Threads, Blocks & Warps

In This Lecture

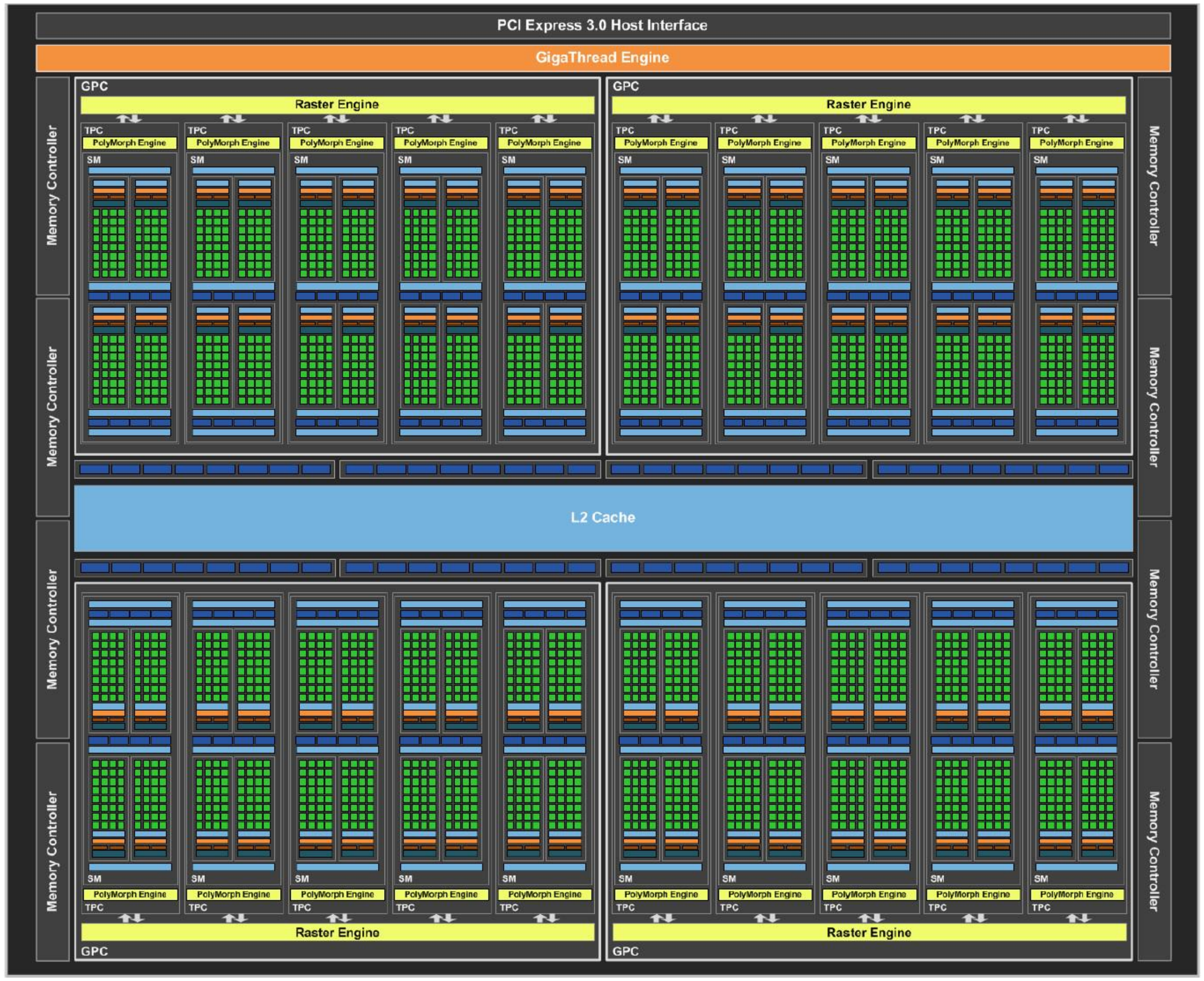
- Thread and Block
- Indexing Block & Thread
- Example: Matrix Summation/Multiplication
- Warp & Thread Divergence

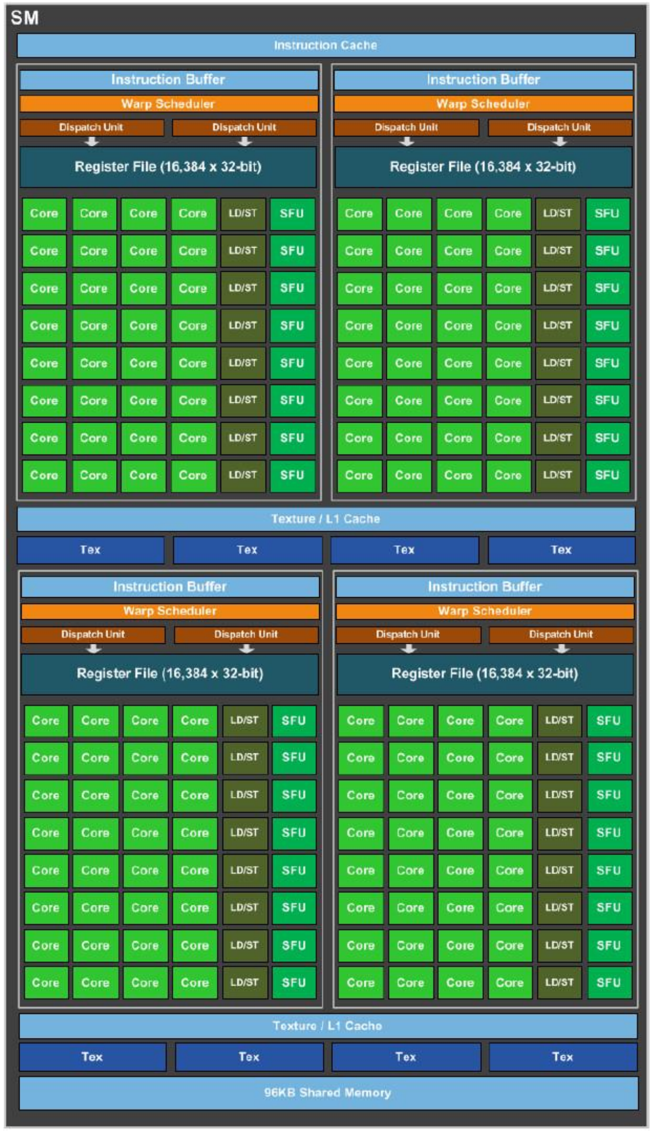
Review: Vector Addition

```
#define N 256
#include <stdio.h>
__global__ void vecAdd (int *a, int *b, int *c);
int main() {
  int a[N], b[N], c[N];
   int *dev_a, *dev_b, *dev_c;

  // initialize a and b with real values (NOT SHOWN)

  size = N * sizeof(int);
  cudaMalloc((void**)&dev_a, size);
  cudaMalloc((void**)&dev_b, size);
  cudaMalloc((void**)&dev_c, size);
  cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

  vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);

  cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);
  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);
  // Print Result
  exit(0);
}
__global__ void vecAdd (int *a, int *b, int *c) {
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}
```

- CUDA gives each thread a unique ThreadID to distinguish between each other even though the kernel instructions are the same.
- In our example, in the kernel call the memory arguments specify 1 block and N threads.

