CUDA Threads

Performance considerations

Sergio Orts Escolano
Albert García García

sorts @ ua.es
agarcia @ dtic.ua.es

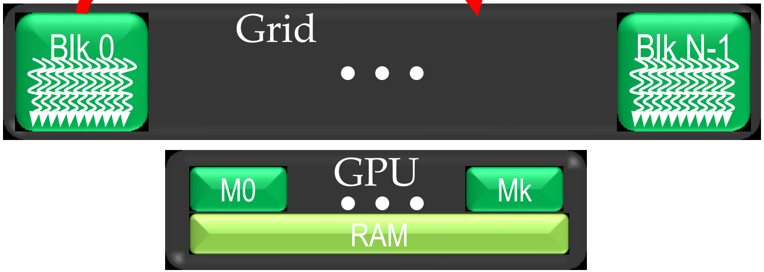
Objectives

- To understand how CUDA threads execute on SIMD Hardware
  - Warp partitioning
  - SIMD Hardware
  - Control divergence
- To learn to analyze the performance impact of control divergence
  - Boundary condition checking
  - Control divergence is data-dependent
- Synchronization

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B
,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
  int i = blockIdx.x * blockDim.x
          + threadIdx.x;

  if( i<n ) C[i] = A[i]+B[i];
}
```

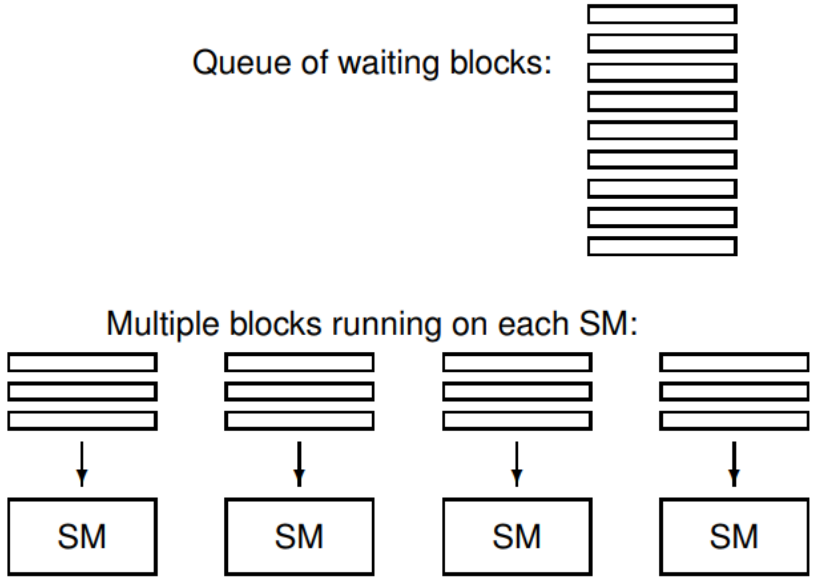


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

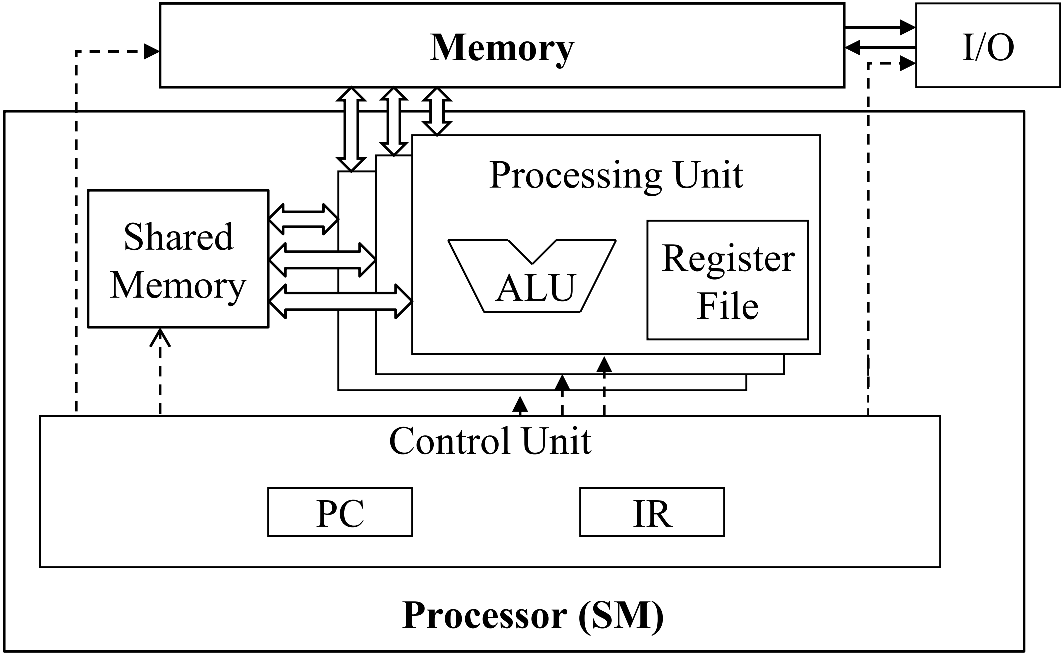
Thread block scheduling



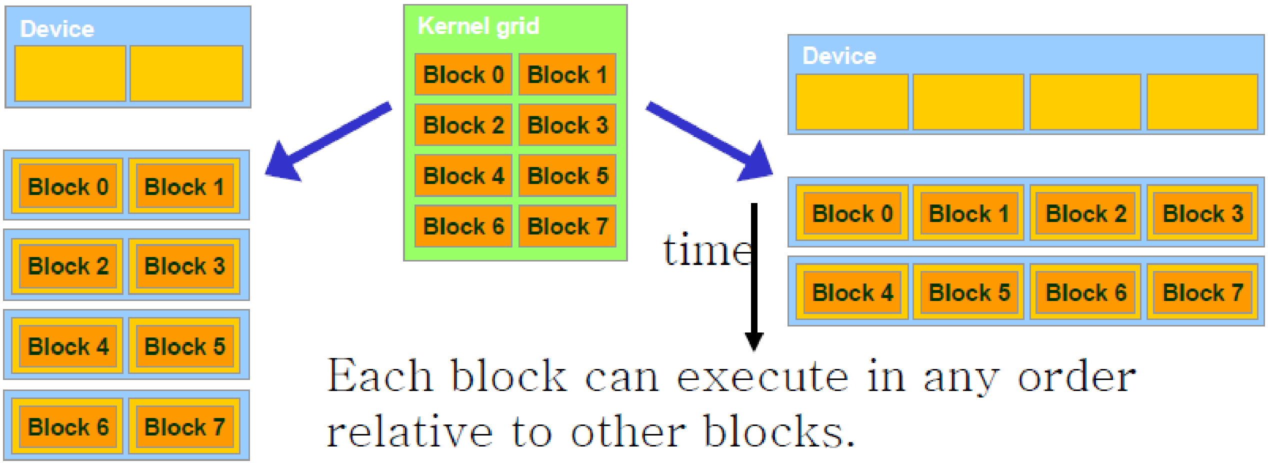
