Introduction to CUDA

CME343 / ME339 | 18 May 2011

James Balfour [jbalfour@nvidia.com]
NVIDIA Research

CUDA

- Programing system for machines with GPUs
  - Programming Language
  - Compilers
  - Runtime Environments
  - Drivers
  - Hardware

CUDA : Heterogeneous Parallel Computing

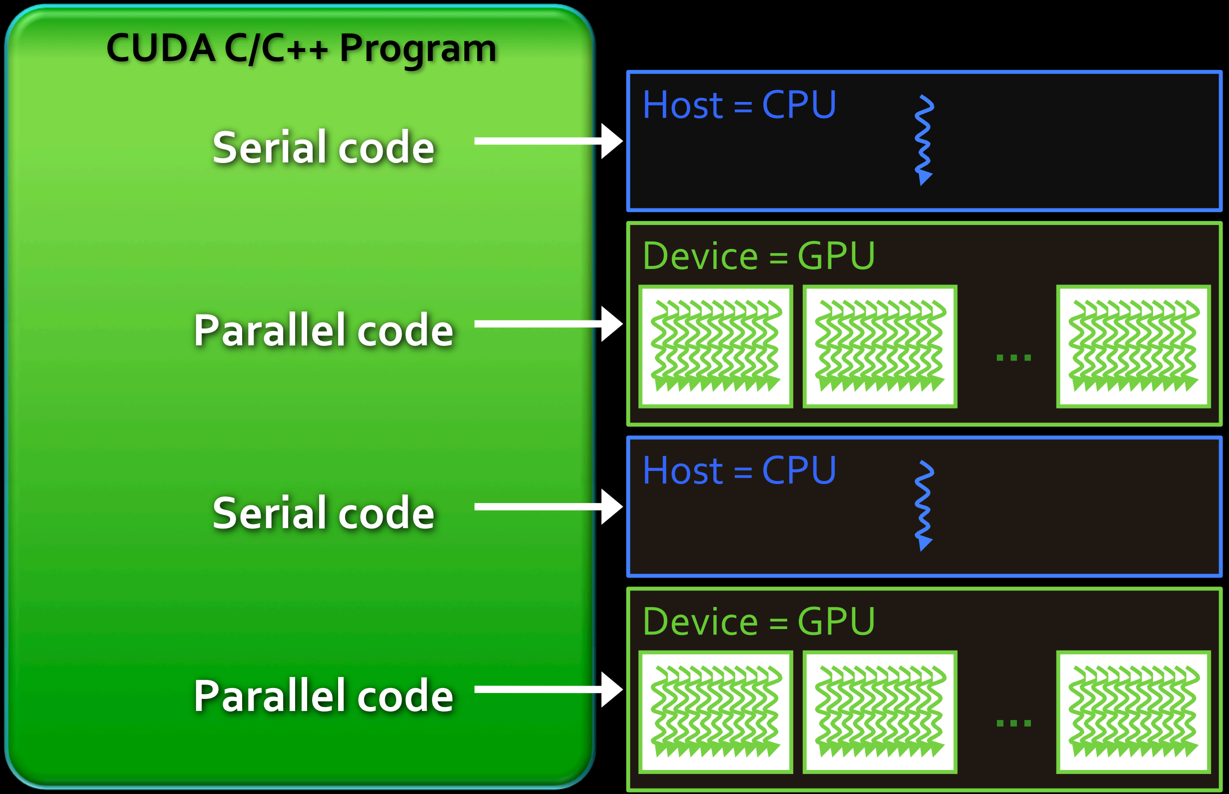
- CPU optimized for fast single-thread execution
  - Cores designed to execute 1 thread or 2 threads concurrently
  - Large caches attempt to hide DRAM access times
  - Cores optimized for low-latency cache accesses
  - Complex control-logic for speculation and out-of-order execution

+

- GPU optimized for high multi-thread throughput
  - Cores designed to execute many parallel threads concurrently
  - Cores optimized for data-parallel, throughput computation
  - Chips use extensive multithreading to tolerate DRAM access times

Anatomy of a CUDA C/C++ Program

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements

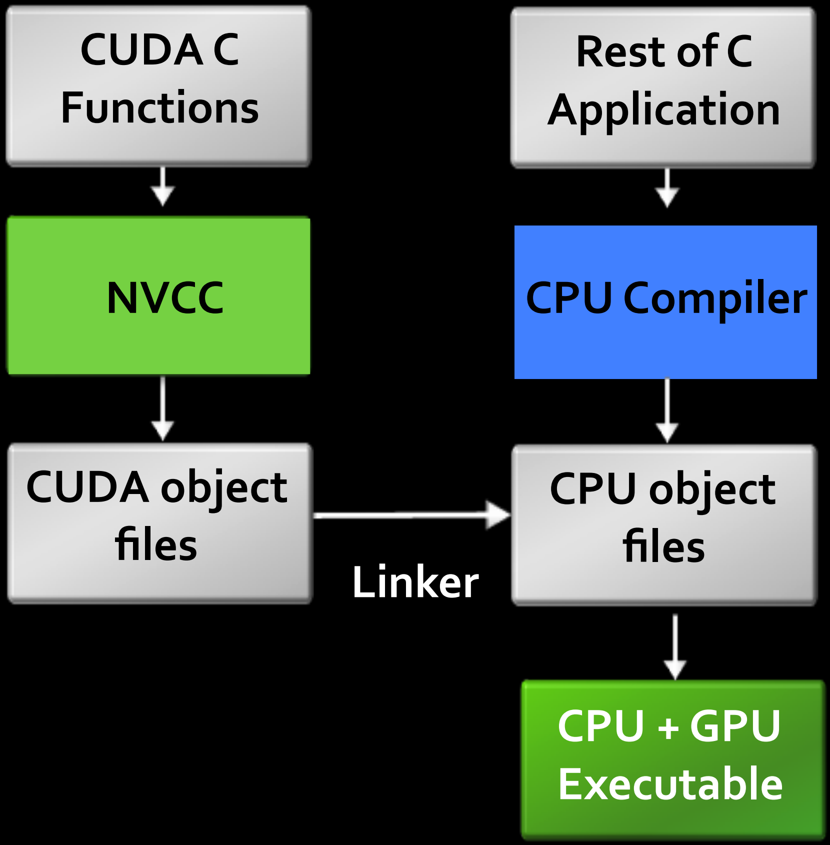


Compiling CUDA C/C++ Programs

```
// foo.cpp
int foo(int x)
{
  ...
}
float bar(float x)
{
  ...
}
```

```
// saxpy.cu
__global__ void saxpy(int n, float ... )
{
  int i = threadIdx.x + ... ;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

