Code of the Day

- Can you re-write the following code to a branchless version?

```
for (int i = 0; i < N; i++) {
    if (a[i] < 50) {
        s += a[i];
    }
}
```

~14 clock cycles per elements

```
for (int i = 0; i < N; i++) {
    s += (a[i] < 50) * a[i];
}
```

~7 clock cycles per elements

ECE 455

GPU Algorithm and System Designs

[Fall 2025]

Shared Memory and Thread Synchronization

10/13/2025

Before we get started…

- Quick overview, things discussed last time
  - Kernel scheduling: Device level and SM level

- Purpose of today's lecture:
  - Shared memory & caches
  - The `__synchtreads()` function
  - Issues related to the global memory
  - Local memory, constant memory & other aspects related to the movement of data between GPU & CPU

- Miscellaneous
  - Proposal due project proposal – due on **10/17 at 23:59 PM**
  - Lab assignment #5 – due **10/17 at 23:59 PM**

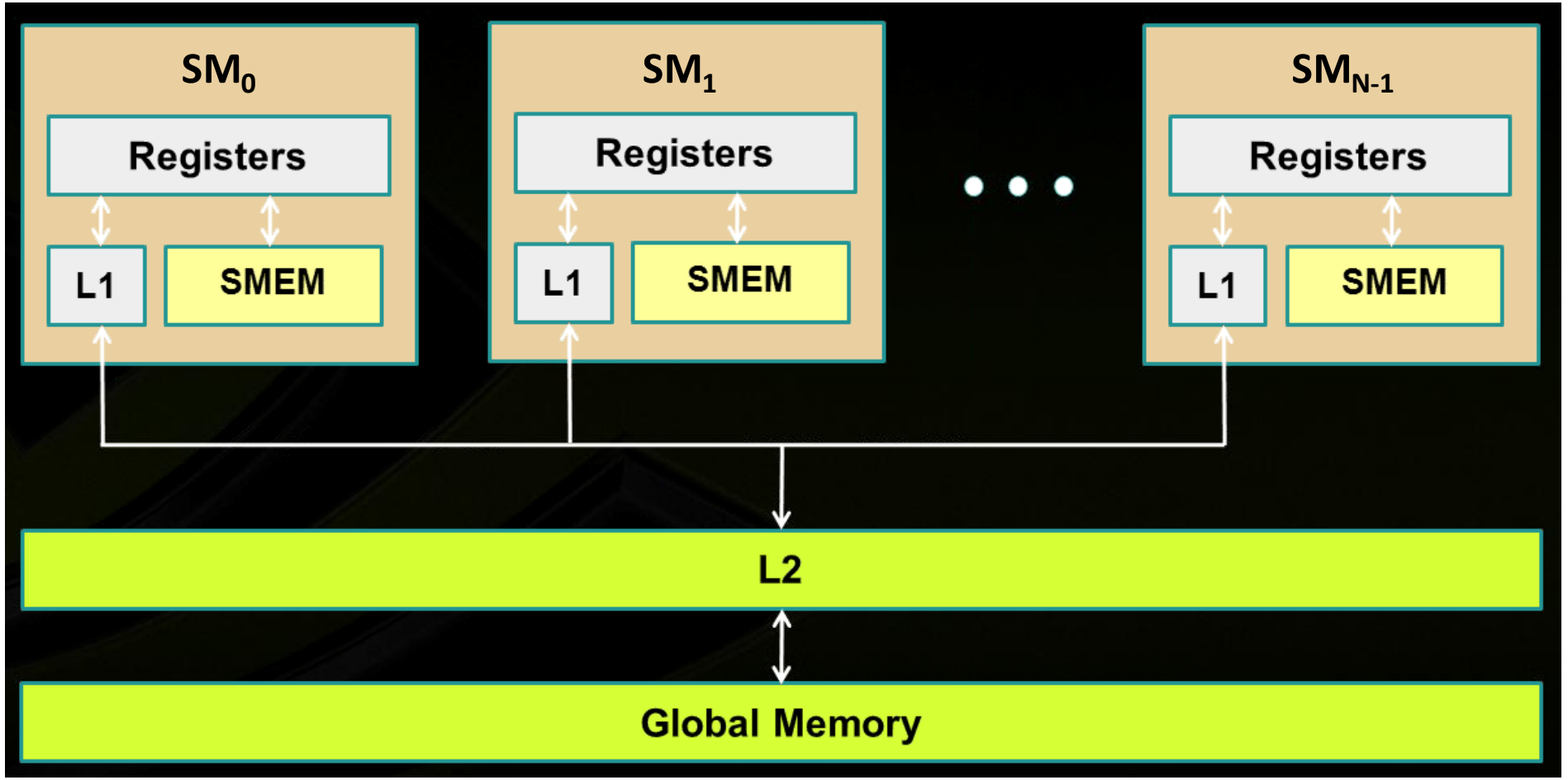
Caches on today's GPU

- Caches on the GPU come in two flavors: L1 and L2
  - There is no L3 cache on the GPU
  - CPU typically has a three-level cache structure: L1, L2, and L3

- Per SM we have:
  - L1 cache is per SM (like L1 cache on CPU is specific to a core) – used to automatically cache global memory loads and stores, providing faster access to frequently used data
  - Shared memory – a programmer-controlled memory space shared among threads within the same thread block, allowing threads to cooperate and share data without the need to access slower global memory

- L2 cache is per device (called uniform memory; on the CPU, we'd call it Last Level Cache)
  - Temporarily store frequently accessed data from global memory to reduce latency

Memory Layout: All GPU units communicate to global memory through L2 cache



Note: shared memory is not cached (a programmer-controlled memory space shared among threads within the same thread block)

L1 cache vs. the Shared Memory

- Physically, there is no difference between the L1 cache and shared memory
  - The bigger the shared memory you use, the fewer the L1 cache is configured, and vice versa
- Logically, there is a distinction between the L1 cache and shared memory
  - That is, they are handled differently by the runtime
- Caches (both L1 & L2): controlled by the runtime
- Shared memory: controlled by you (think of it as “fast scratchpad memory”)
  - A type of high-speed, user-managed memory located on-chip
  - Unlike cache memory, which is managed automatically by the hardware, scratchpad memory is managed explicitly by the programmer or compiler

