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

Utkarsh
MIT CSAIL
Cambridge, Massachusetts, USA
utkarsh5@mit.edu

Valentin Churavy
MIT CSAIL
Cambridge, Massachusetts, USA
vchuravy@mit.edu

Yingbo Ma
JuliaHub
Cambridge, Massachusetts, USA
yingbo.ma@juliahub.com

Tim Besard
JuliaHub
Ghent, Belgium
tim@juliahub.com

Tim Gymnich
MIT CSAIL
Technical University of Munich
Cambridge, Massachusetts, USA
tgymnich@mit.edu

Adam R. Gerlach
United States Air Force Research Laboratory
Wright-Patterson AFB, OH, USA
adam.gerlach.1@us.af.mil

Alan Edelman
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
edelman@math.mit.edu

Christopher Rackauckas
JuliaHub, Pumas-AI
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
crackauc@mit.edu

ABSTRACT

We demonstrate a high-performance vendor-agnostic method for massively parallel solving of ensembles of ordinary differential equations (ODEs) and stochastic differential equations (SDEs) 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 compared to hand-optimized CUDA-C++ kernels, while performing $20-100\times$ faster than the vectorized-map (`vmap`) approach implemented in JAX and PyTorch. Performance evaluation on NVIDIA, AMD, Intel, and Apple GPUs demonstrates performance portability and vendor-agnosticism. We show composability with MPI to enable distributed multi-GPU workflows. The implemented solvers are fully featured, supporting 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.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Ordinary differential equations**; • **Computing methodologies** → **Massively parallel algorithms**; **Modeling and simulation**; • **Applied computing** → *Physical sciences and engineering*.

KEYWORDS

Differential Equations, Numerical Simulation, GPU, Data-parallelism, Computer Kernel, HPC

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 [54], performing uncertainty quantification [32, 40, 42], and calculating global sensitivity analysis [29, 40]. While such an embarrassingly parallel problem lends itself to being well-suited for acceleration via GPU hardware, the programming requirements has 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 [51], Python (SciPy [59]), or Julia (DifferentialEquations.jl [47]) to reduce the barrier to entry for scientists and engineers. In order to target GPUs, previous software such as MPGOS [25] 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 [36] or JAX [13]. 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. Our software transforms code which targets a widely used differential equation solver library in a high level language (Julia's DifferentialEquations.jl [47]) 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 $20-100\times$ 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 incorporating of datasets via the GPU's texture memory. Together, this software allows scientists to target all major GPU platforms without loss of performance.

2 RELATED WORK

While researchers have used GPUs to accelerate computations extensively in applications including molecular simulation, biological systems, and physics [20, 35, 63, 64], 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 a general ODE solver software, previous attempts have generally targeted hardware using array abstraction frameworks like ArrayFire [39], Thrust [5] and VexCL [18], JAX [13], and PyTorch [36]. These frameworks allow one to adapt code written on high-level array abstractions and generates a highly optimized code to backends like OpenCL [53], CUDA [44]. Boost's ODEINT [1, 2, 43] allows user to use ODE solvers which without any modification works with GPU backends like CUDA and OpenCL. JAX's Diffrax [30] generates solvers for ensembles of ODEs on GPUs via JAX's vectorized map functionality (vmap). PyTorch's torchdiffeq [16] allows for defining ensembles of ODEs directly with GPU-based arrays, although their vmap provided by functorch support with ODEs is still primitive as of April 2023.

However, recent results have demonstrated that using array-based abstractions for generating GPU-parallel ODE ensemble solvers greatly lags in performance against the state of the art. In particular, MPGOS [25] demonstrated that ODEINT was 10x-100x 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 [64], a CUDA translation of the widely used LSODE solver [27, 28], which was similarly limited due to requiring ODE models to be written in CUDA and compiled into the kernels.

3 NUMERICAL METHODS FOR DIFFERENTIAL EQUATIONS

3.1 Ordinary Differential Equations (ODEs)

ODEs are models given by an evolution equation:

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

with an initial condition $u(t_0) = u_0$ on a fixed time span (t_0, t_f). There exist many different methods for numerically solving ODEs [23, 24], 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 [26, 52]. One of the most common classes of solvers for ODE software are explicit Runge-Kutta methods [33, 49]. These methods are specified by a coefficient tableau $\{A, b, c\}$, with "s" stages and "k" order, where $k \leq s$. It produces the approximation for $u(t+h)$ as:

$$k_s = f\left(u(t) + \sum_{i=1}^{s} a_{s,i} k_i, p, t + c_s h\right) \tag{2}$$

$$u(t+h) = u(t) + h \sum_{i=1}^{s} b_i k_i \tag{3}$$

Some of the examples of Runge-Kutta methods include dopri5 [19] and MATLAB's ODE suite ode45 [51].

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 $E = \|\tilde{u}(t+h) - u(t+h)\|$ [3, 24]. Adaptivity ensures that error remains below certain tolerance and these tolerances are absolute (atol) and relative (rltol). Mathematically, the proportion of error against tolerance is:

$$q = \left\| \frac{E}{atol + max(|u(t)|, |u(t+h)|) \cdot rtol} \right\|. \tag{4}$$

$q < 1$ means the step-size h is accepted, otherwise it 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 $h_{new} = \eta q_{n-1}^{\beta_2} q_n^{\beta_1} h$, where β_1, β_2 are tuned parameters [24], q_{n-1} is the previous proportion error and η is the safety factor.

3.2 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 [12, 41], systems biology [62], and simulation chemical reaction networks [22]. SDEs are formally defined as:

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t, \tag{5}$$

where W_t is a wiener process and dW_t is a Gaussian random variable $W_{t+dt} - W_t \sim \mathcal{N}(0, dt)$ [31]. We note that generally one is interested in ensembles of SDE solutions for the purpose of calculating 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:

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

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

where $\alpha_j, \beta_j^k, \mu_j, \lambda_{ij}, \gamma_{ij}^k$ are the constants which define the particular stochastic Runge-Kutta method and R is the fit term [31, 55]. Adaptive time-stepping techniques using similar rejection sampling and PI-controller approaches to ODEs have been adapted to SDE solver software [48].

4 THE GPU ECOSYSTEM IN JULIA AND CROSS PLATFORM GPGPU PROGRAMMING

Using high-level languages to program hardware accelerators traditionally means to use either use 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 [39] and CuNumpy [46]. 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, like JAX [13], embed a new language into the host language that provide domain specific concepts and limits expressability 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 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 [34] and JuliaGPU [10] retarget a subset of the language for execution on hardware accelerators. In contrast to Numba, which is a re-implementation 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 to provide the basis for 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 [9] provided by Julia.

4.1 Supporting multiple GPU platforms

Originally JuliaGPU only supported hardware accelerators by NVIDIA (CUDA). As hypothesized [10] the same approach could be extended to target other hardware platforms. Instead of reimplementing a full-fledged compiler for each new platform, the common infrastructure pieces were abstracted into a single unified compiler interface GPUCompiler.jl [7] and a unified array interface GPUArrays.jl [6]. 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 [10] for NVIDIA GPUs using the CUDA toolkit, AMDGPU.jl [50] for AMD GPUs through ROCm, oneAPI.jl[8] for Intel GPUs with oneAPI and Metal.jl [11] 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 note yet supported [1]. Similarly, 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 isn't able to leverage different GPU vendors where that could have been beneficial for, e.g., HPC and AI/ML workloads [14, 45].