```
// main.cpp
void main( ) {
  float x = bar(1.0)
  if (x<2.0f)
    saxpy<<<...>>>(foo(1), ...);
  ...
}
```



CUDA C Functions

NVCC

CUDA object files

Rest of C Application

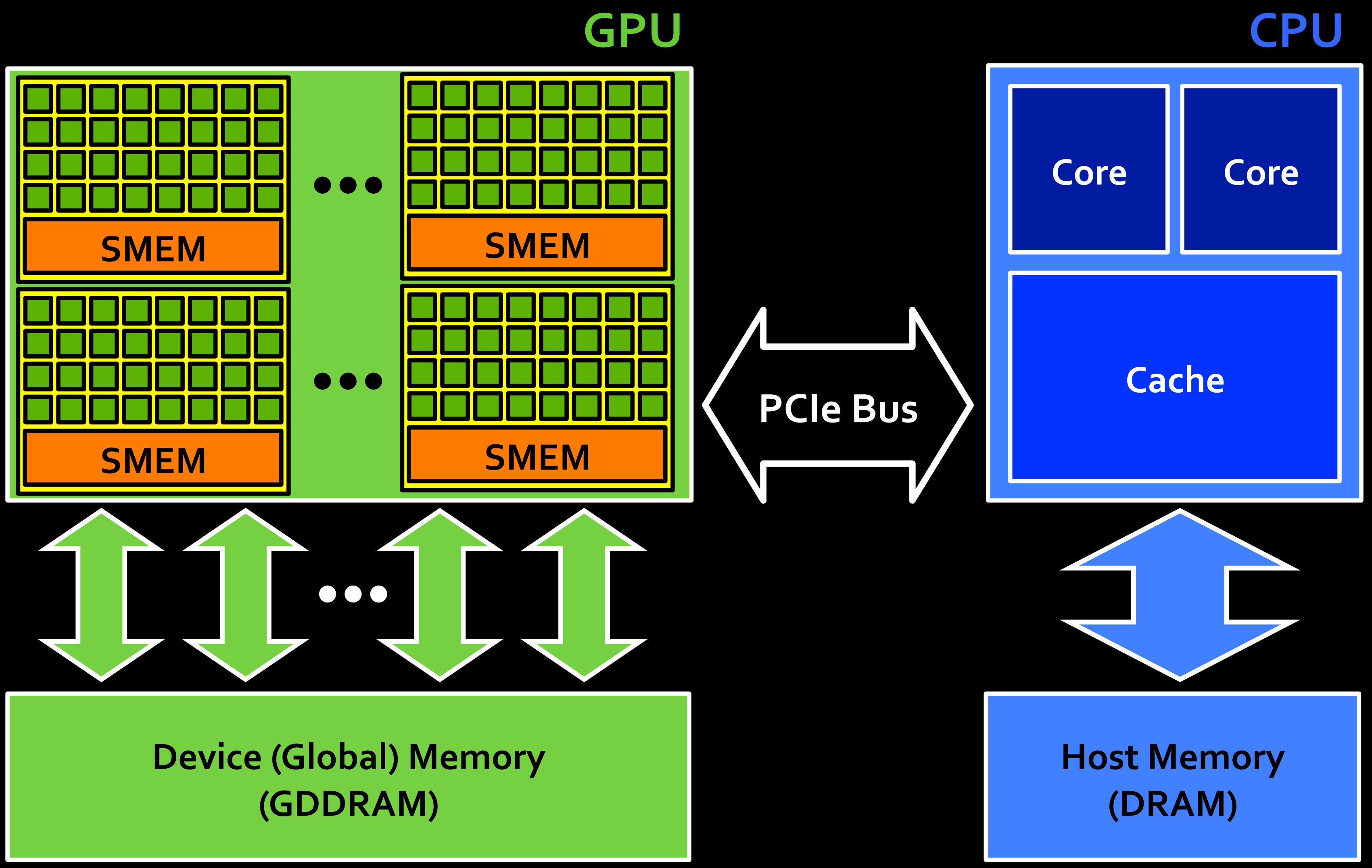
CPU Compiler

CPU object files

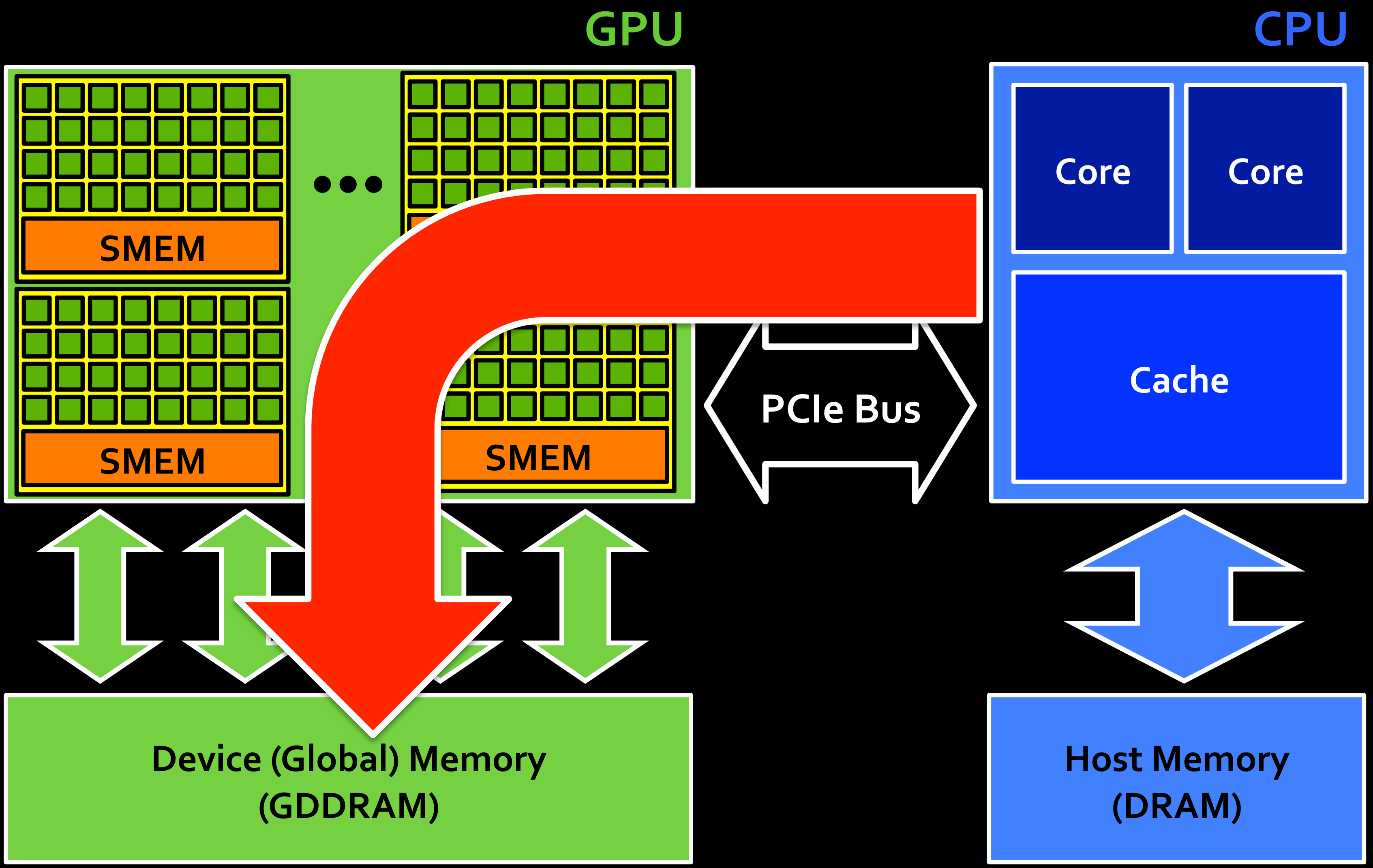
Linker

CPU + GPU Executable

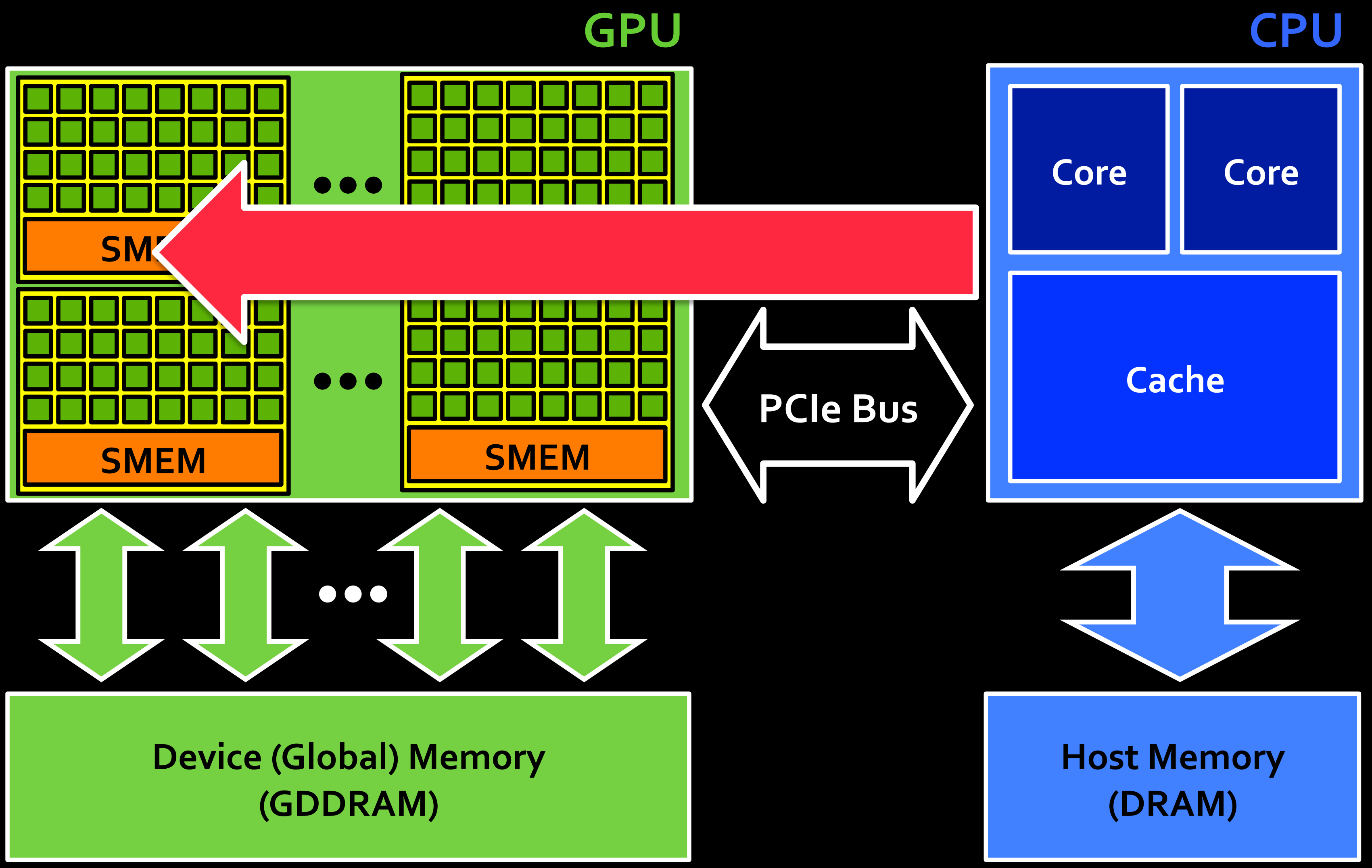
Canonical execution flow



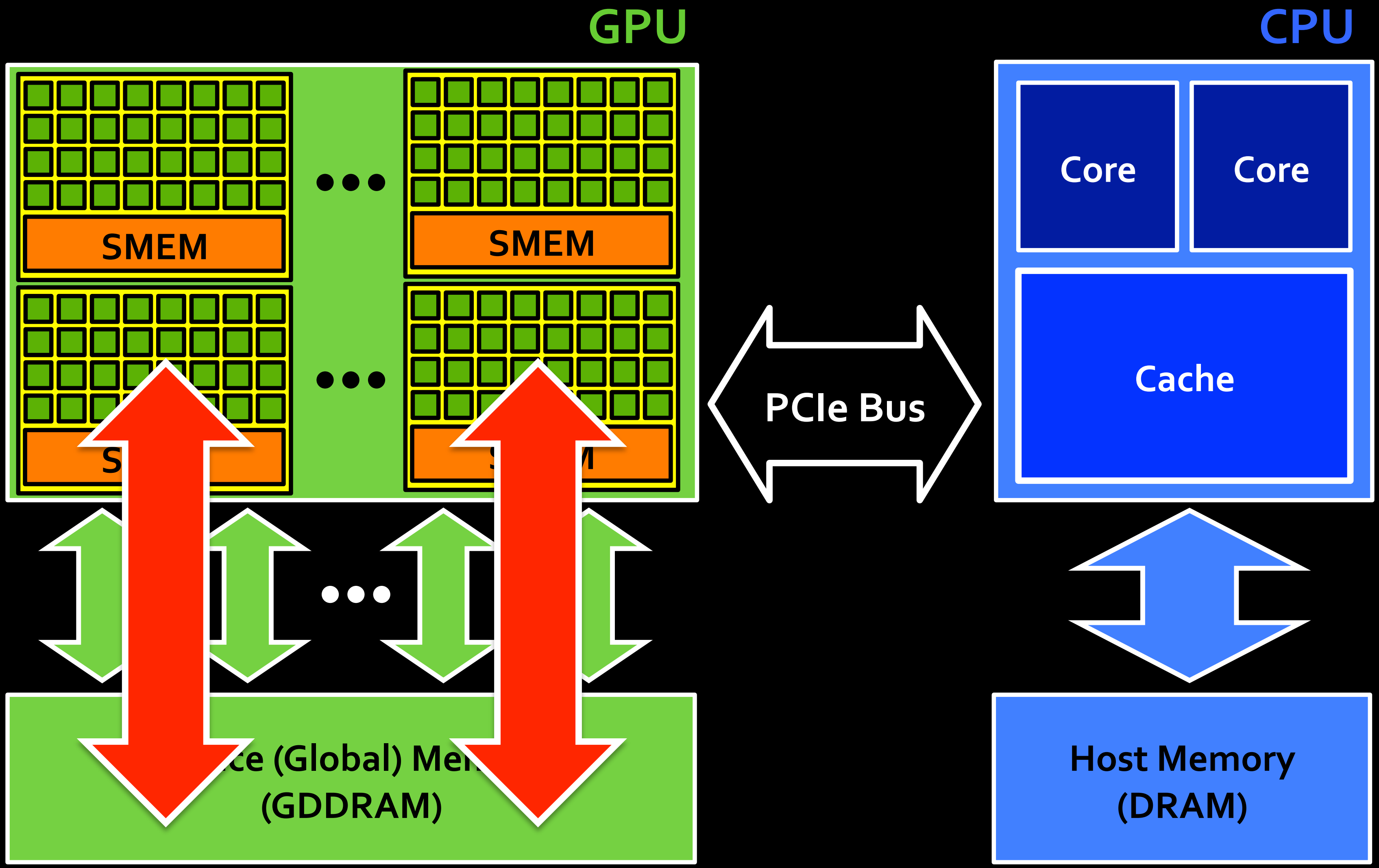
Step 1 – copy data to GPU memory



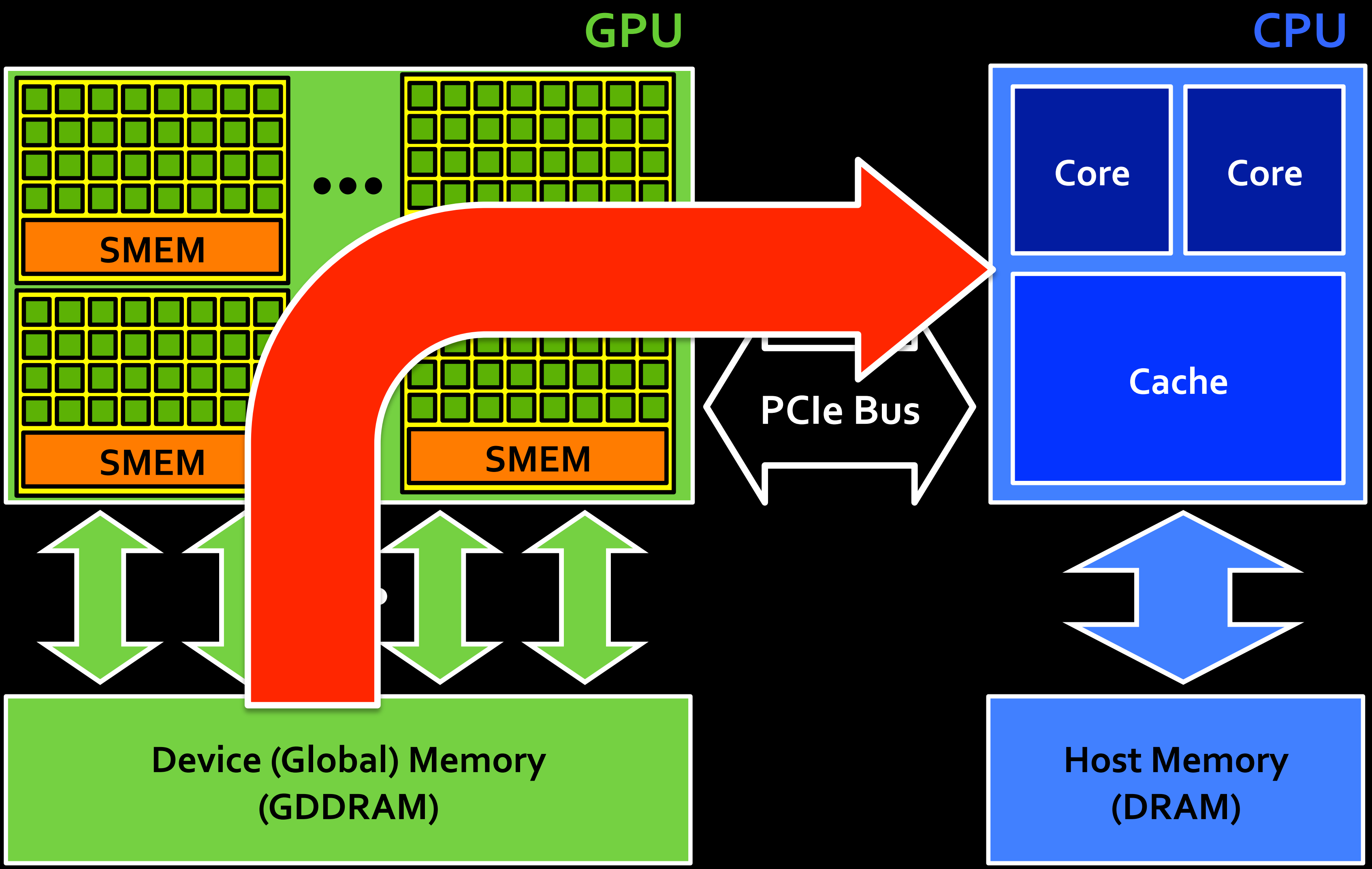
Step 2 – launch kernel on GPU



Step 3 – execute kernel on GPU



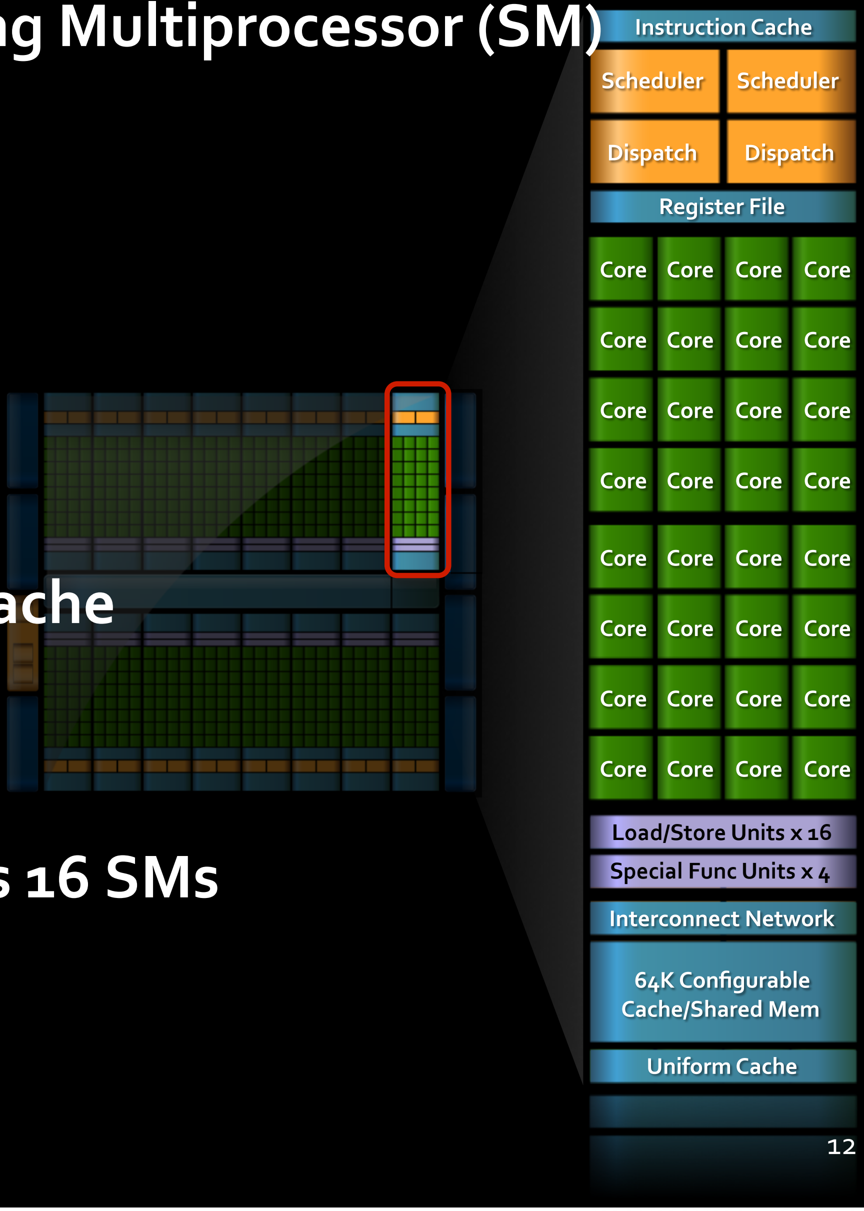
Step 4 – copy data to CPU memory



CUDA ARCHITECTURE

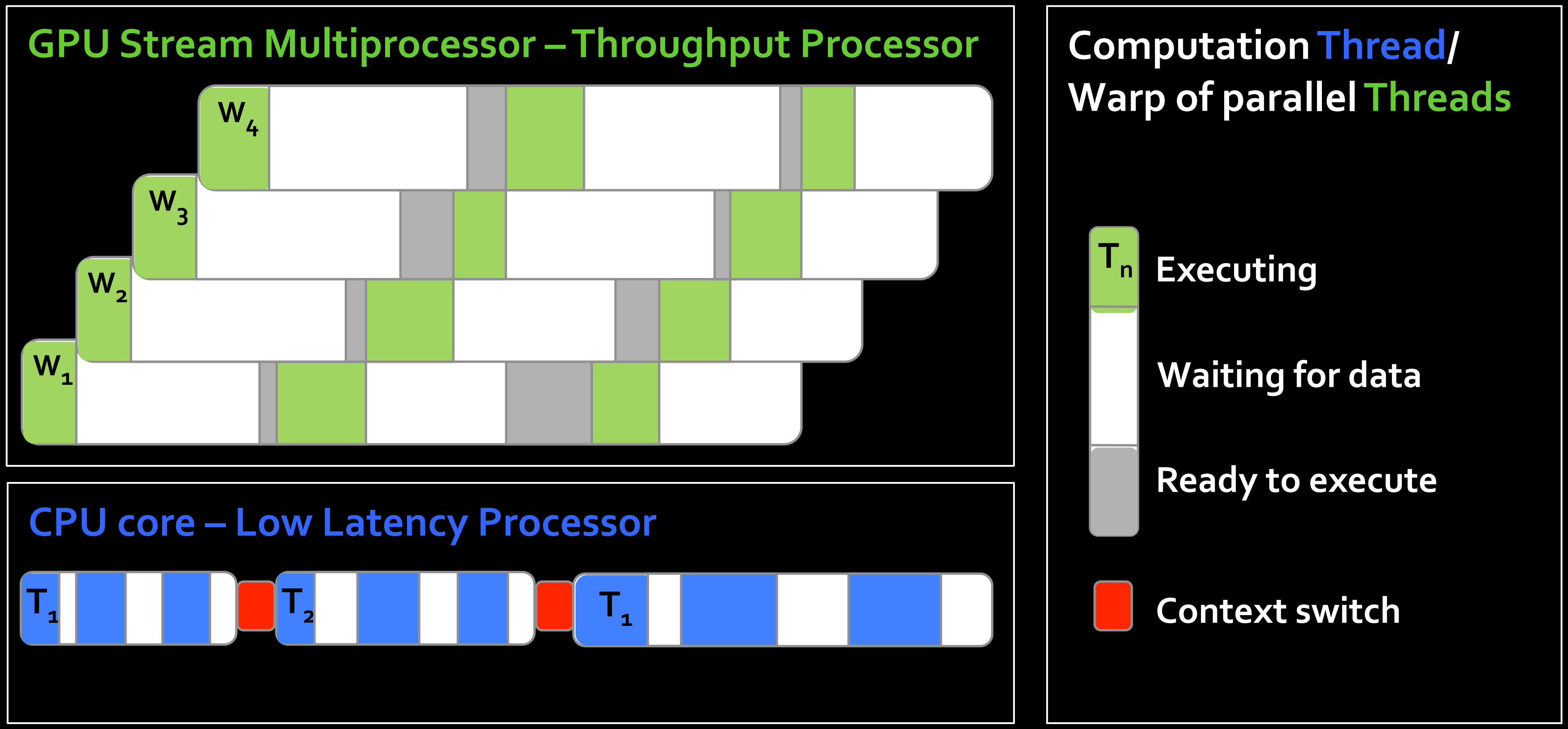
Contemporary (Fermi) GPU Architecture

- 32 CUDA Cores per Streaming Multiprocessor (SM)
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 Warp schedulers per SM
  - 1,536 concurrent threads
