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)

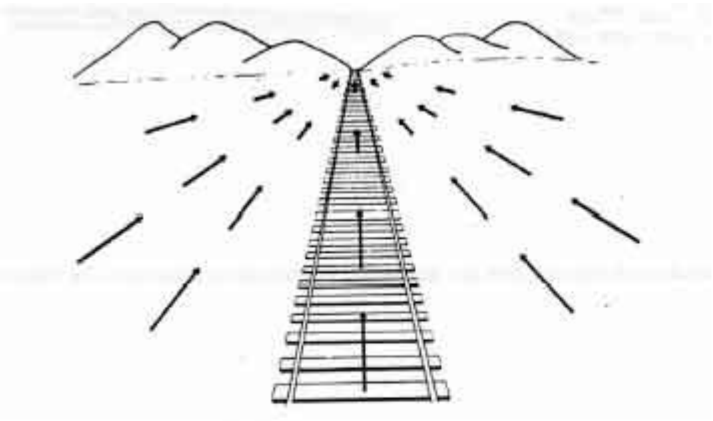
CODE

Programming breakout

Gaussian Filtering (gauss.cu)

Optical Flow – Horn & Schunck'81

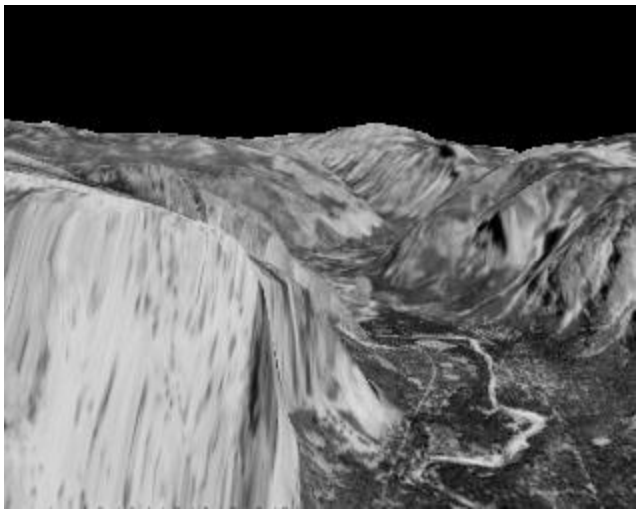
- Apparent motion of brightness patterns in an image sequence (typically two frames)
- For images: $\vec{u}(\vec{x}): R^2 \rightarrow R^2$, is a vector valued fct.
- Often visualized as vector field or color coded



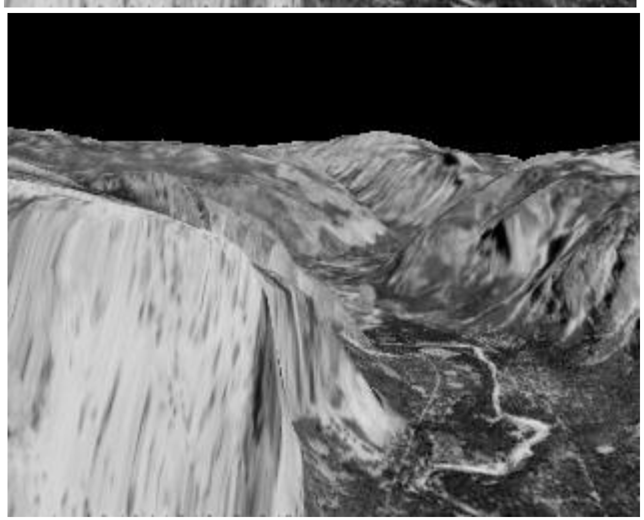
Optical Flow as seen by a person at the back of a train

Example Yosemite sequence

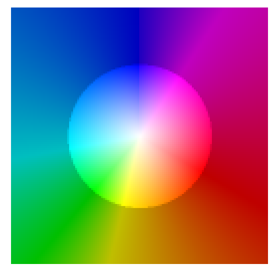
left

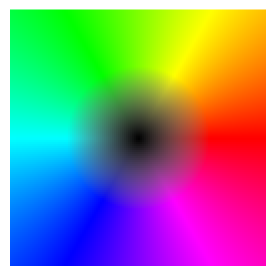


right



middlebury coding





IPOL coding

Flow field (middlebury coding)

Flow field (IPOL coding)

Example Implementation

- 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

$$I(\vec{x}, t): R^3 \rightarrow R$$

- Brightness constancy implies

$$I(\vec{x} + \vec{u}(\vec{x}, t), t) = I(\vec{x}, t + 1)$$

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

$$I(\vec{x}, t + 1) \approx I(\vec{x}, t) + \frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} + O(\nabla^2)$$

