中国科学院大学
University of Chinese Academy of Sciences

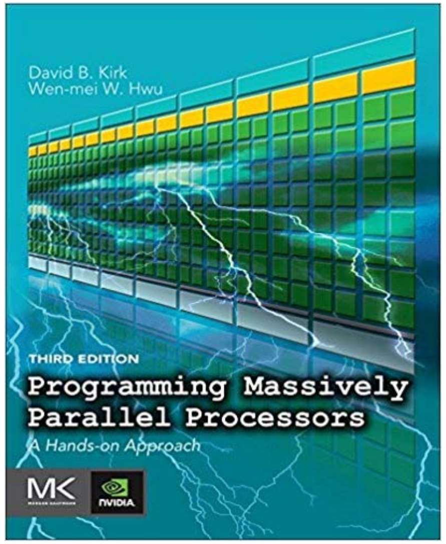
Introduction to CUDA

(2) Programming Model

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

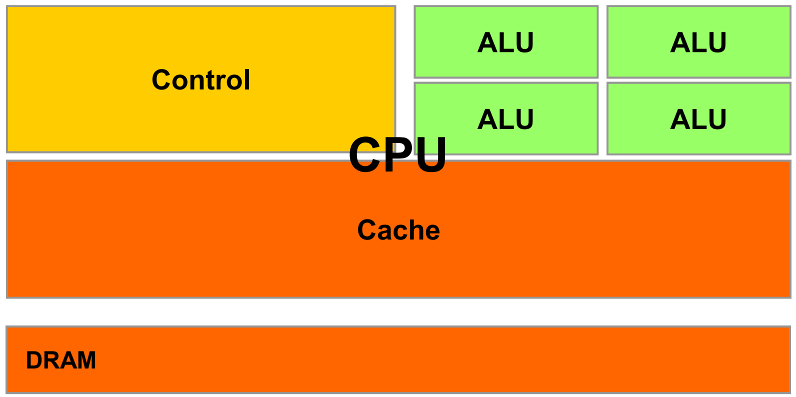


Content

- Heterogeneous parallel computing

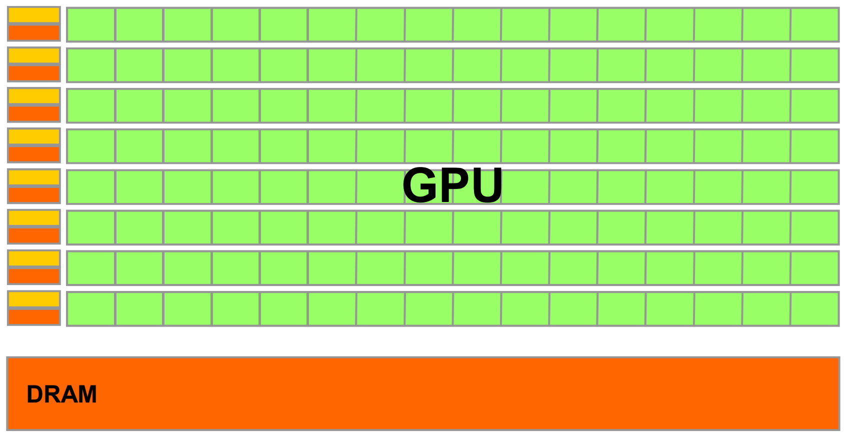
CPUs: Latency Oriented Design

- High clock frequency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



GPUs: Throughput Oriented Design

- Moderate clock frequency
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



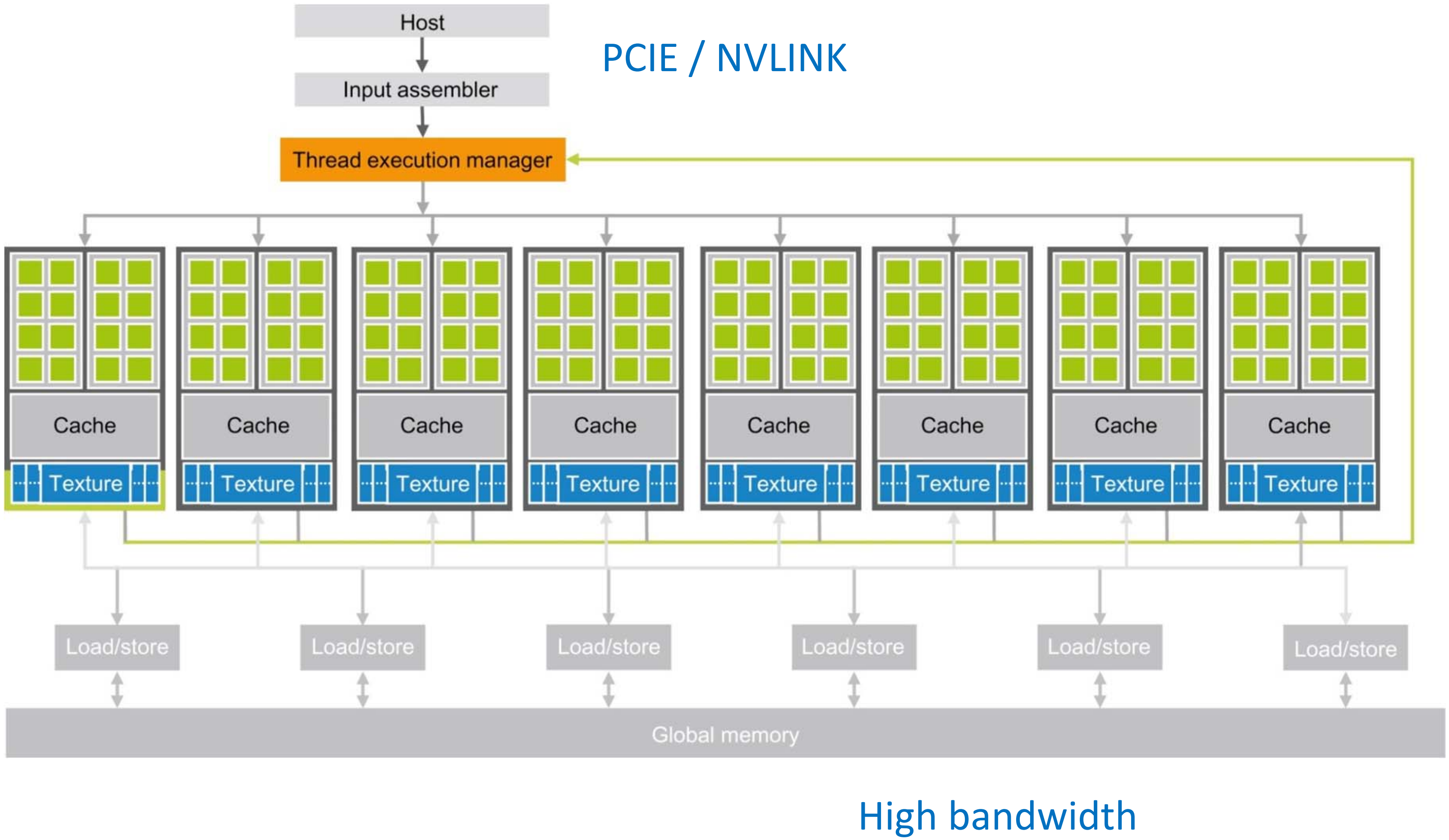
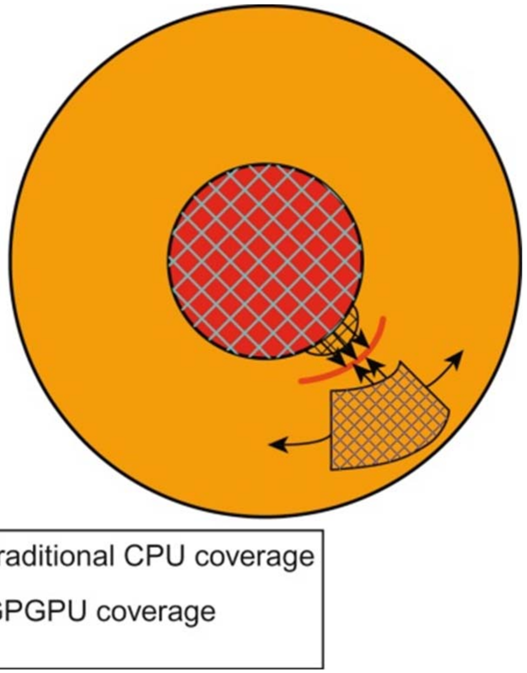


FIGURE 1.2: Architecture of a CUDA-capable GPU.

Applications Benefit from Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code



Heterogeneous parallel computing is catching on.