Summary of the Difference

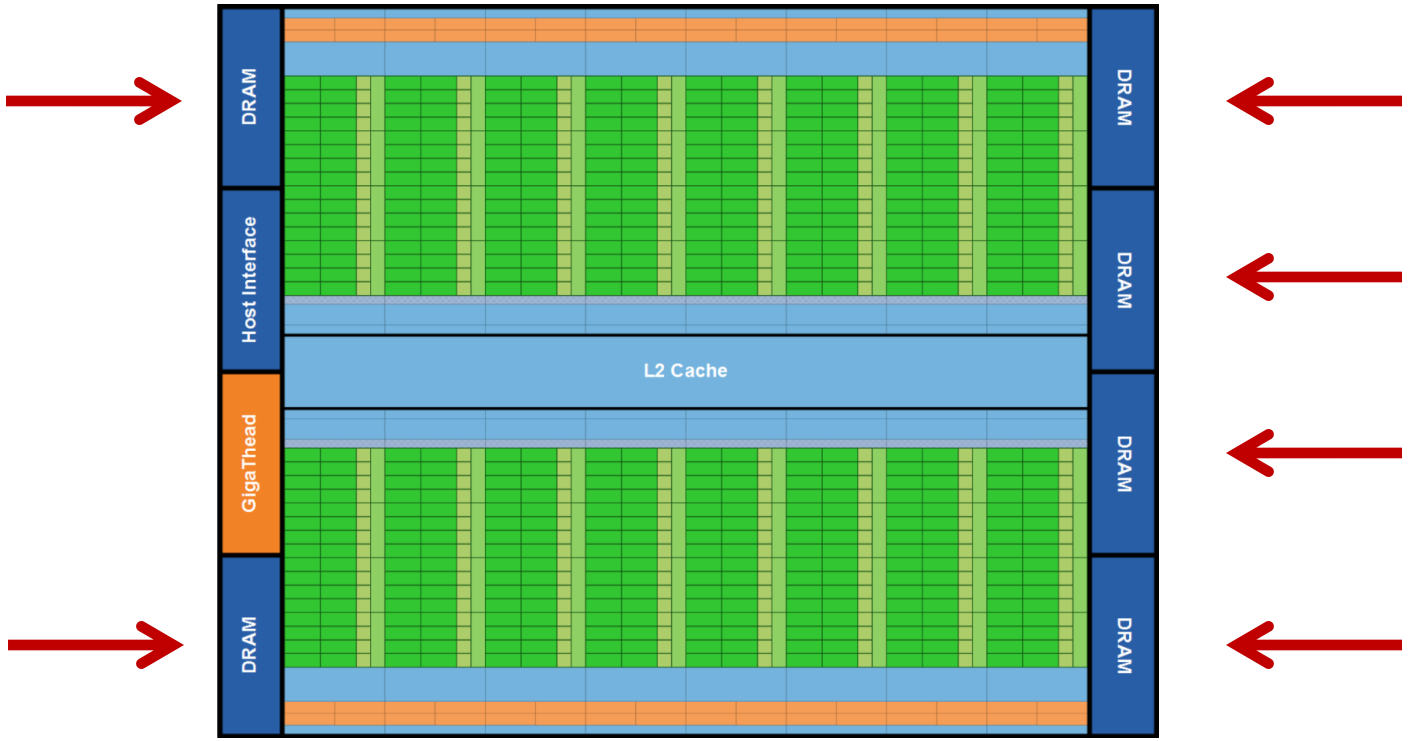
Feature	GPU Shared Memory	L1 Cache in CPU
Management	Managed by programmers	Automatically managed by hardware
Access scope	Shared among threads in a block	Dedicated per core (private)
Size	Typically 48-128 KB per SM	Typically 32-64 KB per core
Cache coherence	No cache coherence	Cache coherence protocols (MESI, MOESI, etc.)
Read/Write	Read and write within a block	Read and write by the owner core
Data persistence	Lives together with the launching block	Data is evicted based on a cache replacement policy
Access pattern optimization	Optimized for coalesced accesses and minimizing bank conflict	Optimized for temporal and spatial locality while avoiding false sharing
Use case	Matrix multiplication, reduction, etc.	Fast access to frequently used data and instructions

Caches & the Shared Memory

- Hopper:
  - L1 + SharedMem: 256 KB per SM.
  - L2 cache: 80 MB, common pool available to all SMs
- Volta:
  - L1 cache + Shared Mem = 128 KB per SM
  - One can carve out of 128KB shared memory as follows:
    - 0 KB, 8 KB, 16 KB, 32 KB, 64 KB, or 96 KB
  - What is not taken by the ShMem becomes L1 cache:
    - 128 KB, 120 KB, 112 KB, 96 KB, 64 KB, 32 KB
  - All SMs tap into 6144 KB of L2 cache memory, visible to all SMs
- Pascal:
  - *Each* SM: Unified 24 KB L1 cache
  - *Each* SM: 64 KB shared memory
  - All SMs tap into 4096 KB of L2 cache memory
- Fermi
  - *Each* SM: 64 KB L1 cache & shared memory
  - All SMs tap into a common 768 KB L2 cache memory

Largest Memory on GPU: Global Memory (shown for a Fermi GPU)

- "Global" in the sense that it doesn't belong to an SM but rather all SMs can access it
  - Global memory is off-chip DRAM and typically have a much larger space than SM and caches
- Up to 6 GB of "global memory" on Fermi, see pic below
  - Kepler goes up to 12 GB, Pascal up to 16 GB, Volta up to 32 GB, Ampere up to 40 GB, Hopper up to 80 GB , Blackwell goes up to 192 GB , …



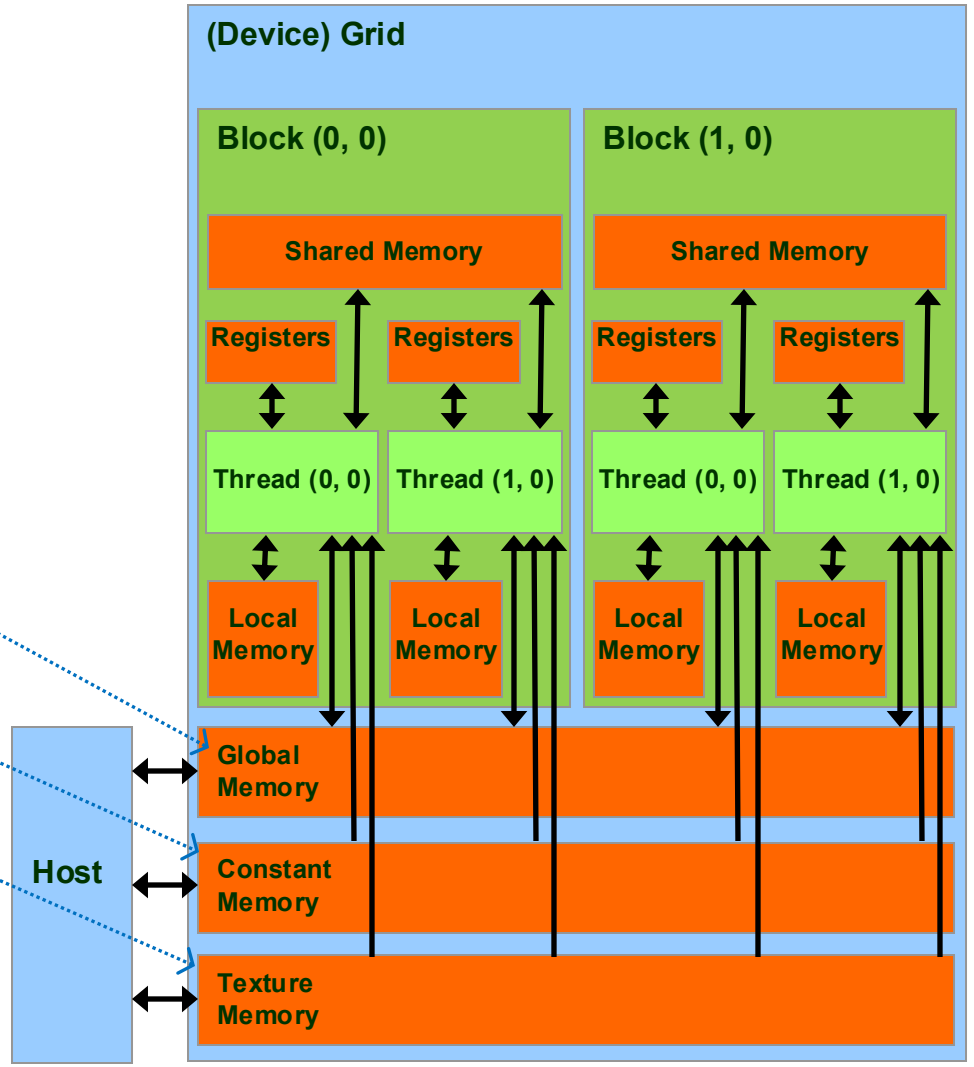
Example of Ampere Arch: “Global Memory” is Off-chip

*Off-chip refers to any hardware components or resources that are physically located outside the main chip or integrated circuit (IC) of a device.

