CME213/ME339
Lecture 5

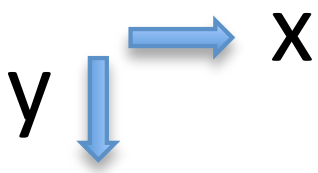
Eric Darve
Erich Elsen
Austin Gibbons

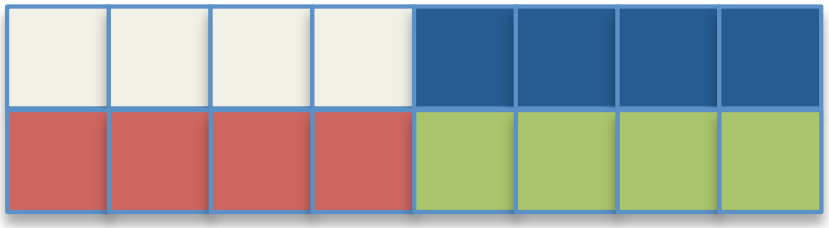
Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

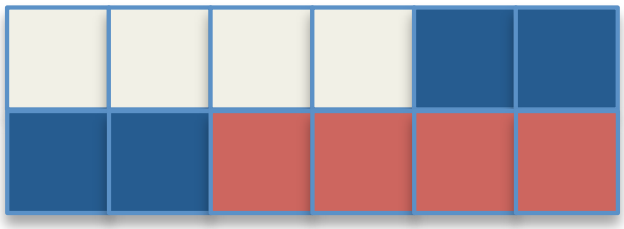
Spring 2012

CUDA warps

- How does the hardware manage so many threads? A special architecture is used for that purpose.
- Unique architecture called SIMT (Single-Instruction, Multiple-Thread) is used.
- Each SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps.**
- When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a **warp scheduler** for execution.
- Each warp contains threads of consecutive, increasing thread IDs. The first warp has thread 0.
- Thread IDs follow the 1D, 2D or 3D thread index (ordering is x, then y, then z), e.g., $x + yD_x$.


y
x





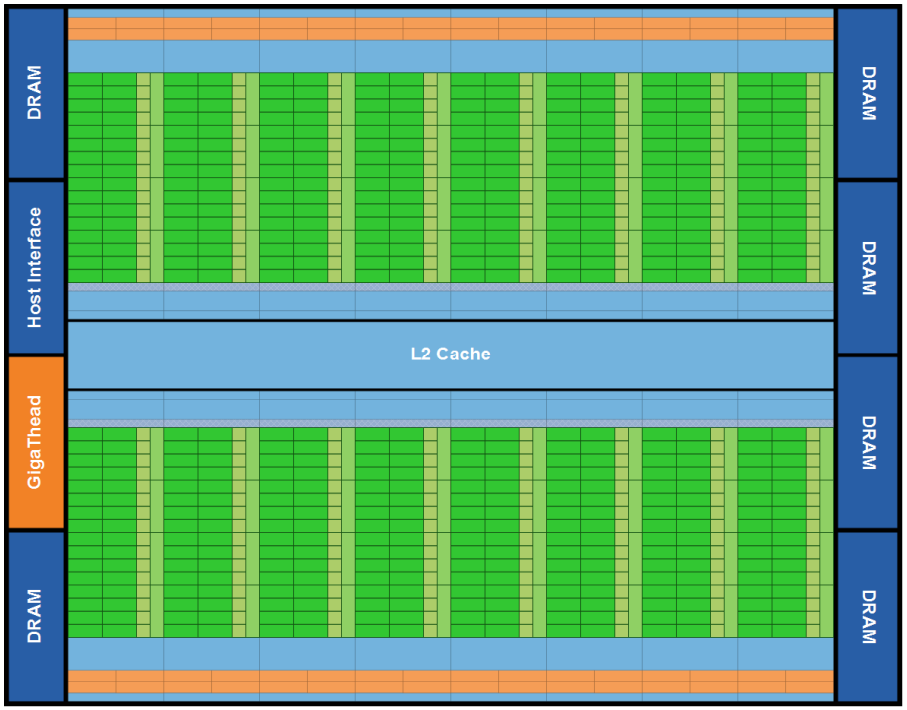
Warp execution

- A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.
- If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path.
- Possible causes of thread divergence:
  - Conditional statement like `if`
  - For loop with a size that is thread-dependent

CPU/GPU threads design and optimization

- If the GPU must wait on one warp of threads, it simply begins executing work on another.
- Because separate registers are allocated to all active threads, no swapping of registers or other state need occur when switching among GPU threads. Resources stay allocated to each thread until it completes its execution.
- In short, CPU cores are designed to *minimize latency* for one or two threads at a time each.
- GPUs are designed to handle a large number of concurrent, lightweight threads in order to *maximize throughput.*

