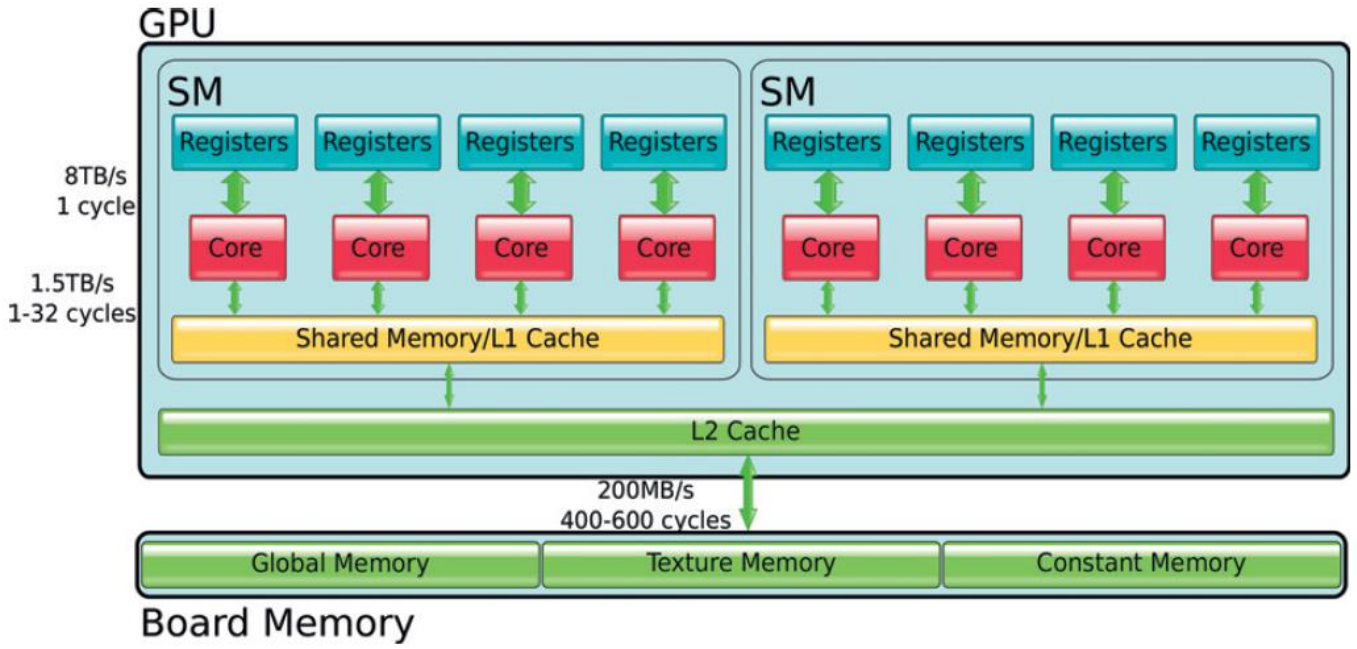
Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Memory Types

There are multiple types of memories, some on-chip and some off-chip.



- **Registers:** Holding local variables
- **Shared Memory:** Fast on-chip memory to hold frequently used data. Can be used to exchange data between the corse of the same SM.
- **L1/L2 Cache:** Transparent to the programmer
- **Global memory:** Main part of the off-chip memory. High capacity but relatively slow. The only part accessible by the host through CUDA functions
- **Texture and surface memory:** Content managed by special hardware that permits fast implementation of some filtering/interpolation operator
- **Constant memory:** Can only store constants. It is cached, and allows broadcasting of a single value to all threads in a warp (less appealing on newer GPUs that have a cache anyway)

Memory Types

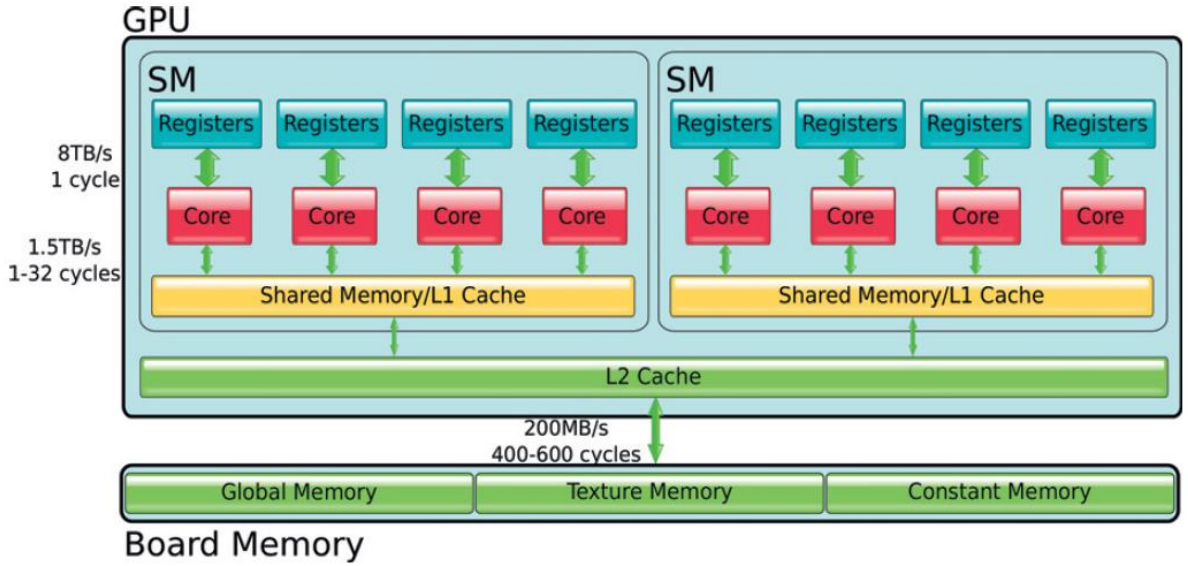


Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
__device__ __shared__ `int SharedVar;`	Shared	Block	Kernel
__device__ `int GlobalVar;`	Global	Grid	Application
__device__ __constant__ `int ConstVar;`	Constant	Grid	Application

Registers

- The number of maximum registers per thread influences the maximum number of resident threads we can have on an SM
- E.g.,
  - A kernel is using 48 registers
  - It is invoked as blocks of 256 threads
  - Each block requires 48*256=12,288 registers
  - Let's assume the target GPU has 32,000 registers per SM
  - Let's assume the target GPU can have up to 1,536 resident threads per SM
  - Thus, each SM could have 2 resident blocks (12,288*2 < 32,000)
  - I.e., each SM could have 2*256 = 512 resident threads
  - This is much below that the maximum limit of 1,536 threads
  - This undermines the possibility to hide latency
- NVIDIA defins as *occupancy* the ratio of resident warps over the maximum possible resident warps

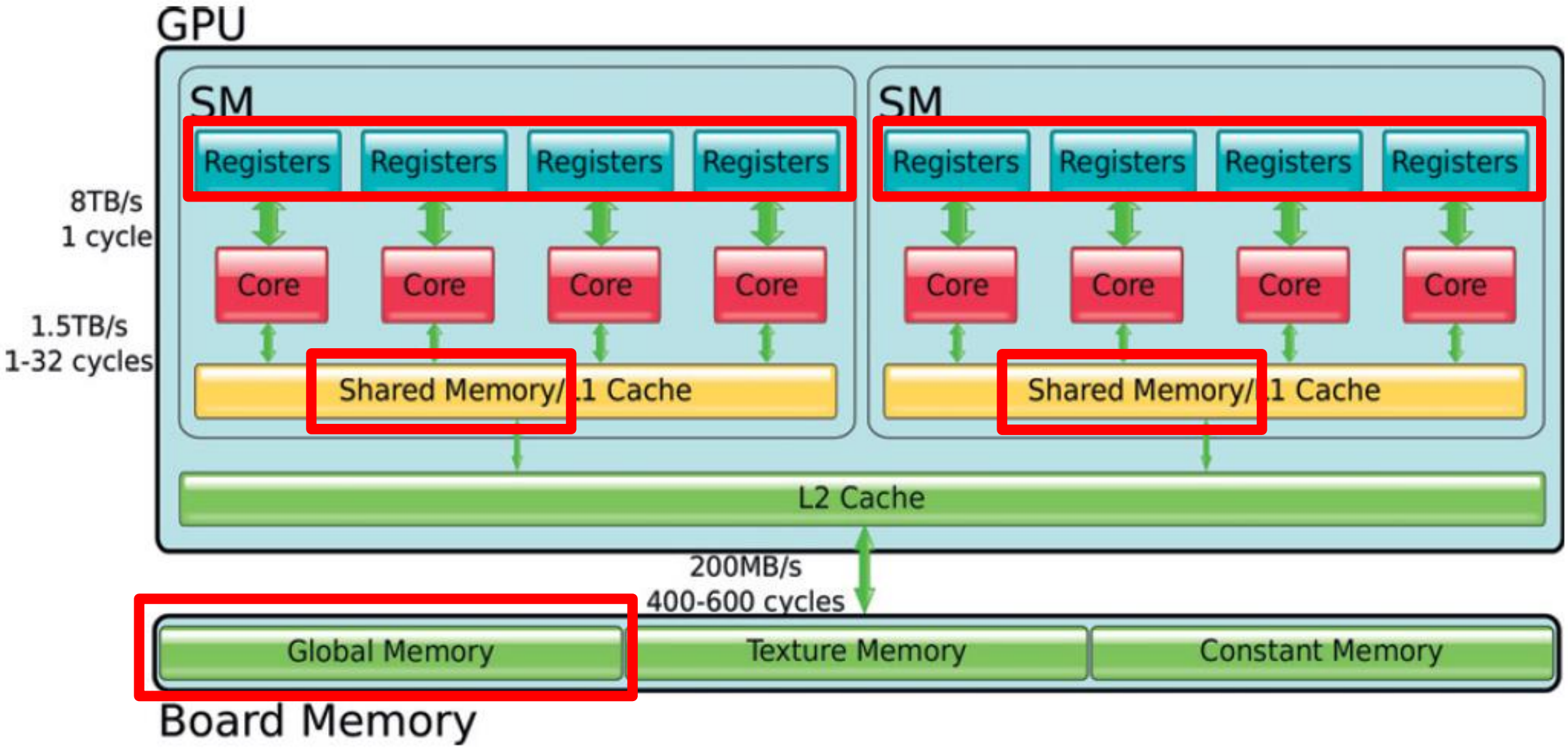
$$occupancy = \frac{resident\_warps}{maximum\_warps} \qquad \frac{2 \cdot \frac{256 \text{ threads}}{32 \text{ threads/warp}}}{48 \text{ warps}} = \frac{16}{48} = 33.3\%$$

Shared Memory (on-chip)

- On-chip memory
  - Different from registers. Registers data is private to threads, shared memory is shared among threads
- Can be seen as a user-managed L1 cache (scratchpad)
- Can be used as:
  - A place to hold frequently used data that would otherwise require a global memory access
  - As a way for cores on the same SM to share data
- The __shared__ specifier can be used to indicate that some data must go in the shared on-chip memory rather than on the global memory
- Shared memory vs. L1 cache:
  - Both are on-chip. The former is managed by the programmer the latter automatically
  - In some cases, managing it manually (i.e., using the shared memory), might provide better performance
  - E.g., you do not have any guarantee that the data you need will be in the L1 cache, but with the explicitely managed shared memory, you can control that

Questions?

