Technical University of Crete, Greece
Department of Electronic and Computer Engineering

Particle Localization
CUDA implementation

Nikolaos Kofinas

Chania, July 2012

Chapter 1

Introduction

In this project we worked on the problem of localization in the 3D space. More specifically, we worked on the problem from the view of RoboCup competition. The team Kouretes from Technical University of Crete has been participating in this competition for more than 5 years. In this competition, the robots are programmed to play football in a field with dimensions 9×6 meters. Furthermore, the teams have to solve a lot of robotic and artificial intelligence problems, one of which is to find the accurate position of the robot in the field. This problem is the localization problem.

More specifically, the position of the robot can be described by the variables x, y,and θ. The posterior distribution of our state space in unknown, thus we must create a model of it using some parameters. The most common algorithm for this problem is the particle filtering. This algorithm tries to find an approximate model of the posterior priority, using a set of samples, the particles.

The particle filter has a lot of variations, but in our team we are using the Sequential Importance Re-sampling (SIR) method. This method has four distinct steps, the prediction, the update, the normalization and the re-sampling step. Also, in our implementation we have implemented one more step to find the average particle, because we use it as our estimated position.

The RoboCup environment is a real-time environment, thus, our implementation must be fast and not take more time than the threshold that we have set. For those reasons, the team's code has a lot of optimizations like loop unrolling and loop merges. The code that is the baseline of this project is the same that the team uses, and its goal is to find if

the CUDA implementation can help our team,in the future, to have a faster localization with better approximation of the real position.

Brief abstract of the project

The particle filter is the most known solution for the localization problem in the RoboCup community. Most teams that use the particle localization, use a small number of particles (the average is in the range of 80-120 particles). Kouretes in the current implementation are using 120 particles, but they have only two landmarks, the right and the left goalpost. Other teams have more landmarks, for example the lines, but this is a vision problem. The only problem with the one landmark is that the position has a lot of noise if we don't get an observation for a period of time.

The code, that has been developed for the team, is implemented in c++ and it is fully optimized for real time execution. Some of the algorithm's steps mentioned above, have been nested to minimize the loops and gain all we can get from the cache of the cpu. The code is single-threaded because we don't have the computing power to support a lot of threads. Thus, the CUDA implementation will be compared with the above code, but it will run on a computer and not on the robot, in order to obtain similar timing results.

First steps

The goal of the project is to implement in CUDA the particle filter algorithm for localization in the 3D-space. Before that, some other things must be done.

First of all, we made the localization code to run locally, and outside the Kouretes code architecture. This step was not so easy as someone can image, because the code includes libraries from the architecture, so we had to separate the functions that we used from these libraries. The input of the localization code was coming from some other modules inside the architecture (like the Vision module) in form of Google protobuff messages. For the local run, the code has been rewritten to read the input from log files and not from those messages.

Secondly, we had to keep logs from a real world scenario for later testing and benchmarking. The robot was free to move inside the field for some time and we logged all the incoming observation along with some other useful data. During those runs, because the

team does not have a ground truth system for the validation of the localization output, we wrote down the movements of the robot inside the field for later validation of the algorithm.

SIR Particle filtering algorithm

As we mentioned before, the algorithm has the following four steps, prediction, update, normalization and re-sampling. These steps are executed in this order and they can not be overlapped.

The prediction step

The prediction step is pretty simple. For every particle in the particle filter, we update its position (x, y and θ) according to the odometry of the robot. This step tries to predict the position of the robot after a movement, regardless if an observation is taken. The odometry is quite noisy, thus, we must add random noise in the movement of our particles. After this step, every particle will have been moved to a new position, near the old one. After some prediction steps, we will see our particle form a cloud, because of the noise that we put in every one of them. In figure 1 we present the pseudo-code for the prediction step.

Algorithm 1 The prediction step of the SIR particle filter

for $i = 0 \rightarrow particlesNumber$ **do**
  $particle[i].x \leftarrow particle[i].x + odometry.x + noise$
  $particle[i].y \leftarrow particle[i].y + odometry.y + noise$
  $particle[i].phi \leftarrow particle[i].phi + odometry.phi + noise$
end for

The update step

The update step is more complicated and mathematical unlike the prediction step. Its goal is to weight each particle given the current observation that was produced by the vision module (in our case goalposts). In this step, we are trying to find which particles are better matched in the theoretical position of the robot, in order to have this observation.

