ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

The NVIDIA GPU Memory Ecosystem

October 5, 2015

ECE/ME/EMA/CS 759 UW-Madison

Quote of the Day

“Do you see over yonder, friend Sancho, thirty or forty hulking giants? I intend to do battle with them and slay them.”

— Miguel de Cervantes Saavedra, Don Quixote (1605)

Before We Get Started

- Issues covered last time:
  - Scheduling for execution on the NVIDIA GPUs

- Today's topics
  - The CUDA memory ecosystem
  - [Atomic operations]

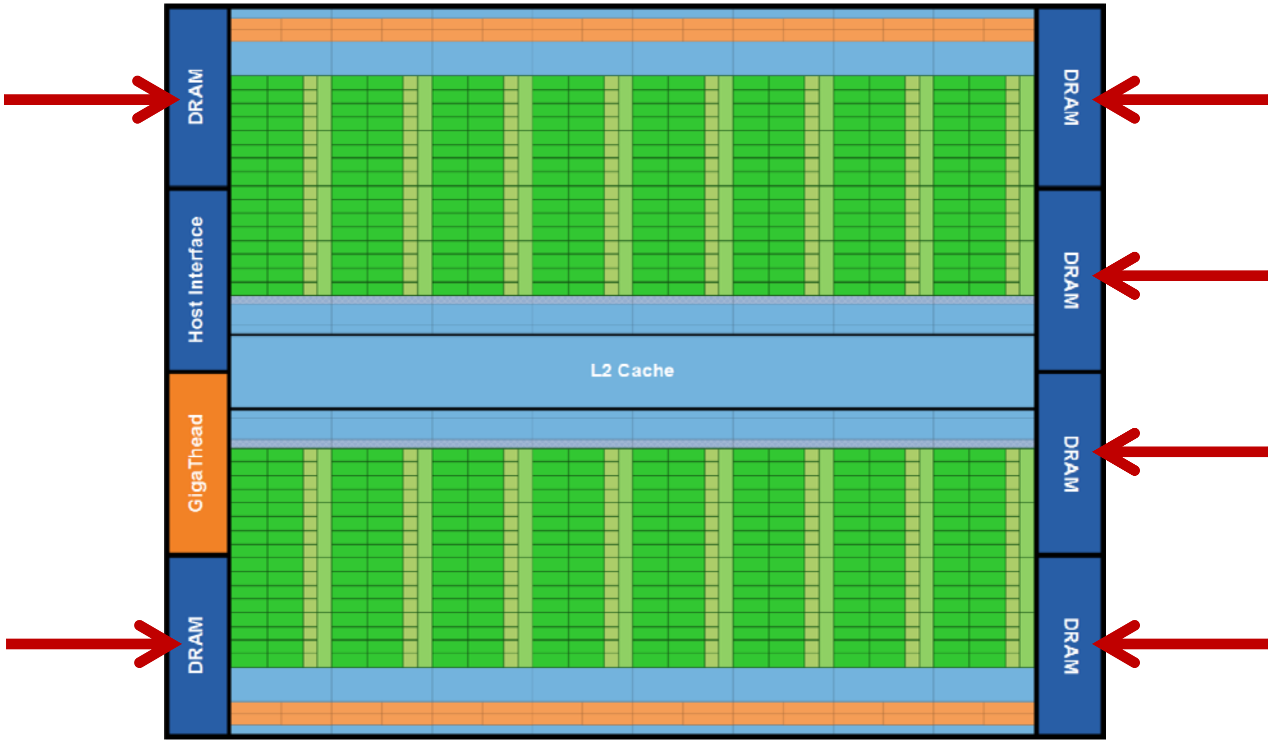
- Assignment:
  - HW04 –due on Oct. Oct. 7 at 11:59 PM

- Midterm Exam: 10/09 (Friday)
  - Review on Th 10/08, at 7:15 PM, room TBA
  - Exam is "open everything"
    - Not allowed to communicate with anybody during duration of the exam
    - Having a laptop/table is ok
    - Exam made up of a collection of questions