Memory Types



More on Memory

Shared Memory Example: 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

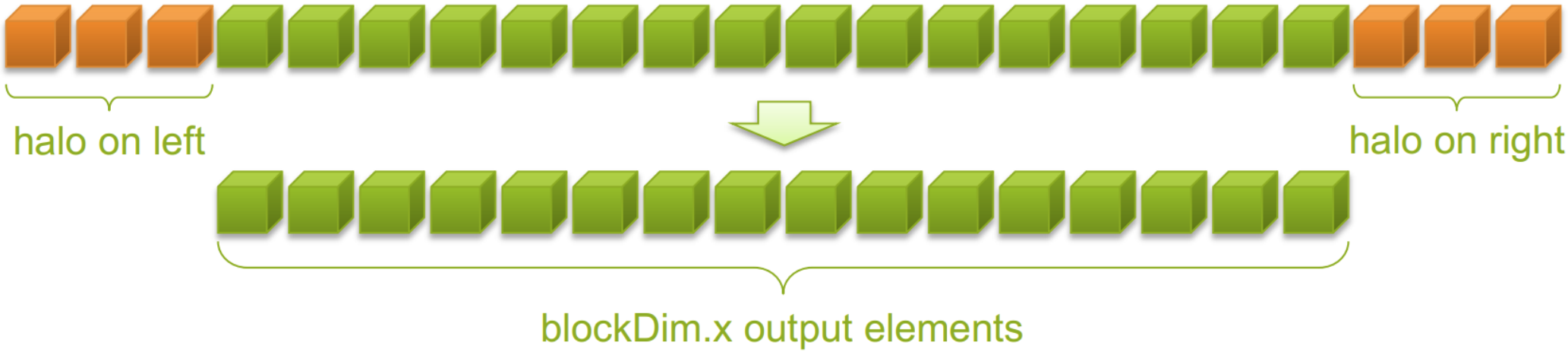
- Each thread processes one output element
  - **`blockDim.x`** elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

- Cache data in shared memory
  - Read (**`blockDim.x`** + 2 * radius) input elements from global memory to shared memory
  - Compute **`blockDim.x`** output elements
  - Write **`blockDim.x`** output elements to global memory
- Each block needs a halo of **`radius`** elements at each boundary



https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

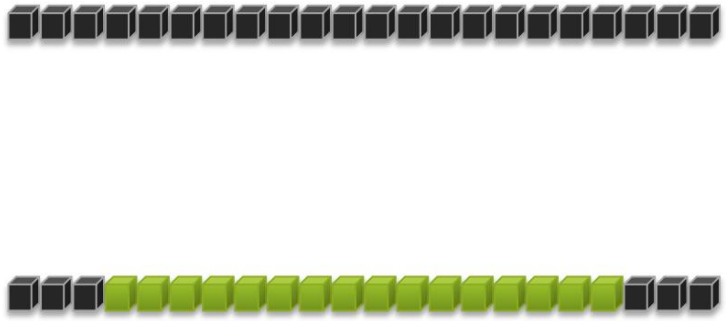
Shared Memory Example: 1D Stencil

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
```

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

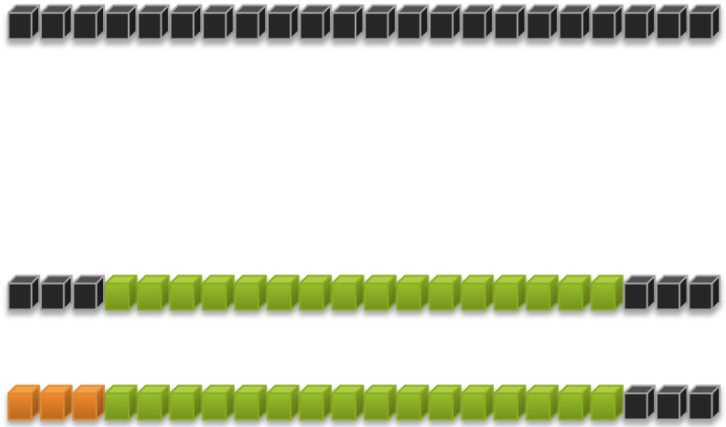
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
```



https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

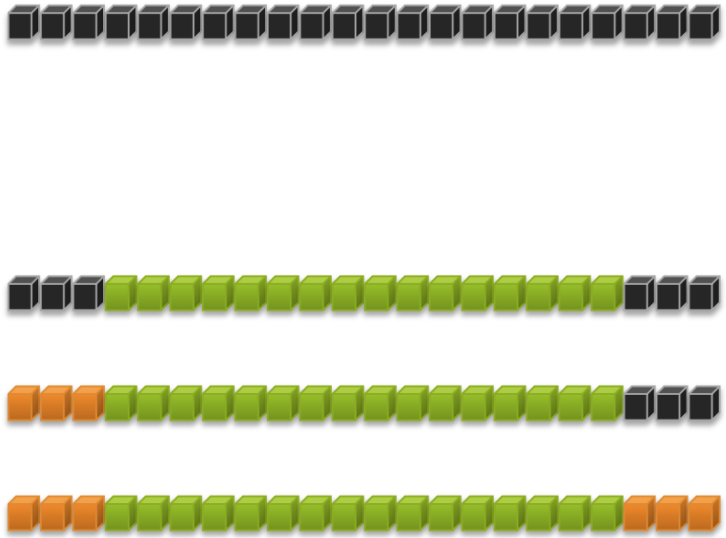
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
```



https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

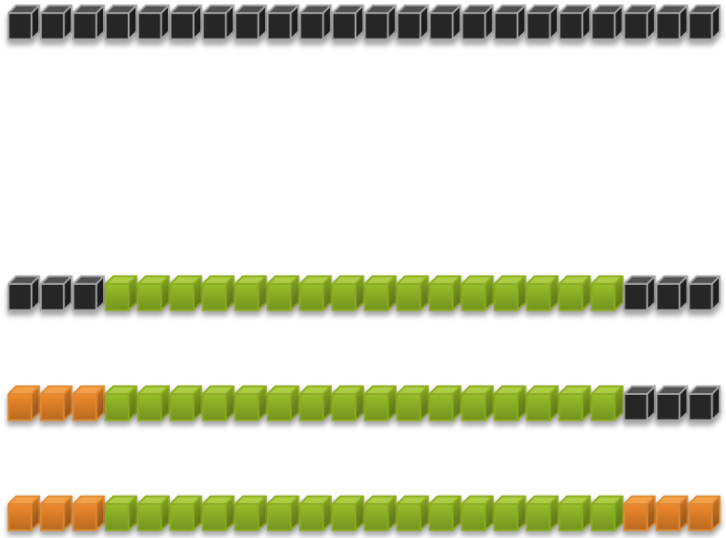
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
   // Apply the stencil
  int result = 0;
   for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
     result += temp[lindex + offset];

   // Store the result
   out[gindex] = result;
}
```



What if the first warp of the block gets executed, and thread 31 starts computing the result before thread 32 copies in temp the element in position 32?

We need to synchronize the threads, to guarantee that the stencil is applied only after the data has been loaded in the shared memory

https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

Shared Memory Example: 1D Stencil

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier

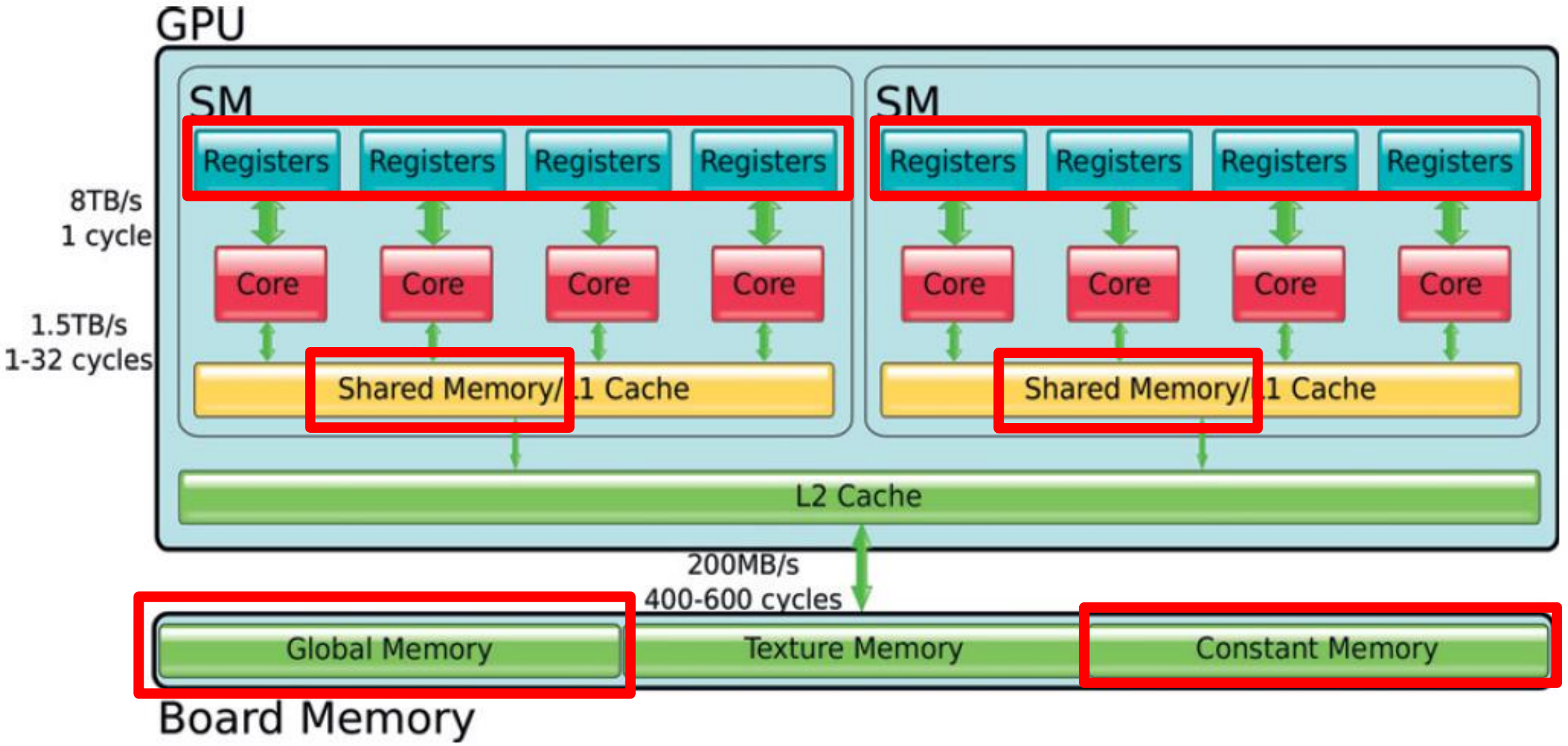
Shared Memory Example: 1D Stencil (Correct)

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

We allocated this statically (the size is known at compile time)
Wee can also do it dynamically (you can find more info on the Barlas book)

Questions?

Memory Types



Constant Memory

Shared Memory (on-chip)

- Constant memory != ROM
  - It is just a memory that can hold constant values (i.e., the host can write it)
- Two main advantages
  - It is cached
  - Supports broadcasting of a single value to all threads in a warp
- E.g., suppose to have 10 warps on an SM, and all request the same variable
  - If data is on global memory:
    - all warp will request the same segment from global memory
    - the first time segment is copied into L2 cache
    - if other data pass through L2, there are good chances it will be lost
    - there are good chances that data should be requested multiple times
  - ff data is in constant memory:
    - during first warp request, data is copied in constant-cache
    - since there is less traffic in constant-cache , there are good chances all other warp will find the data already in cache

