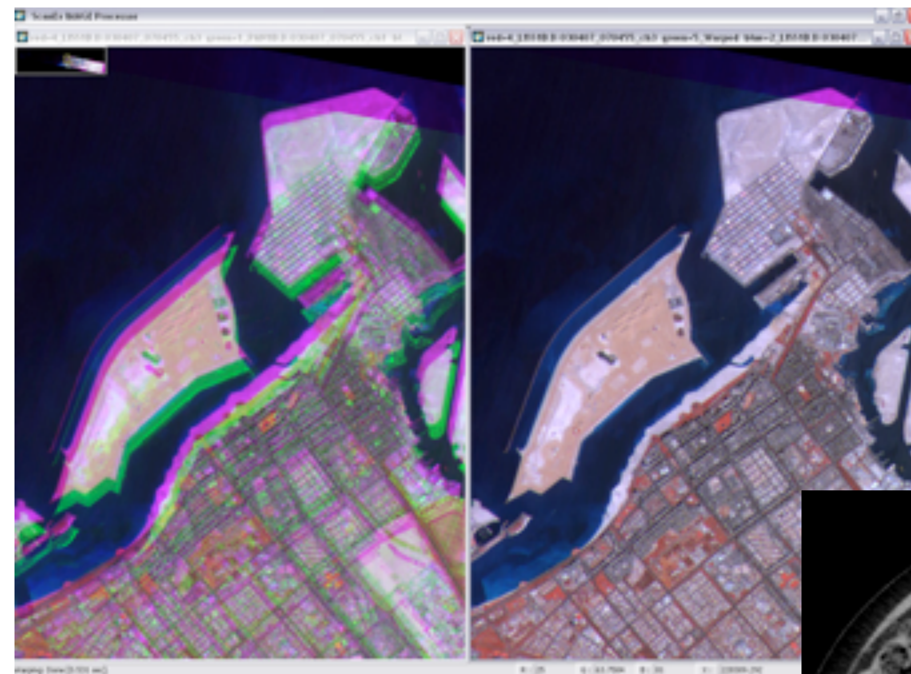
Image Processing

- Spatial / GIS
- Medical Imaging
- Oil and gas exploration
- ...
- A good starting use case for GPUs



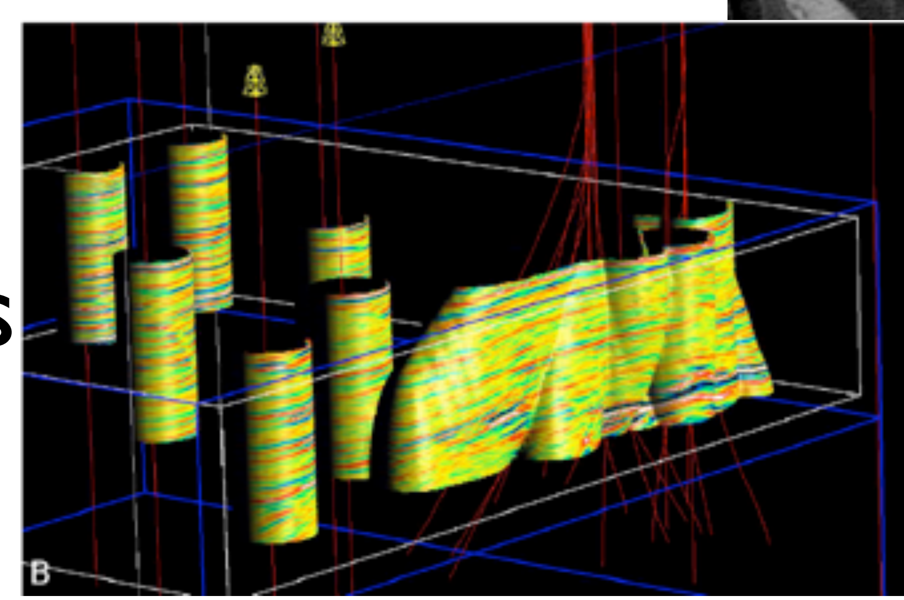
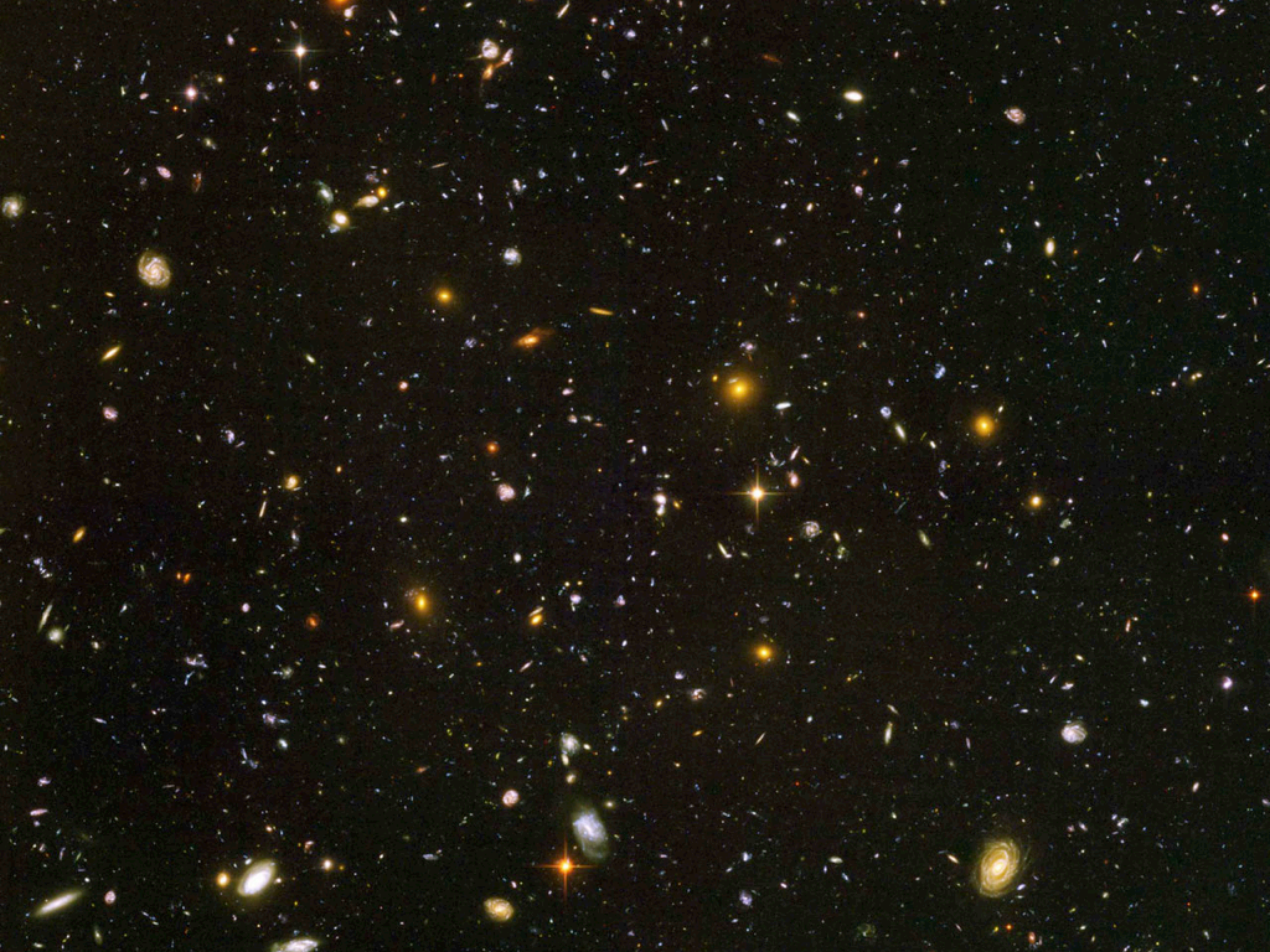


Image Processing

- Today:
  - Greyscaling (~contrast enhancement)
  - Smoothing (de-speckling/de-noising)
- Today's processing are simple, butbut often part of real image processing pipeline
- (eg, astronomical image processing)



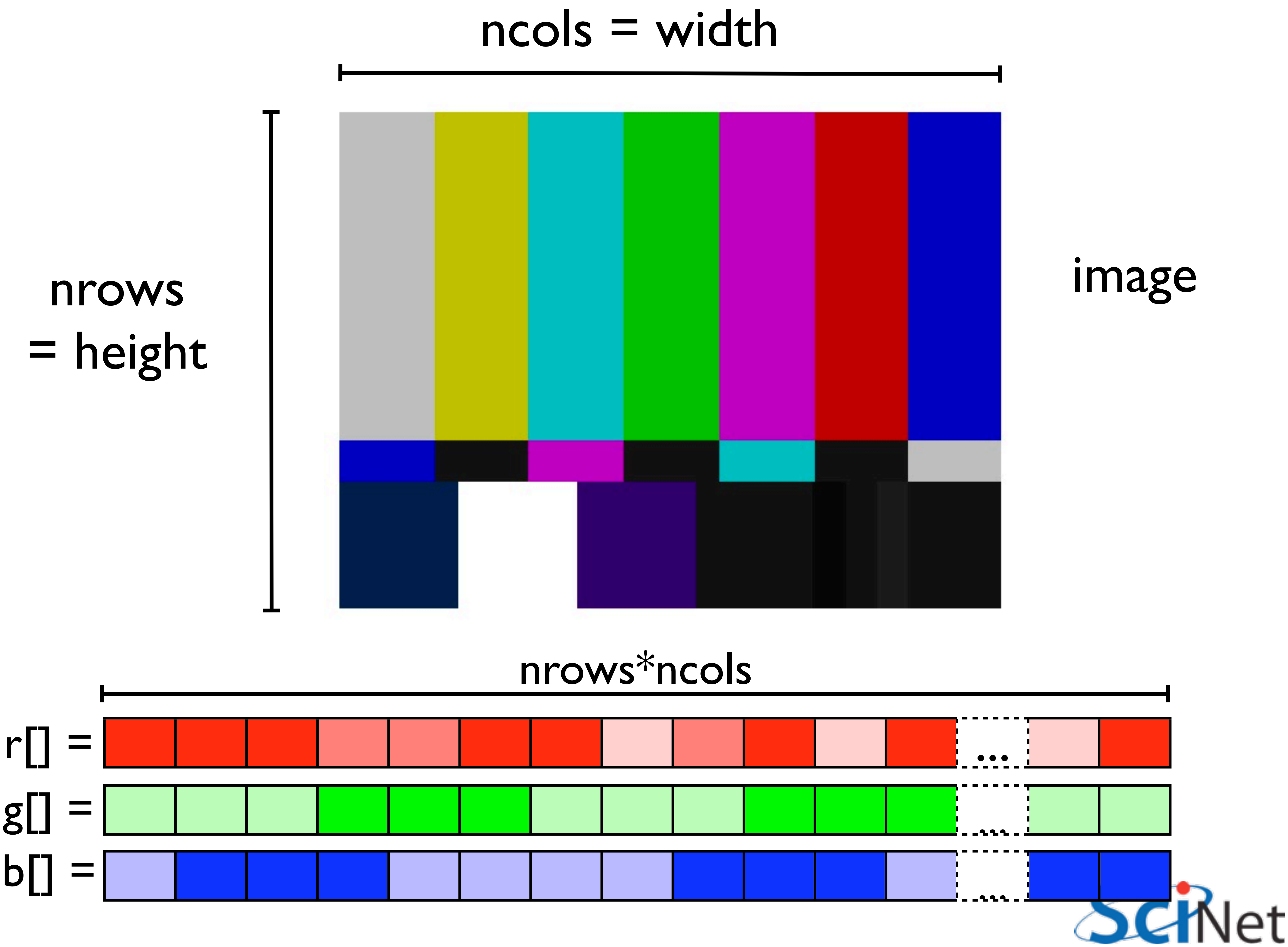
Allocate arrays

```
writeHTMLTitle(fp, "Input Test image");
int *r = (int *)malloc(dimx*dimy*sizeof(int));
int *g = (int *)malloc(dimx*dimy*sizeof(int));
int *b = (int *)malloc(dimx*dimy*sizeof(int));
testpattern(dimx, dimy, r, g, b);
writeHTMLImage(fp, "input", dimx, dimy, r, g, b, 4);
```

