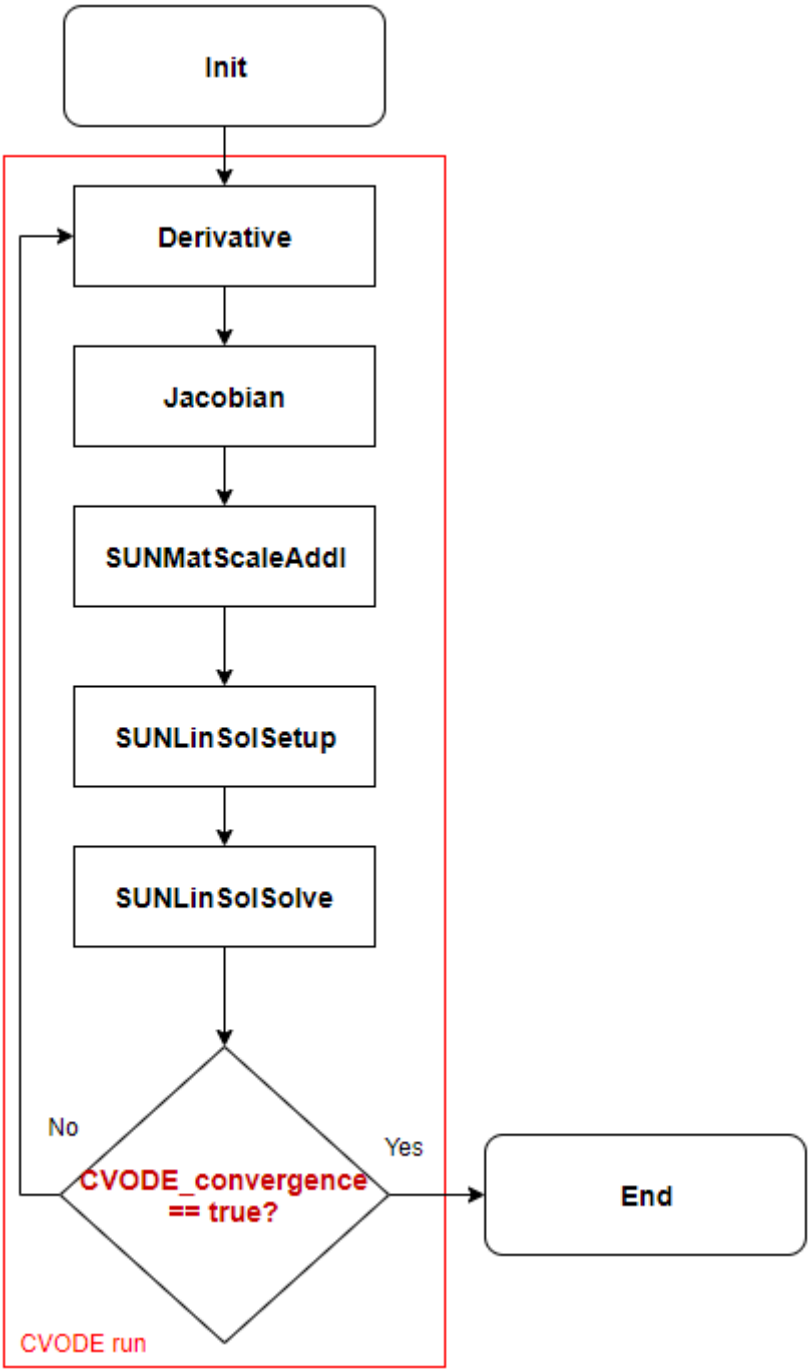


Figure 13. Phlex-chem CVODE run flow diagram. The name functions that start with SUN prefix are functions from the CVODE library.

The configuration is the same as section 7.3 (10,000 calls and CB05 mechanism), with the difference that now we want to test our implementation for cases with a different number of reactions. For example, the CB05 mechanism has 186 reactions, and bigger mechanisms can have till 10,000 reactions.

To test different reaction numbers without modifying the code so much, we duplicate the JSON reaction data from CB05 various times. Thus, the results have no mathematical sense

but are fine to do a scalability test. In figure 14 we can see these results in the form of GPU speedup.

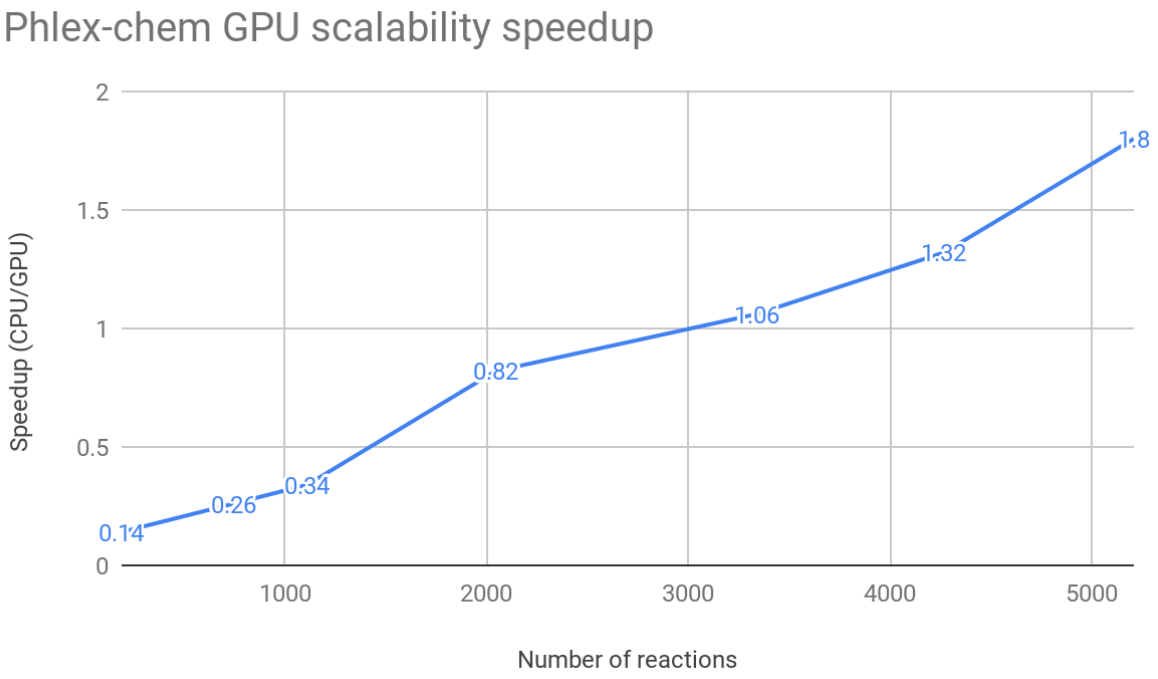


Figure 14. Derivative speedups for different reactions number.

