Speeding-Up the Particle Filter Algorithm for Tracking Multiple Targets Using CUDA Programming

Jinhua Zhang*, Wenkai Guan, Cristinel Ababei, Henry Medeiros, Richard J. Povinelli
Dept. of Electrical and Computer Engineering
Marquette University
Milwaukee, WI, USA
jinhua.zhang94@gmail.com (Contact Author)
{wenkai.guan,cristinel.ababei,henry.medeiros,richard.povinelli}@mu.edu

***Abstract*—Object detection and tracking are essential tasks in many computer vision applications. One of the most popular tracking algorithms is the particle filter, which is widely used for real-time object tracking in live video streams. While very popular, the particle filter algorithm suffers from increased computational runtimes for high-resolution frames and large numbers of particles. In this paper, we investigate the use of CUDA programming as a method to parallelize portions of the particle filter algorithm in order to speed-up its execution time on compute systems that are equipped with NVIDIA GPUs. Experiments that compare a CPU sequential version, as the base case, with the CUDA parallelized version demonstrate an achievable speed-up of up to 7.5x for a 3840x2160 video resolution, and 9216 particles on a computer equipped with an NVIDIA Tesla K40c GPU.**

***Index Terms*—object tracking; particle filter algorithm; GPU; CUDA programming;**

Type of submission: **Full/Regular Research Paper**
The acronym that this paper is being submitted to: **Symposium on Signal & Image Processing, Computer Vision & Pattern Recognition (CSCI-ISPC)**;

I. Introduction and Related Work

There have been previous attempts to speed-up the particle filter algorithm. A first category of such attempts include Field Programmable Gate Array (FPGA) based or application specific circuit approaches [1]–[3]. Although many of the FPGA-based implementations provide improvements, they suffer from not being easily portable. In this paper, we focus on parallelization of the particle filter algorithm using Compute Unified Device Architecture (CUDA) [4], which has the advantages of being more platform independent as well as more accessible to a larger number of programmers, compared to FPGA approaches that require very specialized VHDL or Verilog programming. Thus, our approach falls in the second category of attempts to speed-up the particle filter algorithm, that of using Graphics Processing Units (GPUs).

There have been several previous studies that also used CUDA programming to speed up the particle filter algorithm. The study in [5] proposed a CUDA implementation of the particle filter algorithm for tracking the hands of a car driver. The performance of the algorithm achieved 30 fps, for 8192 particles. The study in [6] focused on pedestrian detection and tracking at night-time. Most of the previous studies used low-resolution videos to test the performance of the CUDA based implementation. For example, the study in [7] used 96x64 videos as input. As a result, the execution time for each frame in [7] is only 7.51ms. However, in real tracking applications, most of the cameras use at least 640x480 resolution. In addition, although some previous studies had dramatic speed-up, they used a large number of particles in their experiment. This is because as the number of the particles increases, the execution time of the particle filter algorithm on CPU increases dramatically. The dramatic speed-up is obtained by using a large number of particles to the algorithm. For example, in [8] 10,000 particles were used to track the object. The study in [6] is the closest to the proposed work in this paper. It is based on the HSV histogram to update the weights of the particles. A similar model to update the weights of the particles is used in our paper as well. We achieve a speed-up of up to 7.5x, which is better than their reported speed-up of 6.4x. The study in [9] used a testing platform that is the closet to the one in this paper. They reported a speed-up of 5x.

In contrast to the above studies, there is literature that directly addressed the computational runtime by optimizing specific algorithmic components of the particle filter algorithm. The study in [10] proposed a parallel redistribute method using OpenMP to provide 3x speed-up compared to its equivalent on CPU. The study in [11] focused on the resample step and proposed metropolis and rejection resamplers to shorten the computational runtime. The study in [12] implemented the implicit equal-weights particle filter algorithm for ocean drift trajectory forecasting. Their experimental results show that short-term drift forecasting is improved with up to twelve hours. The study in [13] presented a method to reduce the communication cost of the particle information. Their method has the potential to be the method of choice in certain specific settings. These previous studies have focused primarily on speeding up the particle filter algorithm by optimizing the algorithm itself. In this paper, we have not done any work

