

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

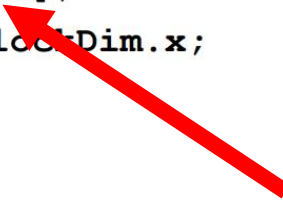
Shared Memory Example: 1D Stencil (Correct)

```
__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)
We can also do it dynamically (you can find more info on the Barlas book)

Constant Memory

```
__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

Grayscale 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;
}
}
```

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

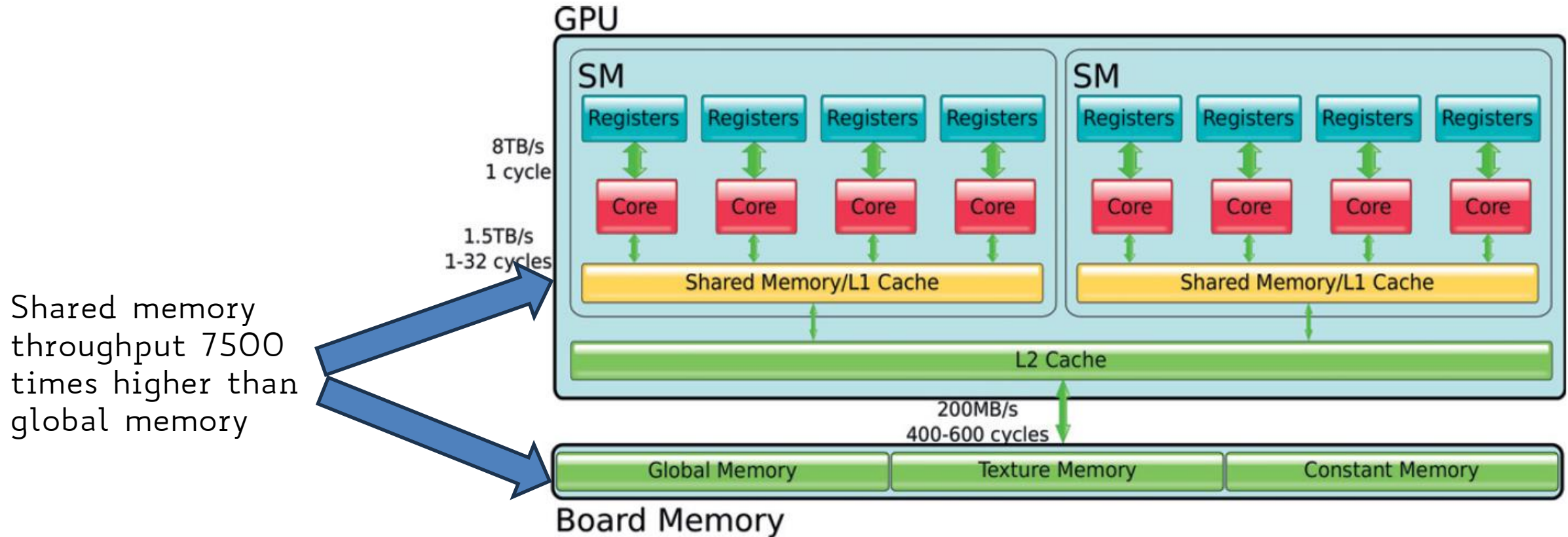
```
pixVal += in[curRow * w + curCol];
```

- 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? $(200 \text{ GB/s}) / (4 \text{ bytes}) = 50 \text{ G operands / s}$
- 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 $1.5 \text{ T} / 50 \text{ G}$
 - 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

