GPU Teaching Kit

Accelerated Computing

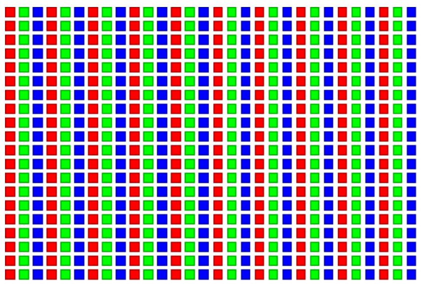
CUDA Parallelism Model

Color-to-Grayscale Image Processing Example

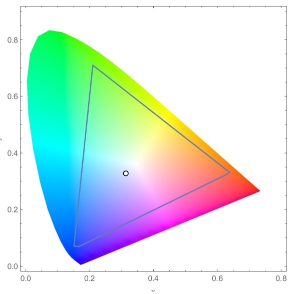
Objective

– To gain deeper understanding of multi-dimensional grid kernel configurations through a real-world use case

RGB Color Image Representation



- Each pixel in an image is an RGB value
- The format of an image's row is (r g b) (r g b) … (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdobeRGB color space
  - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R
  - The triangle contains all the representable colors in this color space

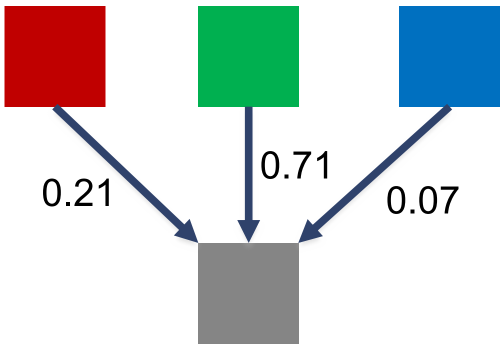


RGB to Grayscale Conversion

A grayscale digital image is an image in which the value of each pixel carries only intensity information.

Color Calculating Formula

– For each pixel (r g b) at (I, J) do:
  grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b
– This is just a dot product <[r,g,b],[0.21,0.71,0.07]> with the constants being specific to input RGB space



RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {










 }
}
```

RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
    unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
    unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel



 }
}
```

RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
    unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
    unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

