



GPU Parallel Programming: CUDA

National Tsing Hua University
2025, Fall Semester

Outline

- Programming Model
- CUDA Language
- Example Code Study
- CPU & GPU Synchronization
- Multi-GPU

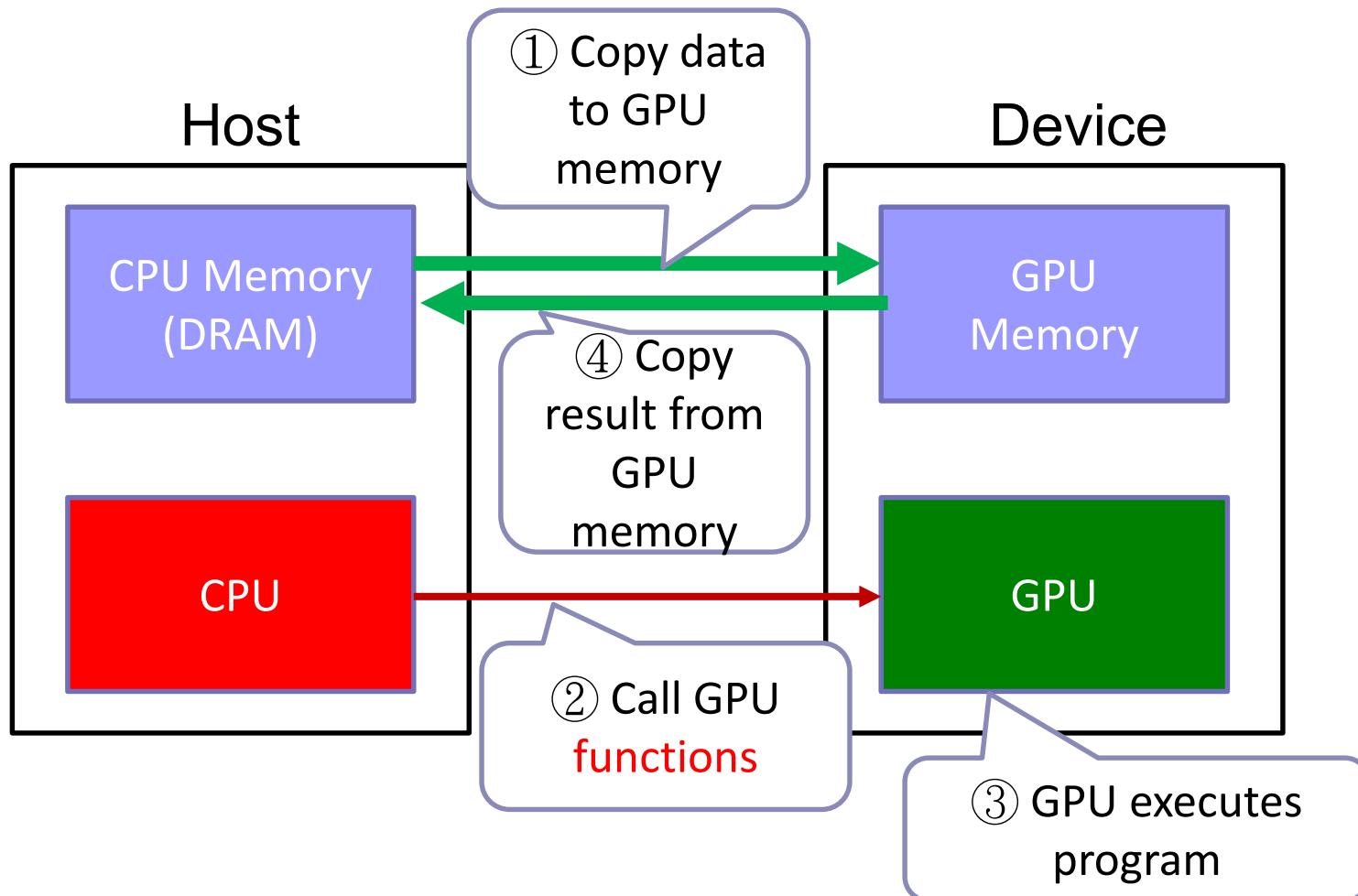
What is CUDA?

■ CUDA: Compute Unified Device Architecture

- CUDA is a **compiler** and **toolkit** for programming NVIDIA GPUs
- Enable heterogeneous computing and horsepower of GPUs
- CUDA API extends the C/C++ programming language
- Express SIMD parallelism
- Give a high level abstraction from hardware

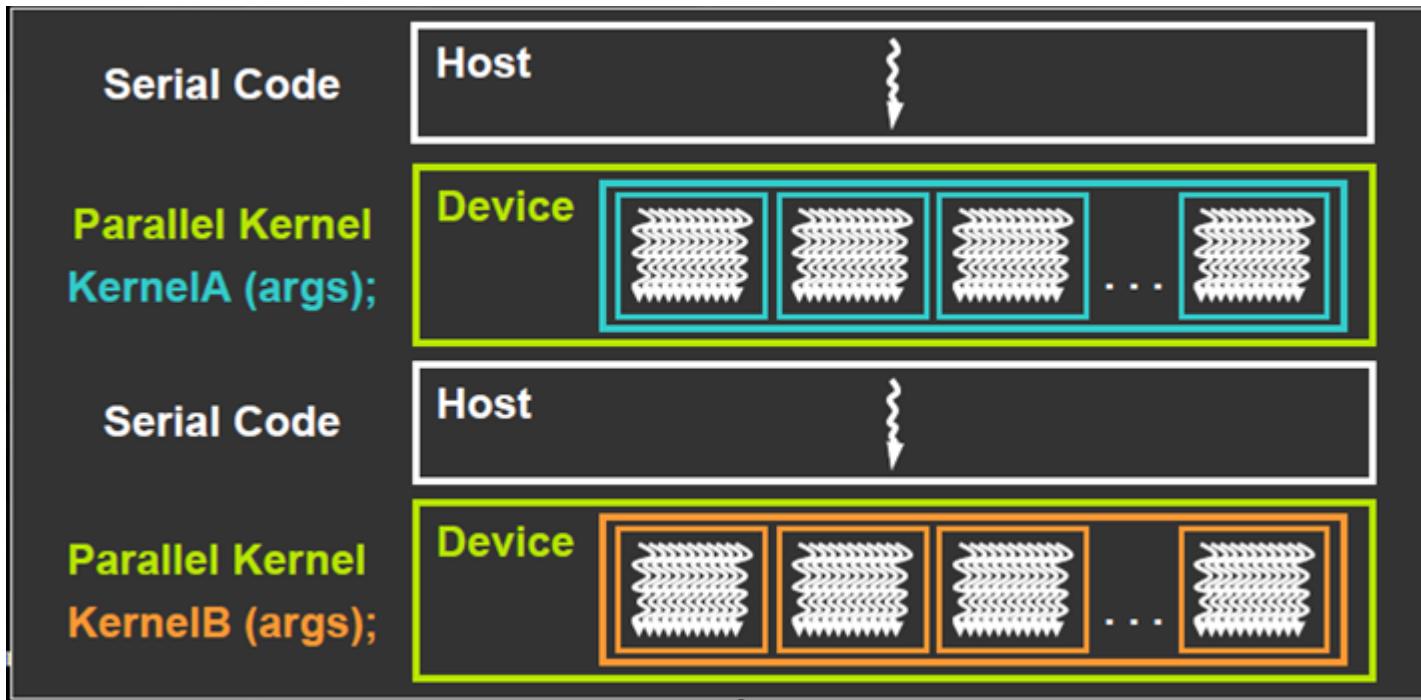
CUDA SDK Version	Compute Capability	Architecture
6.5	1.X	Tesla
7.5	2.0-5.x	Fermi, Kepler, Maxwell
8.0	2.0-6.x	Fermi, Kepler, Maxwell, Pascal
9.0	3.0-7.x	Kepler, Maxwell, Pascal, Volta

CUDA program flow



CUDA Programming Model

- CUDA = serial program with parallel kernels, all in C
 - Serial C code executes in a host thread (i.e. CPU thread)
 - Parallel kernel C code executes in many devices threads across multiple processing elements (i.e. GPU threads)



CUDA program framework

GPU code
(parallel)

CPU code
(serial or
parallel if
p-thread/
OpenMP/T
BB/MPI is
used.)

