ME964 High Performance Computing for Engineering Applications

Memory Layout in CUDA

February 21, 2012

"Computers are good at following instructions, but not at reading your mind."
Donald Knuth

Before We Get Started…

- Last time
  - API related issues
  - Simple matrix multiplication example
  - Memory allocation, copying, freeing, etc.

- Today
  - Memory layout
  - Revisit matrix multiplication (use of shared memory)
  - Thread execution scheduling in CUDA (if we have time)

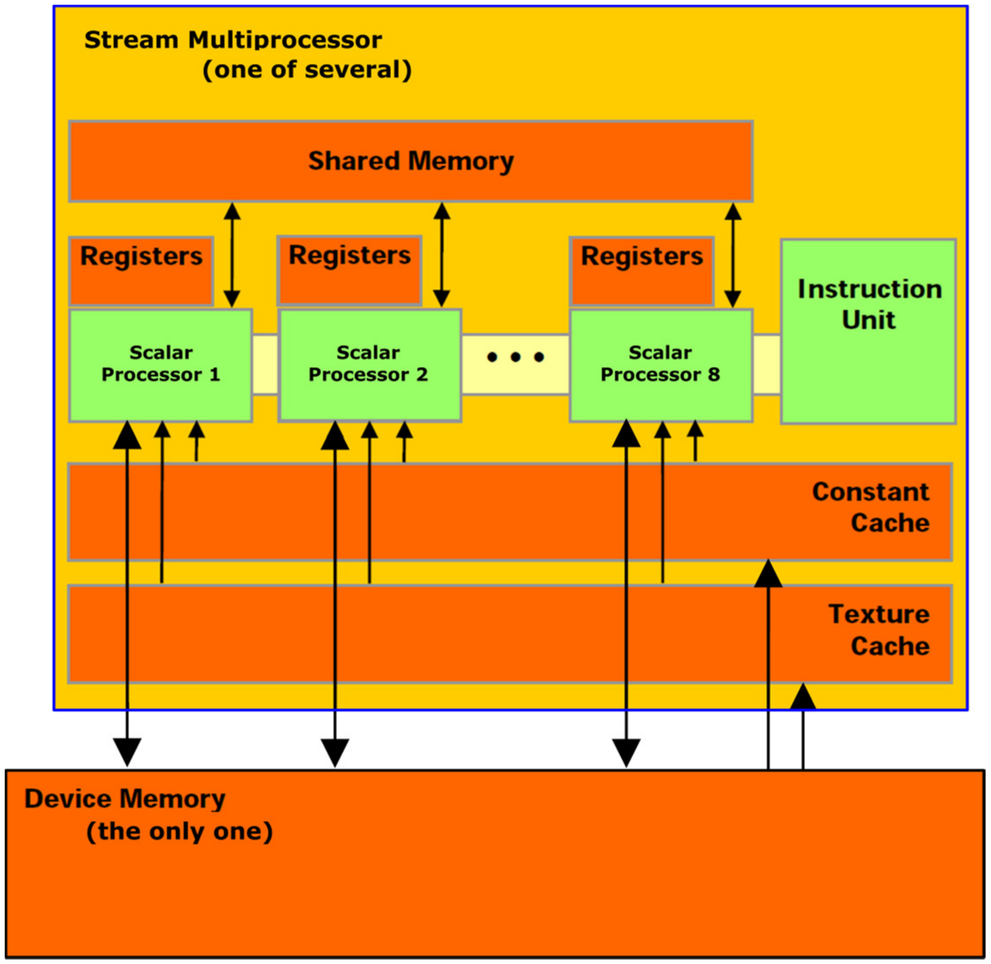
- HW
  - HW4: posted online & due on Th at 11:59 PM

The Memory Ecosystem
[NVIDIA cards specific]

- The memory space is the union of
  - Registers
  - Shared memory
  - Device memory, which can be
    - Global memory
    - Constant memory
    - Texture memory

- Remarks
  - The constant memory is cached
  - The texture memory is cached
  - The global memory is cached only in devices of compute capability 2.X