Financial Analysis	Scientific Simulation	Engineering Simulation	Data Intensive Analytics	Medical Imaging
Digital Audio Processing	Digital Video Processing	Computer Vision	Biomedical Informatics	Electronic Design Automation
Statistical Modeling	Ray Tracing Rendering	Interactive Physics	Numerical Methods	

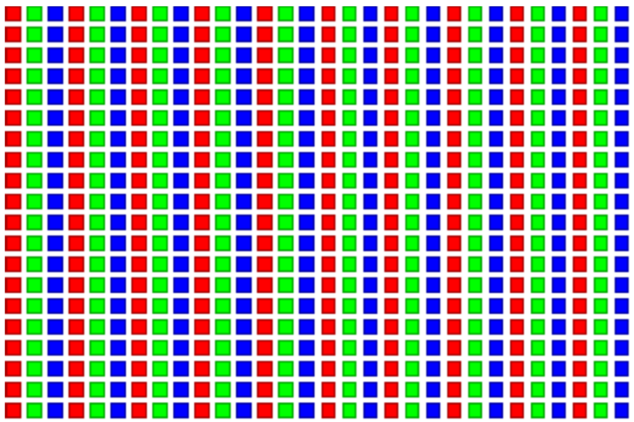
- 280 submissions to GPU Computing Gems
  - 110 articles included in two volumes

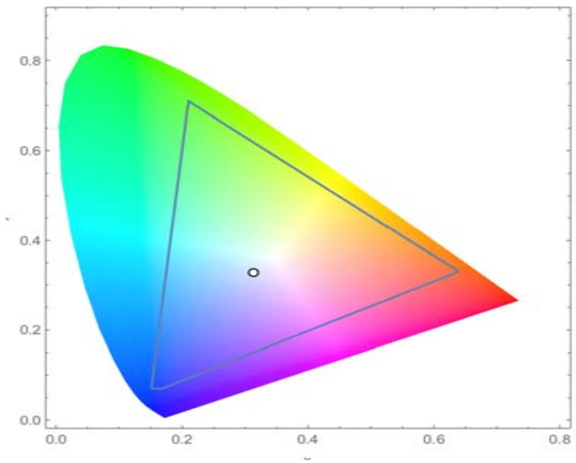
Content

- Data Parallel Programming

Example of data parallel

Conversion of a color image to grey–scale image





The pixels can be calculated independently of each other

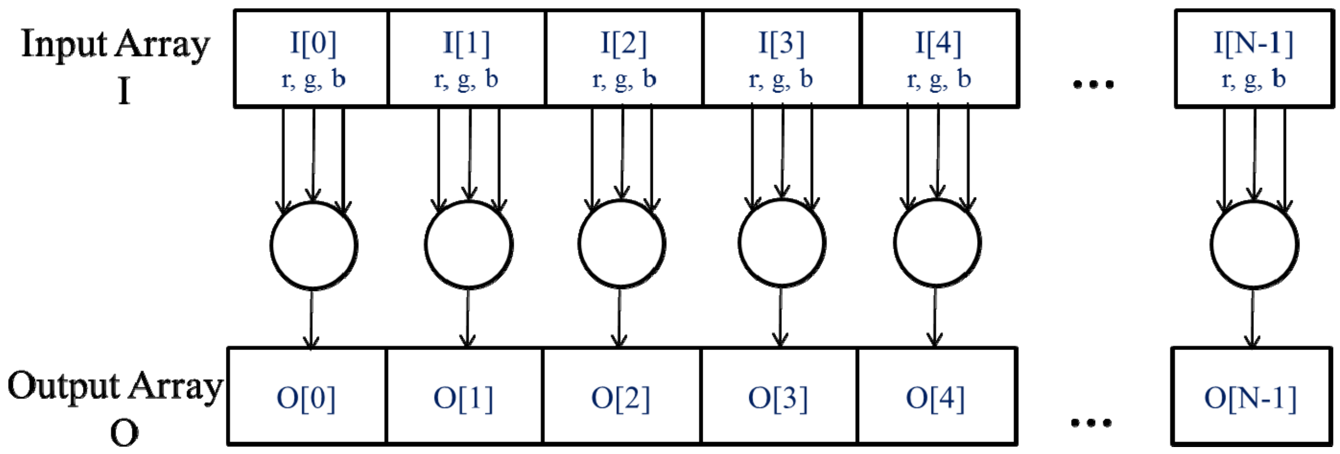
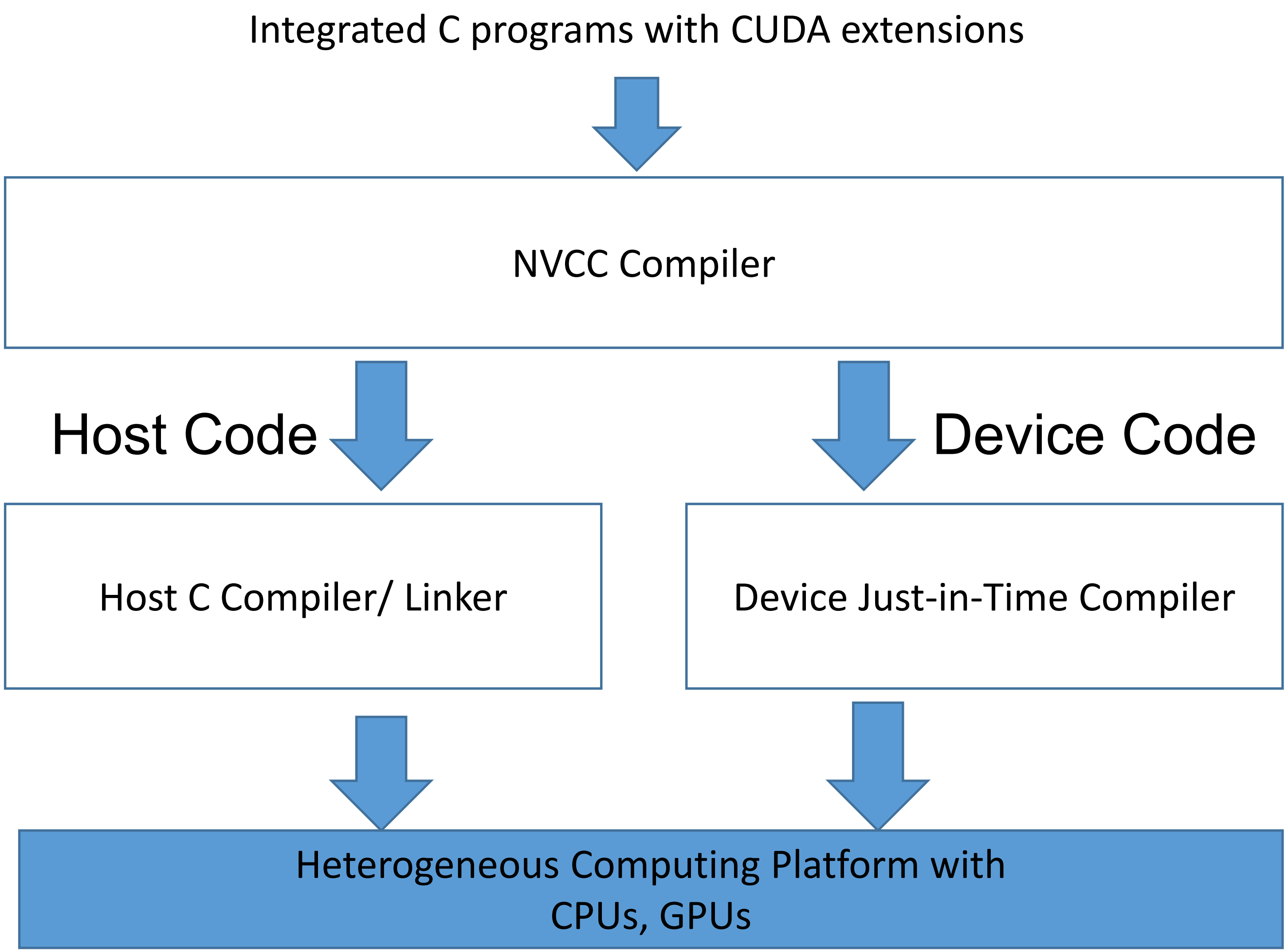


FIGURE 2.2: The pixels can be calculated independently of each other during color to greyscale conversion.

Content

- CUDA C program Structure

Compiling A CUDA Program



CUDA/OpenCL – Execution Model

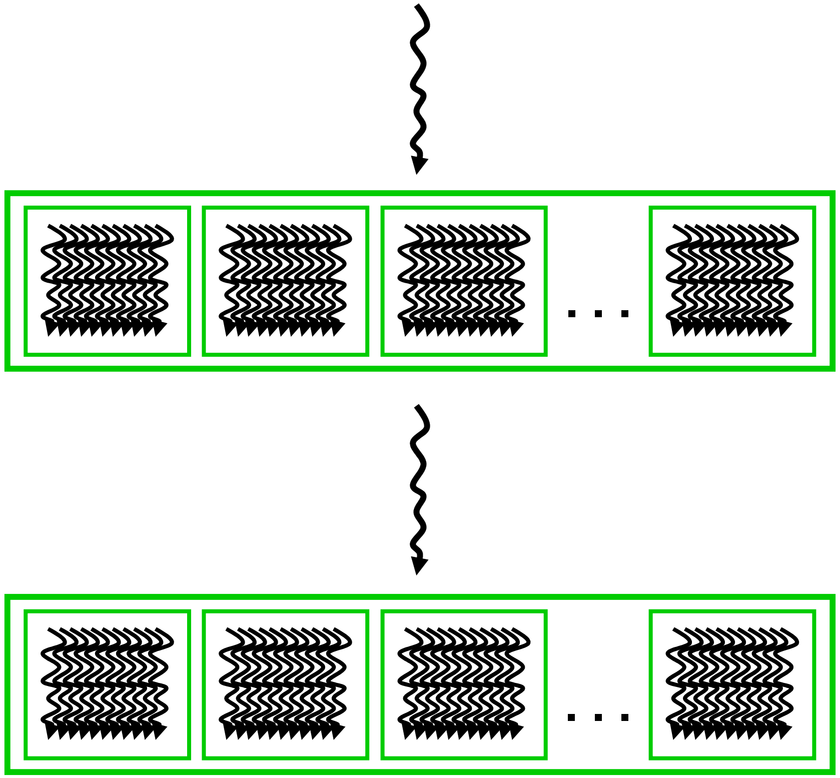
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