  - implies

$$\frac{\partial I}{\partial x} u_x + \frac{\partial I}{\partial y} u_y + \frac{\partial I}{\partial t} = 0$$

  - Alternative form: $\nabla I \cdot \vec{u} + \frac{\partial I}{\partial t} = 0$

Variational methods

- Variational methods work as follows

$$\min_{\vec{u}} \int_{\Omega} D(\vec{u}) + \alpha R(\vec{u})\, d\vec{x}$$

data term regularization term

- Data term measures quality of fit to the data
- Regularization term measures some prior knowledge about $\vec{u}$
- α is a user parameter

Horn&Schunck flow

- Practical minimization of

$$\min_u \int_\Omega L(x, u(x), u'(x))\, d\vec{x}$$

- 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

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

data term   regularization term

- Data term measures fit to brightness constancy constraint
- Regularization term measures smoothness of $\vec{u}$
- α is a user parameter

Horn&Schunck flow

- Euler-Lagrange Equations for multiple functions of multiple variables

$$I[f_1, f_2, \dots, f_m] = \int_\Omega \mathcal{L}(x_1, \dots, x_n, f_1, \dots, f_m, f_{1,1}, \dots, f_{1,n}, \dots, f_{m,1}, \dots, f_{m,n})\, \mathrm{d}\mathbf{x}\,;\ \ f_{j,i} := \frac{\partial f_j}{\partial x_i}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial f_1} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{1,i}} &= 0 \\
\frac{\partial \mathcal{L}}{\partial f_2} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{2,i}} &= 0 \\
\dots \quad \dots \qquad \dots & \\
\frac{\partial \mathcal{L}}{\partial f_m} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{m,i}} &= 0.
\end{aligned}$$

Horn&Schunck flow

- Euler-Lagrange equation for Horn&Schunck:

$$\frac{\partial I}{\partial x}\left(\frac{\partial I}{\partial x}u_x + \frac{\partial I}{\partial y}u_y + \frac{\partial I}{\partial t}\right) - \alpha^2 \Delta u_x = 0$$

$$\frac{\partial I}{\partial y}\left(\frac{\partial I}{\partial x}u_x + \frac{\partial I}{\partial y}u_y + \frac{\partial I}{\partial t}\right) - \alpha^2 \Delta u_y = 0$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial f_1} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{1,i}} &= 0 \\
\frac{\partial \mathcal{L}}{\partial f_2} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{2,i}} &= 0 \\
\cdots \quad \cdots \quad & \cdots \\
\frac{\partial \mathcal{L}}{\partial f_m} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial f_{m,i}} &= 0.
\end{aligned}$$

Horn&Schunck discretization

- Approximation of Δu_x

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

- With

$$\bar{u}_x{}^{i,j} = \frac{1}{12}\left(u_x{}^{i-1,j-1} + u_x{}^{i-1,j+1} + u_x{}^{i+1,j-1} + u_x{}^{i+1,j+1}\right) + \frac{1}{6}\left(u_x{}^{i-1,j} + u_x{}^{i,j-1} + u_x{}^{i+1,j} + u_x{}^{i,j+1}\right)$$

1/12	1/6	1/12
1/6	0	1/6
1/12	1/6	1/12

Horn&Schunck discretization

- Easy: these are static derivatives of the images

$$\frac{\partial I}{\partial x}\left(\frac{\partial I}{\partial x}u_x + \frac{\partial I}{\partial y}u_y + \frac{\partial I}{\partial t}\right) - \alpha^2 \Delta u_x = 0$$

$$\frac{\partial I}{\partial y}\left(\frac{\partial I}{\partial x}u_x + \frac{\partial I}{\partial y}u_y + \frac{\partial I}{\partial t}\right) - \alpha^2 \Delta u_y = 0$$

- In practice: average left and right values for spatial derivatives

Horn&Schunck discretization

- Practical update scheme (iteration count k):

$$u_x{}^{k+1} = \bar{u}_x{}^k - \frac{\partial I}{\partial x}\left(\frac{\frac{\partial I}{\partial x}\bar{u}_x{}^k + \frac{\partial I}{\partial y}\bar{u}_y{}^k + \frac{\partial I}{\partial t}}{\alpha^2 + \left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}\right)$$

$$u_y{}^{k+1} = \bar{u}_y{}^k - \frac{\partial I}{\partial y}\left(\frac{\frac{\partial I}{\partial x}\bar{u}_x{}^k + \frac{\partial I}{\partial y}\bar{u}_y{}^k + \frac{\partial I}{\partial t}}{\alpha^2 + \left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}\right)$$

