Automating Heterogeneous Parallelism in Numerical Differential Equations

by

Utkarsh

B.Tech. (Double Major), Indian Institute of Technology Kanpur (2022)

Submitted to the Center of Computational Science and Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTATIONAL SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Utkarsh Center of Computational Science and Engineering May 20, 2024
Certified by:	Alan Edelman Professor of Applied Mathematics, Thesis Supervisor
Accepted by:	Youssef M. Marzouk Professor of Aeronautics and Astronautics Co-Director, Center for Computational Science and Engineering

Automating Heterogeneous Parallelism in Numerical Differential Equations

by

Utkarsh

Submitted to the Center of Computational Science and Engineering
on May 20, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTATIONAL SCIENCE AND ENGINEERING

ABSTRACT

Scientific computing is an amalgamation of numerical methods and computer science. Developments in numerical analysis have allowed stable and accurate numerical schemes, whereas computer algorithms have been successfully adopted to standard multicore systems of today, enabling parallelism. Combining efficient numerical algorithms with efficient parallelism presents a challenge mainly due to the independent development of these fields and is, therefore, typically solved on a domain-specific basis by domain experts. The development of general-purpose tools that integrate parallelism into algorithms, accessible through high-level languages, signifies the future direction for addressing computational demands across various domains.

This thesis work represents a culmination of efforts in general-purpose parallel numerical algorithms for solving differential equations. We make them accessible by choosing the Julia programming language to implement the high-level framework. Solving differential equations appears to be an intrinsically serial process due to progressive time-stepping that proves challenging to parallelize. Most of the approaches are linked to two broad categories; The first is the parallelism of the solver operations by making each solve faster, and the latter is the parallelism between the solves, i.e., solving multiple batches at a time. We automate the parallelization process in both these domains while keeping the algorithms general-purpose. Parallelization with different hardware accelerators, such as CPUs and GPUs, is also investigated.

Parallelism for sufficiently large stiff ODEs is traditionally linked to the parallelization of the matrix factorization stage. However, these methods still need to overcome the threading overhead for ODEs having less than approximately 200 states. We propose implementing adaptive-order, adaptive time-stepping stiff ODE solvers such as extrapolation methods, which can parallelize a single instance of an ODE solve even for small ODEs.

The other need for parallelization of ODE solvers arises from solving ODEs for batches of data, a typical workflow in inverse problems, global sensitivity analysis, and uncertainty quantification. Traditionally, GPU-accelerated ODE solvers were specially developed for high-dimensional PDE systems, which can be easily adapted for batched ODE solvers. The approach for parallelization is to convert an array-based ODE solver to work with GPU-based arrays. These approaches have shortcomings, such as implicit synchronization of time steps for all the ODEs and GPU overheads. We propose that these approaches can be improved significantly where GPU acceleration for ODE solvers is device-agnostic, general-purpose, and accessible from a high-level language.

Thesis supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgments

I am deeply grateful to everyone who supported and contributed to the completion of my master's thesis. Their invaluable assistance, encouragement, and guidance made this research possible.

I am especially grateful to my advisors, Prof. Alan Edelman and Dr. Chris Rackauckas. Alan has always encouraged me to think deeply about problems and helped me emphasize effective communication in my work, shaping me into a better researcher. Chris has been my supporter and motivator for the past several years, and I am always indebted to him for giving me the opportunity to collaborate with him. I hope to continue to carry his trust and faith in me. Their encouragement, subject expertise, and constructive criticism formed the foundation of this thesis.

I am also thankful to my peers at Julia Lab and MIT CSAIL. Engaging in discussions about technical computing helped me expand my horizons of knowledge, while their camaraderie and banter provided a necessary work-life balance. I would also like to thank my friends from the broader MIT community, who made this place feel like home with their presence.

I would like to extend my heartfelt gratitude to my parents, my mother, Soni, and my father, Vinod Kumar, whose unwavering support and indomitable spirit have been the foundation of my achievements. My father has always provided me with the best resources and encouragement to step into academia, while my mother has consistently supported my decisions with love. They have been constant pillars of support and motivation. I always cherish my time with my sister, Prachi Rajput, whose encouragement and companionship I deeply miss.

Lastly, I would like to thank everyone who has contributed to my work in any way possible. Completing this thesis would not have been possible without your support, faith in my abilities, and dedication. I also wish to thank the Almighty for maintaining their grace and blessings throughout this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Differential equations are fundamental to nearly all scientific research. Advances in scientific computing have enabled the large-scale simulation of these equations on computers. Efficiently scaling these techniques necessitates leveraging parallel computing, which has driven significant scientific discoveries in fields such as Biology, Physics, and Artificial Intelligence [1]–[4]. Despite the success of parallelism in enabling large-scale models and exascale simulations, many scientists and practitioners face barriers to adoption due to two main reasons:

1. They often lack access to the necessary parallel computing resources, typically working with standard x86 computers or, at best, modest GPUs.

2. Efficient utilization of these techniques requires technical expertise, which may be beyond the grasp of those who are less technically savvy.

Integrating parallelism with differential equation solvers is particularly challenging due to the inherently serial nature of the time-stepping integration procedure. Accelerating the simulation of Ordinary Differential Equations (ODEs) generally requires either the development of faster and more stable numerical methods or the use of special techniques within existing ODE solvers to exploit hardware parallelism. A common application in scientific computing involves solving large-scale ODE systems, which arise from the semi-discretization of Partial Differential Equations (PDEs), biological models, and ensemble problems.

Large-scale ODE systems are often parallelized using array-based approaches, such as broadcasts and GPU kernels for non-stiff systems and through matrix factorization for stiff systems. However, parallel computing remains unoptimized and often inaccessible for models that cannot overcome multi-threading overheads. These models, which are prevalent in the scientific community, are typically low-dimensional and expensive to simulate. Parallelism can significantly accelerate the forward pass of these models. One way to achieve this is through within-step parallelism, where special numerical methods are designed to allow parallel computation [5]–[7]. Another form of parallelism arises from ensemble simulations, where the same model is simulated for different initial conditions and parameters. This is useful for tasks such as global sensitivity analysis, uncertainty quantification, and inverse design [8]–[10]. However, parallel computing and numerical methods have historically been developed in isolation, often leading to trade-offs between computational efficiency and numerical accuracy.

This sets up our task: *"How can we efficiently enable parallel computing with efficient numerical methods?"* This thesis addresses this challenge through a two-step approach to overcome the barriers to adoption identified above.

First, we optimize specialized numerical methods, particularly extrapolation methods in ODEs, to enable parallelism for relatively smaller models with fewer than 200 states [5], [11]. This approach is detailed in Chapter 4. We unify heterogeneous acceleration by generating specialized GPU custom kernels for ODE solvers to address the need for parallelism in ensemble simulations. These custom kernels are 20 to 100 times faster than high-level approaches in popular Machine Learning libraries [12], [13]. We discuss our approach in detail and the suitability of the Julia programming language [14] for this task in Chapter 5.

Secondly, We automate this approach by developing or integrating open-source software such as DiffEqGPU.jl [15] and DifferentialEquations.jl [16] that allows the same high-level code to leverage parallelism with heterogeneous compute (CPUs and GPUs) without significant overheads, thus maintaining performance and composability. By building upon previous work and integrating these solutions, we aim to make efficient parallel computing and numerical methods accessible and practical for a wider range of users.

1.1 Originality of the Work

The works presented here are original works of the author or done in collaboration with others. The material presented here has been derived from the manuscripts that have appeared as pre-print, published at either a conference or a journal by the author, or the works of others which are duly cited.

1.1.1 Publications

This thesis contains works from these manuscripts published by the author in reverse chronological order:

- **Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms [15]**
  Utkarsh, Valentin Churavy, Yingbo Ma, Tim Besard, Prakitr Srisuma, Tim Gymnich, Adam R Gerlach, Alan Edelman, George Barbastathis, Richard D Braatz, Christopher Rackauckas
  Computer Methods in Applied Mechanics and Engineering

- **Parallelizing Explicit and Implicit Extrapolation Methods for Ordinary Differential Equations [7]**
  Utkarsh, Chris Elrod, Yingbo Ma, Konstantin Althaus, Christopher Rackauckas
  IEEE High-Performance Extreme Computing Conference (HPEC)

[15] was written and published during my Masters at MIT. [7] was also published during my time at MIT; however, the project was started before I joined MIT. A chapter for [7] has been included in the thesis to provide a complete story and motivation for [15], both of which are algorithms to parallelize solving of differential equations efficiently. [15] builds up on the different parallelization perspectives, which would lack coherence without the mention of [7].

1.2 Outline of the Thesis

The outline of this thesis is as follows:

1. Chapter 2 provides relevant background on numerical simulation of Ordinary and Stochastic Differential Equations.

2. Chapter 3 introduces ways to achieve parallelism in numerical methods, with greater emphasis on differential equation solvers.

3. Chapter 4 describes a way to parallelize a single ODE solve for low-dimensional stiff and non-stiff systems.

4. In Chapter 5, we address the other need for parallelism in differential equations, such as solving them in batches for different initial conditions or parameters. We demonstrate a performant, composable, and vendor-agnostic method that is orders of magnitude faster than current high-level implementations.

5. Chapter 6 highlights the development of open-source package, `DiffEqGPU.jl`, for GPU-based ODE solvers.

6. Chapter 7 concludes our thesis and provides some ideas for future work.

Chapter 2

Theory

2.1 Numerical Methods for Differential Equations

2.1.1 Non-stiff Ordinary Differential Equations (ODEs)

ODEs are models given by the evolution equation:

$$\frac{du}{dt} = f(u, p, t), \tag{2.1}$$

with the initial condition $u(t_0) = u_0$ over the time span (t_0, t_f), where u is the solution, p is the parameter, t is time, t_0 is the initial time, and t_f is the final time. In the subsequent sections of this article, the parameter p is omitted from the formulation to avoid confusion as it is not an important part of our numerical methods and algorithms. There exist many different methods for numerically solving ODEs [11], [17], though generally the most performant method is determined by a property known as the stiffness of the ODE, which is related to the pseudo-spectra of the Jacobian [18], [19].

One of the most common classes of solvers for ODE software are explicit Runge-Kutta methods [20], [21]. These methods are specified by a coefficient tableau $\{A, b, c\}$, with "s" stages and order "k", where $k \leq s$. It produces the approximation for $u_n = u(t_0 + nh)$ which is the solution at the current time step t_n, h is the time step ($t_{n+1} = t_n + h$), where h is the time-step, as

$$k_s = f\left(u_n + \sum_{i=1}^{s} a_{s,i} k_i, t + c_s h\right) \tag{2.2}$$

$$u_{n+1} = u_n + h \sum_{i=1}^{s} b_i k_i \tag{2.3}$$

Some examples of Runge-Kutta methods include dopri5 [22] and MATLAB's ODE suite ode45 [23].

For adaptive step-size control, the Runge-Kutta methods require an extra computation as $\tilde{u}(t+h) = u(t) + h \sum_{i=1}^{s} \tilde{b}_i k_i$, where $\tilde{b}_i$ are another linear combiners, which approximates the solution by one order less than the original solution. The local error estimate can be written as $E = \|\tilde{u}(t+h) - u(t+h)\|$ [11], [24]. Adaptivity ensures that error remains below certain tolerance, and these tolerances are absolute (atol) and relative (rtol). Mathematically, the proportion of error against tolerance is

$$q = \left\| \frac{E}{\text{atol} + \text{rtol} \cdot \max\{|u(t)|, |u(t+h)|\}} \right\|. \tag{2.4}$$

The step-size h is accepted for $q < 1$; otherwise, h is reduced and a new step is attempted. The new step-size in Runge-Kutta methods is proposed through proportional-integral control (PI-control) via $h_{\text{new}} = \eta q_{n-1}^{\beta_2} q_n^{\beta_1} h$, where β_1, β_2 are tuned parameters [11], q_{n-1} is the previous proportion error, and η is the safety factor.

2.1.2 Stiff Ordinary Differential Equations

Rosenbrock Methods

There exist various numerical methods for solving stiff ODEs [17], [23]. One of the most common algorithms used in various ODE solver packages, e.g., Julia, MATLAB, is known as the *Rosenbrock method* [17], [23], [25], [26]. The general formulas of an s-stage Rosenbrock method is given by

$$k_i = hf\left(u_n + \sum_{j=1}^{i-1} \alpha_{ij} k_j, t_n + \alpha_i h\right) + \beta_i h^2 \frac{\partial f}{\partial t}(u_n, t_n) + h \frac{\partial f}{\partial u}(u_n, t_n) \sum_{j=1}^{i} \beta_{ij} k_j, \tag{2.5}$$

$$u_{n+1} = u_n + \sum_{j=1}^{s} \delta_j k_j, \tag{2.6}$$

where u_n is the solution at the current time step t_n, u_{n+1} is the solution at the next time step t_{n+1}, h is the time step ($t_{n+1} = t_n + h$), $\alpha_{ij}, \beta_{ij}, \delta_j$ are the coefficients, and

$$\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij}, \tag{2.7}$$

$$\beta_i = \sum_{j=1}^{i} \beta_{ij}. \tag{2.8}$$

The Rosenbrock-type methods are ideally suited for GPU compilation because they are devoid of the typical Newton's method performed per step in stiff ODE integrators [17]. The Newton's method requires multiple linear solves and is computationally expensive due to repeated Jacobian calculation due to no reuse of previous matrix factorizations. Rosenbrock methods only require one Jacobian evaluation and have a constant number of linear solves per step, where the matrix factorization can be cached to achieve $\mathcal{O}(N^2)$ computational cost of the linear solves, where N is the dimension of the ODE. Since GPUs are efficient in performing multiple small tasks in parallel, Rosenbrock methods are expected to achieve massive speedups on GPUs in ensemble simulations.

Diagonally Implicit Runge-Kutta Methods

Another class of algorithms considered in this work is the Diagonally Implicit Runge-Kutta (DIRK) methods. The computation of a single step is given by

$$k_i = f\left(u_n + h\sum_{j=1}^{i} a_{ij}\kappa_j, t_n + c_i h\right), \tag{2.9}$$

$$u_{n+1} = u_n + h\sum_{i=1}^{s} b_i\kappa_i, \tag{2.10}$$

where a_{ij}, b_i, and c_j are the scalar constants. The class of DIRK methods considered in our benchmarking is the Explicit Singly Diagonal Implicit Runge-Kutta (ESDIRK) methods. ESDIRK methods are characterized by $a_{11} = 0$ and $a_{i,i} = \mu$, for some constant μ. Particularly, we are interested in Kvaerno methods, a class of A-L stable stiffly-accurate ESDIRK method [27]. The methods are suitable for benchmarking comparison as these are only available methods in other open-source software such as in Diffrax [28].

2.2 Extrapolation Methods

The extrapolation methods are variable-order and variable-step methods which generate higher precision approximations of ODE solutions computed at different time step-sizes. For the N^{th} current order of the algorithm, we generate $f(u,\ p,\ t+dt)$ at the current time-step dt for each order from 1 to N. For the approximation at $t+dt$, the algorithm chooses a subdividing sequence which discretizes into further fixed smaller steps between t and $t+dt$. Choosing a sequence of the form:

$$n_1 < n_2 < n_3 < n_4 < \cdots < n_N \tag{2.11}$$

Generates internal step-sizes $h_1 > h_2 > h_3 > h_4 > \cdots > h_N$ by $h_i = \frac{dt}{n_i}$. The subdividing sequences vary with order, having smaller time-steps in higher orders for finer resolution. The algorithm chosen for this is suitably an efficient implicit/explicit method between t and $t+dt$ is of p^{th} order.

The tabulation of 1^{st} N^{th} calculations generate starting-stage of extrapolation computation, denoted by:

$$\begin{aligned} &u_{h_i}(t+dt) = T_{i,1} \\ &i = j, j-1, j-2, \cdots, j-k+1. \end{aligned} \tag{2.12}$$

Extrapolation methods use the interpolating polynomial

$$p(h) = \tilde{u} - e_p h^p - e_{p+1}h^{p+1} - \cdots - e_{p+k-2}h^{p+k-2} \tag{2.13}$$

such that $p(h_i) = T_{i,1}$ to obtain a higher order approximation by extrapolating the step size h to 0. Concretely, we define $T_{j,k} := p(0) = \tilde{u} = u(t+dt) + \mathcal{O}(dt^{p+k})$. In order to find $\tilde{u}$, we can solve the linear system of k variables and k equations formed by equations (2.12) and (2.13). This conveniently generates an array of approximations with different orders which allows simple estimates of local error and order variability techniques:

$$\begin{array}{llll} T_{1,1} & & & \\ T_{2,1} & T_{2,2} & & \\ T_{3,1} & T_{3,2} & T_{3,3} & \\ \dots & \dots & \dots & \dots \end{array}$$

Aitken-Neville's algorithm uses Lagrange polynomial interpolation formulae [29], [30] to make: (2.13):

$$T_{j,k+1} = T_{j,k} + \frac{T_{j,k} - T_{j-1,k}}{\frac{n_j}{n_{j-k}} - 1} \tag{2.14}$$

Finally, the N^{th} order is $u(t + dt) = T_{N,N}$.

As shown in (2.11), the subdividing sequence should be positive and strictly increasing. Common choices are:

1. Harmonic [31]: $n = 1, 2, 3, 4, 5, 6, 7, 8 \dots$
2. Romberg [32]: $n = 1, 2, 4, 8, 16, 32, 64, 128, 256 \dots$
3. Bulrisch [32], [33]: $n = 1, 2, 3, 4, 6, 8, 12, 16 \dots$

The "Harmonic" sequence generates the most efficient load balancing and utilization of multi-threading in parallel computing which is discussed more in Section III.

2.2.1 Explicit Methods

Extrapolation Midpoint Deuflhard and HairerWanner

Both the algorithms use explicit midpoint method for internal step-size computations. The representation used here is in the two-step form, which makes the algorithm symmetric and has even powers in asymptotic expansion [34]

$$\begin{aligned} u_{h_i}(t_0) &= u_0 \\ u_{h_i}(t_1) &= u_0 + h_i f(u_0, p, t_0) \\ u_{h_i}(t_n) &= u_{h_i}(t_{n-2}) + 2h_i f(u_{h_i}(t_{n-1}), p, t_{n-1}) \end{aligned} \tag{2.15}$$

The difference between them arises from the step-sizing controllers. The ExtrapolationMidpointDeufhard is based on the Deuflhard's DIFEX1 [31] adaptivity behaviour and ExtrapolationMidpointHairerWanner is based on Hairer's ODEX [11] adaptivity behaviour.

The extrapolation is performed using barycentric formula which based on the lagrange barycentric interpolation [35]. The interpolation polynomial is given by:

$$\begin{aligned} w_j &= \prod_{i=1:N+1, i\neq j} \frac{1}{n_j^{-2} - n_i^{-2}} \\ \rho(h) &= \prod_{i=1:N+1} h - n_i^{-2} \\ p(h) &= \rho(h) \sum_{j=1}^{N+1} \frac{w_j}{h - n_j{}^{-2}} T_{j,1} \end{aligned} \tag{2.16}$$

Extrapolating the limit $h \to 0$, we get:

$$u(t+dt) = \rho(0) \sum_{j=1}^{N+1} \frac{w_j}{-n_j{}^{-2}} T_{j,1} \tag{2.17}$$

Where w_j are the extrapolation weights, $\rho(0)$ are the extrapolation coefficients and n_j denotes the subdividing sequence.

The choice for baryentric formula instead of Aitken-Neville is mainly due to reduced computation cost of ODE solution at each time-step [36]. The extrapolation weights w_j and coefficients $\rho(0)$ can be easily computed and stored as tableau's. The yields the computation cost to be $O(N_{max}^2)$ for the precomputation where N_{max} is the maximal order than can be achieved by the method.The extrapolation method of order N generates a method of error in order of $2(N+1)$ [11], [36].

It can be analysed that computational cost of with Aitken-Neville for $T_{N,N}$ is $\mathcal{O}(N^2 d)$ $(d(1+2+\cdots+N) = \mathcal{O}(N^2 d))$, where N is the extrapolation order and d is the dimension of the u [36]. The computation cost of (2.17) is simply $\mathcal{O}(Nd)$ (a linear combination all $T_{j,1}$ across d dimensions) [36].

