An Investigation Into Parallelizing the RNEA Algorithm Using PyTorch And CUDA.jl

Eden Chung[1], Annie Wang[1], Siying Ding[1], Neasha Mittal[1]

***Abstract*— This research paper investigates parallelizing the dynamics algorithm, the Recursive Newton-Euler Algorithm (RNEA), using the PyTorch library in Python and CUDA.jl in Julia. The focus is on improving computational efficiency by leveraging CPU and GPU/CUDA programming. The study presents a comparative analysis of both libraries' performance in running this algorithm. Our contributions include developing optimized parallel implementations of the RNEA algorithm across Python and Julia, which are then evaluated across various benchmarks. The results demonstrate that PyTorch and CUDA.jl both have advantages and disadvantages, and should be used in different situations.**

I. Introduction

Being able to parallelize code and algorithms would allow for solving problems even faster, but lots of code, especially the ones written in high-level languages, remain unparallelized due to parallization challenges. Especially for robotics dynamics algorithms, it is essential to have quick algorithms that can make decisions in real time so that the mechanics of the robot can make an accurate and timely decision. Currently, much of the GPU code is written in C/C++ due to its speed, but are there tools that can do the same, while not sacrificing speed, in Python or Julia? Some Python and Julia tools allow programmers to leverage GPU parallelism, and we explore two of them: PyTorch (Python library) and CUDA.jl (Julia package).

Our contributions include:

1) Benchmark tests using PyTorch and CUDA.jl upon basic linear algebra computations
2) An implementation of the RNEA algorithm written in Python using PyTorch
3) An implementation of the RNEA algorithm written in Julia using CUDA.jl
4) A comparison of these tools and their advantages and disadvantages

The rest of the paper is organized as follows: Section II summarizes an overview of related work. Section III provides background on the libraries used and the RNEA algorithm. Section IV provides an overview of the implementation of the parallelized algorithms. Section V gives the results from the fully parallelized RNEA algorithms for both PyTorch and CUDA.jl. Finally, section VI summarizes our conclusions.

Our work can be found at `github.com/eden-chung/RNEA_GPU_Parallelization`.

[1]Eden Chung, Annie Wang, Siying Ding, Neasha Mittal are with Barnard College, Columbia University. {`ec3661, aw3515, sd3609, nm3401`}`@barnard.edu`

II. Related Work

Brian Plancher's A2R Lab has worked on implementing the RNEA algorithm in Python through the GRiD project (a GPU-accelerated library for computing rigid body dynamics with analytical gradients) [1], which establishes the baseline algorithm we will further explore in this paper. Previous research done in the field of GPU parallelism has investigated several Python libraries for parallelizing code, including JAX, OpenML, Numba, and CuPy, but these libraries also have downsides, such as the necessity to restructure the code which poses a steep learning curve for programmers [2]. In this paper, we will explore the potential of utilizing a more commonly used Python library called PyTorch as well as CUDA.jl in Julia to leverage GPU parallelism for implementations of the RNEA algorithm.

III. Background

A. CUDA Programming

CUDA Programming is an API that allows users to write parallel low-level code by utilizing GPU functionality. It is designed to run code on GPU to exploit the benefits of parallelism. Applications using CUDA, such as machine learning and data modeling, can run in parallel on lots of data.

The high level architectures used by CUDA are Grids, Blocks and Threads. The GPU launches one kernel at a time, and each kernel consists of b number of blocks, with each block consisting of t number of threads. Each thread runs in parallel and can share the same memory. (Figure 1)

B. PyTorch library

This project investigates a specific Python library, PyTorch, which is an optimized library that leverages GPUs and CPUs. PyTorch, originally targeted for machine learning tasks such as building neural networks and training models, has been designed to allow Python programmers to seamlessly integrate CUDA/GPU code under the hood.

The PyTorch library uses a data structure known as tensors, which are used in a similar way to arrays and matrices. The advantage of tensors is that they can perform parallelization of operations when running on GPUs.