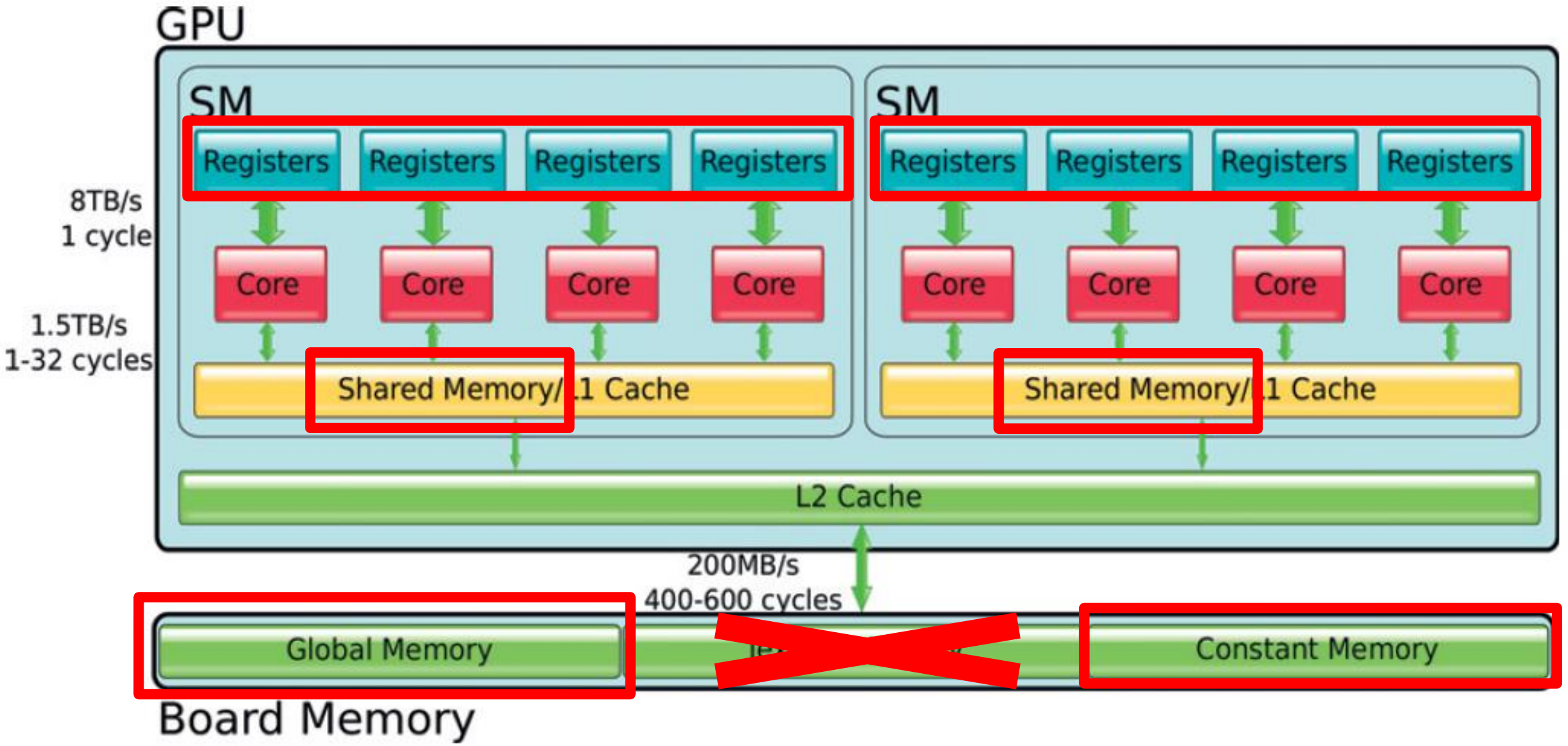
Shared Memory (on-chip)

```
__constant__ type variable_name; // static
cudaMemcpyToSymbol(variable_name, &host_src, sizeof(type), cudaMemcpyHostToDevice);
// warning: cannot be dynamically allocated
```

- data will reside in the constant memory address space
- has static storage duration (persists until the application ends)
- readable from all threads of a kernel

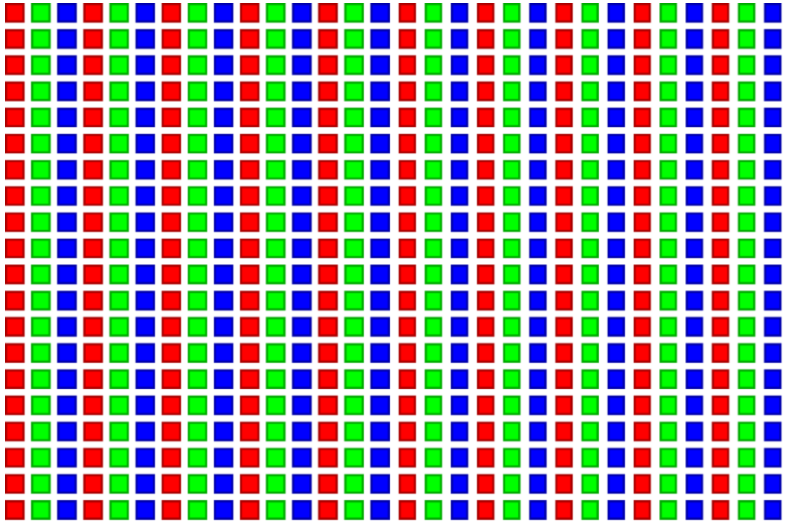
Questions?

Memory Types



Example: Grayscale Image

Image to Grayscale



An image can be seen as a 2D matrix of pixels, each pixel has 3 values for the 3 RGB channels
Number of columns = 3 x Number of pixels in a row

To convert the pixel to grayscale:

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

Dynamically allocated 2D matrix linearized

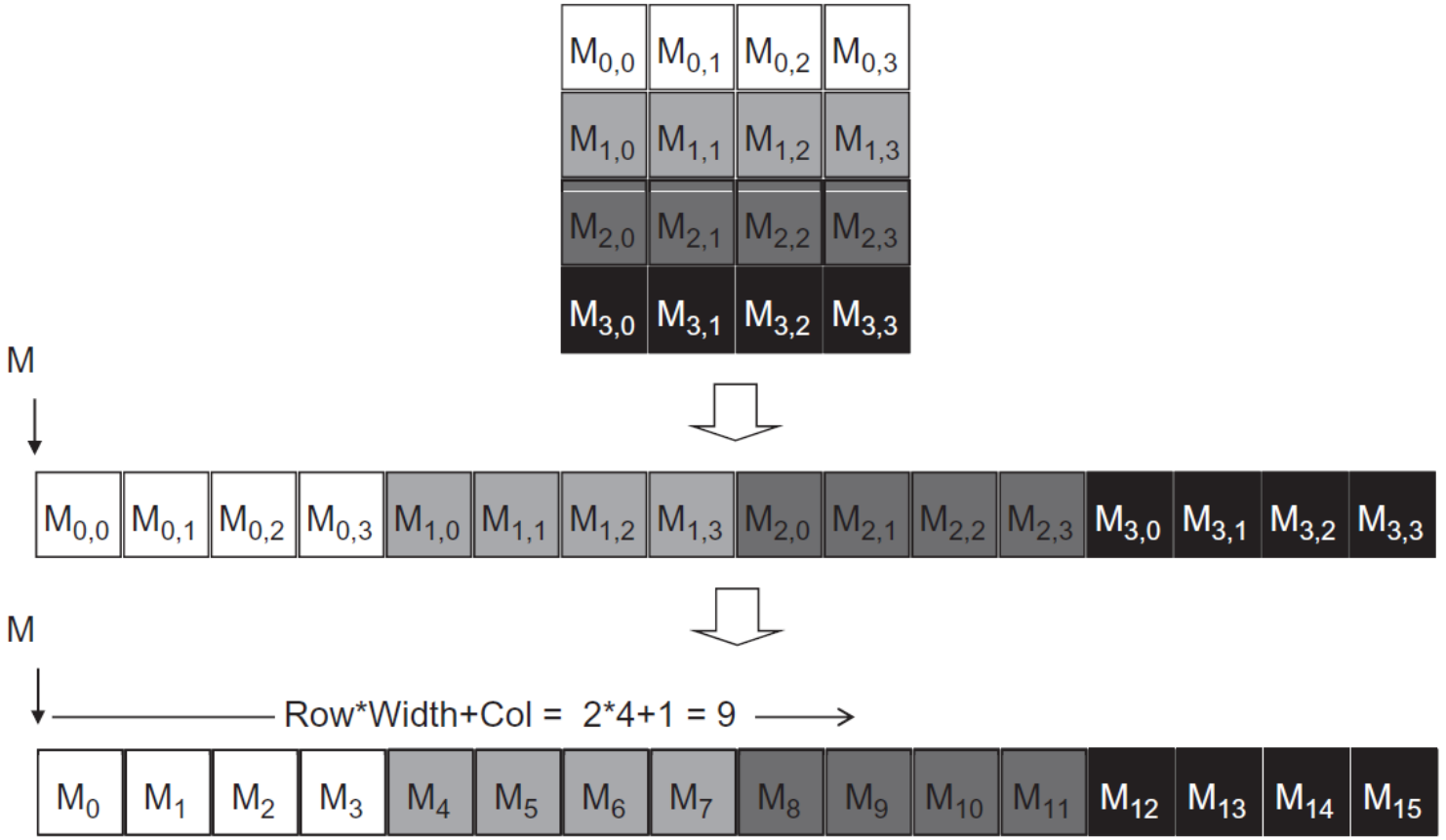
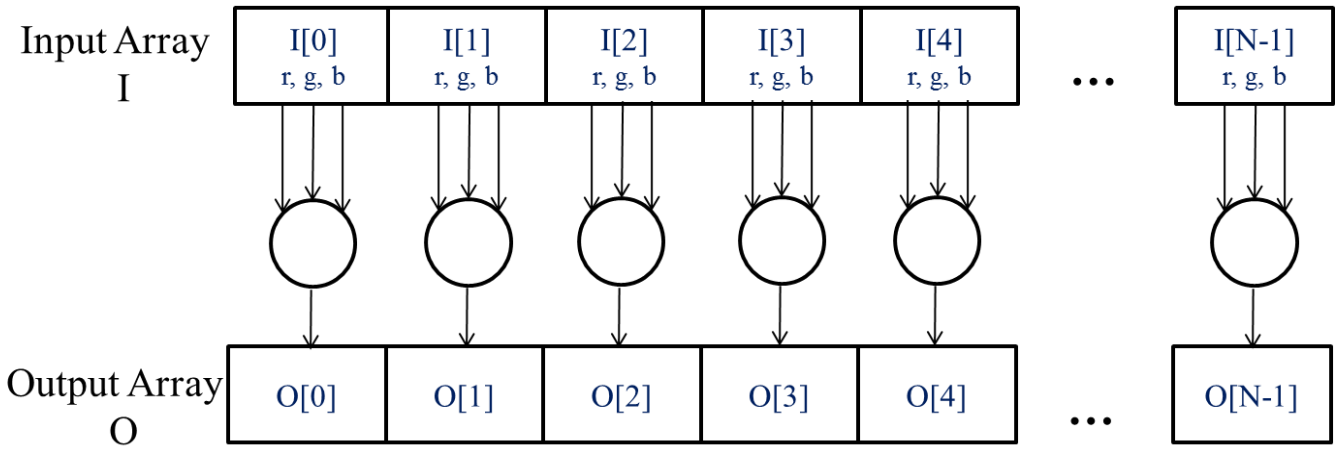


FIGURE 3.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression *j**Width+ *i* for an element that is in the *j* th row and *i* th column of an array of Width elements in each row.

The pixels can be calculated independently of each other



Block Size

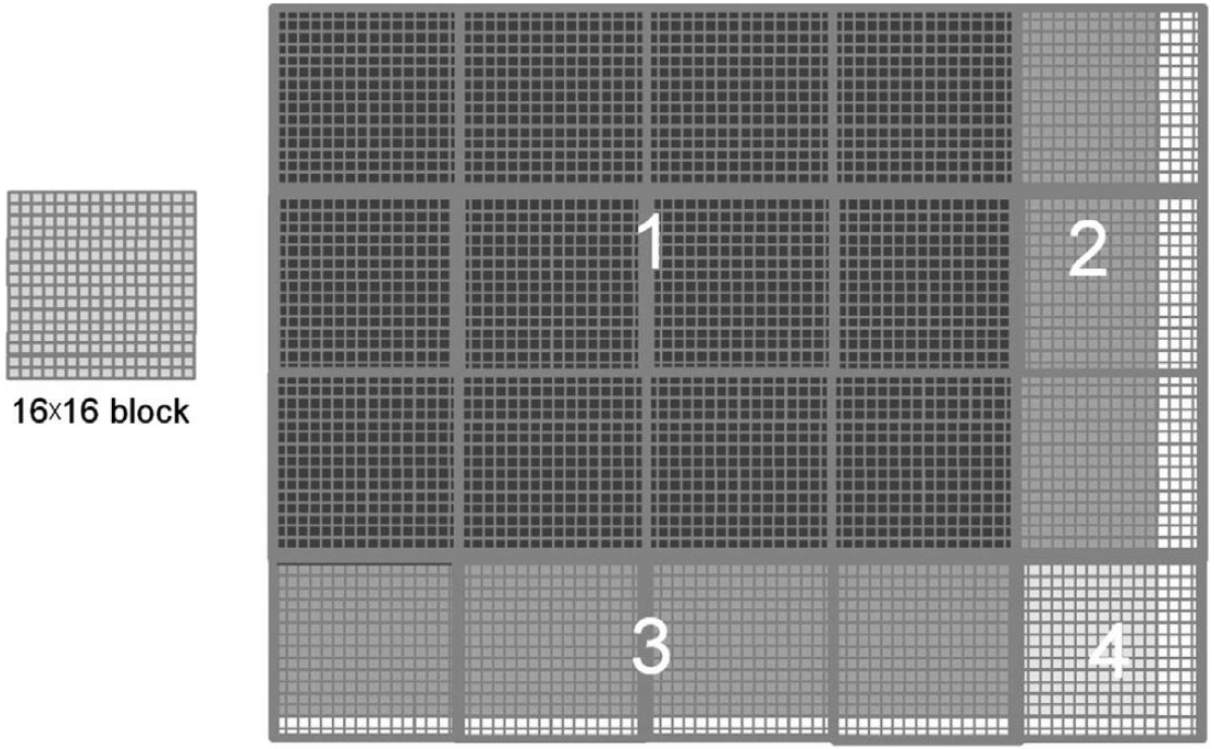


FIGURE 3.5

Covering a 76 × 62 picture with 16 × 16 blocks.

