NVISION 08
THE WORLD OF VISUAL COMPUTING

Image Processing & Video Algorithms with CUDA

Eric Young & Frank Jargstorff

NVIDIA

introduction

- Image processing is a natural fit for data parallel processing
  - Pixels can be mapped directly to threads
  - Lots of data is shared between pixels
- Advantages of CUDA vs. pixel shader-based image processing
- CUDA supports sharing image data with OpenGL and Direct3D applications

overview

- CUDA for Image and Video Processing
  - Advantages and Applications
- Video Processing with CUDA
  - CUDA Video Extensions API
  - YUVtoARGB CUDA kernel
- Image Processing Design Implications
  - API Comparison of CPU, 3D, and CUDA
- CUDA for Histogram-Type Algorithms
  - Standard and Parallel Histogram
  - CUDA Image Transpose Performance
  - Waveform Monitor Type Histogram

advantages of CUDA

- Shared memory (high speed on-chip cache)
- More flexible programming model
  - C with extensions vs HLSL/GLSL
- Arbitrary scatter writes
- Each thread can write more than one pixel
- Thread Synchronization

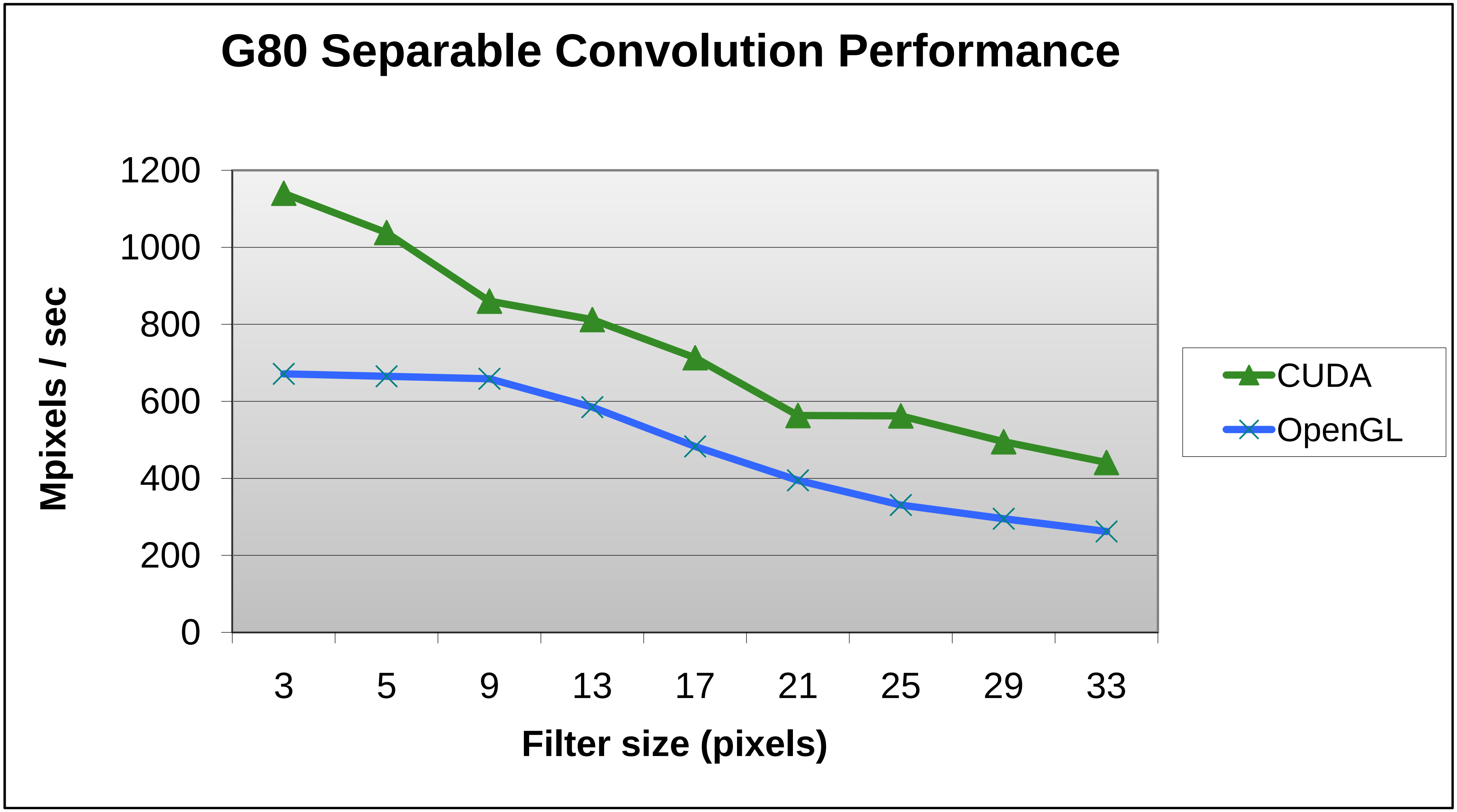
applications

- Convolutions
- Median filter
- FFT
- Image & Video compression
- DCT
- Wavelet
- Motion Estimation
- Histograms
- Noise reduction
- Image correlation
- Demosaic of CCD images (RAW conversion)

shared memory

- Shared memory is fast
  - Same speed as registers
  - Like a user managed data cache
- Limitations
  - 16KB per multiprocessor
  - Can store 64 x 64 pixels with 4 bytes per pixel
- Typical operation for each thread block:
  - Load image tile from global memory to shared
  - Synchronize threads
  - Threads operate on pixels in shared memory in parallel
  - Write tile back from shared to global memory
- Global memory vs Shared
  - Big potential for significant speed up depending on how many times data in shared memory can be reused

convolution performance



separable convolutions

- Filter coefficients can be stored in constant memory
- Image tile can be cached to shared memory
- Each output pixel must have access to neighboring pixels within certain radius *R*
- This means tiles in shared memory must be expanded with an apron that contains neighboring pixels
- Only pixels within the apron write results
  - The remaining threads do nothing

tile apron

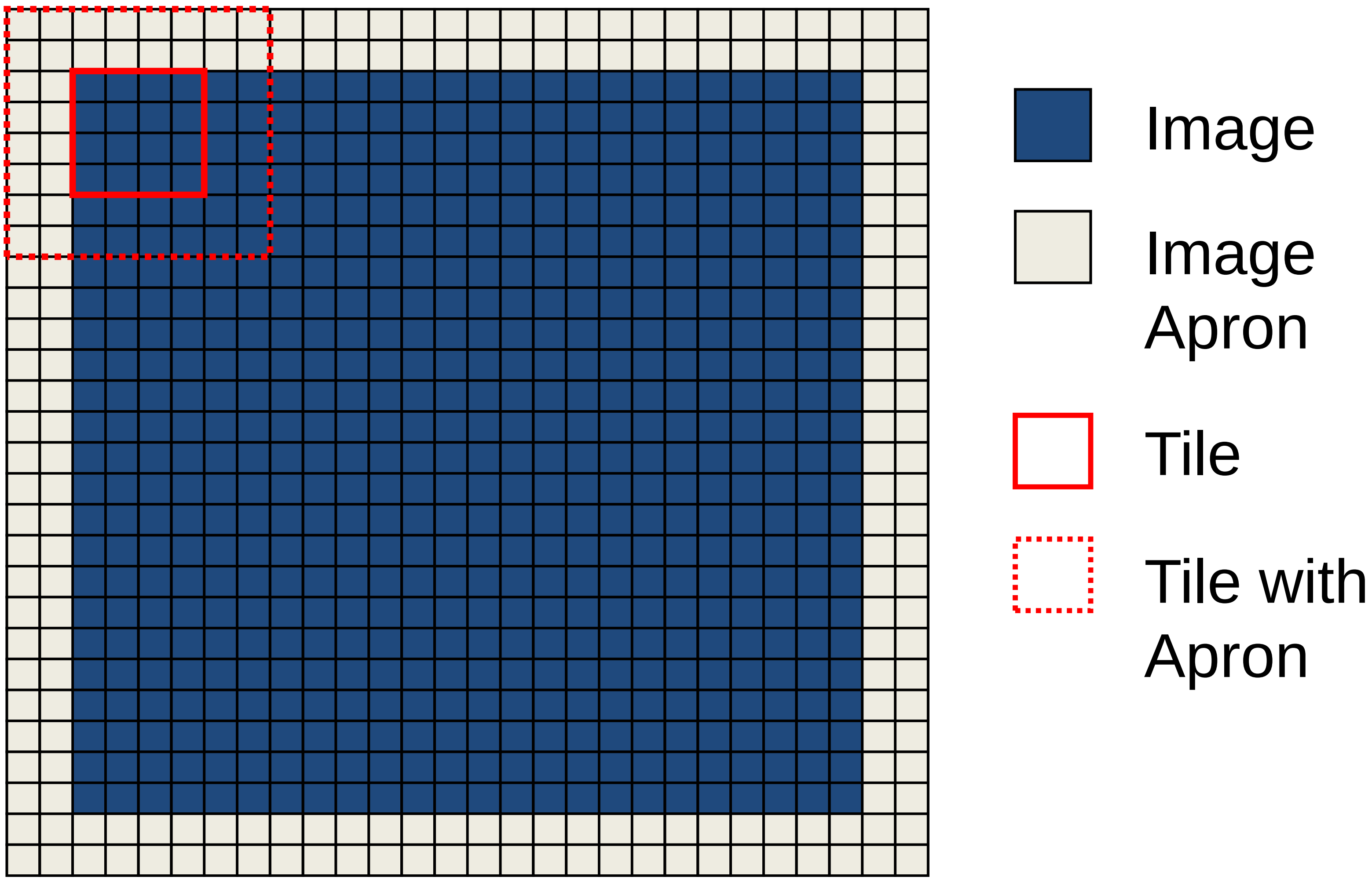


image processing with CUDA

- How does image processing map to the GPU?
  - Image Tiles ⟷ Grid/Thread Blocks
  - Large Data ⟷ Lots of Memory BW
  - 2D Region ⟷ Shared Memory (cached)