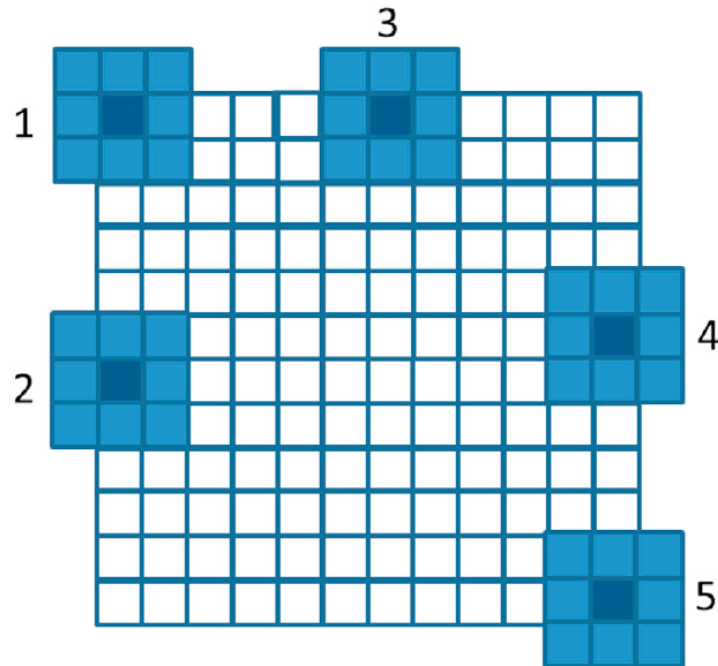
Memory Types



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

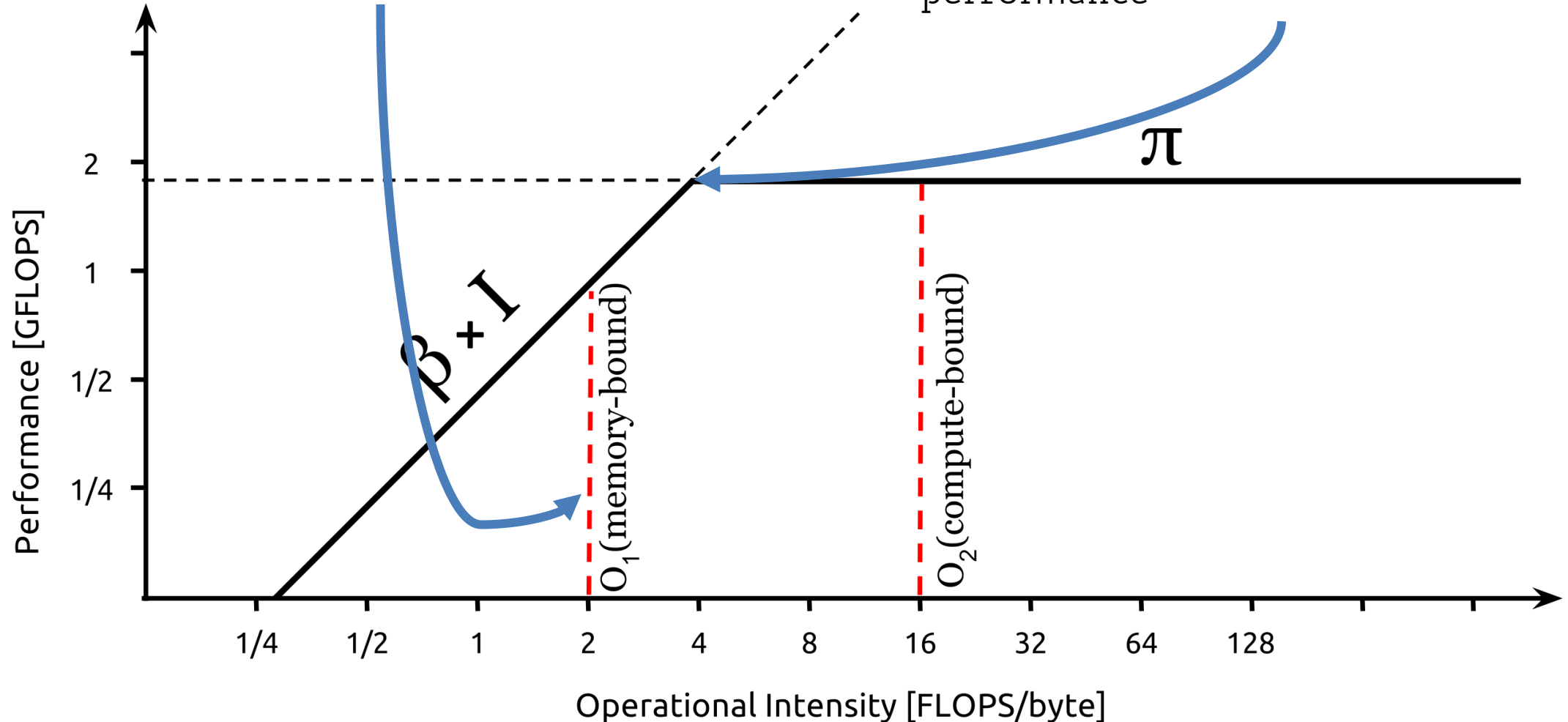
<https://docs.nersc.gov/tools/performance/roofline/>
<https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf>

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

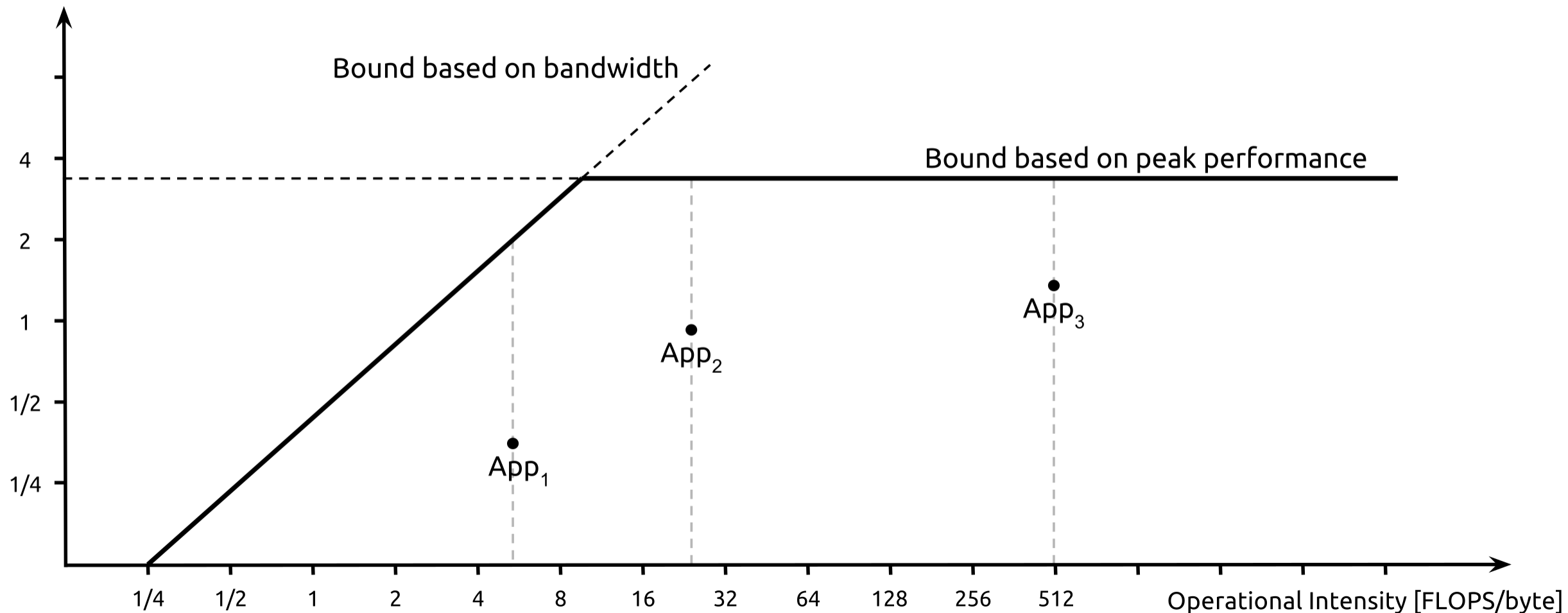
Ridge point

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

