GPU-accelerated Object Tracking Using Particle Filtering and Appearance-adaptive Models

Boguslaw Rymut* and Bogdan Kwolek

Rzeszów University of Technology
W. Pola 2, 35-959 Rzeszów, Poland
bkwolek@prz.edu.pl

Summary. In this work we present an object tracking algorithm running on GPU. The tracking is achieved by a particle filter using appearance-adaptive models. The main focus of our work is parallel computation of the particle weights. The tracker yields promising GPU/CPU speed-up. We demonstrate that the GPU implementation of the algorithm that runs with 256 particles is about 30 times faster than the CPU implementation. Practical implementation issues in the CUDA framework are discussed. The algorithm has been tested on freely available test sequences.

1 Introduction

Driven by the huge market for multimedia and games, graphics processors have evolved more quickly than CPUs, and currently outperform them not only in terms of processing power, but also in terms of memory performance. The increasing programmability and computational power of the graphics processing unit (GPU) provides great capability for acceleration of computer vision algorithms. The GPU computations are done in parallel and algorithms must work in multi-thread mode in order to exploit the computational power of the GPU [1], which is not a feature of many vision algorithms. Unlike traditional CPU-based programs, GPU-based programs have several limitations on how memory can be accessed. Thus, a majority of vision algorithms either cannot be implemented on current GPUs or can be implemented with considerable difficulties, in consequence leading to unsatisfactory speed-up.

The key to using the GPU for accelerating the computer vision algorithm is to view it as a streaming, data-parallel computer, and the computations in the form of SIMD, data-parallel kernels. GPU implementations should access constant memory efficiently, avoid shared memory bank conflicts, coalesce global

* B. Rymut is presently a student, doing his MSc thesis on GPU-based object tracking

memory accesses, and overlap arithmetic with global memory latency. In general, the number of arithmetic operations must be high enough to effectively hide memory latency.

Visual tracking of objects of interest has received significant attention in the vision community. It is the key to the effective use of more advanced technologies, like human identification, event recognition, crowd analysis, etc. In the last decade a number of robust tracking strategies was proposed, which are able to tolerate changes in target appearance and track targets in complex scenes. One such successful approach is the particle filter [2][3]. The most important property of the particle filter (PF) is its ability to handle complex, multi-modal (non-Gaussian) posterior distributions. Such distributions are approximated by a collection of the particles. Essentially, the number of particles required to adequately approximate the distribution grows exponentially with the dimensionality of the state space. PFs are computationally expensive as the number of particles needs to be large for precise results. Moreover, the observation models are often built on complex appearance models, and as the result the trackers have difficulties to operate with 25/30 frames per second.

Adaptive appearance models have demonstrated great effectiveness in object tracking. In [4], the appearance model is based on phase information derived from the image intensity. The appearance models [5][6] consist of three components, namely W, S, F, where the W component represents the two-frame variations, the S component models temporally stable pixel intensities, and the F component is a fixed template of the target to prevent the model from drifting away. The particle filters built on adaptive appearance models algorithms produce good tracking results, but require considerable computational power. This motivated us to elaborate a GPU implementation of such an algorithm. Since the objects are represented as 2D arrays of pixels data, our algorithm takes advantages of GPU effectively.

The contribution of our work is an object tracking algorithm running on GPU. The tracking is achieved by a particle filter using appearance-adaptive models. The tracker yields promising GPU/CPU speed-up. We demonstrate that the GPU implementation of the algorithm that runs with 256 particles is about 30 times faster than the CPU implementation.

2 Object tracking using appearance-adaptive models in particle filter

In this section we overview the particle filtering. The section explains also how the object undergoing tracking is modeled.