*On-chip refers to components or memory that are located directly on the same silicon die as the main processor (CPU or GPU). These components are integrated into the same physical chip, meaning there are no physical separation or external connections between them.

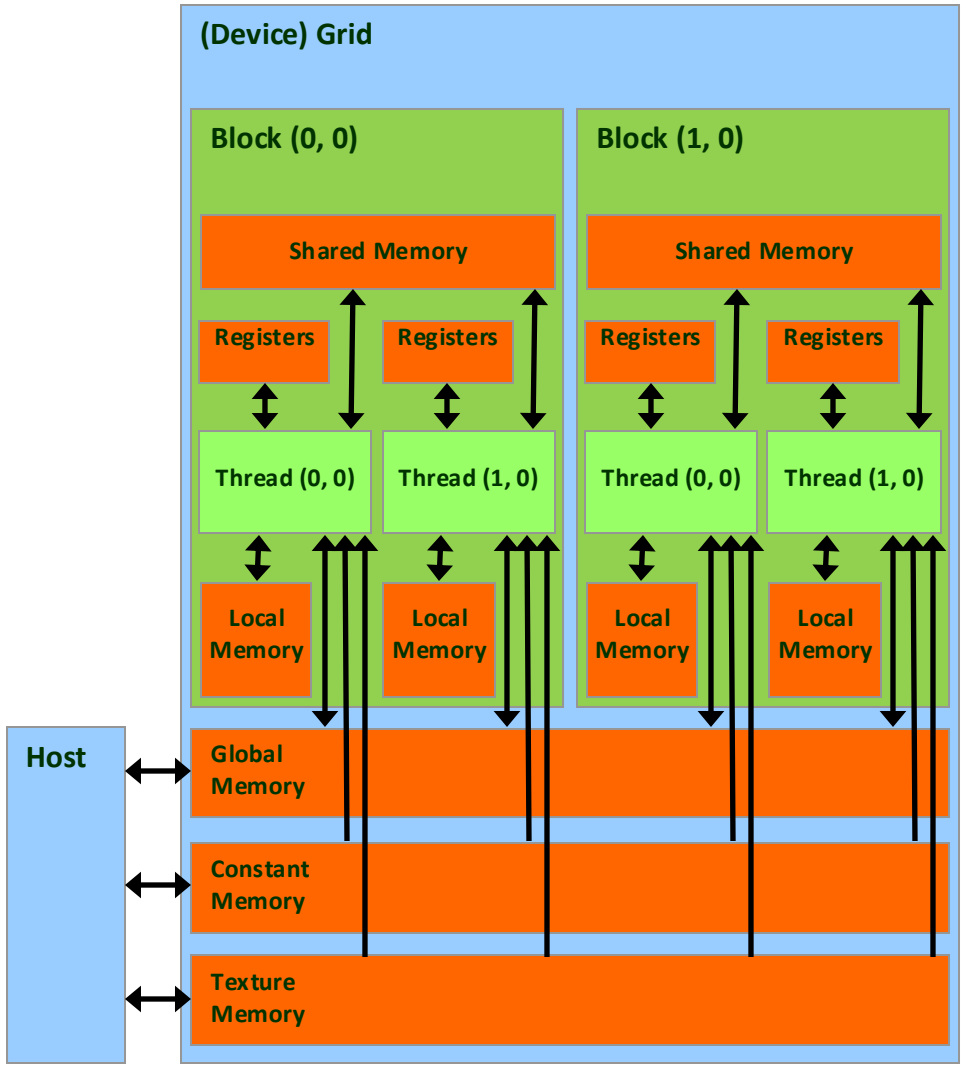
Global, Constant, and Texture Memories (Typically Read-only)

- Global memory
  - Main gateway 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
- Global, texture and constant memories are accessible by host
  - Typically store read-only constant data
  - Done at high latency, low bandwidth



The Concept of Local Memory

- Physically, local memory does not exist
  - In reality, data stored in “local memory” is placed in cache or the global memory at run time or by the compiler
- What gets stored in local memory?
  - Register spill, if too many registers are needed for computation (“high register pressure”)
  - The stack of the thread used to store function address, call stack, activation frame, etc.
- Quick observations:
  - “Local” means the memory is specific to one thread and not visible to any other thread
  - Local memory in the worst case has the same latency as global memory



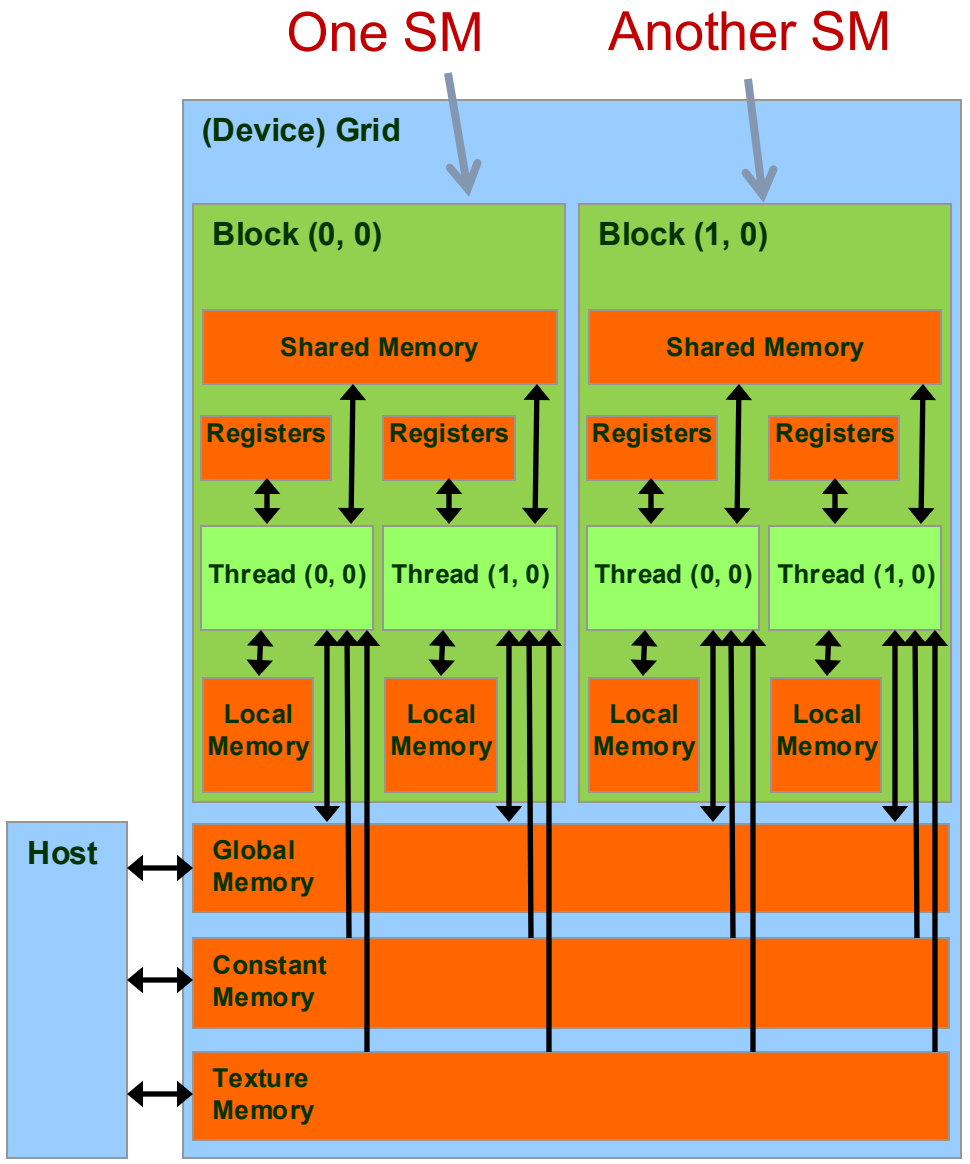
CUDA Memory Hierarchy

[Note: picture assumes two blocks, each with two threads; we have one SM running one block]

