



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# GPU programming basics

Prof. Marco Bertini



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# **Data parallelism: GPU computing**



# CUDA: blocks and threads

# SIMT execution

- Warp
  - smallest unit of concurrency: 32 threads
  - thread = single CUDA core
  - all threads execute same program
- Block
  - can synchronize (barriers)
  - can exchange data (common "shared" memory, etc.)
- Grid
  - grids/blocks serve as work distribution/sharing mechanism on device (occupancy)
  - blocks dispatched to SM (in turn run warps)



# Kernel functions

- A kernel function must be called with an execution configuration:
- `__global__ void kernelFoo(...); // declaration`
- `dim3 DimGrid(100, 50); // 5000 thread blocks`  
`dim3 DimBlock(4, 8, 8); // 256 threads per block`
- `kernelFoo<<< DimGrid, DimBlock>>>(...);`
- Recall that any call to a kernel function is asynchronous
  - By default, execution on host doesn't wait for kernel to finish

# Kernel functions

- A kernel function must be called with an execution configuration:
- `__global__ void kernelFoo(...); // declaration`
- `dim3 DimGrid(100, 50); // 5000 thread blocks`  
`dim3 DimBlock(4, 8, 8); // 256 threads per block`

$5000 \times 256 = 1.280.000 \text{ threads}$
- `kernelFoo<<< DimGrid, DimBlock>>>(...);`
- Recall that any call to a kernel function is asynchronous
  - By default, execution on host doesn't wait for kernel to finish

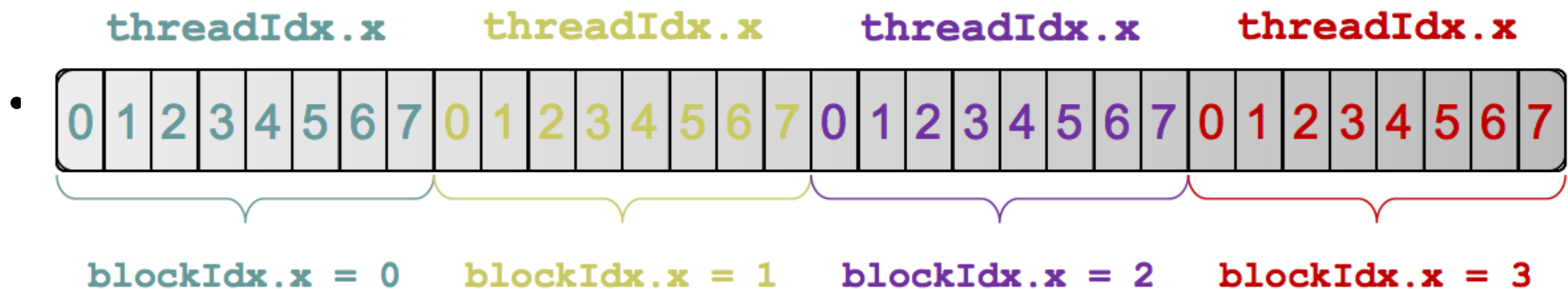
# Block

- The concept of block is important since it represents the entity that gets executed by an SM (streaming multiprocessor)
- The grid of blocks can be organized as a 3D structure (dim3: x, y, z): max of 65,535 by 65,535 by 65,535 grid of blocks (about 280,000 billion blocks)
- The threads can be organized as a 3D structure (dim3: x, y, z)
- The total number of threads in each block cannot be larger than 1024

# 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:  
$$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * M;$$
  
Where  $M$  is the size of the block of threads; i.e.,  $\text{blockDim.x}$

# Array indexing: example

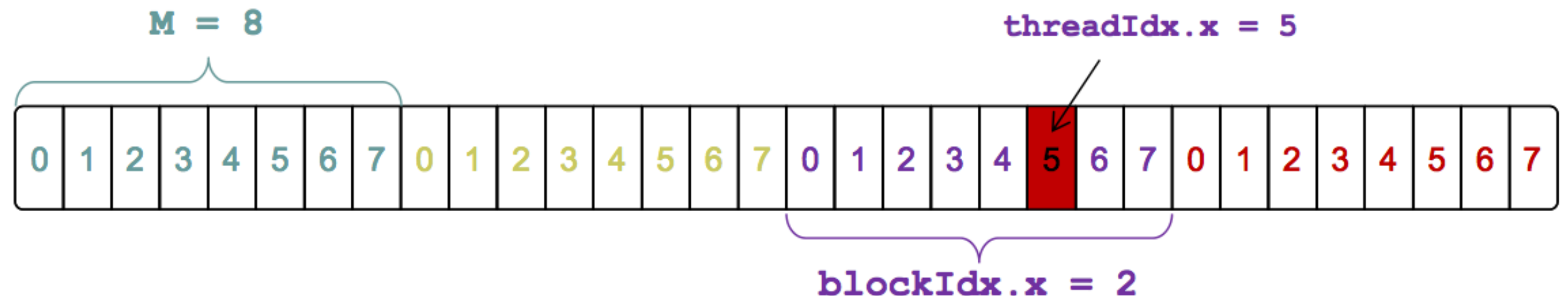
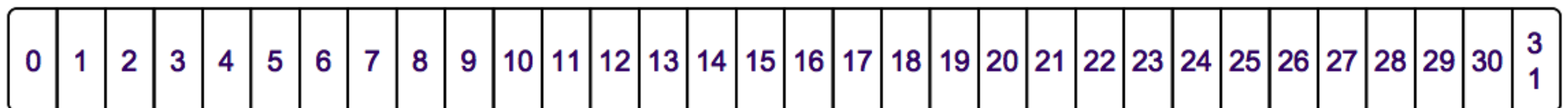


- Assume you launch with  $M=8$  threads per block and the array is 32 entries long
- With  $M$  threads per block a unique index for each thread is given by:  

$$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * M;$$
Where  $M$  is the size of the block of threads; i.e.,  
 $\text{blockDim.x}$

# Array indexing: example

- What is the array entry on which the thread with index 5 in block of index 2 will work ?
- $$\begin{aligned} \text{int index} &= \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}; \\ &= 5 + 2 * 8; \\ &= 21; \end{aligned}$$



# Thread indexes

- Given block and thread indexes a thread must compute on which part of the data it has to work on
- In some cases it does not need to work: check if it is the case
  - E.g.: two blocks with 512 threads working on an array of only 1000 elements long.  
Then 24 threads at the end do nothing.



# Using threads

- In GPU computing you use as many threads as data items [tasks] [jobs] you have to perform
  - This replaces the typical “for” loop.
  - Number of threads & blocks is established at run-time.
  - Number of threads = Number of data items [tasks] [jobs]

It means that you'll have to come up with a rule to match a thread to a data item [task] [job] that this thread needs to process.

If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data.

- Common source of errors and frustration in GPU computing.



# Built-in variables

- Each thread when executing a kernel has access to the following read- only built-in variables:
  - `threadIdx (uint3)` – contains the thread index within a block
  - `blockDim (dim3)` – contains the dimension of the block
  - `blockIdx (uint3)` – contains the block index within the grid
  - `gridDim (dim3)` – contains the dimension of the grid

# Execution

- Blocks are assigned to SM. Possibly multiple blocks are running at the same time on a SM.
- Threads of blocks are divided in warps (32 threads). Multiple warps are running at the same time.
- At each clock tick, SM warp scheduler decides which warp to execute next, choosing from those not waiting for
  - data coming from device memory (memory latency)
  - completion of earlier instructions (pipeline delay)

# Execution

- Blocks are assigned to SM. Possibly multiple blocks are running at the same time on a SM.
- Threads of blocks are divided in warps (32 threads). Multiple warps are running at the same time.

So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

warp to execute next, choosing from those not waiting for

- data coming from device memory (memory latency)
- completion of earlier instructions (pipeline delay)

# Execution

- Blocks are assigned to SM. Possibly multiple blocks are running at the same time on a SM.
- Threads of blocks are divided in warps (32 threads). Multiple warps are running at the same time.

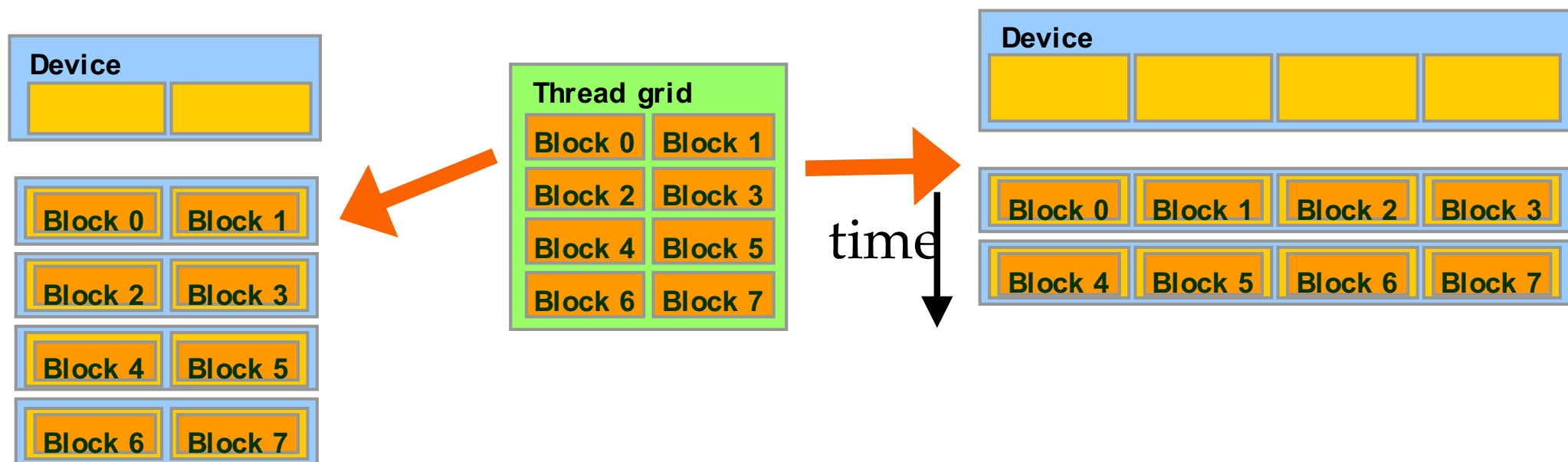
So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

**warp to execute next, choosing from those not waiting for**

Different GPU models have different number of resident blocks (e.g. 32, 16, 8) and different number of resident warps (e.g. 64, 48, 32).