- 4 special-function units
- 64KB shared memory + L1 cache
- 32K 32-bit registers
- Fermi GPUs have as many as 16 SMs
  - 24,576 concurrent threads



Multithreading

- CPU architecture attempts to minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps



GPU Stream Multiprocessor – Throughput Processor

CPU core – Low Latency Processor

Computation Thread/
Warp of parallel Threads

Executing

Waiting for data

Ready to execute

Context switch

CUDA PROGRAMMING MODEL

CUDA Kernels

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel
  - Kernel launches create thousands of CUDA threads efficiently

CPU	Host	Executes functions
GPU	Device	Executes kernels

- CUDA threads
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously
- Kernel launches create hierarchical groups of threads
  - Threads are grouped into Blocks, and Blocks into Grids
  - Threads and Blocks represent different levels of parallelism

CUDA C : C with a few keywords

- Kernel : function that executes on device (GPU) and can be called from host (CPU)
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables

- Functions must be declared with a qualifier

  __global__ : GPU kernel function launched by CPU, must return void

  __device__ : can be called from GPU functions

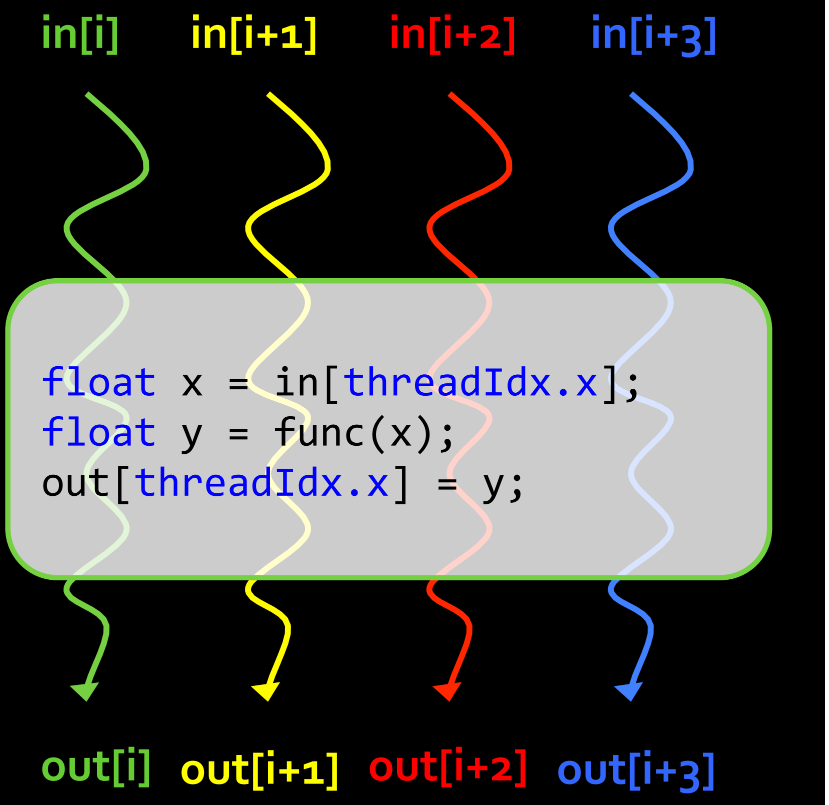
  __host__ : can be called from CPU functions (default)

  __host__ and __device__ qualifiers can be combined

- Qualifiers determines how functions are compiled
  - Controls which compilers are used to compile functions

CUDA Kernels : Parallel Threads

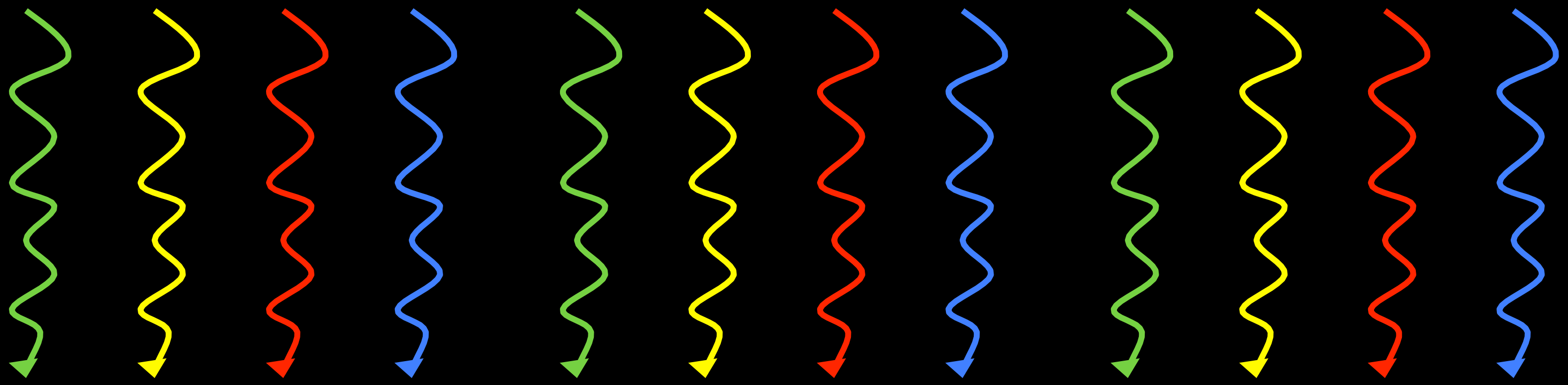
- A kernel is a function executed on the GPU as an array of parallel threads
- All threads execute the same kernel code, but can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



