COSC 407
Intro to Parallel Computing

Topic 13: CUDA Threads – Part 2

COSC 407: Intro to Parallel Computing

Outline

Previously:

- Error Handling, cudaDeviceSynchronize
- Hardware architecture: sp → SM → GPU
- Thread Organization: threads → blocks → grids
  - Dimension variables (blockDim, gridDim)
- Thread Life Cycle From the HW Perspective
- Kernel Launch Configuration: 1D grids/blocks

Today:

- Kernel Launch Configuration: nD grids/blocks
- CUDA limits
- Thread Cooperation
- Running Example: Matrix Multiplication

Higher Dimensional Grids / Blocks

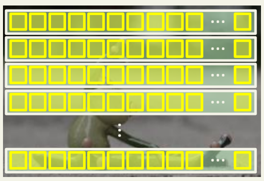
Remember: choose the breakdown of threads and blocks that **make sense to your problem**.

Example:

- Assume you want to process a 100 pixel x 70 pixel image
  (each 1 thread processes 1 pixel).
- We will have many options, e.g.:

Option: (1 block/row, 1 thread/pixel)
A grid of 1x70 blocks (gx=1, gy=70)
each block with 100x1 threads (dx=100,dy=1)

Another Option (1 block/segment)
A grid of 10x7 blocks (gx=10, gy=7)
each block with 10x10 threads



Higher Dimensional Grids/Blocks

```
kernelFunction <<< gridSize , blockSize >>>
```

- **gridSize**: **dimension and size of the grid in terms of blocks**
  – could be one of the following:
  - dim3(gx, gy, gz) → in case of 3D grid
    - Where gx, gy, gz define the three dimensions
  - dim3(gx, gy) → in case of 2D grid
    - equivalent to dime3(gx, gy, 1)
  - dim3(gx) **, or** an integer → in case of 1D grid
    - equivalent to dim3(gx,1,1) **or** simply **gx** (the integer)
    - e.g., dim3(8, 1, 1) = dim3(8) = 8
- **blockSize**: **dimension and size of each block in threads.**
  - dim3(bx, by, bz) → in case of 3D block
  - dim3(bx, by) → in case of 2D block
  - dim3(bx) **, or** an integer → in case of 1D block

Hello Again….

```
__global__ void hello(){
 printf("Thread(%d,%d,%d) in Block(%d,%d,%d) says:Hello!\n",
    threadIdx.x, threadIdx.y, threadIdx.z,
    blockIdx.x, blockIdx.y, blockIdx.z);
}
int main(){
    hello<<<dim3(2,1,1),dim3(2,1,1)>>>();    // same as hello<<<2,2>>>()
    cudaDeviceSynchronize();                 // force the printf() in
                                             // device to flush here
    printf("That's all!\n");
    return 0;
}
```

```
Thread(0,0,0) in Block(0,0,0) says:Hello!
Thread(1,0,0) in Block(0,0,0) says:Hello!
Thread(0,0,0) in Block(1,0,0) says:Hello!
Thread(1,0,0) in Block(1,0,0) says:Hello!
That's all!
```


Hello Again (same function))

```
__global__ void hello(){
 printf("Thread(%d,%d,%d) in Block(%d,%d,%d) says:Hello!\n",
threadIdx.x, threadIdx.y, threadIdx.z,
blockIdx.x, blockIdx.y, blockIdx.z);
}
int main(){
    dim3 gridSize(2,1,1), blockSize(2,1,1);
    hello<<<gridSize, blockSize>>>();
    cudaDeviceSynchronize();          // force the printf() in
                                      // device to flush
   printf("That's all!\n");
   return 0;
}
```

```
Thread(0,0,0) in Block(0,0,0) says:Hello!
Thread(1,0,0) in Block(0,0,0) says:Hello!
Thread(0,0,0) in Block(1,0,0) says:Hello!
Thread(1,0,0) in Block(1,0,0) says:Hello!
That's all!
```

Aside: `printf` on the kernel?