- data coming from device memory (memory latency)
- completion of earlier instructions (pipeline delay)

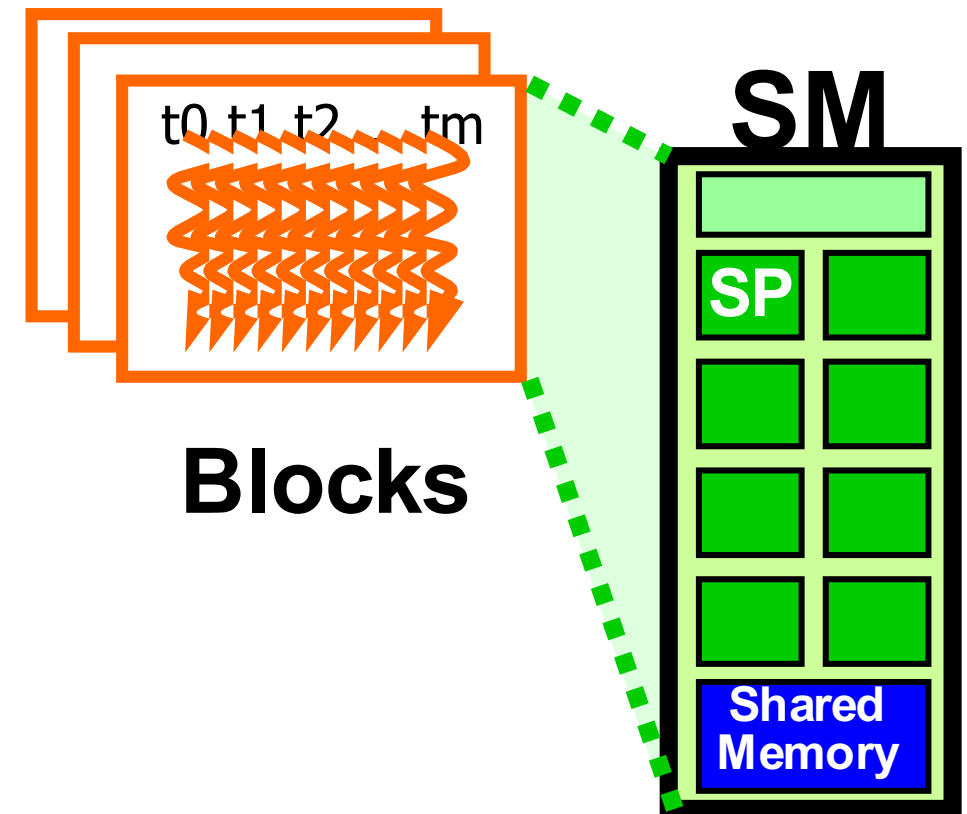
# Transparent scalability



- 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

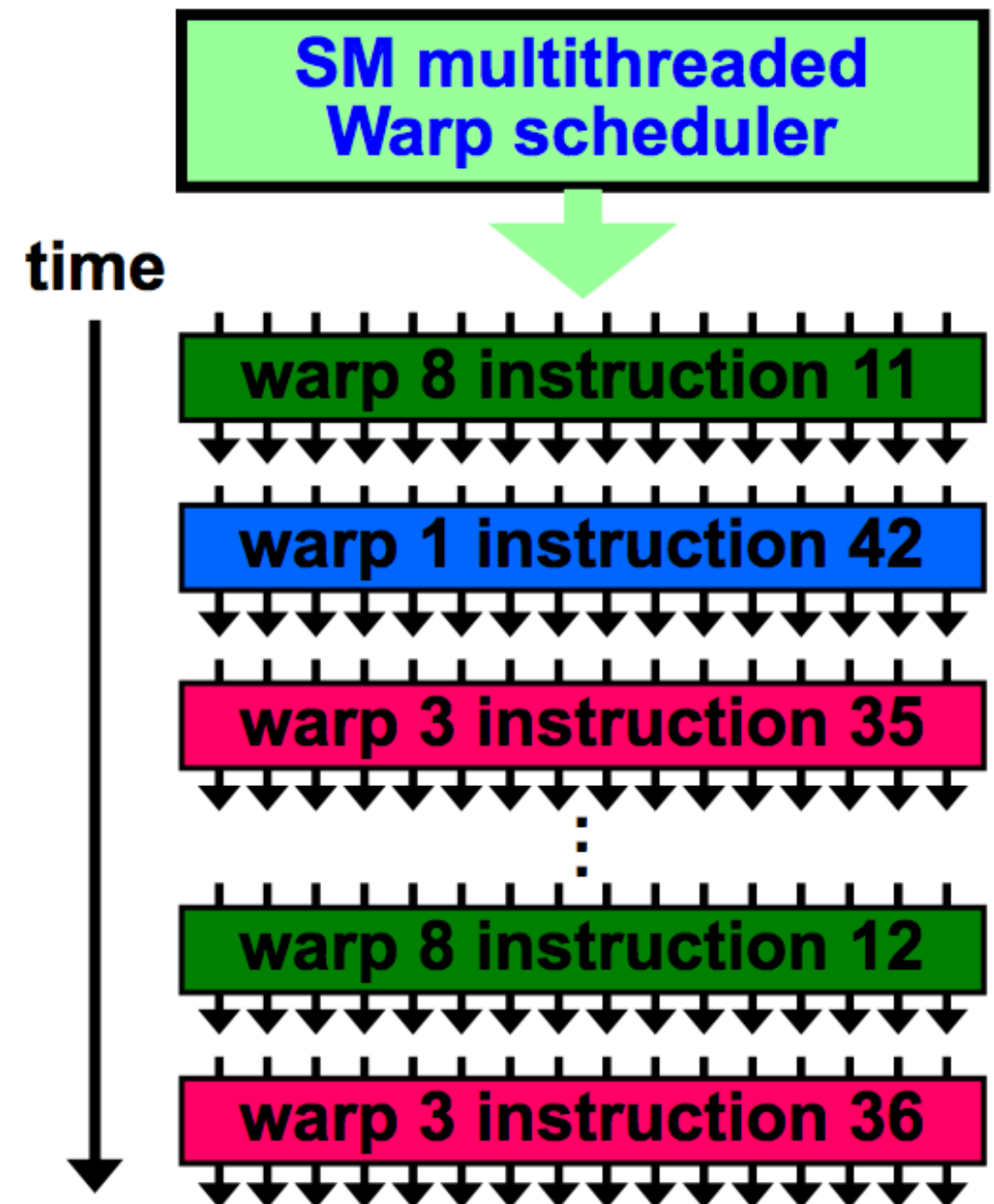
# Executing thread blocks

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to 8 blocks to each SM as resource allows
  - Fermi SM can take up to 1536 threads
  - Could be  $256 (\text{threads/block}) * 6$  blocks
  - Or  $512 (\text{threads/block}) * 3$  blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



# Warp scheduling

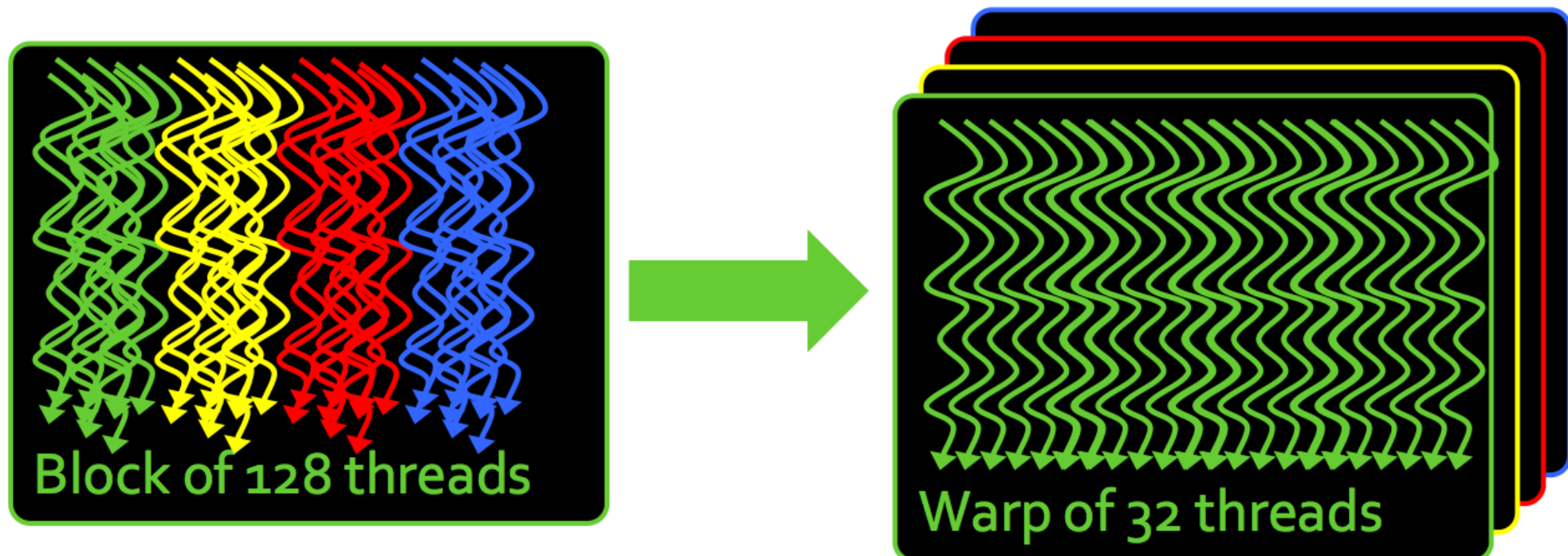
- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a Warp execute the same instruction when selected