NVIDIA GP104 Architecture (GTX 1080)

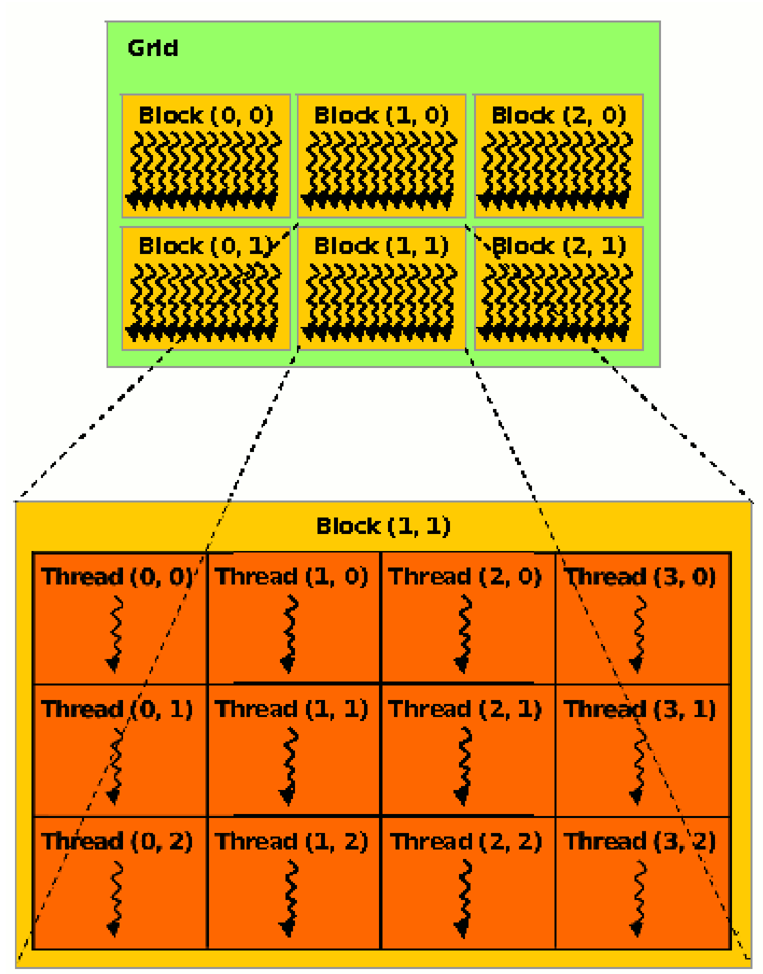




20 SMs, each has 128 CUDA cores (SP)

NVIDIA GPU Hierarchy

- Grids map to GPUs
- Blocks map to the Stream Multiprocessors (SM)
- Threads map to Stream Processors (SP)
- Warps are groups of 32 threads that execute simultaneously (back to this later)



Defining Grid/Block Structure

- Need to provide each kernel call with values for two key structures:
  - Number of blocks in each dimension
  - Number of threads per block in each dimension

- `myKernel<<< B,T >>>(arg1, … );`

- B – a structure that defines the number of blocks in grid in each dimension (1D or 2D or 3D).
- T – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).

1D Grids and/or 1D Blocks

- If want a 1-D structure, can use integers for B and T in:

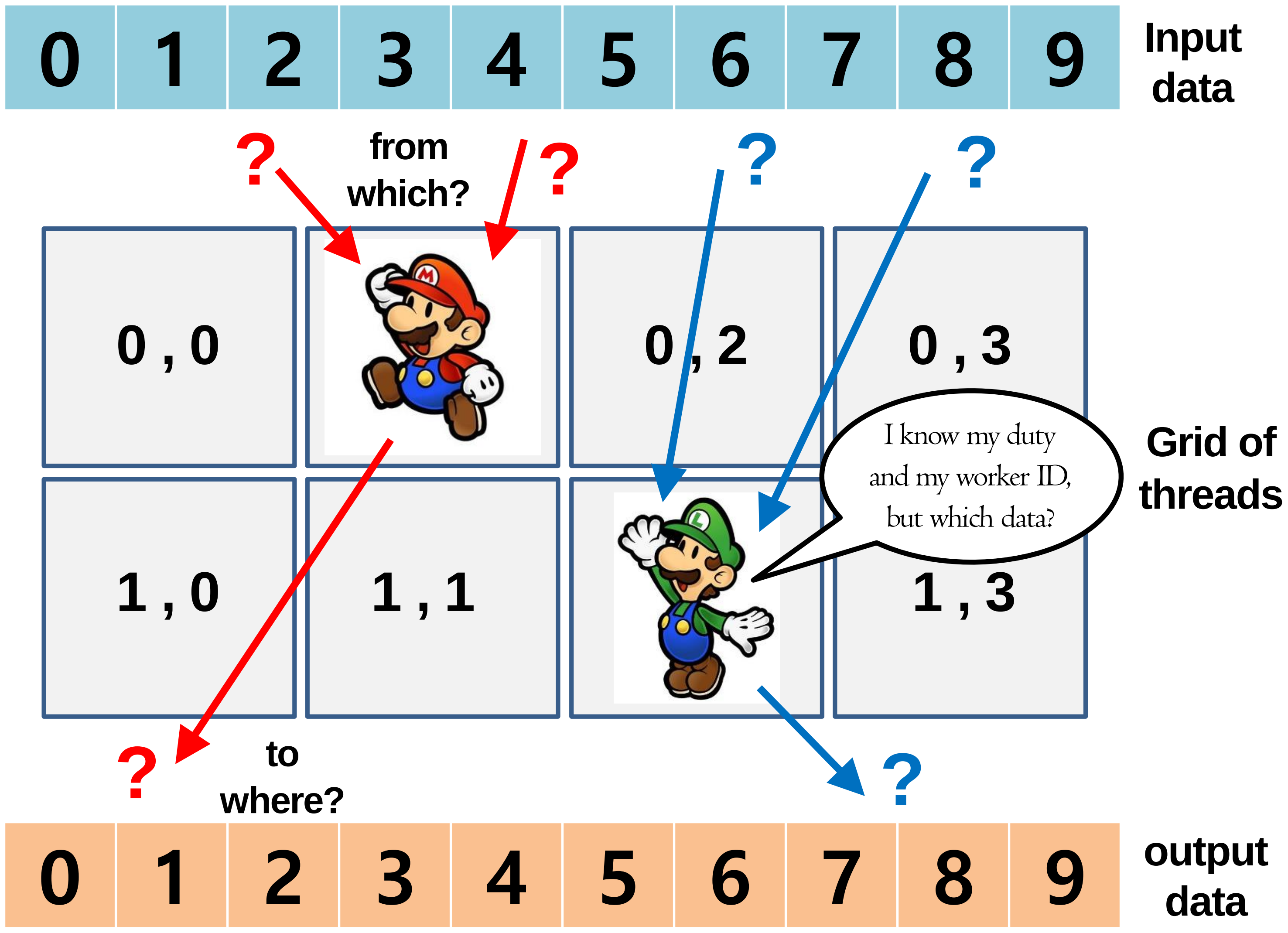
  ```
  myKernel<<< B, T >>>(arg1, … );
  ```

- B – An integer would define a 1D grid of blocks of that size
- T – An integer would define a 1D block of threads of that size

- Example:

  ```
  myKernel<<< 10, 100 >>>(arg1, ... );
  ```

Why Thread ID Matters ?



CUDA Built-In Variables

- **blockIdx.x**, **blockIdx.y**, **blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.

- **threadIdx.x**, **threadIdx.y**, **threadIdx.z** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.

