BACHELOR OF APPLIED MATHEMATICS

DELFT UNIVERSITY OF TECHNOLOGY

NUMERICAL ANALYSIS

Time Integration Parallel in Time

Author
Ariena BUDKO

Supervisors
DR. M. MOÏLER
IR. C.W.J LEMMENS

August 20, 2020

Abstract

In this paper the research behind the parallelization on the GPU of the time parallel time integration method Parareal. Firstly, the theory behind Parareal and its convergence theorems will be detailed. Then, two test models, the Lorenz system and Heat diffusion equation, will be introduced. Additionally, the derivation of the Forward Euler and Backward Euler methods for these problems will be discussed. Secondly, an overview of development in parallel programming will be given, with a focus on architecture, memory organization and GPU properties. Thirdly, the implementation of Parareal in Python using the CuPy library will be shown, including the Parareal convergence plots and the code profiling results. In the second half of the paper, there will be an outline of the improvements that were made for a better speedup of the Parareal implementation. A discussion on linear solvers and their efficiency in regard to matrix properties will be presented. Moreover, the reason why a different linear solver for the Heat diffusion equation was needed, than the one built into CuPy, will be explained. Furthermore, the creation of a separate linear solver based on the Thomas algoithm as a CUDA-kernel in Python will be shared. The construction of CuPy elementwise kernels for the Lorenz system will be described as well. Lastly, the speedup results for the CuPy built-in functions will be compared to the self-made kernels utilizing a self-derived speedup formula. A reflection on the implementation of Parareal in practice will conclude this paper.

Contents

1 Introduction: What is Parareal?

When faced with large-scale physical problems, like the prediction of the weather and ocean waves, or a very detailed small scale problem, like the simulation of a particle-trajectory, science-based models can lack in accuracy and efficiency. A multitude of solutions have been proposed. Sometimes, dividing a problem into smaller independent sub problems and solving them separately, but simultaneously, is the best approach to a big problem. This is usually achieved by parallel computing. The partitioning of problems into smaller ones to approximate the exact solution is not as trivial as one might think. Often a picture comes to mind of placing a spatial grid onto a large area. Unfortunately, not all problems are large in spatial sense. Sometimes one has to study a physical phenomenon for a long period of time. Splitting a period of time into smaller intervals does not seem like a good idea, at first at least. Time is perhaps the most sequential thing one can think of. Time moves into one direction and one direction only, every second comes after the last. Still, there is a mathematical algorithm that is able to parallelize a given problem in time sense. This algorithm is called Parareal. The Parareal algorithm was introduced as ”un schéma en temps $<$ pararéel $>$” by Lions et al. in 2001 [7]. To actually comprehend the Parareal algorithm in one go from a textual explanation might be a bit convoluted or confusing. That is why Figure 1 is presented to summarize how exactly the algorithm works in practice.

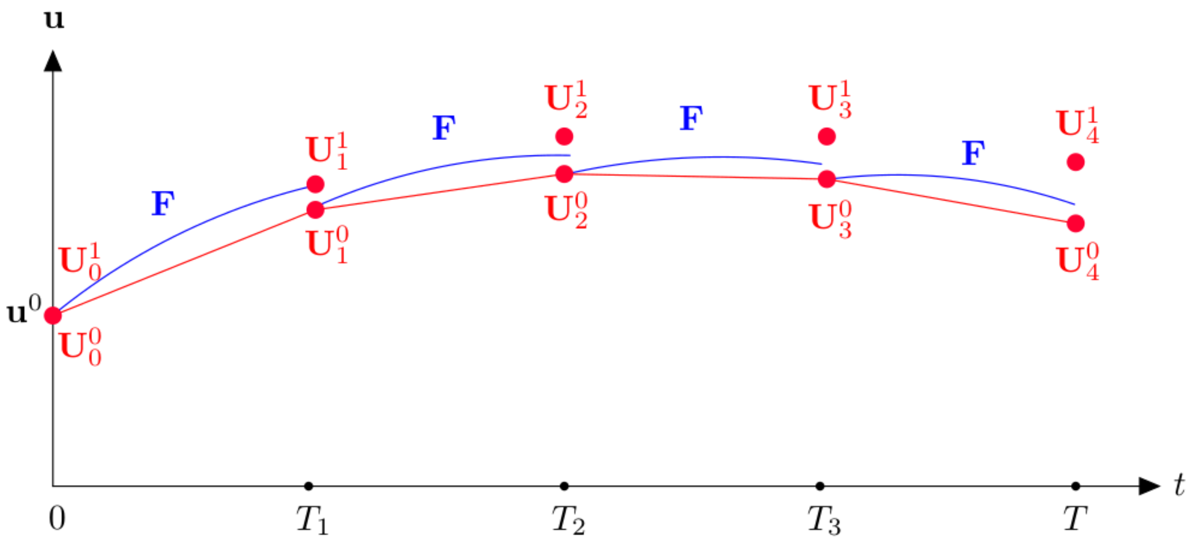


Figure 1: **A visual representation of the Parareal Algorithm [2]**

The baseline idea behind Parareal is the division of overall time into coarse and fine time intervals. From the given initial condition $\mathbf{u}_0$, the initial solution, also called the initial guess, is sequentially determined for the coarse intervals by using a coarse

mathematical solver. This is shown by the red line in Figure (1) Then, inside the subintervals, finer solutions are calculated by a fine mathematical solver (indicated with blue in Figure 1), using initial conditions from the initial guess. These fine solutions can be computed in parallel, as they are now independent from one another. Results from both the coarse and fine solver are utilized to update the final solution. This process is repeated in an iterative manner until satisfaction.
The concept of parallelizing time does probably not seem as foreign as before after hearing this explanation. Although Parareal is quite a new algorithm and therefore still has to be researched on its own, it has already been used on a couple of mathematical and physical problems. For example, Schöps et al. have applied the Parareal algorithm on the eddy current problem [14]. They did have to adapt Parareal to fit their need to incorporate non-conducting regions. In 2019, Waghamare et al. implemented Parareal to optimize the transport of nanoparticles in porous media [15]. Perhaps the most surprising development is the application of Parareal in machine learning. Meng et al. constructed a physics-informed neural network for time dependent PDEs, PPINN for short, in 2019 [9].

The applicability of the Parareal algorithm is very diverse, as can bee seen from the three examples given above. In this paper, Parareal will be tested on two different cases. These two science-based models will be thoroughly explored, one being the model for the Lorenz system and the other for the Heat diffusion equation. They are of particular interest as the Lorenz system consists of three Ordinary Differential Equations, ODEs for short, while the Heat diffusion equation is a one dimensional Partial Differential Equation (PDE). These models will later be used to show the workings and the effectiveness of the Parareal algorithm for these types of mathematical problems. Furthermore, we will explore if the theoretical success of Parareal can be translated into practice. In particular, we will look at existing tools in the parallel programming world, specifically for the GPU, and we will investigate what it takes to build an efficient implementation of Parareal.

1.1 Lorenz system

Firstly we will take a look at the model for the Lorenz system. The system has been developed by the American mathematician and meteorologist Edward N. Lorenz [4]. It consists of three ordinary differential equations. These three equations form a very simplified approximation of the Navier-Stokes equations. E.N. Lorenz constructed his model, while attempting to computationally simplify the Rayleigh-Bénard problem. The Rayleigh-Bénard problem concerns a fluid inside a container, where the bottom and top surfaces of said container have different temperatures. Processes such as atmospheric convection are modeled in this fashion.

In the previously mentioned conditions, the fluid is able to transition into three types of flow: stationary, steady or chaotic. The flow is called stationary, when the fluid does not move at all, whereas a steady flow has a constant and non-zero velocity in the entire fluid. However, for the construction of the mathematical model for the Lorenz system, we are only interested in chaotic flow, also known as chaotic mixing. A flow is deemed chaotic, if its behavior heavily depends on the initial conditions. This sensitivity causes complex patterns in flow direction, magnitude, velocity and circulation. A type of chaotic flow researched quite frequently is a turbulent flow[1]. Nonetheless, there are more kinds of chaotic flows.

Returning to the formulation of the Lorenz system, E.N. Lorenz described it as follows [8]:

$$
\begin{aligned}
\frac{dx(t)}{dt} &= \sigma(y(t) - x(t)),\\
\frac{dy(t)}{dt} &= \rho x(t) - y(t) - x(t)z(t),\\
\frac{dz(t)}{dt} &= x(t)y(t) - \beta z(t).
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
&t > 0, \qquad x(t),\ y(t),\ z(t) \in \mathbb{R}.\\
&t = 0, \qquad x(0) = x_0,\ y(0) = y_0,\ z(0) = z_0.
\end{aligned}
$$

where the whole system can be interpreted as the rate of change for multiple quantities with respect to time, namely temperature, density and velocity. Due to the over-simplified nature of this problem, some variables and parameters are difficult to connect to actual physical phenomena. In several works of literature $x(t)$ is considered proportional to the rate of convection, $y(t)$ to a horizontal temperature variation and $z(t)$ to a vertical one [4]. σ, ρ and β are parameters that determine temperature differences across the fluid and other fluid properties. Sometimes, σ and ρ are seen as synonymous to the Prandtl and Rayleigh number, respectively.
In our computational experiments, we choose $\sigma = 10$, $\beta = 8/3 \approx 2.7$ and $\rho = 28$. Furthermore x, y and z are from here on out to be considered as the coordinates of one particle in $\mathbb{R}^3$ over time t, with $t \in (0, 5]$. We also set $x_0 = 20$, $y_0 = 5$ and $z_0 = -5$ as our main initial condition at $t = 0$. To illustrate the behaviour of the particle over time for these initial conditions, Figure (2) has been plotted. A butterfly pattern can be seen. The trajectory of the particle will be vastly different if other slightly modified initial conditions will be chosen, due to the chaotic nature of the

[1]Planes experience turbulence in air while travelling across the Atlantic Ocean.

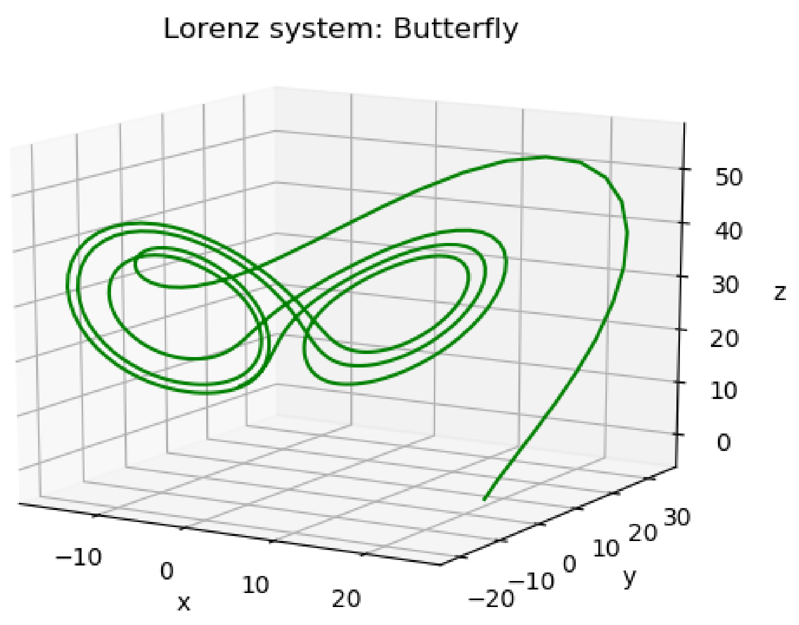


Figure 2: **Lorenz System in action, initial conditions** $x_0 = 20, y_0 = 5, z_0 = -5$

model.

Now, as we have the basics behind the Lorenz model under our belt, we can now proceed to numerically solve the system of ordinary differential equations in (1). Before we attempt this though, we will examine the second model, namely the heat diffusion problem.

1.2 Heat diffusion equation

The Heat diffusion equation or just the Heat equation is a type of PDE that illustrates the diffusion of temperature inside an object. This object can be one, two or three dimensional. The temperature diffusion depends on space and time. We choose a one-dimensional model to derive the Heat equation. This model can be seen as either that of a nail between two ice cubes or of a pipe with both ends in some fire. It does not really matter as long as the chosen rod is perfectly insulated, to ascertain the one-dimensionality.[2] To acquire an abstract formulation for the equation in question, we follow the steps from Haberman 2014 [5]. Thereafter we will modify that version by setting the proper initial and boundary conditions. On this final form the Parareal algorithm will eventually be applied.

Assume we have a rod with length L and cross-sectional surface area A. The rod is oriented in the x-direction, so the left and right boundary of the rod are chosen as

[2]The left and right bounds of the rod do have a cross-sectional area though, because designating them as points would make the future calculations impossible.

coordinates, $x = a$ and $x = b$ respectively. Two components are needed to construct the one-dimensional Heat equation: the conservation of heat energy and Fourier's law. We will take a look at the conservation of energy first.

If it is said that if the heat energy inside a object is conserved, the following is implied:

The rate of change of heat energy in time equals the heat energy flowing across the boundaries of the object per unit time plus the heat energy generated inside the object per unit time.

To rewrite this statement into an equation, we introduce a couple of concepts, namely heat energy, heat flux and heat sources. We define the thermal energy density $e(x,t)$ as the amount of thermal energy per unit volume. $\int_a^b e(x,t)Adx$ is then the total heat energy, the thermal energy density times the surface area integrated over the length of the rod. The heat flux $\phi(x,t)$ is the amount of thermal energy (per unit time) flowing from a to b per unit surface area. The heat energy flowing on the left boundary is therefore $\phi(a,t)A$ and $-\phi(b,t)A$ on the right. Internal sources of thermal energy (per unit volume generated per unit time) are denoted as $Q(x,t)$. Thus $\int_a^b Q(x,t)Adx$ is the total heat energy generated inside (per unit time). We combine all these these concepts into the statement above about the conservation of energy as follows:

$$\frac{d}{dt}\int_a^b e(x,t)Adx = \phi(a,t)A - \phi(b,t)A + \int_a^b QAdx, \tag{2}$$

where $\frac{d}{dt}\int_a^b e(x,t)Adx$ is the rate of change of heat energy in time. We quickly notice that A can be cancelled, as it is a constant. Moreover, with a and b being constants and $e(x,t)$ continuous (the rod does not have any gaps), it is possible to take the ordinary derivative d/dt inside the integral, turning it into a partial derivative. Recognize that $\phi(a,t) - \phi(b,t) = -\int_a^b \frac{\partial\phi}{\partial x}dx$. Expressing thermal density energy $e(x,t)$ through $c(x)$, specific heat[3], $\rho(x)$, mass density and $u(x,t)$, temparature, we get:

$$\int_a^b \left(c(x)\rho(x)\frac{\partial u(x,t)}{\partial t} + \frac{\partial\phi(x,t)}{\partial x} - Q(x,t)\right)dx = 0, \tag{3}$$

[3]c(x) is a material-dependant constant that fixes what amount of heat energy is required to raise the temperature of the material by one unit

where the integral is zero for arbitrary a and b, meaning that the area under the curve must be zero no matter what limit is taken, hence the integrand must be zero, which exactly concludes the (integral) conservation law of heat:

$$c\rho \frac{\partial u}{\partial t} = \frac{\partial \phi}{\partial x} + Q. \tag{4}$$

The second component for the construction of the Heat equation is Fourier's law of heat conduction, namely:

$$\phi(x,t) = -K_0 \frac{\partial u}{\partial x}, \tag{5}$$

where four properties of heat flow are incorporated into one equation.[4]. When substituting Fourier's law into (4), it results in the most abstract version of the one-dimensional Heat equation:

$$c\rho \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(K_0 \frac{\partial u}{\partial x}\right) + Q \tag{6}$$

To complete the derivation for our case, we assume the rod is uniform, so all the thermal coefficients do not depend on x anymore. Then setting $\frac{K_0}{c\rho} = 1$ and adding the initial and boundary conditions into the mix, we get:

$$\begin{aligned} &\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2} + Q(x,t). \\ &t = 0, \qquad u(x,0) = 0 \\ &t > 0, \qquad u(a,t) = 0, \qquad u(b,t) = 0, \end{aligned} \tag{7}$$

where $a = 0$ and $b = L$. The boundary conditions are called Dirichlet boundary conditions. The source function Q is defined as:

[4] K_0 is a material-linked constant indicating thermal conductivity.

$$Q(x,t) = x^4 \times (1-x)^4 + 10 \times \sin(8t). \tag{8}$$

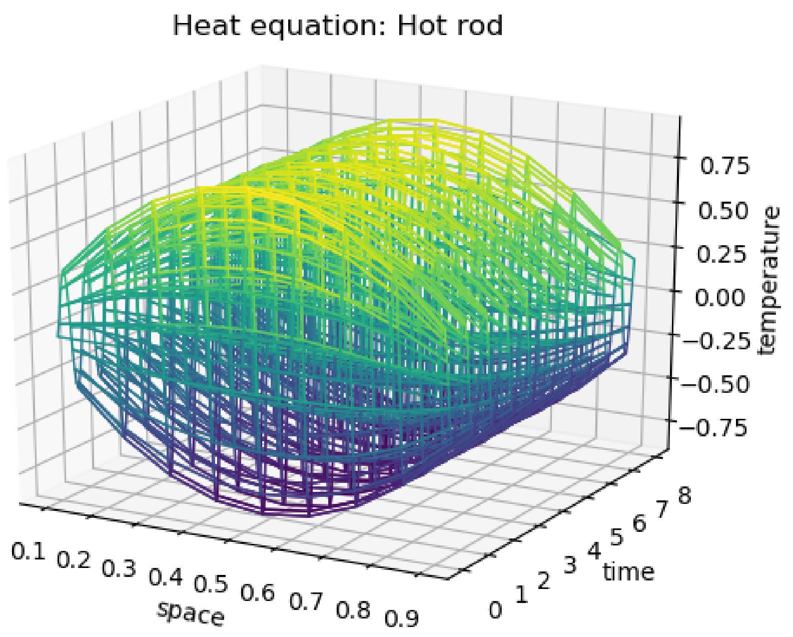


Figure 3: **Heat equation for Dirichlet boundary conditions**

Q acts as a pulsating source of heat on the entire rod. In Figure (3) the effect of the source and the solution to the heat equation in space and time is presented. The yellow indicates warmth, the blue indicates cold. Only the inner points are shown. In time the symmetric parabola goes up and down, due to the pulse.