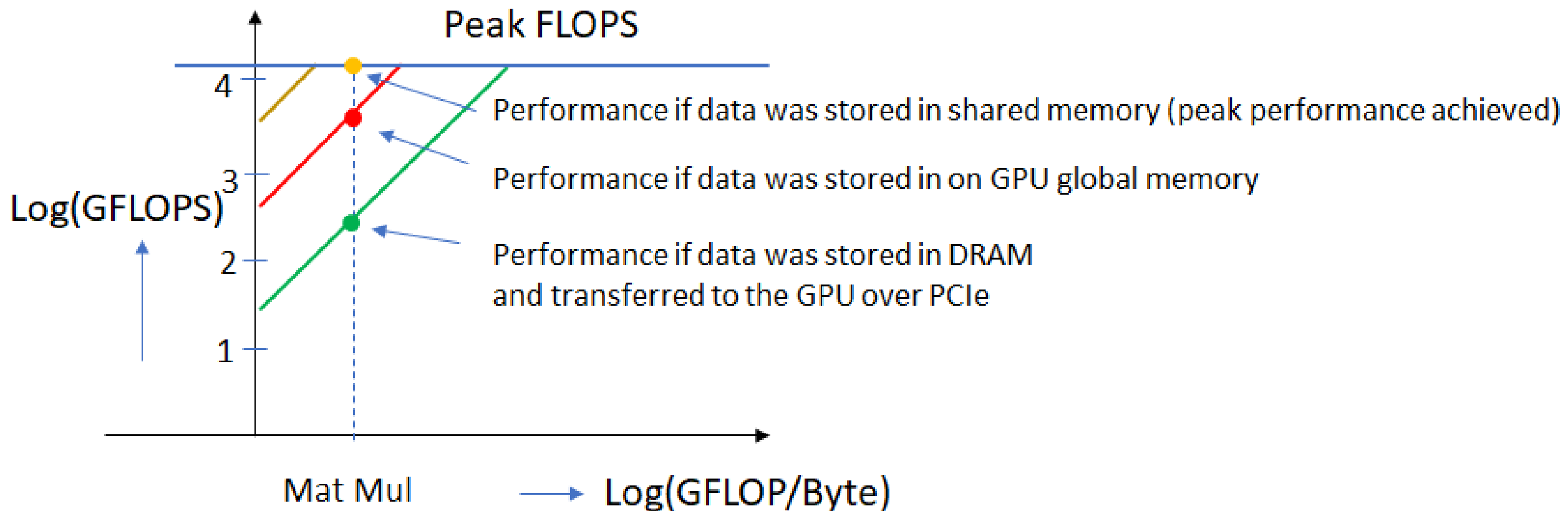


Roofline model

Performance [GFLOPS]



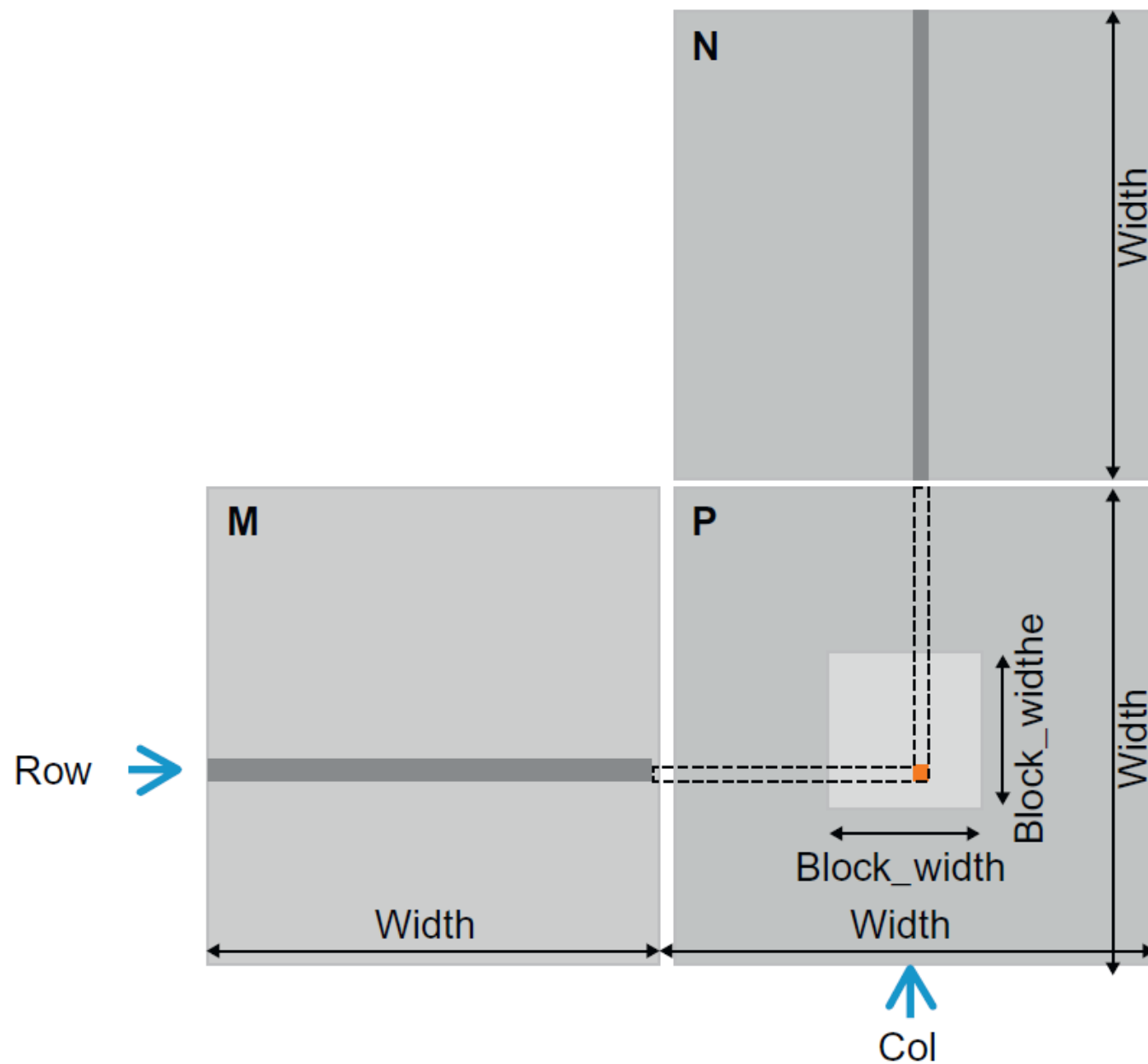
Roofline model



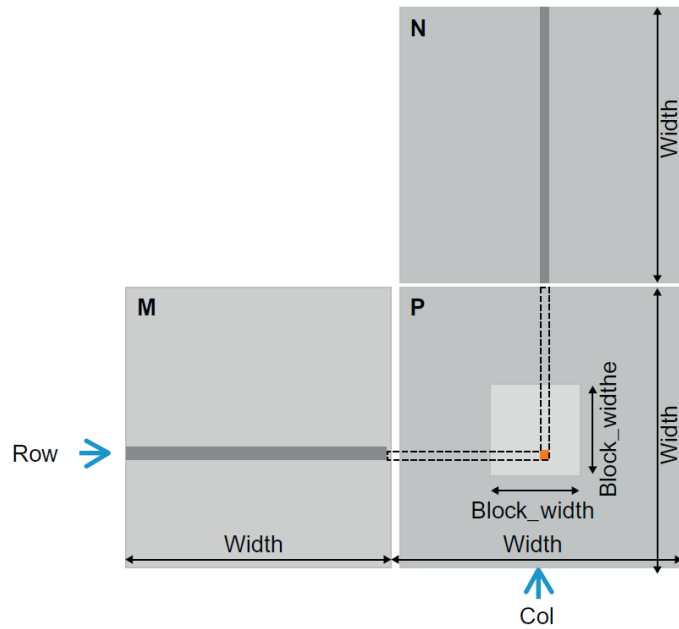
Questions?

Example: Matrix Multiplication with Tiling

Matrix Multiplication

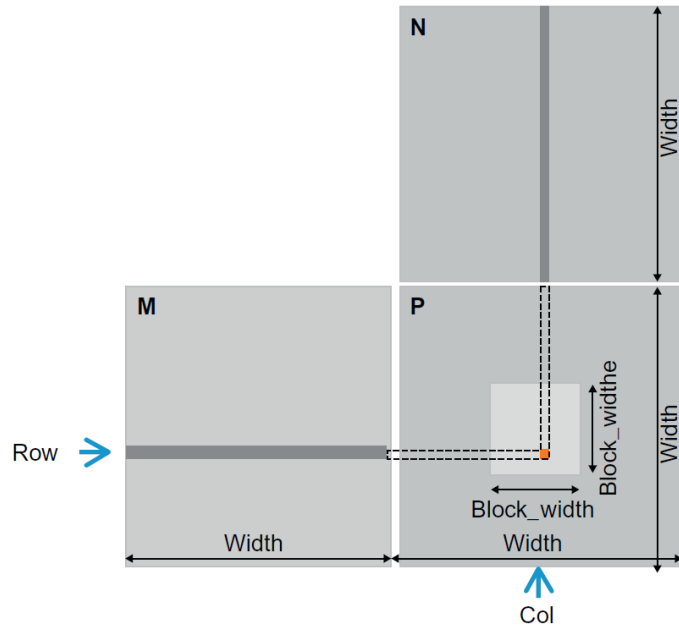


Matrix Multiplication



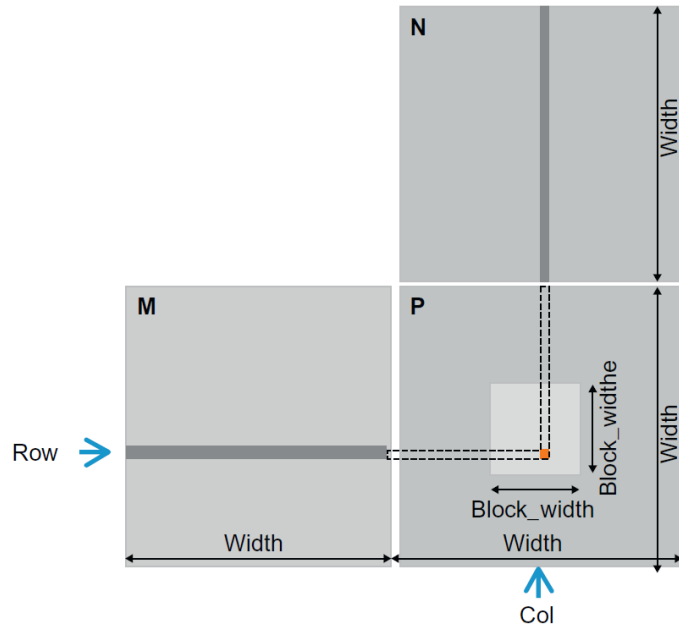
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,  
int Width) {
```