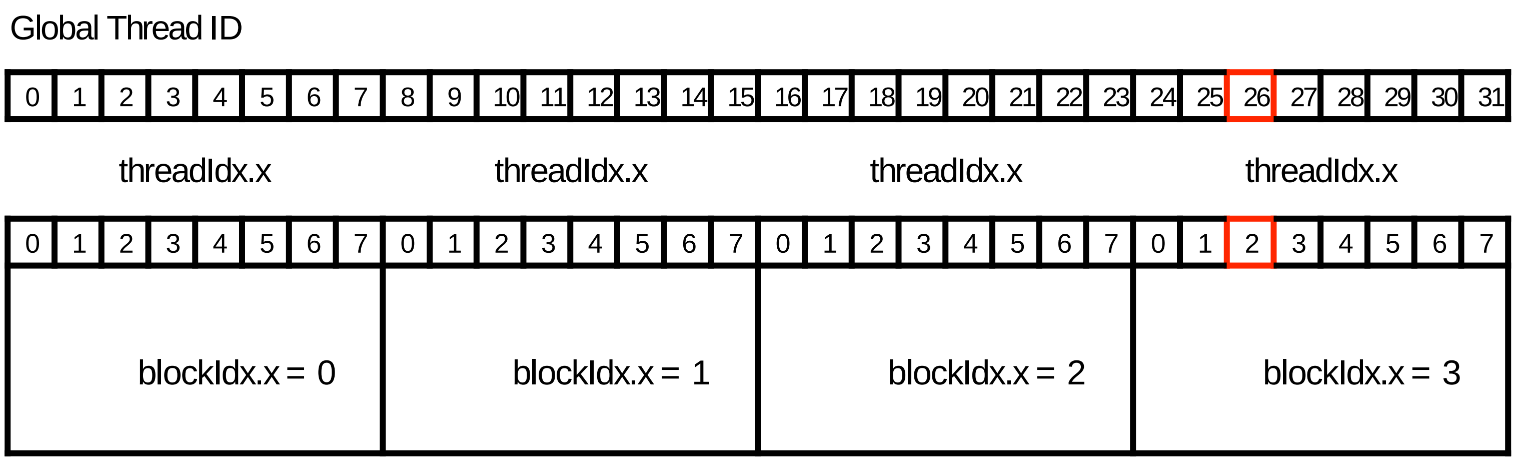
- **blockDim.x**, **blockDim.y**, **blockDim.z** are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

- So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.
- These can be helpful when thinking of your data as 2D or 3D.

- The full global thread ID in x dimension can be computed by:

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

Thread Identification Example: x-direction

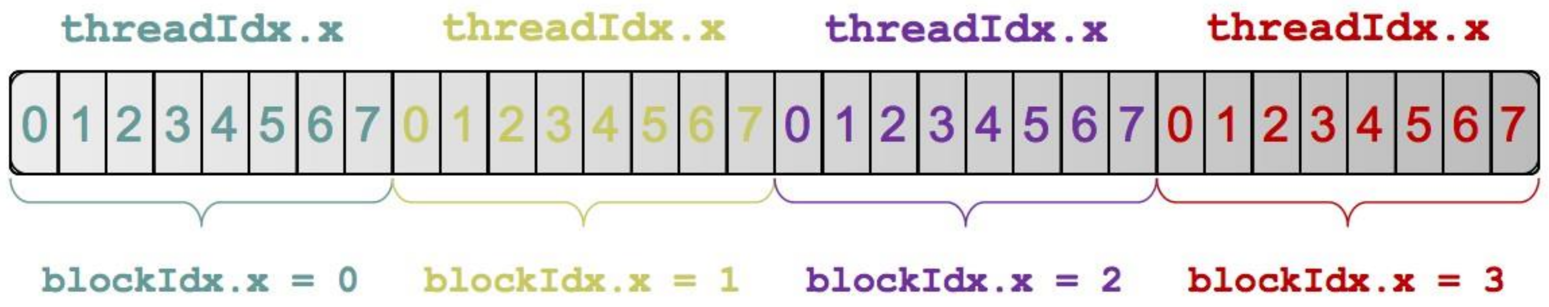


- Assume a hypothetical 1D grid and 1D block architecture:
  4 blocks, each with 8 threads.

- For Global Thread ID 26:
  - gridDim.x = 4 x1
  - blockDim.x = 8 x1
  - Global Thread ID = blockIdx.x * blockDim.x + threadIdx.x = 3 x 8 + 2 = 26

Array indexing: example

- **Consider indexing into an array, one thread accessing one element**

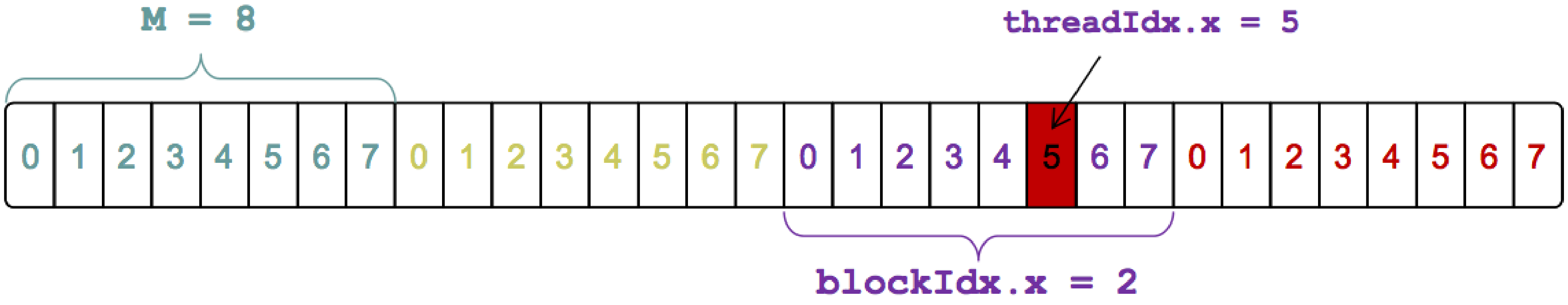


  - Assume you launch with M=8 threads per block and the array is 32 entries long

  - With M threads per block a unique index for each thread is given by:
    - `int index = threadIdx.x + blockIdx.x * M;`
    - Where M is the size of the block of threads; i.e., blockDim.x

Array indexing: example

- What is the array entry on which the thread with index 5 in block of index 2 will work ?

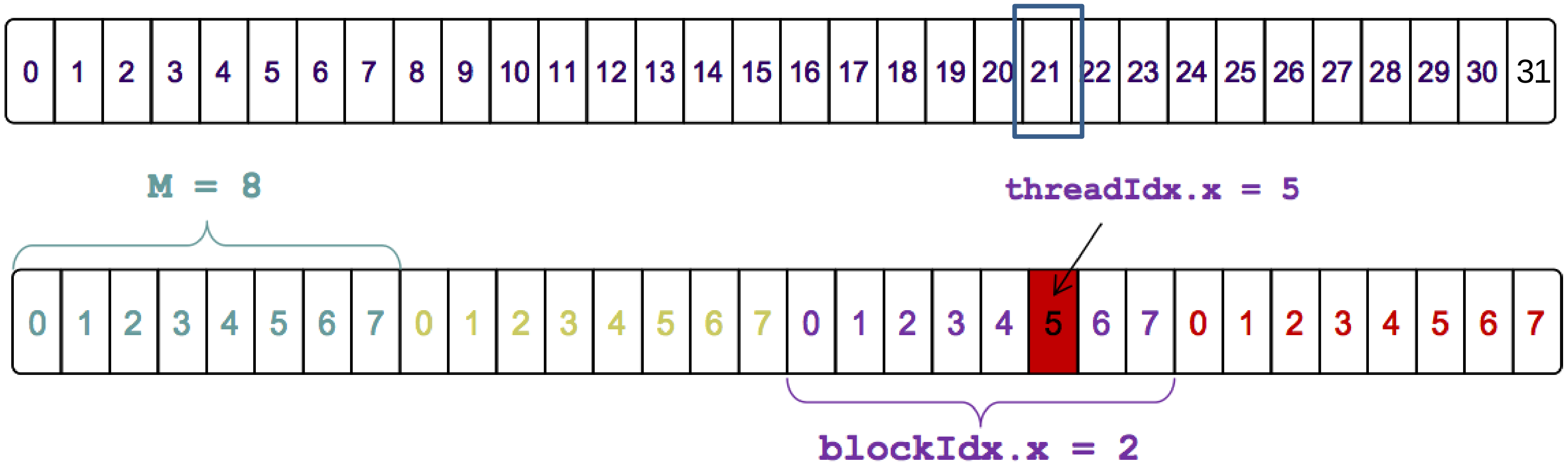


Array indexing: example

- **What is the array entry on which the thread with index 5 in block of index 2 will work ?**

