Device Agnostic GPU Parallelization of Stiff Ordinary Differential Equations

Utkarsh
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
utkarsh5@mit.edu

Prakitr Srisuma
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
prakitrs@mit.edu

***Abstract*—We demonstrate a high-performance and vendor-agnostic method for massively parallel solving of ensembles of stiff ordinary differential equations (ODEs) on GPUs. The method is integrated with a widely used differential equation solver library in a high-level language (Julia's DifferentialEquations.jl) and enables GPU acceleration without requiring code changes by the user. Our approach achieves state-of-the-art performance by performing 20–50× faster than the vectorized-map (`vmap`) approach implemented in Julia and JAX. Performance evaluation on NVIDIA, AMD, Intel, and Apple GPUs demonstrates performance portability and vendor-agnosticism. The implemented solvers successfully extend the support of all the features existing in `DiffEqGPU.jl` solvers, such as MPI-compatibility, event handling, automatic differentiation, and incorporating of datasets via the GPU's texture memory, allowing scientists to take advantage of GPU acceleration on all major current architectures without changing their model code and without loss of performance.**

I. 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 [1], performing uncertainty quantification [2–4], and calculating global sensitivity analysis [3, 5]. In spite of the fact that such an embarrassingly parallel issue lends itself to being well-suited for acceleration via GPU hardware, the traditional hurdle to the adoption of GPU-parallel solvers by scientists and engineers who are less programming knowledgeable has been the programming requirements. The fact that the user must provide a function to define the ordinary differential equation (ODE) presents the primary challenge that must be overcome in order to successfully target GPUs with general ODE solver software. Therefore, high-level ODE solver software has typically consisted of higher order functions that take as input a function written in a high-level language such as MATLAB [6], Python (SciPy [7]), or Julia (DifferentialEquations.jl [8]). This is done to lower the entry barrier for scientists and engineers. In order to target GPUs, previous software such as MPGOS [9] 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 like those found in machine learning libraries like PyTorch [10] or JAX [11].

In this project, we demonstrate a performant, composable, and vendor-agnostic method for model-specific kernel generation to solve massively parallel ensembles of stiff ordinary differential equations (ODEs). Our software transforms code which targets a widely used differential equation solver library in a high level language (Julia's DifferentialEquations.jl [8]) and automatically generates optimized GPU kernels without requiring code changes by the end user. Our kernel generation approach achieves state-of-the-art performance by performing 20–50× faster than the vectorized-map (`vmap`) approach implemented in JAX. We showcase the vendor-agnostic aspect of our approach by benchmarking the results against 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. Together, these applications enable scientists and engineers to target all main GPU platforms without performance loss.

II. Related Work

Although GPUs have been extensively used to accelerate computations in applications such as molecular simulation, biological systems, and physics [12–15], these implementations are typically CUDA kernels written for the specific models and are therefore not general ODE solver software. Array abstraction frameworks such as ArrayFire [16], Thrust [17], and VexCL [18], JAX [11], and PyTorch [10] have been used in previous attempts to simplify the targeting of GPUs with a general ODE solver software. These frameworks enable the adaptation of code written on high-level array abstractions and generate highly optimized code for backends such as OpenCL and CUDA. Boost's ODEINT [19–21] enables the use of ODE solvers that are compatible with GPU backends such as CUDA and OpenCL without modification. JAX's Diffrax generates solvers for ensembles of ODEs on GPUs using the vectorized map functionality (`vmap`) of JAX [22].

Similar results were observed in culsoda [12], a CUDA translation of the widely used LSODE solver [23, 24], which was similarly limited due to requiring ODE models to be written in CUDA and compiled into the kernels. The ODE and SDE

solvers with custom GPU kernels and multiple GPU platform support were developed in Julia, achieving orders of magnitude of performance compared to array-based parallelism implementations [25]. However, the currently available solvers are based on explicit methods, which generally are not efficient for solving stiff ODEs. Stiff ODEs are common when modeling natural phenomena and are harder to simulate [26, 27].

III. Numerical Methods for Stiff Ordinary Differential Equations

A system of ordinary differential equations (ODEs) is given by

$$\frac{du}{dt} = f(u, p, t), \tag{1}$$

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 various numerical methods for solving stiff ODEs [28, 29]. One of the most common algorithms used in various ODE solver packages, e.g., Julia, MATLAB, is known as the *Rosenbrock method* [29, 30]. The general formulas of an s-stage Rosenbrock method is given by

$$\begin{aligned} k_i =& hf(u_n + \sum_{j=1}^{i-1} \alpha_{ij} k_j, t_n + \alpha_i h) + \beta_i h^2 f_t(u_n, t_n) \\ &+ hf_u(u_n, t_n) \sum_{j=1}^{i} \beta_{ij} k_j, \end{aligned} \tag{2}$$

$$u_{n+1} = u_n + \sum_{j=1}^{s} \delta_j k_j, \tag{3}$$

where f_u and f_t are the Jacobians given by

$$f_u \approx \frac{\partial f}{\partial u}(u_n, t_n), \tag{4}$$

$$f_t \approx \frac{\partial f}{\partial t}(u_n, t_n), \tag{5}$$

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 ($t_{n+1} = t_n + h$), and $\alpha_{ij}, \beta_{ij}, \delta_j$ are the coefficients, with α_i, β_i satisfying