As we can see, the speedup for 5000 reactions is very similar to the basic version (1.8), as we can expect. However, below 2000 reactions we have a negative speedup, having 0.14 speedup on the common mechanism used on NMMB-MONARCH (CB05). Moreover, a mechanism with 2000 reactions or more is considered big and rarely use. Thus, GPU implementation is only profitable for big mechanisms.

Although the results may seem bad, we have to remark that we are working with few data and little time executions per iteration (remember from section 7.3 that the time executions per iteration are in the order of µs). So, the initial results are not really bad. Moreover, in the next section, we will explain an optimization that improves these results a bit.

8.4 GPU optimizations

Remember from section 8.3 that we are parallelizing RXN data packets or *rows*. This *row's* name is because it can be interpreted as *rows* in the case that we have a static matrix. Like the rows of a matrix, in C we need to jump some memory positions to reach the next row [58]. Since we are processing all the rows at the same time, we are quasi loading on the memory all the RXN structure at once.

However, we can improve this data access. To do this, we need to reorder the data to read at the same time aligned memory. So, the first value accessed of each thread should be in the same *row*, instead of the same *column*, the second value accessed on the second *row* and so on. In conclusion, we have to "*reverse*" the RXN *matrix* and transform the *rows* of the old *matrix* into the columns of the new one.

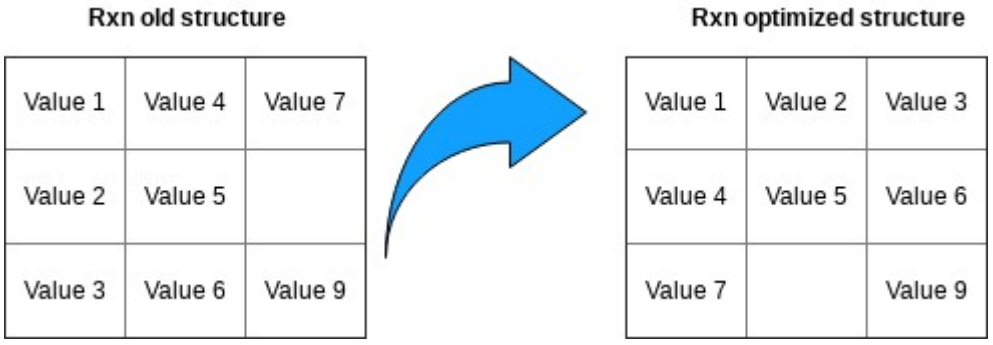


Figure 15. Illustrative image of RXN structure optimization for GPU. The value numbers represent the GPU memory access order.

The results of this optimization are shown in figure 16. Notice also that the Derivative operations only must coincidence on the first memory access, the consequent access depends on each reaction, making it notably random for each reaction and the improve of this optimization depends on the similarities of this memory access.

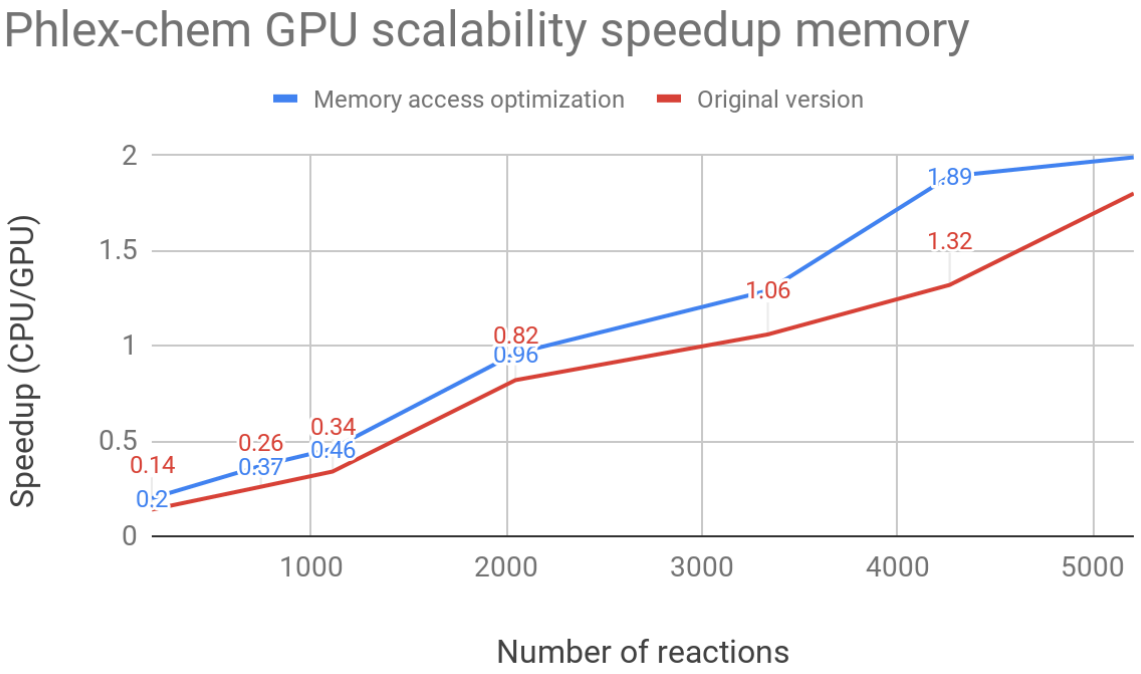


Figure 16. Derivative speedups for different reactions number with RXN memory access optimization.

All the speedups are better from the original version, going from 1.10 to 1.43 better. The average improvement from the speedups measured is 1.3. This is not a bad improvement, but we still have a negative speedup below 2000 reactions.

In conclusion, GPU is only viable computing large amounts of data, as for the case of a mechanism with a big number of reactions. This conclusion leads us to the next section: computing all the data coming from cells at once.

Chapter 9

Phlex-chem: Multiple cells computation

In the previous section, we explained that we need to work with a lot of data at the same time on the GPU to make it profitable. Moreover, in section 7.1, we mention that NMMB-MONARCH computes a lot of independent cells, typically 10,800 cells per core. These cells till now were computed by *phlex-chem* on a cell loop, as we mentioned in section 7.3 where the number of cells is translated to *phlex-chem* iterations.

In this section, we will explain an implementation where we move the cells loop represented in figure 3 to inside the independent functions (Derivative, Jacobian and less relevant functions). In consequence, we are reducing the total number of *phlex-chem* calls. In figure 17, we can see the Derivative data flow for this implementation. Note in advance that this new loop will be avoided for the GPU implementation since there is no dependence between cells.

