Introduction to GPUs and CUDA

Isaac van Til
Texas State University- CS 3339
Fall 2018

Serial Performance Scaling is Over

- Cannot continue to scale processor frequencies (no 10GHz chips)
- Cannot continue to increase power consumption (melting chips)

Solution: Parallelism

- Instruction-level parallelism (superscalar, etc.)
  - Diminishing returns due to power constraints
- Thread-level parallelism (multicore & multithreading architecture)
- Data-level parallelism (vector units, GPU)

What is a GPU?

- Special purpose processor, designed for fast graphics processing
- Development driven by demand for increased quality in 3D graphics (gaming industry)
- Massively parallel
- Graphics pipeline- throughput oriented

CPU vs GPU

CPU

- Latency oriented
- Optimized for serial code performance
- Good for single complex tasks
- Multi-core

GPU

- Throughput oriented
- Massively parallel
- Optimized for performing many similar tasks simultaneously (think data parallelism)
- Many-core (cores are less complex)

CPU vs GPU

CPU

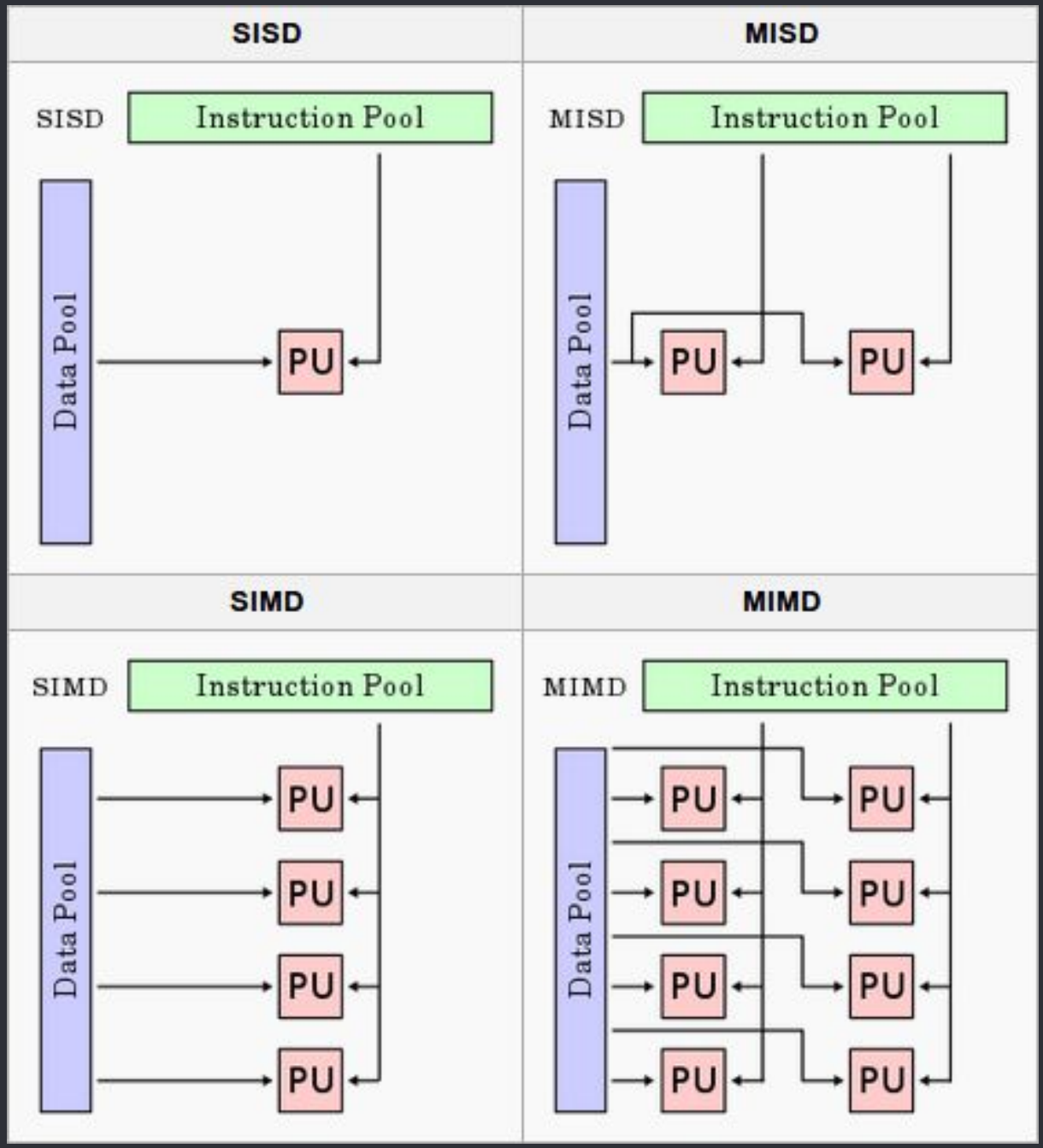
- Large cache
- More powerful ALUs
- Complex control mechanisms (branch prediction, etc.)

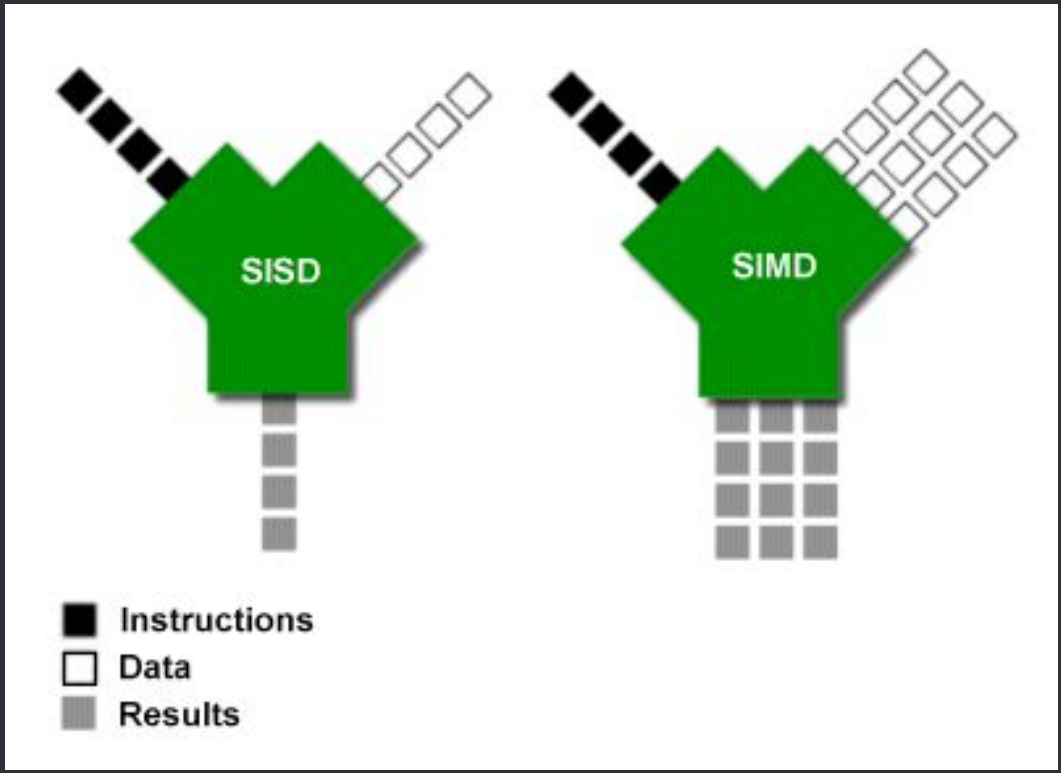
GPU

- Small cache
- More energy efficient ALUs
- Simple control (no branch prediction)