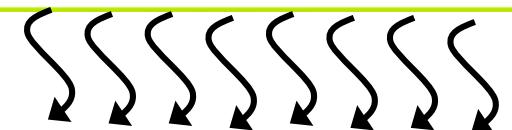
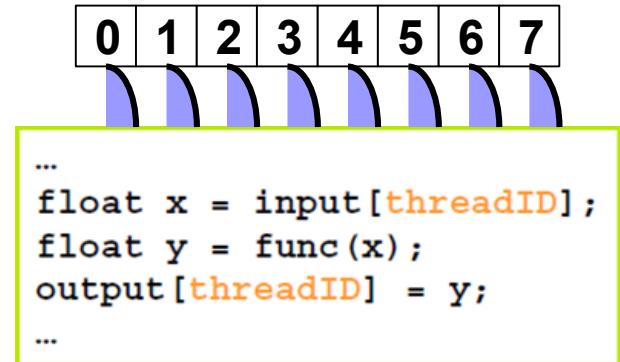
```
#include <cuda_runtime.h>

__global__ void my_kernel(...) {
    ...
}

int main() {
    ...
    cudaMalloc(...)
    cudaMemcpy(...)
    ...
    my_kernel<<<nblock,blocksize>>>(...)
    ...
    cudaMemcpy(...)
    ...
}
```

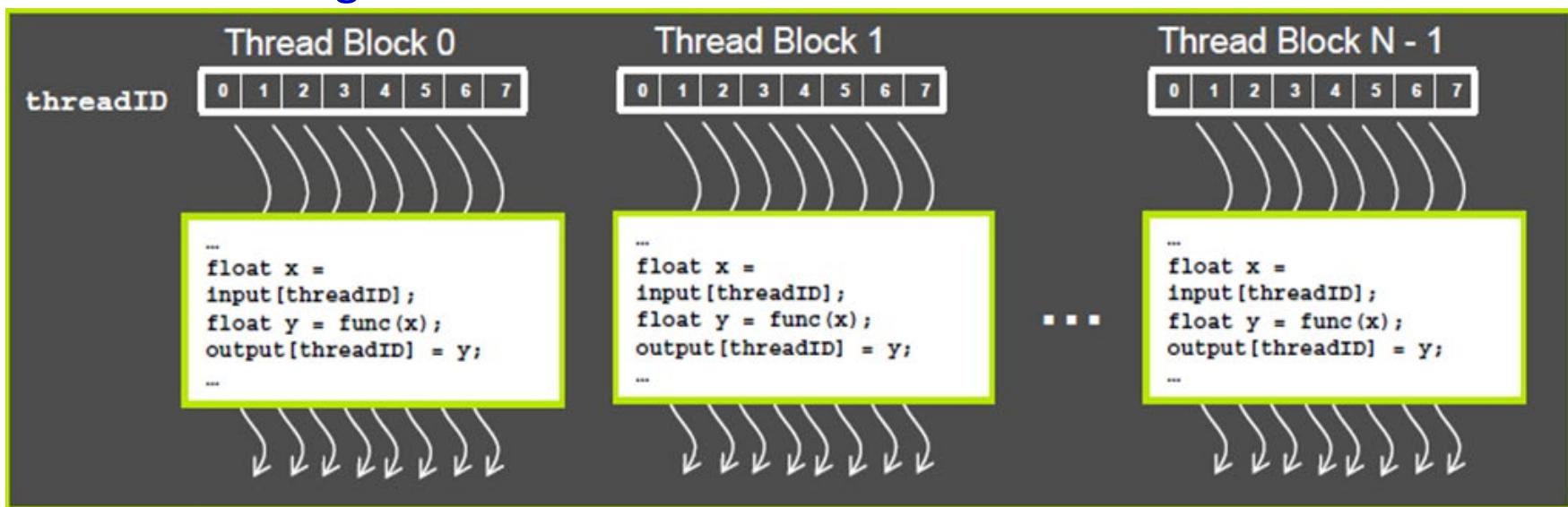
Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
 - Each thread executes the same code
 - ... on the different data based on its threadID
- CUDA thread might be
 - Physical threads
 - ◆ As on NVIDIA GPUs
 - ◆ GPU thread creation and context switching are essentially free
 - Or virtual threads
 - ◆ E.g. 1 CPU core might execute multiple CUDA threads



