Moving Object Detection in Video Using CUDA

Miroslav Schery
Bachelor Degree Programme (3), FIT BUT
E-mail: xscher01@stud.fit.vutbr.cz

Supervised by: Adam Herout
E-mail: herout@fit.vutbr.cz

ABSTRACT

There are many algorithms for moving object detection, that are more or less effective. Particle Filter algorithm performs very well in object tracking, but implementation with really great results is problematic to run in real-time. This is an opportunity for applying Cuda Platform. With Cuda, part of the algorithm can be run using parallel kernels on GPU, to achieve real-time execution or even better performance.

1. Introduction

The objective is to study and describe moving object detection algorithms in video and explore the possibility to apply parallel execution on Cuda (Compute Unified Device Architecture) Platform. Cuda is an invention of nVidia Company that is working on their GPGPU (General-purpose computing on graphics processing units) graphic cards. It enables the software developers, to access the computational power of GPU parallel architecture. CUDA parallel computing architecture is now shipping in GeForce, ION, Quadro, and Tesla GPUs which enables them to choose the right device for their needs.

Millions of CUDA-enabled GPUs were sold to date and software developers, scientists and researchers are using it in image and video processing, computational biology and chemistry, fluid dynamics simulation, CT image reconstruction, ray tracing, and much more. Cuda offers C and C++ interface extensions, which makes it quite easy for programmers to get used to.

2. Analysis

There is a decent amount of algorithms for moving object detection in video. In this work, particle filter algorithm[1] will be implemented. Really complex particle filter can achieve more than 90% of successful detections. Unfortunately, this tracking is usually slower than real-time. A lot of CPU versions offer several options for modifying the object detection method, which are slowing down the computation. For implementation of particle filter on Cuda, this work will focus on creating basic algorithm, to find out what performance improvement is possible to gain by running it in parallel. After that, some upgrades to the method of object detection itself will be included, to make previous basic implementation comparable with complex CPU versions.

Cuda is designed for applying arithmetic operations on large data sets (such as matrices, images), where the same operation can be performed across thousands of elements at the same time. This algorithm is therefore suitable for Cuda architecture.

2.1. Particle Filter

Particle filter generates a set of samples that are uniformly displaced around last (or predicted) position of tracked object. These samples are called particles and each particle is weighted by a weighting function. An example of weighting function is a sum of squared values of corresponding pixel difference or a difference between histograms. The new occurrence of tracked object is a particle with the weight corresponding to the best match between particle and object sample.

The amount of the generated particles should be modified to find a compromise between performance and satisfying tracking results. The more particles are used, the better results are received, but also with performance degradation.

2.2. Cuda Architecture

Cuda program is running on graphic card and is controlled by the CPU. The processor is unable to access the memory on GPU directly and vice versa. The communication between them is provided by Cuda functions. Because of that, the data needs to be copied to GPU for processing and then back to system memory.

The program contains one or more kernels. A kernel is a function callable from the host and executed on the GPU simultaneously by many threads in parallel. In each kernel call a thread/block configuration needs to be specified for effective distribution of work between threads.

Cuda offers many types of memory[2], each has its own positives and negatives. Global memory (ram installed on GPU) has big capacity but access time is really long so its utilization should be minimal. Each multiprocessor has its own 16kB of shared memory, which isn't much, but in right conditions it could reach the speed of registers. Registers are the fastest. Finally we have texture and constants memory. Their advantage is caching.

2.3. Cuda Implementation

The main objective was to implement particle filter algorithm, which would be capable of tracking in real-time or even faster. The program is developed and tested on a notebook with GeForce 9600M GT and Intel Core2Duo 2,5GHz processor. The tests of kernel configurations were also run on desktop configuration with GeForce GTX 285. The notebook GPU has 32 Cuda cores (For a comparison GeForce GTX 285 has 240 Cuda cores), which means it has only 4 multiprocessors. The first implementation without optimizations took about 110ms for one frame on notebook (10ms on GTX 285). As a test, an object with 122x92 pixels is tracked across 435 frames using 500 particles.

2.4. Program optimization

In a kernel function call, a block and thread configuration is required. It has a strong influence on kernel performance. To this date, each multiprocessor is capable of executing kernel function over one block with 512 threads maximum or multiple blocks with 786 threads altogether.

The time results of various kernel configurations are in the Table 1. The kernel with 512 threads was fastest on the notebook, but on desktop it was kernel with 256 threads per block. This is caused by different compute capability of these devices.

To bypass the necessity for data transfer back to system RAM for results display, OpenGL is used to display it from the GPU memory. Access to the global memory from the kernel is time expensive therefore a shared memory should be used. This memory is used for storing loaded data of an object sample, particle data from latest video frame and for results. Loading data from texture memory helps because it is cached and the algorithm loads data within quite small area of the frame.

Threads per block	8	512			256		384
Specifications	no optimization	no optimization	shared memory	shared memory, texture memory	shared memory	shared memory, texture memory	shared memory
GeForce 9600M GT (ms)	110,15	84,7	36,45	33,38	37,57	40,95	44,44
GeForce GTX 285 (ms)	9,8	6,11	3,17	3,1	3,06	2,981	3,15

Table 1) - Time results of various kernel configurations

2.5. Other future optimization methods

Particle filter algorithm requires pseudorandom number generating, which can't be done in parallel, so it's computed on CPU. While the CPU is working, GPU is idle. It would be useful to optimize the program, so that CPU and GPU are working in parallel.

Another time saving optimization could be asynchronous data transfer between system an GPU memory.

3. Conclusion

In the case of computer vision algorithms, which are mostly graphics processing, parallelization offers an easy performance boost. After some kernel rewritings, this kernel achieved real-time video processing even on the notebook graphic card. Thanks to optimal work distribution to threads and blocks, a massive amount of image data is processed in every moment in each multiprocessor and with using shared and texture memory, expensive accesses to global memory were reduced to minimum.

For algorithm that can be parallelized, using cuda is good decision. A lot of research and medical programs are using it and they are achieving a great performance. Cuda isn't just an experiment that would be forgotten, this project surely got future ahead.

Acknowledgement: This work was partially supported by the BUT FIT grant FIT-S-10-2 and the research plan MSM0021630528.

REFERENCES

[1] Musa, Z.B., Watada, J.: Motion Tracking Using Particle Filter, In: Knowledge-Based Intelligent Information and Engineering Systems, Volume 5179/2008, Heidelberg, 2008, (s. 119 - 126)

[2] NVIDIA Corporation, Programming Guide Version 2.3.1, Santa Clara, California, USA