CUDA GPGPU

Ivo Ihrke  Tobias Ritschel  Mario Fritz
Today

• 3 Topics
  – Intro to CUDA (NVIDIA slides)
    • How is the GPU hardware organized?
    • What is the programming model?
  – Simple Kernels
    • Hello World
    • Gaussian Filtering (again)
  – Advanced Application: Horn & Schunck Optical Flow
    • variational method (iterative updates)
    • use of device functions
    • Interoperation with OpenGL for visualization
CODE

Programming breakout
Minexample (minexample.cu)
CODE

Programming breakout
Gaussian Filtering (gauss.cu)
Optical Flow – Horn & Schunck’81

- Apparent motion of brightness patterns in an image sequence (typically two frames)
- For images: \( \vec{u}(\vec{x}): \mathbb{R}^2 \to \mathbb{R}^2 \), is a vector valued fct.
- Often visualized as vector field or color coded
Example Yosemite sequence

Flow field (middlebury coding)

Flow field (IPOL coding)

left

right

middlebury coding

IPOL coding
Example Implementation

- [http://demo.ipol.im/demo/sm_horn_schunck/](http://demo.ipol.im/demo/sm_horn_schunck/)

- “classical” algorithm and multi-scale version available

- In class: discuss classical algorithm
  - can compute flow field for small displacements (1-2 pixels)

- Assignment: multi-scale version
  - Can handle arbitrary displacements for objects of sufficient size
Optical Flow - Derivation

• Assume a video

\[ I(\mathbf{x}, t) : \mathbb{R}^3 \rightarrow \mathbb{R} \]

• Brightness constancy implies

\[ I(\mathbf{x} + \mathbf{u}(\mathbf{x}, t), t) = I(\mathbf{x}, t + 1) \]

• Look at one particular time step with flow vectors \( \mathbf{u} = (u_x, u_y) \)
  - perform Taylor expansion of \( I(\mathbf{x}, t + 1) \):

\[ I(\mathbf{x}, t + 1) \approx I(\mathbf{x}, t) + \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} + O(\nabla^2) \]
  - implies

\[ \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} = 0 \]
  - Alternative form: \( \nabla I \cdot \mathbf{u} + \frac{\partial I}{\partial t} = 0 \)
Variational methods

- Variational methods work as follows

\[
\min_{\tilde{u}} \int_{\Omega} D(\tilde{u}) + \alpha R(\tilde{u}) \, d\tilde{x}
\]

- Data term measures quality of fit to the data
- Regularization term measures some prior knowledge about \( \tilde{u} \)
- \( \alpha \) is a user parameter
Horn&Schunck flow

• Practical minimization of

\[
\min_u \int_\Omega L(x, u(x), u'(x)) \, dx
\]

• via solution of Euler-Lagrange equation (1D case)

\[
\frac{\partial L}{\partial u} + \frac{\partial}{\partial x} \frac{\partial L}{\partial u/\partial x} = 0
\]
Horn&Schunck flow

• Horn & Schunck’s variational formulation

\[
\min_{\bar{u}} \int_{\Omega} (\nabla I \cdot \bar{u} + \frac{\partial I}{\partial t})^2 + \alpha^2 (|\nabla u_x|^2 + |\nabla u_y|^2) \, d\tilde{x}
\]

data term  regularization term

• Data term measures fit to brightness constancy constraint
• Regularization term measures smoothness of \( \bar{u} \)
• \( \alpha \) is a user parameter
Horn&Schunck flow

- Euler-Lagrange Equations for multiple functions of multiple variables

\[
I[f_1, f_2, \ldots, f_m] = \int_{\Omega} \mathcal{L}(x_1, \ldots, x_n, f_1, \ldots, f_m, f_1, \ldots, f_{1,n}, \ldots, f_{m,1}, \ldots, f_{m,n}) \, dx; \quad f_{j,i} := \frac{\partial f_j}{\partial x_i}
\]

\[
\frac{\partial L}{\partial f_1} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \frac{\partial L}{\partial f_{1,i}} = 0
\]

\[
\frac{\partial L}{\partial f_2} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \frac{\partial L}{\partial f_{2,i}} = 0
\]

\[
\cdots \quad \cdots \quad \cdots
\]

\[
\frac{\partial L}{\partial f_m} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \frac{\partial L}{\partial f_{m,i}} = 0.
\]
Horn&Schunck flow

