GPU Particle Filter Implementation and Performance Analysis

Amy Phung

Abstract. Particle filters are frequently used in robotics and other applications for state estimation based on noisy data. Particle filters estimate the current state of a system by using a large number of "particles" to sample over the range of probable states, and iteratively reweight these particles based on their consistency with observed data. Compared to other state estimation techniques, particle filters are particularly good at addressing problems which involve nonlinear models, high-dimensional data, or significant amounts of uncertainty and measurement noise. However, these factors increase the number of particles the filter needs to accurately estimate the state, which increases the filter's computation time. This project presents parallelized CPU and GPU-based particle filter implementations in Julia, and compares their respective performance to the existing ParticleFilters.jl package.

Key words. particle filter, robot localization, parallelization, GPU

1. Motivation.

Many practical data analysis tasks require estimating unknown quantities based on observed data, whether its estimating the position of an airplane based on radar measurements, deciphering a noisy communications signal, or estimating parameters to improve stock market predictions [3]. In all of these applications, prior knowledge enables us to formulate models for estimating the likelihood of these observations and how these systems evolve over time. As these examples illustrate, it is often the case that the observed data is not available all at once in real-world applications. Instead, data often arrives sequentially over time, which provides motivation for an "online" method for continuously estimating the true state of a system based on the observable incoming data, such as a particle filter.

Particle filters are used to estimate the posterior density of state variables given observation variables. Given a prior density, sample "particles" are randomly drawn, and each particle's next state is estimated based on a known dynamics model. Each particle is then assigned a weight based on how likely the observation variable was, given the state represented by the updated particle. The collection of these weighted particles represents the posterior distribution of the state given the observation, and is used as the "prior" for the next observation step.

Within the field of robotics, particle filters have been widely used to address the challenge of estimating a robot's state based on its imperfect sensor data over the last two decades [6]. Particle filters are particularly relevant in current robotics research on localization methods since all real-world sensors have noise and often cannot be used to measure state variables directly, and thus posterior distributions of vehicle state must be estimated based on measurements provided by its sensors. In the case of underwater robotic vehicles, which cannot rely on GPS signals while below the surface, the robot's position can only be estimated based on its other sensors.

One approach for subsurface navigation relevant to my research is a method called Multi-Factor Terrain-Aided Navigation (MF-TAN), which uses a particle filter to estimate an underwater robot's position based on measurements from a doppler-velocity log (DVL), a vehicle dynamics model, and a known bathymetry map [4]. In shallow regions, the Terrain-Aided

Navigation (TAN) process is straightforward - the vehicle obtains a GPS fix to initialize its location while at the surface, then compares each new DVL measurement of the seafloor with its known bathymetry map to update its position estimate sequentially. However, in deep regions, the vehicle must travel significant distances without this "bottom-lock" since the DVL sensor has a limited range. During these transits, the vehicle relies on dead-reckoning to estimate its current position, which causes the positioning uncertainty to grow exponentially. With this large uncertainty in the vehicle's state, a large number of particles are needed in the particle filter for accurate state estimation. Finding a computationally efficient method for continually updating all of these particles while the vehicle is dead-reckoning, and discarding ones that are unlikely once bottom-lock is regained is an important step for enabling the MF-TAN approach to be used on the vehicle in real-time.

This project centers around exploring the use of parallelization to reduce the execution time required for using particle filters to process data with a large number of samples. During this project, I developed parallelized CPU and GPU-based particle filter implementations, and compared their performance to the existing ParticleFilters.jl package [5]. All code used in this project is available online at https://github.com/AmyPhung/GPUParticleFilter.jl

2. Background.

To start, let's formally formulate the problem we intend to solve with the particle filter (based on the forumlation in [1]). Consider a problem where we have an evolving state sequence

$$\mathbf{x}_t = \mathbf{f}_t(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}, \mathbf{v}_{t-1}) \tag{2.1}$$

where the function $\mathbf{f}_t : \mathbb{R}^{n_x} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_x}$ is the system's dynamics model, which describes how the state evolves over time based on the control input $\mathbf{u}_{t-1}$ and some process noise $\mathbf{v}_{t-1}$. Our objective is to estimate the state at each time step, $\mathbf{x}_t$, based on the observed data

$$\mathbf{z}_t = \mathbf{h}_t(\mathbf{x}_t, \mathbf{n}_t) \tag{2.2}$$

The function $\mathbf{h}_t : \mathbb{R}^{n_x} \times \mathbb{R}^{n_n} \to \mathbb{R}^{n_z}$ is the system's observation model, which describes the measurements based on state $\mathbf{x}_t$ with some observation noise $\mathbf{n}_t$. The "state estimation problem" aims to compute the posterior probability density function (PDF) of the state at all time steps given the observations and control inputs, which can be written as $p(\mathbf{x}_{0:t}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$.