Hierarchy of Concurrent Threads

- Threads are grouped into thread blocks
 - Kernel = grid of thread blocks

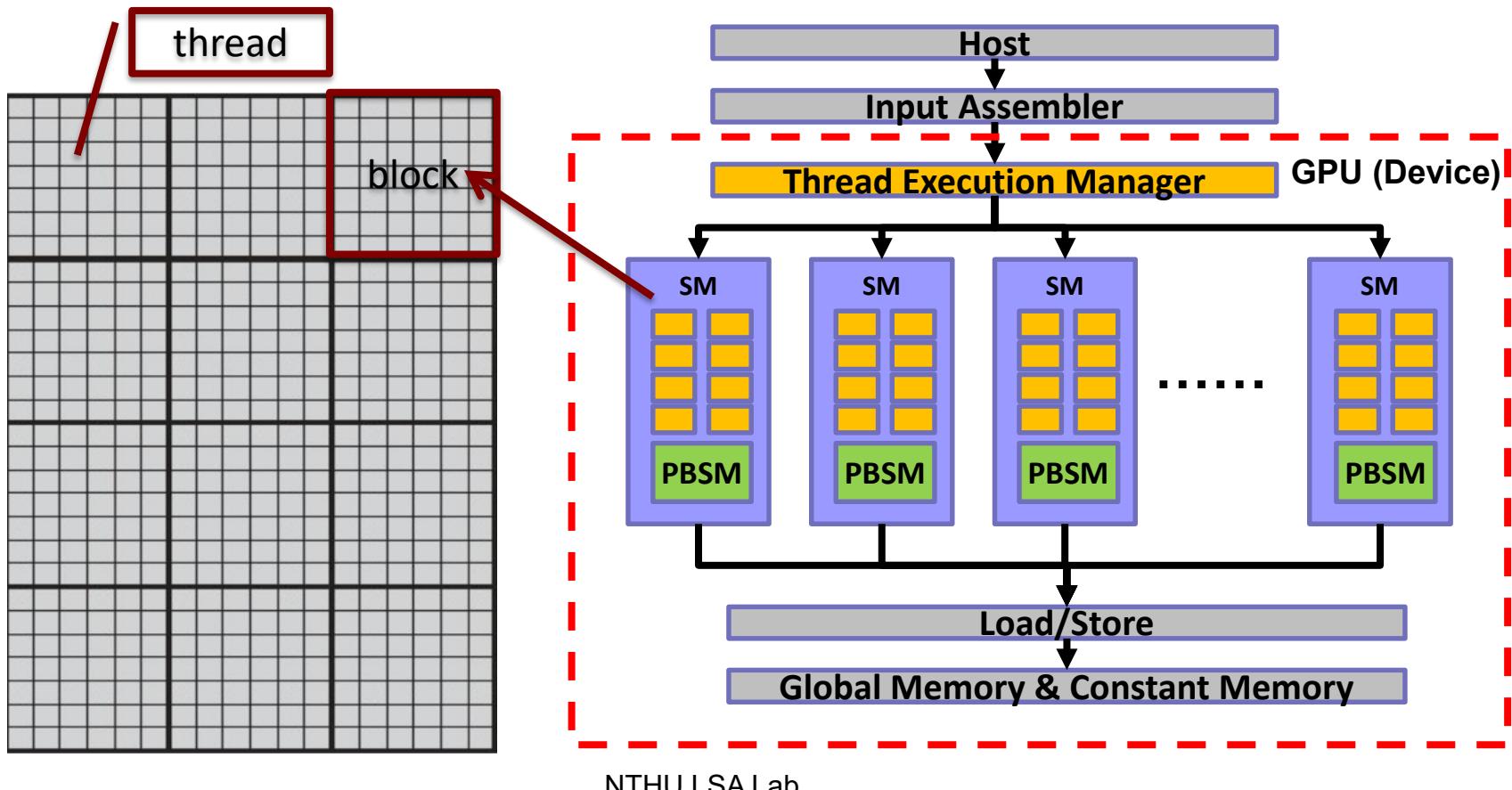


- By definition, threads in the same block may synchronize with barriers, but not between blocks

```
scratch[threadID] = begin[threadID];  
__syncthreads();  
int left = scratch[threadID - 1];
```

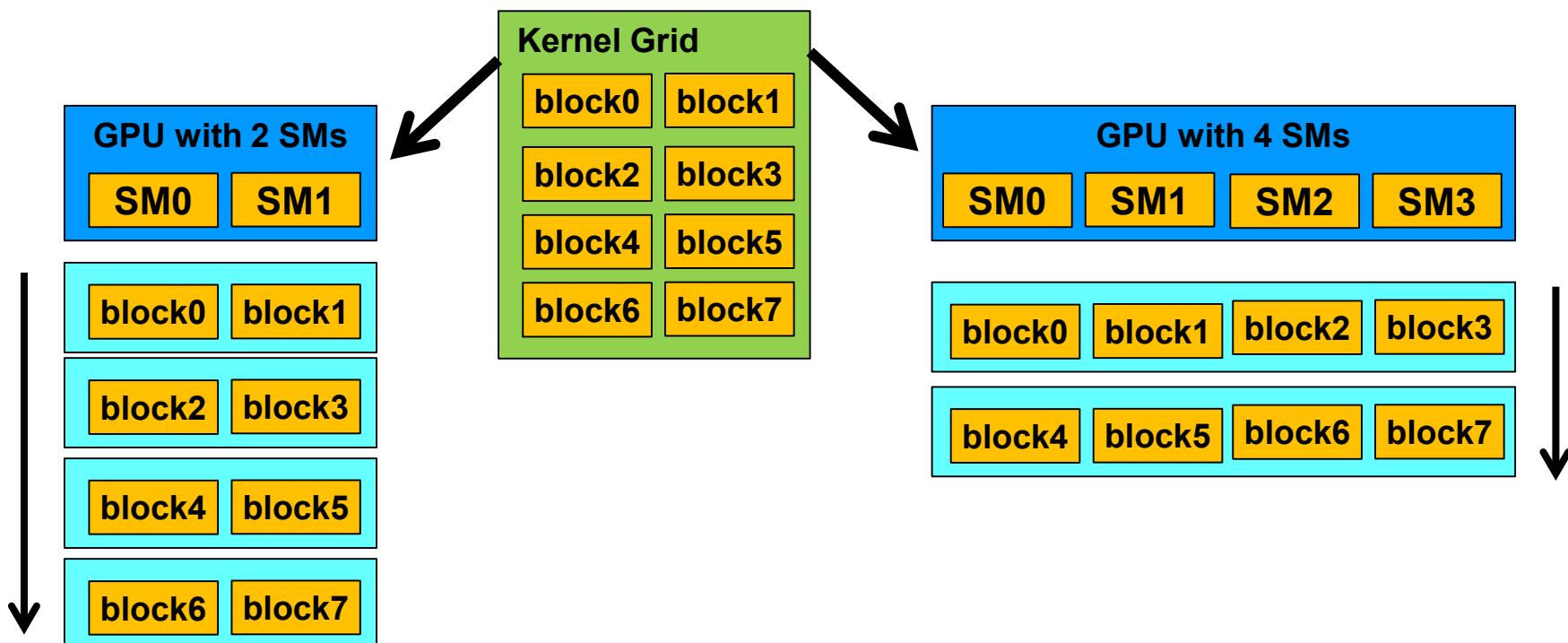
Software Mapping

- Software: grid → blocks → threads
- Hardware: GPU(device) → SM(multicore processor) → core