# 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

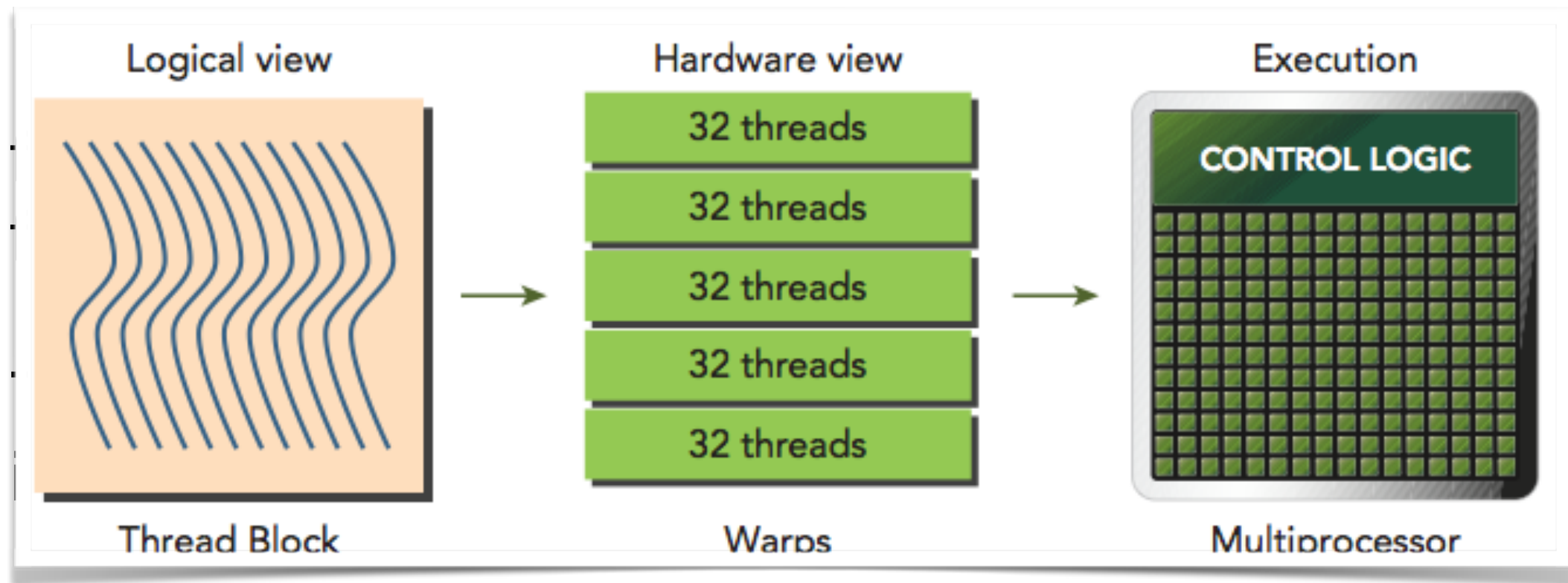




# Threads and Warps

- 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 is always the same
  - You can use this knowledge in control flow

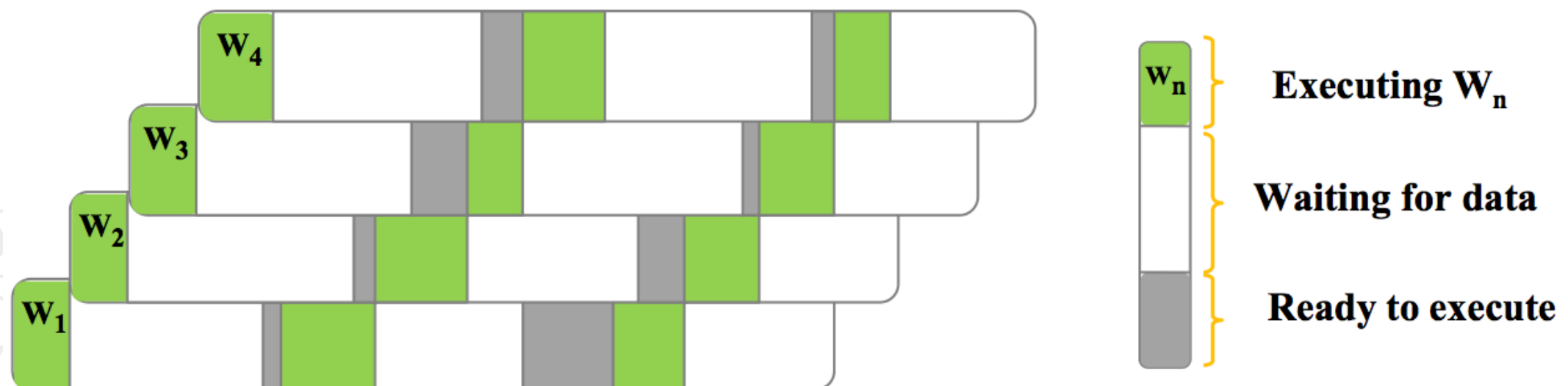
# Threads and Warps



- Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.
- Partitioning of threads in warps 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)

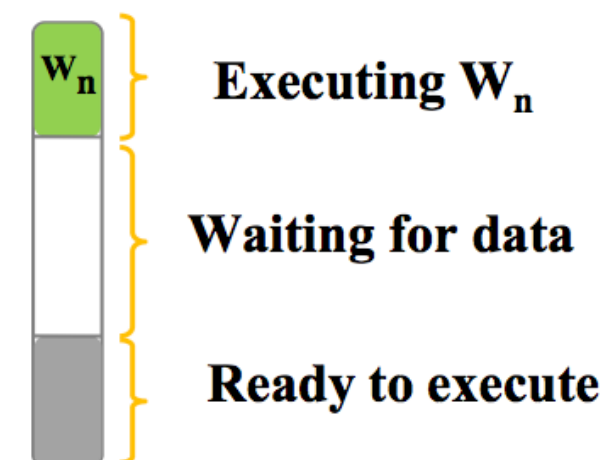
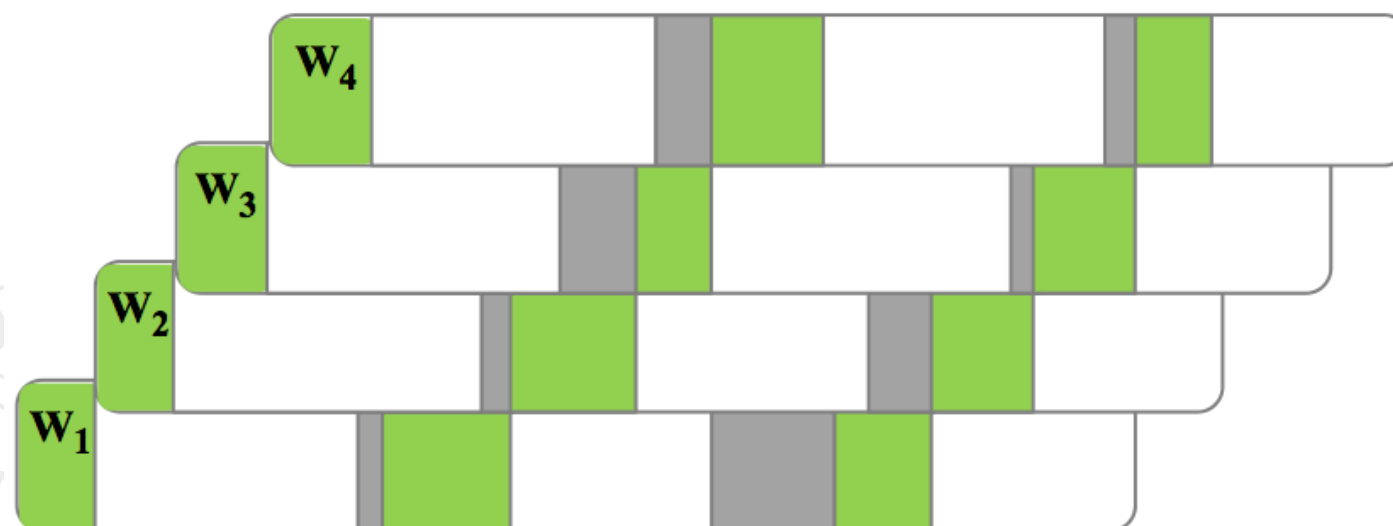


# Thread and Warp scheduling

- 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

hide (memory) latency by pipelining active warps

- 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**:  $\text{kernel} \lll N, 1 \ggg ( \dots )$
  - **Okay**:  $\text{kernel} \lll (N+31) / 32, 32 \ggg ( \dots )$
  - **Better**:  $\text{kernel} \lll (N+127) / 128, 128 \ggg ( \dots )$
- 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