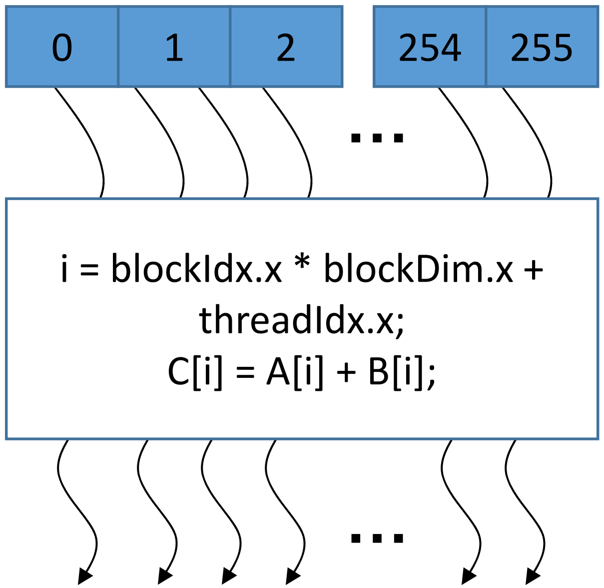
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



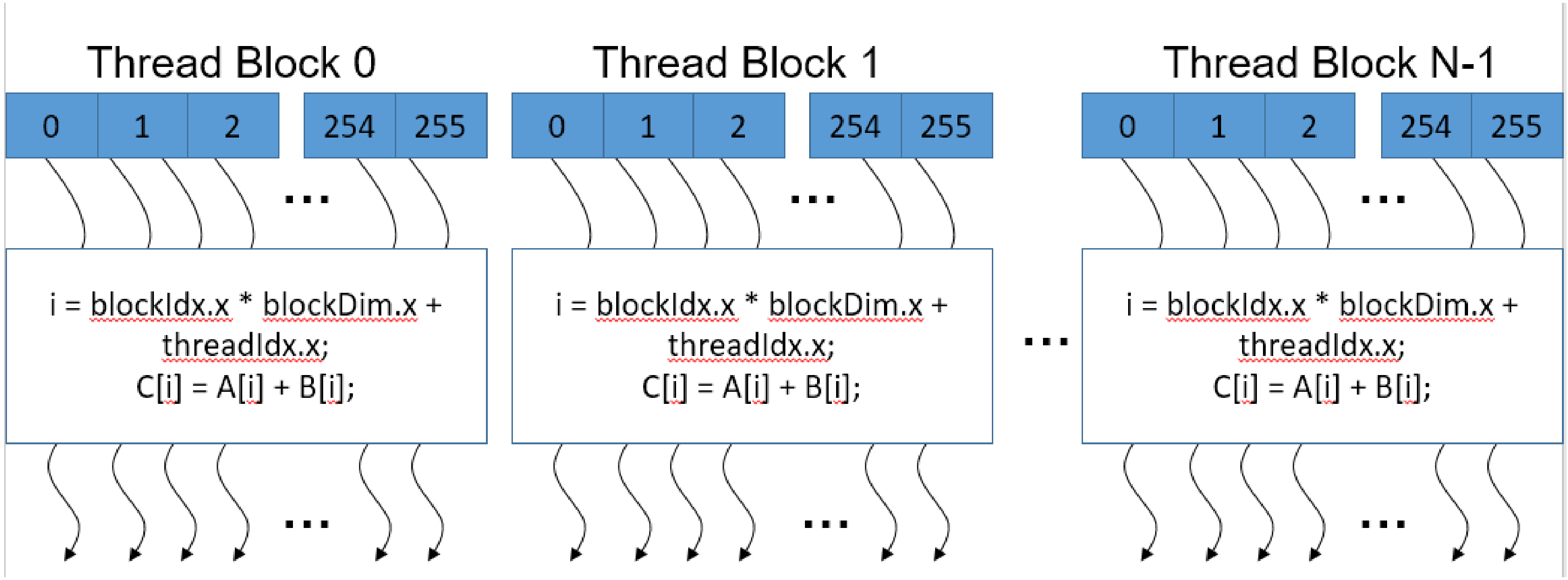
Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions



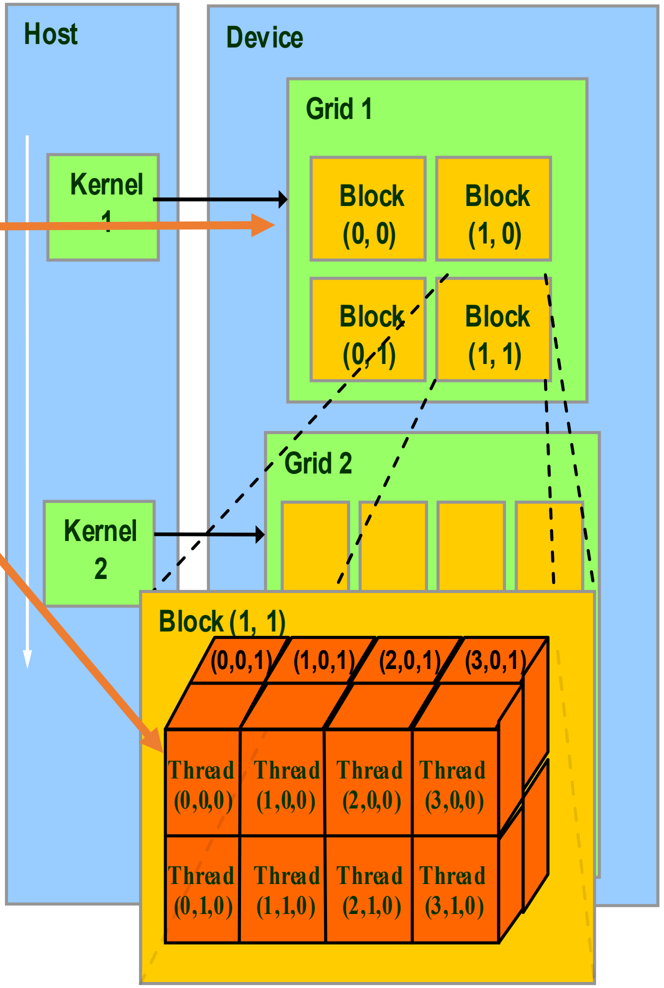
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Content

- Example: A Vector Addition Kernel
-

Vector Addition: traditional

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

A simple traditional vector addition C code example.

Vector Addition: CUDA

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
  int  size = n* sizeof(float);
  float  *d_A *d_B, *d_C;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

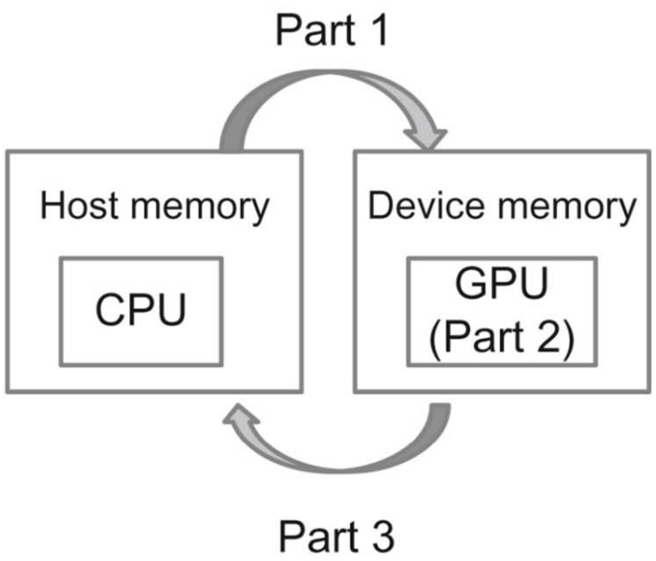
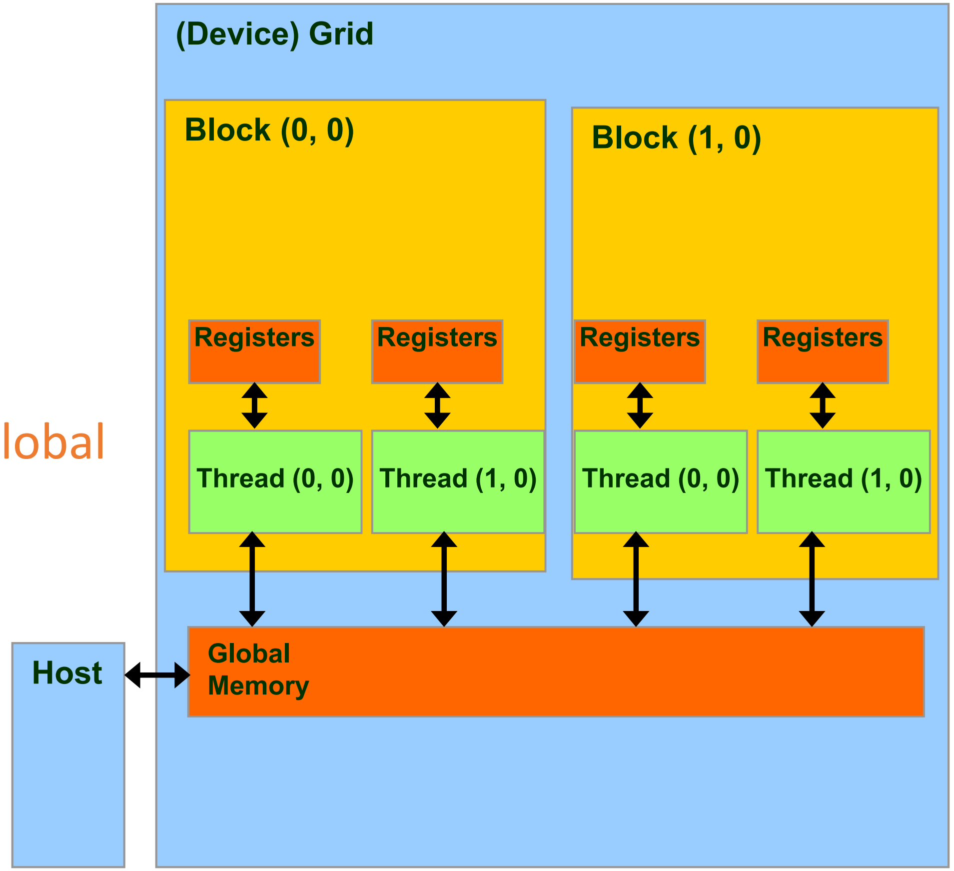


FIGURE 2.6: Outline of a revised vecAdd function that moves the work to a device.