- Yes, although not a great idea..
  - Specific use cases
- Need to use `cudaDeviceSynchronize()`
  - Kernel runs asynchronously from host
  - See the code in previous slide

Computing # of Blocks for 2D Grids

Lets say we have an image of the size WIDTH x HEIGHT

And assume we use 2D blocks of # of threads `TILE_WIDTH` x `TILE_HEIGHT`

– E.g. TILE_WIDTH = TILE_HEIGHT= 32, totaling 1024 threads

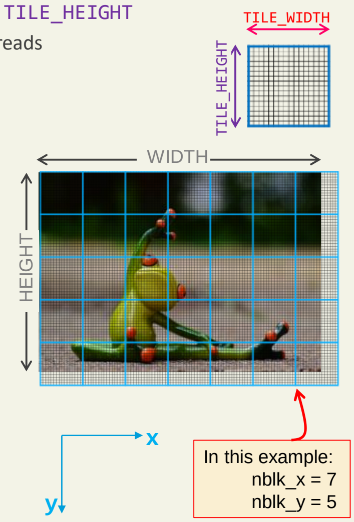
How do we determine the grid & block organization?

```
//block dimensions
int TILE_WIDTH  = 32;    //num of threads along x
int TILE_HEIGHT = 32;    //num of threads along y
dim3 blocksize(TILE_WIDTH,TILE_HEIGHT);

//grid dimensions
int nblk_x = (WIDTH - 1) / TILE_WIDTH + 1;
int nblk_y = (HEIGHT - 1)/ TILE_HEIGHT + 1;
dim3 gridsize(nblk_x, nblk_y);

//launch kernel
kernel<<<gridsize,blocksize>>(…);
```

```
void kernel(…){
 int x = blockIdx.x * blockDim.x + threadIdx.x;
 int y = blockIdx.y * blockDim.y + threadIdx.y;
...}
```



CUDA Limits

For CUDA compute capability 3.0+:

- Within a grid:
  - Total of $(2^{31} – 1)$ x 65535 x 65535 blocks
    - Maximum x-dimension of a grid: $2^{31} - 1$
    - Maximum y- and z- dimension of a grid: $2^{16} – 1$ (= 65535)
  - That is, **launch as many blocks as you want** (almost)!
- Within a block
  - Maximum total number of threads per block:
    - 1024 (or 512 on older GPUs supporting compute capability < 2)
  - Maximum dimension of a block (# of thread per dimension)
    - x- or y- dimension: 1024 (or 512)
    - z-dimension: 64
- The first assignment on CUDA walks you through this (see last lecture)
- ***Check full specs here.***

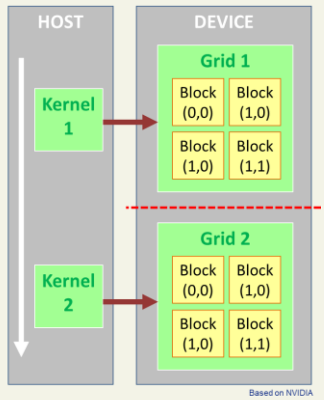
FAQs

- Organization of OpenMP threads vs. CUDA threads?
  - OpenMP:
    - number of threads p close to number of processors
  - CUDA:
    - many many threads, organized in 1D, 2D or 3D arrays

Threads Cooperation

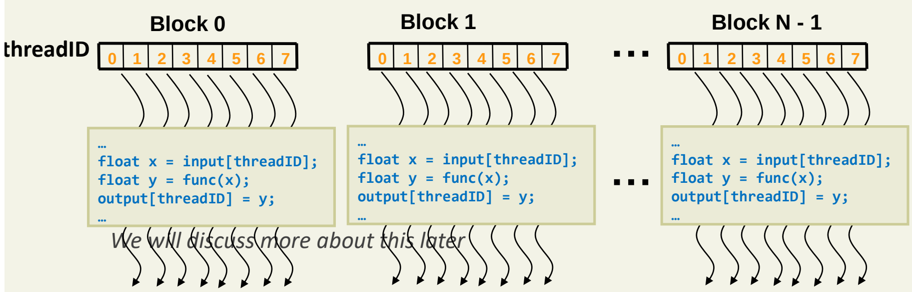
(more about this later)

- Threads in <u>same</u> **block can cooperate**
  - ***Synchronize*** their execution
  - Communicate via ***shared memory***
  - thread/block index is used to assign work and address shared data
- Threads in <u>different</u> **blocks cannot cooperate**
  - Blocks can execute in **any order** relative to other blocks.
  - There is no native way to synchronize all threads in all blocks.
    - **To synchronize threads in all blocks**, terminate your kernel at the synchronization point, and then launch a new kernel which would continue with your job



Threads Cooperation

- For now, all you need to remember is:
  - All threads in <u>all</u> blocks run the same kernel.
  - Threads within the <u>same block</u> cooperate via shared memory, atomic operations and barrier synchronization.
  - Threads in different blocks CANNOT cooperate.