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 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 user code, often obviating the need for custom kernels. Work by Besard et al. [9] 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.

4.2 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 use 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 device backend without modifications of the higher-level code, all without sacrificing performance. In Julia, this is possible with the KernelAbstractions.jl [17] 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 paper KernelAbstractions.jl is used to target these different backends, essentially from the same high-level kernel code.

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

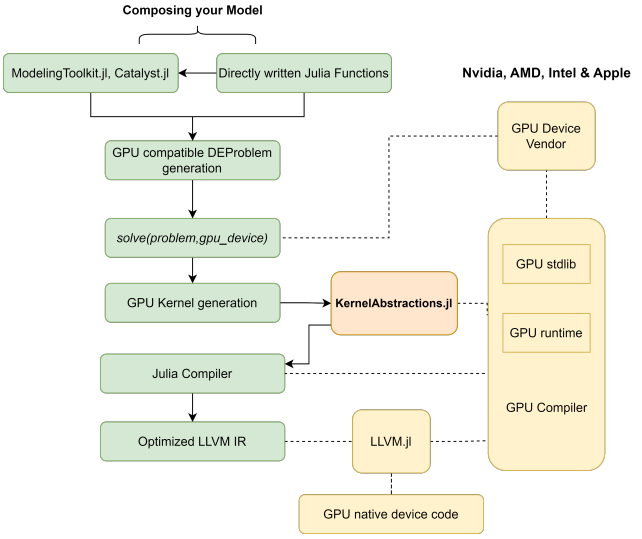

Figure 1: Overview of the automated translating and solving of differential equations for GPUs for massively data-parallel problems. The solid lines indicate the code flow, whereas the dashed indicate the extension interactions.

5 MASSIVELY DATA-PARALLEL GPU SOLVING OF INDEPENDENT ODE SYSTEMS

In this paper, we discuss two approaches to parallelize 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 1. Both of the programs are composable with Julia's SciML [47] 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 [4].

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

5.1.1 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:

$$\frac{dU}{dt} = F(U, P, t), \tag{6}$$

where:

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1N} \\ u_{21} & u_{22} & \dots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nN} \end{bmatrix}_{n \times N} \tag{7}$$

,

$$P = \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1N} \\ p_{21} & p_{22} & \dots & p_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \dots & p_{mN} \end{bmatrix}_{m \times N} \tag{8}$$

$$F = \begin{bmatrix} f(u, p_{1:m,1}, t) & \dots & f(u, p_{1:m,N}, t) \end{bmatrix}_{n \times N} \tag{9}$$

. 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.

5.1.2 Translating ODE solves over GPU using KernelAbstractions.jl. The GPU kernels are written using KernelAbstractions.jl [17], 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.

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 [47]. 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 2 summarizes an overview of the process.

5.1.3 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 5.1.1 has a block diagonal structure:

$$W = \begin{bmatrix} -\gamma I + J_1 & & & \\ & -\gamma I + J_2 & & \\ & & \ddots & \\ & & & -\gamma I + J_N \end{bmatrix}, \tag{10}$$

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

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

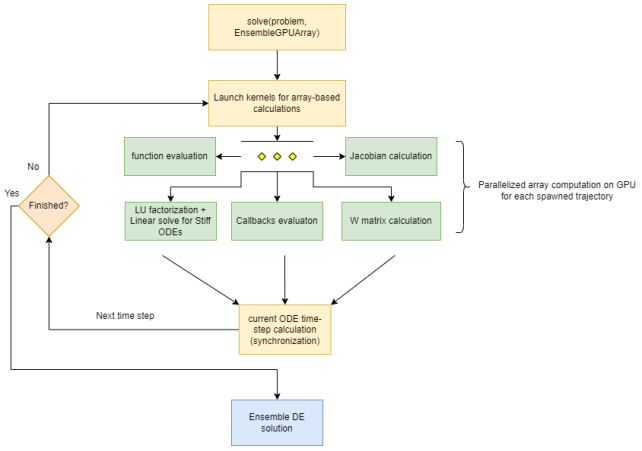

Figure 2: The EnsembleGPUArray flowchart

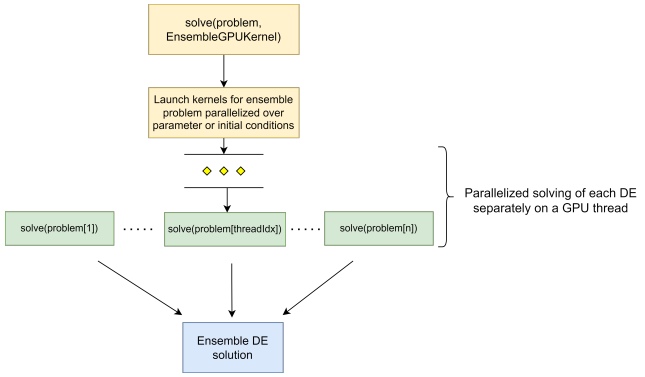

Figure 3: The EnsembleGPUKernel flowchart

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.

5.1.4 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 3.1 and 3.2, most of the operations are linear combinations and column-wise parallel application of the ODE model f, and are thus $O(N)$ operations. Array-based GPU DSLs are typically designed to be used with $O(N^3)$ 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 [61], we will see in the later benchmarks (section 6.2) that each of the array-ensemble GPU ODE solvers have a high fixed cost due to the total overhead of kernel launching.

In additional, 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.

