GPU-Supported Object Tracking Using Adaptive Appearance Models and Particle Swarm Optimization

Boguslaw Rymut[*] and Bogdan Kwolek

Rzeszów University of Technology
W. Pola 2, 35-959 Rzeszów, Poland
http://www.prz.edu.pl

Abstract. This paper demonstrates how CUDA-capable Graphics Processor Unit can be effectively used to accelerate a tracking algorithm based on adaptive appearance models. The object tracking is achieved by particle swarm optimization algorithm. Experimental results show that the GPU implementation of the algorithm exhibits a more than 40-fold speed-up over the CPU implementation.

1 Introduction

While the central processing unit (CPU) is a general purpose microprocessor, which carries out the instructions of a computer program and is capable of processing a wide range of instructions, a graphics processor unit (GPU) is a dedicated microprocessor for the performing of graphical operations of the program. Modern GPUs are designed to operate in a SIMD fashion, which is a natural computational paradigm for graphical tasks. Recent research demonstrates that they are capable of accelerating a much broader scope of applications than the real-time rendering applications for which they were originally designed. GPUs offer potential for considerable increase in computation speed in applications that are data parallel.

Object tracking is an important problem in computer vision. It is a prerequisite for analyzing and understanding visual data, and has been an active research topic in the computer vision community over the last two decades. The goal of the object tracking is to automatically find the same object in an adjacent frame from an image sequence once it is initialized. Tracking algorithms are now employed in a wide variety of domains, such as robotics, human-computer-communication, vehicular traffic and surveillance. The challenge is to track the object irrespective of scale, rotation, perspective projection, occlusions, changes of appearance and illumination. Therefore, reliable vision-based object tracking is typically time-consuming process. However, it should be fast enough to maintain transparent interaction with the user.

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

Bayesian filtering techniques are often employed to achieve reliable tracking. For example, the Kalman filter has been used to track object in [1]. Unfortunately, object tracking in real-world environment rarely satisfies Kalman filter's requirements. Particle filtering [2] is superior to Kalman filtering without being constrained to linear models and Gaussian observations. However, particle filter (PF) being a sequential Monte Carlo method is time-consuming tracking technique. One of the major drawbacks of particle filters is that a huge number of particles are usually required for accurate estimation of state variables lying in a high dimensional space.

One way to achieve object tracking is searching for the best match of the predefined object model in the image. Recently, particle swarm optimization (PSO), a population based stochastic optimization technique has received considerable attention. Unlike the independent particles in the PF, the particles in a PSO interact locally with one another and with their environment in the course of searching for the best solution. Each particle in the swarm represents a candidate solution to the optimization problem. The most time consuming operation in PSO-based object tracking is evaluation of the fitness function. Since multiple candidate solutions are evaluated in each iteration, PSO-based tracking algorithms are computationally demanding for real-time applications.

Adaptive appearance models have acknowledged their great usefulness in visual tracking. In [3], the appearance model is based on phase information derived from the image intensity. Similar to this work, the appearance models that are utilized in [4][5] consist of three components, W, S, F, where the W component models the two-frame variations, the S component characterizes temporally stable images, and the F component is a fixed template of the target to prevent the model from drifting away. The algorithms mentioned above produce good tracking results, but are quite time-consuming. This motivated us to develop a GPU implementation of the tracking using particle swarm optimization with adaptive appearance models. Since in adaptive appearance model based tracking the objects are represented as 2D arrays of pixels data, our algorithm takes the advantage of SIMD architecture effectively.

2 Visual appearance modeling using 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 $A_t = \{W_t, S_t, F_t\}$ represents the appearances existing in all observations up to time $t-1$. It is a mixture of Gaussians [3] with centers $\{\mu_{k,t} \mid k = w, s, f\}$, their corresponding variances $\{\sigma^2_{k,t} \mid k = w, s, f\}$ and mixing probabilities $\{m_{k,t} \mid k = w, s, f\}$.

Let $I(p,t)$ denote the brightness value at the position $p = (x, y)$ in an image $\mathcal{I}$ that was acquired in time t. Let $\mathcal{R}$ be a set of J locations $\{p(j) \mid j = 1, 2, ..., J\}$ defining a template. $Y_t(\mathcal{R})$ is a vector of the brightness values at locations $p(j)$

in the template. The observation model has the following form:

$$p(z_t|x_t) = \prod_{j=1}^{J} \sum_{k=w,s,f} \frac{m_{k,t}(j)}{\sqrt{2\pi\sigma^2_{k,t}(j)}} \exp\left[-\frac{1}{2}\left(\frac{Y_t(j)-\mu_{k,t}(j)}{\sigma_{k,t}(j)}\right)^2\right] \tag{1}$$

where z_t is the observation corresponding to template parameterization x_t. In the object likelihood function we utilize a recursively updated appearance model, which depicts stable structures seen so far, two-frame variations as well as initial object appearance. Owing to normalization by subtracting the mean and dividing by standard deviation, the template becomes invariant to global illumination changes.

