BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SOLUTION OF DIFFERENTIAL EQUATIONS ON GPU USING A NUMERICAL METHOD BASED ON THE TAYLOR SERIES

ŘEŠENÍ DIFERENCIÁLNÍCH ROVNIC NA GPU POMOCÍ NUMERICKÉ METODY ZALOŽENÉ NA TAYLOROVĚ POLYNOMU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. PETR BÉM**
AUTOR PRÁCE

SUPERVISOR **Ing. PETR VEIGEND, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2026

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

172978

Institut:	Department of Intelligent Systems (DITS)
Student:	**Bém Petr, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	High Performance Computing
Title:	**Solution of differential equations on GPU using a numerical method based on the Taylor series**
Category:	Modelling and Simulation
Academic year:	2025/26

Assignment:

1. Get acquainted with the methods used for the numerical solution of ordinary differential equations.
2. Get acquainted with the GPU programming. Study how to implement the algorithms effectively and if it is possible to use variable-precision arithmetic on GPUs.
3. Design a high-order numerical algorithm based on the Taylor series that could be effectively used on GPUs.
4. Implement the algorithm on a selected GPU.
5. Test the implemented solution on a set of benchmark problems. Focus on analysing stability, accuracy and how the GPU is utilised.
6. Evaluate the implemented solution and propose possible improvements.

Literature:

- Dle doporučení vedoucího.
- Václav Šátek: Analýza stiff soustav diferenciálních rovnic, disertační práce, Brno, FIT VUT v Brně, 2011, https://www.vut.cz/studenti/zav-prace/detail/99803
- VEIGEND, Petr. High order numerical method in modelling and control systems. Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šátek, Ph.D, https://www.vut.cz/studenti/zav-prace/detail/161489

Requirements for the semestral defence:

- Complete points 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Veigend Petr, Ing., Ph.D.**
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2025
Submission deadline:	20.5.2026
Approval date:	3.11.2025

Abstract

Ordinary differential equations are often used to model and simulate problems in many fields, such as mathematics, physics, economics, chemistry. Even though, that these equations are well studied and have analytical solution, for big problems – systems of thousands or even millions of equations – it is impossible to obtain the analytical or obtain the solution by hands. Therefore, we use numerical methods to approximate the solution of such system of ordinary differential equations.

The general purpose graphics processing units are massively parallel devices containing of dozens to thousands arithmetic units. Which means that they are efficient for accelerating computations that consist of thousands to millions lightweight threads.

This thesis implements and compares several numerical methods for computation on the graphical processing units. The implemented methods are well-known Euler method, popular Runge-Kutta with 4 stages and higher-order Modern taylor series method.

The efficient use of GPU scales with the number of equations of the systems of linear ordinary differential equations. The results for 4-stage Runge-Kutta and modern Taylor series method with 4-th order has similar results. However, the precalculated variant of modern Taylor series method proven faster and was able to use better the GPUs resources with smaller problems.

Abstrakt

Obyčejné diferenciální rovnice jsou často používané pro modelování a simulování v mnoha různých oborech, mezi které patří matematika, fyzika, ekonomika, chemie. Přestože jsou tyto rovnice důkladně studované a je pro ně známo analytické řešení, pro velké problémy – systémy tisíců až milionů rovnic – je nemožné získat jejich analytické řešení. Z toho důvodu se používají numerické metody pro aproximaci řešení takovýchto systémů obyčejných diferenciálních rovnic.

Grafické karty (GPU) jsou masivně paralelní zařízení, které obsahují desítky až tisíce aritmetických jednotek. To znamená, že je možné efektivně je využívat pro akceleraci náročných výpočtů používajících tisíce až milióny lightweight vláken.

Tato práce implementuje a porovnává několik numerických metod pro výpočet na grafických kartách. Implementované metody jsou velmi známá Eulerova metoda, populární čtyřstupňová metoda Runge-Kutta a moderní metoda vyššího řádu Taylorova rozvoje.

V práci je zjištěno, že efektivní využití grafické karty škáluje s počtem diferenciálních rovnic. Výsledky pro čtyřstupňovou metodu Runge-Kutta a moderní metodu Taylorovy řady (s řádem 4) jsou podobné. Naopak varianta moderní Taylorovy řady s metodou předpočítání dosahuje lepších výsledků, co se týče času a využití grafické karty.

Keywords

Numerical methods, ordinary differential equations, GPU, OpenMP, modern Taylor series method.

Klíčová slova

numerické metody, diferenciální rovnice, grafické karty, openmp, metoda moderní Taylorovy řady.

Reference

BÉM, Petr. *Solution of differential equations on GPU using a numerical method based on the Taylor series.* Brno, 2026. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Petr Veigend, Ph.D.

Rozšířený abstrakt

Obyčejné diferenciální rovnice jsou rovnice obsahující derivaci podle jedné proměnné. Tyto obyčejné diferenciální rovnice jsou následujícího tvaru

$$y'(t) = F(t, y(t)),$$

kde řešení takových rovnic je funkce $y(t)$, pro kterou je derivace $y'(t) = \frac{\mathrm{d}y}{\mathrm{d}t}$.

Obyčejné diferenciální rovnice jsou často používané pro modelování a simulování v mnoha různých oborech (matematika, fyzika, ekonomika, chemie) a jsou velmi důkladně studované, včetně jejich analytického řešení. Ovšem v případě velkých problémů, jako jsou tisíce až miliony rovnic, nelze jejich analytické řešení získat. Analytické řešení pro takovéto rovnice nemusí existovat nebo je obdržení řešení ručně značně časově náročné. Řešením tohoto problému jsou numerické metody. Numerické metody jsou zde využity pro aproximaci řešení rozsáhlých systémů obyčejných diferenciálních rovnic. Výpočet numerických metod nemusí být vhodné provádět na procesorech (CPU), neboť se často jedná o vysoce paralelní výpočty s čísly s pohyblivou řádovou čárkou. V poslední době se stává populárním implementace podobných výpočtů na akceleračních zařízeních, mezi které patří grafické karty (GPU).

Grafické karty mají oproti CPU rozdílnou architekturu. Neobsahují sice tak velké hierarchické paměti cache, ale za to obsahují větší množství aritmeticko-logických jednotek (ALU). Ty jsou schopny efektivně vykonávat instrukce s velkou paralelizací pro jemné rozdělení na tisíce až miliony lightweight vláken. Rozdělení práce pro jednotlivá vlákna je důležité pro efektivní využití grafické karty.

Tato diplomová práce implementuje a porovnává několik numerických metod pro výpočet na grafických kartách. Implementované metody jsou velmi známá Eulerova metoda, populární čtyřstupňová metoda Runge-Kutta a moderní metoda vyššího řádu Taylorova rozvoje. Eulerova metoda používá jednoduché, ale za to rychlé řešení pouze s použitím první derivace. Runge-Kutta oproti tomu používá aproximaci ve více stupních, což zvyšuje přesnost a stabilitu této metody. Moderní metoda vyššího řádu Taylorova rozvoje pracuje s myšlenkou výpočtu vyšších řádu derivace pro zvýšení přesnosti a stability, tento řád lze nastavit podle potřeby. V této práci je dále popsána a implementována právě tato metoda pro systémy lineárních obyčejných diferenciálních rovnic.

Pro implementaci je využito OpenMP, které umožňuje spuštění jak na GPU, tak na (více i jedno vláknovém) CPU. Je použit vlastní formát nad souborovým formátem HDF5 pro definování problému. Implementace má jednotné rozhraní pro všechny metody a umožňuje je tak jednoduše porovnávat. Jsou implementovány dvě možnosti pro zadaní matice Jakobiánu – husté a řídké.

Pro testování je použita desktopová grafická karta AMD RX 9060 XT. Výsledky ukazují, že využití grafické karty roste s počtem počítaných rovnic. Následně je ukázáno, že metody Runge-Kutta se čtyřmi stupni a moderní metoda Taylorova rozvoje s řádem čtyři mají srovnatelné výsledky. Avšak varianta s přepočítáním pro metodu Taylorovy řadem, je schopna efektivněji využít GPU a počítat rychleji.

Solution of differential equations on GPU using a numerical method based on the Taylor series

Declaration

I hereby declare that this Master's thesis was prepared as an original work by myself under the supervision of Ing. Petr Veigend, Ph. D. The supplementary information for Stokes problem was provided by Ing. Václav Šátek, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. No LLMs and generative AI was used for the text and implementation of this thesis. All thoughts are my own.

.....
Petr Bém
May 20, 2026

Acknowledgements

Firstly, I would like to thank my supervisor Ing. Petr Veigend, Ph.D., without his supervising and guidance I would not be able to finish this Thesis. I would not be able to accomplish the same with any other supervisor.

I would also like to thank consultant Ing. Václav Šátek, Ph.D. for introducing me to the topic of numerical differential equation methods and providing help with the Stokes problem experimentation.

As last but not least, I want to thank my family and my girlfriend for all the help and support I received during the time of writing this Thesis.

Contents

1 Introduction **6**

2 Methods of solving differential equations **7**
2.1 Differential equations 7
2.2 Numerical differential equation methods 8
2.3 Numerical computation errors 11
2.4 Stability of single-step methods 13
2.5 Higher order method — Modern Taylor series method 18

3 General-purpose computing on graphics processing units **23**
3.1 Architecture 24
3.2 Options 27
3.3 OpenMP 28

4 Implementation of ODE solver **37**
4.1 Problem definition format – HDF5 file 38
4.2 Dense algorithm 39
4.3 Sparse matrix algorithm 45
4.4 Issues with implementation 47

5 Experiments **52**
5.1 First experiments 52
5.2 Telegraph equation 54
5.3 Incompressible Stokes system 64
5.4 Future work 74

6 Conclusion **76**

Bibliography **77**

Appendices **79**
List of Appendices 80

A Digital attachments **81**

B Specification of benchmarks CPU **82**

C Specification of benchmarking GPU **84**

List of Figures

2.1 Visualization of derivative at time step t_n. 9
2.3 Correlation between round-off and truncation error with dependence on the step size (h). 13
2.4 Example of non-convergent (divergent) solution. 14
2.5 Stability domain of Euler method. 16
2.6 Stability domain of explicit Runge-Kutta4. 18
2.7 Stability domain of explicit modern Taylor series method. 21

3.1 Comparison between CPU and GPU architecture. 25
3.2 GPU and CPU system diagram. 26
3.3 Visual representation of grid blocks. 27
3.4 Illustration of different OpenMP directives for parallelization. 32

4.1 HDF5 File structure 38
4.2 HDF5 Structure of sparse matrix A 39
4.3 Visualization of parallelization for GPU of dense matrix algorithm. 44
4.4 Illustration of compressed sparse row (CSR) format. 45

5.1 Comparison of solutions for exponential functions. 53
5.2 Comparison of Euler method and MTSM solving the ODE for sin and cos. . 54
5.3 Model of telegraph line of S segments in series as introduced in the article [17]. 54
5.4 Simulation of telegraph line using adjusted line model. 58
5.5 Simulation of telegraph line using model without adjustment. 59
5.6 Nonzero elements of sparse matrix. 59
5.7 Graph of measured time for telegraph equation depending on problem size. 60
5.8 Visualization of Stokes domain and Dirichlet boundary. 68
5.9 The Stokes domain Ω discretized with 23 nodes in each dimension. 68
5.10 Visualization of velocities in incompressible Stokes problem. 69
5.11 Graph of measured time for Stokes problem depending on problem size. . . 70

List of Tables

3.1 Different naming between GPU vendors 23
3.2 Different naming between GPU programming platforms 24

5.1 Measured mean time for comparison of all methods on GPU vs single-thread CPU. 61
5.2 GPU analysis of implemented methods for telegraph equation. 62
5.3 Parallelization error of each method for telegraph equation. 62
5.5 Comparison of errors between individual solutions for telegraph equation. . 63
5.6 Measured mean time for comparison of all method on GPU vs single-thread CPU vs multiple-threads CPU 71
5.7 GPU analysis of implemented methods for Stokes problem. 72
5.8 Parallelization error of each method for Stokes problem. 73
5.9 Comparison of errors between individual solutions for Stokes problem. . . . 74

List of abbreviations

CPU	central processing unit.
CSR	compressed sparse row.
CU	compute unit.
CUDA	Compute Unified Device Architecture.
FLOPS	floating-point operations per second.
FMA	fused-multiply-accumulate.
FP32	32-bit floating-point.
GPGPU	general-purpose computing on a graphics processing unit.
GPU	graphics processing unit.
HDF5	hierarchical data format.
HIP	C++ Heterogeneous-Compute Interface for Portability.
HPC	high-performance computer.
IVP	Initial Value Problem.
LSU	load/store unit.
lte	Local truncation error of nth step: τ_n.
MTSM	modern taylor series method.
ODE	Ordinary differential equation.
PDE	Partial differential equation.
SFU	special function unit.
SHM	shared memory.
SIMD	Single Instruction, Multiple Data.
SIMT	Single Instruction, Multiple Threads.
SM	streaming multiprocessor.
VALU	vector ALU.

WGP workgroup processor.

Chapter 1

Introduction

Ordinary differential equation (ODE) are well studied equations with wide application in many fields. Even though many of those studied equations have analytical solution, for simulations and analysis of big systems of ODEs (thousands to millions equations) the solution analytic is not feasible to be obtained by hands. Therefore the numerical approximation methods for solving differential equations are used.

With graphics processing units (GPUs) being massively parallel devices with efficient parallel computation, arises the question of effective use of GPUs for numerical solving big systems of ODE. Continuing the research of the University on higher-order method based on the Taylor series expansion, brings another question to mind, whether this method would benefit from the use of GPU in comparison to the state-of-the-art methods.

Which is the main goal of this Thesis, to research whether the numerical approximation on GPU would be able to use the GPU effectively.

The remaining part of this chapter is for describing the structure of this Thesis. The next chapter Chapter 2 provides basic definitions regarding the topic of solving differential equations using numerical methods, state of the art methods used for numerically solving differential equations, the topic of errors introduced by the numerical approximation, the definition of stable methods and the theoretical introduction to the higher order method based on Taylor series.

The Chapter 3 provides basic introduction to the programming of GPUs. Starting with the detailed description of general GPU architecture and continuing with comparison of options for general-purpose computing on a graphics processing unit (GPGPU). After that there is explanation of OpenMP concepts for offloading code to the GPU and using parallelizable directives for effective use of the GPU.

The Chapter 4 describes the implementation of the state of the arts methods together with the modern taylor series method (MTSM). It starts with description of custom file format for defining problems of systems of linear ODEs to be solved by the implementation. The next section is the detailed explanation of the implementation for solving problem with *dense* Jacobian matrix and continuing with implementation for problems with *sparse* Jacobian matrix. The chapters ends with some of the problems encountered during the implementation and whether solution to the issue was found.

The Chapter 5 first introduces experiments for checking the correctness of the implementation on GPU. The chapter continues in with defining two experiments that are able to create big enough problems for benchmarking the implementation of GPU and comparing it to the run on the central processing unit (CPU). The first big experiment is the Telegraph line equation. The other one is incompressible Stokes problem.

Chapter 2

Methods of solving differential equations

This chapter contains only basic definitions relevant to this Thesis on the topic of numerical methods for solving differential equations. The chapter is summary of many resources, which are cited within the relevant part of text.

More detailed definitions can be found in the books [6], [5] and [8]

2.1 Differential equations

A first-order differential equation is in the following form:

$$y'(t) = F(t, y(t))$$

The solution of differential equation is the function $y(t)$ with derivative of $y(t)$ function $y'(t) = \frac{\mathrm{d}y}{\mathrm{d}t}$. For differential equation to have an unique solution it is required to have *initial value* $y(t_0) = y_0$.

Definition 2.1.1 (Initial Value Problem)**.** The pairing of initial value and differential equation creates an *Initial Value Problem (IVP).*

$$y'(t) = F(t, y(t)), \qquad y(t_0) = y_0, \tag{2.1}$$

where the "independent" variables are called "time" for t and "position" for x, meanwhile the y is called "dependent variable".

The order of differential equation is determined by the highest differentiated term. For example the following equation is of the third-order: $y''' = f(t, y', y'')$. For using fourth-order or higher the notations switches to roman numbers $y^{(iv)}, y^{(v)}, \ldots$ or arabic numbers $y^{(4)}, y^{(5)}, \ldots$ The benefit of arabic numbers is that they allow for generic notation such as $y^{(i)}, y^{(n)}$.

ODEs are differential equations with single independent variable. As opposed to Partial differential equations (PDEs), which contain multiple independent variables. For example Heat equation with time $\frac{\partial u}{\partial t}$ and space $\frac{\partial^2 u}{\partial x^2}$ as independent variables.

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

Definition 2.1.2 (Autonomous differential equation)**.** A differential equation is called "autonomous" if the function f does not contain independent variable (t).

2.1.1 Solution

The following definitions and theorems can be found in more detailed explanations at [6, p. 22–25], [5, p. 261–263] and [8, p. 36].

Definition 2.1.3 (Lipschitz condition)**.** Function $f : [a, b] \to \mathbb{R}$ is said to satisfy the *Lipschitz condition* if there is a constant $L > 0$ such that

$$|f(z) - f(y)| \leq L\,|z - y|\,, \quad \forall y, z \in [a, b]. \tag{2.2}$$

The smallest constant L satisfying (2.2) is called *Lipschitz constant*.

Theorem 2.1.1. There exists a *unique solution* $y(t)$ for $t \in [a, b]$ in an *initial value problem*

$$\begin{aligned} y'(t) &= F(t, y(t)), \\ y(a) &= y_0, \end{aligned} \tag{2.3}$$

if function $F : [a, b] \times \mathbb{R}^n \to \mathbb{R}^n$ is continuous in its first variable (t) and satisfies Lipschitz condition in its second variable $(y(t))$.

Proof of Theorem 2.1.1 can be found at [6, p. 23].

2.2 Numerical differential equation methods

Sometimes finding the exact analytical solution to a differential equations might be impossible or too time expensive to obtain. This is when numerical methods come handy. Since numerical methods approximates the solution of the differential equations, they are quicker and better suited to computers.

With numerical methods we solve the differential equations in specified time interval, usually $[t_0, t_f]$, in $N \in \mathbb{N}$ number of steps and initial condition $y(t_0) = y_0$.

Numerical method then computes the solution $y(t)$ for time points

$$t_n = t_0 + nh, \text{ for each } n = 0, 1, 2, \ldots, N, \tag{2.4}$$

where h is "step size", time between two consecutive time steps.

$$h = (t_f - t_0)/N = t_{n+1} - t_n. \tag{2.5}$$

2.2.1 Explicit and implicit methods

As this thesis contains primarily explicit methods, I will briefly mention implicit methods here and differences between explicit and implicit methods. For those interested in implicit methods I can suggest the following literature in Czech [11].

One of the differences are that the implicit methods are better at computing stiff systems. Whereas explicit methods require very small time step to compute stiff systems at reasonable precision. However, implicit methods require more computation and are harder to implement.

For better illustration of the difference between the two approaches we can compare explicit Euler method

$$y_{n+1} = y_n + hF(t_n, y_n) \tag{2.6}$$

and implicit euler method

$$y_{n+1} = y_n + hF(t_{n+1}, y_{n+1}). \tag{2.7}$$

Explicit Euler method is also called forward Euler method, while implicit Euler method is called backward Euler method. The explanation how to obtain the explicit Euler method is described in the Subsection 2.2.2 and for the implicit Euler method there are many online resources such as Wikipedia[1].

As you can see the explicit method (2.6) uses the current value y_n and time t_n to compute the next value at the next time step y_{n+1}. In contrast to that the implicit method (2.7) uses both current value and next value. This means that in the explicit method, the next value y_{n+1} is given explicit, but in implicit method the value y_{n+1} is usually unknown in the equation and requires editing the equation to the needed form.

2.2.2 Euler method

Euler's method is the most simple numerical method for solving IVPs. Euler method has the following form (2.8). It is *first-order* explicit method, numerical methods are explained in Section 2.3.

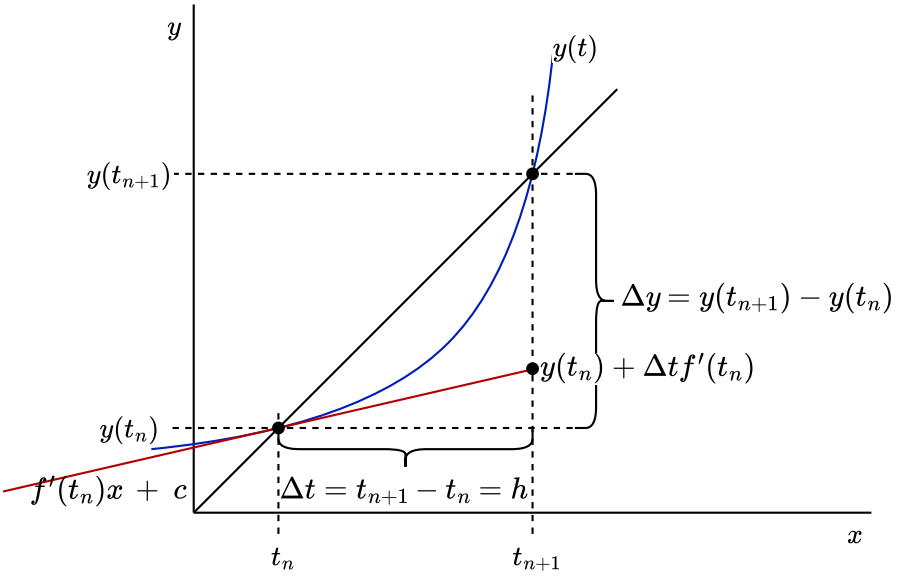
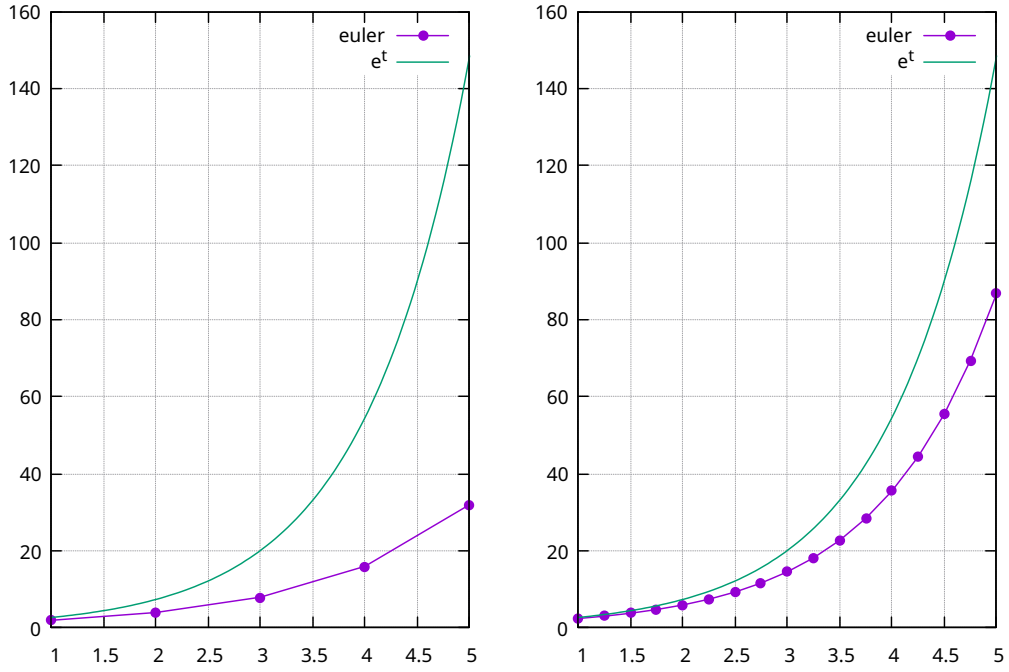


Figure 2.1: Visualization of derivative at time step t_n.

$$\begin{aligned} y_{n+1} &= y_n + hy'(t_n) \\ &= y_n + h \cdot F(t_n, y_n) \\ t_{n+1} &= t_n + h \end{aligned} \tag{2.8}$$

[1] https://en.wikipedia.org/wiki/Backward_Euler_method#Derivation



(a) Numerical solution of e^t using euler method with step size $h = 1$.

(b) Numerical solution of e^t using euler method with step size $h = 0.25$.

Derivation

To obtain this equation several methods can be used. One way to achieve this is by using Taylor's theorem, which will be also used later in Section 2.5.