Intrinsic Parallelism in GPUs

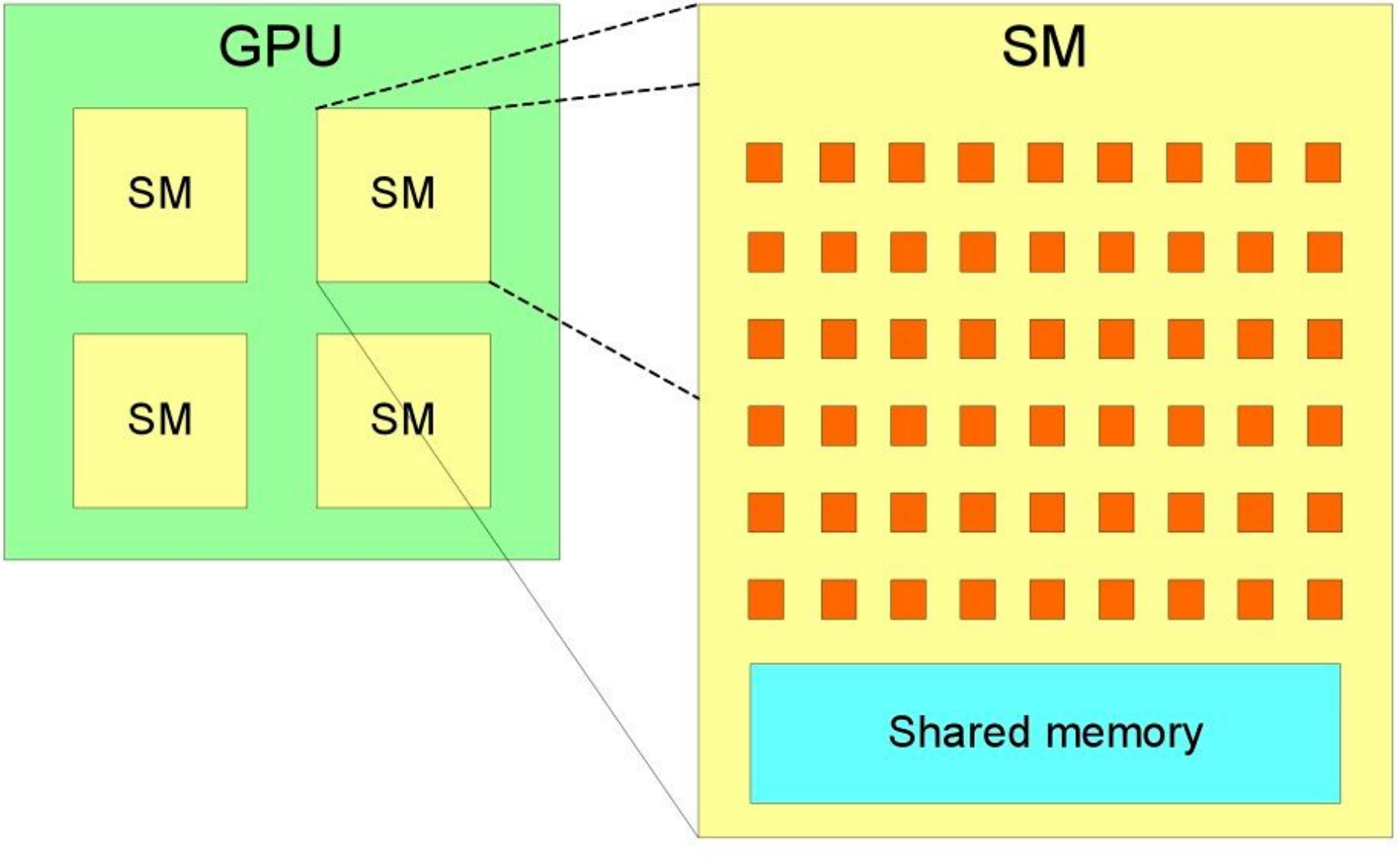
- Access to memory incurs a long latency
  - Mitigated in the CPU using caches
  - Mitigated in the GPU by supporting many threads (thousands) and having very fast switching between them
    - 1 stalled thread is OK is 100 are ready to run


SISD
SISD
Instruction Pool
Data Pool
PU
MISD
MISD
Instruction Pool
Data Pool
PU
PU
SIMD
SIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
MIMD
MIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
PU
PU
PU
PU


SISD
SIMD
Instructions
Data
Results

Data parallelism in GPUs

- Single Instruction Multiple Data
- Threads are not completely independent
- Ordered in thread blocks
- All threads in a block execute the same instruction
- Becomes more apparent with hardware design- Streaming Multiprocessors

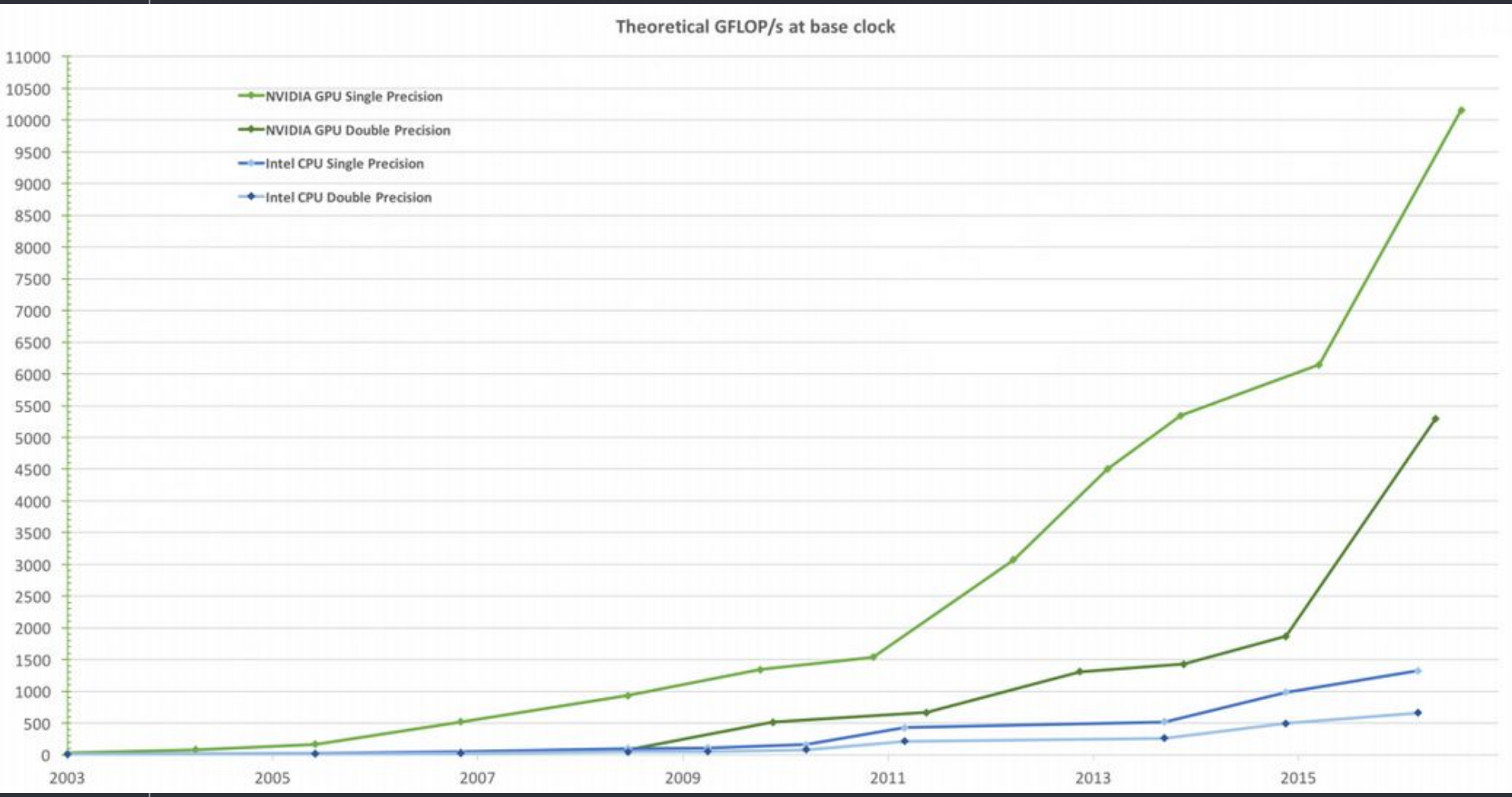

GPU
SM
SM
SM
SM
SM
Shared memory

Thread Parallelism in GPUs

- Threads run in groups of 32 called “warps”
- Hardware handles resource allocation & thread scheduling, while relying on threads to hide latency
- Threads have all resources needed to run
  - Any warp not waiting for something can run
  - Context switching is (basically) free