Block Level Scheduling

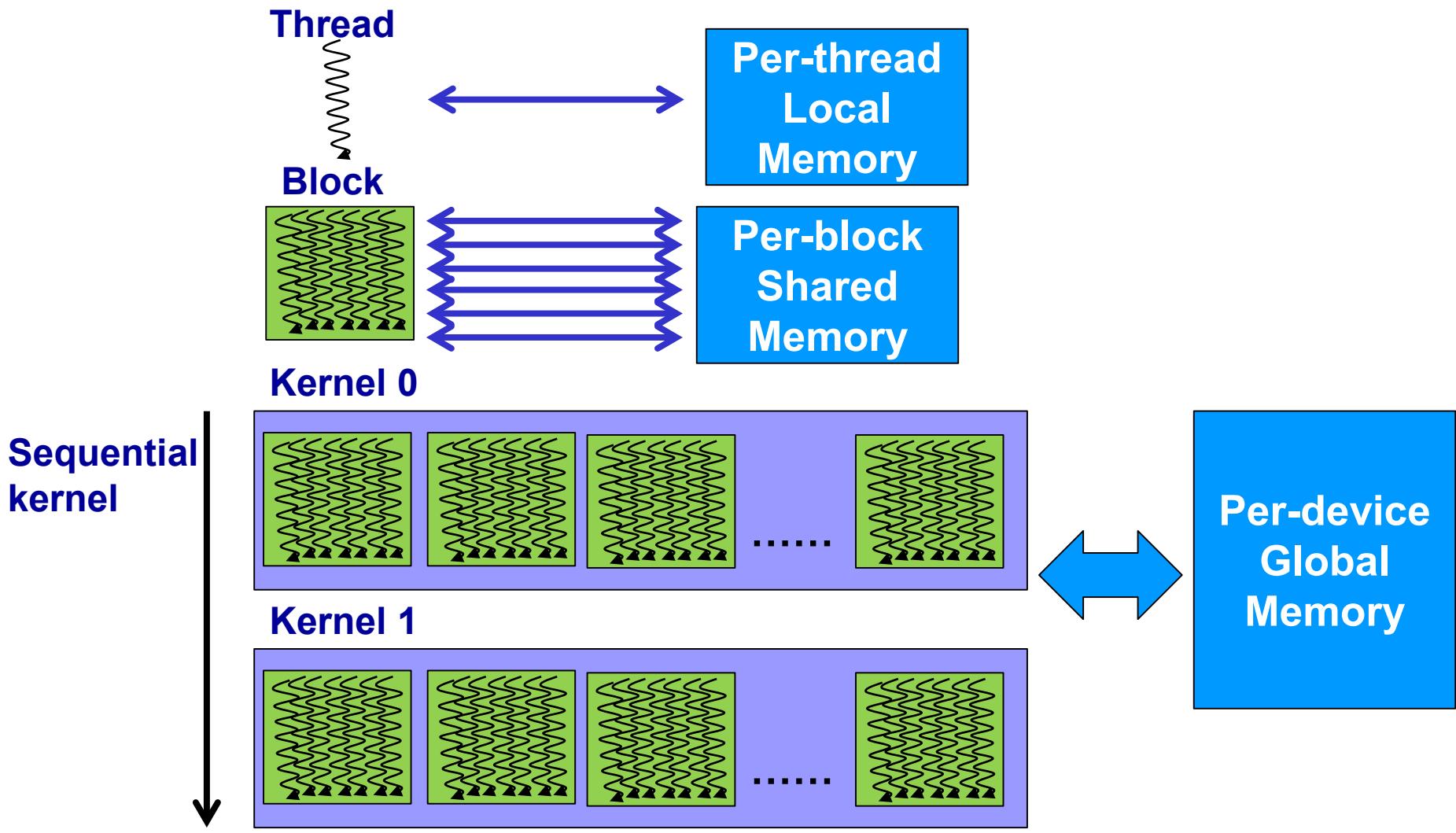
- Blocks are independent to each other to give scalability
 - A kernel scales across any number of parallel cores by scheduling blocks to SMs



Thread group limits

- **GTX1080 CUDA Capability 6.1**
 - Use `deviceQuery.cpp` to find out your limits
- Total number of threads per kernel = threads per block * number of blocks
 - Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
 - Max dimension size of a grid size (x,y,z): ($2^{31}-1$, $2^{16}-1$, $2^{16}-1$)
- Maximum execution concurrency
 - Maximum number of resident grids per device (Concurrent Kernel Execution): 32
 - Maximum number of threads per multiprocessor: 2048
 - Maximum number of threads per block: 1024

Memory Hierarchy



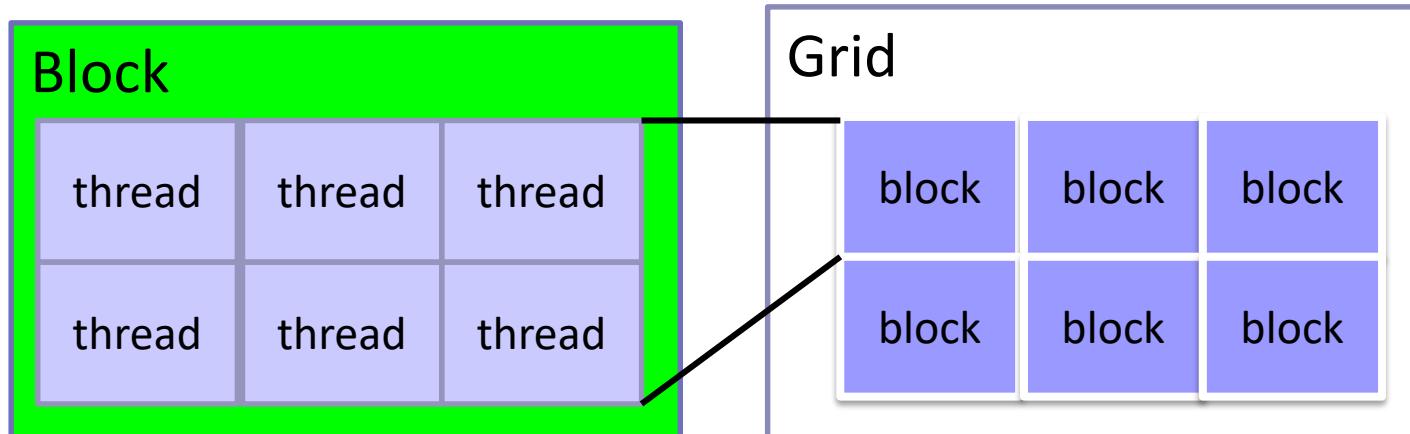
Memory size limits

■ GTX1080 CUDA Capability6.1

- Use `deviceQuery.cpp` to find out your limits
- Total amount of global memory: 11171 MBytes
- Total amount of constant memory: 64 KBytes
- Total amount of shared memory per block: 48 KBytes
- Total number of registers available per block: 65536

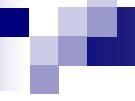
CUDA Programming Terminology

- Host : CPU
- Device : GPU
- Kernel : functions executed on GPU
- Thread : the basic execution unit
- Block : a group of threads
- Grid : a group of blocks



Quiz

- Can a kernel run across multiple SM processors?
- Does the kernel below have to run on 10 different SM processors?
 - `my_kernel<<< 10, 10 >>>();`
- Can the threads from a same block run across multiple SM processors?
- What is the difference between the two kernels below?
 1. `my_kernel<<< 1, 100 >>>(A);`
 2. `my_kernel<<< 100, 1 >>>(A);`
- Why shared block memory can only be accessed by the threads in the same blocks?
- Why we have to call `__syncthreads()` within a block if there are data dependency between statements
- Why `__syncthreads()` is not supported across blocks?



Outline

- Programming Model
- CUDA Language
- Example Code Study
- CPU & GPU Synchronization
- Multi-GPU

CUDA Language

Philosophy: provide minimal set of extensions necessary

■ Kernel launch

```
kernelFunc<<< nB, nT, nS, Sid >>>(...); // nS and Sid are optional
```

- nB : number of blocks per grid (grid size)
- nT : number of threads per block (block size)
- nS : shared memory size (in bytes)
- Sid : stream ID, default is 0