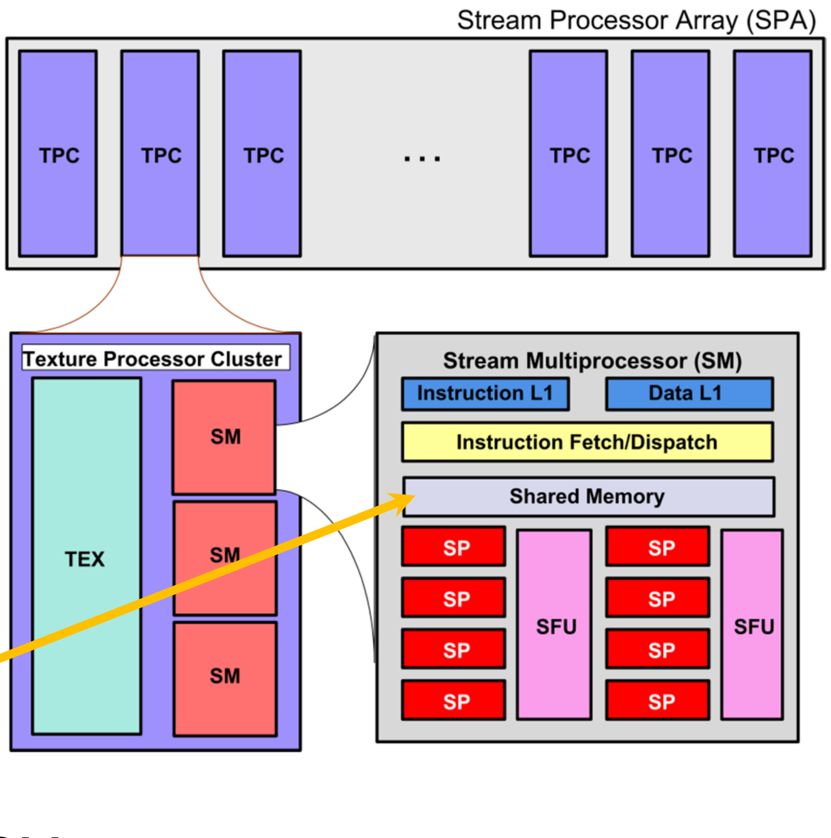
- Mem. Bandwidth, Device Memory:
  - Approx. 140 GB/s



GPU: Underlying Hardware
[Tesla C1060]

$$\mathrm{SPA} \xrightarrow[10]{has} \mathrm{TPC} \xrightarrow[3]{each\ has} \mathrm{SM} \xrightarrow[8]{each\ has} \mathrm{SP}$$

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)

