Bayesian Particle Filter Tracking with CUDA

Geoffrey Ulman
CSI702

April 2010

Contents

1 Background

A simple geographic tracking problem traditionally consists of estimating the state of a target using errored observations of some function of the target state. The problem explored here assumes a four-dimensional state space with two Cartesian position dimensions and two velocity dimensions. To simplify the problem, the possibility of false alarms (observations which do not correspond to an actual target) is ignored. Further, all observations are assumed to correspond to a single target of interest (associating observations with the correct target track is an additional complicating concern in multi-target tracking problems).

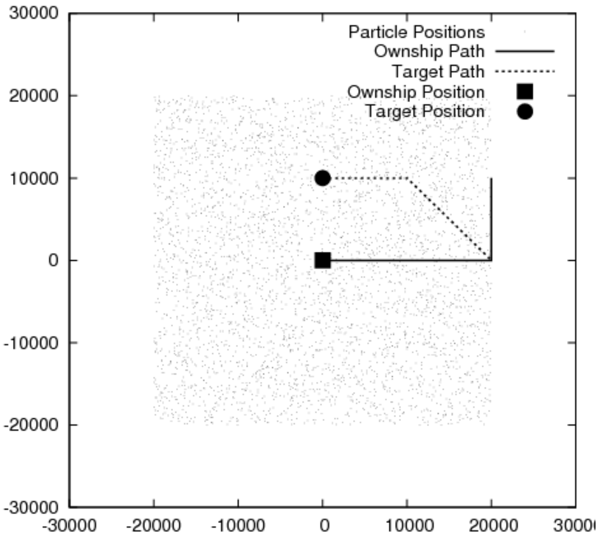

Figure 1: Prior Particle Position Distribution

1.1 Prior Distribution

Employing Bayesian tracking to estimate the true state $x \in S$ of a target in state space S requires a prior distribution $p(x)$. This probability density function describes knowledge about the target's true state x prior to receiving any observations. This prior distribution is generally based on engineering knowledge of the targets and of the sensors used to generate observations.

The simple priors used in this tracking problem assume the sensors have a maximum detection range and thus that targets will not be detected until they are inside that range. The velocity dimension of the target state is also restricted by the known maximum speed of the target being tracked. Together, these form a simple uniform bounded prior distribution on the two position and two velocity state space dimensions. Figure 1 displays the x and y position components of a random sample of particles drawn from this prior with a 20000 meter maximum detection range. Because all particles are initially weighted equally, any particle has an equal likelihood of being the true state of the target.

1.2 Likelihood Functions

Equation 1 describes a likelihood function L for observation $Y = y$ and random variable X which takes on values in a state space S. The function $P(\cdot|x)$ is a probability density function describing the probability of obtaining observations y from the sensor given a known target state x.

However, in Bayesian tracking, the function $P(y|\cdot)$ is far more interesting. This is because once an observation y is received it is fixed and we wish to determine what that observation tells us about the true target state x. The name likelihood function arises from the fact that if $L(y|x_1) > L(y|x_2)$ then the observation y is more likely to have come from a target with state x_1 than a target with state x_2. Is should be noted that unlike $P(\cdot|x)$, the likelihood function usually not a probability density function.[2]

$$L(y|x) = P(Y = y|X = x) \texttt{ for } x \in S \tag{1}$$

The likelihood function, in conjunction with Bayes' rule, tells us how to modify our prior distribution $p(x)$ (discussed in Section 1) to incorporate the information from an observation y. Applying Bayes' rule we arrive at Equation 2. The probability density function $P(\cdot|y)$ is known as the posterior distribution and represents the prior adjusted to reflect the addition of the new information contained in the observation y. This posterior distribution can have further likelihood functions applied to it using Equation 2 to incorporate additional information. This iterative process is known as Bayesian tracking.