```
float x = in[threadIdx.x];
float y = func(x);
out[threadIdx.x] = y;
```

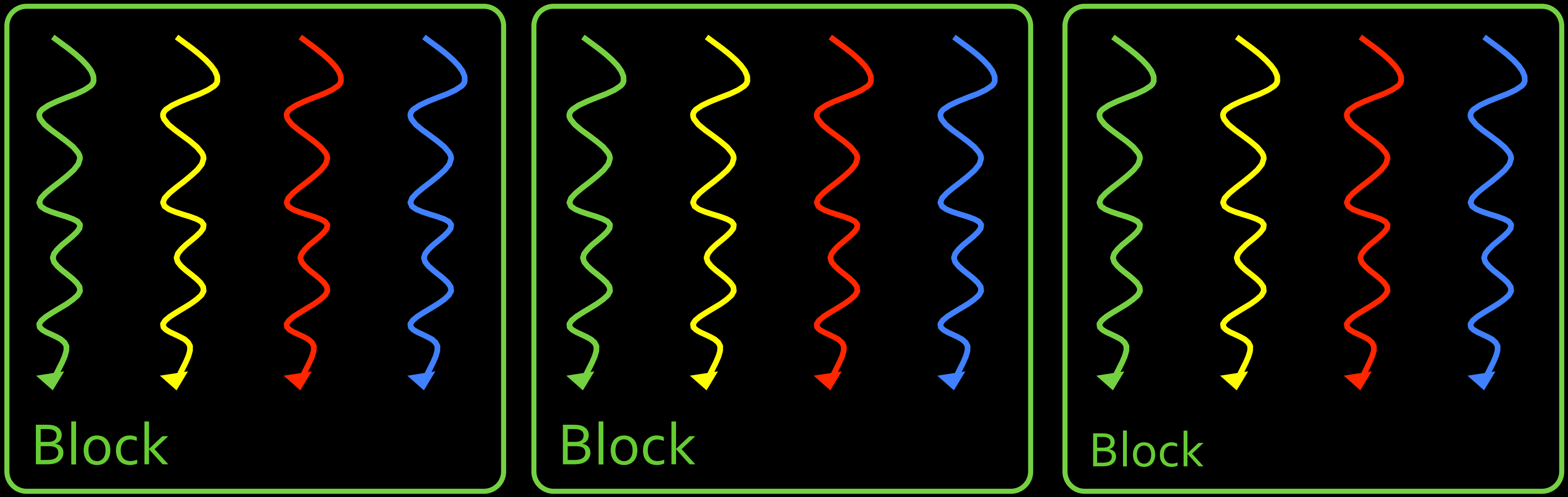
CUDA THREADS

CUDA Thread Organization



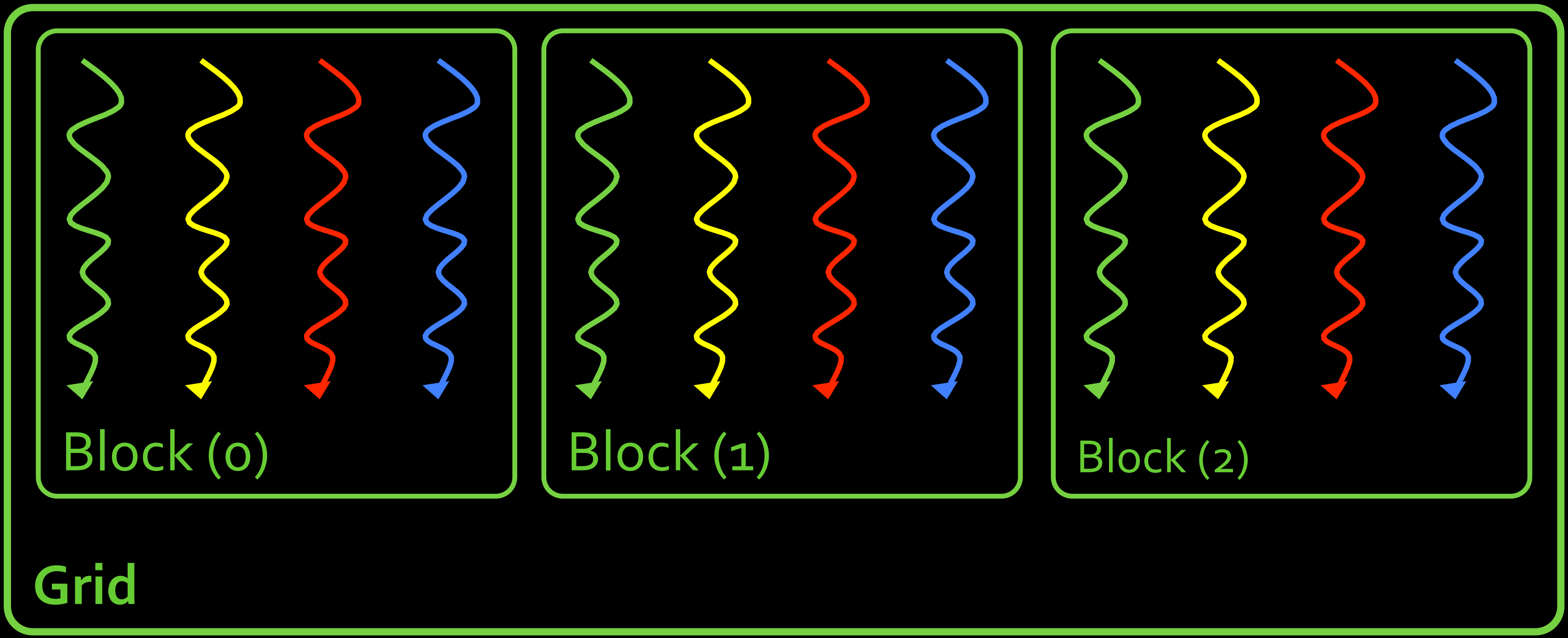
- GPUs can handle thousands of concurrent threads
- CUDA programming model supports even more
  - Allows a kernel launch to specify more threads than the GPU can execute concurrently
  - Helps to amortize kernel launch times

Blocks of threads



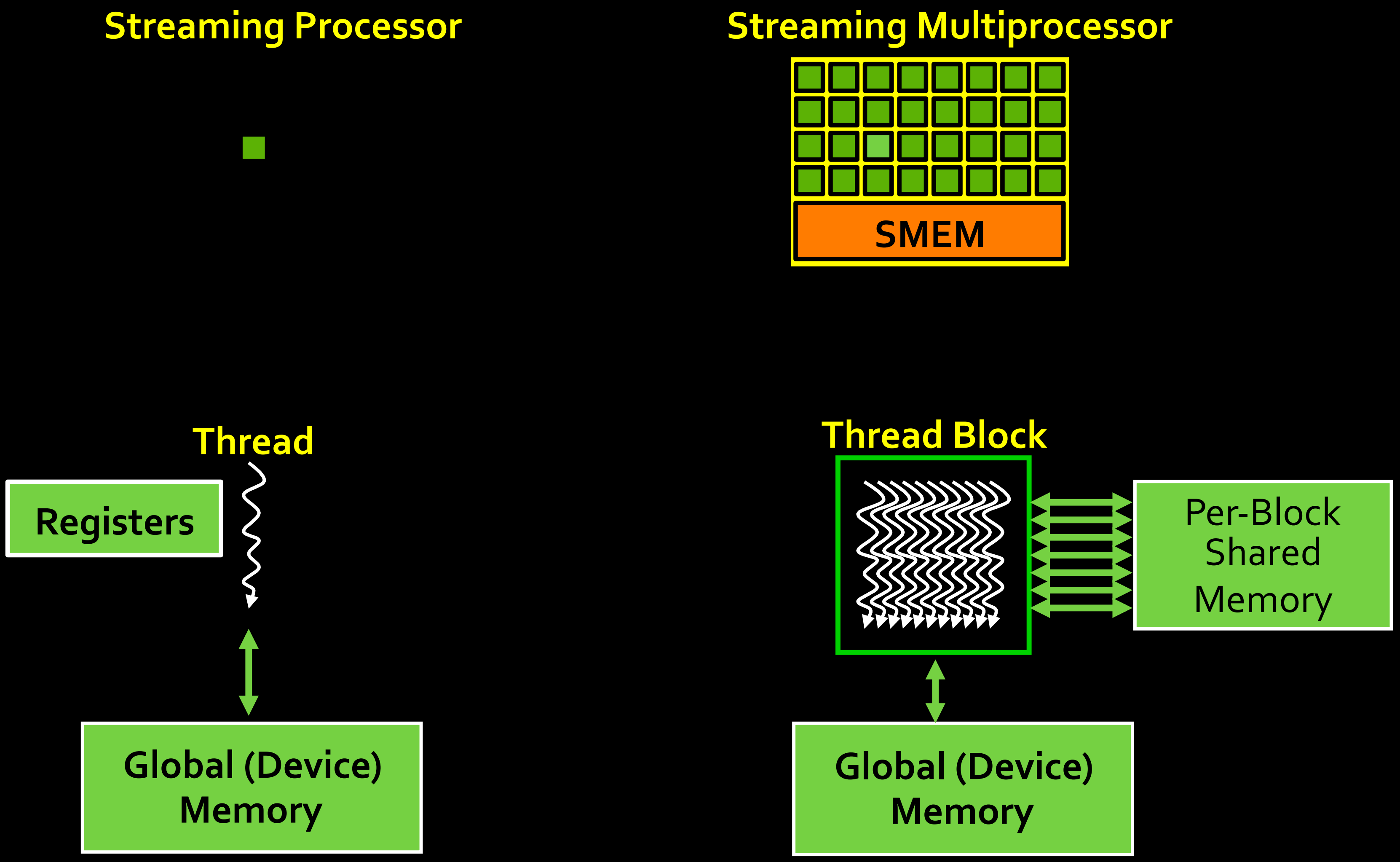
- Threads are grouped into **blocks**

Grids of blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Blocks execute on Streaming Multiprocessors



Streaming Processor

Streaming Multiprocessor

SMEM

Thread

Registers

Global (Device) Memory

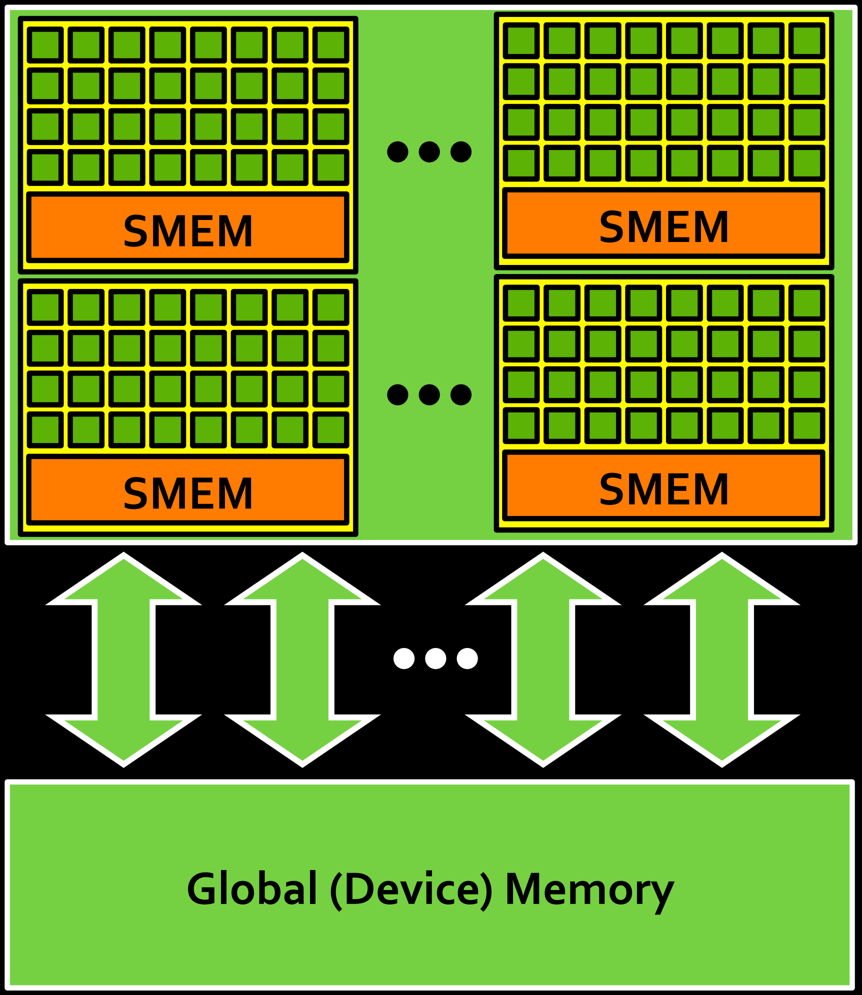
Thread Block

Per-Block Shared Memory

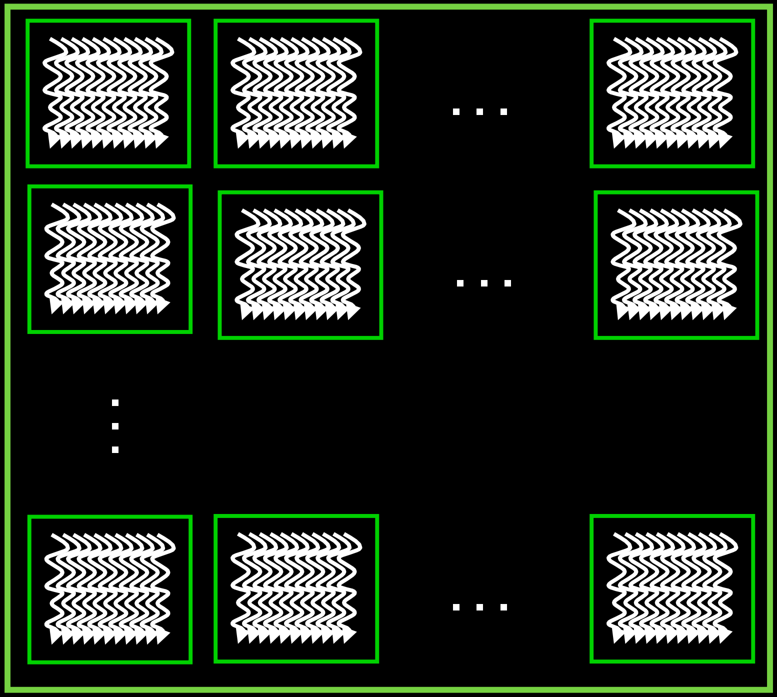
Global (Device) Memory

Grids of blocks executes across GPU

GPU



Grid of Blocks

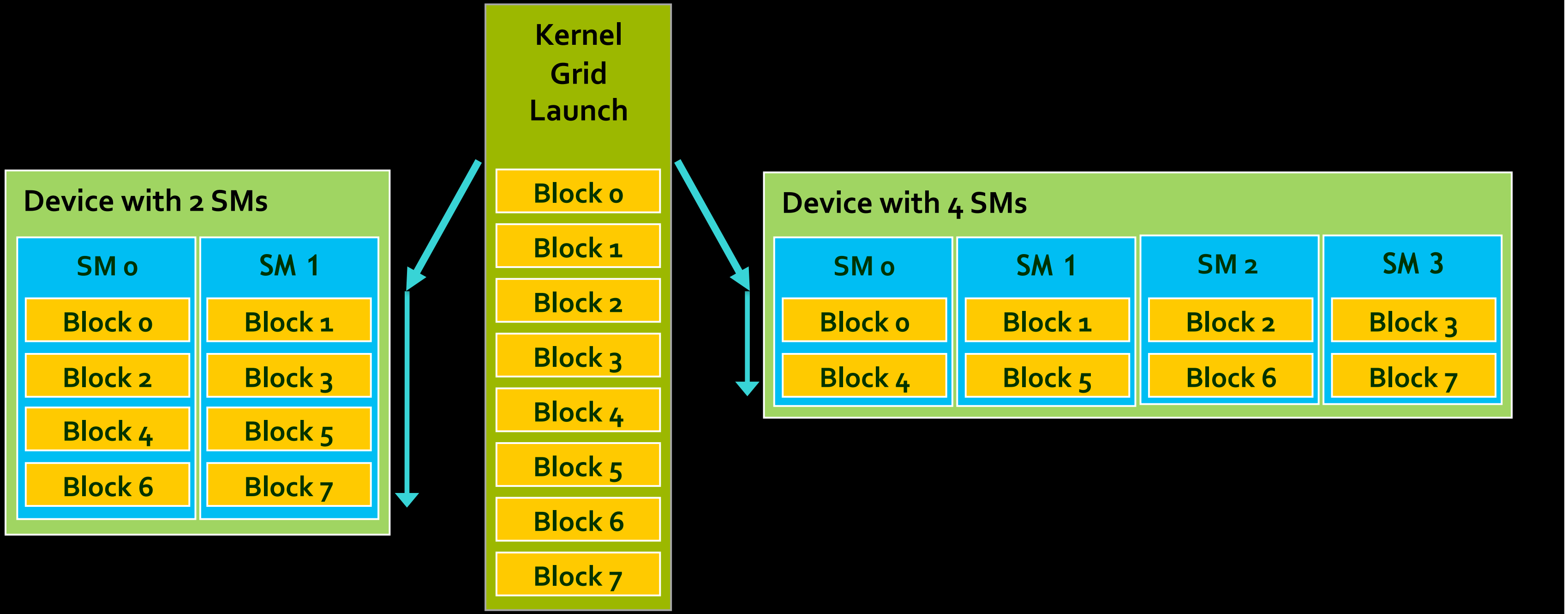


Kernel Execution Recap

- 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 in 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 from multiple grids may execute concurrently
  - A device can execute multiple kernels concurrently

Block abstraction provides scalability

- Blocks may execute in arbitrary order, concurrently or sequentially, and parallelism increases with resources
  - Depends on when execution resources become available
- Independent execution of blocks provides scalability
  - Blocks can be distributed across any number of SMs



Kernel Grid Launch

Block 0

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Device with 2 SMs

SM 0	SM 1
Block 0	Block 1
Block 2	Block 3
Block 4	Block 5
Block 6	Block 7

Device with 4 SMs