5.2 `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 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.

```
@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

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 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.

5.2.1 Kernel-Specialized ODE Solvers. The following ODE solvers are currently available with `EnsembleGPUKernel`:

- `GPUTsit5`: A custom GPU-kernelized implementation of Tsitouras' 5^{th} order ODE solver, a Runge-Kutta 5(4) order method [57]. Serves 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) [24] Runge-Kutta pair which is a common default solver choice in packages like MATLAB [51], torchdiffeq [16] etc. It has free 4^{th} order interpolation support.
- `GPUVern7`: A custom GPU-kernelized implementation of Verner's 7^{th} order, a Runge-Kutta 7(6) order method [58]. Performs best at medium and low tolerances. Has a $7th$ order lazy interpolation scheme.
- `GPUVern9`: A custom GPU-kernelized implementation of Verner's 9^{th} order, a Runge-Kutta 9(8) order pair [58]. Performs best at extremely low tolerances. Has a $9th$ order lazy interpolation scheme.

These choices are based on the SciMLBenchmarks which has extensive comparisons between ODE solvers [6].

5.2.2 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 [31]. Supports diagonal and non-diagonal noise.
- `GPUSIEA`: A custom GPU-kernelized fixed time-step implementation of weak order 2.0 [56], stochastic generalization of midpoint method. Supports only diagonal noise.

6 BENCHMARKS AND CASE STUDIES

6.1 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]

[6] https://docs.sciml.ai/SciMLBenchmarksOutput/stable/
[7] https://docs.sciml.ai/DiffEqGPU/dev/tutorials/lower_level_api/

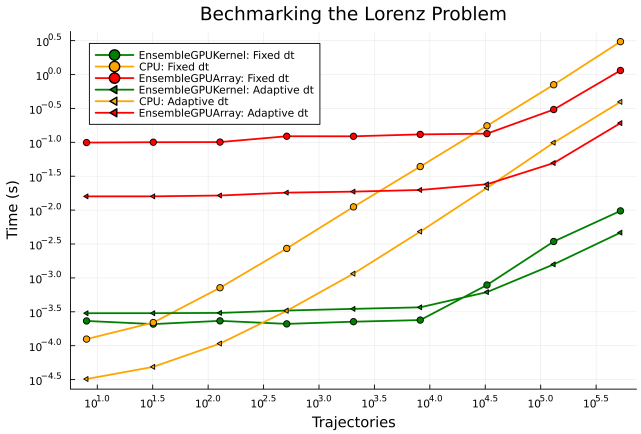

Figure 4: A comparison of an ODE solve timings with CPU vs GPU. The EnsembleGPUKernel performs the best with up to 100× acceleration and lower cutoff for parallelism to take advantage.

that closely resembles other APIs. The timings for each software only report time spent solving the ensemble ODE. 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. Both programs are tested on Python 3.9. The link for the complete benchmark suite is removed currently due to the double-blind review policy.

6.2 Establishing efficiency of solving ODE ensembles with GPU over CPU

Prior researches [25, 43] demonstrates 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 multi-threading with 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. It can be inferred from the figure 4, at approximately 100 − 1000 trajectories, GPU parallelism becomes superior to CPU parallelism. The solver used to benchmark the Lorenz Attractor [38] is Tsitouras' 5(4) Runge-Kutta method [57], both with adaptive and fixed time-stepping. The table 1 lists the relative speedup obtained using `EnsembleGPUKernel`.

[8] https://github.com/nnagyd/ode_solver_tests

Table 1: Summary of mean speed-ups of ODE integrators with different hardware *(lower the better)*

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

6.3 Solving one billion ODEs together: Scaling `EnsembleGPUKernel` with MPI

The Message Passing Interface (MPI) [21, 60] can be directly used with Julia using MPI.jl [15]. Additionally, there exists support for CUDA by using a CUDA-aware MPI backend. As a testimonial of scalability, 1, 073, 741, 824 (1 billion) ODE solutions on the Lorenz problem were calculated using 5 V100 GPU cluster nodes, in which four GPUs explicitly performed the computation of approximately 250 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 250 million trajectories was around 1.6 seconds. To reduce these latencies, one can use GPU with larger memory or multi-GPU per node which was not done in our demonstration due to hardware availability.

6.4 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 these libraries. Choosing systems with low-dimensional ODEs, such as the Lorenz equation, is a suitable candidate owing to the simplicity $f(u, p, t)$ 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, the $\sigma = 10.0 \quad \& \quad \gamma = \frac{8}{3}$ and ρ in equation [38] is uniformly varied from $(0.0, 21.0)$ generating N instances of independent, parallel ODE solves, which is also the no. of trajectories in DiffEqGPU.jl API.

The results for fixed and adaptive time-stepping are shown in separate figures 5 and 6 for equitable comparison. There is not any common ODE solver between these packages, so we use methods belonging to the class of $4^{th} - 5^{th}$ order Runge-Kutta methods, which perform similarly in the benchmarks [47]. "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 it 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

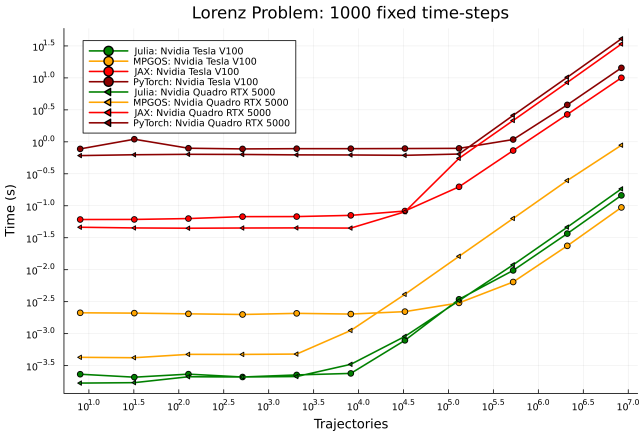

Figure 5: A comparison of ODE solve timings with other programs with fixed time-stepping. EnsembleGPUKernel is able to reach and sometimes outperform speed of light measure (MPGOS) and approximately faster by $20 - 100\times$ in comparison to JAX and $100 - 200\times$ for PyTorch.

Table 2: A summary of the range of slowdowns of the benchmarks in figures 5 and 6 *(lower the 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×	—

different NVIDIA GPUs is listed in tables 2 and 3. While our solvers able to reach and sometimes outperform speed of light measure (C++, MPGOS), we observe a greater acceleration of approximately $20 - 100\times$ in comparison to JAX and $100 - 200\times$ for PyTorch, both of which rely on vectorized map style of parallelism. The authors weren't able to compile adaptive time-stepping results for PyTorch as `vmap` currently does not support all of 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 implementation of the approach.

6.5 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 7 demonstrates that there's minimal overhead in our ODE solvers and

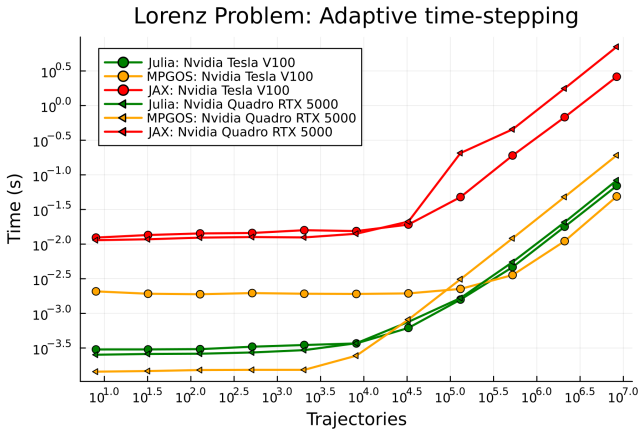

Figure 6: A comparison of ODE solve timings for with other programs with adaptive time-stepping. EnsembleGPUKernel is able to reach and sometimes outperform speed of light measure (MPGOS) and approximately faster by $20 - 100\times$ in comparison to JAX.

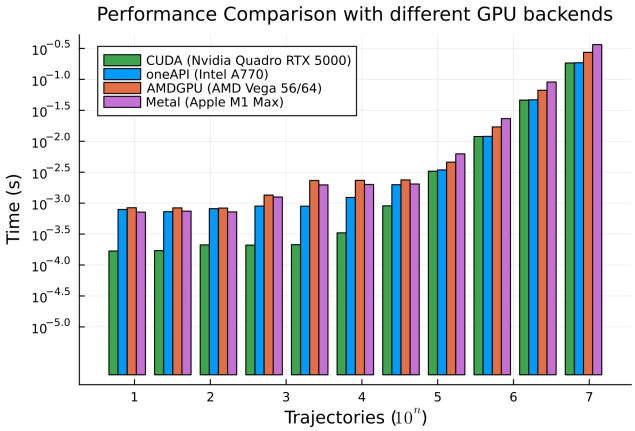

Figure 7: A comparison of ODE solve timings with fixed time-stepping, measured on different GPU platforms. We measure the time *(lower the better)* versus number of parallel solves. Here, the NVIDIA GPUs perform the best owing to the most-optimized library and matured ecosystem with JuliaGPU.

Table 3: A summary of the range of slowdowns of the benchmarks in figures 5 and 6 *(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×	—

users can expect performance one-to-one with mentioned Floating Point Operations per Second (FLOPS) in GPUs mentioned in the section 6.1. To allude other performance impacts like thread-divergence and equitable selection of the dimension of the ODE, we simulate the Lorenz problem with fix time-stepping. We run the benchmarks on major vendors: NVIDIA, AMD, Intel & Apple. The peakflops are listed in section 6.1. To our knowledge, this is the first showcase of GPU-parallel software for solving DEs that supports more than NVIDIA GPUs.

6.6 Event handling and automatic differentiation

Software for simulating dynamical systems allows for injecting discontinuous events and termination of integration with a specified criteria causing discontinuities within the integration. Event handling in differential equations is used 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

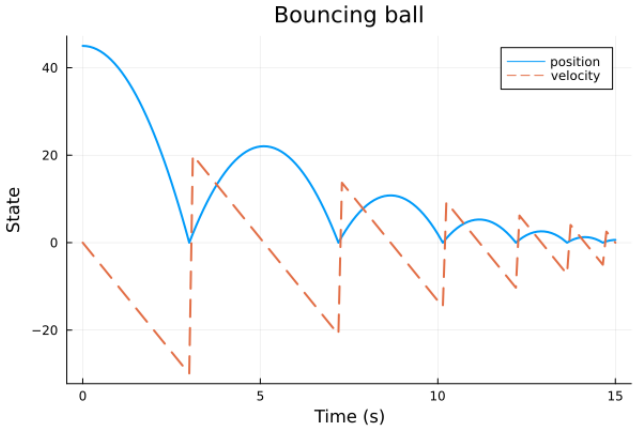

Figure 8: A simulation of bouncing ball problem on GPU. The blue trajectory is the displacement and red trajectory is velocity, across time. This demonstrates the ability to inject code within ODEs via callbacks.

a demonstration of event capabilities with `EnsembleGPUArray` and `EnsembleGPUKernel`, we demonstrate the famous surface-ball collision (bouncing ball) dynamics simulated on GPUs in Figure 8

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[9].

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