Partial Overview of CUDA Memories

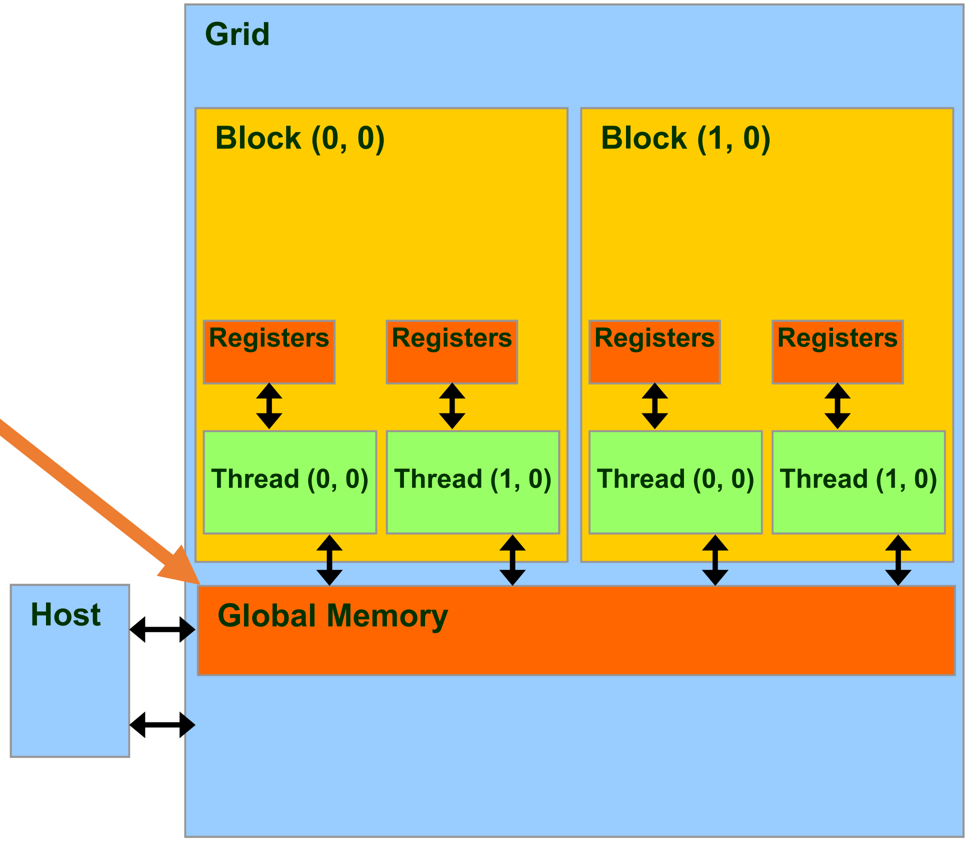
- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

- Host code can
  - Transfer data to/from per grid global memory



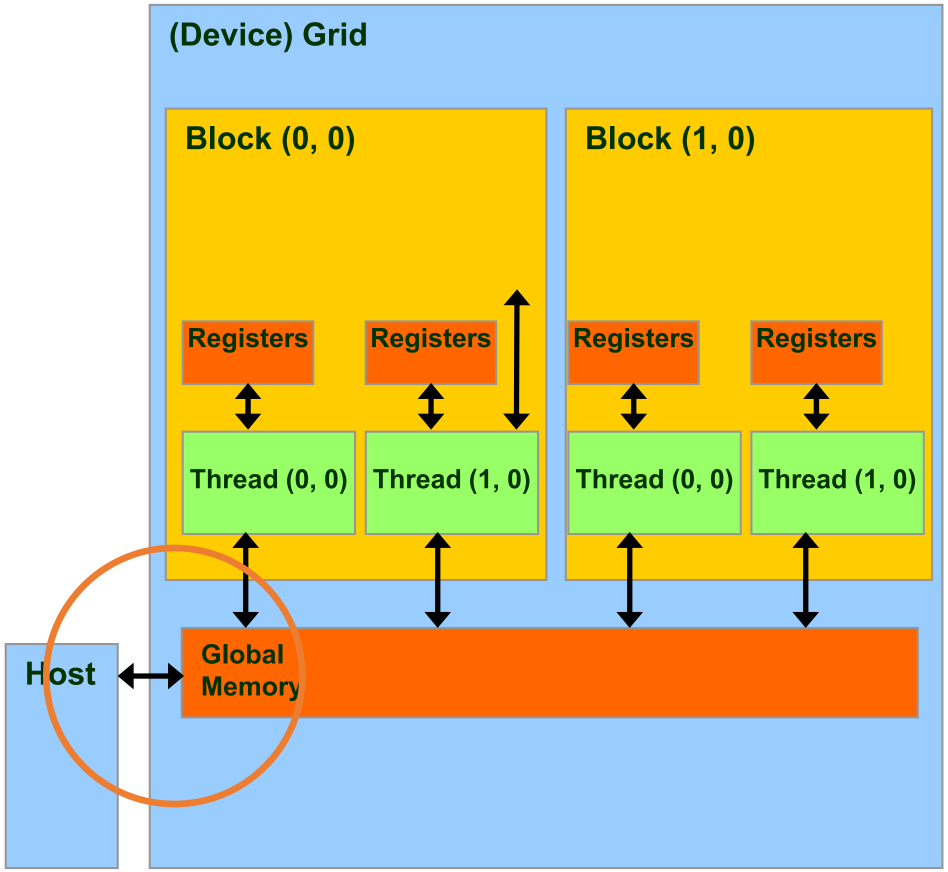
CUDA Device Memory Management

- cudaMalloc()
  - Allocates object in the device <u>global memory</u>
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **Pointer** to freed object



Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous



A more complete version of vecAdd()

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

```
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

Host Code

More on Kernel Launch

Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(n/256, 1, 1);
  if (n%256) DimGrid.x++;
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

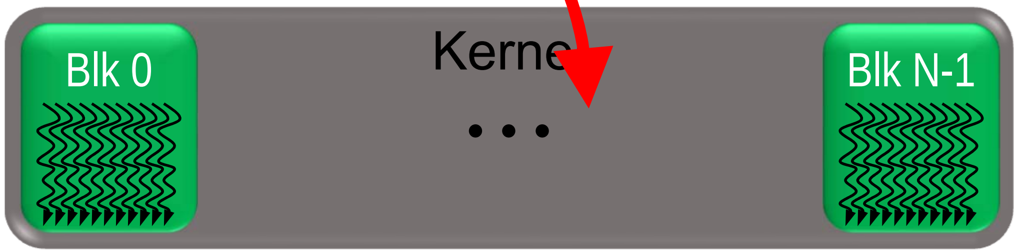
Kernel execution in a nutshell

```
__host__
Void vecAdd()
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

vecAddKernel<<<DimGrid,DimBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
      float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



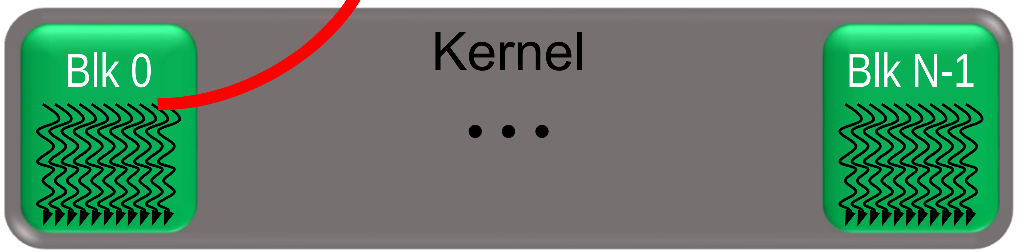
Kernel execution in a nutshell

```
__host__
Void vecAdd()
{
  dim3
DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

vecAddKernel<<<DimGrid,DimBlock>
>>(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
      float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
      + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



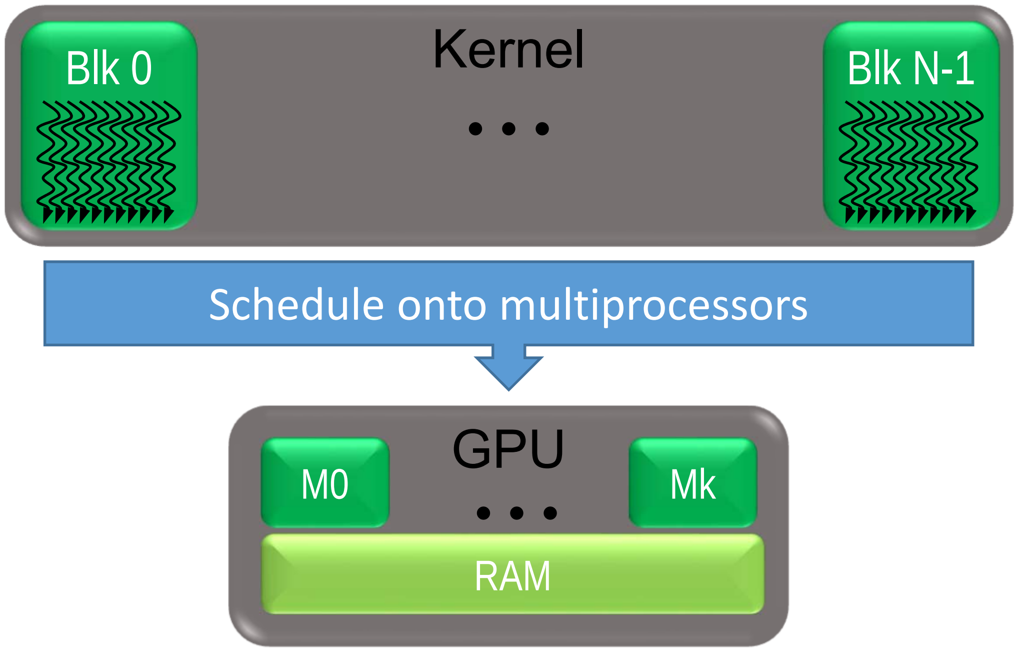
Kernel execution in a nutshell