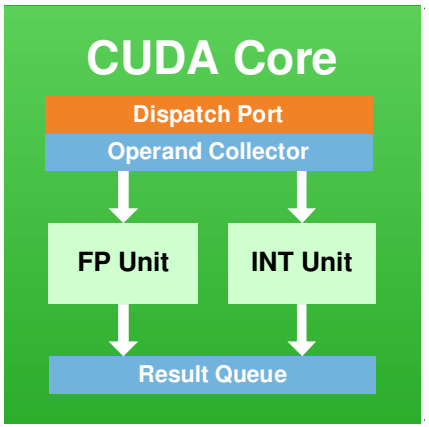
Programming reflects hardware

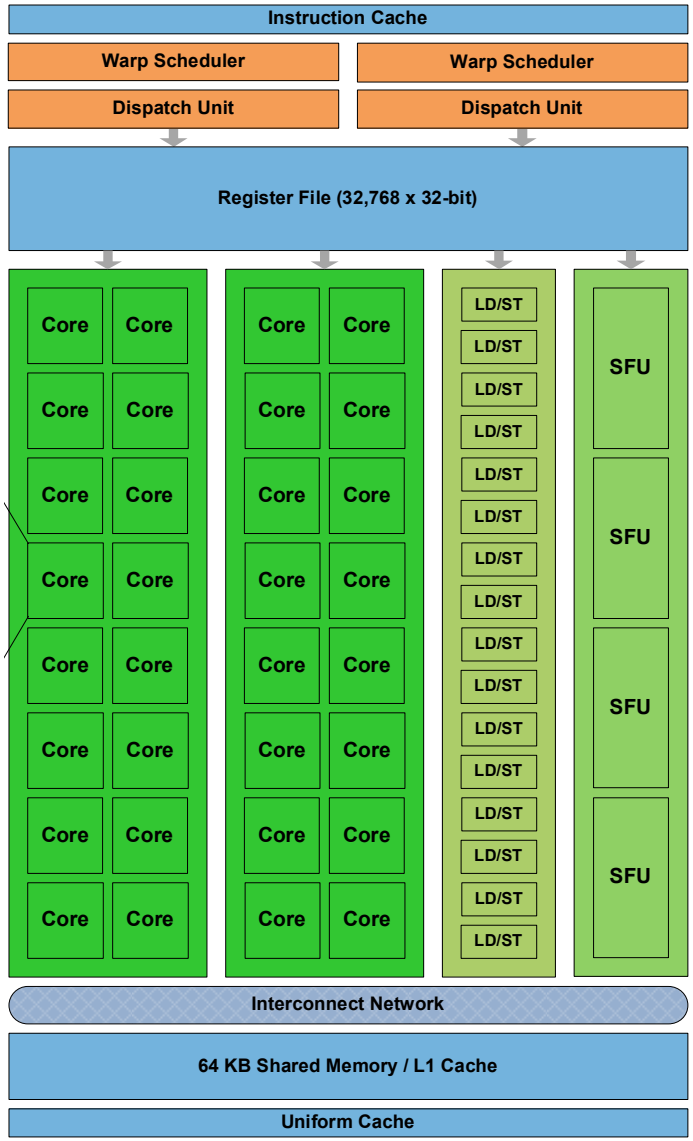


Fermi's 16 streaming-multiprocessor (SM) are positioned around a common L2 cache.

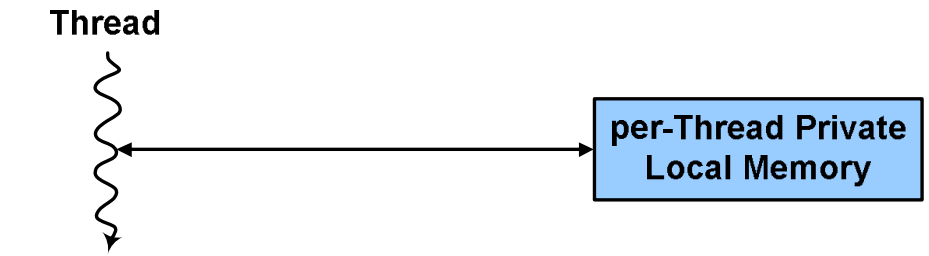
Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