- 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
  - Read only per-grid surface memory

- Host can R/W global, constant, and texture memory
  - Ex: `cudaMalloc`

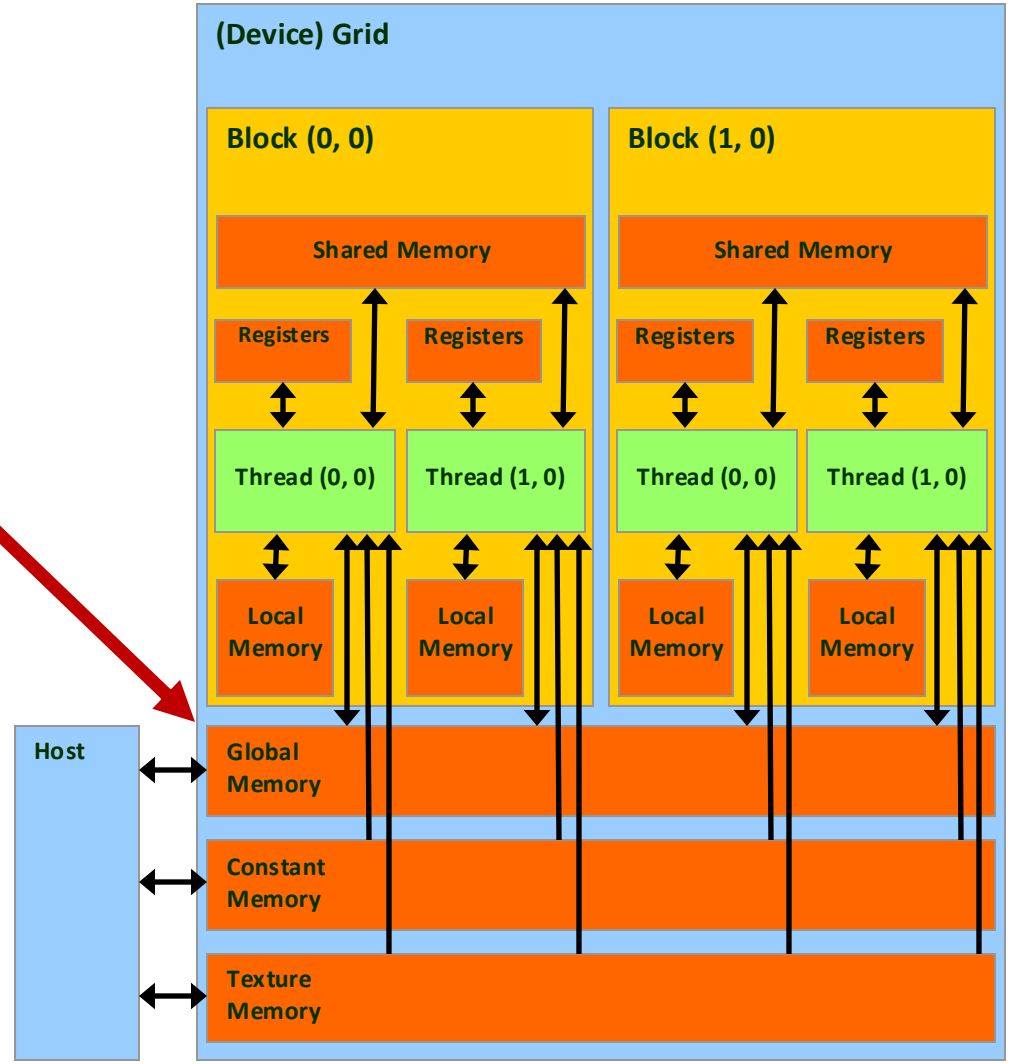


Global, Constant, and Texture memory are Persistent

- These memory spaces retain their data across multiple kernel launches within the same execution of a host application
  - Global, constant, and texture memory are not automatically reset or cleared between the execution of different CUDA kernels
  - If you write data to these memory spaces in one kernel, that data will still be available for subsequent kernels unless explicitly modified or cleared
- Programmers must manage the lifetime of data in these memory spaces to avoid unintentional reuse or data corruption

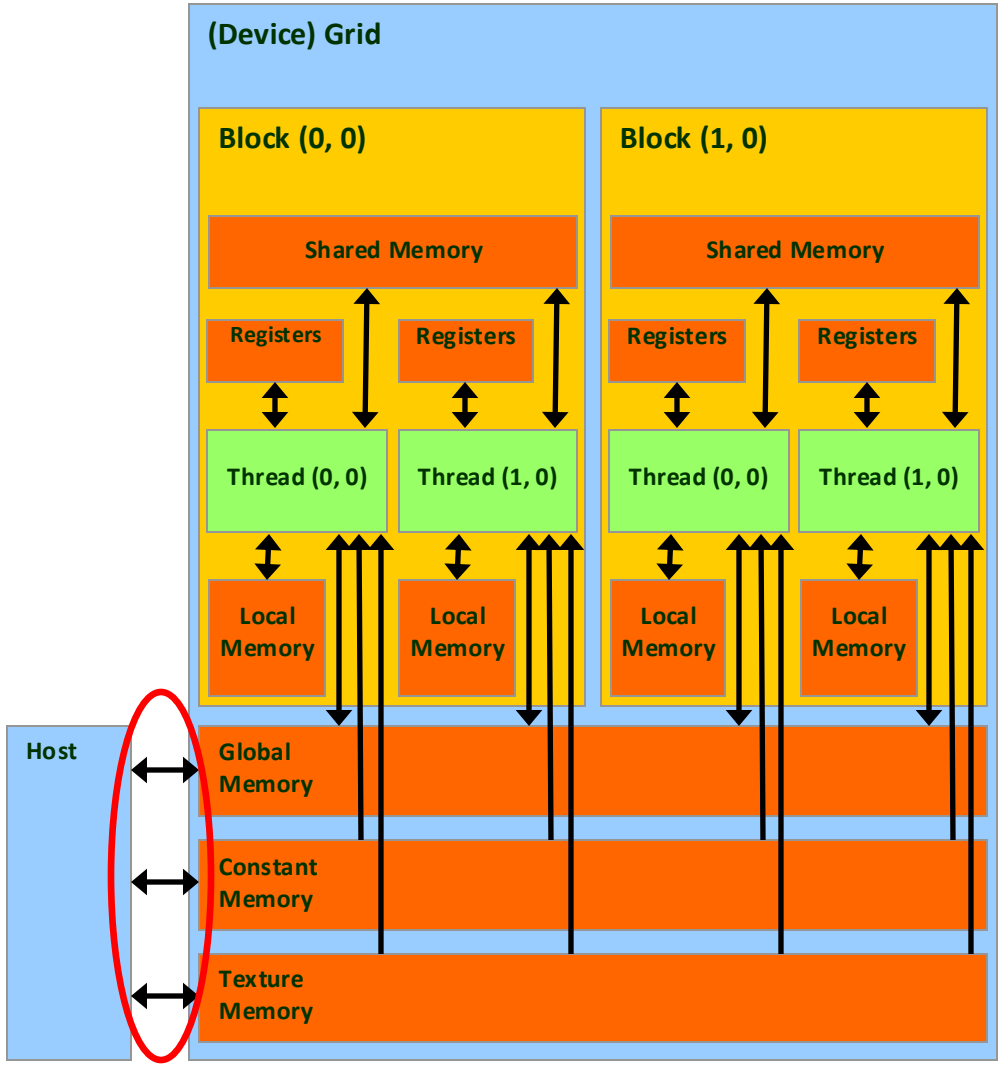
CUDA Device Memory Allocation/Deallocation