The Memory Ecosystem

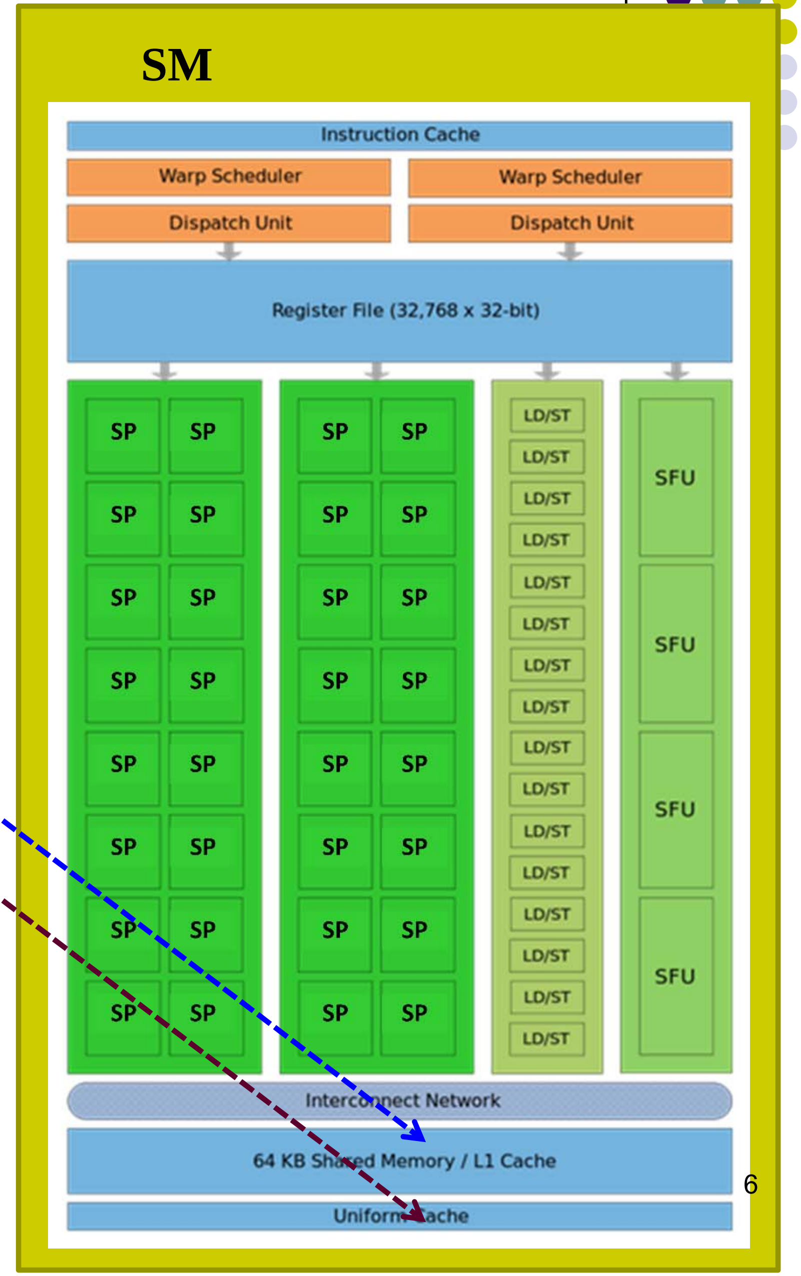
Fermi: Global Memory

- Up to 6 GB of “global memory”
- “Global” in the sense that it doesn’t belong to an SM but rather all SM can access it



The Fermi Architecture

- 64 KB L1 cache & shared memory
- 768 KB L2 uniform cache (shared by all SMs)
- Memory operates at its own clock rate
- High memory bandwidth
  - Close to 200 GB/s



CUDA Device Memory Space Overview

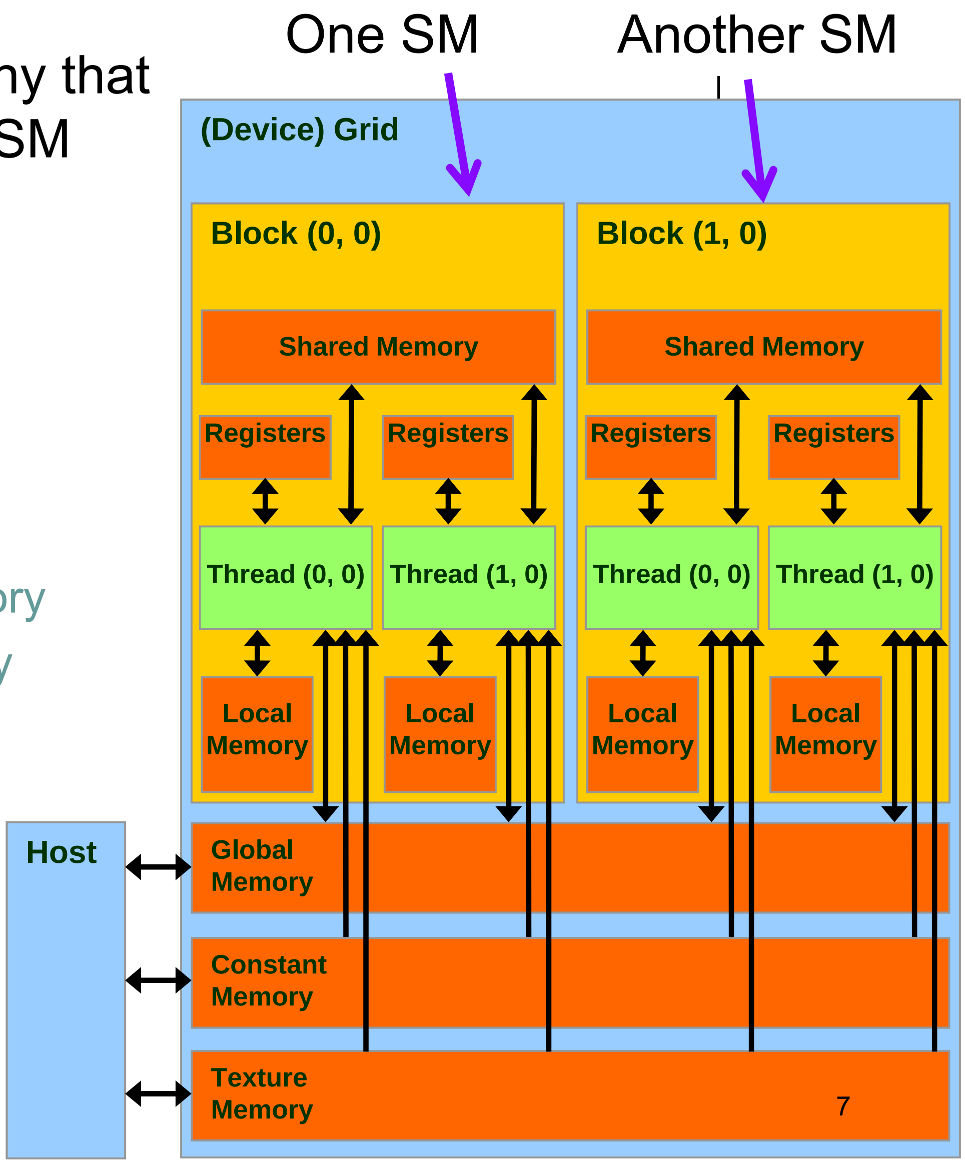
[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on an SM