Definition 2.2.1 (Taylor's Theorem)**.** The Taylor series of a real or complex-valued function $f(x)$, that is infinitely differentiable at a real or complex number a, is the power series

$$f(x) = f(a)+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\cdots+\frac{f^{(n)}(a)}{n!}(x-a)^n+\mathcal{O}((x-a)^{n+1}) \tag{2.9}$$

To obtain the Euler method use the (2.9) for the first order ($n = 1$). Doing so we are left with $f(x) = f(a) + \frac{f'(a)}{1!}(x - a) = f(a) + (x - a)f'(a)$ and substituting that $x = t_{n+1}$ and $a = t_n$ we get $f(t_{n+1}) = f(t_n) + (t_n - t_{n+1})f'(t_n) = f(t_n) + hf'(t_n)$

2.2.3 Runge-Kutta methods

The problem with Euler method is that it is only single-step method and at the single step it only computes linear approximation for the next step. There are multiple ways to fix this problem. For example multi-step methods use the value of multiple steps, rather then only the last value, to compute the next one. One such method is the two-step Adams–Bashforth:

$$y_{n+2} = y_{n+1} + \frac{3}{2}hF(t_{n+1}, y_{n+1}) - \frac{1}{2}hF(t_n, y_n), \tag{2.10}$$

where y_{n+2} is the approximated value for the next step. y_{n+1} is the last computed value. The two derivative approximations from previous step $F(t_{n+1}, y_{n+1})$ and the step before $F(t_n, y_n)$.

However, it is also possible to increase accuracy while using only single-step computation. One of the methods is the Runge-Kutta method, or rather it is a family of methods. The Runge-Kutta methods compute the function f at multiple points between the steps and combine the result to approximate the value of next step more accurately. The Runge-Kutta method or classic Runge-Kutta method is the "RK4", which is Runge-Kutta method with 4 stages, that are shown in the (2.11).

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

$$\begin{aligned} k_1 &= F(t_n, y_n) \\ k_2 &= F(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}) \\ k_3 &= F(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}) \\ k_4 &= F(t_n + h, y_n + hk_3) \end{aligned} \tag{2.11}$$

This method is popular for the implementation on the GPU, for example see the following articles [2] and [9].

2.3 Numerical computation errors

This section defines usual terminology associated with errors in numerical computations. The section is summary of the book [19]. For resources in English using slightly different notation the topics can be found in [6, p. 65, Section 2.1],

2.3.1 Local truncation error

The following is the definition of local truncation error and local error as is written in [19, p. 7-8]. Another definitions in English can be found the in books [5, p. 276], [6, p. 66, Section 2.1.1] and [8, p. 422].

Local truncation error (lte) is the error in a single-step computation with "localization condition", that is the approximate solution is equal to the analytical one $y_n = y(t_n)$

$$\tau_n = y(t_{n+1}) - y_{n+1} = y(t_{n+1}) - (y(t_n) + h\phi(t_n, y, h)),$$

where $y_{n+1} = y(t_n) + h\phi(t_n, y, h)$ is the computation of single-step method.

Notice that the lte does not happen in real computation as the localization condition is not fulfilled, which means $y_n \neq y(y_n)$.

"Local error" is the real error that arises while computing single-step from t_n to t_{n+1}.

$$\mathrm{le}_n = u_n(t_{n+1}) - y_{n+1},$$

where $u_n(t)$ is "local solution" of IVP

$$u_n'(t) = f(t, u_n(t)), \quad u_n(t_n) = y_n.$$

Local truncation error of Euler method

Euler method has the lte of $\mathcal{O}(h^2)$. For the derivation the Taylor expansion for the order of two is used.

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + (t_{n+1} - t_n)y'(t_n) + \frac{1}{2}(t_{n+1} - t_n)^2 y''(t_n) + \mathcal{O}\left((t_{n+1} - t_n)^3\right) \\ &= y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3) \end{aligned}$$

When we compare the Taylor's expansion value of the value and result of euler's method, we obtain the lte of Euler method.

$$\begin{aligned} \tau_{n+1} &= \left(y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3)\right) - \left(y_n + hy'(t_n)\right) \\ &= \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3) \\ &= \mathcal{O}(h^2) \end{aligned}$$

2.3.2 Global truncation error

The following is the definition of global truncation as defined in [19, p. 8]. Another definition in the English language can be found in [5, p. 339].

The "global truncation error" is the cumulative effect of the ltes committed in each step. It is the error at fixed time t_n.

$$e_n = y(t_n) - y_n.$$

Definition 2.3.1 (Order of numerical method)**.** The single-step method has the order of p if the global error is $\mathcal{O}(h^p)$.

Global truncation error of Euler method

For euler's method the number of steps n to reach $t_n = t_0 + n * h$ it is $n = (t_n - t_0)/h$, which is proportional to $n \propto 1/h$ and as previous mentioned the error in each step (lte) is $\mathcal{O}(h^2)$, which means the global truncation error of euler's method is proportional to h ($e_n = \mathcal{O}(1/h) * \mathcal{O}(h^2) = \mathcal{O}(h)$). Thus, euler's method is *first*-order numerical method.

Global truncation error of Runge-Kutta 4-stage method.

The Runge-Kutta method with 4 stages has the global truncation $e_n \in \mathcal{O}(h^4)$, which means the method has order 4. This statement is from [19, p. 14].

2.3.3 Round-off error

An alternative definition in the english form exist in the book [5, p. 18-20].

"Round-off error" is the error caused by rounding in the result of arithmetic operations, due to the use of finite precision floating-point numbers.

$$|\Delta y_{n+1}| = |\tilde{y}_{n+1} - y_{n+1}|,$$

where $\tilde{y}_{n+1}$ is the exact solution in real number of the arithmetic operations. The round-off grows bigger as step size gets smaller.

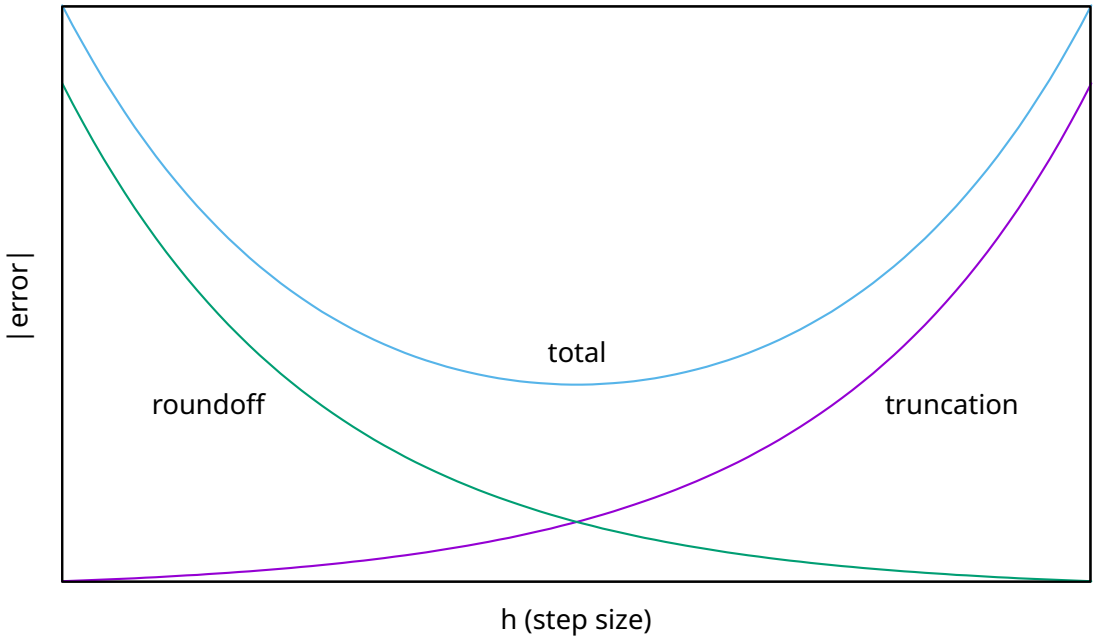


Figure 2.3: Correlation between round-off and truncation error with dependence on the step size (h).

One improvement for increasing the precision (lower the step size) while not increasing the round-off error is the use of larger floating-point datatype, such as multi-precision floating-point arithmetic libraries.

2.4 Stability of single-step methods

When talking numerical methods it is important to consider stability of the method. The stability of numerical method ensures the method is able to reach solution of given IVP.

This sections provides summary in stability of numerical methods, for more detailed information the following books are recommended [5, p. 339], [6, p. 74] and [8, p. 80]. For Czech speaking readers also the thesis [11, p. 62] and book [19] is recommended.

Definition 2.4.1 (Consistency)**.** A one-step difference-equation method with local truncation error $\tau_i(h)$ at the ith step is said to be "consistent" with the differential equation it approximates if

$$\lim_{h\to 0} \max_{1\leq i\leq N} |\tau_i(h)| = 0.$$

Using the definition from [5, p. 339, Definition 5.18]

Definition 2.4.2 (Convergence)**.** The following is definition for convergent method as defined in [5, p. 339, Definition 5.19]. Method is *convergent* if

$$\lim_{h\to 0} \max_{1\leq i\leq N} |e_i| = 0,$$

where e_i is the global discretization error approximation obtained from the difference method at the ith step.

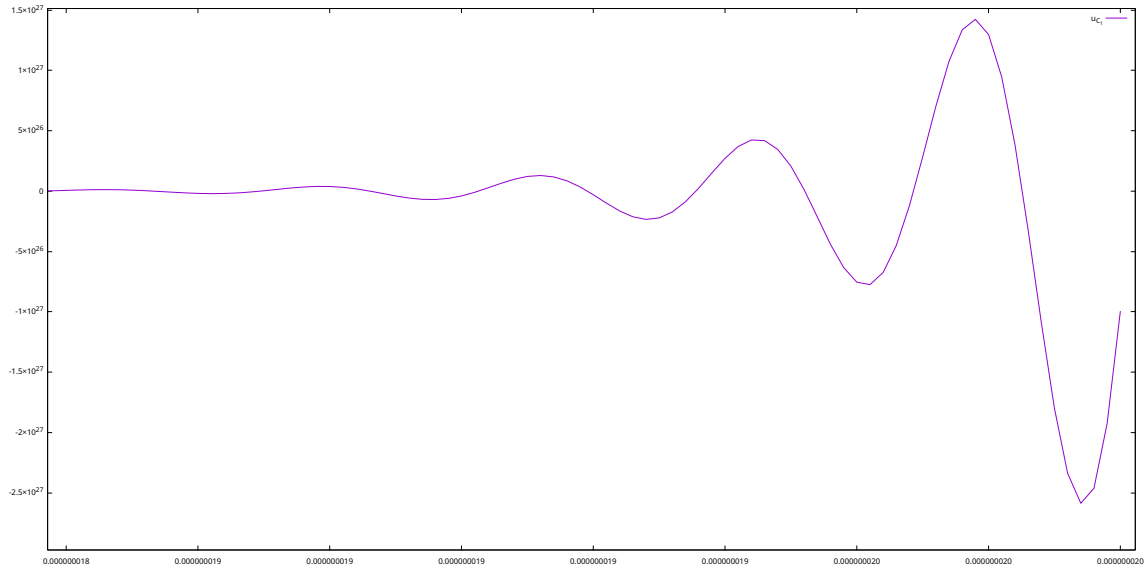


Figure 2.4: Example of non-convergent (divergent) numerical solution. The function rapidly oscillates between big positive and negative numbers and ends up with NAN (Not a number).

Definition 2.4.3 (Dahlquist test problem)**.** Dahlquist test problem is defined as follows

$$y' = \lambda y, \quad y(0) = 1, \quad \text{for } \lambda \in \mathbb{C}, \tag{2.12}$$

where the obvious analytical solution is $y(t) = e^{\lambda t}$.

Definition 2.4.4 (Stability function $R(z)$)**.** The function $R(z)$ is called the stability function of the method. In can be interpreted as the numerical solution after one step for the Dahlquist test equation from Definition 2.4.3, where $z = h\lambda \in \mathbb{C}$.

Without loss of generality that our one-step numerical method gives

$$y_{n+1} = y_n + h\Phi(t_i, y_n, h) = R(z)y_n,$$

as notes the article [7, p. 7, Chapter 3]. From which we get the stability function as

$$R(z) = \frac{y_{n+1}}{y_n}, \quad z = h\lambda \in \mathbb{C}, h > 0$$

Definition 2.4.5 (Stability domain)**.** Then the following set is called "stability domain" of the method

$$S = \{z \in \mathbb{C};\ |R(z)| \leq 1\}.$$

Using definition from the book [10, p. 17, Definition 2.9 & 2.10].

Definition 2.4.6 (A-stability)**.** A method is "A-stable" if its stability domain satisfies

$$S \supset \mathbb{C}^- = \{z \in \mathbb{C};\ \mathrm{Re}(z) \leq 0\}.$$

As defined in the book [10, p. 43, Definition 3.3].

Definition 2.4.7 (L-stability)**.** A method is "L-stable" if it is A-stable and if in addition

$$\lim_{z\to\infty} R(z) = 0.$$

The definition is used from the book [10, p. 45, Definition 3.7].

Theorem 2.4.1. The theorem as written in the book [5, p. 340, Theorem 5.20]. Suppose the initial-value problem

$$y' = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha,$$

is approximated by one-step difference method in the form

$$\begin{aligned} w_0 &= \alpha, \\ w_{i+1} &= w_i + h\phi(t_i, w_i, h). \end{aligned}$$

Suppose also that a number $h_0 > 0$ exists and that $\phi(t,w,h)$ is continuous and satisfies a Lipschitz condition, as defined in Definition 2.1.3, in the variable w with Lipschitz constant L on the set

$$D = \{(t,w,h) \mid a \le t \le b \text{ and } -\infty < w < \infty, 0 \le h \le h_0\}.$$

Then

1. The method is stable;
2. The difference method is convergent if and only if it is consistent, which is equivalent to
$$\phi(t,y,0) = f(t,y), \qquad \text{for } \forall a \le t \le b;$$
3. If a function τ exists and, for each $i = 1,2,\ldots,N$, the local truncation error $\tau_i(h)$ satisfies $|\tau_i(h)| \le \tau(h)$ whenever $0 \le h \le h_0$, then
$$|y(t_i) - w_i| \le \frac{\tau(h)}{L} e^{L(t_i-a)}.$$

2.4.1 Euler

Using the Dahlquist test problem (2.12) with explicit Euler method (2.6).

$$\begin{aligned} y' &= \lambda y, \\ y_{n+1} &= y_n + hf(t_n, y_n) \\ y_{n+1} &= y_n + h\lambda y_n \end{aligned}$$

where $y(0) = 1$ and $\lambda \in \mathbb{C}, \mathrm{Re}(\lambda) < 0$. From which we can obtain the stability function

$$\begin{aligned} y_{n+1} &= y_n + h\lambda y_n = y_n(1 + h\lambda) = y_n R(h\lambda) \\ \frac{y_{n+1}}{y_n} &= R(h\lambda) = 1 + h\lambda \\ R(z) &= 1 + z \end{aligned} \tag{2.13}$$

Using the obtained stability function we can now find the stability domain

$$\begin{aligned} S &= \{z \in \mathbb{C};\ |R(z)| \leq 1\} \\ &= \{z \in \mathbb{C};\ |1+z| \leq 1\} \\ &= \{\lambda \in \mathbb{C}, h > 0;\ |1+h\lambda| \leq 1\} \end{aligned} \tag{2.14}$$

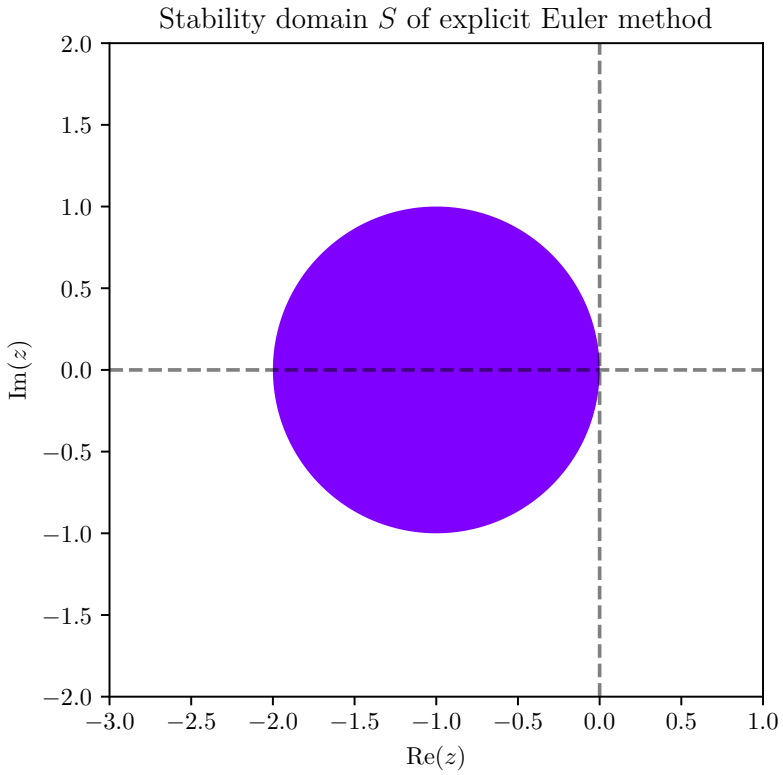


Figure 2.5: Stability domain S of Euler method (2.14) visualized on complex plane.

2.4.2 Runge-Kutta 4

Analogously to the previous Subsection 2.4.1, it is possible to obtain the stability region for the method 4-th order Runge-Kutta method combining the definition of the method (2.11)

with the Dahlquist test problem (2.12).

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}(\lambda y_n + 2\lambda\left(y_n + \frac{1}{2}h\lambda y_n\right) + 2\lambda\left(y_n + \frac{1}{2}h\lambda\left(y_n + \frac{1}{2}h\lambda y_n\right)\right) \\
&\quad + \lambda\left(y_n + h\left(\lambda\left(y_n + \frac{1}{2}h\lambda\left(y_n + \frac{1}{2}h\lambda y_n\right)\right)\right)\right)) \\
y_{n+1} &= y_n + \frac{h}{6}(\lambda y_n + 2\lambda y_n + h\lambda^2 y_n + 2\lambda y_n + h\lambda^2 y_n + \frac{1}{2}h^2\lambda^3 y_n \\
&\quad + \lambda y_n + h\lambda^2 y_n + \frac{1}{2}h^2\lambda^3 y_n + \frac{1}{4}h^3\lambda^4 y_n) \\
\frac{y_{n+1}}{y_n} &= 1 + \frac{h}{6}(\lambda + 2\lambda + h\lambda^2 + 2\lambda + h\lambda^2 + \frac{1}{2}h^2\lambda^3 + \lambda + h\lambda^2 + \frac{1}{2}h^2\lambda^3 + \frac{1}{4}h^3\lambda^4) \\
&= 1 + \frac{h}{6}(6\lambda + 3h\lambda^2 + h^2\lambda^3 + \frac{1}{4}h^3\lambda^4) \\
&= 1 + (h\lambda + \frac{1}{2}h^2\lambda^2 + \frac{1}{6}h^3\lambda^3 + \frac{1}{24}h^4\lambda^4) \\
&= 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4
\end{aligned}
\tag{2.15}
$$

From which we receive the following stability region:

$$
\begin{aligned}
S &= \{z \in \mathbb{C};\ |R(z)| \leq 1\} \\
&= \{z \in \mathbb{C};\ |1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4| \leq 1\} \\
&= \{\lambda \in \mathbb{C}, h > 0;\ |1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4| \leq 1\}
\end{aligned}
\tag{2.16}
$$

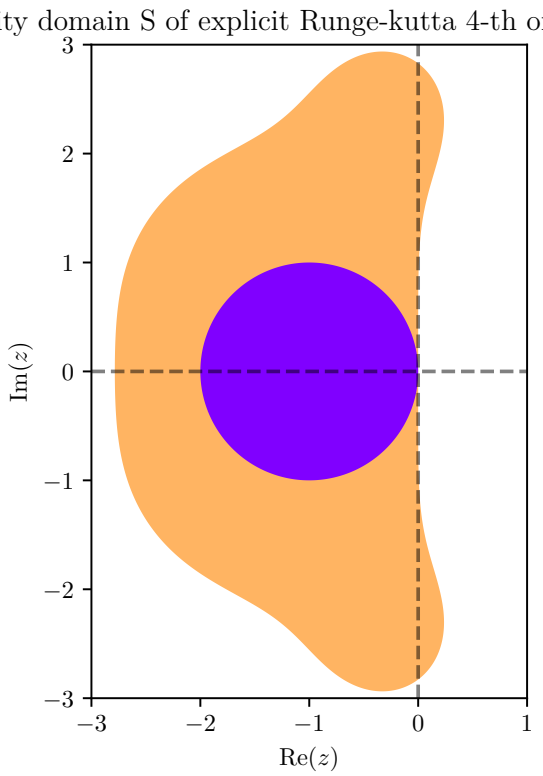


Figure 2.6: Stability domain S of explicit Runge-Kutta 4-th order method (2.16) visualized on complex plane.

2.5 Higher order method — Modern Taylor series method

This section defines the modern taylor series method as defined in [16, p. 37, Chapter 3], [14, p. 69, Chapter 4] with the notation from [18].

As the name implies this method is based on the Taylor series expansion (2.9). And in comparison to the previously mentioned numerical methods (Subsection 2.2.2 and Subsection 2.2.3) the MTSM uses higher order terms of the function derivation than only the first derivatives.

Consider IVP as defined in Definition 2.1.1. Let's assume the current time step is i with the IVP's solution at the current timestep being $y(t)_i$. Using the Taylor expansion we try to find the solution at the next time step $i+1$, where the step size is $h = x - a$. The taylor's function f is y for us and specifically the approximated function is the solution we are looking for $f(x) = y_{i+1}$ and already known point $f(a) = y_i$. The biggest change in notation is the function F, which is the first derivative of our solution $y_i = hF(t_i, y_i)$. The above mentioned substitution combines into the (2.17) below. Then each term can be rewritten as a special function for n-th term in i-th step $p(n)_i$

$$y_{i+1} = y_i + hF(t_i, y_i) + \frac{h^2}{2!}F^{(1)}(t_i, y_i) + \ldots + \frac{h^n}{n!}F^{(n-1)}(t_i, y_i) \tag{2.17}$$

$$y_{i+1} = p(0)_i + p(1)_i + p(2)_i + \ldots + p(n)_i \tag{2.18}$$

Linear differential equation

Considering only linear differential equations we can rewrite the IVP into following form:

$$\mathbf{y}(t)' = \mathbf{A}\mathbf{y} + \mathbf{b}, \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{2.19}$$

where $\mathbf{A}$ is Jacobian of the linear differential equations and $\mathbf{b}$ is the constant right-hand side. Using the previous definition and chain rule we can derive higher derivatives.

$$\begin{aligned}
\mathbf{y}''(t) = \frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{y}'(t)] &= \frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{A}\mathbf{y}(t) + \mathbf{b}] \\
&= \frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{A}\mathbf{y}(t)] + \frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{b}] \\
&= \mathbf{A}\frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{y}(t)] + 0 \\
&= \mathbf{A}\mathbf{y}'(t) \\
&= \mathbf{A}\left(\mathbf{A}\mathbf{y}(t) + \mathbf{b}\right)
\end{aligned} \tag{2.20}$$

Analogically we can derive other higher derivatives.

$$\begin{aligned}
\mathbf{y}'''(t) = \frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{A}\mathbf{y}'(t)] &= \mathbf{A}\frac{\mathrm{d}}{\mathrm{d}t}[\mathbf{y}'(t)] \\
&= \mathbf{A}\mathbf{y}''(t) \\
&= \mathbf{A}\left[\mathbf{A}\left(\mathbf{A}\mathbf{y}(t) + \mathbf{b}\right)\right] \\
&= \mathbf{A}^2\left(\mathbf{A}\mathbf{y}(t) + \mathbf{b}\right)
\end{aligned} \tag{2.21}$$

Generally, we are able to tell that:

$$\begin{aligned}
\mathbf{y}^{(n)}(t) &= \mathbf{A}^{(n-1)}\left(\mathbf{A}\mathbf{y}(t) + \mathbf{b}\right) \\
&= \mathbf{A}\mathbf{y}^{(\mathbf{n-1})}(t)
\end{aligned} \tag{2.22}$$

Coming back to our function we can see that

$$\begin{aligned}
\mathbf{p}(0)_i &= \mathbf{y}_i \\
\mathbf{p}(1)_i &= h\left(\mathbf{A}\mathbf{y}_i + \mathbf{b}\right) \\
\mathbf{p}(n)_i &= \frac{h}{n}\mathbf{A}\mathbf{p}(n-1)_i, \qquad \text{for } n > 1
\end{aligned} \tag{2.23}$$

2.5.1 Stability

Applying the MTSM on Dahlquist test problem Definition 2.4.3.

$$y' = \lambda y,$$

$$y_{i+1} = y_i + hF(t_i, y_i) + \frac{h^2}{2!}F^{(1)}(t_i, y_i) + \ldots + \frac{h^n}{n!}F^{(n-1)}(t_i, y_i) \tag{2.24}$$

$$y_{i+1} = \sum_{n=0}^{\text{maxOrder}} \frac{h^n}{n!}F^{(n-1)}(t_i, y_i) \tag{2.25}$$

$$= \sum_{n=0}^{\text{maxOrder}} \frac{h^n}{n!}\lambda^n y_i \tag{2.26}$$

$$R(z) = \sum_{n=0}^{\text{maxOrder}} \frac{1}{n!}z^n \tag{2.27}$$

where $y(0) = 1$ and $\lambda \in \mathbb{C}, \text{Re}(\lambda) < 0$. Interestingly, putting the maxOrder $= 1$ we have the stability function of

$$R(z) = 1 + z, \tag{2.28}$$

which is exactly same as stability function of explicit Euler method (2.13) in Subsection 2.4.1.

