GSoC 2023 Project Proposal

Exploration of GPU Computing: GPU Acceleration for PDE Discretization in Trixi.jl using CUDA.jl

Author: Huiyu Xie (GitHub)
Last Updated: Apr 3, 2023
References: https://julialang.org/jsoc/gsoc/trixi/, https://trixi-framework.github.io/Trixi.jl/stable/, https://cuda.juliagpu.org/stable/

Things have been done during the application period:

- Solve an open issue from SciML SimpleNonlinearSolve.jl, check here https://github.com/huiyuxie/SimpleNonlinearSolve.jl
- Start a draft to implement the discretization and rewrite it for 1D advection equation, check here https://github.com/huiyuxie/linear_advection_cuda
  The raw implementation of discretization has been completed by following the tutorials, and the rewriting part is still in process. In order to make the rewriting process clear and easy to understand, a demo about how to transform discretization into pure matrix-vector operations was created, check here rewrite_demo. The demo rewrites the loop parts in the raw implementation and uses the same notation as in the raw implementation.

Note: now in the process of importing CuArray to represent existing arrays(vectors and matrices), and up to now all the processes are based on raw implementation. Rewriting methods provided by Trixi.jl (e.g. TreeMesh() and semidiscretize()) are temporarily not considered. In the future, if packing the implementation into different methods is needed, the methods will be rewritten.

Background

GPUs can provide considerable speedups compared to CPUs for computational fluid dynamic simulations of the kind performed in Trixi.jl. Julia provides several ways to implement efficient code on GPUs, including CUDA.jl, AMDGPU.jl, and KernelAbstractions.jl. CUDA.jl is a popular choice for NVIDIA GPUs, as it provides a mature and feature-rich interface to the CUDA platform.

The combination of Trixi.jl and CUDA.jl enables users to perform high-performance simulations of complex physical systems on NVIDIA GPUs, providing the speed and accuracy required for cutting-edge scientific research and engineering applications.

Problem

By default, Trixi.jl implementations are CPU-based, meaning that the code is executed on the CPU rather than on a GPU. However, Trixi.jl provides support for GPU acceleration via the use of packages such as CUDA.jl, allowing users to take advantage of the performance benefits of GPUs when available.

However, not all functionality in Trixi.jl has been ported to GPUs yet. Trixi.jl uses a method of semi-discretization to solve partial differential equations (PDEs). First discretize the PDE in space using meshes and then solve the resulting ODE system numerically. The execution of these processes can achieve speedups on GPUs compared to CPUs.

Therefore, it is necessary to implement the semi-discretization functionality of Trixi.jl on GPUs platform to achieve better performance.

Goal and Tasks

In this project, the goal is to implement the semi-discretization functionality of Trixi.jl on GPUs. The basic building blocks of a semi-discretization are mesh, equations, and solver. This project focuses on cartesian meshes in Trixi.jl, specifically Trixi.TreeMesh, to perform numerical schemes.

The TreeMesh can be generated in multiple spatial dimensions, including 1D, 2D, and 3D. The project starts with 1D cartesian meshes, then expands to 2D, 3D, and also more complex geometries and sophisticated discretizations.

Here are the specific tasks to implement this project:

1. Write a simple 1D code for CPUs by taking the methods implemented in Trixi.jl as a blueprint.

- Choose the sample problem: example PDE problems that can be solved with 1D TreeMesh can be found in Trixi.jl/examples/tree_1d_dgsem, such as the advection equation or the Burgers' equation
- Analyze the methods: take the linear advection equation(basic) as an example, the main parts need to implement are 1) semi-discretization, 2) ODE solvers and callbacks, and 3) simulation
- Implement the methods:
    1) DGSEM, TreeMesh, SemidiscretizationHyperbolic
    2) semidiscretize, other callbacks
    3) solve (from OrdinaryDiffEq.jl, possibly not considered)

2. Port the simple 1D CPU code to GPUs using one of the GPU packages as a prototype.

- Identify the acceleration sections: find out the parts of code that can benefit from the high parallelism offered by GPUs, possibly DGSEM and TreeMesh
- Convert the data structures: move data from the CPU to GPU by converting the data structures to CUDA-compatible types, such as CuArrays (Array programming in CUDA.jl)
- Write and launch CUDA kernels: write CUDA kernels and launch CUDA kernels on GPUs with reference to Kernel programming in CUDA.jl
- Keep data on the GPUs: when the initial data is loaded onto the GPU memory, keep data on the GPUs and all the subsequent kernel functions operate on the data stored in the GPU memory (Memory management in CUDA.jl)
- Prototype GPU implementations: implement the whole data pipeline, that is, moving data from the CPU to GPU and back again explicitly, to help develop GPU prototype.
- Optimize the GPU code: optimize the performance by CUDA kernel parallelization (Synchronization, similar to Parallelization in Trixi.jl), and further optimize by use Profiling provided by CUDA.jl
- Compare the performance: compare the code performance on the CPU and GPU by benchmarking

3. Extend the GPU implementations to more complex numerical methods and settings, and to different types of GPUs.

- Extend to 2D and 3D: after the completion of 1D GPU prototype, it is easy to extend implementation to 2D and 3D GPU prototype by repeating the sequential substeps in above point 2

- Extend to more methods and setting: focus on curvilinear meshes such as Trixi.StructuredMesh and another kind of solver like DGMulti
- (Optional) Extend to different types of GPUs: explore GPU computing on different GPU programming packages in Julia, such as AMDGPU.jl for AMD GPUs, and KernelAbstractions.jl, which provides a single frontend to generate code for multiple GPU backends

Timeline

This project is scheduled to be completed within 350 hours, starting on `May 29, 2023`, and ending on `Nov 6, 2023`. (GSoC 2023 Timeline). The following week-by-week timeline provides a guideline of how the project will be done:

1. Start up (before May 29, 2023):

    - ☑ Get familiar with Trixi.jl and CUDA.jl repositories
    - ☑ Review materials about Numerical Analysis and GPU Computing (CUDA Toolkit Doc)
    - ☑ Prepare cloud computing resources (AWS) for running NVIDIA GPUs

2. In progress (May 29, 2023 - Nov 6, 2023):

    First Stage----------

    Week 1, 2, 3 (May 29 - Jun 18):
    - ☑ Choose the sample problem
    - ☑ Analyze the methods
    - ☑ Implement the methods

    Week 4, 5, 6 (Jun 19 - Jul 9):
    - ☑ Identify the acceleration sections
    - ☑ Convert the data structures
    - ☑ Write and launch CUDA kernels

    Week 7 (Jul 10 - Jul 16):
    - ☑ Keep data on the GPUs

    Week 8, 9, 10 (Jul 17 - Aug 6):
    - ☑ Prototype GPU implementations
    - ☑ Optimize the GPU code

    Week 11 (Aug 7 - Aug 13)

- ☑ Compare the performance

Second Stage----------

Week 12, 13, 14 (Aug14 - Sep 17)

- ☑ Extend to 2D and 3D

Third Stage----------

Week 15, 16, 17, 18, 19, 20 (Sep 18 - Nov 6)

- ☐ Extend to more methods and setting

Future Work

Probably will continue to work on extending functionality of Trixi.jl to different types of GPUs, like AMDGPU.jl and KernelAbstractions.jl if permitted.