Numerical Stability and Analysis: In comparison The numerical performance follows similar behaviour as that of Aitken-Neville in the subdividing sequences of Romberg and Bulrisch, where the absolute error decreases as extrapolation order q increases [37]. However, in the harmonic sequence, the numerical stability in both cases of extrapolation remains up-to as much as up-to 15 order and then it diverges [36].

2.2.2 Implicit Methods

We will consider implicit methods as basis for internal step-size calculation using subdividing sequences in extrapolation methods. They are widely used for solving stiff ODEs (maybe write more why implicit methods are suited for it).

Implicit Euler Extrapolation

The ImplicitEulerExtrapolation uses the Linearly-Implicit Euler Method for internal step-sizing. Mathematically:

$$\begin{aligned} (I - h_i J)(u_{h_i}(t_{n+1}) - u_{h_i}(t_n)) &= hf(u_{h_i}(t_n), p, t_n) \\ \therefore u_{h_i}(t_{n+1}) &= u_{h_i}(t_n) + (I - h_i J)^{-1} hf(u_{h_i}(t_{n+1}), p, t_n) \end{aligned} \tag{2.18}$$

where I and $J \approx \frac{\partial f(u_{h_i}, p, t)}{\partial u_{h_i}}$ is the $\mathbb{R}^{d \times d}$ identity and jacobian matrix respectively. Clearly, the method is non-symmetrical and consequently would have non-even powers of h global error expansion. The methods are $A(\alpha)$ stable with $\alpha \approx 90^o$ [17]. The extrapolation scheme used is Aitken-Neville (2.14).

Implicit Euler Barycentric Extrapolation

We experimented with barycentric formulas [35] to replace extrapolation algorithm in ImplicitEulerExtrapolaton. Since there are no even powers in global error expansion, the barycentric formula [35] needs to changed as follows:

$$\begin{aligned} w_j &= \prod_{i=1:N+1, i \neq j} \frac{1}{n_j^{-1} - n_i^{-1}} \\ \rho(h) &= \prod_{i=1:N+1} h - n_i^{-1} \\ p(h) &= \rho(h) \sum_{j=1}^{N+1} \frac{w_j}{h - {n_j}^{-1}} T_{j,1} \end{aligned} \tag{2.19}$$

Extrapolating the limit $h \to 0$, we get:

$$u(t + dt) = \rho(0) \sum_{j=1}^{N+1} \frac{w_j}{-{n_j}^{-1}} T_{j,1} \tag{2.20}$$

Consequently, the extrapolation method of order N generates a method of error in order of $N + 1$.

Implicit Hairer Wanner Extrapolation

For implicit extrapolation, symmetric methods would be beneficial to provide higher order approximations. The naive suitable candidate is the trapezoidal rule, but the resulting method is not stiffly stable and hence undesirable for solving stiff ODEs [38], [11].

G. Bader and P. Deuflhard [39], [31] developed the linearly-implicit midpoint rule with Gragg's smoothing [40], [41] as extension of popular Gragg-Bulirsch-Stoer (GBS) [33] method. The algorithm is given by:

$$\begin{aligned} (I - h_i J)(u_{h_i}(t_1) - u_0) &= h_i f(u_0, p, t_0) \\ (u_{h_i}(t_{n+1}) - u_{h_i}(t_n)) &= (u_{h_i}(t_n) - u_{h_i}(t_{n-1})) \\ &+ 2(I - h_i J)^{-1} h_i f(u_{h_i}(t_n), p, t_n) \\ &- 2(I - h_i J)^{-1}(u_{h_i}(t_n) - u_{h_i}(t_{n-1})) \end{aligned} \tag{2.21}$$

Followed by Gragg's Smoothing [40], [41]:

$$\begin{aligned} t &= t_0 + 2nh_i \\ T_{j,1} = S_{h_i}(t) &= \frac{u_{h_i}(2n+1) + u_{h_i}(2n-1)}{2} \end{aligned} \tag{2.22}$$

The Gragg's smoothing [40], [41] implementation in OrdinaryDiffEq.jl [1] improved the stability of algorithm resulting in lower f evaluations, time steps and linear solves. Consequently, it increased the accuracy and speed of the solvers.

The extrapolation scheme used is similar to extrapolation midpoint methods, namely the barycentric formula [35] (2.17). One of the trade-offs for the increased stability from Gragg's smoothing [40], [41] is the reduced order of error for the extrapolation method. As stated earlier, the barycentic formula provides and error of order $2(N+1)$ which gets reduced to $2(N+1)-1 = 2N+1$ in the case of Implicit Hairer Wanner Extrapolation [11], [39]. The algorithm requires to use even subdividing sequence and hence we are using multiples of 4 of the common sequences in the implementation. Furthermore, the convergence tests for these methods[2] are written in the test suite of OrdinaryDiffEq.jl.

2.2.3 Stochastic Differential Equations (SDEs)

SDEs are extensions of ODEs which include inherent randomness. SDEs are used as models in many domains such as quantitative finance [42], [43], systems biology [44], and simulation of chemical reaction networks [45]. SDEs are formally defined as

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t, \tag{2.23}$$

where W_t is a wiener process and dW_t is a Gaussian random variable $W_{t+dt} - W_t \sim \mathcal{N}(0, dt)$, where $\mathcal{N}(0, dt)$ is the normal distribution with zero mean and variance dt [46]. The general interest is in ensembles of SDE solutions for the purpose of calculating moments or simple analytical functions of moments, such as the mean and variance of the solution, and thus SDEs are a particularly strong application for ensemble parallelization of the solution process. For this reason, methods which have accelerated convergence for the calculation of the moments, known as high weak order solvers, have gained traction in the literature as a potentially performant method for numerically analyzing such solutions. One such class of methods are the stochastic explicit "s" stage Runge-Kutta methods:

$$\eta_j = \tilde{X}_t + h\sum_{j=1}^{s} \lambda_{ji} a(\eta_j, t + \mu_j h) + \sum_{k=1}^{m} \Delta\tilde{W}_n^k \sum_{j=1}^{s} \lambda_{ji}^k b(\eta_j, t + \mu_j h), \quad j = 1, \ldots, s,$$

$$\tilde{X}_{t+h} = \tilde{X}_t + h\sum_{j=1}^{s} \alpha_j a(\eta_j, t + \mu_j h) + \sum_{k=1}^{m} \Delta\tilde{W}_n^k \sum_{j=1}^{s} \beta_j^k b(\eta_j, t + \mu_j h) + R,$$

[1] https://github.com/SciML/OrdinaryDiffEq.jl/pull/1212

[2] https://github.com/SciML/OrdinaryDiffEq.jl/blob/master/test/algconvergence/ode_extrapolation_tests.jl

where $\alpha_j, \beta_j^k, \mu_j, \lambda_{ij}, \gamma_{ij}^k$ are the constants that define the particular stochastic Runge-Kutta method and R is the fit term [46], [47]. Adaptive time-stepping techniques using similar rejection sampling and PI-controller approaches to ODEs have been adapted to SDE solver software [48].

Chapter 3

Parallelism in Differential Equations

Numerical methods for solving differential equations exploit parallelism at different levels. Briefly, the levels of parallelism in hardware exist as:

1. Instruction-Level Parallelism (ILP)
2. Data-Level Parallelism
    (a) Single Instruction Multiple Data (SIMD)
    (b) Graphics Processing Units (GPUs)
3. Thread-Level Parallelism

This thesis will mainly focus on some ideas for Thread-Level Parallelism and GPUs, and the treatment for other types of parallelism is out of the scope of the thesis.

3.1 Within-step Parallelism

Step parallelism in the context of solving ODEs involves executing different stages or steps of the numerical solution process concurrently. This can significantly speed up the computation, especially for large-scale problems or systems of ODEs. Avenues for parallelism exist in Runge-Kutta methods, where the stage evaluation depends on $f(u, p, t)$ and can somewhat be parallelized. For explicit Runge-Kutta methods, later stages often depend on the results of earlier stages, which limits parallelism. However, within each stage, there might be opportunities for parallel computations, especially for systems of ODEs [49]. A common solver available widely is PETSc, which provides tools for parallel computation, including support for parallel Runge-Kutta methods [50]. Similarly, there exist avenues of parallelism in extrapolation methods [5], [6], which we will describe in Chapter 4 in greater detail.

In the context of stiff systems, the most computationally expensive part is the linear solve. Matrix factorization provides efficient computation of the inverse, which scales at $\mathcal{O}(N^3)$. Hence, most of the parallelization is linked to parallel factorization, commonly provided by LAPACK [51]. LAPACK routines are structured to maximize the use of the Basic Linear Algebra Subprograms (BLAS) for their computations. Specifically, LAPACK is initially designed to leverage Level 3 BLAS—Fortran subprogram specifications for various types of matrix multiplication. The coarse granularity of Level 3 BLAS operations enhances efficiency on many high-performance computers, especially when manufacturers provide specially optimized implementations. These optimized implementations are sufficient for within-step parallelism, especially for large systems, where the multi-threading overhead is amortized.

3.2 Parallel-in-time (PIT) Methods

Parallel-in-time (PIT) methods [52], [53] are a type of time-parallel technique that can help ODEs solve cases by simultaneously stepping the ODE at multiple time points. The basic idea of PIT methods revolves around splitting a large time interval of integration, where these splits, i.e., small time intervals, can be solved in parallel. This is followed by iterative correction of the solutions until convergence. Briefly, the method consists of two key solvers: Coarse and Fine solver. The coarse solver is responsible for generating initial approximations and is relatively fast, however being less accurate. A projection to a more accurate solution is computed using a fine solver, which is trivially more computationally expensive. The method scales with the number of processors available and is suitable for integrating large-time scales.

3.3 GPU Computing for Differential Equations

Graphics Processing Units (GPUs) differ remarkably from CPUs, and hence, avenues for parallelism are somewhat specific and restricted. GPUs follow the Single Threaded Multiple Thread (SIMT) form of parallelism, where there are thousands of workers capable of performing relatively less complex and similar tasks on a thread. Broadly, there two different ways to accelerate differential equations with GPUs:

3.3.1 Implicit Parallelism

In the context of ODEs, implicit parallelism occurs when the vector field, i.e., the RHS or the $f(u, p, t)$ is relatively expensive to compute. The caveat in this parallelization is that 'f' needs to be very structured, i.e., constructs used to compute f have efficient parallelization algorithms, such as matrix-matrix multiplication. A very common example, in this case, is the evaluation of neural networks in Neural ODEs [3], which has a trivial parallel forward-pass evaluation. Another more relevant example of scientific computing is the evaluation of semi-discretized PDEs, which results in large ODEs and has the structure to exploit due to the semi-linear form.

3.3.2 Batched Evaluation

Another need for parallelism arises from ensemble simulations, where the ODE is required to evaluate different initial conditions or parameters. The pre-requisite of a huge number of states is not necessary here, as parallelization acts on the independent evaluation of the ODE. Traditionally, GPU-accelerated ODE solvers were specially developed for high-dimensional

PDE systems. They can be easily adapted for batched ODE solvers. The approach for parallelization was to convert an array-based ODE solver to work with GPU-based arrays. PDE cases have completely different performance characteristics than the GPU ensemble cases since they focus on solving one big (structured) ODE rather than many small ODEs. However, the tooling required to support the GPU acceleration of PDE cases is the same as what is required for the synchronized ensemble form, which is the reason why major ODE solver libraries that focus heavily on PDE support (Sundials [54]) end up supporting the GPU ensembling through the array approach (which then results in the downsides noted above). However, as we will showcase in Chapter 5 ; one should use completely separate approaches (the kernel generation) as otherwise, there will be a massive loss to performance since the inherent synchronization of the array-based methods is only optimized (and required) in the PDE case.

Chapter 4

Parallelizing Explicit and Implicit Extrapolation Methods for Ordinary Differential Equations

4.1 Introduction

The numerical approximation of ordinary differential equations (ODEs) is a naturally serial time stepping process, meaning that methods for parallelizing the solution of such ODEs requires either tricks or alternative solvers in order to exploit parallelism. The most common domain, and one of the most common applications in scientific computing, is the parallel solution of large-scale systems of ODEs (systems of millions or more equations) which arise from the semi-discretization of partial differential equations. For such cases, the size of the system allows for parallel efficiency to be achieved by parallelizing some of the most expensive computations, such as implicit parallelism of BLAS [55]/LAPACK [51], sparse linear solvers, or preconditioner computations, on compute clusters and with heterogeneous GPU+CPU compute. Commonly used open source softwares such as Sundials [54] and DifferentialEquations.jl [16] help users automate the parallelism in such cases. However, for this manuscript we look in the opposite direction at parallelism for small systems of ODEs (< 200).

Parallel computing on small systems of equations comes with a separate set of challenges, namely that computations can easily become dominated by overhead. For this reason, most software for accelerating the parallel solution of ODEs focus on the parallelization of generating ensembles of solutions, i.e. generating the solution to a small system of ODEs over a set of parameters or initial conditions. Examples of this include the ensembles interface in DifferentialEquations.jl [16]. However, in many cases the ensemble form may not be easy to exploit. One key example is in parameter estimation or model calibration routines which are typically done with some gradient-based optimization technique and requires one realization of the model at a given parameter before advancing. Given questions on the Julia [14] Discourse demonstrate that the vast majority of users both typically solve < 200 ODE systems and have multi-core compute available (e.g. Core i5/i7 chips with 2-16 processors), investigated whether the most standard ODE solve cases could benefit from some form of parallelism.

Prior research has developed a large amount of theory around potential solvers for such problems which fall into the categories of parallel-in-time (PIT) methods and within-step parallelism methods [56]. Parallel-in-time methods address the issue by effectively stepping the ODE at multiple time points simultaneously, similar to the ensemble approach, and impose a nonlinear system constraint to relax the initial conditions of the future points to arrive at a sufficiently smooth solution. An open source software for PIT methods, XBraid, does exist but focuses on large compute hardware [57]. Documents from the XBraid tutorials [58] estimate current PIT methods outperform classical solvers at ~ 256 cores[1], making them impractical for standard compute hardware. Thus while PIT methods are a promising direction for accelerating ODE solving for the exascale computers targeted by the XBraid project [57], these methods do not suffice for the ≤ 16 core nature of standard consumer computing devices.

However, promising prior results exist within the within-step parallelism research. Nowak 1998 parallelized the implicit extrapolation methods and showed that the parallelization did improve performance [5], though its tests did not compare against multithreaded (sparse) factorizations like is seen in modern suites such as UMFPACK [59] in SuiteSparse [60], no comparison was made against optimized solver softwares such as CVODE [54] or LSODA [61], and no open source implementation exists for this method. Additionally, Ketcheson (2014) [6] demonstrated that within-step parallel extrapolation methods for non-stiff ODEs can outperform dop853, an efficient high-order Runge-Kutta method [11] . Two notable issues with this study are that (a) no optimized open source software exists for this method for further investigation, (b) the study did not investigate whether such techniques could be useful for stiff ODEs, which tend to be the most common type of ODEs for many applications from biology and chemistry to engineering models. These results highlight that it may be possible to achieve state-of-the-art (SOA) performance by exploiting parallelism, though no software is readily available.

In this work we build off of the prior work in within-method parallel extrapolation to build an open source software in Julia [14] targeting < 200 ODE systems on standard compute architectures. We demonstrate that this new solver is the most efficient solver for this class of equations when high accuracy is necessary, and demonstrate this with benchmarks that include many methods from DifferentialEquations.jl [16], SUNDIALS [54] (CVODE), LSODA [61], and more. To the best of our knowledge, this is the first demonstration that a within-step parallel solver can be the most efficient method for this typical ODE and hardware combination.

This chapter borrows its contents from previously appeared publication: Utkarsh, C. Elrod, Y. Ma, K. Althaus and C. Rackauckas, "Parallelizing Explicit and Implicit Extrapolation Methods for Ordinary Differential Equations," 2022 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2022, pp. 1-9, doi: 10.1109/HPEC55821.2022.9926357. [7]

[1]Private correspondences and tests with DifferentialEquations.jl also confirm a similar cutoff point with PIT methods

4.2 Extrapolation Methods

Our exposition of extrapolation methods largely follows that of Hairer I [11] and II [17], and [36]. Take the ODE:

$$u' = f(u, p, t), \tag{4.1}$$

with some known initial condition $u(t_0) = u_0$ over a time span $t \in (t_0, t_f)$. Extrapolation methods essentially create a series of approximations of $u(t + dt)$ with cheaper numerical algorithms such as explicit and implicit euler. These approximations are computed for different "internal step-sizes" denoted as $T_{1,j}$ with $h = \frac{dt}{n_j}$ $\forall j = 1, \ldots, N$, where N is the current approximation order. From these set of computation points, we compute interpolation polynomial which essentially allows us to extrapolate to the limit $h \to 0$, to compute N^{th} order approximation for the $u(t + dt)$. The process creates a tableau of the form:

$$\begin{array}{llll} T_{1,1} & & & \\ T_{2,1} & T_{2,2} & & \\ T_{3,1} & T_{3,2} & T_{3,3} & \\ \cdots & \cdots & \cdots & \cdots \end{array}$$

Where we extrapolate to get $u(t + dt) = T_{N,N}$ with the standard theory of interpolation polynomials [35]. We refer the reader to the Section 2.2 in Chapter 2, which discusses extrapolation methods for ODEs in greater detail.

4.2.1 Adaptive time-stepping and order of algorithms

The methods follow a comprehensive step-sizing and order selection strategy. The adaptive time-stepping is mostly followed on the lines of Hairer's ODEX [11] adaptivity behaviour. The error at k^{th} order is expressed as:

$$err_k = \|T_{k,k-1} - T_{k,k}\| \tag{4.2}$$

For the optimal order selection, the computation relies on the "work calculation" which is stage-number A_k per step-size h_k, $\frac{A_k}{h_k}$.

The stage-number includes the no. of $f(u, p, t)$ RHS function evaluations, jacobian matrix evaluation, and forward and backward substitutions. These are pre-computed according the subdividing sequence and stored as a cache. More details about these calculations can be found here.[2] These work calculations form the basis of order-selection. The order-selection is restricted between the window $(k - 1, k + 1)$ and appropriate conditions are passed to convergence monitor. Briefly describing, it checks the convergence in the window and subsequently accepts the order which has the most significant work reduction during increasing/decreasing the order and $error < 1$. Detailed description can be found at order and step-size control in [11].

[2] https://github.com/utkarsh530/DiffEqBenchmarks/blob/master/Extrapolation_Methods/Extrapolation_Methods.pdf

4.3 Parallelization of the Algorithm

4.3.1 Choosing Subdividing Sequences for Static Load Balancing

The exploit of parallelism arises from the computation of $T_{j,1}$. k evaluations of $T_{j,1}$ are done to find the solution $u(t + dt)$. These evaluations are independent and are thus parallelizable. However, because of the small systems being investigated, dynamic load balancing overheads are much too high to be relied upon and thus we tailored the parallelism implementation to make use of a static load balancing scheme as follows. Each $T_{j,1}$ requires n_j calls to the function f, and if the method is implicit then there is an additional n_j back substitutions and one LU factorization. While for sufficiently large systems the LU-factorization is the dominant cost due to its $O(n^3)$ growth for an n ODE system, in the < 200 ODE regime with our BLAS [55]/LAPACK [51] implementations[3] we find that this cost is close enough to the cost of a back substitution that we can roughly assume $T_{j,1}$ requires n_j "work units" in both the implicit and explicit cases method cases.

If we consider the multiples of harmonic subdividing sequence is 2, 4, 6, 8, 10, 12, ... then the computation of $T_{j,1}$ needs $2j$ work. Because $T_{k,k}$ is a numerical approximation of order $p = 2k$ we can simply give each $T_{j,1}$ a processor if k processors are available. The tasks would not finish simultaneously due to different amount of work. With Multiple Instruction and Multiple Data Stream (MIMD) processors, multiple computations can be loaded to one processor to calculate say $T_{1,1}$ and $T_{k-1,1}$ on a single processor, resulting in $2 + 2(k - 1) = 2k$ function evaluations . Consequently, each processor would contain $T_{j,1}$ and $T_{k-j,1}$ computations and we require $\frac{k}{2}$ processors. This is an effective load-balancing and all processes would finish simultaneously due to an approximately constant amount of work per chunk.