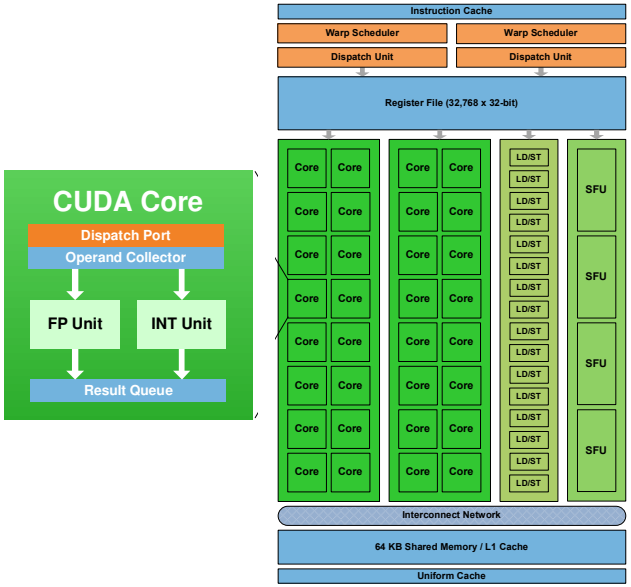
Fermi SM and core

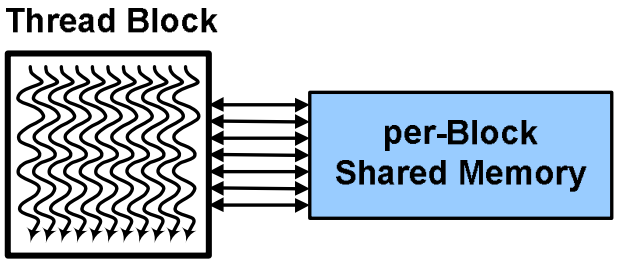


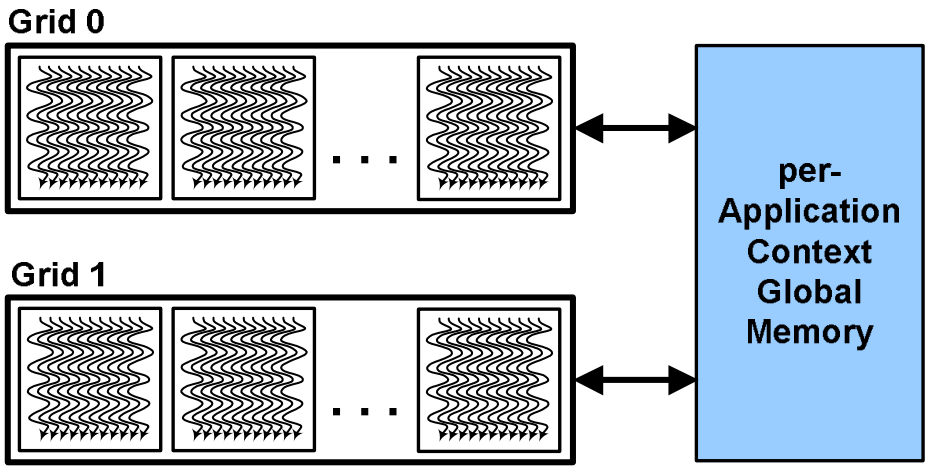


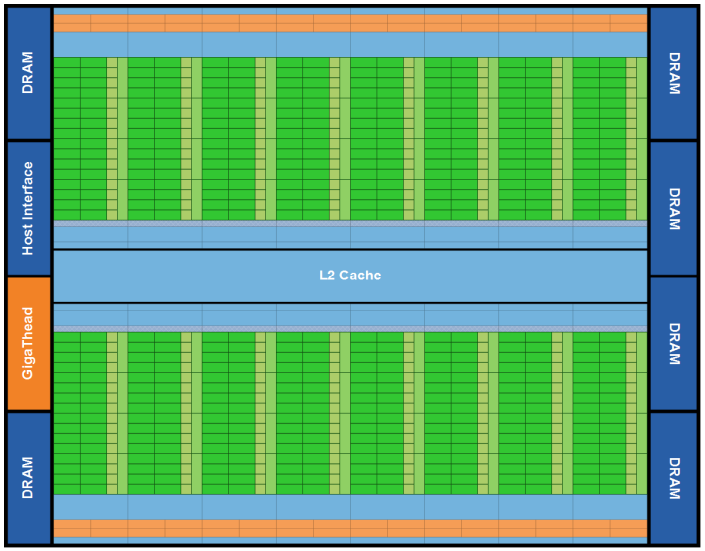
Grid, blocks, SMs, and cores











Parallel kernel execution

A thread executes on a single streaming processor
  Allows use of familiar scalar code within kernel

A block executes on a single streaming multiprocessor
  Threads and blocks do not migrate to different SMs
  All threads within block execute concurrently, in parallel