C. Julia

This project also investigates Julia. Known for its fast performance, it is an open source project created for parallelism and designed for numerical, scientific, and technical computing. In addition, it can be used for general purpose programming. Like Python, Julia offers a wide range of

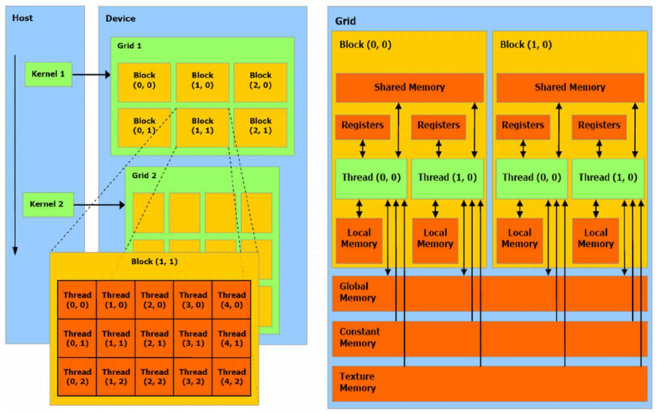

Fig. 1. Overview of CUDA programming architecture application.

libraries and packages. In this project, the library CUDA.jl[3] is used to leverage GPU and parallel programming.

D. Recursive Newton-Euler algorithm (RNEA)

The RNEA algorithm is a recursive inverse dynamics algorithm to calculate the forces required for a specific trajectory based on inputted parameters and constraints.

The dynamics can be written as follows:

$$\tau = fID(q, \dot{q}, \ddot{q}) = H(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q)$$

where τ is the torque, q is the position, $\dot{q}$ is the velocity, and $\ddot{q}$ is the acceleration[4].

More specifically, the algorithm can be divided into three parts:

1) Calculating the velocity and acceleration of each body in the tree[5]
   a) The base configuration is computed using the base of the robot, which is typically at rest or moving with a known motion. Starting from this base configuration, the algorithm computes the velocity and acceleration for each subsequent body (or joint) in the kinematic chain.
   b) Using the joint angles and their time derivatives (angular velocities and angular accelerations), along with the link configurations, it calculates the linear and angular velocities and accelerations for each link. This involves transforming the velocities and accelerations from one joint to the next in the tree, accounting for the relative motion and orientation between connected links.
2) Calculating the forces necessary to produce these accelerations
   a) Once the velocities and accelerations of all bodies are known, the algorithm computes the forces and moments (torques) that must act on each body to produce these accelerations. This calculation uses Newton's second law (Force = Mass × Acceleration) and Euler's equations for rotational motion.
   b) For each link, the necessary force and torque include contributions from the mass of the link, its acceleration, and external forces such as gravity. This stage is essential to understand the dynamic interactions within the robot due to its motion and external forces acting upon it.
3) Calculating the forces transmitted across the joints from the forces acting on the bodies
   a) This final stage involves back-propagating the forces and torques from the end-effector back to the base of the robot. Each joint transmits forces to its preceding joints in the kinematic chain.
   b) The forces and torques computed at each link are resolved into joint torques, which are the outputs necessary for the actuators at the joints. This step considers the geometry and type of each joint (e.g., revolute, prismatic) to properly calculate how forces are transmitted through the mechanical structure.
   c) This resolution of forces ensures that all dynamic interactions and load distributions are accounted for, providing the necessary joint torques to drive the robot as intended, under the dynamics modeled by the previous steps.

IV. Implementation

Initially, the base code for the RNEA algorithm from the A2R lab[1] consists of a Python algorithm for a batch size of 1.

The code is modified to be able to run an arbitrary number of batches at once, in order to best test the quality of the parallel code. To do this, new functions are created to allow the additional dimension for the matrices. For example, the following code block (Figure 2) demonstrates modifications to allow the code to be batched.

For both Python and Julia, this base batched code was then parallelized using PyTorch and CUDA.jl respectively. To do so, the RNEA algorithm was split into smaller functions: `cross_operator()`, `mxS()`, `vxIv()`, `rnea_fpass()`, `rnea_bpass()`, which were then parallelized.