GPU Teaching Kit

Accelerated Computing

CUDA Parallelism Model

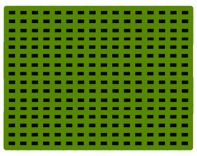
Image Blur Example

Objective

– To learn a 2D kernel with more complex computation and memory access patterns

Image Blurring

Blurring Box



Pixels processed by a thread block

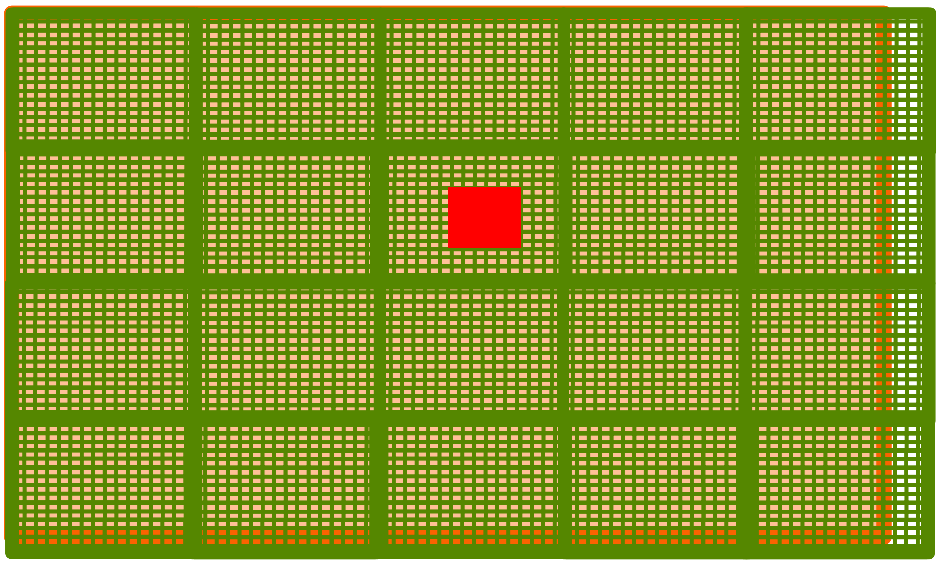


Image Blur as a 2D Kernel

```
 __global__
  void blurKernel(unsigned char * in, unsigned char * out,
int w, int h)
  {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        ... // Rest of our kernel
    }
  }
```

```
 __global__
  void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
      int Col  = blockIdx.x * blockDim.x + threadIdx.x;
      int Row  = blockIdx.y * blockDim.y + threadIdx.y;

      if (Col < w && Row < h) {
          int pixVal = 0;
          int pixels = 0;

          // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
          for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
              for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                  int curRow = Row + blurRow;
                  int curCol = Col + blurCol;
                  // Verify we have a valid image pixel
                  if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                      pixVal += in[curRow * w + curCol];
                      pixels++; // Keep track of number of pixels in the
accumulated total
                  }
              }
          }

          // Write our new pixel value out
          out[Row * w + Col] = (unsigned char)(pixVal / pixels);
      }
  }
```