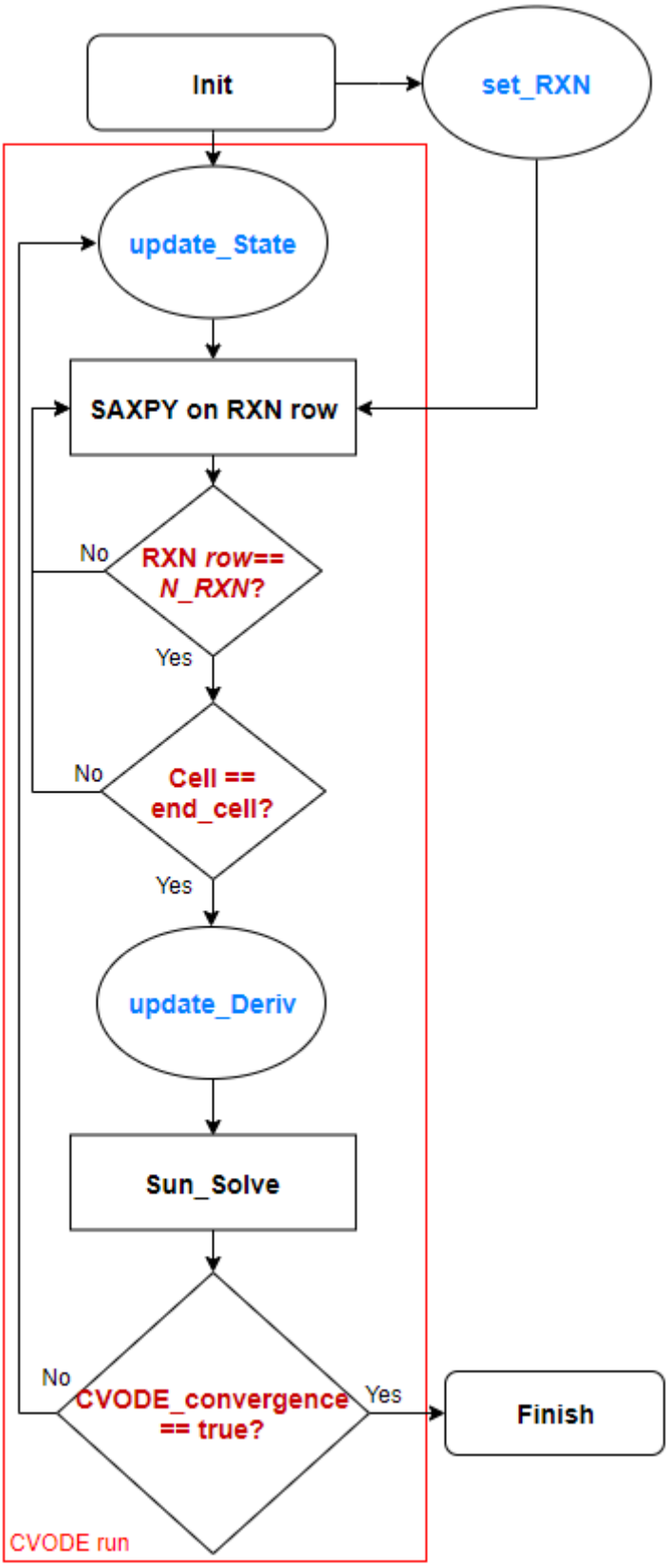


Figure 17. Derivative data flow diagram for N cells implementation. The name functions that start with SUN prefix are functions from the CVODE library. Blue names refer to data variables defined in section 7.1.

9.1 Mock-monarch test

To realize this implementation, we work around a *phlex-chem* basic test that imitates to NMMB-MONARCH data structures and *phlex-chem* call, named *mock-monarch* test. Since NMMB-MONARCH is developed on Fortran, this test is also written in Fortran and uses an interface to communicate the data and functions between the C *phlex-chem* files.

This NMMB-MONARCH test was compared to NMMB-MONARCH phlex-chem call and is very similar, with the exception that NMMB-MONARCH computes and sent to *phlex-chem* some extra data, plus the mechanism used to use more reactions. The variables configuration for this test are the following:

- **Environmental array:** Array composed by temperature and pressure. These variables are necessary at the start to compute the rate constants for each cell.
- **Species**: Only 3 species (A, B and C).
- **State:** Array of initial species concentration with the length of 3. One concentration is initialized at one and the others at zero. During the solving, this concentration generates a concentration of one of the other species, and the new species concentration generates also concentration for the other species, with different rate constants each.
- **Reactions:** Only two Arrhenius reaction, with the same number of *React_params*, but with different values.

All the data is initialized to fixed values, making no difference between cells. This is correct for reaction parameters and rate constants since we are computing the same reactions in each cell, but for environmental variables and state arrays, we can have different values on each cell in the NMMB-MONARCH case.

Thus, we improved the test by adding different offset values to all the concentrations and environmental variables. For example, at the first temperature value we sum a 0.01 value, at the second 0.02, and so on till N cells temperatures. Note that the resultant values are revised to be in a typical range, avoiding nonsense values like a negative pressure.

We also have to mention that NMMB-MONARCH organizes the data for each cell a bit different. NMMB-MONARCH saves this data on a 4D matrix, where the first dimension represents the axis X of a map (from West to East), the second dimension the axis Y (from North to Sud), the third dimension the axis Z (depth, also called vertical levels), and the fourth dimension the data values (for example, the state array).

Since we leak this 4D matrix structure on *phlex-chem*, we had to adapt the matrix to our structures, especially for the multiple cells implementation.

Our *phlex-chem* structures are both arrays, the state array, and the environmental array. Thus, we have to translate the 4D NMMB-MONARCH matrix to a 1D array. To do this, we have to remember the memory layout of multi-dimensional arrays [58]. In fact, for the memory, a matrix is merely an array. The memory (and our implementation) access the matrix updating a variable offset, dependent on the row and column assigned. Since our test is designed on Fortran, the offset is calculated in the following way:

$$offset = n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots))) = \sum_{i=1}^{d} \left(\prod_{j=1}^{i-1} N_j \right) n_i$$

Where N_i is the size of the dimension *i*, *d* is the number of dimensions and n_i is the index of an element. Something equivalent for a 2D matrix is the following:

$$offset = n_1 + N_1 \cdot n_2 \quad \text{and} \quad offset = i_{col} * NROWS + i_{row}$$

Using this offset we can put all the cells data inside our state and environmental. Now the test can send all the data in one *phlex-chem* call.

9.2 General changes on the *phlex-chem* code

Now that our test is sending a state and environmental arrays with all the data cells, we need a method to access correctly the first data value of each cell. To do this, we have to know all the time two values: The array size for each data cells and the total number of cells.