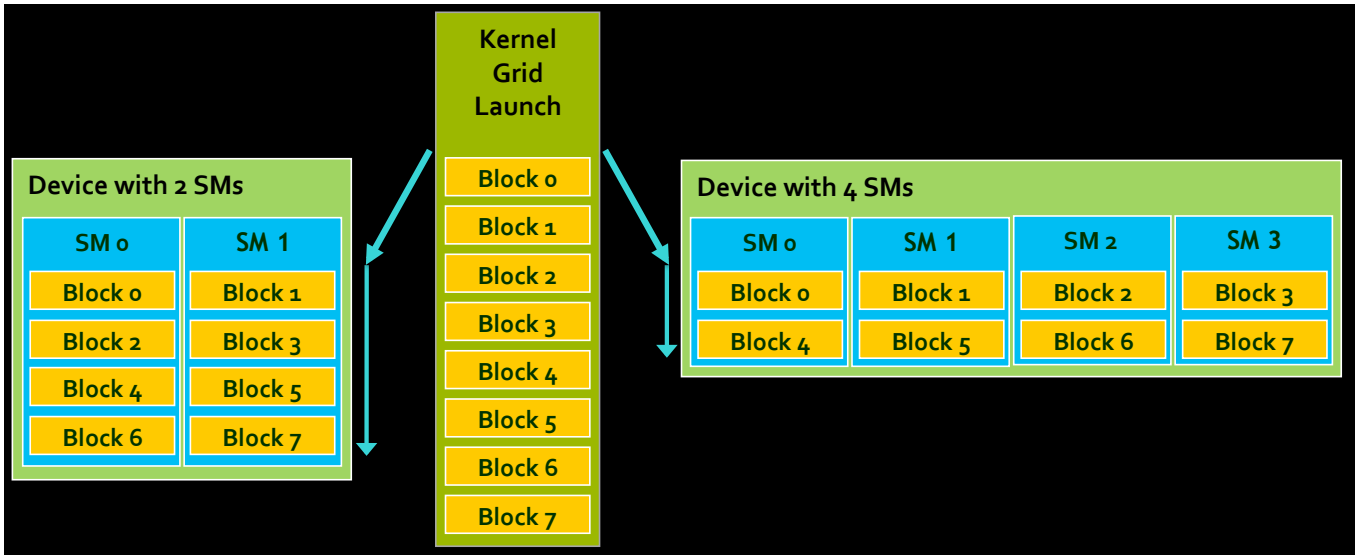
A streaming multiprocessor may execute multiple blocks
  Must be able to satisfy aggregate register and memory demands

A grid executes on a single device (GPU)
  Blocks from the same grid may execute concurrently or serially