- `int index  = threadIdx.x + blockIdx.x * blockDim.x;`

```
            =       5       +       2    *      8;
            =      21;
```



Vector Addition Revisited

```
#define N 1618
#define T 1024 // max threads per block
#include <stdio.h>
__global__ void vecAdd (int *a, int *b, int *c);
int main() {
  int a[N], b[N], c[N];
  int *dev_a, *dev_b, *dev_c;
  // initialize a and b with real values (NOT SHOWN)
  size = N * sizeof(int);
  cudaMalloc((void**)&dev_a, size);
  cudaMalloc((void**)&dev_b, size);
  cudaMalloc((void**)&dev_c, size);
  cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);
  vecAdd<<<(int)ceil((float)N/T),T>>>(dev_a,dev_b,dev_c);
  cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);
  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);
  exit(0);
}
__global__ void vecAdd (int *a, int *b, int *c) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i < N){
    c[i] = a[i] + b[i];
  }
}
```

- Since the maximum number of threads per dimension in a block is **1024**, if you must use more than one block to access more threads.
- Divide the work between different blocks.
- Notice that each block is reserved completely; in this example, two blocks are reserved even though most of the second block is not utilized.
- WARNING: CUDA does not issue warnings or errors if your thread bookkeeping is incorrect.

Higher Dimensional Grids/Blocks

- 1D grids/blocks are suitable for 1D data, but higher dimensional grids/blocks are necessary for:
  - higher dimensional data.
  - data set larger than the hardware dimensional limitations of blocks.

- CUDA has built-in variables and structures to define the number of blocks in a grid in each dimension and the number of threads in a block in each dimension.

CUDA Built-In Vector Types and Structures

- uint3 and dim3 are CUDA-defined structures of unsigned integers: x, y, and z.
  - struct uint3 {x; y; z;};   for indexing
  - struct dim3 {x; y; z;};   for defining dimension

- The unsigned structure components are automatically initialized to 1.

- These vector types are mostly used to define grid of blocks and threads.

CUDA Built-In Variables for Grid/Block Sizes

- `dim3 gridDim` -- Grid dimensions, x, y and z

- Number of blocks in grid =

  ```
  gridDim.x * gridDim.y * gridDim.z
  ```

- `dim3 blockDim` -- Size of block dimensions x, y, and z.

- Number of threads in a block =

  ```
  blockDim.x * blockDim.y * blockDim.z
  ```

Initializing Values

- To set dimensions:

```
dim3 grid(16,16);     // grid = 16 x 16 blocks
dim3 block(32,32);    // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:

```
gridDim.x  = 16;
gridDim.y  = 16;
gridDim.z  = 1;

blockDim.x = 32;
blockDim.y = 32;
blockDim.z = 1;
```

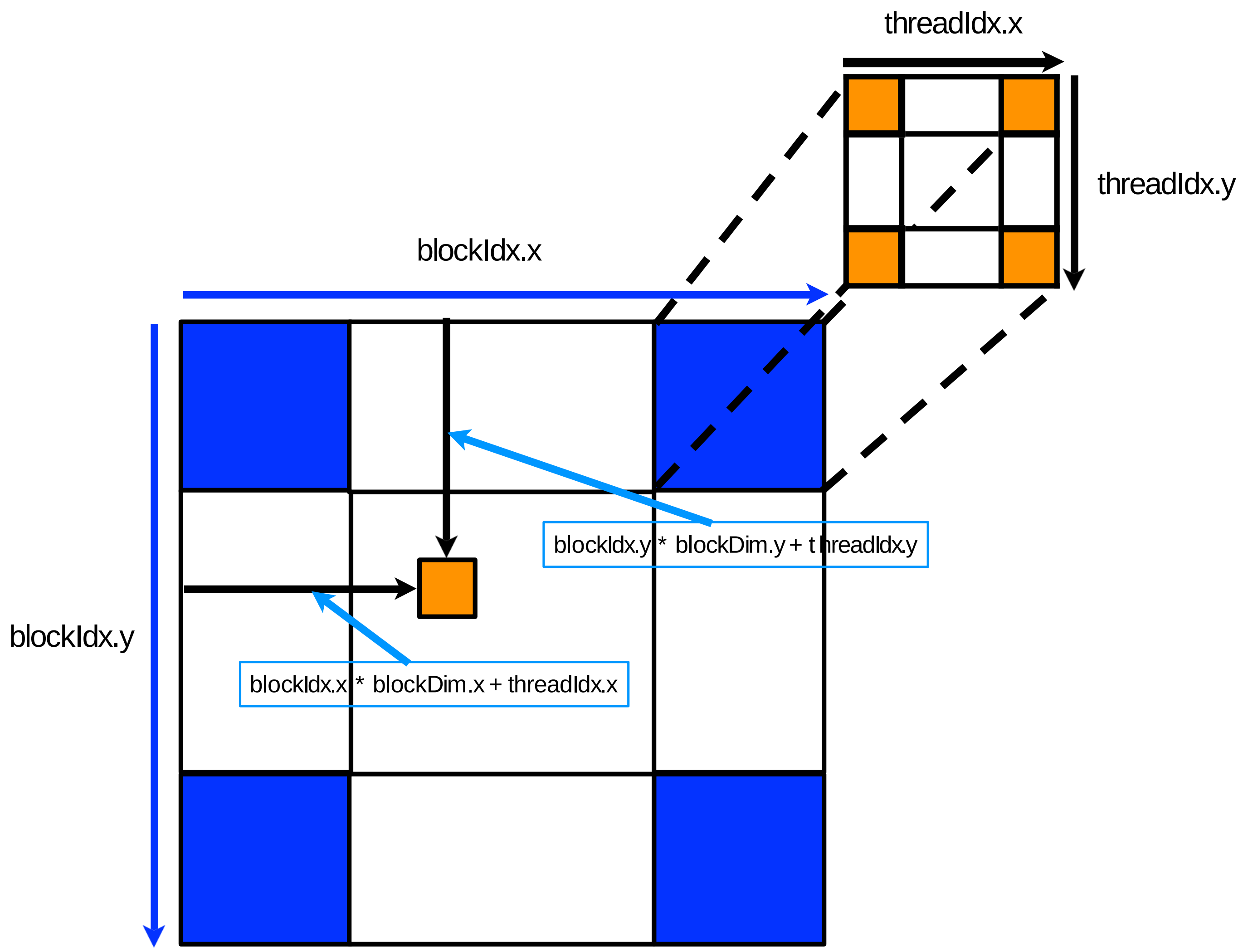
CUDA Built-In Variables for Grid/Block Indices

- uint3 blockIdx -- block index within a grid:
  - blockIdx.x, blockIdx.y, blockIdx.z
- uint3 threadIdx -- thread index within a block:
  - threadIdx.x, threadIdx.y, threadIdx.z

- Full global thread ID in x, y and z dimensions can be computed by:

```
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;
z = blockIdx.z * blockDim.z + threadIdx.z;
```

2D Global ThreadID in 2D Grids and 2D Blocks

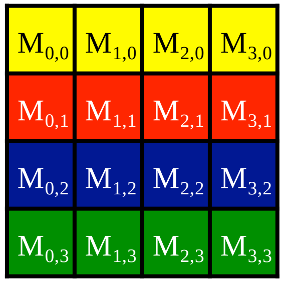

threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.x * blockDim.x + threadIdx.x

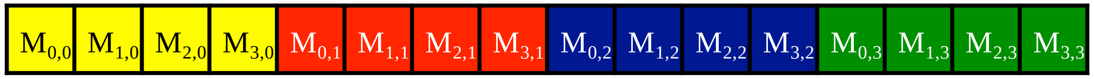
Flatten Matrices into Linear Memory

- Generally memory cannot be allocated dynamically on device (GPU) and we cannot not use two-dimensional indices (e.g.A[row][column]) to access matrices (when data is transferred between host & device).

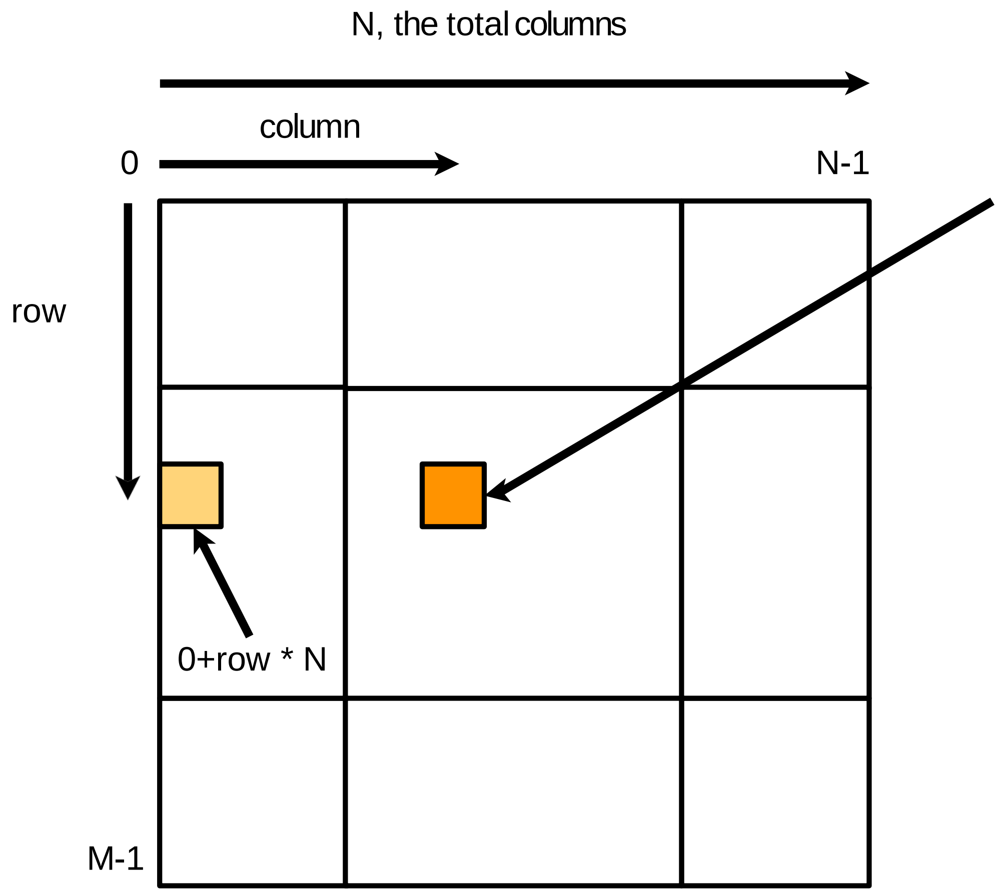
- We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.

- C uses **`row-major`** order --- rows are stored one after the other in memory, i.e. row 0 then row 1 etc.


$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$


$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Accessing Matrices in Linear Memory



```
a[row][col]  = A[offset]

offset = col + row * N
```

In CUDA:

```
int col  =  blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row *  N;
A[index] = …
```

† what if 3D matrix?

```
a[row][col][dep]  = A[offset]
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int dep = blockIdx.z*blockDim.z+threadIdx.z;
offset = col + row*N+depth*N*M
```

Matrix Addition: Add two 2D matrices

- Corresponding elements of each array (a and b) added together to fill the element of third array (c):

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \le i < n,\ 0 \le j < m)$$

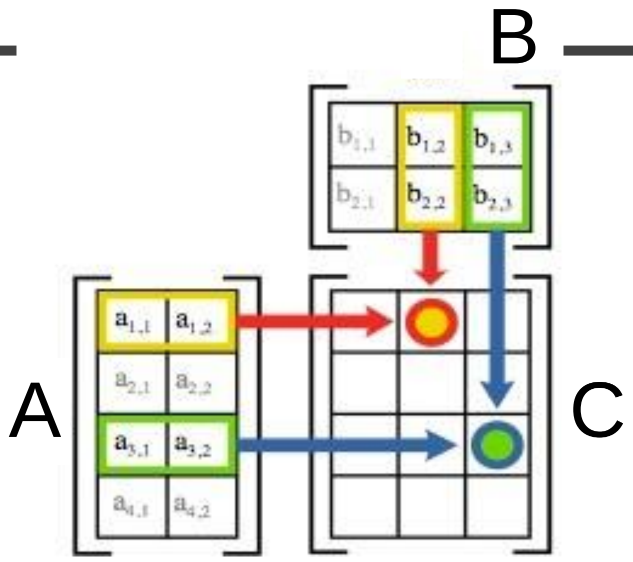
Example: Matrix Summation

```
#define N 100
#define BLOCK_DIM 10
__global__ void matrixAdd (int *a, int *b, int *c);
int main() {
  int *a, *b, *c;
  int *dev_a, *dev_b, *dev_c;
  int size = N * N * sizeof(int);
  // allocate & initialize a and b with input values (NOT SHOWN)
  cudaMalloc((void**)&dev_a, size);
  cudaMalloc((void**)&dev_b, size);
  cudaMalloc((void**)&dev_c, size);
  cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
  dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
  int gridSize = (int)ceil((float)N/BLOCK_DIM);
  dim3 dimGrid(gridSize , gridSize);
  matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);
  cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
  cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}
__global__ void matrixAdd (int *a, int *b, int *c) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int index = col + row * N;