main(), example1.c

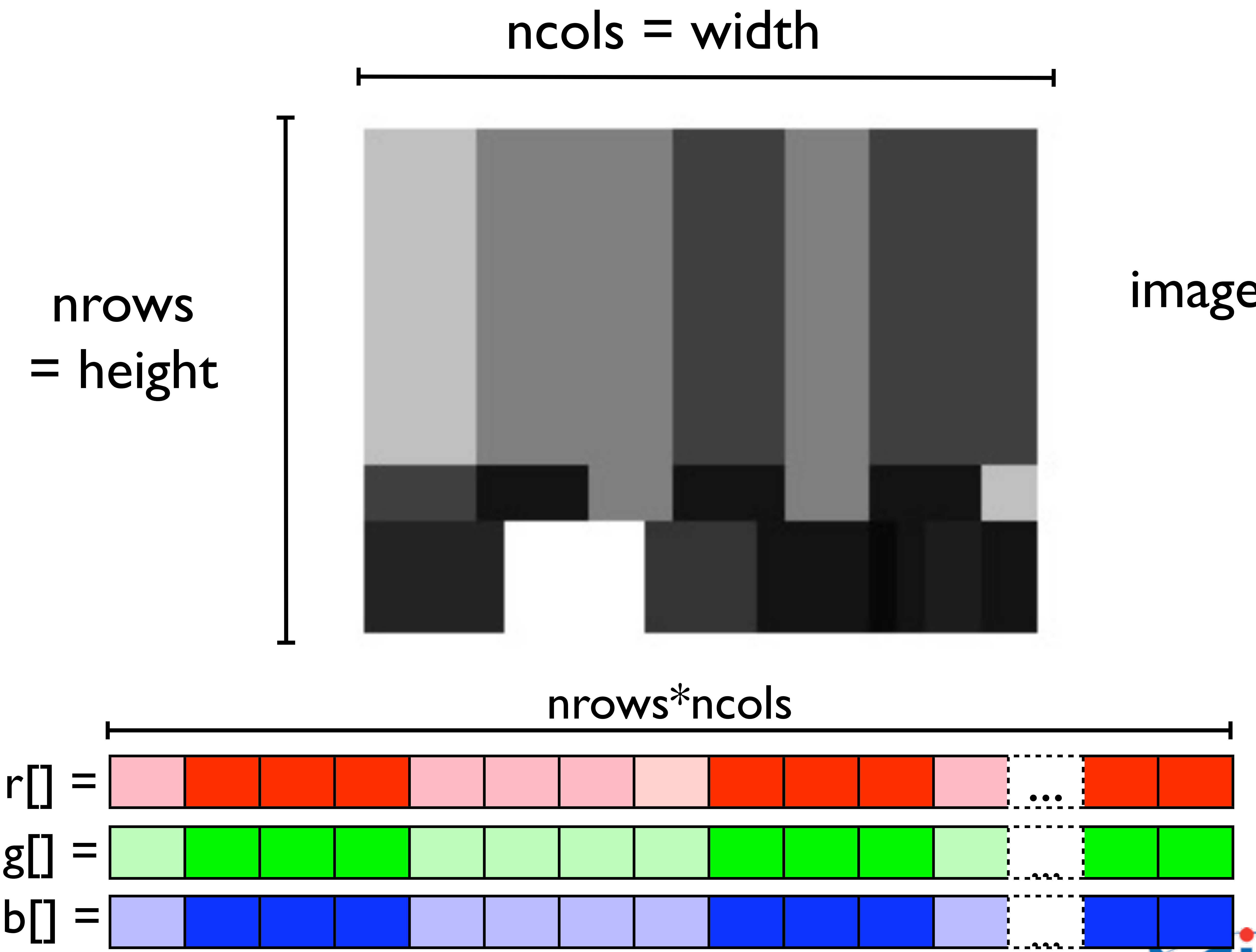
Generate image

Write image


ncols = width
nrows
= height
image
nrows*ncols
r[] =
g[] =
b[] =
...
SciNet

Greyscaling the image (CPU)

```
/* do a greyscale image by averaging the colors at each point */
void cpuGreyscale(const int nrows, const int ncols,
                  const int *in_r, const int *in_g, const int *in_b,
                  int *out_r, int *out_g, int *out_b) {

    int pix = 0;
    for (pix = 0; pix < nrows*ncols; pix++) {
        int avg = (in_r[pix] + in_g[pix] + in_b[pix])/3;
        out_r[pix] = avg;
        out_g[pix] = avg;
        out_b[pix] = avg;
    }
    return;
}
```


ncols = width
nrows
= height
image
nrows*ncols
r[] =
g[] =
b[] =
...

Workflow

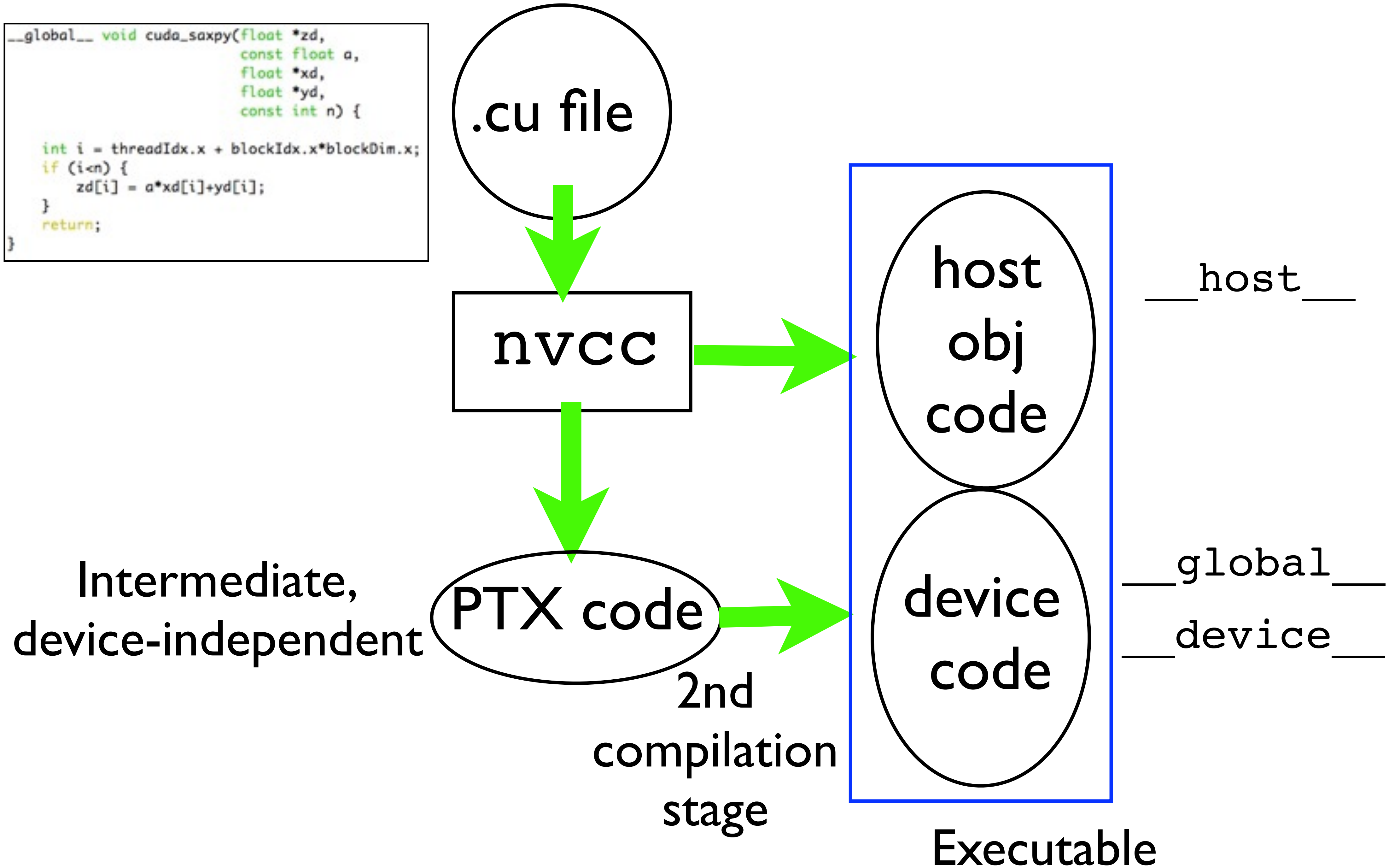
- Allocate memory
- Greyscale the image (taking data from input, processing it, putting it in output)
- Writing to a file
- Freeing memory

```
writeHTMLTitle(fp, "CPU-greyscaled Test image");
int *cpu_grey_r = (int *)malloc(dimx*dimy*sizeof(int));
int *cpu_grey_g = (int *)malloc(dimx*dimy*sizeof(int));
int *cpu_grey_b = (int *)malloc(dimx*dimy*sizeof(int));
cpuGreyscale(dimx, dimy, r, g, b, cpu_grey_r, cpu_grey_g, cpu_grey_b);
writeHTMLImage(fp, "grey_cpu", dimx, dimy, cpu_grey_r, cpu_grey_g, cpu_grey_b, 4);
free(cpu_grey_r); free(cpu_grey_g); free(cpu_grey_b);
```

SciNet

Compilation process

```
__global__ void cuda_saxpy(float *zd,
                           const float a,
                           float *xd,
                           float *yd,
                           const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        zd[i] = a*xd[i]+yd[i];
    }
    return;
}
```



Workflow

- GPU looks the same
- ...but let's go a little deeper:

```
writeHTMLTitle(fp, "CPU-greyscaled Test image");
int *cpu_grey_r = (int *)malloc(dimx*dimy*sizeof(int));
int *cpu_grey_g = (int *)malloc(dimx*dimy*sizeof(int));
int *cpu_grey_b = (int *)malloc(dimx*dimy*sizeof(int));
cpuGreyscale(dimx, dimy, r, g, b, cpu_grey_r, cpu_grey_g, cpu_grey_b);
writeHTMLImage(fp, "grey_cpu", dimx, dimy, cpu_grey_r, cpu_grey_g, cpu_grey_b, 4);
free(cpu_grey_r); free(cpu_grey_g); free(cpu_grey_b);
```