Fortunately, the array sizes are the same for all cells, since we are contemplating all the system species on each state array, and all the environmental variables on each environmental array. Moreover, this array size is already present on all the *phlex-chem* code and can be accessed easily all the time.

The only part that needs a bit of code modification is sent the number of cells. To do this, we imitate the implementation used on *phlex-chem* for this type of variable: Save them on a global C struct. This global struct is passed around most of the program parts, making easier to access his variables.

Once we know all the time the number of cells and offsets (cell array sizes), we have to iterate over this number of cells all the functions related to this data (Derivative, Jacobian,

update_rates…), either on GPU and CPU versions. Note that we only need to update the pointer to the first data value when we refer to the next cell value.

RXN structure should be bigger since the rate constants included on RXN depends on temperature and pressure, and these values are different for each cell. We have two possible implementations to fix this issue:

1. Add all the different cell rate constants on the RXN structure. This will increase considerably the data size of the RXN structure.
2. Move the rate constants away from the RXN structure. In this way, we left the fixed variables among different cells on RXN structure, and treat the different values among cells (in our case, the rate constants) independently.

We decided to implement the second option in order to identify better the data with different sizes, especially when dealing with large data sizes. We create a rate constants array with the size of the number of reactions per number of cells. When we need a rate constant for the desired reaction, we will access this array instead of the RXN structure, leaving this structure with only fixed values for all the *phlex-chem* iterations.

Note also that we are increasing the size of the CVODE data structures *deriv* and Jacobian matrix. *Deriv* is only an array, but Jacobian is a matrix and can escalate quickly. Writing some numbers, if before we were considering a Jacobian matrix of N*N size, where N is the cell state size, now considering 10,800 cells from NMMB-MONARCH this matrix will scale by a factor of 10^8. However, most of the new values will be zero. But as we mentioned in section 6.3, KLU SPARSE works very well with matrices with a large number of zeros. So, KLU SPARSE implementation should shine with this implementation.

9.3 CPU results - Multiple cells computation

In this section, we present and discuss the results for the N cells implementation without using the GPU.

With this optimization, we improve a lot of elements that reduce time execution. Below are the most relevant:

- Avoided CVODE and some *phlex-chem* variables reinitialization in each iteration.
- Saved cache memory errors by accessing all the data at the same time.
- Allowed vectorization on RXN structure, since for simple test it has the same sizes on all *rows*.
- Calls to Derivative and Jacobian functions are lowered a lot.
- Improved efficiency of the KLU SPARSE matrix structure for the Jacobian.

In figure 18 we can see the speedup of the N cells implementation versus the original version according to the number of cells. As we can see, the average speedup is very high (around 12), and the variability is low (range from 11.92 to 14.11).

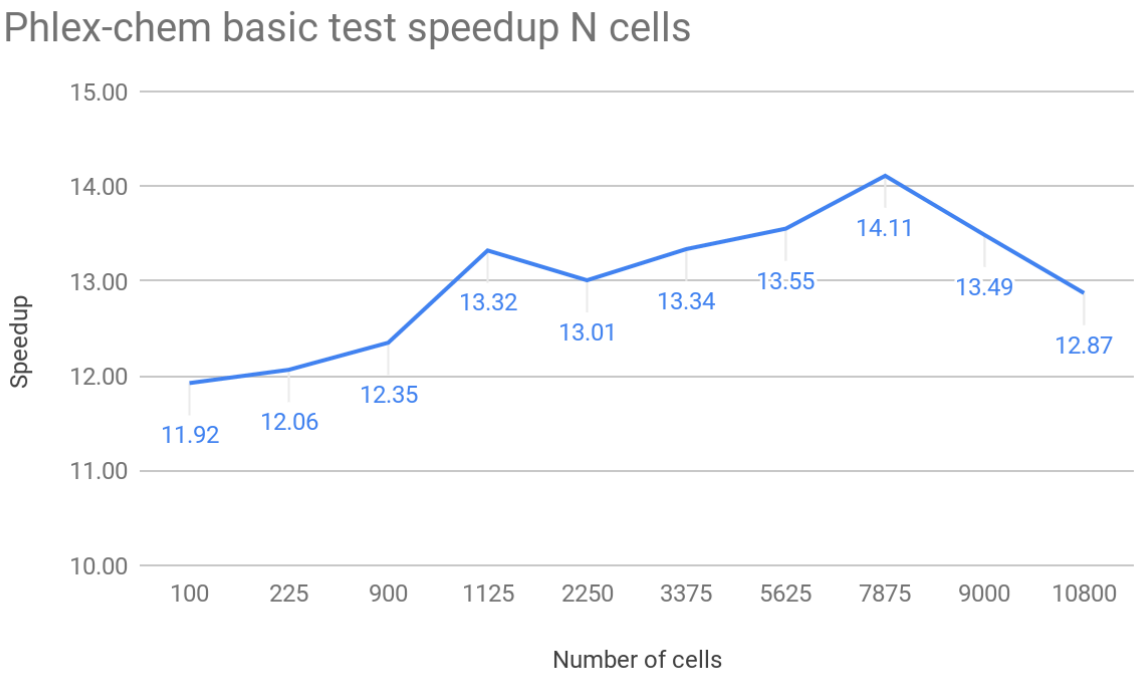


Figure 18. Phlex-chem NMMB-MONARCH basic test speedup of N cells optimization for different cells number.

To confirm the concept expected that we reduce Derivative and Jacobian calls, we measured the Derivative calls of the original version and N cells version. We calculated the scale between these two calls (Original version calls / N cell calls) for a various number of cells. We have to mention that this scale is very similar to the one corresponding to the Jacobian function. In figure 19 we can see these measures. As we can see, the graph is very linear, showing that the number of calls increases by the number of cells on factor 1 approximately. Moreover, the number of calls reduced is quite big.