define tile sizes

```
#define TILE_W   16
#define TILE_H   16
#define R         2       // filter radius
#define D        (R*2+1)  // filter diameter
#define S        (D*D)    // filter size
#define BLOCK_W (TILE_W+(2*R))
#define BLOCK_H (TILE_H+(2*R))
```

simple filter example

```
__global__ void d_filter(int *g_idata, int *g_odata,
                         unsigned int width, unsigned int height)
{
   __shared__ int smem[BLOCK_W*BLOCK_H];
   int x = blockIdx.x*TILE_W + threadIdx.x - R;
   int y = blockIdx.y*TILE_H + threadIdx.y - R;

   // clamp to edge of image
   x = max(0, x);
   x = min(x, width-1);
   y = max(y, 0);
   y = min(y, height-1);

   unsigned int index = y*width + x;
   unsigned int bindex = threadIdx.y*blockDim.y+threadIdx.x;

   // each thread copies its pixel of the block to shared memory
   smem[bindex] = g_idata[index];
   __syncthreads();
```

simple filter example (cont.)

```
// only threads inside the apron will write results
if ((threadIdx.x >= R) && (threadIdx.x < (BLOCK_W-R)) &&
     (threadIdx.y >= R) && (threadIdx.y < (BLOCK_H-R)))
  {
     float sum = 0;
     for(int dy=-R; dy<=R; dy++) {
        for(int dx=-R; dx<=R; dx++) {
           float i = smem[bindex + (dy*blockDim.x) + dx];
           sum += i;
        }
     }
     g_odata[index] = sum / S;
  }
}
```

nVISION 08 THE WORLD OF VISUAL COMPUTING

NVIDIA.

sobel edge detect filter

- Two filters to detect horizontal and vertical change in the image
- Computes the magnitude and direction of edges
- We can calculate both directions with one single CUDA kernel

$$C_{horizontal} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

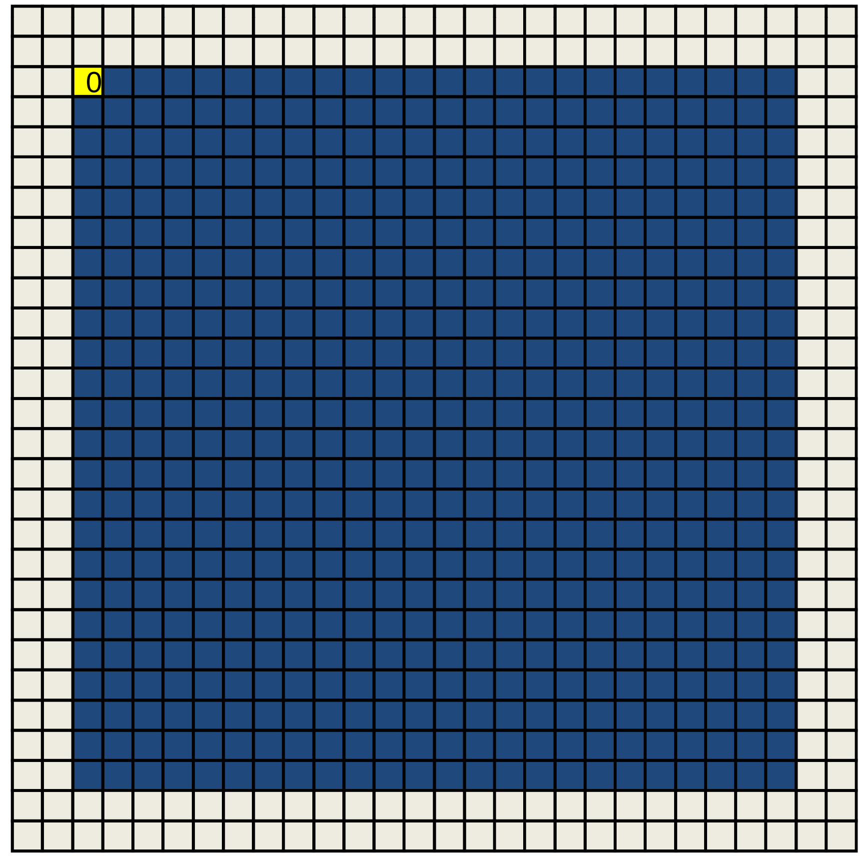
$$C_{vertical} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

$$Direction_{Sobel} = \arctan\left(\frac{G_{vertical}}{G_{horizontal}}\right)$$

sobel edge detect filter

- 3x3 window of pixels for each thread



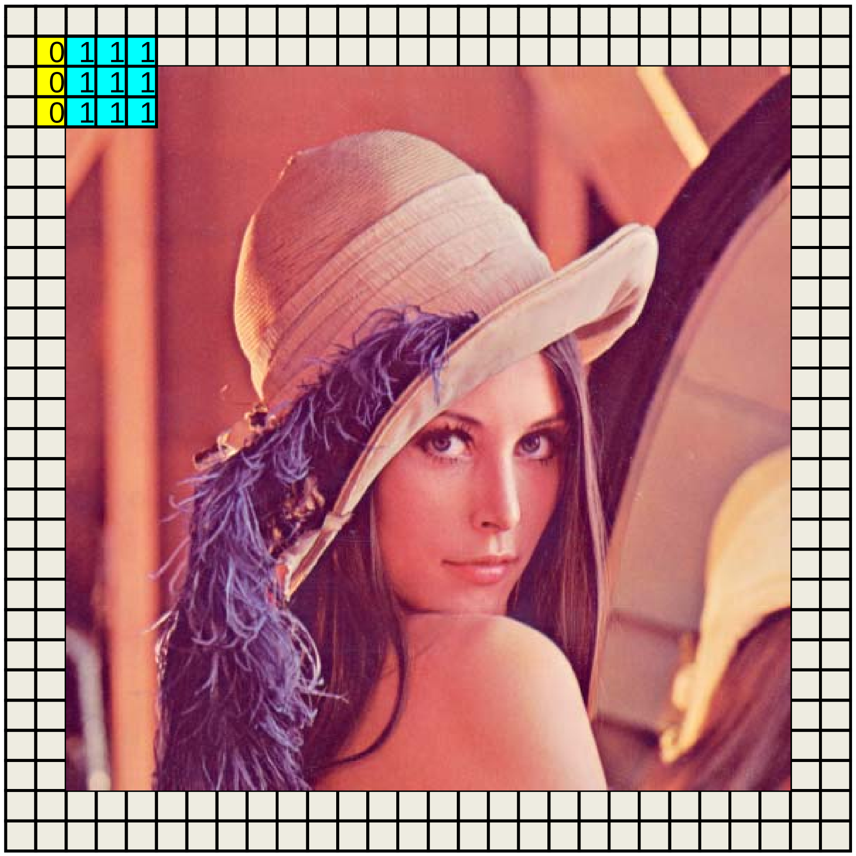
- $$C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

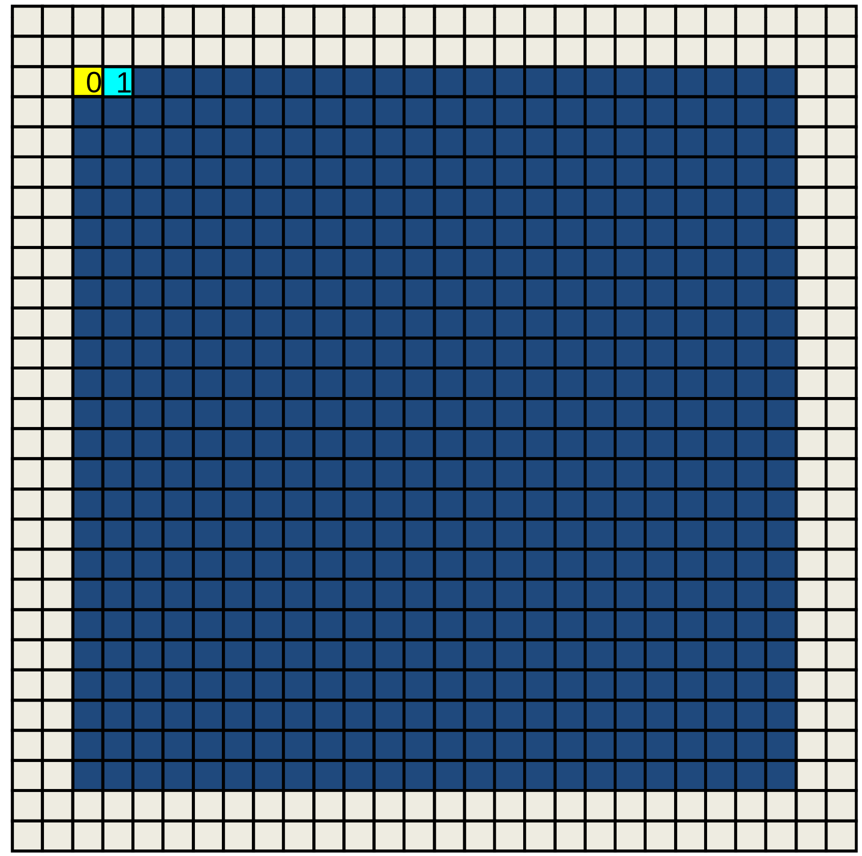
- $$C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