Advantages

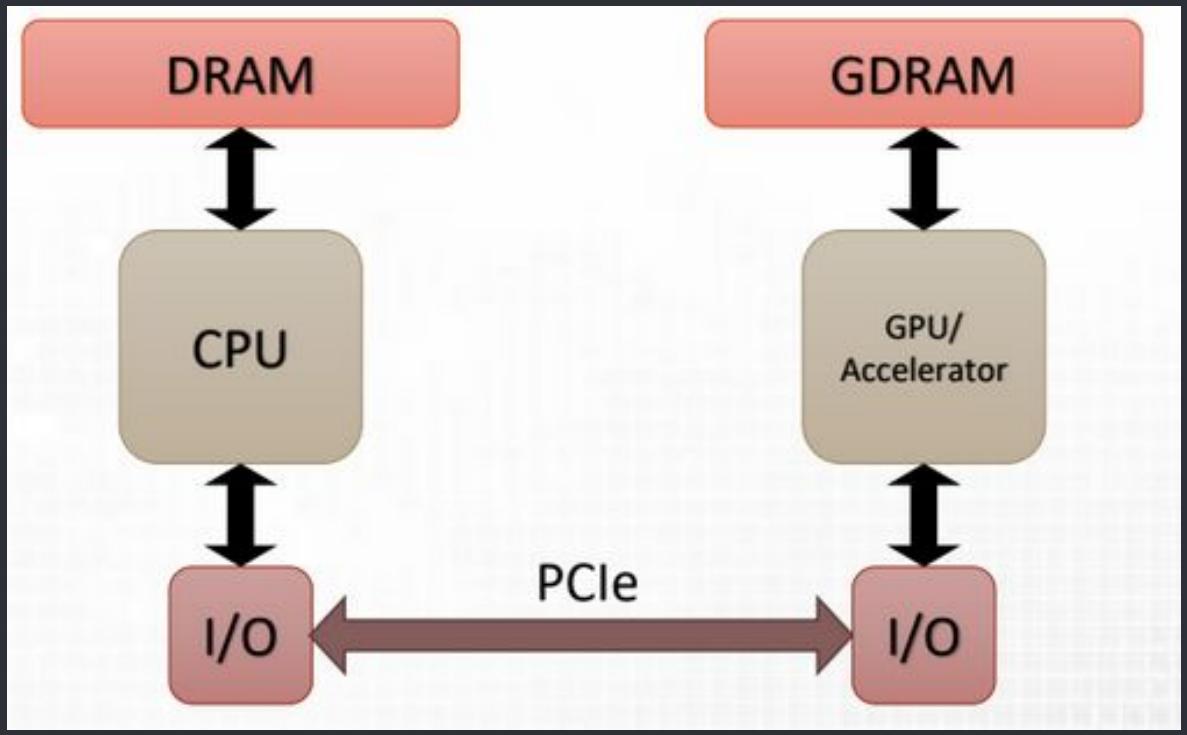
- VERY high performance and cost effectiveness
- But only for certain types of computations (data parallelism)
- Given enough parallelism in the problem, it is very easy to achieve high performance with very little effort


Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015

GPGPU

- GPUs are designed as numeric computing engines, will not perform well on CPU specific tasks
- General Purpose computation on Graphics Hardware
- Term coined by Mark Harris in 2002
- GPUs for non graphics applications

GPU/CPU Cooperation

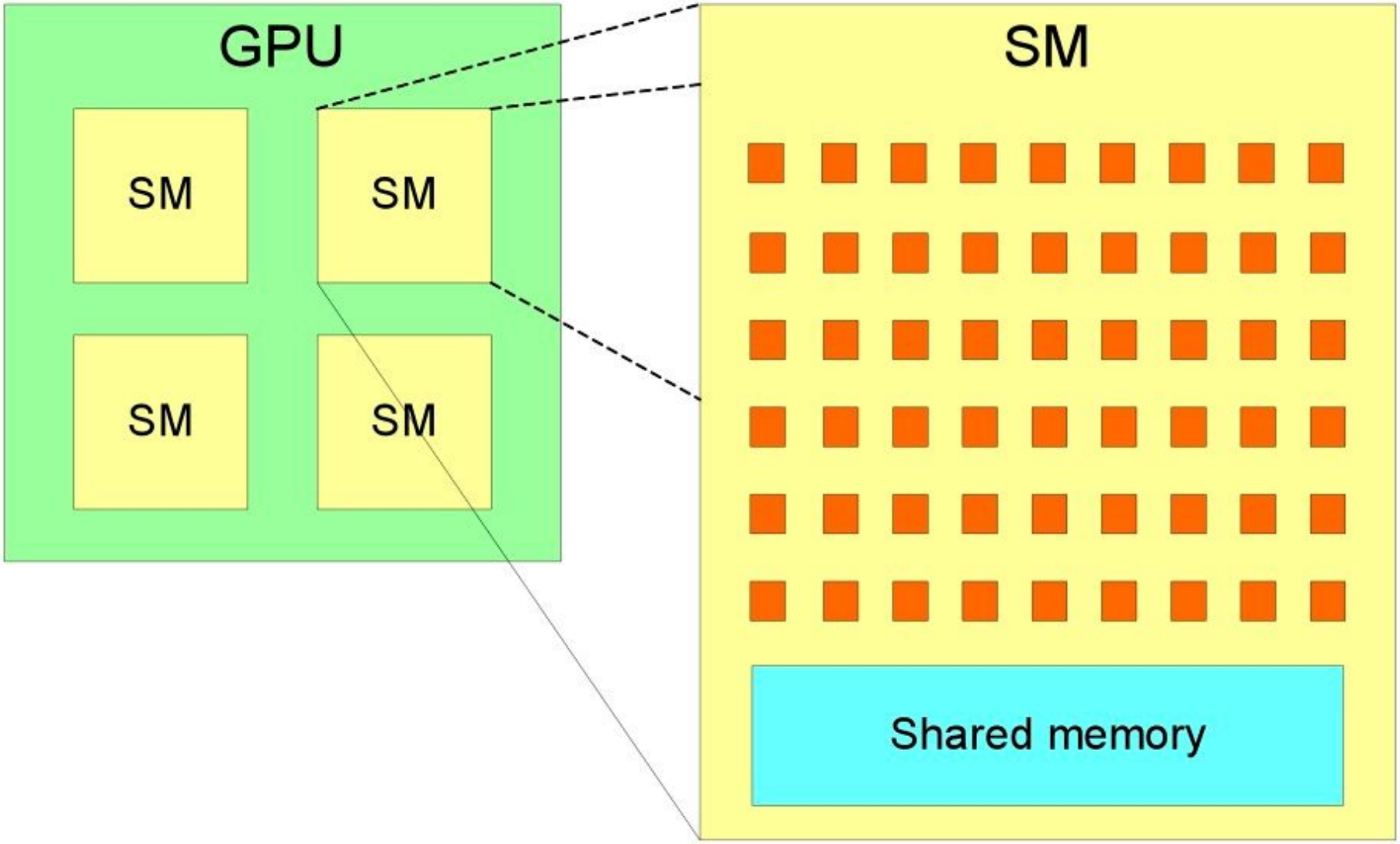


- Design software accordingly- sequential parts on the CPU and number crunching on the GPU