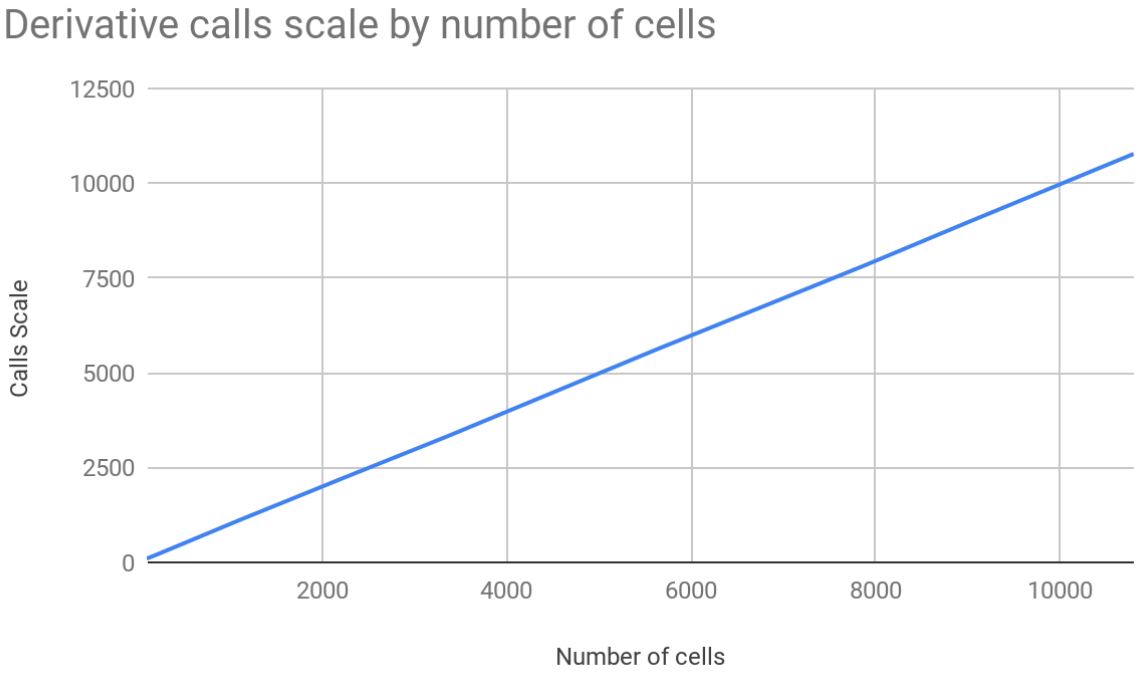


Figure 19. Scale (Original version / N cells version) of derivative calls for different number of cells

9.4 GPU implementation - Multiple cells computation

In this section, we comment on some implementation differences derived from working with large data. Also, we present and discuss the results for the N cells implementation computing the Derivative on the GPU.

We keep the GPU strategy of parallelizing reactions on each thread. The difference now is that we compute again the reactions for each cell with different state and rate constant values. In figure 20, we can see a summary of the parallelization done now.

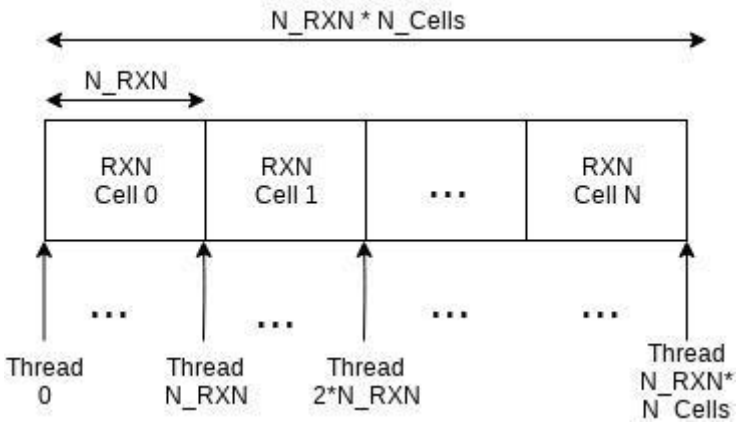


Figure 20. Thread distribution for the N Cells optimization. N_RXN represents the number of reactions and N_Cells the number of cells.

Note that parallelizing the N_Cells and N_RXN, the Derivative flow diagram of N Cells will be the same as the one represented in figure 10.

9.4.1 Shared implementation per N Cells

This section explains an implementation based on taking advantage of the CUDA shared memory [59].

We consider the shared memory as a buffer, where we compute as much data as the maximum shared memory allows, copy the results into global memory, reset the buffer and repeat. In figure 21 we can see an illustrative image of this method.

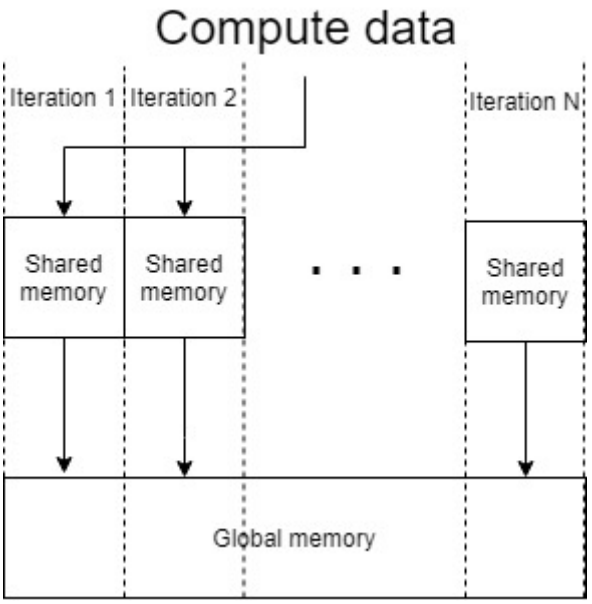


Figure 21. Schema of the buffer shared memory implementation

However, this method gives similar time executions. We suppose the GPU compiler already uses this shared memory as a cache buffer on the compiled code. In consequence, in the results section, we present the results without this implementation.

9.5 GPU measurements & analysis

With this optimization, we expect to reduce the computation time by this parallelization of the cell loop. However, we must take into account the extra time of moving the data to the GPU. In order to see his impact, we measured both times, the computation and the data movement time on GPU for different numbers of cells. In figure 22 we can see the results.