$$\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij}, \tag{6}$$

$$\beta_i = \sum_{j=1}^{i} \beta_{ij}. \tag{7}$$

For adaptive step-size control, we define

$$q = \left\| \frac{E}{\text{atol} + \max(|u_n|, |u_{n+1}|) \cdot \text{rtol}} \right\|, \tag{8}$$

where E is the error, atol is the absolute tolerance, and rtol is the relative tolerance. The formula of E is method-specific, and so will be defined later when discussing each algorithm. If $q < 1$, the step-size h is accepted. Otherwise, it is reduced and a new step is attempted. The new step size is proposed through proportional-integral control (PI-control) via $h_{\text{new}} = \eta q_{n-1}^{\lambda_2} q_n^{\lambda_1} h$, where λ_1, λ_2 are tuned parameters [31], q_{n-1} is the previous proportion error, and η is the safety factor.

Many variations of the Rosenbrock method can be derived by modifying above the formulas and coefficients. In this work, we consider three formulations of the Rosenbrock method, resulting in three stiff ODE solvers, namely `GPURosenbrock23`, `GPURodas4`, and `GPURodas5P`.

A. `GPURosenbrock23`: Second-order Method

`GPURosenbrock23` is a 2^{nd} order, L-stable Rosenbrock-W method and with 3^{rd} order accurate error control, which is suitable for stiff problems at high tolerances as well as problems with oscillations. The algorithm of `GPURosenbrock23` is as described in [29], where the procedure for advancing from (u_n, t_n) to (u_{n+1}, t_{n+1}) is

$$\begin{aligned} F_0 &= f(u_n, t_n), \\ k_1 &= W^{-1}(F_0 + hdf_t), \\ F_1 &= f(u_n + 0.5hk_1, t_n + 0.5h), \\ k_2 &= W^{-1}(F_1 - k_1) + k_1, \\ u_{n+1} &= u_n + hk_2. \end{aligned}$$

Here $W = I - hdf_u$, $d = \frac{1}{2+\sqrt{2}}$, and I is the identity matrix. For adaptive time-stepping, the error E can be calculated by

$$\begin{aligned} F_2 &= F(u_{n+1}, t_{n+1}), \\ k_3 &= W^{-1}(F_2 - e_{32}(k_2 - F_1) - 2(k_1 - F_0) + hdf_t), \\ E &= \frac{h}{6}(k_1 - 2k_2 + k_3), \end{aligned}$$

with $e_{32} = 6 + \sqrt{2}$.

Interpolation for the solution at $t_{n+\theta} = t_n + \theta h$, denoted as $u_{n+\theta}$, can be done by

$$u_{n+\theta} = u_n + h\left(\frac{\theta(1-\theta)}{1-2d}k_1 + \frac{\theta(\theta-2d)}{1-2d}k_2\right).$$

B. `GPURodas4`: Fourth-order Method

`GPURodas4` is based on a fourth-order A-stable stiffly stable Rosenbrock method. The integration scheme of `GPURodas4` is identical to that of `Rodas4` employed in DifferentialEquations.jl [30], where stepping from (u_n, t_n) to (u_{n+1}, t_{n+1}) follows

$$\begin{aligned} F_1 &= f(u_n, t_n), \\ k_1 &= -W^{-1}(F_1 + l_{d_1} f_t), \end{aligned}$$

$$
\begin{aligned}
F_2 &= f(u_n + a_{21}k_1, t_n + c_2h),\\
k_2 &= -W^{-1}(F_2 + l_{d_2}f_t + l_{C_{21}}k_1),\\
F_3 &= f(u_n + \sum_{i=1}^{2} a_{3i}k_i, t_n + c_3h),\\
k_3 &= -W^{-1}(F_3 + l_{d_3}f_t + \sum_{i=1}^{2} l_{C_{3i}}k_i),\\
F_4 &= f(u_n + \sum_{i=1}^{3} a_{4i}k_i, t_n + c_4h),\\
k_4 &= -W^{-1}(F_4 + l_{d_4}f_t + \sum_{i=1}^{3} l_{C_{4i}}k_i),\\
F_5 &= f(u_n + \sum_{i=1}^{4} a_{5i}k_i, t_n + h),\\
k_5 &= -W^{-1}(F_5 + \sum_{i=1}^{4} l_{C_{5i}}k_i),\\
F_6 &= f(u_n + \sum_{i=1}^{4} a_{5i}k_i + k_5, t_n + h),\\
k_6 &= -W^{-1}(F_6 + \sum_{i=1}^{5} l_{C_{6i}}k_i),\\
u_{n+1} &= u_n + \sum_{i=1}^{4} a_{5i}k_i + k_5 + k_6.
\end{aligned}
$$

Here $W = f_u - I/(h\gamma)$, I is the identity matrix, $l_{C_{ij}}$ is defined by

$$l_{C_{ij}} = \frac{C_{ij}}{h},$$

e.g., $l_{C_{21}} = C_{21}/h$, and l_{d_i} is defined by

$$l_{d_i} = hd_i,$$

e.g., $l_{d_1} = hd_1$.