■ Build-in device variables

- threadIdx; blockIdx; blockDim; gridDim

■ Intrinsic functions that expose operations in kernel code

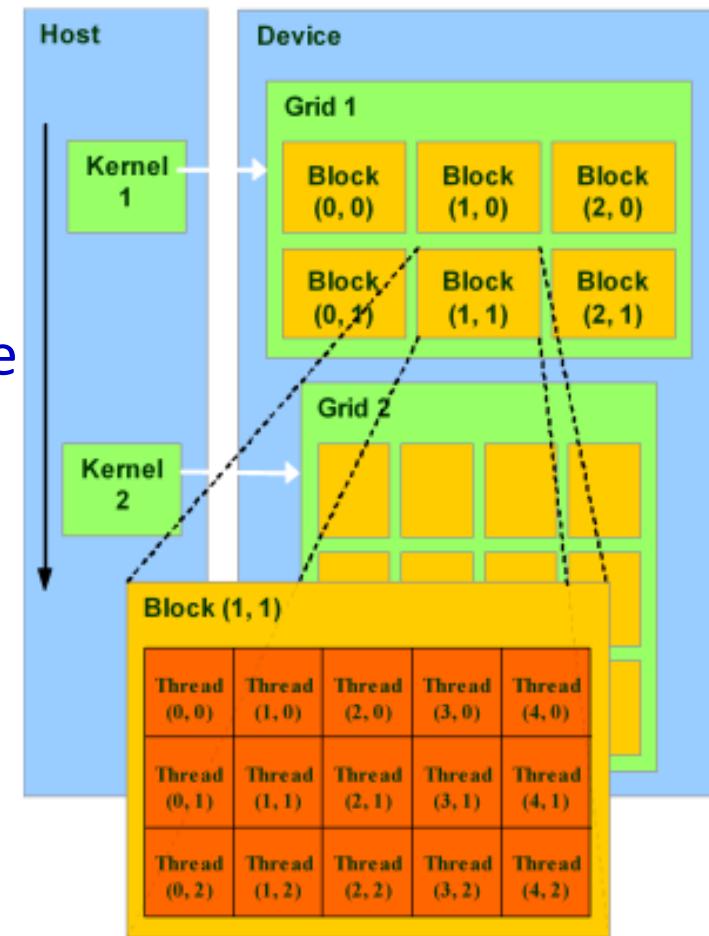
```
__syncthreads();
```

■ Declaration specifier to indicate where things live

```
__global__ void KernelFunc(...); // kernel function, run on device
__device__ void GlobalVar;      // variable in device memory
__shared__ void SharedVar;     // variable in per-block shared memory
```

Thread and Block IDs

- Build-in device variables
 - `threadIdx; blockIdx; blockDim; gridDim`
- The index of threads and blocks can be denoted by a 3 dimensional struct
 - `dim3` defined in `vector_types.h`
struct dim3 { x; y; z; };
- Example:
 - `dim3 grid(3, 2);`
 - `dim3 blk(5, 3);`
 - `my_kernel<<< grid, blk >>>();`
- Each thread can be uniquely identified by a tuple of index (x,y) or (x,y,z)



Q: When will we use multi-dimensional index?

Quiz

- How to index a 100 elements of an array under the following kernel launch setting?

```
// Kernel definition
__global__ void VecAdd(float* A)
{
    int i = [REDACTED]
    A[i] = A[i] + 1;
}
```

1. my_kernel<<< 1, 100 >>>(A) ;
2. my_kernel<<< 100, 1 >>>(A) ;
3. my_kernel<<< 10, 10 >>>(A) ;
4. size=10; dim3 blk(size, size);
 my_kernel<<< 1, blk >>>(A, size) ;

Quiz

- How to index a 100 elements of an array under the following kernel launch setting?

```
// Kernel definition
__global__ void VecAdd(float* A)
{
    int i = [REDACTED]
    A[i] = A[i] + 1;
}
```

1. my_kernel<<< 1, 100 >>>(A); → int i = threadIdx;
2. my_kernel<<< 100, 1 >>>(A); → int i = blockIdx;
3. my_kernel<<< 10, 10 >>>(A);
 → int i=blockIdx*blockDim+threadIdx;
4. size=10; dim3 blk(size, size);
 my_kernel<<< 1, blk >>>(A, size);
 → int i = threadIdx.x * blockDim.x + threadIdx.y;

Function Qualifiers

Function qualifiers	limitations
<code>__device__</code> function	Executed on the device Callable from the device only
<code>__global__</code> function	Executed on the device Callable from the host only (must have <code>void</code> return type!)
<code>__host__</code> function	Executed on the host Callable from the host only
Functions without qualifiers	Compiled for the host only
<code>__host__ __device__</code> function	Compiled for both the host and the device

Variable Type Qualifiers

Variable qualifiers	limitations
<code>__device__ var</code>	<ul style="list-style-type: none">• Resides in device's global memory space• Has the lifetime of an application
<code>__constant__ var</code>	<ul style="list-style-type: none">• Is accessible from all the threads within the grid and from the host through the runtime library• Resides in device's constant memory space
<code>__shared__ var</code>	<ul style="list-style-type: none">• Resides in the shared memory space of a thread block• Has the lifetime of the block• Is only accessible from all the threads within the block

Device memory operations

■ Three functions:

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Similar to the C's `malloc()`, `free()`, `memcpy()`

1. `cudaMalloc(void **devPtr, size_t size)`

- `devPtr`: return the address of the allocated device memory
- `size`: the allocated memory size (**bytes**)

2. `cudaFree (void *devPtr)`

3. `cudaMemcpy(void *dst, const void *src,
size_t count, enum cudaMemcpyKind kind)`

- `count`: size in **bytes** to copy

cudaMemcpyKind

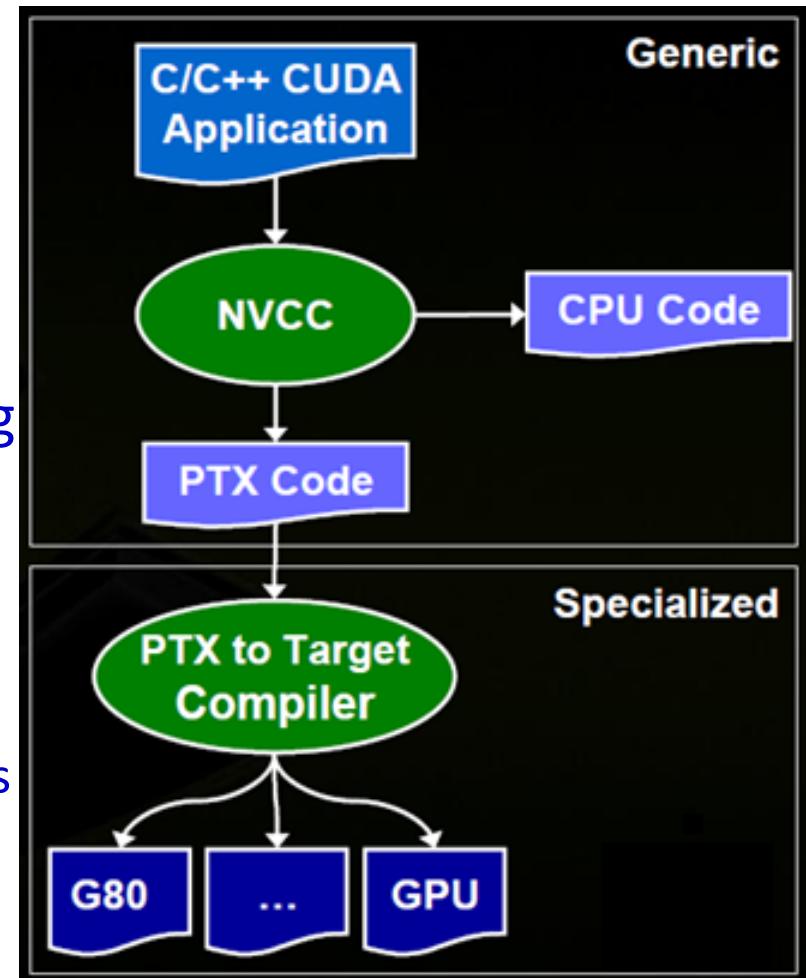
- one of the following four values

cudaMemcpyKind	Meaning	dst	src
cudaMemcpyHostToHost	Host → Host	host	host
cudaMemcpyHostToDevice	Host → Device	device	host
cudaMemcpyDeviceToHost	Device → Host	host	device
cudaMemcpyDeviceToDevice	Device → Device	device	device

host to host has the same effect as memcpy()

Program Compilation

- Any source file containing CUDA language must be compiled with NVCC
 - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
 - Virtual ISA
 - ◆ PTX: Parallel Threads eXecutions
 - Device-specific binary object



Outline

- Programming Model
- CUDA Language
- **Example Code Study**
- CPU & GPU Synchronization
- Multi-GPU

Example 1: Hello World!