2.1 Particle filtering

The particle filter simulates the behavior of the dynamical system. Each sample predicts future behavior of the system in a Monte-Carlo fashion, and the

samples that match the observed system behavior are kept, whereas ones that are unsuccessful in predicting tend to die out. The evolution of the state of the target as well as its measurement process is modeled by a set of (possibly non-linear) equations perturbed by (possibly non-Gaussian) i.i.d. noise:

$$\mathbf{x}_k = f_k(\mathbf{x}_{k-1}, \mathbf{v}_k) \tag{1}$$

$$\mathbf{z}_k = h_k(\mathbf{x}_k, \mathbf{n}_k) \tag{2}$$

where $\mathbf{x}_k$ denotes the state of the target at discrete time k, $\mathbf{v}_k$ is the process noise vector, $\mathbf{z}_k$ is the measurement vector, and $\mathbf{n}_k$ is the measurement noise vector. The aim is to estimate the distribution of the target state given all the previous measurements, that is, $p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1})$, where $\mathbf{z}_{1:k-1} = \{\mathbf{z}_1, \ldots, \mathbf{z}_{k-1}\}$. Given the initial distribution of the target, we can recursively predict the state of the target using:

$$p(\mathbf{x}_k|\mathbf{z}_{1:k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1}) p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1}) d\mathbf{x}_{k-1} \tag{3}$$

If a new measurement becomes available, the state can be updated using Bayes' rule:

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) = \frac{p(\mathbf{z}_k|\mathbf{x}_k) p(\mathbf{x}_k|\mathbf{z}_{1:k-1})}{p(\mathbf{z}_k|\mathbf{z}_{1:k-1})} \tag{4}$$

The complete tracking scheme, known as the recursive Bayesian filter first calculates the *a priori* density $p(\mathbf{x}_k|\mathbf{z}_{1:k-1})$ using the system model and then evaluates *a posteriori* density $p(\mathbf{x}_k|\mathbf{z}_{1:k})$ given the new measurement.

In the PF, the distribution $p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1})$ is approximated by a set of M particles $\{\mathbf{x}^i_{k-1}\}_{i=1\ldots M}$ and associated weights $\{w^i_{k-1}\}_{i=1\ldots M}$ as follows: $p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1}) \approx \sum w^i_{k-1} \delta(\mathbf{x}_k - \mathbf{x}^i_{k-1})$, where $w^i_k \propto w^i_{k-1} \frac{p(\mathbf{z}_k|\mathbf{x}^i_k) p(\mathbf{x}^i_k|\mathbf{x}^i_{k-1})}{q(\mathbf{x}^i_k|\mathbf{x}^i_{k-1}, \mathbf{z}_k)}$, whereas $\sum w^i_{k-1} = 1$ and $\delta(\cdot)$ is the Kronecker delta function. The term $q(\mathbf{x}^i_k|\mathbf{x}^i_{k-1}, \mathbf{z}_k)$ stands for an importance density, which is typically obtained by approximating $p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{z}_k)$ with a Gaussian distribution, or by using $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ like in CONDENSATION [2].

One of the practical difficulties that is associated with particle filters is degeneration of the particle population after a few iterations because weights of several particles are negligible, and, eventually, only a very small number of particles contributes to the posterior distribution. To mitigate this problem the resampling should be used in order to eliminate particles with low importance weights and multiply particles with high importance weights. Resampling can be carried out at every iteration or only when a substantial amount of degeneracy is observed [3].

The algorithm of the particle filter can be expressed in the pseudo-code:

1. For $i = 1, 2, \ldots, M$ sample or propose particles using $p(\mathbf{x}_k|\mathbf{x}_{k-1})$
2. For $i = 1, 2, \ldots, M$ calculate the weights, $\tilde{w}^i_k = w^i_{k-1} p(\mathbf{z}_k|\mathbf{x}^i_k)$
3. Normalize the weights w^i_k using $\tilde{w}^i_k$

4. Calculate the state estimates, $\hat{\mathbf{x}}_k = \sum_{i=1}^{M} w_k^i \mathbf{x}_k^i$
5. Resample $\{\mathbf{x}_k^i, w_k^i\}$ to get new set of particles $\{\mathbf{x}_k^j, w_k^j = 1/M\}$

2.2 Appearance-adaptive models

Our intensity-based appearance model consists of three components, namely, the W-component expressing the two-frame variations, the S-component characterizing the stable structure within all previous observations and F component representing a fixed initial template. The model $A_k = \{W_k, S_k, F_k\}$ represents the appearances existing in all observations up to time $k-1$. It is a mixture of Gaussians [4] with centers $\{\mu_{k,l} \,|\, l = w, s, f\}$, their corresponding variances $\{\sigma_{k,l}^2 \,|\, l = w, s, f\}$ and mixing probabilities $\{m_{k,l} \,|\, l = w, s, f\}$.