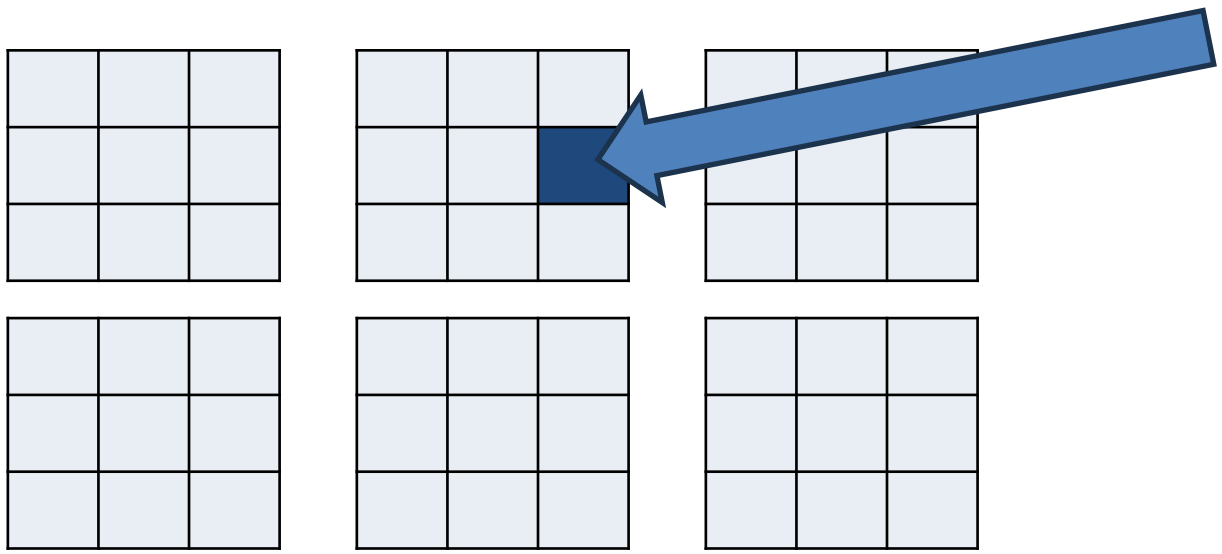
ATTENTION: Some threads in some blocks (those in the 2,3,4 areas) do not have any pixed to process. Each thread should check if it has a pixel to process or not.

Kernel Code

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

Kernel Code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(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;
```



Block (1, 0)
Thread (2,1)

blockIdx.x = 1
blockIdx.y = 0
threadIdx.x = 2
threadIdx.y = 1
blockDim.x = 3
blockDim.y = 3

Col = 2 + 1*3 = 5
Row = 1 + 0*3 = 1

Kernel Code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(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;
```

Kernel Code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(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 than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
```

Kernel Code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(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 than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
```

Kernel Code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(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 than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

Questions?

Example: Image Blur

Image Blur (Simplified)

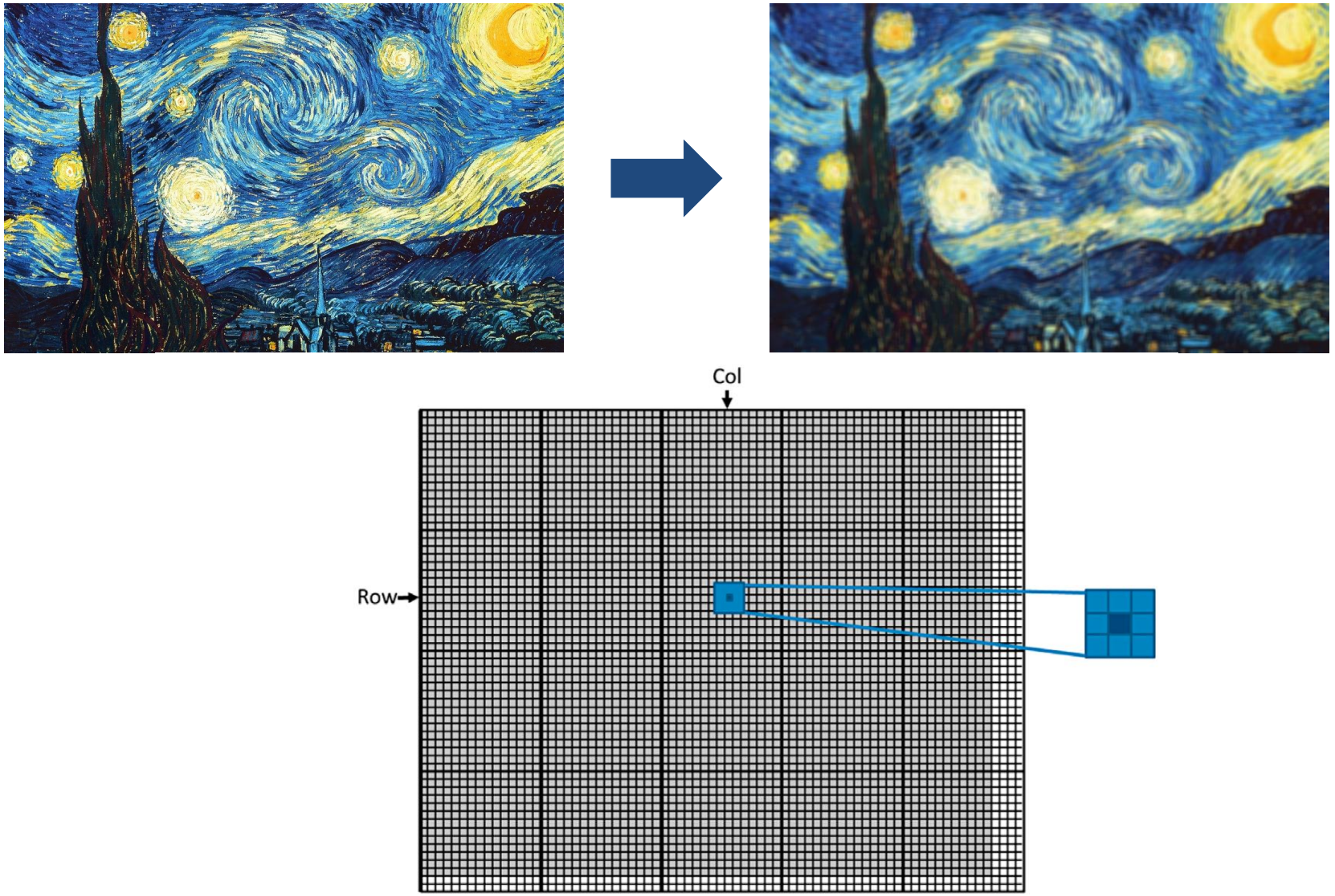


FIGURE 3.7

Each output pixel is the average of a patch of pixels in the input image.

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) {
```

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)
     {
```

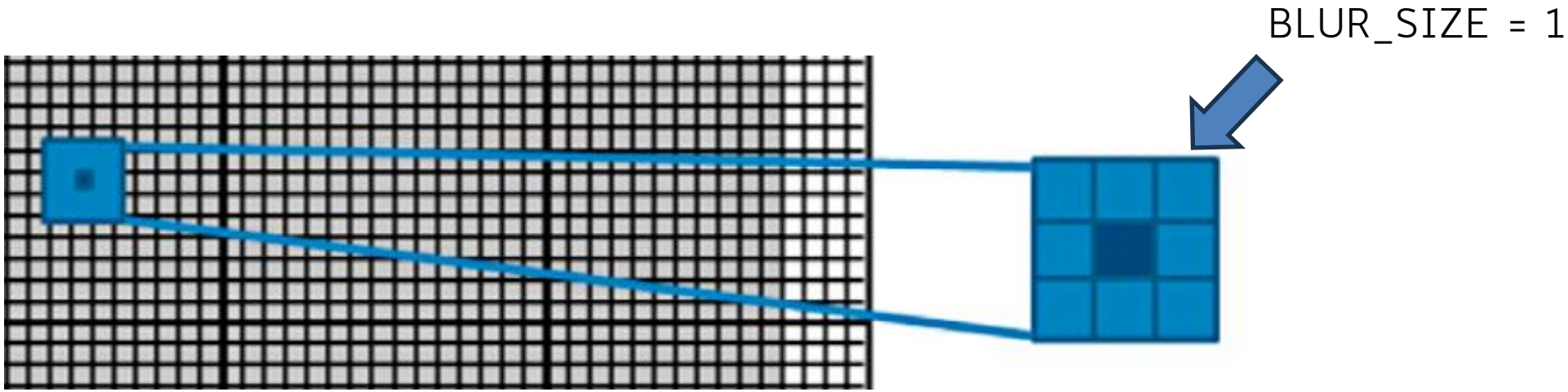
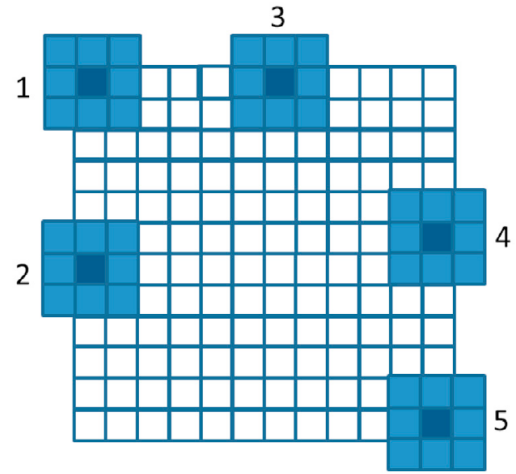


Image Blur Kernel

```
 __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;
```

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
          }
        }
      }
```

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);
  }
}
```

Performance Estimation

- How can we measure performance to have an idea of whether we are saturating the computational capabilities of the hardware?
  - FLOP/s (i.e., floating-point operations per second)
    - What type of floating-point? 64-bit, 32-bit, 16-bit, …? (This must be specified!)
  - Today we have systems capable of 1 ExaFLOP/s (i.e., 10^{18} FLOP/s)
  - You can check top500.org to have a (partial) view
  - FLOP/s vs FLOPs/s vs FLOPs ….. (religious battle)
    - But is "floating point operations" FLOP or FLOPs?
    - We will use:
      - FLOP ("floating-point operations")
      - FLOP/s ("floating-point operations per seconds")
    - But be aware that different books/resources might denote it differently

Performance Estimation

```
pixVal += in[curRow * w + curCol];
```

- All threads access global memory for their input matrix elements
- Let's suppose that the global memory bandwidth is 200 GB/s
  - How many operands can we load? (200 GB/s) / (4 bytes) = 50G operands / s
- We do one floating-point operation (+=) on each operand
  - Thus, we can expect, in the best case, a peak performance of 50 GFLOP/s
- Let's suppose that the peak floating-point rate of this GPU is 1,500 GFLOP/s
- This limits the execution rate to **3.3%** (50/1500) of the peak floating-point execution rate of the device!
- I.e., the memory movement to/from the memory (rather than the compute capacity) is limiting our performance
- We say that this application is ***memory bound***
- Need to drastically cut down memory accesses to get close to the 1,500 GFLOP/s

Performance Estimation

- We define the **compute-to-global-memory-access ratio** as the number of floating-point calculation performed for each access to the global memory within a region of a program.
  - Also known as arithmetic/operational intensity (measured in FLOP/byte)