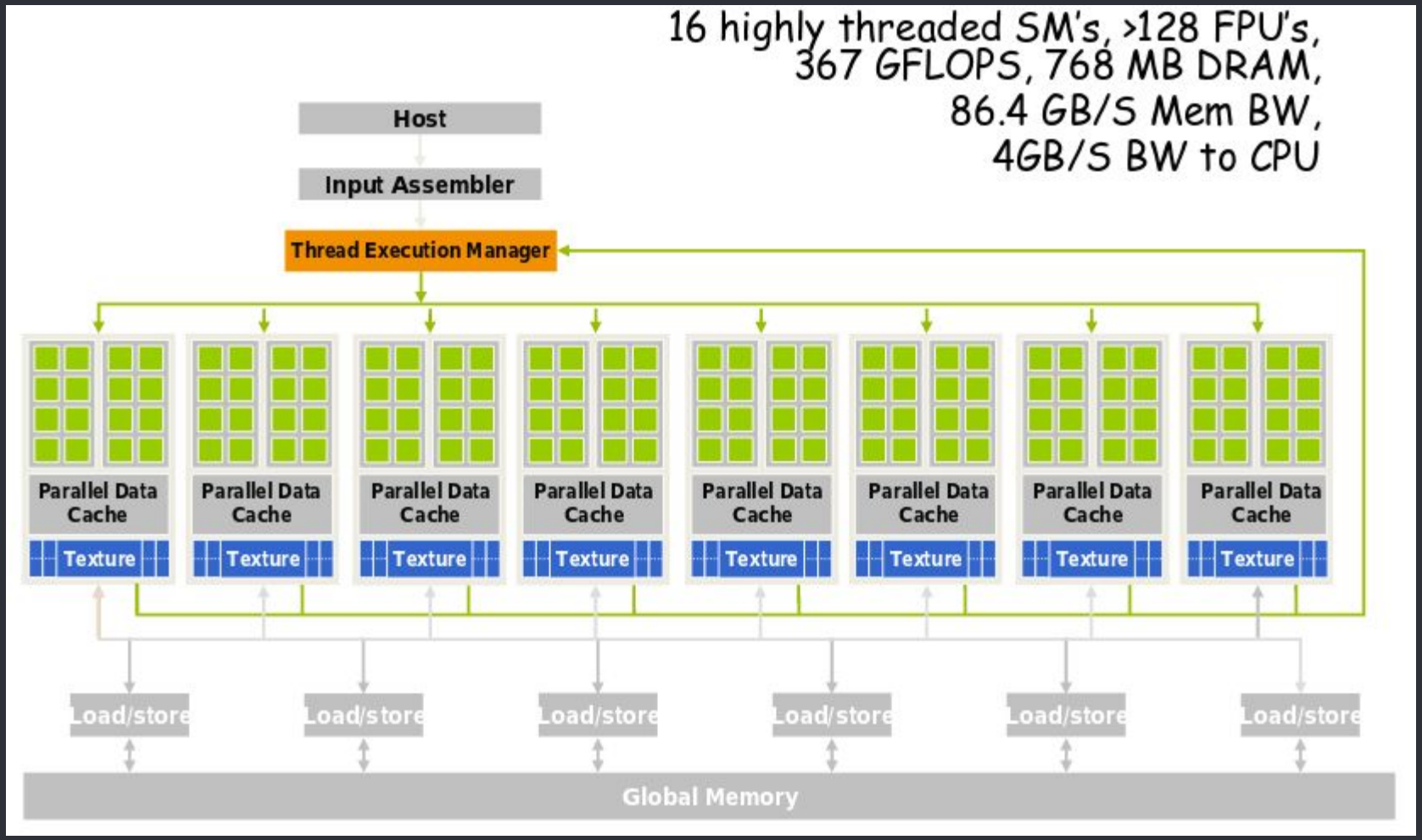
More on GPU Hardware

- CUDA GPUs have a 2-level hierarchy
- Each Streaming Multiprocessor (SM) has multiple Streaming Processor cores