```
// Building textured memory of the datset
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

6.7 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 bouncing ball problem from above extended to include drag forces imparted on the ball via the interpolation of a spatially varying wind-field. Furthermore, one may need to interpolate digital terrain elevation data for ground collision event handling. One can simply use textured memory defined in listing 2.

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 texture memory results in 2× faster simulation, comparing with interpolation and solve on CPU.

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

6.8 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 $O(\sqrt{N})$ convergence rate to the expectation. The generation of these trajectories are 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 Pseudoandom Number Generator (PRNG).

6.8.1 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 [12, 41]. The benchmarks in figure 9 demonstrate that simulations on GPU are faster than CPU on average by 8×.

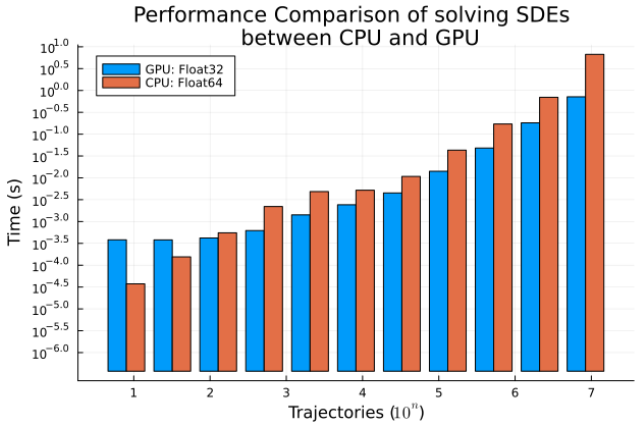

Figure 9: The plot shows the parameter parallel simulation time *(lower the better)*. It demonstrates that GPU parallelism super cedes CPU parallelism at about 1000 trajectories. The simulation is performed on the Linear SDE, as defined in section 6.8.1.

GPU parallelism dominates CPU parallelism over approximately over 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.

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 change in its gene expression through sigma factors [37]. These biological processes are inherently noisy in nature, and it is simulated by transforming the CRN to a SDE via the Chemical Langevin Equation (CLE) [22]. 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 benchmark investigates 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 are uniformly sampled and the set of Cartesian product of parameters are simulated, generating approximately $> 1,000,000$ unique trajectories. The benchmarks show that our GPU implementation for SDEs is 4.5× faster than multi-threading over a CPU.

7 DISCUSSION

We have demonstrated that many programs written for standard CPU usage of DifferentialEquations.jl can be re-targeted 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 the 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

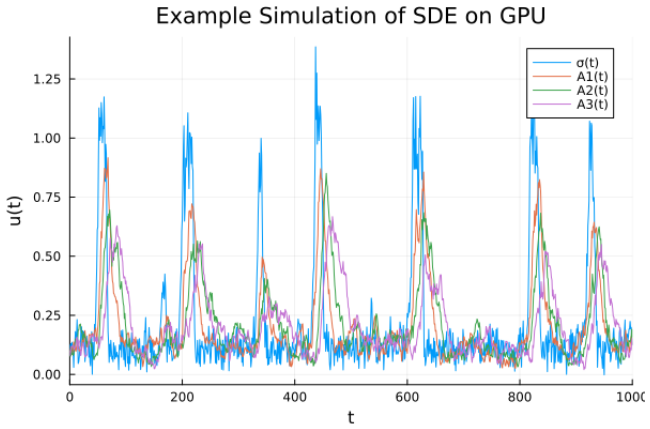

Figure 10: A example simulation plot of the system vs. time. The model is written with Catalyst.jl and automatically works with GPU solvers, showcasing the ability to simulate complex models seamlessly on GPU.