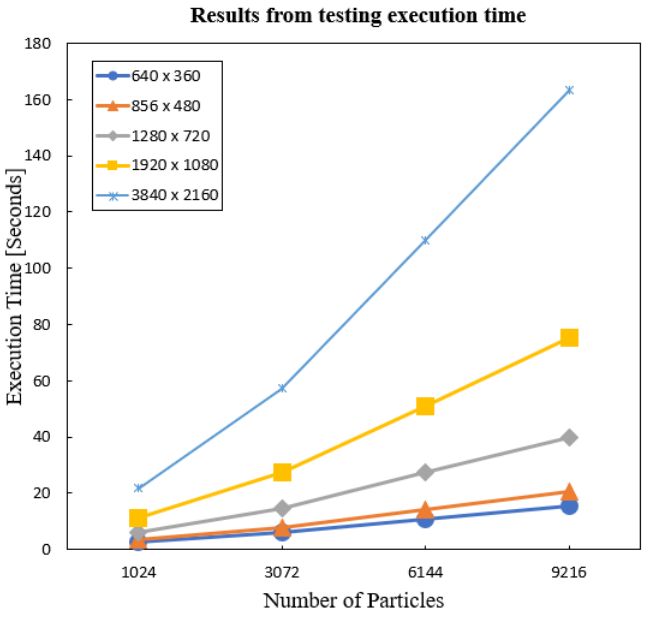


Fig. 1. Variation of the execution time of the particle filter algorithm with respect to the number of particles when executed sequentially on a CPU.

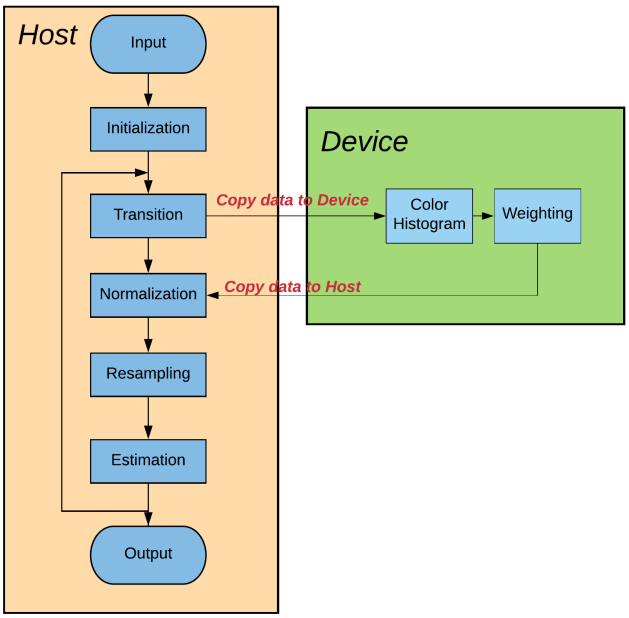


Fig. 2. Illustration of the CUDA-based implementation of the particle filter algorithm.

related to that. The main contribution of our paper is to use CUDA programming as a method for parallelization. In theory, this method could be applied to any algorithmically optimized particle filter to further improve the execution time.

II. Sequential Version of the Particle Filter Algorithm

Due to lack of space, for background information on the particle filter algorithm, we refer the reader to previous literature [14]–[16]. In this paper, we use as a base or reference case an implementation of the particle filter algorithm in C/C++. This implementation integrates methods provided by OpenCV [17] and initially executes sequentially (i.e., single threaded) on a general purpose sequential CPU. It is already quite efficient when the algorithm is run with a relatively small number of particles. However, when the number of particles is increased, the execution time increases significantly, as shown in Fig. 1. The datapoints in this figure were obtained by executing the reference implementation on the same machine as the experiments discussed later in the simulations results section.