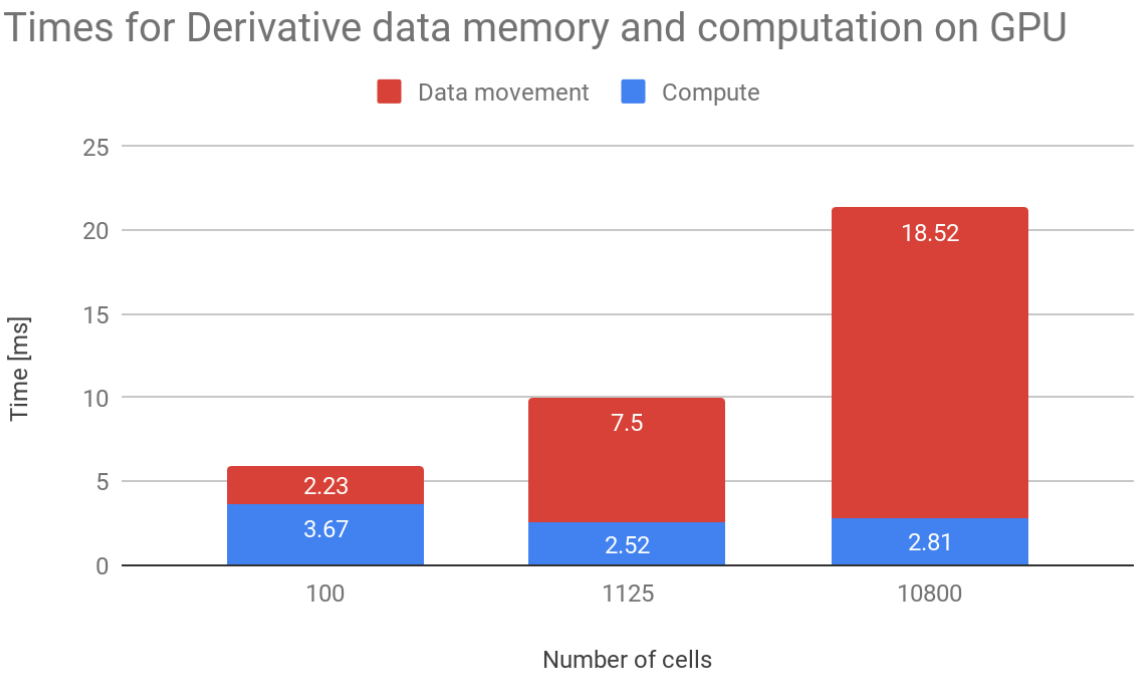


Figure 22. Data movement and computation time executions of Derivative for 100, 1125 and 10800 cells.

Notice that the computation does not scale when we increase the number of cells. In contrast, data movement time increase in factor 8.3 from 100 cells to 10,800. This is not a big factor taking into account that we increase the number of cells per 100, so we can expect a positive speedup from the CPU version. Note also that data movement is taking most of the execution times for a large number of cells, (around 85% of the total time execution for 10,800 cells) being the bottleneck to study in future implementations.

Finally, we measured the speedup of the CPU time execution versus the GPU time execution for the N cells implementation. In figure 23 we can see the results for different numbers of cells.

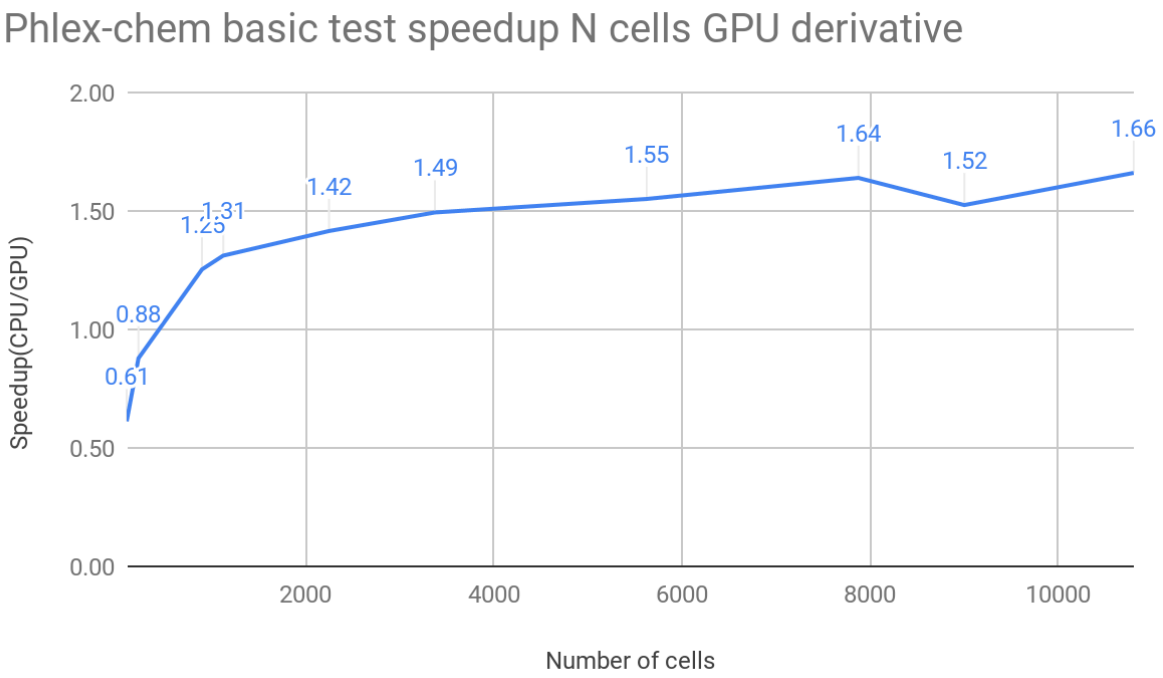


Figure 23. Phlex-chem NMMB-MONARCH basic test speedup of N cell optimization and GPU Derivative computation for different cell numbers.

As we can see, above 1000 cells we have a positive speedup, which increases slowly with the number of cells, being 1.5 as average. Remember this speedup is achieved only calculating the Derivative function on the GPU, and the future Jacobian implementation can also improve this speedup since both functions are very similar.

Note the maximum data computed on these tests is: 3 state values per 10.800 cells per 8 bytes/state value ≅ 259KB of data. This is little data for the GPU, which has the capacity to work with GBs of data. Moreover, the only increment of data expected on NMMB-MONARCH is this 3 state values to 100, which give 8MB of data approximately. So, we are not using all the GPU capacity.

Chapter 10

Conclusions and future work

10.1 Conclusions

In this thesis, we analyzed the complex weather model NMMB-MONARCH. Taking a general view through traces, we discover that his chemistry solver *phlex-chem* is taking 48.6% of the execution time, making it an interesting objective to optimize and our main focus. We also take advantage of the opportunity to study the use of the GPU in order to improve the computation.

We started doing a performance overview of the *phlex-chem* module. From this analysis, we conclude that near 70% of the *phlex-chem* time execution comes from the solver library CVODE, and the 30% left comes from two functions: Derivative and Jacobian.

These functions are very similar and the optimizations for one can be applied to the other. Then, we described and analyzed an isolated basic version of Derivative function that simulates the *phlex-chem* Derivative. From this version, we conclude that using the GPU can speed up the program by a factor 1.8 on the CTE-POWER cluster.

We translated the GPU implementation to the Derivative on *phlex-chem*. Moreover, we apply optimization to improve the memory access, based on reordering the data of the main structure (RXN structure). We improve the speedups measured by 30% on average with this optimization. However, the results give that GPU implementation is only efficient for big mechanisms that include a lot of reactions (from 2000 reactions), which ones are rarely used on the chemistry.

We decide to not only parallelizing the reactions and add to the focus the *phlex-chem* calls. NMMB-MONARCH calls per default 10,800 times *phlex-chem* with different data, representing that each call computes a cell on a map. These calls are independent of each other and can be parallelized, which should work nicely for the GPU. Thus, we start working with the idea to send all the data from the 10,800 calls to *phlex-chem* in one call and compute them simultaneously.

