Parallelized Particle Filter for Robotic State Estimation

Kelvin Silva

June 8, 2018

Abstract

Particle Filters used in Robotic Localization, also known as Monte Carlo Localization are a commonly used method to obtain robotic position within an environment where sensor readings are inaccurate. Particle Filters are components of every robotic system which requires autonomous navigation.

Particle Filters are one of the most versatile methods of obtaining state estimation, rivaling Kalman Filters and other methods due to its robustness in dealing with sensor errors (no assumption of Gaussian error distribution), and are relatively easy to implement within systems.

Particle Filters are mostly implemented using serial algorithms. This paper presents a parallelized approach to particle filtering, as well as runtime analysis of both serial and parallelized approaches, and discusses the advantages and disadvantages of both serial and parallelized approaches to particle filtering.

1 Introduction

A particle filter is a type of Monte Carlo algorithm used to solve filtering problems in various areas of signal processing, statistical inference, and other problem areas. In this paper, I introduce an example of a particle filter used to estimate a Robots position in a virtual 2-D world based on its sensor readings, also known as the Monte Carlo Localization method.

One iteration of the Particle Filtering algorithm consists of four main steps:

Initialization: Particle Generation

1. Motion Updates
2. Measurement Updates
3. Particle Resampling

After one iteration a particle cloud will appear around various areas on the map, approximating the Robot's true position. With more iterations the estimation will become more accurate, and inaccurate sensor readings will have little effect on the state (position) estimation. Within a extremely large 2-D world or state space for the particle filter, more particles are needed to increase accuracy, as such the runtime for a serial approach to particle filtering will grow quadratically with N particles.

A parallelized particle filter will have either linear or constant runtime as the number of particles increases. In this approach, CUDA Nvidia architecture was used to implement the parallelized particle filter.

For a more detailed explanation, refer to [3].

2 Method

The approach taken to create the parallel particle filter (PPF) example and code structure was taken from Sebastian Thrun's Udacity course in Artificial Intelligence For Robotics. The python code for the Robot class was translated [2] into a C style struct coupled with data structure specific functions (device CUDA functions) to modify the data. The next step was to take lines 115 - 150 in [2] and translate it into CUDA kernels. Originally three separate kernels were used, one for each

step in the particle filter, but were later combined into one single kernel to take away unnecessary memory transfers.

One area of special mention is the resampling step in the PPF. Thrun's code [2] is serialized and the weight calculation for each particle is an inherently serial algorithm were one iteration of the resampling step is dependent on the previous iteration (specifically lines 140 - 143 in [**?**]) and cannot be easily parallelized.

To fix this problem, I employed the Metropolis resampling method which is easily parallelizable since its weight calculations are based on ratios of weights instead of a running sum. (More detail explained in [4]) [4]

2.1 Parallelized Particle Generation

The host allocates memory on the device to the amount of particles needed. A CUDA kernel is called to initialize this memory according to N particles. Each particle is assigned one thread to become initialized.

2.2 Parallelized Motion Updates

A CUDA kernel is called to assign a thread to each particle which calls the appropriate device function to modify the respective particle and update its motion. More info found here : [3]

Since this step is embarrassingly parallel, no effort will be taken to discuss it.

2.3 Parallelized Measurement Updates

A CUDA kernel is called to assign a thread to each particle which calls the appropriate device function to modify the respective particle and update its sensor readings. More info found here : [3]

Since this step is embarrassingly parallel, no effort will be taken to discuss it.

2.4 Parallelized Resampling

The resampling method is where unlikely particles are to be filtered out and likely particles that represent the Robot's true position stays in memory. The method that I used was the Metropolis resampling algorithm [4]. The Metropolis resampler takes in a parameter B which specifies the number of iterations to be performed before a particle is to be sampled from the set [4]. For each particle K; and iterating B times: the Metropolis resampler calculates a random number U, and chooses random other particle J in the set. It then calculates the ratio of weight of particle J and K and if it is less than U, that particle J will be resampled (it is a particle likely to approximate the Robot's true position).

Pseudocode for Metropolis resampling.