To identify which portions of the main steps of the particle filter algorithm account for most of the computational runtime, we profiled the sequential implementation of the algorithm on a Linux machine with *gprof* [19]. This tool helps to track the runtime of individual functions and provides information on the number of times functions are called. By analyzing the profiling logs, we found that approximately 80% of the execution time is taken by the calculation of the likelihood function, which computes the histograms and weights of the particles. Note that this is consistent with the findings in previous studies [20]. The algorithm computes the histogram for a rectangular region of pixels associated with a particle, and this must be done separately for all particles. Thus, when the video resolution and the number of particles increase, the computational runtime is thus dominated by this portion of the overall particle filter algorithm. Our implementation will specifically focus on that portion of the algorithm.

III. Parallelization Using CUDA Programming

Based on the analysis from the previous section, the CUDA-based parallelization is implemented by re-coding portions of the *Transition* and *Normalization* steps as illustrated in Fig. 2. The CUDA programming model is a heterogeneous model in which both CPU and GPU are utilized. Within the CUDA programming context, the host refers to the CPU and its memory, and the device refers to the GPU and its memory. Code executed on the host is capable of managing memory on both the host and the device and also launch *kernels*, which are special functions executed on the device. These kernels are executed by many GPU threads in parallel.

After the initialization step is done, the algorithm moves to the transition step. In this step, the predicted particles are generated from a Gaussian distribution on the host first. It is after the initialization step that the likelihood function is called. This function is parallelized using CUDA. The same memory size of particles is allocated on the device, and then the particles are copied from the host to the device. Once we have the same particles on the device, the kernel function of the likelihood calculation is launched. On the device, the calculation of the likelihood function for each particle is executed by separate threads. Specifically, in each thread, the ROI is converted from RGB color channels to HSV channels. In order to compare the Bhattacharyya distance between predicted particles and initial particles to decide the weight for each predicted particle, the histogram of the Hue

channel is computed. The Hue histogram is normalized to the interval [0,1]. This is because after normalization, the Hue histogram is easily used to compare the Bhattacharyya distance between the particles. Once all the CUDA threads finish the processing of the kernel function, the particles are copied back to the host. The CPU then continues with the *Resampling* step, as shown in Fig. 2.

IV. Experiments

In order to evaluate our CUDA-based parallelized version of the particle filter algorithm we conduct several experiments. All tests were performed on a system with the following specifications: Linux Ubuntu 18.04 LTS, 3.5 GHz Intel Xeon CPU with 8 Cores, 745 MHz Nvidia Tesla K40C, PCIe version 3.0, Nvidia driver version 390.116. CUDA Toolkit version 9.1, and OpenCV library 3.4.2. The complete source code of our implementation is made publicly available at github [22].

A. Impact of Number of Particles and of Video Frame Size

To test the performance of the CUDA-based implementation for different image resolutions, a pre-recorded video with varying resolutions from 640 x 360 pixels to 3840 x 2160 pixels was recorded and tested. This video shows a tennis ball rolling on the floor from left to right. The video was recorded at a 3840 x 2160 resolution and then down-sampled to other lower resolutions to test the performance for different resolutions. For each resolution, experiments were conducted for several different numbers of particles. Both the reference sequential and paralel implementations were able to track the ball.

The results of these experiments are summarized in Fig. 3, which shows the speed-up obtained by the CUDA implementation compared to the CPU implementation. It can be observed that, as expected, the runtime of the CUDA-based implementation is shorter than that of the sequential CPU version for all tested video resolutions. This is because the likelihood function for all particles is executed as a parallelized function (i.e., kernel) by multiple threads. Fig. 3 indicates that the speed-up of the CUDA implementation is affected by the number of particles. Looking at the tests with the same number of particles, the speed-up does not change when the video resolution increases. The minimum speed-up is around 1.6x when the number of particles is 1024. The maximum speed-up is 7x when the number of the particles is 9216. This indicates that the video resolution does not have a significant impact on the variation of the speed-up for a given number of particles. That is because calculation for all the particles is parallelized and executed by different threads, but, calculation of the Hue histogram for each particle is processed sequentially in each thread. The execution time of the calculation of the Hue histogram for a particle increases because of the higher number of pixels in a frame, hence, the runtimes for both implementations increase proportionally. Nevertheless, it needs to be noticed that the video resolution influences the total execution of the CUDA implementation. Higher resolutions require longer computational runtimes. The reason for that is the computation