For adaptive time-stepping, the error E is

$$E = k_6.$$

Interpolation for $u_{n+\theta}$ follows a stiff-aware third-order interpolant

$$u_{n+\theta} = (1-\theta)u_n + \theta(u_{n+1} + (1-\theta)(g_1 + \theta g_2)),$$

with

$$
\begin{aligned}
g_1 &= \sum_{i=1}^{5} h_{2i}k_i,\\
g_2 &= \sum_{i=1}^{5} h_{3i}k_i.
\end{aligned}
$$

A set of coefficients $a_{ij}, C_{ij}, c_i, d_i, h_{ij}, \gamma$ for `GPURodas4` is given in the appendix.

C. `GPURodas5`: Fifth-order Method

`GPURodas5P` is based on a fifth-order A-stable stiffly stable Rosenbrock method [32]. The integration scheme of `GPURodas5P` is identical to that of `Rodas5P` implemented in DifferentialEquations.jl [30], where stepping from (u_n, t_n) to (u_{n+1}, t_{n+1}) can be done by

$$
\begin{aligned}
F_1 &= f(u_n, t_n),\\
k_1 &= -W^{-1}(F_1 + l_{d_1}f_t),\\
F_2 &= f(u_n + a_{21}k_1, t_n + c_2h),\\
k_2 &= -W^{-1}(F_2 + l_{d_2}f_t + l_{C_{21}}k_1),\\
F_3 &= f(u_n + \sum_{i=1}^{2} a_{3i}k_i, t_n + c_3h),\\
k_3 &= -W^{-1}(F_3 + l_{d_3}f_t + \sum_{i=1}^{2} l_{C_{3i}}k_i),\\
F_4 &= f(u_n + \sum_{i=1}^{3} a_{4i}k_i, t_n + c_4h),\\
k_4 &= -W^{-1}(F_4 + l_{d_4}f_t + \sum_{i=1}^{3} l_{C_{4i}}k_i),\\
F_5 &= f(u_n + \sum_{i=1}^{4} a_{5i}k_i, t_n + c_5h),\\
k_5 &= -W^{-1}(F_5 + l_{d_5}f_t + \sum_{i=1}^{4} l_{C_{5i}}k_i),\\
F_6 &= f(u_n + \sum_{i=1}^{5} a_{6i}k_i, t_n + h),\\
k_6 &= -W^{-1}(F_6 + \sum_{i=1}^{5} l_{C_{6i}}k_i),\\
F_7 &= f(u_n + \sum_{i=1}^{5} a_{6i}k_i + k_6, t_n + h),\\
k_7 &= -W^{-1}(F_7 + \sum_{i=1}^{6} l_{C_{7i}}k_i),\\
F_8 &= f(u_n + \sum_{i=1}^{5} a_{6i}k_i + k_6 + k_7, t_n + h),\\
k_8 &= -W^{-1}(F_8 + \sum_{i=1}^{7} l_{C_{8i}}k_i),\\
u_{n+1} &= u_n + \sum_{i=1}^{5} a_{6i}k_i + \sum_{i=6}^{8} k_i.
\end{aligned}
$$

In this case, $W, I, l_{C_{ij}}, l_{d_i}$ are as defined for `GPURodas4`.

For adaptive time-stepping, the error E for step size can be calculated by

$$E = k_8.$$

Interpolation for $u_{n+\theta}$ follows a stiff-aware fourth-order interpolant

$$u_{n+\theta} = (1-\theta)u_n + \theta(u_{n+1} + (1-\theta)(g_1 + \theta(g_2 + \theta g_3))),$$

with

$$g_1 = \sum_{i=1}^{8} h_{2i}k_i,$$

$$g_2 = \sum_{i=1}^{8} h_{3i}k_i,$$

$$g_3 = \sum_{i=1}^{8} h_{4i}k_i.$$

A set of coefficients $a_{ij}, C_{ij}, c_i, d_i, h_{ij}, \gamma$ is given in the appendix.

The Rosenbrock-type methods are ideally suited for GPU compilation because they are devoid of typical Newton's method performed per step in stiff ODE integrators [27]. The Newton's method requires multiple linear solves and is computationally expensive due to repeated Jacobian calculation due to no reuse of previous matrix factorization. Rosenbrock methods only require one Jacobian evaluation and have 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 and perform multiple small tasks in parallel, the above argument prophesies Rosenbrock methods to achieve massive speed-ups on GPUs in ensemble simulations.

IV. The State Of GPU Parallelism In Scientific Computing

Graphics Processing Units (GPUs) were initially meant to use for tasks such as image processing or graphics rendering. However, in the past two decades, there has been a rise in General Purpose GPU (GPGPU) programming, which allows to retarget GPUs to applications in scientific computing & machine learning [33–35]. The biggest difference between CPUs and GPUs is that GPUs have massively parallel architecture, i.e., huge number cores called streaming multiprocessor and thousands of active threads. A simple model to distinguish between CPUs and GPUs is that CPUs have a fewer number of workers but with more computing power per worker, whereas GPUs have more workers with less power per worker. A common bottleneck in GPU parallelism is the latency of the load and store operations from the global memory to GPU processor cache. However, large array operations are effectively able to hide the memory latency to larger computation times, and hence large array operations are greatly parallelized and faster on GPUs.