$$P(x|y) = \frac{L(y|x)P(x)}{\int L(y|x)P(x)\,dx} \tag{2}$$

Figure 2 shows the posterior distribution after the likelihood for a single azimuth observation is applied to the prior distribution from Figure 1. An azimuth observation is a single angle value indicating that the target is somewhere along the ray starting at the sensor's current position and continuing in the given angle. The likelihood function which we associate with such observations, shown in Equation 3, is a simple Gaussian distribution in azimuth space.[2] Here θ is an azimuth observation, $x \in S$ is an element of the state space, σ is the standard deviation assigned to the observation, and $b(\alpha, x)$ is a function which gives the azimuth from the sensor position α to the position x. Applying this function to each particle gives us new weights for the particles with those particles closer to the observed azimuth receiving higher weight.

$$L(\theta|x) = (2\pi\sigma^2)^{\frac{-1}{2}} e^{\frac{-(\theta - b(\alpha,x))^2}{2\sigma^2}} \tag{3}$$

1.3 Motion Model

In kinematic tracking problems, observations often occur at different times. The existence of velocity components in the state space S indicates that the position components are changing over time. In addition, the possibility that the target might change its velocity must be modeled.

Motion updating can be described much more generally and rigorously, but in our case the step is simple. The target is assumed to make instantaneous velocity adjustments with an exponentially distributed mean time between adjustments. During periods where no adjustment is made, the target travels at a constant velocity. If no observations are received for an extended period of

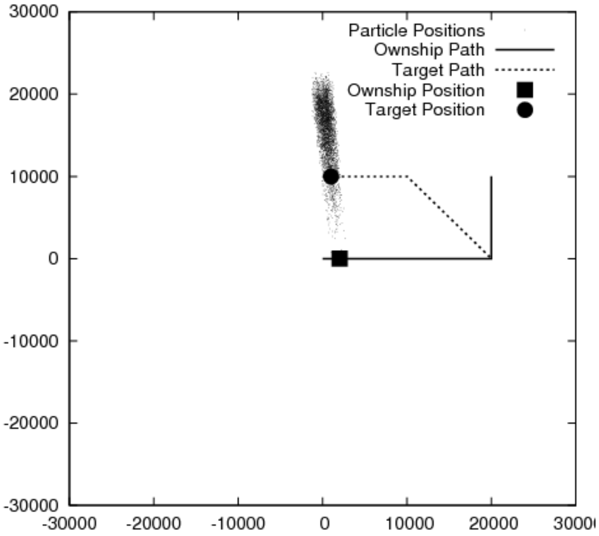

Figure 2: Posterior Particle Position Distribution after Azimuth Observation

time, as has happened in Figure 3, the particles will drift apart on their constant velocity paths and the posterior distribution will become more diffuse to indicate our increasing uncertainty about the current location of the target.

2 Design

Most portions of the Bayesian tracking recursion are embarrassingly parallel operations which map directly to CUDA without much trouble. Initialization of particle positions, time updating, and information updating are all operations on a single particle and are independent of each other particle. However, the CUDA implementation other portions of the algorithm, including particle weight summation (and more generally, particle resampling), and random number generation, provide interesting challenges.

2.1 Resampling

As the information about the target's state contained in observations is incorporated into the prior distribution using appropriate likelihood functions, some particles will match very poorly with the observations and have very low weight as a result. Eventually, keeping such particles around serves very little purpose, since the Bayesian tracker has already indicated that they matches poorly with the observations and are thus unlikely to represent the true target state. Resampling is a technique for remedying this situation by periodically replacing particles with low weight with slightly perturbed copies of particles with high