SM 0	SM 1	SM 2	SM 3
Block 0	Block 1	Block 2	Block 3
Block 4	Block 5	Block 6	Block 7

Blocks also enable efficient collaboration

- 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

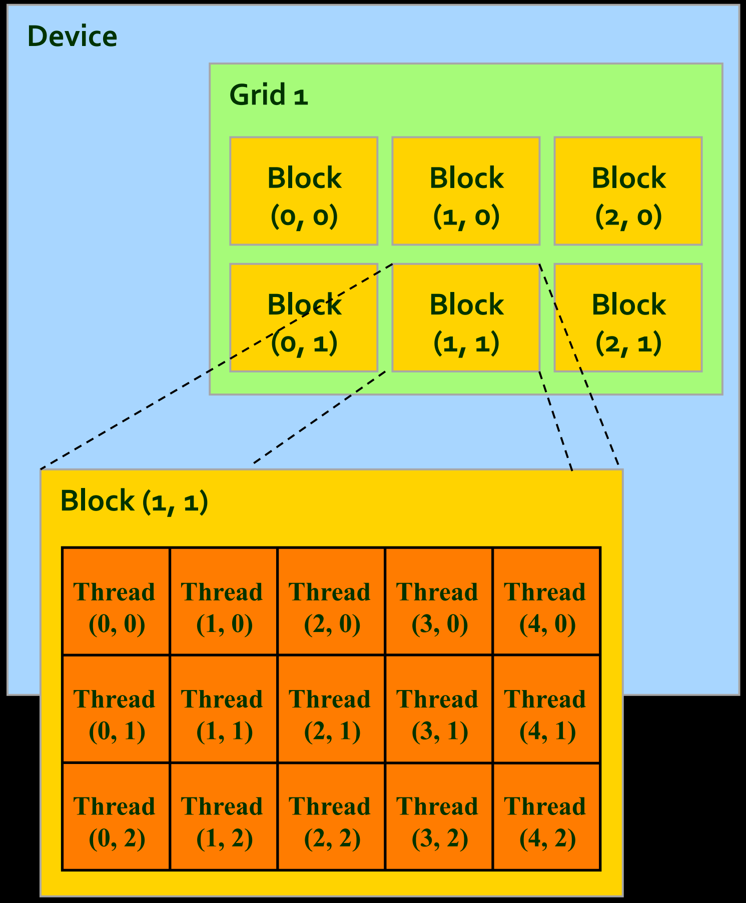
- Threads in different blocks of the same grid
  - Cannot synchronize reliably
  - No guarantee that both threads are alive at the same time

Blocks must be independent

- Any possible interleaving of blocks is allowed
  - Blocks presumed to run to completion without pre-emption
  - May run in any order, concurrently or sequentially
- Programs that depend on block execution order within grid for correctness are not well formed
  - May deadlock or return incorrect results
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock

Thread and Block ID and Dimensions

- Threads
  - 3D IDs, unique within a block
- Thread Blocks
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables
  - `threadIdx`, `blockIdx`
  - `blockDim`, `gridDim`
- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads



Device

Grid 1

Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)

Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Examples of Indexes and Indexing

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}                               Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
                                Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}                               Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Example of 2D indexing

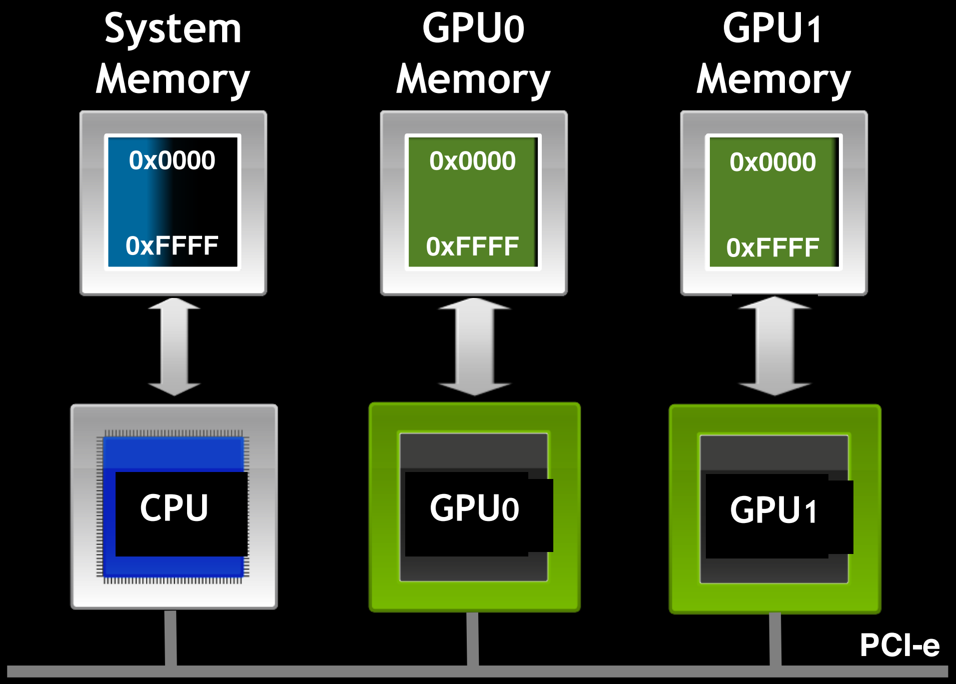
```
__global__ void kernel(int *a, int dimx, int dimy)
{
  int ix  = blockIdx.x*blockDim.x + threadIdx.x;
  int iy  = blockIdx.y*blockDim.y + threadIdx.y;
  int idx = iy * dimx + ix;