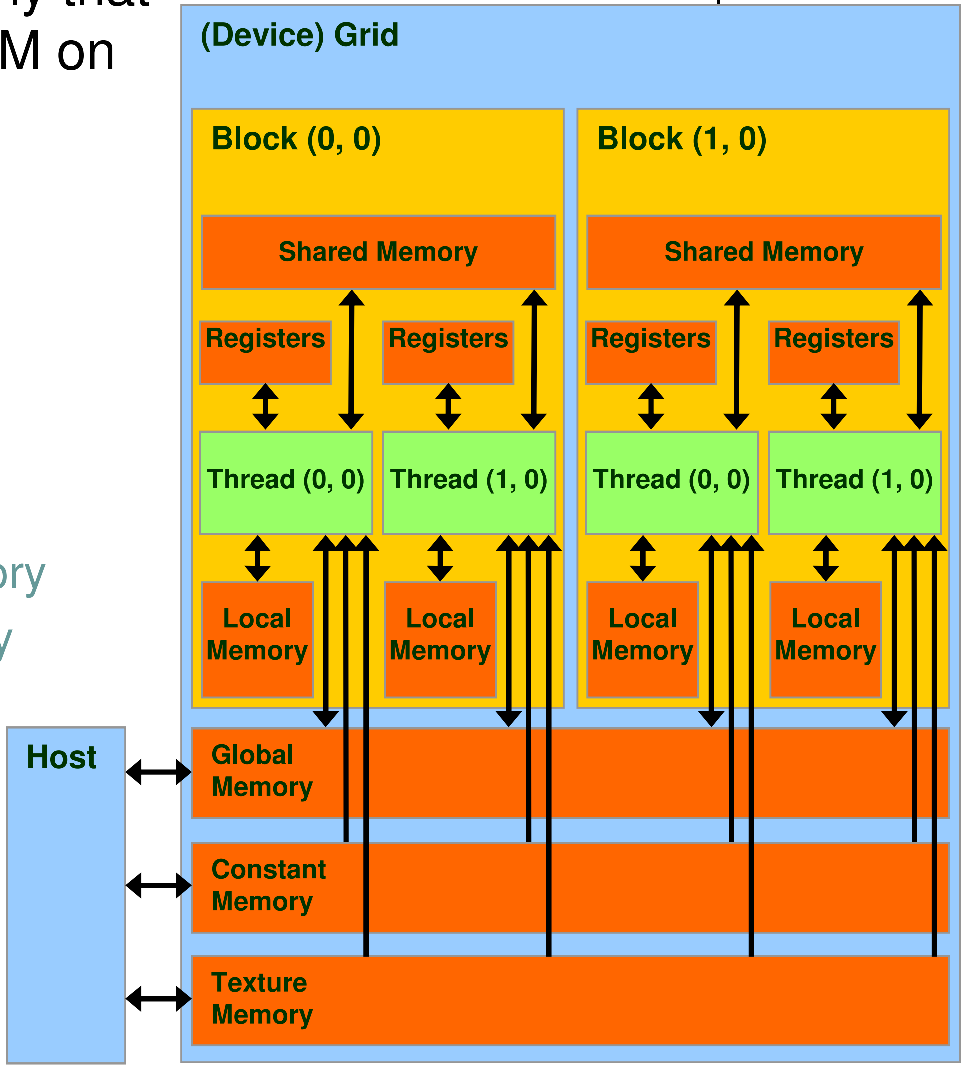

Stream Processor Array (SPA)
TPC
TPC
TPC
. . .
TPC
TPC
TPC
Texture Processor Cluster
TEX
SM
SM
SM
Stream Multiprocessor (SM)
Instruction L1
Data L1
Instruction Fetch/Dispatch
Shared Memory
SP
SP
SP
SP
SFU
SP
SP
SP
SP
SFU

- Look closer…
  - You do see shared memory on the SM
  - You don't see global memory on the SM

CUDA Device Memory Space Overview

[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on a SM on Tesla C1060
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memory