METROPOLIS-ANCESTORS($\mathbf{w} \in [0, \infty)^N, B \in \{1, 2, \ldots\}) \to \{1, \ldots, N\}^N$

```
1  for each i ∈ {1, ..., N}
2      k ← i
3      for n = 1, ..., B
4          u ~ U[0, 1]
5          j ~ U{1, ..., N}
6          if u ≤ w^j / w^k
7              k ← j
8      a^i ← k
9  return a
```

Figure 1: Metropolis Resampling Code Listing [4]

Since there is no data dependency in this algorithm the metropolis resampling approach is perfect for GPU parallelization.

Other methods in [4] include Rejection sampling, which can also replace the Metropolis resampler.

2.4.1 Result

With all major steps of the Particle Filter parallelized, we have attained a fully parallelized particle filter.

3 Run Time Analysis

For the runtime analysis, we are concerned about the trendline or tendency for the algorithm to scale as the number N input increases. In this case, N is the number of particles to be filtered. Extremely complicated robotics applications will require a greater number of particles (large maps, multiple and inaccurate sensor readings), for which a serialized particle filter is unfeasible. A parallelized approach can calculate 32k or more particles depending on the GPU. Throughout my testing, a particle count exceeding 512k incurs significant penalty on the GPU due to scheduling of many more threads than the GPU can handle, as well as memory device penalties. One approach would be to assign more than one particle per thread (which mixes both serialized and parallelized approaches, an interesting and possible implementation for large number of particles).

The Parallelized approach, even if it is not perfectly optimized, is more feasible for 1024, 16k 32k, 512k, 1 million or more particles than any iterative or serialized approach. Any number less than 1024 particles, a serial approach will be easier to implement and integrate into a robotic system.

3.1 Methodology and Results

I tested one iteration of the particle filter (applying the three main steps for all particles only one time) with varying numbers of particles. I chose N particles which would play well with computer memory (powers of 2).

The serialized version of the particle filter that I tested was Sebastian Thrun's Python implementation. The actual runtime of the program does not matter (Python is significantly slower than C), but the progression of the runtime as N increases (the trendline) is the important factor. Thus it doesn't matter if C or Python was used in the runtime comparison.

For each particle in the serialized code, a double for loop is running to do the resampling step. This will incur a $O(N^2)$ runtime penalty on the serialized algorithm.

The parallelized particle filter uses a single for loop for each particle and is parallelized, so all for loops run at the same time, thus setting the runtime to be dependent on constant variables such as the B parameter for the Metropolis resampler, or on the number of sensor readings (not dependent on the input N to the algorithm). In short, the parallelized particle filter algorithm theoretically runs in O(1) time.