Fig. 2. A Code Block

```
#Code before batching
if parent_ind == -1:
    a[:,ind] = np.matmul(Xmat,
        gravity_vec)
else:
    v[:,ind] = np.matmul(Xmat,v[:,
        parent_ind])
    a[:,ind] = np.matmul(Xmat,a[:,
        parent_ind])

#Modified code for batching
if parent_ind == -1:
    a[:, ind, :] = np.sum(Xmat*
        gravity_vec[:, np.newaxis, :],
         axis=1)
else:
    v[:, ind, :] = np.sum(Xmat*v[:,
        ind, :], axis=1)
    a[:, ind, :] = np.sum(Xmat*a[:,
        ind, :], axis=1)
```

V. Results

A. Initial Investigation

We first run initial benchmarking tests in an isolated environment with three computationally expensive algorithms: matrix multiplication, dot product, and matrix inversion. We run tests first for Python, then for Julia.

For Python, the difference in speed is negligible for lower number of iterations, but when calling these functions with a higher order number of times, the difference becomes much more obvious.

However, it is unclear whether running these PyTorch computations on a larger scale such as the full RNEA algorithm would have the same benefits as in an isolated environment.

On the initial investigation of Julia, benchmarks show that it is very fast, especially in comparison to Python. Even before CUDA or GPU parallelization, unaltered Julia code is very quick, suggesting that it could potentially be used over Python in applications such as robotics where the speed of computations is important.

For GPU programming, a Julia package, similar to Python packages, called CUDA.jl [6] is used. This package is the main Julia package for working with NVIDIA GPUs.

As mentioned, the benchmark tests show unaltered Julia code to be very quick, but using CUDA.jl can help prevent exponential blow-up on large matrix sizes.

Ultimately, the results are compared between Python (NumPy), PyTorch, Julia, and CUDA. Across the different computations tested, Python is always the slowest, followed by the serial Julia code. Although Python is the slowest, for matrix multiplication (Figure 3), PyTorch is the fastest (even faster than CUDA.jl by a minimal amount). For matrix inversion (Figure 4), CUDA.jl results in the best performance, surpassing others by by a significant amount. It appears that using either of these two libraries, CUDA.jl or PyTorch, can prevent an exponential blow-up of the runtime on computationally heavy operations, but we still need to investigate which library is better suited for which use-cases, and more specifically for the implementation of the RNEA algorithm.

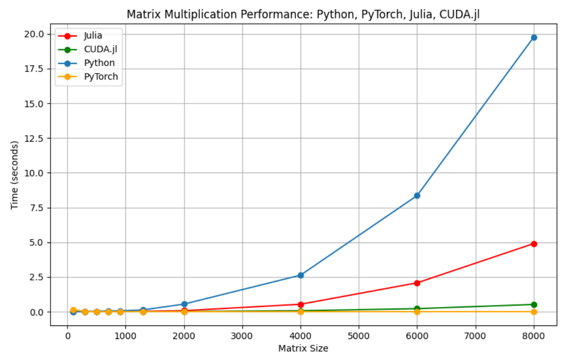

Fig. 3. Benchmarks on matrix multiplication across Python, PyTorch, Julia, CUDA.jl

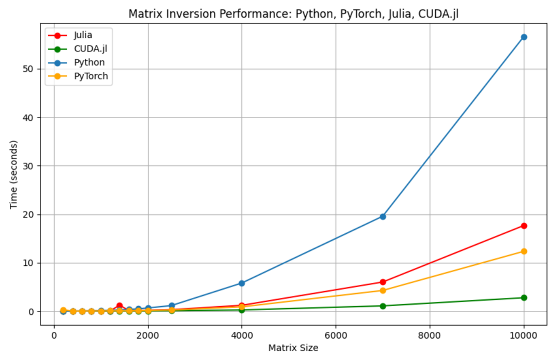

Fig. 4. Benchmarks on matrix inversion across Python, PyTorch, Julia, CUDA.jl

B. PyTorch

We convert the RNEA algorithm implementation from NumPy to PyTorch. For each RNEA-related function implementation in both NumPy and PyTorch, we perform 1 iteration of each function and run 1,000 trials in total to obtain more accurate timing results.