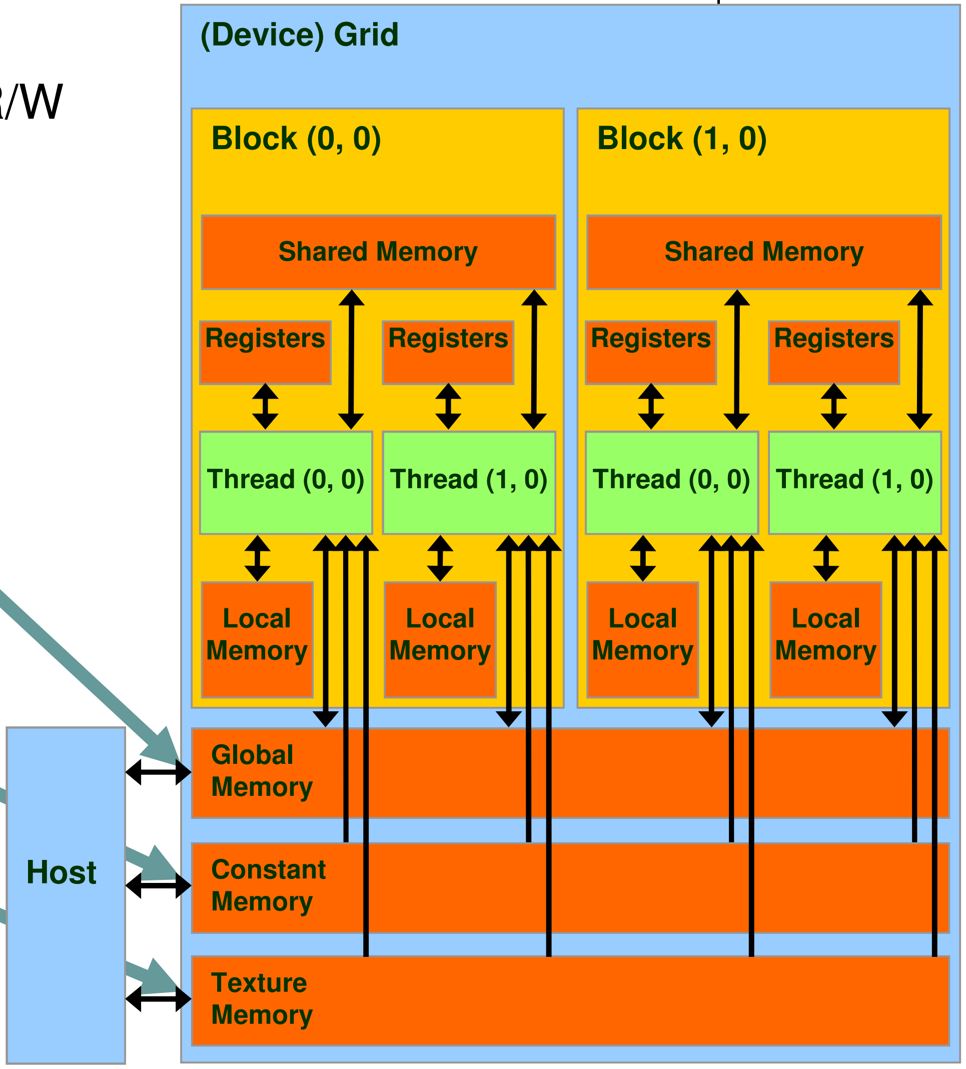


IMPORTANT NOTE: Global, constant, and texture memory spaces are **persistent** across kernels called by the same host application.

Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

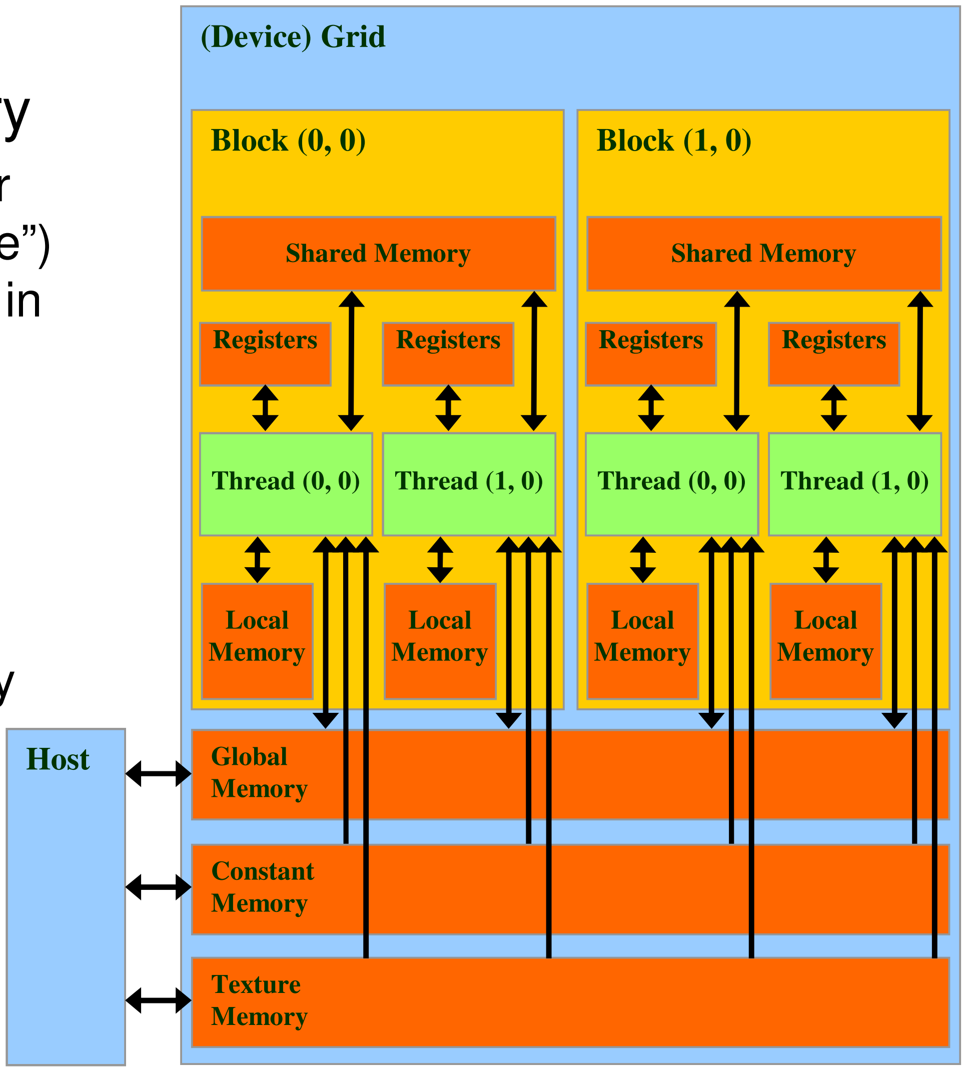
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



NOTE: We will not emphasize texture memory in this class.