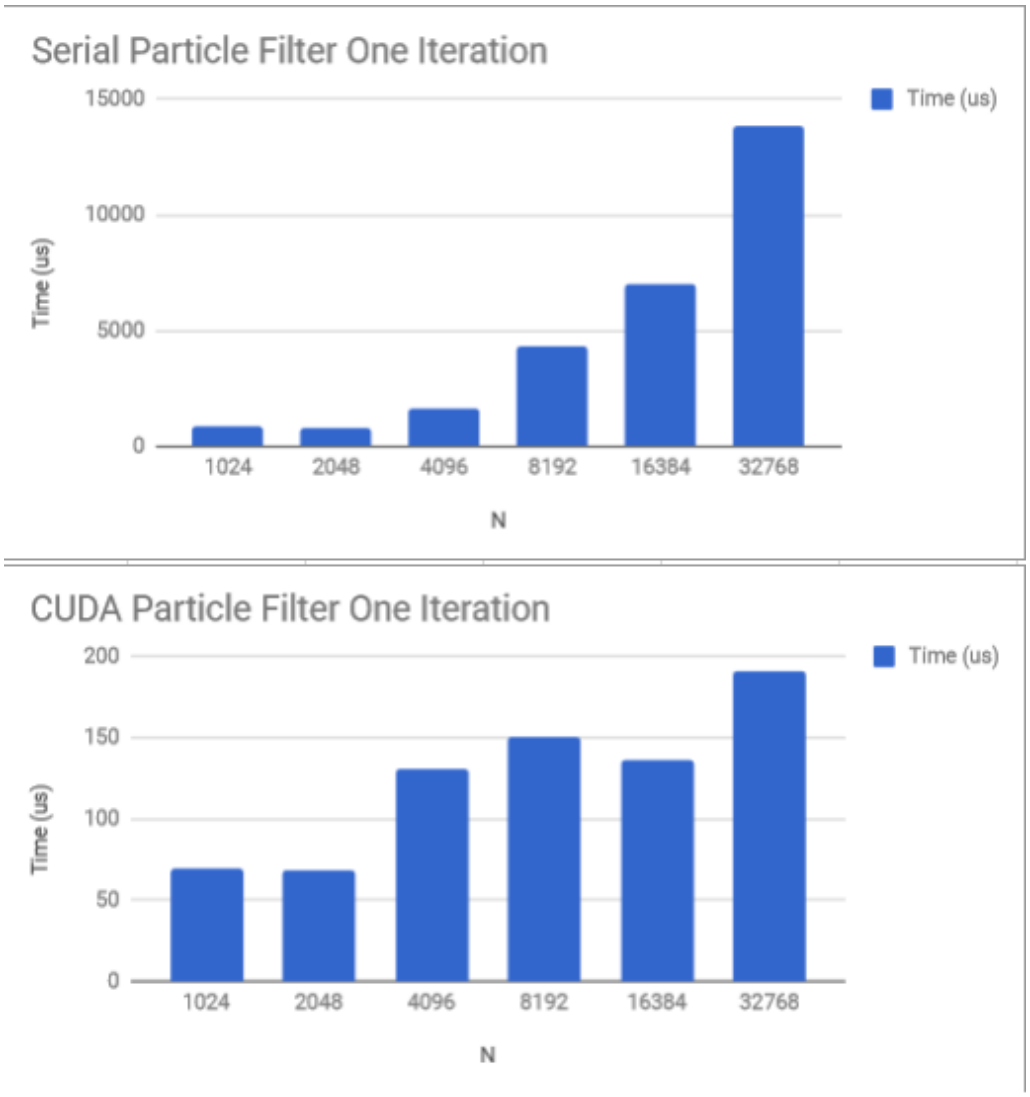

Figure 2: Parallel Constant Time vs. Serial Quadratic Runtime

In a practical sense, the GPU architecture suffers from memory read/write penalties when many calculations are being performed at once, and stored into memory, so we will see some fluctuation in running time as N increases, in either a linear trend or a constant runtime penalty hit.

The runtimes were calculated using system call functions to obtain CPU clock time before and after the particle filter algorithm's execution.

4 Conclusion

From the runtime analysis we see that a Parallelized Particle Filter offers superior performance than its serial implementation. I have also presented a formulation on how to achieve a parallelized particle filter through the introduction of Metropolis based resampling as one of the key components which presents a challenge in attaining a parallelized solution.

Implementing this in a Robotic system is future work which I would like to embark upon, however there are some challenges, such as transferring sensor data onto the device. Currently

this is achieved through the cudaMemcpy function which takes sensor data from the host and copies it to device memory to be used in the parallel particle filter. However, this may prove to be a bottleneck if there are many sensor readings and for a small number of particles, a serial approach may be preferable since data does not have to be transferred from a host to a device (serial approach means that all calculations done on the host).

To conclude, I have presented and exposed some advantages and disadvantages as well as a method to attain a Parallelized Particle filter, which will serve to benefit many applications where an extremely large number of particles may be used where serial implementations do not suffice.

5 Code Listing

The code listing for the CUDA Particle Filter is included in the github directory. [1]

References

[1] Kelvin Silva, *Parallelized Particle Filter CUDA Implementation* https://github.com/kelvinsilva/ams148-gpu-programming-cuda/tree/master/particle-filter-parallel

[2] Sebastian Thrun, *Artificial Intelligence for Robotics Particle Filter Source Code* https://gist.github.com/kelvinsilva/6c19c12c54c5d873aad01f65017cb7d9

[3] Sebastian Thrun, *Particle Filters in Robotics* UAI'02 Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence Pages 511-518

[4] Lawrence M. Murray, Anthony Lee, Pierre E. Jacob, *Parallel Resampling in the Particle Filter* https://www.tandfonline.com/doi/full/10.1080/10618600.2015.1062015