- 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

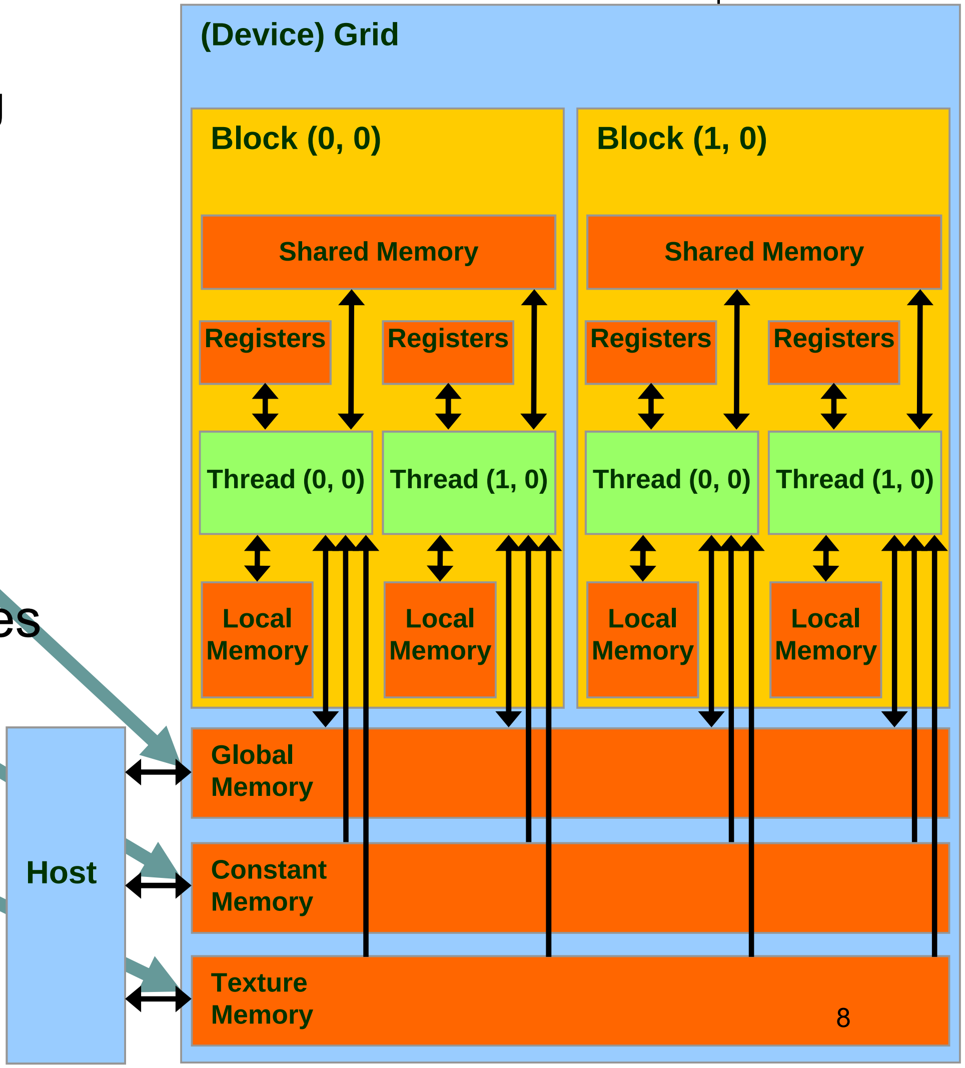
IMPORTANT NOTE: Global, constant, and texture memory spaces are **persistent** between kernels called by the same host application.



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

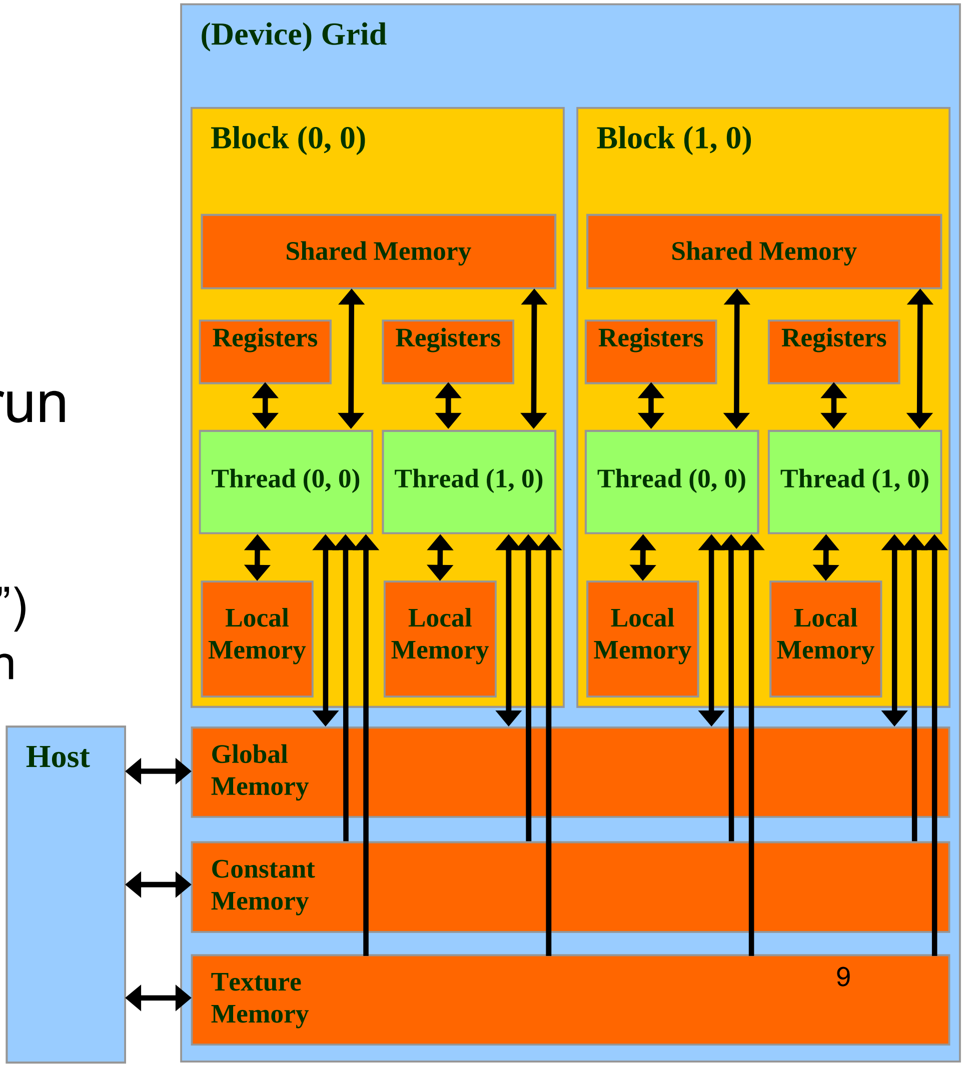
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



NOTE: We will not emphasize texture here.