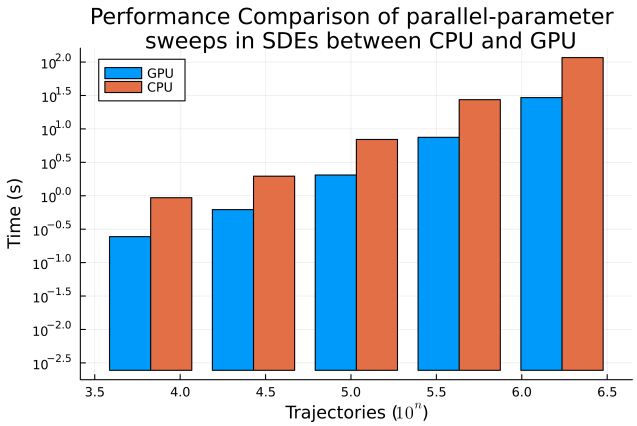

Figure 11: The plot shows the parameter parallel simulation time *(lower the better)* of the SDE simulation of Figure 10. Overall, the comparsion showcases the scalability of speed-ups of using GPUs instead of CPUs, having suitable gains for trajectories as small as 1000**.**

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) [4]. 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.

Although being the performant alternative of `EnsembleGPUArray`, 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 not suitable for higher-dimensional problems due to the limited memory of static allocations. The model might compile, but there might not be any realizable speed-ups. 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 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. The methods do not support solving stiff ODEs, which are more common in nature. 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.

ACKNOWLEDGMENTS

We thank Torkel Loman for his help with stochastic differential equation models. The authors acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC resources that have contributed to the research results reported within this paper. This material is based upon work supported by the National Science Foundation under grant no. OAC-1835443, grant no. SII-2029670, grant no. ECCS-2029670, grant no. OAC-2103804, and grant no. PHY-2021825. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001211 and DE-AR0001222. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This material was supported by The Research Council of Norway and Equinor ASA through Research Council project "308817 - Digital wells for optimal production and drainage". Research was sponsored by the United States Air Force Research Laboratory and Air Force Office of Scientific Research Lab Task #21RQCOR083 in addition to the United States Air Force Artificial Intelligence Accelerator accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Karsten Ahnert, Denis Demidov, and Mario Mulansky. 2014. Solving ordinary differential equations on GPUs. *Numerical Computations with GPUs* 1 (2014), 125–157.

[2] Karsten Ahnert and Mario Mulansky. 2011. Odeint–solving ordinary differential equations in C++. In *AIP Conference Proceedings*, Vol. 1389. American Institute of Physics, American Institute of Physics, Halkidiki, Greece, 1586–1589.

[3] Uri M Ascher and Linda R Petzold. 1998. *Computer methods for ordinary differential equations and differential-algebraic equations*. Vol. 61. Siam, Philadelphia, PA, United States.

[4] Paul Barham and Michael Isard. 2019. Machine Learning Systems Are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro,

Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441
[5] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, Boston, 359–371.
[6] Tim Besard. 2023. *JuliaGPU/GPUArrays.jl: v8.6.4*. Github. https://doi.org/10.5281/zenodo.7807091
[7] Tim Besard. 2023. *JuliaGPU/GPUCompiler.jl: v1.8.0*. Github. https://doi.org/10.5281/zenodo.7807140
[8] Tim Besard. 2023. *oneAPI.jl*. Github. https://doi.org/10.5281/zenodo.7789142 If you use this software, please cite it as below..
[9] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. 2019. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in engineering software* 132 (2019), 29–46.
[10] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2018. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2018), 827–841.
[11] Tim Besard and Max Hawkins. 2023. *Metal.jl*. Github. https://doi.org/10.5281/zenodo.7789146 If you use this software, please cite it as below..
[12] Fischer Black and Myron Scholes. 1973. The pricing of options and corporate liabilities. *Journal of political economy* 81, 3 (1973), 637–654.
[13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. Github. http://github.com/google/jax
[14] Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Design, optimization, and benchmarking of dense linear algebra algorithms on AMD GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, Waltham, MA, USA, 1–7.
[15] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. 2021. MPI.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, Vol. 1. JuliaCon, Online, 68.
[16] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. 2018. Neural ordinary differential equations. *Advances in neural information processing systems* 31 (2018).
[17] Valentin Churavy, Dilum Aluthge, Julian Samaroo, Anton Smirnov, James Schloss, Lucas C Wilcox, Simon Byrne, Maciej Waruszewski, Ali Ramadhan, Meredith, Simeon Schaub, Navid C. Constantinou, Jake Bolewski, Max Ng, Tim Besard, Ben Arthur, Charles Kawczynski, Chris Hill, Christopher Rackauckas, James Cook, Jinguo Liu, Michel Schanen, Oliver Schulz, Oscar, Páll Haraldsson, Takafumi Arakaki, and Tomas Chor. 2023. *JuliaGPU/KernelAbstractions.jl: v0.9.1*. JuliaGPU. https://doi.org/10.5281/zenodo.7770454
[18] Denis Demidov. 2012. VexCL: Vector expression template library for OpenCL.
[19] John R Dormand and Peter J Prince. 1980. A family of embedded Runge-Kutta formulae. *Journal of computational and applied mathematics* 6, 1 (1980), 19–26.
[20] Milinda Fernando, David Neilsen, Eric Hirschmann, Yosef Zlochower, Hari Sundar, Omar Ghattas, and George Biros. 2022. 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, IEEE, Dallas, Texas, 1078–1092.
[21] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. 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, Springer Berlin Heidelberg, Berlin, Heidelberg, 97–104.
[22] Daniel T Gillespie. 2000. The chemical Langevin equation. *The Journal of Chemical Physics* 113, 1 (2000), 297–306.
[23] Hairer and Peters. 1991. *Solving Ordinary Differential Equations II*. Springer Berlin Heidelberg, Berlin, Heidelberg.
[24] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. 1993. Solving Ordinary Differential Equations I: Nonsti Problems, volume Second Revised Edition.
[25] Ferenc Hegedűs. 2021. 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* 97 (2021), 105732.
[26] Desmond J Higham and Lloyd N Trefethen. 1993. Stiffness of odes. *BIT Numerical Mathematics* 33 (1993), 285–303.
[27] Alan C Hindmarsh. 1983. ODEPACK, a systemized collection of ODE solvers. *Scientific computing* 1 (1983).
[28] Alan C Hindmarsh and Linda R Petzold. 1995. Algorithms and software for ordinary differential equations and differential-algebraic equations, Part II: Higher-order methods and software packages. *Computers in Physics* 9, 2 (1995), 148–155.
[29] Bertrand Iooss and Paul Lemaître. 2015. A review on global sensitivity analysis methods. *Uncertainty management in simulation-optimization of complex systems: algorithms and applications* 1 (2015), 101–122.
[30] Patrick Kidger. 2021. *On Neural Differential Equations*. Ph. D. Dissertation. University of Oxford.
[31] Peter E Kloeden, Eckhard Platen, Peter E Kloeden, and Eckhard Platen. 1992. *Stochastic differential equations*. Springer Berlin Heidelberg, Berlin, Heidelberg.
[32] Clemens Kühn, Christoph Wierling, Alexander Kühn, Edda Klipp, Georgia Panopoulou, Hans Lehrach, and Albert J Poustka. 2009. Monte carlo analysis of an ode model of the sea urchin endomesoderm network. *BMC systems biology* 3 (2009), 1–18.
[33] Wilhelm Kutta. 1901. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, Leipzig.
[34] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15, Article 7)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2833157.2833162
[35] Scott Le Grand, Andreas W Götz, and Ross C Walker. 2013. SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications* 184, 2 (2013), 374–380.
[36] Xuechen Li, Ting-Kam Leonard Wong, Ricky TQ Chen, and David K Duvenaud. 2020. Scalable gradients and variational inference for stochastic differential equations. In *Symposium on Advances in Approximate Bayesian Inference*. PMLR, PMLR, -, 1–28.
[37] Torkel Loman. 2022. *How bacteria tune mixed positive/negative feedback loops to generate diverse gene expression dynamics*. Ph. D. Dissertation. University of Cambridge.
[38] Edward N Lorenz. 1963. Deterministic nonperiodic flow. *Journal of atmospheric sciences* 20, 2 (1963), 130–141.
[39] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. 2012. ArrayFire: a GPU acceleration platform. In *Modeling and simulation for defense systems and applications VII*, Vol. 8403. SPIE, SPIE, Baltimore, Maryland, United States, 49–56.
[40] Simeone Marino, Ian B Hogue, Christian J Ray, and Denise E Kirschner. 2008. A methodology for performing global uncertainty and sensitivity analysis in systems biology. *Journal of theoretical biology* 254, 1 (2008), 178–196.
[41] Robert C Merton. 1973. Theory of rational option pricing. *The Bell Journal of economics and management science* 1 (1973), 141–183.
[42] Nicholas Metropolis and Stanislaw Ulam. 1949. The monte carlo method. *Journal of the American statistical association* 44, 247 (1949), 335–341.
[43] Dániel Nagy, Lambert Plavecz, and Ferenc Hegedűs. 2020. 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* 1 (2020).
[44] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (2008), 40–53.
[45] Keith Obenschain, Douglas Schwer, and Alisha Sharma. 2020. Initial assessment of the AMD MI50 GPGPUs for scientific and machine learning applications. Research poster presented at ISC High Performance 2020.
[46] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. 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 (NIPS)*. NIPS, Long Beach, CA, USA. http://learningsys.org/nips17/assets/papers/paper_16.pdf
[47] Christopher Rackauckas and Qing Nie. 2017. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software* 5, 1 (2017).
[48] Chris Rackauckas and Qing Nie. 2020. Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, Waltham, MA, USA, 1–8.
[49] Carl Runge. 1895. Über die numerische Auflösung von Differentialgleichungen. *Math. Ann.* 46, 2 (1895), 167–178.
[50] Julian Samaroo, Valentin Churavy, Anton Smirnov, Torrance Hodgson, Wiktor Phillips, Ludovic Räss, Ali Ramadhan, Jason Barmparesos, Julia TagBot, Michel Schanen, Tim Besard, Takafumi Arakaki, Stephan Antholzer, Alessandro, Alexis Montoison, Chris Elrod, Matin Raayai, and Tom Hu. 2023. *JuliaGPU/AMDGPU.jl: v0.4.8*. Github. https://doi.org/10.5281/zenodo.7641665
[51] Lawrence F Shampine and Mark W Reichelt. 1997. The matlab ode suite. *SIAM journal on scientific computing* 18, 1 (1997), 1–22.
[52] Lawrence F Shampine and Skip Thompson. 2007. Stiff systems. *Scholarpedia* 2, 3 (2007), 2855.
[53] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66.
[54] Albert Tarantola. 2005. *Inverse problem theory and methods for model parameter estimation*. SIAM, Philadelphia.
[55] A Tocino and Ramón Ardanuy. 2002. Runge–Kutta methods for numerical solution of stochastic differential equations. *J. Comput. Appl. Math.* 138, 2 (2002), 219–241.
[56] Angel Tocino and Jesus Vigo-Aguiar. 2002. Weak second order conditions for stochastic Runge–Kutta methods. *SIAM Journal on Scientific Computing* 24, 2