In principle, the posterior density can be computed by starting with the prior distribution $p(\mathbf{x}_0)$, then recursively updating the distribution with a "prediction" and "update" stage. Given the distribution of the previous state $p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})$, the prediction stage uses the model 2.1 to compute the prior PDF of the state $\mathbf{x}_t$ with the equation

$$p(\mathbf{x}_t|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{t-1}) p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}) d\mathbf{x}_{t-1} \tag{2.3}$$

When the measurement $\mathbf{z}_t$ becomes available, the prior PDF can be updated using Bayes' rule

$$p(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t-1}) = \frac{p(\mathbf{z}_t|\mathbf{x}_t) p(\mathbf{x}_t|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \tag{2.4}$$

where the denominator can be written as

$$p(\mathbf{z}_t|\mathbf{z}_{1:t-1}) = \int p(\mathbf{z}_t|\mathbf{x}_t) p(\mathbf{x}_t|\mathbf{z}_{1:t-1}) d\mathbf{x}_t \tag{2.5}$$

In this equation, the term $p(\mathbf{z}_t|\mathbf{x}_t)$ depends on the observation model 2.2. Solving these equations at each timestep gives us the posterior density across all of the timesteps $p(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t-1})$ which provides us with the optimal estimate of the state given the observations and known models. Although the posterior PDF can be computed analytically in principle, it can be extremely computationally expensive or entirely infeasible to compute for high-dimensional problems with nonlinear noise or dynamics. Since the optimal solution is often intractable to compute, approximate methods such as the particle filter are used to estimate the posterior instead of computing the optimal solution in closed-form.

3. Particle Filter Algorithm. The particle filter implements a recursive Bayesian filter with Monte Carlo simulations. Instead of computing the exact posterior distribution, a series of samples are used to appoximately cover the probable regions of the state space, and a particle re-weighting process is used to estimate the posterior density. The posterior PDF can be approximated with the equation

$$p(\mathbf{x}_{0:t}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t-1}) \approx \sum_{i=1}^{N_s} w_k^i \delta(\mathbf{x}_{0:t} - \mathbf{x}_{0:t}^i) \tag{3.1}$$

where the weights are chosen based on the relative likelihood of the samples and are normalized such that $\sum_i w_k^i = 1$. The process for computing this sum can be described as 4 steps:

1. Given a prior density for the state space, draw randomly sampled "particles"
2. Compute weights for each particle based on the relative likelihood of the observed data
3. Resample particles from the re-weighted set
4. Propogate particles based on the control inputs and dynamics model

After the sum is computed, the collection of these weighted particles represents the posterior distribution of the state given the observation. This distribution is then used as the "prior" for the next observation step. In my implementation, I parallelized the reweighting, resampling, and propogation steps.

4. Benchmarking Dataset. For benchmarking, I used simulated data generated from a simplified model of an AUV. In this simplified model, it's assumed at the AUV remains at a constant depth, and can navigate around a 2D map. The AUV's state can be represented as the vector:

$$\mathbf{x}_t = \{x, y, \theta, v, d\theta\} \tag{4.1}$$

where x, y, θ encode the AUV's 2D pose, v is the AUV's linear velocity, and $d\theta$ is the glider's angular velocity. In this model, the AUV's control inputs can be written as:

$$\mathbf{u}_t = \{a, b\} \tag{4.2}$$

where a encodes the AUV's thruster input and b encodes the AUV's rudder input. The AUV's state evolves with the dynamics model $\mathbf{f}_t(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}, \mathbf{v}_{t-1})$. Within this function, the state

variables are updated using the following equations:

$$x_t = x_{t-1} + v_{t-1}\cos(\theta_{t-1})dt + \mathcal{N}(0,1) \tag{4.3}$$

$$y_t = y_{t-1} + v_{t-1}\sin(\theta_{t-1})dt + \mathcal{N}(0,1) \tag{4.4}$$

$$\theta_t = \theta + d\theta_{t-1}dt + 0.002\mathcal{N}(0,1) \tag{4.5}$$

$$v_t = v_{t-1} + (a_{t-1} - v_{t-1})dt \tag{4.6}$$

$$d\theta_t = d\theta_{t-1} + b_{t-1}dt \tag{4.7}$$

where $\mathcal{N}(0,1)$ encodes the noise, which is sampled from a standard gaussian distribution.

In this model, the AUV can use a depthsounder to measure its current depth, and reference it to a known bathymethry map to compute the likelihood. Thus, the observation model $\mathbf{h}_t(\mathbf{x}_t, \mathbf{n}_t)$ can be written as:

$$z_t = \text{LookupDepth}(x_t, y_t) + \eta\mathcal{N}(0,1) \tag{4.8}$$

where the function `LookupDepth` finds the depth value according to the bathymetry map at some position x, y, and the parameter η controls the magnitude of measurement noise added to the signal. The bathymetry map used in this is illustrated in Figure 1.

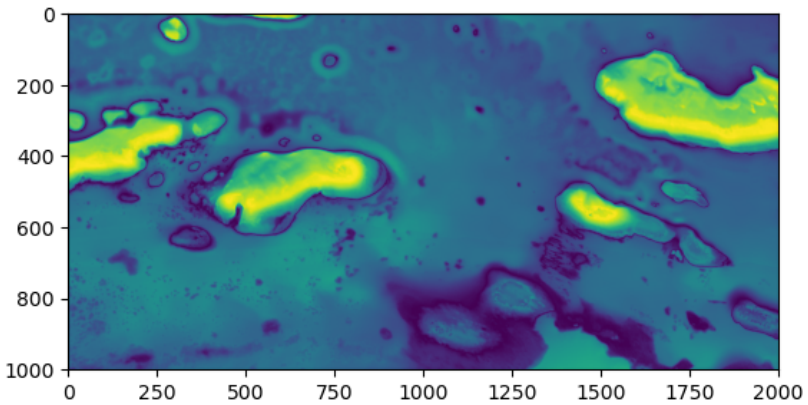

Figure 1. *Bathymetry map used for AUV measurement model. Elevated regions and islands are colored green and yellow, and deeper regions are shaded with darker colors*

5. CPU-based Serialized Implementation. The ParticleFilters.jl package [5] provides support for serialized CPU-based particle filters. For a basic filter, this package requires the user to provide the functions that encode the system's dynamics (2.1) and observation models (2.2). The dynamics function takes in the current state and the control input, and returns a value for the next state based on the dynamics. The observation function returns the likelihood of the measurement given the current state ($p(\mathbf{z}_t|p(\mathbf{x}_t))$). The AUV's measurement and dynamics model (Equations 4.4-4.7) are implemented as follows:

```
function auv_measurement_model(x1, u, x2, y, η, bathy_map, ε)
    x_pos, y_pos, θ, v, dθ = x2
    measured_depth = y
    model_depth = lookup_depth(x_pos, y_pos, bathy_map)
    dist = Normal(model_depth, η)
    pdf_value = pdf(dist, measured_depth)

    # Ensure all particles have at least small likelihood
    if pdf_value < ε
        return ε
    end

    return pdf_value
end
```

```
function auv_dynamics_model(x, u, delta_t)
    """
    inputs:
        x: state
        u: control input
        delta_t: time step
    output:
        updated state (same shape as x)
    """
    x_pos, y_pos, θ, v, dθ = x
    in_thruster, in_rudder = u

    x_step = x_pos + delta_t*v*cos(θ) + randn()*1
    y_step = y_pos + delta_t*v*sin(θ) + randn()*1
    θ_step = θ + delta_t*dθ + randn()*0.002

    # Update velocities based on input
    v_step = v + delta_t*(in_thruster-v)
    dθ_step = dθ + delta_t*in_rudder

    return [x_step, y_step, θ_step, v_step, dθ_step]
end
```