```
__host__
Void vecAdd()
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

vecAddKernel<<<DimGrid,DimBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
      float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



A complete version of the host

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

More on CUDA Function Declarations

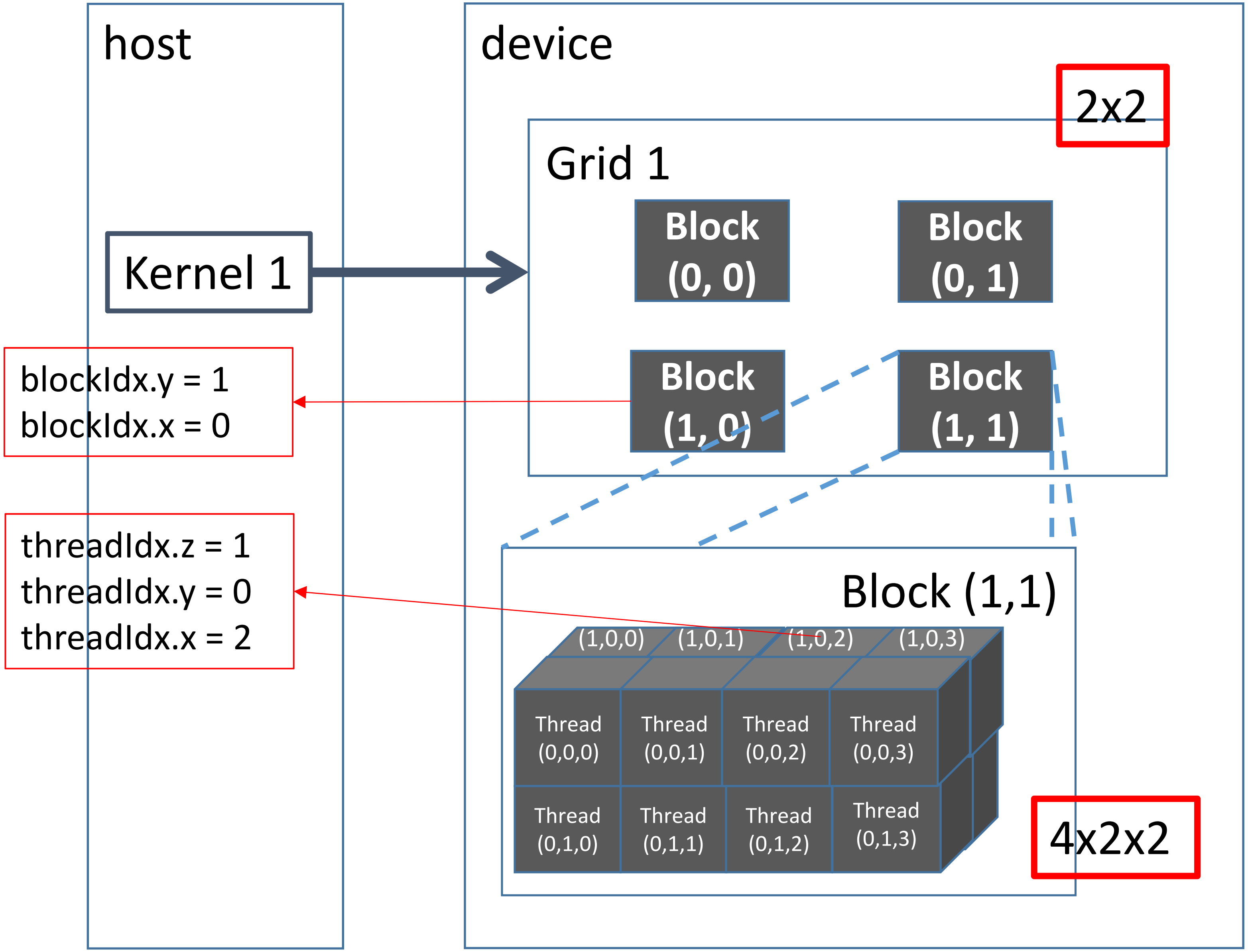
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

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

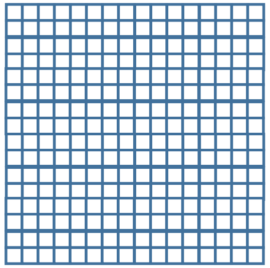
Content

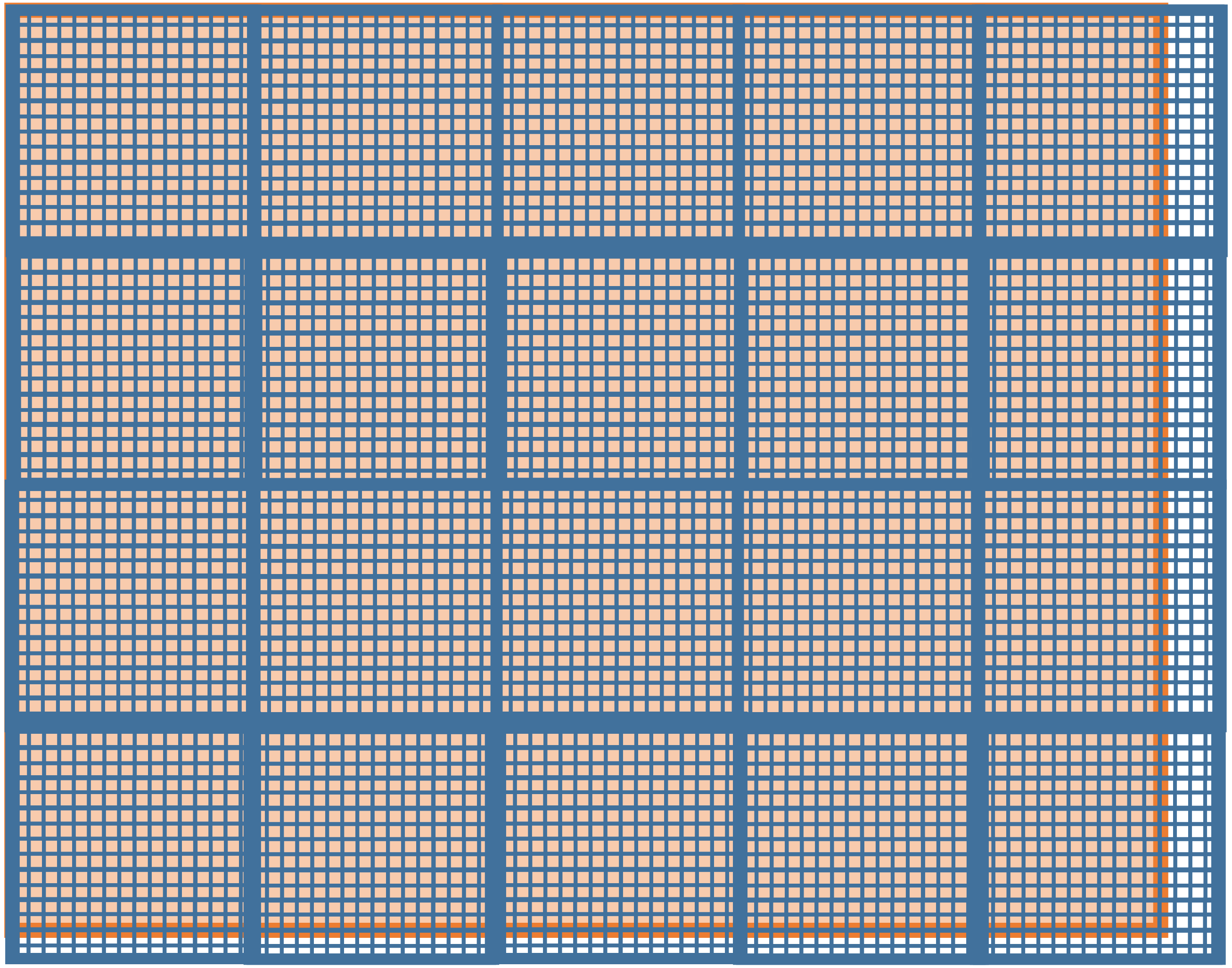
- Mapping Threads to Multi-dimensional Data