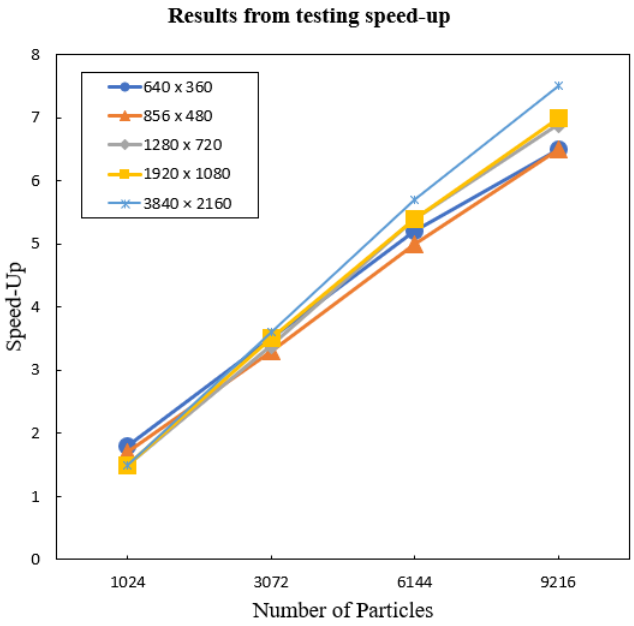


Fig. 3. The relative speed-up obtained by the CUDA implementation compared to the CPU implementation for a varying number particles and different video resolutions.

for each particle is parallelized in the threads executed on GPU using CUDA programming.

B. Realtime Dynamic Surveillance Scenario

To test the performance of the CUDA implementation in a dynamic surveillance scenario, we tested it on a realtime video stream obtained from a web camera. In the initialization step of the particle filter algorithm, targets are selected automatically via face detection using a Haar cascade classifier. We use the standard Haar cascade classifier *haarcascade_frontalface_alt.xml* available inside OpenCV.

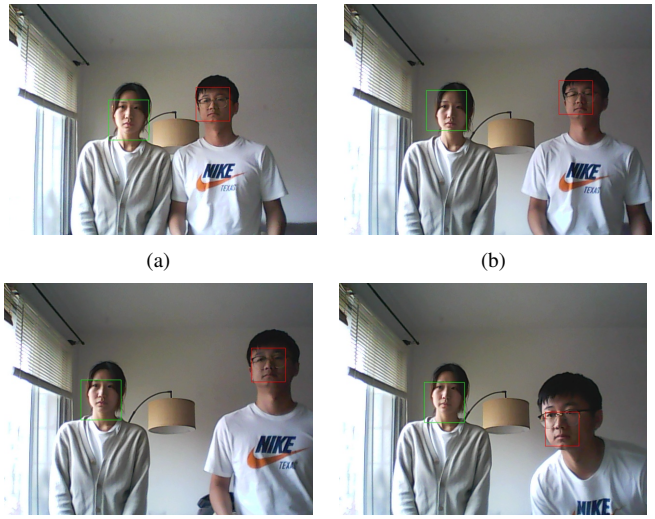
Several selected frames from this experiment are shown in Fig. 4. These frames are recorded at 3 seconds from each other. This figure shows qualitatively that both faces were detected and tracked automatically in the realtime video stream from a web camera, without human intervention. The video resolution of the realtime video stream is 640x480 pixels, which is a common resolution for a web camera, at a 30 fps. In this experiment, the CUDA-based implementation augmented with the automatic face detection and capability achieved an average speed-up of 6.5x compared to the sequential CPU implementation.

C. Results for the Bolt Dataset

In this section, we test the performance of the CUDA-based implementation on video sequences from the *Bolt* dataset, benchmark OTB100 [21]. In the test, we used a varying number of particles in the range of 1024 to 9216. The video resolution is 640x360 pixels. As expected, the CUDA implementation is faster in all cases. When the number of particles increases, the execution time of the CPU implementation increases significantly. However, the execution time of the

TABLE I
EXECUTION TIMES OF THE MAIN STEPS OF THE PARTICLE FILTER ALGORITHM.