Matrix Multiplication



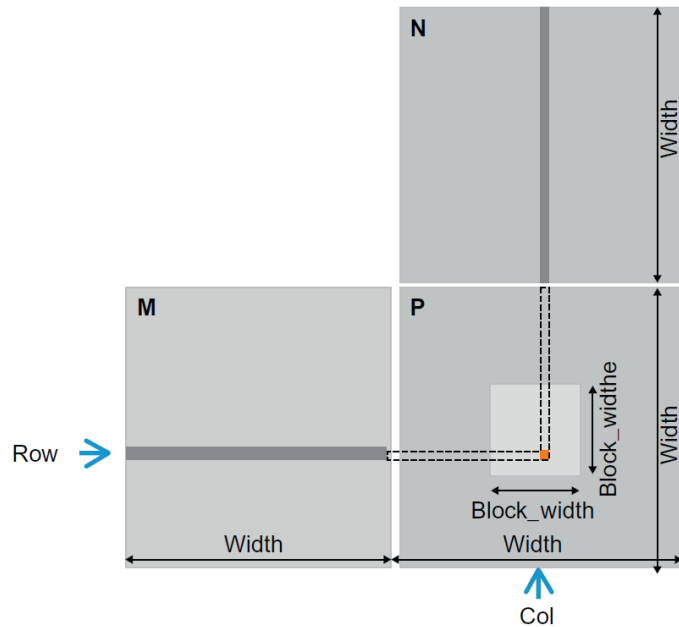
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,  
int Width) {  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
```

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 accesses
(one element from M and
one from N)

One multiplication and
one addition

Arithmetic intensity
(FLOP/operand) = 1

Arithmetic intensity
(FLOP/byte) = 8

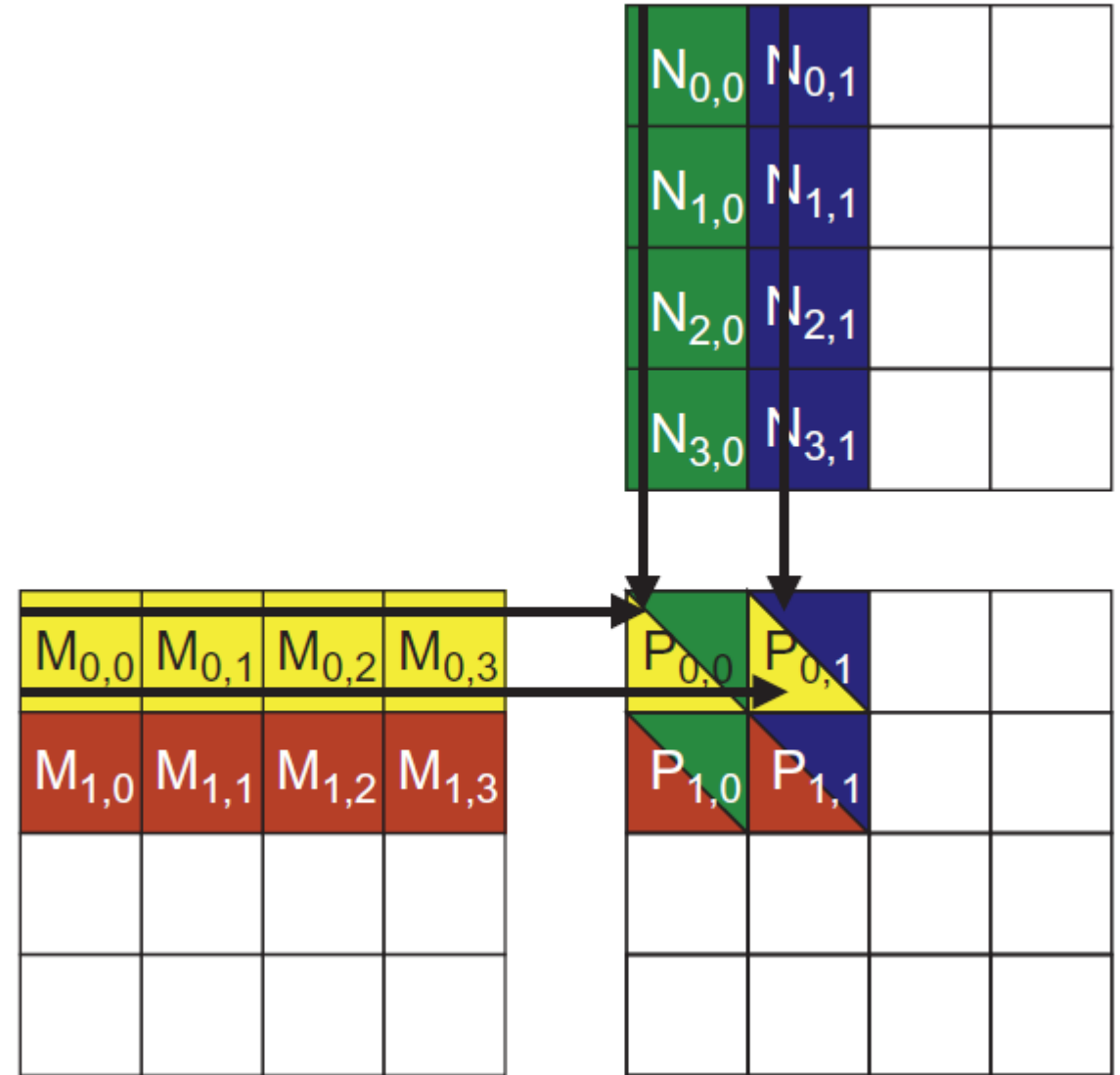
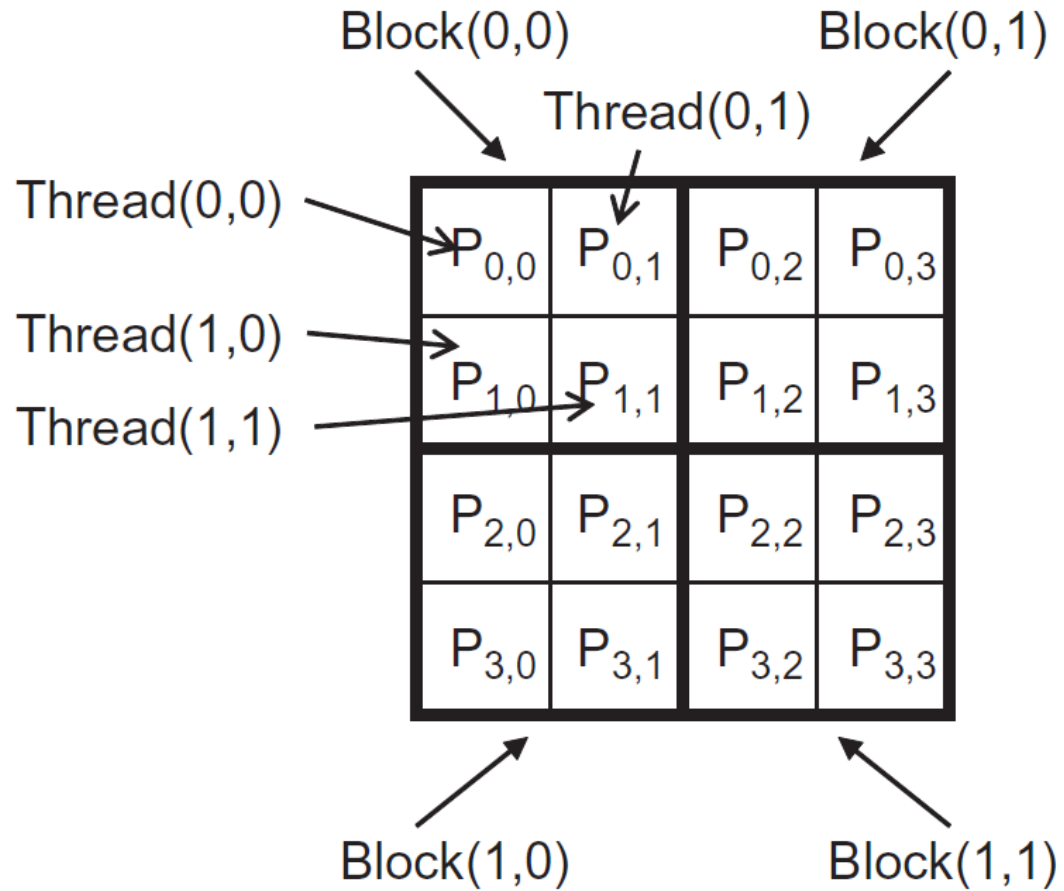
How to increase it? Let's try
to exploit shared on-chip
memory