- Euler-Lagrange equation for Horn&Schunck:

\[
\frac{\partial I}{\partial x} \left( \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} \right) - \alpha^2 \Delta u_x = 0 \\
\frac{\partial I}{\partial y} \left( \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} \right) - \alpha^2 \Delta u_y = 0
\]
Horn & Schunck discretization

- Approximation of $\Delta u_x$

\[
\Delta u_x \approx (\bar{u}_x - u_x)
\]

- With

\[
\bar{u}_x^{i,j} = \frac{1}{12} \left( u_{x_i}^{i-1,j-1} + u_{x_i}^{i-1,j+1} + u_{x_i}^{i+1,j-1} + u_{x_i}^{i+1,j+1} \right) \\
+ \frac{1}{6} \left( u_{x_i}^{i-1,j} + u_{x_i}^{i,j-1} + u_{x_i}^{i+1,j} + u_{x_i}^{i,j+1} \right)
\]
Horn&Schunck discretization

- Easy: these are static derivatives of the images

\[ \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} = -\alpha^2 \Delta u_x = 0 \]

\[ \frac{\partial I}{\partial y} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} = -\alpha^2 \Delta u_y = 0 \]

- In practice: average left and right values for spatial derivatives
Horn & Schunck discretization

• Practical update scheme (iteration count k):

\[ u_{x}^{k+1} = \bar{u}_{x}^{k} - \frac{\partial I}{\partial x} \left( \frac{\partial I}{\partial x} \bar{u}_{x}^{k} + \frac{\partial I}{\partial y} \bar{u}_{y}^{k} + \frac{\partial I}{\partial t} \right) \left( \alpha^2 + \left( \frac{\partial I}{\partial x} \right)^2 + \left( \frac{\partial I}{\partial y} \right)^2 \right) \]

\[ u_{y}^{k+1} = \bar{u}_{y}^{k} - \frac{\partial I}{\partial y} \left( \frac{\partial I}{\partial x} \bar{u}_{x}^{k} + \frac{\partial I}{\partial y} \bar{u}_{y}^{k} + \frac{\partial I}{\partial t} \right) \left( \alpha^2 + \left( \frac{\partial I}{\partial x} \right)^2 + \left( \frac{\partial I}{\partial y} \right)^2 \right) \]
CODE

Programming breakout
Horn&Schunck Optical Flow (oflow.cu)
Debugging: CUDA – OpenGL interaction

• Often reading back and saving out the results is inconvenient
  – Especially for iterative schemes, difficult
    • Browsing of iterations
    • Looking at relevant data
    • Finding the right scale / compression of the data under before saving

• Immediate feedback useful
• Goal: combine CUDA computation with OpenGL visualization without reading back the data from the GPU
• Solution: PixelBufferObjects (PBOs)
  – Can be shared between CUDA and OpenGL
  – Catch: Special buffers owned by OpenGL, both APIs cannot use them simultaneously – need switching
  – Mapping and Unmapping of BufferObjects
    • After mapping: CUDA owns the buffer
    • After Unmapping: OpenGL owns the buffer
CODE

Programming breakout
Horn&Schunck Optical Flow with OpenGL feedback
(oflow_v3.cu)
Assignment – Multi-scale Horn & Schunck

• We can now handle small displacements
• Main idea: *every* displacement is small on an appropriate scale (remember: scale space)
  – 2 options:
    • incrementally more smoothed versions of the images at the same resolution
    • Smoothed and sub-sampled images (less pixels – less work, but more difficult)
  – Important: incremental computation of large scale flow vector (from coarse to fine)
Assignment – Multi-scale Horn&Schunkck

• Algorithm: - Input $I_1, I_2$
  
  - $u_{acc} \rightarrow 0$ (accumulated flow)
  - $v_{acc} \rightarrow 0$ (accumulated flow)

  for $k = 1$ to $K$
  
    - compute $\hat{I}_2$ by warping $I_2$ with $(u_{acc}, v_{acc})$
    - pre-smooth $I_1$ and $\hat{I}_2$ with $\sigma_k$
    - compute $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$, $\frac{\partial I}{\partial t}$
    - $u \rightarrow 0$, $v \rightarrow 0$
    - compute single scale Horn&Schunkck flow on smoothed images

  $u_{acc} += u$; $v_{acc} += v$

end

Note: $u = u_x$ $v = u_y$
Assignment: Adaptive stopping with convergence criterion

• Stop the main Horn-Schunck iteration after convergence instead of using a fixed number of iterations