Similarly, putting the maxOrder $= 4$ the stability function becomes

$$R(z) = 1 + z + \frac{1}{2!}z^2 + \frac{1}{3!}z^3 + \frac{1}{4!}z^4, \tag{2.29}$$

which is exactly same as stability function of explicit Runge-kutta 4-th order method (2.15) in Subsection 2.4.2.

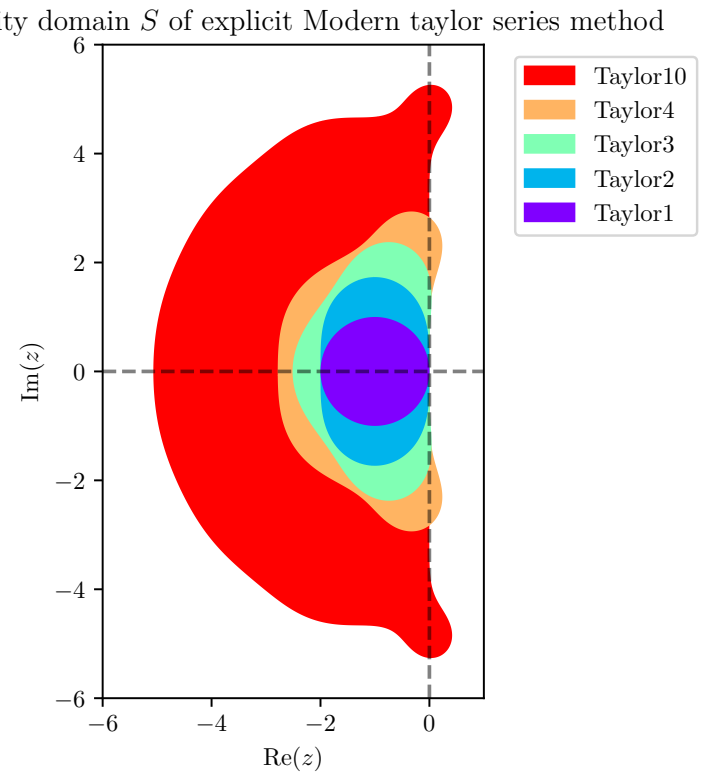


Figure 2.7: Stability domain S of explicit modern Taylor series method (2.27) visualized on complex plane. The visualization contains MTSM with orders 1, 2, 3, 4 and 10.

2.5.2 Precalculation

The idea comes from Ph.D thesis [14, p. 83-84, Section 4.9]. Consider the MTSM with fixed maximum order n, then the MTSM can be rewritten:

$$
\begin{aligned}
\mathbf{y}_{i+1} &= \mathbf{y}_i + h\mathbf{y}'_i + \frac{h^2}{2!}\mathbf{y}''i + \cdots + \frac{h^n}{n!}\mathbf{y}_i^{(n)} + \mathcal{O}(h^{n+1}) \\
&= \mathbf{y}_i + h(\mathbf{A}\mathbf{y}_i + \mathbf{b}) + \frac{h^2}{2!}\mathbf{A}(\mathbf{A}\mathbf{y}_i + \mathbf{b}) + \cdots + \frac{h^n}{n!}\mathbf{A}^{n-1}(\mathbf{A}\mathbf{y}_i + \mathbf{b}) \\
&= \left(1 + h\mathbf{A} + \frac{h^2}{2!}\mathbf{A}^2 + \cdots + \frac{h^n}{n!}\mathbf{A}^n\right)\mathbf{y}_i + \left(h\mathbf{b} + \frac{h^2}{2!}\mathbf{A}\mathbf{b} + \cdots + \frac{h^n}{n!}\mathbf{A}^{n-1}\mathbf{b}\right) \\
&= \left(1 + h\mathbf{A} + \frac{h^2}{2!}\mathbf{A}^2 + \cdots + \frac{h^n}{n!}\mathbf{A}^n\right)\mathbf{y}_i + \left(h + \frac{h^2}{2!}\mathbf{A} + \cdots + \frac{h^n}{n!}\mathbf{A}^{n-1}\right)\mathbf{b} \\
&= \left(1 + h\mathbf{A} + \frac{h^2}{2!}\mathbf{A}^2 + \cdots + \frac{h^n}{n!}\mathbf{A}^n\right)\mathbf{y}_i + \left(1 + \frac{h}{2!}\mathbf{A} + \cdots + \frac{h^{n-1}}{n!}\mathbf{A}^{n-1}\right)h\mathbf{b} \\
&= \left(\sum_{k=0}^{n}\frac{h^k}{k!}\mathbf{A}^k\right)\mathbf{y}_i + \left(\sum_{k=0}^{n-1}\frac{h^{k+1}}{(k+1)!}\mathbf{A}^k\right)\mathbf{b} \\
\mathbf{y}_{i+1} &= \mathbf{A}_y\mathbf{y}_i + \mathbf{A}_b\mathbf{b}
\end{aligned}
\tag{2.30}
$$

It is obvious from the (2.30), that it applies for all steps and that the matrix $\mathbf{A}$ and vector $\mathbf{b}_b = \mathbf{A}_b\mathbf{b}$ is same for all the steps, which means it can be precalculated before the actual numerical computation using the MTSM.

The pre-computed matrix $\mathbf{A}_y$ and vector $\mathbf{b}_b$ can be pre-computed also in parallel, which is in detail explained in the Ph.D. thesis [14, p. 83-84].

Chapter 3

General-purpose computing on graphics processing units

This chapter explains the basic architecture of GPU devices and basic concepts for running OpenMP code on the GPU.

GPGPU is a method of using GPUs to compute general algorithms using parallelization. Sometimes, GPGPUs are referred to as "accelerators". Because it can compute certain type of programs – compute-bound problems – significantly faster than CPUs.

GPUs have a massively parallel architecture consisting of thousands of threads. This means that, to effectively use the computing power of GPGPUs, programs running on them use hundreds or thousands of threads in parallel. Compared to CPU threads, GPGPU's threads are more lightweight – they have fewer registers and smaller caches, but many more computing units. It utilizes the Single Instruction, Multiple Threads (SIMT) execution model, which is similar to Single Instruction, Multiple Data (SIMD). In SIMT single instruction is executed across multiple threads simultaneously. The smallest group of threads being executed is referred to as a "warp", which typically consists of 32 threads for NVIDIA and either 32 or 64 threads for AMD GPUs.

NVIDIA	AMD
CUDA core	vector ALU (VALU)
tensor cores	matrix cores
streaming multiprocessor (SM)	compute unit (CU) / workgroup processor (WGP)

Table 3.1: Different naming between GPU vendors

CUDA (NVIDIA)	HIP (AMD)	OpenMP
warp	wavefront	simd
(cuda) block	block	team
grid	grid	league
`threadIdx.x`	`threadIdx.x`	`omp_get_thread_num()`
`threadIdx.y`	`threadIdx.y`	`omp_get_team_size()`
`blockDim.x`	`blockDim.x`	`omp_get_max_threads()`
`blockIdx.x`	`blockIdx.x`	`omp_get_team_num()`
`__syncthreads()`	`__syncthreads()`	`#pragma omp barrier`

Table 3.2: Different naming between GPU programming platforms

3.1 Architecture

The basic building block of GPUs is the SM. Each SM contains:

- set of registers,
- shared memory (SHM),
- cuda cores, or VALUs for AMD GPUs,
- tensor cores, or matrix cores,
- special function units (SFUs),
- multiple load/store units.

Although the exact number and composition of units depends on generation and type of GPU.

Set of registers The set of registers in SM is shared by all active threads. Even though a warp would be ready to be switched in; SM cannot do that if the warp requires more registers than what the SM can currently provide.

Shared memory (SHM) Shared memory (SHM), which is programmer controlled cache memory on the SM chip with size in order of tens of kilobytes. This memory is structured into memory banks, which requires specific access patterns to avoid bank conflicts. Each bank can hold 32-bit data and usually corresponds to the number of threads in a warp. Accessing the shared memory without bank conflicts should be comparable to accessing registers. And accessing with bank conflicts is slower by an order, but it is still faster than accessing the GPU's memory. There are two rules for avoiding bank conflicts:

1. each thread requests data from different memory bank,
2. each thread requests the same data (memory address) from same memory bank — results in a broadcast.

Cuda cores of VALUs Cuda cores or VALUs, for NVIDIA GPUs and AMD GPUs respectively, compute 32-bit floating-point (FP32) operations: addition, multiplication and fused-multiply-accumulate (FMA). Depending on a specific GPU card, they can also contain units fixed-point operations. When cuda cores contain both FP32 and fixed-point units, using the fixed-point units decreases the maxim possible floating-point operations per second (FLOPS). Therefore, simple computation of indices into array can lead to decreased FLOPS.

Tensor cores or matrix cores, for NVIDIA GPUs and AMD GPUs respectively, are special units to accelerate matrix operations.

Special function units (SFUs) accelerate computation of computation-heavy functions, such as $\sin(x)$, $\cos(x)$, $\sqrt{x}$, e^x.

Load/store units (LSUs), as the name implies, are units that load/store data to/from GPU's memory. LSU have very important role, because they are able to fetch memory in parallel to computation units working. This is done to hide latencies with switching waiting warp with warp ready to compute.

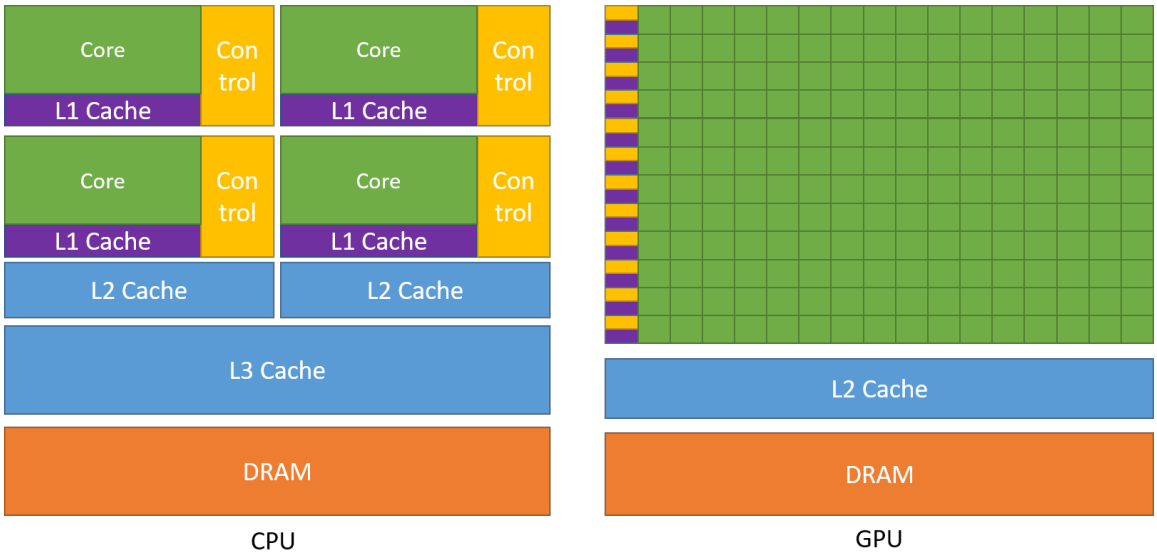


Figure 3.1: Comparison between CPU and GPU architecture [15, https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/introduction.html].

Streaming multiprocessors are cache incoherent, i.e, writes to cache of one SM does not propagate to the other ones. As result, GPUs are trivially scalable by simply adding more streaming multiprocessors.

There is a theoretical limit "Theoretical Occupancy" for the usage of SM. The limiting factors for occupancy are:

- number of warps per SM,
- number of blocks per SM,
- number of registers shared by active threads per SM,
- size of SHM per SM.

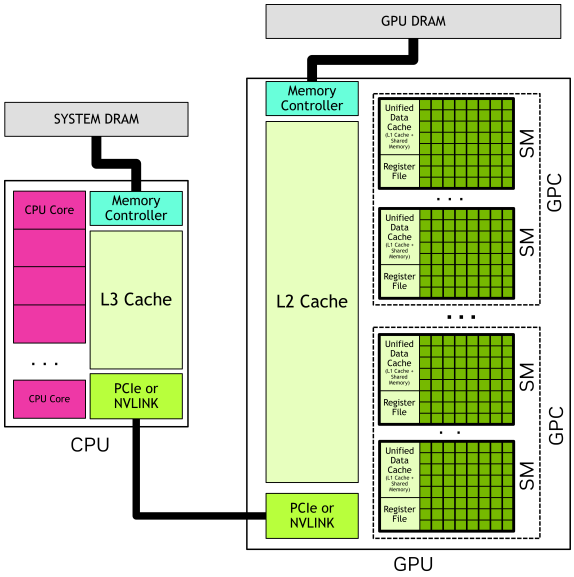


Figure 3.2: GPU and CPU system diagram [15, https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html].

From software perspective the GPU is designed into an unusual hierarchical structure. Going from bottom up the smallest unit is a single thread. Cluster of threads combines into a single "thread block". The block can be defined either in 1, 2 or 3 dimensions, denoted as x, y and z respectively, depending on the structure defined by a programmer.Dimensions are only software abstractions to help programmers decompose the problems into parallelizable programs. Typically, single dimension is used for linear problems and computation over arrays. Two dimensions are used for operations with images and matrices. And spatial problems are done with three dimensions.

All blocks together create a "grid". Similarly to thread blocks grid can be multi-dimensional. Programmer then can retrieve index of the block in grid and index of the thread in the block together with the sizes of each dimension for both thread blocks and the grid. Figure 3.3 shows example of such grid.

The grid gives the full structure of a "kernel". A kernel is the function that is called from host (CPU) and runs on the GPU. Kernel has to be launched together with the grid configuration. Additional non-mandatory configuration for the kernel launch are runtime/-dynamical size of SHM and GPU stream. GPU streams are used for asynchronous execution of the kernel.

One of the most important concepts for parallel computing on the GPU is the mapping between software architecture – threads, blocks – and hardware architecture – VALU, SM, SHM. The most important mapping is between thread block and SM, where the thread block runs only on a single SM. This mapping results in several notable features. The SHM can be only used between threads in a single thread block, since each SM has its own shared memory. Since SM are independent of each other, that means that thread blocks have to be independent on each other. Specifically, there is no synchronization and communication between thread blocks — thread blocks can be scheduled at different time intervals. In newer versions of CUDA and HIP APIs this issue is solved with "cooperative groups", which allow synchronization of multiple thread blocks. For synchronization of threads inside a thread block, there is a special keyword `__syncthreads()`.

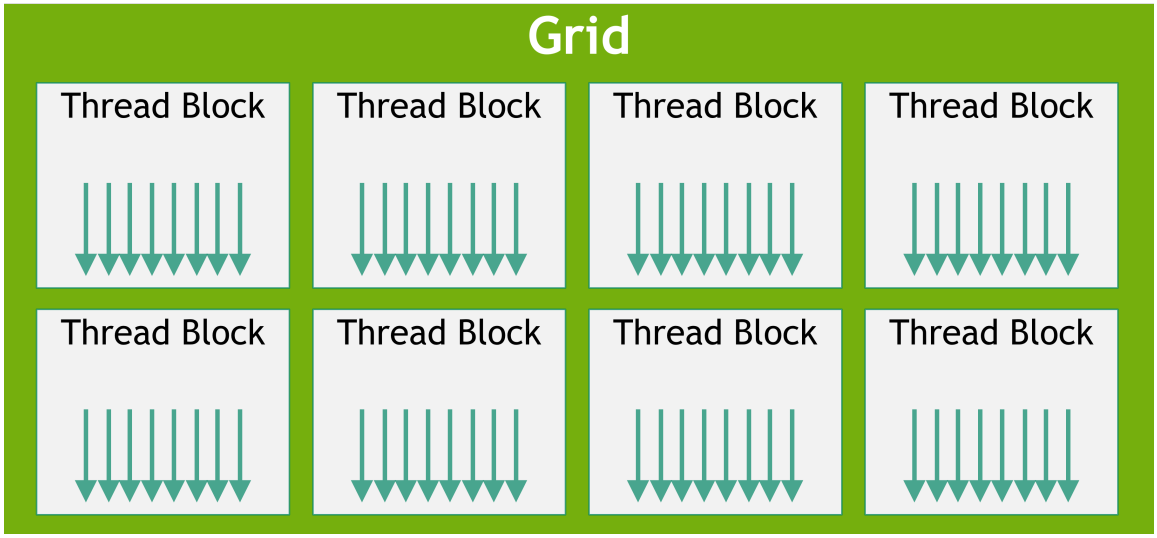


Figure 3.3: Example visualization of a kernel with 1 dimensional thread block structure and 2 dimensional grid [15, https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html].

Another one-to-one mapping is between threads and VALUs / CUDA cores. Even though the mapping is one-to-one, threads are not executed individually. As previously mentioned, the smallest unit of threads that are executed simultaneously is warp. Therefore, the execution of a thread block is split into multiple warps. Such division allows for a mechanism called "warp-swapping", which is replacing currently active threads – which are waiting for data – with a warp that is ready for execution.

"Warp divergence" occurs when one part of the warp wants to execute different instruction than the other part of the warp (whole warp cannot execute single instruction). The issue with warp divergence is that it is a performance hit. Especially, on older GPUs the code had to be serialized to solve the divergence. Nowadays, individually thread scheduling improves the performance, but small penalty is still present.

3.2 Options

This section describes considered options for the implementation of this thesis, their advantages, disadvantages and reasoning for the chosen technology.

The MTSM was previously implemented in Matlab ([16]), Python and Julia ([16]), PETSc ([14]). Matlab, Python, Julia and even PETSc has options to run on GPU, but I am unfamiliar with such platforms and their capabilities on the GPU.

The medium level options were OpenACC and OpenMP, which both are extensions for C, C++ and Fortran. Both are able to run on NVIDIA's and AMD's GPUs. However, it is currently only possible to compile OpenACC with NVIDIA's toolkit and GCC, meanwhile OpenMP is compilable with both GCC and CLANG. The main benefit of these platforms are portability, as the directives allows the code to be run on CPU and devices supported by the used compiler. The other benefit is the interoperability with the low level options, since these platforms provides the possibility to use functions from platform specific libraries. They also provide decent control over the resources, but it still depends heavily on the

compiler. The miscompilation can cause significant performance penalty, as for example described in Subsection 3.3.5.

The Compute Unified Device Architecture (CUDA) and C++ Heterogeneous-Compute Interface for Portability (HIP) platforms were considered as the low level options. Similar to the medium level options CUDA is proprietary NVIDIA's platform and as such supports only NVIDIA's GPUs. Although there is some work to provide CUDA computation platform on other devices, these works are third party and in early stages. On the other hand HIP is supposed to run on both AMD's and NVIDIA's GPU. These platforms provide fined control over the GPU's resources. That being said, at the time of writing, NVIDIA is the leader in the GPU space vastly due to their superior toolchain support. Their toolchain is used by well-known platforms, such as TensorFlow, PyTorch, and contains many highly optimized libraries, for example cuBLAS, cuDNN and others. Also their performance analysis and debugging tools are ahead of their competition, such most known tool is NVIDIA Nsight systems.

The chosen technology was OpenMP with eventual optimizations of kernels in HIP. HIP ended unused in the implementation, since we focused on experimentation on more coarse level – such as dense matrix computation, sparse matrix format CSR, precalculation of Jacobian – instead of more fined optimizations. OpenMP allows to run the code on both GPU and CPU (multi-core and single-core), which provides fast and easy way to establish numerical error cause by the parallelization. OpenMP also supports to parallelize the code incrementally.

3.3 OpenMP

The following section describes basic principles for programming for GPU with OpenMP. The Subsection 3.3.1 shows how to parallelize code regions for running on multiple threads on the GPU. The Subsection 3.3.2 and Subsection 3.3.3 shows how to transfer and allocated data on the GPU. The Subsection 3.3.4 explains environment variables that can be used to control the running program. And last Subsection 3.3.5 show how the unexpected compilation can slow the program.

3.3.1 Parallelization for GPU

This sections explains basics of parallelization for GPU using OpenMP offloading, for additional resource I recommend the website[1], although incomplete, it provides nice overview of some basics. For quick reference OpenMP provides reference guides[2] and for last, but no least, the full OpenMP specification[3]. The compiler that I used for the implementation is using the OpenMP version 4.5 [1].

Because OpenMP is high level programming paradigm not only does it allow for parallelization for GPUs, it also, and mainly, allows parallelization for multi-threaded CPU. The use of offloading to GPU (or any other device such as coprocessor, FPGA and others) can be spotted by the use of OpenMP pragma `target`.

The model of parallelization using OpenMP offloading is host-device model, where there is single host, usually CPU, and one or more target devices. The host and device have separate memory address space, unless they have unified share memory space. In this

[1] https://enccs.github.io/openmp-gpu/
[2] https://www.openmp.org/resources/refguides/
[3] https://www.openmp.org/specifications/

model, the host has the main role – memory management, work distribution and execution for all (host and all devices). The execution with target offloading follows this structure:

1. host prepares and copies the data for the target device(s) – memory allocation, copy of data,
2. host offloads specified code regions (by the `target` pragma) for execution on the target device(s),
3. host collects the data from device(s) and frees allocated memory on the device(s).

```
#pragma omp target [clause [ [,] clause] ... ]
{
    // Code to run on target device
}
```

Listing 3.1: The simplest way of running code region on the target device is using the above mentioned pragma `target` as described in OpenMP specification [1, p. 103, Section 2.10.4]

However, the code region in Listing 3.1 is run as a single "task" on the device, which is equivalent to running a single thread on the GPU and means it is run without parallelism and does not utilize the GPU architecture. The execution of the code region is sequential and synchronous, unless the clause `nowait` is specified, which disables synchronization at the end of the code region.

To create parallel regions on the device the `teams` directive is used as shown in Listing 3.2. The `teams` directive "creates a league of thread teams where the master thread of each team executes the region", as described in [1, p. 114, Section 2.10.7]. *Threads within a team can synchronize but there is no synchronization between teams.* Which follows the GPU architecture of `blocks` as noted in Table 3.2 and as described in Section 3.1. It is possible to limit the number of teams created in a league with the clause `num_teams(number)` in code. which can be received in the teams region using the function `int omp_get_num_teams(void);`. As it specified by the definition and the Figure 3.4b illustrates, the execution thread in each team contains same code region and same data. *There is no implicit barrier at the end of a* ***`teams`*** *construct.*

```
#pragma omp target teams [clause [ [,] clause] ... ]
{
    // Code to run on target device in each team
}
```

Listing 3.2: OpenMP teams pragma as described in OpenMP specification [1, p. 114, Section 2.10.7].