	1024		3072		6144		9216	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Initialzation	0.446 ms	94.6 ms	0.734ms	95.8 ms	2.59 ms	99.2 ms	3.71 ms	93.9 ms
Transition	30.35 ms	10.1 ms	71.15 ms	12.3 ms	129.1 ms	15.2 ms	191.43 ms	18.3 ms
Sort	52.42 μs	83.48 μs	117.29 μs	237.4 μs	427.6 μs	413.82 μs	697.23 μs	564.23 μs
Normalization	7.58 μs	11.75 μs	19.83 μs	25.7 μs	54.79 μs	57.6 μs	90.97 μs	79.18 μs
Resample	74.24 μs	117.95 μs	223.31 μs	345.4 μs	525.1 μs	589.3 μs	880.79 μs	783.06 μs



(a) (b) (c) (d)

Fig. 4. Qualitative illustration of face tracking in realtime. These are four different frames during the execution of the CUDA implementation.

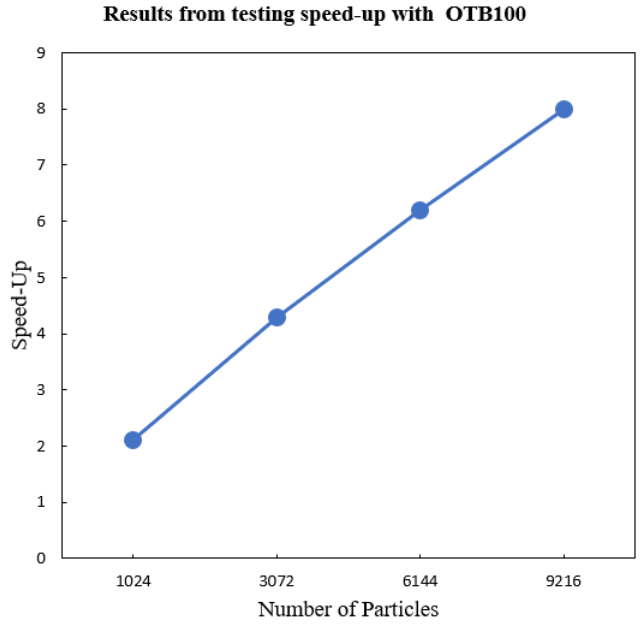


Fig. 5. Relative speed-up obtained by the CUDA implementation compared to the CPU implementation for different numbers of particles, on the OTB100 benchmark.

CUDA implementation increases at a much slower pace, which demonstrates the advantage of the CUDA-based implementation. Fig. 5 shows the speed-up obtained by the CUDA implementation when processing with a different number of particles. As expected, the achieved speed-up in most of tests is the same as the speed-ups reported in Fig. 3. When the particle number is 9216, the maximum speed-up is 8x.

Table I shows the execution time for each of the major processing steps of the particle filter algorithm for both CPU and CUDA implementations. As expected, the execution time of the *Transition* step of the CUDA version is much shorter in all the cases. This is because the likelihood calculation for all particles in the *Transition* step is parallelized. Note that the execution time of the CPU version of this step increases at a faster rate as the number of particles increase. In addition, the *Sort* and *Resample* steps take 126μs and are executed 350 times, which is approximately 10% of the total execution time. This portion could be parallelized in future work.

V. CONCLUSION

We presented a parallelization approach using CUDA programming of the particle tracking algorithm. After identifying the portions that account for the majority of the computational runtime, custom CUDA kernels were developed to change the reference sequential implementation of the algorithm. Experiments demonstrated that the parallelized version benefited from the execution of these kernels on GPUs, which in turn translated into algorithm speed-up of up to 7.5x for a 3840x2160 video resolution, and 9216 particles on a computer equipped with an NVIDIA Tesla K40c GPU. The algorithms were tested on both pre-recorded as well as on realtime video streams that mimicked real surveillance scenarios.

REFERENCES

[1] A. Jarrah, M. M. Jamali, and S.S.S. Hosseini, "Optimized FPGA based implementation of particle filter for tracking applications," *IEEE National Aerospace and Electronics Conference*, 2014.