sobel edge detect filter

- 3x3 window of pixels for each thread



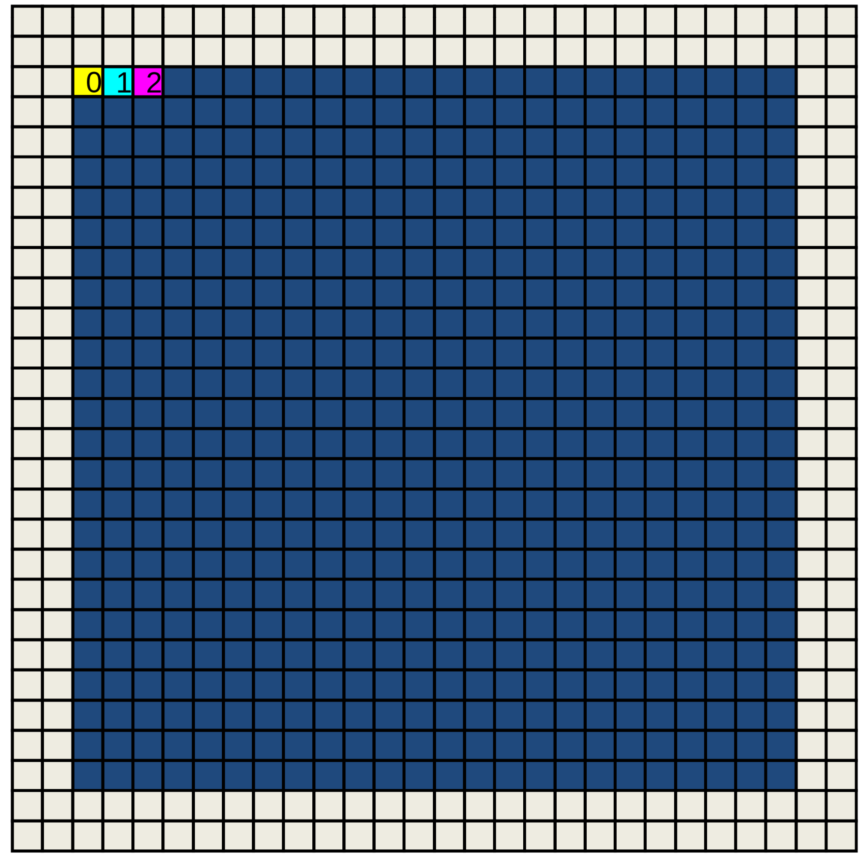


- $$C_{vertical}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$
- $$C_{horizontal}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

sobel edge detect filter

- 3x3 window of pixels for each thread

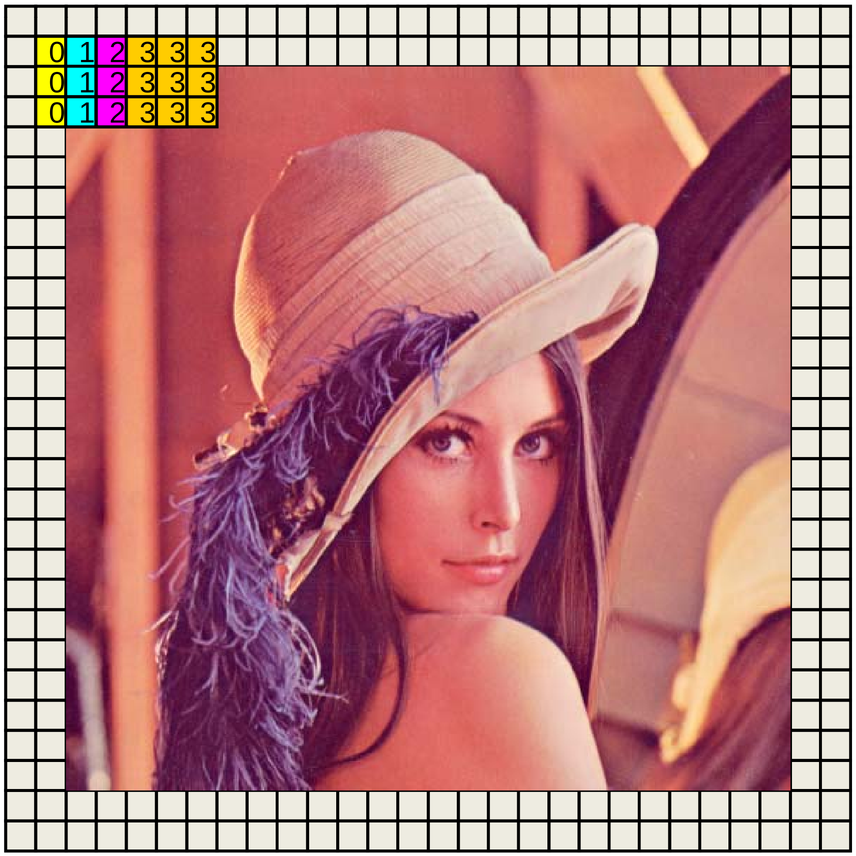


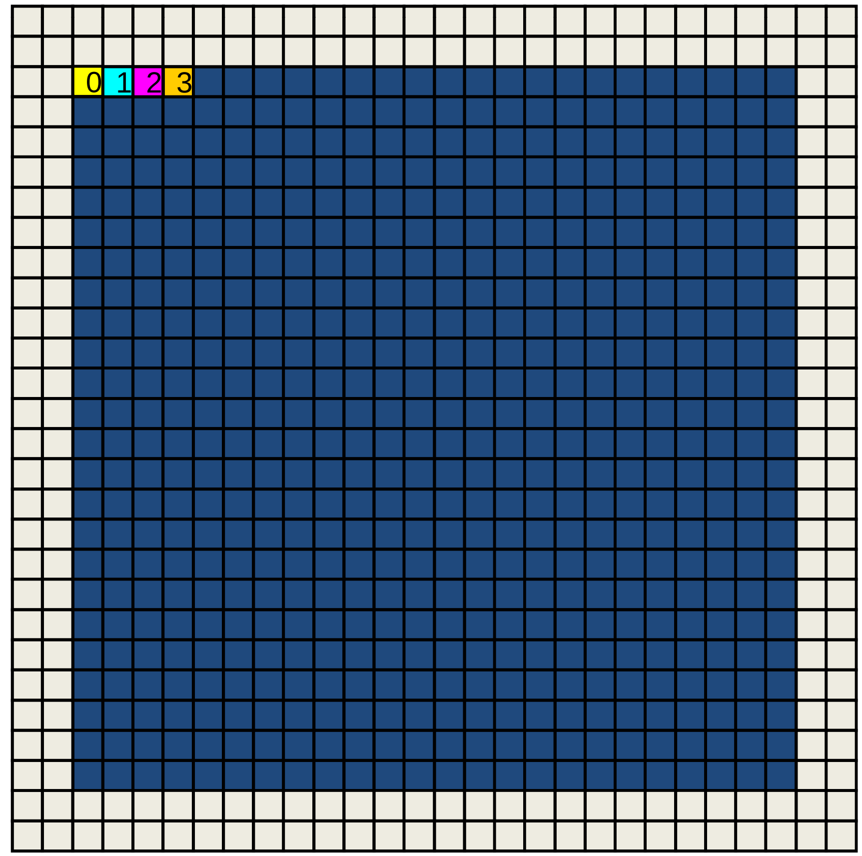
- $C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$
- $C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

sobel edge detect filter

- 3x3 window of pixels for each thread





- $$C_{vertical}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

- $$C_{horizontal}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$$

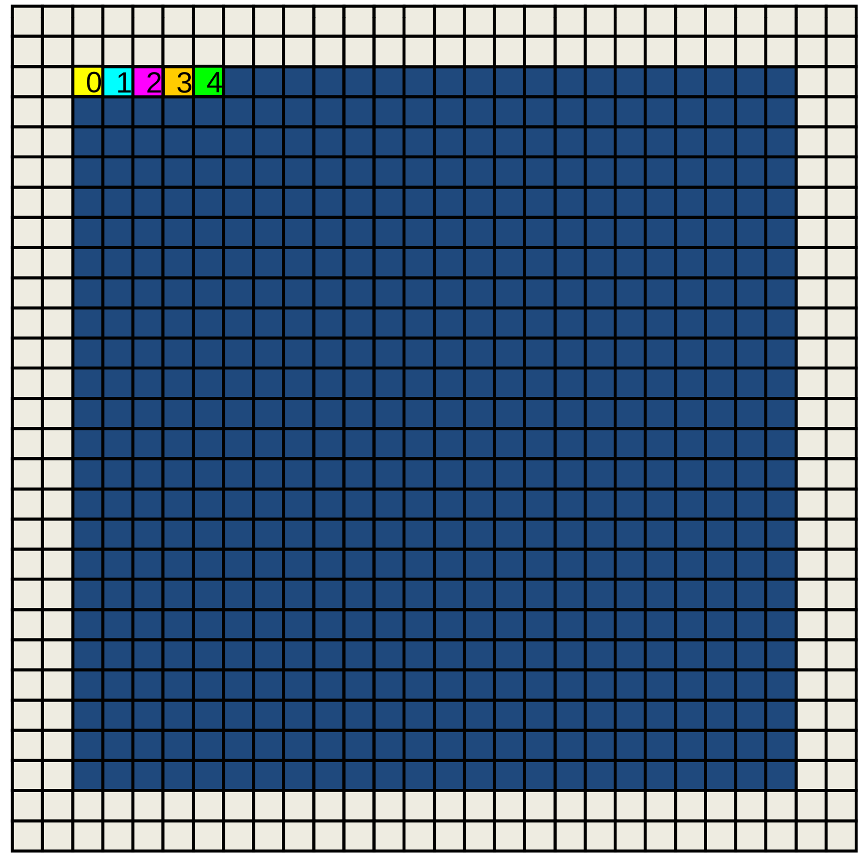
$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

nvision 08
THE WORLD OF VISUAL COMPUTING

NVIDIA.

sobel edge detect filter

- 3x3 window of pixels for each thread



- $$C_{vertical}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$
- $$C_{horizontal}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$$