Julia provides compose-able task-based parallelism. To take advantage of the regularity of our work and this optimized static schedule, we avoid some of the overheads associated with this model such as task-creation and scheduling by using Polyester.jl's [62] `@batch` macro, which allows us to run our computation on long-lived tasks that can remain active with a spin-lock between workloads, thereby avoiding the need for rescheduling. We note that this load-balancing is not as simple in sequences of Romberg and Bulirsch, and thus in those cases we multi-thread computations over constant p processors only. For this reason, the homonic subdividing sequence is preferred for performance in our implementation.

4.3.2 Parallelization of the LU Factorization

One of the most important ways multi-threading is commonly used within implicit solvers is within the LU factorization. The LU factorization is multi-threaded due to expensive $O(N^3)$ complexity. For Jacobian matrices smaller than 100×100, commonly used BLAS/LAPACK implementations such as OpenBLAS, MKL, and RecursiveFactorization.jl are unable to achieve good parallel performance. However, the implicit extrapolation methods require solving linear systems with respect to $I - h_iJ$ for each sub-time step h_i. Thus for the within-method parallel implicit extrapolation implementation we manually disabled the internal LU

[3]RecursiveFactorization.jl notably outperforms OpenBLAS in this regime.

factorization multi-threading and parallelized the LU factorization step by multi-threading the computation of this ensemble of LU factorizations. If the chosen order is sufficiently then a large number of h_i would be chosen and thus parallel efficiency would be achieved even for small matrices. In the results this will be seen as the lower tolerance solutions naturally require higher order methods, and this is the regime where the implicit extrapolation methods demonstrably become SOA.

4.4 Benchmark Results

The benchmarks we use work-precision diagrams to allow simultaneous comparisons between speed and accuracy. Accuracy is computed against reference solutions at 10^{-14} tolerance. All benchmarks were computed on a AMD EPYC 7513 32-Core processor ran at 8 threads[4].

4.4.1 Establishing Implementation Efficiency

To establish these methods as efficient versions of extrapolation methods, we benchmarked these new implementations against standard optimized extrapolation implementations. The most widely used optimized implementations of extrapolation methods come from Hairer's FORTRAN suite [63], with ODEX [11] as a GBS [33] extrapolation method and SEULEX as a linear implicit Euler method. Figure 4.1 illustrates that our implementation outperforms the ODEX [11] FORTRAN method by 6x on the 100 Independent Linear ODEs problem. And similar to what was shown in Ketcheson 2014 [6], we see that the parallelized extrapolation methods were able to outperform the dop853 high order Runge-Kutta method implementation of Hairer by around 4x (they saw closer to 2x). However, when we compare the explicit extrapolation methods against newer optimized Runge-Kutta methods like Verner's efficient 9th order method [64], we only matched the SOA[5], which we suspect is due to the explicit methods being able to exploit less parallelism than the implicit variants. Thus for the rest of the benchmarks we focused on the implicit extrapolation methods. In Figure 4.2 we benchmarked the implicit extrapolation methods against the Hairer SEULEX [63] implementation on the ROBER [65] problem and demonstrated an average of 4x acceleration over the Hairer implementations, establishing the efficiency of this implementation of the algorithm.

4.4.2 State-Of-The-Art Performance for Small Stiff ODE Systems

To evaluate performance of extrapolation methods, we benchmarked the parallel implicit extrapolation implementations on the following set of standard test problems [17]:

1. the Robertson equation [65], 3 ODEs

[4]While more threads were available, we did not find more threads to be beneficial to accelerating these computations.

[5]In the shown Figure 4.1, the methods are approximately 1.5x faster, though on this random linear ODE benchmark, different random conditions lead to different performance results, averaging around 0.8x-2x, making the speedup generally insignificant

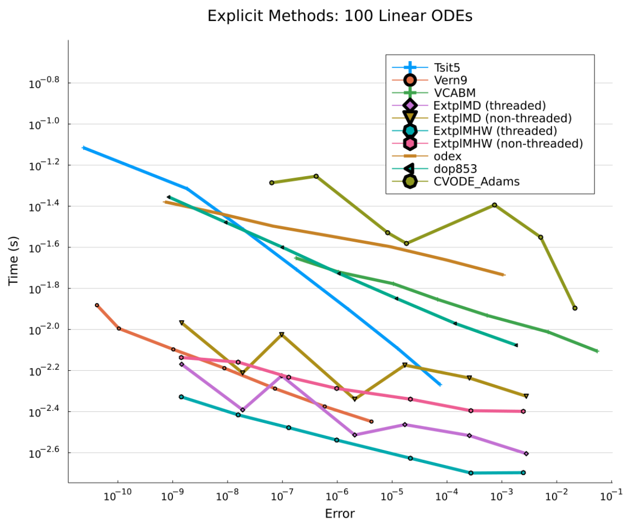

Figure 4.1: Benchmark on the 100 linear ODE problem [A.0.2].

2. Orego (Oregonator) [66], [67], 3 ODEs
3. Hires [68], 8 ODEs
4. Pollution [69], 20 ODEs
5. QSP [70], 109 ODEs

Figures 2-6 show the Work-Precision Diagrams with each of the respective problems with low tolerances. Implicit Euler Extrapolation outperforms other solvers with lower times at equivalent errors. However, very small systems (3-7 ODEs) show a disadvantage when threading is enabled, indicating that overhead is not overcome at this size, but the non-threaded version still achieves SOA by about 2x over the next best methods (Rodas4 and radau) and matching lsoda on HIRES. By 20 ODEs, the multi-threaded form becomes more efficient and is the SOA algorithm by roughly 2.5x. But by the 109 ODEs of the QSP model, the multi-threaded form is noticeably more efficient than the non-multithreaded and the extrapolation methods outperformed lsoda BDF and CVODE's BDF method by about 2x. Together these results show the implicit extrapolation methods as SOA for small ODE systems, with a lower bound on when multi-threading is beneficial.

4.5 Discussion

If one follows the tutorial coding examples for the most common differential equation solver suites in high-level languages (R's deSolve [71], Python's SciPy [72] , Julia's DifferentialEquations.jl [16], MATLAB's ODE Suite [73]), the vast majority of users in common case of <200 ODEs will receive no parallel acceleration even though parallel hardware is readily available[6]. In this manuscript we have demonstrated how the theory of within-method

[6]Unless the f function is sufficiently expensive, like when it is a neural network evaluation

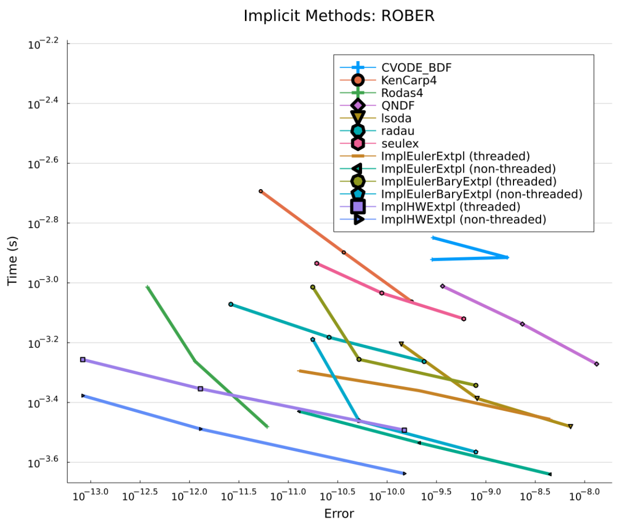

Figure 4.2: Benchmark on ROBER Problem with low tolerances [A.0.2].

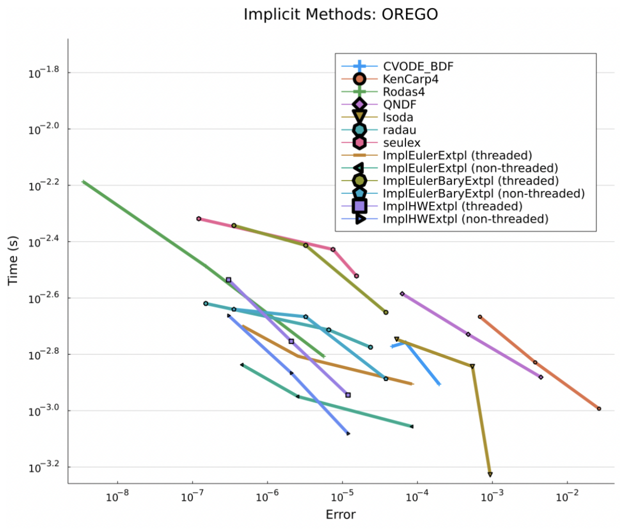

Figure 4.3: Benchmark on Orego Problem with low tolerances [A.0.2].

parallel extrapolation methods can be used to build a method which achieves SOA for the high accuracy approximation regime on this typical ODE case. We distribute this method as open source software: existing users of DifferentialEquations.jl [16] can make use of this technique with no changes to their code beyond the characters for the algorithm choice. As such, this is the first demonstration the authors know of for automated acceleration of ODE solves on typical small-scale ODEs using parallel acceleration.[7]

Lastly, the extrapolation methods were chosen as the foundation because their arbitrary order allows the methods to easily adapt the number of parallel compute chunks to different core counts. This implementation and manuscript statically batched the compute in a way that is independent of the specific core count, leading to generality in the resulting method.

[7]The codes can be found at:
https://github.com/utkarsh530/ParallelExBenchmarks.jl

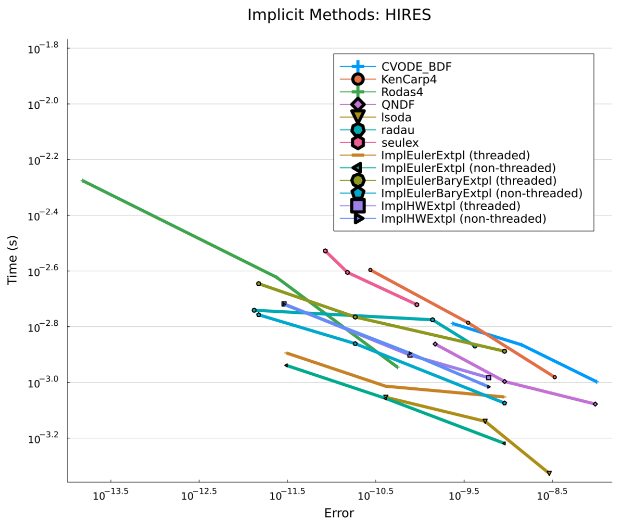

Figure 4.4: Benchmark on Hires Problem with low tolerances [A.0.2].

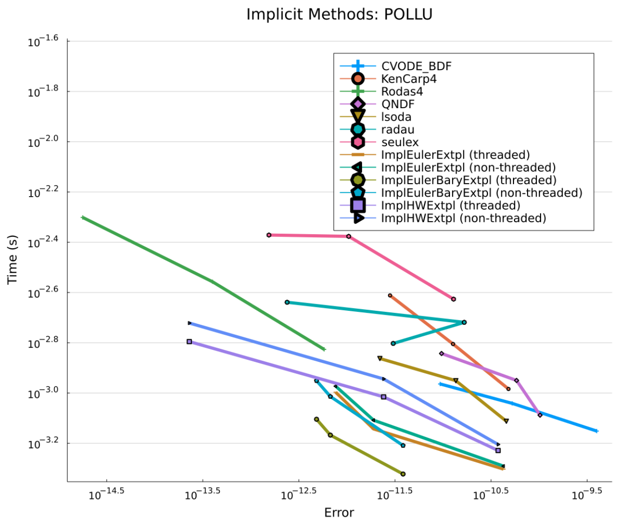

Figure 4.5: Benchmark on Pollution Problem with low tolerances [A.0.2].

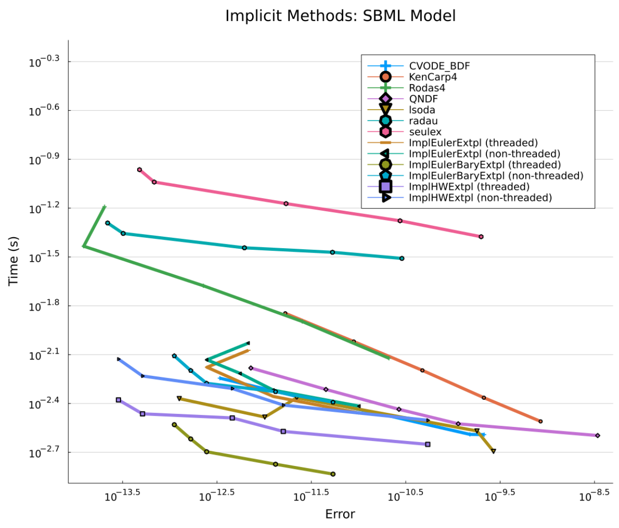

Figure 4.6: Benchmark on QSP model with low tolerances [A.0.2].

Chapter 5

Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms

5.1 Introduction

Solving ensembles of the same differential equation with different choices of parameters and initial conditions is common in many technical computing scenarios such as solving inverse problems [74], performing uncertainty quantification [8], [9], [75], and calculating global sensitivity analysis [9], [10]. While such a naturally parallel problem lends itself to being well-suited for acceleration via GPU hardware, the programming requirements have traditionally been a barrier to the adoption of GPU-parallel solvers by scientists and engineers who are less programming savvy. The core difficulty of targeting GPUs with general ODE solver software is that the definition of the ODE is a function given by the user. Thus, high-level ODE solver software has generally consisted of higher-order functions which take as input a function written in a high-level language such as MATLAB [23], Python (SciPy [72]), or Julia (DifferentialEquations.jl [26]) to reduce the barrier to entry for scientists and engineers. In order to target GPUs, previous software such as MPGOS [76] has required users to rewrite their models in a kernel language such as CUDA C++, which has thus traditionally kept optimized GPU usage out of reach for many scientists. In order to get around this barrier, some software for general GPU-based ODE solving in high-level languages has targeted array-based interfaces such as found in machine learning libraries like PyTorch [77] and JAX [12]. However, we demonstrate in this manuscript that such an approach is orders of magnitude less performant than generating model-specific ODE solver kernels.

In this manuscript, we demonstrate a performant, composable, and vendor-agnostic method for model-specific kernel generation to solve massively parallel ensembles of ordinary differential equations (ODEs) and stochastic differential equations (SDEs) on GPUs. The ODE solvers support both stiff and non-stiff ODEs, allowing a wide range of compatibility with different classes of models. Our software transforms code that targets a widely used differential equation solver library in a high-level language (Julia's DifferentialEquations.jl [26]) and automatically generates optimized GPU kernels without requiring code changes by the end user. We demonstrate an array-based parallelism approach and an automated kernel generation approach, which give a trade-off in extensibility and performance. We demonstrate that the kernel generation achieves state-of-the-art performance by on average outperforming hand-optimized CUDA-C++ kernels provided by MPGOS, and performing 20–100× faster than the vectorized map (vmap) approach implemented in JAX and PyTorch. We showcase the vendor-agnostic aspect of our approach by benchmarking the results on many major GPU vendors' cards like NVIDIA, AMD, Intel (oneAPI), and Apple Silicon (Metal) and demonstrate the composability with MPI to enable distributed multi-GPU workflows. We show that these solvers are fully featured, supporting event handling, forward and reverse (adjoint) automatic differentiation, and incorporation of datasets via the GPU's texture memory. Together, this software allows scientists to target all major GPU platforms without loss of performance.To summarize, the key contributions of this manuscript are:

- A feature-rich open-source library of massively parallel GPU ODE and SDE solvers without trading the high-level interface and performance, allowing composability with the rest of the numerical computing ecosystem.
- According to the author's knowledge, the first known implementation to be completely vendor-agnostic provides a roadmap of the required groundwork for numerical software to achieve vendor-agnosticism.
- Increased the performance of GPU parallelized stiff ODE solvers, which is enabled by leveraging and extending automatic differentiation for GPUs by static compilation.
- Multiple algorithm choices lead to insights in performance engineering for the problem: We showcase that traditional parallelization of numerical solvers such as LSODA, which are known to be performant for large ODEs, are orders of magnitude slower at solving multiple and small ODEs together on GPUs due to their heavy use of branching behavior. Alternative methods (Rosenbrock methods) are thus demonstrated as more suitable for the use of ensemble GPU parallelism.

This chapter borrows its contents from previously appeared publication: Utkarsh, U., Churavy, V., Ma, Y., Besard, T., Srisuma, P., Gymnich, T., Gerlach, A.R., Edelman, A., Barbastathis, G., Braatz, R.D. and Rackauckas, C., 2024. Automated translation and accelerated solving of differential equations on multiple GPU platforms. Computer Methods in Applied Mechanics and Engineering, 419, p.116591 [15].

5.2 Related Work

While researchers have used GPUs to accelerate computations extensively in applications including molecular simulation, biological systems, and physics [78]–[81], these implementations are generally CUDA kernels written for the specific models and thus are not general ODE solver software. In order to simplify the targeting of GPUs with general ODE solver software, previous attempts have generally targeted hardware using array abstraction frameworks such as ArrayFire [82], Thrust [83], VexCL [84], JAX [12], and PyTorch [77]. These frameworks allow the user to adapt code written on high-level array abstractions and generate a highly optimized code to backends such as OpenCL [85] and CUDA [86]. Boost's odeint [87]–[89] allows direct calls to ODE solvers that, without any modification, work with GPU backends such as CUDA and OpenCL. JAX's Diffrax [28] generates solvers for ensembles of ODEs on GPUs via JAX's vectorized map functionality (`vmap`). PyTorch's torchdiffeq [3] allows the defining of ensembles of ODEs directly with GPU-based arrays, although their `vmap` provided by functorch support with ODEs is still primitive as of April 2023.

Recent results have demonstrated that using array-based abstractions for generating GPU-parallel ODE ensemble solvers greatly lags in performance compared to the state of the art. In particular, MPGOS [76] demonstrated that ODEINT was 10–100$\times$ slower than purpose-written ODE solver kernels written in CUDA. In order to achieve this performance, MPGOS requires that the user write CUDA C++ kernels for the ODE definitions, which are then compiled into the solver to reduce the kernel call overhead. Similar results were seen with culsoda [78], a CUDA translation of the widely used LSODE solver [90], [91], which was similarly limited due to requiring ODE models to be written in CUDA and compiled into the kernels.

5.3 The GPU ecosystem in Julia and cross-platform GPGPU programming

Using high-level languages to program hardware accelerators traditionally means either using a library approach or a domain-specific-language (DSL) approach. The library approach focuses on providing array abstractions to call optimized high-level operators written and optimized in another language. Prime examples of this approach are ArrayFire [82] and CuNumpy [92]. Often these systems provide some mechanism of user extendability, but often it is in terms of the underlying system language and not the host language. DSL approaches, such as JAX [12], embed a new language into the host language that provides domain-specific concepts and limits expressibility to a subset of operations that are representable by the DSL. This requires the user to rewrite their application in ways that are compatible with the DSL.

Compiling a high-level language directly to hardware accelerators like GPUs is challenging because these languages often rely on accessing the run-time library, managed memory and garbage collection, interpreted execution, and other constructs that are difficult to use on GPUs or are even in conflict with the hardware design. Numba [93] and JuliaGPU [94] retarget a subset of the language for execution on hardware accelerators. In contrast to Numba, which is a reimplementation of Python, JuliaGPU repurposes Julia's existing CPU-oriented compiler for the purpose of generating code for GPUs.

Over time this has allowed the subset of the language that is directly executable on the GPU to grow and provide the basis for an effective, performant, and highly accessible programming model for GPUs. This model spans from low-level GPU kernel programming with direct access to advanced hardware features to the high-level array abstractions [95] provided by Julia.