The best matched particles will eventually have bigger weight compared with the rest. After this step, the particles will no longer have equal weights and the weights will not sum to one. The pseudo-code 1 is pretty simple as we can see.

Algorithm 2 The update step of the SIR particle filter

```
for i = 0 → particles do
    particle[i].weight ← 0
    for o = 0 → observations do
        for l = 0 → landmarks do
            particle[i].weight ← updateWeight (Obs[o], Landmark[l], particle[i])
        end for
    end for
end for
```

The normalization step

This is the simplest step of the algorithm. As we can see in the pseudo-code 1, in this step, each weight is divided by the total sum of all the weights in order for the weights to be in the range 0 to 1. This step is basic, because the re-sampling is not necessary to be done if the weights' sum is not far from 1. The reason for the sparse re-sampling is time saving.

Algorithm 3 The normalization step of the SIR particle filter

```
totalWeight ← 0
for i = 0 → particles do
    totalWeight ← totalWeight + particle[i].weight;
end for
for i = 0 → particles do
    particle[i].weight ← particle[i].weight/totalWeight;
end for
```

The re-sampling step

The re-sampling step is the most complex step of the particle filter algorithm. Its main goal is to display more particles in the regions where there were particles with big weights

after the update and to lessen the particles from the regions that are now having small weight. After this step, the number of particles will be equal to the ones of the previous step, except that they will have been gathered in the region with the higher weight. There are various implementations of the re-sampling step, but the smartest one is the roulette re-sampling method. This method begins with initial step 1/numberOfParticles and an initial globalWeight equal to this step. Subsequently, we start a loop all over the particles. For every new particle we process a cumsum is updated with the current weight of the visited particle. While the cumsum is less than the value of globalWeight, it moves the current particle into a list with new particles and globalWeight increases by one step. If globalWeight surpasses the cumsum, we move to the next particle. The pseudocode for this step is shown in the pseudo-code 1

Algorithm 4 The re-sampling step of the SIR particle filter

$cumsum \leftarrow particle[0].weight$;
$step \leftarrow 1/particles$;
$r \leftarrow 1/particles$;
$m \leftarrow 0$;
for $i = 0 \rightarrow particles$ **do**
  while $r \geq cumsum$ **do**
    $m++$;
    $cumsum \leftarrow cumsum + particle[m].weight$;
  end while
  $temp[i] \leftarrow particle[m]$;
  $temp[i].weight \leftarrow 1/particles$;
end for
for $i = 0 \rightarrow particles$ **do**
  $particle[i] \leftarrow temp[i]$;
end for

Kouretes implementation

Kouretes have exploited parts of the algorithm's code in order to achieve better efficiency. In the first place, in real-time environment our code runs periodically, thus, we are always interested for the worst case scenario, so in that case we choose to do re-sampling

whenever the weights change, even if this change is not great. Furthermore, we have embedded the normalization step in the roulette re-sampling so as to avoid a surpassing over all the particles, as we are not interested in normalizing their weights.

CUDA implementation

As previously seen, the algorithm is broken into 4 steps but our implementation is divided in 3, so we dealt with each one separately and we attempted to parallelize each step before getting to the next. As we are going to see next, the 2 of the 3 steps are fully parallelized, but the third one is partially parallelized.

Phase one: prediction step

The implementation of the prediction step was quite simple, because we only needed a kernel that would start a thread for each one of the particles, and that would make a mathematical calculation for it based on a specific input and random noise. In the first implementation, random times were being created in the CPU serially and being copied with a cudamemcpy on the GPU which undertook the mathematical calculations. As we observed later, memcopy was very expensive and an order of magnitude larger than the time the kernel was executed.

After we found that we encounter a bottleneck through this implementation, we searched for creating random numbers on the GPU. That became applicable with the use of Cudarand library, which is being provided by Nvidia. The code that is being used to produce random numbers based on this library is not trivial, and required a lot of search in the CUDA documentation.