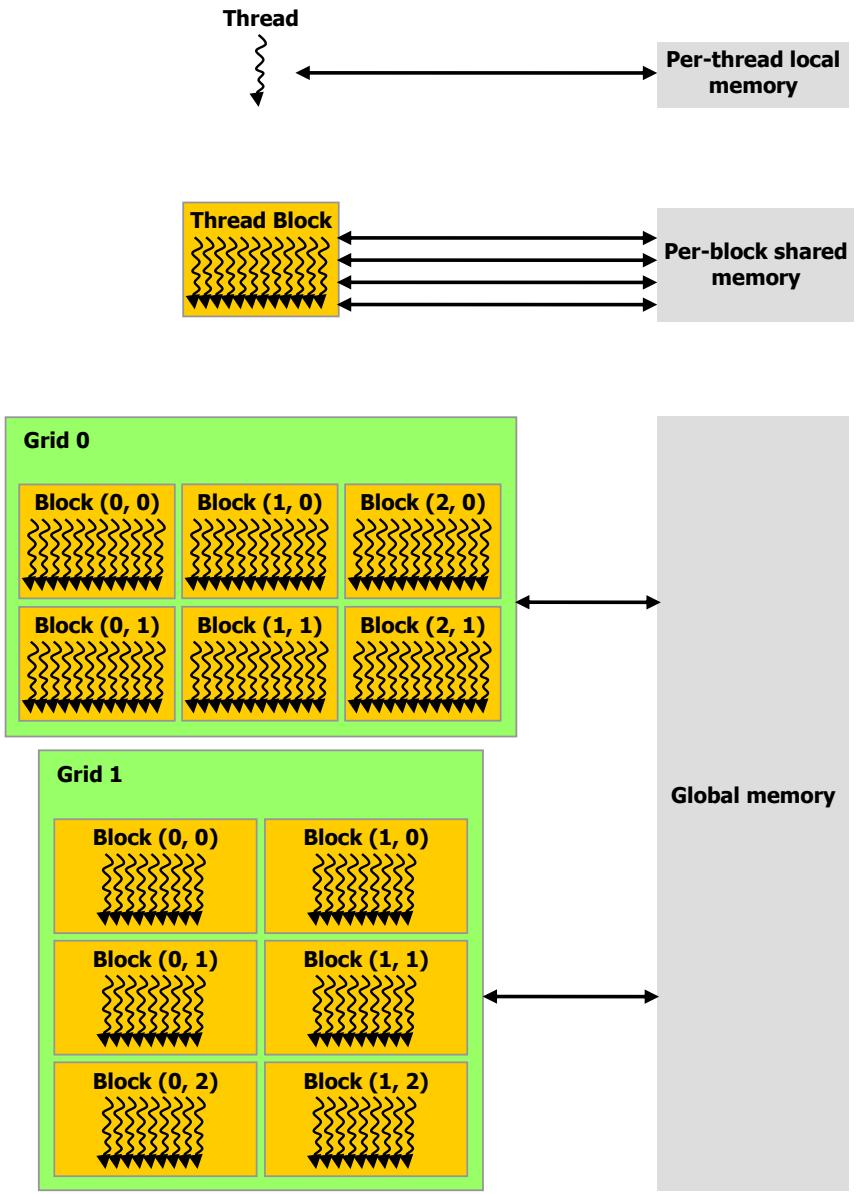
Blocks provide scalability

Performance scales with the number of available SMs. Large number of blocks can be handled without overhead.

- Blocks may execute in arbitrary order, concurrently or sequentially, and parallelism increases with resources
- Block executes when resources become available
- Blocks can be distributed across any number of SMs



CUDA memory hierarchy



Each thread has private local memory.

Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.

All threads have access to the same global memory.

Example: use of shared memory

Classical problem of computing π by generating random numbers.

A poor numerical method to calculate π but a simple example to illustrate the use of shared memory.

Shared memory allows threads **within a block** to exchange data.

Data cannot be exchanged between blocks.

π estimator

```
1   ...
2   // Determine how to divide the work between cores
3   dim3 block, grid;
4   block.x = nthreads;
5   grid.x  = (numSims + nthreads - 1) / nthreads;
6
7   // Aim to launch around blocksPerSM or more times as many blocks
8   // as there are multiprocessors on the target device.
9   unsigned int blocksPerSM = 10;
10  // Reduce grid.x to the smallest value while satisfying
11  // blocksPerSM.x
12  // numSMs: number of SMs on device
13  while (grid.x > 2 * blocksPerSM * numSMs)
14      grid.x >>= 1;
15
16  // Count the points inside the unit quarter-circle
17  computeValue<<<grid, block, block.x * sizeof(unsigned int)>>>
18              (d_results, numSims, ...);
19  ...
```

GPU kernel

```
1  __global__ void computeValue(unsigned int * const results,
2                               const unsigned int numSims, ...) {
3  // Determine thread ID
4  unsigned int bid = blockIdx.x; // Block ID
5  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
6  // Thread ID
7  unsigned int step = gridDim.x * blockDim.x;
8  // Required because numSims > total number of threads
9
10 // Initialise the RNG
11 ...
```

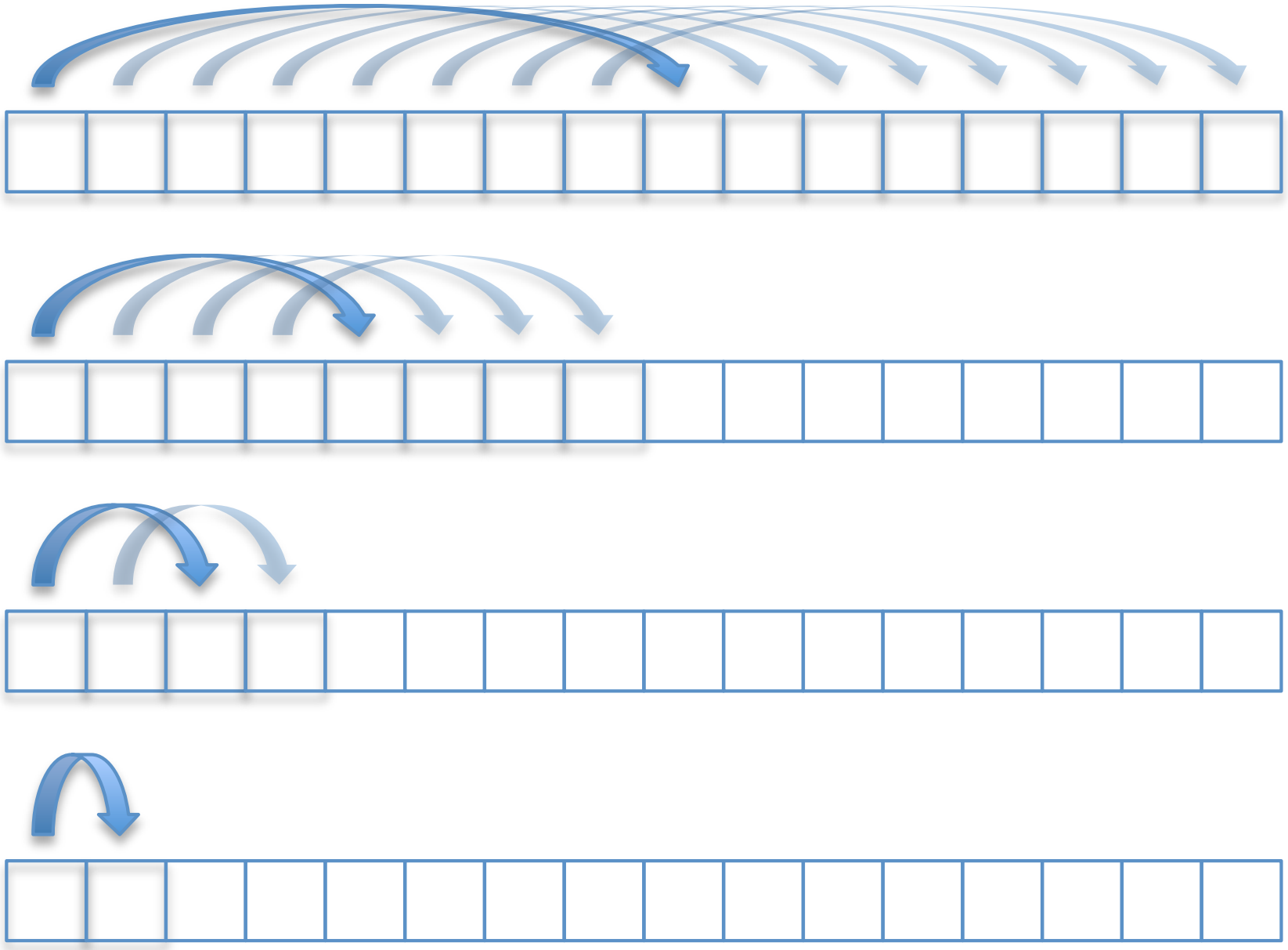
```
1  // Count the number of points that lie inside the unit
2  // quarter-circle
3  unsigned int pointsInside = 0;
4  // Loop is required because numSims > total number of threads
5  for (unsigned int i = tid ; i < numSims ; i += step) {
6    float x, y;
7    ... // Randomly generate points
8    float l2norm2 = x * x + y * y;
9    // Count if inside the unit circle
10   if (l2norm2 < 1.0f) pointsInside++;
11 }
12 ...
```