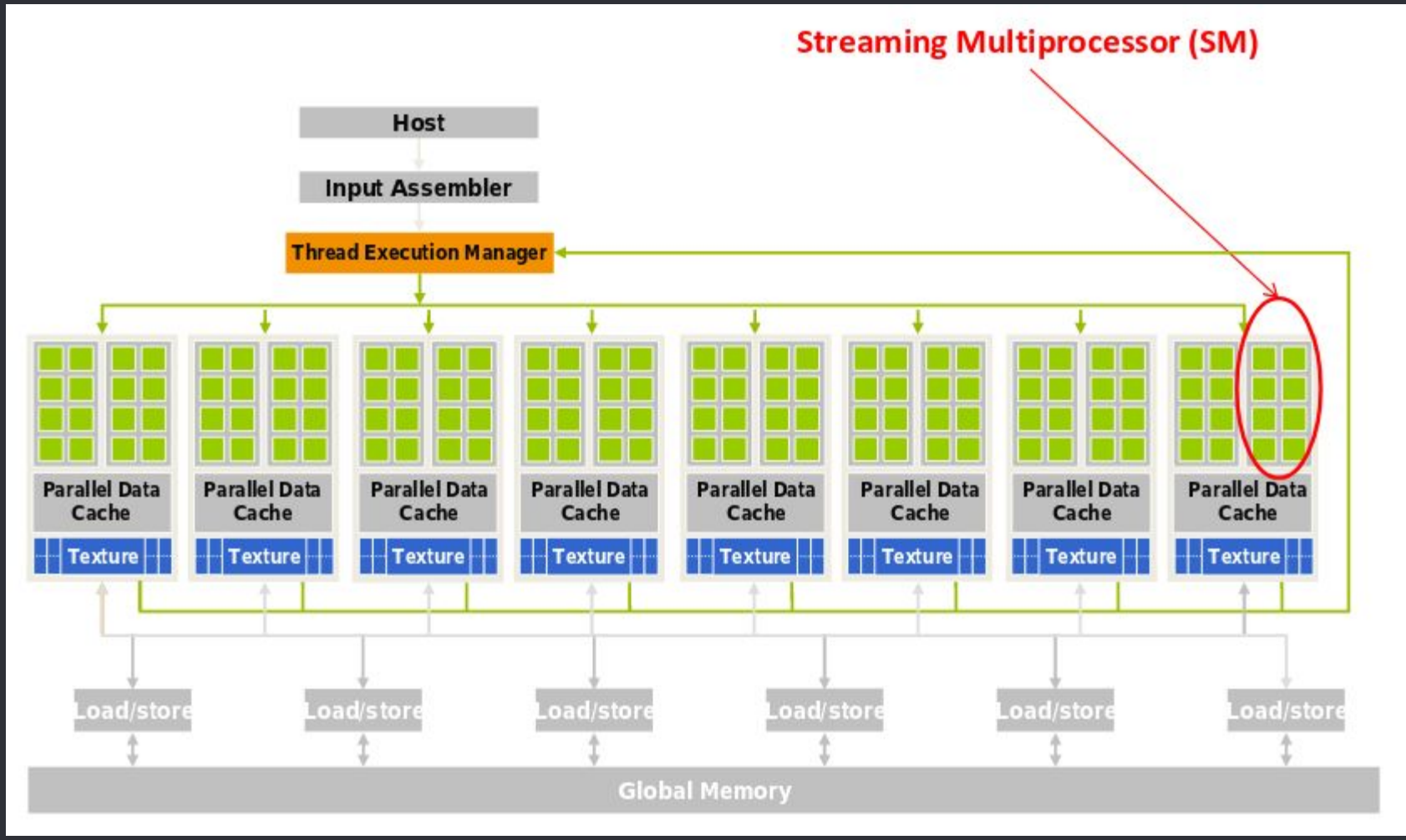
More on GPU Hardware



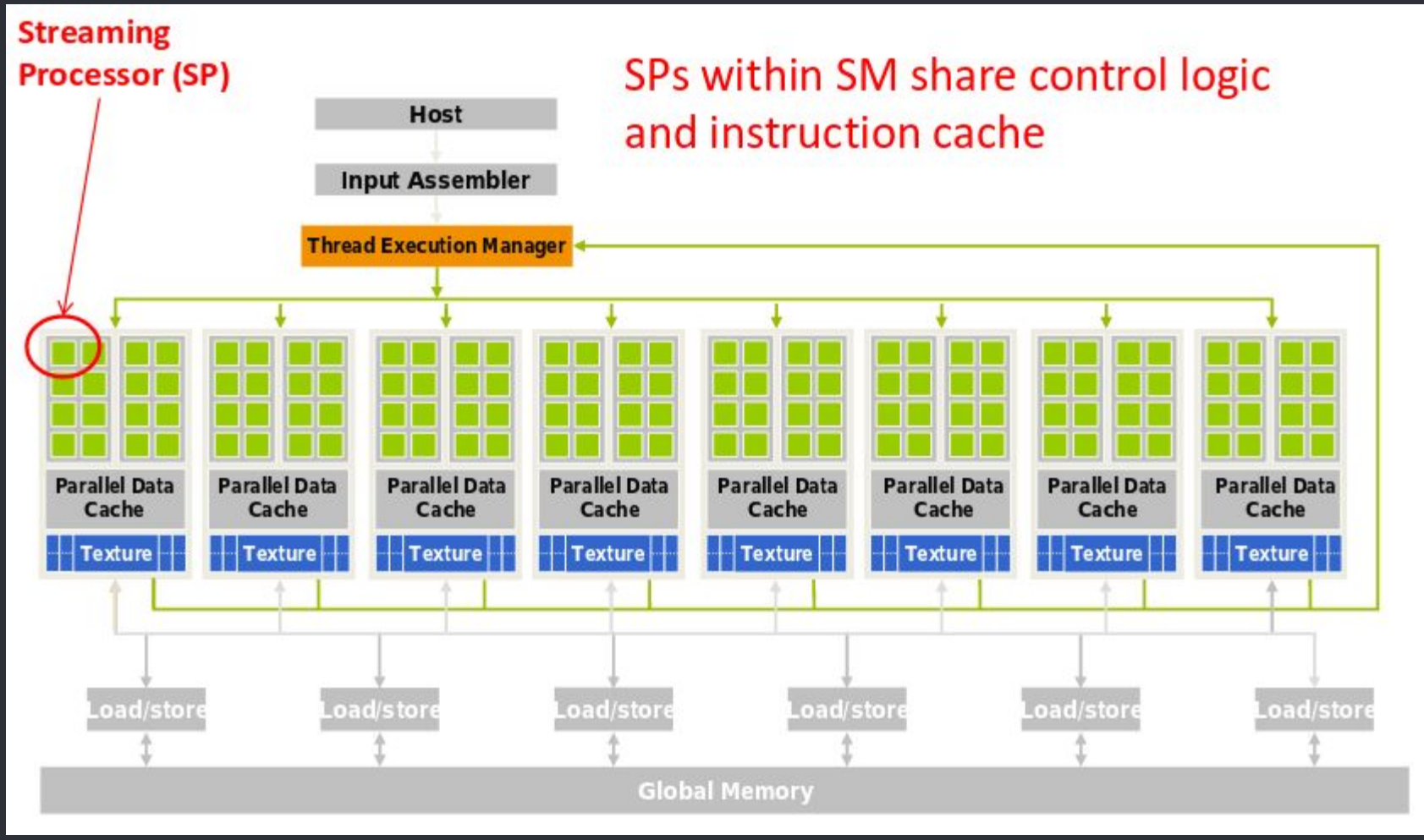
More on GPU Hardware



More on GPU Hardware



More on GPU Hardware



- NVIDIA G80- supporting 5,000 to 12,000 threads at once

“Global Memory”

- Global Memory in previous diagram is also called graphics double data rate (GDDR) DRAM
- GDDR DRAMs differ from CPU DRAMs in that they hold video images and texture information for 3D rendering
- Can function as memory for computing, but with more latency than system memory

How to Take Advantage of GPUs

- Compute Unified Device Architecture (CUDA) (2007)
  - Scalable parallel programming model
  - Minimal extensions to familiar C/C++ environment- remove the need to use OpenGL/Direct3D programming techniques
  - Revolutionized GPU programming for general use
  - Joint CPU/GPU execution

How to Take Advantage of GPUs

- GPU accelerated Libraries/Applications
  - MATLAB, Ansys, etc.
  - GPU mostly abstracted from end user
  - Pros: easy to learn and use
  - Cons: difficult to master (high level of abstraction limits optimization in some cases)

How to Take Advantage of GPUs

- GPU Accelerated Directives (OpenACC)
  - Helps compiler auto generate code for the GPU
  - Very similar to OpenMP
  - Pos: portability, limited understanding of hardware required
  - Cons: limited fine grained control of optimization