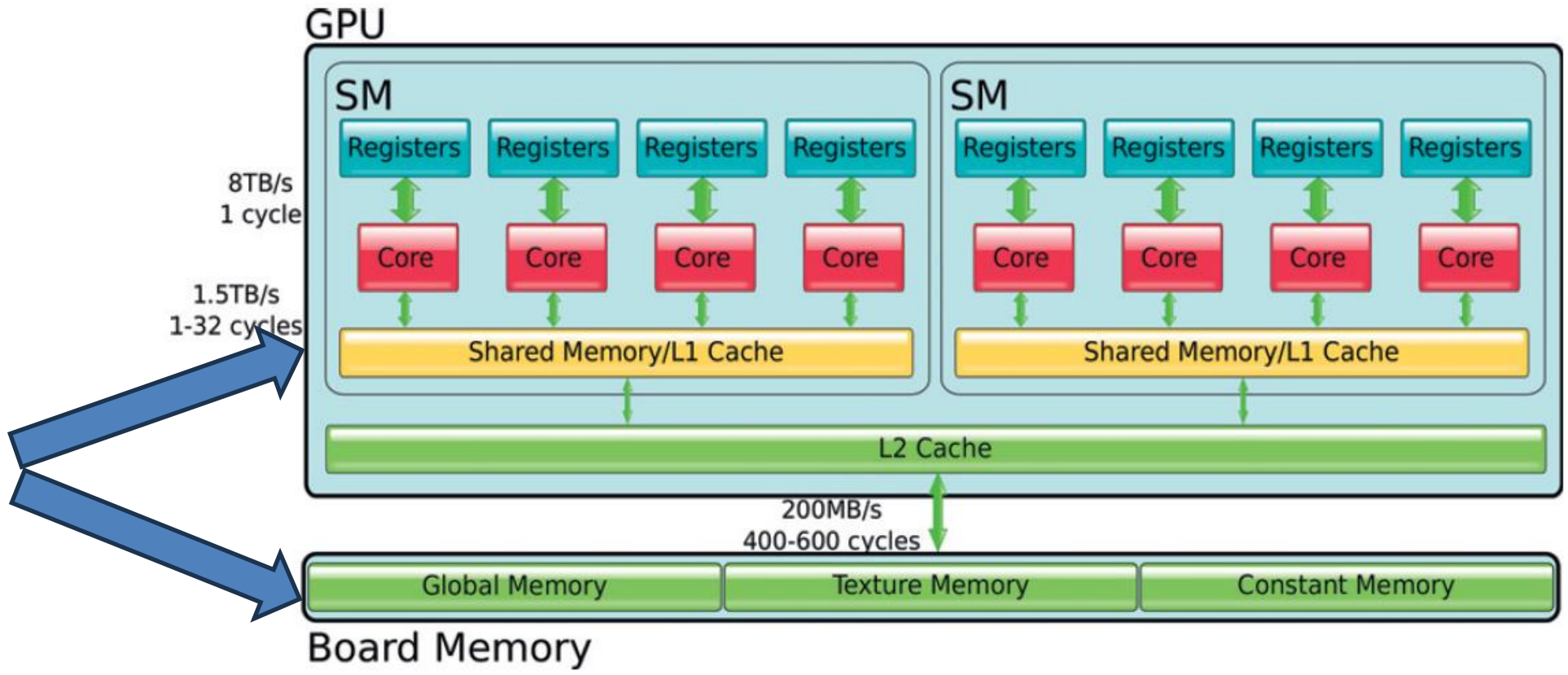
- We can load at most 50 G operands / s

- To achieve the peak 1.5 TFLOP/s rating of the processor, we need a ratio of 30 or higher 1.5 T / 50 G
  - i.e., we would need to perform 30 floating-point operations on every operand

- The technological trend is not encouraging: the computational throughput grows at a faster rate than the memory bandwidth

Memory Types

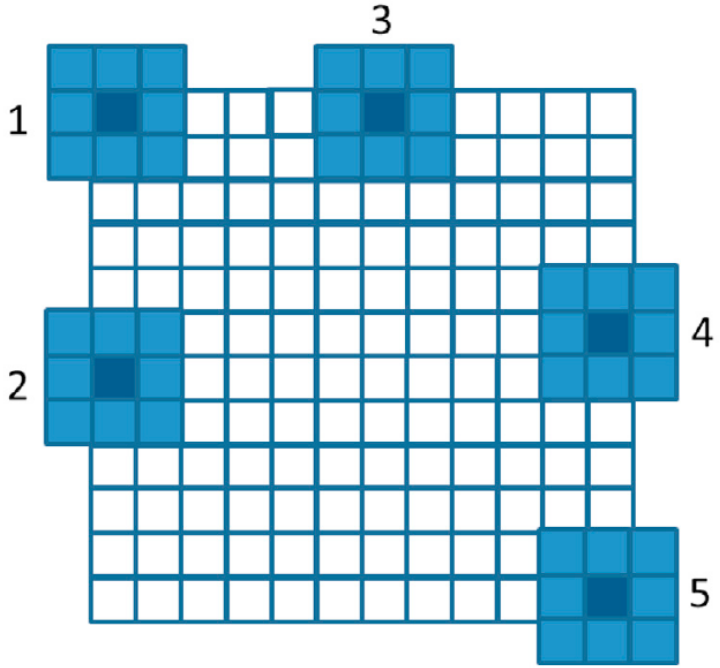
Shared memory throughput 7500 times higher than global memory



Exercise @ Home

Modify the image blur code to use shared memory

- Each thread in a block loads a pixel («its» pixel) into shared memory
- Each thread runs the blur of «its» pixel (as before)
- Pay attention to the border («halo» cells) – Shared memory can only be accessed by the threads running on the same SM



Questions?

Roofline Model

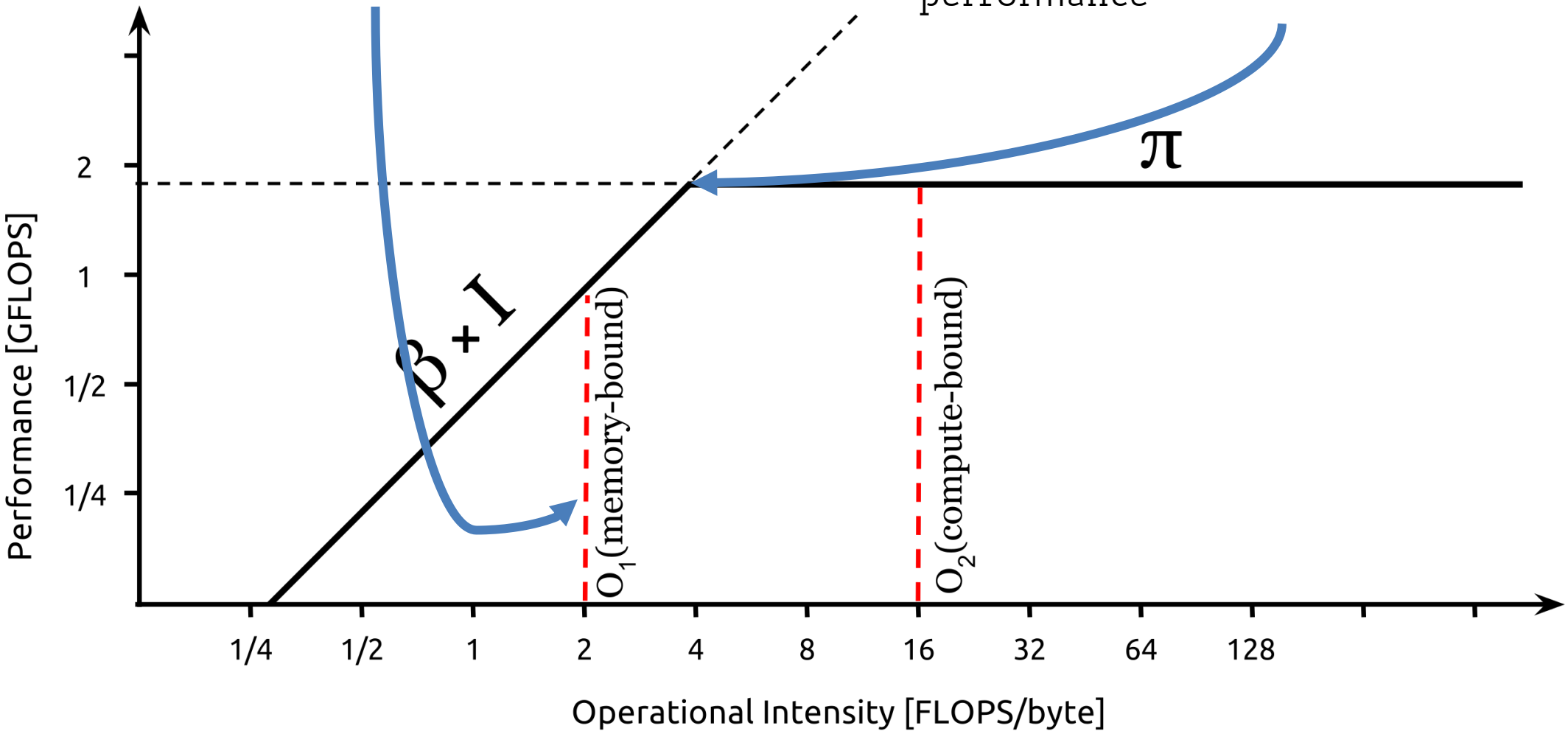
https://docs.nersc.gov/tools/performance/roofline/

https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf

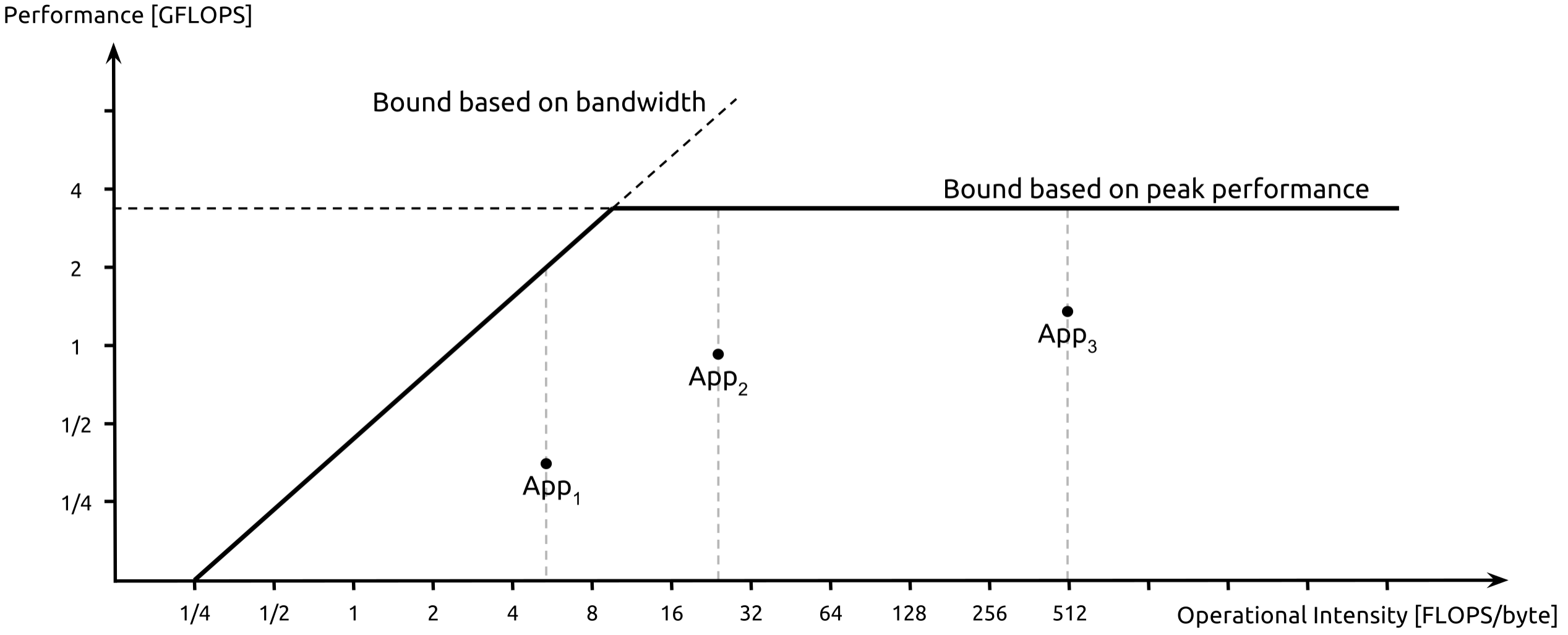
Roofline model

beta = 0.5 GB/s (mem. Bandwidth)
With an op. intensity of 2 flop/byte, in the best case, I can expect a peak performance of 1 GFLOP/s