We will discuss more about this later

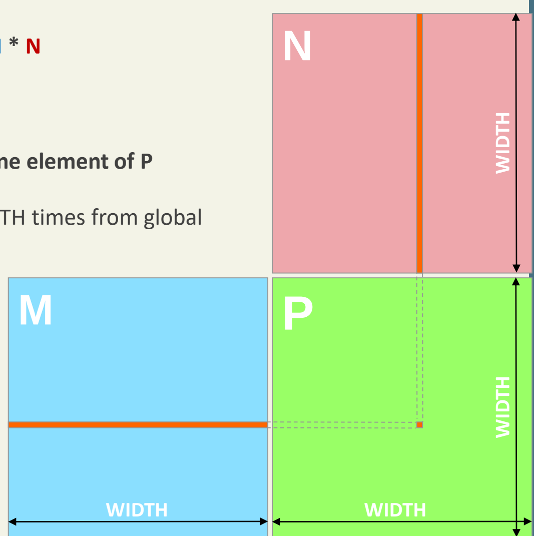
Matrix Multiplication

- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Assume square matrix for simplicity
  - For now, we will discuss
    - Memory data transfer API between host and device
    - Thread ID usage
  - Later
    - How to speed up performance

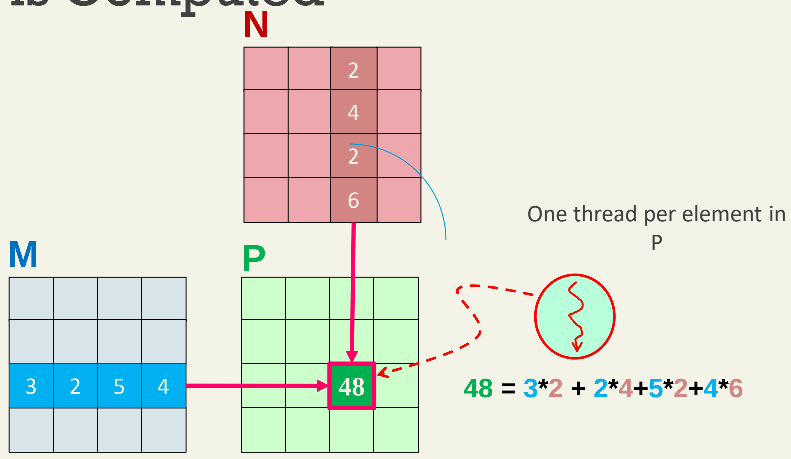
Programming Model

P = M * N

- Size is WIDTH x WIDTH
- **Each thread** calculates **one element of P**
- M and N are loaded WIDTH times from global memory



How Each Element in P is Computed

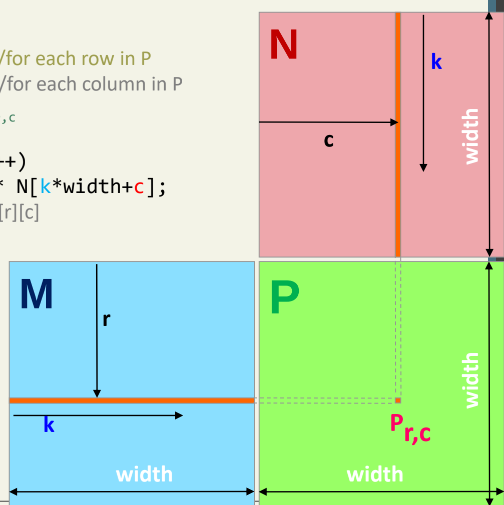


Slide materials based on, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
© David Kirk/NVIDIA and Wen-mei W. Hwu

Serial Code

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulSerial(float* M, float* N, float* P, int width)
{
  //for each element Pr,c
  for (int r=0; r<width; r++){    //for each row in P
    for (int c=0; c<width; c++){ //for each column in P
         //Compute the value of Pr,c
         float value = 0;
         for (int k=0; k<width; k++)
           value += M[r*width+k] * N[k*width+c];
         P[r*width+c] = value; //P[r][c]
    }
  }
}
```