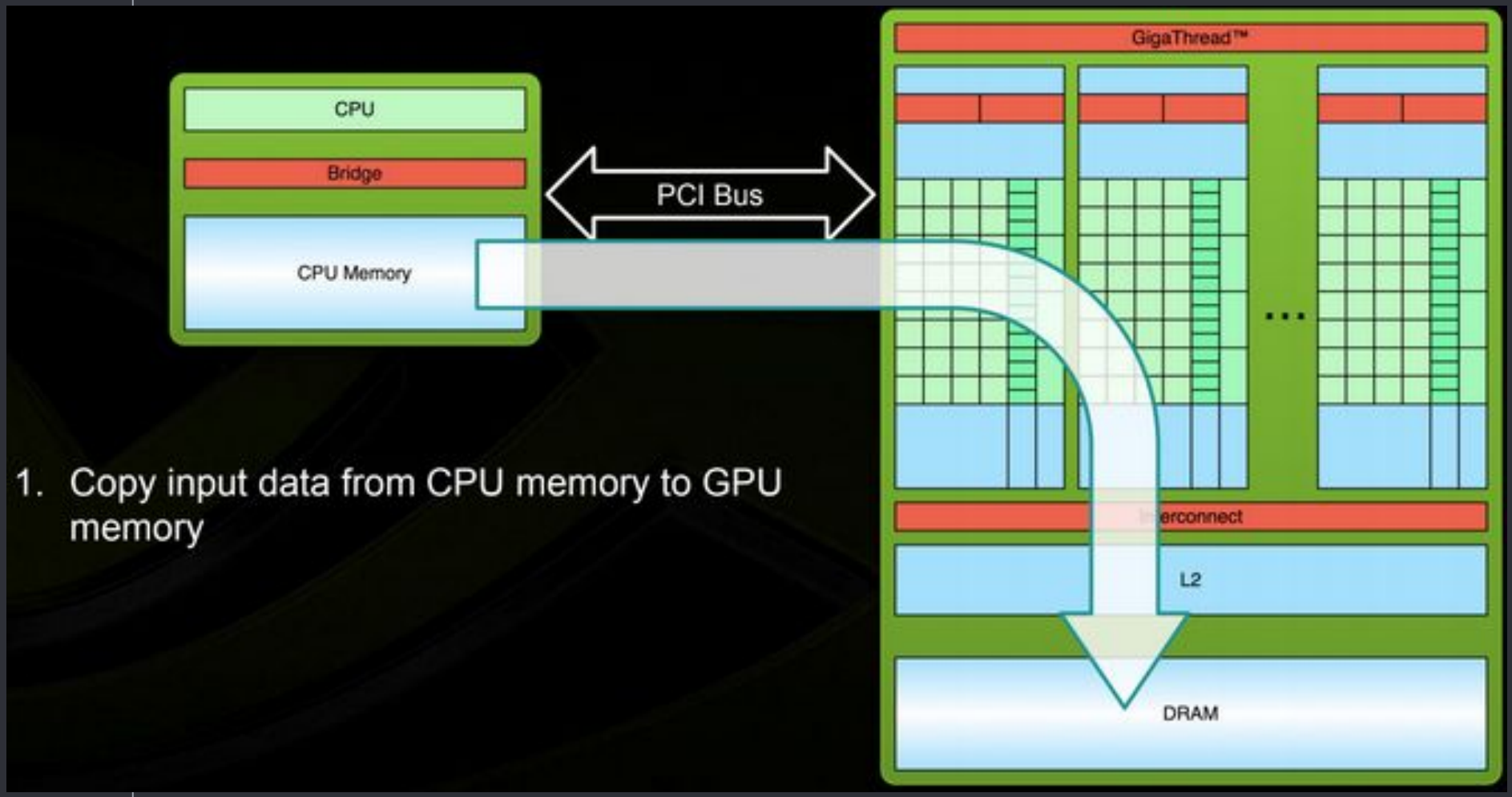
How to Take Advantage of GPUs

- Open Computing Language (OpenCL)
  - Inspired by CUDA but targeted at more general data parallel architectures
  - Pros: cross platform (AMD, Intel, etc)
  - Cons: limited access to cool cutting edge NVIDIA specific functionality

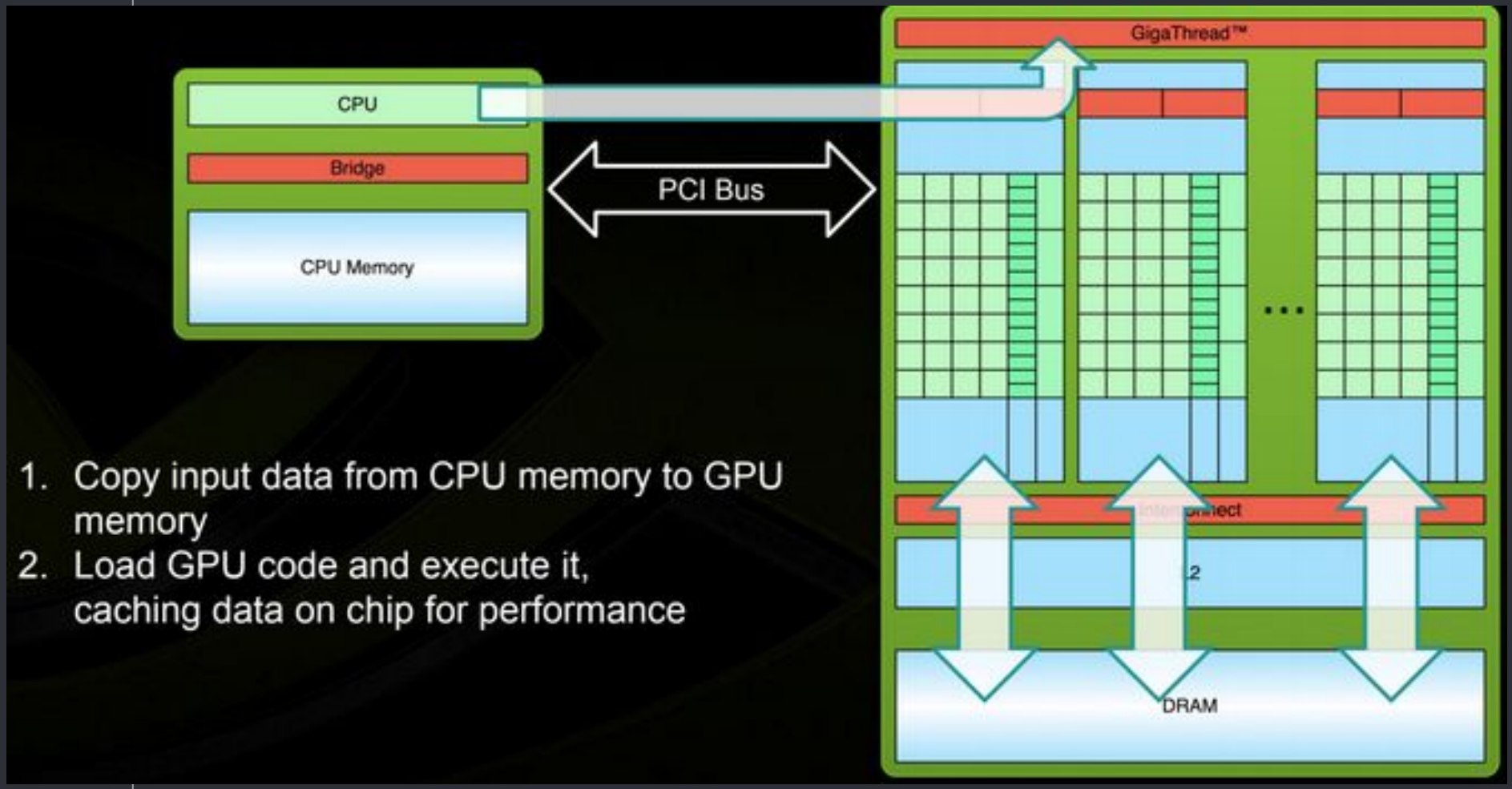
CUDA Terminology

- Host- CPU and its memory
- Device- GPU and its memory
- Kernel- data-parallel function (executed on the device)