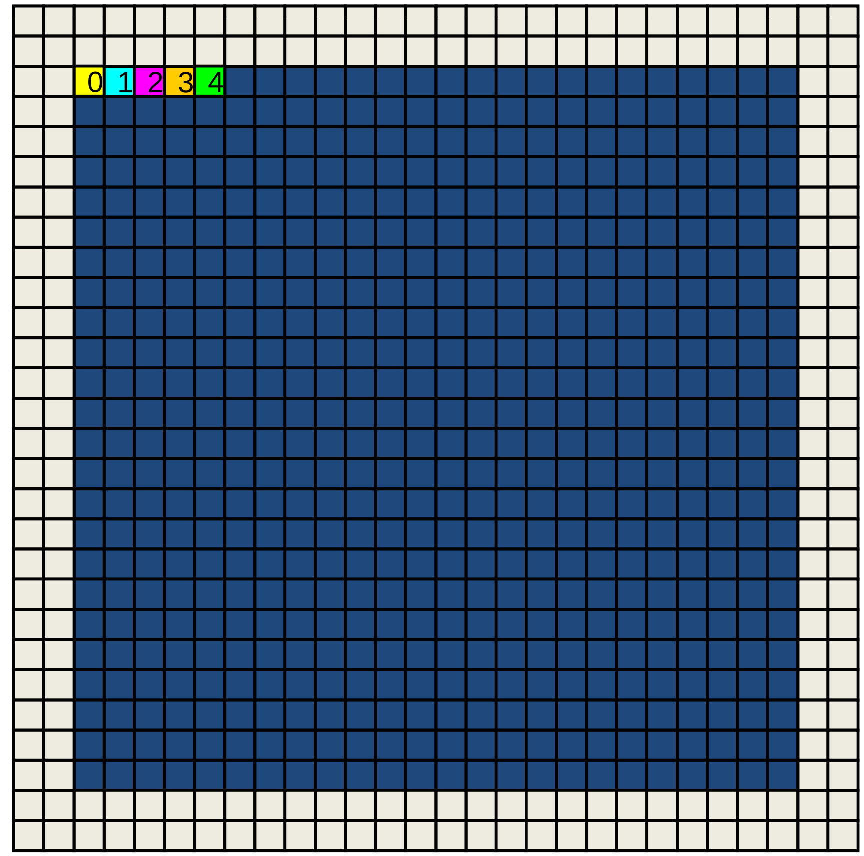
$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

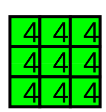
nvision 08
THE WORLD OF VISUAL COMPUTING

NVIDIA.

sobel edge detect filter

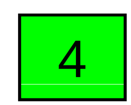
- 3x3 window of pixels for each thread





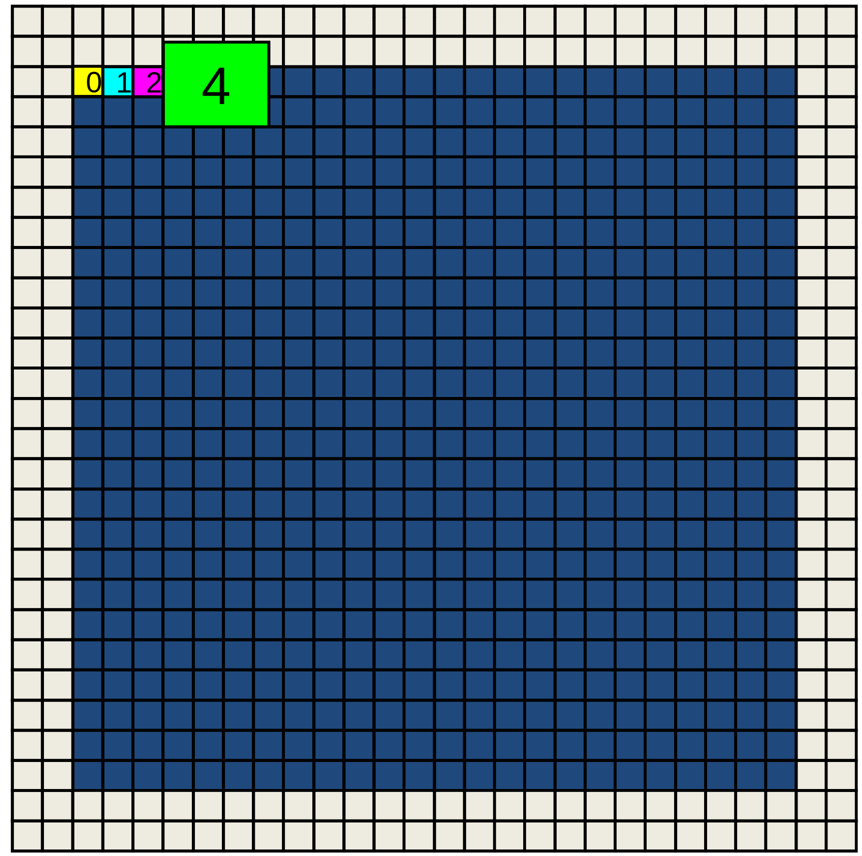
- $$C_{vertical}\begin{pmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{pmatrix} = G_{vertical}$$
- $$C_{horizontal}\begin{pmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{pmatrix} = G_{horizonta}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G^2_{horizontal} + G^2_{vertical}}$$



sobel edge detect filter

- 3x3 window of pixels for each thread



- $$\overset{C_{vertical}}{\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}} = G_{vertical}$$
- $$\overset{C_{horizontal}}{\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}} = G_{horizonta}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

sobel edge detect filter

- 3x3 window of pixels for each thread

- $$C_{vertical}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

- $$C_{horizontal}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizonta}$$

$$Magnitude_{Sobel} = norm \bullet \sqrt{G^2_{horizontal} + G^2_{vertical}}$$

fast box filter

- Allows box filter of any width with a constant cost
  - Rolling box filter
- Uses a sliding window
  - Two adds and a multiply per output pixel
  - Adds new pixel entering window, subtracts pixel leaving
- Iterative Box Filter ≈ Gaussian blur
- Using pixel shaders, it is impossible to implement a rolling box filter
  - Each thread requires writing more than one pixel
- CUDA allows executing rows/columns in parallel
  - Uses tex2D to improve read performance and simplify addressing

fast box filter

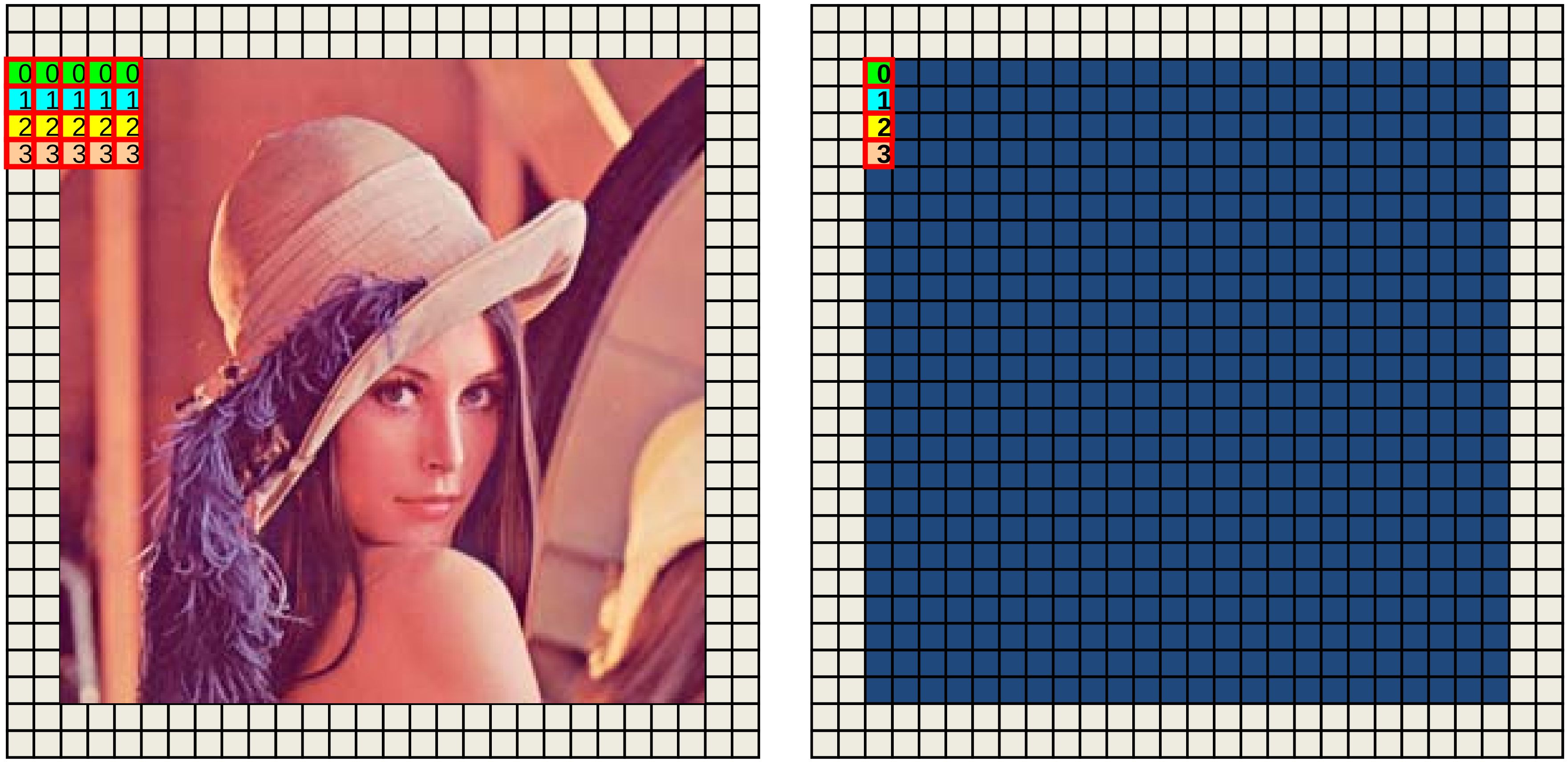
- Separable, two pass filter. First row pass, then column pass