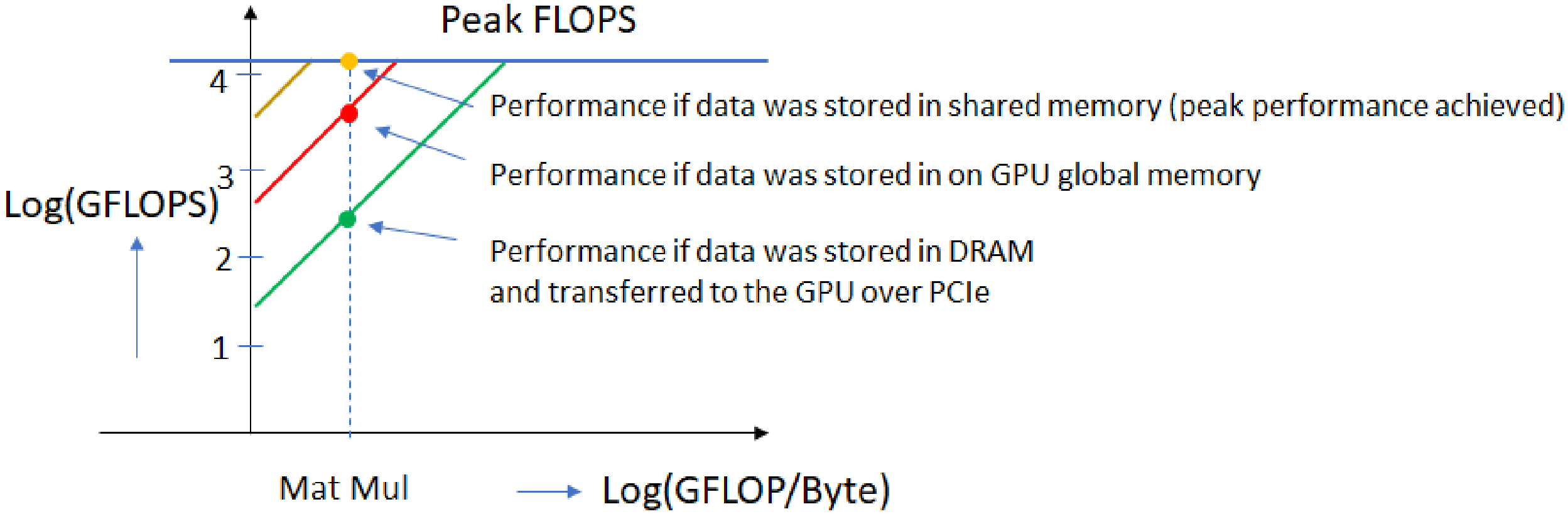
Ridge point
I need to perform at least 4 FLOP for each byte I load in order to get peak performance



Roofline model



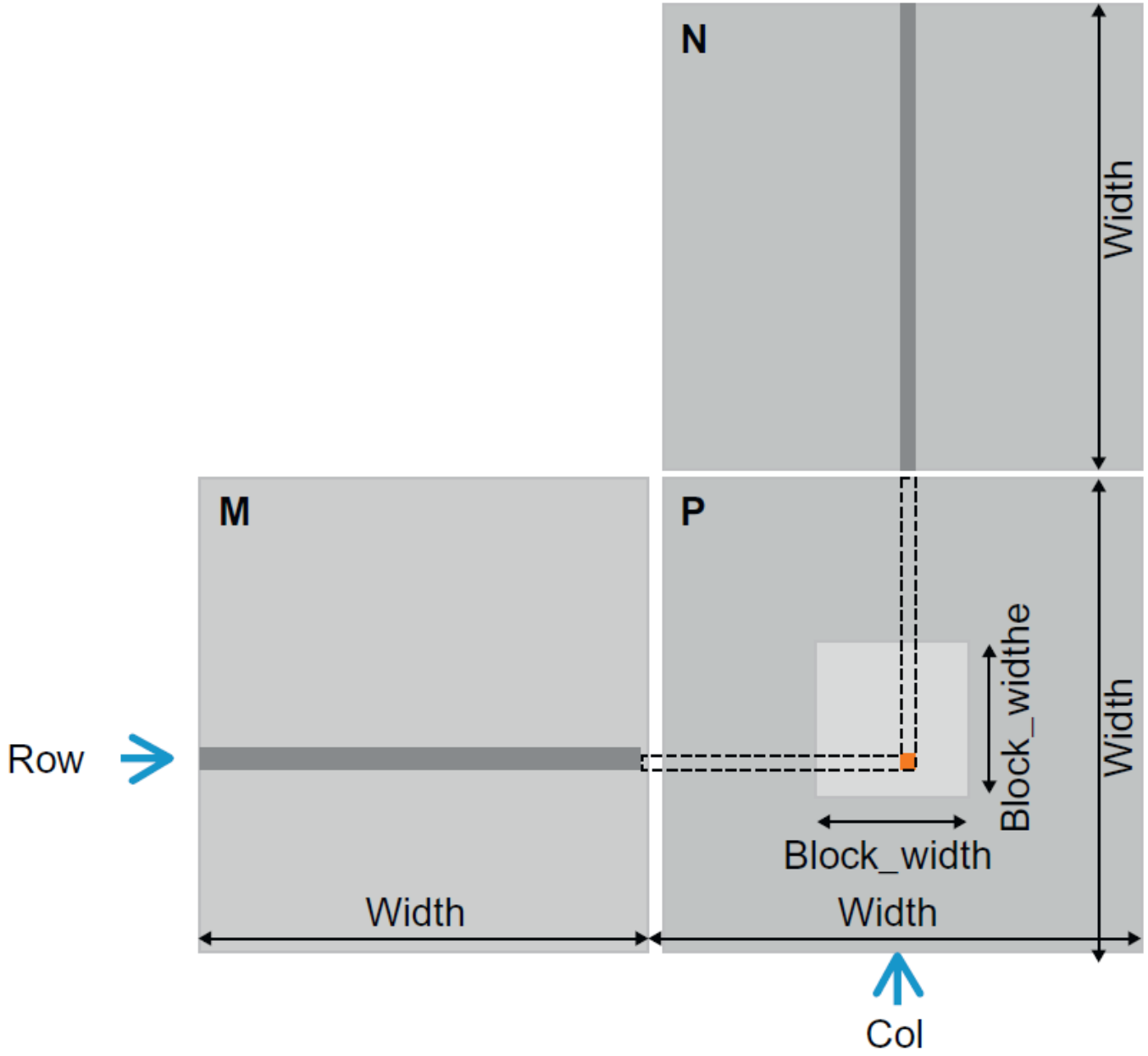
Roofline model



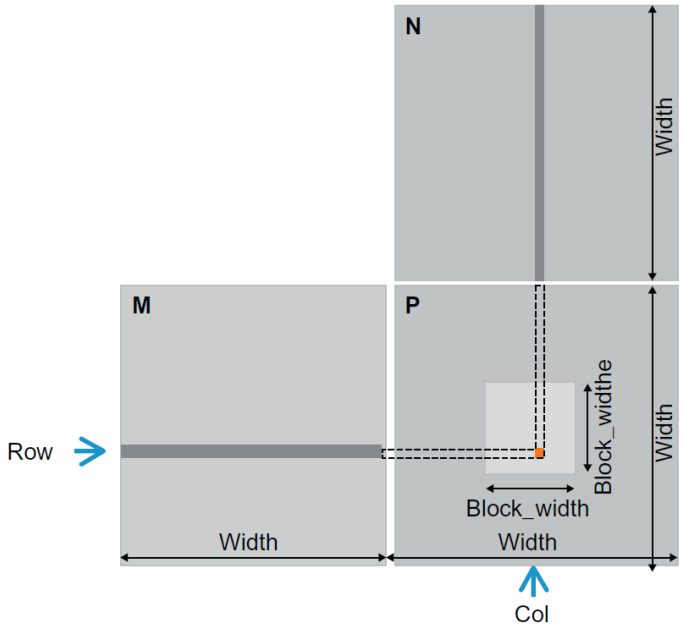
Questions?

Example:
Matrix Multiplication with Tiling

Matrix Multiplication

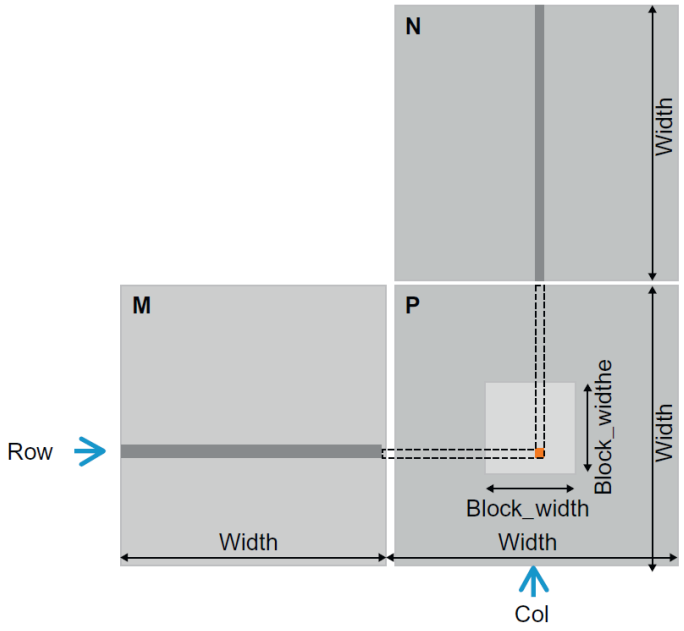


Matrix Multiplication



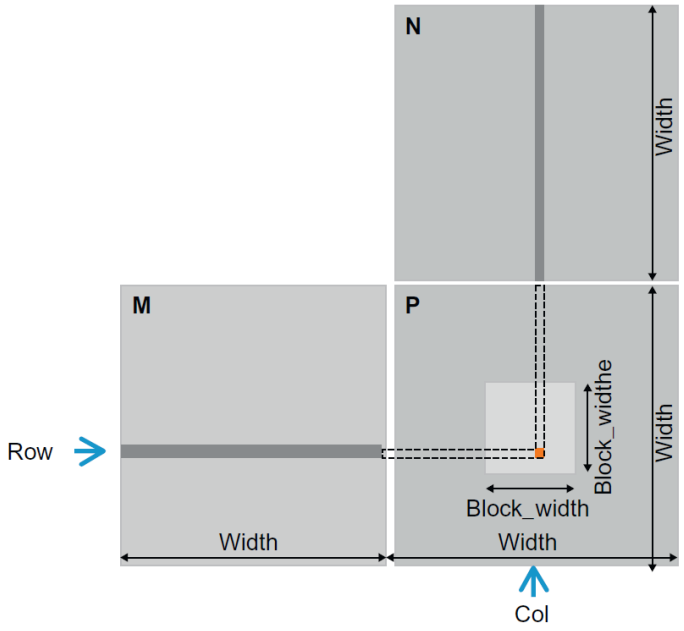

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
```

Matrix Multiplication



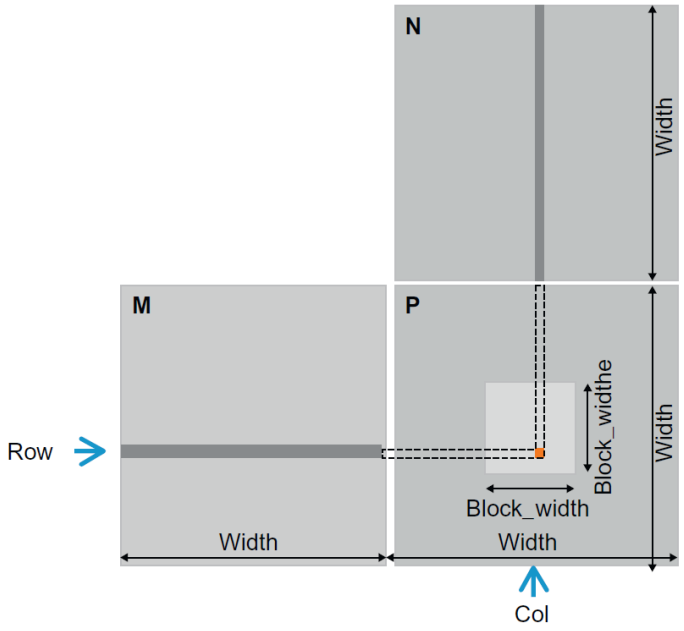

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
```

Matrix Multiplication




```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
```