```
__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;  
    }  
}
```

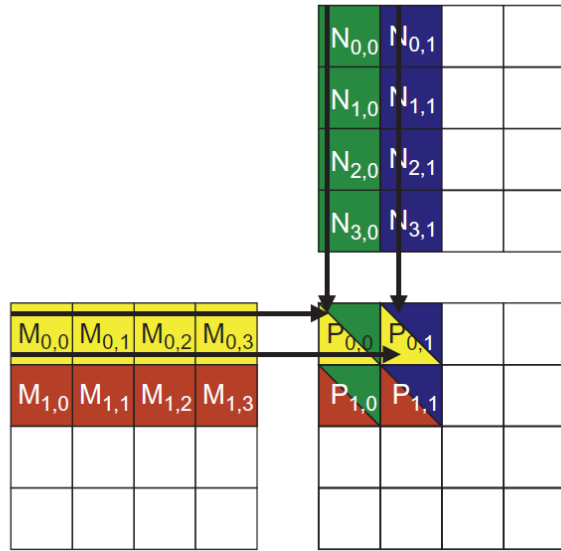
Questions?

Matrix Multiplication

BLOCK_WIDTH = 2



Matrix Multiplication

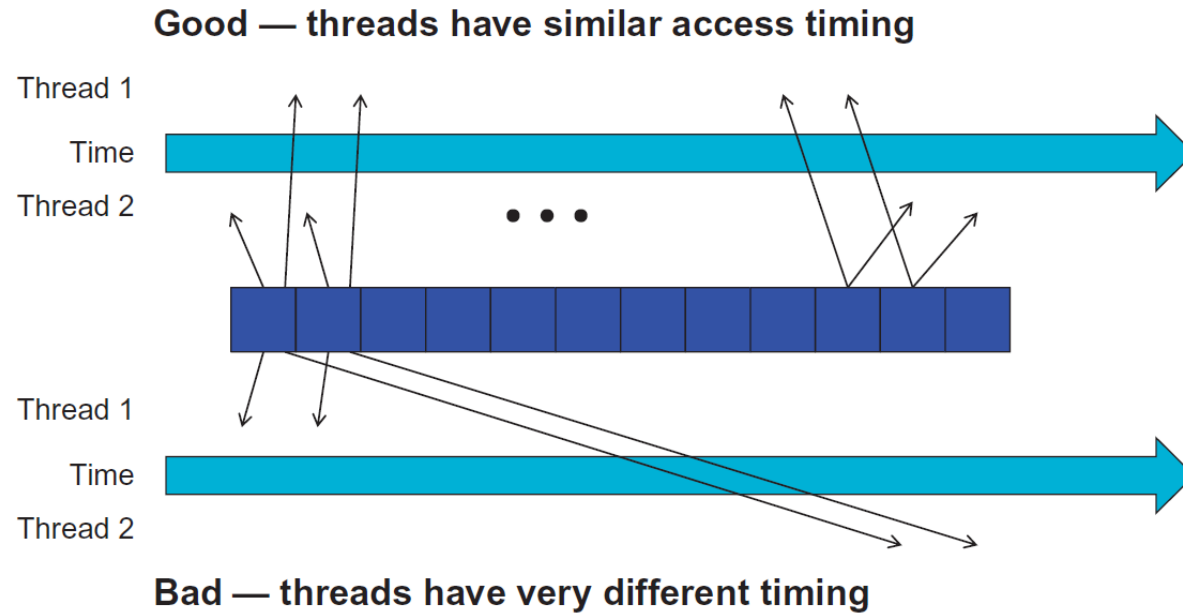


Access order

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

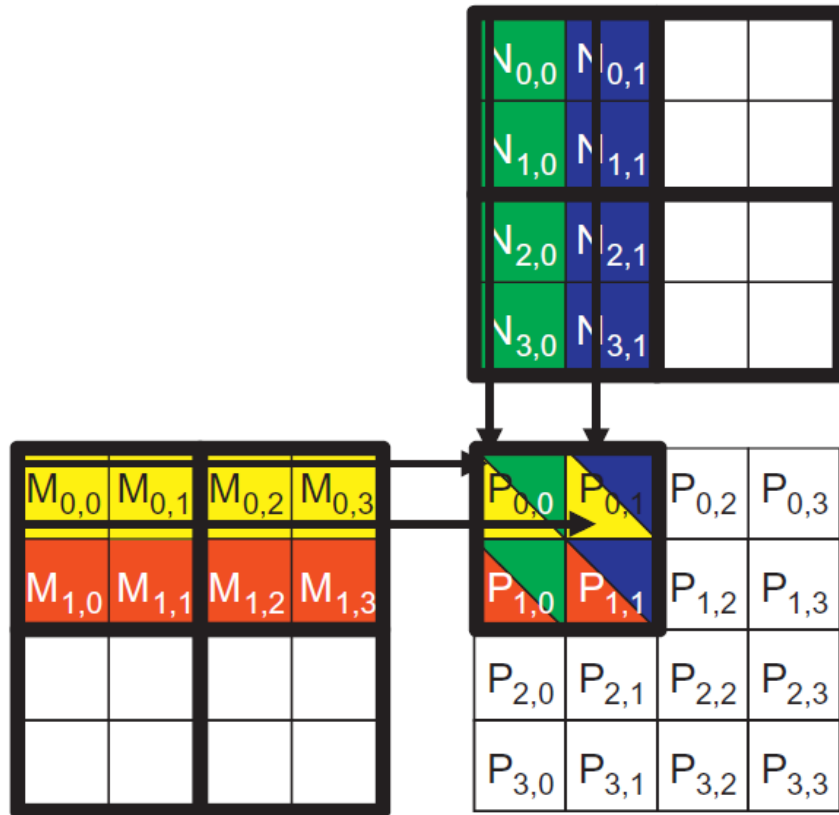
- Each element is accessed by 2 threads at the same time
- I.e., each element is loaded twice from global memory
- We could have one thread loading it from global to shared memory, and the other one reading it from shared memory, reducing traffic to global memory by 2x
- **IMPORTANT:** The potential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

Matrix Multiplication



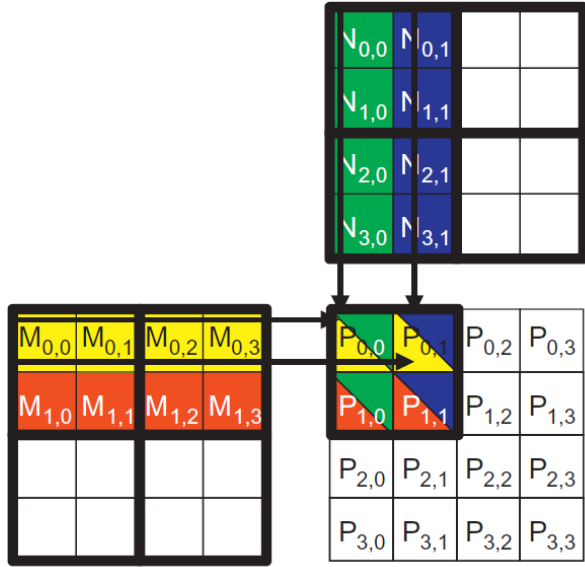
- Shared memory is small, it is important that once we load a value, that value is going to be accessed again soon
- If accesses are too distant in time, then we might have loaded something different in the meanwhile

Tiling



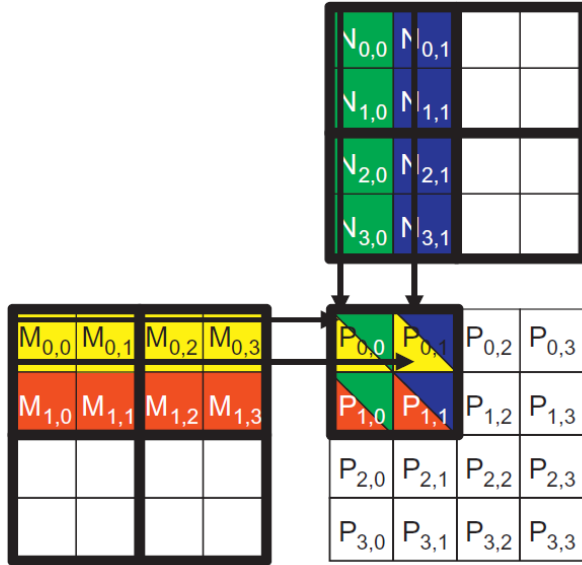
- We also divide the input matrices in tiles (of the same size of the tiles of the output matrix)
- For tile in (0, 2)
