Parallel Implementation of Recursive Background Modeling Technique in CUDA for Tracking Moving Objects in Video Traffic Surveillance

Tomáš Fabián, Jan Gaura
VŠB-Technical University of Ostrava
Department of Computer Science
Table of Content

• Background and Motivation
• Background Subtraction Algorithms
• Mixture of Gaussians
• Algorithm Memory Demands
• Image Decomposition for CUDA Programming Model
• Device Occupancy
• Frame Processing Sequence
• Memory Transfers
• CUDA Code Example
• CUDA-Enabled Hardware
• Kernel Run-times
• Results
• Conclusion
Usage possibilities

Local company providing MAN services including traffic camera system.

Almost 100 installed cameras monitoring traffic situation mostly at a crossroads.

Looking for a statistical evaluation of an acquired video sequences.

At the first stage we are concerned in a simple vehicle counting and a little bit more sophisticated motion direction estimation.

Background subtraction as a fundamental task in CV is a ground stone.
Background and Motivation

Features of a good background subtraction algorithm

Robustness against changes in illumination (varying daylight conditions).

Avoid detection non-stationary background objects (rain, snow, water surface, leaves etc).
Background Subtraction Algorithms

**Simple Non-recursive Techniques** (low computational complexity, average results, buffered):
- Frame differencing – simplest technique.
- Adaptive median filtering.
- Linear predictive filter – difficult to apply in real-time.
- Non-parametric model – use the entire history of images.

**Advanced Recursive Techniques** (computationally expensive, high quality output, recursively update single background model):
- Approximated median filter
- Kalman filter
- **Mixture of Gaussians**
  (Why? Well known approved method. Does not require precise method's parameters tuning, is able to handle multimodal background.)
Mixture of Gaussians

- Tracks density function for each pixel.
- Capable of handling multimodal background distribution.
- Despite the fact that MoG is parametric, parameters can be adaptively updated. Accurate initialization is not necessary.
- No large frame buffer.
- Has to store constant amount of parameters.

$$f_{X|k}(X|k, \Theta_k) = \frac{1}{(2\pi)^{n/2} |\Sigma_k|^{1/2}} e^{-\frac{1}{2} (X-\mu_k)^T \Sigma_k^{-1} (X-\mu_k)}$$

$$\Theta_k = \{\mu_k, \sigma_k\}$$

Three parameters are on-line estimated.

No. modeled surfaces.

$$f_X(X|\Phi) = \sum_{k=1}^{K} P(k) f_{X|k}(X|k, \Theta_k)$$
Algorithm Memory Demands

\[ S(r_x, r_y) = r_x r_y (K (s_\omega + s_{\mu_{r,g,b}} + s_\sigma)) \]

- Unfortunately algorithm need to store quite large amount of data.
- Required memory size \( S \) depends on image resolution \((r_x, r_y)\), number of modeled surfaces \( K \) and data types of values representing Gaussians (32 bit floats).
- For \( K = 5 \) we get \( 5 \cdot 5 \cdot 4 = 100 \) bytes per pixel.
Mixture of Gaussians

PDF for a selected pixel of gray scale image

\[
f_{X}(X|\Phi) = \sum_{k=1}^{K} P(k) f_{X|k}(X|k, \Theta_k)
\]

Bayes’ theorem help us obtain an expression to calculate probability of appearance of the k-th surface.

\[
P(k|X, \Phi) = \frac{P(k) f_{X|k}(X|k, \Theta_k)}{f_X(X|\Phi)}
\]
Every single frame of video sequence is processed by one Kernel call.

Whole image is divided into the equally sized thread blocks.

Every pixel is processed by one thread of corresponding thread block.

In our case:
320 (w) x 240 (h) pixels

Thread Block size: 16 (tx) x 16 (ty) threads

Grid size: 20 (w / tx) x 15 (h / ty)

Kernel Execution Configuration

dim3 threads( tx, ty, 1 );
dim3 grid( w / threads.x, h / threads.y, 1 );

Image Decomposition for CUDA Programming Model
Device Occupancy

Occupyancy = active warps per MPs / maximum number of active warps

Number of registers used per thread is the limiting factor.
16 x 16 x 44 = 11264 reg. per block (16384 reg. available, c.c. 1.3)

Higher occupancy does not necessary lead to higher performance.