5.3.1 Supporting multiple GPU platforms

Originally JuliaGPU only supported hardware accelerators by NVIDIA (CUDA). As hypothesized [94], the same approach could be extended to target other hardware platforms. Instead of re-implementing a full-fledged compiler for each new platform, the common infrastructure pieces were abstracted into a single unified compiler interface GPUCompiler.jl [96] and a unified array interface GPUArrays.jl [97]. Despite its name, GPUCompiler.jl is not limited to only GPU platforms and is also used to target non-GPU accelerators and specific CPU platforms.

With GPUCompiler.jl offering reusable functionality to configure the Julia compiler and LLVM providing the ability to generate high-quality machine code, Julia is well-positioned to target different accelerator platforms. Most major GPU platforms are supported: CUDA.jl [94] for NVIDIA GPUs using the CUDA toolkit, AMDGPU.jl [98] for AMD GPUs through ROCm, oneAPI.jl [99] for Intel GPUs with oneAPI, and Metal.jl [100] for Apple M-series GPUs with the Metal libraries. These backends are relatively simple and can be developed and maintained by small teams. Meanwhile, other languages and frameworks often struggle to provide native support for all but the most popular platforms. In the case of Python, for example, adding a Numba backend involves significant effort, and as such, Apple GPUs are not yet supported.[1] Similarly, as of May 2023, JAX does not support AMD [2] or Apple GPUs [3], because it requires special support in the Accelerated Linear Algebra (XLA) compiler. Unless there is sizable traction, scientific computing with these languages is not able to leverage different GPU vendors where that could have been beneficial for, e.g., HPC and AI/ML workloads [101], [102].

To facilitate working with multiple GPU platforms, Julia offers a powerful array abstraction that makes it possible to write generic code. The abstractions are implemented by each backend, either using native kernels or by reusing existing functionality. For performance reasons, common operations such as matrix multiplication are implemented by dispatching to vendor-specific libraries such as CUBLAS for NVIDIA GPUs and Metal's Performance Shaders for Apple GPUs. Higher-order operations such as `map`, `broadcast`, and `reduce` are implemented using native kernels. This makes it possible to compose them with user code, often obviating the need for custom kernels. Work by Besard et al. [95] has shown that this makes it possible to quickly prototype code for multiple platforms while achieving good performance. To achieve maximum performance, important operations can still be specialized using custom kernels that are optimized for the platform at hand and include application-specific knowledge.

5.3.2 Abstractions for kernel programming

Even though Julia supports multiple GPU platforms, it can quickly become cumbersome to write compatible kernels for each one. For example, kernels need to adhere to specific device APIs that are offered by the platform. With a variety of device backends available, a programmer's dream is to write one kernel that can be instantiated and launched for any

[1]https://github.com/numba/numba/issues/5706
[2]https://github.com/google/jax/issues/2012
[3]https://github.com/google/jax/issues/8074

device backend without modifications of the higher-level code, all without sacrificing performance. In Julia, this is possible with the KernelAbstractions.jl [103] package, which provides a macro-based dialect that hides the intricacies of vendor-specific GPU programming. Kernels can then be instantiated for different hardware accelerators, including CPUs and GPUs. For GPUs, full support is provided for NVIDIA (CUDA), AMD (ROCm), Intel (oneAPI), and Apple (Metal). For the GPU ODE solvers presented in this article, KernelAbstractions.jl is used to target these different backends, essentially from the same high-level kernel code.

The development of the GPU ecosystem was not a separate effort rather, DiffEqGPU.jl has been one of the driving projects of the Julia GPU ecosystem since the inception of the new approaches (GPUCompiler.jl and KernelAbstractions.jl) in 2019. As such, the developers of those tools are co-authors of this work, as this is the first application that displays the full vendor-agnostic feature set that the tooling aims to provide.

5.4 Massively Data-Parallel GPU Solving of Independent ODE Systems

This article considers two approaches for parallelizing ensemble problems on GPUs, both of them automatically translating and compiling the differential equation. The first approach is easily extensible, compatible with any existing solver, and relies on GPU vectorization. This approach is similar to the other high-level software we described in the introduction, and we will show that this approach is not performance optimal and has significant overheads. The second strategy reduces this overhead by generating custom GPU kernels, requiring numerical methods to be programmed within it. A brief overview of the automation is depicted in Figure 5.1. Both of the programs are composable with Julia's SciML [26] ecosystem, where users can write models compatible with standard SciML tools such as DifferentialEquations.jl, and DiffEqGPU.jl will automatically generate the functions which can be invoked from within a GPU kernel. Moreover, SciML is composed of polyglot tools allowing use of its libraries from other languages such as R, allowing even the use of our GPU-accelerated solvers from other programming languages.[4]

5.4.1 `EnsembleGPUArray`: Accelerating Ensemble ODEs with GPU Array Parallelism

Identifying parallelism and problem construction

For an ODE with n states, m parameters, and N required simulations with different parameters, there exist $n \times N$ states to keep track of. This problem can be formulated as solving one ODE

$$\frac{dU}{dt} = F(U, P, t) \tag{5.1}$$

[4]https://cran.r-project.org/web/packages/diffeqr/vignettes/gpu.html

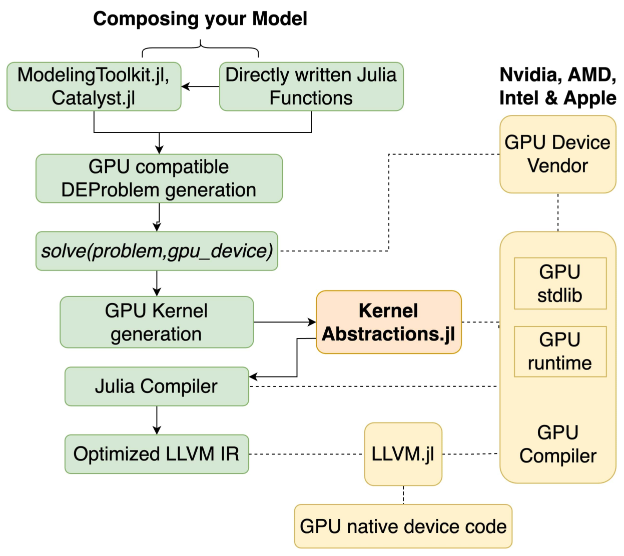

Figure 5.1: Overview of the automated translating and solving of differential equations for GPUs for massively data-parallel problems. The solid lines indicate the code flow, whereas the dashed lines indicate the extension interactions.

where

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1N} \\ u_{21} & u_{22} & \cdots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \cdots & u_{nN} \end{bmatrix}_{n\times N}, \tag{5.2}$$

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1N} \\ p_{21} & p_{22} & \cdots & p_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mN} \end{bmatrix}_{m\times N}, \tag{5.3}$$

$$F = \begin{bmatrix} f(u, p_{1:m,1}, t) & \cdots & f(u, p_{1:m,N}, t) \end{bmatrix}_{n\times N}, \tag{5.4}$$

and $p_{1:m,j}$ denotes the j^{th} column of the P matrix. In this form, we can parallelize the computation over GPU threads, where each thread only accesses and updates the column of U in parallel. This allows the computation of the quantities which depend on U to happen in parallel. When solving ODEs, these quantities are generally the right-hand side (RHS) of ODE f, the Jacobian J, and even the event handling (callbacks). We perform these array-based computations by calling the functions within custom-written GPU kernels, updating each column of the U asynchronously.

Translating ODE solves over GPU using KernelAbstractions.jl

In order to fully leverage the GPU ecosystem in Julia, several changes were made to it to allow DiffEqGPU.jl to have seamless performance and composability. Changes such as the generalization of SciML kernels from CUDA-specific to backend-agnostic, launch parameters

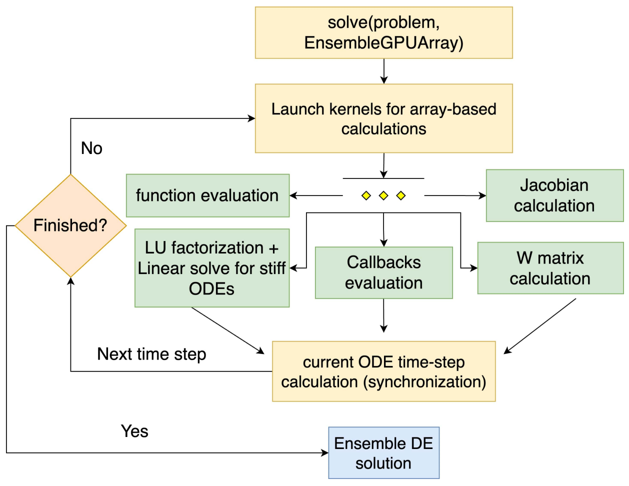
solve(problem, EnsembleGPUArray)
Launch kernels for array-based calculations
No
Finished?
function evaluation
Jacobian calculation
LU factorization + Linear solve for stiff ODEs
Callbacks evaluation
W matrix calculation
Next time step
current ODE time-step calculation (synchronization)
Yes
Ensemble DE solution

Figure 5.2: The EnsembleGPUArray flowchart.

tuning, and delivering usability of newer backends such as Apple metal are some of the initiatives to make this work a reality, which in turn is helping the broader community. Hence, our work provides a roadmap for how others can achieve true vendor agnosticism and the reality of the work necessary.

The GPU kernels are written using KernelAbstractions.jl [103], which allows for the instantiation of the GPU kernels for multiple backends. KernelAbstractions.jl performs a limited form of auto-tuning by optimizing the launch parameters for occupancy. Since these kernels have a high residency, preferring a launch across many blocks has been shown to be beneficial. We instantiate the kernels with the problem defined as normal Julia functions that the kernel is specialized upon. Using a Just-In-Time (JIT) compilation approach, we thus generate a new kernel where the solver and the problem definition are co-optimized.

After calculating the dependents on U is completed, synchronization is required to calculate the next step of the integration. `EnsembleGPUArray` essentially parallelizes the operation involving the state U within the single time step of the ODE integration. This simple approach allows composability and easy integration with the vast collection of numerical integration solvers in DifferentialEquations.jl [26]. An option to simultaneously offload a subset of the solutions to the CPUs provides additional flexibility to the user to leverage the CPU cores. Moreover, users can take advantage of the multiple GPUs over clusters to perform the simulations of the ensemble problems via this tutorial.[5] Figure 5.2 is an overview of the process.

Batched LU: Accelerating Ensemble of Stiff ODEs

Stiff ODE solvers require repeatedly solving the linear system $W^{-1}b$ where $W = -\gamma I + J$, γ is a constant real number, and J is the Jacobian matrix of the RHS of the ODE. The W

[5]https://docs.sciml.ai/DiffEqGPU/dev/tutorials/multigpu/

matrix of the batched ODE problem in Section 5.4.1 has a block diagonal structure:

$$W = \begin{bmatrix} -\gamma I + J_1 & & & \\ & -\gamma I + J_2 & & \\ & & \ddots & \\ & & & -\gamma I + J_N \end{bmatrix}, \tag{5.5}$$

where $(J_k)_{ij} = \frac{\partial f_i}{\partial u_{jk}}$. The block diagonal system can be efficiently solved by computing the LU factorization, and forward, and backward substitutions of each block of W in the GPU kernel.

Drawbacks of the Array Ensemble Approach

The main drawback of this approach is that each array operation inside of the ODE solver requires a separate GPU kernel launch. However, in explicit Runge-Kutta methods as described in Sections 2.1.1 and 2.2.3, most of the operations are linear combinations and column-wise parallel applications of the ODE model f, and are thus $\mathcal{O}(N)$ operations. Array-based GPU DSLs are typically designed to be used with $\mathcal{O}(N^3)$ operations which are common in neural network applications (such as matrix multiplication) in order to more easily saturate the kernels to overcome the overhead of kernel launch. While the ODE solvers are written in a form that automatically fuses the linear combinations to reduce the total number of kernel calls, thus reducing the overall cost [104], we will see in the later benchmarks (Section 5.5.2) that each of the array-ensemble GPU ODE solvers has a high fixed cost due to the total overhead of kernel launching.

In addition, the parallel array computations of each step of the solver method need to be completed before proceeding to the next time step of the integration. Adaptive time-stepping in ODEs allows variable time steps according to the local variation in the ODE integration, allowing optimal time-stepping. Trivially, the ODE can have different time-stepping behavior for other parameters, as they form part of the "forcing" function $f(u, p, t)$. The implicit synchronization of the parallel computations necessitates the same time-stepping for all the trajectories by virtue of solving all trajectories as a single ODE.

5.4.2 `EnsembleGPUKernel`: Accelerating Ensemble of ODEs with specialized kernel generation for entire ODE integration

The `EnsembleGPUArray` requires multiple kernel launches within a time step, which causes large overheads due to the numerous load and storage operations to global memory. In order to completely eliminate the overhead of kernel launches, a separate implementation denoted `EnsembleGPUKernel` generates a single model-specific kernel for the full ODE integration. For stiff ODEs, the Jacobian can be calculated with Automatic Differentiation (AD) invoked within the kernels. Each thread accesses the data-augmented ODE to analyze, and the solving of all the ODEs is completely asynchronous. The process is briefly outlined in Figure 5.3 and an example is given in Listing 1.

The approach described seems deceptively simple but requires clever maneuvers to successfully compile the kernel on GPU. Allocating arrays within GPU kernels is not possible,

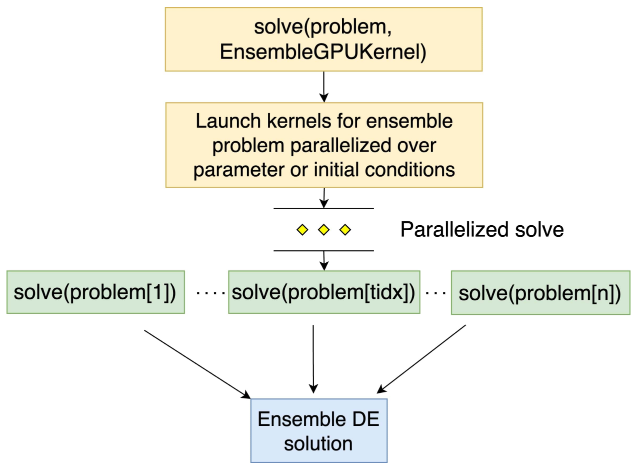
solve(problem, EnsembleGPUKernel)
Launch kernels for ensemble problem parallelized over parameter or initial conditions
Parallelized solve
solve(problem[1])
solve(problem[tidx])
solve(problem[n])
Ensemble DE solution

Figure 5.3: The EnsembleGPUKernel flowchart.

```julia
@kernel function tsit5_kernel(@Const(probs), _us, _ts, dt)
    # Get the thread index
    i = @index(Global, Linear)
    # get the problem for this thread
    prob = @inbounds probs[i]
    # get the input/output arrays for this thread
    ts = @inbounds view(_ts, :, i)
    us = @inbounds view(_us, :, i)
    # Setting up initial conditions and integrator
    integrator = init(...)
    # Perform ODE integration until completion
    while cur_time < final_time
      step!(integrator, ts, us)
      savevalues!(integrator, ts, us)
    end
    # Perform post-processing
    ...
end
```

Listing 1: Example of the kernel performing ODE integration

as Julia's CUDA.jl does not support dynamic memory allocation on the GPU. However, solving ODEs requires storing intermediate computations, normally using array allocations. The vast features of DifferentialEquations.jl rely on operations such as broadcast operations, dynamic allocations, and dynamic function invocation – most of which are GPU incompatible. The solution is to fully stack allocate all intermediate arrays and to perform the ODE integration within a custom GPU kernel implementing the numerical integration procedure. This restricts the user to the set of the already defined ODE solvers in the package and requires simple versions of the ODE solvers to be manually written as GPU kernels.

Kernel-Specialized ODE Solvers

The ODE solvers that are currently available with `EnsembleGPUKernel` are:

- `GPUTsit5`: A custom GPU-kernelized implementation of Tsitouras' 5th-order solver, a Runge-Kutta 5(4) order method [105]. Serves as a go-to choice for solving non-stiff ODEs. Performs well for medium to high tolerances. More efficient and precise than the popular Dormand-Price 5(4) [11] Runge-Kutta pair, which is a common default solver choice in packages such as MATLAB [23] and torchdiffeq [3]. Has free 4th-order interpolation support.

- `GPUVern7`: A custom GPU-kernelized implementation of Verner's 7th-order solver, a Runge-Kutta 7(6) order method [64]. Performs best at medium and low tolerances. Has a 7th-order lazy interpolation scheme.

- `GPUVern9`: A custom GPU-kernelized implementation of Verner's 9th-order solver, a Runge-Kutta 9(8) order pair [64]. Performs best at extremely low tolerances. Has a 9th-order lazy interpolation scheme.

- `GPURosenbrock23`: A custom GPU-kernelized implementation of an order 2/3 L-Stable Rosenbrock-W method [17], [23], [25]. Is good for very stiff equations with oscillations at high tolerances. Employs a second-order stiff-aware interpolation. Also supports Differential Algebraic Equations (DAEs) in mass-matrix form, i.e., $M\frac{du}{dt} = f(u, p, t)$, where M is the mass matrix.

- `GPURodas4`: A custom GPU-kernelized implementation of a fourth-order A-stable stiffly stable Rosenbrock method [17], [25]. Is suitable for problems at medium tolerances. Employs a third-order stiff-aware interpolation.

- `GPURodas5P`: A custom GPU-kernelized implementation of a fifth-order A-stable stiffly stable Rosenbrock method [25], [106]. Is suitable for problems at medium tolerances. Employs a fourth-order stiff-aware interpolation which has better stability in the adaptive time-stepping embedding.

These choices are based on the SciMLBenchmarks, which has extensive comparisons between ODE solvers.[6] For stiff ODEs, the Rosenbrock-type methods are ideally suited for GPU compilation because they are devoid of the typical Newton's method performed per step

[6] https://docs.sciml.ai/SciMLBenchmarksOutput/stable/

in stiff ODE integrators [17]. On CPU, implicit ODE solvers achieve top performance by using the property that the inverted Jacobian of Newton's method does not need to be exact, and thus efficient integrators such as CVODE [54] and those in DifferentialEquations.jl adaptively reuse the same Jacobian factorization from multiple time steps. For sufficiently large equations, this trades the $\mathcal{O}(N^3)$ factorization operation for more iterations of Newton's method with the same inverse factor and thus $\mathcal{O}(N^2)$ linear solves. However, these properties do not transfer well to the GPU setting since (a) branching within a GPU warp is resolved such that every concurrent solve requires the same number of iterations, thus leading to more linear solves than necessary for most of the systems at each step and (b) the systems are sufficiently small so that the factorization is not the dominating factor in the compute cost.

Rosenbrock methods only require one Jacobian evaluation and have a constant number of linear solves per step, where the matrix factorization can be cached to a single factorization per time step. This more static integration procedure is also compensated by higher stage order (i.e., convergence rate on highly stiff ODEs and DAEs) and smaller leading truncation error coefficients, effectively leading to less steps being required to reach the same error. For this reason, Rosenbrock methods are the most efficient solver on CPU for sufficiently small ODEs (approximately less than 20), competitive until the LU-factorization cost becomes dominant (around 100 ODEs). Some prior research has shown results with ODE extrapolation methods where the LU factorization does not have the dominating cost and hence no suitable gains from its parallelization other than parallelizing multiple computations of LU [7].

Integration of automatic differentiation: Automatic differentiation is a way to compute exact derivatives of mathematical computer programs [107]. With the change to high-order Rosenbrock methods, having accurate Jacobians (non-finite difference) is a necessary requirement for Rosenbrock methods to achieve high-order convergence [17]. Thus, while previous LSODA approaches could get away with finite difference approximations through the relaxation in nonlinear solver steps, our approach, which would be more GPU performant due to the greatly reduced branching, requires mixing this automatic differentiation to achieve full accuracy and performance. Integration of solvers is done with forward-mode automatic differentiation within a GPU kernel. In particular, we extended the LU factorization in the nonlinear solve step to statically compile seamlessly for arbitrary size of the ODE.

Given the extra advantages of the GPU and the fact that the embarrassingly parallel context is limited to this size of systems, we will see that Rosenbrock methods are outperformed by the Newton-based stiff ODE solvers quite handily.