Source Image (input)

Output Result

fast box filter (row pass pixel 0)

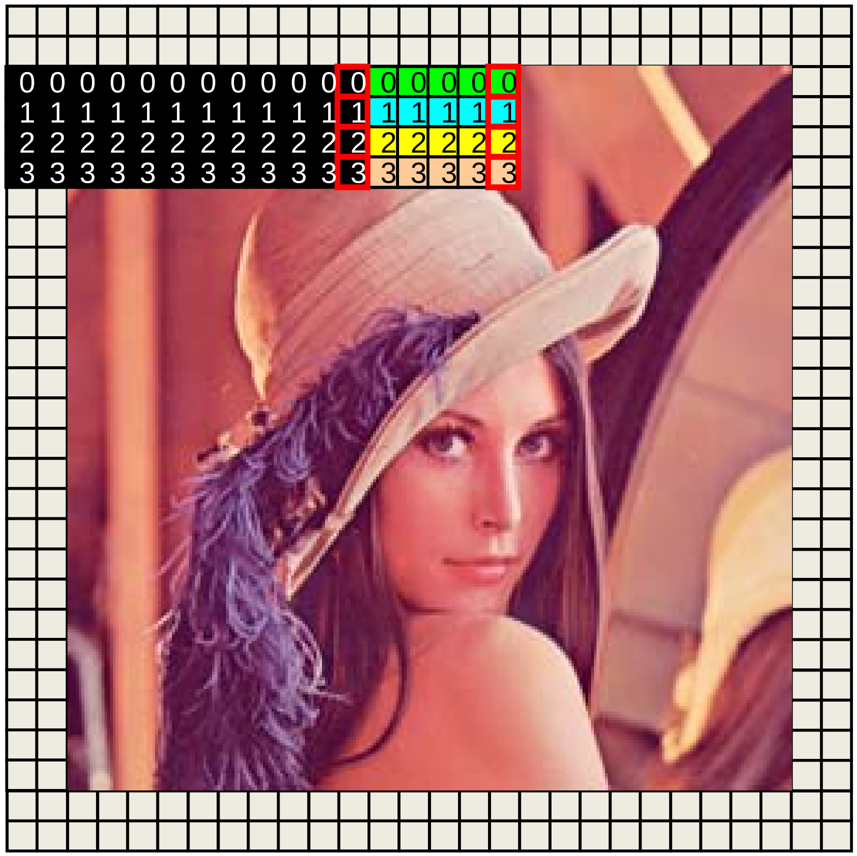
- Assume $r = 2$, each thread works pixels along the row and sums $(2r+1)$ pixels
- Then average $(2r+1)$ pixels and writes to destination $(\ i, j\ )$

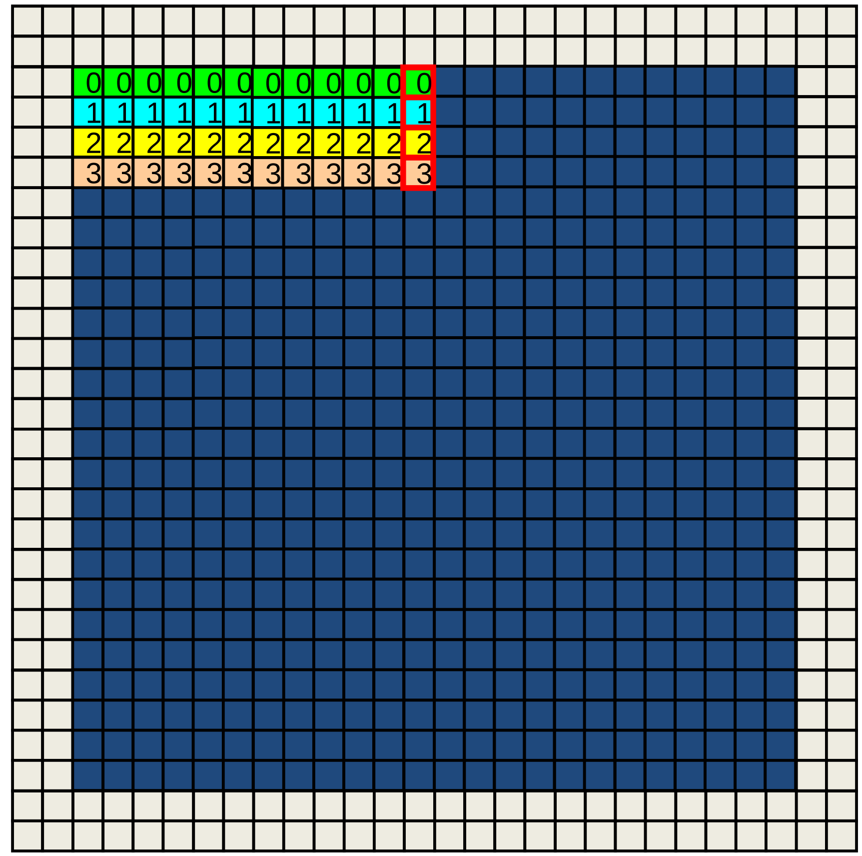


$$\frac{\boxed{3} + \boxed{3} + \boxed{3} + \boxed{3} + \boxed{3}}{(2r + 1)} = \boxed{3}$$

fast box filter (row pass pixel 11)

- Take previous sum from pixel 10, -1 pixel $(i - (r+1), j)$, +1 pixel $(i + (r+1), j)$
- Average $(2r+1)$ pixels and Output to (i, j)



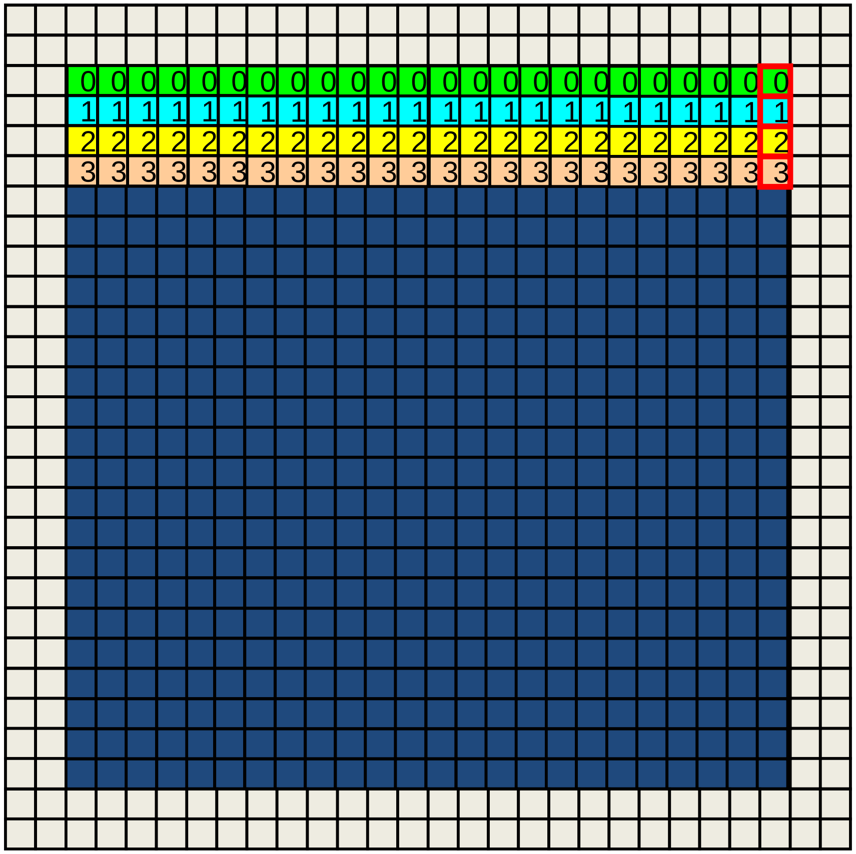


nvision 08 THE WORLD OF VISUAL COMPUTING

NVIDIA.

fast box filter (finish row pass)

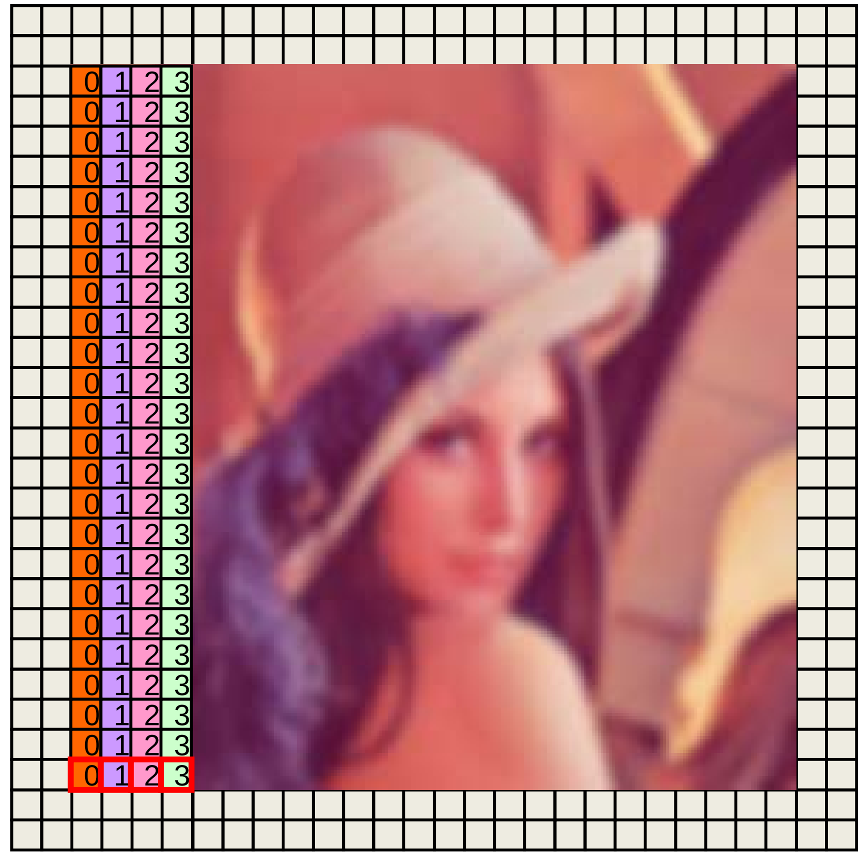
- Each thread continues to iterate until the entire row of pixels is done
- Average then Write to *(i, j)* in destination image
- A single thread writes the entire row of pixels