To reduce the Heat equation PDE to a system of ODEs, the spatial discretization with the Finite difference method needs to be performed. For this, we need to construct a uniform spatial grid on the aforementioned rod. Let x_j be the spatial nodes of this grid. Then, the second order derivative can be approximated as follows:

$$\frac{\partial^2 u(x_j,t)}{\partial x^2} = \frac{u(x_{j-1},t) - 2u(x_j,t) + u(x_{j+1},t)}{h^2} + \mathcal{O}(h^2), \tag{9}$$

where $h = x_{j+1} - x_j$. Introducing the vector $\mathbf{u}(t)$ with components $u(x_j,t)$ and the finite-difference matrix A the PDE is reduced to the following system of ODEs:

$$\frac{d\mathbf{u}}{dt} = A\mathbf{u} + \mathbf{q}, \tag{10}$$

where the source vector $\mathbf{q}$ has components $Q(x_j, t)$. The matrix A has the structure:

$$A = \frac{1}{h^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & & \ddots & & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix} \tag{11}$$

1.3 Mathematical details

In the previous section the Lorenz System and the Heat Equation were introduced and their origin and derivation summarized. The expected solution was also presented graphically, for the established initial and boundary conditions over a given period of time. The plots shown were not created by using experimental data or by determining an exact solution. On the contrary, they are approximations of the real solution. The approximate solutions were made by applying a time integration method on the problems, specifically the Forward-Euler and the Backward-Euler methods on the Lorenz System and Heat equation, respectively. In this part, we will delve into the mathematics behind time integration methods. Specifically, we will compare the Forward-Euler and Backward-Euler method in terms of order, convergence and stability. Most importantly though, there will be an explanation on the inner workings of the Parareal method and why mathematically speaking it converges to the desired solution for both the Lorenz System and the Heat Equation

1.3.1 Sequential time integration methods

Both the Lorenz system and the Heat equation have been reduced to a general system of M ODEs of the form:

$$\begin{gathered} \frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u}, t). \\ \\ \begin{aligned} 0 < t < T, &\qquad \mathbf{u}(t) \in \mathbb{R}^M. \\ t = 0, &\qquad \mathbf{u}(t_0) \text{ given.} \end{aligned} \end{gathered} \tag{12}$$

where f is a non-linear function in the case of the Lorenz system and linear for the Heat equation. We want to obtain a numerical approximation of the solution of these ODEs in a number of points t_n for which $0 \leq t_n \leq T$ holds. This process is called time integration.

The Forward-Euler method, which will be used with the non-linear Lorenz system, is defined in the following fashion:

$$\begin{aligned}&\mathbf{u}(t_0) \text{ given}\\&\mathbf{u}(t_{n+1}) = \mathbf{u}(t_n)) + \Delta t\,\mathbf{f}(\mathbf{u}(t_n), t_n) + \mathcal{O}(\Delta t)\end{aligned} \tag{13}$$

where $\Delta t = t_{n+1} - t_n$. The Forward-Euler is an explicit method. On the linear Heat equation the Backward-Euler method will be applied:

$$\begin{aligned}\mathbf{u}(t_0) &\text{ given}\\\mathbf{u}(t_{n+1}) &= \mathbf{u}(t_n) + \Delta t\,\mathbf{f}(\mathbf{u}(t_{n+1}), t_{n+1}) + \mathcal{O}(\Delta t)\\&= \mathbf{u}(t_n) + \Delta t\,A\mathbf{u}(t_{n+1}) + \Delta t\,\mathbf{q}(t_{n+1}) + \mathcal{O}(\Delta t),\end{aligned} \tag{14}$$

which is an implicit method. To obtain the approximation at the next time step, one needs to solve a linear system, namely:

$$\mathbf{u}(t_{n+1}) = [I - \Delta t\,A]^{-1}\left(\mathbf{u}(t_n) + \Delta t\,\mathbf{q}(t_{n+1})\right) \tag{15}$$

Both methods are numerically of the order Δt and convergent to the exact solution as $\Delta \leftarrow 0$. Furthermore, the Backward-Euler method is unconditionally stable, which allows to take larger time steps Δt. Both methods are by nature sequential, because to know the next step, the previous step has to have been calculated. For a large interval with a lot of time steps, this is very computationally intensive. Parareal tries to circumvent this problem by partitioning the total time interval into coarse subintervals and repetitively computing solutions on these subintervals in parallel. We are now ready to explore the Parareal algorithm in a more detailed mathemtical sense.

1.3.2 Parareal

There are a couple of different approaches to derive and construct the Parareal algorithm. We will be focusing on one in particular, namely the step by step formulation given by M.J Gander and S. Vandewalle in 2007 [3]. This approach is based on the so-called multiple shooting method and takes an abstract system of non-linear ODEs into account. The original paper on Parareal, Lions 2001 [7], describes a derivation of Parareal on a system of linear scalar ODEs, which is not applicable on the Lorenz system nor the Heat equation. Fortunately, the Lorenz system is exactly a non-linear system of ODEs like in Gander and Vandewalle's article. So Parareal will be built in this section following the steps of [3].

However, let's first give a clear and abstract definition of Parareal for a general scenario, combining the definitions given in [3] and Gander's lecture notes from 2018 [2]. Let there be a system of M Ordinary Differential Equations, like in (12) where the system's solution needs to be determined over a period of time $[0, T]$.

To define Parareal on this case properly, we have to introduce two propagation operators, $\mathcal{G}$ and $\mathcal{F}$. Roughly speaking, these operators compute a coarse and fine approximation of a solution for a chosen point in time from a fixed initial condition. More precisely, we set $(t_1, t_2) \in [0, T]$ a time interval with $\mathbf{u}(t_1)$ being the M-dimensional initial condition and $\mathbf{u}(t_2)$ the solution to be approximated. Then $\mathcal{G}(t_2, t_1, \mathbf{u}(t_1))$ gives a coarse approximation of $\mathbf{u}(t_2)$, an approximation resulting from taking coarse time steps inside the time interval (t_1, t_2) to arrive at the end result. $\mathcal{F}(t_2, t_1, \mathbf{u}(t_1))$, on the other hand, gives a fine approximation of $\mathbf{u}(t_2)$, as the time interval (t_1, t_2) is divided into finer time steps to produce the final solution. This idea can be put into an algorithm.

Let $n = 0, 1, \dots, N$. We set the total time interval $[t_0, t_N]$ for this. We divide $[t_0, t_N]$ into $N + 1$ points in time (or N intervals), creating $(t_0, t_1, \dots t_N)$ accordingly. We start by defining an initial approximation at a given time t_n, we call it $\mathbf{U}_0(t_n) \in \mathbb{R}^M$.[5] This initial approximation can be computed sequentially as follows:

$$\mathbf{U}_0(t_{n+1}) = \mathcal{G}(t_{n+1}, t_n, \mathbf{U}_0(t_n)), \tag{16}$$

where $\mathbf{U}_0(t_0) = \mathbf{u}(t_0)$. From there, the algorithm performs the *correction iteration*, for $k = 1, 2, \dots$ iterations, in the following way:

$$\mathbf{U}_{k+1}(t_{n+1}) = \mathcal{G}(t_{n+1}, t_n, \mathbf{U}_{k+1}(t_n)) + \mathcal{F}(t_{n+1}, t_n, \mathbf{U}_k(t_n)) - \mathcal{G}(t_{n+1}, t_n, \mathbf{U}_k(t_n)). \tag{17}$$

Observing (17) carefully, it becomes apparent that for $k \to \infty$ $\mathcal{G}$-propagators will cancel each other out in (17). This leaves $\mathbf{U}(t_{n+1}) = \mathcal{F}(t_{n+1}, t_n, \mathbf{U})$. What this im-

[5]The notation $U_0(t_n)$ is different from the standard U_n^0 used in literature about Parareal, such as [2], [3] or [7], but our version is arguably more understandable.

plies, is that a series of values $\mathbf{U}(\mathbf{t_n})$ will be generated that satisfy this equality. In other words, when k tends to ∞, approximations of $\mathbf{U}(t_n)$ will have the accuracy as if the $\mathcal{F}$-propagator has been applied.

According to Gander and Vandewalle 2007 [3], the Parareal algorithm is closely comparable to a so-called classical deferred correction method. This means that the method, when applied to a difficult problem $A(\mathbf{u}) = \mathbf{0}$, by performing iterative computations, produces the desired solution after solving easier problems $B(\mathbf{u}) = \mathbf{g}$. Mathematically speaking, we have the following iteration:

$$B(\mathbf{u}_{k+1}) = B(\mathbf{u}_k) - A(\mathbf{u}_k), \qquad k = 0, 1, \ldots, \tag{18}$$

where $\mathbf{u}_0$ is a pre-determined approximation for $\mathbf{u}$. To connect this abstract deferred correction method to the correction iteration in (17), we have to set $\mathbf{u} = (\mathbf{U}(t_1), \mathbf{U}(t_2), \ldots, \mathbf{U}(t_N))^T \in \mathbb{R}^{NM}$ and set $A(\mathbf{u})$ and $B(\mathbf{u})$ to be vectors of length N times M. Then we can define for $n = 0, \ldots, N-1$:

$$\begin{aligned} A(\mathbf{u}(t_{n+1})) &= \mathbf{U}(t_{n+1}) - \mathcal{F}(t_{n+1}, t_n, \mathbf{U}(t_n)) \\ B(\mathbf{u}(t_{n+1})) &= \mathbf{U}(t_{n+1}) - \mathcal{G}(t_{n+1}, t_n, \mathbf{U}(t_n)) \end{aligned} \tag{19}$$

where $A(\mathbf{u}(t_{n+1}))$ and $B(\mathbf{u}(t_{n+1}))$ are vectors, both of length M, that are part of $A(\mathbf{u})$ and $B(\mathbf{u})$ respectively. These vectors correspond to the coarse interval $n+1$.

Now, to return to the construction of Parareal, Gander and Vandewalle (2007) base their unified derivation of the algorithm on the application of the multiple shooting method to (12). Let the total interval $[0, T]$ and divide it into N subintervals, defined by points in time, so $0 < t_0 < t_1 < \cdots < t_{N-1} < t_N = T$ is true. Then, the following N initial value problems are posed in one system of equations:

$$\begin{cases} \frac{d\mathbf{u}_0}{dt} &= \mathbf{f}(\mathbf{u}_0), \qquad \mathbf{u}_0(t_0) = \mathbf{U}(t_0), \\ \frac{d\mathbf{u}_1}{dt} &= \mathbf{f}(\mathbf{u}_1), \qquad \mathbf{u}_1(t_1) = \mathbf{U}(t_1), \\ &\vdots \\ \frac{d\mathbf{u}_{N-1}}{dt} &= \mathbf{f}(\mathbf{u}_{N-1}), \qquad \mathbf{u}_{N-1}(t_{N-1}) = \mathbf{U}(t_{N-1}), \end{cases} \tag{20}$$

where the following extra conditions are added to enforce continuity between the separate initial value problems:

$$\begin{aligned}
\mathbf{U}(t_0) - \mathbf{u}(t_0) &= 0, \\
\mathbf{U}(t_1) - \mathbf{u}_0(t_1, \mathbf{U}(t_0)) &= 0, \\
&\vdots \\
\mathbf{U}(t_N) - \mathbf{u}_{N-1}(t_N, \mathbf{t_N}) &= 0.
\end{aligned} \tag{21}$$

The notation $\mathbf{U}(t_1) - \mathbf{u}_0(t_1, \mathbf{U}(t_0)) = 0$ indicates that the solution of initial problem $\mathbf{u_0}$ at time t_1 with initial condition $\mathbf{U}(\mathbf{t_0})$ is equal to the approximation $\mathbf{U}(\mathbf{t_1})$, so at the end of the subinterval corresponding problem $\mathbf{u_0}$, etc. These extra conditions, often referred to as matching conditions, can together form another, although non-linear, system of equations, shortly notated as:

$$\mathbf{F}(\mathbf{U}) = \mathbf{0}, \qquad \mathbf{U} = (\mathbf{U}(t_0), \mathbf{U}(t_1), \ldots, \mathbf{U}(t_N))^T \tag{22}$$

This system can be solved using the Newton-Raphson method for non-lineaer systems. The Newton-Raphson is a common iterative root-finding method in numerical analysis. Each iteration the inverse of the Jacobian of F at the previous approximation is calculated, i.e, if $k = 0, \ldots, K$ with K the number of iterations, then:

$$\mathbf{U}_{k+1} = \mathbf{U}_k - J^-1_F(\mathbf{U}_k)\mathbf{F}(\mathbf{U}_{_k}) \tag{23}$$

To express $J^-1_F(\mathbf{U}_k)\mathbf{F}(\mathbf{U}_{_k})$, the Newton-Raphson update, we utilize the structure of the matching conditions in (21) to get:

$$J_F^{-1}(\mathbf{U}_k) =$$
$$\begin{bmatrix} I & & & & \\ -\frac{\partial \mathbf{u}_0}{\partial \mathbf{U}(t_0)}(t_1, \mathbf{U}_k(t_0)) & I & & & \\ & -\frac{\partial \mathbf{u}_1}{\partial \mathbf{U}(t_1)}(t_2, \mathbf{U}_k(t_1)) & I & & \\ & & \ddots & \ddots & \\ & & & -\frac{\partial \mathbf{u}_{N-1}}{\partial \mathbf{U}(t_{N-1})}(t_N, \mathbf{U}_k(t_{N-1})) & I \end{bmatrix}^{-1}$$

$$\mathbf{F}(\mathbf{U}_k) = \begin{bmatrix} \mathbf{U}_k(t_0) - \mathbf{u}_0 \\ \mathbf{U}(t_1) - \mathbf{u}_0(t_1, \mathbf{U}(t_0)) \\ \mathbf{U}(t_N) - \mathbf{u}_{N-1}(t_N, \mathbf{t_N}) \end{bmatrix} \tag{24}$$

After multiplying the the iteration step in (23) by the Jacobian on the left on both sides, we see the following recurrence emerging:

$$\begin{cases} \mathbf{U}_{k+1}(t_0) & = \mathbf{u}_0, \\ \mathbf{U}_{k+1}(t_{n+1}) & = \mathbf{u}_n(t_{n+1}, \mathbf{U}_k(t_n)) + \frac{\partial \mathbf{u}_n}{\partial \mathbf{U}(t_n)}(t_{n+1}, U_k(t_n))(\mathbf{U}_{k+1}(t_n) - \mathbf{U}_k(t_n)) \end{cases} \tag{25}$$

To translate this back to the formulation of the Parareal algorithm, we have to take into account that this recurrence formulation is continuous. The differential equations need to be discritized on every subinterval to be able to use the multiple shooting method in practice. In particular, to calculate $\mathbf{u}_n(t_{n+1}, \mathbf{U}_k(t_n))$, a numerical method has to be picked, like Forward or Backward Euler. To approximate the solution on subintervals, we indeed can choose that:

$$\mathbf{u}_n(t_{n+1}, U_k(t_n)) = \mathcal{F}(t_{n+1}, t_n, \mathbf{U}_{k+1}(t_n)), \tag{26}$$

with $\mathcal{F}$ being the $\mathcal{F}$-propagator in the Parareal algorithm that was defined earlier. In the same manner, we can approximate the second part of the right-hand side of (25) by applying the finite difference method with the $\mathcal{G}$-propagator incorporated. Saving the reader from some trivial derivations, we present the result immediately:

$$\begin{aligned}\frac{\partial \mathbf{u}_n}{\partial \mathbf{U}(t_n)}(t_{n+1}, U_k(t_n))(\mathbf{U}_{k+1}(t_n) - \mathbf{U}_k(t_n)) = \\ G(t_{n+1}, t_n, \mathbf{U}_{k+1}(t_n)) - G(t_{n+1}, t_n, \mathbf{U}_k(t_n))\end{aligned} \tag{27}$$

Thus, we conclude the entire construction of the Parareal method. The only thing to consider now is: Does the Parareal method really work? To answer this question mathematically, we have to look at the convergence of the method.

1.4 Convergence theorems for Parareal

There are two important results about the convergence that we need to mention before we can take the Parareal method for granted. The first one is a theorem:

Theorem 1 *The Parareal algorithm has the following property:*

$$\mathbf{U}_k(t_n) = \mathbf{F}(T_n, 0, \mathbf{u}_0) \qquad \textit{if} \qquad k \geq n,$$

in other words, $U_k(t_n)$ coincides with the fine approximation from iteration index $k = n$ onward.

The proof of this theorem is by induction and can be found in [2]. The implication of this theorem is that if the number of iterations is equal to the number of coarse time steps, then the Parareal algorithm will have definitely converged. The take-away is though, that it is not very desirable to have a problem for which Parareal happens to converge for $k = n$. That would be computationally equivalent to using $\mathcal{F}(T_n, 0, \mathbf{u}_0)$ on the entire problem, if not worse because of the correction iteration steps. It is therefore more interesting to know the rate of convergence. The rate of convergence of Parareal is still an open research question, because the rate is heavily dependent on the type of problem. For a simple linear ODE:

$$\frac{du}{dt} = au, \qquad u(0) = 0, \qquad t \in [0, T] \qquad \text{with} \qquad a \in \mathbb{R}, \tag{28}$$

a theorem about convergence rate can be formulated:

Theorem 2 *Let $\Delta t = T/N$, $t_n = n\Delta t$ for $n = 0, 1, \ldots, N$. Let $F(t_{n+1}, t_n, U_{k+1}(t_n))$ be the exact solution at t_{n+1} of 28 with $u(t_n) = U_k(t_n)$, and let*

$G(t_{n+1}, t_n, U_{k+1}(t_n))$ be the corresponding Backward Euler approximation with time step Δt. Then:

$$\max_{1\leq n\leq N} |u(t_n) - U_k(t_n)| \leq C_k \Delta T^{k+1}. \tag{29}$$

We observe that the convergence rate is exponential in k. This is very fast and will be confirmed in numerical experiments for the Heat equation later in this paper. See [2] and the references therein for proof and the extension of this theorem for the Heat equation by the Fourier transform.

2 Parallel programming

At this point, we have a grasp of the theory behind the Parareal algorithm and why it is very compelling for grand-scale mathematical problems. It converges to the desired solution and it promises to be more efficient in the process. Dividing problems into lots of parts and computing them simultaneously seems like a great concept. Nothing can go wrong, right? No, there is one big catch: the computer itself. Although the computer is endless fun, it is not endless in capacity, memory and available tools. In this section, an overview will be provided of things to consider while attempting parallel programming. This section is greatly inspired by Rauber and Rünger 2012, [11] and [12].

2.1 Architectures

One of the crucial internal parts of any computer is the processor chip. This chip, on a base level, consists of transistors. Two properties of this processor chip determine the quality of the overall performance: the clock frequency and the number of transistors. The clock frequency is, roughly speaking, the amount of time it takes to execute one simple instruction or task. The number of transistors improves the complexity of the chip, and thus the performance as well.[6] As the reduction in clock frequency has stagnated in recent years, the increase in number of transistors plays the biggest part in the advancement of computer architecture and causes the most shrinkage in total execution time of instructions. To make the best of the improvements in the complexity of computer architecture, parallelism comes into play.

[6]Moore's law, an observation that the number of transistors on an average processor chip doubles every 18 to 24 months, has been true for at least 40 years. That is why the performance of computers still keeps improving so much in our lifetime.