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&Schunck

- Algorithm: - Input I_1, I_2

  uacc → 0 (accumulated flow)
  vacc → 0 (accumulated flow)

  for k = 1 to K

  compute $\widehat{I_2}$ by warping I_2 with (uacc, vacc)

  pre-smooth I_1 and $\widehat{I_2}$ with σ_k

  compute $\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}, \frac{\partial I}{\partial t}$

  u→ 0, v→ 0

  compute single scale Horn&Schunck flow on smoothed images

  uacc+=u; vacc+=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:

$$\frac{1}{N}\sum_{\vec{x}} ({u_x}^{j+1}(\vec{x}) - {u_x}^{j}(\vec{x}))^2 + ({u_y}^{j+1}(\vec{x}) - {u_y}^{j}(\vec{x}))^2 < \varepsilon$$

- 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

Feature Support	**Compute Capability**					
(Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x, 3.0**	**3.5**
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	No	Yes				
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	No		Yes			
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Warp vote functions (Warp Vote Functions)						
Double-precision floating-point numbers	No			Yes		

Feature Support	**Compute Capability**					
(Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x, 3.0**	**3.5**
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	No				Yes	
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())						
__ballot() (Warp Vote Functions)						
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Funnel shift (see reference manual)	No					Yes

GeForce Desktop Products

GPU	Compute Capability
GeForce GTX 690	3.0
GeForce GTX 680	3.0
GeForce GTX 670	3.0
GeForce GTX 660 Ti	3.0
GeForce GTX 660	3.0
GeForce GTX 650 Ti	3.0
GeForce GTX 650	3.0
GeForce GTX 560 Ti	2.1
GeForce GTX 550 Ti	2.1
GeForce GTX 460	2.1
GeForce GTS 450	2.1
GeForce GTS 450*	2.1
GeForce GTX 590	2.0
GeForce GTX 580	2.0
GeForce GTX 570	2.0
GeForce GTX 480	2.0
GeForce GTX 470	2.0
GeForce GTX 465	2.0
GeForce GTX 295	1.3
GeForce GTX 285	1.3
GeForce GTX 285 for Mac	1.3
GeForce GTX 280	1.3

GeForce Notebook Products

GPU	Compute Capability
GeForce GTX 680MX	3.0
GeForce GTX 680M	3.0
GeForce GTX 675MX	3.0
GeForce GTX 675M	2.1
GeForce GTX 670MX	3.0
GeForce GTX 670M	2.1
GeForce GTX 660M	3.0
GeForce GT 650M	3.0
GeForce GT 645M	3.0
GeForce GT 640M	3.0
GeForce GT 640M LE	3.0
GeForce GT 635M	2.1
GeForce GT 630M	2.1
GeForce GT 625M	2.1
GeForce GT 620M	2.1
GeForce 610M	2.1
GeForce GTX 580M	2.1
GeForce GTX 570M	2.1
GeForce GTX 560M	2.1
GeForce GT 555M	2.1
GeForce GT 550M	2.1
GeForce GT 540M	2.1

GeForce Desktop Products

GPU	Compute Capability
GeForce GTX 275	1.3
GeForce GTX 260	1.3
GeForce GT 640	2.1
GeForce GT 630	2.1
GeForce GT 620	2.1
GeForce GT 610	2.1
GeForce GT 520	2.1
GeForce GT 440	2.1
GeForce GT 440*	2.1
GeForce GT 430	2.1
GeForce GT 430*	2.1
GeForce GT 420*	1.0
GeForce GT 240	1.2
GeForce GT 220*	1.2
GeForce 210*	1.2
GeForce GTS 250	1.1
GeForce GTS 150	1.1
GeForce GT 130*	1.1
GeForce GT 120*	1.1
GeForce G100*	1.1
GeForce 9800 GX2	1.1
GeForce 9800 GTX+	1.1

GeForce Notebook Products

GPU	Compute Capability
GeForce GT 525M	2.1
GeForce GT 520MX	2.1
GeForce GT 520M	2.1
GeForce GTX 485M	2.1
GeForce GTX 470M	2.1
GeForce GTX 460M	2.1
GeForce GT 445M	2.1
GeForce GT 435M	2.1
GeForce GT 420M	2.1
GeForce GT 415M	2.1
GeForce GTX 480M	2.0
GeForce GTS 360M	1.2
GeForce GTS 350M	1.2
GeForce GT 335M	1.2
GeForce GT 330M	1.2
GeForce GT 325M	1.2
GeForce GT 240M	1.2
GeForce G210M	1.2
GeForce 310M	1.2
GeForce 305M	1.2
GeForce GTX 285M	1.1
GeForce GTX 280M	1.1