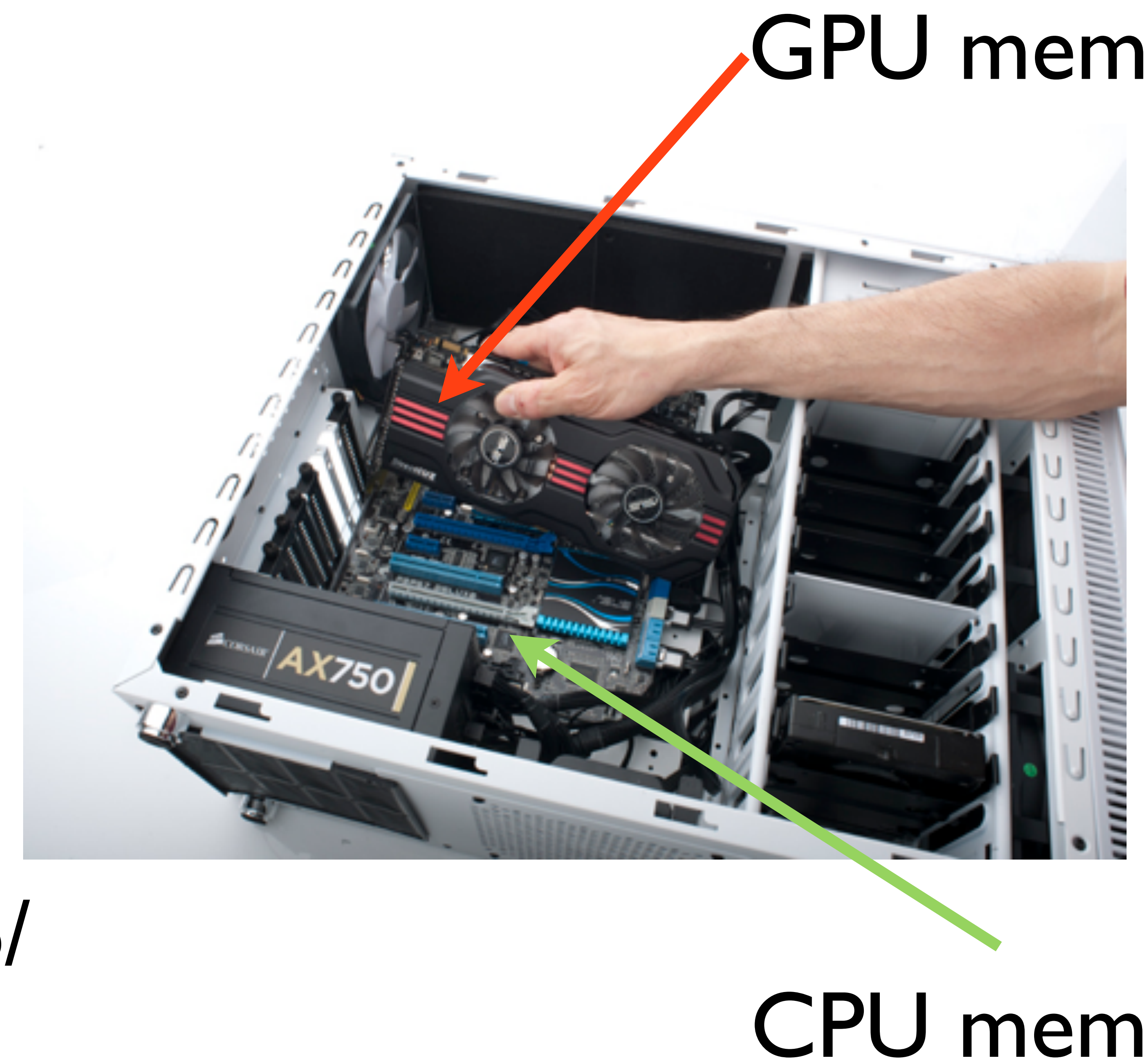
```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

gpuGreyscale(), example1.cu

SciNet

GPU memory is separate

- Different machine, different mem
- “Device” vs “Host”
- Copy back and forth over PCI bus
- Must explicitly allocate, copy data to/from host/device



SciNet

```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

Allocate input, output arrays on gpu

gpuGreyscale(), example1.cu

```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

Copy host input data to GPU input data

gpuGreyscale(), example1.cu

```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

Run GPU code

gpuGreyscale(), example1.cu

```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

Copy output GPU data to host

gpuGreyscale(), example1.cu

SciNet

```
CHK_CUDA( cudaMalloc(&in_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&in_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMalloc(&out_r_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_g_d,  nrows*ncols*sizeof(int)) );
CHK_CUDA( cudaMalloc(&out_b_d,  nrows*ncols*sizeof(int)) );

CHK_CUDA( cudaMemcpy(in_r_d, in_r, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_g_d, in_g, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );
CHK_CUDA( cudaMemcpy(in_b_d, in_b, nrows*ncols*sizeof(int), cudaMemcpyHostToDevice) );

greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

CHK_ERROR ;

CHK_CUDA( cudaMemcpy(out_r, out_r_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_g, out_g_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );
CHK_CUDA( cudaMemcpy(out_b, out_b_d, nrows*ncols*sizeof(int), cudaMemcpyDeviceToHost) );

CHK_CUDA( cudaFree(in_r_d) );
CHK_CUDA( cudaFree(in_g_d) );
CHK_CUDA( cudaFree(in_b_d) );

CHK_CUDA( cudaFree(out_r_d) );
CHK_CUDA( cudaFree(out_g_d) );
CHK_CUDA( cudaFree(out_b_d) );
```

Free GPU mem

gpuGreyscale(), example1.cu

Note all the error checking!

- GPU is essentially an embedded device
- Can't crash, throw error every time an error is encountered
- Will fail silently if you give it invalid data and truck on as best it can
- Need to explicitly test for error conditions.

```
#define CHK_CUDA(e) {if (e != cudaSuccess) {fprintf(stderr,"Error: %s\n", cudaGetErrorString(e)); exit(-1);}}
#define CHK_ERROR   {cudaThreadSynchronize(); cudaError_t e = cudaGetLastError(); if(e != cudaSuccess) {fprintf(
stderr,"Error: %s\n", cudaGetErrorString(e)); exit(-1);}}
```

SciNet

GPU Code:

```
__global__ void greyscaleKernel(const int nrows, const int ncols,
                                const int *in_r_d, const int *in_g_d, const int *in_b_d,
                                int *out_r_d, int *out_g_d, int *out_b_d) {

        int i = threadIdx.x;

        if (i < nrows*ncols) {
            int avg = ( in_r_d[i] + in_g_d[i] + in_b_d[i] )/3 ;
            out_r_d[i] = avg;
            out_g_d[i] = avg;
            out_b_d[i] = avg;
        }

        return;
}
```

SciNet

GPU Code:

__global__: GPU code, callable as a kernel from the host.
Alternatives: __kernel__ (only callable from other GPU code),
__host__ (on host, default).

```
__global__ void greyscaleKernel(const int nrows, const int ncols,
                                const int *in_r_d, const int *in_g_d, const int *in_b_d,
                                int *out_r_d, int *out_g_d, int *out_b_d) {

    int i = threadIdx.x;

    if (i < nrows*ncols) {
        int avg = ( in_r_d[i] + in_g_d[i] + in_b_d[i] )/3 ;
        out_r_d[i] = avg;
        out_g_d[i] = avg;
        out_b_d[i] = avg;
    }

    return;
}
```

GPU Code:

What is our thread index? (which thread are we?)

```
__global__ void greyscaleKernel(const int nrows, const int ncols,
                                const int *in_r_d, const int *in_g_d, const int *in_b_d,
                                int *out_r_d, int *out_g_d, int *out_b_d) {

        int i = threadIdx.x;

        if (i < nrows*ncols) {
            int avg = ( in_r_d[i] + in_g_d[i] + in_b_d[i] )/3 ;
            out_r_d[i] = avg;
            out_g_d[i] = avg;
            out_b_d[i] = avg;
        }

        return;
}
```

GPU vs CPU Code:

- CPU: Loops over pixels
- GPU: Loop over pixels ***implicit***

```
/* do a greyscale image by averaging the colors at each point */
void cpuGreyscale(const int nrows, const int ncols,
                  const int *in_r, const int *in_g, const int *in_b,
                  int *out_r, int *out_g, int *out_b) {

    int pix = 0;
    for (pix = 0; pix < nrows*ncols; pix++) {
        int avg = (in_r[pix] + in_g[pix] + in_b[pix])/3;
        out_r[pix] = avg;
        out_g[pix] = avg;
        out_b[pix] = avg;
    }
    return;
}
```

```
__global__ void greyscaleKernel(const int nrows, const int ncols,
                                const int *in_r_d, const int *in_g_d, const int *
                                int *out_r_d, int *out_g_d, int *out_b_d) {

        int i = threadIdx.x;

        if (i < nrows*ncols) {
            int avg = ( in_r_d[i] + in_g_d[i] + in_b_d[i] )/3 ;
            out_r_d[i] = avg;
            out_g_d[i] = avg;
            out_b_d[i] = avg;
        }

        return;
}
```

GPU Kernel Launch

- Kernel launch starts nrows*ncols threads
- Each has a thread index
- Each thread operates on one pixel
- *Very* fine-grained parallelism

```
__global__ void greyscaleKernel(const int nrows, const int ncols,
                                const int *in_r_d, const int *in_g_d, const int *in_
                                int *out_r_d, int *out_g_d, int *out_b_d) {

        int i = threadIdx.x;

        if (i < nrows*ncols) {
            int avg = ( in_r_d[i] + in_g_d[i] + in_b_d[i] )/3 ;
            out_r_d[i] = avg;
            out_g_d[i] = avg;
            out_b_d[i] = avg;
        }