- Note: Writes are not coalesced
- Solution: Use shared memory to cache results per warp, call __syncthreads(), then copy to global mem to achieve Coalescing

Column Filter Pass (final)

- Threads *(i, j)* read from global memory and sum along the column from row pass image, we get ***Coalesced Reads***
- Compute pixel sums from previous pixel, -1 pixel, +1 pixel
- Average result and Output to *(i, j)*. We get ***Coalesced Writes***



Video processing with CUDA

- GPU has different engines
  - Video Processor (decoding video bitstreams)
  - CUDA (image and video processing)
  - DMA Engine (transfers data host ⟷ GPU)
- CUDA enables developers to access these engines

CUDA Video Extensions

- NVCUVID: video extension for CUDA
- Access to video decoder core requires VP2 (> G80)
- Similar to DXVA API, but will be platform OS independent.
- Interoperates with CUDA (surface exchange) with OpenGL and DirectX
- CUDA SDK 2.0: “cudaVideoDecode”

Video Processor (VP2)

- VP2 is a dedicated video-decode engine on NVIDIA GPUs.
- Supports:
  - MPEG-1, MPEG-2
  - H.264
- Can operate in parallel with GPU's DMA engines and 3D Graphics engine.
- Very low power.

YUV to RGB conversion

- Video Processor
  - Decodes directly to a NV12 surface 4:2:0 that can be mapped directly to a CUDA surface
  - Y samples (bytes) are packed together, followed by interleaved Cb, Cr samples (bytes) sub sampled 2x2

Y0	Y1	Y3	...	...	...	...	
...							
...							
...			...	...	...	...	
U0	V0	U1	V1	...	...	...	
...	...	...	...	...	...	...	...

- CUDA Kernel performs YUV to RGB

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.34413 & -0.714136 \\ 1.0 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

YUV to RGB CUDA kernel

```
__global__ void YUV2RGB(uint32 *yuvi, float *R, float *G, float *B)
{
    float luma, chromaCb, chromaCr;
   // Prepare for hue adjustment (10-bit YUV to RGB)
    luma     = (float)yuvi[0];
    chromaCb = (float)((int32)yuvi[1] - 512.0f);
    chromaCr = (float)((int32)yuvi[2] - 512.0f);

   // Convert YUV To RGB with hue adjustment
   *R    = MUL(luma,      constHueColorSpaceMat[0]) +
           MUL(chromaCb, constHueColorSpaceMat[1]) +
           MUL(chromaCr, constHueColorSpaceMat[2]);
   *G    = MUL(luma,      constHueColorSpaceMat[3]) +
           MUL(chromaCb, constHueColorSpaceMat[4]) +
           MUL(chromaCr, constHueColorSpaceMat[5]);
   *B    = MUL(luma,      constHueColorSpaceMat[6]) +
           MUL(chromaCb, constHueColorSpaceMat[7]) +
           MUL(chromaCr, constHueColorSpaceMat[8]);
}
```

NVCUVID API

- Five entry-points for Decoder object:
  - `cuvidCreateDecoder(...);`
  - `cuvidDestroyDecoder(...);`
  - `cuvidDecodePicture(...);`
  - `cuvidMapVideoFrame(...);`
  - `cuvidUnmapVideoFrame(...);`
- Sample application also uses helper library for Parsing video streams.
  - Provided in binary as part of SDK

cudaVideoDecode Demo

Image Processing (contd.)

- Image Processing:
  - CPU vs. 3D APIs vs. CUDA
  - Design implications
- CUDA for Histogram-Type Algorithms
  - Standard and Parallel Histogram
  - CUDA Image Transpose Performance
  - Waveform Monitor Type Histogram

API Comparison

API	CPU Code	3D API (DX/GL)	CUDA Code
Image Data	Heap Allocated	Texture/FB	CUDA 2D Allocate
Alignment	Matters	n/a	Matters
Cached	Yes	Yes	No
Access (r/w)	Random/random	Random/fixed	Random/random
Access order	Matters (general purpose caches)	Minimized (2D Caching Schemes)	Matters (coalescing, CUDA Array -> Texture HW)
In-Place	Good	n/a	Doesn't matter
Threads	Few per Image (Programmer's decision. But typically one per tile; one tile per core)	One Per Pixel (Consequence of using Pixel Shaders)	One per few Pixels (Programmer's decision. Typically one per input or output pixel)
Data Types	All	32bit-float (half-float maybe)	All (Double precision, native instructions not for all though)
Storage Types	All	Tex/FB Formats	All

Histogram

- Extremely Important Algorithm
  - Histogram Data used in large number of "compound" algorithms:
    - Color and contrast improvements
    - Tone Mapping
    - Color re-quantization/posterize
    - Device Calibration (Scopes see below)

Histogram Performance

- 3D API not suited for histogram computation.
- CUDA Histogram is 300x faster than previous GPGPU approaches:

	64 bins	256 bins
CUDA[1]	6500 MB/s	3676 MB/s
R2VB[2]	22.8 MB/s	42.6 MB/s
CPU[3]	826 MB/s	1096 MB/s

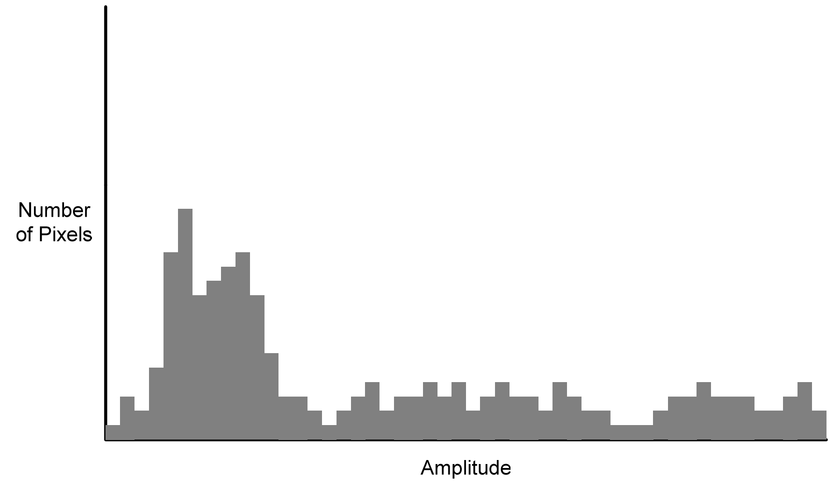
[1] http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#histogram64

[2] Efficient Histogram Generation Using Scattering on GPUs, T. Sheuermann, AMD Inc, I3D 2007

[3] Intel Core 2 @ 2.9 GHz

Histogram Algorithm

- Distribution of intensities/colors in an image
- Standard algoritm:




```
for all i in [0, max_luminance]:
    h[i] = 0;
for all pixel in image:
    ++h[luminance(pixel)]
```

- How to parallelize?

Reinhard HDR Tonemapping operator

Histogram Parallelization

- Subdivide "for-all-pixel" loop
  - Thread works on block of pixels (in extreme, one thread per pixel)
  - Need **`++h[luminance(pixel)]`** to be atomic (global atomics >= compute1_1)
- Breaking up Image I into sub-images I = UNION(A, B):
  - H(UNION(A, B)) = H(A) + H(B)
  - Histogram of concatenation is sum of histograms

Better Parallel Histogram

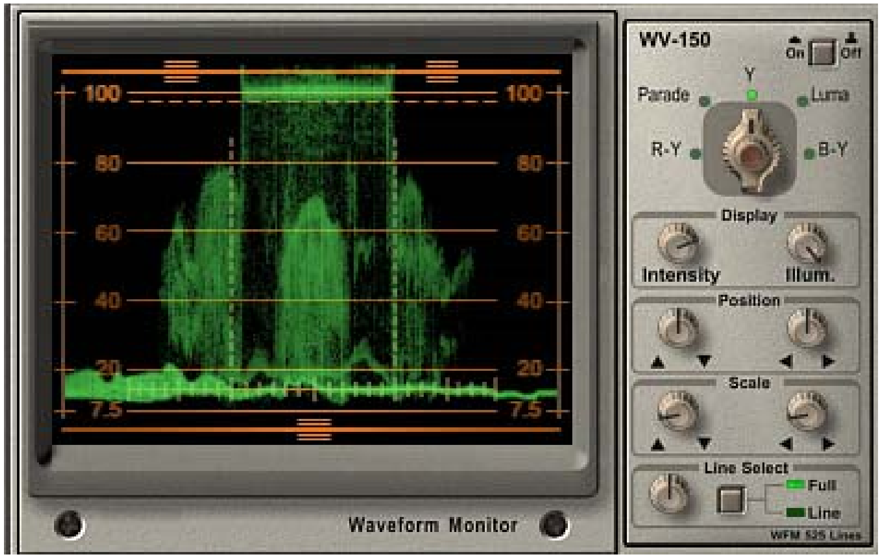
- Have one histogram per thread
  - Memory consumption!
  - Consolidate sub-histograms in parallel (parallel-reduction).
- CUDA:
  - Histograms in shared memory
  - 64bins * 256threads = 16kByte (8bit bins)
  - Approach not feasible for >64 bins

>64bins Parallel Histogram

- Compute capability 1.2 has shared-memory atomic-operations.
- Victor Podlozhnyuk “Manual shared memory per-warp atomics” (CUDA SDK histogram256 sample)
- Have groups of 32 threads work on one sub-histogram, reduce as before.