A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid

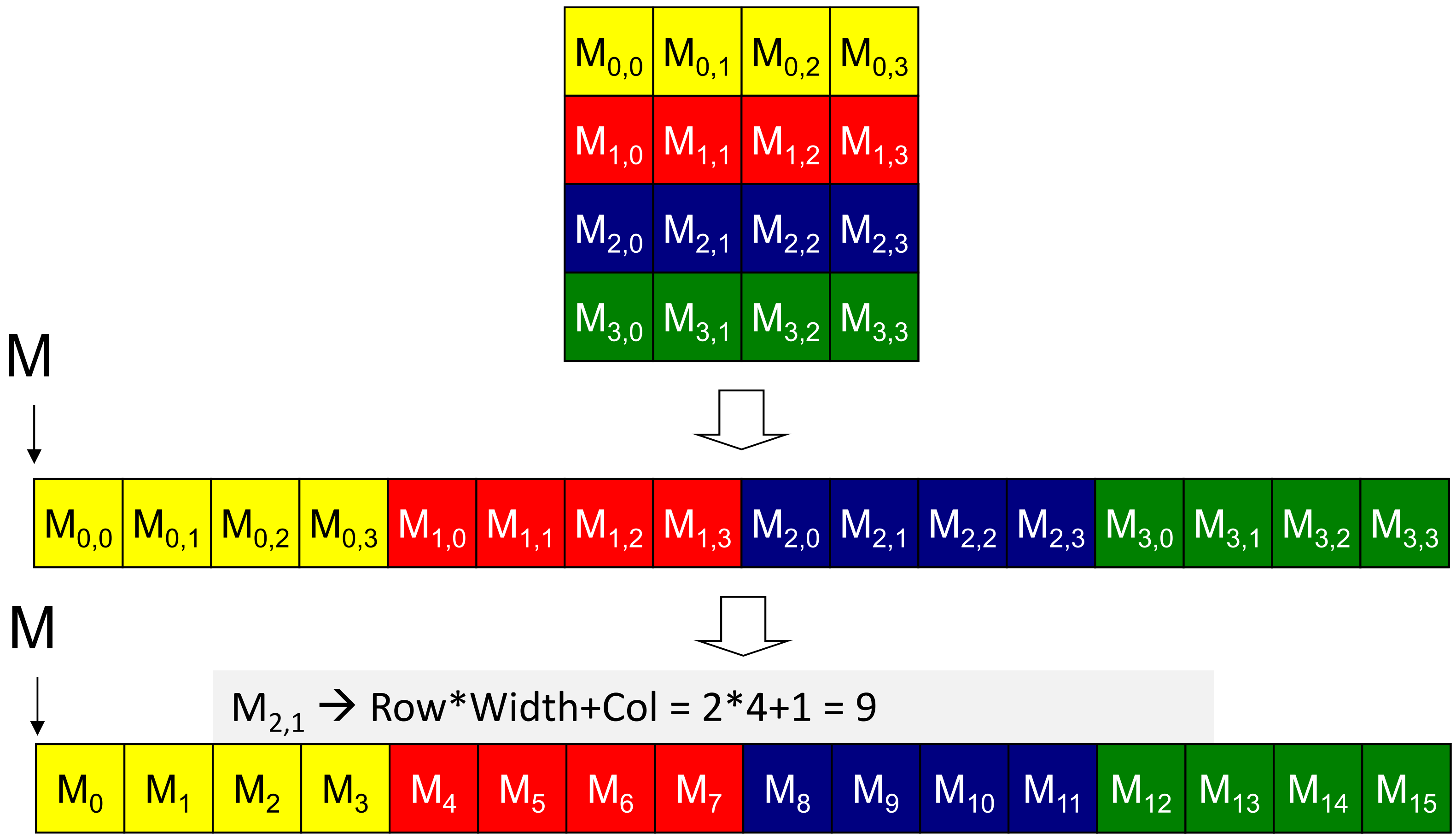




dimGrid = ?
dimBlock= ?

76x62 picture

Row-Major Layout of 2D arrays in C/C++



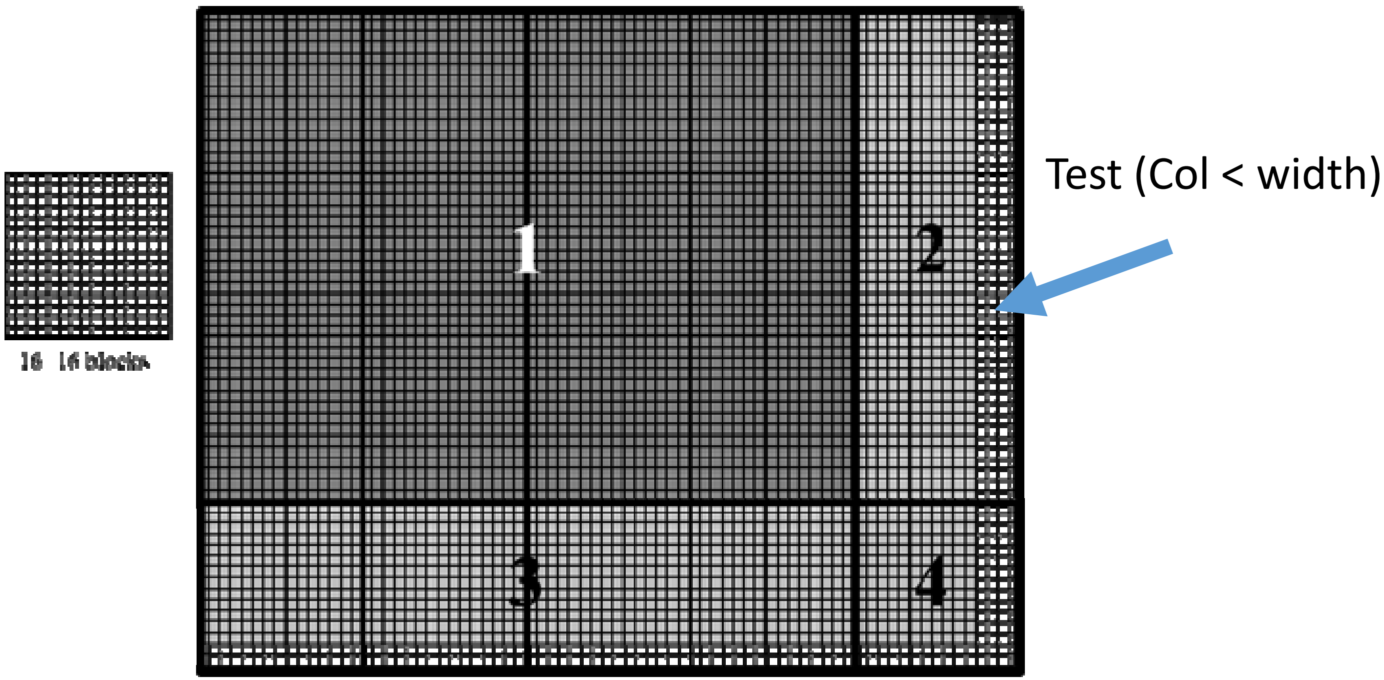
colorToGreyscaleConversion Kernel with 2D thread mapping to data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char * Pout,  unsigned char * Pin,
                int width, int height) {

 int Col =   threadIdx.x + blockIdx.x * blockDim.x;
 int Row = threadIdx.y + blockIdx.y * blockDim.y;

 if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns of the gray scale image
   int rgbOffset = greyOffset*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
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