Parallelism has been applied to four different aspects of processor design, namely at bit level, by pipelining, by multiple functional units and at process or thread level. We will focus on the last one, which is the most relevant to this paper. The other three techniques all have in common that they only require one sequential control flow.[7]. They specifically have more to do with the change of execution order depending on interconnectedness between different instructions and tasks. This results in a more artificial type of parallelism that can be achieved by simple compilation in sequential programming languages. These techniques have reached their limits some time ago though, but transistor numbers keep increasing, just like Gordon Moore has predicted in 1965. This is a great feature (not a bug), since there are enough transistors that can be grouped into multiple processor cores and placed on one single processor chip. These multicore processors, or multiprocessors for short, are a standard part of everybody's average computer for the last 15 years. This means that most computers support some form of parallelism, although in different ways. Flynn's taxonomy [11] is a classification list of four main types of parallel computers, namely: SISD, MISD, SIMD and MIMD. The first two letters stand for either Single or Multiple Instruction(s), while the last two letters indicate if there is access to Single or Multiple Data. The definitions speak for themselves mostly, so we will briefly mention MIMD architecture, because that is commonly part of parallel computers.

When the architecture is Multiple Instruction, Multiple Data it means that there are multiple processing units that have access to (shared or distributed[8]) memory. The multiple processing elements are asynchronous. Hence, they can separately load an instruction and retrieve data, manipulate said data by the instruction and store it again, without getting in each others way. A big piece missing in this explanation though, is how the memory is actually physically stored. In Section 2.2, the two main versions of memory organization will be outlined.

2.2 Memory Organization

There are two sub classes of MIMD computers based on their memory organization. On one hand there are multicomputer systems, these are computers with distributed memory. On the other hand, multiprocessor systems are computers with shared memory. What is the difference exactly and how does it affect the way you program on such a machine?

[7]Control flow is a term from computer science that describes the order in which individual instructions or tasks are executed

[8]More on this in (2.2)

2.2.1 Distributed memory

Computers with distributed memory have a certain amount of processing elements or otherwise called nodes. These nodes are connected by an interconnection network. This network makes it possible for data transfer between nodes to happen. One node has its own processor P and local memory M, see Figure (4). Usually, in the local memory of one or a couple processors the program data is stored. Local memory can only be accessed by a corresponding local processor. Nodes need to request memory data of other nodes by passing messages through the interconnection network. When programming a machine with distributed memory, the parallel programming language makes use of processes. A process has an executable program and all the information necessary for execution. For every process there is an exclusive address space, so only the process can asses its own data. Data exchange has to be done through explicit communication. MPI is a parallel programming language that supports this type of process programming.

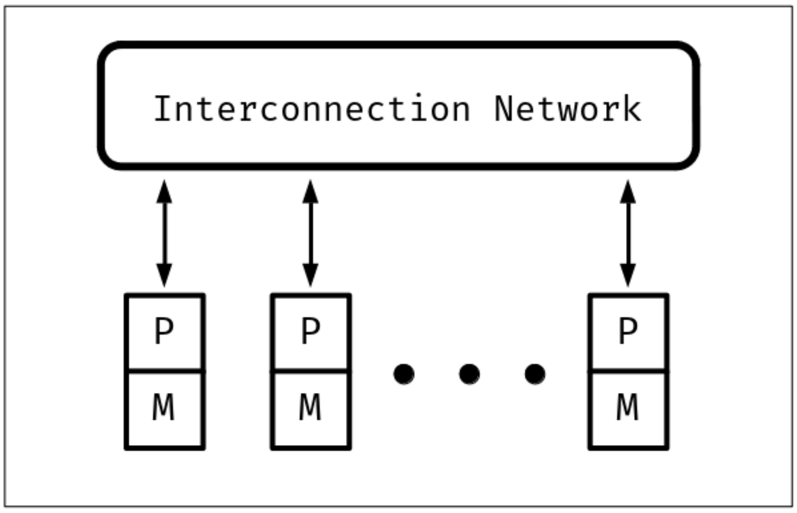


Figure 4: **Illustration of an abstract computer with distributed memory**

2.2.2 Shared memory

If the computer has a shared or global memory on physical level, then it contains multiple processors or cores that are all connected to one shared memory through an interconnection network, see Figure (5). The shared memory can be partitioned in memory modules with common address space that are accessible to all the processors. Exchange of data and communication between processors happens by reading and modifying shared variables in the memory. This has a catch though: If multiple processors try to access the same shared variable simultaneously, race conditions start to arise. Programming for a shared memory type of machine will utilise threads.

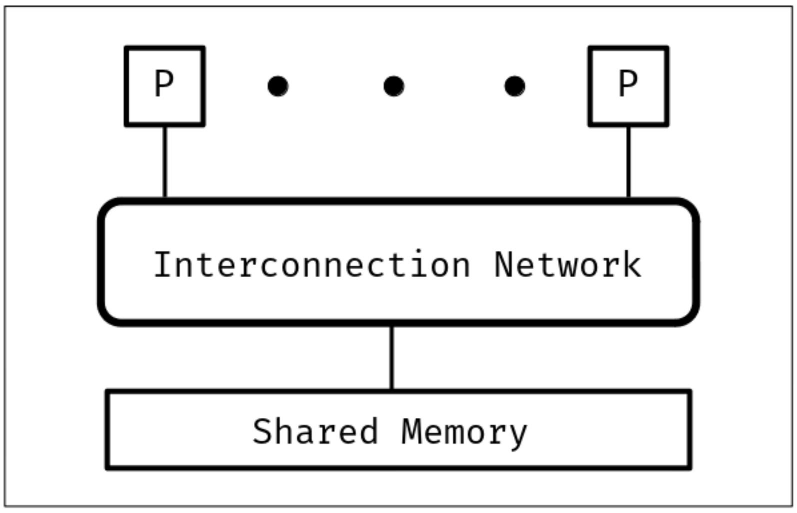


Figure 5: **Illustration of an abstract computer with shared memory**

Threads are multiple independent control flows that are part of one process. Threads thus share the same address space as their process. Threads can store data in their shared address space and all threads then can access it. Programming with threads is more common due to its flexibility. Example of a programming language that uses threads are OpenMP.

2.3 CPU versus GPU

The previous sections discussed the architecture and memory organization of the CPU (Central Processing Unit) inside the computer and how to create parallelism on there. However, another part of the computer can be utilized for parallel programming, the Graphics Processing Unit, the GPU for short [12]. The main purpose of the GPU was previously to process enormous data behind graphics applications. The particular design of the GPU hardware makes it also a desirable tool for general parallel programming and can be more powerful, as it has the capacity for a large amount of cores that can run many threads. For the research in this paper, we will partake in GPU programming, as it has a lot of computing potential and promises of being significantly faster than the CPU for certain types of problems. We have to look at the GPU architecture to understand this.

2.4 GPU architecture

In Figure (6) we see a simplified logical architecture of a GPU. The blue area is a grid containing two blocks. Each block can at maximum have 1024 threads for the type of GPU we are working with (See Section 2.5). Inside a block 48 kilobytes of block

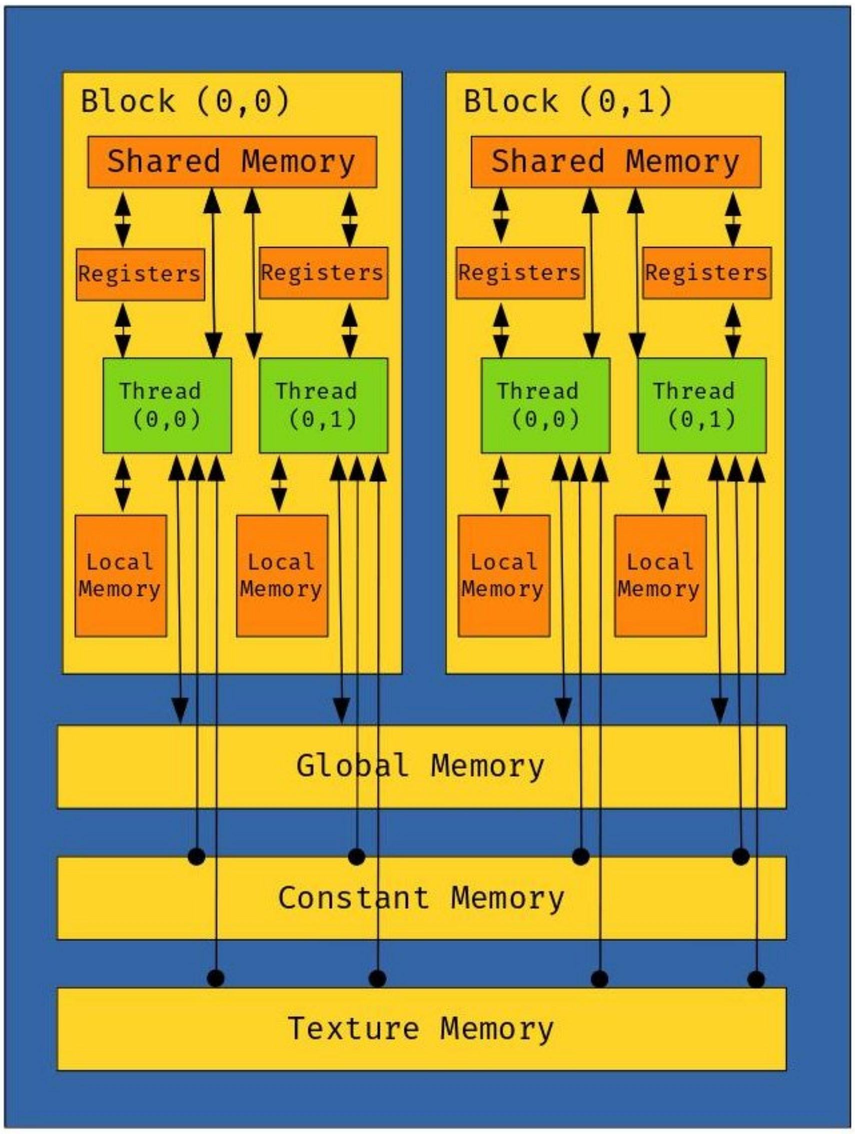


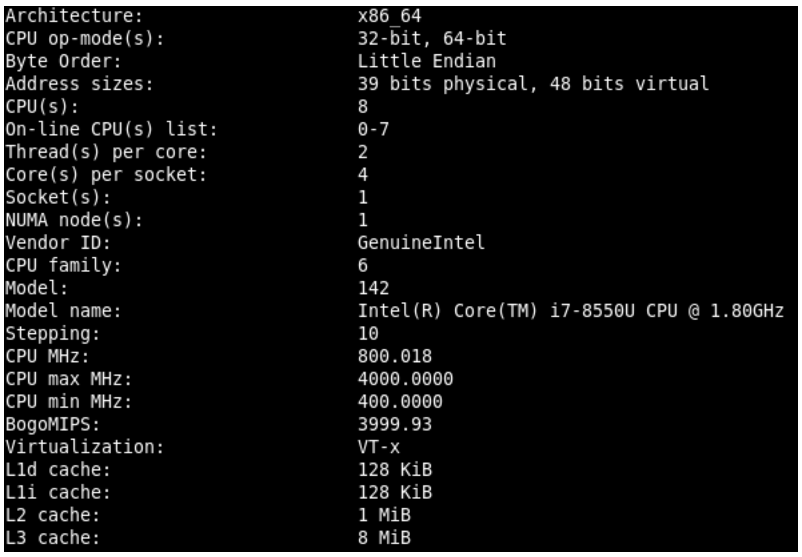
Figure 6: **Illustration of a standard GPU architecture**

shared memory can be stored, that is accessible by the threads and registers, of which there are around 64 kilobytes. Registers are connected to one thread and contain all the local variables and specific information the thread requires. Each thread has a separate local memory or cache with its own address space for data that does not fit into registers. A cache is a small memory space with data readily available for the thread. Outside the blocks there are three types of shared memory: global, constant and texture. The global memory is the slowest and not cached, all the threads have access to it. The constant memory is cashed, connected to all threads, but is read only. Lastly the texture memory is a cashed memory also connected to all threads. It is optimized for 2-dimensional access, read only. and shared by all processors.

The problems that have the most benefit from being programmed on GPU are those that can be implemented as a large number of lightweight threads without too much communication between them. The calculation of the fine solution in the Parareal algorithm is such a problem.

2.5 Tools and laptop specifics

For the research in this paper, one laptop with POP OS (19.04) operating system was utilized. Some specifics of this laptop for the CPU and GPU are listed in Figure (7) and (8).

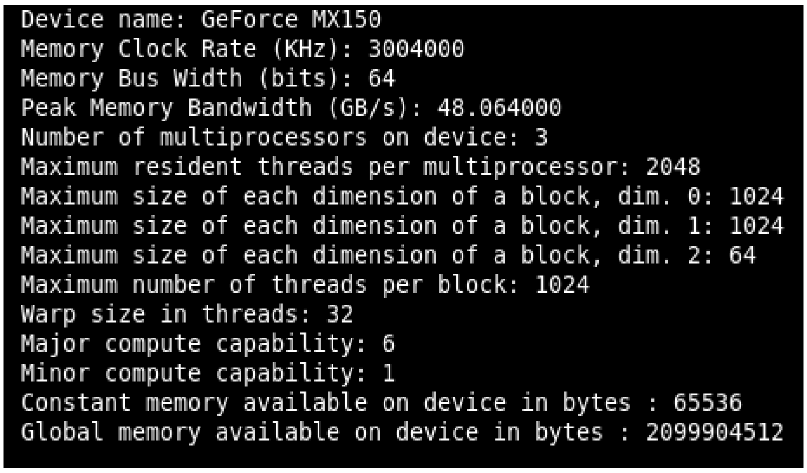


```
Architecture:                x86_64
CPU op-mode(s):              32-bit, 64-bit
Byte Order:                  Little Endian
Address sizes:               39 bits physical, 48 bits virtual
CPU(s):                      8
On-line CPU(s) list:         0-7
Thread(s) per core:          2
Core(s) per socket:          4
Socket(s):                   1
NUMA node(s):                1
Vendor ID:                   GenuineIntel
CPU family:                  6
Model:                       142
Model name:                  Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Stepping:                    10
CPU MHz:                     800.018
CPU max MHz:                 4000.0000
CPU min MHz:                 400.0000
BogoMIPS:                    3999.93
Virtualization:              VT-x
L1d cache:                   128 KiB
L1i cache:                   128 KiB
L2 cache:                    1 MiB
L3 cache:                    8 MiB
```

Figure 7: **CPU specifics**

For Nvidia graphics cards, CUDA was developed. According to the Nvidia website: *"CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.In GPU-accelerated applications, the sequential part of the workload runs on the CPU - which is optimized for single-threaded performance - while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords."*

Specifically, we will be using CUDA with Python, because there has been an interesting module developed for Python, called CuPy. On the CuPy website this is said about it: *"CuPy is an open-source array library accelerated with NVIDIA CUDA. CuPy*



```
Device name: GeForce MX150
Memory Clock Rate (KHz): 3004000
Memory Bus Width (bits): 64
Peak Memory Bandwidth (GB/s): 48.064000
Number of multiprocessors on device: 3
Maximum resident threads per multiprocessor: 2048
Maximum size of each dimension of a block, dim. 0: 1024
Maximum size of each dimension of a block, dim. 1: 1024
Maximum size of each dimension of a block, dim. 2: 64
Maximum number of threads per block: 1024
Warp size in threads: 32
Major compute capability: 6
Minor compute capability: 1
Constant memory available on device in bytes : 65536
Global memory available on device in bytes : 2099904512
```

Figure 8: **GPU specifics**

provides GPU accelerated computing with Python. CuPy uses CUDA-related libraries including cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL to make full use of the GPU architecture."

Apart from ease of installation and the compatibility with NumPy, Cupy allows writing custom made CUDA-kernels in the C-language as well. This possibility will be exploited later to optimize the Parareal code.

2.5.1 Unified memory

Before we move on to the implementation of Parareal in CuPy, we have to mention unified memory, which is a great fairly recent approach to GPU programming. Usually communication between Host (CPU) and Device (GPU) is the biggest bottleneck in programs, as data transfer is very slow between the two. Unified memory is a single memory address space where programs can allocate data and where both codes running on CPU and GPU codes can access it [1]. This implies that no special designation for memory transfer between Host (CPU) and Device (GPU) is required inside the code, making programming a breeze. For codes running on CPU or GPU that access data allocated in this fashion, the parallel-programming language manages to migrate memory pages to the memory of the respective processor.
However, old-fashioned explicit data transfers and memory allocations in CUDA, for example cudaMemcpyAsync, often make code more efficient due to carefully tuned asynchronous execution and data transfers.

3 Implementation of Parareal in Python

In this section the first intuitive implementation of Parareal in Python will be illustrated. The lecture notes of Gander (2018) contain multiple descriptions of sequential MATLAB programs that simulate the Parareal algorithm on a couple of problems [2]. These programs are not a true parallel implementations. Instead, they are used to show the rate of convergence of Parareal. We will reproduce the convergence results in Python for the Lorenz system and Heat equation. This will also be done sequentially utilizing the Numpy library in Python. Afterwards, by applying the Cupy library, the sequential programs will be altered into parallel programs. At this stage excerpts of the code will not be showcased, as the focus will be on the (convergence) results. The reader should take for granted (for now) that the change from Numpy to Cupy arrays and functions went smoothly. The actual performance of the parallel version compared to the sequential one is another story that will be discussed shortly. The entire Python code can be found in Appendices A and B.