- `cudaMalloc(&ptr, size)`
  - Allocates mem in the device Global Memory
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object
- `cudaFree(ptr)`
  - Frees mem from device Global Memory
    - Requires a pointer to freed object



CUDA Host-Device Data Transfer

- `cudaMemcpy(dst, src, bytes, type)`
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
      - Or, use a default value for CUDA to figure it out
  - Things happen over a PCIe connection
    - Up to 32 GB/s (PCIe 4.0 x16, in each direction)
    - Up to 64 GB/s (PCIe 5.0 x16, in each direction)



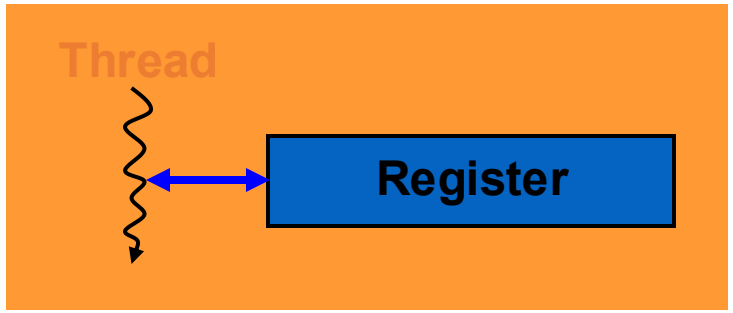
CUDA Host-Device Data Transfer (cont.)

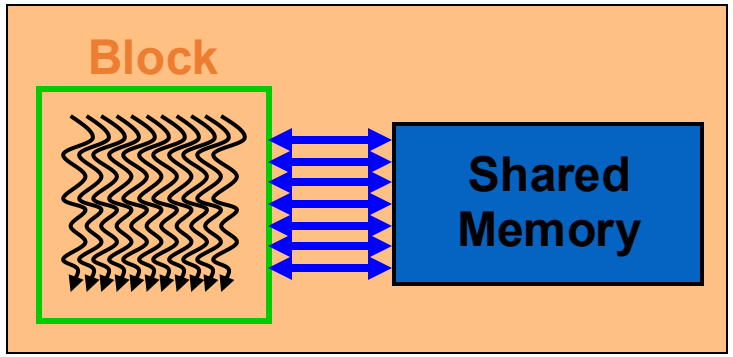
- Example:
  - Transfer a number of "`size`" bytes
  - `M` is in host memory and `Md` is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are CUDA-defined constants

```
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDefault);
```

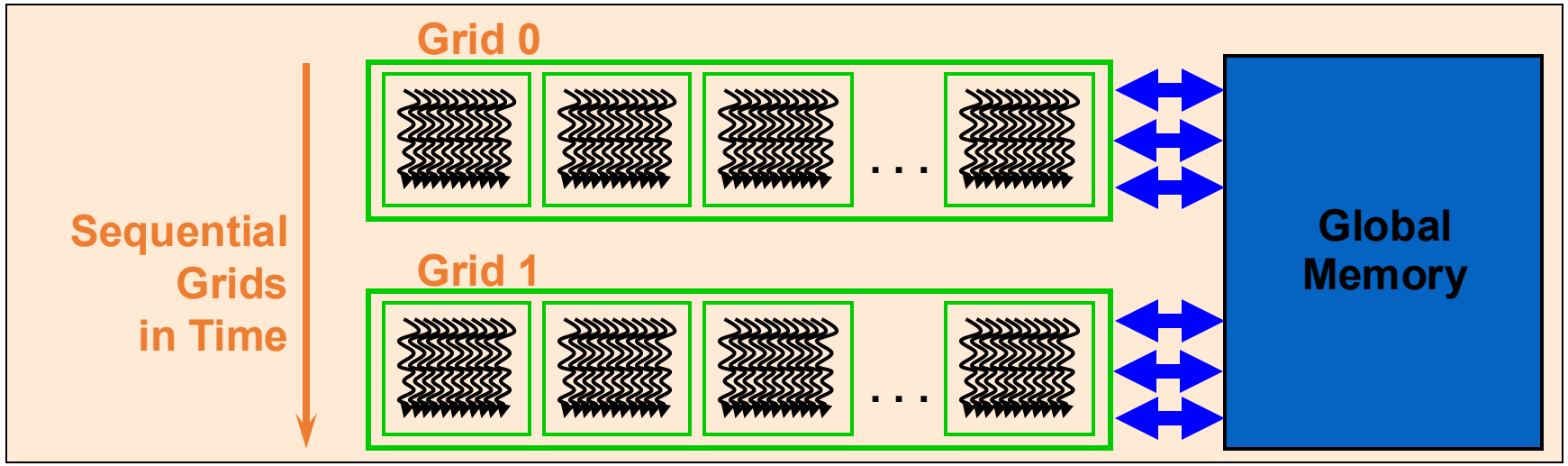
- Note: if you use gdb to debug, don't expect to be able to read memory off a device pointer
  - Debugging for GPU computing relies on `cuda-gdb`

Putting Things Together: The Three Most Important GPU Memory Types





- Register: per-thread basis
  - Private per thread
  - Can spill into local memory and global memory (operated by runtime)
- 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-block and inter-grid communication



Slightly from the left field

- GPU virtual memory: 49-bit virtual address space mapped to physical memory located:
  - On the device
  - On the host, pinned system memory
  - On a different GPU (peer memory)
- Global memory is visible to all threads in the GPU
- Global memory is accessed through the SM L1 and GPU L2
- NVIDIA's first take at fusing CPU& GPU (ARM architecture): white paper

Case Study: Matrix Multiplication, Revisited

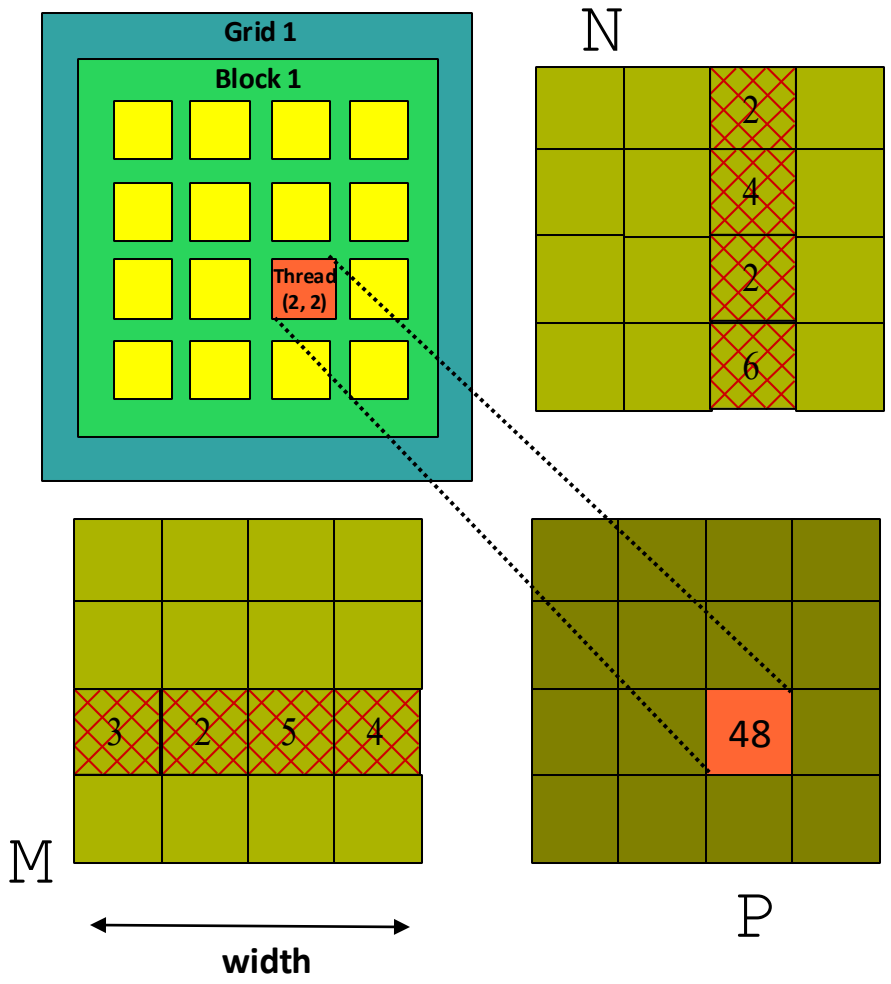
- Previous, we have introduced a simple GPU-parallel matrix multiplication kernel
  - We used only one block of threads to perform the multiplication
  - Each thread fetches one row and one column from the two operand matrices and perform a dot product to reduce the result
  - All data is accessed through the global memory at a cost of high latency