Let $I(\mathbf{x}, k)$ denote the brightness value at the position $\mathbf{x} = (x, y)$ in an image $\mathcal{I}$ that was acquired in time k. Let $\mathcal{R}$ be a set of J locations $\{\mathbf{x}(j) \,|\, j = 1, 2, ..., J\}$ defining a template. $Y_k(\mathcal{R})$ is a vector of the brightness values at locations $\mathbf{x}(j)$ in the template. The object likelihood is evaluated as follows:

$$p(\mathbf{z}_k|\mathbf{x}_k) = \prod_{j=1}^{J} \sum_{l=w,s,f} \frac{m_{k,l}(j)}{\sqrt{2\pi\sigma_{k,l}^2(j)}} \exp\left[-\frac{1}{2}\left(\frac{Y_k(j) - \mu_{k,l}(j)}{\sigma_{k,l}(j)}\right)^2\right] \tag{5}$$

It uses a recursively updated appearance model, which depicts stable structures seen so far, two-frame variations as well as initial object appearance.

The update of the current appearance model A_k to A_{t+1} is done using the Expectation Maximization (EM) algorithm [7]. For a template $\hat{Y}_k(\mathcal{R})$, which is located in the image $\mathcal{I}$ at position $\hat{\mathbf{x}}_k$, we evaluate the posterior contribution probabilities as follows:

$$o_{k,l}(j) = \frac{m_{k,l}(j)}{\sqrt{2\pi\sigma_{k,l}^2(j)}} \exp\left[-\frac{1}{2}\left(\frac{\hat{Y}_k(j) - \mu_{k,l}(j)}{\sigma_{k,l}(j)}\right)^2\right] \tag{6}$$

where $l = w, s, f$ and $j = 1, 2, ..., J$. The posterior contribution probabilities (with $\sum_k o_{k,l}(j) = 1$) are used in updating the mixing probabilities:

$$m_{k+1,l}(j) = \gamma o_{k,l}(j) + (1 - \gamma) m_{k,l}(j) \quad | \, l = w, s, f \tag{7}$$

where γ is accommodation factor. Then, the first and the second-moment images are determined in the following manner:

$$M_{k+1}^{(1)}(j) = (1 - \gamma) M_k^{(1)}(j) + \gamma o_{k,s}(j) \hat{Y}_k(j) \tag{8a}$$

$$M_{k+1}^{(2)}(j) = (1 - \gamma) M_k^{(2)}(j) + \gamma o_{k,s}(j) \hat{Y}_k^2(j) \tag{8b}$$

In the last step the mixture centers and the variances are calculated as follows:

$$\mu_{k+1,s}(j) = \frac{M_{k+1}^{(1)}(j)}{m_{k+1,s}(j)}, \;\; \sigma_{k+1,s}(j) = \sqrt{\frac{M_{k+1}^{(2)}(j)}{m_{k+1,s}(j)} - \mu_{k+1,s}^2(j)} \tag{9}$$

$$\mu_{k+1,w}(j) = \hat{Y}_k(j), \quad \sigma_{k+1,w}(j) = \sigma_{1,w}(j) \tag{10}$$

$$\mu_{k+1,f}(j) = \mu_{1,f}(j), \quad \sigma_{k+1,f}(j) = \sigma_{1,f}(j) \tag{11}$$

In order to initialize the model A_1 the initial moment images are set using the following formulas: $M_1^{(1)} = m_{1,s} Y_{t0}(\mathcal{R})$ and $M_1^{(2)} = m_{1,s}(\sigma_{1,s}^2 + Y_{t0}^2(\mathcal{R}))$.