Kernel-Specialized SDE solvers

Currently, DiffEqGPU.jl only supports fixed time-stepping in SDEs with `EnsembleGPUKernel`.

- `GPUEM`: A custom GPU-kernelized fixed time-step implementation of Euler-Maruyama method [46]. Supports diagonal and non-diagonal noise.
- `GPUSIEA`: A custom GPU-kernelized fixed time-step implementation of weak order 2.0 [108], stochastic generalization of midpoint method. Supports only diagonal noise.

5.5 Benchmarks and Case studies

5.5.1 Setup

To compare different available open-source programs with GPU-accelerated ODE solvers, we benchmark them with several NVIDIA GPUs: one being a typical compute node GPU, Tesla V100, and the other being a high-end desktop GPU, Quadro RTX 5000. Performance comparison of the kernel-based ODE solvers with different GPU vendors is also carried out. Except for Apple having an integrated GPU, we use dedicated desktop GPUs. For NVIDIA, we benchmark on Quadro RTX 5000 (11.15 TFLOPS), Vega 64 for AMD (10.54 TFLOPS), A770 (19.66 TFLOPS) for Intel, and M1 Max (10.4 TFLOPS) for Apple. The DE problems involve single precision (Float32) on GPUs. The CPU benchmarks are executed using double precision (Float64), and are timed on an Intel Xeon Gold 6248 CPU @ 2.50GHz with 16 enabled threads. Using double precision on CPUs is faster than the single precision for our use-case and processor.

To facilitate the seamless transition from CPU to GPU without requiring any modifications to the original code, `EnsembleGPUKernel` was developed to mimic the SciML ensemble interface. Nevertheless, this GPU-aspect-hiding approach always passes the problem to the GPU and returns the result to the CPU, reducing overall performance. To avoid conversion overheads and for a fair comparison among other software, we developed a lower-level API[7] that closely resembles other APIs. The timings for each software only report time spent solving the ensemble ODE. The benchmarking literature discusses the topic of reporting times for benchmarking and uses the minimum as an approximation of the ideal value, the least noisy measurement. Hence, the timings for the programs written in Julia are measured using BenchmarkTools.jl, taking the best timing. The Julia-based benchmarks were tested on DiffEqGPU.jl 1.26, CUDA.jl 4.0, oneAPI.jl 1.0, and Metal.jl 0.2.0, all using Julia 1.8. The test with AMD GPUs was done with AMDGPU.jl 0.4.8, using Julia 1.9-beta3.

The timings script for MPGOS has been borrowed from their available open-source codes.[8] They have been tested with CUDA toolkit 11.6 and C++ 11. The programs are run at least ten times for JAX and PyTorch. The JAX-based programs are run on 0.4.1 with Diffrax 0.2.2. Programs based on PyTorch are tested with PyTorch nightly 2.0.0.dev20230202, and a custom installation of torchdiffeq to extend support to `vmap` is used.[9] Both programs are tested on Python 3.9. The complete benchmark suite can be found at https://github.com/utkarsh530/GPUODEBenchmarks.

5.5.2 Establishing efficiency of solving ODE ensembles with GPU over CPU

Prior researches [76], [89] demonstrate that solving low-dimensional ODEs over parameters and initial conditions (massively parallel) exposes superior parallelism in GPUs over CPUs. The benchmarking is performed on GPUs with single precision and CPU multithreading with

[7]https://docs.sciml.ai/DiffEqGPU/dev/tutorials/lower_level_api/
[8]https://github.com/nnagyd/ode_solver_tests
[9]https://github.com/utkarsh530/torchdiffeq/tree/u/vmap

Table 5.1: Summary of mean slowdowns of ODE integrators, benchmarking on stiff problems with different hardware *(lower the better)*

Time-stepping	GPU (Kernel)	GPU (Array)	CPU
Adaptive (Nonstiff)	1.0×	48.2×	22.2×
Fixed (Nonstiff)	1.0×	377.6×	110.3×
Adaptive (Stiff)	1.0×	180.0×	132.3×

double precision to take advantage of respective optimized floating point math. Indeed, the `EnsembleGPUKernel` supports our claim of lower overhead compared to `EnsembleGPUArray` and being up to 100× faster. For the benchmark problem GPU parallelism becomes superior to CPU parallelism at approximately 100–1000 trajectories (Figure 5.4). The solver used to benchmark the Lorenz Attractor [109] is Tsitouras' 5(4) Runge-Kutta method [105], both with adaptive and fixed time-stepping. Similarly, we also perform the benchmarking of stiff ODE integrators with the Robertson equation[65], using the Rosenbrock23 methods [25]. Figure 5.5 showcases the performance of the solvers. Table 5.1 lists the relative slowdowns of both non-stiff and stiff ODE integrators obtained using `EnsembleGPUKernel`.

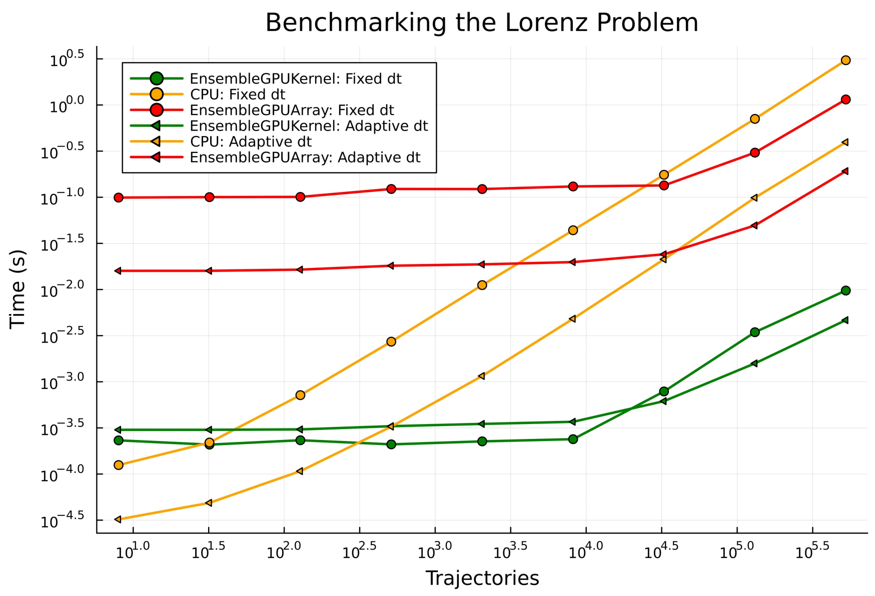

Figure 5.4: A comparison of time for an ODE solve for CPU vs. GPU. The EnsembleGPUKernel performs the best with up to 100× acceleration and a lower cutoff to take advantage of parallelism.

GPU performance at < 1000 trajectories: Massive parallel simulations are needed to utilize GPU cores and amortize overhead costs fully. Figure 5.4 shows that overheads are essentially constant regardless of thread count. In this range, the computations are not latency hiding; one essentially measures the kernel launch time. Hence, one would not expect superior GPU parallelism in comparison to CPU at lower trajectories because CPUs are usually free of these overheads.

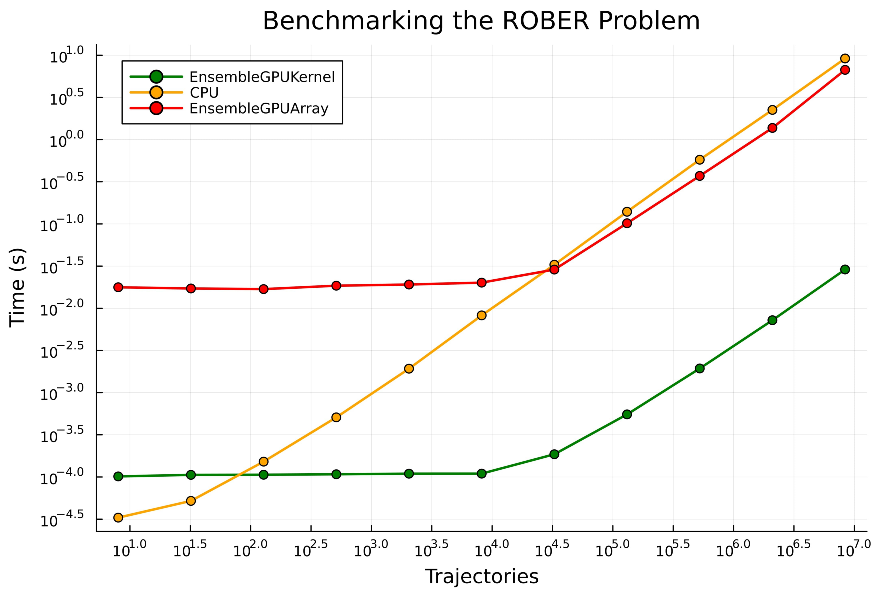

Figure 5.5: A comparison of the time of an ODE solve for CPU vs. GPU. The EnsembleGPUKernel performs the best with up to 100× acceleration and a lower cutoff to take advantage of parallelism.

5.5.3 Solving billions of ODEs together: Scaling `EnsembleGPUKernel` with MPI

The Message Passing Interface (MPI) [110], [111] can be directly used with Julia using MPI.jl [112]. Additionally, there exists support for CUDA by using a CUDA-aware MPI backend. As a testimonial of scalability, 2,147,483,648 (∼2 billion) ODE solutions on the Lorenz problem were calculated using eight V100 GPU cluster nodes, in which seven GPUs amounting to 35,840 CUDA cores explicitly performed the computation of approximately 306 million ODEs. The wall-clock time of this simulation was approximately 50 seconds, which includes other latencies such as package loading, compilation, and GC times. The runtime of the MPI call (creating buffers and transferring arrays to GPUs) and solving ODEs was around 13 seconds. The runtime of only the ODE solve of 306 million trajectories was around 1.6 seconds.[10] We also note in our observations that the methods are scalable, and are limited by the global memory of the GPU required for storing the solutions, which can be roughly calculated by calculating the memory requirements of the number of time-points required multiplied by the different number of parameters in the simulation. Subsequently, the methods are prophesied to scale well with multiple nodes in a cluster.

5.5.4 Comparison with other GPU-accelerated ODE programs

Selecting the problem for fair benchmarking on different implementations of GPU-based solvers is a task in itself, owing to the different use cases, motivations, and optimizations of

[10]These latencies could be reduced by using a GPU with larger memory or a multi-GPU per node, which was not done in our demonstration due to hardware availability.

these libraries. Choosing systems with low-dimensional ODEs, such as the Lorenz equation, is a suitable candidate owing to the simplicity $f(u, p, t)$ of definition, which alludes to any optimizations in calculating $f(u, p, t)$. The right-hand side solely involves additions/subtractions and multiplications, in which each floating-point operation will be interpreted as a complete FMA (Fused Multiply Accumulate) instruction. For our benchmarking purposes, $\sigma = 10.0$, $\gamma = \frac{8}{3}$, and ρ in the Lorenz equation [109] is uniformly varied from $(0.0, 21.0)$ generating N instances of independent, parallel ODE solves, which is also the number of trajectories in DiffEqGPU.jl API.

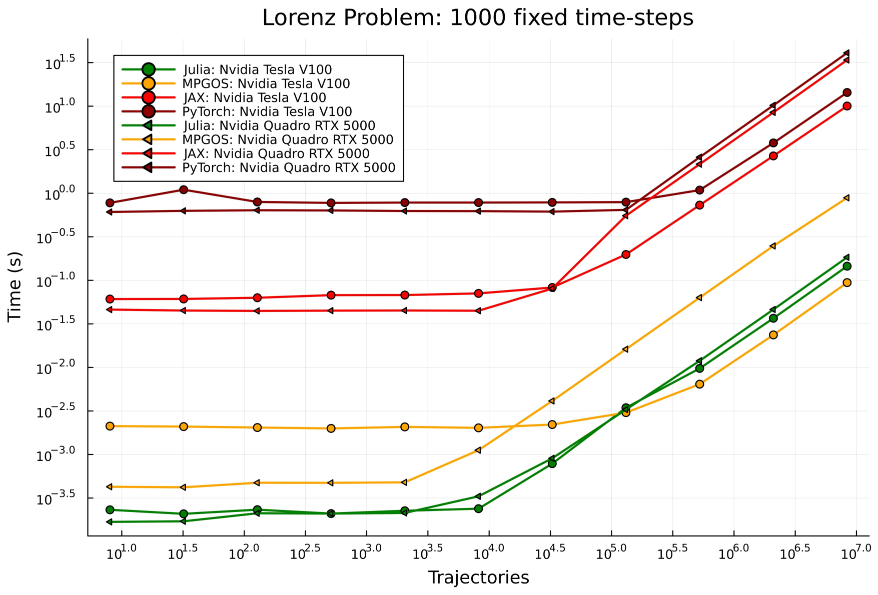

Figure 5.6: A comparison of the time for an ODE solve with other programs with fixed time-stepping. EnsembleGPUKernel is able to reach and sometimes outperform speed of light measure (MPGOS) and is approximately faster by 20–100× in comparison to JAX and 100–200× for PyTorch.

The results for fixed and adaptive time-stepping are shown in separate Figures 5.6 and 5.7 for equitable comparison. There is not any common ODE solver between these packages, so we use methods belonging to the class of 4$^{\text{th}}$, 5$^{\text{th}}$ order Runge-Kutta methods, which perform similarly in the benchmarks [26]. "Tsit5" was used in both Julia and JAX, "Cash-Karp" for MPGOS, and "Dopri5" for PyTorch. Fixed time-stepping implicitly assures a constant work/thread, which is ideal for GPUs. However, adaptive time-stepping within ODE integrators adjusts time steps to ensure stability and error control. This generally results in faster ODE integration times than fixed time-stepping, but may cause thread divergence as effectively different time-stepping across threads, eventually amounting to contrasting work per thread. The slowdown of different packages with respect to our implementation with different NVIDIA GPUs is listed in Tables 5.2 and 5.3. While our solvers are able to reach and sometimes outperform speed of light measure (C++, MPGOS), we observe a greater acceleration of approximately 20–100× in comparison to JAX and 100–200× for PyTorch, both of which rely on vectorized map style of parallelism. The authors were not able to compile adaptive time-stepping results for PyTorch, as `vmap` currently does not support all the internal operations required by the PyTorch code base. Notably, the array abstraction parallelism approach of PyTorch and JAX performs similarly to the demonstrated efficiency of `EnsembleGPUArray`, providing clear evidence that the performance difference is due to the fundamental approach itself and not due to efficiencies or inefficiencies in the implementation of the approach.

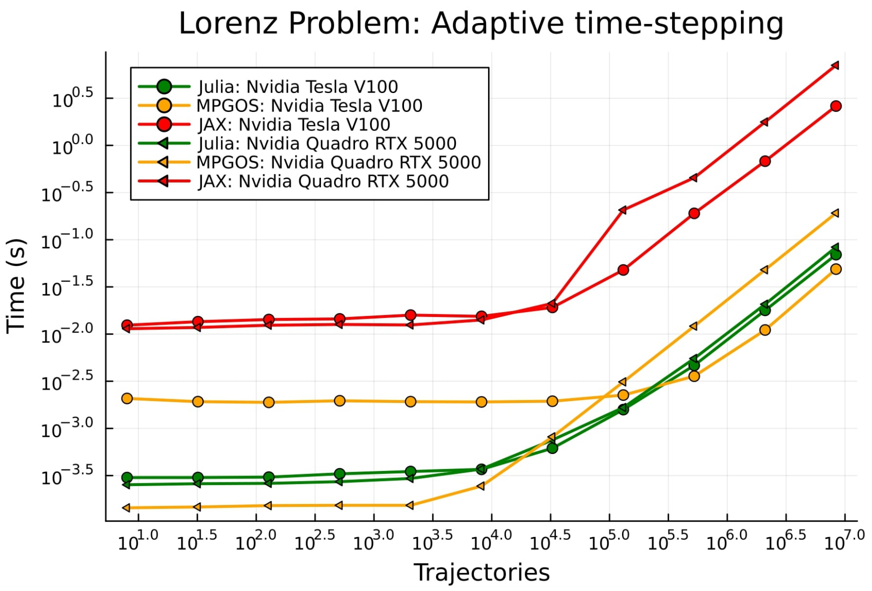

Figure 5.7: A comparison of the time for an ODE solve for with other programs with adaptive time-stepping. EnsembleGPUKernel is able to reach and sometimes outperform speed of light measure (MPGOS) and approximately faster by 20–100× in comparison to JAX.

Similarly, we benchmark the solvers with stiff ODEs. A usual test case for stiff ODEs is the Robertson equation [65], which models a chemical reaction between three species. The rate of the reactions varies at drastically different time scales, which subsequently gives rise to the stiffness characteristic of the problem. We also additionally benchmark with an other massively parallel stiff ODE solver, ginSODA [113], which uses Python as a front-end for their CUDA-C++ kernels. Figure 5.8 summarizes the benchmarks when compared with JAX, Julia's EnsembleGPUArray, and ginSODA. The performance trend remains congruent with non-stiff ODE problems, indicating an average speed-up of 76–130× for JAX, 129–280× for Julia's EnsembleGPUArray, and 170–409× for ginSODA. According to the author's knowledge, the stiff ODE solvers are the first most extensive and the state-of-the-art massively parallel kernel-based solvers available for GPU acceleration.

5.5.5 Vendor agnosticism with performance: Comparison with several GPU platforms

With vendor-agnostic GPU kernel generation, researchers can choose major GPU backends with ease. Our benchmarks in Figure 5.9 demonstrate that overhead is minimal in our ODE solvers and users can expect performance one-to-one with mentioned Floating Point

Table 5.2: A summary of the range of slowdowns of the benchmarks in Figures 5.6 and 5.7 *(lower is better)*, with results compiled on a desktop GPU. The slowdowns are computed by varying the number of trajectories. The Julia-based solvers achieve the best acceleration on average.

Software \ Time-stepping	Fixed	Adaptive
DiffEqGPU.jl (Julia)	1.0×	1.0×
MPGOS (C++)	2.2–5.3×	0.5–2.3×
Diffrax (JAX)	88.9–274.0×	28.0–124.0×
torchdiffeq (PyTorch)	196.4–3617.7×	—

Table 5.3: A summary of the range of slowdowns of the benchmarks in Figures 5.6 and 5.7 *(lower the better)*, with results compiled on a server GPU. The slowdowns are computed by varying the number of trajectories. The Julia-based solvers achieve the best acceleration on average.

Software \ Time-stepping	Fixed	Adaptive
DiffEqGPU.jl (Julia)	1.0×	1.0×
MPGOS (C++)	0.6–10.0×	0.6–6.9×
Diffrax (JAX)	57.4–322.3×	30.2–46.8×
torchdiffeq (PyTorch)	98.8–3693.9×	—

Table 5.4: A summary of the range of slowdowns of the benchmarks in Figure 5.8 *(lower the better)*, with results compiled on a server GPU for stiff ODEs. The slowdowns are computed by varying the number of trajectories. The Julia-based solvers achieve the best acceleration on average.

Software \ Time-stepping	Adaptive
EnsembleGPUKernel (Rodas5P)	1.0×
EnsembleGPUKernel (Kvaerno5)	0.85–1.0×
EnsembleGPUKernel (Rosenbrock23)	1.2–1.6×
EnsembleGPUArray (Kvaerno5)	129.1–279.9×
Diffrax (Kvaerno5)	76.6–130.0×
ginSODA (LSODA)	170.4–409.4×

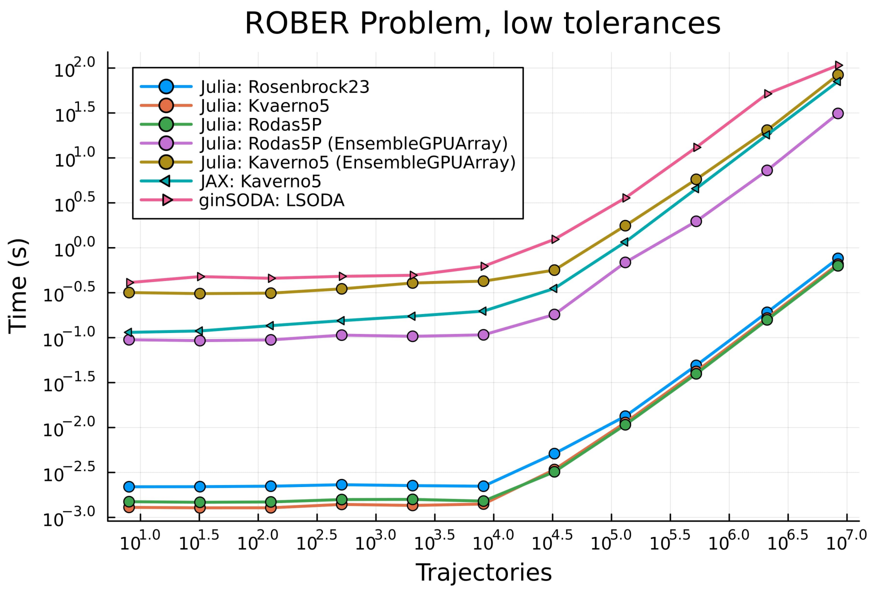

Figure 5.8: A comparison of the time for a stiff ODE solve with other programs with adaptive time-stepping. EnsembleGPUKernel is faster than JAX and EnsembleGPUArray by approximately 76–130×.