        return;
}
```

```
greyscaleKernel<<<1,nrows*ncols>>>(nrows, ncols, in_r_d, ir
```

SciNet

GPUs and Threads

- The kernel launch starts a block of nrows*ncols threads
- Threads run in lock step
- Each operates on a work item
- Data parallelism

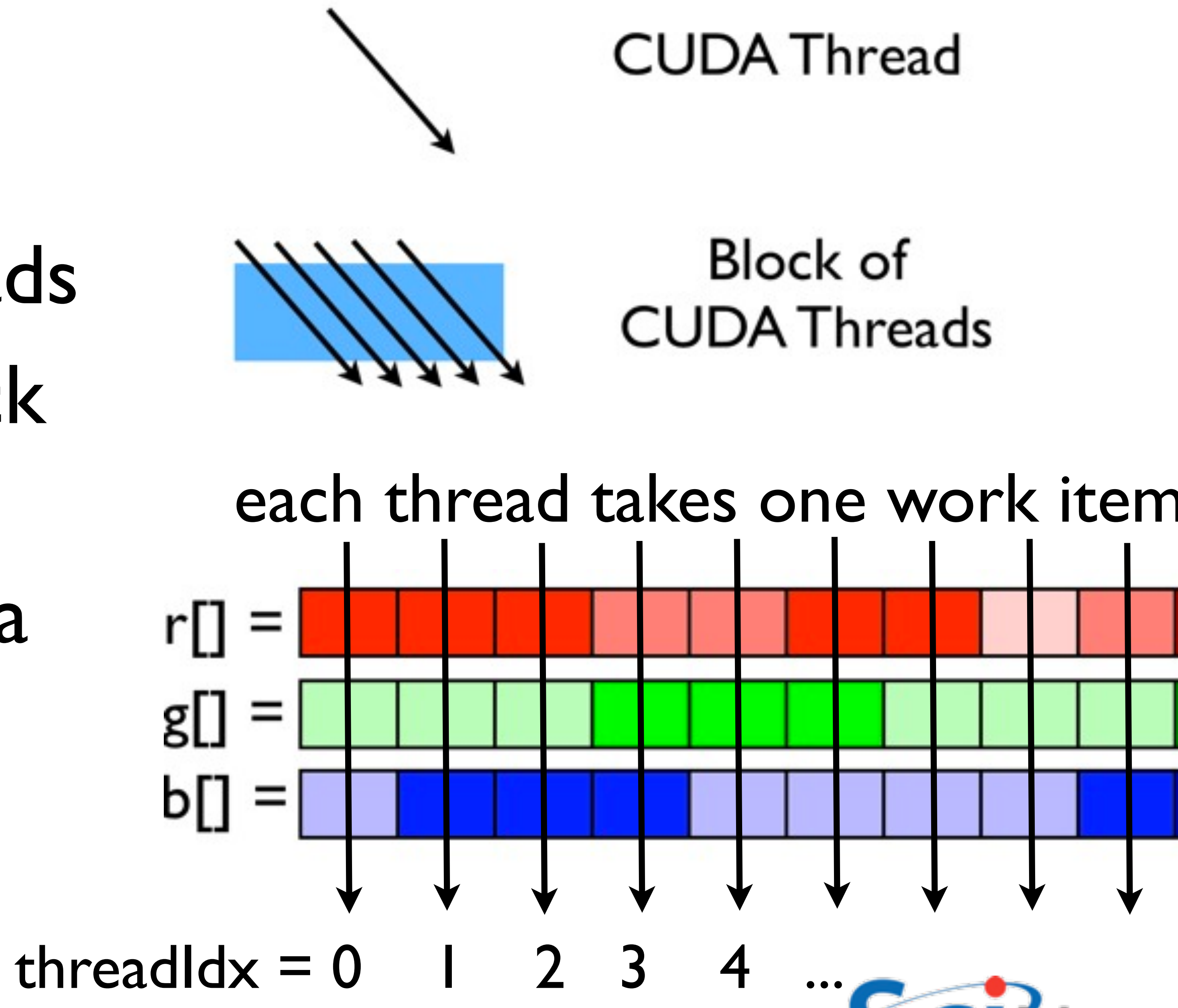
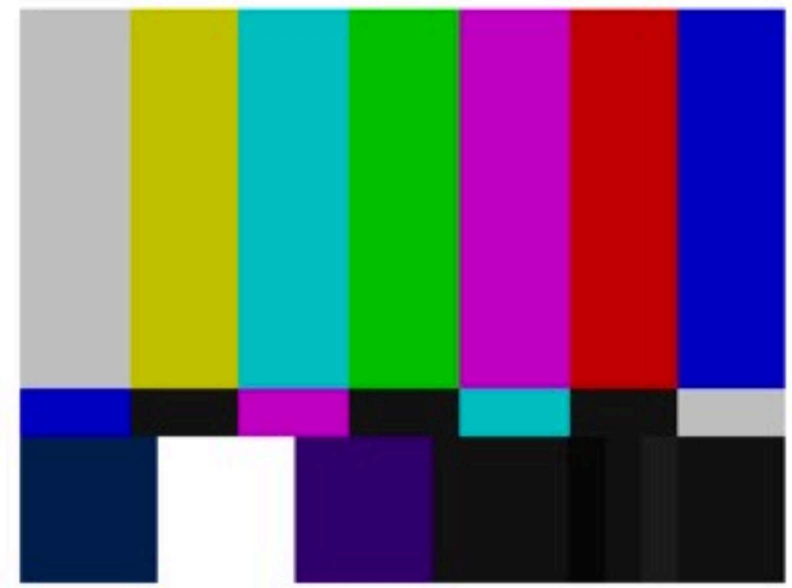


Image size

- Right now, we're working in very small images
- That's no good!
- Increase dimx/dimy to be closer to lgdmix/lgdimy.
- Recompile, run.
- What happens?

```
int main(int argc, char **argv)
{
    const int lgdimx = 256, lgdimy = 192;
    const int dimx = 24, dimy = 18;
```

Large Test image



Input Test image

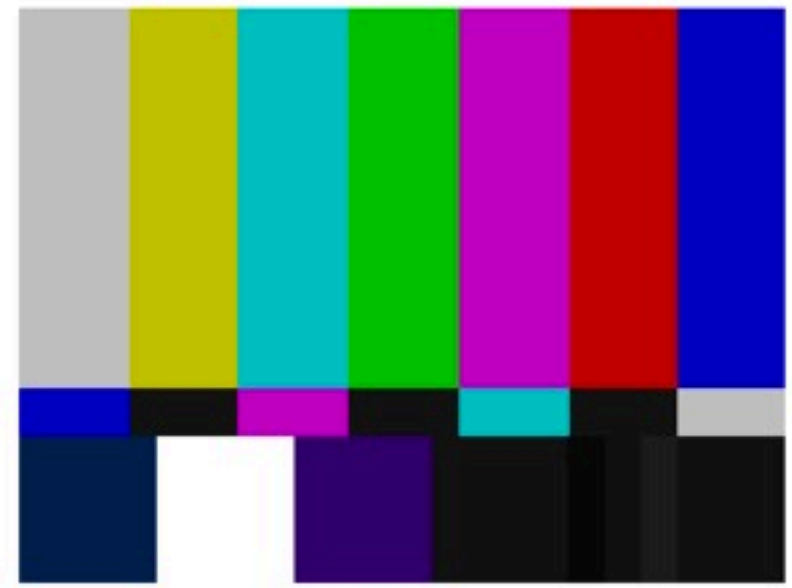
CPU-greyscaled Test image

Image size

- Right now, we're working in very small images
- That's no good!
- Increase dimx/dimy to be closer to lgdmix/lgdimy.
- Recompile, run.
- What happens?

```
user162-16:example1 ljdursi$ ./testpattern
Error: invalid configuration argument
```

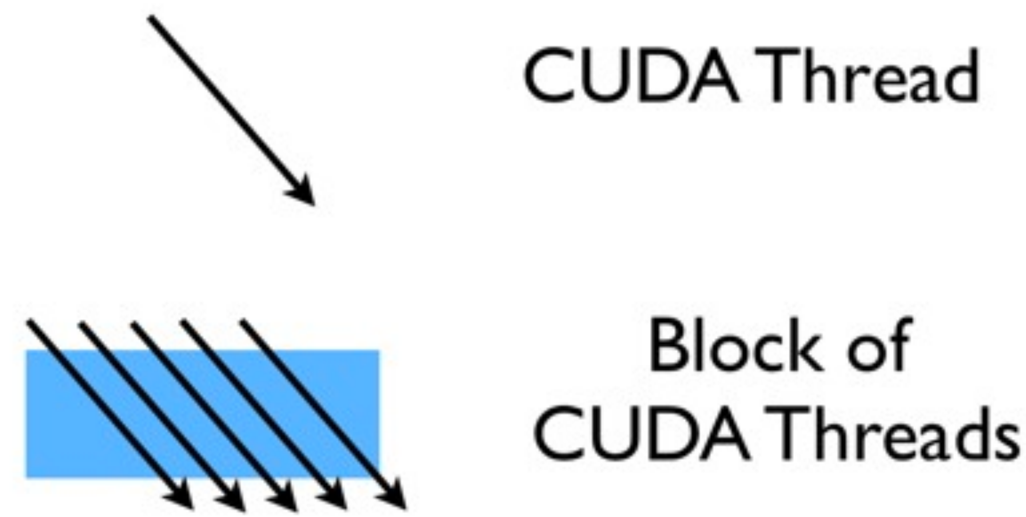
Large Test image



Input Test image

reyscaled Test image

Maximum # threads



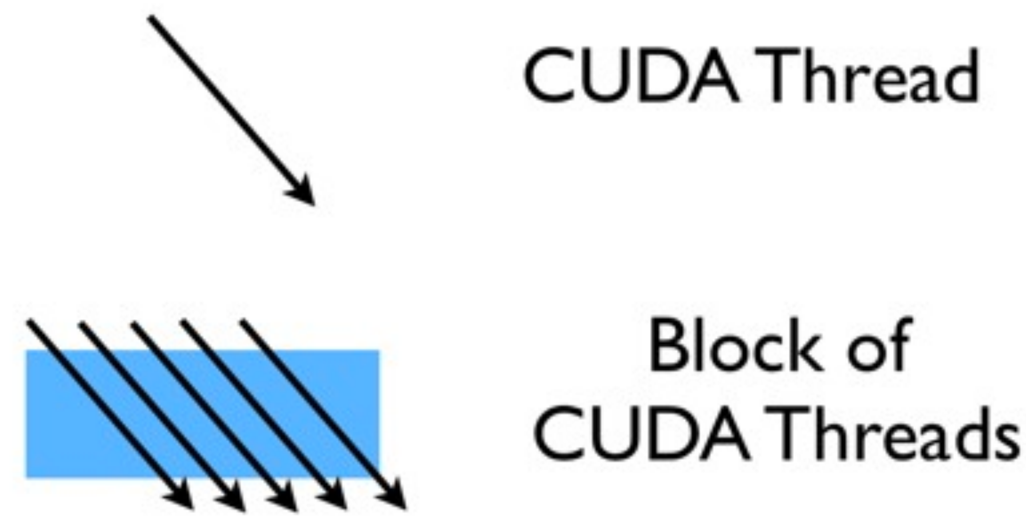
- Compile, run querydevs
- On my laptop:

```
Device 0 has:
        Name                 GeForce GT 650M,
        Number of SMs        2,
        Warp Size            32,
        Max Threads/block    1024,
        Regisgers/block      65536,
```

!!

Maximum # threads

- On a machine with a Tesla M2070 (Fermi):




```
Device 0 has:
        Name                    Tesla M2070,
        Number of SMs           14,
        Warp Size               32,
        Max Threads/block       1024,
        Regisgers/block         32768,
        Compute Capability      2.0,
        Global Mem              5375 MB,
```

!!

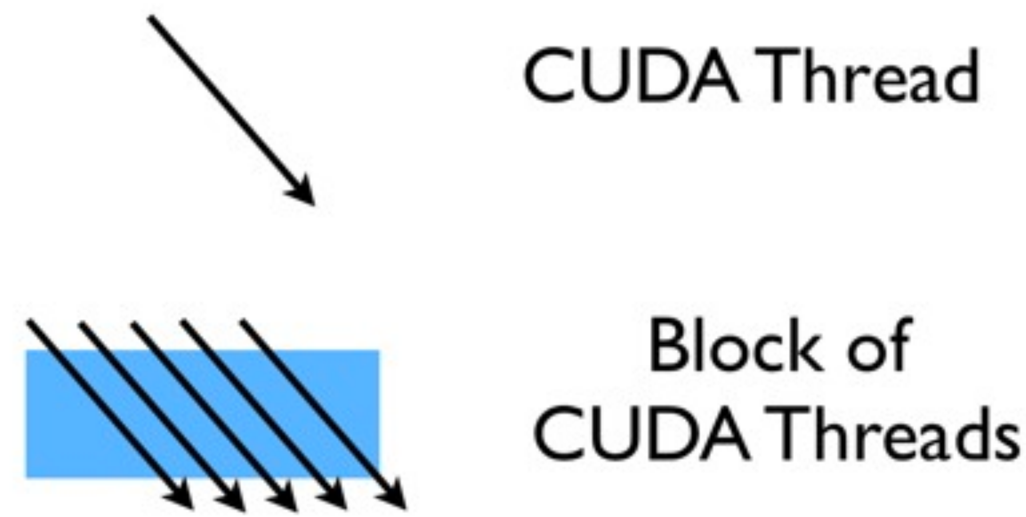
cudaGetDeviceProperty