The Concept of Local Memory

- Local memory does not exist physically
  - “Local” in scope but not in location
- Data that is stored in “local memory” is actually placed in cache or the global memory at run time or by the compiler.
  - 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 got local scope; i.e., it’s specific to one thread
  - Long access times for local memory (on Fermi, local memory is cached)



Storage Locations

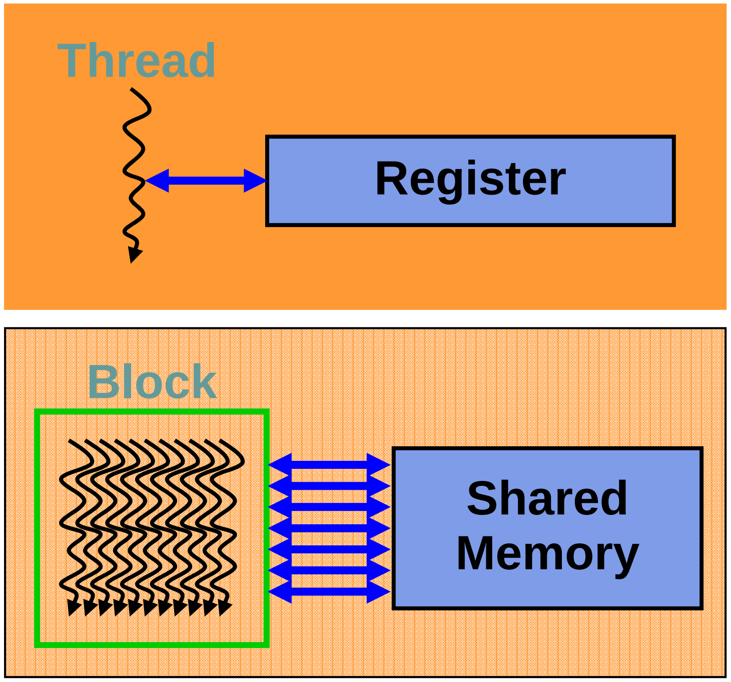
Memory	Location	Cached	Access	Who
Register	On-chip	N/A	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	Yes	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Off-chip means not on the SM; i.e., slow access time.

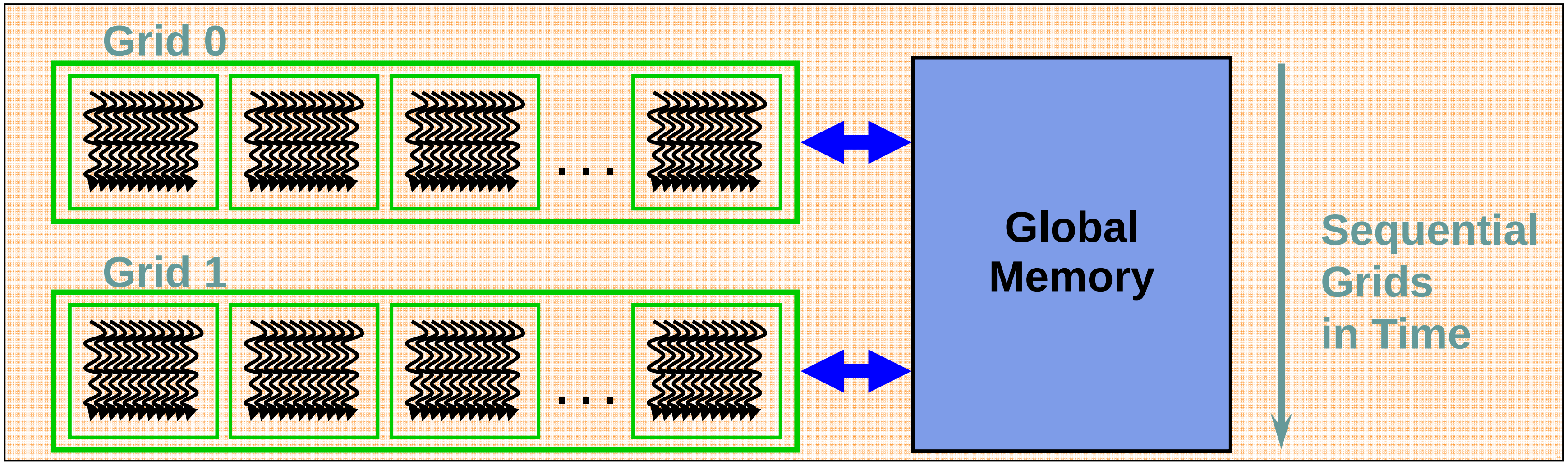
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM: *fast* if cached, otherwise very slow
- Global Memory – DRAM: *slow* (unless if cached)
- 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

The Three Most Important Parallel Memory Spaces



- Register: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit unless cached)
- Shared Memory: per-block basis
  - Shared by threads of the same block
  - Used for: intra-block inter-thread communication
- Global Memory: per-application basis
  - Available for use by all threads
  - Used for: global access, all threads
  - Also used for inter-kernel communication