• Condition for inner iteration index $j$:

\[
\frac{1}{N} \sum_{\tilde{x}} (u_{x}^{j+1}(\tilde{x}) - u_{x}^{j}(\tilde{x}))^2 + (u_{y}^{j+1}(\tilde{x}) - u_{y}^{j}(\tilde{x}))^2 < \varepsilon
\]

• Algorithm modification: need to sum over all elements
  – Easy solution: download to CPU, compute there (expensive)
  – Better solution: compute per block on GPU (best with shared memory), use atomic operations to sum all block results, download sum (needs atomicAdd)
  – Best solution (?): parallel block sum on GPU (shared mem) + atomicAdd
## Common Error Messages and Likely Reasons

**cudaErrorUnknown**
This indicates that an unknown internal error has occurred.

- a very popular one: most often memory access out of bounds
- try cuda-memcheck

**cudaErrorLaunchTimeout**
This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property `kernelExecTimeoutEnabled` for more information. The device cannot be used until `cudaThreadExit()` is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

- this often happens when using cuda-memcheck
- driver may lump together many kernel calls, try reducing #iterations or similar measures to reduce computational burden

**cudaErrorInvalidDeviceFunction**
The requested device function does not exist or is not compiled for the proper device architecture.

- this happens when compiling with incompatible architecture and code settings
- E.g. (nvcc --arch compute_20 --code sm_21) – try removing options

**cudaErrorLaunchFailure**
An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until `cudaThreadExit()` is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

- things are somehow incorrectly set up – something is pretty wrong in this case
- Try creating a minimal example that shows the behavior, figure out the reason for misconfiguration
## Compute Capabilities

Table 9: Feature Support per Compute Capability

<table>
<thead>
<tr>
<th>Feature Support</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Unlisted features are supported for all compute capabilities)</td>
<td>1.0</td>
</tr>
<tr>
<td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td>
<td>No</td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td>
<td>No</td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Warp Vote Functions)</td>
<td>No</td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature Support</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Unlisted features are supported for all compute capabilities)</td>
<td>1.0</td>
</tr>
<tr>
<td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td>
<td></td>
</tr>
<tr>
<td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td>
<td></td>
</tr>
<tr>
<td>__ballot() (Warp Vote Functions)</td>
<td>No</td>
</tr>
<tr>
<td>__threadfence_system() (Memory Fence Functions)</td>
<td></td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)</td>
<td></td>
</tr>
<tr>
<td>Surface functions (Surface Functions)</td>
<td></td>
</tr>
<tr>
<td>3D grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Funnel shift (see reference manual)</td>
<td>No</td>
</tr>
<tr>
<td>GPU</td>
<td>Compute Capability</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>GeForce GTX 690</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 680</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 670</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 660 Ti</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 660</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 650 Ti</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 650</td>
<td>3.0</td>
</tr>
<tr>
<td>GeForce GTX 560 Ti</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GTX 550 Ti</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GTX 560</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GTX 550</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GTX 550*</td>
<td>2.1</td>
</tr>
<tr>
<td>GeForce GTX 590</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 580</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 570</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 470</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 465</td>
<td>2.0</td>
</tr>
<tr>
<td>GeForce GTX 295</td>
<td>1.3</td>
</tr>
<tr>
<td>GeForce GTX 285</td>
<td>1.3</td>
</tr>
<tr>
<td>GeForce GTX 285 for Mac</td>
<td>1.3</td>
</tr>
<tr>
<td>GeForce GTX 280</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Some Differences to the IPOL implementation

• Images are in the range [0..1] (not [0..255])
  – this affects the choice of $\alpha$

• Warp strategy and implementation of multi-scale scheme are slightly different (no new equations, use of out-of-the box classic Horn-Schunck at every scale)

• Continuous decrease of $\alpha$ with decreasing $\sigma_k$
Parallel Sum

• Not an easy per-pixel operation

• Reduce operation

• Common for control flow problems

--global--
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory, padding // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements. See attached figure.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s) x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 now holds the sum of all input values // to this block. Have it add that sum to the running total
    if( tid == 0 ) atomicAdd(total, x[tid]);
}
Parallel Sum

Parallel Sum Reduction Tree

\[ \text{FIG 11} \]

[Nickolls’08]
AtomicAdd (for floats)

• Needs Compute capability 2.0
  – nvcc -arch compute_20 -code sm_20