```
int i, count;
cudaDeviceProp prop;

CHK_CUDA( cudaGetDeviceCount( &count ));
for (i=0; i<count; i++) {
    CHK_CUDA( cudaGetDeviceProperties( &prop, i ));
    printf("Device %d has:\n",i);
    printf("\tName                  %s,\n",prop.name);
    printf("\tNumber of SMs         %d,\n",prop.multiProcessorCount);
    printf("\tWarp Size             %d,\n",prop.warpSize);
    printf("\tMax Threads/block     %d,\n",prop.maxThreadsPerBlock);
```

querydevs.cu

SciNet

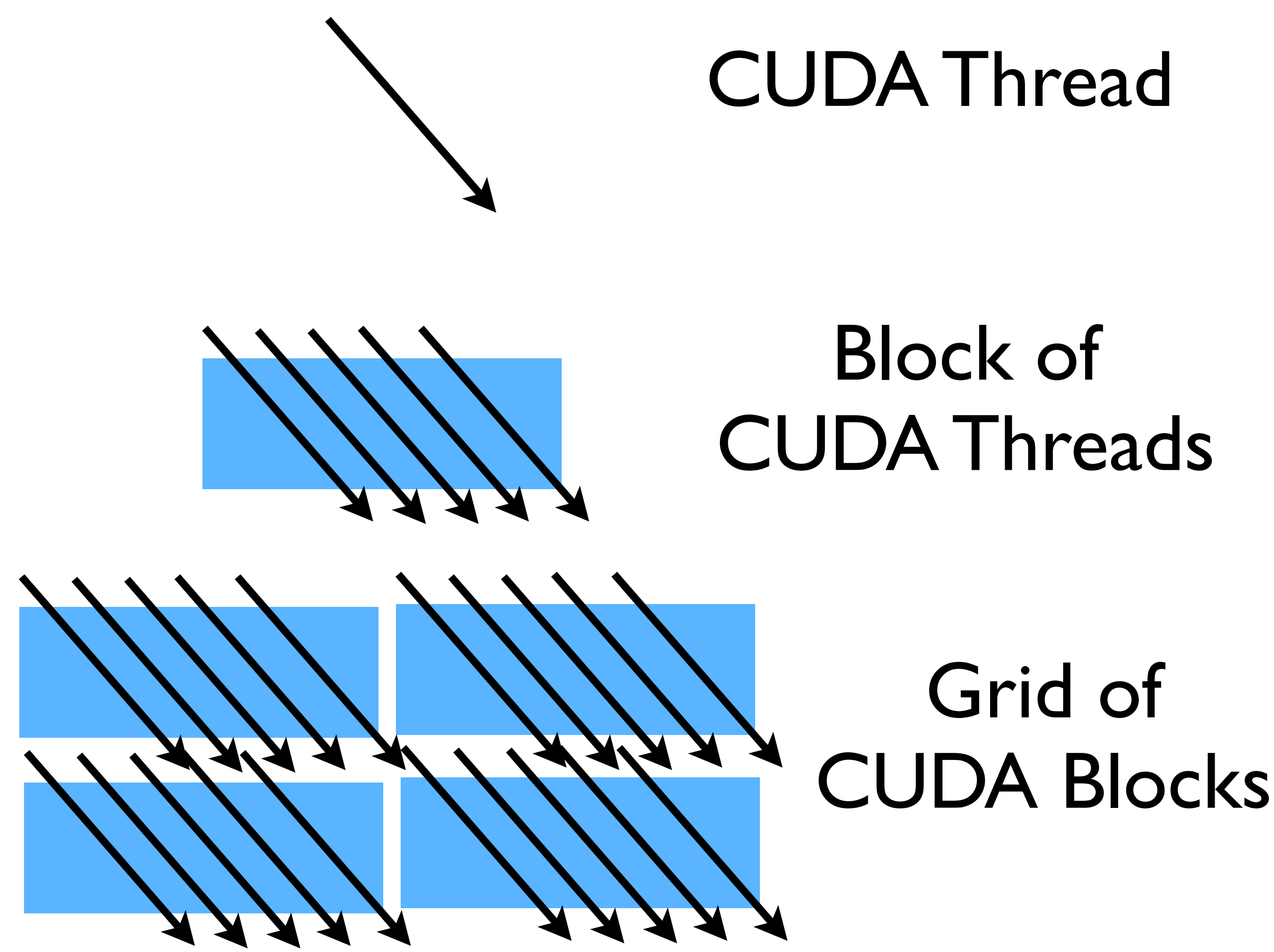
Maximum # threads



- Huh? How can we do big/ fast computing if we can only operate on 1k pixels?

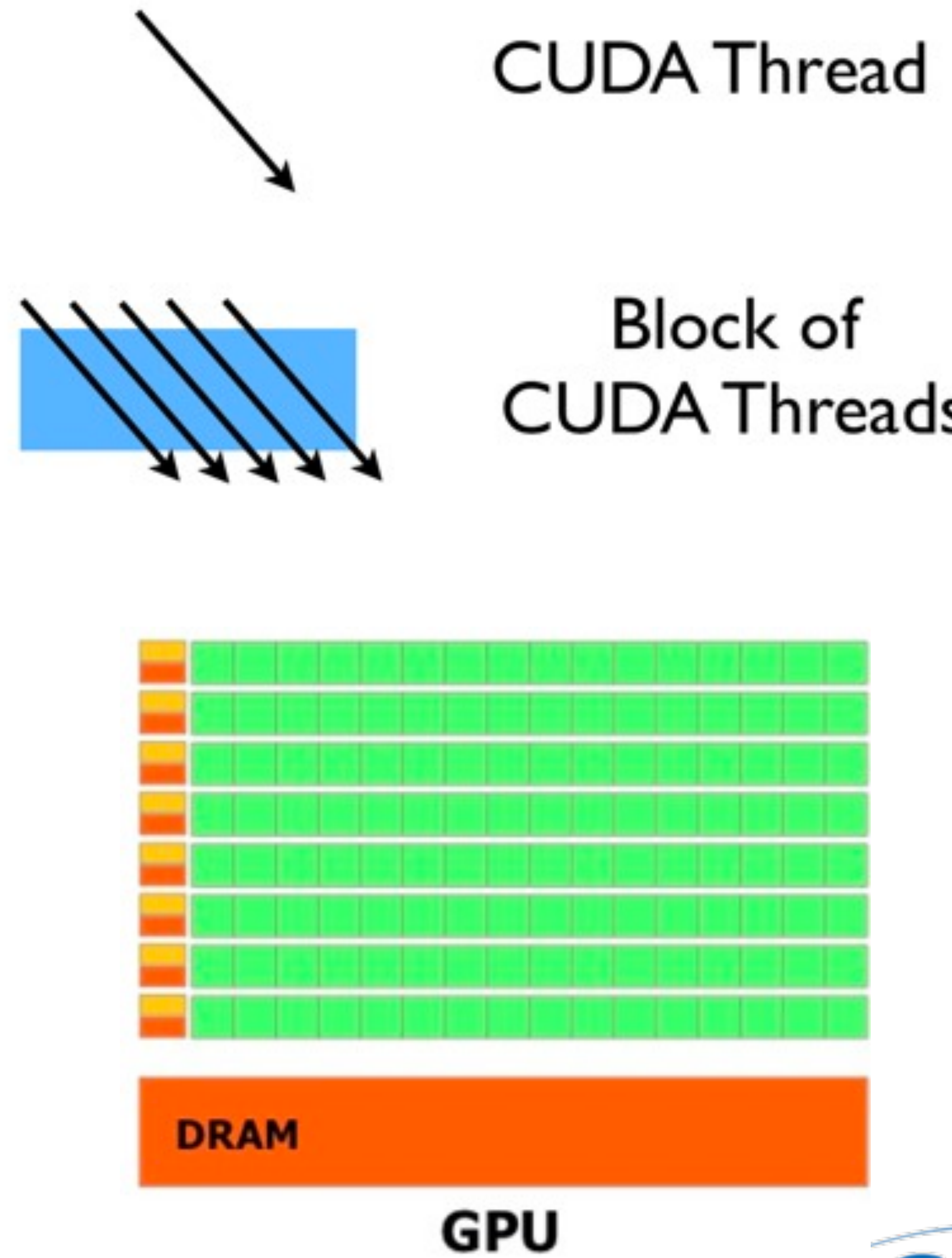
Threads, Blocks, Grids