6. CPU-based Parallelized Implementation. My custom CPU based implementation uses the same measurement model and dynamics model functions as the implementation using the ParticleFilters.jl package, but the particle filter itself is designed to be run in parallel. In this implementation, the particle reweighting and propogation steps are wrapped in a for loop that is run using threads, as illustrated by the pseudocode listed below:

```
function runfilterthreaded!(particles, inputs, measurements,
    dynamics_model, measurement_model, output_particles)

    for idx in range(1, num_steps)
        u = inputs[idx,:]
        y = measurements[idx]

        # STEP 1: Compute weights ------------------
        Threads.@threads for p_idx = 1:num_particles
            p = particles[p_idx]
            weights[p_idx] =
                measurement_model(particle_buffer[p_idx], u, p, y)
        end
        cdf = Weights(weights ./ sum(weights))

        # STEP 2: Resample particles ---------------
        samples = sample(1:num_particles, cdf, num_particles)
        particles = particle_buffer[samples]
        particle_buffer = particles

        # STEP 3: Propogate particles --------------
        Threads.@threads for p_idx = 1:num_particles
            p = particles[p_idx]
            particles[p_idx] = dynamics_model(p, u, y)
        end
    end
end
```

To validate this implementation, the function was first run with serialized for-loops (i.e., `Threads.@threads` was commented out) with various particle sample sizes. For the values tested (n=10, 100 particles), the execution times closely matched that of the ParticleFilters.jl package. However, with threading, there was a notable increase in execution times. More detailed results from testing this threaded implementation are presented in Section 8.

7. GPU-based Parallelized Implementation. My GPU-based implementation used the CUDA.jl package [2], and was implemented with three separate kernels - one for reweighting, one for resampling, and one for propogating the particles. Three separate kernels were used instead of one to separate these steps to increase code reusability and ensure the individual kernels remained relatively simple. With this implementation, the user needs to provide a kernel for particle reweighting and propogation, but the resampling kernel can be reused between different applications. The overhead for initializing each kernel was approximately 200ms, which non-trivially increases the execution time but leads to substantial performance improvements for larger sample sizes, as discussed in Section 8. The pseudocode used for this implementation is listed below:

```
function runfiltergpu!(particles, inputs, measurements,
        reweight_kernel!, propogate_kernel!, output_particles, pf_data)
    reweight_kernel = @cuda launch=false reweight_kernel!(particles,
        inputs[1], measurements[1], weights_d, pf_data)
    reweight_config = launch_configuration(reweight_kernel.fun)
    reweight_threads = min(num_particles, reweight_config.threads)
    reweight_blocks = cld(num_particles, reweight_threads)

    ... # Setup resample and propogate kernels in similar fashion

    synchronize()

    for idx in range(1, num_steps)
        u = inputs[idx]
        y = measurements[idx]

        # STEP 1: Compute weights ------------------
        CUDA.@sync begin
            reweight_kernel(particles, u, y, weights_d, pf_data;
            threads=reweight_threads, blocks=reweight_blocks)
        end

        cumsum!(cdf_d, weights_d)
        cdf_d ./= cdf_d[end]

        # STEP 2: Resample particles ---------------
        CUDA.@sync begin
            resample_kernel(particles, particle_buffer, cdf_d;
            threads=resample_threads, blocks=resample_blocks)
        end

        # STEP 3: Propogate particles --------------
        CUDA.@sync begin
            propogate_kernel(particles, u, pf_data;
            threads=propogate_threads, blocks=propogate_blocks)
        end
    end
end
```

The reweight and propogate kernel functions bear similarity to the measurement and dynamics models from the CPU-based implementation, but needed to be re-written substantially to ensure GPU-compatible datatypes were used and handled properly. While the CPU-based functions for the measurement and dynamics model returned the output directly, this approach was not viable for the GPU implementation since kernels cannot return values. Instead, the

kernels were written to modify variables that are pre-allocated before execution. Although the implementation details differ slightly, the underlying function of these kernels match the modelling functions used for the CPU-based implementations.