GPUs offer a powerful array abstraction that makes it possible to write generic code. The abstractions are implemented by each backend, either using native kernels or reusing existing functionality. For performance, common operations like matrix-multiplication are implemented by dispatching to vendor-specific libraries like CUBLAS for NVIDIA GPUs and Metal's Performance Shaders for Apple GPUs. Higher-order operations like `map`, `broadcast` and `reduce` are implemented using native kernels. This makes it possible to compose them with code provided by the user, often obviating the need for custom kernels. Work by Besard et al. [36] 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.

V. Julia Ecosystem for Massively Data-Parallel GPU Solving of Independent ODEs

In Julia, there are two approaches to parallelize ensemble problems on GPUs. Both approaches automatically translate and compile the ODEs. The first approach, `EnsembleGPUArray`, 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, `EnsembleGPUKernel`, 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 1. Both of the programs are composable with Julia's SciML [8] ecosystem, where users can write models compatible with standard SciML tools like 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 to use of its libraries from other languages like R, allowing even the use of our GPU-accelerated solvers from other programming languages [1].

A. `EnsembleGPUArray`: Accelerating Ensemble ODEs with GPU Array Parallelism

1) 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, which will be required to keep track of. Subsequently, one can formulate this problem to solve this ODE at once:

$$\frac{dU}{dt} = F(U, P, t), \tag{9}$$

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

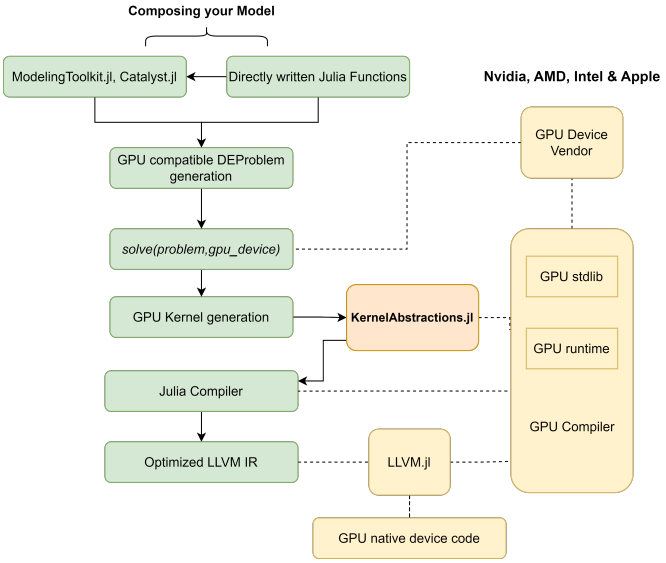

Fig. 1: Overview of the automated translating and solving of differential equations for GPUs for massively data-parallel problems. The solid lines indicate the code flow, whereas the dashed indicate the extension interactions.

where

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1N} \\ u_{21} & u_{22} & \dots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nN} \end{bmatrix}_{n \times N}, \tag{10}$$

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

$$F = \begin{bmatrix} f(u, p_{1:m,1}, t) & \dots & f(u, p_{1:m,N}, t) \end{bmatrix}_{n \times N}, \tag{12}$$

where $p_{1:m,j}$ 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 computation of the quantities which depend on U to happen in parallel. When solving ODEs, these quantities are generally the 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.

2) Translating ODE solves over GPU using KernelAbstractions.jl: The GPU kernels are written using KernelAbstractions.jl [37], this 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.

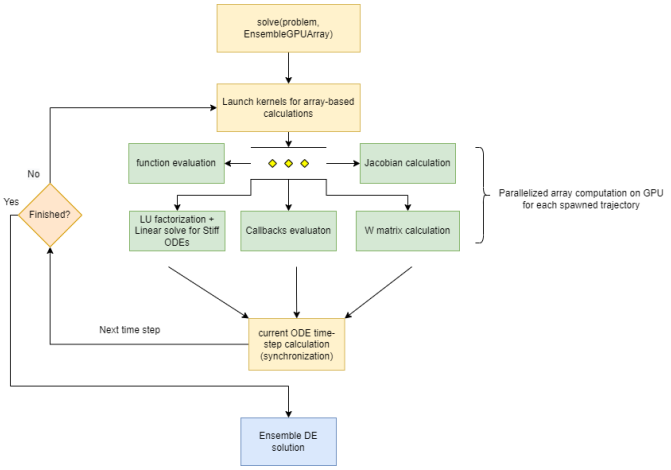

Fig. 2: The EnsembleGPUArray flowchart

After calculating the dependents on U, 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 [8]. 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[2]. Figure 2 summarizes an overview of the process.