- CUDA threads are organized into blocks
- Threads operate in SIMD(ish) manner -- each executing same instructions in lockstep.
- Only difference are thread ids
- Can have a grid of multiple blocks



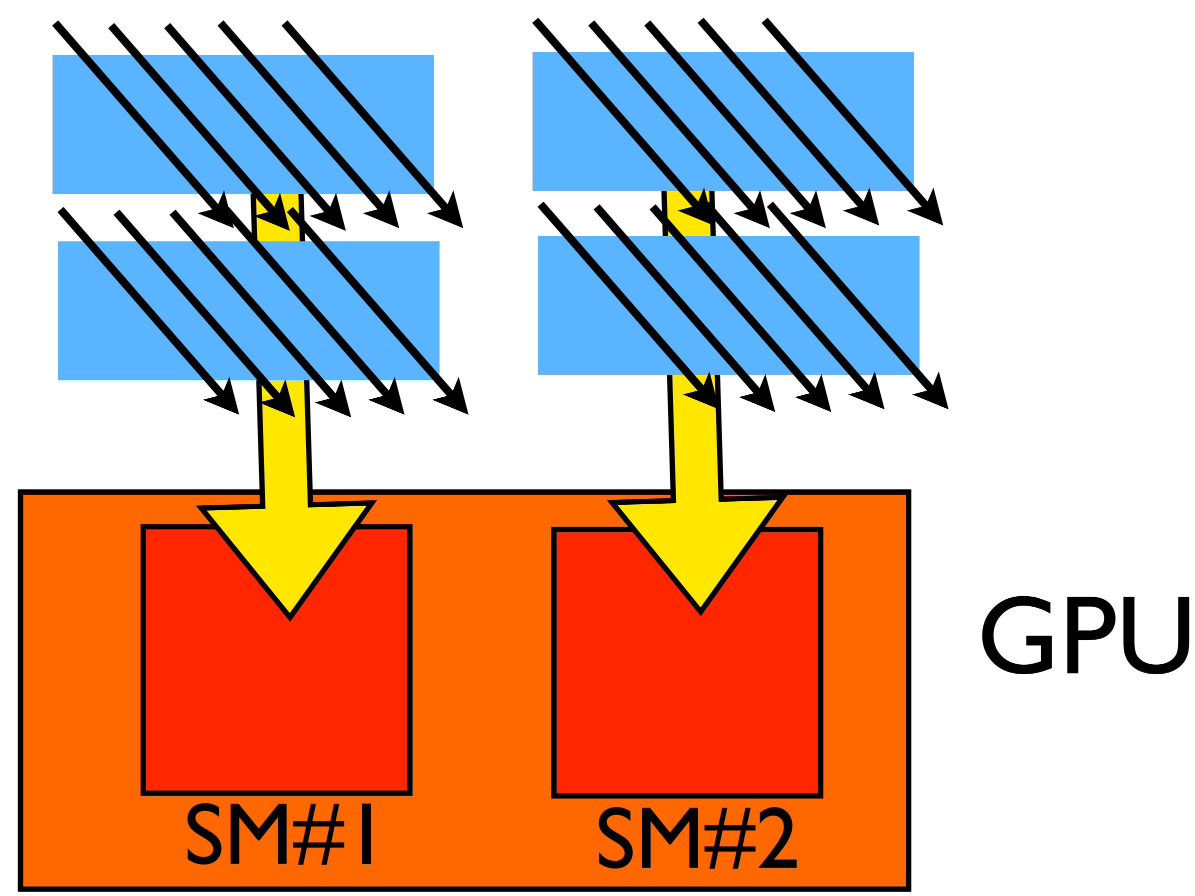
GPUs and Threads

- The GPU is split up into several "streaming multiprocessors" (SMs)
- Each have several cores, all operating in lockstep.



CUDA - H/W mapping

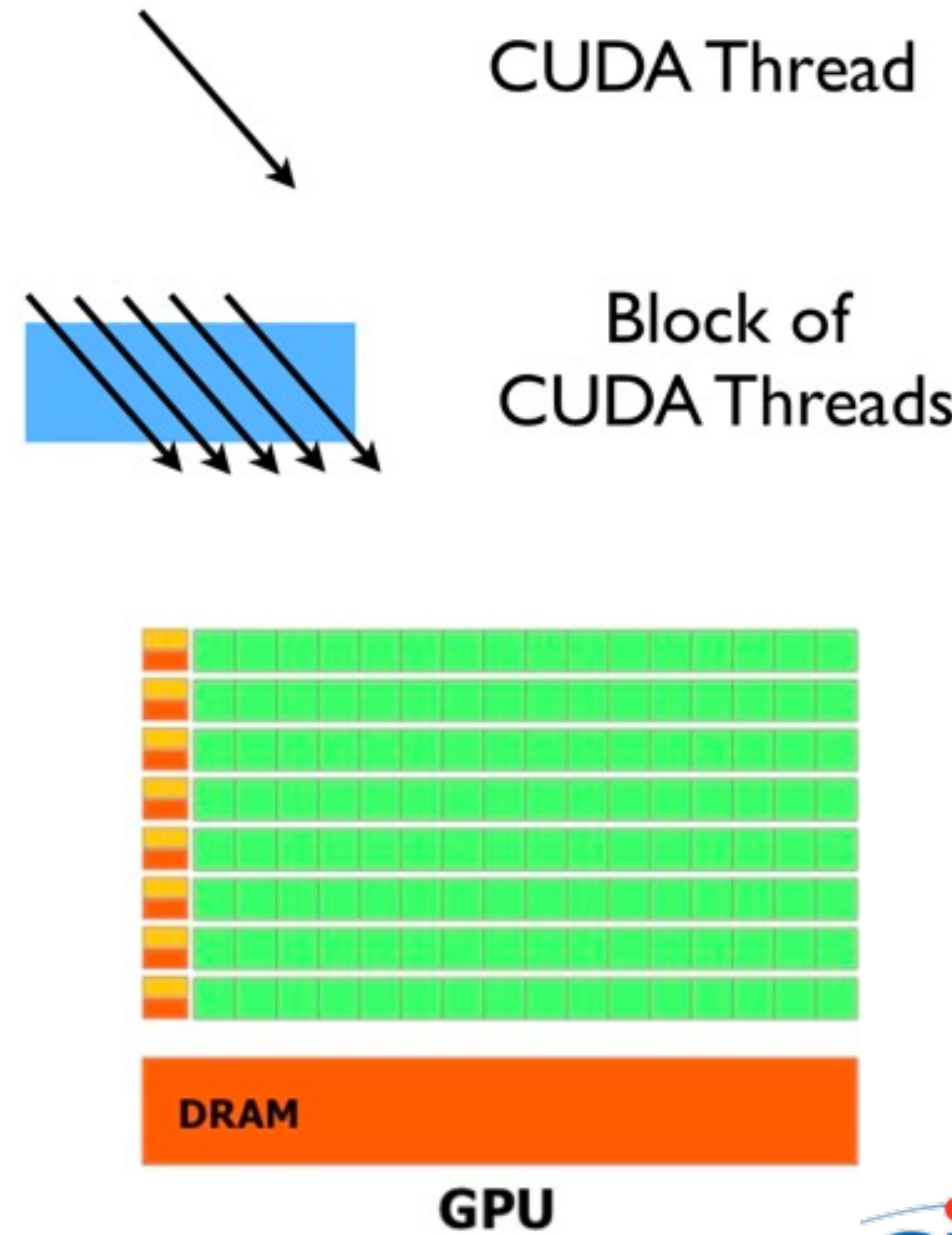
- Blocks are assigned to a particular SM
  - Executed there one 'warp' at a time (typically 32 threads)
- Multiple blocks may be on SM concurrently
  - Good; latency hiding
  - Bad - SM resources must be divided between blocks