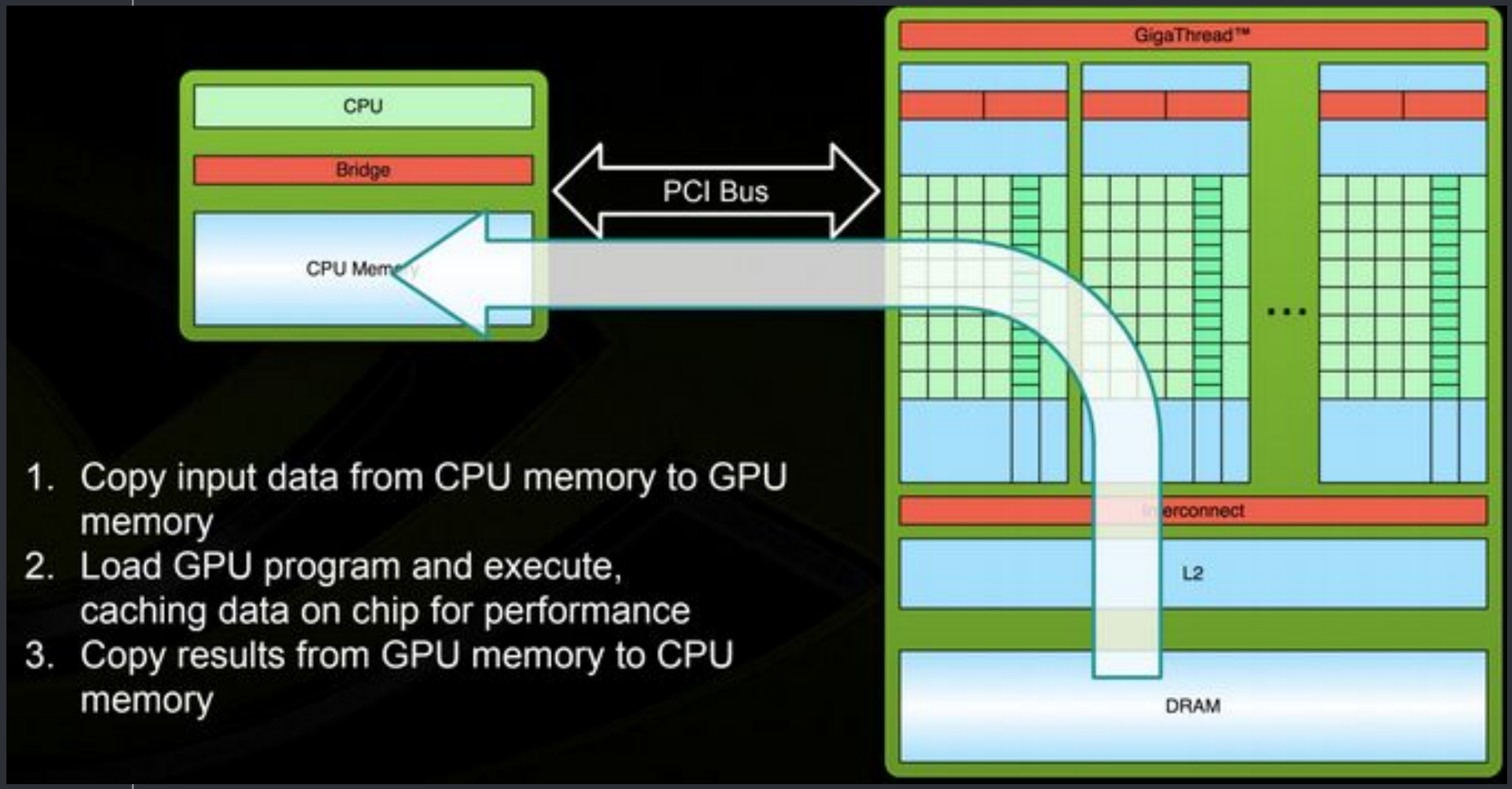
CUDA Execution



CUDA Execution



CUDA Execution



CUDA Execution

```
mykernel<<<1, 1>>>();
```

- Calls CUDA function “mykernel”
- “Kernel launch”
- Marks a call from host code to device code (moving from CPU to GPU)

CUDA Execution

```
__global__ void mykernel(void) {}

int main(void) {
    mykernel<<<1, 1>>>();
    printf("Hello, World!\n");
    return 0;
}
```

CUDA Function Extensions

`__global__`
- Executed on the **device**
- Called from the **host**

`__device__`
- Executed on the **device**
- Called from the **device**

`__host__`
- Executed on the **host**
- Called from the **host**

CUDA Kernel Launch

```
mykernel<<<1, 1>>>();
```

- `nvcc` separates source code into host and device components
- You just have to specify a CUDA kernel with <<< >>>
- Note the parameters (1,1)

Something Slightly More Interesting

```
__global__ void add(int *a, int *b, int*c){
    *c = *a + *b;
}
```

- Simple kernel for adding two integers
- `add()` runs on the device, so `a`, `b`, and `c` must point to device memory
- We need to allocate memory on the GPU