In order to split the work between each team, the directive `distribute` can be used, as shown in the Listing 3.3. "The iterations of the for loop are distributed across the master threads of all teams", as described in [1, p. 117, Section 2.10.8]. Which means there is no distribution between the threads within a team, as it is illustrated in Figure 3.4c. Similarly to the GPU `blocks`, there in no guarantee for the execution order of teams. For example, the team with end iterations of the loop might run before the team with first iterations in the first run, but in different order in the next run. Similarly to `teams`, *there is no implicit barrier at the end of a* ***`distribute`*** *construct.*

```
#pragma omp target teams distribute [clause [ [,] clause] ... ]
for (...) {
    // Code to be distributed and run in each team
}
```

Listing 3.3: OpenMP teams distribute as described in OpenMP specification [1, p. 117, Section 2.10.8].

Since `team distribute` split the work only among teams, not threads in teams, the combined directive `parallel for` can be used. Firstly, the directive `parallel` "forms a team of threads and starts parallel execution." The declaration of a parallel region is shown in Listing 3.4. The code region of a `parallel` is executed on all spawned threads inside a single team. It is, in a way, similar to `teams`, but with more fine-grained parallelization for all created threads in a team, instead of master threads in all teams. The maximum number of threads created in a team can be controlled by the `num_threads(number)` clause or by the `OMP_NUM_THREADS` environment variable.

```
#pragma omp target parallel [clause [ [,] clause] ... ]
{
    // Code to be executed for each thread created in a team
}
```

Listing 3.4: OpenMP parallel as described in OpenMP specification [1, p. 46, Section 2.5].

Analogically to combination of `teams` and `distribute`, the directive `parallel` can be combined with directive `for` to split the iterations of for loop between threads in a team. The use of the combined directive is shown in Listing 3.5 and the Figure 3.4d illustrates the work-sharing.

```
#pragma omp target parallel for [clause [ [,] clause] ... ]
for (...) {
    // Code to be distributed and run for each thread created in a team
}
```

Listing 3.5: OpenMP parallel for as described in openmp specification [1, p. 56, Section 2.7.1].

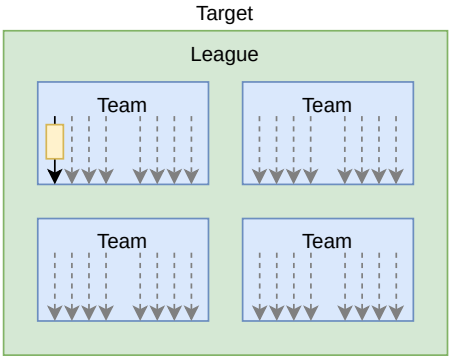
The above mentioned directives can be combined together as specified in [1, p. 124, Section 2.11]. The combination of all above mentioned directives provide effective parallelization for easily parallelizable problems on GPU. The combined constructs are shown in the Listing 3.6 and the work-sharing is illustrated in Figure 3.4e. This parallelism of independent for loop iterations is effective for GPU because:

1. The `teams` construct creates a league of independent execution teams, either specified by compiler of `num_teams` clause.
2. The `distribute` construct gives each team range of the loop iterations to compute.
3. The `parallel` construct creates more fine-grained parallelism by creating more threads inside each time.
4. The `for` construct further splits the range of iterations of each team into work for each thread within the team.

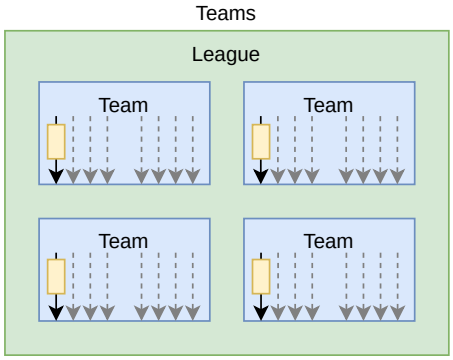
Thus copying the GPU architecture as explained in Section 3.1.

```
#pragma omp target teams distribute parallel for [clause [ [,] clause]
    ... ]
for (...) {
    // 1. Create league of teams
    // 2. Split the loop between teams
    // 3. Create threads in each team
    // 4. Split the work of team into threads
}
```

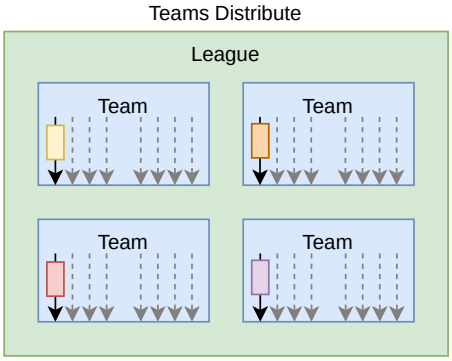
Listing 3.6: OMP teams distribute parallel for as described in OpenMP specification [1, p. 141, Section 2.11.14].



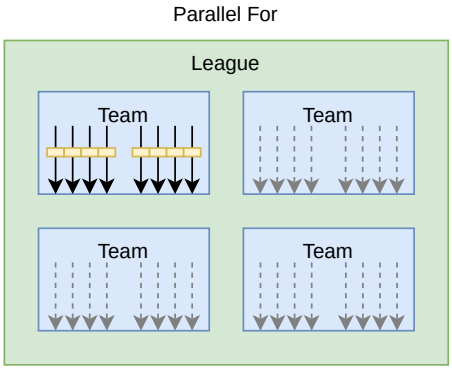
(a) Illustration of created threads and distribution of data for `target` directive.



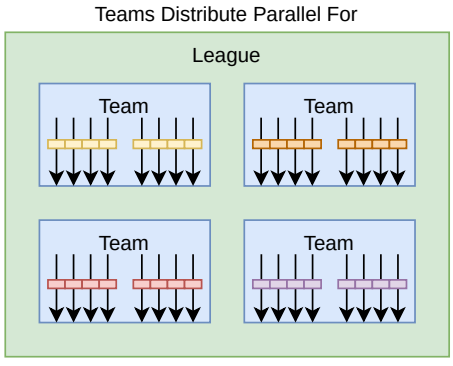
(b) Illustration of created threads and distribution of data for `teams` directive.



(c) Illustration of created threads and distribution of data for `teams distribute` directive.



(d) Illustration of created threads and distribution of data for `parallel for` directive.



(e) Illustration of created threads and distribution of data for `teams distribute parallel for` directive.

Figure 3.4: Illustration of different OpenMP directives for parallelization.

3.3.2 OpenMP data mapping

The following text is summary of data mapping as described in openmp specification [1, p. 215, Section 2.15.5].

```
#pragma omp target map([ [map-type-modifier[,]] map-type : ] list)
{
    // Maps data to device and runs the region
}
```

Listing 3.7: OpenMP target map as described in OpenMP specification [1, p. 215, Section 2.15.5].

Where `map-type-modifer` is `always`, `list` is: "A comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types." and `map-type` is one of the following:

alloc At the beginning of the region, variables in the list are allocated on the device, but are left uninitialized (values on the device are undefined).

release At the end of the region, variables corresponding reference count is decremented by one.

delete At the end of the region, variables corresponding reference count is set to zero.

to At the beginning of the region, variables in the list are allocated and filled with values from the host.

from At the end of the region, the data of variables are copied from the device to the variables on the host and the corresponding reference count is decremented by one. At the begging of the region the data is left uninitialized and if necessary they are allocated on the device.

tofrom Combination of `to` and `from`.

The OpenMP has implicit mapping for used variables:

- scalar variables are mapped as `firstprivate`,
- non-scalar variables are mapped as `tofrom`,
- pointers are mapped as pointer values.

The last point means that when mapping pointer to an array, no array data is copied over, which might not be expected behavior. To properly copy, allocate, delete arrays, OpenMP provides array section notation in specification [1, p. 44, Section 2.4]: section `array[start:length]`, where the `start` and `length` is *number of elements*, not bytes. The notation `array[:length]` corresponds to `array[0:length]`.

3.3.3 OpenMP target data constructs

The following is the summary of target device data constructs as described in openmp specification [1, p. 95, Section 2.10].

The Listing 3.8 shows "structured data region" the `target data` and maps the data with same rules as described in Listing 3.7, but *does not run the code region.* Expects later pragma that runs the on device as described in Subsection 3.3.1.

```
#pragma omp target data map([ [map-type-modifier[,]] map-type : ] list)
{
    // Maps data to device, but does not run the region
}
```

Listing 3.8: OpenMP structured data region as described in openmp specification [1, p. 95, Section 2.10.1].

The structured data regions only allows mapping within single functions and the memory exits only within the data region, which is unsuitable for many applications. In order to provide mapping in different functions, the OpenMP standard provides "unstructured data regions" `target enter data` and `target exit data` to start and end mapping, respectively, as shown in Listing 3.9. With the use of unstructured data regions, the data on the device exists until it is explicitly deallocated.

```
void start(...) {
#pragma omp target enter data map([ [map-type-modifier[,]] map-type : ]
    list)
    {
        // Starting mapping - allocation and copy to device
    }
}

void end(...) {
#pragma omp target exit data map([ [map-type-modifier[,]] map-type : ]
    list)
    {
        // Ending mapping - deallocation and copy from device
    }
}
```

Listing 3.9: OpenMP unstructured data region as described in openmp specification [1, p. 97-102, Section 2.10.2 and 2.10.3] .

The unstructured data regions are useful, but do not allow to copy data to and from device without deallocation (or to be exact without increasing and decreasing the reference count). For this use case the OpenMP standard provides `target update` as shown in Listing 3.10.

```
1 start();
2 // ...
3 #pragma omp target update to(list)
4 // Compute on device
5 #pragma omp target update from(list)
6 // ...
7 end();
```

Listing 3.10: OpenMP target update data as described in openmp specification [1, p. 107, Section 2.10.5].

The `motion-clause to` on Line 3 in Listing 3.10, copies the variables from host *to target device*. Analogically, the `motion-clause from` on Line 5 in Listing 3.10, copies the variables *from target device* back to host.

3.3.4 OpenMP environment variables

OpenMP provides some control over the runtime of program with several environment variables defined in [1, p. 290, Chapter 4].

`OMP_NUM_THREADS` sets the number of threads to be created and used in parallel regions, similarly to `num_threads` for the `parallel` construct.

`OMP_THREAD_LIMIT` specifies the total amount of threads that can be used by the whole program.

`OMP_DEFAULT_DEVICE` sets the default device to use for offloading in case multiple devices are present, such as multiple GPUs.

`OMP_DISPLAY_ENV` controls the information display regarding the OpenMP settings, such as OpenMP version and initial values of environment variables.

`OMP_TARGET_OFFLOAD` Newer standard from version 5.0 defines the environment variable `OMP_TARGET_OFFLOAD` with values of `MANDATORY | DISABLED | DEFAULT`, where the value `DISABLED` forbids the launch of target regions on target device, `MANDATORY` terminates the program if target device is not available.

Non-standard but useful environment variables

`LIBOMPTARGET_INFO` for clang provides useful information about the runtime of target devices[4].

`GOMP_DEBUG` for gcc provides debugging information about offloading[5].

3.3.5 SPMD mode

The initial design had the whole single step of the MTSM as single kernel running on the GPU.Due to the data dependency between orders the compiler was unable to properly parallelize the kernel into "SPMD mode", or Single-Program Multi-Data mode, which is OpenMP's equivalent of SIMT. By default the OpenMP creates the kernels in "generic-mode", which is single thread kernel with a state machine that schedules parallel worker threads. Due to this overhead the generic-mode kernels are much slower than the SPMD mode kernels.

```
// Full step in target
time ./methodsOMP mtsm 0 2e-8 500 1e-10 ../inputs/telegraph100.h5 /dev/null

real    1m34.619s
user    0m1.156s
sys     0m0.050s
```

[4] https://openmp.llvm.org/design/Runtimes.html#libomptarget-info
[5] https://gcc.gnu.org/onlinedocs/gcc-16.1.0/libgomp/GOMP_005fDEBUG.html

By splitting the control of higher order computation onto cpu instead of in the kernel and keeping only the expensive computation on the GPU, the compiler was able to compile the split kernels into the SPMD mode. Which naturally resulted in much faster execution, as can be seen below.

```
// only parallel for kernels
time ./methodsOMP mtsm 0 2e-8 500 1e-10 ../inputs/telegraph100.h5 /dev/null
real    0m2.070s
user    0m2.003s
sys     0m0.056s

time ./methodsCpu mtsm 0 2e-8 500 1e-10 ../inputs/telegraph100.h5 /dev/null

real    0m0.461s
user    0m0.421s
sys     0m0.035s
```

Debugging

The debugging of OpenMP programs can be split into two parts – at compile time and at run time.

To receive debugging information during compile time or compilers reasoning for applied optimizations the following arguments can be used: For Clang based compilers use the remark diagnostics `-Rpass=openmp-opt`. For the GCC based compilers the option is called `-fopt-info`.

Apart from using debuggers to debug the programs at runtime, OpenMP provides some environment variables that can provide additional useful information regarding the running program.

Using the OpenMP runtime environment variable `LIBOMPTARGET_KERNEL_TRACE=TRUE` or `LIBOMPTARGET_INFO=-1` it is possible to obtain detailed information about the launched kernels.

Chapter 4

Implementation of ODE solver

As it goes with parallelization, we must find independent parts of the methods to utilize GPU's parallelization. Consider the computation of single-step $y_{n+1} = y_n + h\Phi(t_n, y_n, h)$. This equation has immediate data dependency between steps, which blocks the parallel computation of multiple steps, e.g., at time step t_n it would not be possible to compute y_{n+2} and y_{n+1} at the same time in parallel as y_{n+2} depends on result of y_{n+1}.

Even if the first directly dependant term was left out – to precompute results of $\Phi(t_n, y_n, h)$ and add them to y_n's in serial computation – the term $\Phi(t_n, y_n, h)$ can be dependant on y_n and in most interesting cases is. The functions $\Phi(t_n, y_n, h)$, that do not contain the result of previous step y_n, could only contain constants, t_n and step size h, which would mean these functions can be only linear function.

Therefore parallelization of step computation is not practical. Consider system of ODEs, which can be written as $\mathbf{y}_{n+1} = \mathbf{F}(t_n, \mathbf{y}_n)$, which can be rewritten per equation as

$$y_{n+1}(0) = F_n(0)(t_n, \mathbf{y}_n) \tag{4.1}$$
$$y_{n+1}(1) = F_n(1)(t_n, \mathbf{y}_n) \tag{4.2}$$
$$\vdots \tag{4.3}$$
$$y_{n+1}(i) = F_n(i)(t_n, \mathbf{y}_n) \tag{4.4}$$

In contrast to previous try to parallelize by time step, *parallelization between equations have no dependency in one time step*! This makes it possible for simple parallelization.

The initial idea behind implementing by hand is to have finer control over implementation and thus detect which parts take the longest and then reuse improved implementation for example `sgemv` instead. This implementation although allows for more precise analysis of execution time and performance of individual parts.

The initial idea with the implementation of compressed sparse row (CSR) format was to create abstraction and provide the underlying `sgemv_basic` or `spmv_basic` functions implemented for matrix-vector multiplication regardless of the actual format used (dense or sparse), but this was not done, due to the differences in dense and sparse matrices for GPU, such as the need of reallocation for certain operations. Although, this principle should be possible except for the precalculated MTSM.

The initial implementation was done on cpu with in the file `methodsCpu.cpp`, however this was abandoned due to the ability of OpenMP to run the code with single thread on CPU. The main code resides in the file `methodsOMP.cpp`, which provides help of runtime parameters with launched without any or with incorrect ones. Similar applies for `genTelegraph.cpp`, which is used to generate file for the Telegraph model as defined in

Section 5.2. The same also applies for `genDense.cpp`, which is used to convert the Jacobian matrix $\mathbf{A}$ and vector $\mathbf{y_0}$ from csv to the custom file format.

4.1 Problem definition format – HDF5 file

This section describes custom format based on the file format HDF5, for defining problems for the implementation. Due to the structure of the file format, the implementation of the methods `methodsOMP.cpp` is able to detect the matrix format from the file (based on the attribute of `Type`) for the dataset of matrix $\mathbf{A}$. The file format is expected to contain group `IVP` at the root of the file as illustrated in Figure 4.1, which contains all the necessary data for defining the linear IVP. Those necessary datasets are jacobian matrix `A` with the shape $N \times N$ numbers, constant right-hand side `b` vector of N numbers and vector of initial conditions `y_initial` of N numbers. Optionally it can contain dataset `Names` containing string name of each equation, such as $u_{C_1}, u_{C_2}, \ldots$ for the telegraph line experiment Section 5.2. All numbers in the dataset are 64-bit floating-point low endianness numbers, since the solver can be used with either 32-bit or 64-bit floating-point numbers. The exact datatype of matrix `A` depend on its attribute `Type`, which can be either string `Dense` or `CSR`. In the case of `Dense` the matrix is stored in the usual $2D$ array of $N \times N$ numbers. In the other case the Figure 4.2 describes the structure of matrix `A`. The `Configuration` attribute is optional one and provides mostly information for user about the parameters of the system of ODEs – such as value for capacitances and inductances and resistors for the telegraph line problem Section 5.2.

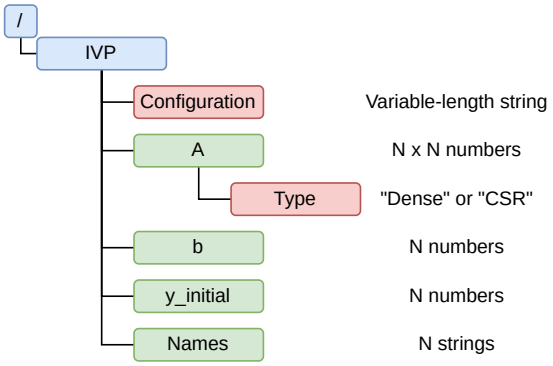


Figure 4.1: Expected structure of input files for problems to be solved by the implemented solver. Blue nodes are groups, green nodes are datasets and red node is an attribute.

The actual object `A` is in the case of CSR matrix is a group, rather than a dataset, since the CSR format expects three arrays, which are explained in Section 4.3. The structure of this group is illustrated in Figure 4.2. Since CSR is a format for sparse matrices, the number of nonzero elements (`numberOfNonzero`) is important parameter of this format. The type `hsize_t` is type defined by hierarchical data format (HDF5) and is alias to 64-bit unsigned integer (`uint64_t`). The array `rowIndices` contains `rows + 1` "pointers" for start and end of the compressed row. The array `columnIndices` has `numberOfNonzero` column indices. And the array `values` has `numberOfNonzero` value of the element in matrix.

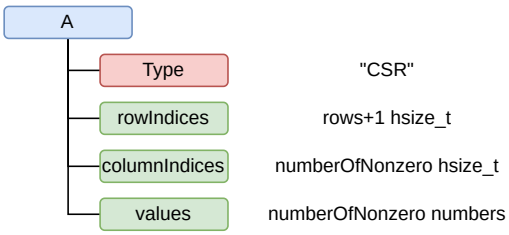


Figure 4.2: Structure of HDF5 object `A` in the case of `Type` being `CSR`.

4.2 Dense algorithm

This section provides explanation for the implementation of methods for solving the system of linear ODEs with *dense* Jacobian matrix **A**.

4.2.1 Dense matrix implementation

Important part of the implementation for dense matrix is, naturally, the matrix itself. In my implementation, the dense matrix is done by the class `Matrix<T>`, which stores the matrix as 1D vector in member `data` and the shape of the matrix as member `dims` with `dims[0]` being the number of columns and `dims[1]` being the number of rows.

4.2.2 Dense ODE solver implementation

The reason why Line 11 in Listing 4.1 is in steps and computes time with multiplication is due to the possible accumulated error of float, see Subsection 4.4.3.

The reason why the `odeSolverManual` function takes two additional template arguments of the functions is that the C++ has issues with taking template function as both function pointers and `std::functions`. The argument `methodFunction` is the computation of a single-step function (Euler, Runge-Kutta, MTSM). The argument `resultConsumer` is a function that consumes one results of single step, ie. printing to output or saving to file. Even though, the exact types of arguments are known for the `methodFunction`, `odeSolverManual` still has to take it as an template argument, due to the use of templates within the `methodFunction`.

The arguments of `odeSolverManual` are two vectors and jacobian matrix of the linear systems of ODEs computing for each step $\mathbf{y}_{n+1} = \mathbf{A} * \mathbf{y}_n + \mathbf{b}$, where the argument $\mathbf{y}$ is both input – as initial condition $\mathbf{y}_0$ – and output argument for space reuse. The method computes the solution for interval `[t_start, t_end]`, where `t_start` is t_0 and is the time of initial conditions $\mathbf{y}(t_0) = \mathbf{y}_0$. The `stepCount` is the number of steps the method takes to compute the solution and is used to calculate the step size h. The last argument of computation is the `maxOrder`, which is order of the MTSM specified by the user, it is unused for the other implemented methods.

```
1 template <std::floating_point FloatType, typename MethodFunction,
      typename ResultConsumer>
2 auto odeSolverManual( ... ) {
3     const FloatType step = (t_end - t_start) / stepCount;
4
```

```
5       // Allocate and prepare variables pointers for GPU
6       // ...
7
8       // Transfer and allocate data on the device
9   #pragma omp target enter data map(to: y_ptr[:size], A_ptr[:A_size],
        b_ptr[:size]) map(alloc: tmp[:size], dy[:size])
10
11      for (std::size_t step_index = 1; step_index <= stepCount;
            step_index++) {
12          // Calculate time at current step
13          const FloatType time = t_start + step_index * step;
14
15  #pragma omp task depend(out: y_ptr[:size])
16          {
17              // Call the single step computation of method on device
18              methodFunction( ... );
19          }
20
21          // Transfer the results from device (GPU) to host (CPU)
22  #pragma omp target update from(y_ptr[:size]) depend(in: y_ptr[:size])
        nowait
23
24          // Wait for the synchronization of compute and transfer before
                using the results.
25  #pragma omp taskwait
26          // Give the current results to consumer function for saving
                the results.
27          resultConsumer( step_index, time, y );
28      }
29
30  }
```

Listing 4.1: `odeSolver` is a function that solves the system of ODEs in the t_{start} to t_{end} interval using provided single-step method.

The `resultConsumer` allows for flexible and more efficient use of the computed results. The implementation uses the following consumer Listing 4.2, which either prints the result by time step on the standard output or saves to file.

```
auto resultPrint = [&]<std::floating_point FloatType>(std::size_t
    timestep, FloatType time, std::vector<FloatType>& y) {
    // time y[0] y[1] ... y[n]
    fmt::print(output, "{} ", time);
    for (const auto& yi : y) {
        fmt::print(output, "{} ", yi);
    }
fmt::println(output, "");
};
```

Listing 4.2: Lambda function for the consumption of single step result of the solver.

The Euler method is the simplest method of the three methods implemented. The three methods (Euler, Runge-Kutta with 4 stages and MTSM) use additional two vectors (`tmp` and `dy`) as temporary space for the computation of the method.

The arguments are current time of the computation `t`, step size `step`, input and output argument `y`, which as inputs is a $vecy_n$ and at the end of the step computation is $\mathtt{y}_{n+1}$, the Jacobian matrix `A` and constant right-hand side vector `b`, the number of equations `equationCount`, which determines the size of `y`, `b` and `A`. The argument `maxOrder` on Line 4 in Listing 4.3 is not used, as it is there only for function argument compatibility with the MTSM Listing 4.5.

The parallelization itself is done by the Line 10 letting the compiler decide the division of work and number of threads used. The assumption is that the work distribution is one equation per thread.

```
1  template <std::floating_point FloatType>
2  void EulerStepManual(
3          ...
4          const std::size_t maxOrder
5  ) noexcept
6  {
7      // tmp = A * y_n
8      sgemv_basic( A, y, tmp, equationCount, equationCount);
9
10 #pragma omp target teams distribute parallel for
11     for (auto i = 0; i < equationCount; i++) {
12         // dy = h * ( (A * y_n) + b)
13         dy[i] = step * (tmp[i] + b[i]);
14         // y_n+1 = y_n + h * f(t, y)
15         y[i] += dy[i];
16     }
17 }
```

Listing 4.3: Implementation of Euler single-step method for dense matrices.