GPU Teaching Kit

Accelerated Computing

Accelerated Computing

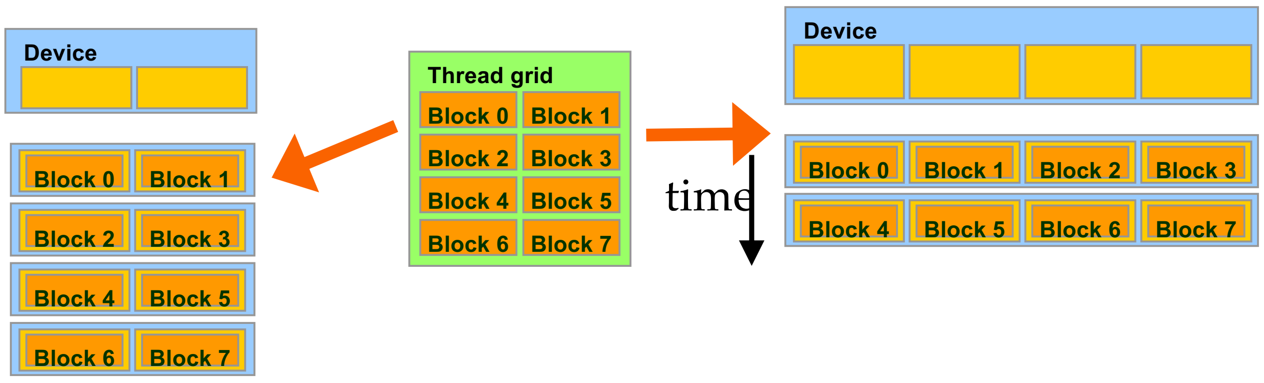
CUDA Parallelism Model

Thread Scheduling

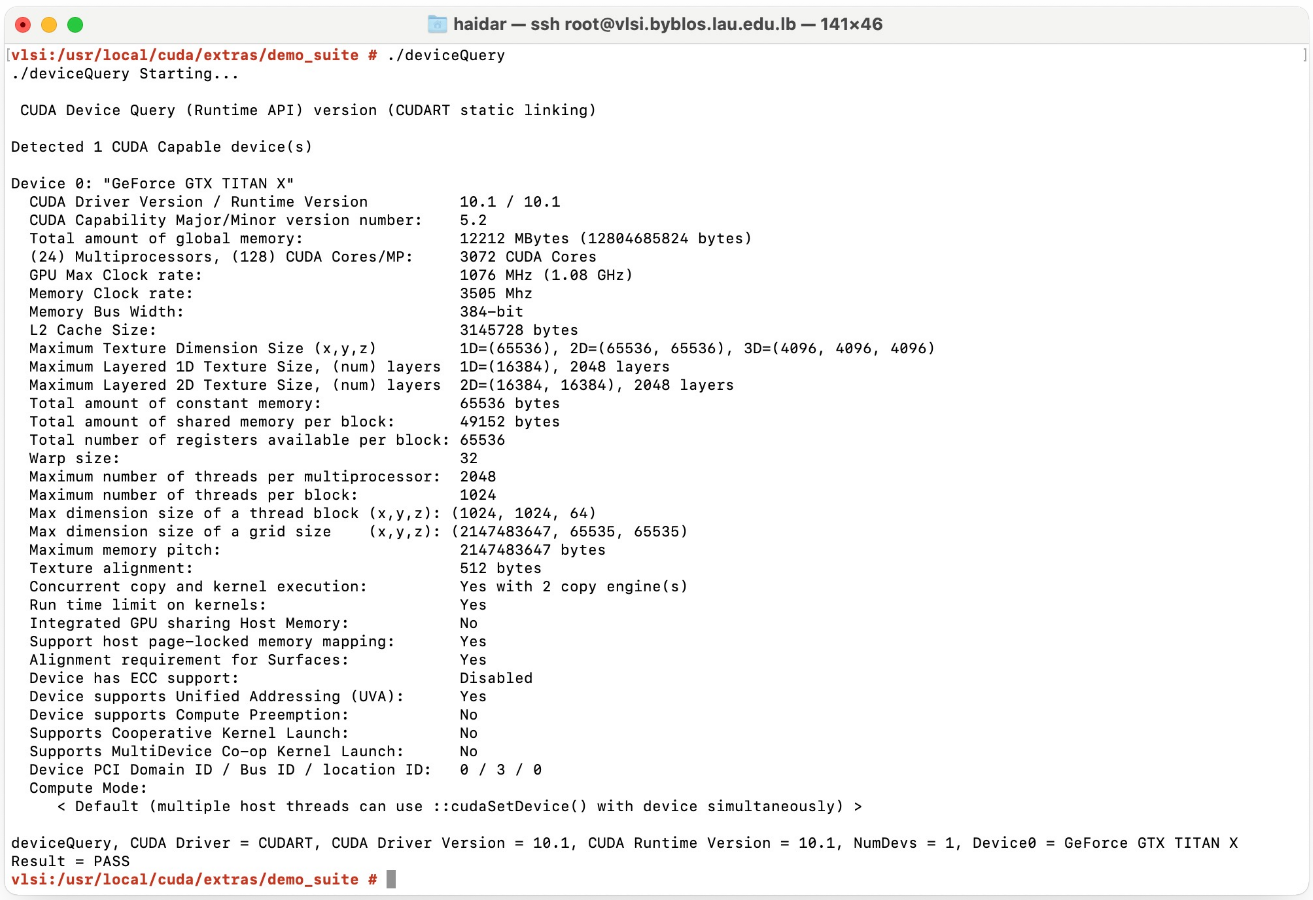
Objective

- To learn how a CUDA kernel utilizes hardware execution resources
  - Assigning thread blocks to execution resources
  - Capacity constrains of execution resources
  - Zero-overhead thread scheduling

Transparent Scalability



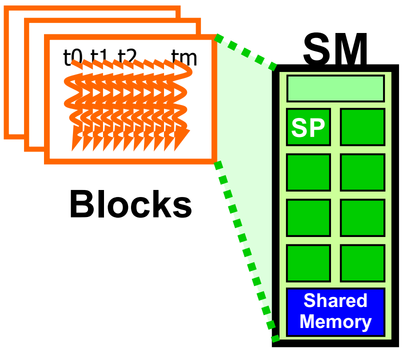
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors



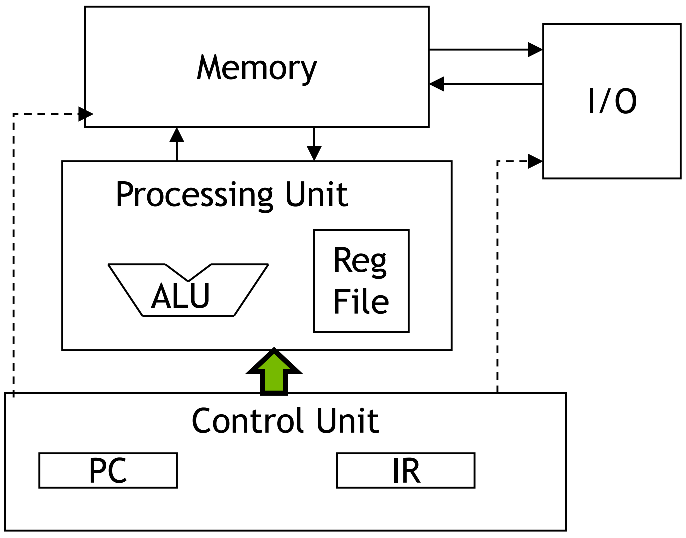
```
vlsi:/usr/local/cuda/extras/demo_suite # ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 12212 MBytes (12804685824 bytes)
  (24) Multiprocessors, (128) CUDA Cores/MP:     3072 CUDA Cores
  GPU Max Clock rate:                            1076 MHz (1.08 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1, Device0 = GeForce GTX TITAN X
Result = PASS
vlsi:/usr/local/cuda/extras/demo_suite #
```

Example: Executing Thread Blocks

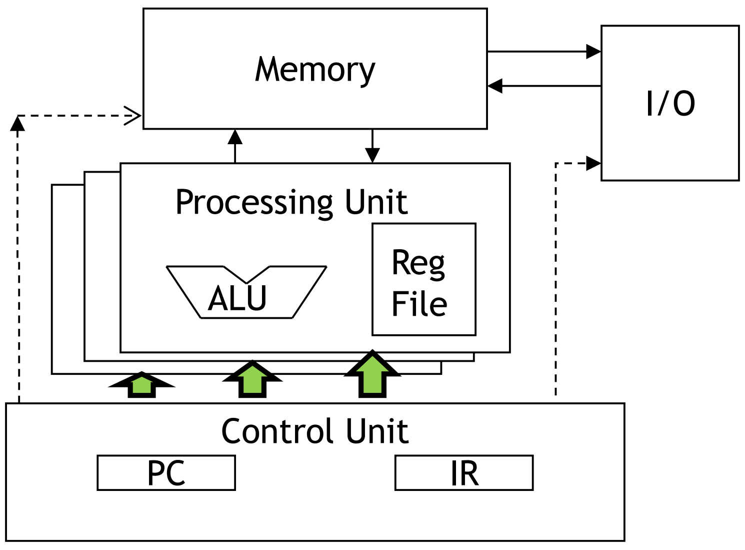
- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to 32 blocks to each SM as resource allows
  - Volta SM can take up to **2048** threads
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



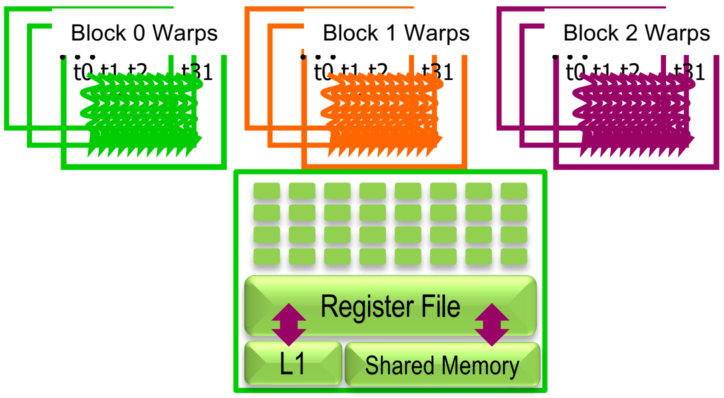
Single Instruction Multiple Data
(SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?
  - For 4X4, we have 16 threads per Block. Each SM can take up to 2048 threads, which translates to 128 Blocks. However, each SM can only take up to 32 Blocks, so only 512 threads will go into each SM!
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 2048 threads, it can take up to 32 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 30X30, we would have 900 threads per Block. Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

GPU Teaching Kit

Accelerated Computing