Operations per Second (FLOPS) in GPUs mentioned in the Section 5.5.1. To elude other performance impacts such as thread divergence and equitable selection of the dimension of the ODE, we simulate the Lorenz problem with fixed time-stepping. We run the benchmarks on major vendors: NVIDIA, AMD, Intel, and Apple. The peak flops are listed in Section 5.5.1. To our knowledge, this is the first showcase of GPU-parallel software for solving DEs that supports more than NVIDIA GPUs.

5.5.6 Event handling and automatic differentiation

Software for simulating dynamical systems allows for injecting discontinuous events and termination of integration with a specified criterion causing discontinuities within the integration. Event handling in differential equations is used to produce non-differentiable points in continuous dynamical systems. Mathematically, an event within an ODE can be specified as a tuple of functions, g, h ("condition" and "affect"), where satisfying the "condition" $g(u, p, t) = 0$ triggers the "affect" $h(u, p, t)$, changing u, t or terminating the integration. As a demonstration of event capabilities with `EnsembleGPUArray` and `EnsembleGPUKernel`, we demonstrate the famous surface-ball collision (bouncing ball) dynamics simulated on GPUs in Figure 5.11.

Additionally, the GPU kernels are automatic differentiation (AD) compatible, both with forward with reverse mode, allowing for GPU-parallel forward and adjoint sensitivity analysis. Tutorials in the library demonstrate the usage of AD for parameter estimation with minibatching.[11]

[11] https://docs.sciml.ai/SciMLSensitivity/stable/tutorials/data_parallel/#Minibatching-Across-GPUs-with-DiffEqGPU

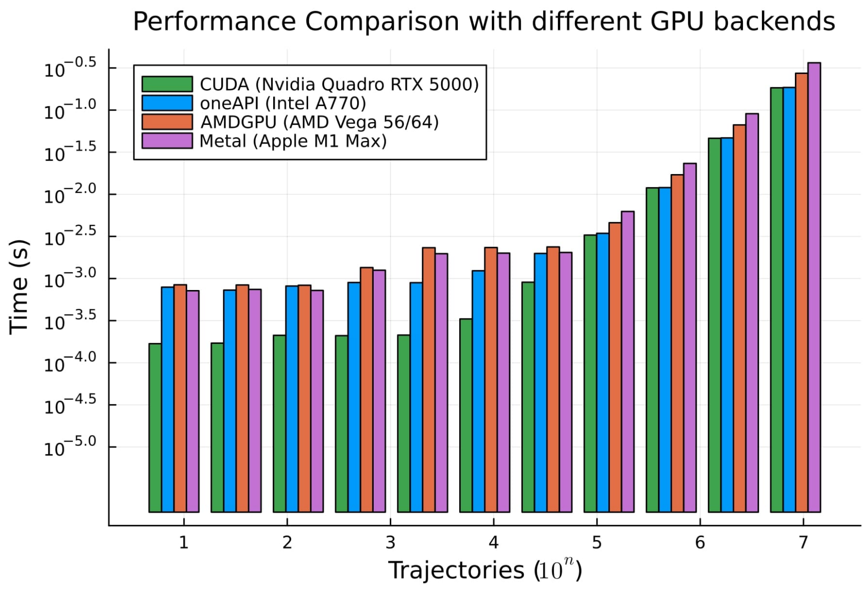

Figure 5.9: A comparison of the time for an ODE solve for fixed time-stepping, measured on different GPU platforms. The non-stiff ODE solver `GPUTsit5` is used here. We measure the time *(lower the better)* versus the number of parallel solves. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

```
// Building textured memory of the dataset
texture = CuTexture(CuTextureArray(dataset))
// Passing the textured memory as parameter of the problem
prob = ODEProblem(f_rhs, u0, tspan, (p, texture))
// Performing the solve, with 0 CPU offloading.
sol = solve(prob, alg, EnsembleGPUKernel(0), ...)
```

Listing 2: An outline of using texture memory with DifferentialEquations.jl

5.5.7 Texture memory interpolation

As the mathematical model used to simulate a dynamical system is oftentimes a low-fidelity approximation of the true physical phenomena, practitioners often account for non-modeled behavior via lookup tables and interpolation of datasets. These datasets may be derived from real-world experimentation and/or higher-fidelity simulation. For cases where the interpolant is a function of the system state, interpolation is required at each time step for each system in the ensemble. A simple example of this is the above bouncing ball problem, extended to include drag forces imparted on the ball via the interpolation of a spatially varying wind field. Furthermore, a user may need to interpolate digital terrain elevation data for ground collision event handling, in which the textured memory defined in Listing 2 can be used.

In addition to exploiting parallelism for time-stepping as described above, GPU texture memory can be leveraged when interpolation is required with multiple benefits. In particular, for NVIDIA GPUs, texture memory provides interpolation, nearest-neighbor search, and automatic boundary handling for the cost of a single memory read. Benchmarking with

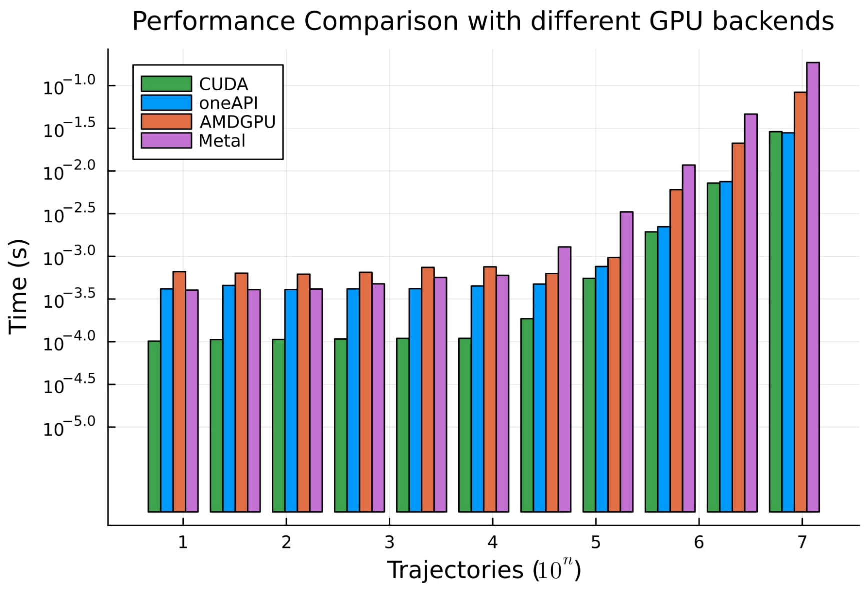

Figure 5.10: A comparison of of the time for an ODE solve for adaptive time-stepping, measured on different GPU platforms. The stiff ODE solver `GPURosenbrock23` is used here. We measure the time *(lower the better)* versus the number of parallel solves. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

texture memory results in 2× faster simulation, when we replace a compute-bound operation such as interpolation on GPU with texture-memory.

Furthermore, texture memory is advantageous for situations where the memory access pattern is not known *a priori*, which is often the case for state-dependent interpolation. Texture memory, however, requires uniformly spaced data.

5.5.8 Accelerating stochastic processes with GPUs

The expectation of SDE solutions is a key metric in many model analyses due to the randomness of the simulation process. Such a calculation is generally done through Monte Carlo estimation via generating many trajectories of SDE solution, typically requiring tens of thousands of trajectories for suitable convergence due to the $\mathcal{O}(\sqrt{N})$ convergence rate to the expectation. The generation of these trajectories is independent of each other, thus fitting the form of ensemble parallelism with the same initial condition and parameter values, with the only difference being the seed supplied to the Pseudorandom Number Generator (PRNG).

Asset Price Model in Quantitative Finance

As a rudimentary example, we examine the ensemble simulation of a linear SDE, popularly known as Geometric Brownian Motion (GBM), which is the common Black-Scholes model used in asset pricing in quantitative finance [42], [43]. The benchmarks in Figure 5.12 demonstrate that simulations on GPU are faster than CPU on average by 8×. GPU

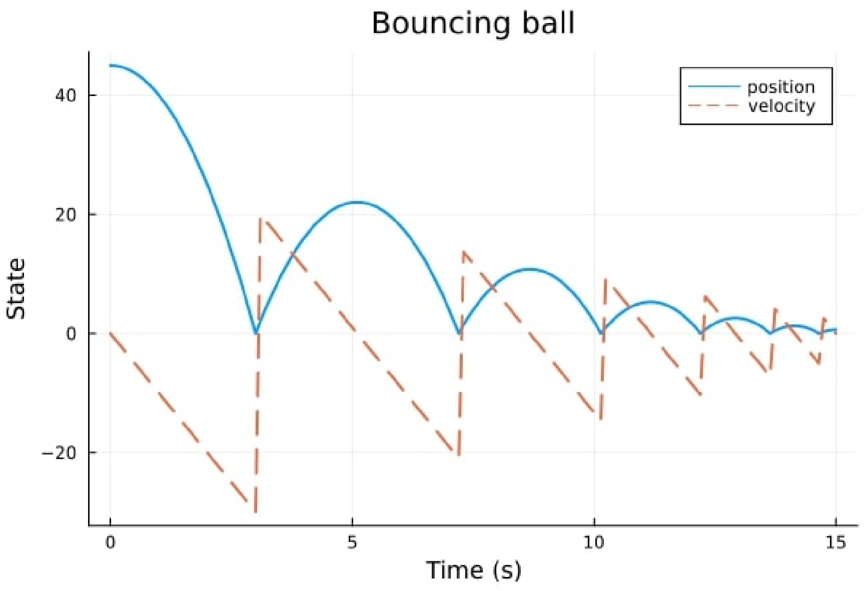

Figure 5.11: A simulation of bouncing ball problem on GPU. The blue trajectory is the displacement and the red trajectory is the velocity, across time. This demonstrates the ability to inject code within ODEs via callbacks.

parallelism dominates CPU parallelism over approximately 1000 trajectories. We note that the decreased performance difference from the ODE case is likely due to the less optimized implementation of the kernel PRNG implementations.

Stochastic Chemical Reaction Networks

Additionally, we benchmark the SDE solvers over a real case study of Chemical Reaction Networks (CRN) generated when microorganisms such as bacteria respond to stimuli, which causes a change in its gene expression through sigma factors [114]. These biological processes are inherently noisy in nature, and it is simulated by transforming the CRN to an SDE via the Chemical Langevin Equation (CLE) [45]. The non-dimensionalized model has notably 4 states, 8 Wiener noise variables, and 6 parameters, making it suitable for our case study as our GPU DE solvers are suited for problems with low-dimensional states. The simulation of the process on the GPU in shown in Figure 5.13. The benchmark investigates the generation of trajectories of solutions for different parameters akin to parameter sweeps, which are widely used in parameter estimation and uncertainty quantification. Each of the parameters is uniformly sampled and the set of the Cartesian products of parameters is simulated, generating approximately >1,000,000 unique trajectories. The benchmarks shown in Figure 5.14 quantitatively shows that our GPU implementation for SDEs is 4.5× faster than multithreading over a CPU, averaged over different trajectories.

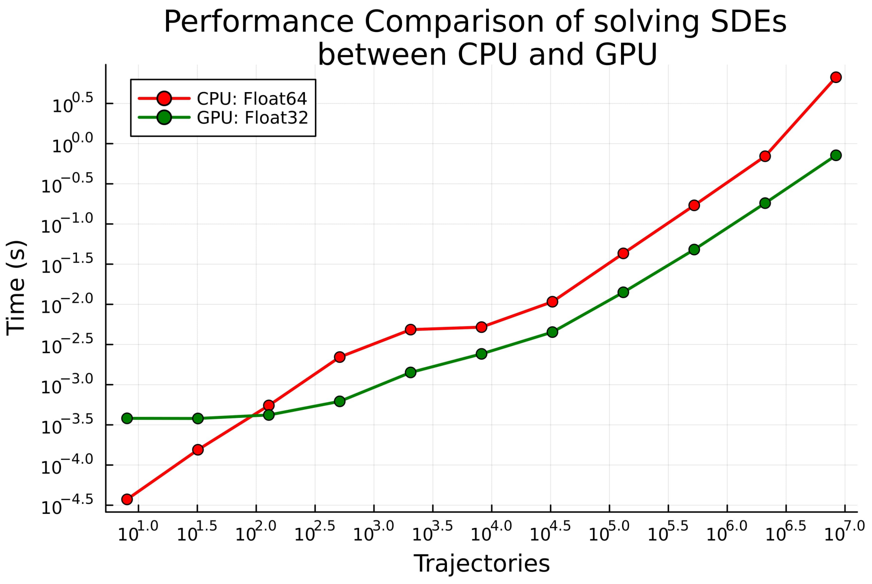

Figure 5.12: The parameter parallel simulation time for the linear SDE defined in Section 5.5.8 *(lower the better)*. The GPU parallelism supercedes CPU parallelism at about 1000 trajectories.

5.6 Discussion

We have demonstrated that many programs written for standard CPU usage of DifferentialEquations.jl can be retargeted to GPUs via DiffEqGPU.jl and achieve state-of-the-art (SOA) performance without requiring changes to user code. This solution democratizes the SOA by not requiring scientists and engineers to learn CUDA C++ in order to achieve top performance. One key result of the paper is that we demonstrate that all approaches which used array abstraction GPU parallelism, PyTorch, JAX, and `EnsembleGPUArray`, achieve similar performance and are orders of magnitude less efficient than the kernel generation approach of `EnsembleGPUKernel` and MPGOS. This suggests that the performance difference is due to a limitation of the array abstraction parallelization formulation, and demonstrates a concrete application where kernel generation is required for achieving SOA. It has previously been noted that "machine learning systems are stuck in a rut" where many deep learning architectures are designed to only use the kernels included by current machine learning libraries (PyTorch and JAX) [115]. Our results further this thesis by demonstrating that orders of magnitude performance improvements can only be achieved by leaving the constrained array-based DSL of pre-defined kernels imposed by such deep learning frameworks.

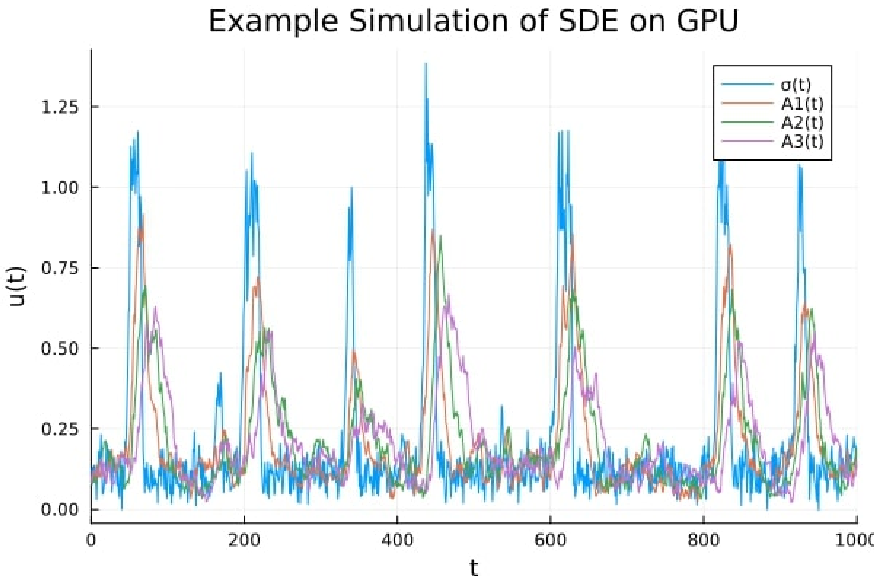

Figure 5.13: An example simulation plot of the system vs. time. The model is written with Catalyst.jl and automatically works with GPU solvers, showcasing the ability to simulate complex models seamlessly on GPU.

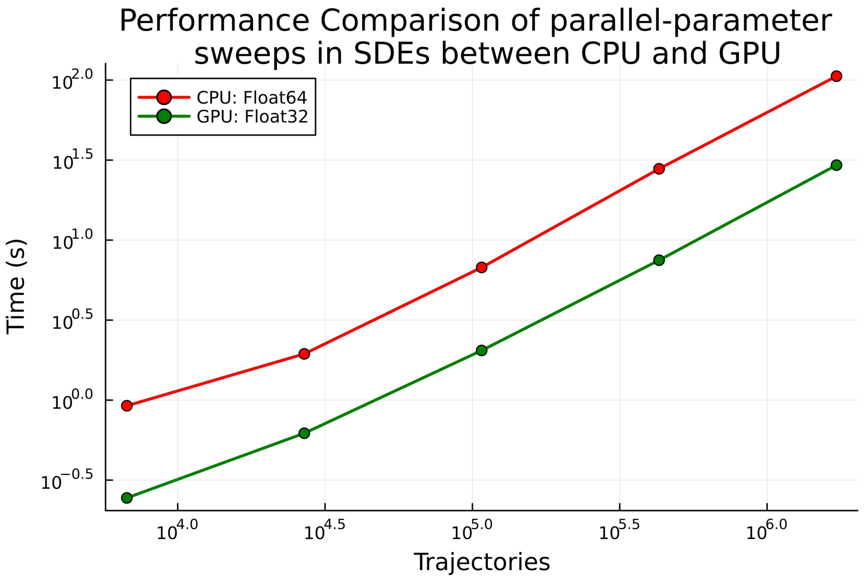

Figure 5.14: The parameter parallel simulation time *(lower the better)* of the SDE simulation of Figure 5.13. Overall, the comparison showcases the scalability of speedups of using GPUs instead of CPUs, having suitable gains for trajectories as small as 1000.

Chapter 6

DiffEqGPU.jl: GPU-acceleration routines for DifferentialEquations.jl and broader SciML ecosystem

DiffEqGPU.jl is an open-source Julia package that provides GPU acceleration for solving differential equations. The process for GPU acceleration is completely automated, where the user provides a high-level function for the differential equation, and the package automatically generates suitable kernels that are amenable to GPU compilation. Moreover, the same high-level code can target different hardware accelerators, such as CPUs and GPUs, and even to different GPU backends. Currently, the platform supports a variety of GPU backends listed in Table 6.1. Our work thus provides a roadmap for how others can achieve true vendor agnosticism and the reality of the work necessary. This work presents Julia's first portable GPU application that supports all four relevant GPU vendors. This was achieved by doing the groundwork to make it possible, i.e., improving the JuliaGPU stack (adding backends, fixing bugs, etc.).

GPU Manufacturer	GPU Kernel Language	Julia Support Language 3	Backend
NVIDIA	CUDA	CUDA.jl	`CUDABackend()`
AMD	ROCm	AMDGPU.jl	`ROCBackend()`
Intel	oneAPI	oneAPI.jl	`oneAPIBackend()`
Apple M-Series	Metal	Metal.jl	`MetalBackend()`

Table 6.1: Backend choices available in DiffEqGPU.