 if (col < N && row < N) {
   c[index] = a[index] + b[index];
 }
}
```

- 2D matrices are added to form a sum 2D matrix.
- We use dim3 variables to set the Grid and Block dimensions.
- We calculate a global thread ID to index the column and row of the matrix.
- We calculate the linear index of the matrix.

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of tw o matrices A and B, we multipl y the rows of A by the column s of B and add them up.

- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

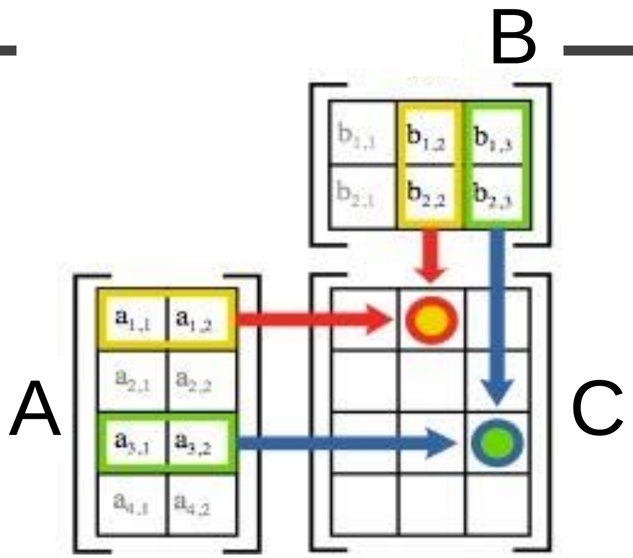
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.

- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

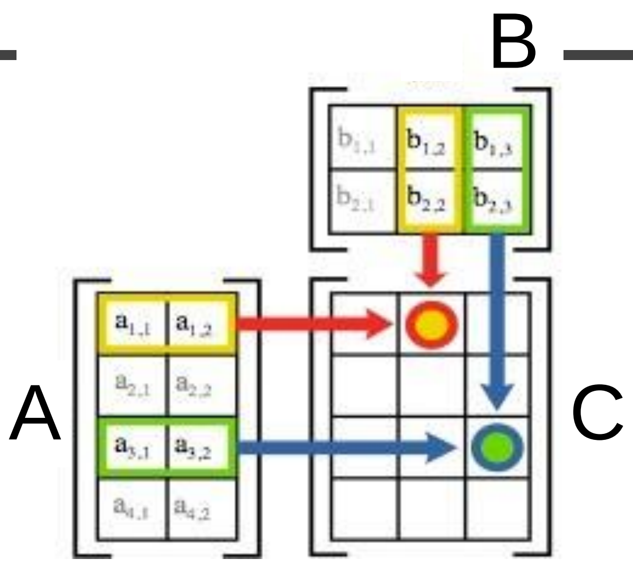
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.

- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

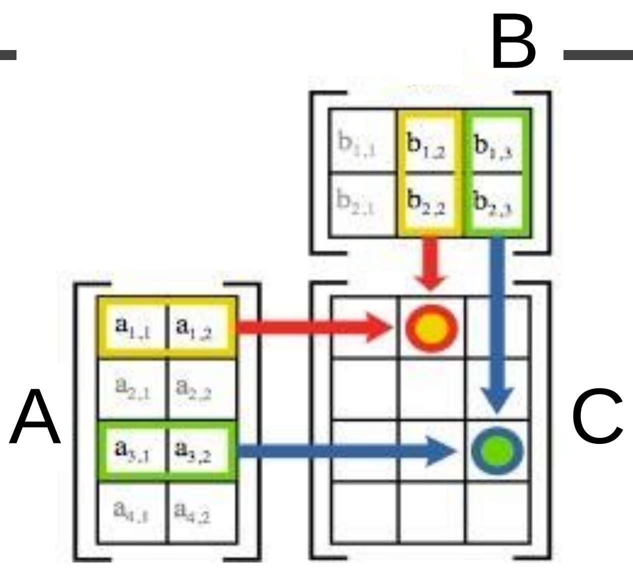
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

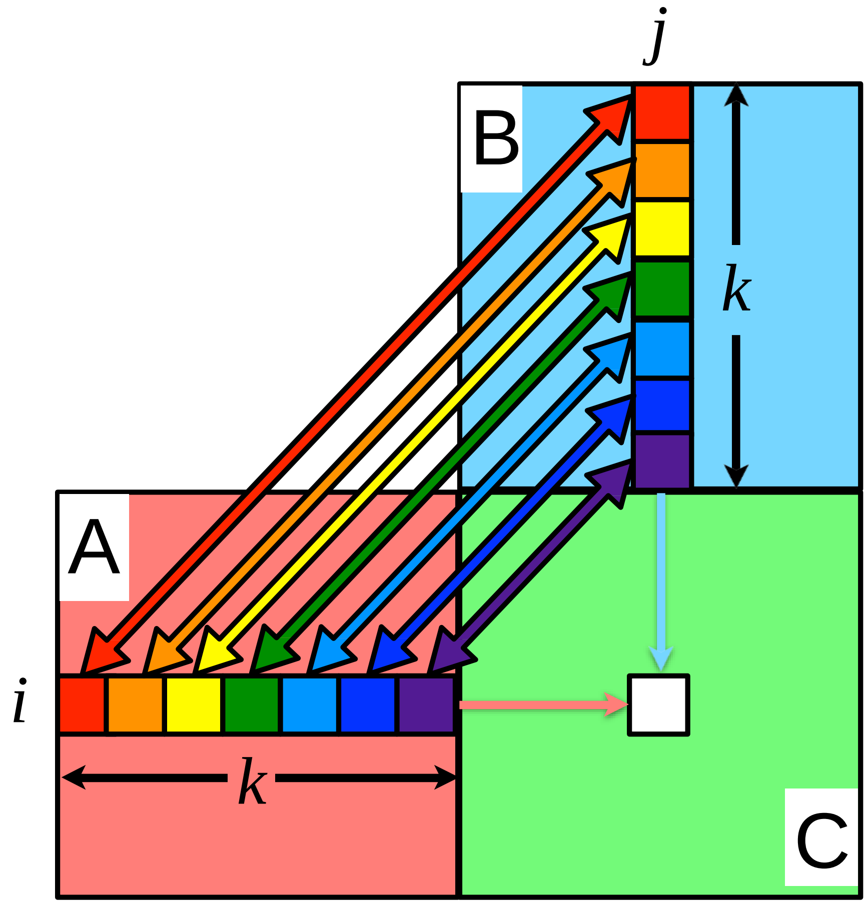
$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Square Matrix Multiplication C Code

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
       int sum = 0;
       for (int k = 0; k < width; k++) {
         int m = a[i][k];
         int n = b[k][j];
         sum += m * n;
       }
}
       c[i][j] = sum;
    }
  }
```

- Sequential algorithm consists of multiple nested <u>for loops</u>.
- Both multiplications and additions are in O(N^3).
- Can it be parallelized?

Parallel Matrix Multiplication



- To compute a single value of C(i,j), only a single thread be necessary to traverse the ith row of A and the jth column of B.

- Therefore, the number of threads needed to compute a square matrix multiply is O(N^2).

Practice: Matrix Multiplication

- **Copy Sample Skeleton Code**
  - cp –r /home/share/17_MatrixMulit ./[FolderName]
  - cd [FolderName]

- **Notepad: MatrixMulti.cu 코드 Kernel function 완성**

- **Compile & run program**
  - make
  - ./EXE

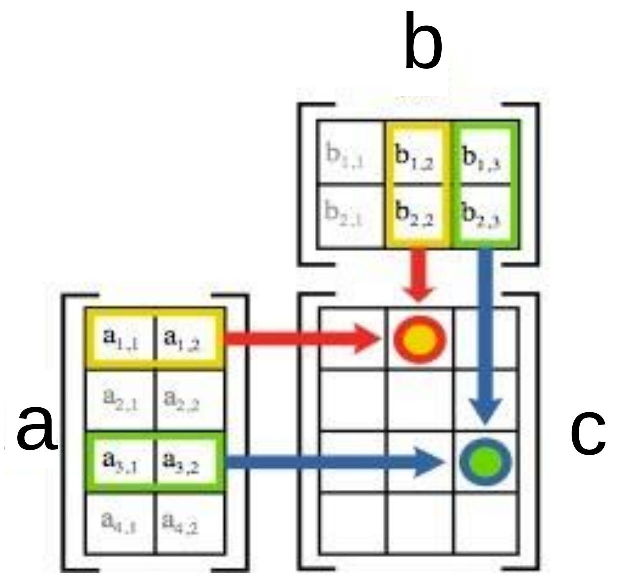
C to CUDA for Dummies Step 1: Write the Kernel

C Function

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;
      }
      c[i][j] = sum;
    }
  }
}
```

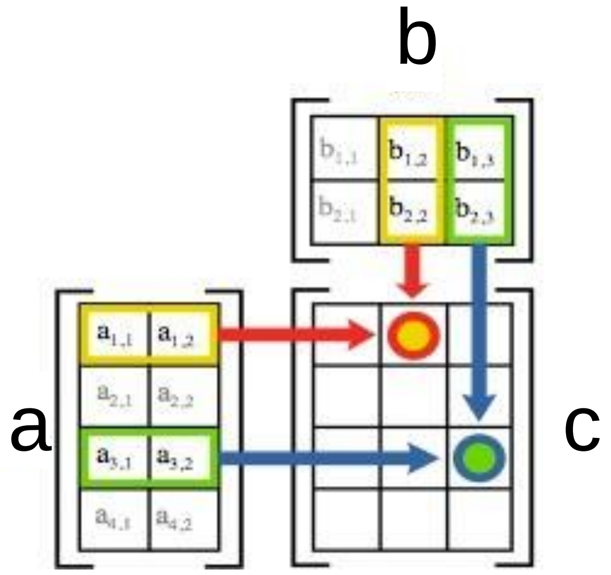
CUDA Kernel