Arguments for 4-th order Runge-Kutta methods are same as for Euler method Listing 4.3. However, the function has addition temporary vector of memory for intermediate computation. There was little to no measured overhead of having the allocation inside the function, since the allocation used previously allocated memory and the offloaded regions are after the allocation. There was another experimented implementation with the `k_1, k_2, k_3, k_4` being thread local variable with single for loop, however, this solution worked for smaller problems, but computed incorrectly for bigger problems.

```
1  template <std::floating_point FloatType>
2  void RungeKuttaManual( ... ) noexcept {
3      std::vector<FloatType>tmp2 (equationCount);
4      FloatType * tmp2_ptr = tmp2.data();
5  #pragma omp target enter data map(alloc: tmp2_ptr[:equationCount])
6
7      // tmp = k_1 &= f(t_n, y_n)
8      // k1 = A * y_n
9      sgemv_basic(A, y, tmp, equationCount, equationCount);
```

```
    #pragma omp target teams distribute parallel for
    for (auto i = 0; i < equationCount; i++) {
        // k1 = A * y_n + b
        tmp[i] += b[i];
        // y_(n+1) &= y_n + h / 6 (k_1 + ...)
        dy[i] = y[i] + tmp[i] * step / 6.0;

        tmp2_ptr[i] = y[i] + tmp[i] * step / 2.0;
    }

    // k_2 &= f(t_n + h / 2, y_n + h k_1 / 2)
    sgemv_basic(A, tmp2_ptr, tmp, equationCount, equationCount);
    #pragma omp target teams distribute parallel for
    for (auto i = 0; i < equationCount; i++) {
        // k1 = A * y_n + b
        tmp[i] += b[i];
        // y_(n+1) &= ... + 2 * k2 * h / 6;
        dy[i] += tmp[i] * 2 * step / 6.0;
        tmp2_ptr[i] = y[i] + tmp[i] * step / 2.0;
    }

        // k_3 &= f(t_n + h / 2, y_n + h k_2 / 2)
    sgemv_basic(A, tmp2_ptr, tmp, equationCount, equationCount);
    #pragma omp target teams distribute parallel for
    for (auto i = 0; i < equationCount; i++) {
        tmp[i] += b[i];
        // y_(n+1) &= ... + 2 * k3 * h / 6;
        dy[i] += tmp[i] * 2 * step / 6.0;
        tmp2_ptr[i] = y[i] + tmp[i] * step;
    }

        // k_4 &= f(t_n + h, y_n + h k_3)
    sgemv_basic(A, tmp2_ptr, tmp, equationCount, equationCount);
    #pragma omp target teams distribute parallel for
    for (auto i = 0; i < equationCount; i++) {
        tmp[i] += b[i];
        dy[i] += tmp[i] * step / 6.0;
        // y_(n+1) &= y_n + h / 6 (k_1 + 2 k_2 + 2 k_3 + k_4)
        y[i] = dy[i];
    }

#pragma omp target exit data map(delete: tmp2_ptr[:equationCount])
}
```

Listing 4.4: Implementation of Runge-Kutta four-stages single-step method for dense matrices.

```
template <std::floating_point FloatType>
void MTSMStepManual( ... ) noexcept {
```

```
    const std::size_t M = equationCount;
    const std::size_t N = equationCount;

    if ( maxOrder > 0 ) {
        // y' = A * y_n + b
        // tmp = A * y_n
        sgemv_basic(A, y, tmp, M, N);

#pragma omp target teams distribute parallel for
        for (auto k = 0; k < equationCount; k++) {
            // p(1) = h * y'
            dy[k] = step * (tmp[k] + b[k]);
            // y_n+1 = y_n + p(1) + ...
            y[k] += dy[k];
        }
    }

    for (auto order = 2; order <= maxOrder; order++) {
        std::swap(tmp, dy);

        // A * p(j-1)
        sgemv_basic(A, tmp, dy, M, N);
#pragma omp target teams distribute parallel for
        for (auto k = 0; k < equationCount; k++) {
            // p(j) = h / j * A * p(j-1)
            dy[k] *= step / order;
            // y_n+1 = ... + p(j) + ...
            y[k] += dy[k];
        }
    }
}
```

Listing 4.5: Implementation of MTSM single-step method for dense matrices.

The function `sgemv_basic` Listing 4.6 is a basic implementation of `sgemv` (Single-precision General Matrix-Vector multiplication) function, that performs matrix-vector multiply operation $\mathbf{y} = \alpha * \mathbf{A} * \mathbf{x} + \beta * \mathbf{y}$. However, the `sgemv_basic` is simplified into $\mathbf{y} = \mathbf{A} * \mathbf{x}$ only, which is equivalent to `sgemv` with parameters $\alpha = 1, \beta = 0$. The simplification does not hurt performance, consider the following:

$$
\begin{aligned}
&\text{SGEMV:} && \mathbf{y} = \alpha * \mathbf{A} * \mathbf{x} + \beta * \mathbf{y} \\
&\text{Euler:} && \mathbf{y}_{n+1} = \mathbf{y}_n + h * (\mathbf{A} * \mathbf{y}_n + \mathbf{b}) = \mathbf{y}_n + h * \mathbf{A} * \mathbf{y}_n + h\mathbf{b} \\
& && \quad = h * \mathbf{A} * \mathbf{y}_n + (\mathbf{y}_n + h\mathbf{b}) \\
&\rightarrow && \alpha = h, \mathbf{A} = \mathbf{A}, \mathbf{x} = \mathbf{y}_n, \beta = 1, \mathbf{y} = \mathbf{y}_n + h\mathbf{b}
\end{aligned}
$$

Based on the arguments of `EulerStepManual`, it would be still necessary to have loop before `sgemv` to compute $\mathbf{y}_n + \mathbf{b}h$. Additionally, because $\mathbf{x}$ and $\mathbf{y}$ are the same vectors or arrays, it would need temporary array to store the results and then move the data from the temporary array into $\mathbf{y}$.

The `sgemv_basic` has two important properties. First one is simple performance optimization of only single write to memory Line 12 of thread local variable Line 6. The other one is marking of tracing region using `tracePush` and `tracePop` for more detailed profiling.

```
1  template <std::floating_point FloatType>
2  void sgemv_basic(const FloatType * A, const FloatType * x, FloatType
       * y, std::size_t M, std::size_t N) {
3      #pragma omp target teams distribute parallel for
4      for (auto i = 0; i < M; i++) {
5          // Initialize output
6          FloatType sum = 0;
7          // Perform the computation
8          for (auto j = 0; j < N; j++) {
9              sum += A[i * N + j] * x[j];
10         }
11         // Write the result to memory
12         y[i] = sum;
13     }
14 }
```

Listing 4.6: Custom simplified implementation of `sgemv` to only matrix-vector multiplication with tracing region for better profilling support. Computes $\mathbf{y} = \mathbf{A} * \mathbf{x}$, where $\mathbf{A}$ has size $M \times N$, $\mathbf{x}$ has size N and $\mathbf{y}$ has size M.

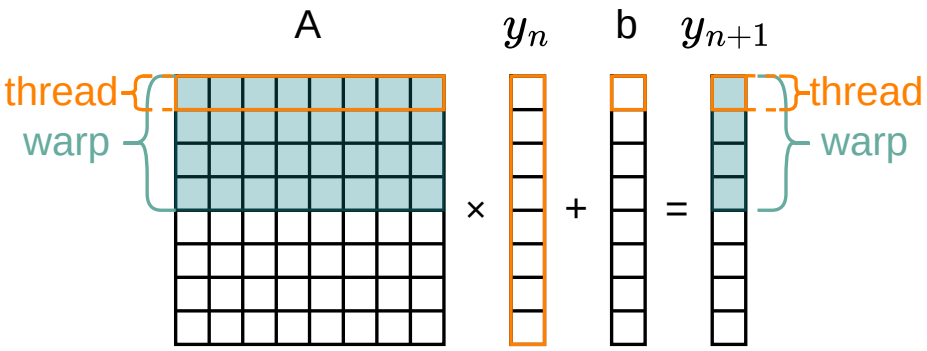


Figure 4.3: Visualization of parallelization for GPU of dense matrix algorithm.

4.2.3 Spatial complexity

For the dense matrix the most dominant element in space domain is the matrix $\mathbf{A}$ with space complexity $\mathcal{O}(n^2)$. The space complexity of current solution $\mathcal{O}(n)$, due to the use of `resultConsumer` function, which allows to hold only single step in memory. If we consider the test system, which has GPU memory of 16 GB, for 32-bit floating-point it would be able to hold at most $n = \sqrt{16\,\mathrm{GB}/4\,\mathrm{B}} \approx 64\,\mathrm{k} = 64 \times 10^3$ equations. If we take high-performance computer (HPC) systems, for example Czech supercomputer Karolina

from IT4I has NVIDIA A100[1], that has 80 GB[2], which would be able to hold matrix with $n \approx 141 \times 10^3$ equations, which is *only twice as much as on a test system*. For problems that are an order of magnitude bigger, it would be necessary to use multi-gpu systems, such as the mentioned HPC.

However, for problems that have sparse matrix $\mathbf{A}$, which is usual – e.g., the telegraph line model Section 5.2 – the dense matrix format would be inefficient for both storage (in memory) and computation. Therefore sparse matrix format is used for such matrices and it is described in detail in Section 4.3.

4.2.4 Precalculation

This subsection follows up on the theoretical introduction to the precalculated version of MTSM in (2.30). Due to the difference in computation of precalculated version of MTSM, this method has its own function `odeSolverManualPrecalc`. The difference between this function and `odeSolverManual` is that, the precalculation version pre-computes the matrix $\mathbf{A}_y = \sum_{k=0}^{n} \frac{h^k}{k!}\mathbf{A}^k$ and vector $\mathbf{b}_b = \left(\sum_{k=0}^{n-1} \frac{h^{k+1}}{(k+1)!}\mathbf{A}^k\right)\mathbf{b}$ and finally the step loop is only $\mathbf{y}_{i+1} = \mathbf{A}_y\mathbf{y}_i + \mathbf{b}_b$.

4.3 Sparse matrix algorithm

As it is mentioned at the end of Section 4.2 many problems that are systems of linear ODEs have the Jacobian matrix $\mathbf{A}$ *sparse*. Using the previously mentioned algorithm Section 4.2 performed on such matrices wastes both time and space, as they contain many zero elements. For this need there are many already existing sparse matrix formats, such as the chosen format compressed sparse row (CSR). The CSR format was chosen because it allows for fast matrix-vector multiplications ($\mathbf{y} = \mathbf{A}\mathbf{x}$) [4].

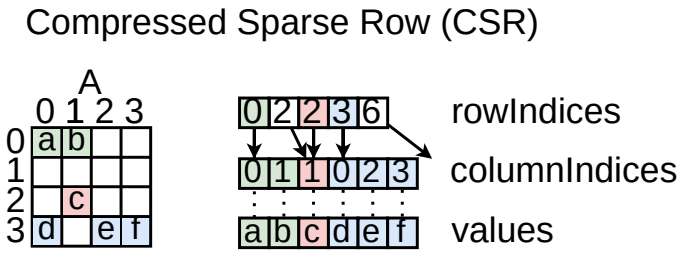


Figure 4.4: Illustration of compressed sparse row (CSR) format.

The compressed sparse row (CSR) or compressed row storage (CRS) or Yale format stores the matrix $\mathbf{A}$ with size $M \times N$ in three (one-dimensional) arrays as illustrated in Figure 4.4: `rowIndices`, `columnIndices` and `values` in the case of HDF5 file format as described in Figure 4.2 or `row_index`, `column_index` and `values` in the case of `MatrixCSR.hpp` implementation. Since the format is for sparse matrix it uses another important parameter – *number of nonzero elements* NNZ. The parameter also defines when

[1] https://www.it4i.cz/infrastruktura/karolina
[2] https://www.nvidia.com/en-us/data-center/a100/

it saves space:

$$NNZ < (M(N-1)-1)/2 \tag{4.5}$$

The array `values` contains all the nonzero elements of matrix $\mathbf{A}$ and thus have the length NNZ. The array `columnIndices` maps each values from `values` to column index of given value in the matrix $\mathbf{A}$, and therefore also has length of NNZ. On the contrary, the array `rowIndices` specifies, which values from the array `values` are part of the specified row and has length of $M + 1$. The range for row with index `row` is the interval `[rowIndices[row], rowIndices[row + 1])`. This provides some interesting properties for the array `rowIndices` – the first element of the `rowIndices` is always zero (`rowIndices[0] = 0`) and the last element is always NNZ (`rowIndices[M] = NNZ`).

This structure gives the format its name, since all values are compressed by rows.

Some advanced algorithms, such as CSR matrix-matrix multiplication and transposition were used from the article [3]. However, these algorithms are for serial programs and do not work in parallel environment. Some of the algorithms change the NNZ, for example those are matrix-matrix multiplication and addition of matrices. This is problematic specifically for GPU parallelization, because the allocation inside kernel can have serious impact on the performance, for example see[3].

Nevertheless, the CSR matrix and dense vector multiplication is trivially parallelizable by splitting the work by "rows", similarly to parallelization of Section 4.2. Since the format is compressed by rows, accessing each rows values and column indices is independent from accessing the other rows.

4.3.1 Implementation

The implementation is mostly done using the `MatrixCSR<T>` class in the file `MatrixCSR.hpp` except the matrix-vector multiplication, which is implemented as `spmv_basic` function. The function `spmv_basic` follows the same methodology as `sgemv_basic` in Section 4.2, which means the function computes $\mathbf{y} = \mathbf{Ax}$.

This design choice was to unify the interface between dense and sparse matrix, since only difference between those algorithms is the structure of matrix. which would enable reusing the same implementation of single-step methods, with only difference in the matrix-vector multiplication, which would be done by providing correct function for given time as function argument. Altough, the implementation of `odeSolver` might differ between the formats, as they need to allocate different memory structures.

```
template <std::floating_point FloatType>
void spmv_basic(const MatrixCSR<FloatType>& A, const FloatType * x,
    FloatType * y, std::size_t M, std::size_t N) {
    // Get the pointers of the CSR matrix struct
    std::size_t * row_index = A.row_index;
    std::size_t * column_index = A.column_index;
    FloatType * values = A.values;

#pragma omp target teams distribute parallel for
    for (auto i = 0; i < M; i++) {
        // Initialize output
        FloatType sum = 0;
```

[3] https://stackoverflow.com/q/9806299

```
        // Get the indices for single row
        for (auto j = row_index[i]; j < row_index[i+1]; j++) {
            // This is the main difference compared to sgemv_basic
            sum += values[j] * x[column_index[j]];
        }
        // Save the result
        y[i] = sum;
    }
}
```

Listing 4.7: The implementation of dense matrix-vector multiplication `spmv_basic`. Arguments are similer to Listing 4.6, except the `MatrixCSR` class.

4.4 Issues with implementation

This section provides description of encountered problems that shaped the implementation into its current form.

4.4.1 Asynchronous transfer to overlap transfer and computation

One of the earliest optimizations were to try overlap computation and memory transfer of a single step. The tried implementation was that the `methodFunction` would be in OpenMP `task` with pragma `depend(out: ...)` and the transfer would be task with `depend(in: ...)`.

```
#pragma omp task depend(out: y_ptr)
{
    methodFunction( ... );
}

#pragma omp task depend(in: y_ptr)
#pragma omp target update from(y_ptr[:size]) nowait
```

Listing 4.8: Failed attemp to overlap the computation of next step and result transfer.

However, this approach did not work with the following error Listing 4.9.

```
code/methodsOMP.cpp:430:28: error:
  '#pragma omp target update' cannot be an immediate substatement
  430 |         #pragma omp target update from(y_ptr[:size]) depend(in:
    y_ptr[:size]) nowait
```

Listing 4.9: Compiler output for the try of overlapping computation and memory transfer from code snippet Listing 4.8.

I was not able to find workable solution to this problem.

4.4.2 OpenMP and Objective-oriented programming

OpenMP did not allow for the use of class attribute as outputted by compiler in Listing 4.10

```
type 'const MatrixCSR<float>' is not trivially copyable and not guaranteed
   to be
      mapped correctly [-Wopenmp-mapping]
  600 |             sum += values[j] * x[A.column_index[j]];
      |                                  ^
```

Listing 4.10: Output of OpenMP compiler when compiling code that uses pointer to class attributes.

The workaround for this problem is to use direct pointers to the attributes instead accessing the pointer through structure.

4.4.3 Floating point arithmetic

Following the round-off error, the following error with computing the time by summing the step size was encountered, where for enough number of steps the end time was different from the expected value. Therefore the other variant with multiplying the step size was chosen.

```
#include <print>
#include <numbers>
#include <limits>

using T = double;

int main() {
    T start = 0.0;
    T end = 4 * 2 * std::numbers::pi_v<T>;
    std::size_t steps = 5000;
    T step = (end - start) / steps;

    T calc_end = start + steps * step;

    std::println("original end\t= {}", end);
    std::println("calculated end\t= {}", calc_end);

    T sum = start;
    for (int i = 0; i < steps; i++) {
        sum += step;
    }

    std::println("accumulated end\t= {}", sum);
    std::println("");

    std::println("rel err calc \t = {}", (calc_end - end) /
        std::max(calc_end, end));
    std::println("rel err acc \t = {}", (sum - end) / std::max(sum,
        end));
}
```

Listing 4.11: Example program of issues with floating-point arithmetic

The following Listing 4.12 is the output for previous code sample Listing 4.11 for `T = float`.

```
original end    = 25.132742
calculated end  = 25.13274
accumulated end = 25.131777

rel err calc     = -7.589099e-08
rel err acc      = -3.840084e-05
```

Listing 4.12: Output for code sample Listing 4.11 for `T = float`.

```
original end    = 25.132741228718345
calculated end  = 25.132741228718345
accumulated end = 25.132741228719556

rel err calc     = 0
rel err acc      = 4.820307317240031e-14
```

Listing 4.13: Output for code sample Listing 4.11 for `T = double`.

4.4.4 C++ function pointers and templates

C++ provides nice type for storing function pointers and using them later, the `std::function`. However, this type has problems when used with templated functions and templated lambdas, similarly to what is described in[4].

The use of templated function with `std::function` resulted in unreadable template error, which I was unable to solve.

The workaround for this issue was to use another template parameter as mentioned with the `odeSolverManual`, which provides some type safety, but doesn't ensure the exact specified format of the single-step function.

4.4.5 OpenMP overlapping allocation

Missing array range (`map(delete: array)` instead of `map(delete: array[:size])`) in deletion of the sparse matrix arrays, caused invalid deletion in the pointer mapping table – keeping the mapping alive for de-allocated host pointer. The next allocation of the same memory address at the host caused OpenMP to crash the program with the following issue due to different device allocation with different sizes mapped to same host pointer of different sizes.

```
omptarget message: explicit extension not allowed: host address specified
    is 0x000000003dd48320 (56 bytes), but device allocation maps to host at
     0x000000003dd48320 (48 bytes)
```

[4] https://ikriv.com/blog/?p=4865

4.4.6 `std::valarray` vs `std::vector`

The initial implementation used `std::valarray` for the vectors and matrix. The class `std::valarray` supports element-wise mathematical operations. This works wonders for serial C++ implementation, however it doesn't have API for accessing the array as continuous block of memory therefore the transfer to gpu would require deep-copy. On the other hand `std::vector` is stored and accessed as an continuous memory block simplifying the memory transfer.

4.4.7 Runge-Kutta implementation

During the implementation of Runge-Kutta 4-stage method, the implementation in Listing 4.14 was faster than the final implementation. This implementation only used single for loop with storing the ks in thread local variables. However, this faster implementation has issue with stability for the benchmarking problems, but worked correctly for the trivial experiments for checking the correctness of a method.

I was unable to find the reason for the unstable computation.

```
template <std::floating_point FloatType>
void RungeKuttaManualFaster(...) noexcept
{
    #pragma omp target teams distribute parallel for
    for (auto i = 0; i < equationCount; i++) {
        // k_1 &= f(t_n, y_n)
        FloatType k1 = 0;
        // k1 = A * y_n
        for (auto j = 0; j < equationCount; j++) {
            k1 += A[i * equationCount + j] * y[j];
        }
        // k1 = A * y_n + b
        k1 += b[i];

        // k_2 &= f(t_n + h / 2, y_n + h k_1 / 2)
        FloatType k2 = 0;
        for (auto j = 0; j < equationCount; j++) {
            k2 += A[i * equationCount + j] * (y[j] + step * k1 / 2);
        }
        k2 += b[i];

        // k_3 &= f(t_n + h / 2, y_n + h k_2 / 2)
        FloatType k3 = 0;
        for (auto j = 0; j < equationCount; j++) {
            k3 += A[i * equationCount + j] * (y[j] + step * k2 / 2);
        }
        k3 += b[i];

        // k_4 &= f(t_n + h, y_n + h k_3)
        FloatType k4 = 0;
        for (auto j = 0; j < equationCount; j++) {
```

```
            k4 += A[i * equationCount + j] * (y[j] + step * k3);
        }
        k4 += b[i];

        // y_(n+1) &= y_n + h / 6 (k_1 + 2 k_2 + 2 k_3 + k_4)
        y[i] = y[i] + step / 6.0 * (k1 + 2 * k2 + 2 * k3 + k4);
    }
}
```

Listing 4.14: Faster implementation of Runge-Kutta using single for loop.

4.4.8 Double comparison

Because the implementation supports the use of double for computation with recompilation, during the development this was tested with yielding the following results of being about 25% slower using the double implementation. This can be tested by setting the `FloatType` in `methodsOMP.cpp` to double.

```
> time OMP_TARGET_OFFLOAD=MANDATORY ./methodsOMP mtsm 0 2e-8 500 100kcsr.h5
    tele100k-csr-double-mtsm-openmp-h500.dat

real    0m44.340s
user    0m43.540s
sys     0m0.315s

> time OMP_TARGET_OFFLOAD=MANDATORY ./methodsOMP mtsm 0 2e-8 500 100kcsr.h5
    tele100k-csr-mtsm-openmp-h500.dat

real    0m35.568s
user    0m35.252s
sys     0m0.252s
```

Chapter 5

Experiments

This chapter provides definitions of experiments tested on the implemented solution. The first section describes experiments used for checking the correctness of method computation on the GPU. The section after defines equations for the Telegraph line model. The last section defines the incompressible Stokes problem. The last two experiments are used for benchmarking, because they provide a way to generate problems with arbitrary number of equations.

All the benchmarks use the MTSM only with order of 4, for closer comparison to the Runge-Kutta 4th order method. The measured data are wall time of the methods is seconds, number of steps, that were needed to reach non-divergent solution, errors of parallelization, difference of solutions between each pair of methods and gpu specific measurements. The benchmarks were measured on my personal computer with the CPU AMD Ryzen 7 9700X 8-Core with total of 16 threads[1] and the GPU AMD Radeon RX 9060 XT with 16GB[2].

Between the GPU specific measurements are the occupancy as is described in Section 3.1 and the settings for parallelization of OpenMP league, i.e. the number of independent teams in the league and number of threads in each team.

The error of parallelization is the comparison of singlecore solution (CPU with 1 thread) to the parallelized GPU solution, with absolute error being $e_{abs} = |y_1 - y_{gpu}|$ and relative error being $e_{rel} = |y_1 - y_{gpu}|/|y_1|$.

5.1 First experiments

This sections describes first two tested experiments – exponential function and system of sin and cos. These experiments were conducted and tested to check the correctness of implemented methods on the GPU, since these equations are simple with known analytical solutions. These experiments are not used for the analysis of methods, because the GPU's resource would not be used efficiently for only one and two equations.

The first considered experiment is done to check correct behavior of implemented methods, because it has known analytical solution. Since the IVP has only single ODE the solver does not scale, which means benchmarking this IVP on the GPU would not be able to use the GPU's resources.

$$u' = \lambda u, \quad u(0) = 1 \tag{5.1}$$

[1] https://www.amd.com/en/products/processors/desktops/ryzen/9000-series/amd-ryzen-7-9700x.html

[2] https://www.amd.com/en/products/graphics/desktops/radeon/9000-series/amd-radeon-rx-9060xt.html

with analytical solution $u(t) = e^{\lambda t}$.