The update of the current appearance model A_t to A_{t+1} is done using the Expectation Maximization (EM) algorithm [6]. For a template $\hat{Y}_t(\mathcal{R})$, which was obtained from the image $\mathcal{I}$ using the estimated parameterization $\hat{x}_t$, we evaluate the posterior contribution probabilities as follows:

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

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

$$m_{k,t+1}(j) = \gamma o_{k,t}(j) + (1-\gamma) m_{k,t}(j) \quad | \; k = w, s, f \tag{3}$$

where γ is accommodation factor. Then, the first and the second-moment images are determined as follows:

$$M_{1,t+1}(j) = (1-\gamma) M_{1,t}(j) + \gamma o_{s,t}(j) \hat{Y}_t(j) \tag{4a}$$

$$M_{2,t+1}(j) = (1-\gamma) M_{2,t}(j) + \gamma o_{s,t}(j) \hat{Y}_t^2(j) \tag{4b}$$

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

$$\mu_{s,t+1}(j) = \frac{M_{1,t+1}(j)}{m_{s,t+1}(j)}, \quad \sigma_{s,t+1}(j) = \sqrt{\frac{M_{2,t+1}(j)}{m_{s,t+1}(j)} - \mu^2_{s,t+1}(j)} \tag{5}$$

$$\mu_{w,t+1}(j) = \hat{Y}_t(j), \quad \sigma_{w,t+1}(j) = \sigma_{w,1}(j) \tag{6}$$

$$\mu_{f,t+1}(j) = \mu_{f,1}(j), \quad \sigma_{f,t+1}(j) = \sigma_{f,1}(j) \tag{7}$$

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

3 PSO-based object tracking

PSO is a population based algorithm that exploits a set of particles representing potential solutions of the optimization task [7]. The particles fly through the n-dimensional problem space with a velocity subject to both stochastic and deterministic update rules. The algorithm seeks for the global best solution through adjusting at each time step the location of each individual according to personal best and the global best positions of particles in the entire swarm. Each particle keeps the position *pbest* in the problem space, which is associated with the best fitness it has achieved personally so far. Additionally, when a particle considers all the population as its topological neighbors, each particle employs *gbest* location, which has been obtained so far by any particle in the swarm. The new positions are subsequently scored by a fitness function f. The velocity of each particle i is updated in accordance with the following equation:

$$v_j^{(i)} \leftarrow w v_j^{(i)} + c_1 r_{1,j}(pbest_j^{(i)} - x_j^{(i)}) + c_2 r_{2,j}(gbest_j - x_j^{(i)}) \tag{8}$$

where $v_j^{(i)}$ is the velocity in the $j-$th dimension of the $i-$th particle, c_1, c_2 denote the acceleration coefficients, $r_{1,j}$ and $r_{2,j}$ are uniquely generated random numbers in the interval $[0.0,\ 1.0]$. The new position of a particle is calculated in the following manner:

$$x_j^{(i)} \leftarrow x_j^{(i)} + v_j^{(i)} \tag{9}$$

The local best position of each particle is updated as follows:

$$pbest^{(i)} \leftarrow \begin{cases} x^{(i)}, & \text{if } f(x^{(i)}) > f(pbest^{(i)}) \\ pbest^{(i)}, & \text{otherwise} \end{cases} \tag{10}$$

and the global best position *gbest* is defined as:

$$gbest \leftarrow \arg \max_{pbest^{(i)}} \{f(pbest^{(i)})\} \tag{11}$$

The value of velocity $v^{(i)}$ should be restricted to the range $[-v_{max}, v_{max}]$ to prevent particles from moving out of the search range. In some optimization problems the local best version of PSO, where particles are influenced by the best position within their neighborhood, as well as their own past experience can give better results. While such a configuration of the PSO is generally slower in convergence than algorithm with *gbest*, it typically results in much better solutions and explores a larger part of the problem space.

At the beginning of the optimization the PSO initializes randomly locations as well as the velocities of the particles. Then the algorithm selects *pbest* and *gbest* values. Afterwards, equations (8)-(11) are called until maximum iterations or minimum error criteria is attained. After that, given $\hat{x}_t = gbest$ we calculate $\hat{Y}_t$, and then update of the object model using formulas (2)-(7).