Executing the code with the add of cudarand, we detected a big improvement in execution time, but we needed to make it faster. We also observed that the odometry data was the input of kernel and thought if it was more efficient instead of them being passed as an input, to be set to constant variables and simply do a memcpy before the algorithm's execution. The result was to observe a small speedup, but not as impressive as the previous one.

Phase two: updating step

The updating step was more tricky than the above one. In this step we set up a kernel that, like in the prediction step, will start a thread for each one of the particles, and then it will do some calculations to find out the new weight of the particle. The difficulty of this step is that we have complex mathematical calculations, and we tried to find the best implementation of those calculations to get as much speedup as we can.

As we mentioned before, the landmarks that we can observe from the vision module are the left and the right goalposts. Sometimes we get an ambiguous observation, because the vision cannot understand if the goalpost is left or right. In order to handle this situation, we need a different kernel for the ambiguous update step, but the calculations are almost the same with the regular update step. Also, we can get and observe both goalposts, but this situation is being handled from the normal update kernel.

To do the calculations, the kernel needs the observation and as we learned from the previous step, it is better to copy the input to a constant variable and then all threads can use it. Furthermore, we saw that the bottleneck of our code was the atan2 and the exp (exponential) functions. Those two functions are complex and in the code that we have for the CPU we use some "fast" implemented functions. These two functions are faster than the ones implemented inside the "math.h" library but as we found out, on the GPU they are slower than the ones that nVidia provides. So, we changed the function back to the default ones.

Another test that we have done is to find out if the call of two kernels is faster than to execute an "if" statement that is "true" or "false" for all the threads. We found out that the syscall that starts a kernel is by far slower than an if statement.

The last test was to find out if we have any speedup when we use the fast, but not accurate, implemented math functions that nVidia provides. The updating step does not need the best accuracy that we can have, thus the fast functions will have no impact in the accuracy of our algorithm. The time results were too disappointing, the fast functions did not gave us any speedup at all and we never found out the reasons for this lack of speedup.endiameso

Phase three: re-sampling step

This step was by far the most difficult, because the slowest code in this step has no parallelism, as we mentioned before. Thus, we focused on some internal steps that can be done with parallelism. These internal steps were the calculation of the totalWeight and the calculation of the mean particle (we put this step inside res-sampling for simple implementation).

For both steps, we have to find the sum of all particles (in the totalWeight step the sum of all the weights and in the mean step the sum of the x, y, and phi variables). As we saw in the class, there is a very good implementation for such problems and we used this implementation for our problem too. Some specifications for this problem is that we use only kernels with block size "1" and thread size "512", thus the number of particles for our implementation must be a multiple of "512", but with small modifications, we can make the code to run for a smaller number of particles.

Now this step works as following:

1. Calculate the totalWeight on the GPU
2. Memcopy the particles back to CPU
3. Do the res-sampling in the CPU
4. Memcopy the particles to GPU
5. Calculate the mean particle

As we can observe, the two memcopies are the bottleneck of this step, thus we tried to implement the intermediate step in a single thread inside the GPU to avoid those memcopies, but the results were bad because of the lack of a caching system on the GPU.

Phase four: speedups

After the implementation of all the steps of the algorithm, we tried to find out if we can do something more to get further speedup. We found some minor code lines which when changed,result in a small speedup. An example of that was in the res-sampling step,

when we copied the new particles back to the original particle vector and then to GPU. As we observed, the memcopy that moves the particles from the new vector to the old one is useless and, thus, we can memcopy the new vector to the GPU from the start and the result of that was a lesser memcopy and a small speedup.

Another small improvement was to put the mean step outside of the re-sampling step, because we desired that the execution of the re-sampling step is necessary only when we have a new observation. This modification does not change the worst case execution of the algorithm, but it changes the mean execution time, because it is unlikely to have a new observation in every execution of the localization.

Results

The system on which we have tested our implementation has an Intel core2duo processor at 3GHz, 2GB of DDR2 Ram and an nVidia Ge-force GT 220 GPU with 1024MB memory. More specifically, the GPU has 1.2 computing capability and it includes 48 CUDA Cores, and the GPU clock is 1380Mhz.