3 Implementation of object tracking on GPU

At the beginning of this section we overview programming in CUDA framework. Afterwards we discuss implementation details of the algorithm on GPU.

3.1 Programming in CUDA

Compute Unified Device Architecture (CUDA) is a programming interface that employs the parallel architecture of NVIDIA GPUs for general purpose computing [8]. In CUDA, programs are expressed as kernels and GPU is viewed as a device that can carry out multiple concurrent threads. Threads are organized in two hierarchical levels, namely blocks, which are groups of threads executed on one of the GPU's multiprocessors, and grids, which are groups of blocks launched concurrently on the device, and which all execute the same kernel [1]. The memory requirements of a kernel determine how many threads can run concurrently on each multiprocessor. The threads in a block can share memory on a single multiprocessor. For a given kernel the block dimensions are chosen to optimize the utilization of the available computational resources. Warp is a group of threads executed physically in parallel in SIMD fashion. If the GPU processor must wait on one warp of threads, it simply starts executing work on a different one. Because registers are allocated to active threads, i.e. they stay allocated to the thread until it completes its execution, no swapping of registers and state takes place between GPU threads. In general, the more threads per block, the better the performance because the scheduler can better hide memory latencies. Large arithmetic calculations also contribute towards hiding the memory latency.

3.2 Implementation details

Porting well known computer vision algorithms to GPUs is a challenging task. Creating efficient data structures for effective use of the GPU memory model is a challenging problem in itself [9]. In order to take the full advantages of the available resources one should make careful decisions according to the data layout, data exchange and synchronization. In order to decompose the algorithm onto GPU the data-parallel portions of the program should be identified and then separated as CUDA kernels.

The predicting of the particles, see pseudo-code in subsection 2.1, is done in a kernel, which uses the normally distributed random numbers. The random

numbers are generated in advance in two kernels. In the first one we generate pseudo-random numbers using the Mersenne Twister [10] kernel provided by the CUDA^{TM} SDK. The second kernel employs the pseudo-random numbers to generate a set of normal random numbers. It uses Box Mueller transform based on trigonometric functions [11]. The random numbers are generated in 32 blocks and each block consists of 128 threads, where each thread generates two random numbers. In the kernel responsible for the prediction of the particles the position of each particle is calculated in a separate thread.

The calculation of the particle weights is done in two separate kernels. The first kernel performs the normalization of the pixels in the template to the unit variance, whereas the second one is executed after the first one and calculates the object likelihood (5). The size of the reference object template is 42×32. In both kernels each thread processes one column of the template. For each particle the number of threads is equal to 32. The product in (5) is calculated using parallel reduction [8]. The results achieved by the first kernel are stored in the shared memory, and both kernels operate on textures.

The normalized weights w_k^i, and the state estimate $\hat{\mathbf{x}}_k$, see pseudo-code in subsection 2.1, are calculated with the use of the parallel reduction. The object state consists in the template location as well as its size. The admissible change of the template size between successive frames is ± 1 pixel. Given the object state, the update of the appearance model takes place. In the resampling step the multinomial algorithm [12] has been utilized. The vector of cumulative sums was extracted with the use of parallel reduction, whereas the random numbers were taken from the set that had been generated in advance.

When a new image becomes available, the algorithm scales down and scales up the input image. The aim of this operation is to provide the images from which we can extract object templates that are smaller/larger about one pixel with regard to the estimated template size. The images are scaled using the bilinear interpolation. In the discussed kernel, the number of blocks is equal to the number of columns of the input images, whereas the number of threads is equal to the number of rows.