3) 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 matrix of the batched ODE problem in Section V-A1 has a block diagonal structure:

$$W = \begin{bmatrix} -\gamma I + J_1 & & & \\ & -\gamma I + J_2 & & \\ & & \ddots & \\ & & & -\gamma I + J_N \end{bmatrix}, \tag{13}$$

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

4) Drawbacks of the Array Ensemble Approach: The main drawback of this approach is that each array operation inside

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

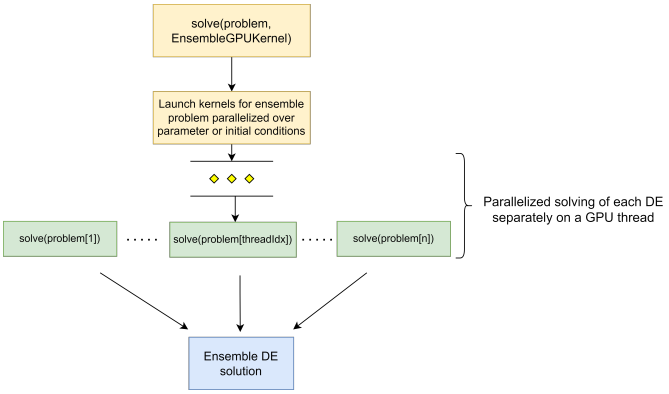

Fig. 3: The EnsembleGPUKernel flowchart

of the ODE solver requires a separate GPU kernel launch. 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 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 reduce the overall cost [38], we will see in the later benchmarks (Section VI-B) that each of the array-ensemble GPU ODE solvers have 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.