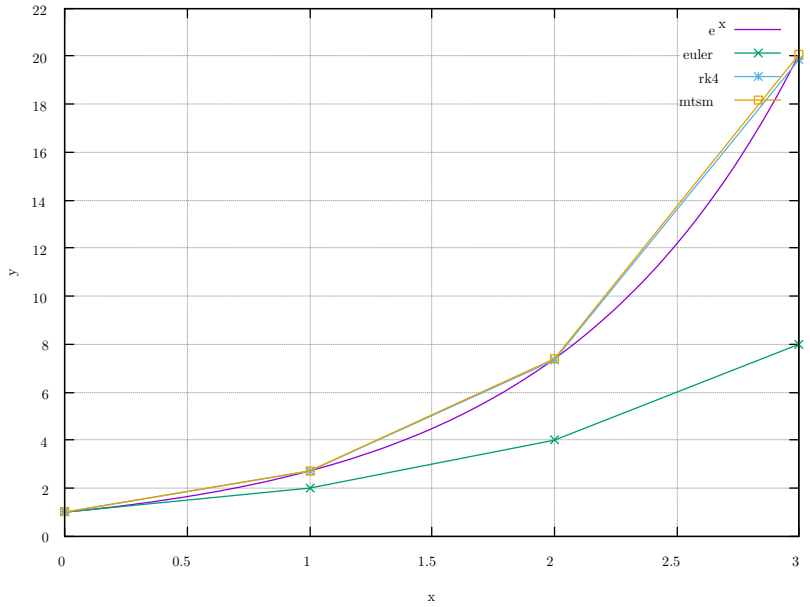
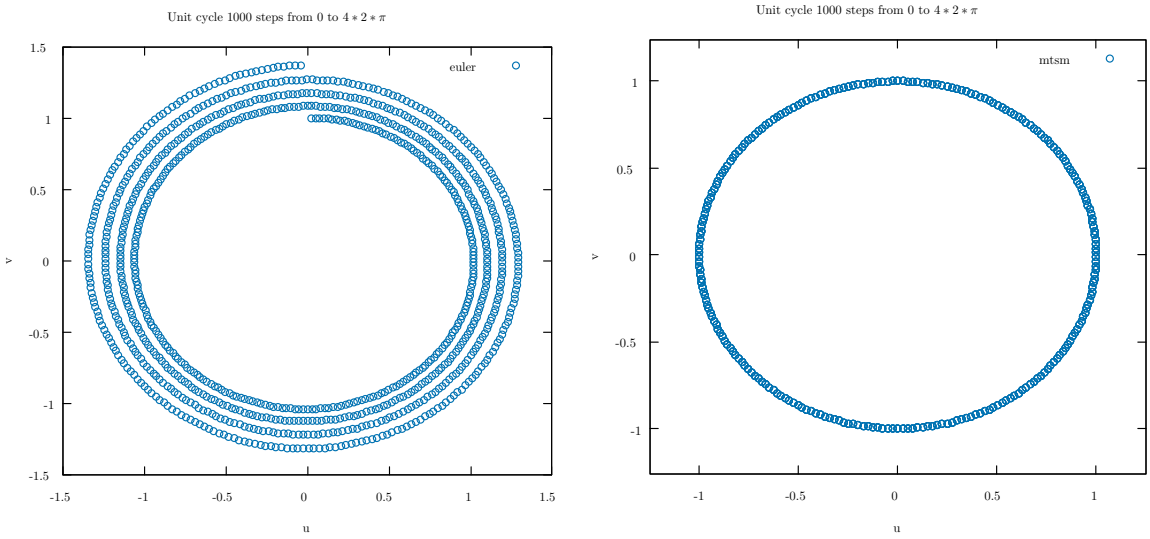


Figure 5.1: Comparison of solutions of different methods and analytical solution. Computed the above IVP for exponential function with $\lambda = 1$ on the interval $[0, 3]$ with step size $h = 1$. As we can see Euler method for this configuration diverges very quickly from the analytical solution, whereas Runge-Kutta 4 and MTSM stays close to the analytical solution.

Similarly to previous experiment, this experiment also serves as a check for the correctness of implementation. The analytical solution is also known and the IVP has only two linear equations. A more detailed experiments can be found in the Ph.D. thesis [14, p. 72, Experiments 1-5].

$$\begin{aligned} u' &= \omega v, \qquad u(0) = 0 \\ v' &= -\omega u, \quad v(0) = 1 \end{aligned} \tag{5.2}$$

where analytical solution is $u(t) = \sin(\omega t)$ and $v(t) = \cos(\omega t)$.



(a) Solution of the Euler method for the sin and cos ODE. The Euler method solution is slowly diverging over time from the analytical solution of unit circle.

(b) Solution of the MTSM for the sin and cos ODE. The solution of MTSM is stable over time and stays close to the the analytical solution of unit circle.

Figure 5.2: Comparison of Euler method and MTSM solving the ODE for sin and cos function. The $\omega = 1$ and the interval is $[0, 4 * 2 * \pi]$ with 1000 steps each.

5.2 Telegraph equation

This section explains the Telegraph equation model for scalable experimentation. The telegraph equation model is introduced in detail the article [17].

This problem was chosen because it allows to specify arbitrary number of ODEs, allowing the examination of how the MTSM behaves on GPU for bigger problems.

5.2.1 Telegraph equation model

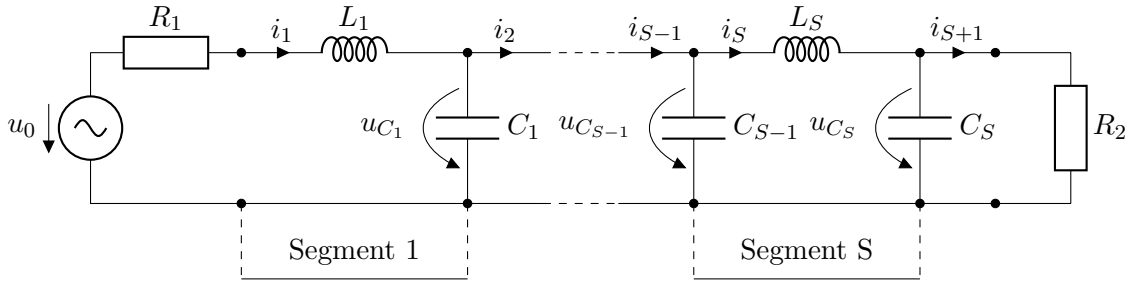


Figure 5.3: Model of telegraph line of S segments in series as introduced in the article [17].

The change of voltage and current can be expressed by the following ODEs

$$u'_{C_j} = \frac{i_{C_j}}{C_j}$$
$$i'_{L_j} = \frac{u_{L_j}}{L_j}$$

Applying Kirchoff's current law

$$\sum_{k=1}^{n} I_k = 0 \tag{5.3}$$

for the segment 1 we obtain:

$$\begin{aligned} i_1 - i_2 - i_{C_1} &= 0 \\ i_1 &= i_2 + i_{C_1} \end{aligned}$$

Applying Kirchoff's voltage law

$$\sum_{k=1}^{n} U_i = 0 \tag{5.4}$$

for the segment 1 we receive:

$$\begin{aligned} u_0 - u_{R_1} - u_{L_1} - u_{C_1} &= 0 \\ u_0 &= u_{R_1} + u_{L_1} + u_{C_1} \end{aligned}$$

Applying Ohm's law

$$u = i \cdot R \tag{5.5}$$

for R_1 and R_2

$$\begin{aligned} u_{R_1} &= i_1 \cdot R_1 \\ u_{R_2} &= i_{S+1} \cdot R_2 \end{aligned}$$

Note that

$$\begin{aligned} i_{L_1} &= i_1 \\ i_{L_2} &= i_2 \\ &\vdots \\ i_{L_S} &= i_S \end{aligned}$$

For the first segment we have the following equations

$$\begin{aligned} u'_{C_1} &= \frac{i_{C_1}}{C_1} \overset{(5.3)}{=} \frac{i_1 - i_2}{C_1} \\ &= \frac{1}{C_1}(i_1 - i_2) \\ i'_1 &= \frac{u_{L_1}}{L_1} \overset{(5.4)}{=} \frac{u_0 - u_{C_1} - u_{R_1}}{L_1} \overset{(5.5)}{=} \frac{u_0 - u_{C_1} - R_1 \cdot i_1}{L_1} \\ &= \frac{1}{L_1}(u_0 - u_{C_1} - R_1 \cdot i_1) \end{aligned} \tag{5.6}$$

For the next segments $k \in \langle 2, S \rangle$ we have:

$$\begin{aligned} u'_{C_k} &= \frac{i_{C_k}}{C_k} \overset{(5.3)}{=} \frac{i_k - i_{k+1}}{C_k} \\ &= \frac{1}{C_k}(i_k - i_{k+1}) \\ i'_k &= \frac{u_{L_k}}{L_k} \overset{(5.4)}{=} \frac{u_{C_{k-1}} - u_{C_k}}{L_k} \\ &= \frac{1}{L_k}(u_{C_{k-1}} - u_{C_k}) \end{aligned} \tag{5.7}$$

And for the last segment we have:

$$i_{S+1} = \frac{1}{R_2} u_{C_S} \tag{5.8}$$

$$\mathbf{A} = \left(\begin{array}{c|c} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \hline \mathbf{A}_{21} & \mathbf{A}_{22} \end{array}\right), \quad \mathbf{y} = \left(\begin{array}{c} u_{C_1} \\ \vdots \\ u_{C_s} \\ \hline i_1 \\ \vdots \\ i_S \end{array}\right), \quad \mathbf{b} = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline \frac{u_0}{L_1} \\ 0 \\ \vdots \\ 0 \end{array}\right), \quad \mathbf{y}_0 = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 0 \\ \vdots \\ 0 \end{array}\right), \tag{5.9}$$

where $\mathbf{A}_{11}$, $\mathbf{A}_{12}$, $\mathbf{A}_{21}$ and $\mathbf{A}_{22}$ are individual block matrices with size $S \times S$

$$\mathbf{A}_{11} = \begin{pmatrix} 0 & \dots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \vdots & \dots & 0 & 0 \\ 0 & \dots & 0 & \frac{-1}{R_2 C_S} \end{pmatrix}, \qquad \mathbf{A}_{12} = \begin{pmatrix} \frac{1}{C_1} & \frac{-1}{C_1} & 0 & \dots & \dots & 0 \\ 0 & \frac{1}{C_2} & \frac{-1}{C_2} & 0 & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & \frac{1}{C_S} \end{pmatrix},$$

$$\mathbf{A}_{21} = \begin{pmatrix} \frac{-1}{L_1} & 0 & \dots & \dots & \dots & 0 \\ \frac{1}{L_2} & \frac{-1}{L_2} & 0 & \dots & \dots & \vdots \\ 0 & \frac{1}{L_3} & \frac{-1}{L_3} & 0 & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & \frac{1}{L_S} & \frac{-1}{L_S} \end{pmatrix}, \qquad \mathbf{A}_{22} = \begin{pmatrix} \frac{-R_1}{L_1} & 0 & \dots & 0 \\ 0 & 0 & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{pmatrix}.$$

Then the model can be written as linear system of ODEs using the matrix-vector notation:

$$\mathbf{y}' = \mathbf{A}\mathbf{y} + \mathbf{b}, \quad \mathbf{y}(0) = \mathbf{y}_0 \tag{5.10}$$

The IVP defined in (5.9) defines the input voltage u_0 as a constant, modelling the circuit with direct current.

5.2.2 Model input

In order to create model for alternating circuit, a change from constant input voltage u_0 to a harmonic signal is needed, that is $u_0 = U_0 \sin(\omega t)$. This harmonic signal can be computed using the system of coupled linear ODEs from the (5.2) with alteration to the initial condition of $\cos(\omega t)$ to the U_0 value.

$$\begin{aligned} u_0' &= \omega x, & u_0(0) &= 0 \\ x' &= -\omega u_0, & x(0) &= U_0, \end{aligned} \tag{5.11}$$

where the analytical solution is $u_0(t) = U_0 \sin(\omega t)$ and $x(t) = U_0 \cos(\omega t)$.

Since the $u_0(t)$ is now a function described by differential equation the matrices and vectors for IVP are changed accordingly:

$$\mathbf{A} = \left(\begin{array}{c|c|cc} A_{11} & A_{12} & 0 & 0 \\ \hline & & \frac{1}{L_1} & 0 \\ A_{21} & A_{22} & 0 & \vdots \\ & & \vdots & 0 \\ \hline 0 & \dots & 0 & \omega \\ 0 & \dots & -\omega & 0 \end{array}\right), \quad \mathbf{y} = \left(\begin{array}{c} u_{C_1} \\ \vdots \\ u_{C_s} \\ \hline i_1 \\ \vdots \\ i_S \\ \hline u_0 \\ x \end{array}\right), \quad \mathbf{b} = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 0 \\ \vdots \\ 0 \\ \hline 0 \\ 0 \end{array}\right), \quad \mathbf{y}_0 = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 0 \\ \vdots \\ 0 \\ \hline 0 \\ U_0 \end{array}\right) \tag{5.12}$$

5.2.3 Simulation

The article also defines a constant for propagation per unit length of one segment for the model as

$$t_{LC} = \sqrt{LC}. \tag{5.13}$$

That is the time it takes for the signal to propagate through one segment as model on Figure 5.3. For example the signal from u_0 would take t_{LC} to propagate to u_{C_1}. Then the total delay of input signal as

$$t_{delay} = S\ t_{LC}, \tag{5.14}$$

which is propagation from the input voltage u_0 to the resistor R_2 at the end of the telegraph line.

For simulation experiments the configuration is same as introduced in article [17, p. 12]. That is the capacitances and inductances are the same, $C_1 = C_2 = \ldots = C_S = 1\,\text{pF}$ and $L_1 = L_2 = \ldots = L_S = 10\,\text{nH}$, which models *homogeneous lossless* telegraph line.

If the condition

$$R_1 = R_2 = \sqrt{L/C} \tag{5.15}$$

is fulfilled, then the line is *lossless and signal on the line is transmitted without change.* Using the previous values for capacitances and inductances the resulting resistor values are $R_1 = R_2 = \sqrt{L/C} = \sqrt{10\,\text{nH}/1\,\text{pF}} = \sqrt{10 \times 10^{-9}\,\text{H}/1 \times 10^{-12}\,\text{F}} = 100\,\Omega$.

The configuration for the following graphs and measurements is the following:

- $C_1 = C_2 = \ldots = C_s = 1\,\text{pF}$,
- $L_1 = L_2 = \ldots = L_S = 10\,\text{nH}$,
- $R_1 = R_2 = 100\,\Omega$,
- $U_0 = 1\,\text{V}$,
- AC circuit $\omega = 3 \times 10^9\,\text{rad}\,\text{s}^{-1}$,
- Propagation time per unit length of one segment: $1 \times 10^{-10}\,\text{s}$,
- Time delay: $S \cdot 1 \times 10^{-8}\,\text{s}$,
- Start $= 0\,\text{s}$,
- End $= 2 \times 10^{-8}\,\text{s}$.

It is important to note that the computation interval stays the same for all sizes of the problem, even though, the delays changes. Reason for this is that the computation time would grow exponentially. Keeping the interval same provides better comparison regarding scaling of parallelization and the number of equations.

To check whether the method has output as expected, the following configuration was used to compare with outputs from the article [17, p. 8, Figure 4a) and 5a)].

- $S = 100$
- Step count $= 200$,
- Step size $h = 1 \times 10^{-10}$.

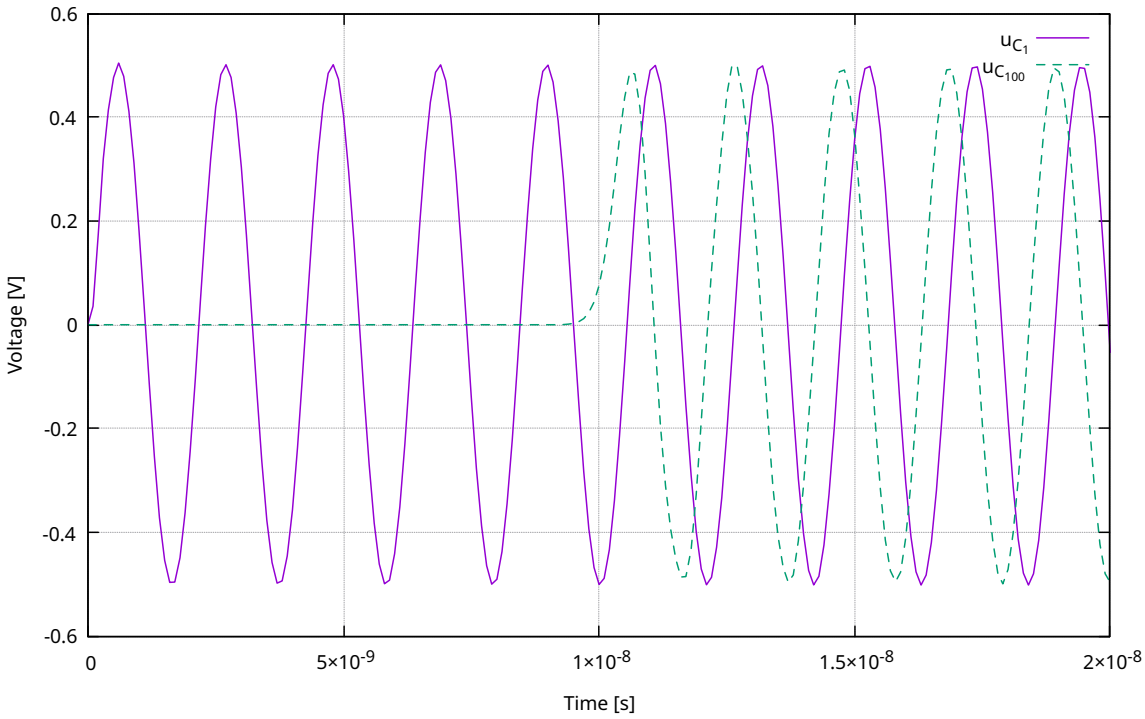


Figure 5.4: Simulation of telegraph line using harmonic input with adjusted line: $R_1 = R_2 = 100\,\Omega$ as specified in article [17, p. 8, Figure 4a)].

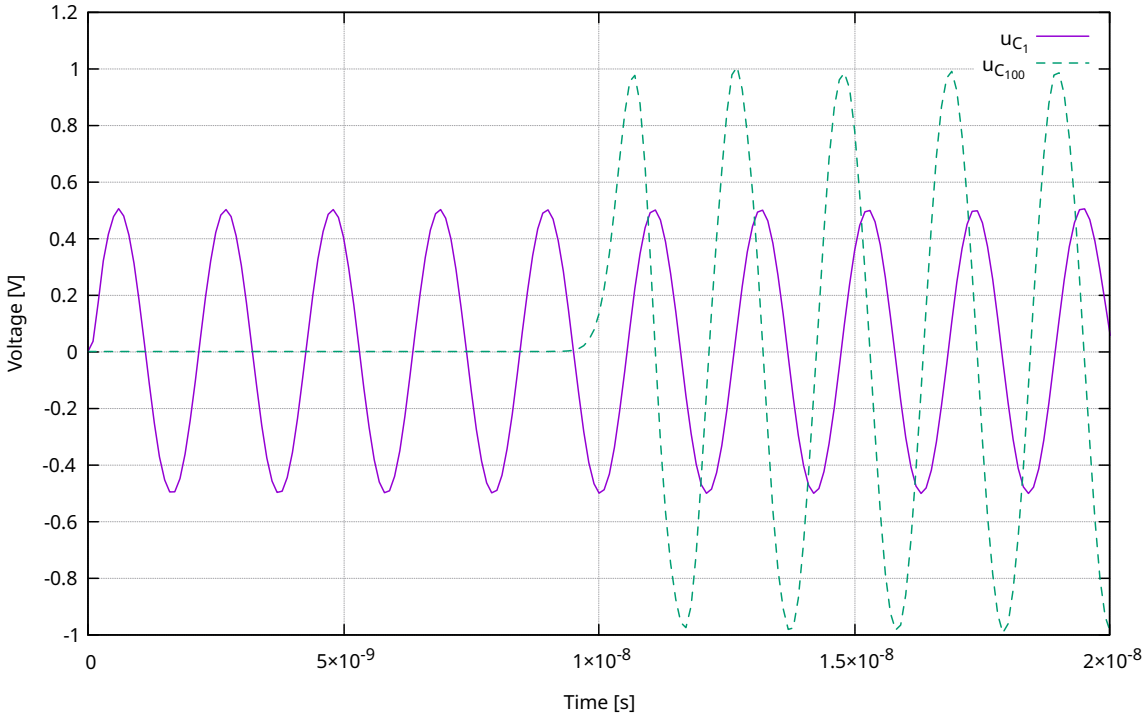


Figure 5.5: Simulation of telegraph line using harmonic input with no adjustment: $R_1 = 100\,\Omega$ and $R_2 = 10 \times 10^{-12}\,\Omega$ as specified in article [17, p. 8, Figure 5a)].

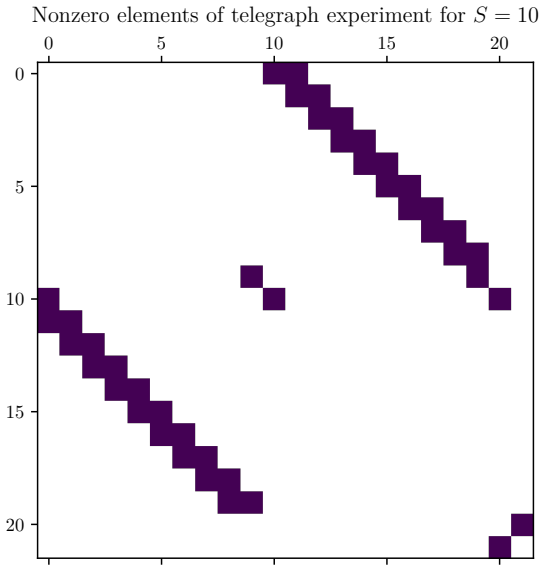


Figure 5.6: Nonzero elements of sparse matrix for telegraph line model with $S = 10$.

5.2.4 Measurements results

This section provides measurement data for the Telegraph line problem described in Section 5.2 with the configuration for adjusted line from Subsection 5.2.3 with variable segment

count. All measurements were done for the sparse matrix format CSR as described in Section 4.3. The measured environments were 1 thread on CPU (singlecore) and the GPU.

For the harmonic input the number of equations is $N = 2 * S + 2$ and the number of nonzero elements in the Jacobian matrix $\mathbf{A}$ is $NNZ = 2 * 2 * S + 3$.

The computation of bigger problem was not able to be performed due to the size of result data of the methods. Precisely, the output of Euler method for non-diverging solution had over 50 GiB. This is due to the CSR format being more significantly smaller than the dense format, which would limit the size of the problem. A computation where only the last result vector $\mathbf{y}$ is stored would have to be used.

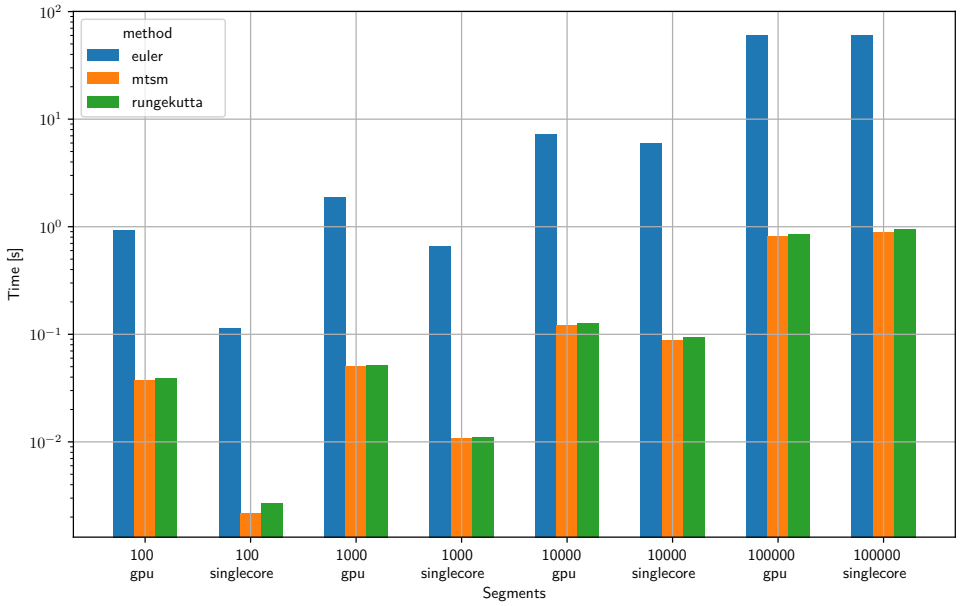


Figure 5.7: Bar graph of measured computation time for reaching solution grouped by problem size and running environment for each method with logarithmic scale on the y-axis. The Euler method takes significantly longer than the other methods, since it required much higher number of steps (lower step size) to reach non-divergent solution (see Table 5.2).