[2] H. Sugano and R. Miyamoto, "Hardware implementation of a cascade particle filter," *IEEE Int. Conference on Image Processing (ICIP)*, 2009.

[3] H.A.A. El-Halym, I.I. Mahmoud, and S.E.-D. Habib, "Efficient hardware architecture for Particle Filter based object tracking," *IEEE Int. Conference on Image Processing (ICIP)*, 2010.

[4] J. Sander and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison Wesley, 2010.

[5] N. Ikoma and T. Ito, "GPGPU implementation of visual tracking by particle filter with pixel ratio likelihood," *IEEE/SICE Int. Symposium on System Integration (SII)*, 2012.

[6] B.-J. Choi, B.-W. Yoon, J.-K. Song, and J. Park, "Implementation of pedestrian detection and tracking with GPU at night-time," *Journal of Broadcast Engineering*, vol. 20, pp. 421-429, May 2015.

[7] B. Rymut and B. Kwolek, "GPU-accelerated object tracking using particle filtering and appearance-adaptive models," *Image Processing and Communications Challenges 2*, R. S. Choras, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 337-344, 2010.

[8] M.A. Goodrum, M.J. Trotter, A. Aksel, S.T. Acton, and K. Skadron, "Parallelization of particle filter algorithms," *Computer Architecture*, A.L. Varbanescu, A. Molnos, and R. van Nieuwpoort, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 139-149, 2012.

[9] P. Jecmen, F. Lerasle, and A.A. Mekonnen, "Trade-off between GPGPU based implementations of multi object tracking particle filter," *Int. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, 2017.

[10] A. Varsi, J. Taylor, L. Kekempanos, E. Pyzer-Knapp, and S. Maskell, "A fast parallel particle filter for shared memory systems," *IEEE Signal Processing Letters*, vol. 27, pp. 1570-1574, 2020.

[11] L.M. Murray, A. Lee, and P.E. Jacob,"Parallel resampling in the particle filter," *Journal of Computational and Graphical Statistics*, vol. 25, no. 3, pp. 789-805, 2016.

[12] H.H. Holm, M.L. Sætra, and P.J. van Leeuwen, "Massively parallel implicit equal-weights particle filter for ocean drift trajectory forecasting," *Journal of Computational Physics*, vol 6. 100053, 2020.

[13] K. Heine, N. Whiteley, and A.T. Cemgil, "Parallelizing particle filters with butterfly interactions," *Scandinavian Journal of Statistics*, Aug. 2019.

[14] A. Doucet, N. Freitas, K. Murphy, and S. Russell, Sequential Monte Carlo Methods in Practice, Springer, 2013.

[15] M. Chao, C. Chu, C. Chao, and A. Wu, "Efficient parallelized particle filter design on CUDA," *IEEE Workshop On Signal Processing Systems*, pp. 299–304, Oct. 2010.

[16] M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Trans. on Signal Processing*, vol. 50, no. 2, pp. 174-188, Feb. 2002.

[17] OpenCV, 2020, [Online]. Available: https://opencv.org

[18] G. Szwoch,"Performance evaluation of the parallel object tracking algorithm employing the particle filter," *Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pp. 119–124, Sep. 2016.

[19] H. Arora, Gprof tutorial – how to use Linux gnu gcc profiling tool, 2020, [Online]. Available: https://www.thegeekstuff.com/2012/08/gprof-tutorial

[20] H. Medeiros, G. Holguin, P.J. Shin, and J. Park, "A parallel histogram-based particle filter for object tracking on SIMD-based smart cameras," *Computer Vision and Image Understanding*, vol. 114, no. 11, pp. 1264-1272, Nov. 2010.

[21] Visual Tracker Benchmark. [Online]. Available: http://cvlab.hanyang.ac.kr/tracker_benchmark/datasets.html

[22] Speeding up particle filter for tracking targets using CUDA, Github repository, [Online]. Available: https://github.com/KevinZhanggg/Speeding-up-particle-filter-for-tracking -targets-using-CUDA