B. `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 store 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. Each thread accesses the data-augmented ODE to analyse, and the solving of all the ODEs is completely asynchronous. The process is briefly outlined in Figure 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 as Julia's CUDA.jl does not support dynamic memory allocation on

```
@kernel function Rodas5P_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

the GPU. However, solving ODEs requires storing intermediate computations, normally using array allocations. The vast features of DifferentialEquations.jl rely on operations like broadcast operations, dynamic allocations, and dynamic function invocation, etc., 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.

VI. Benchmarks and Case Studies

A. Setup

To compare different available open-source programs with GPU-accelerated ODE solvers, we benchmarked them with NVIDIA Tesla V100, a typical compute node GPU. We chose a typical desktop GPU setup to benchmark the ODE solvers with different vendors; Quadro RTX 5000 (11.15 TFLOPS) for NVDIA, Vega 64 (10.54 TFLOPS) for AMD, A770 (19.66 TFLOPS) for Intel, and M1 Max (10.4 TFLOPS) for Apple. The ODE problems involved single precision (Float32) on GPUs. The CPU benchmarks were executed with double precision (Float64) and timed on an Intel Xeon Gold 6248 processor running at 2.50GHz with 16 threads enabled. Using double precision on CPUs is faster for our use case and processor than using single precision.

For timing, we measured only the times used for solving the ensemble ODE. The other latencies in running the solvers might include compilation time, which is only reported during the first run of the function, due to the Just-in-Time (JIT) compilation. The timings for the programs written in Julia can

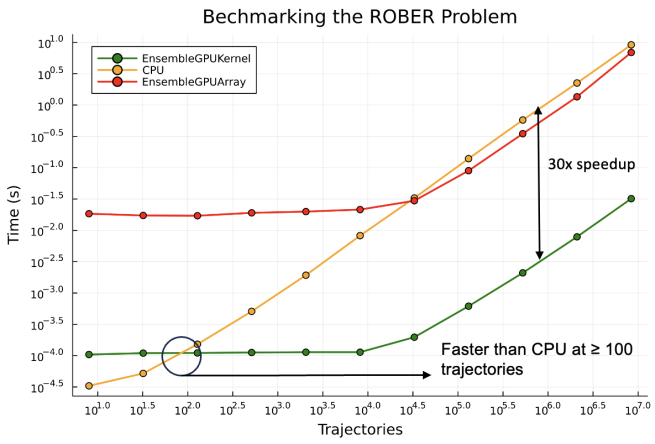

Fig. 4: A comparison of `EnsembleGPUKernel` with Julia's `EnsembleGPUArray` and CPU parallelism. The stiff ODE solvers demonstrates efficient scaling with trajectory count and are on average 30× faster than the CPU implementation.

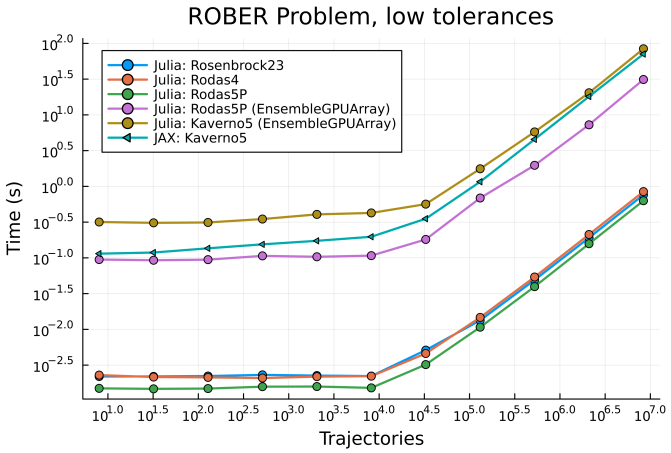

Fig. 5: A comparison of ODE solve timings for with other programs with adaptive time-stepping. `EnsembleGPUKernel` is faster by 20–50× in comparison with JAX and `EnsembleGPUArray`.

be measured using BenchmarkTools.jl [39], taking the fastest run. The Julia-based benchmarks were tested on DiffEqGPU.jl 2.2.1, 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 programs were run at least ten times for programs based on JAX. JAX 0.4.1 with Diffrax 0.2.2 and Python 3.9 forms our software stack for benchmarking.

B. Comparison between CPU and GPU Parallelism

To establish the efficiency of GPU-based parallelism, we compared the solvers with CPU multi-threading. Ensuring a fair comparison between CPUs and GPUs was done by choosing a setup typical in a cluster compute node as described in SectionVI-A. Figure 4 demonstrates the simulation times as a function of the number of trajectories. The `EnsembleGPUKernel` based ODE solvers supersede CPU parallelism at approximately 100 trajectories and demonstrates an average speed-up of 30×. The solver used to benchmark the Roberston Equation [40] is the `Rosenbrock23` method (Section III-A) with adaptive time-stepping. This demonstrates the viability of GPU parallelism in ensemble simulations where traditional CPU parallelism becomes a sub-optimal choice for massively parallel simulations.

C. Comparison between Different GPU-based Parallelism

The idea of GPU-based parallelism for ensemble simulations is not new, but was restricted to relatively lower-level implementations for achieving optimal performance. The barrier in adoption in high-level languages for our application is the $f(u, p, t)$ definition, which consists of high-level definitions often difficult or even incompatible to compile on GPUs. Benchmarking against existing implementations ensures accessibility with high performance. Here, we compare our solvers with existing Julia's `EnsembleGPUArray` and JAX's Diffrax ODE solver libraries. We run the benchmarks on Roberston's equation B, a three-dimensional stiff ODE. Figure 5 shows that `EnsembleGPUKernel` outperforms both JAX and Julia's `EnsembleGPUArray` on average by 20–50×. It is important to note that this performance difference is owing to fundamentally different parallelism methods (i.e., array-based vs. special kernel generation) rather than the efficiency of different libraries.

D. High Performance with Vendor Agnosticism: Comparison between Different GPU Vendors

With vendor agnostic GPU kernel generation, researchers can choose major GPU backends with ease. Our benchmarks in Figure 6 show that, at a sufficiently high number of trajectories, the best performance can be achieved by NVIDIA and Intel, closely followed by AMD and Apple. This is a concrete proof that there is minimal overhead in our ODE solvers, and so users can expect performance one-to-one with the Floating Point Operations per Second (FLOPS) in GPUs mentioned in Section VI-A. Note that we simulated the Lorenz problem [41] using `GPURosenbrock23` with fix time-stepping to allude thread/warp divergence.

VII. Conclusion