Matrix Multiplication




```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

Matrix Multiplication

Two global memory accessess (one element from M and one from N)

One multiplication and one addition

Arithmetic intensity (FLOP/operand) = 1

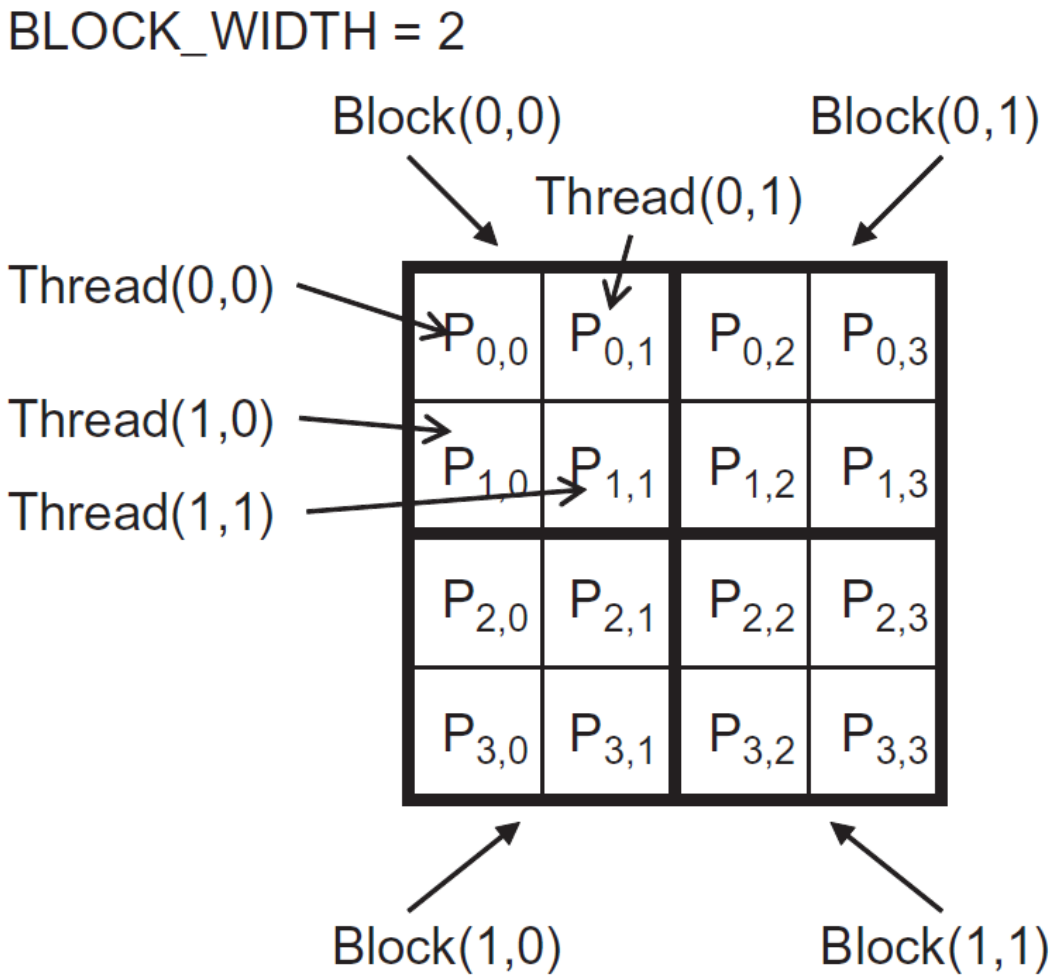
Arithmetic intensity (FLOP/byte) = 8

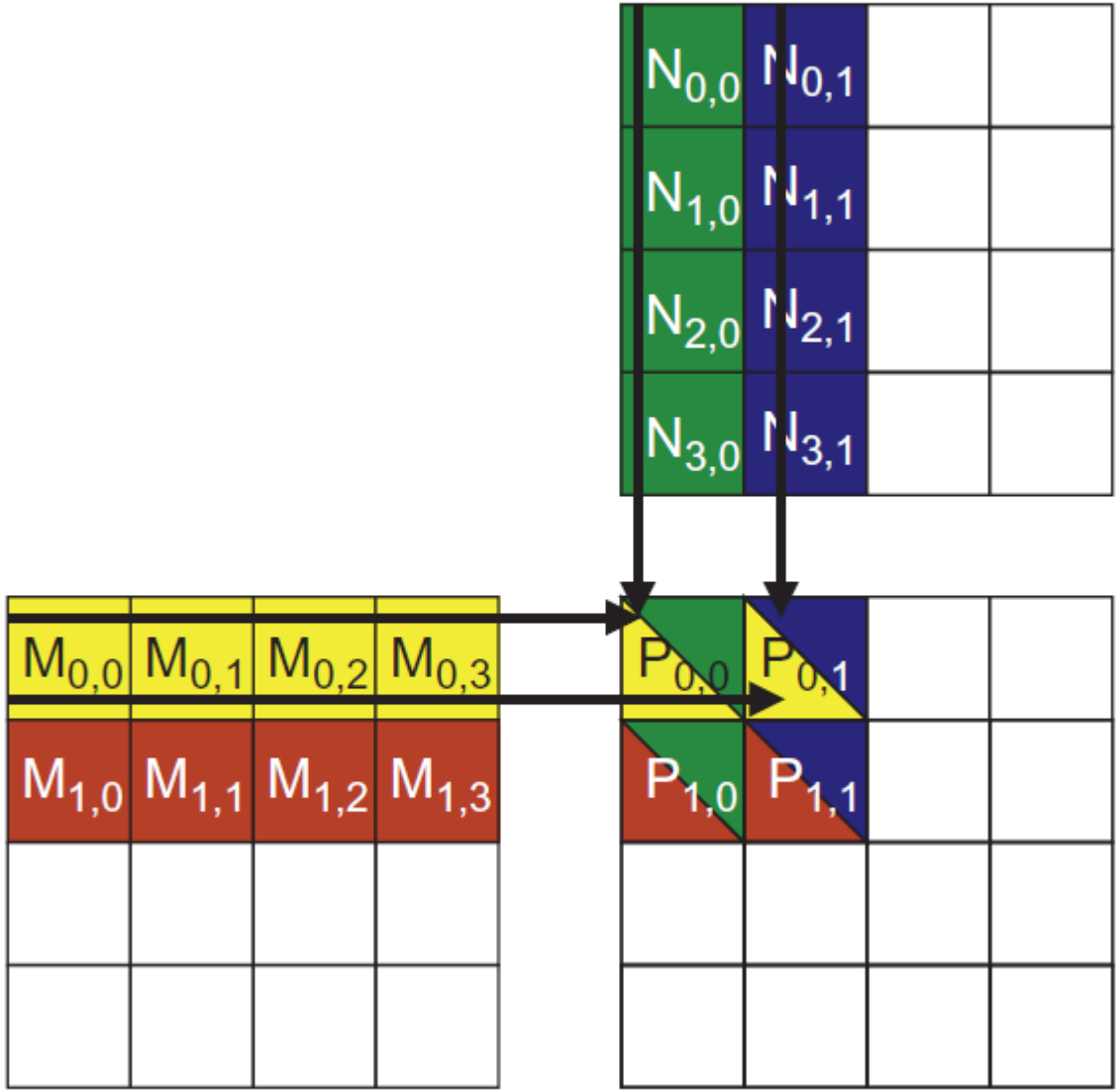
How to increase it? Let's try to exploit shared on-chip memory

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

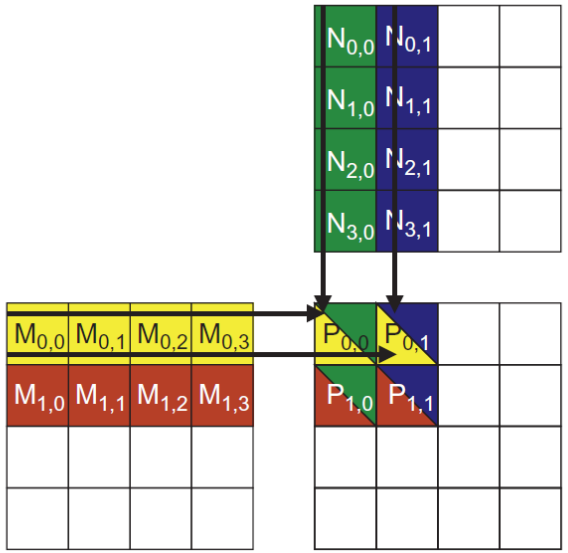
Questions?

Matrix Multiplication





Matrix Multiplication

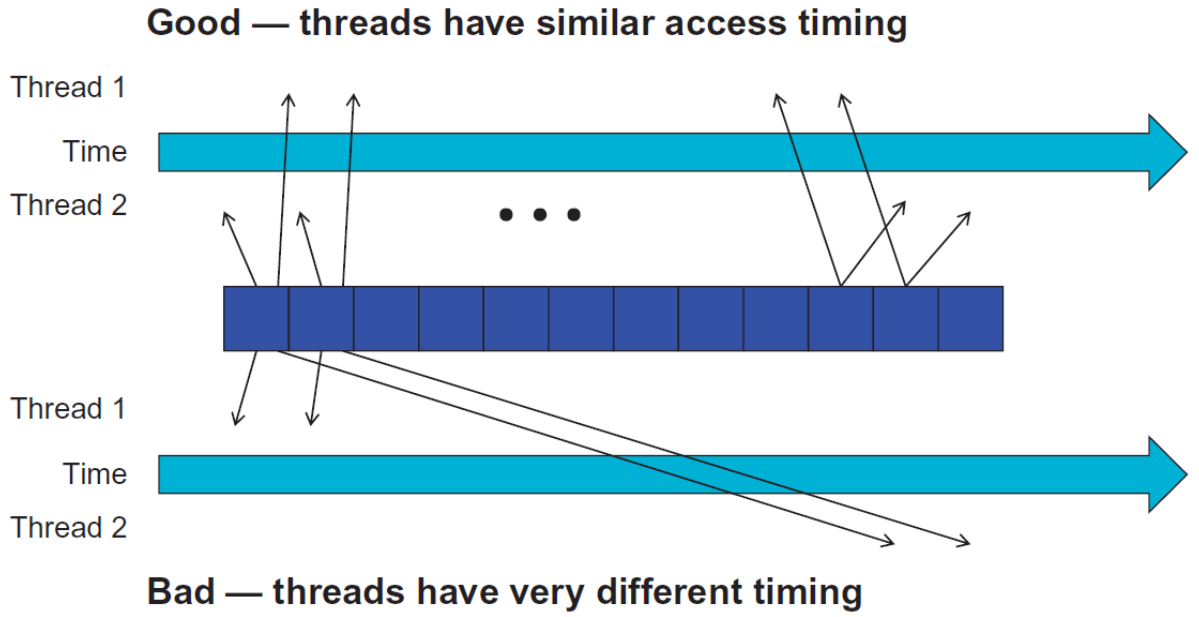


Access order

thread$_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread$_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread$_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread$_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

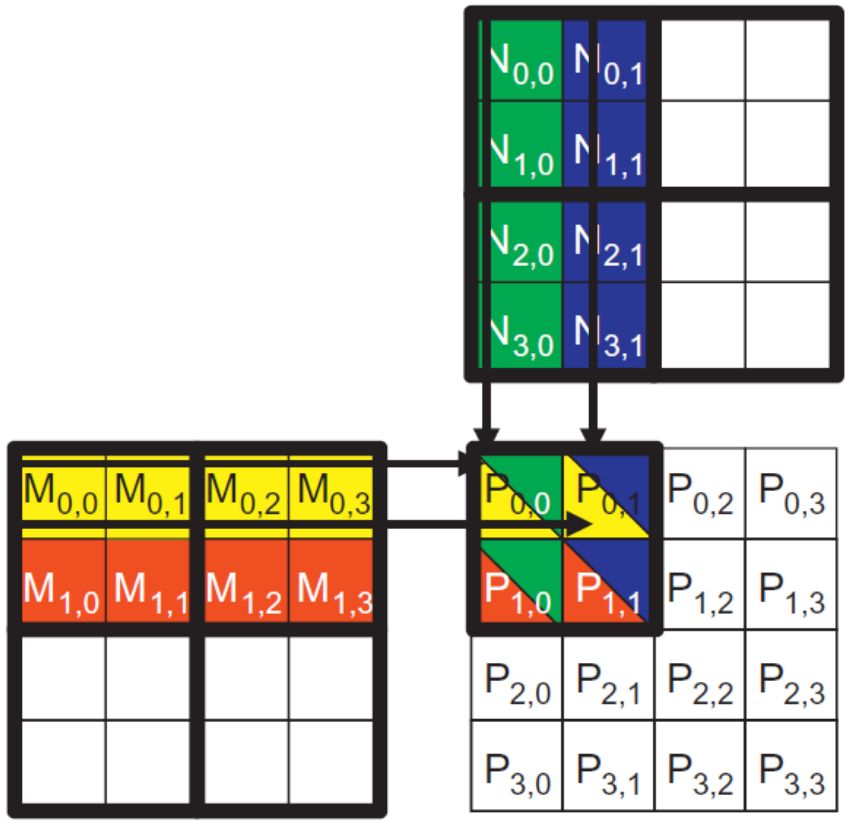
- Each element is accesses by 2 threads at the same time
- I.e., each element is loaded twice from global memory
- We could have one thread loading it from global to shared memory, and the other one reaading it from shared memory, reducing traffic to global memory by 2x
- **IMPORTANT:** The potential for memory traffic reduction depends on the block size (i.e., for 16x16 blocks we could reduce it by 16x)

Matrix Multiplication



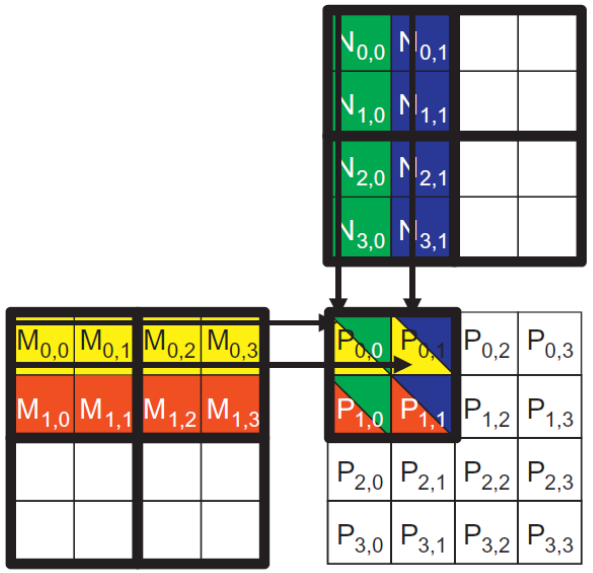
- Shared memory is small, it is important that once we load a value, that value is going to be accessed again soon
- If accesses are too distant in time, then we might have loaded something different in the meanwhile

Tiling



- We also divide the input matrices in tiles (of the same size of the tiles of the output matrix)
- For tile in (0, 2)
  - Each thread now loads one element from the first tile of M and one from the first tile of N from global to shared memory
    - E.g., P_{00} loads N_{00} and M_{00}
  - Sync (barrier)
  - Each thread increments the partial value of the output (using the sub-rows and sub-cols they loaded)

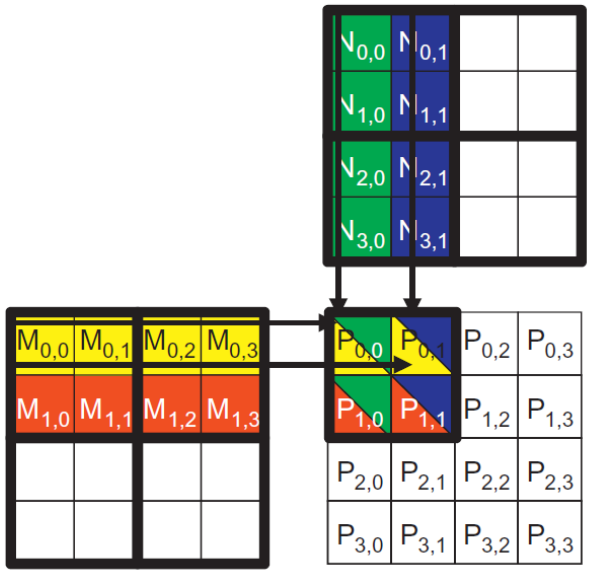
Tiling



	Phase 1		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time

Tiling



	Phase 1			Phase 2		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

- **Without tiling:** 4 threads, each loads one row (4 loads) and one column (4 loads) = 4x(4+4) = 32 accesses to global memory
- **With tiling:** 4 threads: each loads 4 elements from global memory = 16 accesses to global memory **(2x reduction in global accesses)**

Questions?

Matrix multiplication with tiling: code

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

     // Identify the row and column of the d_P element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;
```