8. Comparative Performance Analysis. To evaluate the relative performance of the ParticleFilters.jl implementation, my multithreaded CPU-based implementation, and my parallelized GPU-based implementation, I used each of the implementations to filter simulated data, which was generated using the AUV model described in Section 4. The AUV's "ground truth" trajectory was generated using the output from the dynamics model, with the thruster input set to a constant value of 0.5 and the rudder inputs generated from a randomly generated continous function. Once a sample trajectory was computed, the measurement model was used to generate a simulated set of noisy measurements. Some of the trajectories used for testing are illustrated in Figure 2. All experiments were run on a local machine with an AMD Ryzen 9 5900X processor and an NVIDIA GeForce RTX 3070 GPU.

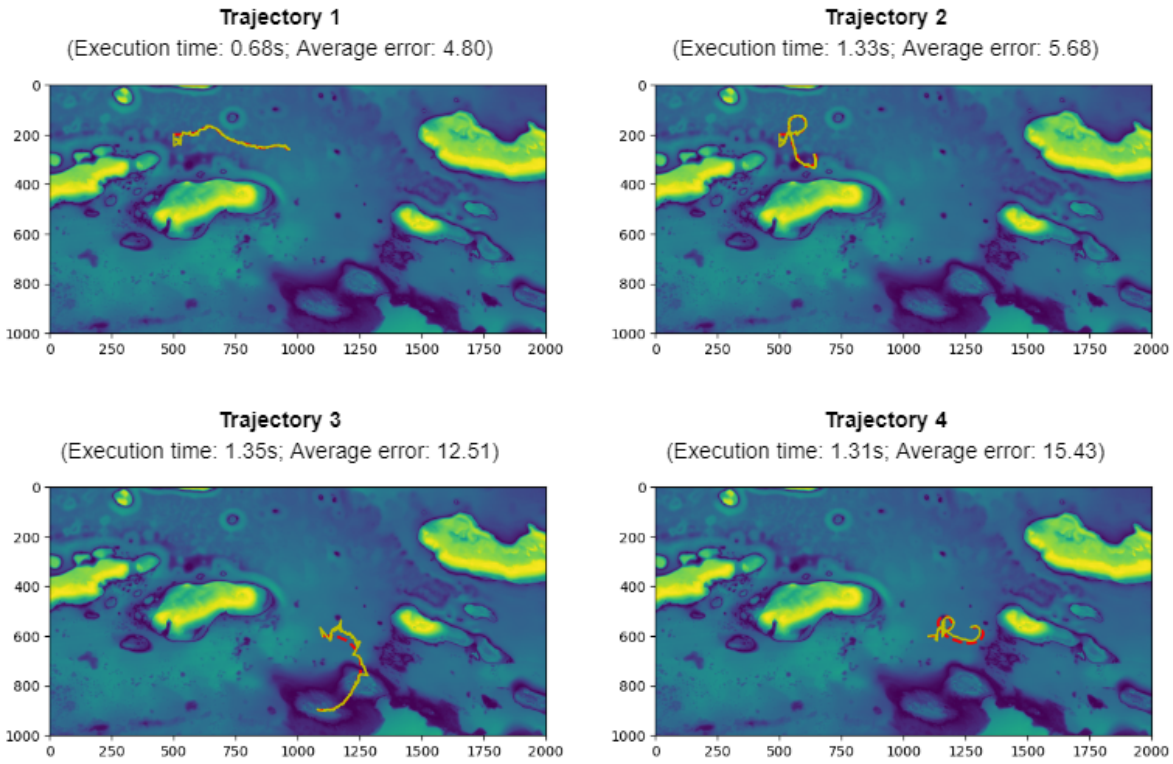

Figure 2. *Example trajectories generated by forward simulating the AUV dynamics model with thruster inputs set to 0.5 and a rudder inputs from a randomly generated function are displayed in red. Timing and error results from filtering with the GPU implementation with N=100000 are displayed in yellow.*

The particle filters generally performed better with trajectories which crossed over regions with more significant variations in depth (e.g., 2) than in those without (e.g. 4). Although Trajectory 3 crosses over significant variations in depth, those variations occur later in the

trajectory (the AUV starts in the middle of the map), which causes significant estimation error in the trajectory before those variations are reached. In contrast, although trajectory 1 (which starts on the left and moves to the right) crosses over relatively fewer regions with significant variation, its average error is significantly less than that of Trajectory 3's because there are a lot of concentrated regions of large depth changes near the start. These early features help the filter converge on the correct state estimate more quickly, which in turn leads to less overall error.