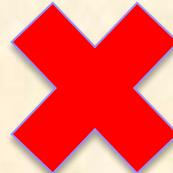
```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

■ Two new syntactic elements...

1. __global__ indicates a function that runs on the device and is called from host code
2. mykernel<<<1,1>>>();
Triple angle brackets mark a call from host code to device code, which is called a “kernel launch”.

Example 2: add 2 numbers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}  
int main(void) {  
    int ha=1,hb=2,hc;  
    add<<<1,1>>>(&ha, &hb, &hc);  
    printf("c=%d\n", hc);  
    return 0;  
}
```



- This does not work!!
- `int ha, hb, hc` are in the host memory (DRAM), which cannot be used by device (GPU).
- We need to allocate variables in “device memory”.

The correct main ()

```
int main(void) {  
    int a=1, b=2, c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **) &d_a, sizeof(int));  
    cudaMalloc((void **) &d_b, sizeof(int));  
    cudaMalloc((void **) &d_c, sizeof(int));  
    // Copy inputs to device  
    cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);  
    // Launch add() kernel on GPU  
    add<<<1,1>>>(d_a, d_b, d_c);  
    // Copy result back to host  
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);  
    // Cleanup  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

Example 3: add 2 vectors

- Let's first look at the sequential code!

```
// function definition
void VecAdd(int N, float* A, float* B, float* C)
{
    for(int i = 0; i<N; i++)
        C[i] = A[i] + B[i];
}

int main()
{
    ...
    VecAdd (N, Ah, Bh, Ch);
    ...
}
```

Parallel CUDA code

- Use `blockIdx.x` as the index of the arrays
 - Each thread processes 1 addition, for the elements indexed at `blockIdx.x`.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{ ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(Ah, Bh, Ch); ...
}
```

Alternative implementation

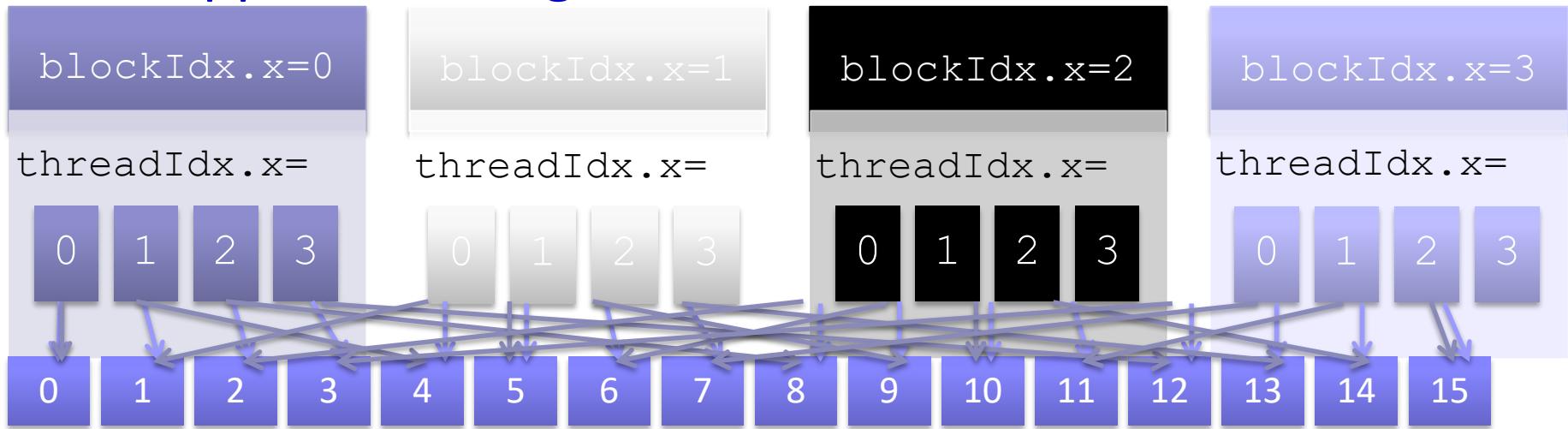
- Using parallel thread instead

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}  
int main(void) {  
    int a[N], b[N], c[N];  
    int *d_a, *d_b, *d_c;  
    ...  
    add<<< N, 1 >>>(d_a, d_b, d_c);  
    ...  
}
```

- N blocks and each block has 1 thread.
- Which one is better?
 - Threads in the same block can communicate, synchronize with others, but the number of threads per block is limited.

3rd implementation

- Using multiple threads and multiple blocks
- Suppose N=16, grid size = 4, and block size = 4



- How to index 16 elements of an array?
 - Method 1: $\text{index} = \text{blockIdx.x} * 4 + \text{threadIdx.x}$
 - Method 2: $\text{index} = \text{threadIdx.x} * 4 + \text{blockIdx.x}$
- Which one is better?

The general case

- Use the built-in variable `blockDim.x` for threads per block.

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}  
int main(void) {  
    int a[N], b[N], c[N];  
    int *d_a, *d_b, *d_c;  
    ...  
    add<<< N/BS, BS>>>(d_a, d_b, d_c);  
    ...  
}
```

What if N is not a multiple of BS?

- BS is block size (number of threads per block)