3.1 Lorenz system

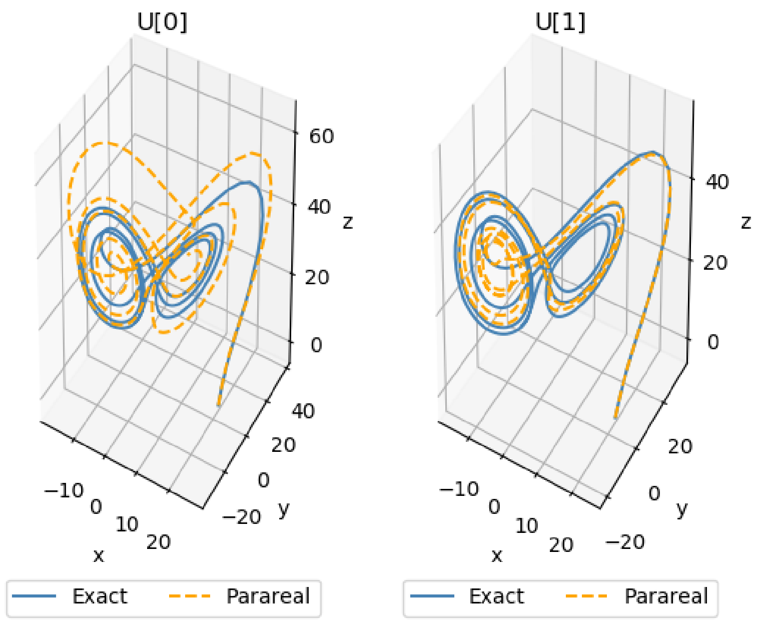


Figure 9: **Parareal-convergence for the Lorenz system, iterations $k = 1$ and $k = 2$**

To simulate the Lorenz system, the same assumptions were made as in Section 1.1, so the initial conditions are `x_0 = 20`, `y_0 = 5` and `z_0 = -5` with the parameters being σ `= 10`, ρ`= 28` and β `= 8/3`. Some new information about the constants for the Parareal algorithm is needed. We set the number of Parareal iterations to be `K = 20`. The total time interval is `T =` $[0, 5]$. The amount of coarse time steps is chosen to be `Mc = 500`, while the number of fine steps for the total time interval are `Mf =`

`5000`, meaning that there are `Mfc = 10` fine time steps inside each subinterval. To approximate the solution numerically, the Forward Euler method was implemented that has already been described in Section 1.3.1. Looking at Figures (9), (10), (11),

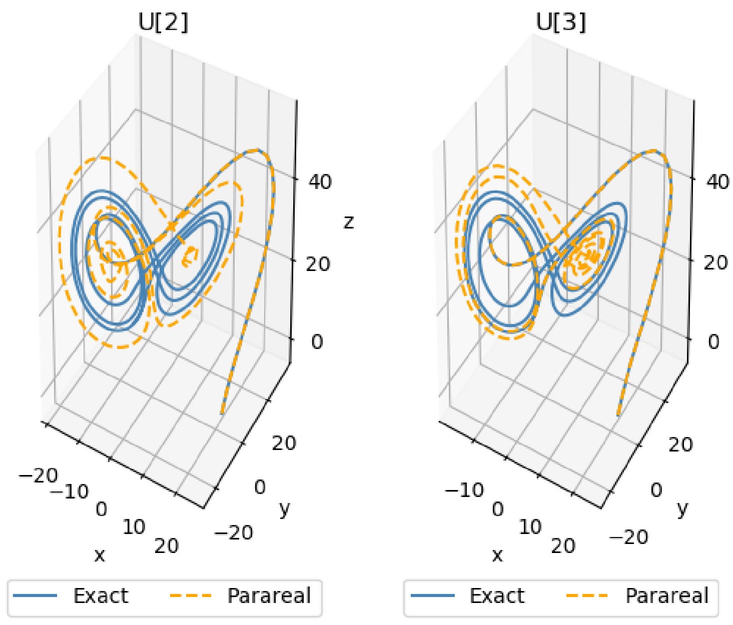


Figure 10: **Parareal-convergence for the Lorenz system, iterations $k = 3$ and $k = 4$**

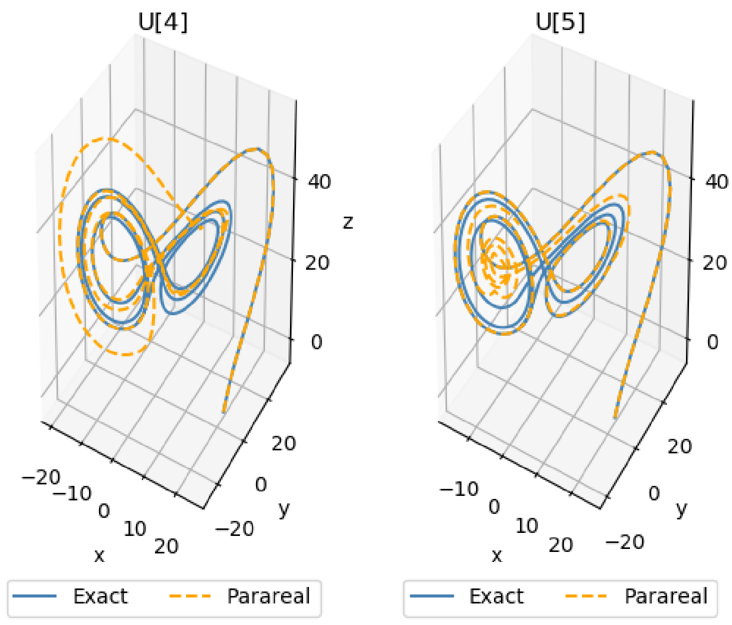


Figure 11: **Parareal-convergence for the Lorenz system, iterations $k = 5$ and $k = 6$**

(12) and (13), it can be see how after ten iterations, the Parareal approximation comes closer and closer to the Forward-Euler approximation taken over the entire time interval with fine steps (called the test or exact solution from now on). Observe how

the initial solution or initial guess of Parareal is way off to the exact solution in Figure (9). From the third until eighth iteration, Figure (10), (11), (12), the Parereal approximation swerves from one side to the other, until it settles down at the nineth and tenth iteration in Figure (13). The Parareal solution seems to have visually converged two times faster than the set number of iterations, that is the reason the rest of the plots are not shown.[9]

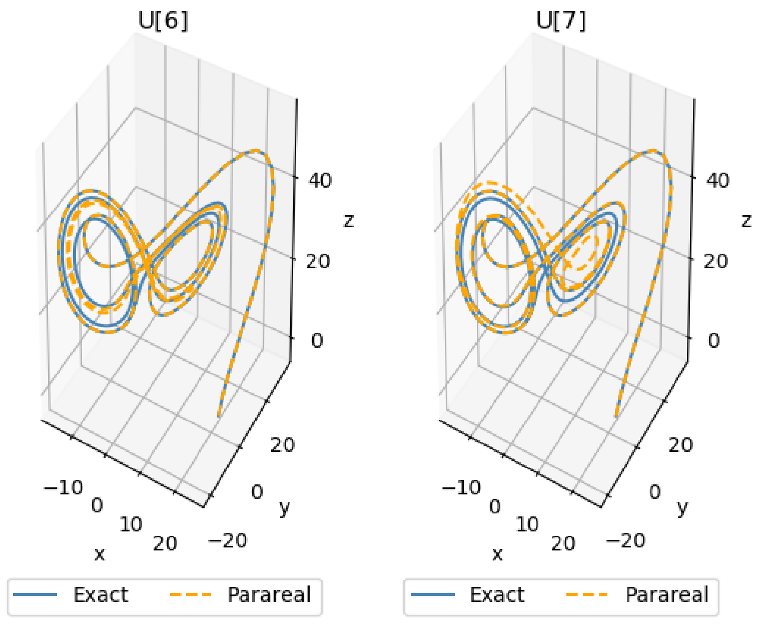


Figure 12: **Parareal-convergence for the Lorenz system, iterations $k = 7$ and $k = 8$**

A rigorous method to check if the approximation comes close to the test solution is to calculate the error. This can be done by determining the l_2-norm for each Parareal iteration k. The (normalized) l_2-error is formulated as follows:

$$||u_{test} - u_{parareal}||_2 = \frac{\sqrt{\sum_i^n |u_{test} - u_{parareal}|^2}}{\sqrt{\sum_i^n |u_{test}|^2}}, \tag{30}$$

where u_{test} and $u_{parareal}$ both contain vectors of size `Mc = 500` three times, because there are three coordinates in total. In Figure (14) there are two error-plots. The x-axis is connected to the twenty Parareal iterations, while the y-axis shows the value of the error. In the linear scale plot, the error seems to jump up and down between the first nine iterations and then exponentially drop. However, in the logarithmic scale graph we notice a super exponential trend, because the error-curve is not linear downwards. Another phenomenon, that is not pictured in Figure (14), is that plotting

[9]It would be the plot of the same picture for another ten times.

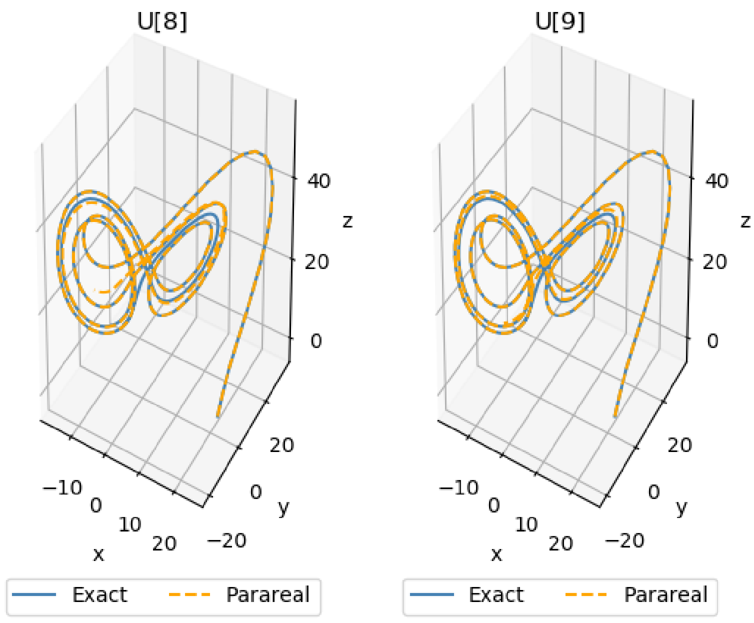


Figure 13: **Parareal-convergence for the Lorenz system, iterations $k = 9$ and $k = 10$**

the logarithmic scale graph for more Parareal iterations, for `K = 50` for example, the curve starts to jump up and down around 10^{-12}, close to machine precision. The first assumption is perhaps that the error decrease is not monotone, but closely examining at what values the jumps happen, it is more likely to just be floating point errors.

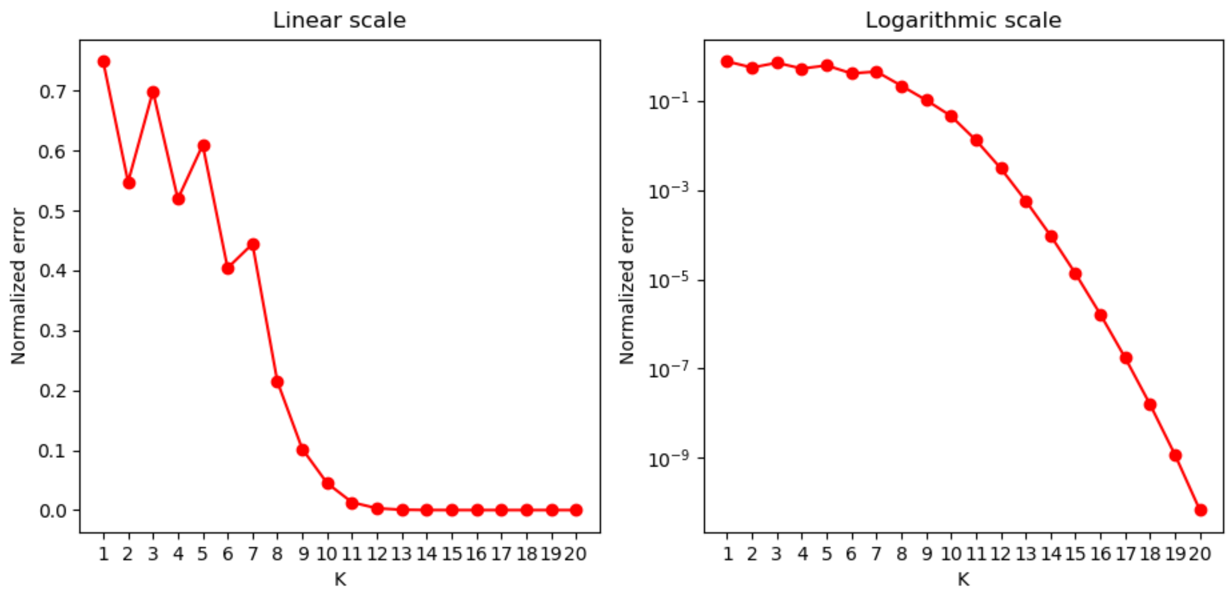


Figure 14: **Super exponential error-decrease for the Lorenz System**

3.2 Heat equation

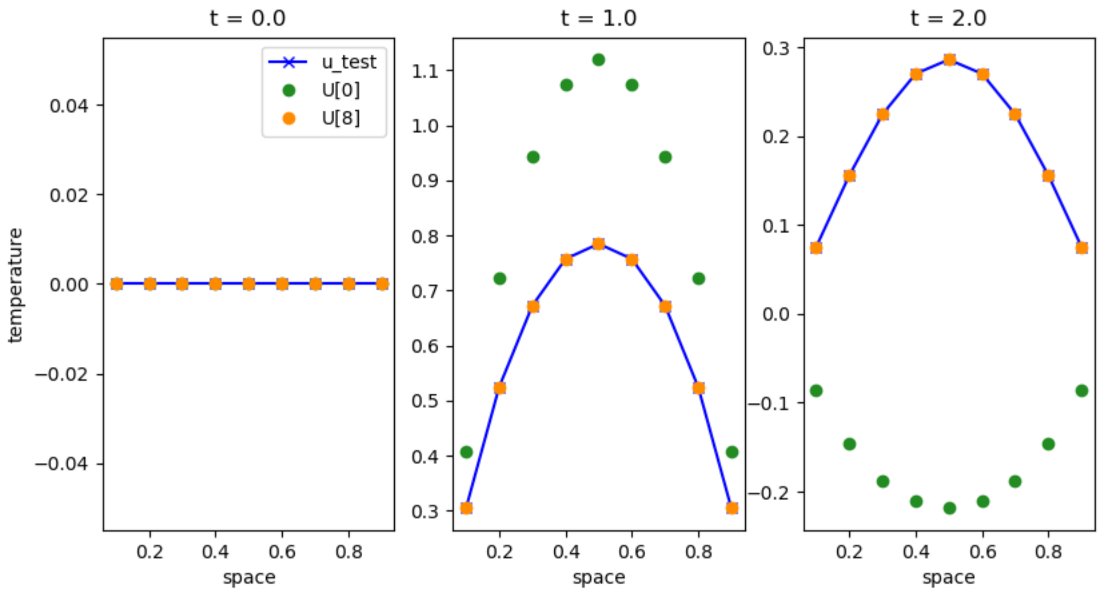


Figure 15: **Parareal-convergence for the Heat equation, time** $t = 0$**,** $t = 1$**,** $t = 2$

To numerically solve the Heat Equation, the Backward-Euler method is utilized, see Section 1.3.1. From Section 1.2, we already know the zero boundary and initial conditions. We have a pulsating source directed towards a rod of L = 1. We discretize the rod into ten uniform intervals. The spatial grid contains `9` inner points. For Parareal, we go through `K = 8` iterations and partition the time interval `T =` $[0, 8]$ into `Mc = 8` coarse time intervals and `Mf = 160` fine time time intervals, resulting into `Mfc = 20` fine time steps per subinterval. In Figures (15), (16) and (17) the temperature is set on

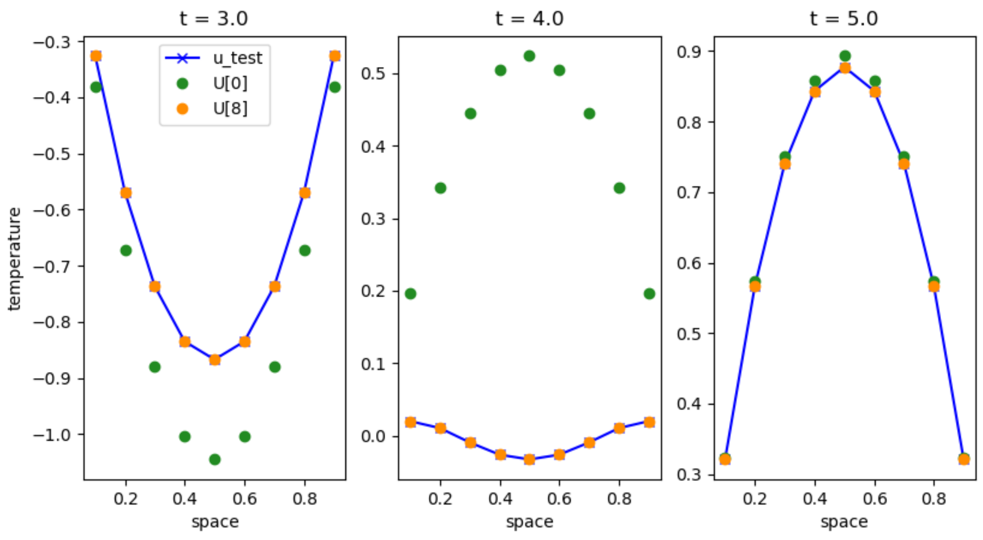


Figure 16: **Parareal-convergence for the Heat equation, time** $t = 3$**,** $t = 4$**,** $t = 5$

the y-axis and the inner points of the rod are set on the x-axis. Figure (15) shows the first three snapshots in time. In the first picture the temperature is zero over the whole rod. When the source starts to work, the rod warms up and cools down periodically and the heat diffuses symmetrically. With blue the exact solution is plotted, just like for the Lorenz system (3.1). The green dots are the placement of the initial guess of

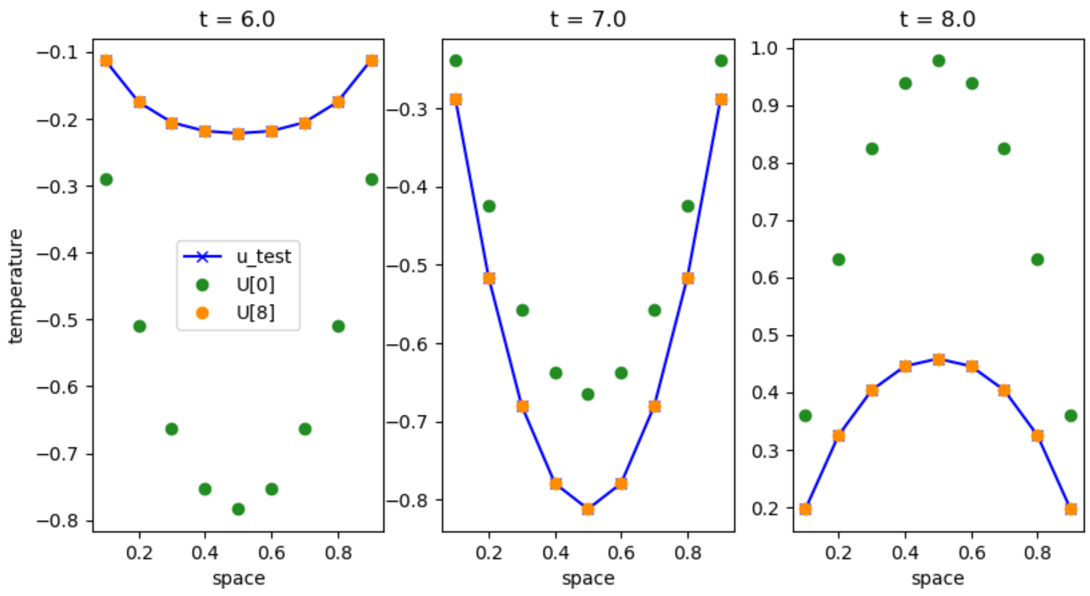


Figure 17: **Parareal-convergence for the Heat equation, time** $t = 6$**,** $t = 7$**,** $t = 8$

the Parareal algorithm, the orange ones indicate the final Parareal solution. The final iteration completely coincides with the exact solution, but we have to look at the error plots before we can make this type of conclusion, of course. In Figure (19) we see a

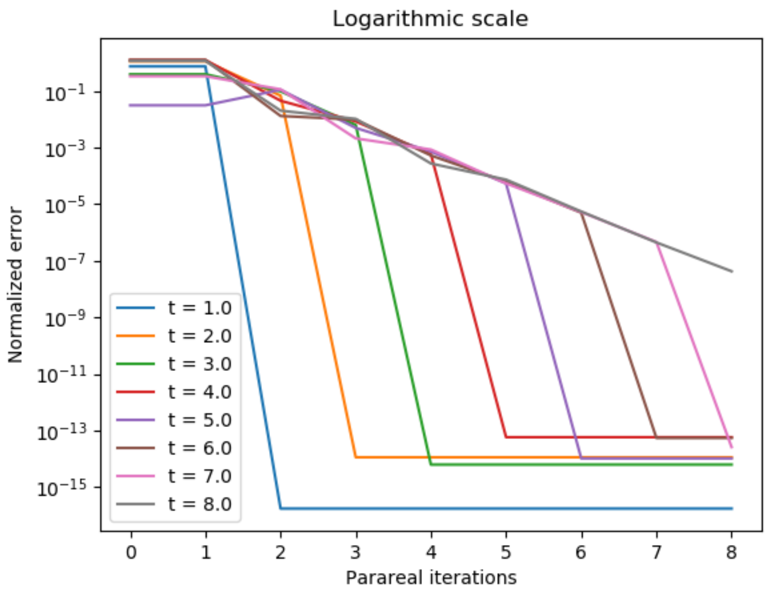


Figure 18: **Exponential error-decrease for the Heat equation per coarse time step**

similar graph as in Figure (14), again plotted by computing the l_2-error as in (30). Except, we have a perfect exponential decrease in error. This corresponds to predictions of Theorem 2 of Section 1.4. Figure (18) shows a non-normalized computation of the l_2-error, plus for each coarse time step the decrease in error is determined. The staircase pattern can be interpreted as a depiction of how after each iteration one of the coarse time intervals has converged to the exact solution. Remembering the analysis about Theorem 1 in Section 1.4 it is not impressive that the Parareal algorithm converges when `K = Mc`, as is exactly in this experimental case. Fortunately, choosing `K = 10` and `Mc = 80` and `Mf = 1600`, we still get an exponential convergence and a linear decrease in error up till machine precision after ten iterations (Not pictured).