(2002), 507–523.
[57] Ch Tsitouras. 2011. Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications* 62, 2 (2011), 770–775.
[58] James H Verner. 2010. Numerically optimal Runge–Kutta pairs with interpolants. *Numerical Algorithms* 53, 2 (2010), 383–396.
[59] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.
[60] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.
[61] Guibin Wang, YiSong Lin, and Wei Yi. 2010. 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 Conference on Cyber, Physical and Social Computing*. IEEE, IEEE, Hangzhou, China, 344–350.
[62] Darren J Wilkinson. 2018. *Stochastic modelling for systems biology*. CRC press, Boc Raton, FL, USA.
[63] Tianchen Zhao, Saibal De, Brian Chen, James Stokes, and Shravan Veerapaneni. 2021. Overcoming Barriers to Scalability in Variational Quantum Monte Carlo. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 62, 13 pages. https://doi.org/10.1145/3458817.3476219
[64] Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael PH Stumpf, and Chris Barnes. 2011. GPU accelerated biochemical network simulation. *Bioinformatics* 27, 6 (2011), 874–876.

A ARTIFACT IDENTIFICATION

The manuscript aims to explore the parallel solving of the ensembles of ordinary differential equations (ODEs) and stochastic differential equations (SDEs) with GPUs. The methods automate the process by generating optimized GPU kernels without requiring any model changes by the user. The methods outperform the vectorized-map approach adopted by JAX and PyTorch and even hand-written CUDA C++ kernels, becoming state-of-the-art for solving ensemble ODEs/SDEs. The solution is high-performant with vendor-agnosticism: it works with NVIDIA, AMD, Intel, and Apple GPUs.

The paper evaluates the performance of the ODE and SDE solvers with benchmarking with CPU multi-threading using Intel Xeon Gold 6248 CPU @ 2.50GHz with 16 enabled threads. The composability of the methods with Julia's SciML ecosystem is also demonstrated by simulating a stochastic process based on a chemical reaction network model, written in a Julia-based Domain Specific Language (DSL) Catalyst.jl.

Further, we also benchmark against GPU-accelerated programs performed with ODE solvers. The GPUs used are Tesla V100 and RTX 500. The following software forms our case study:

- EnsembleGPUArray (Julia's DifferentialEquations.jl)
- MPGOS (CUDA C++)
- JAX (Python)
- PyTorch (Python)

The `EnsembleGPUArray` based code is written in Julia and is a part of DiffEqGPU.jl, the MPGOS program is written in C++, and the benchmarking scripts for JAX and PyTorch are written in Python. A standard workflow to integrate the polyglot software for benchmarking is done via bash scripts, which invoke the benchmarking scripts and store the execution times in a ".txt" file. Finally, the plotting scripts for these benchmarks are written in Julia to visualize the numbers obtained through benchmarking.