For the following experiments, each of the particle filter implementations used the simulated set of measurements generated using Trajectory 3 (illustrated in Figure 2) to estimate the AUV's trajectory, which is subsequently compared to the ground-truth estimate. The initial particles for filtering were generated from a prior belief state, which was encoded using the parameters init_pos_η and init_rot_η. These parameters represent the amplitude of the gaussian noise added to the starting position and rotation, respectively, which encode uncertainty in the initial belief state. To benchmark the particle filters' performance for the challenge of re-localization, these parameters were set to be relatively large compared to the map size and vehicle dynamics.

The following parameters were used for benchmarking estimation error and execution time:

- `start_x = 1100`: Initial x position in map
- `start_y = 600`: Initial y position in map
- `start_θ = 0`: Initial heading in map
- `init_pos_η = 100`: Amplitude of initial translational uncertainty
- `init_rot_η = 1`: Amplitude of initial rotational uncertainty
- `η = 0.2`: Amplitude of measurement noise
- `sim_time = 1000`: Duration of simulation
- `n_steps = 1001`: Number of discrete timesteps to use for simulation
- `ε = 0.0000001`: Minimum likelihood per particle

CPU (library & threaded): # Particles and # Trials													
10	25	50	100	250	500	1K	2.5K	5K	10K	25K	50K	100K	150K
5	5	5	4	3	2	1	1	1	1	-	-	-	-
GPU: # Particles and # Trials													
10	25	50	100	250	500	1K	2.5K	5K	10K	25K	50K	100K	150K
5	5	5	5	5	5	5	5	5	5	5	5	5	5

Table 1

Table delineating number of trials run for each particle sample size

Table 1 lists the number of trials and particles used to benchmark each implementation. The results from these experiments are illustrated in Figure 3. As expected, each of the implementations produced a filtered estimate with approximately the same estimation error given a particular particle sample size. Particle filters approximate the posterior distribution by sampling, and since each sample is computed independently of other samples the filtered results are the same regardless of whether they were computed in a parallel or serial manner.

It's worth noting that with a small number of particles, the variance and magnitude of the estimation error were significantly higher than with a larger number of particles. This is due

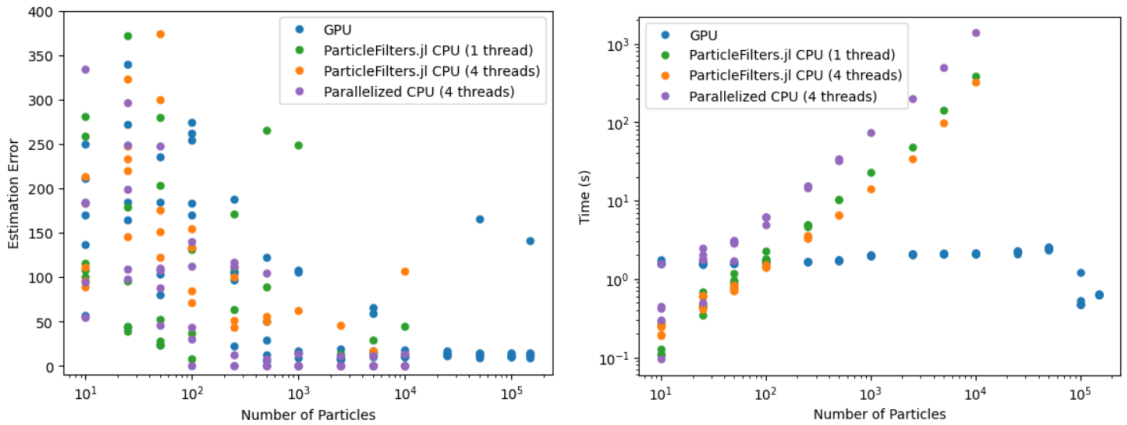

Figure 3. *Comparison of average error after filtering (left) and execution time (right) between GPU, threaded CPU, and ParticleFilters.jl implementation with different particle sample sizes*

to the fact that particle filters depend on having a sufficient number of particles to adequately represent the probability distribution. With a smaller sample size, the filter is less likely to sample the "correct" state, which causes the filter's accuracy to become more random.