- If only use 1 Block - 1 SM



SciNet

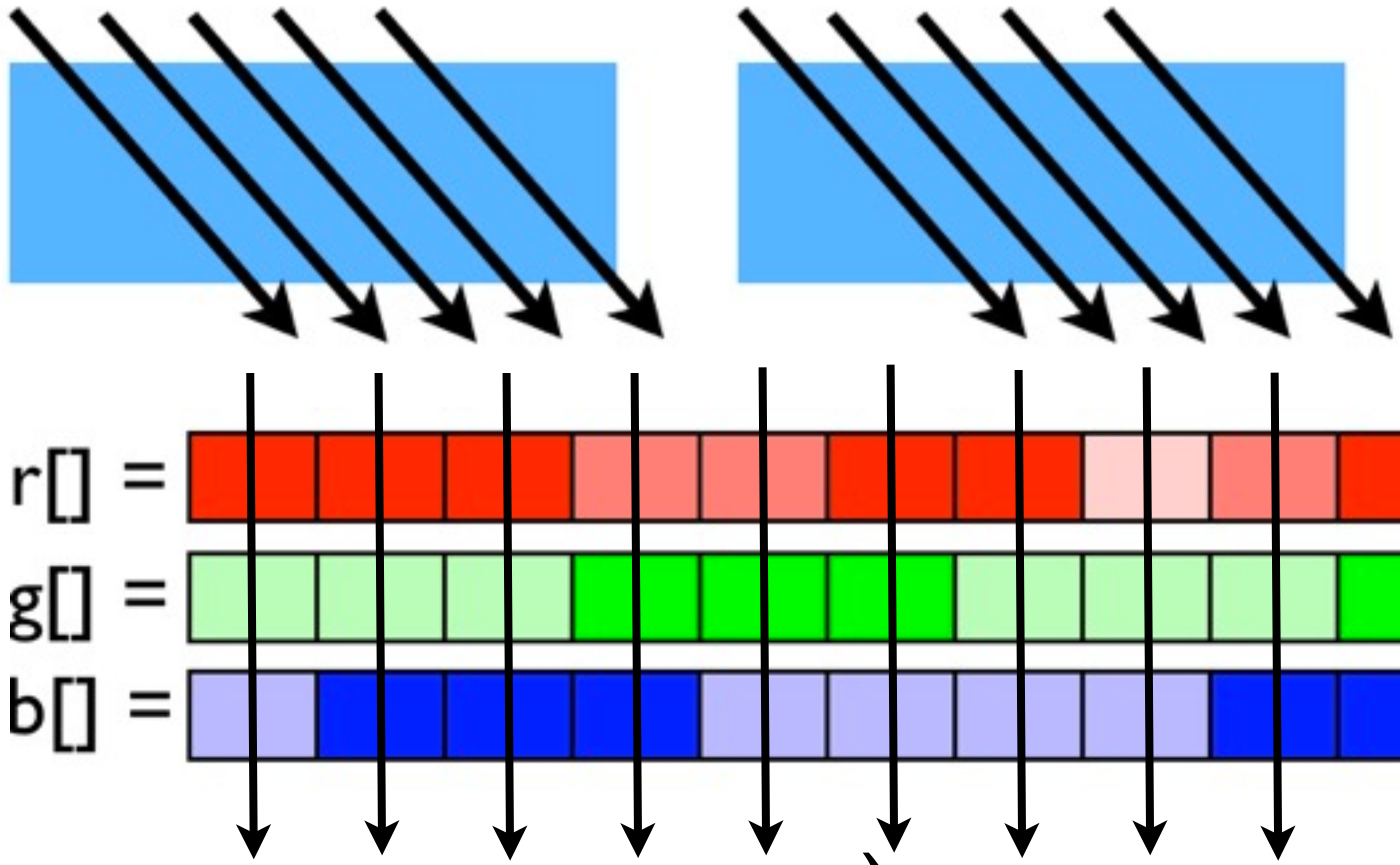
GPUs and Threads

- With only one block, locked onto one SM - using only fraction of your GPU.
- Better is to break computation onto many blocks of threads
- Take advantage of multiple SMs
- Can have many more blocks than SMs, this is often helpful.



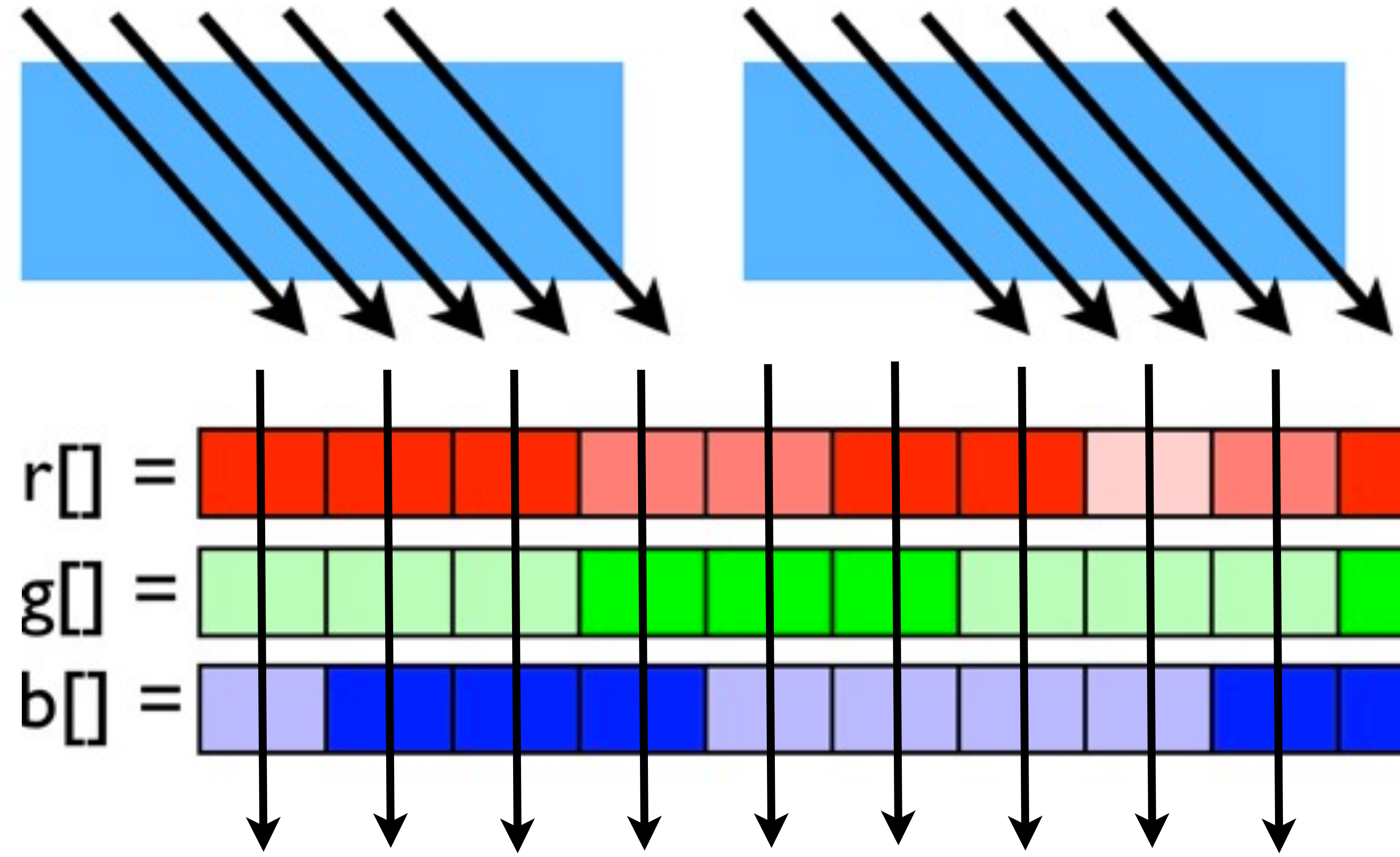
SciNet

Multi-block greyscaling



- Look in example2
- (Nclab: download http://support.scinet.utoronto.ca/~ljdursi/example2.py)
- Break into multiple blocks
- Can take full advantage of GPU

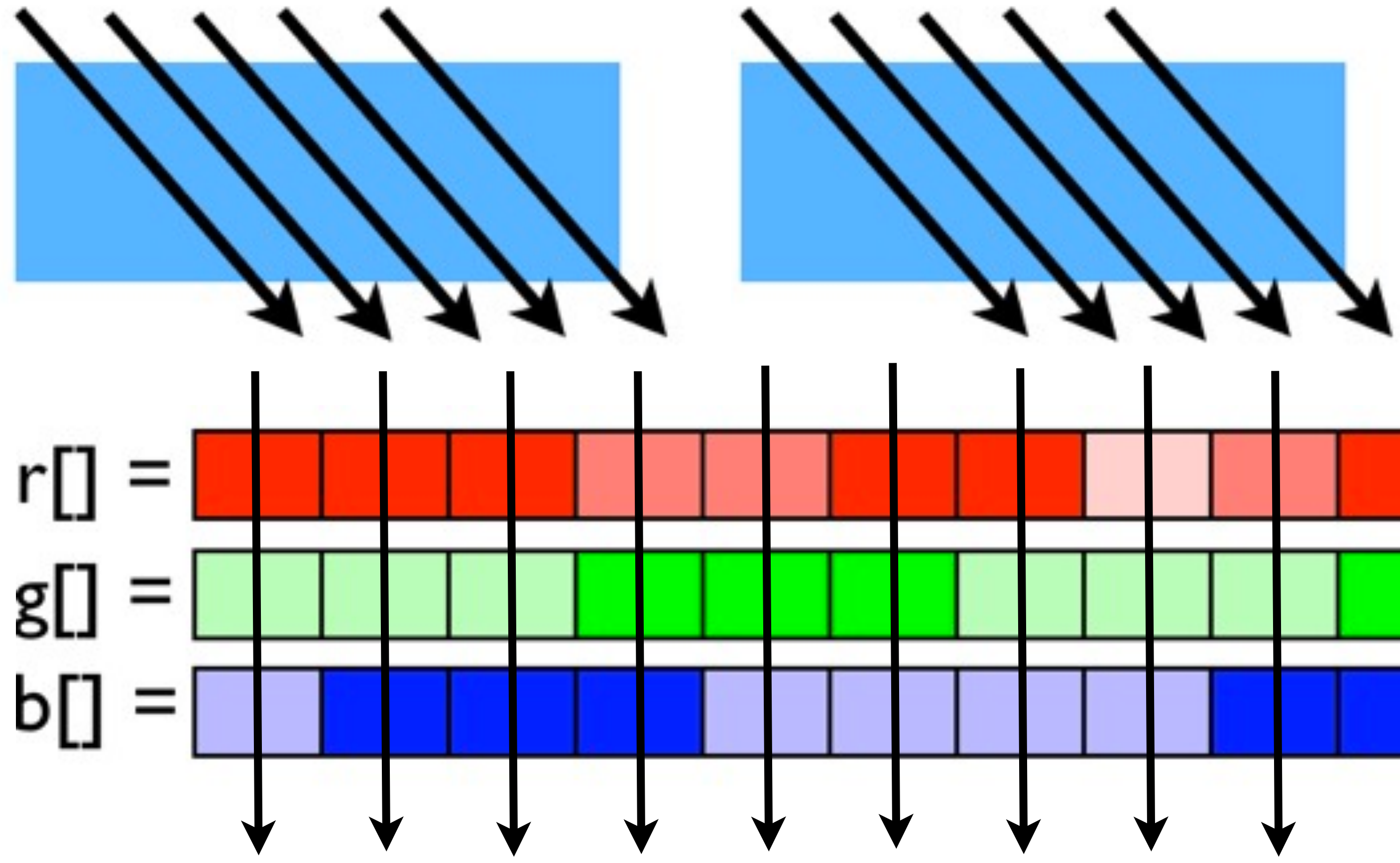
Multi-block greyscaling



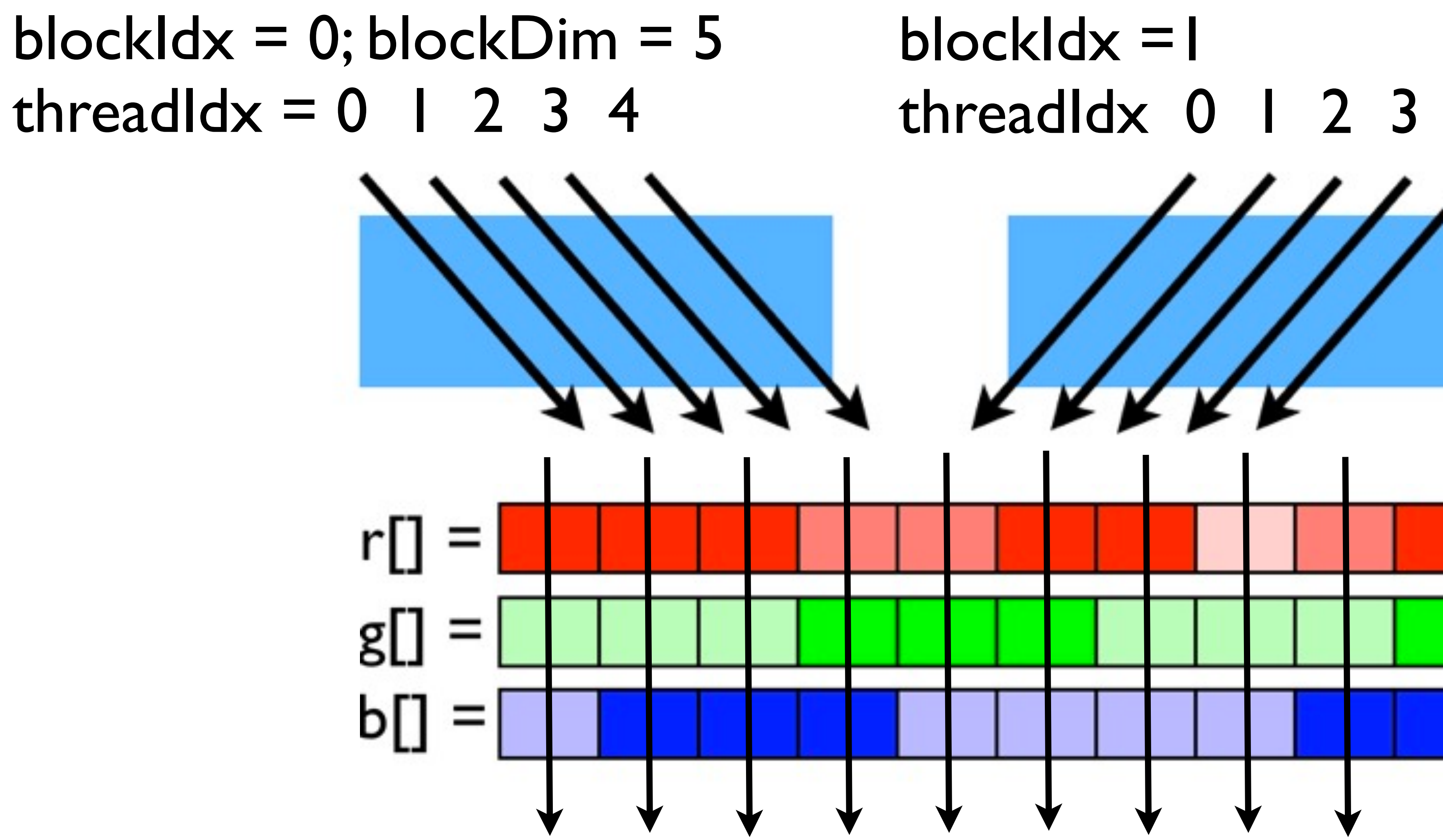