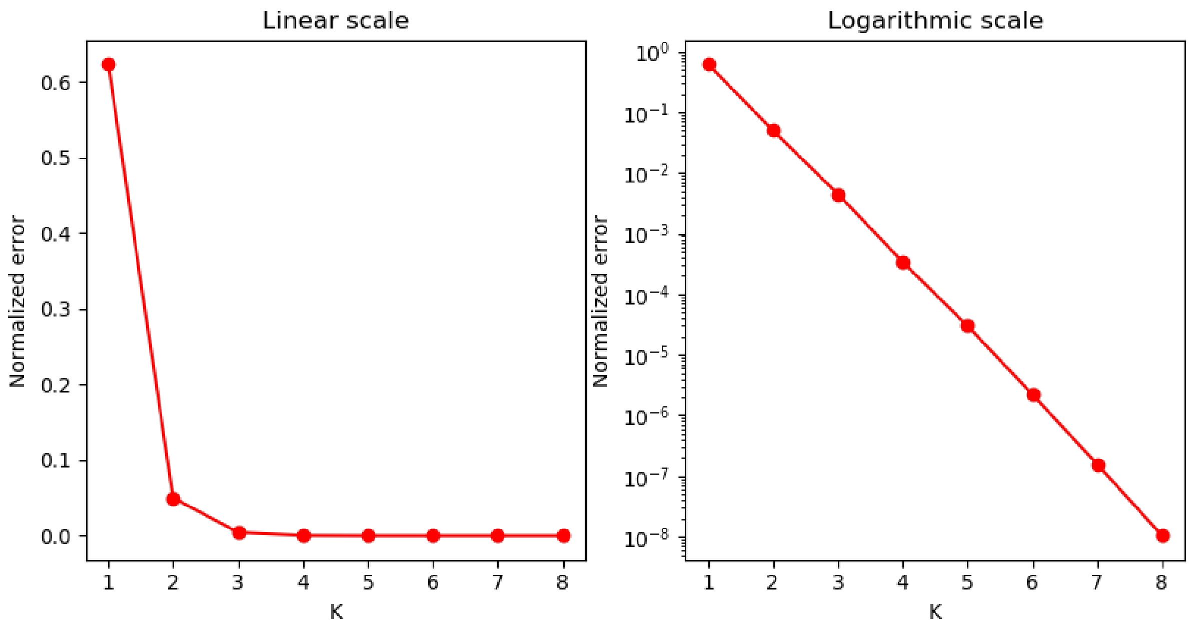


Figure 19: **Exponential error-decrease for the Heat equation)**

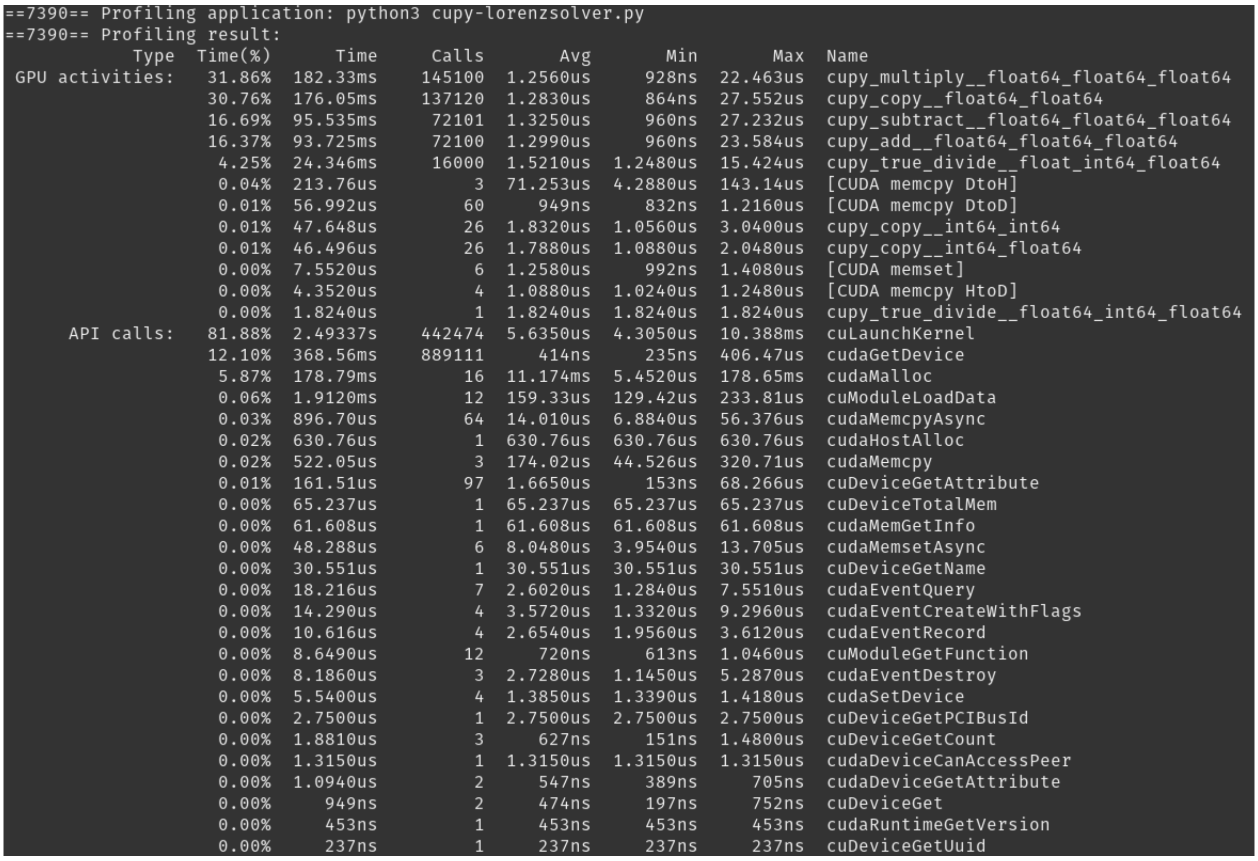
3.3 Code profiling

Both the NumPy and CuPy version of the Python script produced the same results. The difference in the CuPy implementation as opposed to its NumPy version is in the fine solver stage. There, the $\mathcal{F}$-propogators are run in parallel on each coarse subinterval on the GPU. The main question remains: Did the the parallel version outperform the sequential one?

The answer is: not quite. Overall, when first attempts at measuring the time for the Parareal implementation were made (compared to the time for a sequential Foward or Backward-Euler implementation with fine time steps) were made, the sequential version appeared to be faster. To understand where it went ”wrong”, two profilers for Nvidia were applied to the Python programs: NvidiaVisual Profiler (Extention:

`.nvvp`) and the `nvprof` command inside the Pop OS terminal.[10]

3.4 Profiler output: Lorenz system



```
==7390== Profiling application: python3 cupy-lorenzsolver.py
==7390== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   31.86%  182.33ms    145100  1.2560us     928ns  22.463us  cupy_multiply__float64_float64_float64
                   30.76%  176.05ms    137120  1.2830us     864ns  27.552us  cupy_copy__float64_float64
                   16.69%  95.535ms     72101  1.3250us     960ns  27.232us  cupy_subtract__float64_float64_float64
                   16.37%  93.725ms     72100  1.2990us     960ns  23.584us  cupy_add__float64_float64_float64
                    4.25%  24.346ms     16000  1.5210us  1.2480us  15.424us  cupy_true_divide__float_int64_float64
                    0.04%  213.76us         3  71.253us  4.2880us  143.14us  [CUDA memcpy DtoH]
                    0.01%  56.992us        60     949ns     832ns  1.2160us  [CUDA memcpy DtoD]
                    0.01%  47.648us        26  1.8320us  1.0560us  3.0400us  cupy_copy__int64_int64
                    0.01%  46.496us        26  1.7880us  1.0880us  2.0480us  cupy_copy__int64_float64
                    0.00%  7.5520us         6  1.2580us     992ns  1.4080us  [CUDA memset]
                    0.00%  4.3520us         4  1.0880us  1.0240us  1.2480us  [CUDA memcpy HtoD]
                    0.00%  1.8240us         1  1.8240us  1.8240us  1.8240us  cupy_true_divide__float64_int64_float64
      API calls:   81.88%  2.49337s    442474  5.6350us  4.3050us  10.388ms  cuLaunchKernel
                   12.10%  368.56ms    889111     414ns     235ns  406.47us  cudaGetDevice
                    5.87%  178.79ms        16  11.174ms  5.4520us  178.65ms  cudaMalloc
                    0.06%  1.9120ms        12  159.33us  129.42us  233.81us  cuModuleLoadData
                    0.03%  896.70us        64  14.010us  6.8840us  56.376us  cudaMemcpyAsync
                    0.02%  630.76us         1  630.76us  630.76us  630.76us  cudaHostAlloc
                    0.02%  522.05us         3  174.02us  44.526us  320.71us  cudaMemcpy
                    0.01%  161.51us        97  1.6650us     153ns  68.266us  cuDeviceGetAttribute
                    0.00%  65.237us         1  65.237us  65.237us  65.237us  cuDeviceTotalMem
                    0.00%  61.608us         1  61.608us  61.608us  61.608us  cudaMemGetInfo
                    0.00%  48.288us         6  8.0480us  3.9540us  13.705us  cudaMemsetAsync
                    0.00%  30.551us         1  30.551us  30.551us  30.551us  cuDeviceGetName
                    0.00%  18.216us         7  2.6020us  1.2840us  7.5510us  cudaEventQuery
                    0.00%  14.290us         4  3.5720us  1.3320us  9.2960us  cudaEventCreateWithFlags
                    0.00%  10.616us         4  2.6540us  1.9560us  3.6120us  cudaEventRecord
                    0.00%  8.6490us        12     720ns     613ns  1.0460us  cuModuleGetFunction
                    0.00%  8.1860us         3  2.7280us  1.1450us  5.2870us  cudaEventDestroy
                    0.00%  5.5400us         4  1.3850us  1.3390us  1.4180us  cudaSetDevice
                    0.00%  2.7500us         1  2.7500us  2.7500us  2.7500us  cuDeviceGetPCIBusId
                    0.00%  1.8810us         3     627ns     151ns  1.4800us  cuDeviceGetCount
                    0.00%  1.3150us         1  1.3150us  1.3150us  1.3150us  cudaDeviceCanAccessPeer
                    0.00%  1.0940us         2     547ns     389ns     705ns  cudaDeviceGetAttribute
                    0.00%     949ns         2     474ns     197ns     752ns  cuDeviceGet
                    0.00%     453ns         1     453ns     453ns     453ns  cudaRuntimeGetVersion
                    0.00%     237ns         1     237ns     237ns     237ns  cuDeviceGetUuid
```

Figure 20: **nvprof profiler results for the Lorenz system)**

In Figure (20) the output of `nvprof` for the Lorenz system solver is shown. Calling `nvprof` for Python is as simple as writing `sudo nvprof python3 program.py`. There are two categories displayed: GPU activities and API calls. The API (Application Programming Interface) calls in this instance are internal Cupy calls to CUDA. They are not of relevance. The first five lines of GPU activitities should peak our interest. Multiplication (31.87%), copying (30.76%), subtraction (16.69%), addition (16.37%) and division (4.25%) take up most of the time. Indeed, it seems like the operations inside the Lorenz system, the actual solving of the ODE system (1.1) are the main reason for tardiness of the program. Copying is just one unfortunate part of the `for`-loop structure of the code. Furthermore, taking a look at visual profiler output in Figure (21), it becomes immediately clear that Host to Device, Device to Host and Device to Device communication can be neglected. So really the computations themselves take the most time.

[10]Profilers are tools to measure and display CPU and GPU activity.

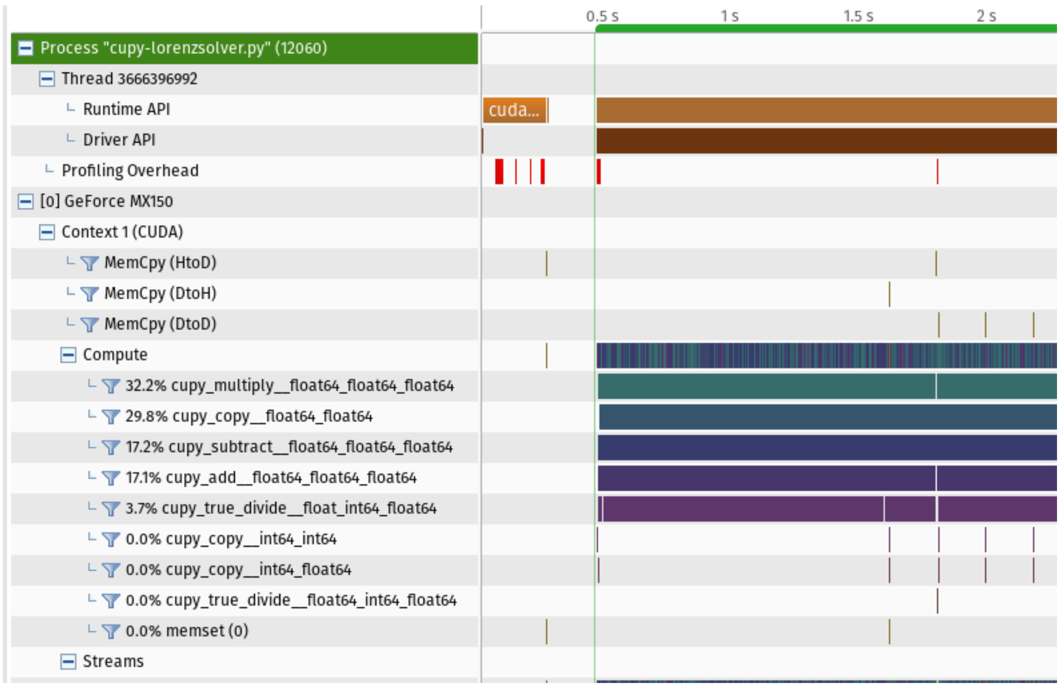
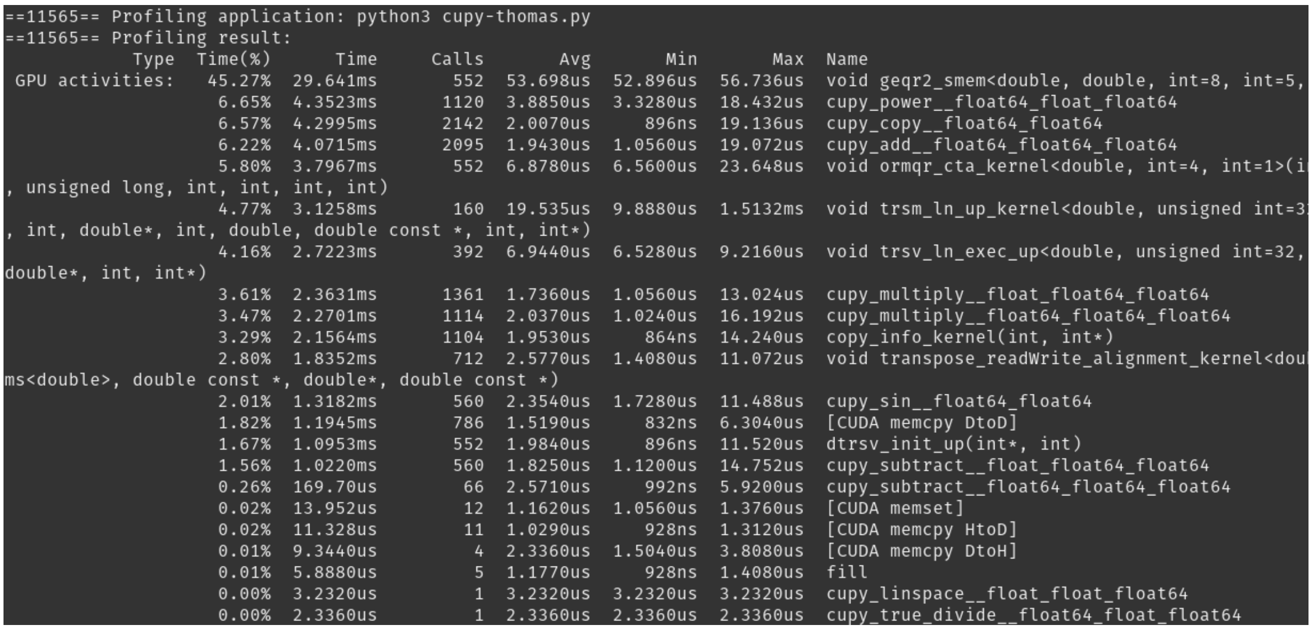

0.5 s
1 s
1.5 s
2 s
Process "cupy-lorenzsolver.py" (12060)
Thread 3666396992
Runtime API
cuda...
Driver API
Profiling Overhead
[0] GeForce MX150
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
32.2% cupy_multiply__float64_float64_float64
29.8% cupy_copy__float64_float64
17.2% cupy_subtract__float64_float64_float64
17.1% cupy_add__float64_float64_float64
3.7% cupy_true_divide__float_int64_float64
0.0% cupy_copy__int64_int64
0.0% cupy_copy__int64_float64
0.0% cupy_true_divide__float64_int64_float64
0.0% memset (0)
Streams

Figure 21: **.nvvp profiler results for the Lorenz system)**

3.5 Profiler output: Heat equation

Analyzing Figure (22) where API calls were not included, the arithmetical operations seem to be scattered between the smaller percentages, although taking the power (6.65%) takes the cake, probably the computation of the source function (1.2) is at fault there. The main reason for slowness is something called geqr2_smem (45.7%).