Shared memory code

```
1  extern __shared__ unsigned int sdata[];
2  // Perform first level of reduction
3  // 1. Write to shared memory
4  unsigned int ltid = threadIdx.x;
5  sdata[ltid] = pointsInside;
6  __syncthreads();
7  // Make sure all threads have written to shared memory
8
9  // 2. Do reduction in shared mem
10 for (unsigned int s = blockDim.x / 2 ; s > 0 ; s >>= 1) {
11   if (ltid < s) sdata[ltid] += sdata[ltid + s];
12   __syncthreads();
13   // Make sure sdata has been updated by all threads
14 }
15
16 // Store the result
17 if (threadIdx.x == 0) results[bid] = sdata[0];
18 }
```

End of host function

```
1  ...
2  // Copy partial results back
3  vector<unsigned int> results(grid.x);
4  cudaMemcpy(&results[0], d_results,
5             grid.x * sizeof(unsigned int),
6             cudaMemcpyDeviceToHost);
7
8  // Complete sum-reduction on host
9  float value =
10       std::accumulate(results.begin(), results.end(), 0);
11
12 // Determine the proportion of points inside the quarter-circle,
13 // i.e., the area of the unit quarter-circle
14 value /= numSims;
15 value  = 4; // This is our estimate of π
16 ...
```



Variable type qualifiers

Four types (mostly):

1. `__shared__`: variable resides in shared memory; only accessible from all the threads within a block.
2. `__device__`: variable resides in global memory and is accessible by all threads.
3. `__constant__`: variable resides in constant memory space, which is in device memory; each SM has a **read-only** uniform cache that is shared by all functional units and speeds up reads from the constant memory space.
4. unqualified
   - Scalars and built-in vector types are stored in registers
   - Arrays may be in registers or local memory (registers are not addressable)

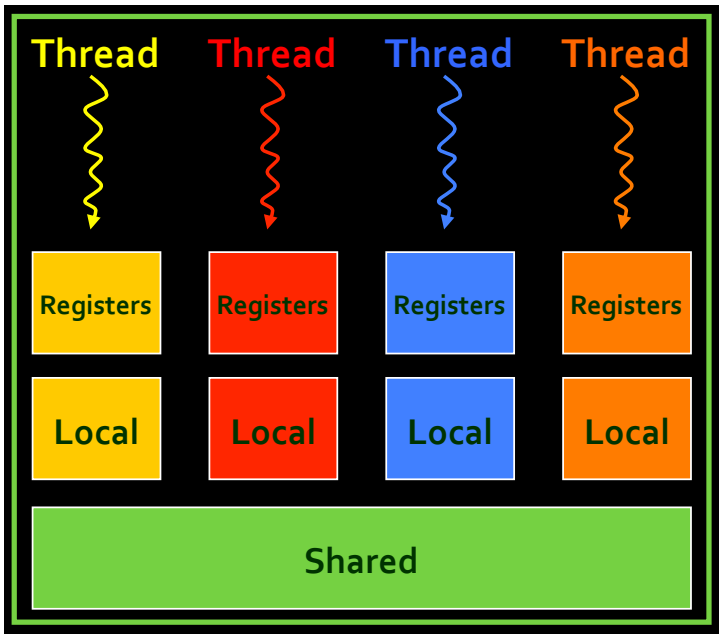
Shared memory

```
1 __shared__ <type> x[<elements>];
```