- In the next example, we multiply large matrices & use multiple blocks of threads
  - Highlight the use/role of the shared memory
  - Highlight the role of the `__syncthreads()` function call

- NOTE: A one-dimensional array stores the entries in the matrix

Problem of the Previous Matrix Multiplication Example

- To calculate one matrix value by a thread
  - One row of `M` is loaded `N.width` times
  - This happens for each row of `M`
  - One column of `N` is loaded `M.height` times
  - This happens for each column of `N`
- Many repetitive (and redundant) global memory access
  - Ex: P[1, 2], P[1, 3], P[1, 4] … all load row 1 of `M`
    - Why don't we load row 1 together and process them in shared?
  - Ex: P[1, 1], P[2, 1], P[3, 1] … all load column 1 of `N`
    - Why don't we load column 1 together and process them in shared?
- Matrices `M` and `N` used multiple times → Keep them in *shared memory* and access them from there to enhance performance



Why should we do this? Why should we use Shared Memory?

- Advantage of shared memory lies in its fast memory access speed
  - Shared memory latency is about 10-30 cycles, whereas global memory latency is 400-800 cycles
- Figuring out whether you should use shared memory
  - Imagine you are a thread executing the code in the kernel
  - If you, as a thread, use data that can also be used by other thread(s) in your block multiple times, then you should consider using shared memory
- Note: you can always use shared memory as scratch pad memory (even if data not shared)
  - Don't let it go unused, use for storing often-used variables even if variables not used by multiple threads

Shared Memory is Highly Recommended for GPU programming

- “Try to organize your data structure and algorithms to make use of shared memory”

- Organize data and use algorithms so that you can bring data in shared memory and subsequently use it repeatedly by multiple threads in the block
  - This sounds like “locality”; the cache should help you too. But explicitly writing shared memory patterns in your kernel will greatly facilitate the use of caches and compiler optimization
    - Again, you are the most knowledgeable person to your code

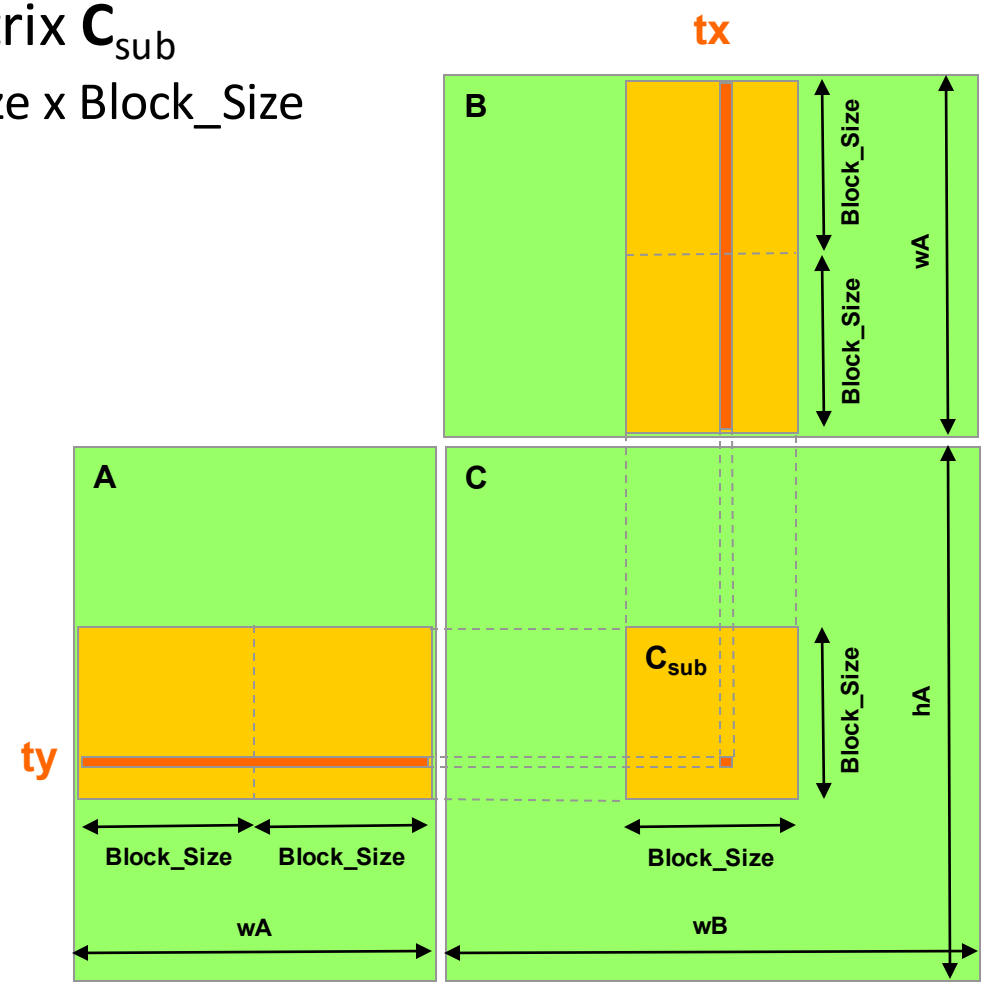
A Common Shared Memory Programming Pattern: Tiling

- Fact: Global memory accesses are much slower when compared to shared memory access

- Consequence: Use shared memory, if possible
  - 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 of the tile