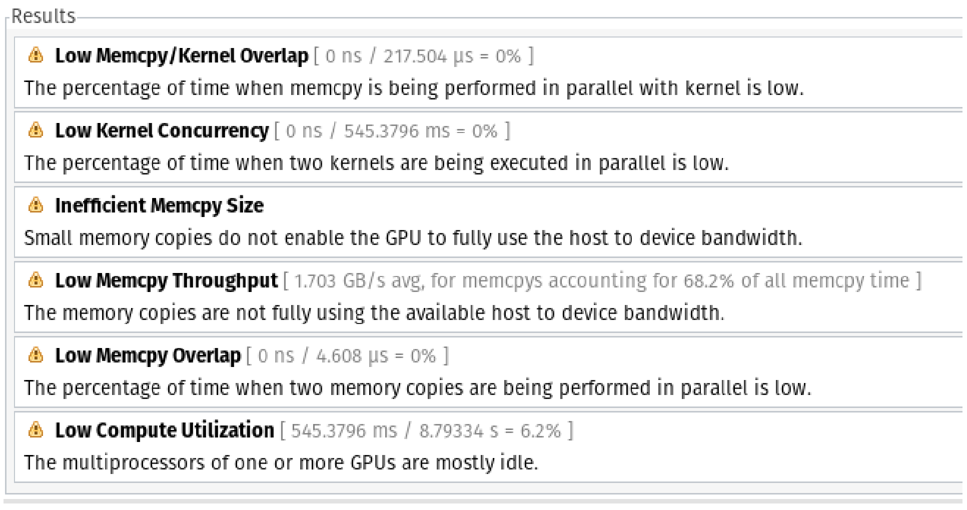


```
==11565== Profiling application: python3 cupy-thomas.py
==11565== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   45.27%  29.641ms       552  53.698us  52.896us  56.736us  void geqr2_smem<double, double, int=8, int=5,
                    6.65%  4.3523ms      1120  3.8850us  3.3280us  18.432us  cupy_power__float64_float_float64
                    6.57%  4.2995ms      2142  2.0070us     896ns  19.136us  cupy_copy__float64_float64
                    6.22%  4.0715ms      2095  1.9430us  1.0560us  19.072us  cupy_add__float64_float64_float64
                    5.80%  3.7967ms       552  6.8780us  6.5600us  23.648us  void ormqr_cta_kernel<double, int=4, int=1>(i
, unsigned long, int, int, int, int)
                    4.77%  3.1258ms       160  19.535us  9.8880us  1.5132ms  void trsm_ln_up_kernel<double, unsigned int=3
, int, double*, int, double, double const *, int, int*)
                    4.16%  2.7223ms       392  6.9440us  6.5280us  9.2160us  void trsv_ln_exec_up<double, unsigned int=32,
double*, int, int*)
                    3.61%  2.3631ms      1361  1.7360us  1.0560us  13.024us  cupy_multiply__float_float64_float64
                    3.47%  2.2701ms      1114  2.0370us  1.0240us  16.192us  cupy_multiply__float64_float64_float64
                    3.29%  2.1564ms      1104  1.9530us     864ns  14.240us  copy_info_kernel(int, int*)
                    2.80%  1.8352ms       712  2.5770us  1.4080us  11.072us  void transpose_readWrite_alignment_kernel<dou
ms<double>, double const *, double*, double const *)
                    2.01%  1.3182ms       560  2.3540us  1.7280us  11.488us  cupy_sin__float64_float64
                    1.82%  1.1945ms       786  1.5190us     832ns  6.3040us  [CUDA memcpy DtoD]
                    1.67%  1.0953ms       552  1.9840us     896ns  11.520us  dtrsv_init_up(int*, int)
                    1.56%  1.0220ms       560  1.8250us  1.1200us  14.752us  cupy_subtract__float_float64_float64
                    0.26%  169.70us        66  2.5710us     992ns  5.9200us  cupy_subtract__float64_float64_float64
                    0.02%  13.952us        12  1.1620us  1.0560us  1.3760us  [CUDA memset]
                    0.02%  11.328us        11  1.0290us     928ns  1.3120us  [CUDA memcpy HtoD]
                    0.01%  9.3440us         4  2.3360us  1.5040us  3.8080us  [CUDA memcpy DtoH]
                    0.01%  5.8880us         5  1.1770us     928ns  1.4080us  fill
                    0.00%  3.2320us         1  3.2320us  3.2320us  3.2320us  cupy_linspace__float_float_float64
                    0.00%  2.3360us         1  2.3360us  2.3360us  2.3360us  cupy_true_divide__float64_float_float64
```

Figure 22: **nvprof profiler results for the Heat equation)**

Although we have no clue why it is called that way, from its input arguments one can deduce it is the function `cp.linalg.solve()`, which is a Cupy solver responsible for finding a solution for a system of linear equations. In this case, the system is a result of discretization in space for the Heat equation by the finite difference method (1.3.1). As what happens inside the function, it is in a black box so to speak, so we cannot figure out why the solver would calculate inefficiently in parallel. Or can we?

It is abundantly clear that for both problems the mathematical computations are the number one cause for the inefficiency of the parallel programs. Another important reason is the low usage of the GPU's capacity. As we looked at relatively small size problems, the GPU could not realize its full potential (See Figure (23)). Lastly, the comparison might have not been fair as well, because we are comparing codes run on CPU versus GPU, which have different clock frequency and computing power. This will later be addressed and a more fair comparisson will be presented (Section (6))


Results

Low Memcpy/Kernel Overlap [0 ns / 217.504 µs = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low.

Low Kernel Concurrency [0 ns / 545.3796 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low.

Inefficient Memcpy Size
Small memory copies do not enable the GPU to fully use the host to device bandwidth.

Low Memcpy Throughput [1.703 GB/s avg, for memcpys accounting for 68.2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 4.608 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Low Compute Utilization [545.3796 ms / 8.79334 s = 6.2%]
The multiprocessors of one or more GPUs are mostly idle.

Figure 23: **Low usage of GPU, .nvvp profiler result**

To salvage these bottlenecks we have to, on one hand, discover a more efficient way to computationally approach the Lorenz system. On the other hand, we have to find a mathematically sound approach to the Heat equation, by analyzing algorithms that work better on this type of problem.

4 Optimization of linear solvers on the GPU

Thankfully, the research on algorithms for systems of linear equations is vast. There are plenty algorithms to choose from, so we have to categorize them properly to decide which one suits our problem the best. The best place to start is the algorithm behind `np.linalg.solve()` and `cp.linalg.solve()`, which is based on the LU decomposition. The LU decomposition is part of the direct solution methods for linear systems. Direct solvers are characterized by being able to determine the exact solution, not counting the rounding errors, in a fixed number of steps. The number of steps hinge on the system size. The algorithms that are part of direct solvers often involve the factorization of the coefficient matrix A, which is part of the considered linear system, abstractly denoted:

$$A\mathbf{x} = \mathbf{b}. \tag{31}$$

Which direct solution method is more appropriate and henceforth more efficient during the parallelization of the problem, is dependant on the structure of the matrix A. This is the moment where linear algebra shows her head. We have to look at matrix properties. Matrices are very diverse. It is a general rule that for linear systems with sparse matrices, the class of so-called iterative solution methods is actually a better fit [13], as factorization of large matrices leads to an increase in computational work [10]. This does not mean there are no direct solvers for sparse linear systems. On the contrary, there are a plethora of direct solvers for matrices with a banded structure, for example tri-diagonal matrices. That is excellent news, because applying the Backward-Euler method on the Heat equation indeed forms a linear tri-diagonal system. We will now dive into the world of direct solvers, by examining a couple of them, and seeing why a generic LU decomposition might not have been the best suited for the Heat equation problem.

4.1 Dense and sparse direct solvers

LU-based solvers have been thoroughly discussed in books such as Arieh Iserles 1997 [6]. The most important part about these solver is their computational cost. An LU solver performs an LU-decomposition of the system matrix A, which gives us $A = LU$, where L is a lower-triangular and U is an upper triangular matrix. The matrix L has the additional property that all diagonal elements equal one. To find the solution for the original system, two sub systems $L\mathbf{y} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$ must be solved. Both, by substitution, can be solved in $\mathcal{O}(M^2)$ floating point operations (FLOPs). The LU-decomposition itself takes $\mathcal{O}(M^3)$ FLOPs in total. Additionally, the storage of elements is M^2. This is the case for a matrix A that is dense. This means most elements of the matrix are non-zero.

For sparse matrices, which have mostly zero elements, the LU-decomposition can be optimized, in particular for sparse banded matrices. Roughly speaking, banded matrices have a bandwidth s, which is defined as the utmost distance between non-zero elements. For a tri-diagonal system, like the one we have in the Heat eqaution, the bandwidth is $s = 1$. The storage of elements for a banded matrix is in general $(2s + 1)M$. Solving the sub systems $L\mathbf{y} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$ takes $\mathcal{O}(sM)$ FLOPs and the total LU-decomposition is $\mathcal{O}(s^2M)$.

4.1.1 Sparse storage formats: COO, CSR and CSC

Sparse solvers, like `sp.linalg.spsolve()`[11], use COO, CSR and CSC format to store sparse matrices [13] (pp. 84-85). While a dense matrix is usually stored as a 2-dimensional array with M^2 non-zero elements, sparse formats store only the non-zero elements and their coordinates. The COO (Coordinate) format consists of three arrays. Array 1 contains all the non-zero (real or complex) elements in any order. Then Array 2 stores the row indices and Array 3 the column indices (all integer). The three arrays are only so long as the number of non-zero elements. In COO format, the number of elements stored for a tri-diagonal matrix would be $3(3M - 2) = 9M - 6$.

The CSR (Compressed Sparse Row) format and the CSC (Compressed Sparse Column) format, the order of the elements is set to be from row to row and from column to column, respectively. Again three arrays are made. Per element only two numbers need to be stored: in Array 1 the number itself and in Array 2 its position inside the row or column. Array 3 then contains how many non-zero elements there are inside each row or column. A tri-diagonal matrix in these formats would take up a storage of $M + 2(3M - 2) = 7M - 4$, a bit less than for COO.

For a general tri-diagonal matrix, using these formats would be redundant, because one only needs to know the position of the diagonals and their elements, which requires $3M - 2 + 3 = 3M + 1$ storage. A symmetric tri-diagonal matrix with constant diagonals like in the Heat equation is completely defined by 4 values, two diagonal positions and two numbers.

4.2 Tri-diagonal solvers: Thomas Algorithm

We know now that the tri-diagonal matrix for the Heat equation matrix is very easily stored. It is therefore a good approach to find a solver designed for this specific type

[11]This function is part of the SciPy module

of problem. That is how we stumble upon the Thomas algorithm. It is first and foremost a simplified version of Gaussian elimination, according to Zhang et al. 2012 [16]. They describe two phases, the forward elimination and backward substitution. Let matrix A be (M, M) and the system of equations $A\mathbf{x} = \mathbf{b}$ of the following form:

$$\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{M-1} \\ & & & & a_{M-1} & b_M \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_M \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ \vdots \\ d_M \end{bmatrix} \tag{32}$$

Then, the first phase, where the lower diagonal is eliminated, is formulated as follows:

$$\begin{aligned} c'_1 &= \frac{c_1}{b_1}, \qquad c'_i = \frac{c_i}{b_i - c'_{i-1}a_i}, \qquad i = 2, 3, \dots, M. \\ d'_1 &= \frac{d_1}{b_1}, \qquad d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}, \qquad i = 2, 3, \dots, M. \end{aligned} \tag{33}$$

The second phase, the backward substitution, calculates the unknowns of $\mathbf{x}$ with length M from last, x_M, to first, x_1. This process can be summarized like this:

$$x_M = d'_M, \qquad x_i = d'_i - c'_i x_{i+1}, \qquad i = M-1, \dots, 2, 1. \tag{34}$$

Counting the FLOPs, we get counting $1 + 3(M-1) + 1 + 5(M-1) + 2(M-1) = 2 + 3M - 3 + 5M - 5 + 2M - 2 = 10M - 8$. The algorithm takes $2M$ sequential computational steps, due to the fact that computing c'_i, d'_i and x_i are based on the previous values, c'_{i-1} , d'_{i-1} and x_{i-1}. However, there are still ways to still parallelize these sequential steps.

4.3 Parallel solvers: Cyclic reduction

Zhang et al. 2012 also give a discription of Cyclic reduction. This method consists of two phases as well, but the first phase is replaced by forward reduction. The advantage of forward reduction as opposed to forward elimination is that with each iteration of the algorithm the system of equations is *reduced* to half it's size. This process is repeated up till two unknowns are left. The second phase, the backward substitution, is the same as in the Thomas algorithm, but it is utilized to consecutively calculate the other half of the unknowns. This is achieved by substituting the values that were previously determined by forward reduction.

To formulate the forward reduction mathematically, we consider the system in **??** we have to take into account that for each step all equations with an even index are updated in parallel. This is done by taking equation i of the system at hand and rewriting it as a linear combination of equation i, $i-1$ and $i+1$ from the same system. Equation i can be then defined as:

$$a_i x_{i-1} + b_i x_i + c i x_{i+1} = d_i. \tag{35}$$

The values of a_i, b_i, c_i and d_i are updated, so to speak, by the next set of equations:

$$\begin{aligned}
a'_i &= -a_{i-1}k_1, \\
b'_i &= b_i - c_{i-1}k_1 - a_{i+1}k_2, \\
c'_i &= -c_{i+1}k_2, \\
d'_i &= d_i - d_{i-1}k_1 - d_{i+1}k_2,
\end{aligned} \tag{36}$$

where k_1 and k_2 are $k_1 = \frac{a_i}{b_{i-1}}$ and $k_2 = \frac{c_i}{b_{i+1}}$. For the backward substitution, all the x_i unknowns with odd index are solved in parallel by taking the x_{i-1} and x_{i+1} that were determined by forward reduction and putting them inside (35), which results in equation:

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}. \tag{37}$$

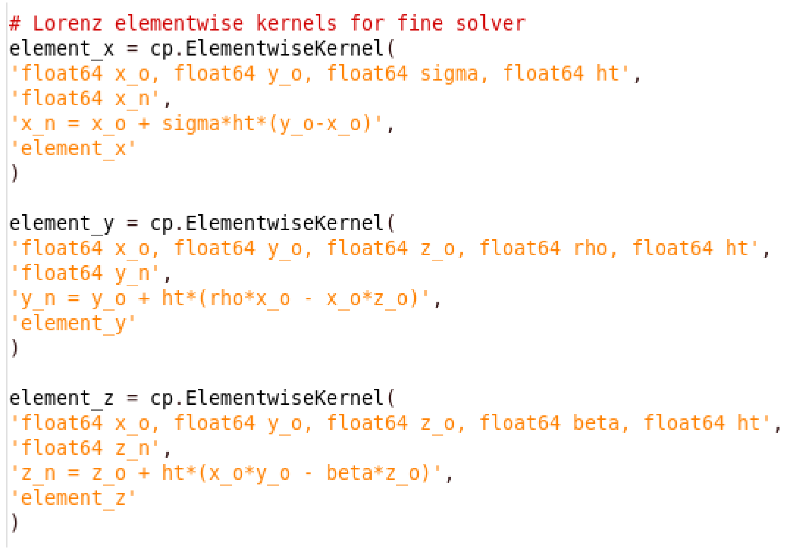
Although the flop count is higher, the Cyclic reduction algorithm performs $2\log_2(M)$ sequential steps due to its parallel nature, while the Thomas algorithm requires $2M$ steps.

5 Implementation of CUDA-based kernels for Parareal

In the previous Section we have discussed a mathematical algorithm that can potentially improve the performance of the Parareal algorithm on the Heat equation. There is still another possibility, namely improvements on software level.

5.1 Elementwise kernel in Cupy: Lorenz system

For the Lorenz system, we can construct a CUDA-kernel in C, instead of using the built-in CuPy array operations. For this purpose, a CuPy elementwise kernel is sufficient. This kernel has a standard input-output structure. The threads are automatically distributed between blocks. In Figure (24) the elementwise kernels are shown. There are three kernels, because we have three coordinates. For the fine solver, each kernel performs one step of the Forward-Euler on an entire array of length `Mc`, with `Mc` being the number of coarse time intervals. In that case, each kernel launches `Mc` number of threads. They are extremely lightweight, as there are no extra local variables created. Furthermore, they only use the input data and the threads do not even communicate with each other.



```
# Lorenz elementwise kernels for fine solver
element_x = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 sigma, float64 ht',
'float64 x_n',
'x_n = x_o + sigma*ht*(y_o-x_o)',
'element_x'
)

element_y = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 z_o, float64 rho, float64 ht',
'float64 y_n',
'y_n = y_o + ht*(rho*x_o - x_o*z_o)',
'element_y'
)

element_z = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 z_o, float64 beta, float64 ht',
'float64 z_n',
'z_n = z_o + ht*(x_o*y_o - beta*z_o)',
'element_z'
)
```

Figure 24: **Elementwise kernel in Cupy**

5.2 Raw kernel in Cupy: Heat equation

For the Heat equation, an elementwise kernel does not suffice. This is due to the fact that the Thomas algorithm is a recurrence relation and cannot be rewritten as pointwise operations. Instead, we have to create a raw kernel in CuPy, which is C-code imported into Python. The input and output arrays are allocated on the GPU, before the kernel executes. There is no Host to Device or Device to Host data transmission.

Three variables are created inside the kernel and at most thirteen variables are accessed by the kernel during the execution per thread. If `Mc` is larger than 1024 more than one block needs to be used.

```
# Thomas algorithm for multiple rhs
thomas_multi = cp.RawKernel(
r'''
extern "C" __global__
void thomas_multi(double *c, double *b, double *f, double *x, double *d, long long n, long long m) {
long long index = blockDim.x * blockIdx.x + threadIdx.x;
for (int i=1; i<n-1; i++)
    {
    c[i] = d[1]/(d[0]-d[1]*c[i-1]);
    }
if (index<m)
    {
    for (long long i=1; i<n; i++)
        {
        b[i*m+index] = (f[i*m+index]-d[1]*b[(i-1)*m+index])/(d[0]-d[1]*c[i-1]);
        }
    x[(n-1)*m+index] = b[(n-1)*m+index];
    for (long long i=1; i<=n; i++)
        {
        x[(n-i-1)*m+index] = b[(n-i-1)*m+index]-c[n-i-1]*x[(n-i)*m+index];
        }
    }
}
''',
'thomas_multi'
)
```

Figure 25: **Raw kernel for multiple right hand sides in Cupy**

6 Speedup analysis

6.1 Theoretical speedup

Let t_{cpu} be the time for one step of Forward or Backward-Euler on the CPU, and t_{gpu} for the GPU likewise. Let the time to calculate the exact solution while traversing with fine steps through the total time interval be $t_{exact} = t_{cpu} M_{fc} M_c$, where M_{fc} are the number of fine steps inside a subinterval and M_c are the number of coarse steps and thus the number of subintervals.