# Block level synchronization

- In CUDA, synchronization can be performed at two levels:
  - System-level: Wait for all work on both the host and the device to complete (`cudaDeviceSynchronize`).
  - Block-level: Wait for all threads in a thread block to reach the same point in execution on the device.
- Because warps in a thread block are executed in an undefined order, CUDA provides the ability to synchronize their execution with a block-level barrier. You can mark synchronization points in the kernel using:  

```
__device__ void __syncthreads(void);
```

# 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

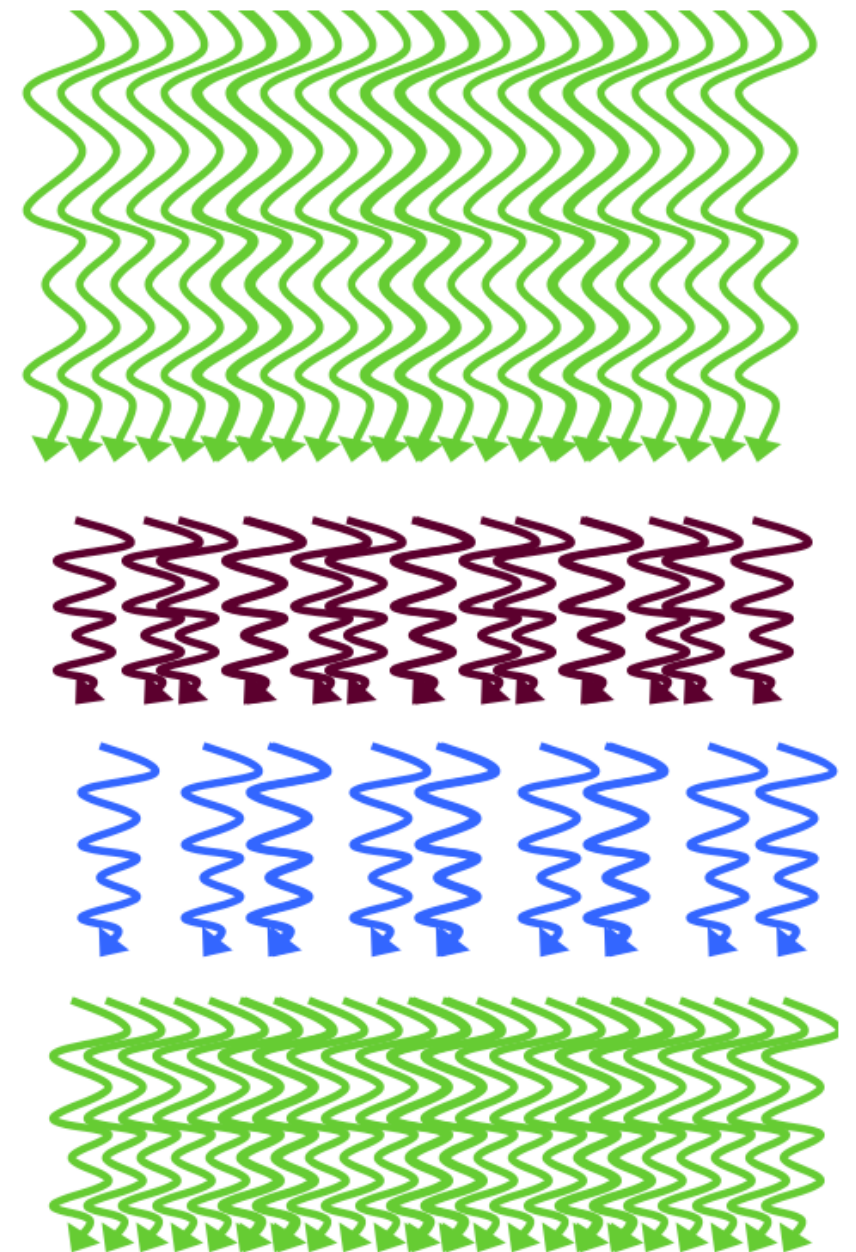
- Intuitively, we would think statements in *then* and *else* blocks should be executed in parallel. However, because of the requirement that threads in a warp cannot diverge, this cannot happen.  
The CUDA platform has a workaround that fixes the problem, but has **negative performance** consequences.
- When executing the **if-then-else** statement, the CUDA platform will instruct the warp to execute the **then** part first, and then proceed to the **else** part.  
While executing the **then** part, all threads that evaluated to false (e.g. the **else** threads) are effectively deactivated. When execution proceeds to the **else** condition, the situation is reversed.
- The **then** and **else** parts are not executed in parallel, but in serial. This serialization can result in a significant performance loss.
- Nested branches are handled similarly.  
Deeper nesting results in more threads being temporarily disabled



# Divergence and execution

```
__global__ void kv(int* x, int* y) {
    int i = threadIdx.x + blockDim.x *
            blockIdx.x;

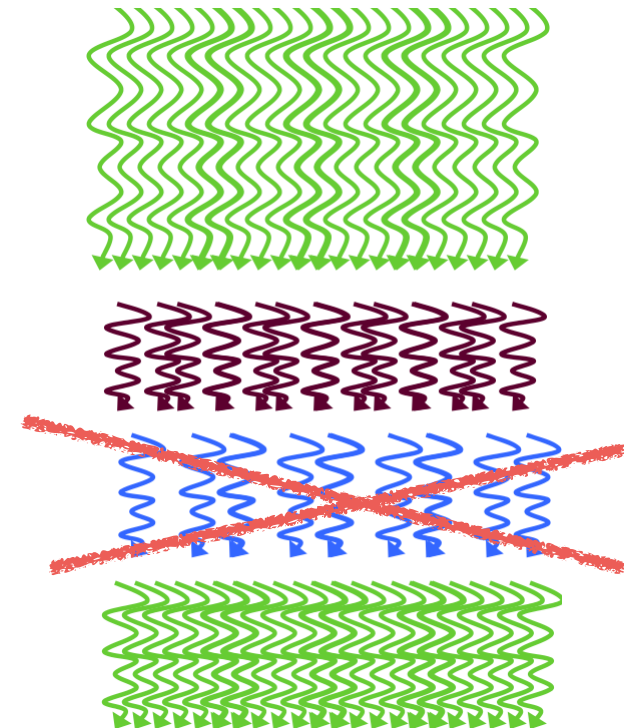
    int t;
    bool b = f(x[i]);
    if( b ) {
        // g(x)
        t = g(x[i]);
    } else {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
```



# 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();  
}
```

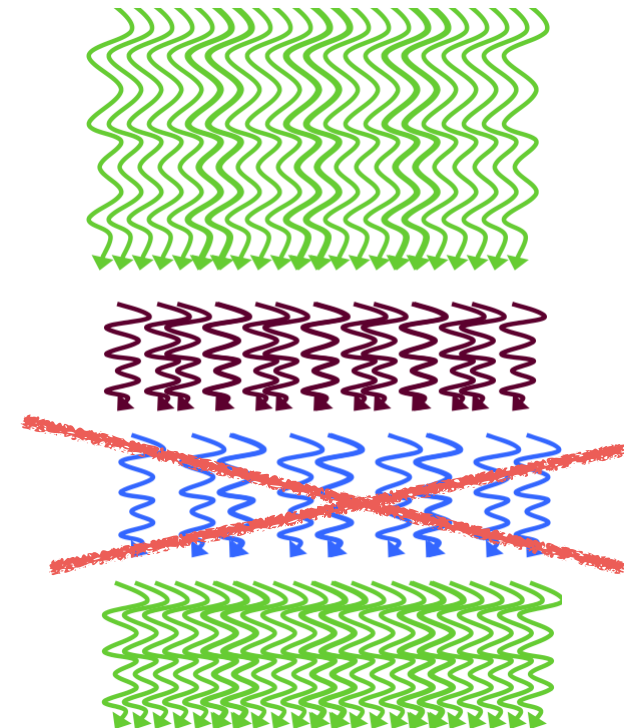


- The first half of the warp will execute the then part, then wait for the second half of the warp to reach `__syncthreads()`. However, the second half of the warp did not enter the then part; therefore, the first half of the warp will be waiting for them forever.

The `__syncthreads()` command is a **block** level synchronization barrier. That means it is safe to be used when all threads in a block reach the barrier.

- It is also possible to use `__syncthreads()` in conditional code **but only when all threads evaluate identically** such code, otherwise the execution is likely to hang

```
//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 `__syncthreads()`. 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.

# \_\_syncthreads and deadlock

- From the NVIDIA Programming guide of Compute Capability 7.x (Volta):

Although `__syncthreads()` has been consistently documented as synchronizing all threads in the thread block, Pascal and prior architectures could only enforce synchronization at the warp level. In certain cases, this allowed a barrier to succeed without being executed by every thread as long as at least some thread in every warp reached the barrier. Starting with Volta, the CUDA built-in `__syncthreads()` and PTX instruction `bar.sync` (and their derivatives) are enforced per thread and thus will not succeed until reached by all non-exited threads in the block. Code exploiting the previous behavior will likely deadlock and must be modified to ensure that all non-exited threads reach the barrier.

# \_\_syncthreads and deadlock

- The pre 7.x (Volta) behavior means that:
- If any thread in a warp executes a PTX bar instruction (e.g. from \_\_syncthreads), it is as if all the threads in the warp have.
- The threads within a warp are not synchronized by \_\_syncthreads(). The instruction will not cause the warp to stall and wait for the threads on divergent paths. Branch execution is serialized, so only when the branches rejoin or the code terminates do the threads in the warp then resynchronize.  
Until that, the branches run in sequence and independently.  
Again, only one thread in each warp of the block needs to hit \_\_syncthreads() for execution to continue.