We have demonstrated GPU-based acceleration of ODEs for ensemble simulations. The methods achieve significant speed-ups of 20–50× compared to the existing solvers. Moreover, we conclude that performance difference is not due to a more optimized implementation of array based parallelism, but inherently due to a different parallelism strategy which results in low overheads and sans implicit synchronization of time-stepping. For achieving peak performance in HPC,

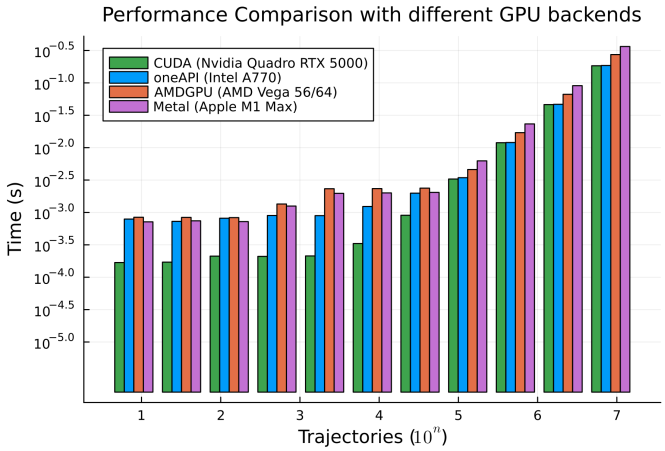

Fig. 6: Demonstration of the vendor agnosticism by measuring run-times on different GPU vendors. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

it is suggested to leave the constraint array-based parallelism approach imposed by common DSLs.

Apart from being a performant alternative to the `EnsembleGPUArray` approach, there are opportunities for further improvements with `EnsembleGPUKernel`. The linear algebra routines such as matrix factorization and linear solve for static allocated arrays need to be rewritten for high-dimensional ODEs specifically for GPUs. This is primarily because the current methods rely on constructs such as heap allocations, which are forbidden in GPU compilation. Flexibility in terms of supporting mutation within the ODE function can be extended as well. This could be achieved by using mutable static arrays, which requires special techniques to compile with the GPU kernels. The user is also limited in terms of using features such as broadcast and calls to BLAS. 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, which are expected to resolve this issue.

Availablity of Code

The code is available at https://github.com/SciML/DiffEqGPU.jl with benchmarking scripts available at https://github.com/utkarsh530/GPUODEBenchmarks/tree/u/stiff.
The code was contributed through these Pull Requests (PRs) to the repository:
https://github.com/SciML/DiffEqGPU.jl/pull/275,
https://github.com/SciML/DiffEqGPU.jl/pull/274,
https://github.com/SciML/DiffEqGPU.jl/pull/272,
https://github.com/SciML/DiffEqGPU.jl/pull/252.

Acknowledgments

We would like to acknowledge Dr Chris Rackauckas with his guidance with numerical methods for SciML, particularly the discussions about optimal choice for ODE solvers on GPUs.

References

[1] A. Tarantola, *Inverse problem theory and methods for model parameter estimation*. Philadelphia: SIAM, 2005.
[2] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American statistical association*, vol. 44, no. 247, pp. 335–341, 1949.
[3] 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.
[4] 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, pp. 1–18, 2009.
[5] B. Iooss and P. Lemaître, "A review on global sensitivity analysis methods," *Uncertainty management in simulation-optimization of complex systems: algorithms and applications*, vol. 1, pp. 101–122, 2015.
[6] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," *SIAM journal on scientific computing*, vol. 18, no. 1, pp. 1–22, 1997.
[7] 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.
[8] 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.
[9] 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. 105732, 2021.
[10] X. Li, T.-K. L. Wong, R. T. 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.
[11] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," Github, 2018. [Online]. Available: http://github.com/google/jax
[12] Y. Zhou, J. Liepe, X. Sheng, M. P. Stumpf, and C. Barnes, "Gpu accelerated biochemical network simulation," *Bioinformatics*, vol. 27, no. 6, pp. 874–876, 2011.

[13] 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 *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society. Dallas, Texas: IEEE, 2022, pp. 1078–1092.

[14] 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. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476219

[15] 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.

[16] 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*, vol. 8403, SPIE. Baltimore, Maryland, United States: SPIE, 2012, pp. 49–56.

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

[18] D. Demidov, "Vexcl: Vector expression template library for OpenCL," 2012.

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

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

[21] D. Nagy, L. Plavecz, and F. Hegedűs, "Solving large number of non-stiff, low-dimensional ordinary differential equation systems on gpus and cpus: performance comparisons of mpgos, odeint and differentialequations. jl," *arXiv preprint arXiv:2011.01740*, vol. 1, 2020.

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

[23] A. C. Hindmarsh, "Odepack, a systemized collection of ode solvers," *Scientific computing*, vol. 1, 1983.

[24] 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.

[25] 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.

[26] G. D. Byrne and A. C. Hindmarsh, "Stiff ode solvers: A review of current and coming attractions," *Journal of Computational physics*, vol. 70, no. 1, pp. 1–62, 1987.

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

[28] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. Berlin, Heidelberg: Springer, 1996.

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

[30] 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.

[31] E. Hairer, S. P. Norsett, and G. Wanner, "Solving ordinary differential equations i: Nonsti problems, volume second revised edition," 1993.

[32] 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.