 - Each thread now loads one element from the first tile of M and one from the first tile of N from global to shared memory
 - E.g., P_{00} loads N_{00} and M_{00}
 - Sync (barrier)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

Tiling



	Phase 1		
thread _{0,0}	$M_{0,0}$ \downarrow $Mds_{0,0}$	$N_{0,0}$ \downarrow $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$
thread _{0,1}	$M_{0,1}$ \downarrow $Mds_{0,1}$	$N_{0,1}$ \downarrow $Nds_{1,0}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$
thread _{1,0}	$M_{1,0}$ \downarrow $Mds_{1,0}$	$N_{1,0}$ \downarrow $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$
thread _{1,1}	$M_{1,1}$ \downarrow $Mds_{1,1}$	$N_{1,1}$ \downarrow $Nds_{1,1}$	$PValue_{1,1} +=$ $Mds_{1,0} * Nds_{0,1} +$ $Mds_{1,1} * Nds_{1,1}$
	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) = $4 \times (4+4) = 32$ 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;

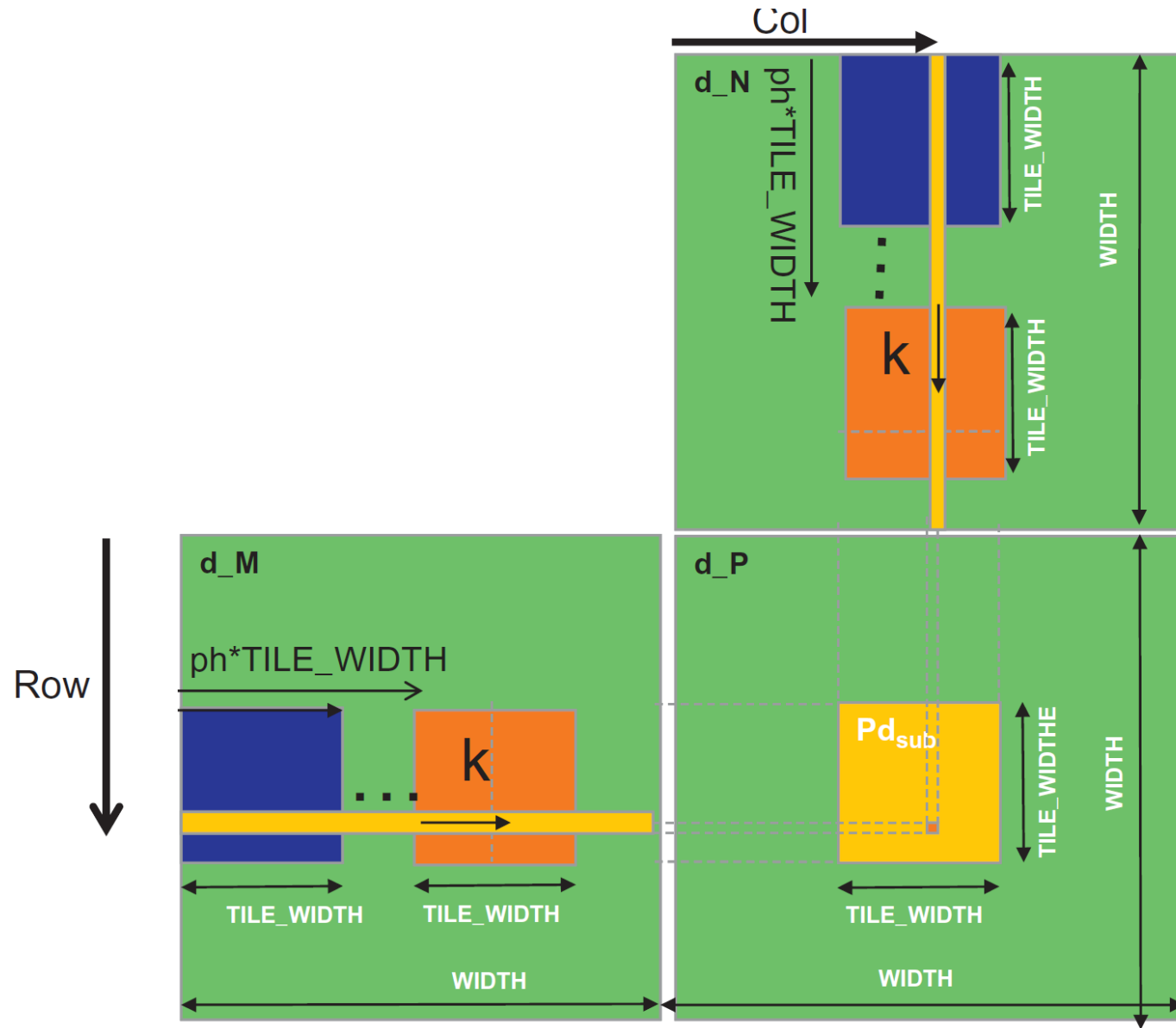
    // Identify the row and column of the d_P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

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.     }
15.     __syncthreads();
    }
    d_P[Row*Width + Col] = Pvalue;
}
```