Programmer doesn't have to worry about this level of detail, just make sure there are lots of threads / warps

SMs are SIMD Processors

- Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized (Module 1)



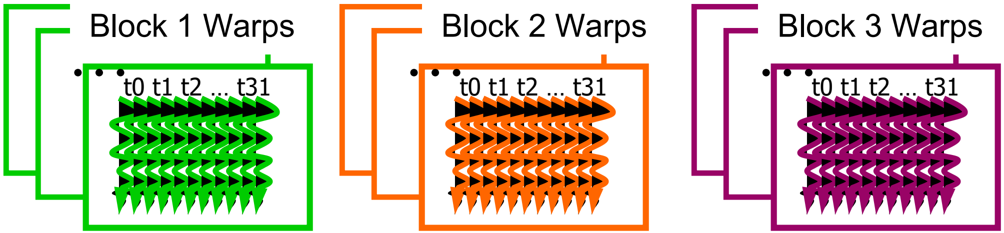
Thread block scheduling



The threads are assigned to execution in a block-by-block basis.
A SM can hold a maximum number of blocks at a certain time.

One of the SM resource limitations is the number of threads that can be simultaneously tracked and schedule.

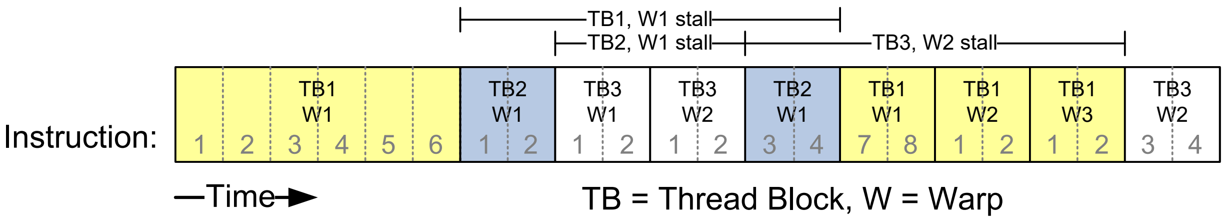
Warps as Scheduling Units



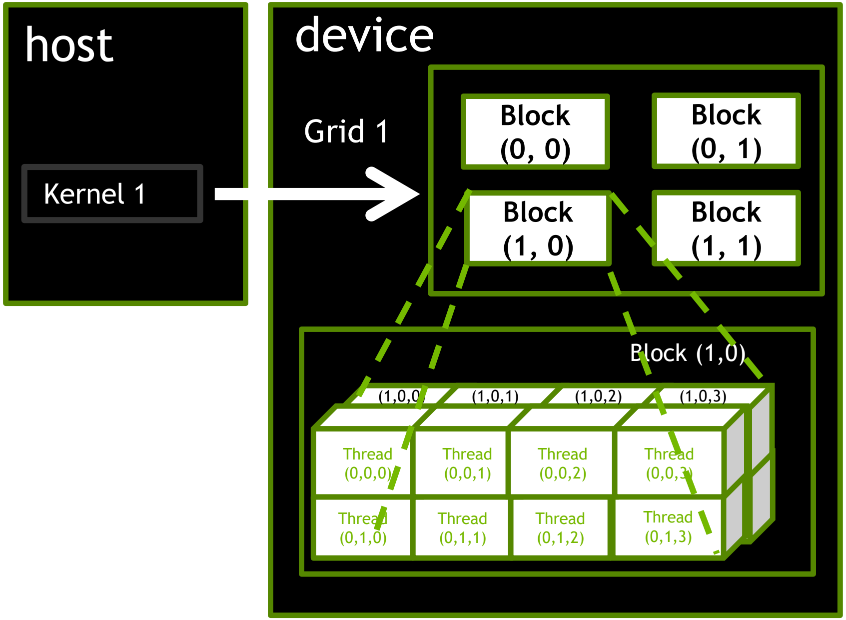
- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps as Scheduling

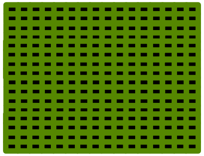
- Zero-overhead warp scheduling
  - Latency hiding
  - Priority mechanism
  - Selection of warps which are ready for execution



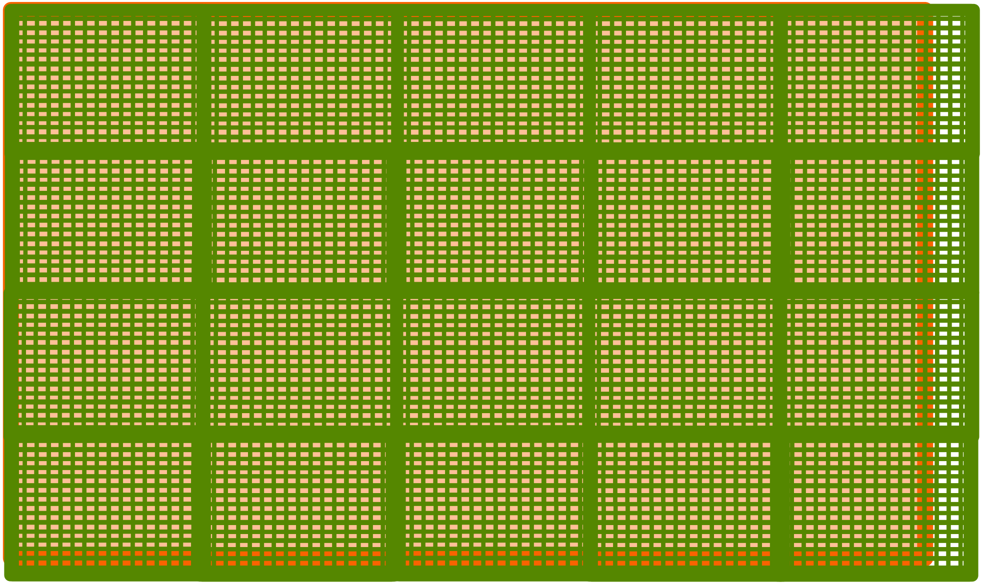