4 Experimental results

The experiments were conducted on a PC with 1 GB RAM, Intel Core 2 Quad, 2.66 GHz processor with NVIDIA GeForce 9800 GT graphics card. The graphics card has 14 stream multiprocessors, clocked at 1.5 GHz, each with 8 cores. It is equipped with 1024 MB RAM, 64 KB constant memory and 16 KB common memory. We implemented the algorithm in CUDA and compared the runtimes with its counterpart that was implemented in C/C++ and executed on the CPU. The CPU code was compiled with Visual Studio 2005 with the SSE2 (Streaming SIMD Extensions 2) option and O2 optimization turned on. Table 1 shows the running times and speed-up of the tracking algorithm both on CPU and GPU. The communication delays for transferring images from

CPU to GPU and vice versa have not been taken into account. The most time-consuming operation of the tracking algorithm is calculation of the likelihood function (5). This operation amounts to 0.82 of the whole processing time.

Table 1. Tracking times [ms] and speed-up obtained on CPU (Intel Core 2, 2.66 GHz) and on GPU (NVIDIA GeForce 9800 GT).

#particles	32	64	128	256	512
CPU	16.53	32.27	62.65	123.73	243.19
GPU	1.30	1.80	2.70	4.17	7.51
CPU/GPU	12.8	18.3	24.4	29.5	32.4

In the experiments we employed various number of particles, see Table 1. As we can observe, the algorithm achieves larger speed-up for larger number of particles. Figure 1 depicts some tracking results, which were obtained on gray images[1].

The dataset exhibits severe illumination conditions with partial shading. The template is a rectangular window initialized manually in the first frame. The initial template size for the Trellis70 dataset was set to 96×64. The object tracking was performed in three dimensional space, i.e. we track the location of the template as well as its scale. The size of the reference frame was set to 42×32. The maximal change of the template size between successive frames was constrained to ± 1 pixel. The example tracking results that are depicted on Fig. 1 were obtained using 32 particles.

Fig. 1. Face tracking using particle filter and adaptive appearance models. Frames #1, 50, 100, 150.

5 Conclusions

The adaptive appearance model-based particle filter is a robust algorithm for tracking objects. However, the computational cost of this algorithm is substantial. In this paper we presented our implementation of this algorithm

[1] Trellis70 dataset is available at: `http://www.cs.toronto.edu/_dross/ivt/`

on GPU. We explained how to design threads and memory structures for high performance. The result is a parallel algorithm that is easy to implement and yields promising GPU/CPU speed-up. The results showed that the GPU implementation of the algorithm running with 256 particles is about 30 times faster than the CPU implementation. Performance comparison on various CPU/GPU configurations of the particle filter is also presented.

References

1. Wasson, S. (2006) Nvidia's GeForce 8800 graphics processor. *Tech Report, November 8, PC Hardware Explored*.
2. Isard, M. and Blake, A. (2006) Condensation - conditional density propagation for visual tracking. *Int. J. of Computer Vision*, **29**, 5–28.
3. Doucet, A., Godsill, S., and Andrieu, C. (2000) On sequential Monte Carlo sampling methods for bayesian filtering. *Statistics and Computing*, **10**, 197–208.
4. Jepson, A. D., Fleet, D. J., and El-Maraghi, T. (2003) Robust on-line appearance models for visual tracking. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **25**, 1296–1311.
5. Zhang, X., Hu, W., Maybank, S., Li, X., and Zhu, M. (2008) Sequential particle swarm optimization for visual tracking. *IEEE Int. Conf. on Computer Vision and Pattern Recognition*, Anchorage, AK, USA, pp. 1–8.
6. Kwolek, B. (2009) Particle swarm optimization-based object tracking. *Fundamenta Informaticae*, **95**, 449–463.
7. Dempster, A., Laird, N., and Rubin, D. (1977) Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B*, **39**, 1–38.
8. Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008) Scalable parallel programming with CUDA. *ACM Queue*, **6**, 40–53.
9. Lefohn, A. E., Sengupta, S., Kniss, J., Strzodka, R., and Owens, J. D. (2006) Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, **25**, 60–99.
10. Matsumoto, M. and Nishimura, T. (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.
11. Box, G. E. P. and Muller, M. E. (1958) A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, **29**, 610–611.
12. Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993) Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEE Proc. part-F, Radar Signal Proc.*, **140**, 107–113.