For small sample sizes (<100 particles), the ParticleFilters.jl implementation exhibited the best performance in terms of execution times. Although the ParticleFilters.jl package does use threading for the particle filter itself, using 4 threads instead of 1 results in slightly faster execution times. This is likely due to the fact that some of the built-in Julia functions it uses implement multithreading by default.

For larger sample sizes (>100 particles) the GPU implementation offers significantly better performance. While the execution time for the CPU-based implementations increases significantly as the number of particles increase, the execution time for the GPU-based implementation remains fairly constant.[1] For $\geq 10^5$ particles and above, the GPU-based implementation took less time to filter the data than for smaller sample sizes, which was unexpected. It's also worth noting that the variance in execution times between repeated trials is rather small with the GPU-based implementations.

Surprisingly, the threaded particle filter implementation on the CPU proved to be slower than the serialized library implementation. [2] One possible explanation for these results is that the additional overhead for coordinating the computation across multiple threads outweighs the benefits of parallelizing with such a small number of threads, but with the GPU, the computation is parallelized across enough threads such that the overhead becomes worthwhile.

To illustrate the effect of increasing the number of particles used in the filter, Figure 4 displays the initial particle samples and the result after filtering with the number of particles set to different orders of magnitude.

[1] The GPU timing results were double-checked to ensure that the actual execution times were recorded, and not just the kernel set-up process

[2] To verify this difference was caused by threading and not the implementation itself, I first compared the timing of my implementation to that of the library before adding threading as discussed in Section 5. Without threading, both implementations had similar execution times