We worked around a simple test simulating the NMMB-MONARCH data structures and *phlex-chem* call. We improve this test by applying different data on each call and adapting

one NMMB-MONARCH complex structure (4D matrix) to the simple one working on *phlex-chem* (state array with chemical concentrations).

Once we implemented the optimization with multiple cells computation, we found that the speedup was grateful. Testing with different numbers of cells from 100 to 10,800, the *phlex-chem* simple test works between 12 and 14 times faster. We find out these good results come from avoiding a lot of iterations even on the CVODE solver, avoiding the cell loop on all the *phlex-chem* module. This phenomena improves a lot the memory access, and avoid some CVODE and *phlex-chem* data reinitializations on each call.

Finally, we tested if the GPU utilization improves this result. Thanks to parallelization, the computation time is not increasing by the number of cells. In contrast, the time for data movement between GPU and CPU increases with the number of cells computed. The time for this data movement represents around 85% of the total time execution for 10,800 cells, leaving it as the GPU bottleneck for big number of cells. Another interesting measure is that this time only increases in factor 8.3 from 100 cells to 10,800, resulting in a 1.5 speedup on the entire *phlex-chem* module.

We conclude that improving a model not always come from optimizing the individual functions. Instead, taking a complete module view with an accurate test can give very powerful ideas to speedup. In addition, the GPU usage can also optimize complex modules like *phlex-chem* without using a lot of capacity but have a lot of work on adapting the code functions individually.

10.2 Future work

For future work, we left the implementation of the optimized *phlex-chem* on NMMB-MONARCH, translate the Jacobian function and treat complex cases with more reactions than the simple test.

Furthermore, the detected bottleneck on GPU data movement leads us to explore new future optimizations. For example, we can explore a heterogeneous implementation where the CPU can do other calculations meanwhile the GPU data transfers. The potential of this optimization depends on the no dependance of CPU calculations with GPU data. We can look for this CPU calculations by exploring with more detail NMMB-MONARCH in the future.

Other optimizations for the GPU data movement include reducing the memory jumps on accessing the RXN data or translating this matrix to the SPARSE structure in order to improve memory management.

Bibliography

[1] S. Hartmann and R. Frigg, “Scientific models,” Jan. 2005.

[2] J. D. Sterman, “All models are wrong: reflections on becoming a systems scientist,” *Syst. Dyn. Rev.*, vol. 18, no. 4, pp. 501–531, 2002.

[3] M. Poznic, “Models in Science and Engineering,” Apr. 2017.

[4] M. Carmen Lemos, C. Kirchhoff, and V. Ramprasad, “Narrowing the Climate Information Usability Gap,” *Nat. Clim. Change*, vol. 2, Oct. 2012.

[5] E. Holzbecher, *Environmental Modeling*. 2012.

[6] N. D Bennett, B. Croke, A. J. Jakeman, L. T H Newham, and J. P Norton, “Performance evaluation of environmental models,” 2010.

[7] “MareNostrum,” *BSC-CNS*. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum. [Accessed: 26-Aug-2019].

[8] “The NMMB/BSC-Dust model — Barcelona Dust Forecast Center.” [Online]. Available: https://dust.aemet.es/methods/the-nmmb-bsc-dust-model. [Accessed: 26-Aug-2019].

[9] “Technical Information,” *BSC-CNS*. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information. [Accessed: 13-Aug-2019].

[10] E. Mizell and R. Biery, “Introduction to GPUs for Data Analytics,” p. 39.

[11] Y. Jararweh, M. Jarrah, A. Bousselham, and S. Hariri, “Gpu-based personal supercomputing,” *2013 IEEE Jordan Conf. Appl. Electr. Eng. Comput. Technol. AEECT*.

[12] “New GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500 | TOP500 Supercomputer Sites.” [Online]. Available: https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-top500/. [Accessed: 13-Aug-2019].

[13] O. Jorba *et al.*, “Potential significance of photoexcited NO2 on global air quality with the NMMB/BSC chemical transport model,” *J. Geophys. Res. Atmospheres*, vol. 117, p. D13301, Jul. 2012.

[14] Z. Janjic and L. Gall, “Scientific documentation of the NCEP nonhydrostatic multiscale model on the B grid (NMMB). Part 1 Dynamics,” 2012.

[15] A. Da Silva *et al.*, “The Earth system modeling framework,” Jan. 2002.

[16] Z. Janjic, “A nonhydrostatic model based on a new approach,” *Meteorol. Atmospheric Phys.*, vol. 82, pp. 271–285, Jan. 2003.

[17] Z. Janjic, “Nonlinear Advection Schemes and Energy Cascade on Semi-Staggered Grids,” *Mon. Weather Rev. - MON WEATHER REV*, vol. 112, Jun. 1984.

[18] Z. I. Janić, *Nonsingular Implementation of the Mellor-Yamada Level 2.5 Scheme in the NCEP Meso Model*. U.S. Department of Commerce, National Oceanic and Atmospheric Administration, National Weather Service, National Centers for Environmental Prediction, 2001.

[19] A. S. Monin and A. M. Obukhov, “Basic laws of turbulent mixing in the surface layer of the atmosphere,” p. 30.

[20] Z. I. Janjić, “The Step-Mountain Eta Coordinate Model: Further Developments of the Convection, Viscous Sublayer, and Turbulence Closure Schemes,” *Mon. Weather Rev.*, vol. 122, no. 5, pp. 927–945, May 1994.

[21] M. Ek *et al.*, “Implementation of NOAH land surface model advances in the NCEP operational mesoscale Eta model,” *J. Geophys. Res.*, vol. 108, Apr. 2003.

[22] A. Vukovic, B. Rajkovic, and Z. Janjic, “Land Ice Sea Surface Model: Short Description and Verification,” 2010.