The Concept of Local Memory

- Note the presence of local memory, which is virtual memory
  - If too many registers are needed for computation ("high register pressure") the ensuing data overflow is stored in local memory
  - "Local" means that it's local, or specific, to one thread
  - In fact local memory is part of the global memory
  - Long access times for local memory (in Fermi, local memory might be cached)



Memory Space, Tesla C1060
[Compute Capability 1.3]

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

- BTW, off-chip means that’s on-device, which translates into slow access time
- NOTE: Fermi caches local memory, as well as global memory

Access Times [Tesla C1060]

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Matrix Multiplication Example, Revisited

- Purpose
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the **`_syncthreads()`** function call
- NOTE: We use the same one dimensional array to store the entries in the matrix
  - Drawing on the **`Matrix`** data structure discussed in a previous lecture

Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device N.width times
  - The matrix N (its entries) is copied from global memory to the device M.height times
- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things

The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU
- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined
- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up…). You're done.

Rules of Thumb for Efficient GPU Computing:

1. Get the data on the GPU and keep it there
2. Give the GPU enough work to do
3. Focus on data reuse within the GPU to avoid memory bandwidth limitations

A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

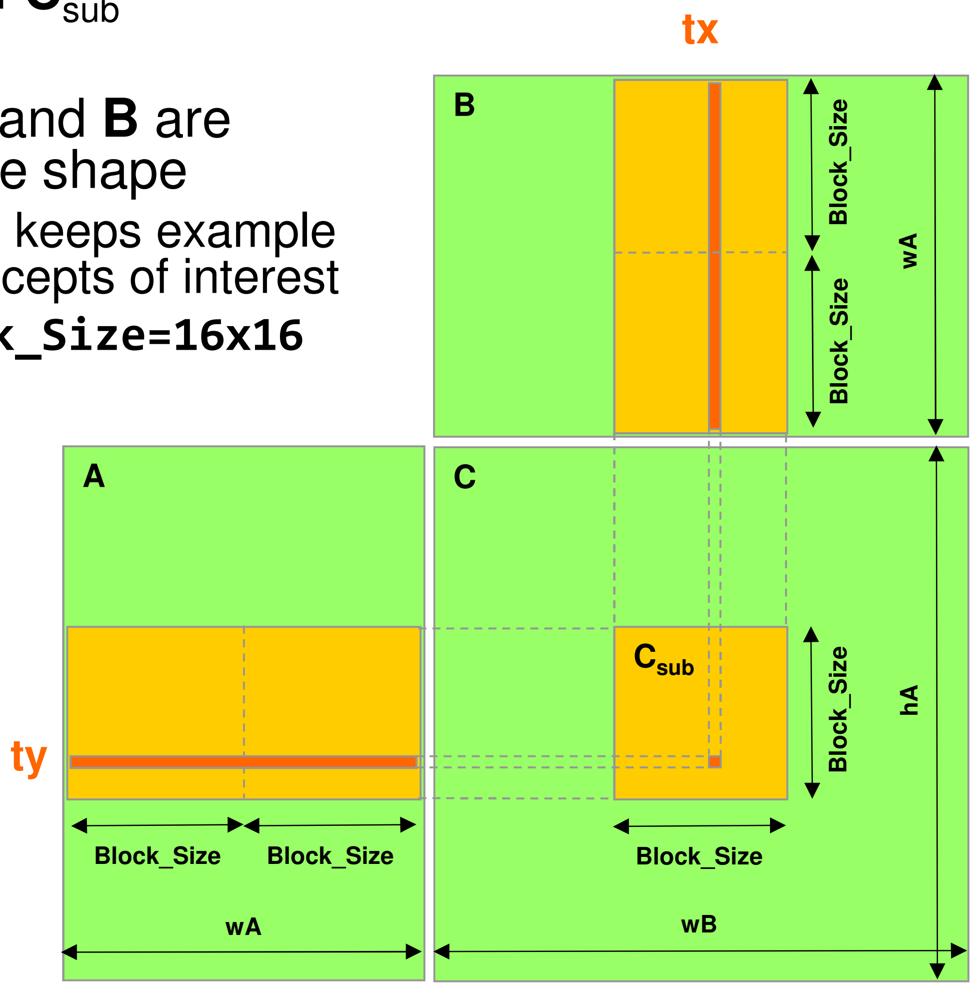
- An advantageous way of performing computation on the device is to partition ("tile") data to take advantage of fast shared memory:

    - Partition data into data subsets (tiles) that each fits into shared memory

    - Handle each data subset (tile) with one thread block by:
        - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
        - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
        - Copying results from shared memory back to global memory

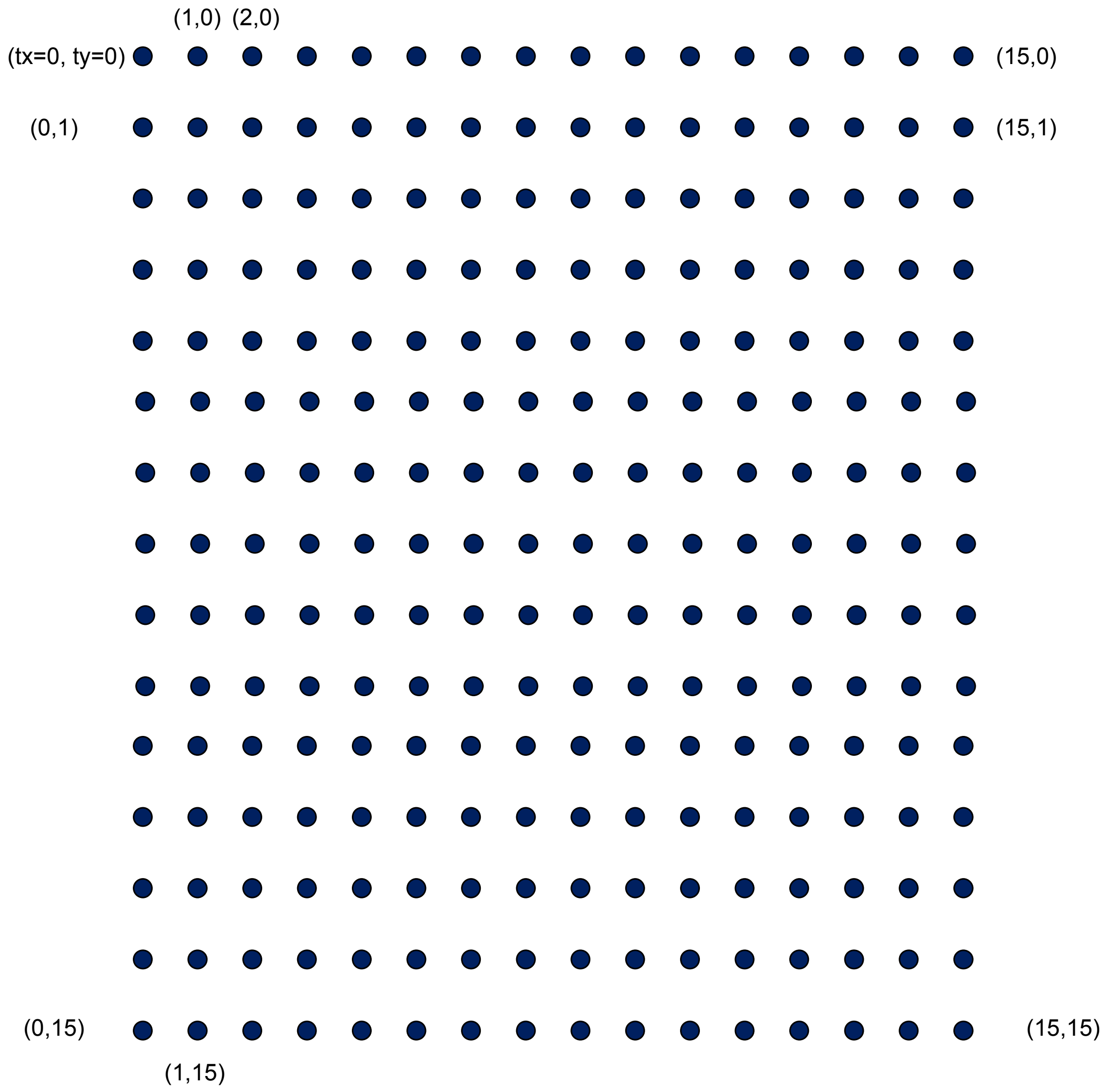
Multiply Using Several Blocks

- One block computes one square sub-matrix $\mathbf{C}_{sub}$ of size Block_Size
- One thread computes one entry of $\mathbf{C}_{sub}$
- 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 3.2
• Available on the 2011 class website