The time to go through the Parareal algorithm is $t_{para} = K(t_{fine} + t_{coarse})$, where K is the number of Parareal iterations, $t_{fine} = t_{gpu} M_{fc}$ is the time it takes for the fine solver to run and $t_{coarse} = t_{gpu} 2M_c$ is similarly corresponding to the coarse solver.[12] Then the ratio between t_{exact} and t_{para} is the following:

$$\begin{aligned} S = \frac{t_{exact}}{t_{para}} &= \frac{t_{cpu} M_{fc} M_c}{K(t_{gpu} M_{fc} + t_{gpu} 2M_c)} \\ &= \frac{t_{cpu}}{t_{gpu}} \frac{M_c}{K\left(1 + \frac{2M_c}{M_{fc}}\right)}. \end{aligned} \tag{38}$$

We call this ratio S. This is exactly the speedup. Here we assumed that t_{fine} is M_c times faster than t_{exact} due to parallelization, but this is a highly ideal situation. So we introduce another ratio, we call it p, the apparent speedup of the fine parallel solver:

$$p = \frac{t_{exact}}{t_{fine}}, \tag{39}$$

where $0 \leq p \leq M_c \frac{t_{cpu}}{t_{gpu}}$. The upper bound is found by dividing t_{exact} by the ideal t_{fine}. Substituting the realistic t_{fine} by $\frac{t_{exact}}{p}$ in (38), we get:

$$S = \frac{t_{cpu}}{t_{gpu}} \frac{M_c}{K\left(\frac{t_{cpu}}{t_{gpu}} \frac{M_c}{p} + \frac{2M_c}{M_{fc}}\right)} \tag{40}$$

[12]The time is multiplied by 2 due to the update step having a correction iteration step and another coarse solver run called inside.

From this formula it follows that the speedup of Parareal depends on the number of iterations K, which is to be expected, the apparent speedup of the fine solver p, as well as the ratio between M_c and M_{fc}. Keeping this in mind, we now can perform a speedup analysis for the Lorenz system and Heat equation. For fairness of comparison, everything is run on GPU, making $t_{cpu} = t_{gpu}$

6.2 Numerical experiments

After performing a number of experiments on the Lorenz system for different values of the parameters from equation 40, Table 1 has been constructed. S_p is the speedup that was achieved in practice, while S_t is the theoretical speedup. The number of K was specifically decided by observing the iteration for which the Parareal algorithm converged to machine precision (10^{-12} for simplicity). For small $M_{fc} = 10$, both the speedup of the fine solver p and S_p (and S_t) approximately coincided. p is quite close to Mc, its upperbound. The values for S_p and S_t fluctuate simply due to the GPU being busy sometimes and not fully dedicated to the numerical computations. The lower value for S_p compared S_t for $M_{fc} = 1000$ might be still somewhat acceptable (around 80%). t_{exact} is definitely two times faster for the kernel version though. This is true for both M_{fc} values. This implies that the elementwise kernel outperforms the native CuPy array operations consistently.

Table 1: **Lorenz system performance**

	M_c	M_{fc}	K	t_{exact}	p	S_p	S_t
`cupy arrays`	500	10	22	0.70	473	0.26	0.22
`cupy arrays`	500	1000	24	75.96	419	8.9	9.5
`elementwise kernel`	500	10	22	0.39	423	0.25	0.22
`elementwise kernel`	500	1000	24	35.5	389	7.06	9.5

For the Heat equation the performance is detailed in Table 2, where we compare the standard CuPy function `cp.linalg.solve()` for dense linear systems and our Thomas fine solver for tri-diagonal systems. J is the number of spatial intervals. The results are arguably more in favor for the custom Thomas (raw) kernel. The values for S_p are better than S_t for every version. Numerical experiments indicate a p that is up to two times bigger than M_c. A possible reason could be the avoidance of Python `for`-loops in both implementations for the fine solver, as they are significantly slower than C-loops. The most impressive result is the difference between $t_e xact$ of the CuPy solver and Thomas kernel for a large spatial grid ($J = 100$). There, we can finally see the true power of the GPU for large problems!

Table 2: **Heat equation performance**

	J	M_c	M_{fc}	K	t_{exact}	p	S_p	S_t
`cp.linalg.solve()`	10	8	20	8	0.05	18.3	1.02	0.81
`cp.linalg.solve()`	10	80	200	18	4.11	137	3.58	3.21
`cp.linalg.solve()`	100	80	200	18	27.67	63	2.63	2.16
`thomas kernel`	10	8	20	8	0.05	19.1	1.04	0.82
`thomas kernel`	10	80	200	18	3.79	160	4.14	3.43
`thomas kernel`	100	80	200	18	4.82	165	4.12	3.47

What is very remarkable, is that for both problems the theoretical and actual speedup are very close. This means that (40) can actually be used to deduce the speedup for much larger problem, without performing the computations.

7 A discussion on Parareal in practice

To conclude, we have shown that the performance of Parareal on the GPU corresponds to the theoretical expectations. We demonstrated that for the Heat equation there was an exponential convergence rate, as predicted. The Lorenz system even had a super exponential convergence. It turned out that the available tools allow for a parallel implementation of Parareal on the GPU. However, it is recommended to use custom-made CUDA-kernels in combination with CuPy, as the built-in solvers and array operations leave a lot to be desired.

We have observed that for systems of linear equations that are integrated with implicit methods, it is essential to investigate the matrix properties to find an optimal algorithm. This reduces the computational cost immensely. In particular, we have constructed a CUDA-kernel implementing a batched Thomas algorithm for the tridiagonal system. It is imperative that sparse direct solvers in CuPy are developed, since the sparse solver in the SciPy library still outperforms in most cases the current CuPy solvers.

The derived speedup formula showed close agreement with actual speedup results. The number of Parareal iterations appeared to of great significance for the speedup. Thus, more needs to be known about convergence rates of the Parareal algorithm with different types of problems.
Lastly, preliminary investigations show that speedup increases for high spatial dimensions. There needs to be more research done theoretically and experimentally on a larger dedicated GPU. The results are promising.

Appendix A

```
#!/usr/bin/env python3

import numpy as np
import cupy as cp
import time

import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d

################################################################################
#
# CUDA KERNELS :D
#
################################################################################

# Lorenz elementwise kernels for fine solver
element_x = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 sigma, float64 ht',
'float64 x_n',
'x_n = x_o + sigma*ht*(y_o-x_o)',
'element_x'
)

element_y = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 z_o, float64 rho, float64 ht',
'float64 y_n',
'y_n = y_o + ht*(rho*x_o - x_o*z_o)',
'element_y'
)

element_z = cp.ElementwiseKernel(
'float64 x_o, float64 y_o, float64 z_o, float64 beta, float64 ht',
'float64 z_n',
'z_n = z_o + ht*(x_o*y_o - beta*z_o)',
'element_z'
)

################################################################################
#
# PARAMETERS AND CONSTANTS
#
################################################################################

K = 24 # number of parareal iterations, 20
T = 5.0 # end time
J = 4 # no space intervals, only one point (with three coordinates)
time_iterations = 5

Mc = 500          # course, 500, 2**5
Mf = 500000# fine, 5000, 2**5 * 2**10
dT = T/Mc          # coarse time step
dt = T/Mf          # fine time step

Mfc = int(Mf/Mc) # number of fine time steps on coarse interval dT
Mcc = 1 # number of coarse time steps on coarse interval dT

M_tjes = cp.array([Mc,Mf,Mfc,Mcc])

# Lorenz parameters
```

```
sigma = 10.
rho = 28.
beta = 8./3.
para = cp.array([sigma,rho,beta])

u0 = np.array([20,5,-5])
u0_GPU = cp.array([20,5,-5])

################################################################################
#
# LORENZ SYSTEM: CONSTRUCTION
#
################################################################################

def lorenz_solver(ht,u_0,M):

    x_t_old = u_0[0]
    y_t_old = u_0[1]
    z_t_old = u_0[2]

    for i in range(M):
        x_t_new = x_t_old + para[0] * (y_t_old - x_t_old)*ht
        y_t_new = y_t_old + (x_t_old * (para[1] - z_t_old))*ht
        z_t_new = z_t_old + (x_t_old*y_t_old - para[2]*z_t_old)*ht

        #x_t_new = element_x(x_t_old,y_t_old,         para[0], ht)
        #y_t_new = element_y(x_t_old,y_t_old,z_t_old, para[1], ht)
        #z_t_new = element_z(x_t_old,y_t_old,z_t_old, para[2], ht)

        x_t_old = cp.copy(x_t_new)
        y_t_old = cp.copy(y_t_new)
        z_t_old = cp.copy(z_t_new)

    return cp.array([x_t_new, y_t_new, z_t_new])

def lorenz_GPU(ht_all,U_matrix,M,Mfine):

    x_GPU_old = U_matrix[0,:]
    y_GPU_old = U_matrix[1,:]
    z_GPU_old = U_matrix[2,:]

    for i in range(Mfine):
        x_GPU_new = x_GPU_old + para[0] * ht_all * (y_GPU_old - x_GPU_old)
        y_GPU_new = y_GPU_old + ht_all*(para[1]*x_GPU_old - x_GPU_old*z_GPU_old)
        z_GPU_new = z_GPU_old + ht_all*(x_GPU_old*y_GPU_old - para[2]*z_GPU_old)

        #x_GPU_new = element_x(x_GPU_old,y_GPU_old,           para[0], ht_all)
        #y_GPU_new = element_y(x_GPU_old,y_GPU_old,z_GPU_old, para[1], ht_all)
        #z_GPU_new = element_z(x_GPU_old,y_GPU_old,z_GPU_old, para[2], ht_all)

        x_GPU_old = cp.copy(x_GPU_new)
        y_GPU_old = cp.copy(y_GPU_new)
        z_GPU_old = cp.copy(z_GPU_new)

    U_matrix[0,1:] = x_GPU_new[0:-1]
    U_matrix[1,1:] = y_GPU_new[0:-1]
    U_matrix[2,1:] = z_GPU_new[0:-1]

    return U_matrix

################################################################################
#
# LORENZ SYSTEM: TEST AND EXACT SOLUTIONS
```

```
#
################################################################################

u_test = cp.zeros((J-1, Mc+1))
u_test[:,0] = cp.copy(u0_GPU)

begin_test = time.time()
for n in range(Mc):
    t0 = n*dT
    t1 = t0 + dT
    hh = (t1-t0)/M_tjes[2]      # M_tjes = cp.array([Mc,Mf,Mfc,Mcc])
    u_test[:,n+1] = lorenz_solver(hh, u_test[:,n], Mfc)
    # change from cupy to element-kernel inside solver
end_test = time.time()
print("Time for test solution: ", end_test - begin_test)

u_test = cp.asnumpy(u_test)

'''
u_exact = cp.zeros((J-1, Mc*Mfc + 1))
u_exact[:,0] = cp.copy(u0_GPU)

begin_exact = time.time()
for n in range(Mc*Mfc):
    t0 = n*dT
    t1 = t0 + dT
    hh = (t1-t0)/M_tjes[2]      # M_tjes = cp.array([Mc,Mf,Mfc,Mcc])
    u_exact[:,n+1] = lorenz_solver(hh,u_exact[:,n],Mcc)
end_exact = time.time()
print("Time for exact solution: ", end_exact - begin_exact)

u_exact = cp.asnumpy(u_exact)
'''

################################################################################
#
# PARAREAL ALGORITHM
#
################################################################################

U_big_GPU = cp.zeros((K+1,J-1,Mc+1))
U_big_GPU[0,:,0] = cp.copy(u0_GPU)

Go_array_GPU = cp.zeros((J-1,Mc+1))
Go_array_GPU[:,0] = cp.copy(u0_GPU)

Fn_array_GPU = cp.zeros((J-1,Mc+1))
Fn_array_GPU[:,0] = cp.copy(u0_GPU)

Gn_array_GPU = cp.zeros((J-1,Mc+1))
Gn_array_GPU[:,0] = cp.copy(u0_GPU)

begin_coarse = time.time()
for n in range(Mc):
    t0 = n*dT
    t1 = t0 + dT
    hh = (t1-t0)/M_tjes[3]      # M_tjes = cp.array([Mc,Mf,Mfc,Mcc])

    Go_array_GPU[:,n+1] = lorenz_solver(hh,Go_array_GPU[:,n],Mcc)
    U_big_GPU[0,:,n+1] = cp.copy(Go_array_GPU[:,n+1])
end_coarse = time.time()
print("Time for coarse solution: ", end_coarse - begin_coarse)
```

```
time_array = cp.array([0,dT])
ht_lol = (time_array[1] - time_array[0])/M_tjes[2]

fine_time = []
update_time = []
begin_total = time.time()
for k in range(K):

    begin_fine = time.time()
    Fn_array_GPU = lorenz_GPU(ht_lol,U_big_GPU[k,:,:],Mc,Mfc)
    end_fine = time.time()
    print("Time for fine solution: ", end_fine - begin_fine)
    fine_time.append(end_fine - begin_fine)

    U_big_GPU[k+1,:,0] = cp.copy(u0_GPU)

    begin_update = time.time()
    for n in range(Mc):
        t0 = n*dT
        t1 = t0 + dT
        hh = (t1-t0)/M_tjes[3] # M_tjes = cp.array([Mc,Mf,Mfc,Mcc])
        Gn_array_GPU[:,n+1] = lorenz_solver(hh,U_big_GPU[k+1,:,n],Mcc)

        # COMBINATION STEP
        U_big_GPU[k+1,:,n+1] = Fn_array_GPU[:,n+1] + Gn_array_GPU[:,n+1] - Go_array_GPU[:,n+1]
    end_update = time.time()
    print("Time for update solution: ", end_update - begin_update)
    update_time.append(end_update - begin_update)

    Go_array_GPU = cp.copy(Gn_array_GPU)
end_total = time.time()
print("Time for total solution: ", end_total - begin_total)

average_fine = sum(fine_time)/len(fine_time)
average_update = sum(update_time)/len(update_time)

print("average_fine = ", average_fine)
print("average_update = ", average_update)

U_big_CPU = cp.asnumpy(U_big_GPU)

################################################################################
#
# ERRORS
#
################################################################################

# normalized errors for x,y,z-dimension

errors = np.zeros(K)

for k in range(K):
        errors[k] = np.sqrt(np.sum((u_test - U_big_CPU[k,:,:])**2))/np.sqrt(np.sum(u_test**2))

################################################################################
#
# PLOTS
#
################################################################################

plt.ion()
plt.clf()
```

```
k_array = np.linspace(1,K,K)
subcount = 1

fig = plt.figure(1)
for k in range(int(10)):
        x_array = U_big_CPU[k,0,:]
        y_array = U_big_CPU[k,1,:]
        z_array = U_big_CPU[k,2,:]
        x_first = u_test[0]
        y_first = u_test[1]
        z_first = u_test[2]
        ax = fig.add_subplot(2,5,subcount, projection='3d')
        ax.plot3D(x_first,y_first,z_first, 'steelblue', label = 'Exact')
        ax.plot3D(x_array,y_array,z_array, '--', color='orange', label = 'Parareal')
        ax.set_title("U["+str(k)+"]")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("z")
        ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), ncol=2)
        subcount = subcount + 1

fig2, (ax1,ax2) = plt.subplots(1,2)

ax1.plot(k_array, errors, '-o', color='red')
ax2.semilogy(k_array, errors, '-o', color='red')
plt.setp((ax1,ax2), xticks=k_array)
ax1.set_xlabel('K')
ax2.set_xlabel('K')
ax1.set_ylabel('Normalized error')
ax2.set_ylabel('Normalized error')
ax1.set_title('Linear scale')
ax2.set_title('Logarithmic scale')


fig3 = plt.figure(3)
x_test = u_test[0]
y_test = u_test[1]
z_test = u_test[2]
ax3 = fig3.gca(projection = "3d")
ax3.plot3D(x_test,y_test,z_test, color = 'green')
ax3.set_title("Lorenz system: Butterfly")
ax3.set_xlabel("x")
ax3.set_ylabel("y")
ax3.set_zlabel("z")

plt.show()
```

