ECE408/CS483/CSE408 Spring 2018

Applied Parallel Programming

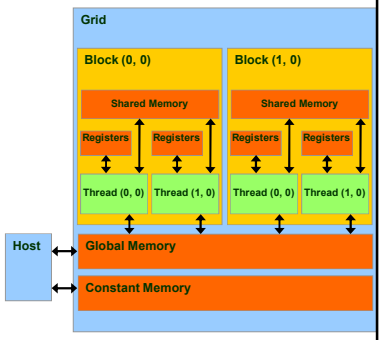
Lectures 4:
Memory Model

Objective

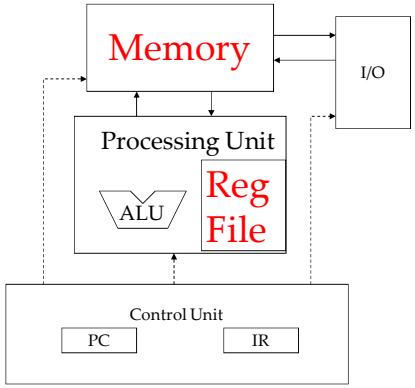
- To learn the basic features of the memories accessible by CUDA threads
- To prepare for MP-2 - basic matrix multiplication
- To learn to evaluate the performance implications of global memory accesses

Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



The Von-Neumann Model



Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
  - The decode stage typically accesses register file
- Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.
- An example instruction cycle is the following:

Fetch | Decode | Execute | Memory

Operate Instructions

- Example of an operate instruction:

  ADD R1, R2, R3

  R1 – dest, R2, R3 - src

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Memory Access Instructions

- Examples of memory access instruction:

  LDR R1, R2, #2

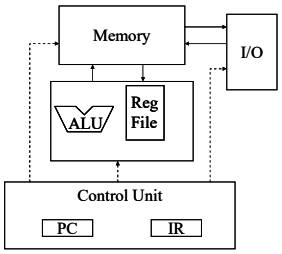
  STR R1, R2, #2

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Registers vs Memory

- Registers are “free”
  - No additional memory access instruction
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large



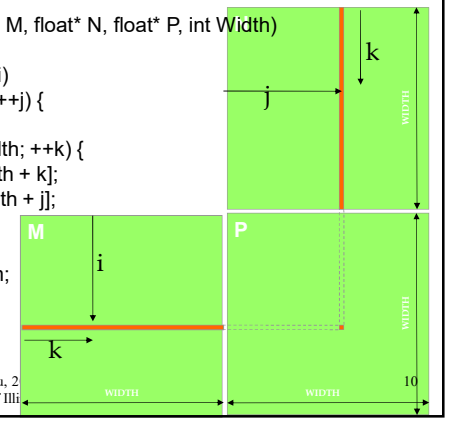
CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except per-thread arrays that reside in global memory

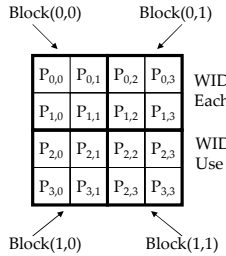
Matrix Multiplication Example
A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in single precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         float sum = 0;
         for (int k = 0; k < Width; ++k) {
            float a = M[i * Width + k];
            float b = N[k * Width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```



Kernel Function - A Small Example

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks



WIDTH = 4; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks

A Slightly Bigger Example
(TILE_WIDTH =2)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4* 4 = 16 blocks

A Slightly Bigger Example (cont.)
(TILE_WIDTH = 4)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 =16 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant
   dim3 dimGrid(ceil((1.0*Width)/TILE_WIDTH),
                ceil((1.0*Width)/TILE_WIDTH), 1);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

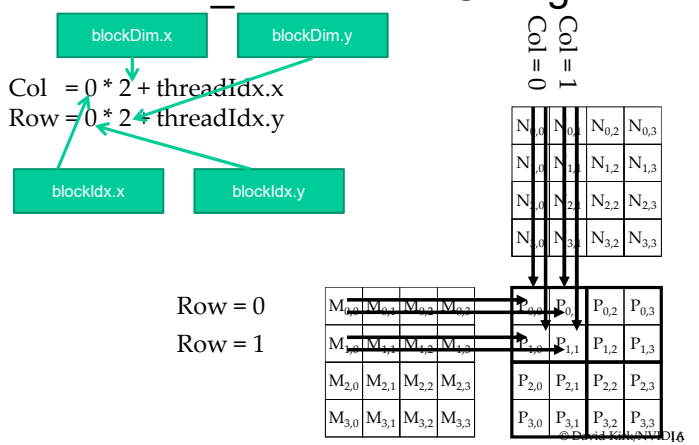

Kernel Function

```
// Matrix multiplication kernel – per thread code

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

   // Pvalue is used to store the element of the matrix
   // that is computed by the thread
   float Pvalue = 0;
```


Work for Block (0,0) in a TILE_WIDTH = 2 Configuration

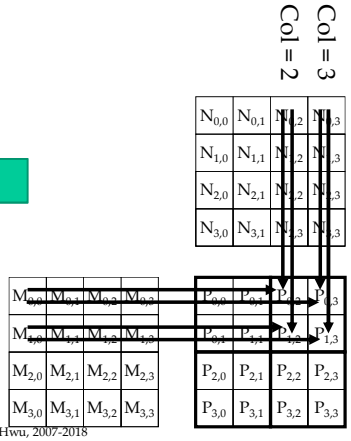


Work for Block (0,1)

Col = 1 * 2 + threadIdx.x
Row = 0 * 2 + threadIdx.y

blockIdx.x

blockIdx.y



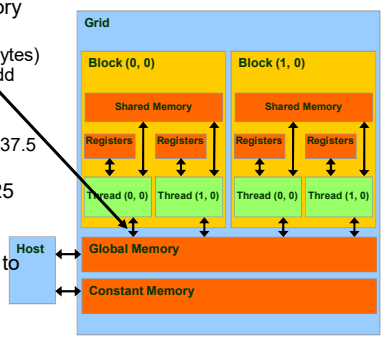
A Simple Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
// Calculate the row index of the d_P element and d_M
 int Row = blockIdx.y*blockDim.y+threadIdx.y;
// Calculate the column idenx 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;
  }
}
```


How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS



ANY MORE QUESTIONS?
READ CHAPTER 4!