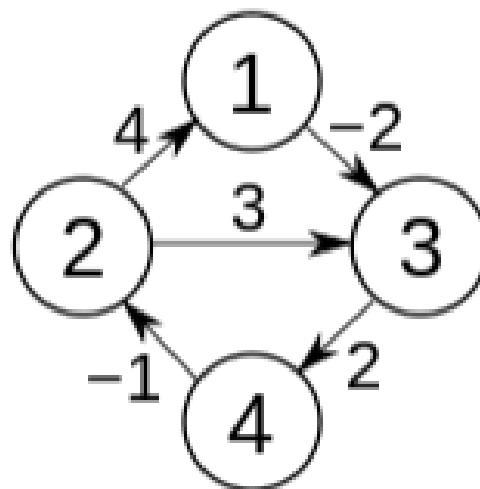
A even more general case

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
int main(void) {
    int a[N], b[N], c[N];
    int *d_a, *d_b, *d_c;
    ...
    add<<< (N+BS-1)/BS, BS>>>(d_a, d_b, d_c, N);
    ...
}
```

- The kernel function can have branches, but with a price to pay...

Example4: APSP

- Given a weighted directed graph $G(V, E, W)$, where $|V| = n$, $|W| = m$, and $W > 0$, find the shortest path of all pairs of vertices (v_i, v_j) .
- Example:



0	INF	-2	INF
4	0	3	INF
INF	INF	0	2
INF	-1	INF	0

Initial weight

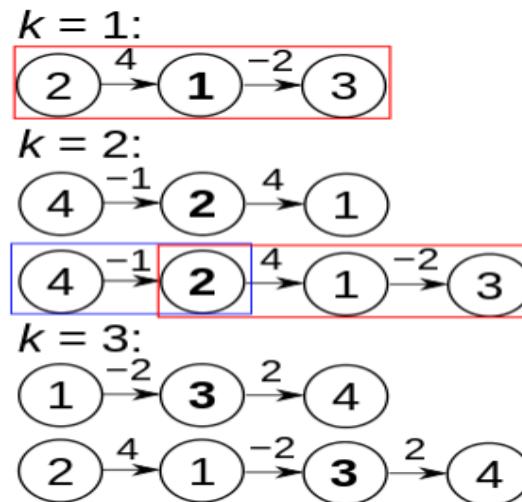
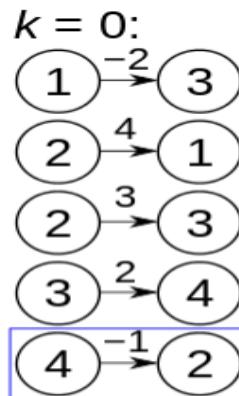
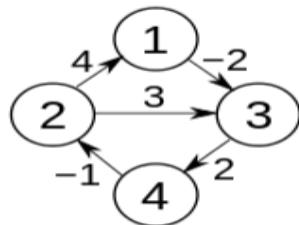
0	-1	-2	0
4	0	2	4
5	1	0	2
3	-1	1	0

Final result

Floyd-Warshall (Sequential code)

```
Floyd-Warshall(G, W)
{
    n ← |V|
    D(0) ← W
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                if D(k-1) [i, j] > D(k-1) [i, k] + D(k-1) [k, j]
                then D(k) [i, j] ← D(k-1) [i, k] + D(k-1) [k, j]
                else D(k) [i, j] ← D(k-1) [i, j]
    return D(n)
}
```

How to parallelize it?



Implementation I

- 1 block and n threads.
- Thread i updates the SP for vertex i .

```
__global__ void FW_APSP(int k, int D[n][n]) {
    int i = threadIdx.x;
    for (int j = 0; j < n; j++)
        if (D[i][j] > D[i][k] + D[k][j])
            D[i][j] = D[i][k] + D[k][j];
}

int main() { ...
    for (int k = 0; k < n, k++)
        FW_APSP<<<1, n>>>(k, D);
}
```

Simple! But can it be faster ?

Implementation 2

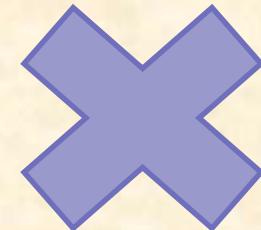
- Each thread updates one pair of vertices
 - Increase parallelism from n to n^2

```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    if (D[i][j] > D[i][k] + D[k][j])  
        D[i][j] = D[i][k] + D[k][j];  
}  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    for (int k = 0; k < n, k++)  
        FW_APSP<<<1, threadsPerBlock >>>(k, D);  
}
```

- How about the for-loop of k ?

Implementation 3

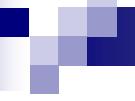
```
__global__ void FW_APSP(int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    for (int k = 0; k<n, k++)  
        if (D[i][j]>D[i][k]+D[k][j])  
            D[i][j]= D[i][k]+D[k][j];  
}  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    FW_APSP<<<1, threadsPerBlock >>>(D);  
}
```



- It is a synchronous computation
 - There are data dependency on k...

Add __syncthreads()

```
__global__ void FW_APSP(int D[n][n]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    for (int k = 0; k<n, k++){
        if (D[i][j]>D [i][k]+D[k][j])
            D[i][j]= D[i][k]+D[k][j];
        __syncthreads();
    }
}
int main() { ...
    dim3 threadsPerBlock(n, n);
    FW_APSP<<<1, threadsPerBlock >>>(D);
}
```



Outline

- Programming Model
- CUDA Language
- Example Code Study
- **CPU & GPU Synchronization**
- Multi-GPU

Asynchronous Functions

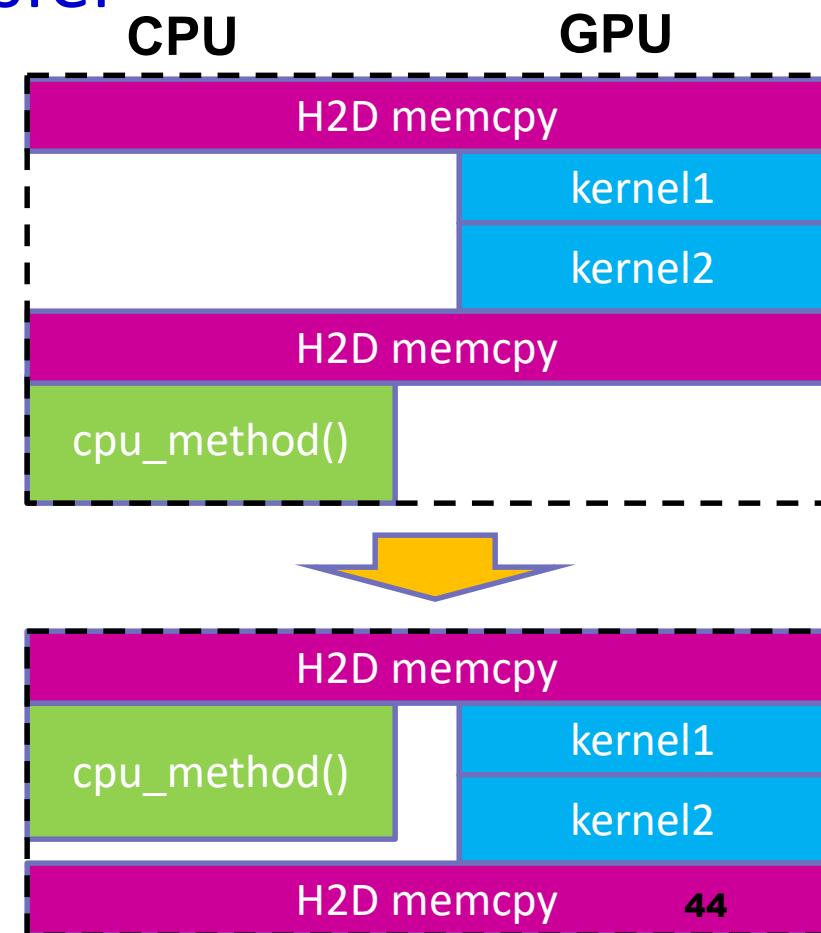
- To facilitate concurrent execution between host and device, most CUDA function calls are asynchronous:
 - Control is returned to the host thread before the device has completed the requested task.
 - But function calls from a kernel are serialized on GPU
- Asynchronous functions:
 - Kernel launches
 - Asynchronous memory copy and set options:
cudaMemcpyAsync, cudaMemcpyAsync
 - **cudaMemcpy** within the same device
 - H2D cudaMemcpy of **64kB or less**
 - **cudaEvent** functions

Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

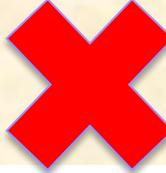
```
void main() {  
    cudaMemcpy ( /**/, H2D ) ;  
    kernel1 <<< grid, block>>> () ;  
    kernel2 <<< grid, block>>> () ;  
    cudaMemcpy ( /**/, D2H ) ;  
    cpu_method() ;  
}
```