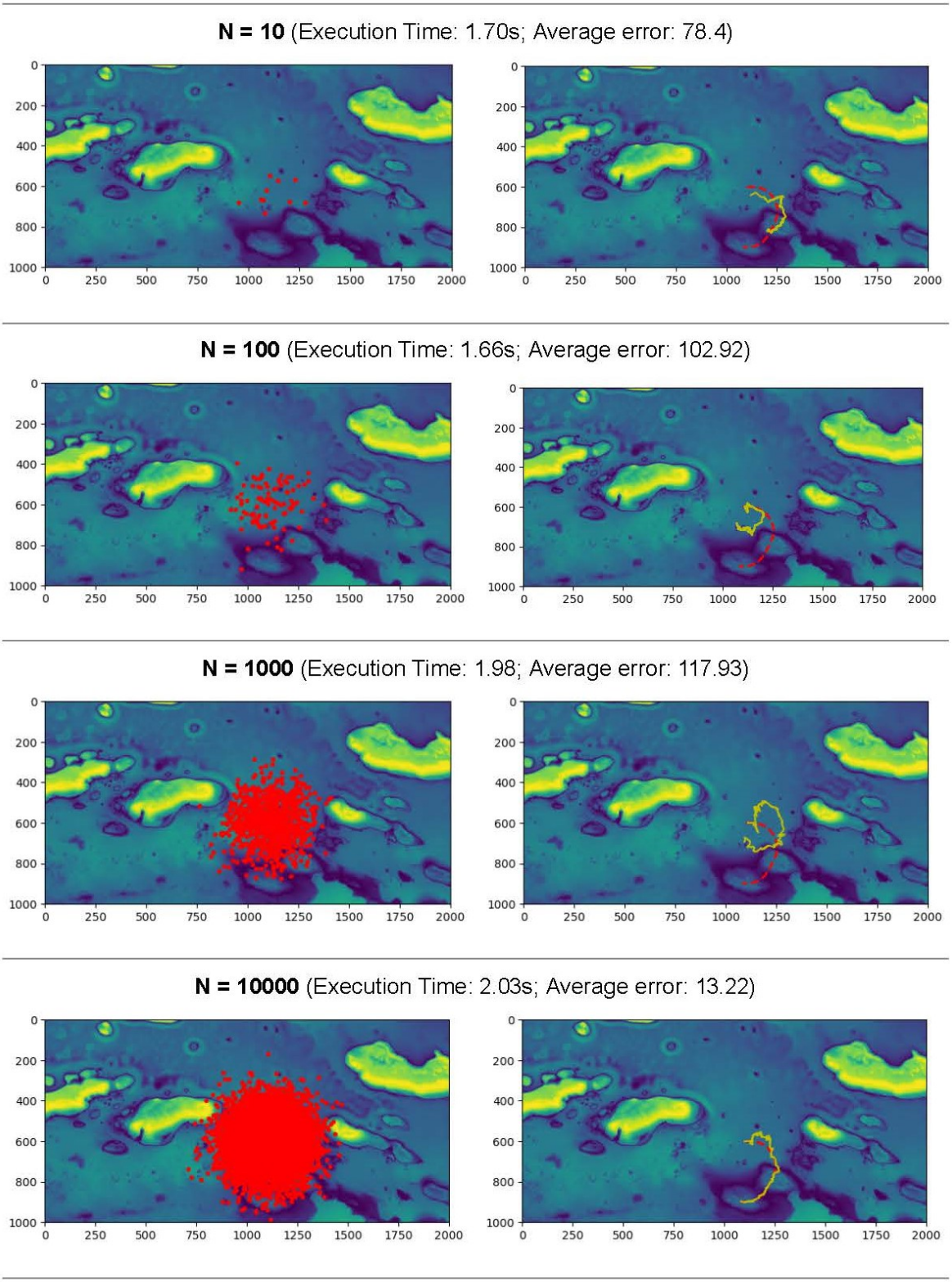

Figure 4. *Visual display of initial particles (left) and filtered results (right) from filtering with different orders of magnitude for sample size*

9. Discussion. The parallelized GPU implementation shows promise in decreasing the execution time of particle filters with a large number of samples, but there are a number of potential improvements that could further improve performance. With the cost of reducing code re-usability, implementing the the particle filter using only one kernel instead of three can potentially reduce the overhead time required for kernel setup. My testing was primarily focused on benchmarking performance based on the particle sample size, but additional testing with other models which have more dimensions or nonlinear noise sources would be needed for a more comprehensive evaluation of the GPU implementation. It's possible that GPU's performance relative to the CPU is dependent on model complexity or data dimensionality, but further testing is needed to evaluate this.

It's interesting to note that the GPU reliably had a shorter execution time with $\geq 10^5$ particles than with fewer. I was interested in evaluating whether this trend held true with even more particles, but ran into memory limits on the GPU with more than 150K particles since my implementation saves the particles generated across all timesteps.[3] Creating an implementation that only keeps track of the average or maximum likelihood particle would significantly reduce the amount of memory that needs to be stored on the GPU, which would enable testing with a larger number of particles.

10. Conclusion. This project implements parallelized CPU and GPU-based particle filter implementations in Julia, and evaluates their performance relative to the serial implementation in the ParticleFilters.jl package. Benchmarking tests with a simplified AUV dynamics model show that a GPU-based implementation outperforms serialized or threaded CPU-based implementations for particle sample sizes greater than 10^2. These results also show that the serialized CPU implementation outperforms the threaded CPU implementation, which suggests that a large number of threads are needed in order to overcome the overhead required for parallelization.

REFERENCES

[1] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, *A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking*, IEEE Transactions on signal processing, 50 (2002), pp. 174–188.

[2] T. Besard, C. Foket, and B. De Sutter, *Effective extensible programming: Unleashing Julia on GPUs*, IEEE Transactions on Parallel and Distributed Systems, (2018), https://doi.org/10.1109/TPDS.2018.2872064, https://arxiv.org/abs/1712.03112.

[3] A. Doucet, N. De Freitas, N. J. Gordon, et al., *Sequential Monte Carlo methods in practice*, vol. 1, Springer, 2001.

[4] Z. Duguid, *Towards basin-scale in-situ characterization of sea-ice using an Autonomous Underwater Glider*, PhD thesis, Massachusetts Institute of Technology, 2020.

[5] JuliaPOMDP, *Particlefilters.jl*. https://github.com/JuliaPOMDP/ParticleFilters.jl, 2021.

[6] S. Thrun, *Particle filters in robotics.*, in UAI, vol. 2, Citeseer, 2002, pp. 511–518.

[3] The ParticleFilters.jl package saves all particles at all timesteps. To ensure a fair comparison, I wanted to ensure my particle filter's functionality closely matched that of the package's