The description of the models used in benchmarking is provided in the next section.

```
function affect!(integrator)
    # Flips the sign of velocity
    # by coefficient of restitution
    integrator.u[2] = -integrator.e*integrator.u[2]
end

function condition(u, t, integrator)
    # returns true when distance
    # to ground is evaluated to zero
    u[1] # == 0
end
```

Listing 3: Implementation of callbacks

A.1 Models: Ordinary Differential Equations

A.1.1 Lorenz Problem. The first test problem is Lorenz attractor:

$$\frac{dy_1}{dt} = \sigma(y_2 - y_1), \tag{11}$$

$$\frac{dy_2}{dt} = \rho y_1 - y_2 - y_1 y_3, \tag{12}$$

$$\frac{dy_3}{dt} = y_1 y_2 - \gamma y_3. \tag{13}$$

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 time-step $dt = 0.001$, essentially generating 1000 fixed time-steps. The initial conditions are $y = [1.0, 0.0, 0.0]$

A.1.2 Bouncing-ball problem. The problem simulated in the callbacks example is the bouncing ball problem:

$$\frac{dx}{dt} = v, \tag{14}$$

$$\frac{dv}{dt} = -g, \tag{15}$$

With $g = 9.8m/s^2$ and e is the coefficient of restitution which varies across the simulation. The problem is simulated for time interval $(0.0, 15.0)$. However, this specifies the kinetics of the ball, but we will need to specify how the object will behave when it hits the ground. This is done via event handling, described in the listing 3.

A.2 Models: Stochastic Differential Equations

A.2.1 Linear SDE. The asset-price model, simply expressed as:

$$dX_t = rX_t dt + VX_t dW_t \tag{16}$$

Where X_t is the state to be simulated with initial condition $X_{t_0} = [0.1, 0.1, 0.1]$. Here, $r = 1.5$ is the risk-free rate of interest and $V = 0.01$ is the volatility.

A.2.2 Biological Chemical Reaction Network (CRN) simulating generalized stress response model of bacterial sFigma factors. The biological CRN models the regulation of sigma factors via sigma factor circuits, which critically regulate gene expression during the bacterial stress response. These phenomena are stochastic in nature. The

resulting process can be expressed as an SDE, generated through Chemical Langevin Equation (CRE) can be written as:

$$d[\sigma] = dt(v_0 + \frac{(S[\sigma])^n}{(S[\sigma])^n + (D[A3])^n + 1} - [\sigma]) \tag{17}$$

$$+ \eta\sqrt{v_0 + \frac{(S[\sigma])^n}{(S[\sigma])^n + (D[A3])^n + 1}}dW_1 - \eta\sqrt{[\sigma]}dW_2, \tag{18}$$

$$d[A_1] = dt(\frac{[\sigma]}{\tau} - \frac{[A_1]}{\tau}) \tag{19}$$

$$+ \eta\sqrt{\frac{[\sigma]}{\tau}}dW_3 - \eta\sqrt{\frac{[A_1]}{\tau}}dW_4, \tag{20}$$

$$d[A_2] = dt(\frac{[A_1]}{\tau} - \frac{[A_2]}{\tau}) \tag{21}$$

$$+ \eta\sqrt{\frac{[A_1]}{\tau}}dW_5 - \eta\sqrt{\frac{[A_2]}{\tau}}dW_6, \tag{22}$$

$$d[A_3] = dt(\frac{[A_2]}{\tau} - \frac{[A_3]}{\tau}) \tag{23}$$

$$+ \eta\sqrt{\frac{[A_2]}{\tau}}dW_7 - \eta\sqrt{\frac{[A_3]}{\tau}}dW_8, \tag{24}$$

$$\tag{25}$$

The parameters being the set $(S, D, \tau, v_0, n, \eta)$. The range of parameters is specified in the table 4. The simulation is performed for the time-span $(0.0, 1000.0)$ with $dt = 0.1$ and initial condition $[[\sigma], [A_1], [A_2], [A_3]]$ being $[v_0, v_0, v_0, v_0]$.

Table 4: Range of parameters used in parameter-parallel SDE solutions

Parameter	Range
S	$0.1 \leq S \leq 100.0$
D	$0.1 \leq D \leq 100.0$
τ	$0.1 \leq \tau \leq 100.0$
v_0	$0.01 \leq S \leq 0.2$
n	$2 \leq S \leq 4$
η	$0.001 \leq S \leq 0.1$

The benchmarking measures the execution time of the methods in seconds. The timings are compared against the number of parallel solves (trajectories). Apart from benchmarks, we also present compatibility of the methods with standard scientific computing methodologies such as event-handling, automatic differentiation, HPC cluster scaling with MPI, and incorporation of the use of scientific datasets via textured memory. The methods are free to use via the open-source library, DiffEqGPU.jl, hosted on GitHub. The specific examples of the case studies and benchmarks are available on the complete benchmark suite:
https://github.com/utkarsh530/GPUODEBenchmarks.

Together, these evaluations allow the paper to convey the SOA performance of the methods, scalability with distributed and multiple GPU vendors, reusability by re-targeting existing CPU-targeted user's code to GPUs, and composability with other tools of the high-level programming language.

B REPRODUCIBILITY OF EXPERIMENTS

The methods are written in Julia, and are part of the repository, https://github.com/SciML/DiffEqGPU.jl. The benchmark suite also consists of the raw data, such as simulation times and plots mentioned in the paper. The supported OS for the benchmark suite is Linux.

B.1 Installing Julia

Firstly, we will need to install Julia. The user can download the binaries from the official JuliaLang website https://julialang.org/downloads/. Alternatively, one can use the convenience of a Julia version multiplexer, https://github.com/JuliaLang/juliaup. The recommended OS for installation is Linux. The recommended Julia installation version is v1.8. To use AMD GPUs, please install v1.9. The Julia installation should also be added to the user's path.

B.2 Setting up DiffEqGPU.jl

B.2.1 Installing backends. The user must install the GPU backend library for testing 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
```

B.2.2 Testing DiffEqGPU.jl. DiffEqGPU.jl is a test suite that regularly checks functionality by testing features like 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 by this command

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

Additionally, the GitHub discussion https://github.com/SciML/DiffEqGPU.jl/issues/224#issuecomment-1453769679 highlights the use of textured memory with ODE solvers, accelerated the code by 2× over CPU.

B.2.3 Continuous Integration and Development. DiffEqGPU.jl is a fully featured library with regression testing, semver versioning, and version control. The tests are performed on cloud machines having a multitude of different GPUs https://buildkite.com/julialang/diffeqgpu-dot-jl/builds/705. These tests approximately completes in 30 minutes. The publicly visible testing framework serves as a testimonial of compatibility with multiple platforms and said features in the paper.

B.3 Testing GPU accelerated ODE Benchmarks with other programs

B.3.1 Benchmarking Julia (DiffEqGPU.jl) methods. We will need to install CUDA.jl for benchmarking. It is the only backend compatible with the ODE solvers in JAX, PyTorch, and MPGOS. To do so, one can follow the below process in the Julia Terminal:

```
$ julia
julia> using Pkg
julia> Pkg.add("CUDA")
```

Let's clone the benchmark suite repository to start benchmarking;

```
$ git clone https://github.com/utkarsh530\
/GPUODEBenchmarks.git
```

We will instantiate and pre-compile all the packages beforehand to avoid the wait times during benchmarking. The folder ./GPU_ODE_Julia contains all the related scripts for the GPU solvers.