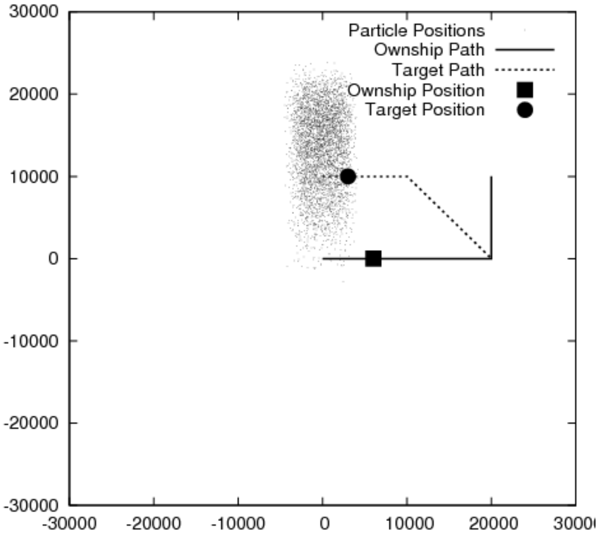

Figure 3: Posterior Particle Position Distribution after Azimuth Observation and Motion Update

weight.

2.1.1 First Resampling Implementation

$$C = \frac{n}{\sum_{i=0}^{n} w_i}$$
$$\overline{w}_i = C w_i \tag{4}$$

$$\hat{w}_i = \text{floor}(C \sum_{j=0}^{i} \overline{w}_j) \tag{5}$$

When copying particles, we want each particle to have a likelihood of being copied proportional to its weight. Equation 4 describes the transformation applied to particle weights w_i to obtain the number of copies to make of each particle. However, to implement the copy operation in parallel in CUDA, each particle must also know where in global memory its copies should be placed.

Equation 5 solves this problem by calculating a cumulative sum of weights. With the cumulative sum in hand, thread i simply makes $\hat{w}_i - \hat{w}_{i-1}$ copies of the particle at index i and places them in the contiguous block of memory from $\hat{w}_{i-1}$ to $\hat{w}_i$.

Unfortunately, as indicated by Figure 4, this approach is very slow on GPU hardware because of the large amount of uncoalesced memory access which it must perform. Further, most threads sit idle (those with low enough weights

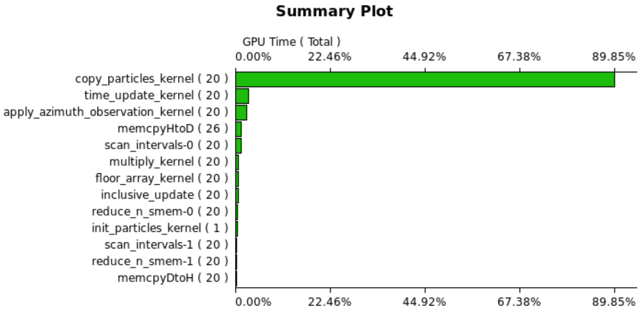

Figure 4: CUDA Visual Profiler Version 1 Results

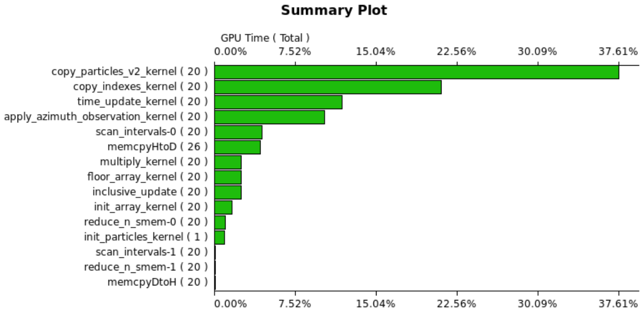

Figure 5: CUDA Visual Profiler Version 2 Results

that they make 0 copies of themselves) while a very few perform all the memory copies.

2.1.2 Second Resampling Implementation

The second resampling algorithm implementation improves on the first dividing the work of duplicating particles more evenly among the threads. Instead of the copy-from thread making $\hat{w}_i - \hat{w}_{i-1}$ copies of itself, it simply overwrites $\hat{w}_{i-1}$ through $\hat{w}_i$ with its index i. Then, in a separate kernel, each copy-to thread duplicates the particle at the index stored in $\hat{w}_i$ and perturbs it. This approach does not solve the uncoalesed memory access issues, but by evenly dividing the work of copying and perturbing among all threads, it still produces a significant speedup as shown in Figure 5.