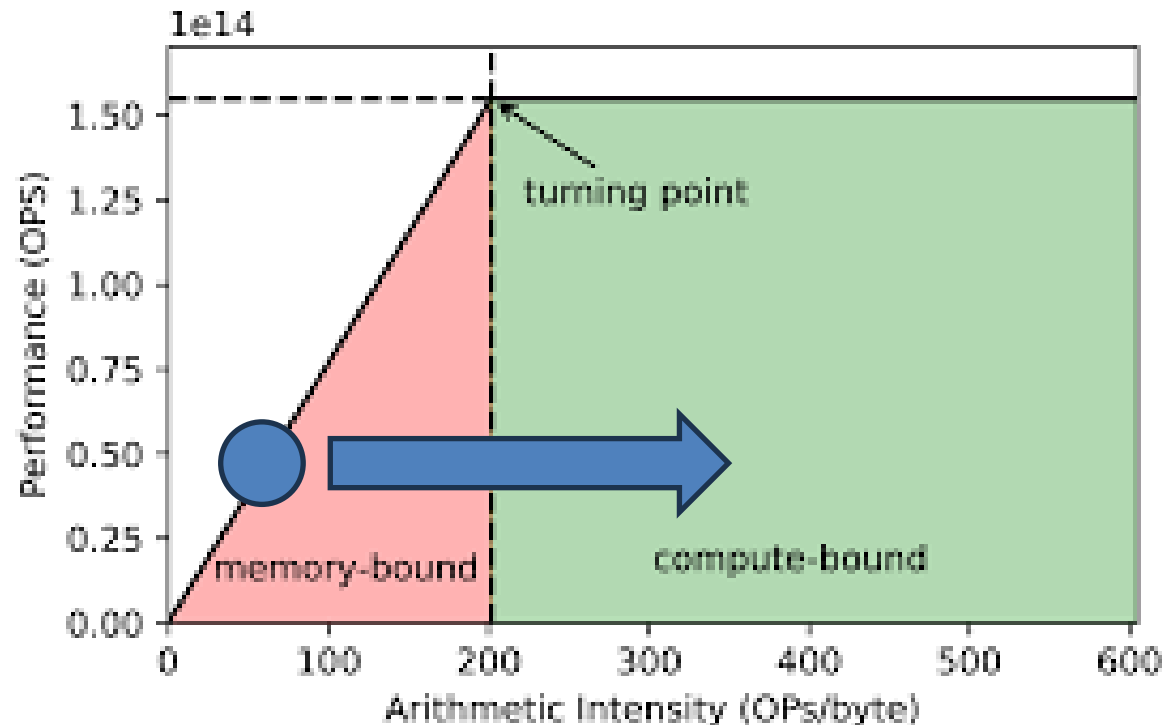
Calculation of tile index



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 thread block uses $2 \times 16 \times 16 \times 4\text{B} = 2\text{KB}$ of shared memory (i.e., we can have at most 8 thread blocks executing – i.e., $8 \times 16 \times 16 = 2048$ threads per SM)
 - Since we can have at most 1536 threads per SM, we would need to run 6 blocks – $6 \times 16 \times 16 = 1536$, **saturating the number of threads**
- For TILE_WIDTH = 32, each thread block uses $2 \times 32 \times 32 \times 4\text{B} = 8\text{KB}$ of shared memory (i.e., we can have at most 2 thread blocks executing – i.e., $2 \times 1024 = 2048$ threads per SM)
 - Since we can have at most 1536 threads per SM, we would need to run 1 block – $1 \times 32 \times 32 = 1024$, **leaving 512 threads unused**

Questions?

Cluster Hands-On

Example: Jacobi 2D

Atomic Operations in CUDA

```
int atomicAdd(int *old, int val);
```

where

old is variable to added to

val is the variable to be added to

result: $old = old + val$

return value: original value of old;

Explore alternatives

- How to do the check of convergence?
 - Compute sum and diffsum on the GPU, copy it to the CPU, and do the check?
 - Copy the data to the CPU and do the check on the CPU?