Real-World Problems with Histograms

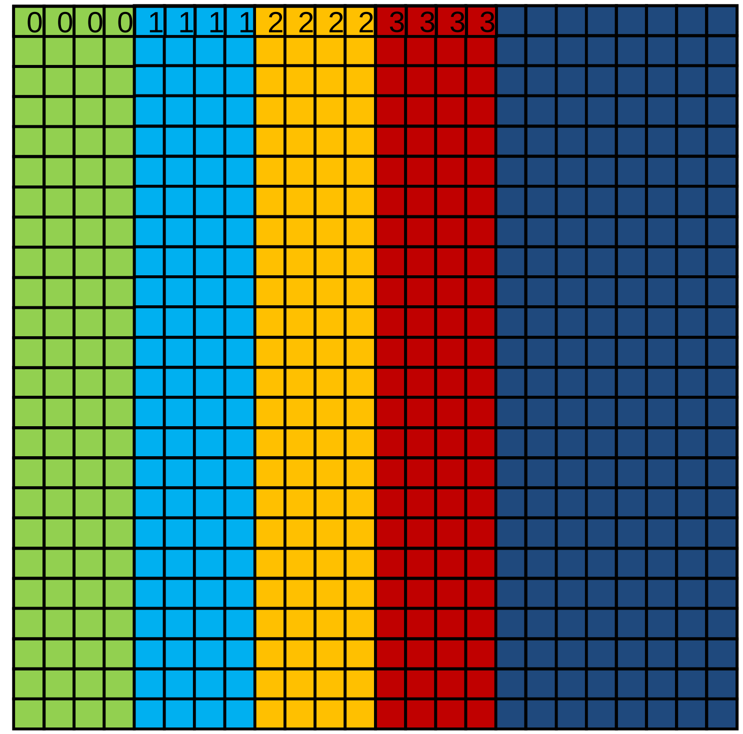
- My attempt to implement a waveform monitor for video using CUDA.
- One histogram per column of the in-put video frame.



- In order to achieve good performance need to solve various memory-access related issues.

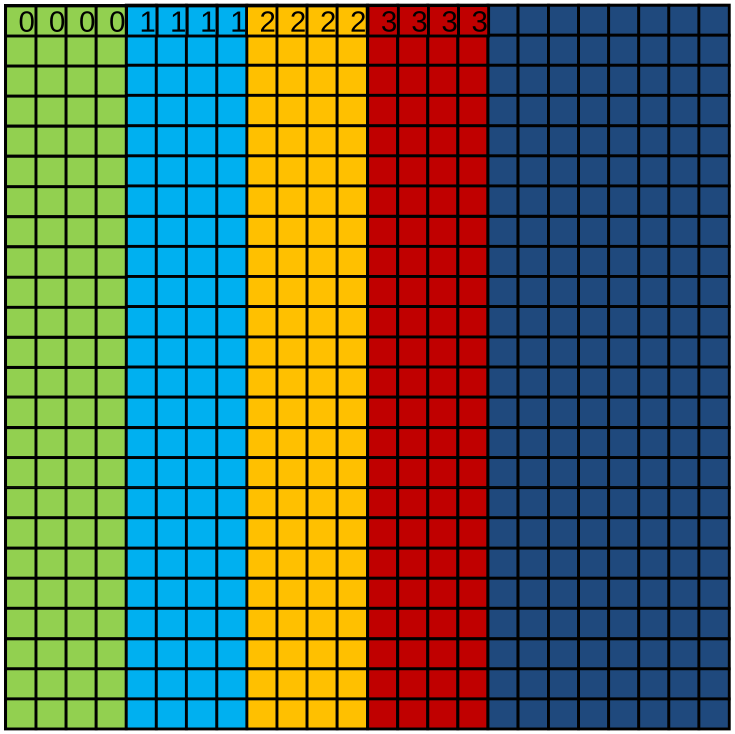
Accessing 8-bit Pixels

- Input video produces Y-channel (luma) as planar data in row-major form.
- Coalescing: 16 threads access 32bit word in subsequent locations.



SMEM Exhausted!

- => 16xM thread blocks => 64 columns processed by each block.
- => 64 histograms in smem:
  64 * 256 * 4 = 64kByte. Max 16 kByte!



Thread-Block Dimensions

- 16xM TB dimensions desirable but impossible for Y-surface read
- Nx32 TB dimensions desirable for “manual atomics” in waveform code
- Solution: Fast transpose input image!
- Also: Result histograms could be copied efficiently (shared->global) horizontally.

Image Transpose

- Problem: Writing a fast Transpose vs. writing Transpose fast.
- Naïve implementation:

```
kernel(char * pi, int si, char * po, int so,
       int w, int h)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (y < w && x < h)
    OUTPUT_PIXEL(y, x) = INPUT_PIXEL(x, y);
}
```

Problems with Naïve Code

- Memory reads AND writes not coalesced because reading/writing bytes, not words.
- Bandwidth: ~3 GByte/s (of ~141max)
- Idea:
  - Need to read (at least) 16 words in subsequent threads.
  - Need to write (at least) 16 words in subsequent threads.

Improved Transpose Idea

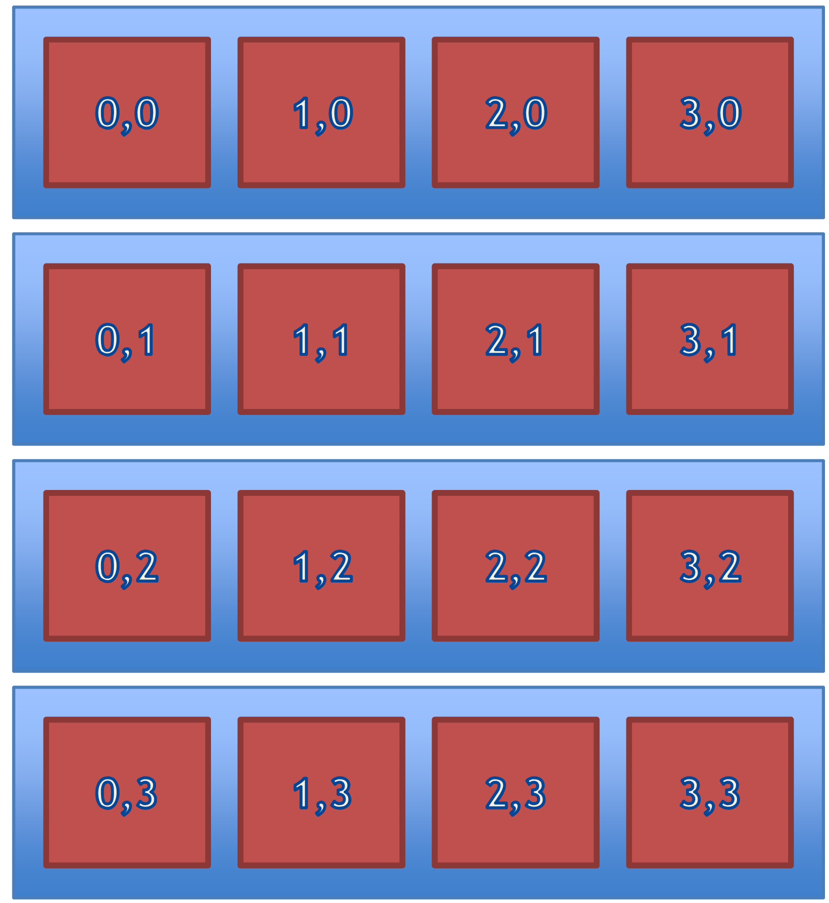
- Subdivide image into “micro-blocks” of 4x4 pixels (16Byte).
- Thread blocks of 16x16 threads.
- Each thread operates on a micro-block.
- Shared memory for micro-blocks: 16 x 16 x 4 x 4 = 4kByte.

Basic Algorithm

- Each thread reads its micro-block into shared memory.
- Each thread transposes its micro-block.
- Each thread writes its micro-block back into global memory.

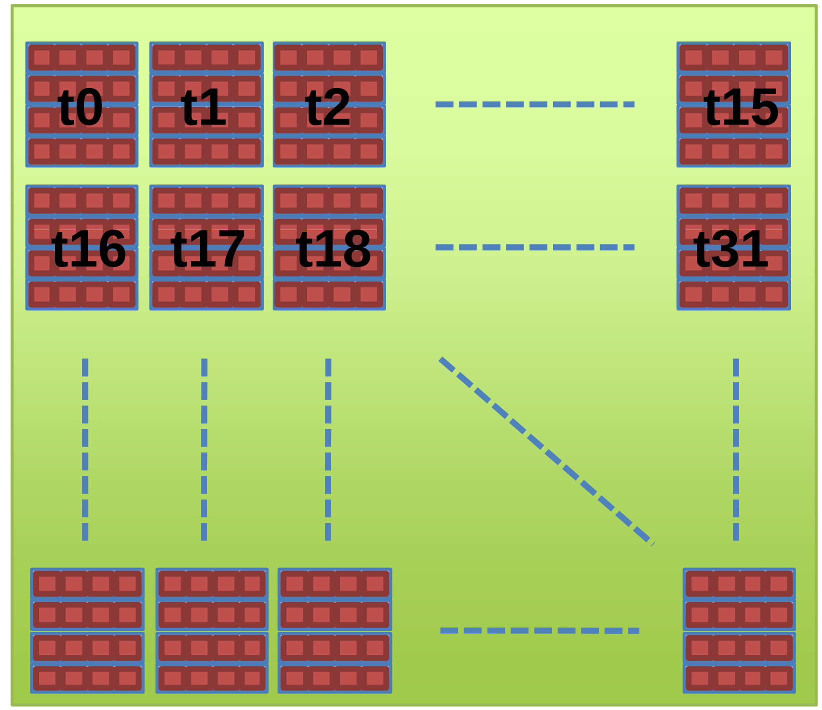
Reading and Writing MicroBlocks

- Reading one row of MicroBlock via **`unsigned int`** rather than 4x **`unsigned char`**



16x16 Thread Blocks

- One (16-thread) warp reads one row of MicroBlocks.
- One 16x16 block of threads deals with a 64x64 pixel region (8-bit luminance pixels).