Use Tiling for Matrix-Matrix multiplication: one block computes one tile

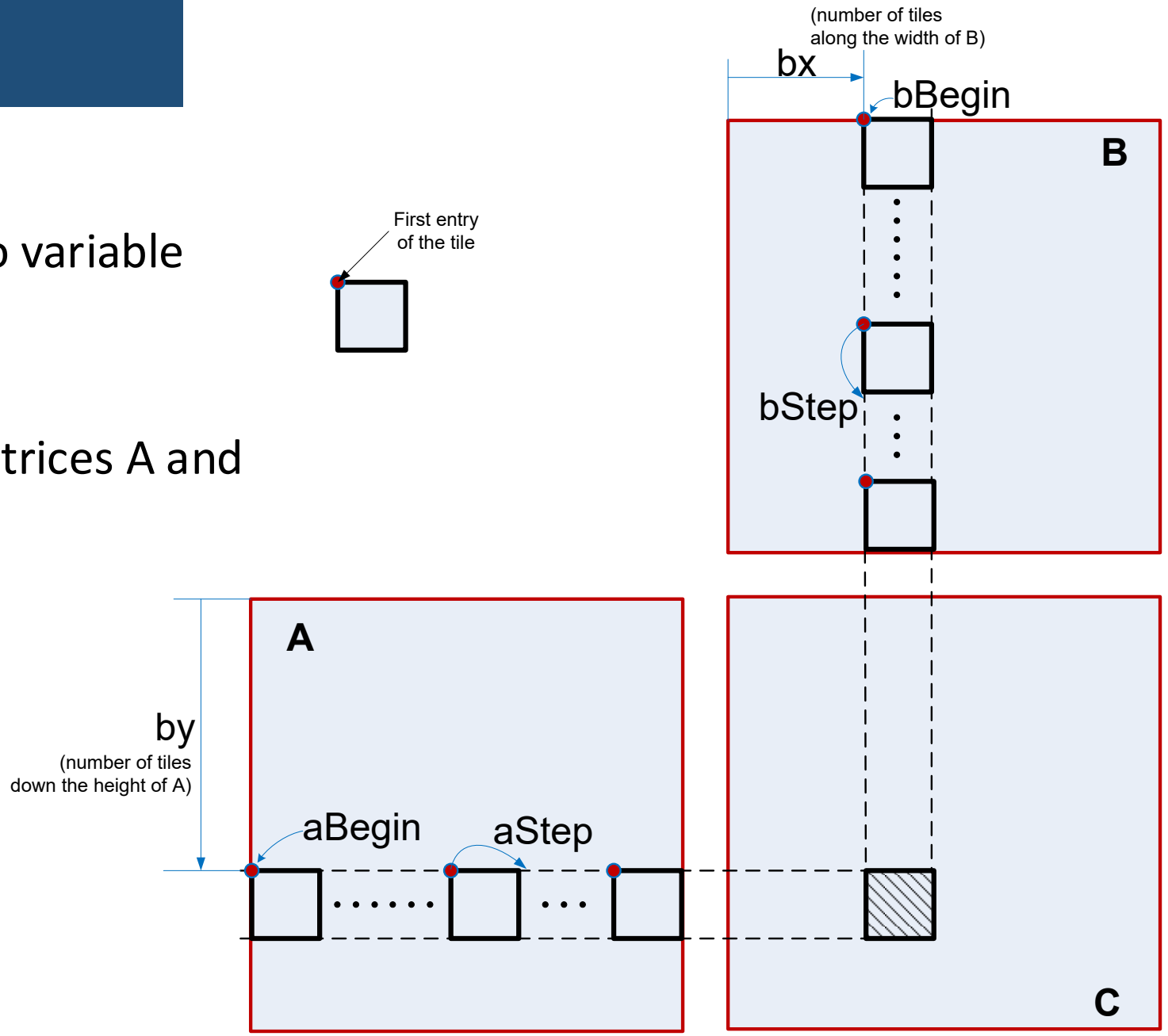
- One block computes one tile, i.e., a square sub-matrix $\mathbf{C}_{sub}$
  - Each dimension is of Block_Size → $\mathbf{C}_{sub}$ is of size Block_Size x Block_Size
- One thread computes one entry of $\mathbf{C}_{sub}$
- Assumption: **A** and **B** are *square matrices* and their dimensions of are *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`**
  - In reality, you need to handle boundary condition



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

Tile-by-Tile Multiplication

- The names in this pic correspond to variable names in the kernel on next slide
- Indexing is messy owing to how matrices A and B are stored in memory



```
// 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)
{
    // Load A and B to the device
    float* Ad;
    int 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);
```

```
    // 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 global memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// 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;
```

```
// Shared memory for the sub-matrices (tiles) of  A and B
   __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

   // Loop over all the sub-matrices (tiles) of A and B required to
   // compute the block sub-matrix; moving in A left to right in
   // a row, and in B from top to bottom in a column
   for (int a = aBegin, b = bBegin; a <= aEnd;
      a += aStep, b += bStep) {

         // Load tiles from global memory into shared memory; each
         // thread loads one element of the two tiles from A & B
         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();

         // Each thread in this block computes one element
         // of the block sub-matrix (tile).  Thread with indexes
         // ty and tx computes in this tile the entry [ty][tx].
        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;
}
```

`__syncthreads()`, some important aspects

- Synchronization accomplished through an SM-level lightweight function
  - `void __syncthreads();`
  - Intra-block synchronization is typically very fast – a few clock cycles to accomplish the synchronization

- This function call synchronizes all threads in a block (acts as barrier for all threads in block)
  - Does **not** synchronize threads that belong to different blocks
  - Synchronization of different blocks are typically done at the kernel level
    - Inter-block synchronization is typically of high latency because of the need to flush and reload data in global memory, plus kernel launch overhead (microseconds to milliseconds, depending on the architecture and kernel complexity)

- Takeaway: prefer intra-kernel synchronization over inter-kernel synchronization
  - This is one reason why you need shared memory

Two Ways to Configure Shared Memory

- First way: Statically, declared inside a kernel
  - This is how we use in the previous matrix multiplication example w/ shared memory use
  - Keep in mind each block has an upper limit on the shared memory you can use

- Second way: Dynamically, declared outside the kernel via execution configuration
  - `Ns` below indicates size (in bytes) to be allocated in shared memory

```
__global__ void MyFunc(float*) // __device__ or __global__ function
{
   extern __shared__ float shMemArray[];
   // Size of shMemArray determined through the execution configuration
   // You can use shMemArray as you wish here…
}
// invoke like this
MyFunc<<< Dg, Db, Ns >>>(parameter);
```

Example: External Configuration of Shared Memory

- Reverse an array of integers for each thread using dynamic shared memory

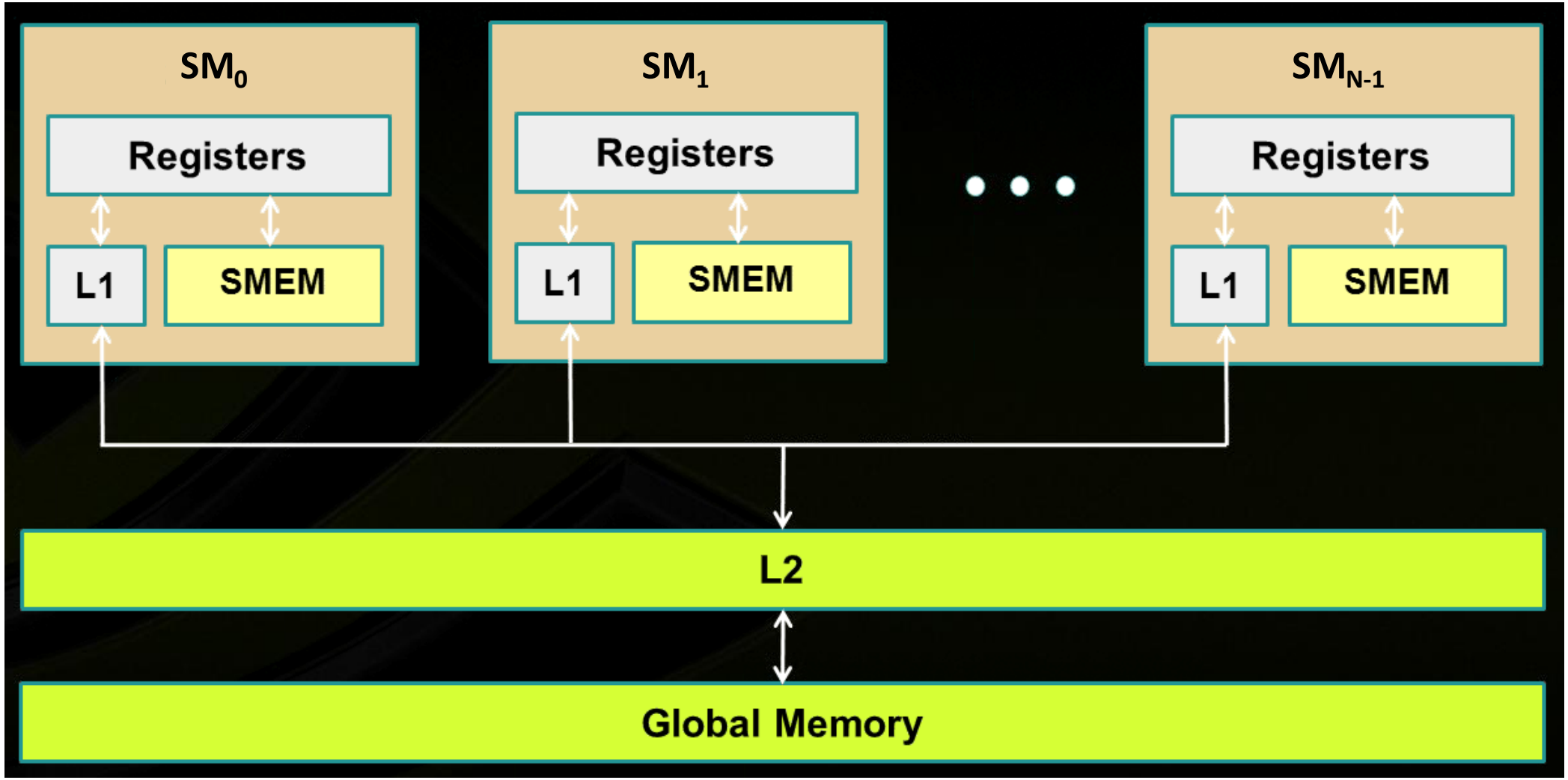
```
__global__ void my_function(int *d, int n) {
    extern __shared__ int s[]; // ShMem allocation determined at run time

    int t = threadIdx.x;
    s[t] = d[t];

    __syncthreads(); // make sure all threads have completed loading into shared memory

    ... // perform work on s[]
}
```

```
my_function<<<1, n, n*sizeof(int)>>>(d_d, n);
```