  a[idx]  = a[idx]+1;
}
```

CUDA MEMORY MODEL

Independent address spaces

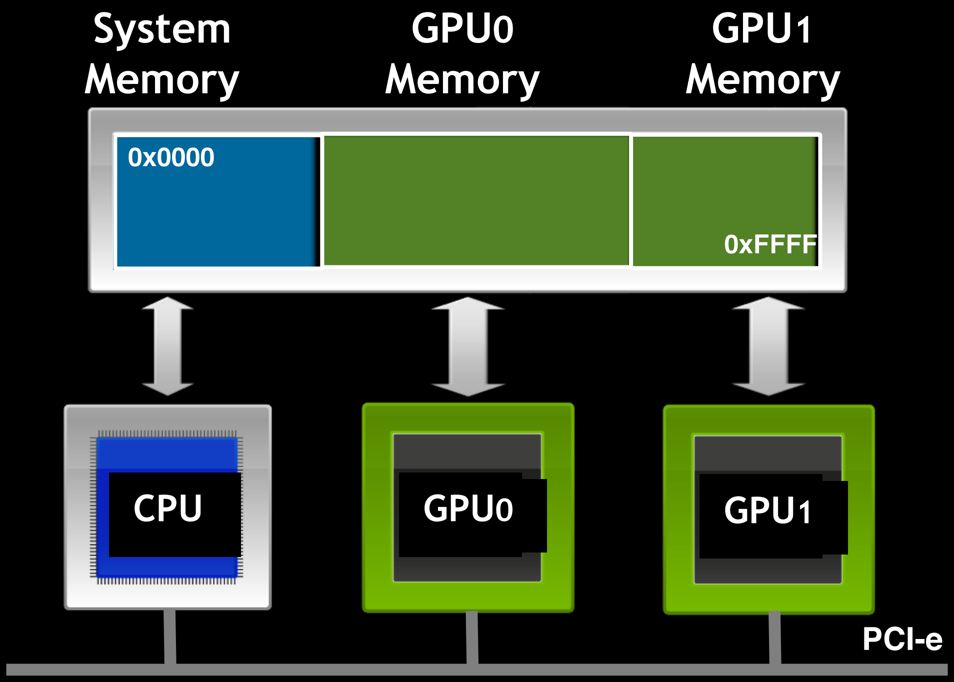
- CPU and GPU have independent memory systems
  — PCIe bus transfers data between CPU and GPU memory systems

- Typically, CPU thread and GPU threads access what are logically different, independent virtual address spaces



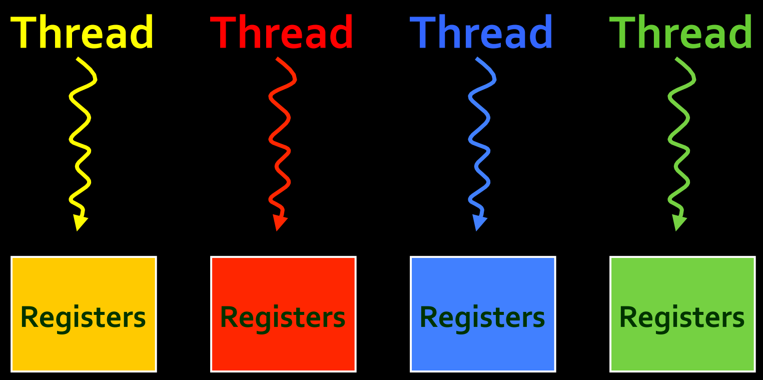
Independent address spaces: consequences

- Cannot reliably determine whether a pointer references a host (CPU) or device (GPU) address from the pointer value
  - Dereferencing CPU/GPU pointer on GPU/CPU will likely cause crash
- Unified virtual addressing (UVA) in CUDA 4.0
  - One virtual address space shared by CPU thread and GPU threads



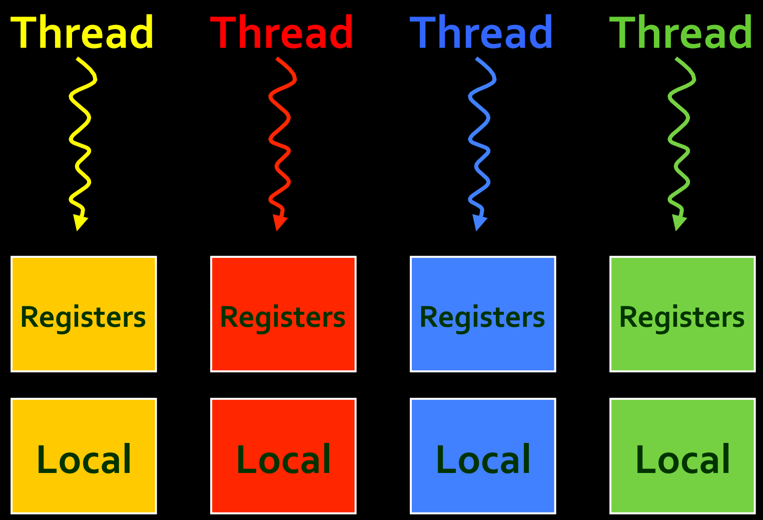
CUDA Memory Hierarchy

- Thread
  - Registers



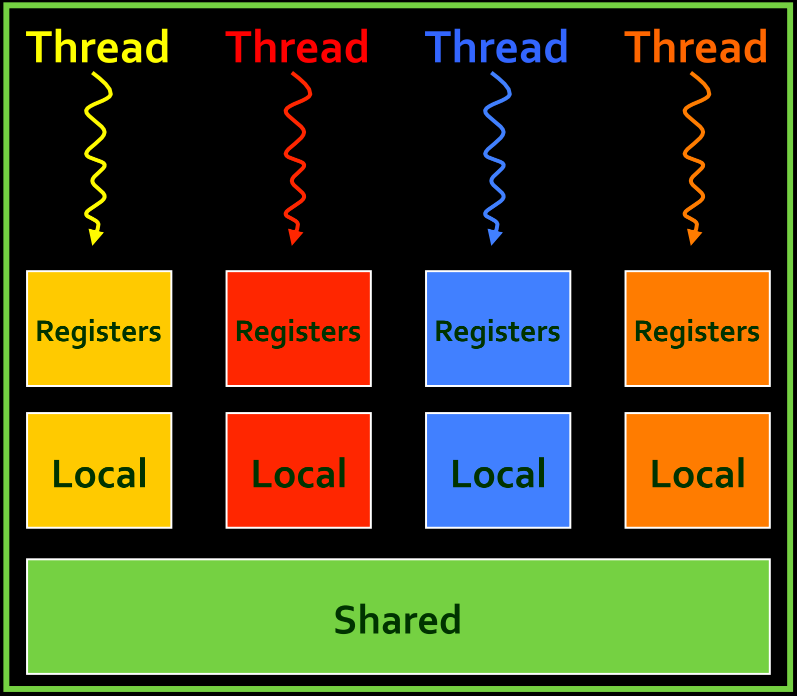
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory



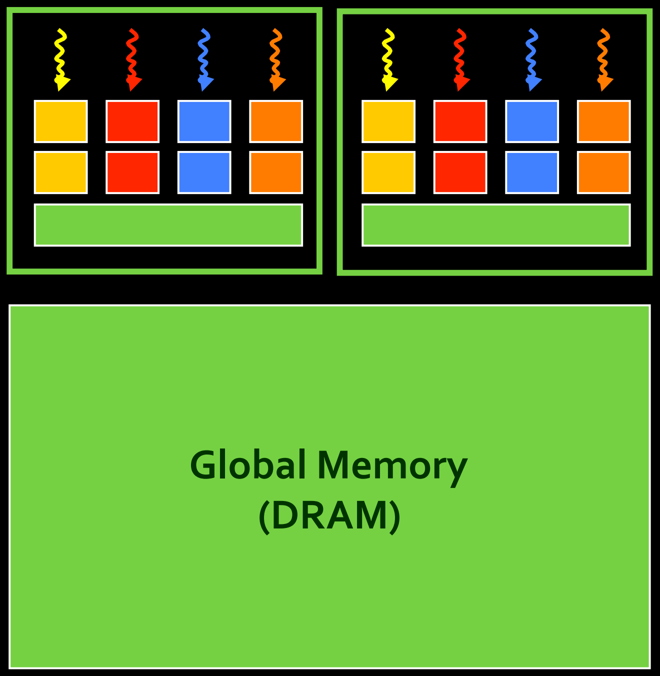
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global Memory



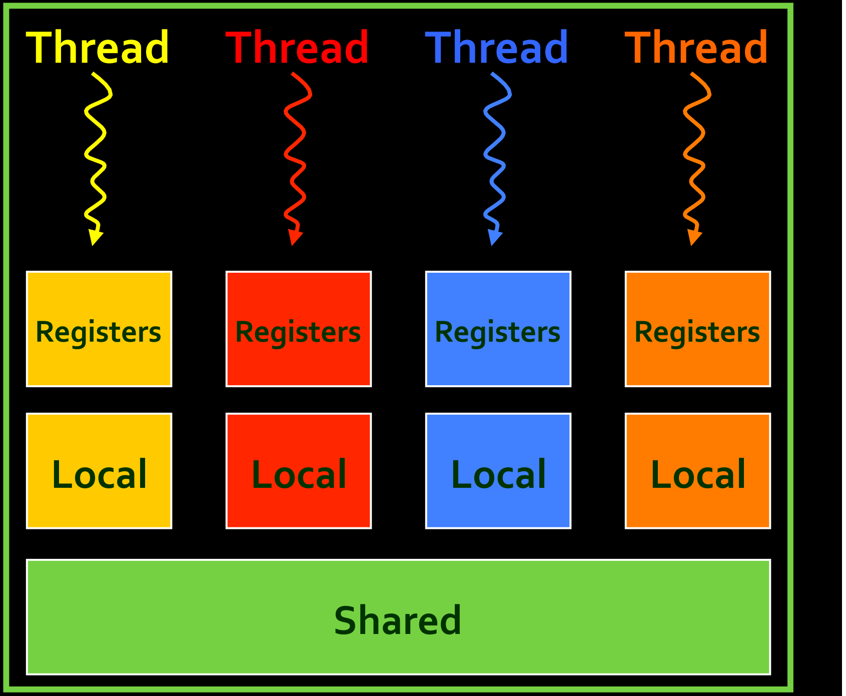
Shared Memory

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

- Allocated per thread block
- Scope: threads in block
- Data lifetime: same as block
- Capacity: small (about 48kB)
- Latency: a few cycles
- Bandwidth: very high

  SM: 32 * 4 B * 1.15 GHz / 2 = 73.6 GB/s

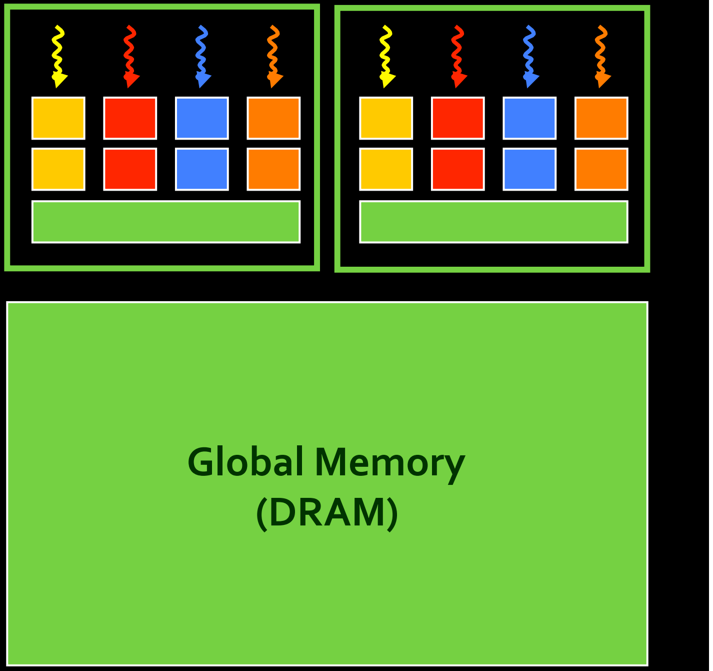
  GPU: 14 * 32 * 4 B * 1.15 GHz / 2 = 1.03 TB/s



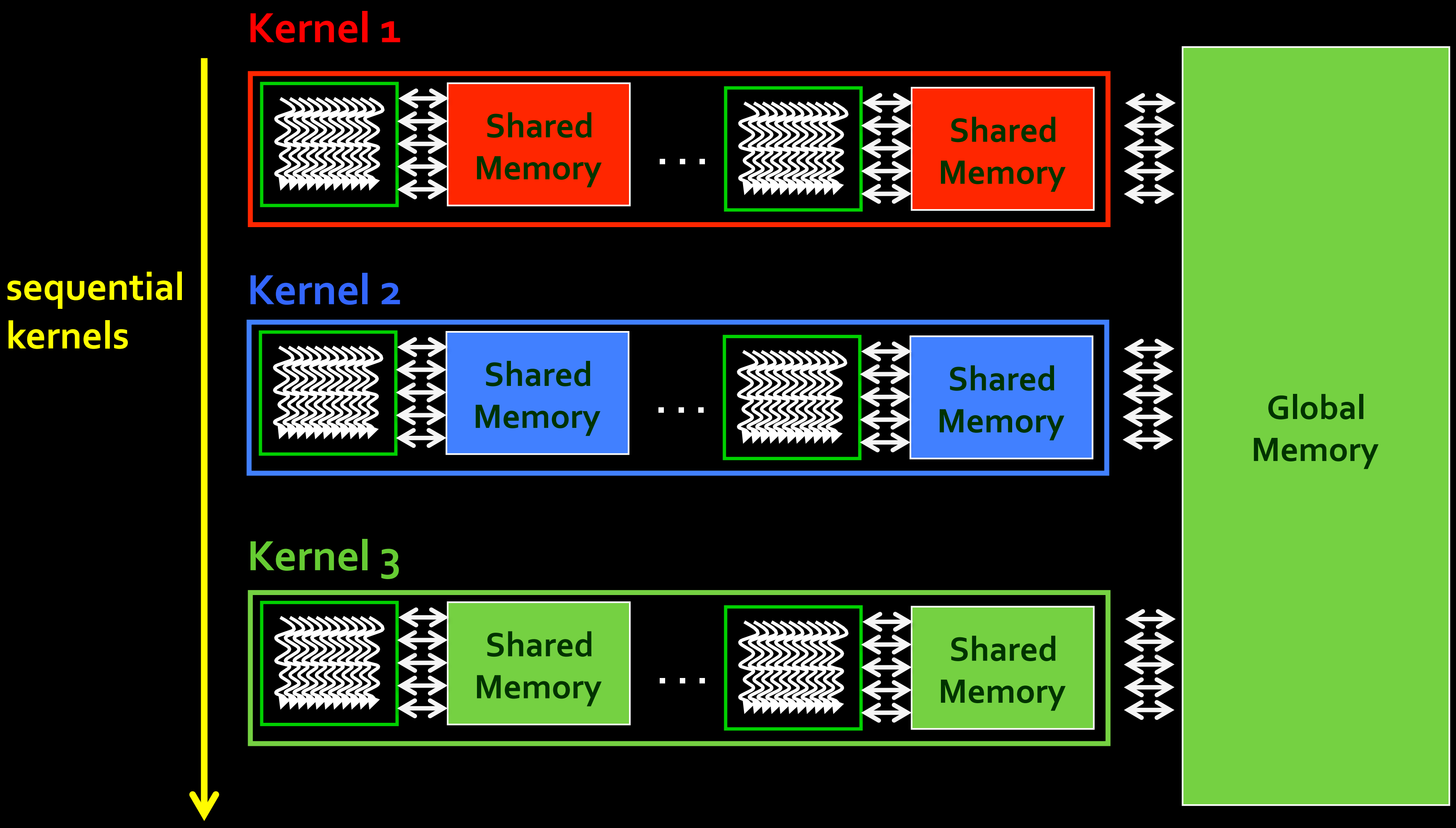
- Common uses
  - Sharing data among threads in a block
  - User-managed cache (to reduce global memory accesses)

Global Memory

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



Communication and Data Persistence



Managing Device (GPU) Memory

- Host (CPU) manages device (GPU) memory
  - cudaMalloc (void ** pointer, size_t num_bytes)
  - cudaMemset (void* pointer, int value, size_t count)
  - cudaFree(void* pointer)

- Example: allocate and initialize array of 1024 ints on device