2.1.3 Third Resampling Implementation

The third resampling algorithm optimization was based on the observation that once the copy-from indexes had been stored in the weight array at the copy-to index, the weights were equivalent to the gather map used in the thrust::gather

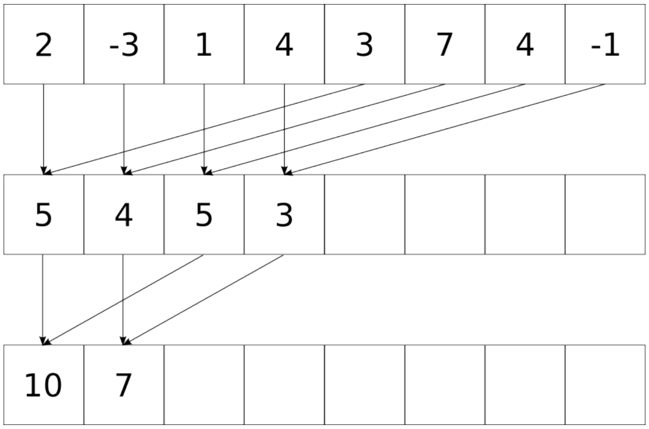

Figure 6: Inner-block Reduction In Shared Memory Example

algorithm. [6] thrust::gather copies items from a source array $s[i]$ into a destination array $d[i]$ using a map array $m[i]$ such that $s[i] = d[m[i]]$.

Surprisingly, especially given the amazing efficiency of thrust's reduction and scan algorithms, thrust::gather provided no speedup over my custom implementation from Section 2.1.2. The inefficiency of the inherently uncoalesced nature of the memory access pattern must simply overwhelm any subtle performance tweaks that thrust::gather provides.

2.2 Parallel Reduction

The resampling calculations in Equation 4 and 5 require calculating the sum and cumulative sum of an array of weights. The thrust library provides prewritten parallel algorithms for common CUDA tasks, including parallel reduction through repeated application of a binary reduction function.[6]

However, before using thrust, a custom parallel array summation implementation was written based on the NVIDIA white paper on the subject included in the CUDA SDK.[3] Understanding the overall summation algorithm requires first understanding the basic shared memory summation algorithm for a single block. Each block copies its values from global memory to shared memory and performs a simple fan-in reduction in shared memory. As Figure 6 indicates, theads add their partial sums together in a tree pattern until a single partial sum for the entire block remains. Each iteration is synchronized using `__syncthreads()`.

However, the loop construct that controls this iteration is expensive, as are the `__syncthreads()` calls. Fortunately, because CUDA threads are grouped into *warps* of 32 threads which execute commands simultaneously, substantial additional savings are possible once the number of partial sums in shared memory drops below 32. By stopping the reduction iteration and unrolling the final iterations, we can avoid synchronization for the iterations where all the summa-

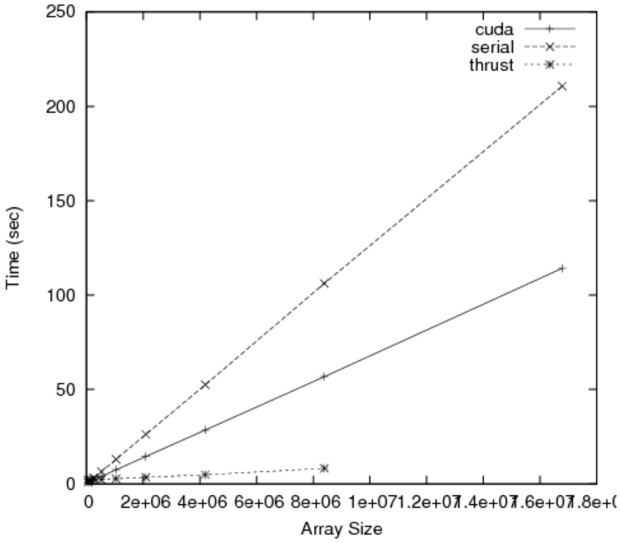

Figure 7: Array Summation Algorithm Performance

tions are occurring within a single warp.