Cubin file:

```c
... code { name = __globfunc__Z16kernelUpdateLRGBPfjb lmem = 100 smem = 25 reg = 44 bar = 0 bincode { 0xd0800215 0x00400780 0x11002204 0x11002410
```
Allocating pinned memory

cvCreateImageHeader(...);
cudaMallocHost(...);

cudaMallocPitch(...);

Video sequence loop follows

cudaMemcpyp(..., HostToDevice);
kernelllll <<< grid, threads >>> (g_img, pitch, ...);
cudaMemcpyp(..., DeviceToHost);
Memory Transfers

- cudaMemcpy(dst, src, nBytes, direction)
  - Limited by PCI-e bus transfer rates
- Use page-locked (pinned) host memory
  - Instead of malloc(...) we use cudaMemcpyHost(...)
  - Prevent OS from paging
  - Async. data transfers requires page-locked host memory.
  - Allow PCI-e DMA to run at full speed

Experimental results

<table>
<thead>
<tr>
<th>Data transfer direction</th>
<th>Pageable</th>
<th>Pinned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host to device</td>
<td>1.252 ms (0.686 GB/s)</td>
<td>0.909 ms (0.944 GB/s)</td>
</tr>
<tr>
<td>Device to host</td>
<td>3.026 ms (0.284 GB/s)</td>
<td>1.335 ms (0.643 GB/s)</td>
</tr>
</tbody>
</table>
CUDA Code Example

Host code:

```c
void kernelUpdateLRGB_CPU( float* g_img, size_t pitch,
    bool init, int width, int height )
{
    #pragma omp parallel for schedule( static )
    for( int y = 0; y < height; y++ ) {
        for( int x = 0; x < width; x++ ) {
            const unsigned int offset = x * 3 +
                y * ( int )pitch / sizeof( float );

            // processing single pixel at specified offset
        }
    }
}
```

Host code: Single function call will process whole image.

Device code:

```c
__global__ void kernelUpdateLRGB( float* g_img, size_t pitch,
    bool init )
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = 3 * x + y * pitch / sizeof( float );

    // processing single pixel at specified offset
}
```

Device code: Function will be executed by many threads in parallel. Every thread will process single pixel.

\( \text{Host} = \text{CPU}, \text{Device} = \text{GPU}, \text{Kernel} = \text{function called from the host that runs on the device} \)
CUDA-Enabled Hardware

Testing platforms

A  HP xw9400 Workstation
   2 × Dual-Core AMD Opteron
   Processor 2218
   2.60 GHz, 4.00 GB of RAM
   PCI Express 1.0 x16 slot
   MS Windows XP Prof x64 Edition

B  Lower Class Office PC
   Single-Core Intel Pentium 4
   With enabled Hyper-Threading Technology
   2.60 GHz, 1.00 GB of RAM
   MS Vista Business

GeForce GTX 260 OC
NVIDIA GPU GT200
27 Multiprocessors
(216 scalar cores @ 1242 MHz)
896 MB GDDR3
448-bit Memory interface
PCI Express 2.0 x16
Compute Capability 1.3
Kernel Run-times

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>host</td>
<td>device</td>
</tr>
<tr>
<td>4 threads</td>
<td>1 thread</td>
<td>8 x 8</td>
</tr>
<tr>
<td>5</td>
<td>31.91 ms</td>
<td>128.34 ms</td>
</tr>
<tr>
<td>4</td>
<td>26.82 ms</td>
<td>109.76 ms</td>
</tr>
<tr>
<td>3</td>
<td>21.78 ms</td>
<td>89.24 ms</td>
</tr>
<tr>
<td>2</td>
<td>17.03 ms</td>
<td>69.13 ms</td>
</tr>
</tbody>
</table>

In case of CPU (host) code OpenMP was used for parallelization.

We get $128.34 / (0.91 + 2.73 + 1.34) \approx 26 \times \text{speed up by CUDA.}$

In case of 4 threads we get $\approx 6 \times \text{speed up.}$

Measured on the following video sequence:
320 x 240 pix, 20 fps, 60 sec. length
Results
Conclusion

- MoG is a reliable method for background subtraction.
- Our CUDA implementation outperforms our reference CPU code.
  - Decrease number of used registers -> higher occupancy -> better latency hiding
  - Use streams (overlapping).
  - Use shared memory (parallel data cache) to minimize global loads.
  - Revise coalescing during global memory accesses. (Half warp (i.e. 16 threads) accesses contiguous regions of device global memory.)
Thank you for attention.