Even though PyTorch exhibits slower performance than NumPy when running `mxS` and `xvIv`, its overall performance surpasses NumPy when running more complicated RNEA-related functions such as `bpass`, `fpass`, and the full RNEA algorithm (Figure 5). This is due to PyTorch's utilization of tensors, which are multi-dimensional arrays that can automatically parallelize operations on GPU. It is worth noting that PyTorch incurs I/O overhead when creating and copying tensors onto the GPU, which becomes apparent when running simpler functions like mxS and xvIv due to their short runtime. In other words, when executing functions that are already optimized and fast on CPU, the I/O overhead outweighs the speedup gained from GPU parallelization. However, for more complex functions such as bpass and fpass, the speedup achieved by GPU parallelization outweighs the I/O overhead, leading to faster performance.

After familiarizing oneself with PyTorch, it becomes fairly straightforward to convert NumPy implementation to PyTorch, such as replacing `np.sum(vec_output * S, axis=1)` (where vec_output and S are NumPy arrays) with `torch.sum(vec_output * S, dim=1)` (where vec_output and S are tensors). This makes PyTorch a desirable Python library to leverage if programmers want to take advantage of GPU parallelism, despite the limitation of the amount of controls the programmers have over how operations are distributed across various blocks and threads on GPU under the hood.

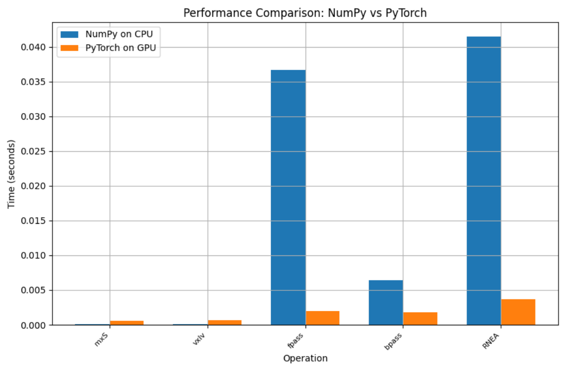

Fig. 5. Performance Comparison Between NumPy and PyTorch Implementations of Various RNEA-related Operations (Averaged Across 1000 trials)

C. CUDA.jl

Converting the Python code to Julia, on its own, without parallelizing, already has beneficial effects on the runtime of the algorithm. In converting Python code to Julia, however, there are several remarks that are important to keep in mind.

1) Julia uses 1-indexing as opposed to 0-indexing like many other programming languages, so this is important to keep in mind to avoid index out-of-range errors.
2) In Julia the * operator does not perform element-wise multiplication when arrays have different dimensions, whereas in Python, this is the case. Instead, in Julia, the .* operator performs element-wise multiplication, while * performs matrix multiplication.

To further accelerate the runtime, we convert each function to run in parallel by using the CUDA.jl library. For example, the cross_operator_batched is parallelized by launching threads and blocks to compute in parallel (Figure 6).

It appears that for Julia, either the startup time to run CUDA.jl is very long or the overhead from moving data from the CPU and GPU and vice versa is very large, so the results are surprising. For the functions that are parallelized successfully, the CUDA.jl code is significantly slower than the original Julia code.

Fig. 6. A Code Block

```
function cross_operator_batched_parallel(d_vec::CuDeviceMatrix{Float64, 1}, d_output::CuDeviceArray{Float64, 3, 1})
    idx = threadIdx().x
    stride = blockDim().x
    for k in idx:stride:size(d_vec, 2)
        d_output[1, 2, k] = -d_vec[2, k]
        d_output[1, 3, k] = d_vec[1, k]
        d_output[2, 1, k] = d_vec[2, k]
        d_output[2, 3, k] = -d_vec[1, k]
        d_output[3, 1, k] = -d_vec[2, k]
        d_output[3, 2, k] = d_vec[1, k]

        ...
    end
    return
end
```

In addition, CUDA.jl is not the simplest to use, as it does sometimes require refactoring code in order to run it in parallel, as opposed to PyTorch, where some functions such as np.sum can be replaced with torch.sum. It requires some knowledge of CUDA programming, so we recommend using CUDA.jl over PyTorch for situations where having more control over the code is wanted and the user has knowledge on and is willing to write CUDA code.