Covering a 76×62 picture with 16×16 blocks

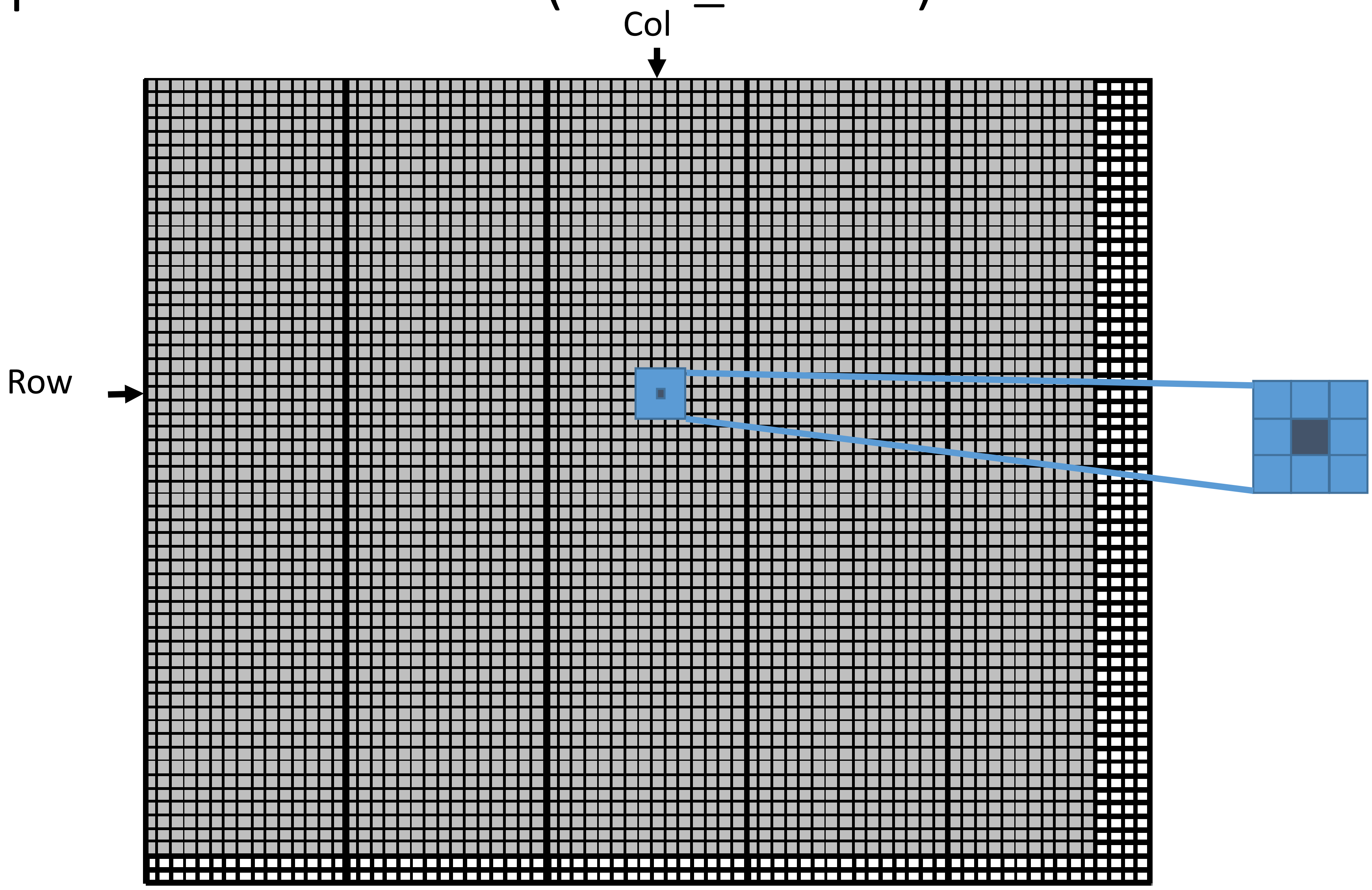


if (Col < width && Row < height)

Content

- Image Blur: A More Complex Kernel
-

Each output pixel is the average of pixels around it (BLRU_SIZE = 1)



An Image Blur Kernel

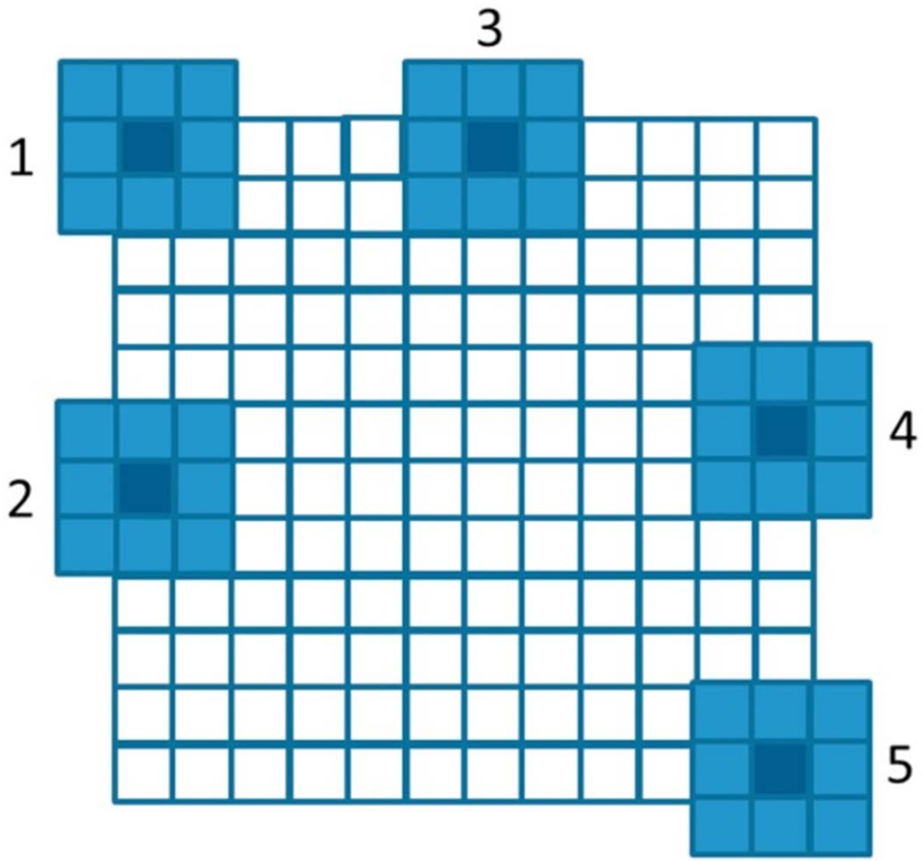
```
__global__
 void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
   int Col  = blockIdx.x * blockDim.x + threadIdx.x;
   int Row  = blockIdx.y * blockDim.y + threadIdx.y;

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

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

5.         int curRow = Row + blurRow;
6.         int curCol = Col + blurCol;
         // Verify we have a valid image pixel
7.         if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.           pixVal += in[curRow * w + curCol];
9.           pixels++; // Keep track of number of pixels in the avg
           }
         }
       }
     // Write our new pixel value out
10   out[Row * w + Col] = (unsigned char)(pixVal / pixels);
   }
 }
```

Handling boundary conditions for pixels near the edges of the image



Content

- Synchronization and Transparent Scalability

Barrier Synchronization

- An API function call in CUDA
  - __syncthreads()

- All threads in the same block must reach the __syncthreads() before any can move on

- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed

Synchronization

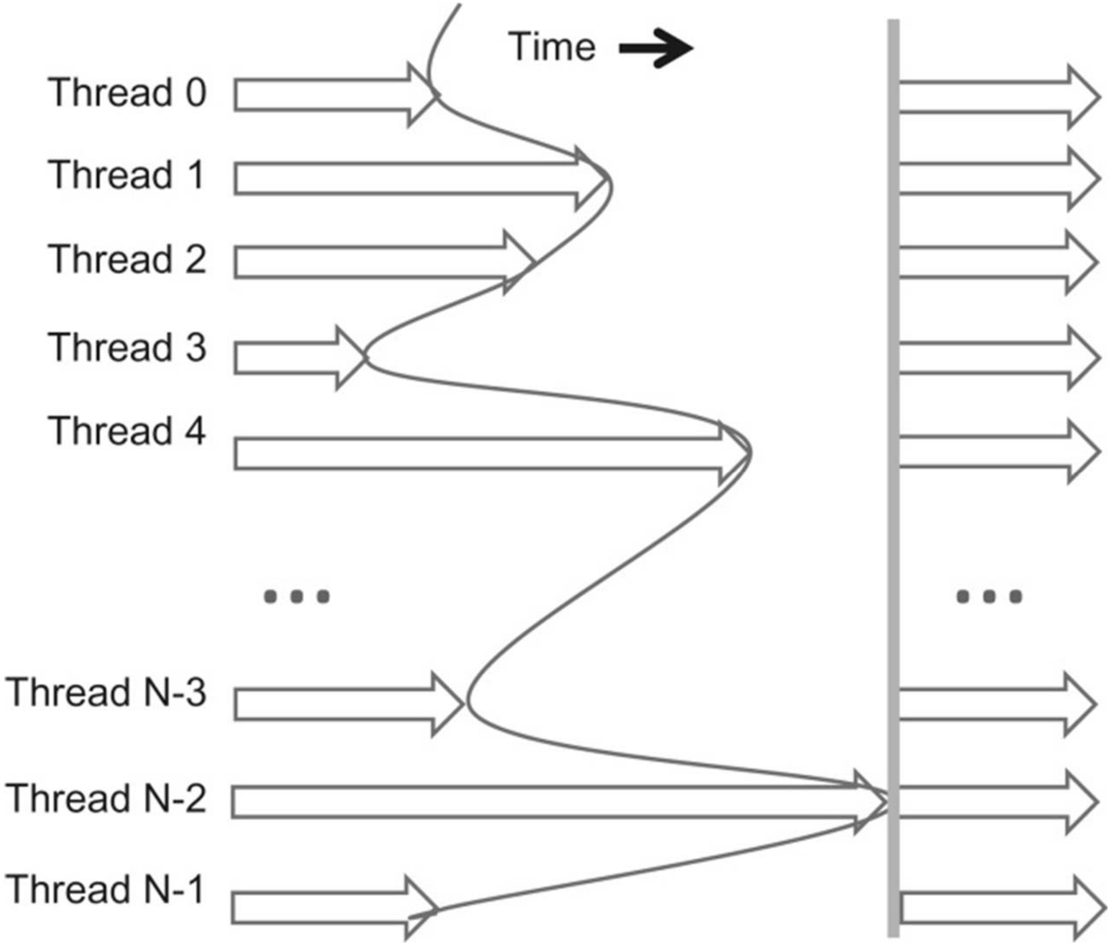


FIGURE 3.10: An example execution timing of barrier synchronization.