- Allocated per thread block
- Scope: threads in block
- Capacity: small (about 48kB)
- Bandwidth: very high
  Per SM: 32 4B 1.15 GHz / 2 = 73.6GB/s
  Across GPU:
  14 32 4B 1.15 GHz / 2 = 1.03TB/s

Common uses:

- Sharing data among threads in a block
- User-managed cache (to reduce global memory accesses)



Using shared memory

Size known at compile time

```
1  __global__ void kernel(...)
2  {
3  ...
4  __shared__ float sdata[256];
5  ...
6  }
7  int main(void)
8  {
9  ...
10 kernel<<<nblocks,nthreads>>>
11                         (...);
12 ...
13 }
```

Size known at kernel launch

```
1  __global__ void kernel(...)
2  {
3  ...
4  extern __shared__ float sdata[];
5  ...
6  }
7  int main(void)
8  {
9  ...
10 sb = blockSize*sizeof(float);
11 kernel<<<nblocks,nthreads,
12                      sb>>>(...);
13 ...
14 }
```

Global memory

- Allocated explicitly by host (CPU) thread
- Scope: all threads of all kernels
- Data lifetime: determined by host (CPU) thread

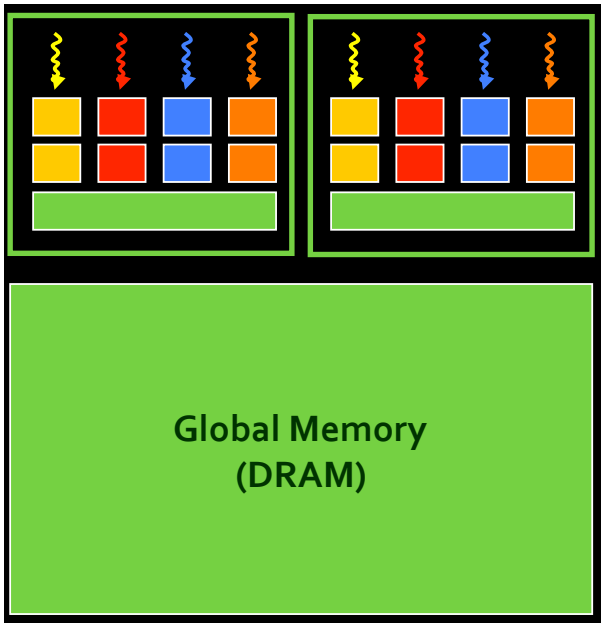
```
cudaMalloc(void** pointer,
           size_t nbytes)
cudaFree(void* pointer)
```