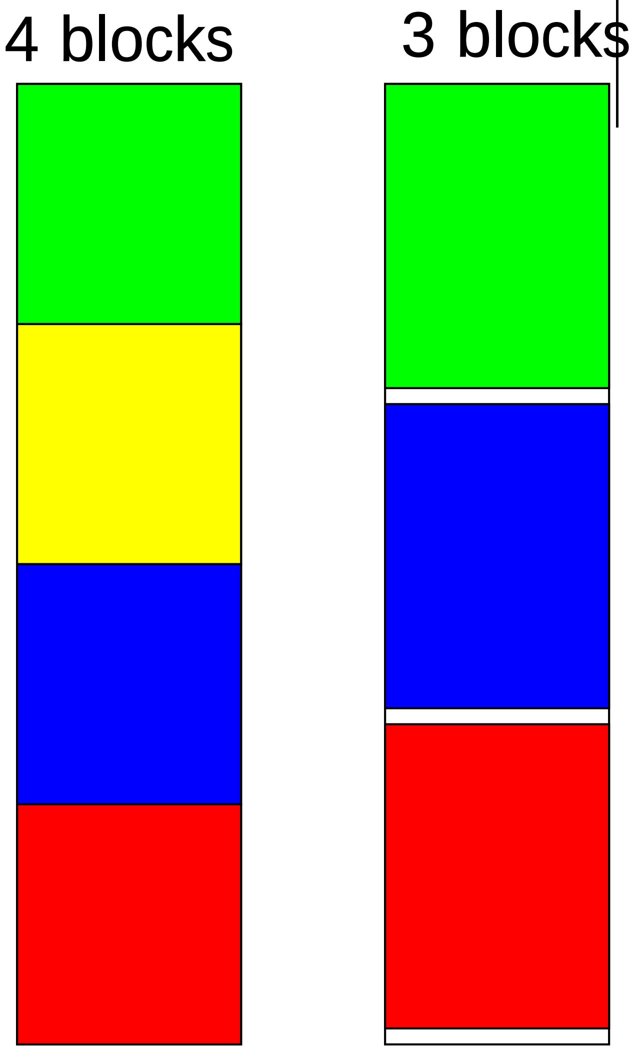


Coming Up Next

- Talk about these three memory spaces
  - Register file
  - Shared Memory
  - Global Memory

Programmer View of Register File

- Number of **32 bit** registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla
  - 32K on Fermi
  - 64K on Kepler and Maxwell
- Registers are dynamically partitioned across all Blocks assigned to the SM
- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks
- A thread in a Block can only access registers assigned to itself
  - Kepler and Maxwell: a thread can have assigned by the compiler up to 255 registers



Possible per-block partitioning scenarios of the RF available on the SM

Shared Memory Discussion

[via revisiting the old Matrix Multiplication Example]

- Purpose of Matrix Multiplication Example
  - See an example where the use of multiple blocks of threads plays a central role
  - Understand, through an example, the use/role of the Shared Memory
  - Emphasize the need for the `_syncthreads()` function call
- NOTE: A one dimensional array stores the entries in the matrix

Why Revisit the Matrix Multiplication Example?

- Matrix Multiplication: In the naïve first implementation, ratio of arithmetic computation to memory transaction (“arithmetic intensity”) 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
- Common sense observation: 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
  - Moving data around: takes a lot of time and advances execution by one instruction only

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:

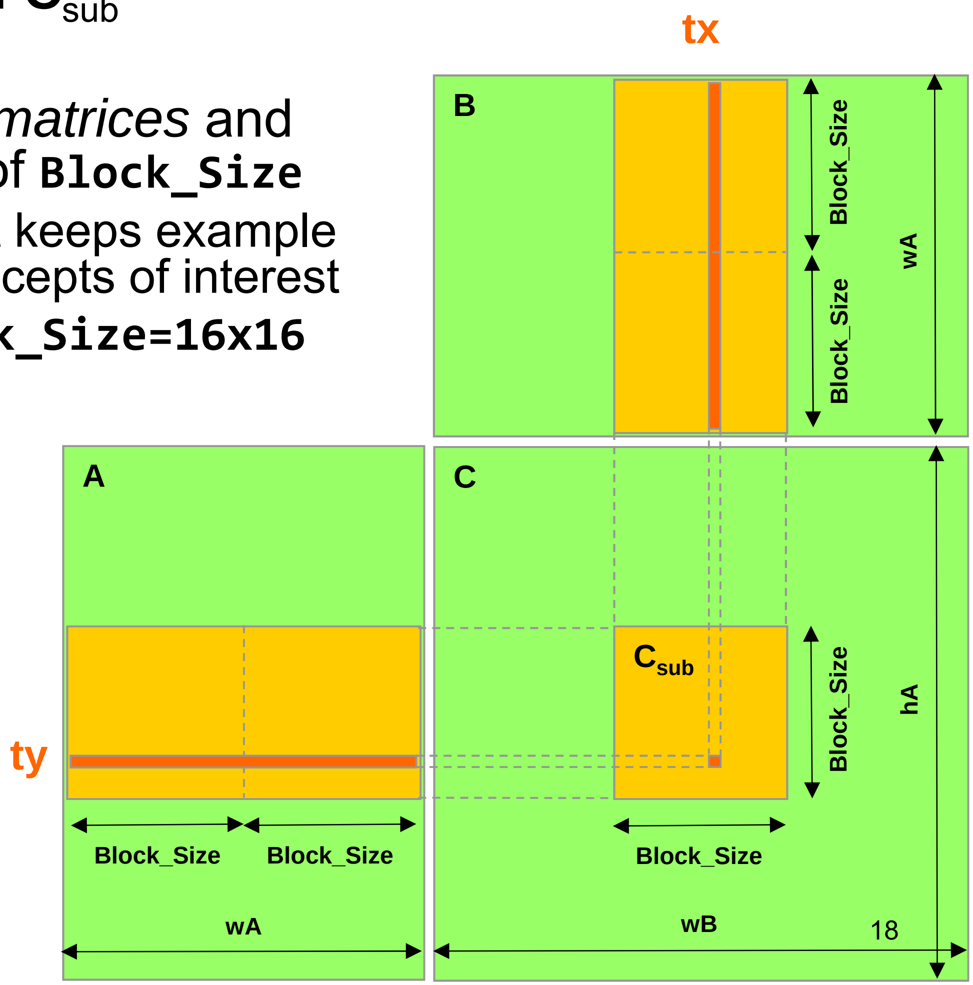
  - Partition data into data subsets (tiles) that each fits into shared memory