```
__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
     sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```



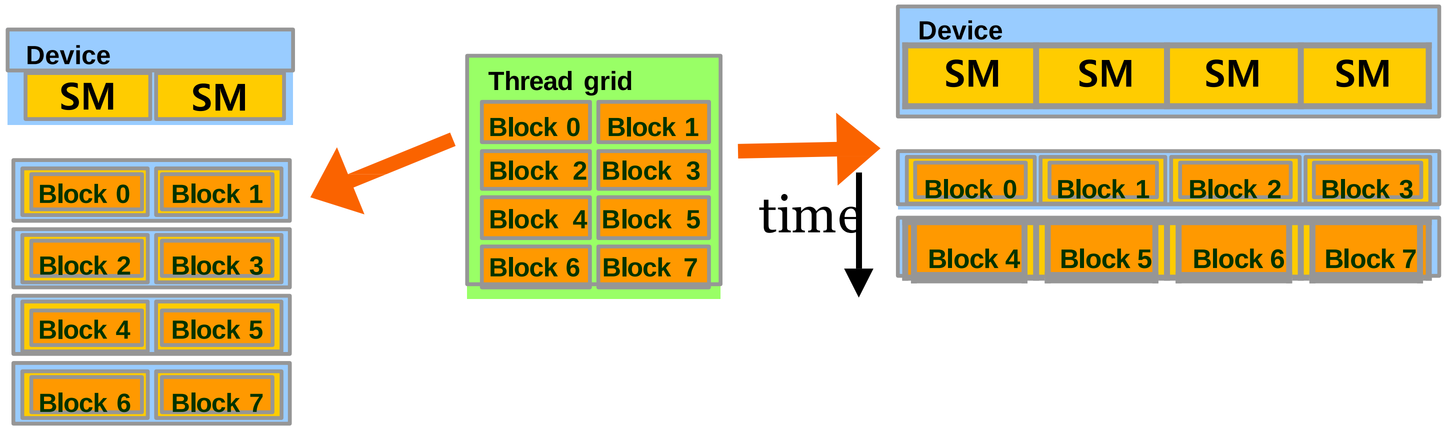
C to CUDA for Dummies Step 2: Do the Rest

```
#define N 16
__global__ void matrixMult (int *a, int *b, int *c, int width);
  int main() {
  int *a, *b, *c;
  int *dev_a, *dev_b, *dev_c;
  int size = N * N * sizeof(int);
  //allocate & initialize a to c with real values (NOT SHOWN)
  cudaMalloc((void **) &dev_a, size);
  cudaMalloc((void **) &dev_b, size);
  cudaMalloc((void **) &dev_c, size);
  cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
  dim3 dimGrid(1,1);
  dim3 dimBlock(N, N);
  matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);
  cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
  cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}
__global__ void matrixMult (int *a, int *b, int *c, int width) {
  int k, sum = 0;
  int col = threadIdx.x + blockDim.x * blockIdx.x;
  int row = threadIdx.y + blockDim.y * blockIdx.y;
  if(col < width && row < width) {
    for (k = 0; k < width; k++)
      sum += a[row * width + k] * b[k * width + col];
    c[row * width + col] = sum;
  }
}
```



Warps and Scheduling

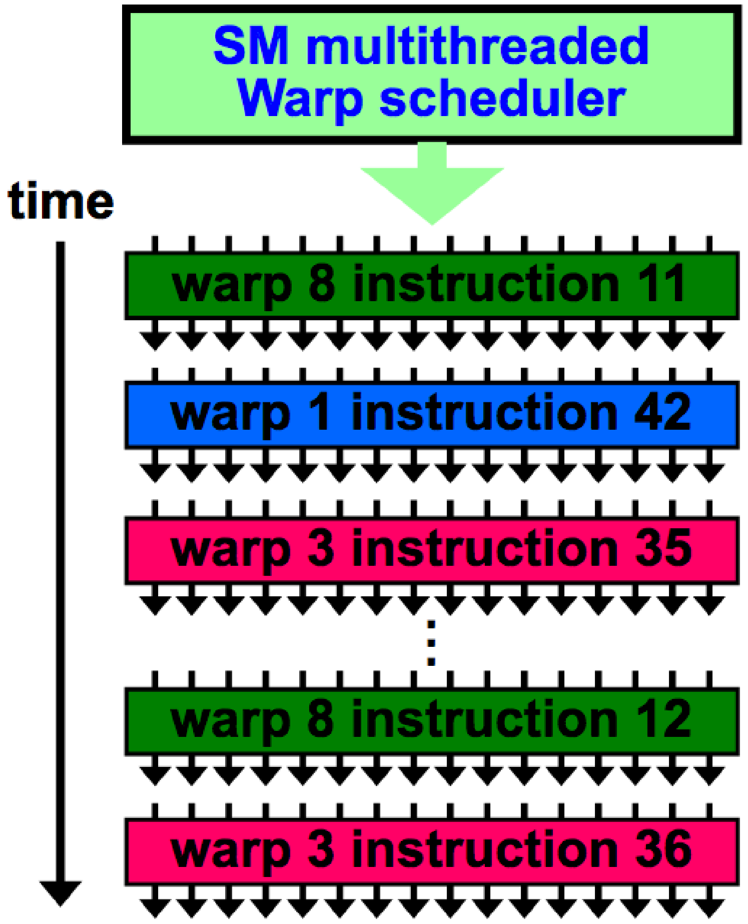
Transparent scalability



- **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

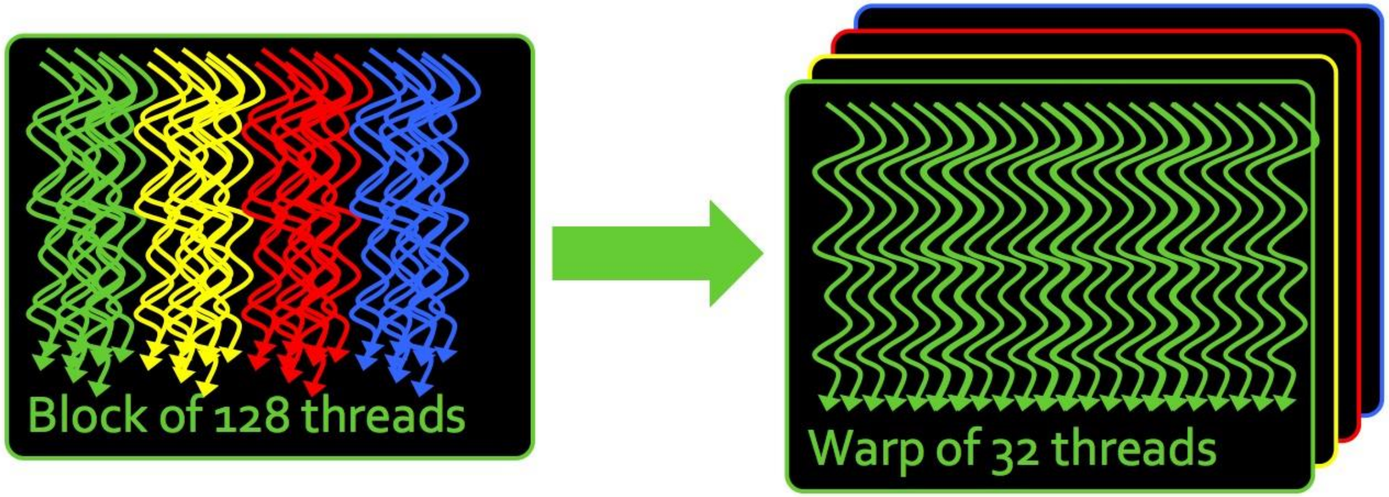
Warp scheduling

- A warp is a group of 32 threads
- 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