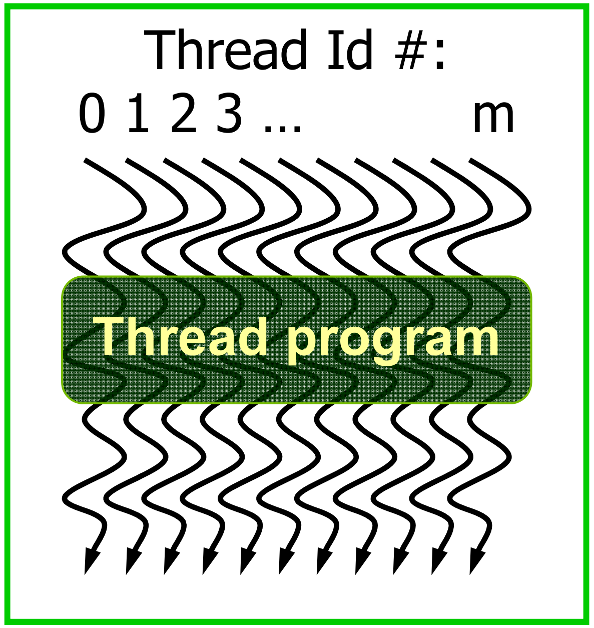
```
If
    ....
    __syncthreads()

else
    ....
    __syncthreads()
```

CUDA Thread Block (review)

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **1024** concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Transparent Scalability

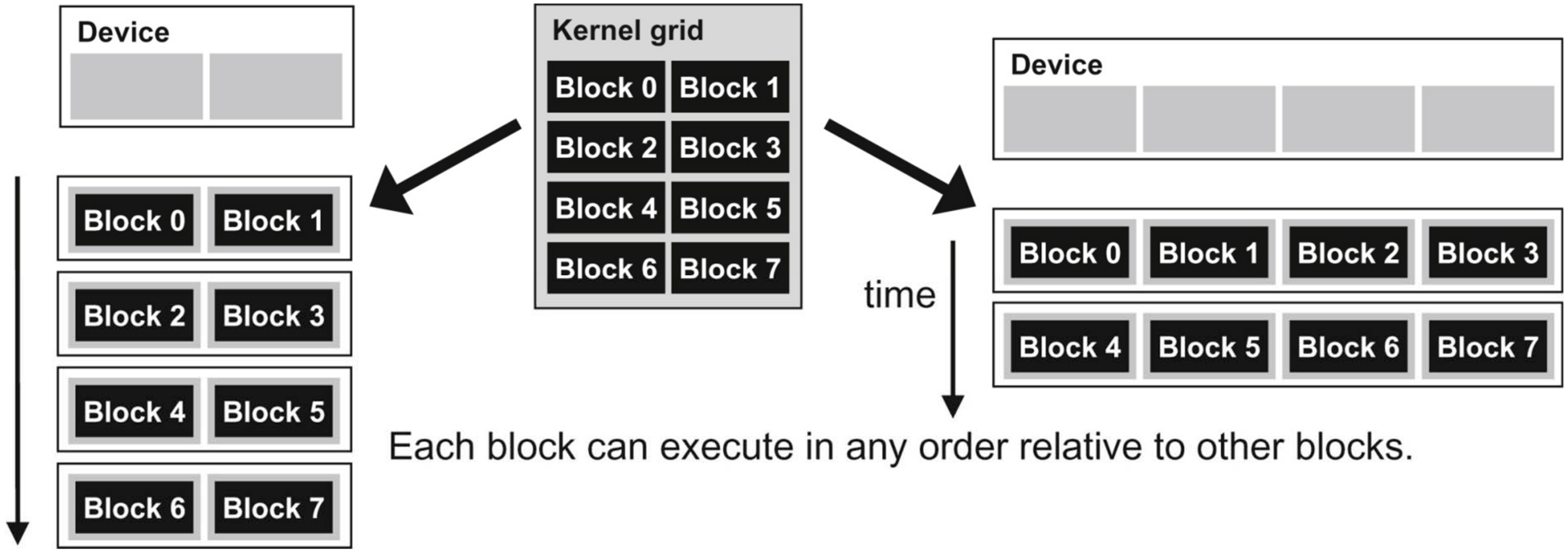
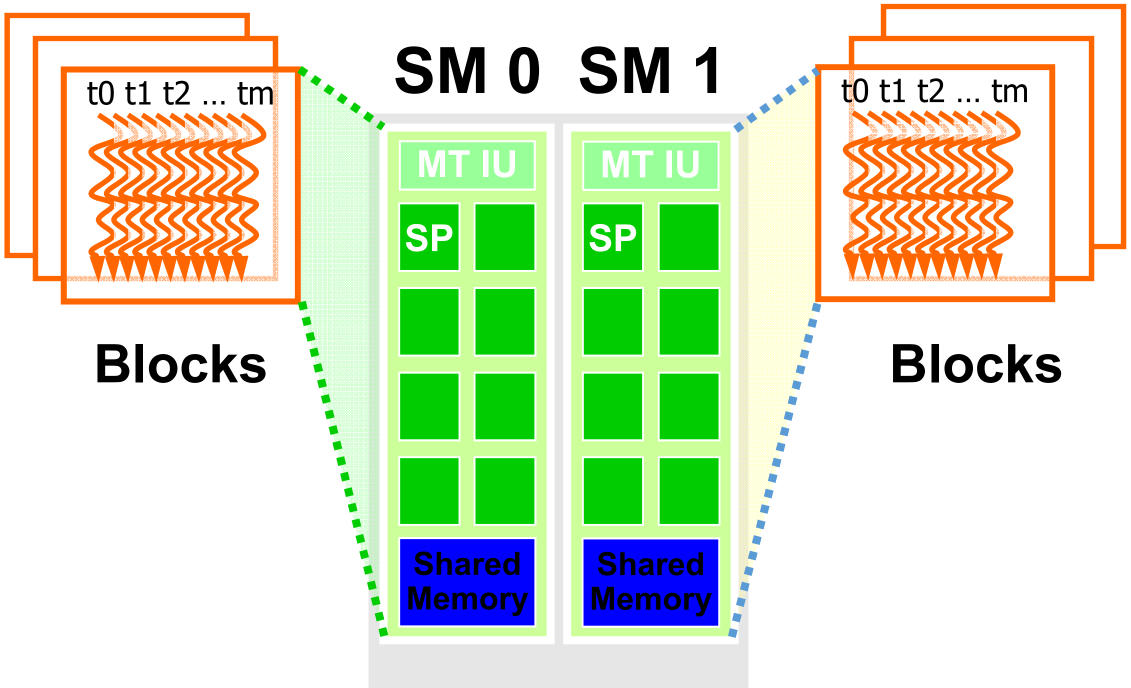


FIGURE 3.11: Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

The ability to execute the same application code on hardware with different numbers of execution resources is referred to as **transparent scalability** .

Executing Thread Blocks



- Limitations:
  - Number of Streaming Multiprocessors
  - Number of Blocks in a SM
  - Number of Threads in a SM

Compute Capabilities are GPU Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/s	86.4 GB/s
L2 Cache Size	256 KB	2048 KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm^2	148 mm^2
Manufactoring Process	28 nm	28 nm

Querying Device Properties

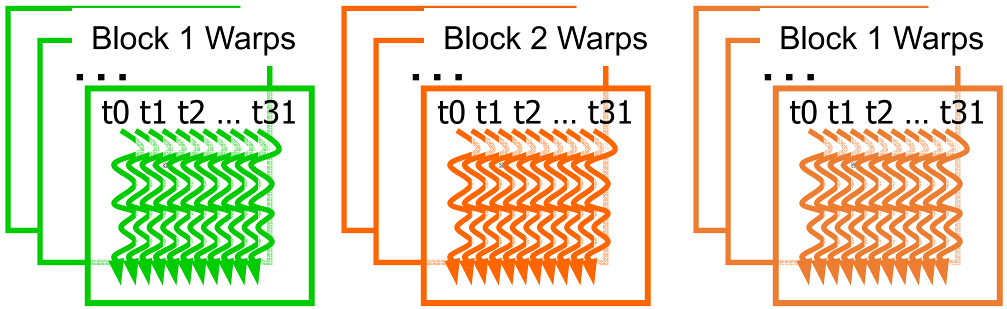
- How do find amount of resources available?
  - Number of SM
  - Number of Blocks in a SM
  - Number of Threads in a SM

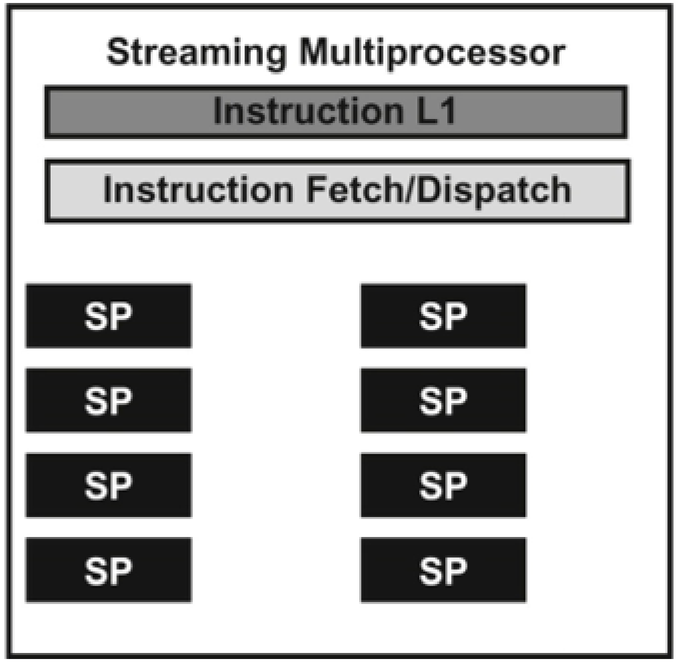
- CUDA Runtime API
  - cudaDeviceProp dev_prop;
  - cudaGetDeviceProperties(&dev_prop,i)

  - dev_prop.maxThreadsPerBlock
  - dev_prop.multiProcessorCount
  - ……
  - dev_prop.warpSize

Thread Scheduling (1/2)

- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- 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
  - 8 warps/blk * 3 blks = 24 warps





Thread Scheduling (2/2)

- 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 on a prioritized scheduling policy
  - **All threads in a warp execute the same instruction when selected**
- The hardware Streaming Processors actually execute instructions:
  - Can only execute small subset of warps