Matrix Multiplication & Cuda Execution Model

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

Outline Today

- A more complex kernel, Matrix-matrix multiplication.
- Query Device Property
- Assign resources to Blocks

vecAddition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C, int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if ( i < n )
      C[i] = A[i] + B[i];
}
//Where is the loop?
```

Matrix-Matrix Multiplication

- Widely used in Science and Engineering
  - Graphics
  - Linear equation systems.
  - Statistics
  - Computer Vision and Image processing.
  - Data Encryption

Matrix-Matrix Multiplication

- d_M (I by J) times d_N(J by K) produces d_P(I by K)
- In this class, we use square matrix for discussion.
- Where I = J = K, we use **Width or n** for I, J and K.

Matrix-Matrix Multiplication

- Sequential Code on CPU

```
void mul(float c[], float a[], float b[], int n){ // n is the Width of matrices
  float sum;                      // a, b, c are linearized 1D array to stored 2d matrix
  int di, dj, i;
  for(di = 0; di < n; di++){
    for(dj = 0; dj < n; dj++){
      sum=0;
      for(i=0; i < n; i++){
         sum += a[di*n + i] * b[i* n + dj]; // compute c(di, dj) in 2D matrix
      }
      c[di*n + dj] = sum;
    }
  }
}//end of mul
```

Matrix-Matrix Multiplication

- Sequential Code on CPU