# Divergence and programming

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - ... of course consider cases that may cause a deadlock like the previous example
- In general, one does need to consider divergence when reasoning about the performance of a program



# Divergence and performance

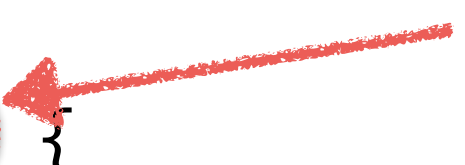
- Performance decreases with degree of divergence in warps

```
__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;  
    }  
}
```

# 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 !



# nvcc optimization

- CUDA compiler optimization replaces branch instructions (which cause actual control flow to diverge) with predicated instructions for short, conditional code segments.
- In branch predication, a predicate variable for each thread is set to 1 or 0 according to a condition. The conditional flow paths become sequential and are coded inline, but only instructions with a predicate of 1 are executed. Instructions with a predicate of 0 do not, but the corresponding thread does not stall either.
- The compiler replaces a branch instruction with predicated instructions only if the number of instructions in the body of a conditional statement is less than a certain threshold.

# nvcc optimization

- ```
__global__ void odd_even(int n, int* x) {  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    bool ipred = (i % 2 == 0);  
  
    if( ipred ) {  
        x[i] = x[i] + 1;  
    }  
    if( !ipred) {  
        x[i] = x[i] + 2;  
    }  
}
```
- The compiler replaces a branch instruction with predicated instructions only if the number of instructions in the body of a conditional statement is less than a certain threshold.

```
if condition
  do this
else
  do that
```

On a system that uses conditional branching, this might translate to machine instructions looking similar to:

```
branch if condition to label 1
  do that
  branch to label 2
label 1:
  do this
label 2:
  ...
```

With branch predication the machine code becomes:

```
(condition) do this
(not condition) do that
```

if condition is FALSE, then the instruction is treated as NOP

```
if condition
  do this
else
  do that
```

On a system that uses conditional branching, this might translate to machine instructions looking similar to:

```
branch if condition to label 1
  do that
  branch to label 2
label 1:
  do this
label 2:
  ...
“predication”
```

With branch predication the machine code becomes:

```
(condition) do this
(not condition) do that
```

if condition is FALSE, then the instruction is treated as NOP



# Organizing blocks and threads

# Grid dimensions

- `gridDim.x`, `gridDim.y`, and `gridDim.z` range from 1 to 65,536
- All threads in a block share the same `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` values
- Among all blocks, the `blockIdx.x` value ranges between 0 and `gridDim.x-1`, the `blockIdx.y` value between 0 and `gridDim.y-1`, and the `blockIdx.z` value between 0 and `gridDim.z-1`.

# Block dimensions

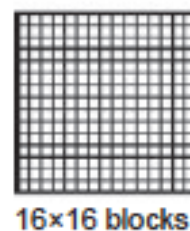
- The total size of a block is limited to 1024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1024.
- For example,  $(512, 1, 1)$ ,  $(8, 16, 4)$ , and  $(32, 16, 2)$  are all allowable `blockDim` values, but  $(32, 32, 2)$  is not allowable (it's 2048).





# Mapping threads to data

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data.  
For example, pictures are a 2D array of pixels or 3D arrays of single channels.
- Let us consider a  $76 \times 62$  pixel image, and we want to process with a  $16 \times 16$  threads block.  
We need  $5 \times 4$  and some of the blocks will have unused threads



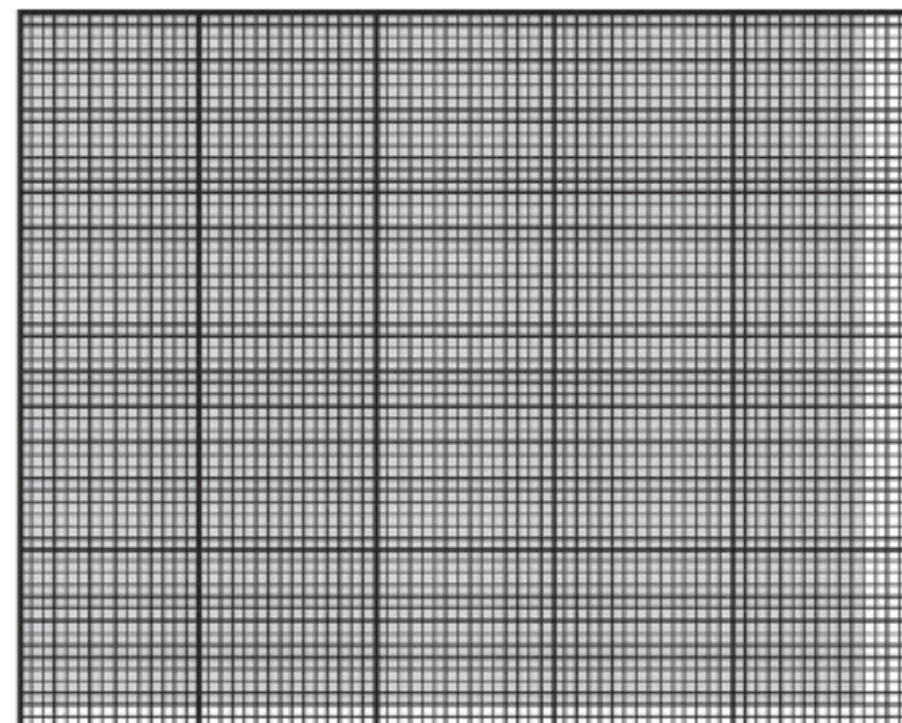
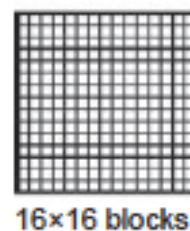


# Mapping threads to data

We will have some code similar to:

```
dim3 dimGrid(ceil(im_width/16.0), ceil(im_height/16.0), 1);  
dim3 dimBlock(16, 16, 1);  
pictureKernel<<<dimGrid,dimBlock>>>(d_Pin,d_Pout,im_width, im_height);
```

- Let us consider a  $76 \times 62$  pixel image, and we want to process with a  $16 \times 16$  threads block.  
We need  $5 \times 4$  and some of the blocks will have unused threads



# 2D kernel example

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int w, int h) {  
  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((row < w) && (col < h)) {  
        d_Pout[ row * w + col ] = 2 * d_Pin[ row * w + col ];  
    }  
  
}
```

# 2D kernel example

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int w, int h) {  
    Number of threads along the y axis  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((row < w) && (col < h)) {  
        d_Pout[ row * w + col ] = 2 * d_Pin[ row * w + col ];  
    }  
}
```

# 2D kernel example

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int w, int h) {  
    Number of threads along the y axis  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    Number of threads along the x axis  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((row < w) && (col < h)) {  
        d_Pout[ row * w + col ] = 2 * d_Pin[ row * w + col ];  
    }  
}
```

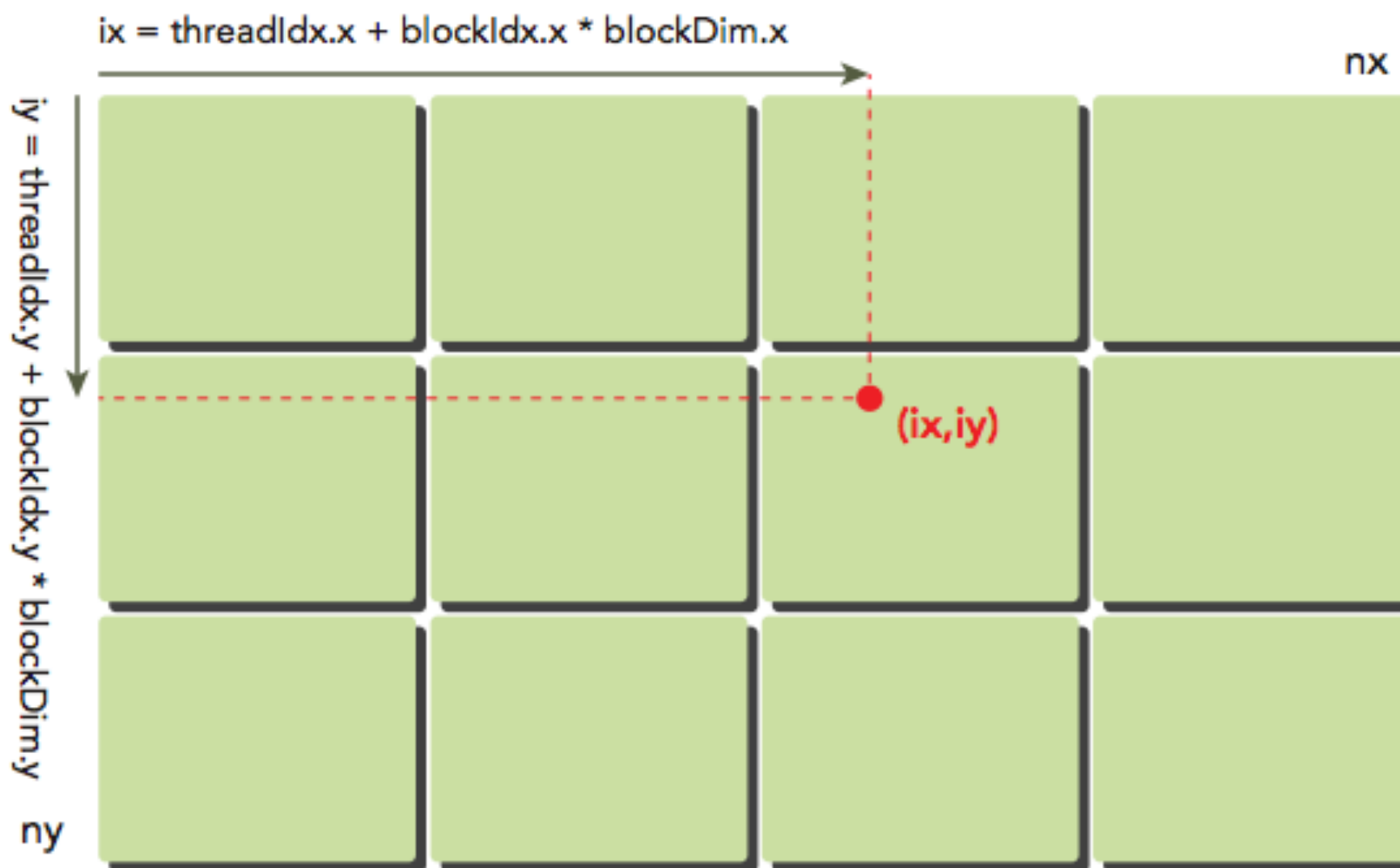
# 2D kernel example

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int w, int h) {  
    Number of threads along the y axis  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    Number of threads along the x axis  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    Check the unused threads  
    // each thread computes one element of d_Pout if in range  
    if ((row < w) && (col < h)) {  
        d_Pout[ row * w + col ] = 2 * d_Pin[ row * w + col ];  
    }  
}
```