  - 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

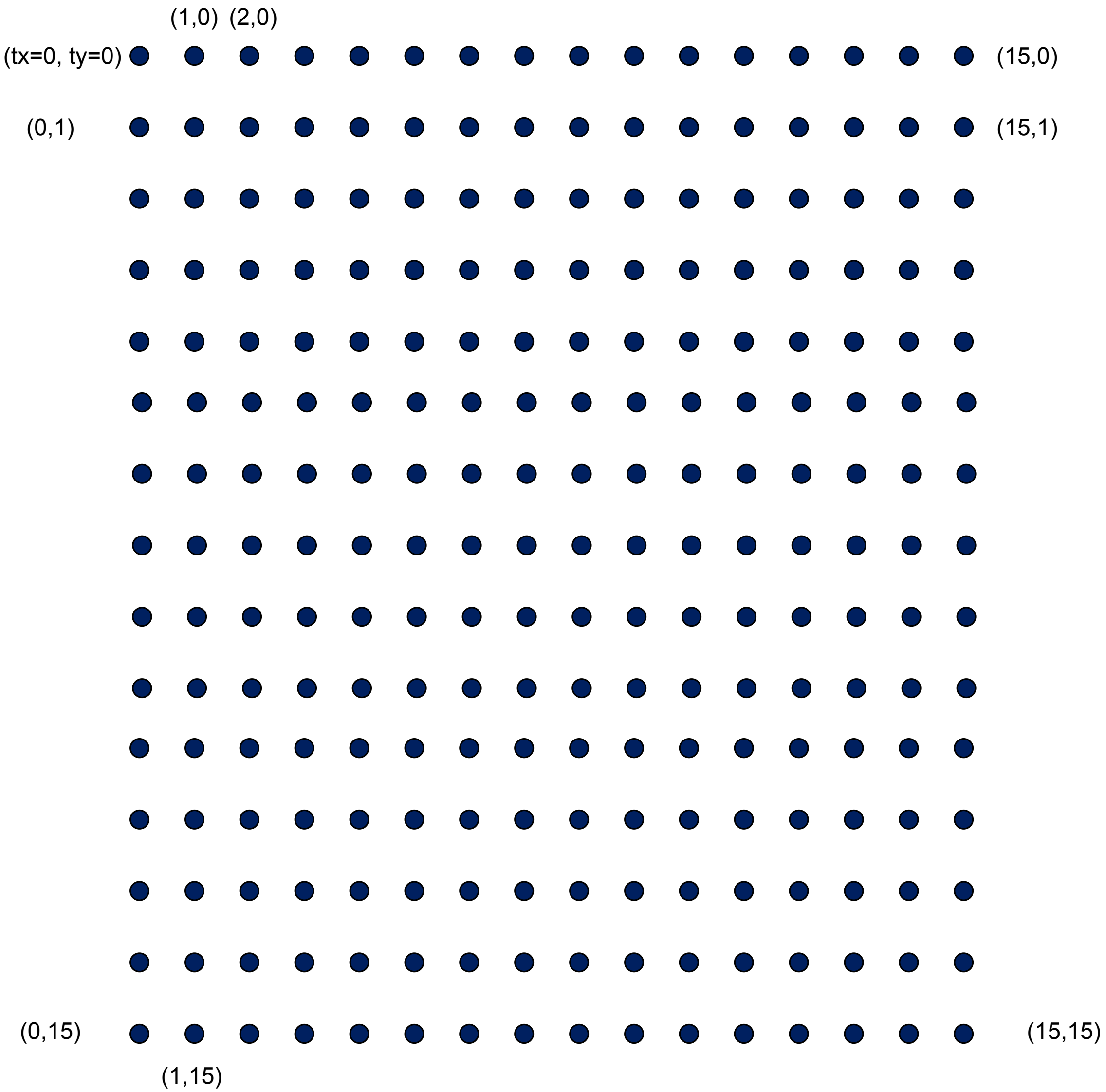
Multiply Using Several Blocks

- One block computes one square sub-matrix **C_{sub}** of size Block_Size
- One thread computes one entry of **C_{sub}**
- Assumption: **A** and **B** are *square matrices* and their dimensions of are *multiples* of `Block_Size`
  - 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: A similar technique is used on CPUs to improve cache hits. See slide "Blocking Example" at http://cseweb.ucsd.edu/classes/fa10/cse240a/pdf/08/CSE240A-MBT-L15-Cache.ppt.pdf