Pseudo Code

- Assume single 64x64 image.

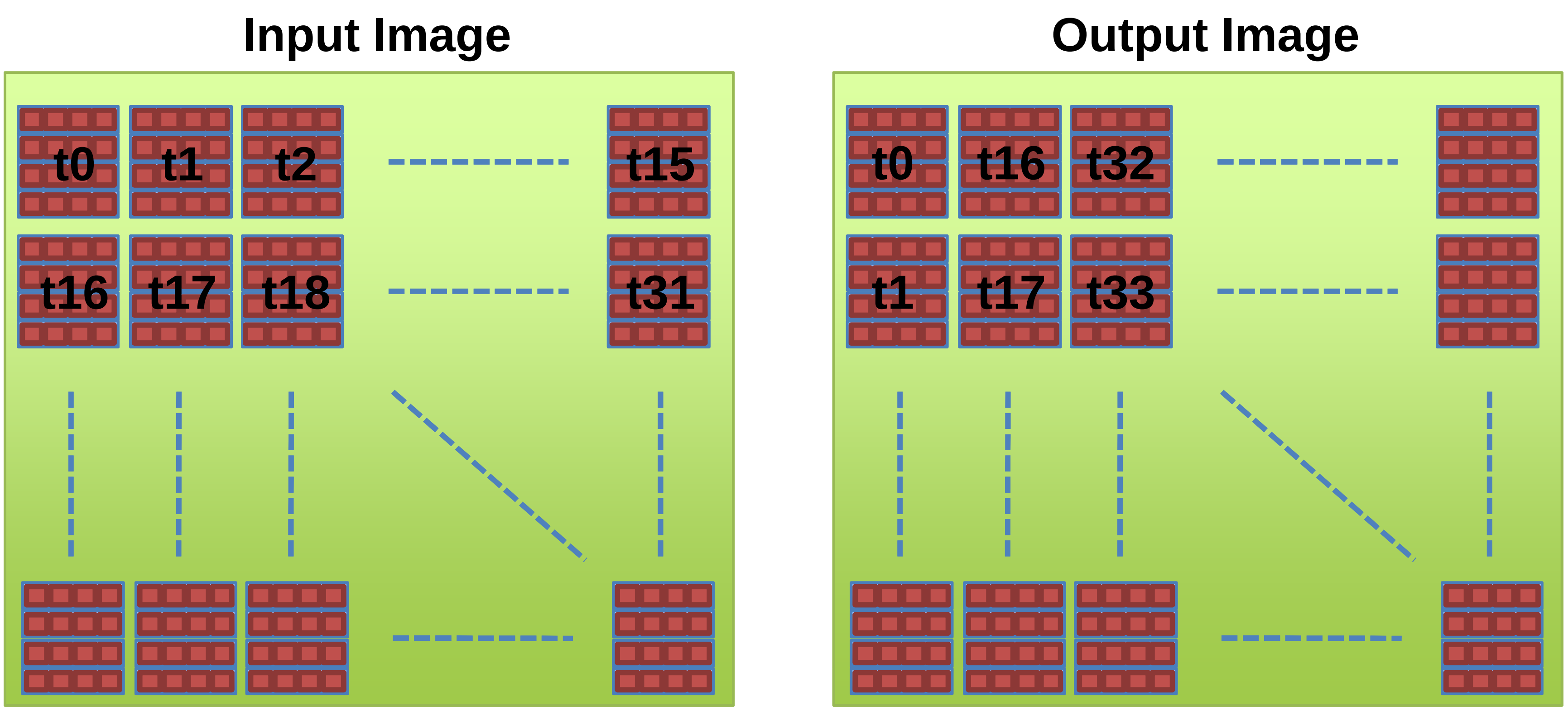
```
kernel(...)
{
  int i = threadIdx.x;
  int j = threadIdx.y;

  readMicroBlock(image, i, j, shared, i, j);
  transposeMicroBlock(shared, i, j);
  writeMicroBlock(shared, i, j, image, j, i);
}
```

- Problem: Non-coalesced writes!

Write Coalescing for Transpose

- `readMicroBlock(image, i, j, shared, i, j);`
- `writeMicroBlock(shared, i, j, image, j, i);`



Coalesced Writes

- Simple fix:

```
kernel(...)
{
   int i = threadIdx.x;
   int j = threadIdx.y;

   readMicroBlock(image, i, j, shared, i, j);
   transposeMicroBlock(shared, i, j);
   __syncthreads();
   writeMicroBlock(shared, j, i, image, i, j);
}
```

- Must __syncthreads() because $T_{i,j}$ now writes data produced by $T_{j,i}$.

Transpose Performance

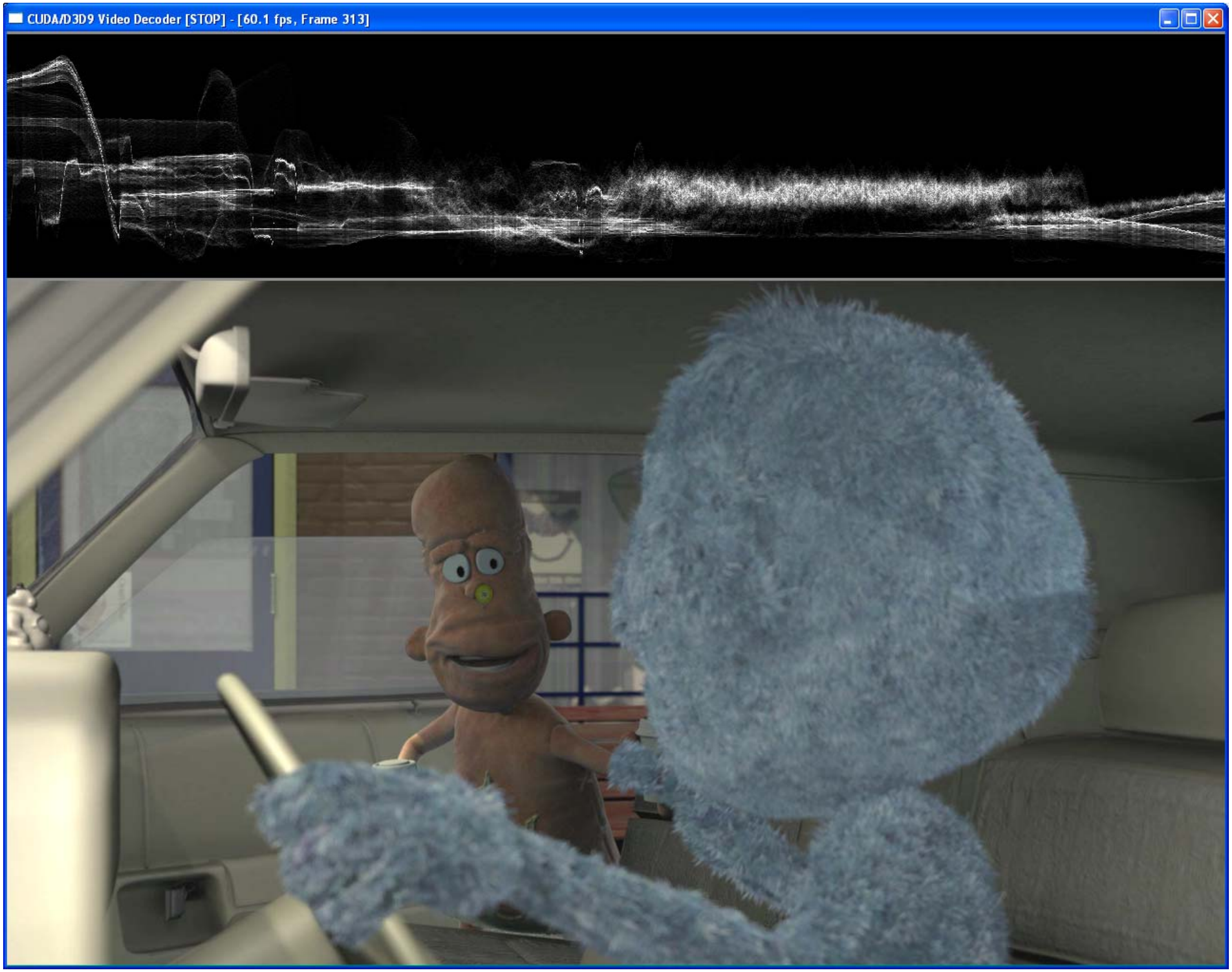
Algorithm	256x256	512x512	1024^2	2048^2	4096^2
CUDA Naive	2.39	3.72	3.43	3.29	2.89
CUDA Opt	16.64	28.73	35.44	38.88	40.33
IPPI	9.03	8.49	5.07	3.83	2.60

Unit: GB/s throughput.
GPU: GeForce GTX 280 (GT200)
CPU: Intel Core 2 Duo X6800 @ 2.93GHz

Summary

- Memory access crucial for CUDA performance.
- Shared memory as user-managed cache.
- 8-bit images especially tricky.
- Extra pass may improve over all performance.

Waveform Demo



NVIDIA

Questions?

- Eric Young
  - (eyoung@nvidia.com)
- Frank Jargstorff
  - (fjargsto@nvidia.com)