As each block completes its parallel summation, it writes its sum to an array in global memory. Thus, after the first iteration, global memory will contain one partial sum for each block. To sum these values, the kernel is called iteratively until fewer partial sums remain than the number of threads in a single block. At this point, the kernel is called one last time with a single block which writes the final overall sum to global memory.

However, as Figure 7 indicates, writing complex parallel algorithms efficiently is very difficult in CUDA. For the largest arrays tested, the custom CUDA implementation was only about twice as fast as the serial implementation whereas the thrust implementation was 12.9 times faster. These results highlight cuda's sensitivity to subtle factors like uncoalesced memory access, shared memory bank conflicts (threads from multiple warps accessing the same sections of shared memory), and expensive gpu operations which can have significant impact on performance.[5]

2.3 Random Number Generation

Particle filter tracking is a stochastic process: as particles are time updated, they maneuver randomly according to their motion model; as particles are resampled, their replacements are randomly perturbed copies of existing particles. Thus, performing particle filter tracking using CUDA required generating random numbers efficiently on the GPU. The thrust library provides random number generation capabilities and the CUDA SDK provides a parallel MersenneTwister example. However, the random number generator implementation used for this problem is a much simpler linear congruential generator which relies on independent seeds stored for each particle.

$$X_{n+1} = (aX_n + c \mod m) \tag{6}$$

With properly chosen a, c and m values linear congruential generator can provide sufficiently random values for particle filtering.[7] The values I used are those used by the java.util.Random class. This approach is attractive because it is theoretically very fast and requires very little state, allowing each particle to generate its own independent stream of random values.

The recurrence relation in Equation 6 relies on the modulus operator, which is extremely slow on NVIDA hardware.[3] However, this recurrence can be rewritten using the much faster shift and bitwise mask operations, avoiding the significant modulus performance hit.

2.4 Effective Particle Count

When resampling is performed after each observation, every particle always has an equal weight and therefore an equal chance of representing the true target state. However, resampling every iteration is not necessary.

Formula 7 calculates the effective particle count N_{eff} from a weight array w_i.[8] This formula is best understood by considering two extreme cases. When all particles have the same weight (like after resampling is performed), we would expect the effective particle count N_{eff} to equal the actual particle count n. In this case each normalized paricle weight $\overline{w}_i$ will equal $\frac{1}{n}$. $Thus, \sum_{i=1}^{n} \overline{w}_i^2$ will equal $\frac{n}{n^2}$ and n_{eff} equals n as expected.

Now consider the case where one particle has weight 1 and all other particles have weight 0. In this case we expect the effective particle count to be 1 and it is trivial to show that this is indeed the case.

Because resampling is the slowest portion of the parallel particle filter code, as discussed in Section 2.1, it makes sense to minimize the number of times it must be performed. The effective particle count metric provides a way to make that determination. Every iteration the effective particle count can be calculated relatively cheaply, and resampling can be performed only when the effective particle count falls below a given threshold.

$$\overline{w}_i = \frac{w_i}{\sum_{i=1}^{n} w_i}$$

$$N_{eff} = \frac{1}{\sum_{i=1}^{n} \overline{w}_i^2} \tag{7}$$

3 Performance

The peak performance improvement using CUDA over the serial implementation was approximately 20 times. While this is nowhere near the theoretical performance gains which CUDA can achieve, it is respectable given the inherently uncoalesed nature of the memory accesses which the resampling strategy performs.

Figure 8 displays a log-log plot of the overall execution time for the serial implementation and the CUDA implementation at various stages of optimization. Note that the CUDA timing results stop at problem sizes smaller than the serial results because of memory constraints on the Quadro FX 1700M GPU used to run the timing tests.