Segments	Method	Hardware	Time [s]
100	euler	singlecore	0.114456
100	euler	gpu	0.920177
100	rungekutta	singlecore	**0.002659**
100	rungekutta	gpu	0.038918
100	mtsm	singlecore	0.002171
100	mtsm	gpu	0.037504
1000	euler	singlecore	0.653903
1000	euler	gpu	1.855362
1000	rungekutta	singlecore	0.011064
1000	rungekutta	gpu	0.051741
1000	mtsm	singlecore	**0.010740**
1000	mtsm	gpu	0.050579
10000	euler	singlecore	5.962509
10000	euler	gpu	7.223816
10000	rungekutta	singlecore	0.094106
10000	rungekutta	gpu	0.124698
10000	mtsm	singlecore	**0.087440**
10000	mtsm	gpu	0.120943
100000	euler	singlecore	59.365885
100000	euler	gpu	60.209152
100000	rungekutta	singlecore	0.932653
100000	rungekutta	gpu	0.839165
100000	mtsm	singlecore	0.889037
100000	mtsm	gpu	**0.820085**

Table 5.1: Measured mean time of 5 iterations for given method and problem size. Each method was ran on GPU, CPU with only 1 thread allowed (singlecore). In bold text are visualized best times for each size. As we can see Euler method proven to be the worst method for this problem, due to the high number of steps it required to reach non-diverging solution.

Segments	Method	StepCount	Occupancy	Threads	Teams	Time [s]
100	euler	15000	2.00%	32	7	0.920177
100	rungekutta	200	2.00%	32	7	0.038918
100	mtsm	200	2.00%	32	7	0.037504
1000	euler	15000	5.00%	32	63	1.855362
1000	rungekutta	200	5.00%	32	63	0.051741
1000	mtsm	200	5.00%	32	63	0.050579
10000	euler	15000	50.00%	256	79	7.223816
10000	rungekutta	200	50.00%	256	79	0.124698
10000	mtsm	200	50.00%	256	79	0.120943
100000	euler	15000	100.00%	256	782	60.209152
100000	rungekutta	200	100.00%	256	782	0.839165
100000	mtsm	200	100.00%	256	782	0.820085

Table 5.2: Detailed information regarding the offloaded functions onto the GPU, such as number of teams launched in the league, number of threads in one team, functions occupancy of the GPU resources and numbers of computed steps to receive non-divergent solution. The number of steps for each method and size were determined experimentally. The most notable results are regarding the problem size and occupancy. The occupancy scales for all methods nicely with the size of the problem, reaching 100% occupancy for size of 100000 segments.

Segments	Method	Absolute mean error	Relative mean error	Absolute maximum error
100	euler	6.70e-06	0.0061%	3.80e-05
100	rungekutta	**4.78e-08**	0.00016%	**2.60e-07**
100	mtsm	6.67e-08	**0.00015%**	4.60e-07
1000	euler	8.11e-07	0.0023%	3.58e-05
1000	rungekutta	**1.08e-08**	**2.5e-05%**	**3.60e-07**
1000	mtsm	1.27e-08	3.3e-05%	3.84e-07
10000	euler	8.12e-08	0.00023%	3.58e-05
10000	rungekutta	**1.08e-09**	**2.6e-06%**	**3.60e-07**
10000	mtsm	1.27e-09	3.3e-06%	3.84e-07
100000	euler	8.12e-09	2.3e-05%	3.58e-05
100000	rungekutta	**1.08e-10**	**2.6e-07%**	**3.60e-07**
100000	mtsm	1.27e-10	3.3e-07%	3.84e-07

Table 5.3: Comparison of method's solution using singlecore setup (CPU with only 1 thread), to eliminate error from parallelization, and run on the GPU. The errors in the table are errors introduced by the parallelization of the method on the GPU. The Runge-Kutta has the smallest error introduced by the parallelization, closely followed by the MTSM.

Segments	MethodA	MethodB	Absolute mean error	Absolute maximum error
100	euler	rungekutta	4.53e-02	2.57e-01
100	euler	mtsm	4.53e-02	2.57e-01
100	rungekutta	euler	4.53e-02	2.57e-01
100	rungekutta	mtsm	**1.73e-07**	**8.70e-07**
1000	euler	rungekutta	7.41e-03	2.15e-01
1000	euler	mtsm	7.41e-03	2.15e-01
1000	rungekutta	euler	7.41e-03	2.15e-01
1000	rungekutta	mtsm	**2.61e-08**	**7.80e-07**
10000	euler	rungekutta	7.42e-04	2.15e-01
10000	euler	mtsm	7.42e-04	2.15e-01
10000	rungekutta	euler	7.42e-04	2.15e-01
10000	rungekutta	mtsm	**2.62e-09**	**7.80e-07**
100000	euler	rungekutta	7.42e-05	2.15e-01
100000	euler	mtsm	7.42e-05	2.15e-01
100000	rungekutta	euler	7.42e-05	2.15e-01
100000	rungekutta	mtsm	**2.62e-10**	**7.80e-07**

Table 5.5: Comparison of errors between individual solutions. For each different size of the problem all possible comparisons between used methods. The best results for each size are highlighted. As we can see the closes to each other across all sizes are Runge-Kutta 4 and MTSM with order 4, which is reasonable, since they have same stability region.

5.3 Incompressible Stokes system

The experiment is incompressible Stokes problem with finite element discretization with P^1-Bubble/P^1 from [12, p. 245, Chapter 3]. This discretization limits the precision of a solution and therefore limits both maximum MTSM order and datatype.

This problem was chosen, because it can be rewritten into linear Initial Value Problem of the following form and its $\mathbf{A}_{aux}$ *is dense*:

$$\dot{\mathbf{y}} = \mathbf{A}_{aux}\mathbf{y} + \mathbf{b}, \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{5.16}$$

where $\mathbf{y} = \begin{pmatrix} \mathbf{u} \\ g_1 \\ g_2 \\ g \end{pmatrix}$.

5.3.1 Final equation

Altered equation to Stokes problem defined in [12, p. 244, Chapter 2]. The difference is that the parameter $\alpha = 0$ and additional term $\frac{\partial \mathbf{u}}{\partial t}$.

Let Ω be a bounded domain in $\mathbb{R}^2$ with Dirichlet boundary $\gamma_D \subset \Omega$. We consider a Stokes flow in Ω with time derivative term by the given two equations – Momentum balance equation (5.17) and Continuity equation in incompressible flow (5.18):

$$\frac{\partial \mathbf{u}}{\partial t} - \nu\nabla^2\mathbf{u} + \nabla p = \mathbf{f} \quad \text{in } \Omega \tag{5.17}$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \tag{5.18}$$

$$\mathbf{u} = 0 \quad \text{in } \gamma_D, \tag{5.19}$$

where

- $\mathbf{u}$ is the flow velocity in m/s,
- t is time in s,
- p is the fluid pressure in Pa,
- $\mathbf{f}$ are forces acting on the fluid,
- Ω is a bounded domain in $\mathbb{R}^2$.

- $\nu = \frac{\mu}{\rho} > 0$ is kinematic viscosity m^2/s,
- ∇ is gradient operator,
- $\nabla\cdot$ is the divergence operator,
- $\Delta = \nabla \cdot \nabla = \nabla^2$ is the Laplace operator,

For 2D the equations would have the following form:

$$\frac{\partial u_1}{\partial t} - \nu\left(\frac{\partial^2 u_1}{\partial x_1^2} + \frac{\partial^2 u_1}{\partial x_2^2}\right) + \frac{\partial p}{\partial x_1} = f_1 \tag{5.20}$$

$$\frac{\partial u_2}{\partial t} - \nu\left(\frac{\partial^2 u_2}{\partial x_1^2} + \frac{\partial^2 u_2}{\partial x_2^2}\right) + \frac{\partial p}{\partial x_2} = f_2 \tag{5.21}$$

$$\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} = 0$$

5.3.2 Analytic solution

Functions $g_1(t)$, $g_2(t)$, $g(t)$ are only time dependent, no dependency on space.

$$\begin{aligned} u_1 &= g_1(t) \cdot h_1(x_1, x_2) \\ u_2 &= g_2(t) \cdot h_2(x_1, x_2) \\ p &= g(t) \cdot h(x_1, x_2), \end{aligned}$$

where $h_1(x_1, x_2), h_2(x_1, x_2)$ and $h(x_1, x_2)$ are solutions of Stokes system (5.17) without the term $\frac{\partial \mathbf{u}}{\partial t}$ from the article [13, p. 13, Section 6.1].

$$\begin{aligned} h_1(x_1, x_2) &= -\cos(2\pi x_1)\sin(2\pi x_2) + \sin(2\pi x_2) \\ h_2(x_1, x_2) &= \sin(2\pi x_1)\cos(2\pi x_2) - \sin(2\pi x_1) \\ h(x_1, x_2) &= 2\pi(\cos(2\pi x_2) - \cos(2\pi x_1)) \end{aligned}$$

Functions $g(t)$, $g_1(t)$ and $g_2(t)$ are chosen such that they satisfy the incompressible flow constraint (5.18)

$$\begin{aligned} & & \nabla \cdot \mathbf{u} &= 0 \\ \implies & \frac{\partial u_1}{\partial x_1} + & \frac{\partial u_2}{\partial x_2} &= 0 \\ g_1(t) & \frac{\partial h_1(x_1, x_2)}{\partial x_1} + g_2(t) & \frac{\partial h_2(x_1, x_2)}{\partial x_2} &= 0 \end{aligned} \tag{5.22}$$

The (5.23) should be fulfilled from Stokes solution.

$$\frac{\partial h_1(x_1, x_2)}{\partial x_1} + \frac{\partial h_2(x_1, x_2)}{\partial x_2} = 0 \tag{5.23}$$

Therefore to satisfy (5.22) the choice of $g_1(t)$ and $g_2(t)$ should be

$$g_1(t) = g_2(t). \tag{5.24}$$

There is no constraint for $g(t)$.

5.3.3 Partial derivatives

The following section uses the notation for partial derivatives as $f_x = \frac{\partial f}{\partial x}(x, y, \ldots)$ and $f_{xy} = (f_x)_y = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right)$.

Time derivatives $\left(\frac{\partial}{\partial t}\right)$

$$\begin{aligned} u_{1,t} &= \frac{\partial}{\partial t}\left(g_1(t) \cdot h_1(x_1, x_2)\right) = \frac{\partial g_1(t)}{\partial t} \cdot h_1(x_1, x_2) \\ u_{2,t} &= \frac{\partial}{\partial t}\left(g_2(t) \cdot h_2(x_1, x_2)\right) = \frac{\partial g_2(t)}{\partial t} \cdot h_2(x_1, x_2) \end{aligned}$$

Analytic solution derivatives

$$
\begin{aligned}
h_{1,x_1} &= \frac{\partial h_1(x_1,x_2)}{\partial x_1} = \frac{\partial}{\partial x_1}(-\cos(2\pi x_1)\sin(2\pi x_2) + \sin(2\pi x_2)) \\
&= 2\pi \sin(2\pi x_1)\cdot\sin(2\pi x_2) \\
h_{1,x_1x_1} &= \frac{\partial h_{1,x_1(x_1,x_2)}}{\partial x_1} = (2\pi^2)\cos(2\pi x_1)\cdot\sin(2\pi x_2) \\
h_{1,x_2} &= \frac{\partial h_1(x_1,x_2)}{\partial x_2} = -2\pi\cos(2\pi x_1)\cdot\cos(2\pi x_2) + 2\pi\cos(2\pi x_2) \\
h_{1,x_2x_2} &= \frac{\partial h_{1,x_2}(x_1,x_2)}{\partial x_2} = (2\pi)^2\cos(2\pi x_1)\sin(2\pi x_2) - (2\pi)^2\sin(2\pi x_2) \\
h_{2,x_1} &= \frac{\partial h_2(x_1,x_2)}{\partial x_1} = 2\pi\cos(2\pi x_1)\cos(2\pi x_2) - 2\pi cos(2\pi x_1) \\
h_{2,x_1x_1} &= \frac{\partial h_{2,x_1}(x_1,x_2)}{\partial x_1} - (2\pi)^2\sin(2\pi x_1)\cos(2\pi x_2) + (2\pi)^2\sin(2\pi x_1) \\
h_{2,x_2} &= \frac{\partial h_2(x_1,x_2)}{\partial x_2} = -2\pi\sin(2\pi x_1)\sin(2\pi x_2) \\
h_{2,x_2x_2} &= \frac{\partial h_{2,x_2}x_1,x_2)}{\partial x_2} = -(2\pi)^2\sin(2\pi x_1)\cos(2\pi x_2) \\
h_{x_1} &= \frac{\partial h(x_1,x_2)}{\partial x_1} = (2\pi)^2\sin(2\pi x_1) \\
h_{x_2} &= \frac{\partial h(x_1,x_2)}{\partial x_2} = (2\pi)^2\sin(2\pi x_2)
\end{aligned}
$$

Pressure derivatives

$$
\begin{aligned}
p_{x_1} &= \frac{\partial}{\partial x_1}\left(g(t)\cdot h(x_1,x_2)\right) = g(t)\cdot h_{x_1} \\
p_{x_2} &= \frac{\partial}{\partial x_2}\left(g(t)\cdot h(x_1,x_2)\right) = g(t)\cdot h_{x_2}
\end{aligned}
$$

Velocity derivatives

$$
\begin{aligned}
u_{1,x_1} &= \frac{\partial}{\partial x_1}\left(g_1(t)\cdot h_1(x_1,x_2)\right) = g_1(t)\cdot h_{1,x_1} \\
u_{1,x_1x_1} &= \frac{\partial^2 u_1}{\partial x_1^2} = \frac{\partial}{\partial x_1}\left(g_1(t)\cdot h_{1,x_1}\right) = g_1(t)\cdot h_{1,x_1x_1} \\
u_{1,x_2} &= \frac{\partial}{\partial x_2}\left(g_1(t)\cdot h_1(x_1,x_2)\right) = g_1(t)\cdot h_{1,x_2} \\
u_{1,x_2x_2} &= \frac{\partial^2 u_1}{\partial x_2^2} = \frac{\partial}{\partial x_2}\left(g_1(t)\cdot h_{1,x_2}\right) = g_1(t)\cdot h_{1,x_2x_2} \\
u_{2,x_1} &= \frac{\partial}{\partial x_1}\left(g_2(t)\cdot h_2(x_1,x_2)\right) = g_2(t)\cdot h_{2,x_1} \\
u_{2,x_1x_1} &= \frac{\partial^2 u_2}{\partial x_1^2} = \frac{\partial}{\partial x_1}\left(g_2(t)\cdot h_{2,x_1}\right) = g_2(t)\cdot h_{2,x_1x_1} \\
u_{2,x_2} &= \frac{\partial}{\partial x_2}\left(g_2(t)\cdot h_2(x_1,x_2)\right) = g_2(t)\cdot h_{2,x_2} \\
u_{2,x_2x_2} &= \frac{\partial^2 u_2}{\partial x_2^2} = \frac{\partial}{\partial x_2}\left(g_2(t)\cdot h_{2,x_2}\right) = g_2(t)\cdot h_{2,x_2x_2}
\end{aligned}
$$

5.3.4 Right-hand side

The $\dot{g}(t)$ notation is used for time derivative $\dot{g}(t) = \frac{\partial g(t)}{\partial t}$. The rand-hand side, i.e. forces of (5.20) and (5.21) respectively, are the following:

$$f_1 = \dot{g}_1(t) \cdot [h_1(x_1, x_2)] + g_1[-\nu \cdot (h_{1,x_1x_1} + h_{1,x_2x_2})] + g[h_{x_1}]$$
$$f_2 = \dot{g}_2(t) \cdot [h_2(x_1, x_2)] + g_2[-\nu \cdot (h_{2,x_1x_1} + h_{2,x_2x_2})] + g[h_{x_2}],$$

where terms inside $[\dots]$ are generated constants (spatial integrals) in assembly of matrices and right-hand side.

Functions g_1, g_2 and g will be solved by using ODE system of IVP. Since (5.24) is the only constraint for the functions g, g_1 and g_2, we are able to choose for example:

$$g_1(t) = g_2(t) = g(t) = e^{-\lambda t}$$

Which can generated by the following IVP with system of ODEs:

$$\begin{aligned} \dot{g_1} &= -g_1, & g_1(0) &= 1, \\ \dot{g_2} &= -g_2, & g_2(0) &= 1, \\ \dot{g} &= -g, & g(0) &= 1. \end{aligned} \tag{5.25}$$

5.3.5 Algebraic form

The Stokes system (5.17), (5.18) and (5.19) can be rewritten after finite element method discretization in a linear algebraic formulation.

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B}^\top \\ -\mathbf{B} & -\tilde{\mathbf{E}} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \tilde{c} \end{bmatrix},$$

where $\mathbf{A}$ is (symmetric) positive definite block diagonal matrix, $\mathbf{C}$ is positive semi-definite matrix coming from elimination of bubble velocity component in finite element discretization [12, p. 253, (5.1)].

The equation $Bu = 0$ is rewritten (5.18), to which the stabilization term $-\tilde{E}p = \tilde{c}$ is added. Which allows for elimination of $\mathbf{p}$ using Schur complement, which reduces the numbers of equations.

$$\begin{aligned} M\dot{u} + Au + B^T p &= f \\ Bu - \tilde{E}p &= \tilde{c} \\ \rightarrow p &= \tilde{E}^{-1}(Bu - \tilde{c}) \end{aligned}$$

$$\begin{aligned} M\dot{u} &= (-A - B^T\tilde{E}^{-1}B)u + B^T\tilde{E}^{-1}\tilde{c} + f \\ \dot{u} &= M^{-1}(-A - B^T\tilde{E}^{-1}B)u + M^{-1}B^T\tilde{E}^{-1}\tilde{c} + M^{-1}f, \quad u(0) = u_0, \end{aligned} \tag{5.26}$$

where

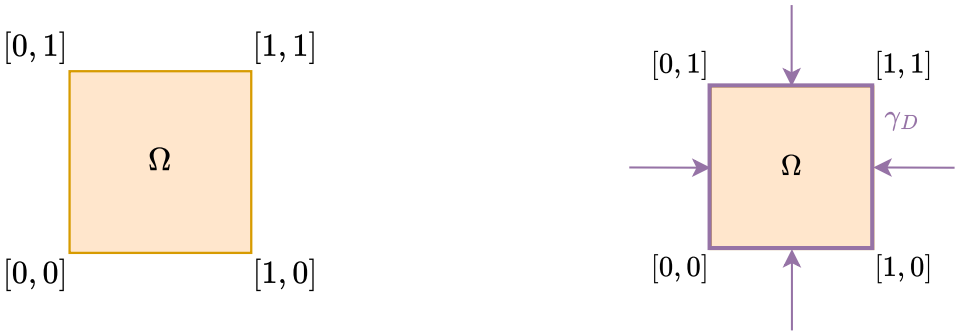
- M is the "mass matrix",
- A contains the matrix K, which is "stiffness matrix".

Combining (5.25) and (5.26) we obtain the IVP (5.16).

5.3.6 Example domain

Let $\Omega = (0,1)\times(0,1)$ and $\gamma_D = (0,1)\times\{0,1\}\cup\{0,1\}\times(0,1)$ (Dirichlet boundary condition). Data are defined as follows:

- $\mathbf{f} = -\nu\Delta\mathbf{u}_{exp} + \nabla p_{exp}$,
- $\nu = 1$,
- $\mathbf{u} = \mathbf{0}$ in γ_D.



(a) Visualization of specified Stokes domain Ω for experiments.

(b) Visualization of Dirichlet boundary γ_D over the specified Stokes domain Ω.

Figure 5.8: The Dirichlet boundary γ_D – square of purple color with arrows pointing to it – is the edge of Stokes domain Ω, which is visualized as square filled with orange color.

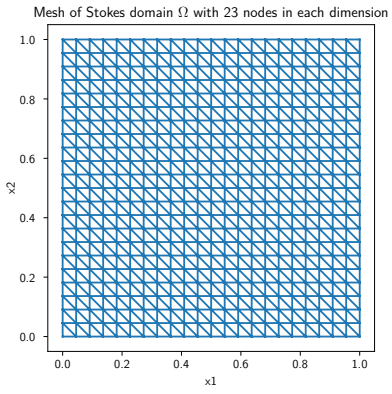
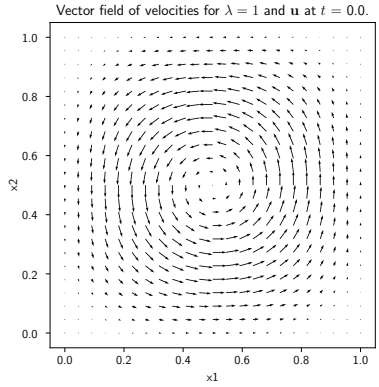
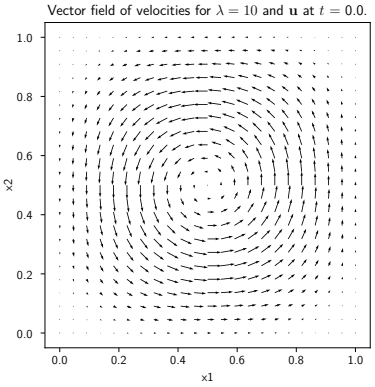


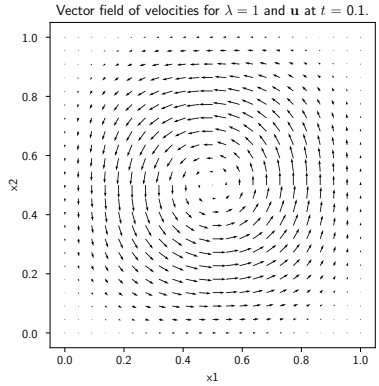
Figure 5.9: The Stokes domain Ω discretized with 23 nodes in each dimension.



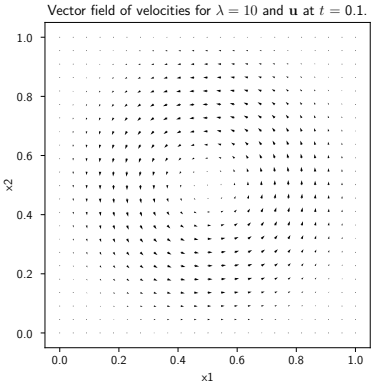
(a) Velocities $\mathbf{u}$ at $t = 0[s]$ for $\lambda = 1$.



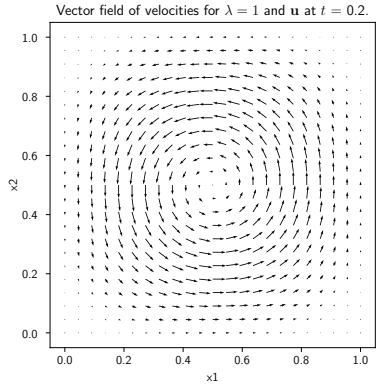
(b) Velocities $\mathbf{u}$ at $t = 0[s]$ for $\lambda = 10$.



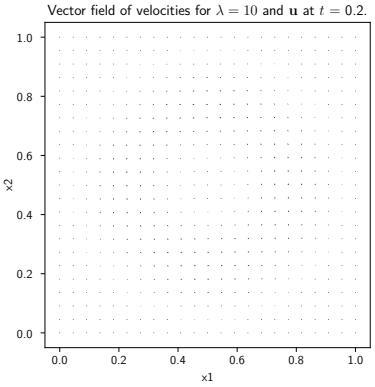
(c) Velocities $\mathbf{u}$ at $t = 0.1[s]$ for $\lambda = 1$.



(d) Velocities $\mathbf{u}$ at $t = 0.1[s]$ for $\lambda = 10$.



(e) Velocities $\mathbf{u}$ at $t = 0.2[s]$ for $\lambda = 1$.



(f) Velocities $\mathbf{u}$ at $t = 0.2[s]$ for $\lambda = 10$.

Figure 5.10: Visualization of velocities $\mathbf{u}$ as a vector field over Stokes domain Ω with two different λ parameters. The λ parameter controls the slowdown speed of velocities, as it controls the time dependant variable in the system.

5.3.7 Measured results

This section contains measured results of the earlier introduced incompressible Stokes problem (5.16). The measurements contain time execution of implemented methods ran on the

CPU (with single core and multiple cores) and on the GPU, error between different methods, occupancy and league size for the GPU offloading.

The measurements were ran with domain explained in previous section Subsection 5.3.6, with the λ parameter set to 10. The segments in the problem size is the number of nodes +1. Then for number of segments S, the discretized domain has $(S-1)^2$ nodes and the linear system has $2(S-1)^2+3$ equations.