6.1 Setting up DiffEqGPU.jl

Installing backends

The user must install the GPU backend library to test the DiffEqGPU.jl-related code.

```
julia> using Pkg
julia> #Run either of them
julia> Pkg.add("CUDA") # NVIDIA GPUs
julia> Pkg.add("AMDGPU") #AMD GPUs
julia> Pkg.add("oneAPI") #Intel GPUs
julia> Pkg.add("Metal") #Apple M series GPUs
```

Testing DiffEqGPU.jl

DiffEqGPU.jl is a test suite that regularly checks functionality by testing features such as multiple backend support, event handling, and automatic differentiation. To test the functionality, one can follow the below instructions. The user needs to specify the "backend," for example, "CUDA" for NVIDIA, "AMDGPU" for AMD, "oneAPI" for Intel, and "Metal" for Apple GPUs. The estimated time of completion is 20 minutes.

```
$ julia --project=.
julia> using Pkg
julia> Pkg.instantiate()
julia> Pkg.precompile()
```

Finally, test the package with this command

```
$ backend="CUDA"
$ julia --project=. test_DiffEqGPU.jl $backend
```

6.2 API

The API of DiffEqGPU.jl is based on Julia's SciML ecosystem. It conveniently allows the user to work with SciML API and automatically leverage GPU acceleration. We demonstrate an example in showcasing the GPU acceleration of an ODE with parameter-parallel simulations in Listing 3.

6.3 Composability

DiffEqGPU.jl enables composability with the rest of the SciML ecosystem. This means most of the features work on the box: Automatic Differentiation, High-level modeling via ModelingToolkit.jl [116], and differential equation solvers from `OrdinaryDiffEq.jl`. We showcase an example in Listing 4 that allows composability from ModelingToolkit.jl to allow GPU acceleration of parameter-parallel sweeps of a Chemical Reaction Network with Catalyst.jl [117].

```
using Catalyst, Plots, StochasticDiffEq, StaticArrays, CUDA

@show ARGS
#settings

```

```julia
using DiffEqGPU, OrdinaryDiffEq, StaticArrays, CUDA

function lorenz(u, p, t)
    du1 = p[1] * (u[2] - u[1])
    du2 = u[1] * (p[2] - u[3]) - u[2]
    du3 = u[1] * u[2] -p[3]  * u[3]
    return SVector{3}(du1, du2, du3)
end

u0 = @SVector [1.0f0; 0.0f0; 0.0f0]
tspan = (0.0f0, 10.0f0)
p = @SVector [10.0f0, 28.0f0, 8 / 3.0f0]
prob = ODEProblem{false}(lorenz, u0, tspan, p)
prob_func = (prob, i, repeat) -> remake(prob, p = (@SVector rand(Float32, 3)) .* p)
monteprob = EnsembleProblem(prob, prob_func = prob_func, safetycopy = false)

sol = solve(monteprob, GPUTsit5(), EnsembleGPUKernel(CUDA.CUDABackend()),
    trajectories = 10_000)
```

Listing 3: Code example for accelerating ensemble simulation using DiffEqGPU.jl.

```julia
N = 10

### Example 4: Ensemble SDE simulations (varioous parameter values) at steady state
↪   behaviours of 4 variable CRN (Generalised bacterial stress response model).
↪   ###

## Credits: Torkel Loman

# Declare the model (using Catalyst).
σGen_system = @reaction_network begin
    (v0 + (S * σ)^n / ((S * σ)^n + (D * A3)^n + 1), 1.0), ∅ ↔ σ
    (σ / τ, 1 / τ), ∅ ↔ A1
    (A1 / τ, 1 / τ), ∅ ↔ A2
    (A2 / τ, 1 / τ), ∅ ↔ A3
end S D τ v0 n η

# Declares the parameter values.
σGen_parameters = [:S => 2.3, :D => 5.0, :τ => 10.0, :v0 => 0.1, :n => 3, :η =>
↪   0.1]

# Set ensemble parameter values.
S_grid = Float32.(10 .^ (range(-1.0, stop = 2, length = N)))
D_grid = Float32.(10 .^ (range(-1, stop = 2, length = N)))
τ_grid = Float32[0.1, 0.15, 0.20, 0.30, 0.50, 0.75, 1.0, 1.5, 2.0, 3.0, 5.0, 7.50,
↪   10.0,
                  15.0, 20.0, 30.0, 50.0, 75.0, 100.0][1:2:19]
v0_grid = Float32[0.01, 0.02, 0.03, 0.05, 0.075, 0.1, 0.15, 0.20]
n_grid = Float32[2.0, 3.0, 4.0]
η_grid = Float32[0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1]

parameters = collect(Iterators.product(S_grid, D_grid, τ_grid, v0_grid, n_grid,
 ↪  η_grid));

numberOfParameters = length(parameters)

@show numberOfParameters

function σGen_p_func(prob, i, repeat)
    for parameter in parameters
        remake(prob; p = SVector{6, T1}(parameter))
    end
end

# Declare initial condition.
σGen_u0 = [:σ => 0.1, :A1 => 0.1, :A2 => 0.1, :A3 => 0.1] # (for some S values, the
 ↪  system will start far away from the steady state).

# Create EnsembleProblem.
σGen_sprob = SDEProblem(σGen_system, σGen_u0, (0.0, 1000.0), σGen_parameters,
                        noise_scaling = (@parameters η)[1])

### Experimentation
sys = modelingtoolkitize(σGen_sprob)
T1 = Float32
prob = SDEProblem{false}(sys, SVector{length(σGen_sprob.u0), T1}(σGen_sprob.u0),
                         Float32.(σGen_sprob.tspan),
                         SVector{length(σGen_sprob.p), T1}(σGen_sprob.p),
                         noise_rate_prototype = SMatrix{

                                                                  ↪  size(σGen_sprob.noise_rate_prototype

                                                                  ↪  T1}(σGen_sprob.noise_rate_prototype)

using DiffEqGPU

# parameter as cartesian product of the ranges, initial condition as [v0,v0,v0,v0]
function prob_func(prob, i, repeat)
    remake(prob; p = SVector{6, T1}(parameters[i]...),
           u0 = SVector{4, T1}(parameters[i][4], parameters[i][4],
           ↪  parameters[i][4],
                               parameters[i][4]))
end
```

```julia

eprob = EnsembleProblem(prob, prob_func = prob_func, safetycopy = false)

saveat = T1(0.0f0):T1(1.0f0):T1(1000.0f0)
dt = T1(0.1f0)

esol = solve(eprob, GPUTsit5(), EnsembleGPUKernel(CUDA.CUDABackend()); trajectories
 ↪  = numberOfParameters)
```

Listing 4: Code example for demonstrating composability to GPU acceleration for reaction models using DiffEqGPU.jl.

Chapter 7

Conclusion

Parallel computing faces two significant challenges today. Firstly, many algorithms are inherently complex. As a result, there is often a divide between experts in numerical analysis, who focus on developing efficient algorithms, and those in computer science, who specialize in parallelizing algorithms. This distinction is evident in fields like ODE solving, where projects such as XBraid [57] from the exascale initiative utilize parallel-in-time integrators. However, these integrators do not exhibit improved performance until utilizing 256 or more cores. While XBraid may not feature an optimal algorithm, parallelizing is relatively easy. The second challenge lies in the integration of effective parallelism with robust algorithms. This task is inherently intricate and typically resolved on a domain-by-domain basis. Examples include the development of climate models like General Circulation Models (GCMs) [118] and molecular dynamics codes like AMBER [119]. However, these codes are often specialized, opaque, and time-consuming to develop, lacking versatility across different domains. We identified the need for general-purpose differential equation solvers for users where efficient parallelism should be easily accessible.

In this thesis, we demonstrated two general-purpose methods for parallelizing differential equation solvers. We demonstrated algorithms for within-step parallelism by parallelizing extrapolation methods, resulting in the first general-purpose, extensively benchmarked, and open-source implementation. The GPU-based ODE solvers for solving ensemble problems present Julia's first portable GPU application that supports all four relevant GPU vendors. This was achieved by doing the groundwork to make it possible, i.e., improving the JuliaGPU stack (adding back-ends, fixing bugs, etc.). Our work thus provides a roadmap for how others can achieve true vendor agnosticism and the reality of the work necessary.

7.1 Future works

As noted in Chapter 4, Extrapolation methods are generally inefficient in comparison to other methods such as Rosenbrock [120] or SDIRK [121] methods. In none of the numerous stiff ODE benchmarks by Hairer II [17] or the SciML suite[1] do the implicit extrapolation methods achieve SOA performance. Thus, while the parallelization done here was able to achieve SOA, even better results could likely be obtained by developing parallel Rosenbrock

[1]https://github.com/SciML/SciMLBenchmarks.jl

[120] or SDIRK [121] methods (using theory described in [56]) and implementing them using our parallelization techniques. This would have the trade-off of being tied to a specific core count but could achieve better performance through tableau optimization.

Based on Chapter 5, there are opportunities for improvements with `EnsembleGPUKernel`. For example, while using stack-allocated arrays provides a workaround to using arrays inside GPU kernels, they are unsuitable for higher-dimensional problems due to the limited memory of static allocations. The model might compile, but there might not be any realizable speedups. Flexibility in supporting mutation within the ODE function can also be extended. This could be achieved using mutable static arrays, which require special tricks to compile them with the GPU kernels. The user is also limited in terms of using features such as broadcast and calls to BLAS. Support for other differential equation problems, such as differential-algebraic equations (DAEs) and stiff SDEs, remains an area of contribution. Experimental support exists for event handling; however, some callbacks can generate GPU-incompatible code due to limitations in the Julia compiler. Improvements to the compiler's escape analysis and effects modeling are currently being implemented, and are expected to resolve this issue.

Notably, the work of incorporating parallelism into efficient numerical methods can be extended to model calibration routines such as parameter estimation of differential equations. As a first step toward that goal, we want to address the limitation in training high-fidelity models currently done by gradient-based optimizers by handling only one solution at a time. A potential solution involves developing specialized global optimization tools, like multi-start methods or evolutionary approaches, leveraging GPU ensembles for search exploration. Given the highly non-convex landscapes of many ODE optimization problems, efficient GPU-accelerated global optimization is crucial in domains like pharmacometrics, chemical engineering, and robotics. I aim to contribute to this field by creating optimization libraries and solvers that exploit GPU-batched evaluations, enabling rapid global optimization where only local optimization was previously tractable. [122] showcased some early results by combining Particle Swarm Optimization (PSO) [123] with the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm [124], for non-convex optimization problems. We aim to follow up on this work and demonstrate the combining efficient parallelism with numerical optimization methods.

Appendix A

Parallel Extrapolation Methods

A.0.1 Models

The first test problem is the ROBER Problem:

$$\begin{aligned}\frac{dy_1}{dt} &= -k_1y_1 + k_3y_2y_3,\\ \frac{dy_2}{dt} &= k_1y_1 + -k_2y_2^2 - k_3y_2y_3,\\ \frac{dy_3}{dt} &= k_2y_2^2.\end{aligned} \tag{A.1}$$

The initial conditions are $y = [1.0, 0.0, 0.0]$ and $k = (0.04, 3e7, 1e4)$. The time span for integration is $t = (0.0\,s, 1e5\,s)$.[65], [17] The second test problem is OREGO:

$$\frac{dy_1}{dt} = -k_1(y_2 + y_1(1 - k_2y_1 - y_2)), \tag{A.2}$$

$$\frac{dy_2}{dt} = \frac{y_3 - (1 + y_1)y_2}{k_1}, \tag{A.3}$$

$$\frac{dy_3}{dt} = k_3(y_1 - y_3). \tag{A.4}$$

The initial conditions are $y = [1.0, 2.0, 3.0]$ and $k = (77.27, 8.375 \times 10^{-6}, 0.161)$. The time span for integration is $t = (0.0\,s, 30.0\,s)$. The third test problem is HIRES:

$$\frac{dy_1}{dt} = -1.71y_1 + 0.43y_2 + 8.32y_3 + 0.0007, \tag{A.5}$$

$$\frac{dy_2}{dt} = 1.71y_1 - 8.75y_2, \tag{A.6}$$

$$\frac{dy_3}{dt} = -10.03y_3 + 0.43y_4 + 0.035y_5, \tag{A.7}$$

$$\frac{dy_4}{dt} = 8.32y_2 + 1.71y_3 - 1.12y_4, \tag{A.8}$$

$$\frac{dy_5}{dt} = -1.745y_5 + 0.43y_6 + 0.43y_7, \tag{A.9}$$

$$\frac{dy_6}{dt} = -280.0y_6y_8 + 0.69y_4 \tag{A.10}$$

$$+ 1.71y_5 - 0.43y_6 + 0.69y_7, \tag{A.11}$$

$$\frac{dy_7}{dt} = 280.0y_6y_8 - 1.81y_7, \tag{A.12}$$

$$\frac{dy_8}{dt} = -280.0y_6y_8 + 1.81y_7. \tag{A.13}$$

The initial conditions are:

$$y = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0057]. \tag{A.14}$$

The time span for integration is $t = (0.0\,s, 321.8122\,s)$. The fourth test problem is POLLU:

$$\frac{dy_1}{dt} = -k_1y_1 - k_{10}y_{11}y_1 - k_{14}y_1y_6 - k_{23}y_1y_4 \tag{A.15}$$

$$- k_{24}y_{19}y_1 + k_2y_2y_4 + k_3y_5y_2 + k_9y_{11}y_2 \tag{A.16}$$

$$+ k_{11}y_{13} + k_{12}y_{10}y_2 + k_{22}y_{19} + k_{25}y_{20}, \tag{A.17}$$

$$\frac{dy_2}{dt} = -k_2y_2y_4 - k_3y_5y_2 - k_9y_{11}y_2 - k_{12}y_{10}y_2 \tag{A.18}$$

$$+ k_1y_1 + k_{21}y_{19}, \tag{A.19}$$

$$\frac{dy_3}{dt} = -k_{15}y_3 + k_1y_1 + k_{17}y_4 + k_{19}y_{16} + k_{22}y_{19}, \tag{A.20}$$

$$\frac{dy_4}{dt} = -k_2y_2y_4 - k_{16}y_4 - k_{17}y_4 - k_{23}y_1y_4 + k_{15}y_3, \tag{A.21}$$

$$\frac{dy_5}{dt} = -k_3y_5y_2 + 2k_4y_7 + k_6y_7y_6 + k_7u_9 \tag{A.22}$$

$$+ k_{13}y_{14} + k_{20}y_{17}y_6, \tag{A.23}$$

$$\frac{dy_6}{dt} = -k_6y_7y_6 - k_8y_9y_6 - k_{14}y_1y_6 - k_{20}y_{17}y_6 \tag{A.24}$$

$$+ k_3y_5y_2 + 2k_{18}u_{16}, \tag{A.25}$$

$$\frac{dy_7}{dt} = -k_4y_7 - k_5y_7 - k_6y_7y_6 + k_{13}y_{14}, \tag{A.26}$$

$$\frac{dy_8}{dt} = k_4y_7 + k_5y_7 + k_6y_7y_6 + k_7y_9, \tag{A.27}$$

$$\frac{dy_9}{dt} = -k_7y_9 - k_8y_9y_6, \tag{A.28}$$

$$\frac{dy_{10}}{dt} = -k_{12}y_{10}y_2 + k_7y_9 + k_9y_{11}y_2, \tag{A.29}$$

$$\frac{dy_{11}}{dt} = -k_9y_{11}y_2 - k_{10}y_{11}y_1 + k_8y_9y_6 + k_{11}y_{13}, \tag{A.30}$$

$$\frac{dy_{12}}{dt} = k_9y_{11}y_2, \tag{A.31}$$

$$\frac{dy_{13}}{dt} = -k_{11}y_{13} + k_{10}y_{11}y_1, \tag{A.32}$$

$$\frac{dy_{14}}{dt} = -k_{13}y_{14} + k_{12}y_{10}y_2, \tag{A.33}$$

$$\frac{dy_{15}}{dt} = k_{14}y_1y_6, \tag{A.34}$$

$$\frac{dy_{16}}{dt} = -k_{18}y_{16} - k_{19}y_{16} + k_{16}y_4. \tag{A.35}$$

$$\frac{dy_{17}}{dt} = -k_{20}y_{17}y_6, \tag{A.36}$$

$$\frac{dy_{18}}{dt} = k_{20}y_{17}y_6, \tag{A.37}$$

$$\frac{dy_{19}}{dt} = -k_{21}y_{19} - k_{22}y_{19} - k_{24}y_{19}y_1 + k_{23}y_1y_4 + k_{25}y_{20}, \tag{A.38}$$

$$\frac{dy_{20}}{dt} = -k_{25}y_{20} + k_{24}y_{19}y_1. \tag{A.39}$$

$$k = [0.35, 26.6, 12300.0, 0.00086, 0.00082, 15000.0, 0.00013, 24000.0, 16500.0, 9000.0, 0.022, 12000.0, 1.88, 16300.0, 4.8e6, 0.00035, 0.0175, 1.0e8, 4.44e11, 1240.0, 2.1, 5.78, 0.0474, 1780.0, 3.12].$$

$$y = [0.0, 0.2, 0.0, 0.04, 0.0, 0.0, 0.1, 0.3, 0.017, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.007, 0.0, 0.0, 0.0].$$

The time span for integration is $t = (0.0\,s, 60.0\,s)$.

The QSP model is "BIOMD0000000452 QSP Model, Bidkhori2012 - normal EGFR signalling" [70] from BioModels Database [125]. This is a Systems Biology Markup Model (SBML) [126], parsed using SBMLToolkit.jl and Catalyst.jl [16], [116]. It notably has 109 states and 188 parameters

A.0.2 Benchmarks

100 Linear ODEs

The test tolerances for the solvers in the benchmark Fig. 4.1 are relative tolerances in the range of $(10^{-7}, 10^{-13})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-16})$.

Table A.1: Tuned parameters for 100 Linear ODEs

Extrapolation Method/Order	minimum	initial	maximum
Midpoint Deuflhard	5	10	11
Midpoint Hairer Wanner	5	10	11

Rober

The test tolerances for the solvers in the benchmark Fig. 4.2 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [54]. The parameters for the solvers are tuned to these settings:

Table A.2: Tuned parameters for ROBER

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	3	5	12
Implicit Euler Barycentric	4	5	12
Implicit Hairer Wanner	2	5	10

Orego

The test tolerances for the solvers in the benchmark Fig. 4.3 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is Rodas5 [17]. The parameters for the solvers are tuned to these settings:

Table A.3: Tuned parameters for OREGO

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	3	4	12
Implicit Euler Barycentric	3	4	12
Implicit Hairer Wanner	2	5	10

Hires

The test tolerances for the solvers in the benchmark Fig. 4.4 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is Rodas5 [17]. The parameters for the solvers are tuned to these settings:

Table A.4: Tuned parameters for HIRES

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	4	7	12
Implicit Euler Barycentric	4	7	12
Implicit Hairer Wanner	3	6	10

POLLUTION

The test tolerances for the solvers in the benchmark Fig. 4.5 are relative tolerances in the range of $(10^{-8}, 10^{-10})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-13})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [54]. The parameters for the solvers are tuned to these settings:

Table A.5: Tuned parameters for POLLU

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	5	6	12
Implicit Euler Barycentric	5	6	12
Implicit Hairer Wanner	3	6	10

QSP Model

The test tolerances for the solvers in the benchmark Fig. 4.6 are relative tolerances in the range of $(10^{-6}, 10^{-10})$ and corresponding absolute tolerances are $(10^{-9}, 10^{-13})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [54]. The parameters for the solvers are tuned to these settings:

Table A.6: Tuned parameters for QSP Model

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	8	9	12
Implicit Euler Barycentric	7	8	12
Implicit Hairer Wanner	2	5	10

References

[1] D. S. Jones, M. Plank, and B. D. Sleeman, *Differential equations and mathematical biology*. Chapman and Hall/CRC, 2009.

[2] R. Courant and D. Hilbert, *Methods of mathematical physics: partial differential equations*. John Wiley & Sons, 2008.

[3] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in Neural Information Processing systems*, vol. 31, 2018.

[4] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman, "Universal differential equations for scientific machine learning," *arXiv preprint arXiv:2001.04385*, 2020.