Appendix B

```
import math
import numpy as np
import cupy as cp
import time
import scipy.sparse as sp
import os

import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d

########################################################################
#
# CUDA KERNELS :D
#
########################################################################

# Thomas algorithm for one rhs
thomas_single = cp.RawKernel(
r'''
extern "C" __global__
void thomas_single(double *c, double *b, double *f, double *x,
double *d, long long n) {
for (int i=1; i<n-1; i++)
    {
    c[i] = d[1]/(d[0]-d[1]*c[i-1]);
    }
for (int i=1; i<n; i++)
    {
    b[i] = (f[i]-d[1]*b[i-1])/(d[0]-d[1]*c[i-1]);
    }
x[n-1] = b[n-1];
for (long long i=1; i<=n; i++)
    {
    x[n-i-1] = b[n-i-1]-c[n-i-1]*x[n-i];
    }
}
''',
'thomas_single'
)

# Thomas algorithm for multiple rhs
thomas_multi = cp.RawKernel(
r'''
extern "C" __global__
void thomas_multi(double *c, double *b, double *f, double *x,
double *d, long long n, long long m) {
long long index = blockDim.x * blockIdx.x + threadIdx.x;
for (int i=1; i<n-1; i++)
    {
    c[i] = d[1]/(d[0]-d[1]*c[i-1]);
    }
if (index<m)
    {
    for (long long i=1; i<n; i++)
        {
        b[i*m+index] = (f[i*m+index]-d[1]*b[(i-1)*m+index])/(d[0]-d[1]*c[i-1]);
        }
    x[(n-1)*m+index] = b[(n-1)*m+index];
    for (long long i=1; i<=n; i++)
```

```
        {
        x[(n-i-1)*m+index] = b[(n-i-1)*m+index]-c[n-i-1]*x[(n-i)*m+index];
        }
    }
}
''',
'thomas_multi'
)

########################################################################
#
# PARAMETERS AND CONSTANTS
#
########################################################################

K = 18                                              # parareal iterations
T = 8.0                                             # end time

uleft = 0.0                                         # left BC
uright = 0.0                                        # right BC

# mesh parameters
J = 100                                             # space intervals, need to be a lot! :)
Jin = J-1
x_grid = np.linspace(0,1,J+1)
x_inner = x_grid[1:-1]                              # without boundaries
x_grid_GPU = cp.linspace(0,1,J+1)
x_inner_GPU = x_grid_GPU[1:-1]
dx = 1./J                                           # grid size

u0 = np.zeros(len(x_grid[1:-1]))                    # initial condition
u0_GPU = cp.zeros(len(x_grid_GPU[1:-1]))

Mc = 80                                             # coarse
Mf = 80*200                                         # fine
dT = T/Mc                                           # coarse time step
dt = T/Mf                                           # fine time step

# number of fine time steps on coarse interval dT
Mfc = int(Mf/Mc)

# number of coarse time steps on coarse interval dT
Mcc = 1

########################################################################
#
# HEAT EQUATION: CONSTRUCTIONS
#
########################################################################

# source function
def source(x,t):
        return x**4 * (1.-x)**4 + 10*cp.sin(8*t)

def source_CPU(x,t):
    return x**4 * (1.-x)**4 + 10*np.sin(8*t)

# construction of matrix A for Thomas Algorithm
Ad = -2.0
Au = 1

# I - dT*A/dx**2
dd_c = 1 - dT*Ad/dx**2
```

```
du_c = -dT*Au/dx**2
d_array_c = cp.array([dd_c,du_c])

# I - dt*A/dx**2
dd_f = 1 - dt*Ad/dx**2
du_f = -dt*Au/dx**2
d_array_f = cp.array([dd_f,du_f])

# construction of matrix A for heat equation solver (GPU)
A0_GPU           = -2*cp.ones(len(x_inner_GPU))
A1min_GPU        = cp.ones(len(x_inner_GPU)-1)
A1plus_GPU       = cp.ones(len(x_inner_GPU)-1)
A_matrix_GPU     = (cp.diag(A1min_GPU,-1) + cp.diag(A0_GPU,0) + cp.diag(A1plus_GPU,1))/dx**2
I_GPU            = cp.eye(len(u0_GPU))

#################################
########### MULTI-DIM RIGHT-HAND SIDE
#################################

U_source = cp.zeros((J-1,Mc,Mfc+1))
time_array = cp.array([0,dT])
step_array = cp.array([dt,dT])

# awkward time-array extraction
t_c = time_array[0]
t_f = time_array[0]
inter_time = cp.array([t_c,t_f])

for ii in range(Mfc+1): ###
    inter_time[0] = cp.copy(inter_time[1])
    for jj in range(Mc):
        U_source[:,jj,ii] = source(x_inner_GPU, inter_time[0])
        inter_time[0] = inter_time[0] + step_array[1]
    inter_time[1] = inter_time[1] + step_array[0]

SysMat_course = I_GPU-step_array[1]*A_matrix_GPU
SysMat_fine = I_GPU-step_array[0]*A_matrix_GPU

#######################################################################
#
# HEAT EQUATION: TEST AND EXACT SOLUTIONS
#
#######################################################################

# Test solution with course step on [0,T]
u_test = cp.zeros((J-1,Mc+1))
u_test[:,0] = cp.copy(u0_GPU)

inter_time = cp.array([t_c,t_f])

time_array = cp.array([0,dT])
step_array = cp.array([dt,dT])

begin_test = time.time()
for n in range(Mc):

    inter_time[0] = n*step_array[1]
    inter_time[1] = inter_time[0] + step_array[1]
    u_old = u_test[:,n]

    # Backward Euler
    for ii in range(Mfc):
        u_array = cp.linalg.solve(SysMat_fine,u_old + step_array[0]*
        source(x_inner_GPU,inter_time[0] + step_array[0]))
        u_old = cp.copy(u_array)
        inter_time[0] = inter_time[0] + step_array[0] ### fine step

    u_test[:,n+1] = u_array
end_test = time.time()
print("Time for test solution: ", end_test - begin_test)

t_test = end_test - begin_test
u_test = cp.asnumpy(u_test)

'''
# Exact solution with fine step on [0,T] with cp.linalg.solve()

u_exact = cp.zeros((J-1,Mc*Mfc+1))
u_exact[:,0] = cp.copy(u0_GPU)

inter_time = cp.array([t_c,t_f])

begin_exact = time.time()
for n in range(Mc*Mfc):

    inter_time[0] = n*step_array[0]
    inter_time[1] = inter_time[0] + step_array[0]
    u_old = u_exact[:,n]

    u_array = cp.linalg.solve(SysMat_fine,u_old + step_array[0]*
    source_CPU(x_inner_GPU,inter_time[0] + step_array[0]))
    u_exact[:,n+1] = u_array
end_exact = time.time()
print("Time for exact solution: ", end_exact - begin_exact)

u_exact = cp.asnumpy(u_exact)


# Exact solution with fine step on [0,T] with thomas_single()

u_thomas = cp.zeros((J-1,Mc*Mfc+1))
u_thomas[:,0] = cp.copy(u0_GPU)

cp.cuda.runtime.deviceSynchronize()
cgpu = cp.zeros(J-2)
bgpu = cp.zeros(J-1)
u_array = cp.zeros(J-1)
cp.cuda.runtime.deviceSynchronize()

inter_time = cp.array([t_c,t_f])

begin_thomas = time.time()
for n in range(Mc): #*Mfc

    #inter_time[0] = n*step_array[0]
    #inter_time[1] = inter_time[0] + step_array[0]

    inter_time[0] = n*step_array[1]
    inter_time[1] = inter_time[0] + step_array[1]
    u_old = u_thomas[:,n]

    #step_array[1]*source(x_inner_GPU,inter_time[0] + step_array[1]

    for i in range(Mfc): #
        fgpu = cp.copy(u_old + step_array[0]*source(x_inner_GPU,inter_time[0] + step_array[0]))
        cgpu[0] = d_array_c[1]/d_array_c[0]
        bgpu[0] = fgpu[0]/d_array_c[0]
        thomas_single((1,),(1,),(cgpu,bgpu,fgpu,u_array,d_array_c,Jin))
        u_old = cp.copy(u_array)
        inter_time[0] = inter_time[0] + step_array[0] ### fine step

    u_thomas[:,n+1] = u_array
end_thomas = time.time()
print("Time for thomas solution: ", end_thomas - begin_thomas)

u_thomas = cp.asnumpy(u_thomas)
t_thomas = end_thomas - begin_thomas

'''
################################################################################
#
# PARAREAL ALGORITHM
#
################################################################################

U_big_GPU = cp.zeros((K+1,J-1,Mc+1))
U_big_GPU[0,:,0] = cp.copy(u0_GPU)

Go_array_GPU = cp.zeros((J-1,Mc+1))
Go_array_GPU[:,0] = cp.copy(u0_GPU)

Fn_array_GPU = cp.zeros((J-1,Mc+1))
Fn_array_GPU[:,0] = cp.copy(u0_GPU)

Gn_array_GPU = cp.zeros((J-1,Mc+1))
Gn_array_GPU[:,0] = cp.copy(u0_GPU)

#################################
########### COARSE PARAREAL ITERATION
#################################

### Thomas algorithm, initialize coefficients and solution for one rhs


cp.cuda.runtime.deviceSynchronize()
cgpu = cp.zeros(J-2)
bgpu = cp.zeros(J-1)
u_array = cp.zeros(J-1)
cp.cuda.runtime.deviceSynchronize()


### Initial guess

inter_time = cp.array([t_c,t_f])
begin_coarse = time.time()
for n in range(Mc):

    inter_time[0] = n*time_array[1]
    inter_time[1] = inter_time[0] + step_array[0]
    #u_old = cp.copy(Go_array_GPU[:,n])
    u_oldcupy = cp.copy(Go_array_GPU[:,n])

    # Backward Euler
    u_cupy = cp.linalg.solve(SysMat_course,u_oldcupy + step_array[1]*
    source(x_inner_GPU,inter_time[0] + step_array[1]))

    #fgpu = cp.copy(u_old + step_array[1]*source(x_inner_GPU,inter_time[0] + step_array[1]))
    #cgpu[0] = d_array_c[1]/d_array_c[0]
    #bgpu[0] = fgpu[0]/d_array_c[0]
    #thomas_single((1,),(1,),(cgpu,bgpu,fgpu,u_array,d_array_c,Jin))

    #errs = cp.sqrt(cp.sum((u_cupy-u_array)**2))
    #print("errs = ", errs)

    #u_old = cp.copy(u_array)
    u_oldcupy = cp.copy(u_cupy)

    Go_array_GPU[:,n+1] = u_cupy # u_array
    U_big_GPU[0,:,n+1] = cp.copy(Go_array_GPU[:,n+1])
end_coarse = time.time()
print("Time for coarse solution: ", end_coarse - begin_coarse)

#################################
########### FINE PARAREAL ITERATION
#################################


### Thomas algorithm, initialize coefficients and solution for multiple rhs
cp.cuda.runtime.deviceSynchronize()
cgpu_f = cp.zeros(J-2)
bgpu_f = cp.zeros((J-1,Mc))
U_matrix = cp.zeros((J-1,Mc))
cp.cuda.runtime.deviceSynchronize()


inter_time = cp.array([t_c,t_f])

### beginning of the Parareal-loop
fine_time = []
update_time = []
begin_total = time.time()
for k in range(K):

    U_m = U_big_GPU[k,:,:]
    U_mcupy_old = U_m[:,0:-1]  # cupy.linalg.solve loop on GPU
    #U_matrix_old = U_m[:,0:-1]

    # Backward Euler
    begin_fine = time.time()
    for ii in range(Mfc):

        U_mcupy = cp.linalg.solve(SysMat_fine, U_mcupy_old + step_array[0]*U_source[:,:,ii+1])
        # cupy.linalg.solve loop on GPU

        #fgpu = cp.copy(U_matrix_old + step_array[0]*U_source[:,:,ii+1])
        #cgpu_f[0] = d_array_f[1]/d_array_f[0]
        #bgpu_f[0,:] = fgpu[0,:]/d_array_f[0]
        #thomas_multi((1,),(Mc,),(cgpu_f,bgpu_f,fgpu,U_matrix,d_array_f,Jin,Mc))

        U_mcupy_old = cp.copy(U_mcupy)  # cupy.linalg.solve loop on GPU
        #U_matrix_old = cp.copy(U_matrix)

        #errs = cp.sqrt(cp.sum((U_mcupy-U_matrix)**2))
        #print("errs matrix = ", errs)
    Fn_array_GPU[:,0]  = U_m[:,0]
    Fn_array_GPU[:,1:] = U_mcupy #U_matrix
    end_fine = time.time()
    print("Time for fine solution: ", end_fine - begin_fine)
    fine_time.append(end_fine-begin_fine)
    U_big_GPU[k+1,:,0] = cp.copy(u0_GPU)
    ### Thomas algorithm, initialize coefficients and solution for one rhs
    cp.cuda.runtime.deviceSynchronize()
    cgpu = cp.zeros(J-2)
    bgpu = cp.zeros(J-1)
    u_array = cp.zeros(J-1)
    cp.cuda.runtime.deviceSynchronize()

    begin_update = time.time()
    for n in range(Mc):

        # Backward Euler Method
        u_oldcupy = cp.copy(U_big_GPU[k+1,:,n])
        #u_old     = cp.copy(U_big_GPU[k+1,:,n])
        inter_time[1] = n*time_array[1]

        u_cupy = cp.linalg.solve(SysMat_course,u_oldcupy + step_array[1]*
        source(x_inner_GPU,inter_time[1] + step_array[1]))

        #fgpu = cp.copy(u_old + step_array[1]*source(x_inner_GPU,inter_time[1] + step_array[1]))
        #cgpu[0] = d_array_c[1]/d_array_c[0]
        #bgpu[0] = fgpu[0]/d_array_c[0]
        #thomas_single((1,),(1,),(cgpu,bgpu,fgpu,u_array,d_array_c,Jin))

        u_oldcupy = cp.copy(u_cupy)
        #u_old     = cp.copy(u_array)

        Gn_array_GPU[:,n+1] = u_cupy # u_array

        # COMBINATION STEP
        U_big_GPU[k+1,:,n+1] = Fn_array_GPU[:,n+1] + Gn_array_GPU[:,n+1] - Go_array_GPU[:,n+1]

    Go_array_GPU = cp.copy(Gn_array_GPU)
    end_update = time.time()
    print("Time for update solution: ", end_update - begin_update)
    update_time.append(end_update - begin_update)
end_total = time.time()
print("Time for total solution: ", end_total - begin_total)
t_total = end_total - begin_total

U_big_CPU = cp.asnumpy(U_big_GPU)

average_fine = sum(fine_time)/len(fine_time)
average_update = sum(update_time)/len(update_time)

print("t_exact = ",t_test)
print("Mc = ", Mc)
print("Mfc = ", Mfc)
print("K = ", K)
print(" p = ", t_test/average_fine)
print("Sp = ", t_test/t_total)
print("St = ", Mc/(K*(Mc/(t_test/average_fine) + 2*Mc/Mfc)))
print("average_fine = ", average_fine)
print("average_update = ", average_update)
################################################################################
#
# ERRORS
#
################################################################################

U_big_CPU = cp.asnumpy(U_big_GPU)

errors = np.zeros((Mc,K+1))
for n in range(Mc):
        errors[n,0] = np.linalg.norm(U_big_CPU[0,:,n+1]-u_test[:,n+1])

for k in range(K):
        for n in range(Mc):
                errors[n,k+1] = np.linalg.norm(U_big_CPU[k,:,n+1]-u_test[:,n+1])

################################################################################
#
# PLOTS
#
################################################################################

t_span = np.linspace(0,T,Mf+1)
T_span = np.linspace(0,T,Mc+1)
xx,tt = np.meshgrid(x_inner,t_span)
XX,TT=np.meshgrid(x_inner,T_span)

subsub = [331,332,333,334,335,336,337,338,339]
subcount = 0

'''
plt.figure(0)
for m in range(Mc+1):
    plt.subplot(subsub[subcount])
    plt.plot(x_inner,u_test[:,m],'-xb', label = 'u_test')
    plt.plot(x_inner,U_big_CPU[K,:,m], 'or', label = 'U[8]')
    plt.plot(x_inner,U_big_CPU[0,:,m], 'og' , label = 'U[0]')
    if m == 6 or m == 7 or m == 8:
        plt.xlabel("space")
    if m == 0 or m == 3 or m == 6:
        plt.ylabel("temperature")
    if m == 0:
        plt.legend()
    plt.title('t = '+str(dT*m))
    subcount = subcount + 1

# Plotting u_test

norm2 = plt.Normalize(u_test.min(), u_test.max())
colors1 = cm.viridis(norm2(u_test.transpose()))
rcount, ccount, _ = colors1.shape
fig2 = plt.figure(2)
ax2 = fig2.gca(projection='3d')
surf = ax2.plot_surface(XX, TT, u_test.transpose(),
rcount=rcount, ccount=ccount, facecolors=colors1, shade=False)
surf.set_facecolor((0,0,0,0))
ax2.set_title("Test solution")
ax2.set_xlabel("space")
ax2.set_ylabel("time")
ax2.set_zlabel("temperature")
'''
# Plotting the errors

plt.figure(3)
for n in range(Mc):
        plt.semilogy(errors[n,:], label="t = "+str(n*dT+1))
plt.xlabel("Parareal iterations")
plt.ylabel("Logarithmic scale")
plt.legend()

# Plotting parareal
'''
fig4 = plt.figure(4)
```

```
norm4 = plt.Normalize(U_big_CPU[0,:,:].min(), U_big_CPU[0,:,:].max())
colors = cm.inferno(norm4(U_big_CPU[0,:,:].transpose()))
rcount, ccount, _ = colors.shape
ax4 = fig4.add_subplot(1,2,1, projection='3d')
surf = ax4.plot_surface(XX, TT, U_big_CPU[k,:,:].transpose(),
rcount=rcount, ccount=ccount, facecolors=colors, shade=False)
surf.set_facecolor((0,0,0,0))
ax4.set_title("U["+str(0)+"]")
ax4.set_xlabel("space")
ax4.set_ylabel("time")
ax4.set_zlabel("temperature")
norm5 = plt.Normalize(U_big_CPU[K,:,:].min(), U_big_CPU[K,:,:].max())
colors = cm.inferno(norm5(U_big_CPU[K,:,:].transpose()))
rcount, ccount, _ = colors.shape
ax5 = fig4.add_subplot(1,2,2, projection='3d')
surf = ax5.plot_surface(XX, TT, U_big_CPU[K,:,:].transpose(),
rcount=rcount, ccount=ccount, facecolors=colors, shade=False)
surf.set_facecolor((0,0,0,0))
ax5.set_title("U["+str(K)+"]")
ax5.set_xlabel("space")
ax5.set_ylabel("time")
ax5.set_zlabel("temperature")
'''
plt.show()
```

References

[1] A. Chrzeszczyk and J. Anders. *Matrix Computations on the GPU. CUBLAS, CUSOLVER and MAGMA by example*. Notes. 2017.

[2] M.J. Gander. *Time Parallel Time Integration [PDF Document]*. Retrieved from `https://www.unige.ch/~gander/`. Lecture Notes. Sept. 2018.

[3] M.J. Gander and S. Vandewalle. "Analysis of the Parareal Time-Parallel Time-Integration Method". In: *SIAM J. Sci. Comput.* 29. No. 2 (2007), pp. 556–578.

[4] N.J. Giordano and H. Nakanishi. "Oscillatory Motion and Chaos". In: *Computational Physics*. Pearson Education, Inc, 2006. Chap. 3, pp. 75–82.

[5] R. Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*. Pearson Education Limited, 2014.

[6] A. Isereles.

[7] J-L. Lions, Y. Maday, and G. Turinici. "Résolution d'EDP par un schéma en temps pararéel". In: *Académie des sciences* Série I (2001). Éditions scientifique & médicales Elsevier SAS, pp. 661–668.

[8] E.N. Lorenz. "Deterministic nonperiodic flow". In: *Journal of the atmospheric sciences* Vol. 20(2) (1963), pp. 130–141.

[9] X. Meng et al. "PPINN: Parareal physics-informed neural network for time-dependent PDEs". In: *Computer Methods in Applied Mechanics and Engineering, Elsevier* Vol. 370 (2019).

[10] T. Rauber and G. Rünger. "Algorithms for Systems of Linear Equations". In: *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2012. Chap. 8, pp. 417–499.

[11] T. Rauber and G. Rünger. "Parallel Computer Architecture". In: *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2012. Chap. 2, pp. 9–103.

[12] T. Rauber and G. Rünger. "Parallel Computer Architecture". In: *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2012. Chap. 7, pp. 387–415.

[13] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[14] S. Schöps, I. Niyonzima, and M. Clemens. "Parallel-In-Time Simulation of Eddy Current Problems Using Parareal". In: *IEEE TRANSACTIONS ON MAGNETICS* Vol. 54. No. 3 (2018).

[15] A. Waghamare and P. Seshaiyer. “Implementation of the Parareal Algorithm to Optimize Nanoparticle Transport in Porous Media Simulations”. In: *International Journal of Computational Methods* Vol. 16. No. 5 (2019). DOI: 10.1142/S0219876218400017.

[16] Y. Zhang et al. “A Hybrid Method for Solving Tridiagonal Systems on the GPU”. In: *GPU Computing Gems*. Elsevier Inc., 2012. Chap. 11, pp. 117–132.