```
// allocate and initialize int x[1024] on device
int n = 1024;
int num_bytes = 1024*sizeof(int);
int* d_x = 0;  // holds device pointer
cudaMalloc((void**)&d_x,  num_bytes);
cudaMemset(d_x, 0, num_bytes);
cudaFree(d_x);
```

Transferring Data

-
```
cudaMemcpy(void* dst, void* src, size_t num_bytes,
                                enum cudaMemcpyKind
direction);
```
  - returns to host thread after the copy completes
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete
- Direction controlled by enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- CUDA also provides non-blocking
  - Allows program to overlap data transfer with concurrent computation on host and device
  - Need to ensure that source locations are stable and destination locations are not accessed

CUDA EXAMPLE

Example: SAXPY Kernel [1/4]

```
// [compute] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example: SAXPY Kernel [1/4]

```
// [computes] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Device Code

```
int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example: SAXPY Kernel [1/4]

```
// [computes] for (i=0; i < n; i++) y[i] = a * x[i] + y[i];
//   Each thread processes one element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Host Code

```
int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [3/4]

```
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result…


  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y;

  return 0;
}
```

Example: SAXPY Kernel [3/4]

```
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result…


  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y;

  return 0;
}
```

Example: SAXPY Kernel [4/4]

```c
void saxpy_serial(int n, float a, float* x, float* y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
// invoke host SAXPY function
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```c
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)  y[i] = a*x[i] + y[i];
}
// invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA C Code

CUDA DEVELOPMENT RESOURCES

CUDA Programming Resources

- CUDA Toolkit
  - Compiler, libraries, and documentation
  - Free download for Windows, Linux, and MacOS
- GPU Computing SDK
  - Code samples
  - Whitepapers
- Instructional materials on NVIDIA Developer site
  - CUDA introduction & optimization webinar: slides and audio
  - Parallel programming course at University of Illinois UC
  - Tutorials
  - Forums

GPU Tools

- Profiler
  - Available for all supported OSs
  - Command-line or GUI
  - Sampling signals on GPU for
    - Memory access parameters
    - Execution (serialization, divergence)

- Debugger
  - Linux: cuda-gdb
  - Windows: Parallel Nsight
  - Runs on the GPU

Questions?

jbalfour@nvidia.com

Acknowledgements

- Some slides derived from decks provided by Jared Hoberock and Cliff Woolley