```
__global__ void greyscaleKernel(const int nrows, const int ncol
                                 const int *in_r_d, const int *in_g
                                 int *out_r_d, int *out_g_d, int *o

        int i = threadIdx.x + blockDim.x*blockIdx.x;
```

example2/testpattern.cu : gpuGreyscaleWithBlocks()

Multi-block greyscaling



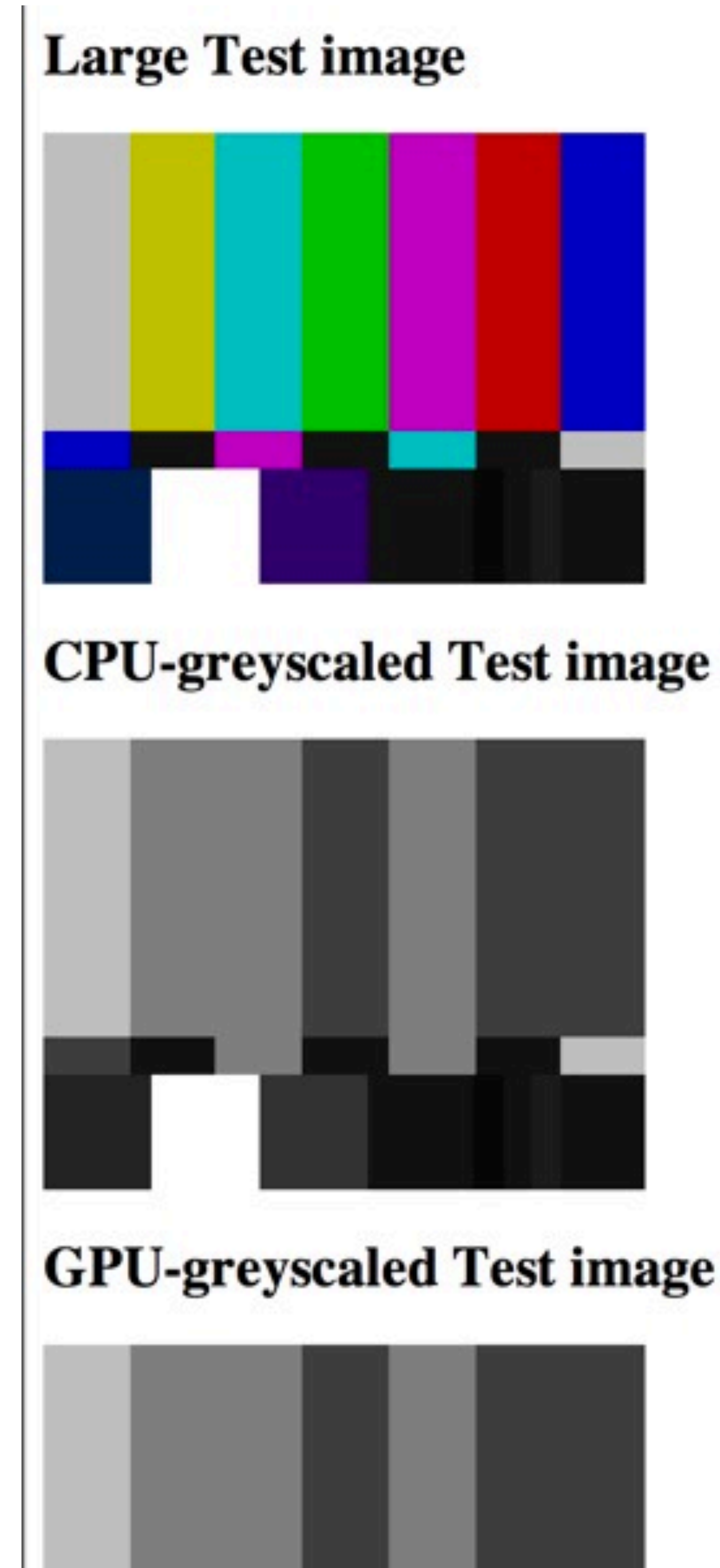

```
__global__ void greyscaleKernel(const int nrows, const int ncol
                                         const int *in_r_d, const int *in_g
                                         int *out_r_d, int *out_g_d, int *o

        int i = threadIdx.x + blockDim.x*blockIdx.x;
```




```
__global__ void greyscaleKernel(const int nrows, const int ncol
                                const int *in_r_d, const int *in_g
                                int *out_r_d, int *out_g_d, int *o

        int i = threadIdx.x + blockDim.x*blockIdx.x;
```

Multi-block greyscaling

- Now we can operate on full-sized image
- Only limit here is size of memory on GPU
- (can get from querydevs.cu; ~1GB on my laptop)



Let’s take 15 minutes to get familiar with this; modify example 2 so that it does something else to image:

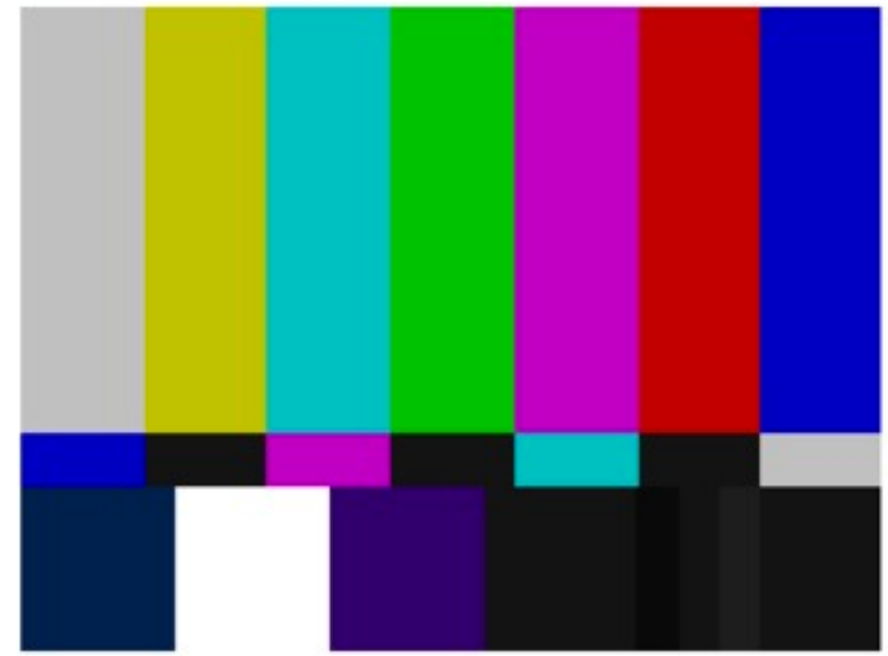
- Makes image red-only
- Puts big blue square in top right corner

...

Smoothing

- Smoothing/Blurring
- So far, we've done operations that only depend on the local pixel values.
- Many/most image processing algorithms also depend on neighbouring values.

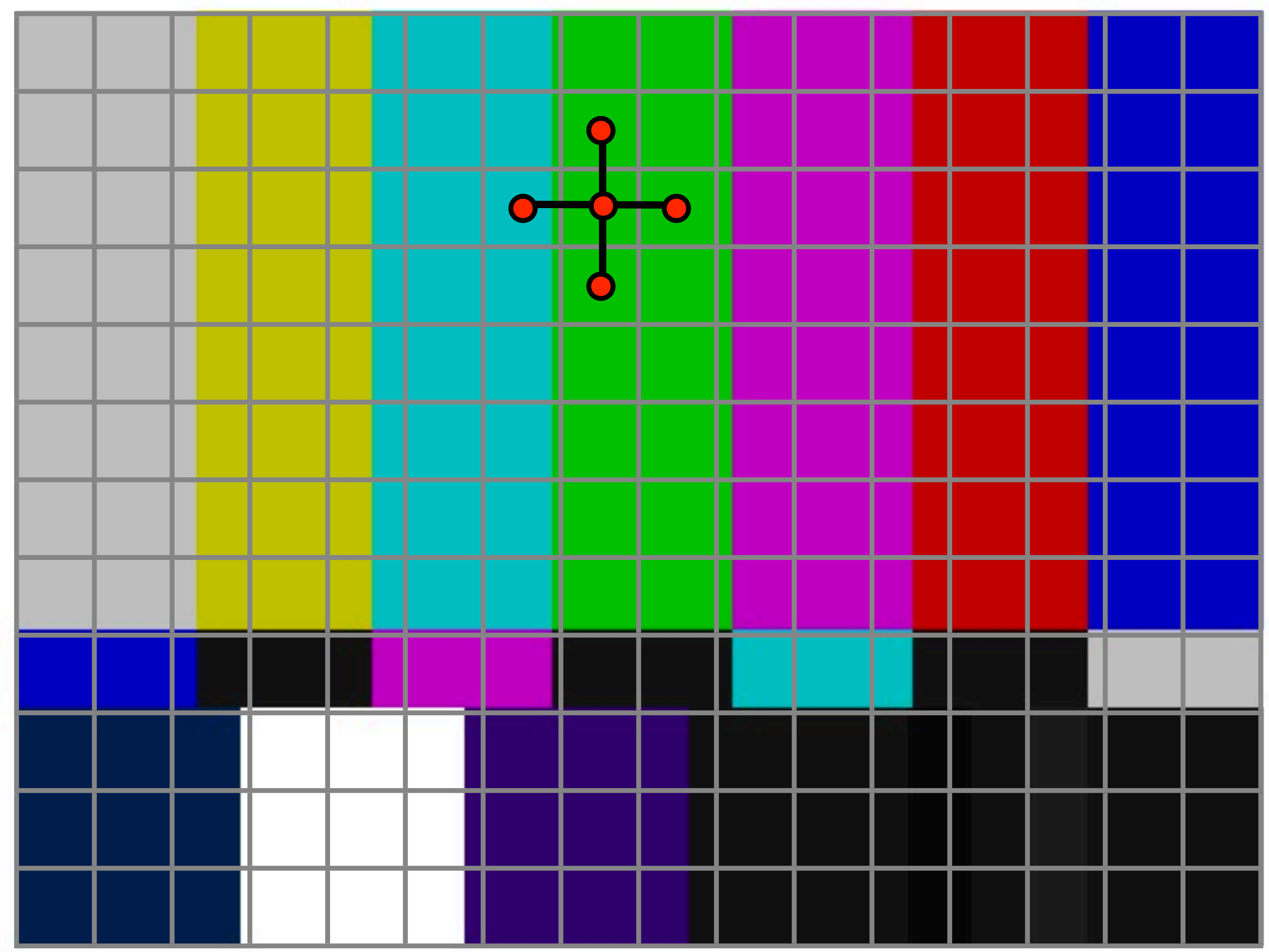
Large Test image



CPU-smoothed Test image

Smoothing

- "Stencil"
- For each point, consider it and it's nearest neighbour
- Take weighted average of r, g, b values
- Averages out noise
- Can use different stencil size - tradeoff between reducing noise and washing out small scale features.



SciNet

CPU Code

example3/testpattern.cu

```
__device__ __host__ int avg(const int *color, const int pix, const int nrows, const int ncols) {

    int avg = (                  color[pix-ncols]
              + color[pix-1] + color[pix] + color[pix+1] +
                             color[pix+ncols]     ) / 5;

    return (int)avg;
}
```

```
void cpuSmooth(const int nrows, const int ncols,
               const int *in_r, const int *in_g, const int *in_b,
               int *out_r, int *out_g, int *out_b) {

    int row, col;
    int pix = 0;
    for (row = 0; row<nrows; row++) {
        for (col = 0; col<ncols; col++) {
            if (row == 0 || row == nrows-1 || col == 0 || col == ncols-1) {
                out_r[pix] = in_r[pix];
                out_g[pix] = in_g[pix];
                out_b[pix] = in_b[pix];
            } else {
                out_r[pix] = avg(in_r, pix, nrows, ncols);
                out_g[pix] = avg(in_g, pix, nrows, ncols);
                out_b[pix] = avg(in_b, pix, nrows, ncols);
            }
            pix++;
```

SciNet

GPU Code

example3/testpattern.cu

```
/* do a greyscale image by averaging the colors at a point */
__global__ void smoothKernel(const int nrows, const int ncols,
                             const int *in_r_d, const int *in_g_d, const int *in_b_d,
                             int *out_r_d, int *out_g_d, int *out_b_d) {

        int pix = /* something else goes here */ 0;
        int row = pix/ncols;
        int col = (pix - row*ncols);

        if (pix < nrows*ncols) {
            /* do the appropriate work */
        }

        return;
}
```

GPU Code

example3/testpattern.cu

```
/* launch the kernel on our input image */
void gpuSmooth(const int nrows, const int ncols,
               const int *in_r, const int *in_g, const int *in_b,
               int *out_r, int *out_g, int *out_b, int blocksize) {

    int *in_r_d, *in_g_d, *in_b_d;
    int *out_r_d, *out_g_d, *out_b_d;

    /* cudaMalloc the device arrays*/

    /* cudaMemcpy the input data */

    /* calculate number of blocks we need -- round up */
    int nblocks = (nrows*ncols + blocksize - 1)/blocksize;
    smoothKernel<<<nblocks, blocksize>>>(nrows, ncols, in_r_d, in_g_d, in_b_d, out_r_d, out_g_d, out_b_d);

    CHK_ERROR ;

    /* cudaMemcpy the output data */

    /* cudaFree the device arrays */

    return;
}
```