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…)

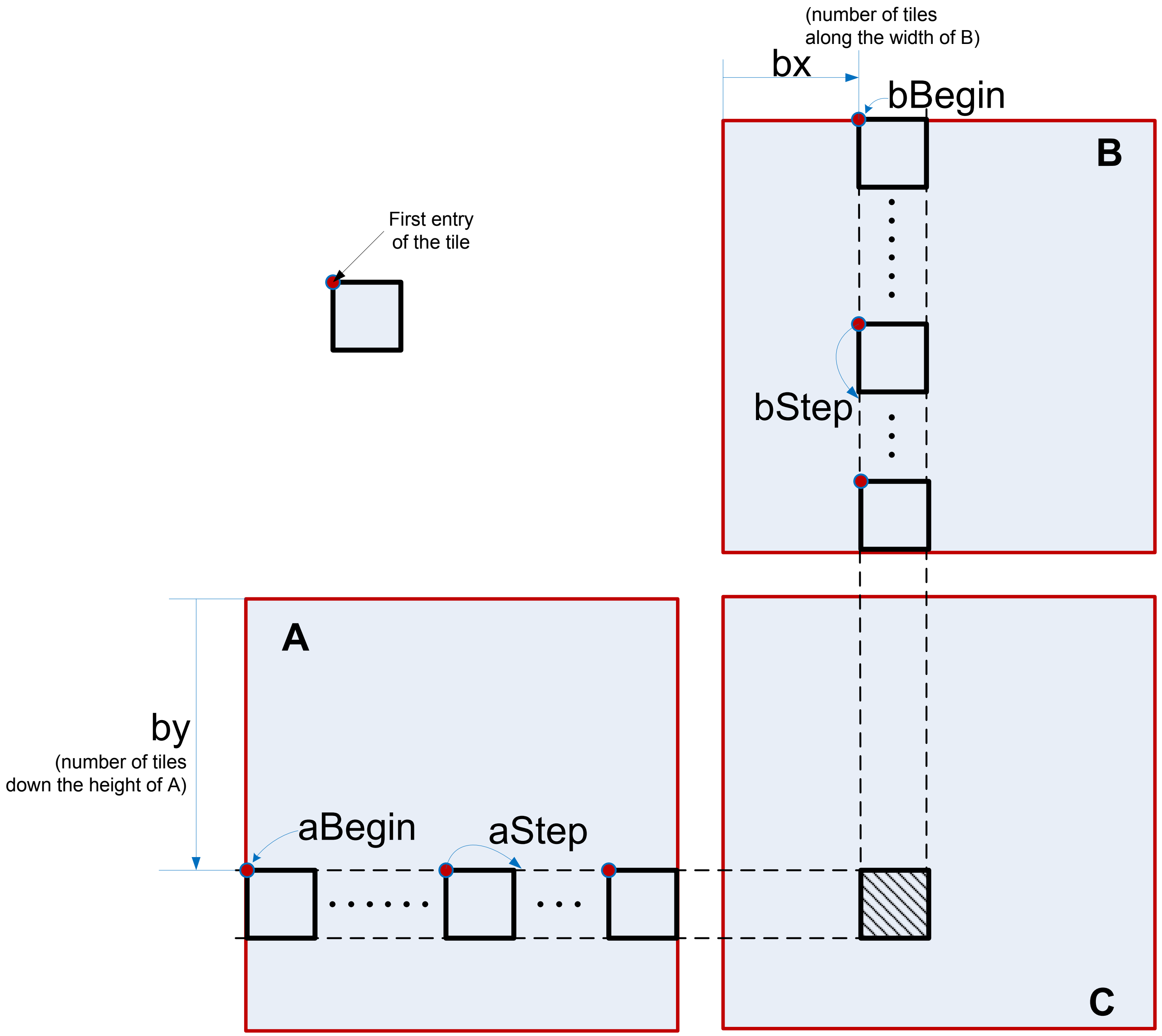
```
  // Allocate C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);

  // 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;
}
```

Synchronization Function

- It’s a device lightweight runtime API function
  - `void __syncthreads();`
- Synchronizes all threads in a block (acts as a barrier for all threads of a block)
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block