We ran the code, both for CPU and GPU, with a looped input, because we did not want to check the accuracy of the localization, but the worst time. Thus, we took three inputs that have two observations and we provided localization continuously with those inputs. All the results that we present in the Table 1.1 1.6 are from the fake set of inputs. We want to run the localization process twenty five times per second, thus each run must take less than 40 ms. The problem is that the resources of the machine are not only for us. There are some other heavy tasks that run and they need a lot of computer time. Thus, we have two limits, one soft and one hard. The soft limit is that we want to have an execution time below 25 ms and the hard is set to be 15 ms. We must choose the number of particles correctly, because the localization must have execution time under the soft limit and it is preferable to be under the hard limit. In the Figure 1.1 we can see the total execution times, for a variation of particle numbers, for both CPU and GPU implementations and in Figure 1.2 we can see the overall speedup of our implementation. As we can see, the GPU can handle a lot more particles than the CPU. The total number of inputs that we used in order to extract those numbers are 10000 inputs.

Table 1.1: Execution times in the CPU

Particles	**Prediction** (ms)	**Update** (ms)	**Re-sampling** (ms)	**Find average** (ms)
512	0.168919	0.532831	0.00396432	0.0395371
1024	0.322394	1.06459	0.0168484	0.0790743
2048	0.65444	2.15501	0.0198216	0.166827
4096	1.29826	4.28525	0.0475719	0.322083
8192	2.62066	8.5845	0.086224	0.646095
16384	5.3388	17.4198	0.18335	1.30569
32768	10.6023	34.5425	0.371655	2.62295
65536	20.7519	71.423	0.890981	5.12343
131072	41.7674	137.467	2.83251	10.3905
262144	82.5637	271.758	6.98216	20.4976

Table 1.2: Execution times on the GPU of the final implementation

Particles	**Predict** (ms)	**Update** (ms)	**Re-sample** (ms)	**Find average** (ms)
512	0.0704633	0.0667384	0.169475	0.0867888
1024	0.103282	0.0721206	0.11001	0.0684667
2048	0.170849	0.080732	0.130823	0.0838959
4096	0.309846	0.0850377	0.184341	0.112825
8192	0.57529	0.134553	0.282458	0.145612
16384	1.12548	0.199139	0.483647	0.243973
32768	2.22008	0.344456	0.919722	0.437801
65536	4.42761	0.587729	1.87611	0.817743
131072	8.91023	1.13132	4.27156	1.22179
262144	17.7288	2.03552	10.663	2.01543

Discussion of the results

From the tables we can see that all the functions that we implement on the GPU are a lot faster, compared to the CPU base functions. The only exception is the re-sampling

