GPU Computing with CUDA

Hands-on: Shared Memory Use (Dot Product, Matrix Multiplication)

Dan Melanz & Andrew Seidl
Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Portland, Oregon
August 4, 2013

CUDA Programming

- Remember, CUDA programs have a basic flow:
  1)The host initializes an array with data.
  2)The array is copied from the host to the memory on the CUDA device.
  3)The CUDA device operates on the data in the array.
  4)The array is copied back to the host.

Quick examples...

Example 1: Vector Dot Product

- Recall the dot product example from last time:
  - Given vectors a and b each with size N, store the result in scalar c

$$c = \mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \ldots + a_N b_N$$

Purpose of the exercise: use shared memory

Example 1: Vector Dot Product

- We originally used a global memory vector to store the product of the vector elements
  - The C array was passed in as a function argument

```
if(i<N) C[i] = A[i] * B[i];
```

- This time around, use a shared memory vector instead:

```
__shared__ float C_shared[N];
 if(i<N) C_shared[i] = A[i] * B[i];
```

Example 1: Vector Dot Product

- To compile this code:

```
>> nvcc dotProductShared.cu
```

- To run this code:

```
>> qsub submit_example.sh
```

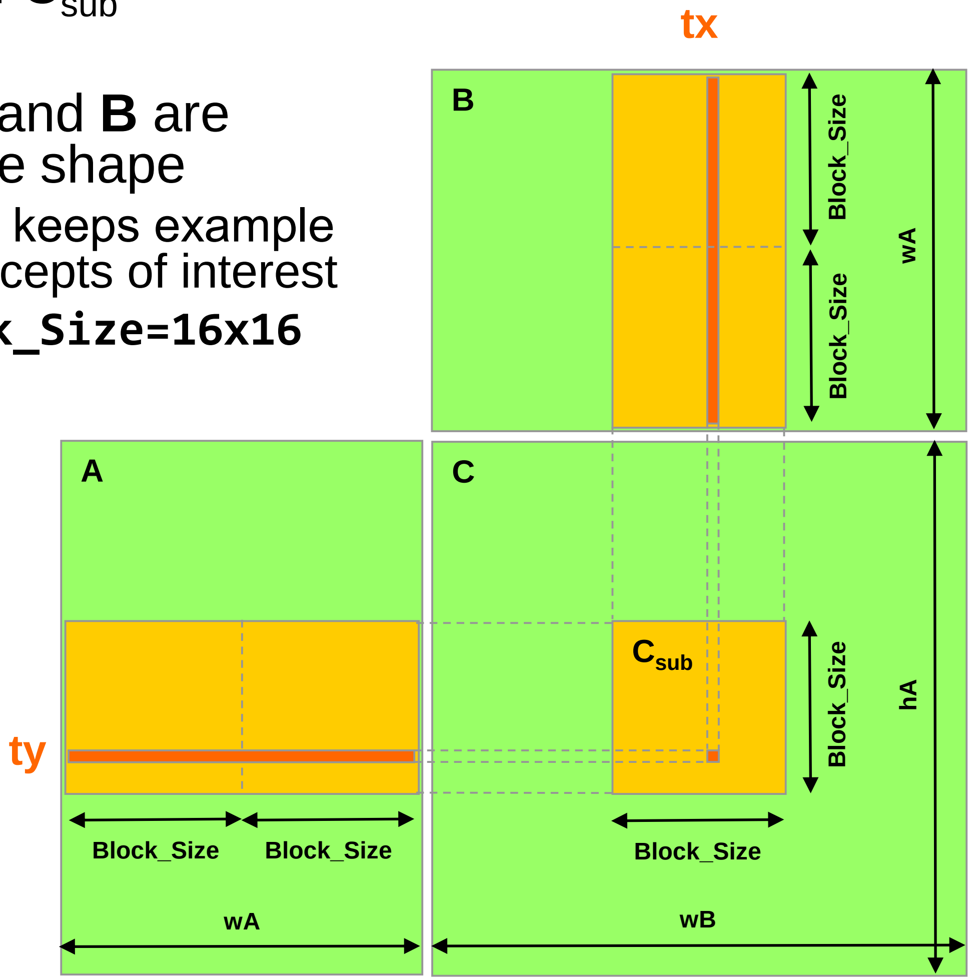
End Programming Job #1

Example 2: Matrix Multiplication

- In the basic implementation, the ratio of arithmetic computation to memory transaction is very low → BAD
  - Each computation required one fetch from global memory
  - Matrix M copied from the global memory to the device `N.width` times
  - Matrix N copied from the global memory to the device `M.height` times