A Block of 16 X 16 Threads



Code on the host side

```
// 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…)

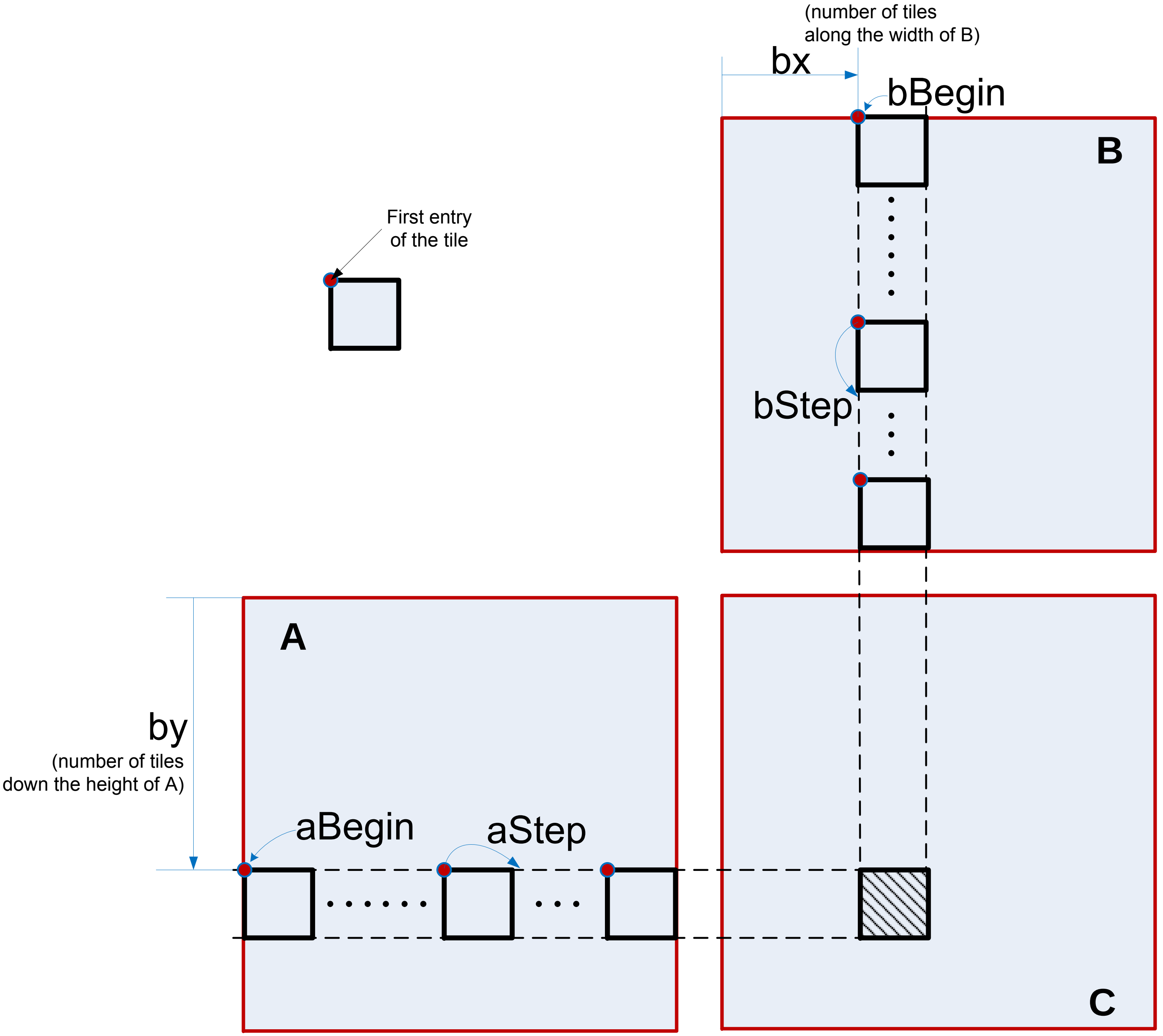
```
  // Allocate C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);