```
void mul(float c[], float a[], float b[], int n){ // n is the Width of matrices
  float sum;                    // a, b, c are linearized 1D array to stored 2d matrix
  int di, dj, i;
  for(di = 0; di < n; di++){
    for(dj = 0; dj < n; dj++){
      sum=0;
      for(i=0; i < n; i++){
         sum += a[di*n + i] * b[i* n + dj]; // compute c(di, dj) in result matrix
      } //this is the dot product of row di in matrix a and one column dj in matrix b
       // a[di *n + i ] is the a(row di, column i); increasing i will go through the whole row di.
       // To visit the next element in column dj in b, we have to skip a whole row of elements,
       // because we use row-major storage of 2D matrix.
      c[di*n + dj] = sum;
    }
  }
}//end of mul
```

Matrix-Matrix Multiplication CUDA

- When perform matrix multiplication, each element in matrix d_P is an inner product of a row of d_M and a column of d_N.
- When we design our kernel, we map 2D grid of threads to the result matrix d_P.
  - So that each thread will produce one element in d_P matrix.
  - No matter how much data each thread will use or read from input array d_M and d_N.

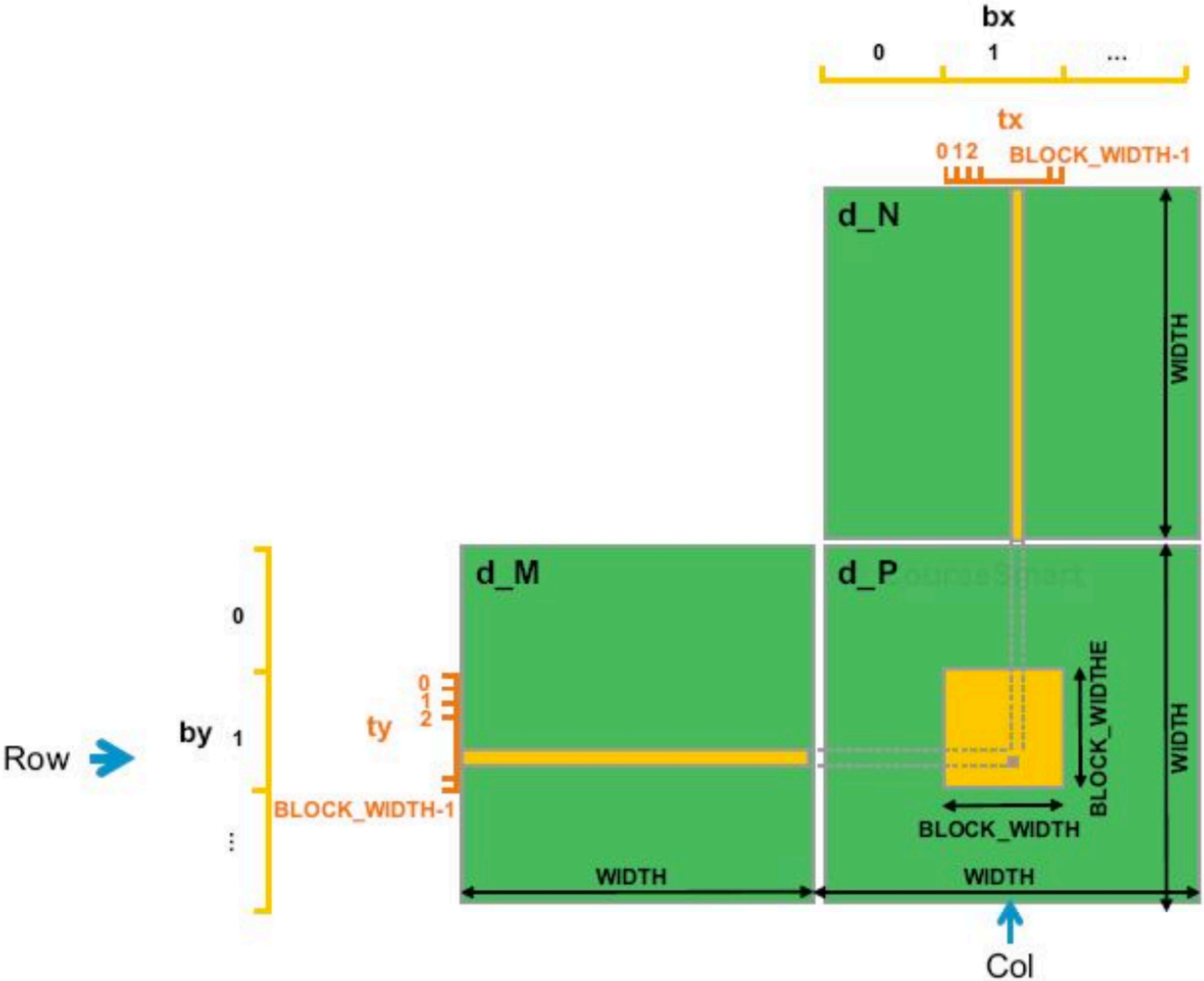


FIGURE 4.6

Matrix multiplication using multiple blocks by tiling d_P.

Matrix-Matrix Multiplication CUDA

- In figure 4.6, we imagine we put a 2D grid of threads on top of matrix d_P,
  - The yellow area is one of the thread block.
- Now we focus on the thread at (Row, Col) in grid,
  - Row is the global row index of the thread in the grid in y direction.
  - Col is the global column index of the thread in the grid in x direction.

Matrix-Matrix Multiplication CUDA

- Now we focus on the thread at (Row, Col) in grid,
  - Remember, this thread calculates d_P(Row,Col) by performing inner product of a row at Row in d_M and a column at Col in d_N.
  - This thread reads one row from d_M in global memory.
  - This thread reads one column from d_N in global memory.

Matrix-Matrix Multiplication CUDA

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

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_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 += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 4.7

In text, 16 by 16 thread blocks is used.

A simple matrix–matrix multiplication kernel using one thread to compute each d_P element.

Query Device Properties

- Find out questions like:
  - Number of SMs in device?
  - Max # of threads could be assigned to each SM?
- APIs
  - int dev_count;
  - cudaGetDeviceCount(&dev_count);

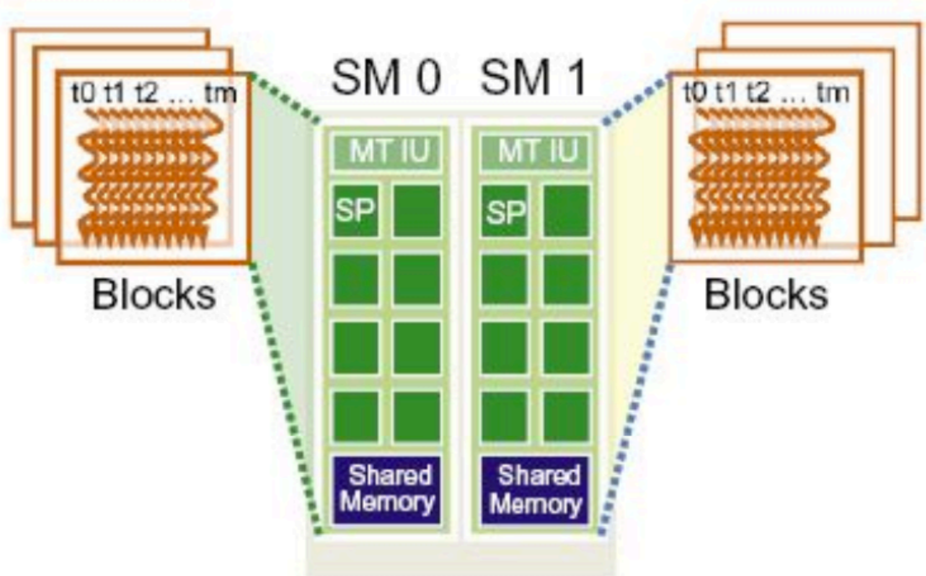
Query Device Properties