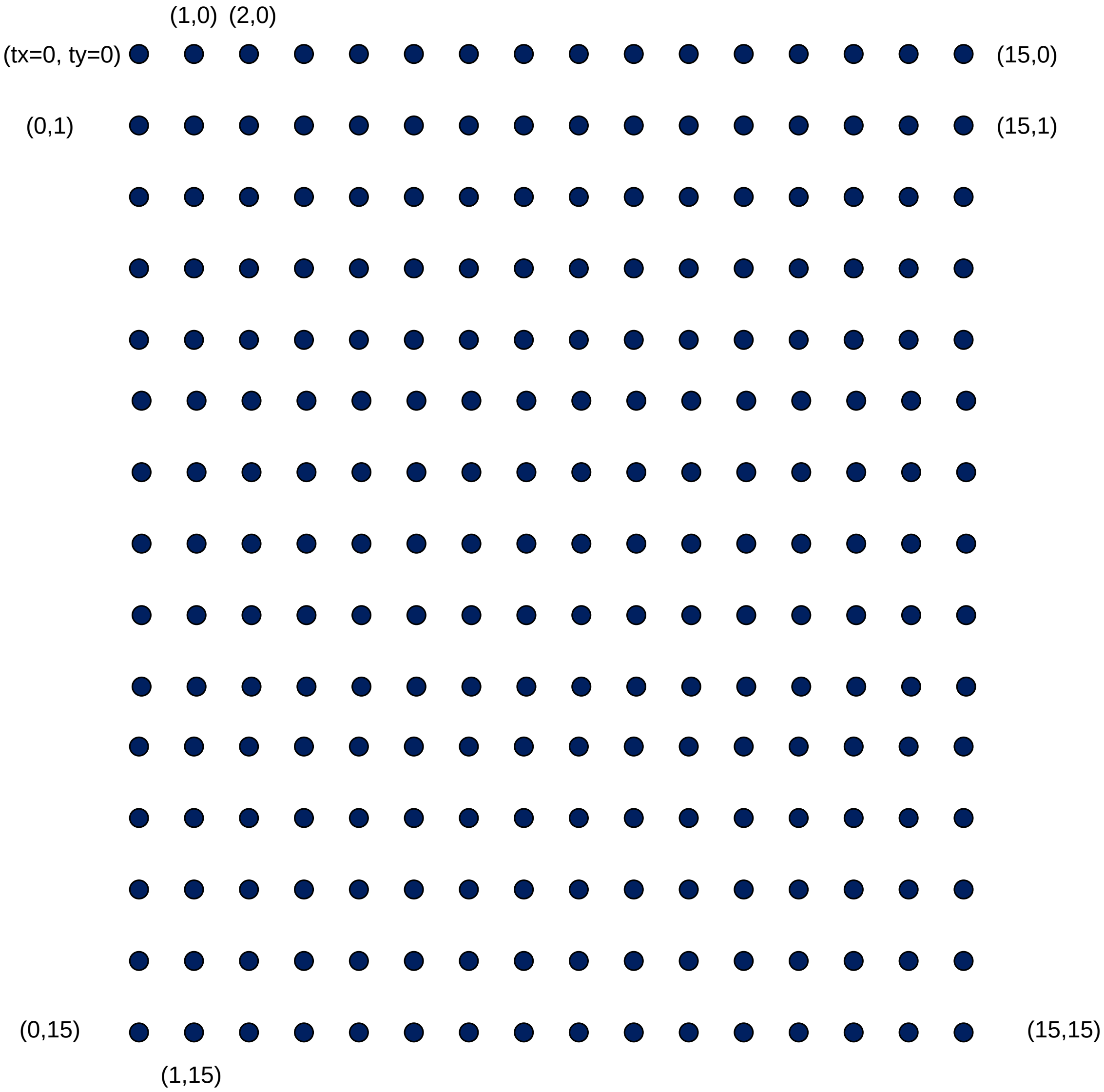
Multiply Using Several Blocks

- One block computes one square sub-matrix **C**$_{sub}$ of size Block_Size
- One thread computes one entry of **C**$_{sub}$
- Assume that the dimensions of **A** and **B** are multiples of **`Block_Size`** and square shape
  - Doesn't have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with **`Block_Size=16x16`**



NOTE: Similar example provided in the CUDA Programming Guide 4.2

A Block of 16 X 16 Threads



```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
  int size;

  // Load A and B to the device
  float* Ad;
  size = hA * wA * sizeof(float);
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

  float* Bd;
  size = wA * wB * sizeof(float);
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
```

(continues with next block…)

(continues below…)