  // 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 accumulator = 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)
      accumulator += 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] = accumulator;
}
```

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)
  - Does **not** synchronize threads from two different blocks
- 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

Short detour, on answering this question: "Does it make sense to use Shared Memory?"

- Imagine you are a thread and execute the kernel

- If data that you use turns out that can be used by any other thread in your block then you should consider using shared memory

- Note: if nothing else, you can use shared memory as scratch pad memory
  - Don't let it go wasted… use it as "quasi-registers" (that you control)

Memory Facts, Fermi GPUs

- There is 64 KB of fast memory on each SM that gets split between L1 cache and Shared Memory
  - You can split 64 KB as “L1/Sh: 16/48” or “L1/Sh: 48/16”

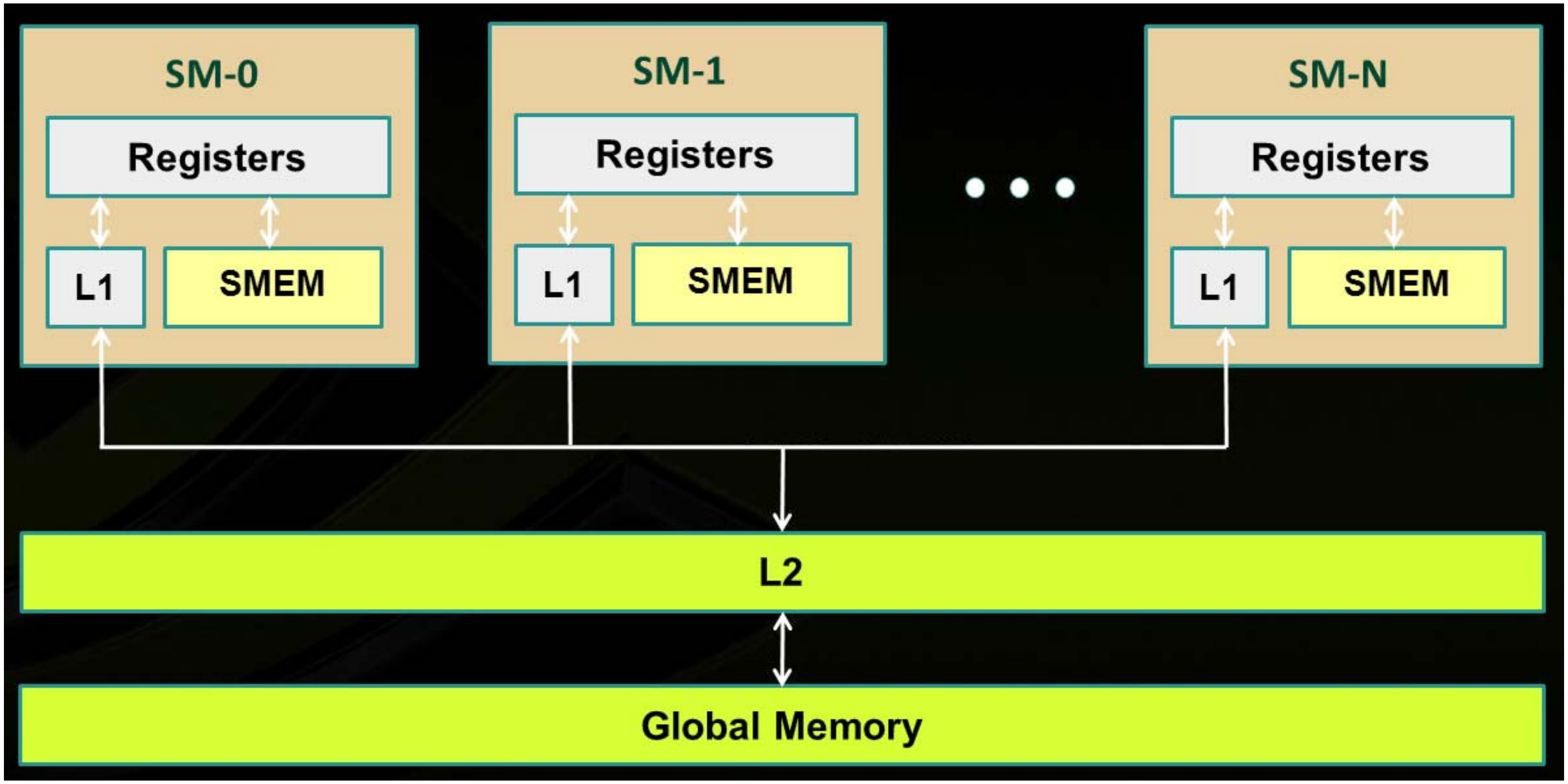
- L2 cache: 768 KB – one big pool available to *all* SMs on the device

- L1 and L2 cache used to cache accesses to
  - Local memory, including register spill
  - Global memory

- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction

Fermi Memory Layout

[credits: NVIDIA]



More Memory Facts
[Fermi GPUs]

- All global memory accesses are cached

- A cache line is 128 bytes
  - It maps to a 128-byte aligned segment in device memory
  - Note: it so happens that 128 bytes = 32 (warp size) * 4 bytes
    - In other words, 32 floats or 32 ints can be brought over in fell swoop

- If the size of the type accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently

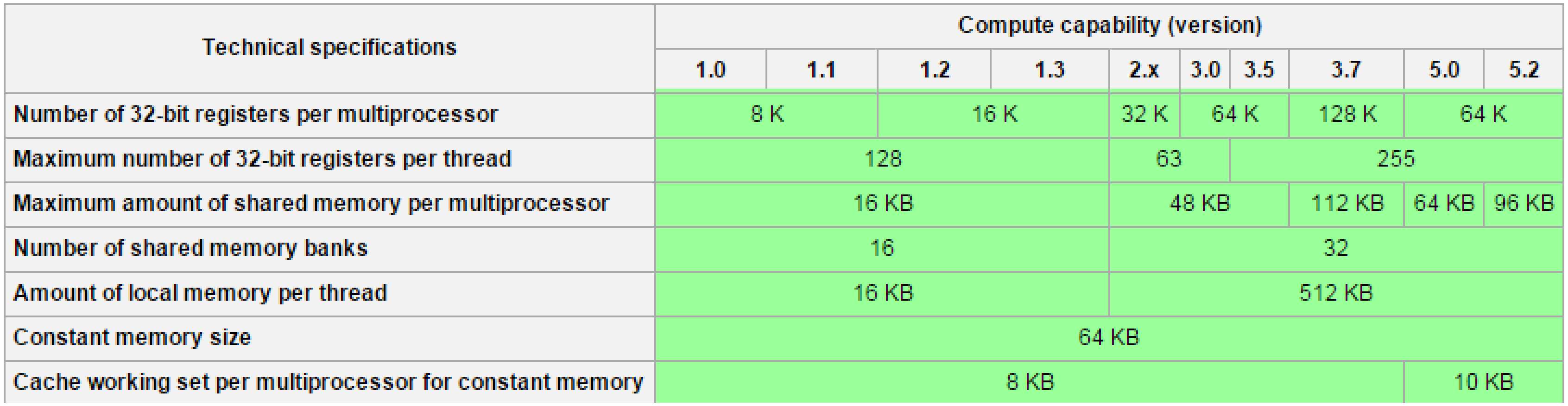
More Memory Facts
[Fermi GPUs]

- The memory access schema is as follows:
  - Two memory requests, one for each half-warp, if the size of data manipulated by a thread is 8 bytes
  - Four memory requests, one for each quarter-warp, if the size of data manipulated by a thread is 16 bytes

- Each memory request is then broken down into cache line requests that are issued independently

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise

Technical Specifications and Features

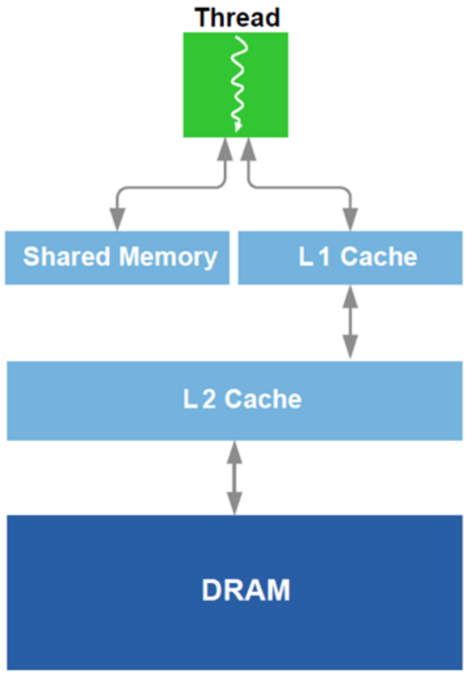


Technical specifications	Compute capability (version)									
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		128 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255			
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			112 KB	64 KB	96 KB
Number of shared memory banks	16				32					
Amount of local memory per thread	16 KB				512 KB					
Constant memory size	64 KB									
Cache working set per multiprocessor for constant memory	8 KB								10 KB	

[Wikipedia]→

The Cache vs. Shared Mem. Conundrum

- On Fermi and Kepler you can split “fast memory banks” between shared memory and cache



- Fermi: you can go 16/48 or 48/16 KB for ShMem/Cache
- Lots of Cache & Little ShMem:
  - Cache handled for you by the scheduler
  - No control over it
  - Can’t have too many blocks of threads running if blocks use ShMem
- Lots of ShMem & Little Cache:
  - Good in tiling, if you want to have full control
  - ShMem pretty cumbersome to manage

Memory Issues Not Addressed Yet…

- Not all *global* memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question
  - Discussed next

- Not all *shared* memory accesses are equivalent
  - How can you optimize shared memory accesses?
  - Moderately relevant questions
  - Not discussed in this course