```
cudaDeviceProp dev_ prop;
for ( i = 0; i < dev_count; i ++)
{
   cudaGetDeviceProperties( &dev_ prop,  i );
   // decide if device has sufficient resources and capabilities
}
```

- The built-in type cudaDeviceProp is a C structure with fields that represent the properties of a CUDA device.
- Demo Code on GPU server
  - /usr/local/cuda-8.0/samples/1_Utilities/deviceQuery

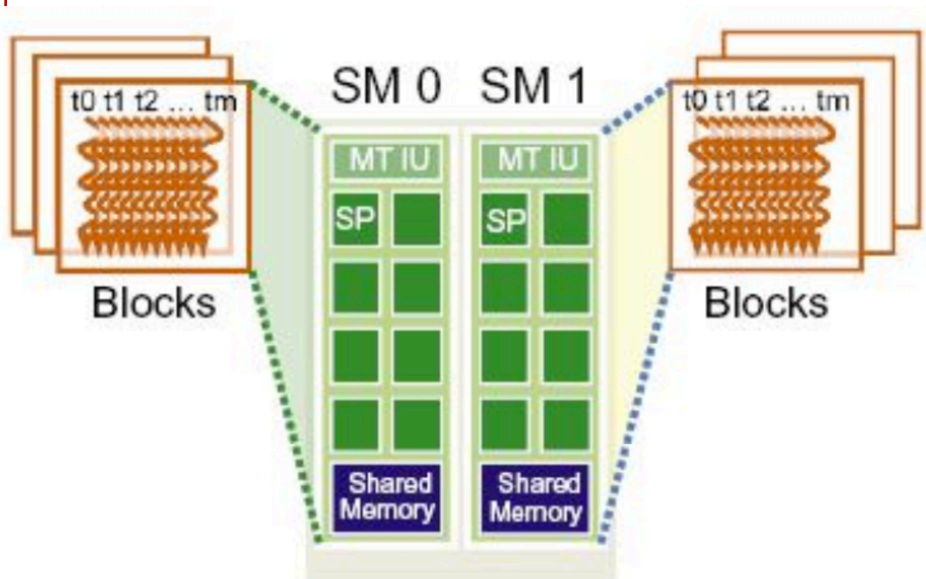
Assigning Resources to Blocks

- Threads are assigned to SMs on a block-by-block basis,
- In figure right, multiple thread blocks can be assigned to each SM.



Assigning Resources to Blocks

- But there is limit on the max # of blocks to be assigned to each SM.
  - E.g. may allow up to 8 blocks that can be assigned to each SM.
    - Threads of 8 blocks actively residing in the SM.



Assigning Resources to Blocks

- Therefore, a limit on the # of blocks that can be actively executing in GPU device.
- Most grid contains many more blocks than this limit.
- Runtime system
  - maintains a list of blocks that need to execute,
  - assigns new blocks to SMs once they finish the blocks previously assigned to them.

Assigning Resources to Blocks

- Also, another SM resource limitation
  - Max # of threads that can be simultaneously tracked and scheduled.
  - Recent CUDA device designs, **X** threads can be assigned to each SM, E.g. X = 1536 or 2048.
    - Could be 6 block of 256 threads each Or 3 blocks of 512 threads each if it is 1536.
    - If we a device has 30 SMs, the device can have up to 46080 threads **simultaneously residing** in the device, for 1536.

Summary

- A more complex kernel, Matrix-matrix multiplication.
- Query Device Property
- Assign resources to Blocks
- Please read chapter 4.3, 4.5 and 4.6

Future Classes

- Chapter 4.7 Thread scheduling and latency tolerance.
- Set up double pointers
- Text processing on GPU.
- Memory Model of GPU device