In the simplest solution the object tracking can be realized as deterministic searching of window location whose content best matches a reference window content. PSO allows us to avoid such time consuming exhaustive searching for the best match. It provides an optimal or sub-optimal match without the complete knowledge of the searching space. In PSO based tracking, at the beginning of each frame in the initialization stage, an initial position is assigned to each particle

$$x_t^{(i)} \leftarrow \mathcal{N}(gbest, \Sigma) \tag{12}$$

given the location *gbest* that has been estimated in the previous frame $t-1$. In the evaluation phase the fitness value of each particle is determined by a predefined observation model according to the following formula:

$$f(x_t^{(i)}) = p(z_t^{(i)} | x_t^{(i)}) \tag{13}$$

4 Programming of GPU

In this section we outline the architectural properties of G80 [8], which are the most relevant to our implementation. CUDA$^{\text{TM}}$ is a new language and development environment developed by NVIDIA, allowing execution of programs with thousands of data-parallel threads on NVIDIA G80 class GPUs [9]. Such threads are extremely lightweight and almost no cost for creation and context switch is needed. In CUDA, programs are expressed as kernels and GPU is viewed as a device, see Fig. 1, which can carry out multiple concurrent threads. Each kernel consists of a collection of threads arranged into blocks. A thread block is a group of threads, which can cooperate jointly through efficiently sharing data via some fast shared memory, and synchronizing their execution to coordinate memory accesses. A kernel should have enough blocks to simultaneously utilize all the multiprocessors in a given GPU. Many thread blocks can be assigned to a single multiprocessor, which are executed concurrently in a time-sharing fashion to keep GPU as busy as possible.

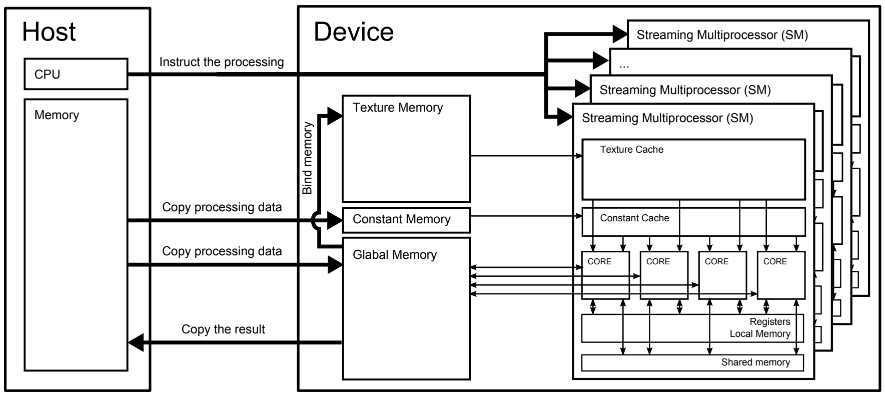

Fig. 1. G80 GPU architecture

As is illustrated in Fig. 1, both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. On the device side the data structures can be mapped to the texture memory to take advantages of special hardware on the GPU, which supplies a small amount of on-chip caching and a little more efficient access to off-chip memory. The device memory is accessible by all multiprocessors. Each multiprocessor has a set of registers, shared memory, constant cache, and texture cache. Constant/texture cache are read-only and have faster access than shared memory. Because only threads within the same block can cooperate via shared memory and thread synchronization, the user must partition the computation into multiple blocks. The number of threads per block is restricted by the limited memory resources of a multiprocessor core. On current GPUs, a thread block may contain up to 512 threads. In general, the more threads per block, the better the performance because the hardware can hide memory latencies. High arithmetic intensity also hides the memory latency. Storing the data locally reduces the need to access off-chip memory, thereby improving the performance.

5 Implementation details

Writing effective computer vision applications for GPU is not a trivial task. One should make careful decisions according the data layout, data exchange and synchronization to ensure that the program can take the full advantages of the available resources. In order to decompose the algorithm into GPU we should identify data-parallel portions of the program and separate them as CUDA kernels. The decomposition was done with regard to the main steps of the algorithm:

1. Assign each particle a random position in the problem hyperspace.
2. Evaluate the fitness function for each particle by applying (13).
3. For each particle i compare the particle's fitness value with its $f(pbest^{(i)})$. If the current value is better than the value $f(pbest^{(i)})$, then set this value as the $f(pbest^{(i)})$ and the current particle's position $pbest^{(i)}$ as $x^{(i)}$.
4. Find the particle that has the best fitness value *gbest*.
5. Update the velocities and positions of all particles according to (8) and (9).
6. Given *gbest*, update of the object model using formulas (2) - (7).
7. Repeat steps $2-6$ until maximum number of iterations is attained.