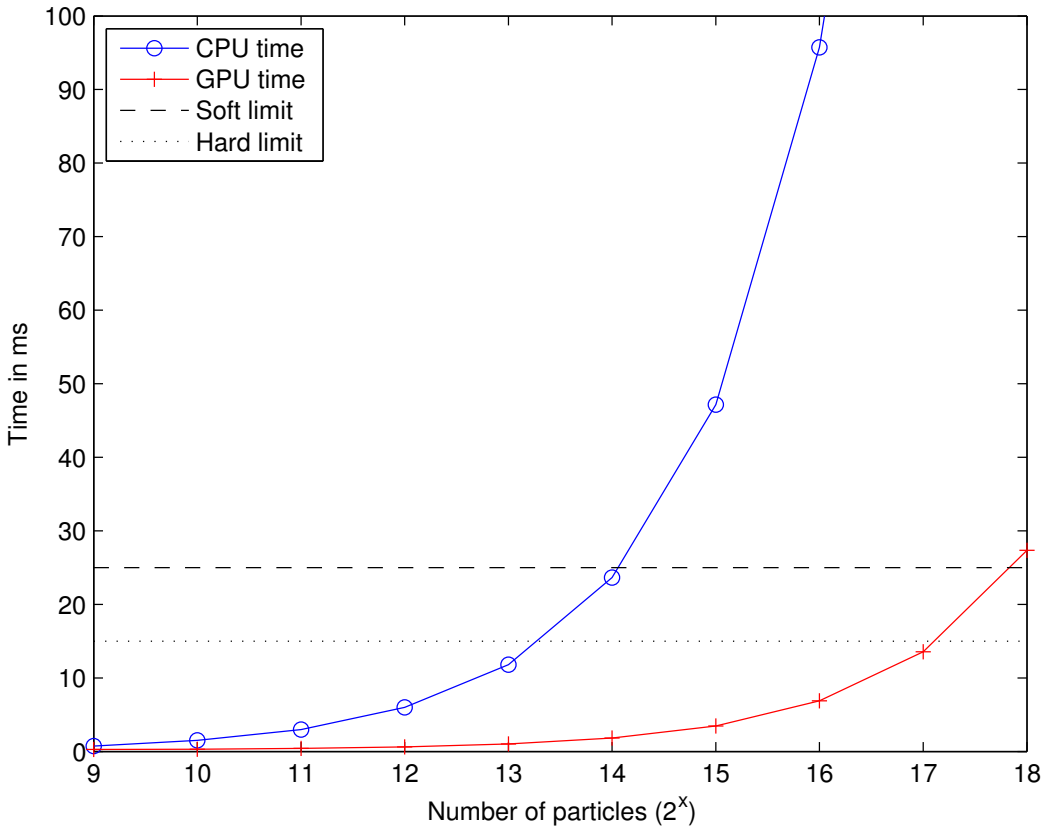

Figure 1.1: Total time of the localization step

function, because it can not run on the GPU,so we must retrieve all the particles from the GPU, run this function and then move the particles back to the GPU. The result of that, is to have a negative speedup in this function, but this negative speedup is nothing compared to the positive speedup that we got from the prediction and update steps.

As we can assume, the system on which we benchmarked our implementation is far better than the system that will be on-board in any small robot that works with batteries. Thus, we must scale our results to find out th profit that we would have on a smaller system. An on-board system may have a two cored CPU with a clock near 1 GHz and a small GPU processor like the ones that are now in every smart-phone.