Threads and Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently



Threads and Warps

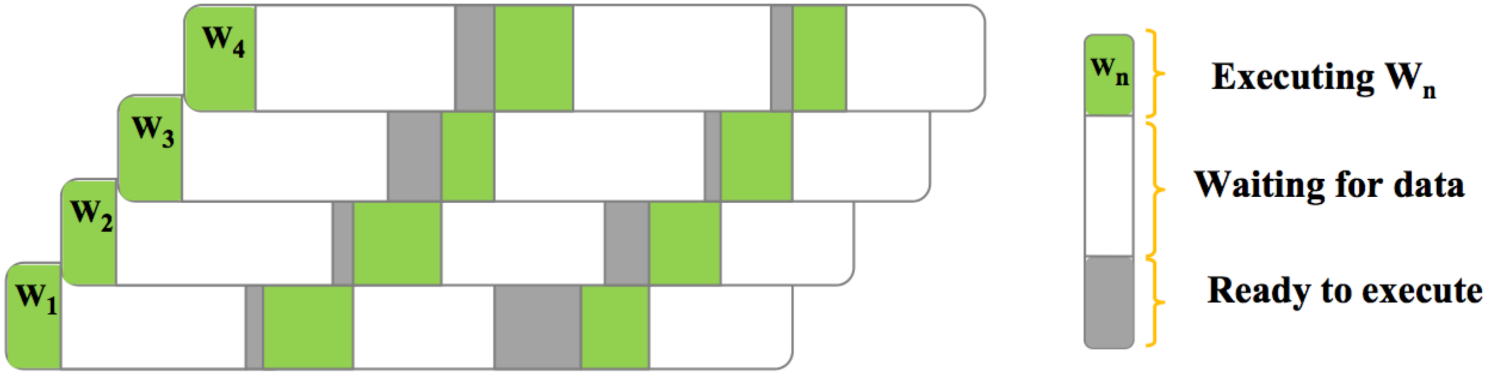
- **In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index**

- **Thread IDs within a warp are consecutive and increasing**
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.
  - Partitioning of threads in warps by SM is always the same
    - You can use this knowledge in control flow

Thread and Warp scheduling

- An SM can switch between warps with no apparent overhead
- Warps with instructions whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all active threads execute the same instruction in lockstep fashion

  (i.e. the exact same operation in parallel on different data)



Thread and Warp scheduling

- Prefer thread block sizes that result in mostly full warps:
  - **Bad**: `kernel<<< N,1 >>>( ... )`
  - **Okay**: `kernel<<<(N+31)/32, 32>>>( ... )`
  - **Better**: `kernel<<<(N+127)/128, 128>>>( ... )`
- Prefer to have enough threads per block to provide hardware with many warps to switch between (hides memory latency)
- When a thread block finishes, a new block is launched on the vacated SM

Thread Divergence

- **The lockstep execution of threads means that all threads must execute the same instruction at the same time. In other words, threads cannot diverge.**
- **The most common code construct that can cause thread divergence is branching for conditionals in an if-then-else statement.**

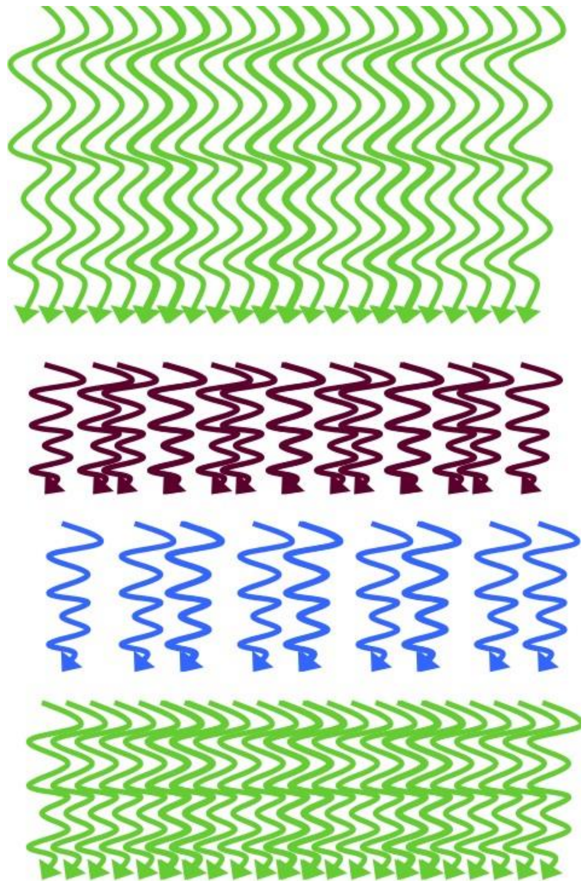
```
__global void odd_even(int n, int* x)
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
      if( i % 2 == 0 ) {
          x[i] = x[i] + 1;
      } else {
          x[i] = x[i] + 2;
      }
}
```

  - Half the threads (even i) in the warp execute the if clause, the other half (odd i) the else clause
  - The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel
  - Often, this requires that the hardware execute multiple paths through a kernel for a warp.

Divergence and execution

```
__global__ void kv(int* x, int* y)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
     int t;
     bool b = f(x[i]);

     if( b )
     {
         // g(x)
         t = g(x[i]);
     }
     else
     {
         // h(x)
         t = h(x[i]));
     }
     y[i] = t;
}
```

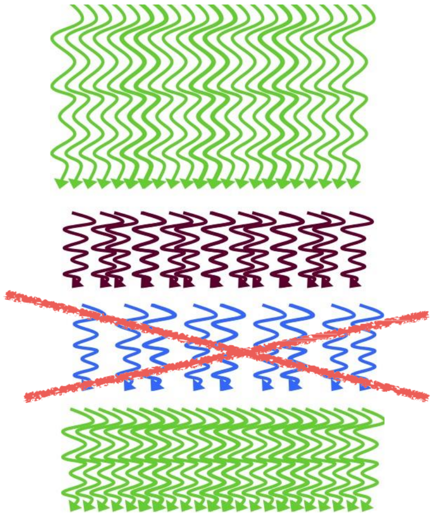
Instruction divergence



Divergence and deadlock

- Thread divergence can also cause a program to deadlock.

```
//my_Func_then and my_Func_else are
//some device functions
if (threadIdx.x <16) {
   myFunc_then();
   __syncthreads();
} else if (threadIdx.x >=16) {
   myFunc_else();
   __syncthreads();
}
```



- The first half of the warp will execute the then part, then wait for the second half of the warp to reach `__syncthread()`.
  However, the second half of the warp did not enter the then part; therefore, the first half of the warp will be waiting for them forever.

Divergence and programming

- **In general, one does not need to consider divergence for the correctness of a program**
  - Of course the deadlock case must be considered.

- **In general, one does need to consider divergence when reasoning about the performance of a program**

Divergence and performance

- Performance decreases with degree of divergence in warps

```
__global void dv(int* x) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  switch (i % 32) {
    case 0 : x[i] = a(x[i]);
             break;
    case 1 : x[i] = b(x[i]);
             break;

    ...

    case 31: x[i] = v(x[i]);
             break;
  }
}
```

Warp size !