[5] U. Nowak, R. Ehrig, and L. Oeverdieck, "Parallel extrapolation methods and their application in chemical engineering," in *International Conference on High-Performance Computing and Networking*, Springer, 1998, pp. 419–428.

[6] D. Ketcheson and U. bin Waheed, "A comparison of high-order explicit runge–kutta, extrapolation, and deferred correction methods in serial and parallel," *Communications in Applied Mathematics and Computational Science*, vol. 9, no. 2, pp. 175–200, 2014.

[7] Utkarsh, C. Elrod, Y. Ma, K. Althaus, and C. Rackauckas, "Parallelizing explicit and implicit extrapolation methods for ordinary differential equations," in *IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA: IEEE, 2022, pp. 1–9. DOI: 10.1109/HPEC55821.2022.9926357.

[8] C. Kühn, C. Wierling, A. Kühn, E. Klipp, G. Panopoulou, H. Lehrach, and A. J. Poustka, "Monte carlo analysis of an ode model of the sea urchin endomesoderm network," *BMC Systems Biology*, vol. 3, p. 83, 2009.

[9] S. Marino, I. B. Hogue, C. J. Ray, and D. E. Kirschner, "A methodology for performing global uncertainty and sensitivity analysis in systems biology," *Journal of Theoretical Biology*, vol. 254, no. 1, pp. 178–196, 2008.

[10] B. Iooss and P. Lemaitre, "A review on global sensitivity analysis methods," in *Uncertainty Management in Simulation-Optimization of Complex Systems: Algorithms and Applications*, G. Dellino and C. Meloni, Eds., 1 vols., Boston, MA: Springer, 2015, pp. 101–122.

[11] E. Hairer, S. Norsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*. Berlin, Heidelberg: Springer, Jan. 1993, vol. 8, ISBN: 978-3-540-56670-0. DOI: 10.1007/978-3-540-78862-1.

[12] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, Github, 2018. URL: http://github.com/google/jax.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[14] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[15] U. Utkarsh, V. Churavy, Y. Ma, T. Besard, T. Gymnich, A. R. Gerlach, A. Edelman, and C. Rackauckas, "Automated translation and accelerated solving of differential equations on multiple GPU platforms," *arXiv preprint arXiv:2304.06835*, 2023.

[16] C. Rackauckas and Q. Nie, "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia," *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[17] E. Hairer and G. Peters, *Solving Ordinary Differential Equations II*. Berlin Heidelberg: Springer, 1991.

[18] L. F. Shampine and S. Thompson, "Stiff systems," *Scholarpedia*, vol. 2, no. 3, p. 2855, 2007.

[19] D. J. Higham and L. N. Trefethen, "Stiffness of odes," *BIT Numerical Mathematics*, vol. 33, pp. 285–303, 1993.

[20] C. Runge, "Über die numerische auflösung von differentialgleichungen," *Mathematische Annalen*, vol. 46, no. 2, pp. 167–178, 1895.

[21] W. Kutta, "Beitrag zur näherungsweisen integration totaler differentialgleichungen," Ph.D. dissertation, Ludwig Maximilian University of Munich, Germany, 1901.

[22] J. R. Dormand and P. J. Prince, "A family of embedded runge-kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980.

[23] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," *SIAM Journal on Scientific Computing*, vol. 18, no. 1, pp. 1–22, 1997. DOI: 10.1137/S1064827594276424.

[24] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, PA, USA: SIAM, 1998, vol. 61.

[25] H. H. Rosenbrock, "Some general implicit processes for the numerical solution of differential equations," *The Computer Journal*, vol. 5, no. 4, pp. 329–330, 1963. DOI: 10.1093/comjnl/5.4.329.

[26] C. Rackauckas and Q. Nie, "Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia," *The Journal of Open Research Software*, vol. 5, no. 1, 2017. DOI: 10.5334/jors.151.

[27] A. Kværnø, "Singly diagonally implicit runge–kutta methods with an explicit first stage," *BIT Numerical Mathematics*, vol. 44, no. 3, pp. 489–502, 2004.

[28] P. Kidger, "On Neural Differential Equations," Ph.D. dissertation, University of Oxford, 2021.

[29] A. Aitken, "On interpolation by iteration of proportional parts, without the use of differences," *Proceedings of the Edinburgh Mathematical Society*, vol. 3, no. 1, pp. 56–76, 1932.

[30] E. H. Neville, *Iterative interpolation.* St. Joseph's IS Press, 1934.

[31] P. Deuflhard, "Order and stepsize control in extrapolation methods," *Numerische Mathematik*, vol. 41, no. 3, pp. 399–422, 1983.

[32] W. Romberg, "Vereinfachte numerische integration," *Norske Vid. Selsk. Forh.*, vol. 28, pp. 30–36, 1955.

[33] R. Bulirsch and J. Stoer, "Numerical treatment of ordinary differential equations by extrapolation methods," *Numerische Mathematik*, vol. 8, no. 1, pp. 1–13, 1966.

[34] H. J. Stetter, "Symmetric two-step algorithms for ordinary differential equations," *Computing*, vol. 5, no. 3, pp. 267–280, 1970.

[35] J.-P. Berrut and L. N. Trefethen, "Barycentric lagrange interpolation," *SIAM review*, vol. 46, no. 3, pp. 501–517, 2004.

[36] K. Althaus, *Theory and implementation of the adaptive explicit midpoint rule including order and stepsize control*, 2018. URL: https://github.com/AlthausKonstantin/Extrapolation/blob/master/Bachelor%20Theseis.pdf.

[37] M. Webb, L. N. Trefethen, and P. Gonnet, "Stability of barycentric interpolation formulas for extrapolation," *SIAM Journal on Scientific Computing*, vol. 34, no. 6, A3009–A3015, 2012.

[38] G. G. Dahlquist, "A special stability problem for linear multistep methods," *BIT Numerical Mathematics*, vol. 3, no. 1, pp. 27–43, 1963.

[39] G. Bader and P. Deuflhard, "A semi-implicit mid-point rule for stiff systems of ordinary differential equations," *Numerische Mathematik*, vol. 41, no. 3, pp. 373–398, 1983.

[40] W. B. Gragg, "Repeated extrapolation to the limit in the numerical solution of ordinary differential equations.," CALIFORNIA UNIV LOS ANGELES, Tech. Rep., 1964.

[41] B. Lindberg, "On smoothing and extrapolation for the trapezoidal rule," *BIT*, vol. 11, no. 1, pp. 29–52, Mar. 1971. DOI: 10.1007/bf01935326. URL: https://doi.org/10.1007%2Fbf01935326.

[42] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[43] R. C. Merton, "Theory of rational option pricing," *The Bell Journal of Economics and Management Science*, vol. 1, pp. 141–183, 1973.

[44] D. J. Wilkinson, *Stochastic modelling for systems biology*. Boca Raton, FL, USA: CRC Press, 2018.

[45] D. T. Gillespie, "The chemical langevin equation," *The Journal of Chemical Physics*, vol. 113, no. 1, pp. 297–306, 2000.

[46] P. E. Kloeden and E. Platen, *Stochastic Differential Equations*. Berlin Heidelberg: Springer, 1992.

[47] A. Tocino and R. Ardanuy, "Runge–kutta methods for numerical solution of stochastic differential equations," *Journal of Computational and Applied Mathematics*, vol. 138, no. 2, pp. 219–241, 2002.

[48] C. Rackauckas and Q. Nie, "Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations," in *IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA: IEEE, 2020, pp. 1–8.

[49] A. Iserles and S. Nørsett, "On the theory of parallel runge—kutta methods," *IMA Journal of numerical Analysis*, vol. 10, no. 4, pp. 463–488, 1990.

[50] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, *et al.*, "Petsc users manual," 2019.

[51] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, *et al.*, *LAPACK users' guide*. SIAM, 1999.

[52] M. J. Gander and S. Vandewalle, "Analysis of the parareal time-parallel time-integration method," *SIAM Journal on Scientific Computing*, vol. 29, no. 2, pp. 556–578, 2007.

[53] Y. Maday and G. Turinici, "A parareal in time procedure for the control of partial differential equations," *Comptes Rendus Mathematique*, vol. 335, no. 4, pp. 387–392, 2002.

[54] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363–396, Sep. 2005. DOI: 10.1145/1089014.1089020. URL: https://doi.org/10.1145%2F1089014.1089020.

[55] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[56] K. Burrage, *Parallel and sequential methods for ordinary differential equations*. Clarendon Press, 1995.

[57] X. Team, *XBraid: Parallel multigrid in time*, http://llnl.gov/casc/xbraid, 2013.

[58] J. Schroder and R. Falgout, "Xbraid tutorial," in *18th Copper Mountain Conference on Multigrid Methods*, Colorado: Colorado, 2017.

[59] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.

[60] T. Davis, W. Hager, and I. Duff, "Suitesparse," *URL: faculty. cse. tamu. edu/davis/-suitesparse. html*, 2014.

[61] L. Petzold, "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations," *SIAM journal on scientific and statistical computing*, vol. 4, no. 1, pp. 136–148, 1983.

[62] C. Elrod, *Polyester.jl*, https://github.com/JuliaSIMD/Polyester.jl, 2021.

[63] E. Hairer, *Fortran and matlab codes*, 2004.

[64] J. H. Verner, "Numerically optimal runge–kutta pairs with interpolants," *Numerical Algorithms*, vol. 53, no. 2, pp. 383–396, 2010.

[65] H. H. Robertson, "Numerical integration of systems of stiff ordinary differential equations with special structure," *IMA Journal of Applied Mathematics*, vol. 18, no. 2, pp. 249–263, 1976.

[66] A. M. Zhabotinsky, "Belousov-zhabotinsky reaction," *Scholarpedia*, vol. 2, no. 9, p. 1435, 2007.

[67] R. J. Field and R. M. Noyes, "Oscillations in chemical systems. iv. limit cycle behavior in a model of a real chemical reaction," *The Journal of Chemical Physics*, vol. 60, no. 5, pp. 1877–1884, 1974.

[68] E. Schäfer, "A new approach to explain the "high irradiance responses" of photomorphogenesis on the basis of phytochrome," *Journal of Mathematical Biology*, vol. 2, no. 1, pp. 41–56, 1975.

[69] J. G. Verwer, "Gauss–seidel iteration for stiff odes from chemical kinetics," *SIAM Journal on Scientific Computing*, vol. 15, no. 5, pp. 1243–1250, 1994.

[70] G. Bidkhori, A. Moeini, and A. Masoudi-Nejad, "Modeling of tumor progression in nsclc and intrinsic resistance to tki in loss of pten expression," *PloS one*, vol. 7, no. 10, e48004, 2012.

[71] K. Soetaert, T. Petzoldt, and R. W. Setzer, "Solving differential equations in r: Package desolve," *Journal of statistical software*, vol. 33, pp. 1–25, 2010.

[72] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[73] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," *SIAM Journal on Scientific Computing*, vol. 18, no. 1, pp. 1–22, 1997.

[74] A. Tarantola, *Inverse Problem Theory and Methods for Model Parameter Estimation.* Philadelphia: SIAM, 2005.

[75] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

[76] F. Hegedűs, "Program package mpgos: Challenges and solutions during the integration of a large number of independent ode systems using gpus," *Communications in Nonlinear Science and Numerical Simulation*, vol. 97, p. 105 732, 2021.

[77] X. Li, T.-K. L. Wong, R. T. Q. Chen, and D. K. Duvenaud, "Scalable gradients and variational inference for stochastic differential equations," in *Symposium on Advances in Approximate Bayesian Inference*, PMLR, -: PMLR, 2020, pp. 1–28.

[78] Y. Zhou, J. Liepe, X. Sheng, M. P. H. Stumpf, and C. Barnes, "Gpu accelerated biochemical network simulation," *Bioinformatics*, vol. 27, no. 6, pp. 874–876, 2011.

[79] M. Fernando, D. Neilsen, E. Hirschmann, Y. Zlochower, H. Sundar, O. Ghattas, and G. Biros, "A gpu-accelerated amr solver for gravitational wave propagation," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, Dallas, Texas: IEEE, 2022, pp. 1078–1092.

[80] T. Zhao, S. De, B. Chen, J. Stokes, and S. Veerapaneni, "Overcoming barriers to scalability in variational quantum monte carlo," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021, pp. –, ISBN: 9781450384421. DOI: 10.1145/3458817.3476219. URL: https://doi.org/10.1145/3458817.3476219.

[81] S. Le Grand, A. W. Götz, and R. C. Walker, "Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations," *Computer Physics Communications*, vol. 184, no. 2, pp. 374–380, 2013.

[82] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos, "Arrayfire: A gpu acceleration platform," in *Modeling and simulation for defense systems and applications VII*, SPIE, vol. 8403, Baltimore, Maryland, USA: SPIE, 2012, pp. 49–56.

[83] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems, Jade edition*, Boston: Elsevier, 2012, pp. 359–371.

[84] D. Demidov, *Vexcl: Vector expression template library for OpenCL*, 2012.

[85] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, p. 66, 2010.

[86] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[87] K. Ahnert and M. Mulansky, "Odeint–solving ordinary differential equations in c++," in *AIP Conference Proceedings*, American Institute of Physics, vol. 1389, Halkidiki, Greece: American Institute of Physics, 2011, pp. 1586–1589.

[88] K. Ahnert, D. Demidov, and M. Mulansky, "Solving ordinary differential equations on GPUs," *Numerical Computations with GPUs*, vol. 1, pp. 125–157, 2014.

[89] D. Nagy, L. Plavecz, and F. Hegedűs, "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on gpus and cpus," *Communications in Nonlinear Science and Numerical Simulation*, vol. 112, p. 106 521, 2022.

[90] A. C. Hindmarsh, “Odepack, a systemized collection of ode solvers,” in *Scientific Computing*, R. S. Stepleman, Ed., 1 vols., Amsterdam: North-Holland, 1983, pp. 55–64.

[91] A. C. Hindmarsh and L. R. Petzold, “Algorithms and software for ordinary differential equations and differential-algebraic equations, part ii: Higher-order methods and software packages,” *Computers in Physics*, vol. 9, no. 2, pp. 148–155, 1995.

[92] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, “Cupy: A numpy-compatible library for nvidia gpu calculations,” in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems*, Long Beach, CA, USA: NIPS, 2017, pp. –. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[93] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15, New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–6, ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[94] T. Besard, C. Foket, and B. De Sutter, “Effective extensible programming: Unleashing Julia on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2018.

[95] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, “Rapid software prototyping for heterogeneous and distributed platforms,” *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.

[96] T. Besard, *Juliagpu/gpucompiler.jl: V1.8.0*, version v0.18.0, Github, Mar. 2023. DOI: 10.5281/zenodo.7807140. URL: https://doi.org/10.5281/zenodo.7807140.

[97] T. Besard, *Juliagpu/gpuarrays.jl: V8.6.4*, version v8.6.4, Github, Mar. 2023. DOI: 10.5281/zenodo.7807091. URL: https://doi.org/10.5281/zenodo.7807091.

[98] J. Samaroo, V. Churavy, A. Smirnov, *et al.*, *Juliagpu/amdgpu.jl: V0.4.8*, version v0.4.8, Github, Feb. 2023. DOI: 10.5281/zenodo.7641665. URL: https://doi.org/10.5281/zenodo.7641665.

[99] T. Besard, *Oneapi.jl*, version v1.1.0, Github, Mar. 2023. DOI: 10.5281/zenodo.7789142. URL: https://doi.org/10.5281/zenodo.7789142.

[100] T. Besard and M. Hawkins, *Metal.jl*, version v0.3.0, If you use this software, please cite it as below., Github, Mar. 2023. DOI: 10.5281/zenodo.7789146. URL: https://doi.org/10.5281/zenodo.7789146.

[101] K. Obenschain, D. Schwer, and A. Sharma, *Initial assessment of the amd mi50 gpgpus for scientific and machine learning applications*, Research poster presented at ISC High Performance 2020, 2020.

[102] C. Brown, A. Abdelfattah, S. Tomov, and J. Dongarra, “Design, optimization, and benchmarking of dense linear algebra algorithms on amd gpus,” in *IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA: IEEE, 2020, pp. 1–7.

[103] V. Churavy, D. Aluthge, J. Samaroo, *et al.*, *Juliagpu/kernelabstractions.jl: V0.9.1*, version v0.9.1, JuliaGPU, Mar. 2023. DOI: 10.5281/zenodo.7770454. URL: https://doi.org/10.5281/zenodo.7770454.

[104] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, IEEE, Hangzhou, China: IEEE, 2010, pp. 344–350.

[105] C. Tsitouras, "Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption," *Computers & Mathematics with Applications*, vol. 62, no. 2, pp. 770–775, 2011.

[106] G. Steinebach, "Construction of rosenbrock–wanner method rodas5p and numerical benchmarks within the julia differential equations package," *BIT Numerical Mathematics*, vol. 63, no. 2, p. 27, 2023. DOI: 10.1007/s10543-023-00967-x.

[107] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, *Automatic differentiation of algorithms: from simulation to optimization.* Springer Science & Business Media, 2002.

[108] A. Tocino and J. Vigo-Aguiar, "Weak second order conditions for stochastic runge–kutta methods," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 507–523, 2002.

[109] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of Atmospheric Sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[110] D. W. Walker and J. J. Dongarra, "MPI: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[111] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, Springer, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104.

[112] S. Byrne, L. C. Wilcox, and V. Churavy, "Mpi.jl: Julia bindings for the message passing interface," in *Proceedings of the JuliaCon Conferences*, vol. 1, Online: JuliaCon, 2021, p. 68.

[113] M. S. Nobile, P. Cazzaniga, D. Besozzi, and G. Mauri, "Ginsoda: Massive parallel integration of stiff ode systems on gpus," *The Journal of Supercomputing*, vol. 75, no. 12, pp. 7844–7856, 2019.

[114] T. Loman, "How bacteria tune mixed positive/negative feedback loops to generate diverse gene expression dynamics," Ph.D. dissertation, University of Cambridge, U.K., 2022.

[115] P. Barham and M. Isard, "Machine learning systems are stuck in a rut," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19, Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 177–183, ISBN: 9781450367271. DOI: 10.1145/3317550.3321441. URL: https://doi.org/10.1145/3317550.3321441.

[116] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, *Modelingtoolkit: A composable graph transformation system for equation-based modeling*, 2021. arXiv: 2103.05244 [cs.MS].

[117] T. Loman, Y. Ma, V. Ilin, S. Gowda, N. Korsbo, N. Yewale, C. V. Rackauckas, and S. A. Isaacson, "Catalyst: Fast biochemical modeling with julia," *bioRxiv*, pp. 2022–07, 2022.

[118] S. L. Grotch and M. C. MacCracken, "The use of general circulation models to predict regional climatic change," *Journal of climate*, vol. 4, no. 3, pp. 286–303, 1991.

[119] D. A. Pearlman, D. A. Case, J. W. Caldwell, W. S. Ross, T. E. Cheatham III, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman, "Amber, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules," *Computer Physics Communications*, vol. 91, no. 1-3, pp. 1–41, 1995.

[120] H. Rosenbrock, "Some general implicit processes for the numerical solution of differential equations," *The Computer Journal*, vol. 5, no. 4, pp. 329–330, 1963.

[121] R. Alexander, "Diagonally implicit runge–kutta methods for stiff ode's," *SIAM Journal on Numerical Analysis*, vol. 14, no. 6, pp. 1006–1021, 1977.

[122] Utkarsh, V. K. Dixit, J. Samaroo, A. Pal, A. Edelman, and C. V. Rackauckas, "Efficient GPU-accelerated global optimization for inverse problems," in *ICLR 2024 Workshop on AI4DifferentialEquations In Science*, 2024. URL: https://openreview.net/forum?id=nD10o1ge97.

[123] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[124] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.

[125] C. Li, M. Donizelli, N. Rodriguez, H. Dharuri, L. Endler, V. Chelliah, L. Li, E. He, A. Henry, M. I. Stefan, *et al.*, "Biomodels database: An enhanced, curated and annotated resource for published quantitative kinetic models," *BMC systems biology*, vol. 4, no. 1, pp. 1–14, 2010.

[126] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, *et al.*, "The systems biology markup language (sbml): A medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.