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid



16×16 blocks



62×76 picture

Source Code of a PictureKernel

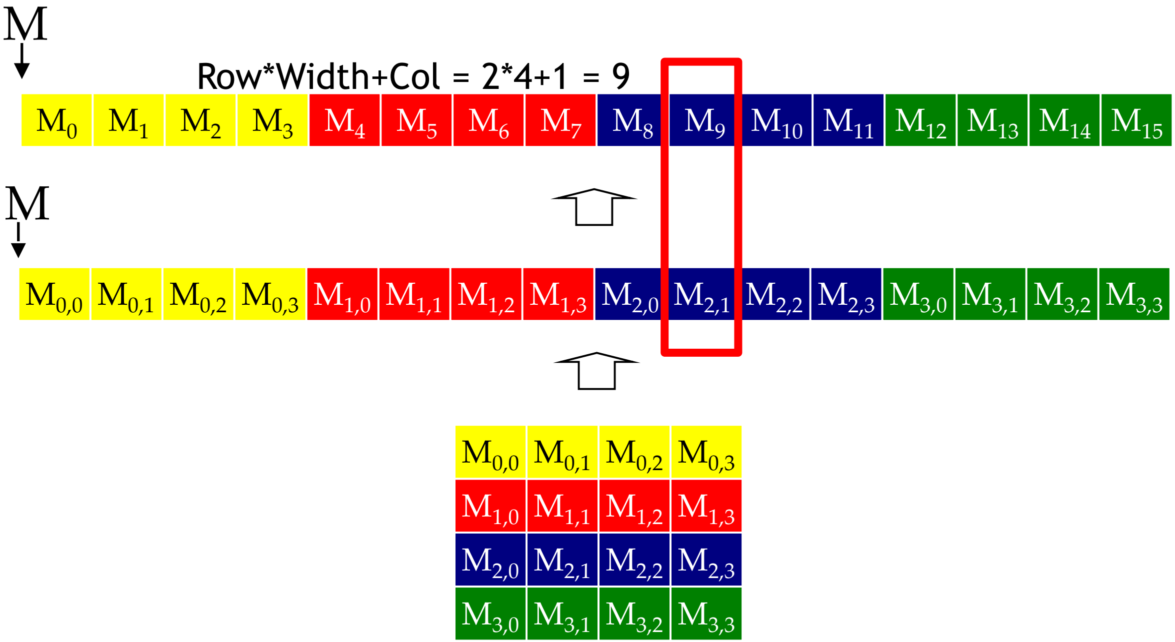
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                               int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

Scale every pixel value by 2.0

Host Code for Launching PictureKernel

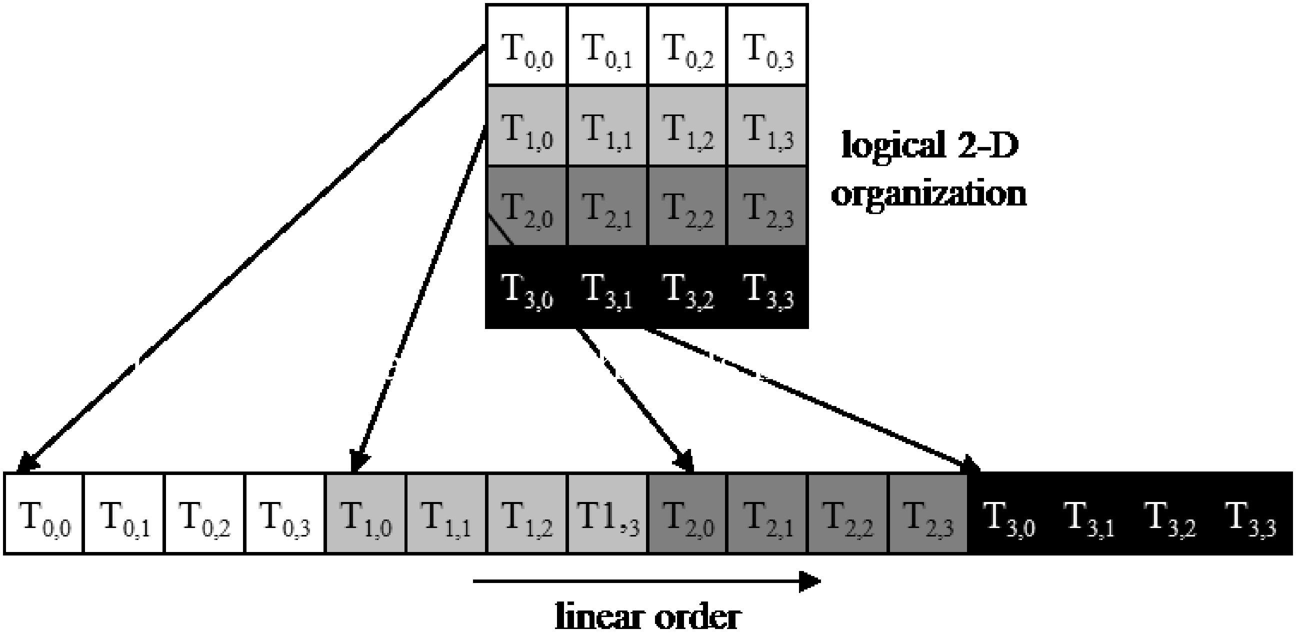
```
// assume that the picture is m×n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
…
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
…
```

Row-Major Layout in C/C++



Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last



Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0

- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation

- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time

- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times

Control Divergence

- **Control divergence** occurs when threads in a warp take different control flow paths
  - Some take the **then-path** and others take the **else-path** of an if-statement
  - Some threads take **different number of loop iterations** than others

- The execution of threads taking different paths are serialized in current GPUs (loss of performance)
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - `if (threadIdx.x > 2) { }`
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - `If (blockIdx.x > 2) { }`
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path
- Doesn't matter what is happening with other warps
  - Each warp is treated separately
  - if each warp only goes one way that's very efficient

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C,
  int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i<n) C[i] = A[i] + B[i];
}
```

Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
  - 8 warps in each block
- All threads in Blocks 0, 1, and 2 are within valid range
  - i values from 0 to 767
  - There are 24 warps in these three blocks, none will have control divergence
- Most warps in Block 3 will not control divergence
  - Threads in the warps 0-6 are all within valid range, thus no control divergence
- One warp in Block 3 will have control divergence
  - Threads with i values 992-999 will all be within valid range
  - Threads with i values of 1000-1023 will be outside valid range
- Effect of serialization on control divergence will be small
  - 1 out of 32 warps has control divergence
  - The impact on performance will likely be less than 3%

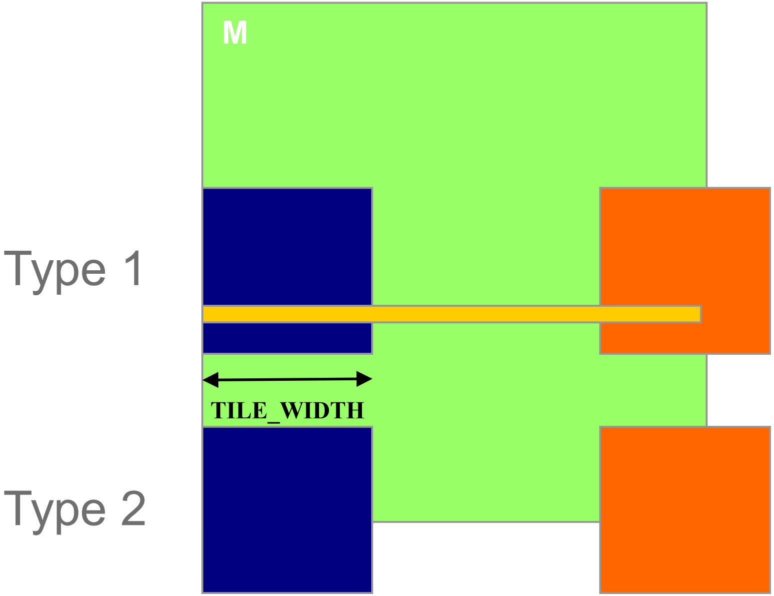
Performance Impact of Control Divergence

- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation
  - For example, see the tile loading code below:

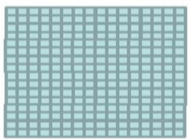
```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
   ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
} else {
 ds_M[ty][tx] = 0.0;
}

if (p*TILE_WIDTH+ty < Width && Col < Width) {
   ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
} else {
   ds_N[ty][tx] = 0.0;
}
```

Two types of blocks in loading M Tiles

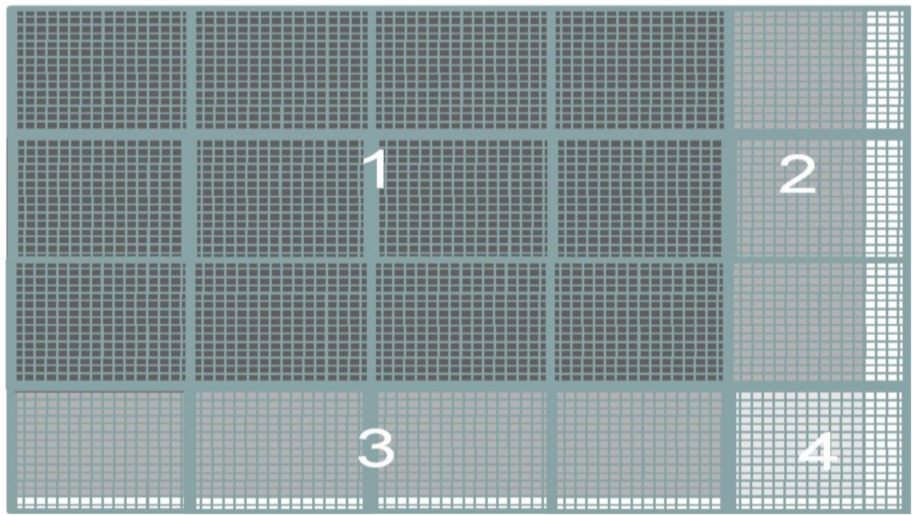
- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way



Covering a 62×76 Picture with 16×16 Blocks



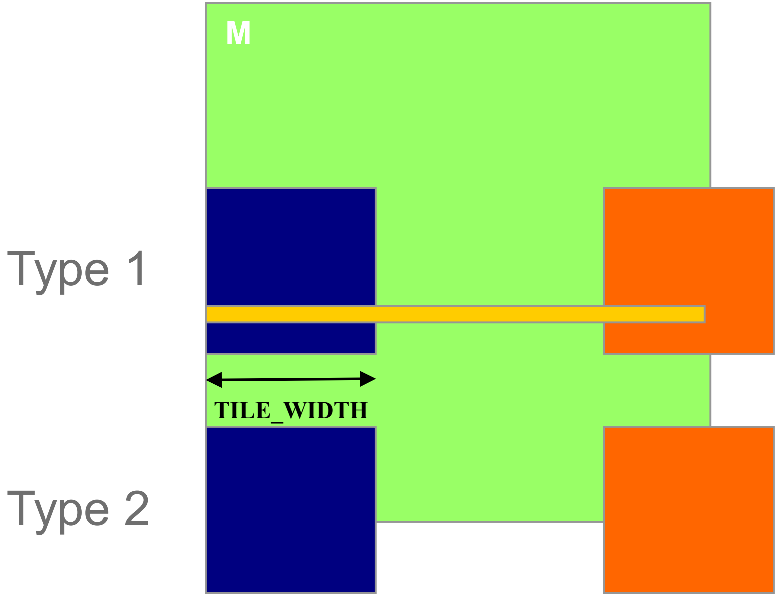
16×16 block



Not all threads in a Block will follow the same control flow path.

Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12%



Additional Comments

- The estimated performance impact is data dependent.
  - For larger matrices, the impact will be significantly smaller
- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant
  - One should not hesitate to use boundary checks to ensure full functionality
- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence
- If boundary conditions are cheap, loop over all nodes and branch as needed for boundary conditions
- if boundary conditions are expensive, use two kernels:
  - first for interior points, second for boundary points

Synchronization

- `__syncthreads();` which forms a barrier – all threads wait until every one has reached this point (block-level).
- When writing conditional code, must be careful to make sure that all threads do reach the `__syncthreads();`
- Otherwise, can end up in deadlock
- __threadfence_block();
  wait until memory accesses are visible to block
- __threadfence();
  wait until memory accesses are visible to block and device

Synchronization

- There are other synchronisation instructions which are similar but have extra capabilities:
  - `int __syncthreads_count(predicate)`

    counts how many predicates are true

  - `int __syncthreads_and(predicate)`

    returns non-zero (true) if all predicates are true

  - `int __syncthreads_or(predicate)`

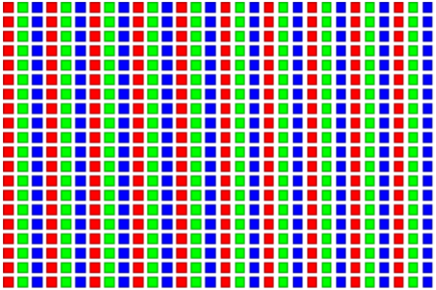
    returns non-zero (true) if any predicate is true

- There are similar warp voting instructions which operate at the level of a warp:
  - `int __all(predicate)`, `int __any(predicate)`, `unsigned int __ballot(predicate)`

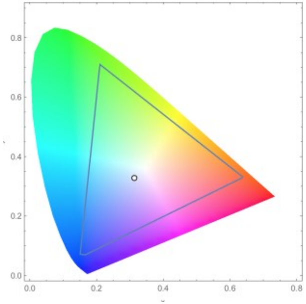
Synchronization

- There are similar warp voting instructions which operate at the level of a warp:
  - `int __all(predicate)`
    returns non-zero (true) if all predicates in warp are true
  - `int __any(predicate)`
    returns non-zero (true) if any predicate is true
  - `unsigned int __ballot(predicate)`
    sets n^{th} bit based on n th predicate

CV example: Color to Grayscale
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 AbobeRGB 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



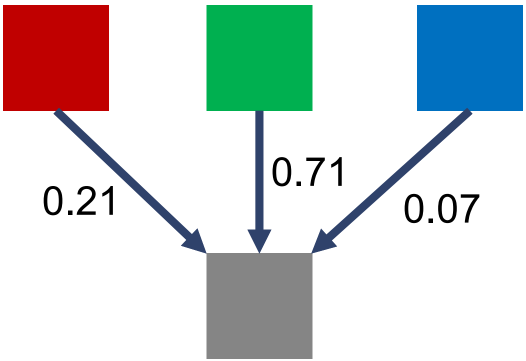
3DPERCEPTIONLAB

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



3DPERCEPTIONLAB

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;
 }
}
```

Timing CUDA kernels

cudaEventCreate ( ), cudaEventRecord ( ), cudaEventSynchronize ( ), cudaEventElapsedTime ( ) y **cudaEventDestroy ( )**

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
for (int i=0; i<2; ++i)
    cudaMemcpyAsync(inputDevPtr+i*size, inputHost+i*size,
                    size, cudaMemcpyHostToDevice, stream[i]);

for (int i=0; i<2; ++i)
    mikernel<<<100, 512, 0, stream[i]>>>
            (outputDev+i*size, inputDev+i*size, size);

for(int i=0; i<2; ++i)
    cudaMemcpyAsync(outputHostPtr+i*size, outputDevPtr+i*size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

CUDA Threads

Performance considerations

Thanks for your attention!

Sergio Orts Escolano
Albert García García

sorts @ ua.es
agarcia @ dtic.ua.es