- Capacity: large (1–6GB)
- Latency: 400–800 cycles
- Bandwidth: 156 GB/s

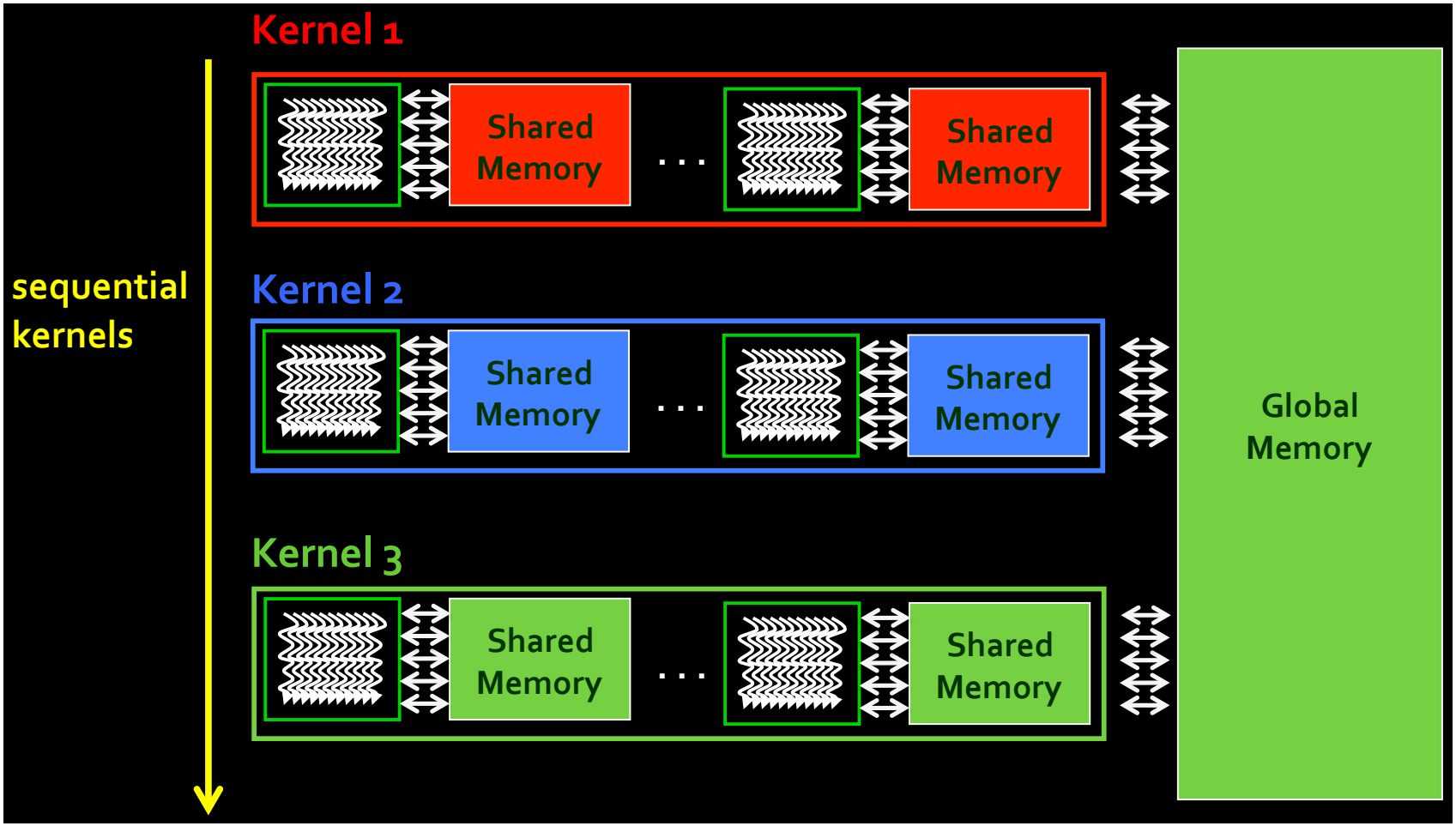
  Data access patterns will limit bandwidth achieved in practice

Common uses:

- Staging data transfers to/from CPU
- Staging data between kernel launches



Data persistence



GPU thread synchronization

```
1  void __syncthreads();
```

- Synchronizes all threads in a block
- Generates barrier synchronization instruction
- No thread can pass this barrier until all threads in the block reach it
- Used to avoid RAW / WAR / WAW hazards when accessing shared memory

Recap on collaboration between threads

- Threads often need to collaborate
  - Cooperatively load/store common data sets
  - Share results or cooperate to produce a single result
  - Synchronize with each other
- Threads in the same block
  - Can communicate through shared and global memory
  - Can synchronize using fast synchronization hardware
- Threads in different blocks of the same grid
  - Cannot synchronize reliably
  - No guarantee that both threads are alive at the same time

Blocks can collaborate only in limited ways [advanced topic]

- Any possible interleaving of blocks is allowed
  - Blocks must be able to run to completion without pre-emption (that is, one cannot presume to temporarily interrupt a block, and plan to return to it later)
  - May run in any order, concurrently or sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK (e.g., schedule a task by adding to queue / grab a task and execute it)
  - shared lock: BAD; it may deadlock (e.g., all blocks running on SMs are waiting on the lock)

Note on host synchronization

- All kernel launches are asynchronous
  Control returns to CPU immediately
  Kernel executes after all previous CUDA calls have completed
- `cudaMemcpy()` is synchronous
  Control returns to CPU after copy completes
  Copy starts after all previous CUDA calls have completed
- `cudaThreadSynchronize()`: blocks until all previous CUDA calls complete

The big table of sizes and dimensions for 2.x

Max x-, y-, or z-dimension of a grid of thread blocks	65535
Max x- or y-dimension of a block	1024
Max z-dimension of a block	64
Max # threads per block	1024
Warp size	32
Max # resident blocks per SM	8
Max # resident threads per SM	1536
# 32-bit registers per SM	32 K
Max amount of shared mem per SM	48 KB
Amount of local memory per thread	512 KB
Max # instructions per kernel	512 M

References and books

- Google: `cuda-zone developer`
- Google: `cuda-training short course`
- University Of Illinois: ECE 498AL course
- Stanford University: CS193G course
- NVIDIA CUDA C Programming Guide, Version 4.1 11/18/2011
- "Programming Massively Parallel Processors: A Hands-on Approach," by David Kirk and Wen-Mei Hwu