```
#pragma omp parallel for private (j, k)
for (r=0; r< n; r++)
 for (c=0; c<n; c++)
  for (k=0; k<m; k++)
    P[r][c] += M[r][k] * N[k][c];
```



Slide materials based on, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
© David Kirk/NVIDIA and Wen-mei W. Hwu

Processing 2D arrays in C

Review

Let's say we have two 2D arrays that we want to process them in loops (e.g. initialize to 0's)

Serial Code:

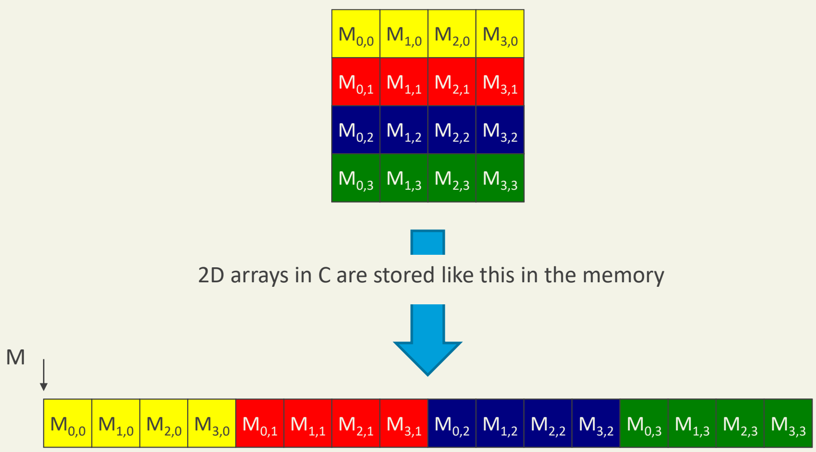
```
int  A[10][10];
int* B = malloc(100 * sizeof(int));

for (int r = 0; r < 10; r++){
    for (int c = 0; c < 10; c++) {
        A[r][c] = 0; // this is ok
        B[r][c] = 0; // ERROR!
    }
}
```

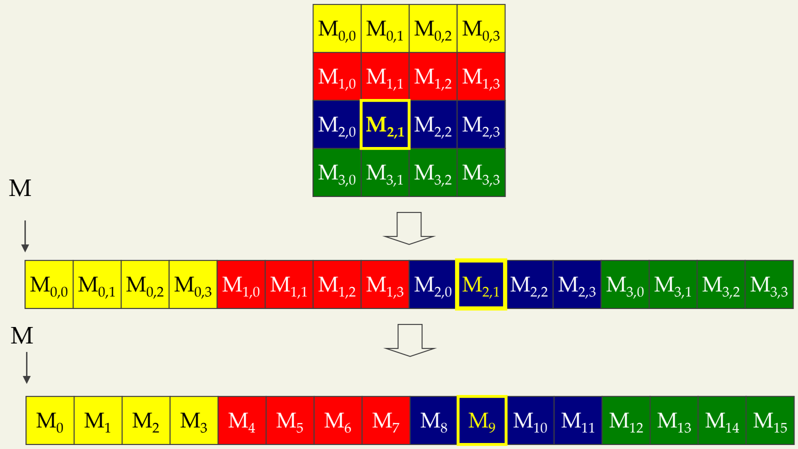
You cannot use [r][c] with dynamically allocated arrays (B can only be a pointer or a vector). **Solution: use row-major format!**

Memory Layout: Matrix in C

Review



Row-Major Layout in C/C++



`Row*Width+Col = 2*4+1 = 9` → M_9

```
M[r][c] = M[r * Width + c]
```


Processing 2D arrays in C

Review cont.

Let's say we have two 2D arrays that we want to process them in loops (e.g. initialize to 0's)

Serial Code:

```
int  A[10][10];
int* B = malloc(100 * sizeof(int));

for (int r = 0; r < 10; r++){
    for (int c = 0; c < 10; c++) {
        A[r][c] = 0;          // this is ok
        B[r*10+c] = 0;        // this is ok now!
    }
}
```

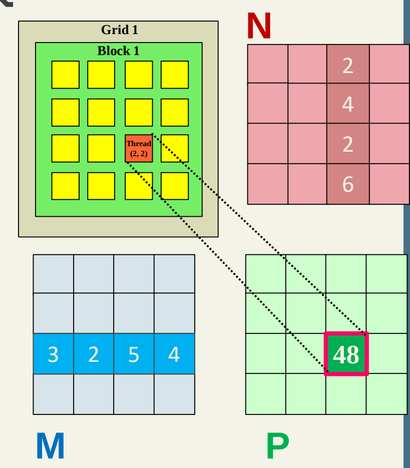
Parallel Code: Using *One* Block

Basic Idea

- Only **ONE** block used to compute the output matrix P
- Each thread computes one element of P as follows:
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute and stores the result on an off-chip memory (DRAM)

Limitation:

- Size of P is limited to 32x32
  - i.e. the number of threads allowed in a thread block.



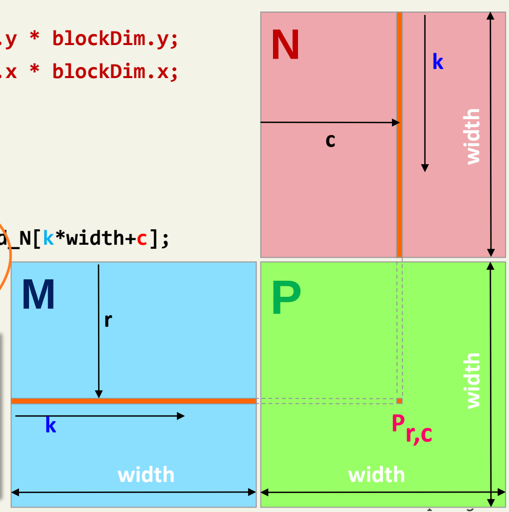
Parallel: Kernel – *One* Block