[23] E. J. Mlawer, S. J. Taubman, P. D. Brown, M. J. Iacono, and S. A. Clough, “Radiative transfer for inhomogeneous atmospheres: RRTM, a validated correlated-k model for the longwave,” *J. Geophys. Res. Atmospheres*, vol. 102, no. D14, pp. 16663–16682, 1997.
[24] B. S. Ferrier, Y. Jin, Y. Lin, T. Black, E. Rogers, and G. Dimego, “Implementation of a new grid-scale cloud and precipitation scheme in the NCEP Eta model,” *15th Conf Numer. Weather Predict.*, pp. 280–283, Jan. 2002.
[25] A. Betts and M. J. Miller, “A new convective adjustment scheme. Part II: Single column tests using GATE wave, BOMEX, ATEX and arctic air-mass data sets,” *Q. J. R. Meteorol. Soc.*, vol. 112, pp. 693–709, Jul. 1986.
[26] Z. Janjic, “The Step-Mountain Coordinate: Physical Package,” *Mon. Weather Rev. - MON WEATHER REV*, vol. 118, Jul. 1990.
[27] M. Guevara, C. Tena, M. Porquet, O. Jorba, and C. Pérez García-Pando, “HERMESv3, a stand-alone multi-scale atmospheric emission modelling framework – Part 1: global and regional module,” *Geosci. Model Dev.*, vol. 12, no. 5, pp. 1885–1907, May 2019.
[28] 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 Chem. Phys.*, vol. 6, no. 11, pp. 3181–3210, Aug. 2006.
[29] “Photolysis scheme Fast-J (Wild et al., 2000) Gas phase chemistry CBM-Z...,” *ResearchGate*. [Online]. Available: https://www.researchgate.net/figure/Photolysis-scheme-Fast-J-Wild-et-al-2000-Gas-phase-chemistry-CBM-Z-Zavier-et-al_fig1_328396780. [Accessed: 02-Aug-2019].
[30] A. Atkinson, V. Siegel, E. Pakhomov, and P. Rothery, “Long-term decline in krill stock and increase in salps within the Southern Ocean,” *Nature*, vol. 432, pp. 100–3, Nov. 2004.
[31] D. J. Luecken, S. R. Phillips, C. Jang, and N. Possiel, “Effects Of Using The CB 05 vs . SAPRC 99 vs . CB 4 Chemical Mechanism On Model Predictions,” 2006.
[32] J. H. Seinfeld and S. N. Pandis, *Atmospheric chemistry and physics: from air pollution to climate change*. Wiley, 1998.
[33] L. Zhang, S. Gong, J. Padro, and L. Barrie, “A size-segregated particle dry deposition scheme for an atmospheric aerosol model,” *Atmos. Environ.*, vol. 35, pp. 549–560, Dec. 2001.
[34] W. G. N. Slinn, “Predictions for particle deposition to vegetative canopies,” *Atmospheric Environ. 1967*, vol. 16, no. 7, pp. 1785–1794, Jan. 1982.
[35] K. W. Appel *et al.*, “Geoscientific Model Development A multi-resolution assessment of the Community Multiscale Air Quality (CMAQ) model v 4 . 7 wet deposition estimates for 2002 – 2006,” 2011.
[36] C. Pérez *et al.*, “Atmospheric dust modeling from meso to global scales with the online NMMB/BSC-Dust model – Part 1: Model description, annual simulations and evaluation,” *Atmospheric Chem. Phys.*, vol. 11, no. 24, pp. 13001–13027, Dec. 2011.
[37] G. Yarwood, S. Rao, M. A. Yocke, and G. Z. Whitten, “Updates to the Carbon Bond Chemical Mechanism: CB05,” 2005.
[38] V. Damian, A. Sandu, M. Damian, F. Potra, and G. Carmichael, “The kinetic preprocessor KPP—A software environment for solving chemical kinetics,” *Comput. Chem. Eng.*, vol. 26, pp. 1567–1579, Nov. 2002.
[39] J. H. Curtis, M. D. Michelotti, N. Riemer, M. T. Heath, and M. West, “Accelerated simulation of stochastic particle removal processes in particle-resolved aerosol models,” *J. Comput. Phys.*, vol. 322, pp. 21–32, Oct. 2016.
[40] “Matthew West: PartMC.” [Online]. Available: http://lagrange.mechse.illinois.edu/partmc/. [Accessed: 02-Aug-2019].
[41] *Particle-resolved stochastic atmospheric aerosol model: compdyn/partmc*. compdyn, 2019.

[42] “Introduction — autosubmit 3.9.1 documentation.” [Online]. Available: https://autosubmit.readthedocs.io/en/latest/introduction.html. [Accessed: 01-Sep-2019].
[43] B. J. Finlayson-Pitts and J. N. Pitts, *Chemistry of the Upper and Lower Atmosphere: Theory, Experiments, and Applications*. Academic Press, 2000.
[44] S. D. Cohen, A. C. Hindmarsh, and P. F. Dubois, “CVODE, A Stiff/Nonstiff ODE Solver in C,” *Comput. Phys.*, vol. 10, no. 2, p. 138, 1996.
[45] R. D. Skeel, “Construction of variable-stepsize multistep formulas,” 1986.
[46] J. R. Cash, “Second Derivative Extended Backward Differentiation Formulas for the Numerical Integration of Stiff Systems,” *SIAM J. Numer. Anal.*, vol. 18, no. 1, pp. 21–36, 1981.
[47] A. C. Hindmarsh, R. Serban, and D. R. Reynolds, “User Documentation for cvode v4.1.0 (sundials v4.1.0),” p. 316.
[48] E. PALAMADAI NATARAJAN, “KLU{A HIGH PERFORMANCE SPARSE LINEAR SOLVER FOR CIRCUIT SIMULATION PROBLEMS,” Jan. 2005.
[49] 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.
[50] 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.
[51] “suitesparse : a suite of sparse matrix software.” [Online]. Available: http://faculty.cse.tamu.edu/davis/suitesparse.html. [Accessed: 05-Aug-2019].
[52] “(PDF) Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems.” [Online]. Available: https://www.researchgate.net/publication/220492376_Algorithm_907_KLU_A_Direct_Sparse_Solver_for_Circuit_Simulation_Problems. [Accessed: 05-Aug-2019].
[53] S. Li, “An overview of SuperLU: Algorithms, implementation, and user interface.,” *ACM Trans Math Softw*, vol. 31, pp. 302–325, Jan. 2005.
[54] J. W. Demmel, J. R. Gilbert, and S. Li, “An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination,” *SIAM J. Matrix Anal. Appl.*, vol. 20, Aug. 1998.
[55] A. C. Hindmarsh *et al.*, “SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers,” *ACM Trans. Math. Softw. TOMS*, vol. 31, pp. 363–396, Nov. 2004.
[56] “Tesla V100 Data Center GPU | NVIDIA.” [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100/. [Accessed: 12-Aug-2019].
[57] “NVIDIA Developer Tools Overview,” *NVIDIA Developer*, 24-Feb-2014. [Online]. Available: https://developer.nvidia.com/tools-overview. [Accessed: 01-Sep-2019].
[58] “Memory layout of multi-dimensional arrays - Eli Bendersky’s website.” [Online]. Available: https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays. [Accessed: 12-Aug-2019].
[59] “Using Shared Memory in CUDA C/C++,” *NVIDIA Developer Blog*, 29-Jan-2013. [Online]. Available: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/. [Accessed: 08-Aug-2019].