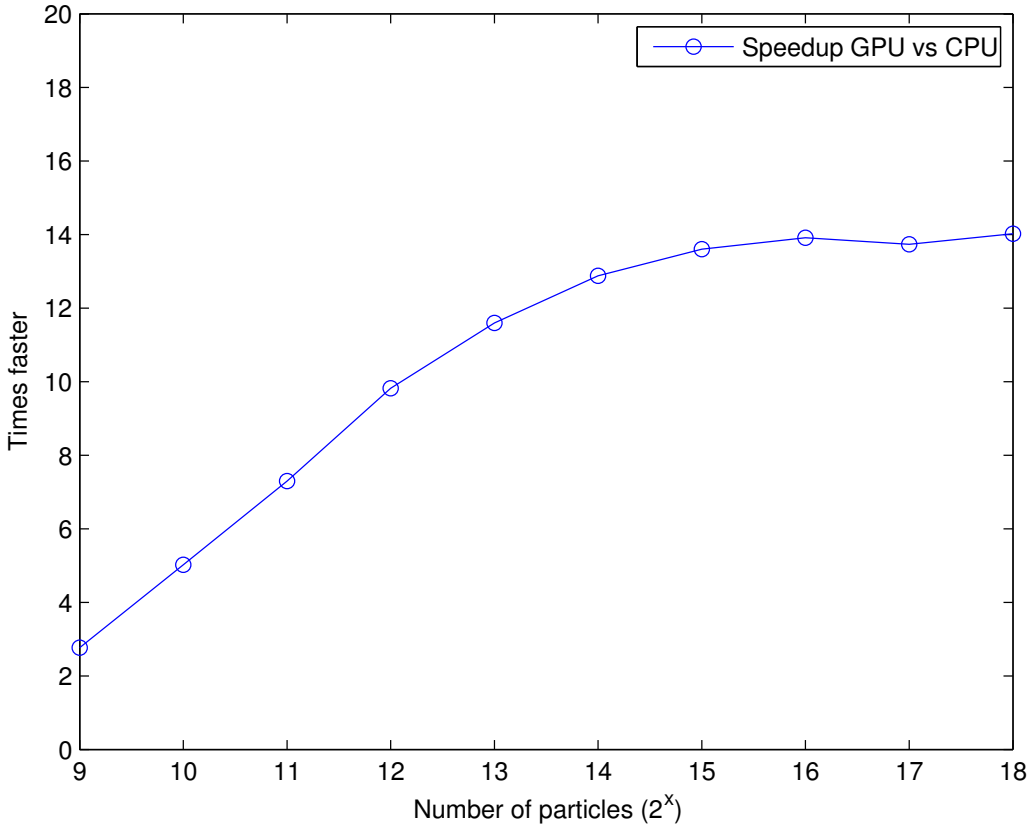

Figure 1.2: Total speedup

Future work

The next step of this project is to find an actual real time system and try to find out how better results we can achieve with the GPU implementation. As we know, it is possible that in the next generation of our robot (Aldebaran NAO) we will have a small GPU together with the CPU. If we have the opportunity to obtain the new version, we will move the localization module from the CPU to the GPU, because with the current implementation the vision does not have enough time to process the entire image.

Another update that we want to attempt in this project is to find a GPU card with computing capability 3+ and try out some special commands, like atomic add on the share memory. We believe that with these commands, we can make some steps, like

"find average", a lot faster.

Appendix

Table 1.3: All the implementation times of the Prediction step

Particles	**CPU Random** (ms)	**Function Variables** (ms)	**Const Var** (ms)
512	0.312741	0.078214	0.0704633
1024	0.582046	0.112544	0.103282
2048	1.1139	0.19125	0.170849
4096	2.24614	0.34187	0.309846
8192	4.36583	0.62428	0.57529
16384	8.68147	1.27429	1.12548
32768	17.4015	2.43078	2.22008
65536	35.7847	5.062643	4.42761
131072	69.0946	9.702342	8.91023
262144	137.958	19.32412	17.7288

Table 1.4: All the implementation times of the Update step

Particles	**Two Updates** (ms)	**Custom Maths** (ms)	**NVidia maths** (ms)
512	0.112385	0.067564	0.0667384
1024	0.124754	0.083245	0.0721206
2048	0.14545	0.091451	0.080732
4096	0.187163	0.094484	0.0850377
8192	0.246521	0.145864	0.134553
16384	0.373507	0.215464	0.199139
32768	0.551806	0.375486	0.344456
65536	0.916857	0.658452	0.587729
131072	1.74929	1.243456	1.13132
262144	2.89586	2.14541	2.03552

Table 1.5: All the implementation times of the Re-sampling step

Particles	**Run on the GPU** (ms)	**Normalize on the GPU** (ms)	**Final** (ms)
512	542.851	0.170457	0.169475
1024	2432.472	0.112165	0.11001
2048	10458.986	0.139487	0.130823
4096	-	0.198784	0.184341
8192	-	0.305481	0.282458
16384	-	0.524542	0.483647
32768	-	1.14547	0.919722
65536	-	1.93245	1.87611
131072	-	4.98764	4.27156
262144	-	11.954	10.663

Table 1.6: Execution times on the mhl GPU of the final implementation

Particles	**Predict** (ms)	**Update** (ms)	**Re-sample** (ms)	**Find average** (ms)
1024	0.0318533	0.0473628	0.0584737	0.0298939
2048	0.0318533	0.0473628	0.0782953	0.0356798
4096	0.0318533	0.0484392	0.11893	0.0520733
8192	0.034749	0.0484392	0.198216	0.0858245
16384	0.0530888	0.050592	0.354807	0.150434
32768	0.0868726	0.0775027	0.675917	0.274831
65536	0.153475	0.130248	1.31715	0.527483
131072	0.281853	0.235737	2.39643	0.882353
262144	0.523166	0.44564	5.51338	1.05593
524288	0.986486	0.861141	12.0704	1.31244
1048576	1.91506	1.69214	24.8811	2.17165
2097152	3.76062	3.35414	49.8821	3.93443
4194304	7.45946	6.676	99.7384	7.46287