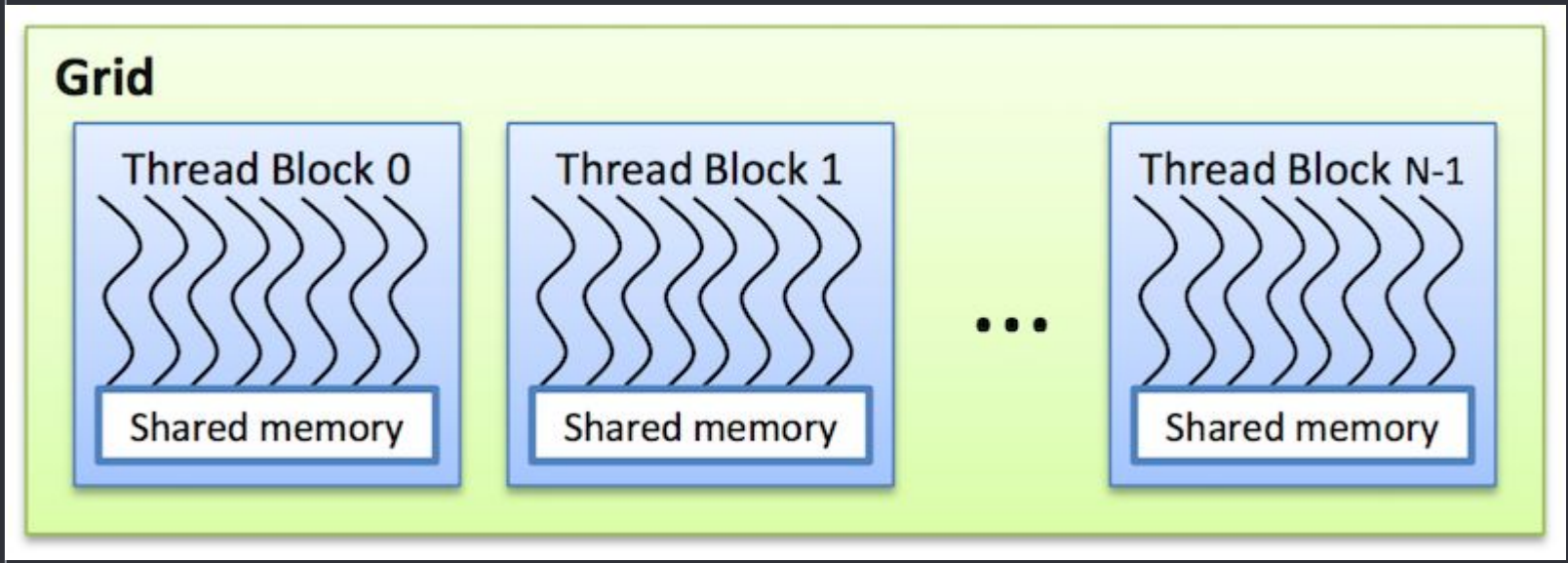
Memory Management

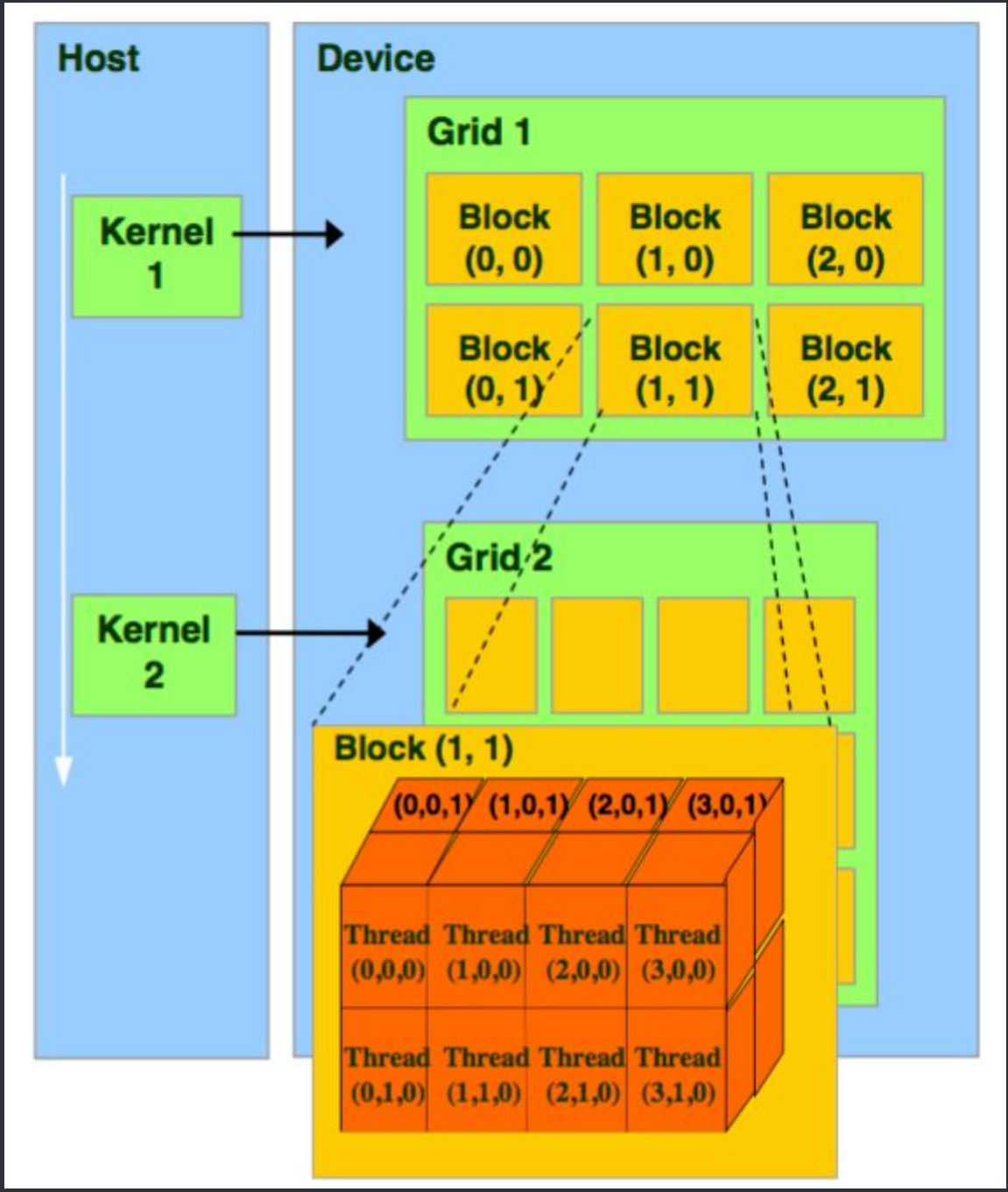
- Host and device memory are separate entities
- A simple CUDA API exists for handling device memory
  - `cudaMalloc(), cudaFree(), cudaMemcpy()`
  - Similar to C equivalents `malloc(), free(), memcpy()`

```c
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main(void) {
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2; b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Running in Parallel

- GPU computing is about massive parallelism!
- CUDA uses blocks of threads organized into a “grid”




Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
(0,0,1)
(1,0,1)
(2,0,1)
(3,0,1)
Thread (0,0,0)
Thread (1,0,0)
Thread (2,0,0)
Thread (3,0,0)
Thread (0,1,0)
Thread (1,1,0)
Thread (2,1,0)
Thread (3,1,0)

High Level Thread Organization

- CUDA provides the size and shape of the block (`blockDim`), and the size and shape of the grid (`gridDim`)
- You can specify what dimensions to make your blocks and grids:
  - `dim3 dimGrid(128, 1, 1);`
  - `dim3 dimBlock(32, 1, 1);`
  - `kernel<<<dimGrid,dimBlock>>>();`
- Grids and blocks can be 1, 2, or 3-dimensional

Thread Organization

```
add<<<1, 256>>>(d_a, d_b, d_c);
```

- Kernel exists with one dimensional block of size 256, which means 256 threads are running (keep in mind CUDA organizes threads into warps of 32 threads)

```
add<<<128, 1>>>(d_a, d_b, d_c);
```

- Kernel exists with a one dimensional grid of size 128, which means 128 blocks are running

Thread Organization

- All threads in a grid execute the same kernel function- so they rely on unique coordinates to distinguish themselves from each other
- Two level hierarchy of unique coordinates
  - `threadIdx` (thread index)
  - `blockIdx` (block index)
  - These are built in variables pre-initialized by the CUDA runtime system (that can be accessed from within kernel functions)

Parallelism in Practice

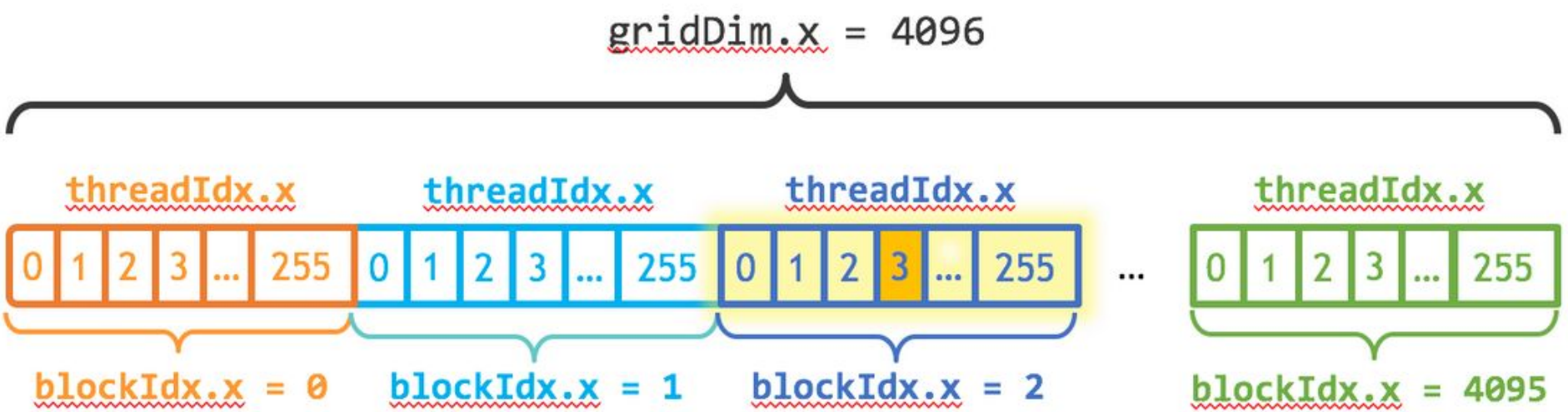
- Refer to block indexes using `blockIdx.x`
- By using the block index to index an array, each block handles a different element of the array

```
__global__ void add(int *a, int *b, int*c) {
    c[blockIdx.x] = a[blockIdx.x] +
                    b[blockIdx.x];
}
```

- Thread indexes use `threadIdx.x`

Using Blocks and Threads

- Indexing arrays is no longer as simple as using block id and thread id




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

Add Vectors With Threads & Blocks

```
__global__
void add(int *a, int *b, int *c) {
    int index = threadIdx.x +
                blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

```
__global__ void vector_add(int *a, int *b, int *c)
{
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        c[index] = a[index] + b[index];
}

#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main()
{
    int *a, *b, *c;
        int *d_a, *d_b, *d_c;
        int size = N * sizeof( int );

        cudaMalloc( (void **) &d_a, size );
        cudaMalloc( (void **) &d_b, size );
        cudaMalloc( (void **) &d_c, size );

        a = (int *)malloc( size );
        b = (int *)malloc( size );
        c = (int *)malloc( size );

        for( int i = 0; i < N; i++ )
        {
                a[i] = b[i] = i;
                c[i] = 0;
        }

        cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );
        cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice );

        add<<< (N + THREADS_PER_BLOCK) / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( d_a, d_b, d_c );

        cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );

        free(a); free(b); free(c);
        cudaFree( d_a ); cudaFree( d_b ); cudaFree( d_c );

        return 0;
```

Sharing Data between Threads

- Threads can share memory within a block
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks
- Use `__synchthreads()` to prevent hazards when dealing with shared data

Coordinating Host & Device

- Kernel launches are asynchronous, control returns to the CPU immediately
- CPU needs to synchronize before consuming the results
  - `cudaDeviceSynchronize()` blocks the CPU until all preceding CUDA calls have completed

Additional Resources

- Victor Eijkhout, Introduction to High-Performance Scientific Computing http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html
- NVIDIA developer portal https://developer.nvidia.com/
- CUDA documentation https://docs.nvidia.com/cuda/
- OpenCL resources https://www.khronos.org/opencl/
- Stanford University Course on Parallel Programming https://github.com/jaredhoberock/stanford-cs193g-sp2010
- Oxford University Course on CUDA http://people.maths.ox.ac.uk/~gilesm/cuda/
- NVIDIA Parallel Programming Course on Udacity https://developer.nvidia.com/udacity-cs344-intro-parallel-programming
- NVIDIA AI & Deep Learning (this is really cool) https://developer.nvidia.com/deep-learning