Matrix multiplication with tiling: code

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

     // Identify the row and column of the d_P element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
```

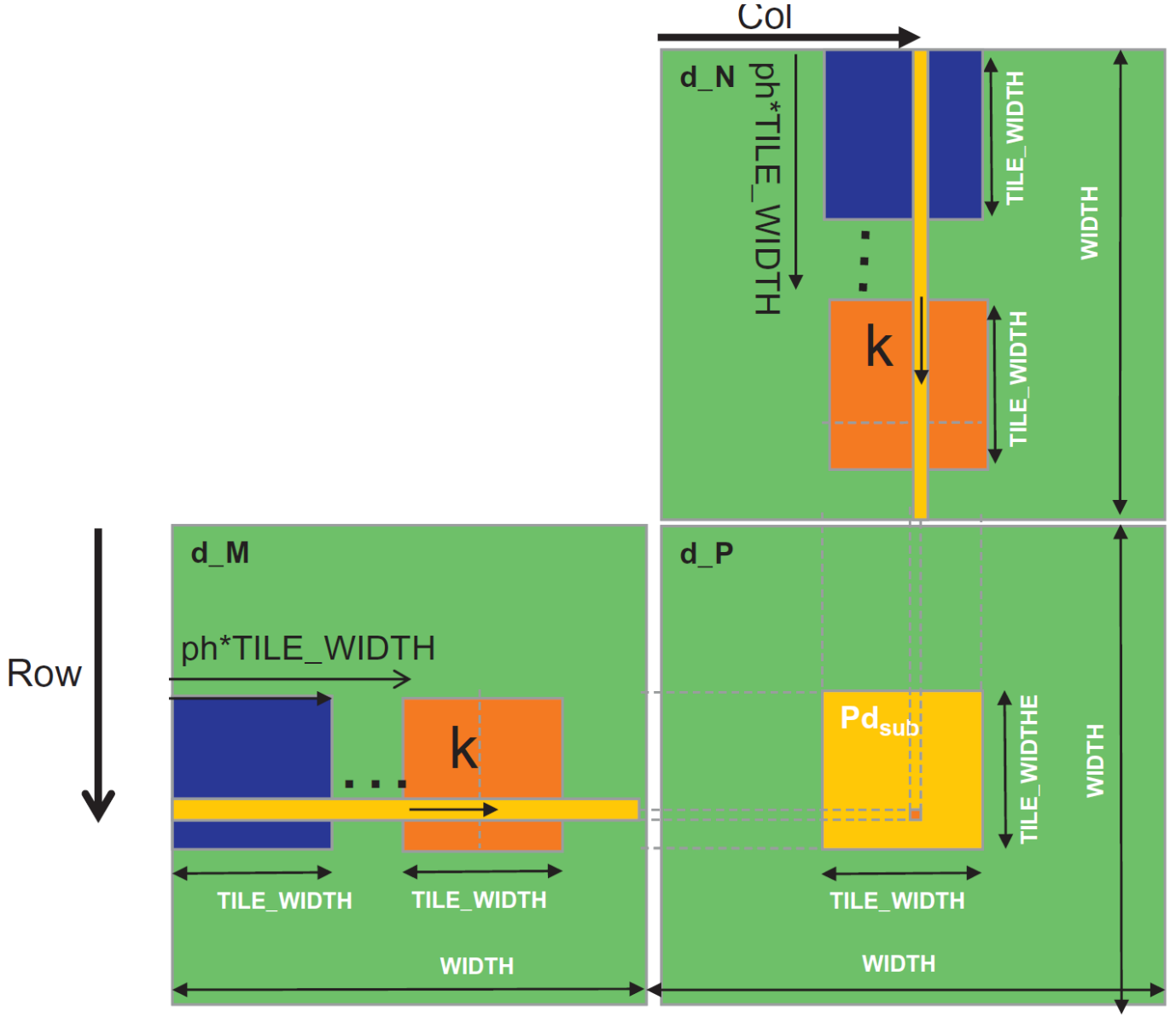
Matrix multiplication with tiling: code

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
8.  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

      // Collaborative loading of d_M and d_N tiles into shared memory
9.    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.   Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();
```

Matrix multiplication with tiling: code

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
8.  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

      // Collaborative loading of d_M and d_N tiles into shared memory
9.    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.   Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k) {
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
      }
14.   __syncthreads();
    }
15. d_P[Row*Width + Col] = Pvalue;
  }
```

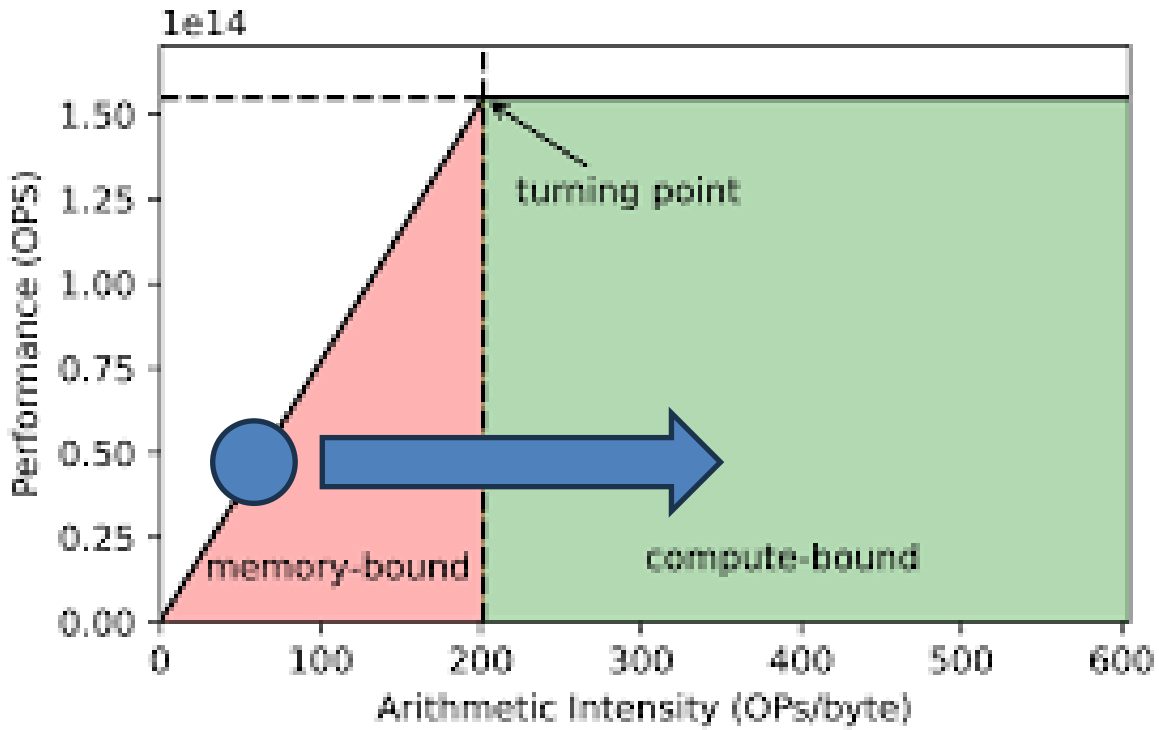
Calculation of tile index



How many blocks etc?

```
for (int k = 0; k < TILE_WIDTH; ++k) {
  Pvalue += Mds[ty][k] * Nds[k][tx];
}
```

- **Each block works on a tile, so TILE_WIDTH == BLOCK_WIDTH**
- For 16x16 tiles, in each phase, the inner loop is executed 16 times. Before the loop each thread loaded two elements from global memory (one from M and one from N). In the loop we do 2 FLOP per iteration. Thus, 2 loads and 32 FLOP, we have an arithmetic intensity of 16
- For 32x32 tiles, it would be 32



How many blocks etc?

- A GPU has a fixed size for its shared memory
- E.g., let's assume it has a 16KB memory size and at most 1536 threads per SM
- For TILE_WIDTH = 16, each thead block uses 2*16*16*4B = 2KB of shared memory (i.e., we can have at most 8 thread blocks executing – i.e., 8*16*16 = 2048 threads per SM)
  - Since we can have at most 1536 threads per SM, we would need to run 6 blocks – 6*16*16= 1536, saturating the number of threads
- For TILE_WIDTH = 32, each thread block uses 2*32*32*4B = 8KB of shared memory (i.e., we can have at most 2 thread blocks executing – i.e., 2*1024 = 2048 threads per SM)
  - Since we can have at most 1536 threads per SM, we would need to run 1 block – 1*32*32 = 1024, leaving 512 threads unused

Questions?