```
$ cd ./GPUODEBenchmarks
$ julia --project=./GPU_ODE_Julia --threads=auto
julia> using Pkg
julia> Pkg.instantiate()
julia> Pkg.precompile()
julia> exit()
```

It may take a few minutes to complete (< 10 minutes). After this, we can generate the timings of ODE solvers written in Julia. There is a script to benchmark ODE solvers for the different number of trajectories to demonstrate scalability and performance. The script invocation and timings can be generated through the following:

```
$ bash ./run_benchmark.sh -l julia -d gpu -m ode
```

It might take around 20 minutes to finish. The flag `-n N` can be used to specify the upper bound of the trajectories to benchmark. By default $N = 2^{24}$, where the simulation runs for $n \in 8 \leq n < N$, with the multiples of 4.

The data will be generated in the `data/Julia` directory, with two files for fixed and adaptive time-stepping simulations. The first column in the ".txt" file will be the number of trajectories, and the section column will contain the time in milliseconds.

Additionally, to benchmark ODE solvers for other backends:

```
$ N = $((2**24))
Benchmark
$ backend = "Metal"
$ ./runner_scripts/gpu/run_ode_mult_device.sh\
$N $backend
```

B.3.2 Benchmarking C++ (MPGOS) ODE solvers. Benchmarking MPGOS ODE solverse requires the CUDA C++ compiler to be installed correctly. The recommended CUDA Toolkit version is >= 11. The installation can be checked through:

```
$ nvcc
If the installation exists, it will return
something like this:
nvcc fatal   : No input files specified;
use option --help for more information
```

If nvcc is not found, the user needs to install CUDA Toolkit. The NVIDIA website lists out the resource https://developer.nvidia.com/cuda-downloads for installation.

The MPGOS scripts are in the `GPU_ODE_MPGOS` folder. The file `GPU_ODE_MPGOS/Lorenz.cu` is the main executed code. However, the MPGOS programs can be run with the same bash script by changing the arguments as:

```
$ bash ./run_benchmark.sh -l cpp -d gpu -m ode
```

It will generate the data files in `data/cpp` folder.

B.3.3 Benchmarking JAX (Diffrax) ODE solvers. Benchmarking JAX-based ODE solvers require installing Python 3.9 and conda. First, we will install all the Python packages for benchmarking:

```
$ conda env create -f environment.yml
$ conda activate venv_jax
```

It should install the correct version of JAX with CUDA enabled and the Diffrax library. The GitHub https://github.com/google/jax#installation is a guide to follow if the installation fails.

For our purposes, we can benchmark the solvers by:

```
$ bash ./run_benchmark.sh -l jax -d gpu -m ode
```

B.3.4 Benchmarking PyTorch (torchdiffeq) ODE solvers. Benchmarking PyTorch based ODE solvers is a similar process compared to JAX ones.

```
$ conda env create -f environment.yml
$ conda activate venv_torch
```

`torchdiffeq` does not fully support vectorized maps with ODE solvers. To circumvent this, we extended the functionality by rewriting some library parts. To download it:

```
(venv_torch)$ pip uninstall torchdiffeq
(venv_torch)$ pip uninstall torchdiffeq
(venv_torch)$ pip install git+https://github.com/
\utkarsh530/torchdiffeq.git@u/vmap
```

Then run the benchmarks by:

```
$ bash ./run_benchmark.sh -l pytorch -d gpu -m ode
```

B.4 Comparing GPU acceleration of ODEs with CPUs

The benchmark suite can also be used to test the GPU acceleration of ODE solvers in comparison with CPUs. The process for generating simulation times for GPUs can be done by following the section B.3.1. The following bash script allows the generation of CPU simulation times for ODEs:

```
$ bash ./run_benchmark.sh -l julia -d cpu -m ode
```

The simulation times will be generated in `data/CPU`. Each of the workflow approximately takes around 20 minutes to finish.

B.5 Benchmarking GPU acceleration of SDEs with CPUs

The SDE solvers in Julia are benchmarked by comparing them to the CPU-accelerated simulation. This will benchmark the linear SDE with three states, as described in the "Benchmarks and case studies" section. To generate simulation times for GPU, do:

```
$ bash ./run_benchmark.sh -l julia -d gpu -m sde
```

We can generate the simulation times for CPU accelerated codes through:

```
$ bash ./run_benchmark.sh -p julia -d cpu -m sde
```

The results will get generated in `data/SDE` and `data/CPU/SDE`, taking around 10 minutes to complete.

B.6 Composability with MPI

Julia supports Message Passing Interface (MPI) to allow Single Program Multiple Data (SPMD) type parallel programming. The composability of the GPU ODE solvers enables seamless integration with MPI, enabling scaling the ODE solvers to clusters on multiple nodes.

```
$ julia --project=./GPU_ODE_Julia
julia> using Pkg
# install MPI.jl
julia> Pkg.add("MPI")
```

An example script solving the Lorenz problem for approximately 1 billion parameters is available in the `MPI` folder. A SLURM-based script is shown below.

```
#!/bin/bash
# Slurm Sbatch Options
# Reqeust no. of GPUs/node
#SBATCH --gres=gpu:volta:1
# 1 process per node
#SBATCH -n 5 -N 5
#SBATCH --output="./mpi_scatter_test.log-%j"
# Loading the required module

# MPI.jl requires memory pool disabled
export JULIA_CUDA_MEMORY_POOL=none
export JULIA_MPI_BINARY=system
# Use local CUDA toolkit installation
export JULIA_CUDA_USE_BINARYBUILDER=false

source $HOME/.bashrc
module load cuda mpi

srun hostname > hostfile
time mpiexec julia --project=./GPU_ODE_Julia\
./MPI/gpu_ode_mpi.jl
```

B.7 Plotting Results

The plotting scripts to visualize the simulation times. The scripts are located in `runner_scripts/plot` folder. These scripts replicates the benchmark figures in the paper. The benchmark suite contains the simulation data generated by authors, which can be used to verify the plots. Various benchmarks can be plotted, which are described in the different sections. The plotting scripts are based on Julia. As a preliminary step:

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

The plot comparison between Julia, C++, JAX, and PyTorch mentioned in the paper can be generated by using the below command:

```
$ julia --project=. ./runner_scripts/plot\
/plot_ode_comp.jl
```

The plot will get saved in `plots` folder.

Similarly, the other plots in the paper can generated by running the different scripts in the folder `runner_scripts/plot`.

```
plot performance of GPU ODE solvers
with multiple backends
$ julia --project=. ./runner_scripts/plot\
/plot_mult_gpu.jl
plot GPU ODE solvers comparsion with CPUs
$ julia --project=. ./runner_scripts/plot\
/plot_ode_comp.jl
plot GPU SDE solvers comparsion with CPUs
$ julia --project=. ./runner_scripts/plot\
/plot_sde_comp.jl
plot CRN Network sim comparsion with CPUs
$ julia --project=. ./runner_scripts/plot\
/plot_sde_crn.jl
```

To plot data generated by running the scripts, specify the location of the `data` as the argument to the mentioned command.

```
$ julia --project=. ./runner_scripts/plot/\
plot_mult_gpu.jl /path/to/data/
```