# 2D kernel example

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int w, int h) {  
    Number of threads along the y axis  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    Number of threads along the x axis  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    Check the unused threads  
    // each thread computes one element of d_Pout if in range  
    if ((row < w) && (col < h)) {  
        d_Pout[ row * w + col ] = 2 * d_Pin[ row * w + col ];  
    }  
    Standard 2D array coordinate conversion  
}
```

# Block and thread indices



matrix coordinate:  $(ix, iy)$

global linear memory index:  $idx = iy * nx + ix$

- Correspondences with previous code:
- $row = ix$
- $col = iy$
- $nx = w$  (width)
- $ny = h$  (height)



# Another example

```
#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
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

# Another example

```
#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
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Same as before



# Another example

```
#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
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Same as before

Standard 2D array coordinate conversion



# Another example

```
#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;
```

Same as before

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

Standard 2D array coord

```
        // 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
```

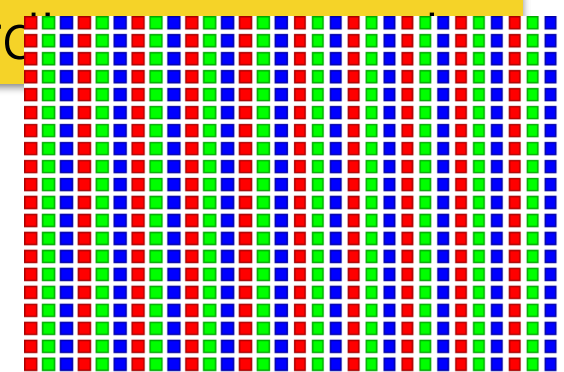
```
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
```

```
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
```

```
        // perform the rescaling and store it
```

```
        // We multiply by floating point constants
```

```
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
```



# Another example

```
#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;
```

Same as before

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

Standard 2D array coord

```
        // 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
```

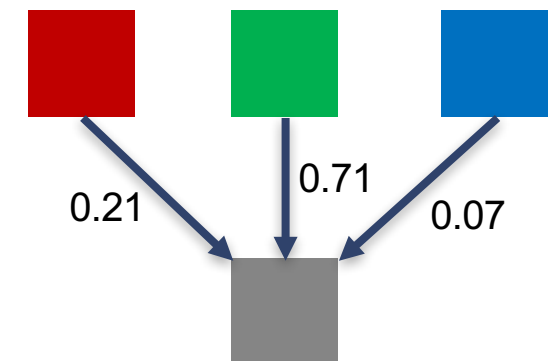
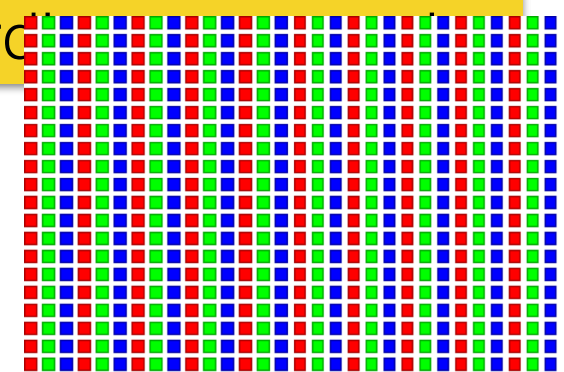
```
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
```

```
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
```

```
        // perform the rescaling and store it
```

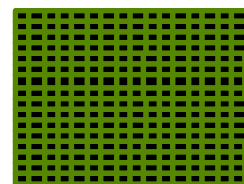
```
        // We multiply by floating point constants
```

```
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
```

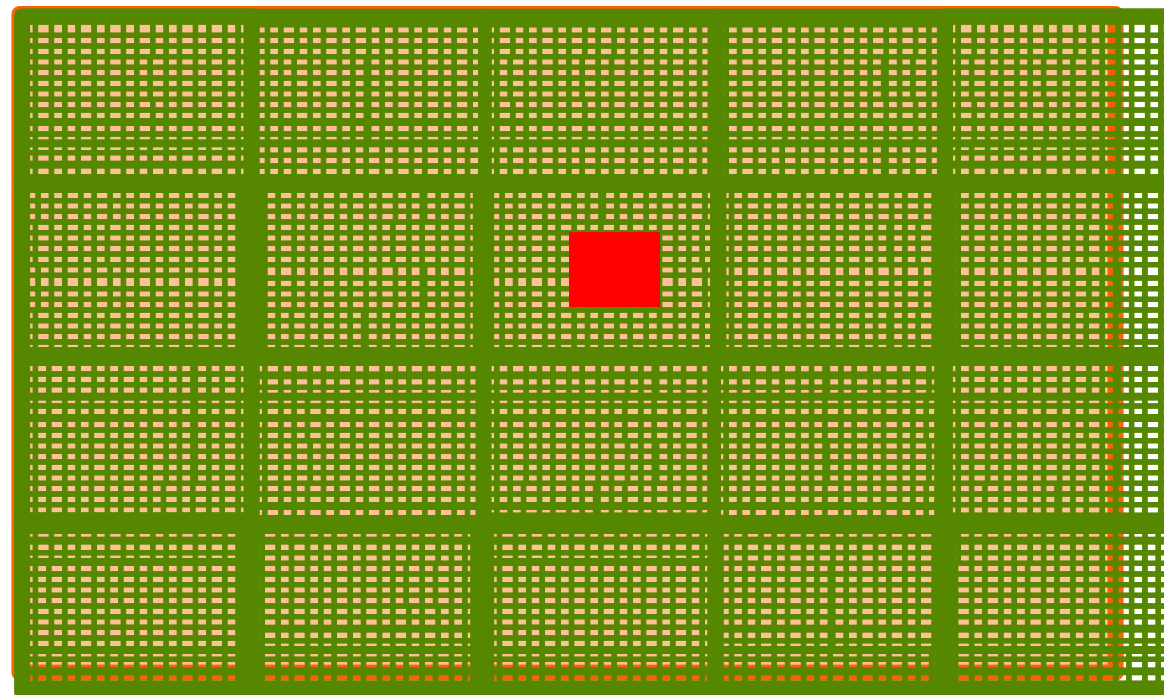


# More complex example

- E.g. image blurring: we have to deal with more computations and more complex memory access patterns



Pixels  
processed  
by a thread  
block



\_\_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
    }
}
```

The main structure is the same  
of the first example