Some Differences to the IPOL implementation

- Images are in the range [0..1] (not [0..255])
  - this affects the choice of α

- 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 α with decreasing σ_k

Parallel Sum

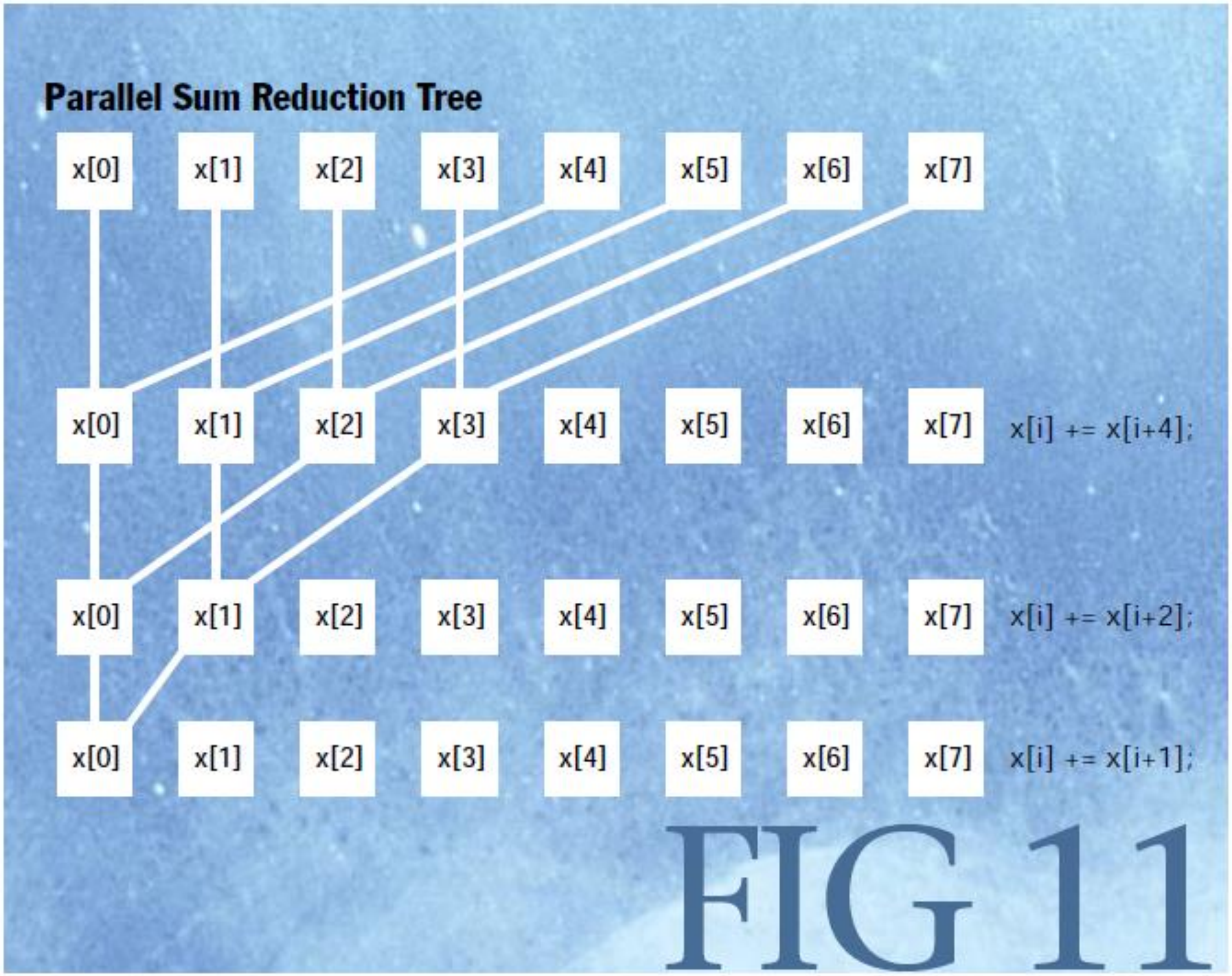
- Not an easy per-pixel operation
- Reduce operation
- Common for control flow problems

```
__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]);
}
```

FIG 10

[Nickolls'08]

Parallel Sum



[Nickolls'08]

AtomicAdd (for floats)

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