```
// Matrix multiplication kernel – each thread computes one P element
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int width){
    //find index of P_r,c element
    int r = threadIdx.y + blockIdx.y * blockDim.y;
    int c = threadIdx.x + blockIdx.x * blockDim.x;
    //compute P's element
    if(r<width && c<width){
      float value = 0;
      for (int k=0; k<width; k++)
        value += d_M[r*width+k] * d_N[k*width+c];
      d_P[r*width+c] = value;
    }
}
```

Also ok to use

```
int r = threadIdx.y;
int c = threadIdx.x;
```

But it is better to use the general formula (here, we use only one block, and thus blockIdx = 0). ***WHY BETTER?***

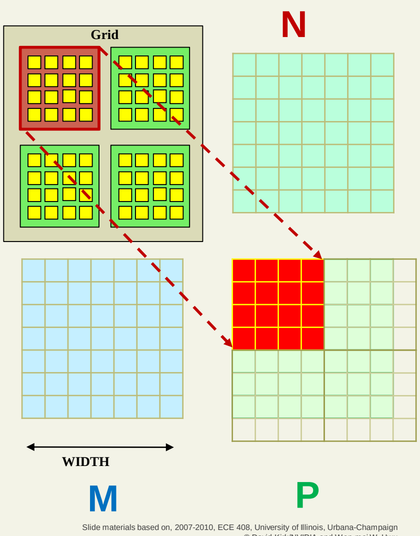


Parallel : Host – *One* Block

```
void MatrixMulOnDevice(float* M, float* N, float* P, int width)
{
  int size = width * width * sizeof(float);
  float *d_M, *d_N, *d_P;
  //1) Allocate M, N, P on device memory. Copy M,N to device
  cudaMalloc(&d_M, size);
  cudaMalloc(&d_N, size);
  cudaMalloc(&d_P, size);
  cudaMemcpy(d_M, M, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_N, N, size, cudaMemcpyHostToDevice);
  //2) Kernel invocation code
  dim3 blockSize(width, width);
  MatrixMulKernel<<<1 , blockSize>>>(d_M, d_N, d_P, width);
  //3) Read P from the device
  cudaMemcpy(P, d_P, size, cudaMemcpyDeviceToHost);
  //4) Free device matrices
  cudaFree(d_M); cudaFree(d_N); cudaFree(d_P);
}
```


Using *Multiple* Blocks

- We saw that using only **one block has a serious limitation**: size of matrix limited by 1024.
- Also, you are not fully using your GPU
- **Solution**: use multiple blocks
  - We shall apply the method explained previously
- More on this next day



Remember…

Why we need to divide threads into blocks with the grid?

- To make thread organization better fit the problem
  - e.g., 2D blocks for 2D images.
- To satisfy CUDA limits (only 1024 threads per block)
  - We also need to avoid GPU hardware limits
    - **For example, G80** has 16 SMs.
      - Each **SM** can process up to **8 blocks** at a time and up to **768 threads** at a time (more later)
- To exploit the GPU full power
  - E.g., one block means one SM is functioning and remaining are not
- To allow for threads communication at different levels
  - Threads within same block have “shared memory” and can sync. Threads in different blocks cannot sync (at least directly) and can only share data through the global memory.
  - ***More about this next…***

Summary

Today:

– Kernel Launch Configuration: nD grids/blocks
– CUDA limits
– Thread Cooperation
– Running Example: Matrix Multiplication

Next:

– Tiling
– CUDA Scalability
– Thread Scheduling on the H/W: Thread Lifecycle
  - zero-overhead and latency tolerance
– GPU limits
– CUDA Memories Types (and Performance)
– Example: Improving Performance of Matrix Multiplication

Topic 13: CUDA Threads – Part 2 COSC 407: Intro to Parallel Computing