[33] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.

[34] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 3–3.

[35] D. Steinkraus, I. Buck, and P. Simard, "Using gpus for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, 2005, pp. 1115–1120 Vol. 2.

[36] 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.

[37] V. Churavy, D. Aluthge, J. Samaroo, A. Smirnov, J. Schloss, L. C. Wilcox, S. Byrne, M. Waruszewski, A. Ramadhan, Meredith, S. Schaub, N. C. Constantinou, J. Bolewski, M. Ng, T. Besard, B. Arthur, C. Kawczynski, C. Hill, C. Rackauckas, J. Cook, J. Liu, M. Schanen, O. Schulz, Oscar, P. Haraldsson, T. Arakaki, and T. Chor, "Juliagpu/kernelabstractions.jl: v0.9.1," JuliaGPU, Mar. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7770454

[38] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Confer-*

ence on Cyber, Physical and Social Computing, IEEE. Hangzhou, China: IEEE, 2010, pp. 344–350.

[39] J. Chen and J. Revels, "Robust benchmarking in noisy environments," *arXiv e-prints*, Aug 2016.

[40] 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.

[41] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of atmospheric sciences*, vol. 20, no. 2, pp. 130–141, 1963.

APPENDIX

A. Lorenz Problem

The first test problem is Lorenz attractor:

$$\frac{dy_1}{dt} = \sigma(y_2 - y_1), \tag{14}$$

$$\frac{dy_2}{dt} = \rho y_1 - y_2 - y_1 y_3, \tag{15}$$

$$\frac{dy_3}{dt} = y_1 y_2 - \gamma y_3. \tag{16}$$

It consists of three parameters σ, ρ, γ, where $\sigma = 10.0$ & $\gamma = \frac{8}{3}$ and $\rho = 21.0$. The integration is performed from $t = (0.0, 1.0)$ with the time-step $h = 0.001$, essentially generating 1000 fixed time-steps. The initial conditions are $y = [1.0, 0.0, 0.0]$

B. ROBER Problem

The second test problem is the Robertson Equation:

$$\frac{dy_1}{dt} = -0.04y_1 + 10^4 y_2 y_3, \tag{17}$$

$$\frac{dy_2}{dt} = 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2, \tag{18}$$

$$\frac{dy_3}{dt} = 3 \times 10^7 y_2^2. \tag{19}$$

The integration is performed from $t = (0.0, 10^5)$ with the initial time-step of $h = 0.0001$. The initial conditions are $y = [1.0, 0.0, 0.0]$.

C. Coefficients for Rodas4 and Rodas5P

TABLE I: Coefficients for the Rodas4 algorithm.

Coefficient	Value
γ	0.25
a_{21}	1.544000000000000
a_{31}	0.9466785280815826
a_{32}	0.2557011698983284
a_{41}	3.314825187068521
a_{42}	2.896124015972201
a_{43}	0.9986419139977817
a_{51}	1.221224509226641
a_{52}	6.019134481288629
a_{53}	12.53708332932087
a_{54}	−0.6878860361058950
C_{21}	−5.668800000000000
C_{31}	−2.430093356833875
C_{32}	−0.2063599157091915
C_{41}	−0.1073529058151375
C_{42}	−9.594562251023355
C_{43}	−20.47028614809616
C_{51}	−7.496443313967647
C_{52}	−10.24680431464352
C_{53}	−33.99990352819905
C_{54}	11.70890893206160
C_{61}	8.083246795921522
C_{62}	−7.981132988064893
C_{63}	−31.52159432874371
C_{64}	16.31930543123136
C_{65}	−6.058818238834054
c_2	0.386
c_3	0.21
c_4	0.63
d_1	0.2500000000000000
d_2	−0.1043000000000000
d_3	0.1035000000000000
d_4	−0.0362000000000023
h_{21}	−10.12623508344586
h_{22}	−7.487995877610167
h_{23}	−34.80091861555747
h_{24}	−7.992771707568823
h_{25}	1.025137723295662
h_{31}	−0.6762803392801253
h_{32}	6.087714651680015
h_{33}	16.43084320892478
h_{34}	24.76722511418386
h_{35}	−6.594389125716872

TABLE II: Coefficients for the Rodas5P algorithm.

Coefficient	Value
γ	0.21193756319429014
a_{21}	3.0
a_{31}	2.849394379747939
a_{32}	0.45842242204463923
a_{41}	−6.954028509809101
a_{42}	2.489845061869568
a_{43}	−10.358996098473584
a_{51}	2.8029986275628964
a_{52}	0.5072464736228206
a_{53}	−0.3988312541770524
a_{54}	−0.04721187230404641
a_{61}	−7.502846399306121
a_{62}	2.561846144803919
a_{63}	−11.627539656261098
a_{64}	−0.18268767659942256
a_{65}	0.030198172008377946
C_{21}	−14.155112264123755
C_{31}	−17.97296035885952
C_{32}	−2.859693295451294
C_{41}	147.12150275711716
C_{42}	−1.41221402718213
C_{43}	71.68940251302358
C_{51}	165.43517024871676
C_{52}	−0.4592823456491126
C_{53}	42.90938336958603
C_{54}	−5.961986721573306
C_{61}	24.854864614690072
C_{62}	−3.0009227002832186
C_{63}	47.4931110020768
C_{64}	5.5814197821558125
C_{65}	−0.6610691825249471
C_{71}	30.91273214028599
C_{72}	−3.1208243349937974
C_{73}	77.79954646070892
C_{74}	34.28646028294783
C_{75}	−19.097331116725623
C_{76}	−28.087943162872662
C_{81}	37.80277123390563
C_{82}	−3.2571969029072276
C_{83}	112.26918849496327
C_{84}	66.9347231244047
C_{85}	−40.06618937091002
C_{86}	−54.66780262877968
C_{87}	−9.48861652309627
c_2	0.6358126895828704
c_3	0.4095798393397535
c_4	0.9769306725060716
c_5	0.4288403609558664
d_1	0.21193756319429014
d_2	−0.42387512638858027
d_3	−0.3384627126235924
d_4	1.8046452872882734
d_5	2.325825639765069

Coefficients for the Rodas5P algorithm (continued).

Coefficient	Value
h_{21}	25.948786856663858
h_{22}	−2.5579724845846235
h_{23}	10.433815404888879
h_{24}	−2.3679251022685204
h_{25}	0.524948541321073
h_{26}	1.1241088310450404
h_{27}	0.4272876194431874
h_{28}	−0.17202221070155493
h_{31}	−9.91568850695171
h_{32}	−0.9689944594115154
h_{33}	3.0438037242978453
h_{34}	−24.495224566215796
h_{35}	20.176138334709044
h_{36}	15.98066361424651
h_{37}	−6.789040303419874
h_{38}	−6.710236069923372
h_{41}	11.419903575922262
h_{42}	2.8879645146136994
h_{43}	72.92137995996029
h_{44}	80.12511834622643
h_{45}	−52.072871366152654
h_{46}	−59.78993625266729
h_{47}	−0.15582684282751913
h_{48}	4.883087185713722