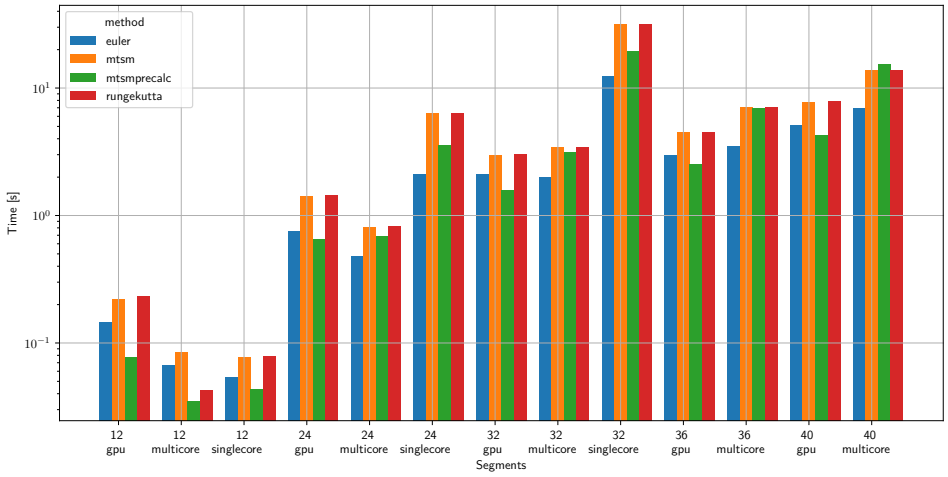


Figure 5.11: Bar graph of measured computation time grouped by problem size and running environment for each method with logarithmic scale on the y-axis. The singlecore results are missing for sizes bigger than 32, since single run for those sizes took over 2 and 5 minutes for sizes 36 and 40, respectively.

Segments	Method	Hardware	Time [s]
12	euler	singlecore	0.053461
12	euler	gpu	0.145413
12	euler	multicore	0.066439
12	rungekutta	singlecore	0.078725
12	rungekutta	gpu	0.230070
12	rungekutta	multicore	0.042560
12	mtsm	singlecore	0.076944
12	mtsm	gpu	0.221274
12	mtsm	multicore	0.084548
12	mtsmprecalc	singlecore	0.043020
12	mtsmprecalc	gpu	0.076458
12	mtsmprecalc	multicore	**0.034551**
24	euler	singlecore	2.113388
24	euler	gpu	0.754134
24	euler	multicore	**0.474342**
24	rungekutta	singlecore	6.325740

Continued on next page

Segments	Method	Hardware	Time [s]
24	rungekutta	gpu	1.437499
24	rungekutta	multicore	0.821570
24	mtsm	singlecore	6.310087
24	mtsm	gpu	1.411801
24	mtsm	multicore	0.806813
24	mtsmprecalc	singlecore	3.568807
24	mtsmprecalc	gpu	0.645680
24	mtsmprecalc	multicore	0.683483
32	euler	singlecore	12.333441
32	euler	gpu	2.108623
32	euler	multicore	1.986528
32	rungekutta	singlecore	31.664917
32	rungekutta	gpu	3.030217
32	rungekutta	multicore	3.408804
32	mtsm	singlecore	31.625379
32	mtsm	gpu	2.980663
32	mtsm	multicore	3.417567
32	mtsmprecalc	singlecore	19.447555
32	mtsmprecalc	gpu	**1.572505**
32	mtsmprecalc	multicore	3.118901
36	euler	gpu	2.961507
36	euler	multicore	3.457246
36	rungekutta	gpu	4.512470
36	rungekutta	multicore	7.006524
36	mtsm	gpu	4.468597
36	mtsm	multicore	6.998016
36	mtsmprecalc	gpu	**2.507617**
36	mtsmprecalc	multicore	6.932106
40	euler	gpu	5.120002
40	euler	multicore	6.896033
40	rungekutta	gpu	7.841741
40	rungekutta	multicore	13.755517
40	mtsm	gpu	7.755309
40	mtsm	multicore	13.656343
40	mtsmprecalc	gpu	**4.251438**
40	mtsmprecalc	multicore	15.206521

Table 5.6: Measured mean time of 5 iterations for given method and problem size. Each method was ran on GPU, CPU with only 1 thread allowed (singlecore) and CPU with all cores allowed (multicore). The fastest combination from the size of 32 segments is the precalculated MTSM on the GPU.

Segments	Method	StepCount	Occupancy	Threads	Teams	Time [s]
12	euler	2000	2.00%	32	8	0.145413
12	rungekutta	1000	2.00%	32	8	0.230070
12	mtsm	1000	2.00%	32	8	0.221274
12	mtsmprecalc	1000	**100.00%**	256	235	0.076458
24	euler	5000	5.00%	32	34	0.754134
24	rungekutta	4000	5.00%	32	34	1.437499
24	mtsm	4000	5.00%	32	34	1.411801
24	mtsmprecalc	4000	**100.00%**	256	4398	0.645680
32	euler	9000	5.00%	32	61	2.108623
32	rungekutta	6000	5.00%	32	61	3.030217
32	mtsm	6000	5.00%	32	61	2.980663
32	mtsmprecalc	6000	**100.00%**	256	14476	1.572505
36	euler	11000	7.00%	32	77	2.961507
36	rungekutta	8000	7.00%	32	77	4.512470
36	mtsm	8000	7.00%	32	77	4.468597
36	mtsmprecalc	8000	**100.00%**	256	23505	2.507617
40	euler	15000	7.00%	32	96	5.120002
40	rungekutta	10000	7.00%	32	96	7.841741
40	mtsm	10000	7.00%	32	96	7.755309
40	mtsmprecalc	10000	**100.00%**	256	36219	4.251438

Table 5.7: Detailed information regarding the offloaded functions onto the GPU, such as number of teams launched in the league, number of threads in one team, functions occupancy of the GPU resources and numbers of computed steps to receive non-divergent solution. The number of steps for each method and size were determined experimentally. The most notable results are regarding the precalculated MTSM, with occupancy of 100% and biggest size of league, since the analyzed function is matrix multiplication.

Segments	Method	Absolute mean error	Relative mean error	Absolute maximum error
12	euler	1.73e-08	0.71%	**9.00e-08**
12	rungekutta	**1.72e-08**	0.76%	1.20e-07
12	mtsm	2.94e-08	**0.23%**	1.70e-07
12	mtsmprecalc	1.85e-07	1.1%	6.00e-07
24	euler	4.91e-08	0.16%	3.40e-07
24	rungekutta	**2.32e-08**	0.079%	**1.30e-07**
24	mtsm	4.43e-08	0.36%	3.00e-07
24	mtsmprecalc	1.75e-07	**0.039%**	7.40e-07
32	euler	7.24e-08	0.034%	5.00e-07
32	rungekutta	**3.40e-08**	0.02%	**2.10e-07**
32	mtsm	5.36e-08	0.12%	3.40e-07
32	mtsmprecalc	1.99e-07	**0.0084%**	1.15e-06
36	euler	6.19e-08	0.032%	3.90e-07
36	rungekutta	**4.31e-08**	0.0064%	**2.60e-07**
36	mtsm	7.36e-08	0.014%	4.50e-07
36	mtsmprecalc	2.51e-07	**0.0048%**	1.36e-06
40	euler	5.70e-08	0.0091%	3.60e-07
40	rungekutta	**3.61e-08**	**0.0045%**	**2.20e-07**
40	mtsm	9.36e-08	0.018%	6.00e-07
40	mtsmprecalc	3.48e-07	0.02%	1.78e-06

Table 5.8: Comparison of method's solution using singlecore setup (CPU with only 1 thread), to eliminate error from parallelization, and run on the GPU. The smallest error introduced by parallelization for absolute value is usually for the Runge-Kutta 4 stage method. However, the relative error is closest be MTSM, precalculated MTSM and for the last problem Runge-Kutta again.

Segments	MethodA	MethodB	Absolute mean error	Absolute maximum error
12	euler	rungekutta	9.17e-05	2.49e-04
12	euler	mtsm	9.20e-05	2.50e-04
12	euler	mtsmprecalc	9.68e-05	2.63e-04
12	rungekutta	mtsm	**2.81e-07**	**8.70e-07**
12	rungekutta	mtsmprecalc	5.07e-06	1.40e-05
12	mtsm	mtsmprecalc	4.79e-06	1.33e-05
24	euler	rungekutta	3.64e-05	1.06e-04
24	euler	mtsm	3.52e-05	1.03e-04
24	euler	mtsmprecalc	2.12e-05	6.19e-05
24	rungekutta	mtsm	**1.19e-06**	**3.77e-06**
24	rungekutta	mtsmprecalc	1.53e-05	4.75e-05

Continued on next page

Segments	MethodA	MethodB	Absolute mean error	Absolute maximum error
24	mtsm	mtsmprecalc	1.41e-05	4.41e-05
32	euler	rungekutta	2.03e-05	5.98e-05
32	euler	mtsm	2.92e-05	8.64e-05
32	euler	mtsmprecalc	4.18e-05	1.34e-04
32	rungekutta	mtsm	**8.91e-06**	**2.74e-05**
32	rungekutta	mtsmprecalc	6.16e-05	1.85e-04
32	mtsm	mtsmprecalc	7.04e-05	2.09e-04
36	euler	rungekutta	1.64e-05	4.91e-05
36	euler	mtsm	**2.18e-07**	**3.71e-06**
36	euler	mtsmprecalc	1.09e-04	3.48e-04
36	rungekutta	mtsm	1.64e-05	4.97e-05
36	rungekutta	mtsmprecalc	1.25e-04	3.95e-04
36	mtsm	mtsmprecalc	1.09e-04	3.48e-04
40	euler	rungekutta	1.19e-05	3.62e-05
40	euler	mtsm	**3.94e-06**	**1.25e-05**
40	euler	mtsmprecalc	2.91e-04	8.95e-04
40	rungekutta	mtsm	1.59e-05	4.79e-05
40	rungekutta	mtsmprecalc	3.03e-04	9.27e-04
40	mtsm	mtsmprecalc	2.87e-04	8.83e-04

Table 5.9: Comparison of errors between individual solutions. For each different size of the problem all possible comparisons between used methods. For the first half of the problem sizes the closest methods are Runge-Kutta and MTSM. For the other half the Euler method and MTSM are closest.

5.4 Future work

The topic of higher precision libraries was research and it was discovered that GPU do not have proper libraries for the use of higher-precision types. For example the article for Cuda multiple precision arithmetic library[3] was found, however the link for freely provided source do no longer work and the library seems to not be updated since the release from the year 2016. The working solution for multi-precision libraries seems to be the NVIDIA's experimental library CGBN[4], however this is library only for NVIDIA GPUs and is marked as for experimental use only. The last option for higher-precision types that was considered was the 128-bit datatype for floating-point[5]. This is also not used, since I did not find portable version for the use with OpenMP.

Another work for the future would be testing more possible optimizations for computing the higher-order of the MTSM similar to the precalculation. One such consider possibility was the sub-matrix multiplication.

[3] https://rd.springer.com/chapter/10.1007/978-3-319-42432-3_29
[4] https://github.com/NVlabs/CGBN
[5] https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__QUAD.html

An interesting work to be done is the implementation of Stopping rule similar to usual implementations of numerical methods using the tolerance instead of user provided step size and observe the impact on efficiency.

A more fine grained optimization would be the overlap of computation and transferring the result of last step, since I was unable to implement this using the OpenMP directives.

Chapter 6

Conclusion

The main goal of this Thesis was to implemented and test the numerical method for solving ODE using Taylor series expansion on the GPU. Which was successfully implemented implemented and compared to other methods.

The first part of the Thesis provided necessary mathematical introduction to the topic together with the widely used method for numerical approximation of ODEs. Then the thesis provided description of the implementation with definitions for big problems to test the effectivity of the implementation compared to the other methods.

The unimplemented part from the assignment is the use of higher-precision libraries on the GPU. This topic was researched and is described in more detail in the future work section. In general GPUs do not support higher-precision libraries and tend to go the other way of having types for lower precision floating-point datatypes. The found solutions very considered too specific for my implementation as they require specific solution and specific environment (NVIDIA GPU only).

The results were tested with the fixed order of 4 for the better comparison to the Runge-Kutta 4-stage method. The results of benchmarks showed that the occupancy of GPU resources *grow with the number of equations* in system of ODEs. The precalculated version of the MTSM proved to be faster for full occupancy usage (for the matrix multiplication function) and faster in time than the other methods. The smallest error introduced by the parallelization was for the Runge-Kutta method very closely followed by MTSM.

Bibliography

[1] *OpenMP Application Program Interface Version 4.5*. Available at: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[2] Ahnert, K.; Demidov, D. and Mulansky, M. Solving Ordinary Differential Equations on GPUs. In: Kindratenko, V., ed. *Numerical Computations with GPUs*. Springer International Publishing, p. 125–157. ISBN 978-3-319-06547-2 978-3-319-06548-9. Available at: https://link.springer.com/10.1007/978-3-319-06548-9_7.

[3] Bank, R. E. and Douglas, C. C. Sparse matrix multiplication package (SMMP), vol. 1, no. 1, p. 127–137. ISSN 1572-9044. Available at: https://doi.org/10.1007/BF02070824.

[4] Buluç, A.; Fineman, J. T.; Frigo, M.; Gilbert, J. R. and Leiserson, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, p. 233–244. ISBN 978-1-60558-606-9. Available at: https://dl.acm.org/doi/10.1145/1583991.1584053.

[5] Burden, R. L. and Faires, J. D. *Numerical analysis*. 9. ed., International edth ed. Brooks/Cole, Cengage Learning. ISBN 978-0-538-73351-9 978-0-538-73564-3.

[6] Butcher, J. C. *Numerical methods for ordinary differential equations*. Wiley. ISBN 978-0-470-72335-7 978-0-470-75376-7.

[7] Corless, R. M.; Kaya, C. Y. and Moir, R. H. C. Optimal residuals and the Dahlquist test problem, vol. 81, no. 4, p. 1253–1274. ISSN 1017-1398, 1572-9265. Available at: http://link.springer.com/10.1007/s11075-018-0624-x.

[8] E. Hairer; S. P. Nørsett and G. Wanner. Solving ordinary differential equations. 1: Nonstiff problems. In:. 2., rev. ed., 1. softcover printingth ed. Springer, no. 8. Springer series in computational mathematics. ISBN 978-3-540-56670-0. Num Pages: 528.

[9] Garcia Molla, V.; Liberos, A.; Vidal, A.; Guillem, M.; Millet, J. et al. Adaptive step ODE algorithms for the 3D simulation of electric heart activity with graphics processing units, vol. 44, p. 15–26. ISSN 00104825. Available at: https://www.cinc.org/archives/2011/pdf/0233.pdf.

[10] Hairer, E. and Wanner, G. *Solving Ordinary Differential Equations II*. Springer Berlin Heidelberg. Springer Series in Computational Mathematics. ISBN

978-3-662-09949-0 978-3-662-09947-6. Available at: http://link.springer.com/10.1007/978-3-662-09947-6.

[11] Ing. Václav Šátek, Ph.D.. *Analýza stiff soustav diferenciálních rovnic*. Phdthesis. Available at: https://www.vut.cz/studenti/zav-prace/detail/99803.

[12] Koko, J. Efficient MATLAB Codes for the 2D/3D Stokes Equation with the Mini-Element, vol. 30, no. 2, p. 243–268. ISSN 0868-4952, 1822-8844. Available at: https://informatica.vu.lt/doi/10.15388/Informatica.2019.205.

[13] Koko, J. Vectorized Matlab Codes for the Stokes Problem with P 1Bubble/P 1 Finite Element.

[14] Nečasová, G. *PARALLEL NUMERIC SOLUTION OF DIFFERENTIAL EQUATIONS*. Phdthesis.

[15] NVIDIA Corporation. *CUDA Programming Guide*. Available at: https://docs.nvidia.com/cuda/cuda-programming-guide/index.html.

[16] Veigend, P. *High order numerical method in modelling and control systems*. Brno, CZ. Phdthesis. Available at: https://www.fit.vut.cz/study/phd-thesis/813/.

[17] Veigend, P.; Nečasová, G. and Šátek, V. Model of the telegraph line and its numerical solution, vol. 8, no. 1, p. 10–17. ISSN 2299-1093. Available at: https://www.degruyter.com/document/doi/10.1515/comp-2018-0002/html.

[18] Veigend, P.; Šátek, V. and Nečasová, G. Effective solution of nonlinear DAEs problems using Taylor series method. De Gruyter Open Access, vol. 59, no. 1. ISSN 2391-4661. Available at: https://www.degruyterbrill.com/document/doi/10.1515/dema-2025-0216/html?lang=en.

[19] Čermák, L. and Hlavička, R. *Numerické metody*. VUT v Brně, Fakulta strojního inženýrství. ISBN 978-80-214-6322-6. Available at: https://hdl.handle.net/11012/250025.

Appendices

List of Appendices

A Digital attachments 81

B Specification of benchmarks CPU 82

C Specification of benchmarking GPU 84

Appendix A

Digital attachments

The digital attachments for this Thesis contain folders `code` with sources of implementation, `tex` with sources for this thesis with all figures, `Executables` with compiled binaries from the source code, `Inputs` containing two example inputs for the first experiments together with csv files for the incompressible Stokes problem and `Outputs` with the files that were captured during measurements, such as times and method results.

Appendix B

Specification of benchmarks CPU

```
> lscpu
Architecture:                x86_64
  CPU op-mode(s):            32-bit, 64-bit
  Address sizes:             48 bits physical, 48 bits virtual
  Byte Order:                Little Endian
CPU(s):                      16
  On-line CPU(s) list:       0-15
Vendor ID:                   AuthenticAMD
  Model name:                AMD Ryzen 7 9700X 8-Core Processor
    CPU family:              26
    Model:                   68
    Thread(s) per core:      2
    Core(s) per socket:      8
    Socket(s):               1
    Stepping:                0
    Frequency boost:         enabled
    CPU(s) scaling MHz:      61%
    CPU max MHz:             5582.3008
    CPU min MHz:             605.3100
    BogoMIPS:                7585.25
    Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep
   mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx
   mmxext fxsr_opt pdpe1gb rdtscp lm
                             constant_tsc rep_good amd_lbr_v2 nopl
   xtopology nonstop_tsc cpuid extd_apicid aperfmperf rapl pni pclmulqdq
   monitor ssse3 fma cx16 sse4_1 sse4_2
                             movbe popcnt aes xsave avx f16c rdrand
   lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3
   dnowprefetch osvw ibs skinit wdt tce top
                            oext perfctr_core perfctr_nb bpext perfctr_llc
    mwaitx cpuid_fault cpb cat_l3 cdp_l3 hw_pstate ssbd mba perfmon_v2
   ibrs ibpb stibp ibrs_enhanced v
```

```
                          mmcall fsgsbase tsc_adjust bmi1 avx2 smep bmi2
    erms invpcid cqm rdt_a avx512f avx512dq adx smap avx512ifma clflushopt
    clwb avx512cd sha_ni avx512
                          bw avx512vl xsaveopt xsavec xgetbv1 xsaves
   cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local user_shstk avx_vnni
   avx512_bf16 clzero irperf xsavee
                          rptr rdpru wbnoinvd cppc arat npt lbrv
   svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists
   pausefilter pfthreshold avic v_vmsave_vm
                          load vgif x2avic v_spec_ctrl vnmi avx512vbmi
   umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni
   avx512_bitalg avx512_vpopcntdq rdpid bu
                          s_lock_detect movdiri movdir64b overflow_recov
    succor smca fsrm avx512_vp2intersect flush_l1d amd_lbr_pmc_freeze
Virtualization features:
  Virtualization:         AMD-V
Caches (sum of all):
  L1d:                    384 KiB (8 instances)
  L1i:                    256 KiB (8 instances)
  L2:                     8 MiB (8 instances)
  L3:                     32 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-15
```

Appendix C

Specification of benchmarking GPU

```
=====================
HSA System Attributes
=====================
Runtime Version:         1.18
Runtime Ext Version:     1.15
System Timestamp Freq.:  1000.000000MHz
Sig. Max Wait Duration:  18446744073709551615 (0xFFFFFFFFFFFFFFFF) (
    timestamp count)
Machine Model:           LARGE
System Endianness:       LITTLE
Mwaitx:                  DISABLED
XNACK enabled:           NO
DMAbuf Support:          YES
VMM Support:             YES

==========
HSA Agents
==========
*******
Agent 2
*******
  Name:                    gfx1200
  Uuid:                    GPU-8e0a0c3cadb3f51b
  Marketing Name:          AMD Radeon Graphics
  Vendor Name:             AMD
  Feature:                 KERNEL_DISPATCH
  Profile:                 BASE_PROFILE
  Float Round Mode:        NEAR
  Max Queue Number:        128(0x80)
  Queue Min Size:          64(0x40)
  Queue Max Size:          131072(0x20000)
  Queue Type:              MULTI
```

```
Node:                    1
Device Type:             GPU
Cache Info:
  L1:                      32(0x20) KB
  L2:                      4096(0x1000) KB
  L3:                      32768(0x8000) KB
Chip ID:                 30096(0x7590)
ASIC Revision:           1(0x1)
Cacheline Size:          256(0x100)
Max Clock Freq. (MHz):   2700
BDFID:                   768
Internal Node ID:        1
Compute Unit:            32
SIMDs per CU:            2
Shader Engines:          2
Shader Arrs. per Eng.:   2
WatchPts on Addr. Ranges:4
Coherent Host Access:    FALSE
Memory Properties:
Features:                KERNEL_DISPATCH
Fast F16 Operation:      TRUE
Wavefront Size:          32(0x20)
Workgroup Max Size:      1024(0x400)
Workgroup Max Size per Dimension:
  x                        1024(0x400)
  y                        1024(0x400)
  z                        1024(0x400)
Max Waves Per CU:        32(0x20)
Max Work-item Per CU:    1024(0x400)
Grid Max Size:           4294967295(0xffffffff)
Grid Max Size per Dimension:
  x                        2147483647(0x7fffffff)
  y                        65535(0xffff)
  z                        65535(0xffff)
Max fbarriers/Workgrp:   32
Packet Processor uCode:: 178
SDMA engine uCode::      662
IOMMU Support::          None
Pool Info:
  Pool 1
    Segment:                 GLOBAL; FLAGS: COARSE GRAINED
    Size:                    16695296(0xfec000) KB
    Allocatable:             TRUE
    Alloc Granule:           4KB
    Alloc Recommended Granule:2048KB
    Alloc Alignment:         4KB
    Accessible by all:       FALSE
  Pool 2
```

```
      Segment:                 GROUP
      Size:                    64(0x40) KB
      Allocatable:             FALSE
      Alloc Granule:           0KB
      Alloc Recommended Granule:0KB
      Alloc Alignment:         0KB
      Accessible by all:       FALSE
  ISA Info:
    ISA 1
      Name:                    amdgcn-amd-amdhsa--gfx1200
      Machine Models:          HSA_MACHINE_MODEL_LARGE
      Profiles:                HSA_PROFILE_BASE
      Default Rounding Mode:   NEAR
      Default Rounding Mode:   NEAR
      Fast f16:                TRUE
      Workgroup Max Size:      1024(0x400)
      Workgroup Max Size per Dimension:
        x                        1024(0x400)
        y                        1024(0x400)
        z                        1024(0x400)
      Grid Max Size:           4294967295(0xffffffff)
      Grid Max Size per Dimension:
        x                        2147483647(0x7fffffff)
        y                        65535(0xffff)
        z                        65535(0xffff)
      FBarrier Max Size:       32
    ISA 2
      Name:                    amdgcn-amd-amdhsa--gfx12-generic
      Machine Models:          HSA_MACHINE_MODEL_LARGE
      Profiles:                HSA_PROFILE_BASE
      Default Rounding Mode:   NEAR
      Default Rounding Mode:   NEAR
      Fast f16:                TRUE
      Workgroup Max Size:      1024(0x400)
      Workgroup Max Size per Dimension:
        x                        1024(0x400)
        y                        1024(0x400)
        z                        1024(0x400)
      Grid Max Size:           4294967295(0xffffffff)
      Grid Max Size per Dimension:
        x                        2147483647(0x7fffffff)
        y                        65535(0xffff)
        z                        65535(0xffff)
      FBarrier Max Size:       32
*** Done ***
```