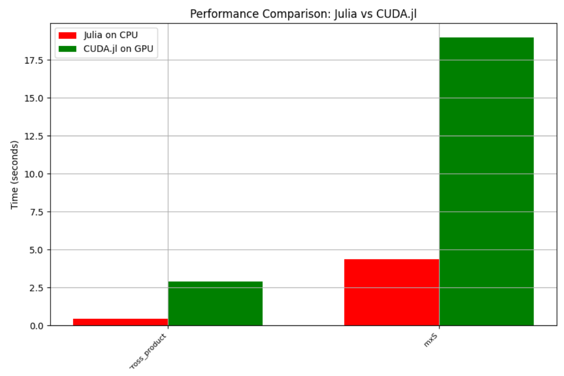

Fig. 7. Performance Comparison Between Julia and CUDA.jl implementations of RNEA-related Operations (Averaged across 100 trials)

It would be important to investigate exactly what is making the CUDA.jl code slow in order understand how to best leverage the library.

D. Methodology

Running and timing the Python code is done on Google Colab, using the Tesla T4 GPU, whereas a Dell XPS 15 9560 with a NVIDIA GPU, the GeForce GTX 1050 Mobile, is used for running benchmarks on Julia code.

Benchmark and timing tests are conducted by using time.time() in Python and time() in Julia. We consider time to be the elapsed time between the start of the computation and the end when the computation completes.

VI. Conclusion and Future Work

In this work, we investigate Python, Julia, PyTorch, and CUDA.jl with regard to a robotics dynamics algorithm called RNEA.

Among various implementations of the RNEA algorithm, PyTorch code is very fast, but it is not easy to customize, meaning that you must use PyTorch's built-in functions and have limited control over how operations are distributed and parallelized under the hood. Julia, on its own, is relatively fast, especially in comparison to Python, but our implementation of CUDA.jl does not seem to result in better runtime performances, potentially due to startup time or I/O overhead. However, the advantage of CUDA.jl over PyTorch is that a user can launch threads and blocks, allowing the code to be customized to a specific use case.

For future work, it will be useful to investigate if we can further parallelize NumPy implementation of RNEA with `ThreadPoolExecutor` or PyTorch implementation with `torch.multiprocessing`. In addition, it would be interesting to investigate PyCuda, which can potentially give users more control over how to parallelize the operations. Exploring ways to further optimize the CUDA.jl code and investigating what in particular is slowing the code down could be useful to learn how to best leverage the CUDA.jl library to make Julia implementation of RNEA even faster.

References

[1] B. Plancher, "Grid," https://github.com/A2R-Lab/GRiD, 2023.

[2] N. Demeure, T. Kisner, R. Keskitalo, R. Thomas, J. Borrill, and W. Bhimji, "High-level gpu code: a case study examining jax and openmp." in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1105–1113. [Online]. Available: https://doi.org/10.1145/3624062.3624186

[3] "Cuda programming in julia." [Online]. Available: https://cuda.juliagpu.org/stable/

[4] G. Sutanto, A. S. Wang, Y. Lin, M. Mukadam, G. S. Sukhatme, A. Rai, and F. Meier, "Encoding physical constraints in differentiable newton-euler algorithm," *CoRR*, vol. abs/2001.08861, 2020. [Online]. Available: https://arxiv.org/abs/2001.08861

[5] G. Buondonno and A. De Luca, "A recursive newton-euler algorithm for robots with elastic joints and its application to control," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 5526–5532.

[6] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.