size in bytes of the shared memory

[https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/]→

Revisit the GPU Memory Hierarchy – Hopefully it Makes More Sense Now



Details on Fetching Data into Shared Memory

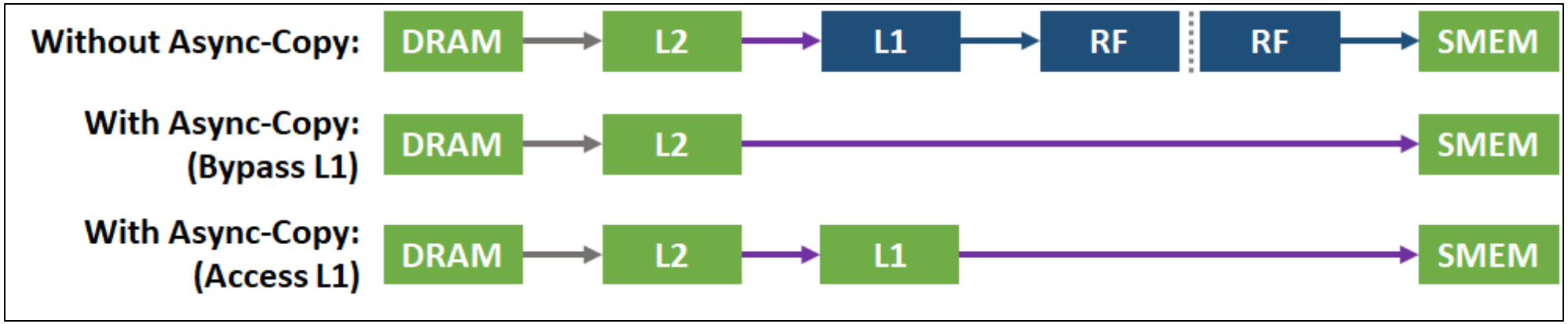
- Load data from global memory into shared memory
  - Data might be in global mem or in L2 cache

- On Volta and older architectures:
  - Data was first loaded through L1 cache into the register file with load-global instructions
  - Then, data was transferred from the register file to shared memory with store-shared instructions

- As with Ampere, there is a new asynchronous copy instruction that loads data directly from global memory into SM shared memory
  - Called “load-global-store-shared asynchronous copy”
  - Bypasses the register file
    - Less pressure on registers
    - Less overhead

Ampere A100 specific, Cache and Shared Memory aspects

- New asynchronous copy instruction loads data directly from global memory into shared memory, optionally bypassing L1 cache, and eliminating the need for intermediate register file (RF) usage



- Two other Ampere A100 footnotes:
  - L1+ShMem combo: 192 KB/SM in A100 (versus 128 KB/SM in V100)
  - SECDED ECC protection for L2 cache, as well as the L1 caches and register files inside all SMs

[NVIDIA Ampere whitepaper]→

Size Comparison between L1/Shm and L2 Caches

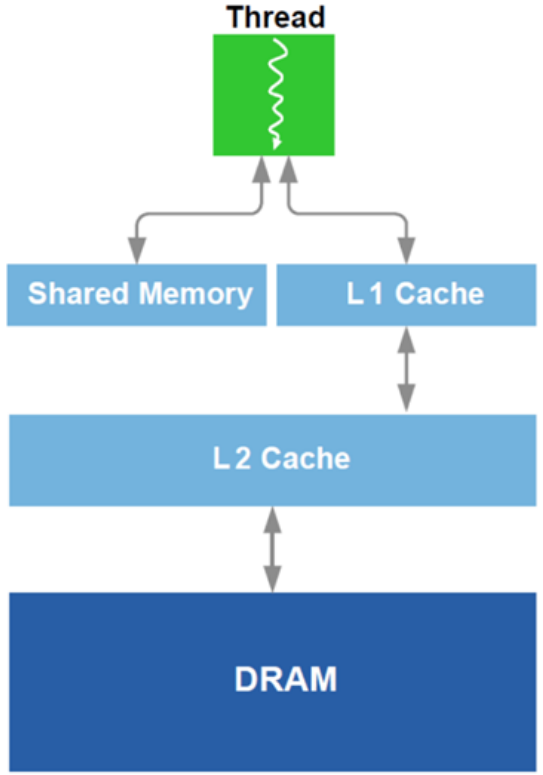
- The amount of L1 cache on the SM is about 4x bigger than L1 cache size on a CPU core
  - Typically shared with the shared memory size

- L2 cache: one big mem pool available to *all* SMs
  - Volta: 6,144 KB
  - Ampere: 40,000 KB (serving 108 SMs → 370.4 KB/SM)
  - Hopper: 50,000 KB (serving 144 SMs → 347.2 KB/SM)

- Level 2 cache on the GPU serves the same role & is similar in size to the L3 cache on the CPU
  - Example: Intel® Xeon® Platinum 8452Y Processor (about $4000 apiece) – 67 MB L3 cache
  - Note: L3 cache on the CPU called Last Level Cache

L1 Cache vs. Shared Memory

- Starting w/ Fermi architecture, you can adjust how much shared memory and how much cache
  - A zero-sum game; they share the physical banks



- Example, Fermi: you can go 48/16 or 16/48 KB for Shared Mem/Cache
- OPTION 1: you choose to go w/ "Lots of Cache & Little Shared Mem"
  - Cache handled for you by the scheduler
  - No control over it
  - Can't have too many blocks of threads running if blocks use Shared Mem
- OPTION 2: you choose to go w/ "Lots of Shared Mem & Little Cache"
  - Good in tiling, you have full control
  - Shared Mem is rather cumbersome to manage