Both the serial and CUDA implementations scale roughly linearly with particle count. This makes intuitive sense because most operations are independent of other particles. The parts of the algorithm that do depend on other particles, like summation and cumulative summation, are performed once for all particles and scale linearly in time complexity with particle count.

The speedup factors in Figure 9 acheived by the version one, version two, and version two using effective particle count (EPC) all increase with problem size. Because GPUs do not have cached memory heirarchies to preload data, they rely on data parallelism instead of caching to hide memory latecncy. With smaller problem sets, there are fewer blocks which reduces the ability of the GPU to swap out blocks that are stalled waiting for memory access.

It should also be noted that the CUDA version two test using EPC to determine when resampling is needed is not really a fair comparison. Its speedup factor was calculated compared to a serial problem with the same particle count. However, without resampling every iteration the results will be less accurate (although quantifying exactly how much less accurate is a difficult problem beyond the scope of this paper).

4 Challenges

Text...

References

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall PTR, New Jersey, 2009.

[2] Stone, Barlow, and Corwin, *Bayesian Multiple Target Tracking*, Artech House, Boston, 1999.

[3] Harris, Mark, *Optimizing Parallel Reduction in CUDA*, NVIDIA Developer Technology
http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[4] Volume I: Introduction to CUDA Programming
http://www.nvidia.com/docs/IO/47904/VolumeI.pdf

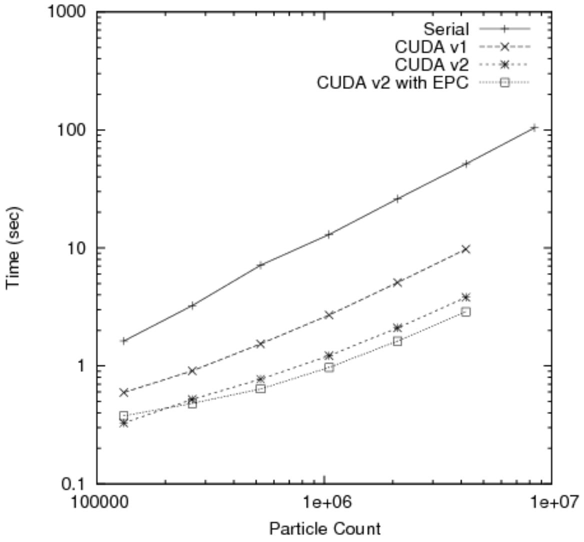

Figure 8: Timing Results with 40 Observations and Variable Particle Count

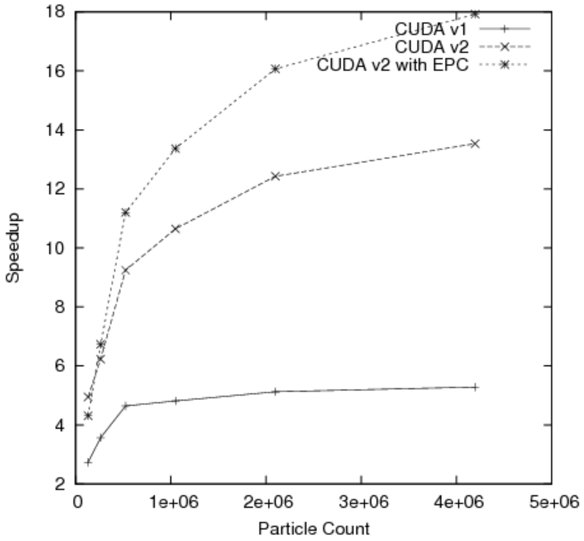

Figure 9: CUDA Speedup Results with 40 Observations and Variable Particle Count

[5] CUDA Best Practices Guide – CUDA 2.2
http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[6] Thrust C++ Template Library for CUDA
http://code.google.com/p/thrust/

[7] Linear Congruential Generator
http://en.wikipedia.org/wiki/Linear_congruential_generator

[8] Particle Filter
http://en.wikipedia.org/wiki/Particle_filter