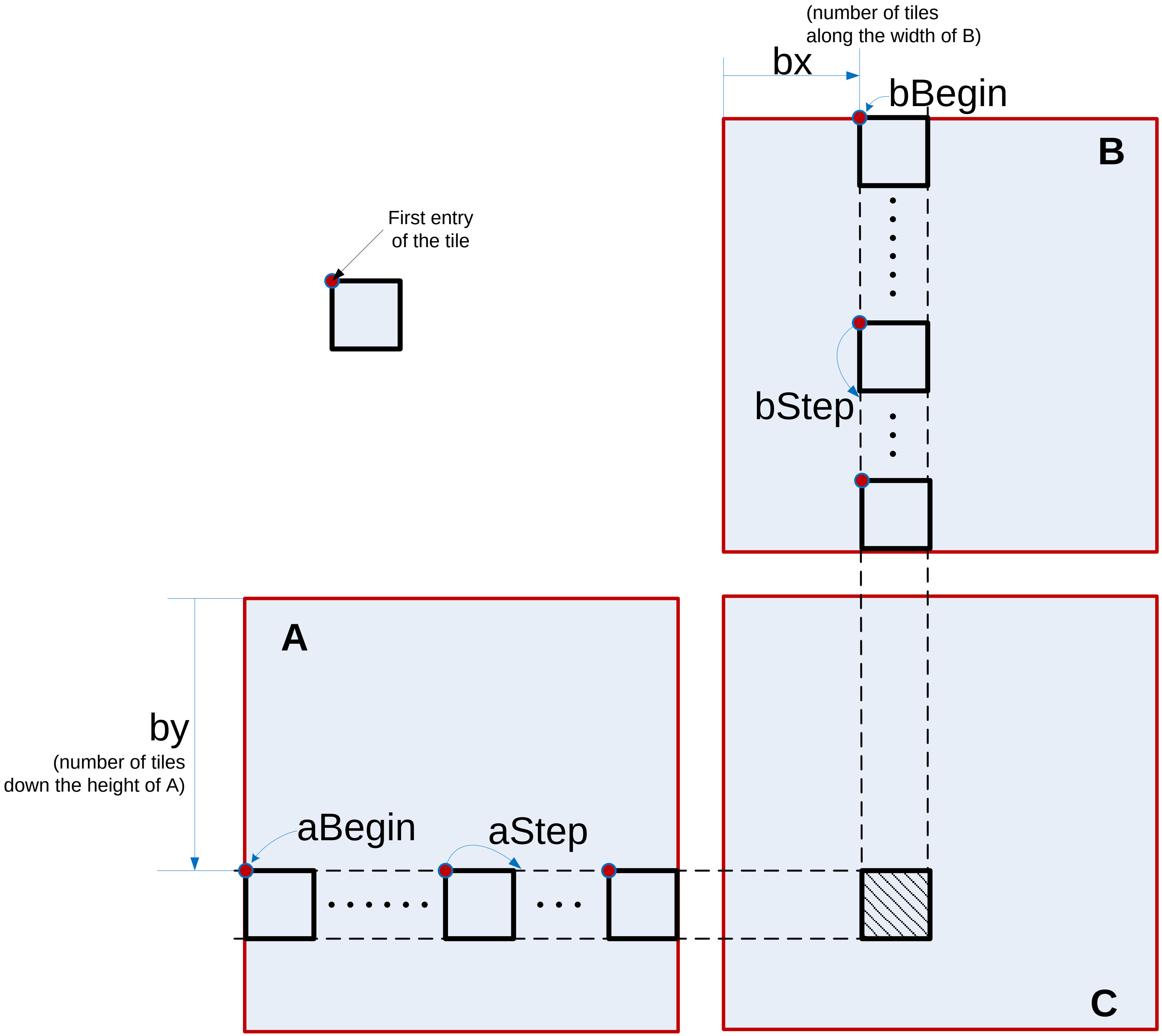
```
  // Allocate C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);

  // Compute the execution configuration assuming
  // the matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}
```


(number of tiles
along the width of B)
bx
bBegin
B
First entry
of the tile
bStep
A
by
(number of tiles
down the height of A)
aBegin
aStep
C

```
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
  // Block index
  int bx = blockIdx.x;  // the B (and C) matrix sub-block column index
  int by = blockIdx.y;  // the A (and C) matrix sub-block row index

  // Thread index
  int tx = threadIdx.x; // the column index in the sub-block
  int ty = threadIdx.y; // the row index in the sub-block

  // Index of the first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;

  // Index of the last sub-matrix of A processed by the block
  int aEnd = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;

  // Index of the first sub-matrix of B processed by the block
  int bBegin = BLOCK_SIZE * bx;

  // Step size used to iterate through the sub-matrices of B
  int bStep = BLOCK_SIZE * wB;

  // The element of the block sub-matrix that is computed
  // by the thread
  float Csub = 0;
```

(continues with next block…)

```
 // Shared memory for the sub-matrix of A
 __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

 // Shared memory for the sub-matrix of B
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  // Loop over all the sub-matrices of A and B required to
  // compute the block sub-matrix
  for (int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {

// Load the matrices from global memory to shared memory;
      // each thread loads one element of each matrix
      As[ty][tx] = A[a + wA * ty + tx];
      Bs[ty][tx] = B[b + wB * ty + tx];

      // Synchronize to make sure the matrices are loaded
      __syncthreads();

      // Multiply the two matrices together;
      // each thread computes one element
      // of the block sub-matrix
      for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];

      // Synchronize to make sure that the preceding
      // computation is done before loading two new
      // sub-matrices of A and B in the next iteration
      __syncthreads();
  }
  // Write the block sub-matrix to global memory;
  // each thread writes one element
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

Example 2: Matrix Multiplication

- To compile this code:

```
>> qsub compile.sh
```

- To run this code:

```
>> qsub submit_example.sh
```

Try to do it alone…

- Remember, CUDA programs have a basic flow:
  1)The host initializes an array with data.
  2)The array is copied from the host to the memory on the CUDA device.
  3)The CUDA device operates on the data in the array.
  4)The array is copied back to the host.

End Programming Job #2