At the beginning of each frame we generate pseudo-random numbers using the Mersenne Twister [10] kernel provided by the CUDA$^{\text{TM}}$ SDK. From uniform random numbers we generate a vector of normal random numbers using Box Mueller transform based on trigonometric functions [11] in order to initialize the particle's positions in the problem hyperspace. The positions are generated using equation (12). The initialization is executed in 32 blocks and each block consists of 128 threads, where each thread generates two random numbers. The evaluation of the fitness function 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 thread is executed after the first one and calculates the fitness score. In both kernels each thread processes a single column of the template, and the parallel reduction technique [9] is used to obtain the final results. The results achieved by the first kernel are stored in the shared memory. Both kernels operate on textures. After the computation of the fitness score the calculation of the *pbest* locations and their corresponding fitness values takes place. Identical number of threads and blocks is used in this and in the initialization stage. Afterwards, the *gbest* value is calculated. Finally, the algorithm updates in parallel the velocities and the locations.

6 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 with 1.5 GHz, each with 8 cores, see Fig. 1. 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 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 of the tracking algorithm both on CPU and GPU as well as the speed-up. The communication delays for copying 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 fitness function (13). This operation amounts to 0.9 of the whole processing time.

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

#particles	32	64	128	256
CPU(2.66 GHz)	30.6 ms	60.0 ms	117.9 ms	234.2 ms
GPU(GF 9800 GT)	1.4 ms	1.9 ms	3.4 ms	5.6 ms
CPU/GPU	22.4	31.5	38.8	41.5

In object tracking experiments we employed various number of particles, see Table 1. We can notice that for larger number of particles the speed-up of the GPU algorithm is larger. The tracking is done in three dimensional space, i.e. we track the location of the template as well as its scale. The scaling is achieved via bilinear interpolation, which is extension of the linear interpolation for interpolating functions of two variables on the regular grid of pixels. Figure 2 depicts some tracking results, which were obtained on color images[1]. The first

[1] Thanks Dr. Birchfield for this sequence, obtained from http://robotics.stanford.edu/~birch/headtracker

image shown in Fig. 2 contains a face in the front of the background with colors that are similar to skin color. The size of the reference frame is 32×42. The change of template size between successive frames is ± 1 pixel. The experimental results depicted on Fig. 2 were obtained using 32 particles.

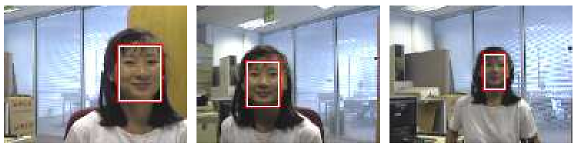

Fig. 2. GPU-based face tracking using adaptive appearance models

7 Conclusions

In this paper, we have shown how the adaptive appearance based tracking algorithm can be accelerated significantly using programmable graphics hardware. The results showed that our algorithm using GPU is about 40 times faster than a CPU implementation. As a result the tracking algorithm runs at frame-rates exceeding 60 frames per second.

References

1. Weng, S., Kuo, C., Tu, S.: Video object tracking using adaptive Kalman filter. J. Vis. Comun. Image Represent. **17** (2006) 1190–1208
2. Isard, M., Blake, A.: Condensation - conditional density propagation for visual tracking. Int. J. of Computer Vision **29** (2006) 5–28
3. Jepson, A.D., Fleet, D.J., El-Maraghi, T.: Robust on-line appearance models for visual tracking. IEEE Trans. on PAMI **25** (2003) 1296–1311
4. Zhang, X., Hu, W., Maybank, S., Li, X., Zhu, M.: Sequential particle swarm optimization for visual tracking. In: IEEE Int. Conf. on CVPR. (2008) 1–8
5. Kwolek, B.: Particle swarm optimization-based object tracking. Fundamenta Informaticae **95** (2009) 449–463
6. Dempster, A., Laird, N., Rubin, D.: Maximum likelihood from incomplete data via the EM algorithm. J. of the Royal Statistical Society. Series B **39** (1977) 1–38
7. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proc. of IEEE Int. Conf. on Neural Networks, IEEE Press, Piscataway, NJ (1995) 1942–1948
8. Wasson, S.: Nvidia's GeForce 8800 graphics processor. Technical report, PC Hardware Explored (2006)
9. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue **6** (2008) 40–53
10. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation **8** (1998) 3–30
11. Box, G.E.P., Muller, M.E.: A note on the generation of random normal deviates. The Annals of Mathematical Statistics **29** (1958) 610–611