```
__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);  
    }  
}
```

# 3D kernel

- Extend the previous case adding a plane variable
- $\text{int plane} = \text{blockIdx.z} * \text{blockDim.z} + \text{threadIdx.z}$
- Linearize the 3D coordinate with:
- $\text{1Dcoord} = \text{Plane} * \text{height} * \text{width} + \text{Row} * \text{width} + \text{Col};$

# Guidelines

- Keep the number of threads per block a multiple of warp size (32).
- Avoid small block sizes: Start with at least 128 or 256 threads per block.
  - When one warp stalls, the SM switches to executing other eligible warps. Ideally, you want to have enough warps to keep the cores of the device occupied.
  - But too many threads per block leads to fewer per-SM hardware resources available to each thread.
- Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism to your device.
- Conduct experiments to discover the best execution configuration and resource usage.
  - Use the CUDA Occupancy Calculator Excel file to get help in selecting parameters
  - Use nvprof to profile the code

# Example: block granularity evaluation

- For Matrix Multiplication using multiple blocks, should I use  $8 \times 8$ ,  $16 \times 16$  or  $32 \times 32$  blocks for Fermi?
  - For  $8 \times 8$ , we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For  $16 \times 16$ , we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For  $32 \times 32$ , we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.

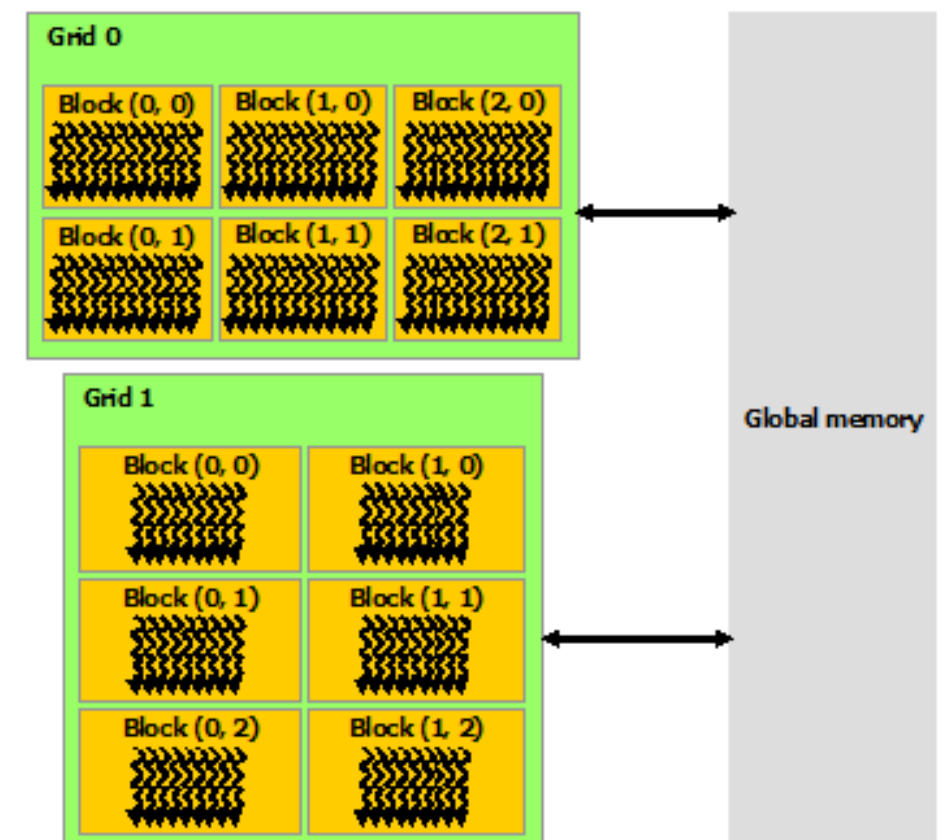
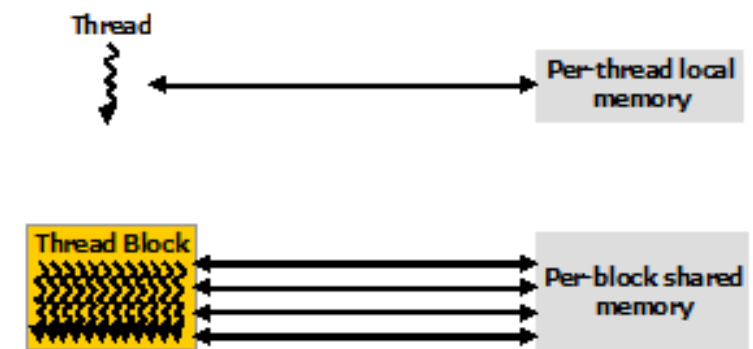


UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# CUDA Memory

# Memory Hierarchy

- CUDA threads may access data from multiple memory spaces during their execution:
  - Each thread has private local memory.
  - Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
  - All threads have access to the same global memory.



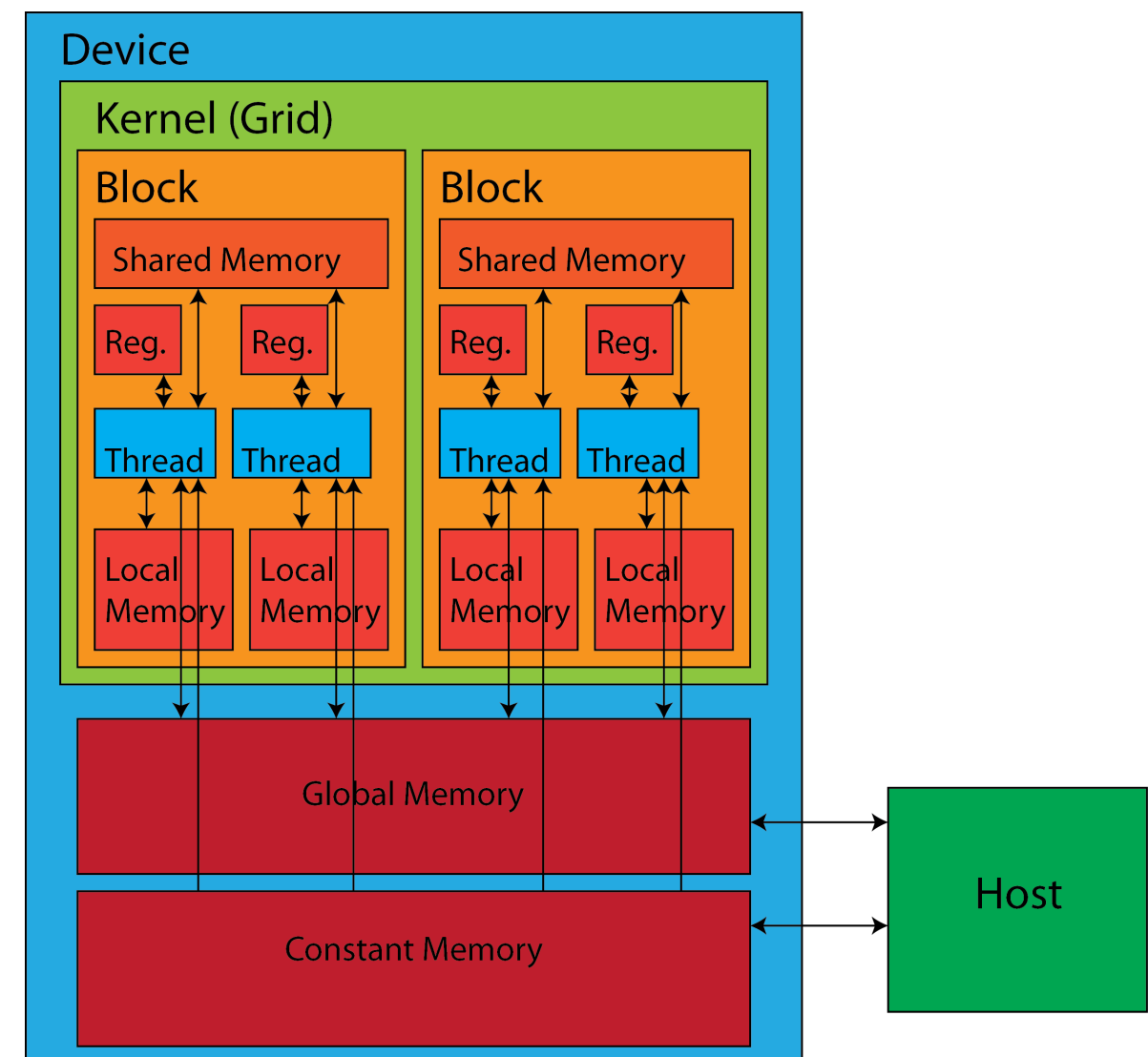
# Memory and GPU performance

- Reconsider the blurKernel code: all threads access global memory for their input matrix elements
  - One memory accesses (4 bytes) per floating-point addition
- Assume a GPU with
  - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
  - $4 \times 1,500 = 6,000$  GB/s required to achieve peak FLOPS rating
  - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS
- This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS



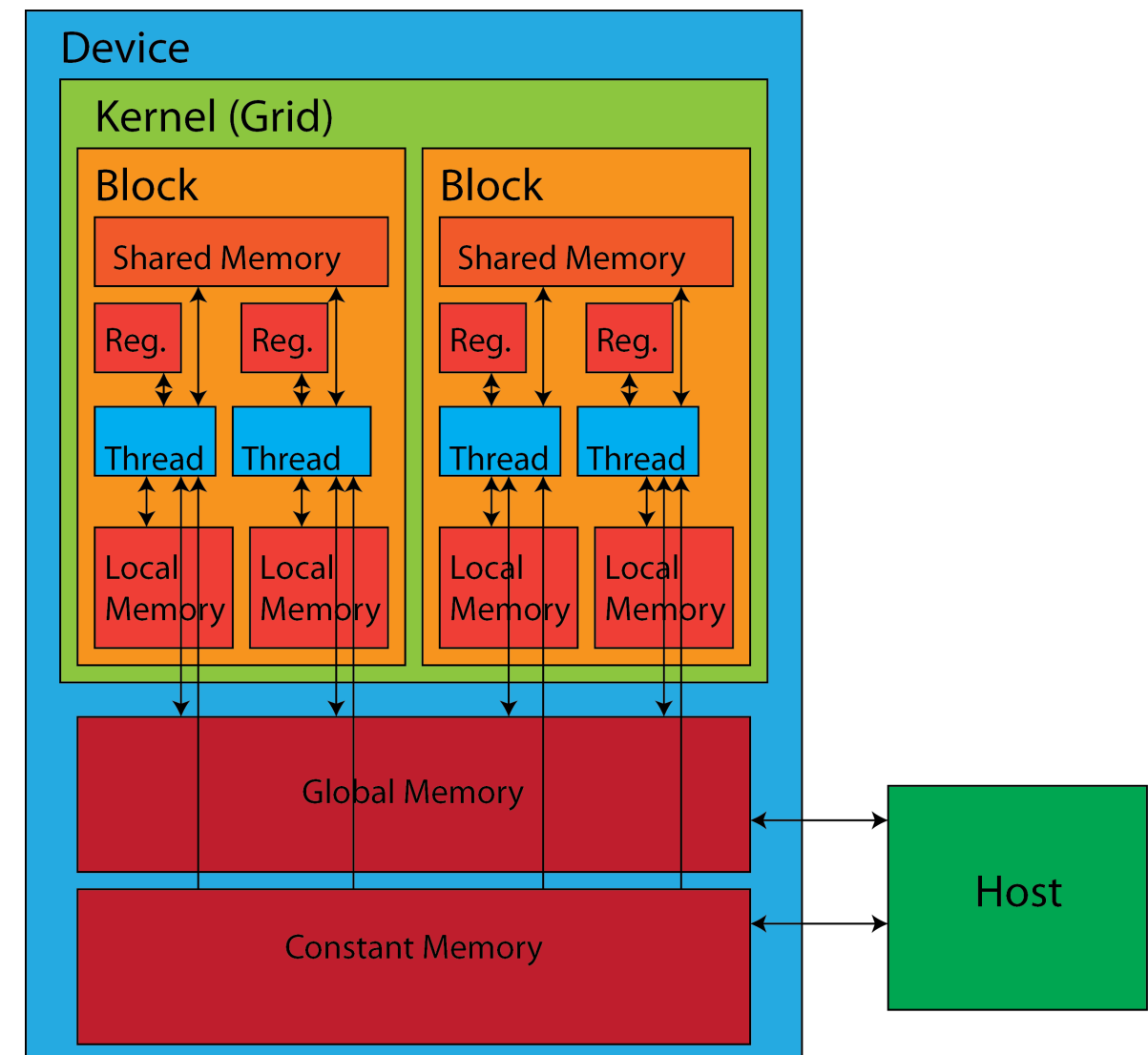
# Memory access

- In general, we use the host to:
  - Transfer data to and from global memory
  - Transfer data to and from constant memory
- Once the data is in the device memory, threads can read and write (R/W) different parts of memory:
  - R/W per-thread register
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - R per-grid constant memory



# Memory access

- The global, and constant memory spaces are persistent across kernel launches by the same application.
  - Transfer data to and from global memory
  - Transfer data to and from constant memory
- Once the data is in the device memory, threads can read and write (R/W) different parts of memory:
  - R/W per-thread register
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - R per-grid constant memory



# Variable qualifiers

- CUDA supports different types of qualifiers to assign variables to different memory components. Different memory components also have a life span

| Qualifiers                           | Location | Scope  | Lifespan    |
|--------------------------------------|----------|--------|-------------|
| Automatic variable                   | Register | Thread | Kernel      |
| Automatic array                      | Local    | Thread | Kernel      |
| __device__ __shared__ int SharedVar  | Shared   | Block  | Kernel      |
| __device__ int GlobalVar             | Global   | Grid   | Application |
| __device__ __constant__ int ConstVar | Constant | Grid   | Application |

# Registers

- Maxwell GPUs have 64K 32 bit registers.
- Registers are dynamically partitioned across all Blocks assigned to the SM.
- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks.
- A thread in a Block can only access registers assigned to itself.
  - On a Maxwell GPU a thread can get up to 255 registers, assigned by the compiler
- Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory.

# Local memory

- Local memory does not exist physically: it is “local” in scope (i.e., it’s specific to one thread) but not in location
- Data that is stored in “local memory” is actually placed in cache or the global memory at run time or by the compiler.
- If too many registers are needed for computation (“high register pressure”) the ensuing data overflow is stored in local memory
- Register and shared memory is within the SM, i.e. very fast. Global and instant are in the DRAM shared by the SMs of the GPU.

# Shared memory

- Shared memory is on-chip, thus it has a much higher bandwidth and much lower latency than local or global memory. It is used similarly to CPU L1 cache, but is also programmable.
- Each SM has a limited amount of shared memory that is partitioned among thread blocks.
  - Do not over-utilize shared memory or you will inadvertently limit the number of active warps.
- Shared memory serves as a basic means for inter-thread communication. Threads within a block can cooperate by sharing data stored in shared memory.
- Access to shared memory must be synchronized using `__syncthreads()`





A block may have 48KB shared memory, but if N blocks are running on a SM each block gets 48/N KB

# Shared memory

- If shared memory size is known, we may use the CUDA qualifier, `__shared__`, to declare the size of shared memory statically as follows:

```
__shared__ int sharememory[size];
```

- If the size is unknown at compile time, we need to dynamically allocate shared memory. Do it by prepending the keyword `extern` to an unsized memory array in the kernel:

```
extern __shared__ int sharememory[];
```

- Then, specify the size when the kernel is launched. Here we want to declare a shared memory array consisting of n integers:

```
kernel_func<<<gridSize, blockSize, n*sizeof(int)>>>  
(...);
```

# Constant Memory

- Constant memory resides in device memory and is cached in a dedicated, per-SM constant cache.
- Kernels can only read from constant memory. Constant memory must therefore be initialized by the host using `cudaMemcpyToSymbol`
- Constant memory performs best when all threads in a warp read from the same memory address.
  - Example: a coefficient for a mathematical formula is a good use case because all threads in a warp will use the same coefficient to conduct the same calculation on different data.

# Automatic Variables and Arrays

- An automatic variable is declared without any qualifiers. It resides in the per-thread register and is only accessible by that thread.

```
int autovar;
```

- An automatic array variable resides in the per-thread local memory and is only accessible by that thread. However, it is possible for the compiler to store automatic arrays in the registers, if all access is done with constant index values.

```
int autoarr[];
```

# Shared, device and constant

- Appending the `__shared__` variable type qualifier explicitly declares that the variable is shared within a thread block.

```
__shared__ int shvar;  
// is the same as  
__device__ __shared__ int shvar2;
```

- When the `__device__` qualifier is used by itself, it declares the variable resides in global memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__device__ int dvvar;
```

- Appending the `__constant__` qualifier declares a constant variable that resides in the constant memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__constant__ int cnvar;  
// is the same as  
__device__ __constant__ int cnvar2;
```

# Shared, device and constant

- Appending the `__shared__` variable type qualifier explicitly declares that the variable is shared within a thread block.

```
__shared__ int shvar;  
// is the same as  
__device__ __shared__ int shvar2;
```

If data that you use turns out that can be used by any other thread in your block then you should consider using shared memory.

- When the `__device__` qualifier is used by itself, it declares the variable resides in global memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__device__ int dvvar;
```

- Appending the `__constant__` qualifier declares a constant variable that resides in the constant memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__constant__ int cnvar;  
// is the same as  
__device__ __constant__ int cnvar2;
```

`__device__` and `__constant__` are accessible through `cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`

- Appending the `__shared__` variable type qualifier explicitly declares that the variable is shared within a thread block.

```
__shared__ int shvar;  
// is the same as  
__device__ __shared__ int shvar2;
```

If data that you use turns out that can be used by any other thread in your block then you should consider using shared memory.

- When the `__device__` qualifier is used by itself, it declares the variable resides in global memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__device__ int dvvar;
```

- Appending the `__constant__` qualifier declares a constant variable that resides in the constant memory. Is accessible from all the threads within the grid and from the host through the runtime library.

```
__constant__ int cnvar;  
// is the same as  
__device__ __constant__ int cnvar2;
```

# Pinned memory: why ?

- To allow programmers to use a larger virtual address space than is actually available in the RAM, CPUs (or hosts, in the language of GPGPU) implement a virtual memory system, in which a memory page may be swapped out to disk. When needed the page is read back from disk.
- The CUDA driver may need to request the CPU to read back the page from disk, slowing down the access to RAM.





# Pinned memory

- To improve data transfer speed between host and device, you can pin device memory to the host memory, which will allow the system to use Direct Memory Access (DMA) transfer between the host and GPU whenever data transfer is specified, which is significantly faster.
- To do so, you can simply replace `malloc()` by the API function, `cudaMallocHost()`. Es.:

```
cudaMallocHost((int **) &h_t1, size);
```

This function prevents the memory from being swapped out.

# Pinned memory: performance

- Pinned memory is more expensive to allocate/deallocate, but has higher transfer throughput.
- It is conveniente when dealing with large buffers, e.g. tens/hundreds of MBs that have to be transferred back and forth.



# Unified memory

- Since CUDA 6.0 (and Kepler architecture) has been introduced a Unified Memory scheme that unifies (virtually) host and GPU memory.
- No need for explicit data transfer (that still happens, of course, since physically memory is separated)
- Similar to OS virtual memory
- Slower execution than manually optimized memory management

Use `cudaMallocManaged()` to allocate Unified Memory, accessible from CPU and GPU:  
`cudaError_t cudaMallocManaged(void** ptr, size_t size);`

# Unified memory

- Since CUDA 6.0 (and Kepler architecture) has been introduced a Unified Memory scheme that unifies (virtually) host and GPU memory.
- No need for explicit data transfer (that still happens, of course, since physically memory is separated)
- Similar to OS virtual memory
- Slower execution than manually optimized memory management



# Unified Memory

- On systems with pre-Pascal GPUs like the Tesla K80, calling `cudaMallocManaged()` allocates size bytes of managed memory on the GPU device that is active when the call is made. Internally, the driver also sets up page table entries for all pages covered by the allocation, so that the system knows that the pages are resident on that GPU.
- Since these older GPUs can't page fault, all data must be resident on the GPU just in case the kernel accesses it (even if it won't). This means there is potentially migration overhead on each kernel launch.

# Unified Memory

- On Pascal and later GPUs, managed memory may not be physically allocated when `cudaMallocManaged()` returns; it may only be populated on access (or prefetching). In other words, pages and page table entries may not be created until they are accessed by the GPU or the CPU. The pages can migrate to any processor's memory at any time, and the driver employs heuristics to maintain data locality and prevent excessive page faults.
- Since Tesla P100 (Pascal) there is hardware page faulting and migration support. kernel launches without any migration overhead, and when it accesses any absent pages, the GPU stalls execution of the accessing threads, and the Page Migration Engine migrates the pages to the device before resuming the threads.

# Credits

- These slides report material from:
  - Prof. Dan Negrut (Univ. Wisconsin - Madison)
  - Philip Nee (Cornell Univ.)
  - NVIDIA GPU Teaching Kit



# Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 2nd edition - Chapt. 4-5

or

Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 3rd edition - Chapt. 3-4

- Professional CUDA C Programming, J. Cheng, M. Grossman and T. McKercher, Wrox - Chapt. 3