```
void main() {  
    cudaMemcpy ( /**/, H2D ) ;  
    kernel1 <<< grid, block>>> () ;  
    kernel2 <<< grid, block>>> () ;  
    cudaMemcpyAsync ( /**/, D2H ) ;  
    cpu_method() ;  
}
```



Risk of Using Asynchronous Functions

- Programmer must enforce synchronization between **GPU** and **CPU** when there is **data dependency**

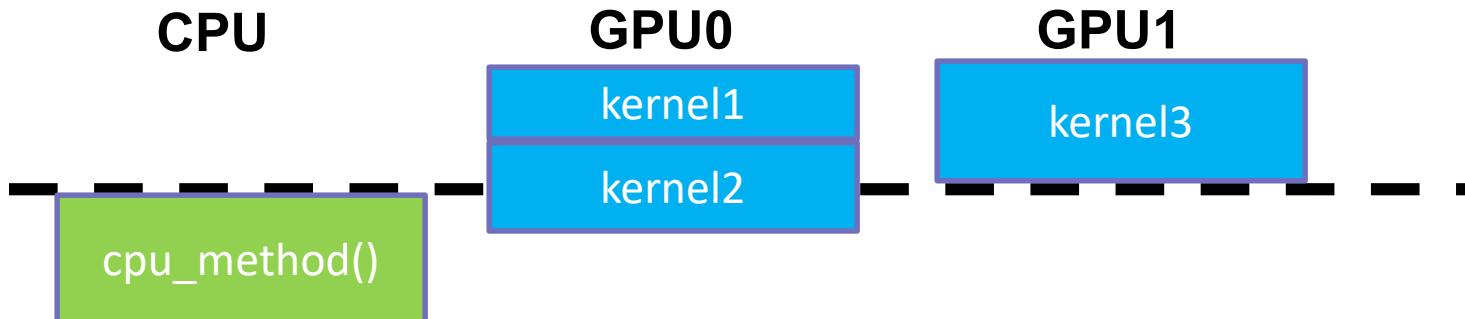
```
void main()  {  
    cudaMemcpyAsync ( d_a, h_a, count, H2D ) ;  
    kernel <<< grid, block>>> (d_a) ;  
    cudaMemcpyAsync ( h_a, d_a, count, D2H ) ;  
    cpu_method(h_a);    
}
```

Synchronization between CPU & GPU

- Device based: `cudaDeviceSynchronize()`
 - Block a **CPU thread** until all issued CUDA calls to a **device** complete
- Context based: `cudaThreadSynchronize()`
 - Block a **CPU thread** until all issued CUDA calls from the **thread** complete
- Stream based: `cudaStreamSynchronize(stream-id)`
 - Block a **CPU thread** until all CUDA calls in stream `stream-id` complete
- Event based:
 - `cudaEventSynchronize(event)`
 - ◆ Block a **CPU thread** until event is recorded
 - `cudaStreamWaitEvent(steam-id, event)`
 - ◆ Block a **GPU stream** until event reports completion

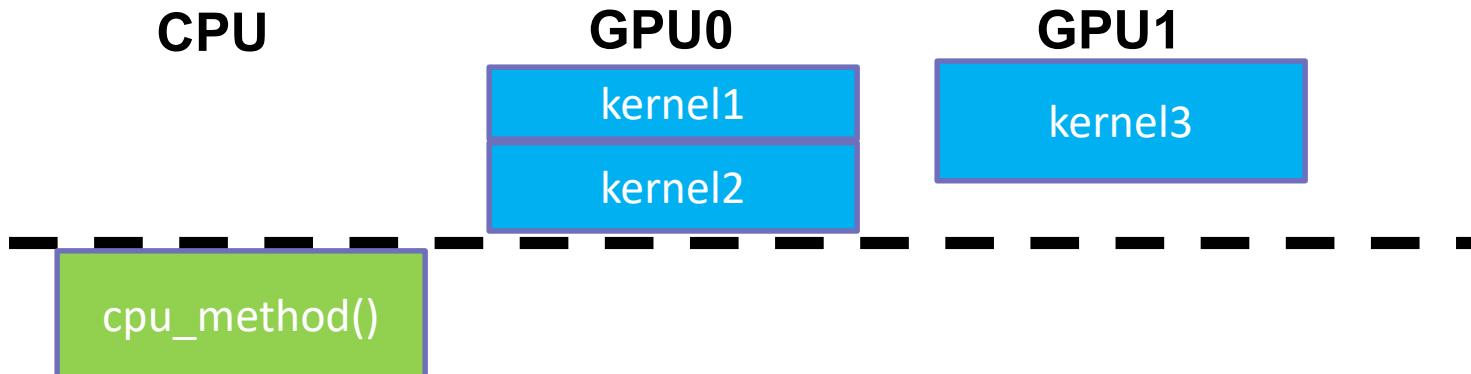
Device Synchronization Example

```
void main()  {  
    cudaSetDevice(0);  
    kernel1 <<< grid, block>>> () ;  
    kernel2 <<< grid, block>>> () ;  
    cudaSetDevice(1);  
    kernel3 <<< grid, block>>> () ;  
    cudaDeviceSynchronize()  
    cpu_method();  
}
```



Thread Synchronization Example

```
void main()  {  
    cudaSetDevice(0);  
    kernel1 <<< grid, block>>> () ;  
    kernel2 <<< grid, block>>> () ;  
    cudaSetDevice(1);  
    kernel3 <<< grid, block>>> () ;  
    cudaThreadSynchronize()  
    cpu_method();  
}
```

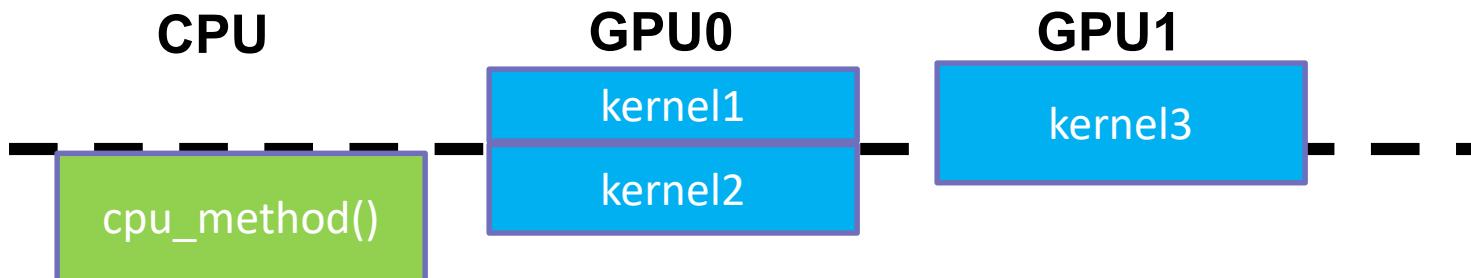


CUDA event

- Data type : `cudaEvent_t`
- `cudaError_t cudaEventCreate(cudaEvent_t* event)`
 - Create CUDA event
- `cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)`
 - Record CUDA event
 - If stream is non-zero, the event is recorded after all preceding operations in the stream have been completed
 - Since **operation is asynchronous**, `cudaEventQuery()` and/or `cudaEventSynchronize()` must be used to determine when the event has actually been recorded
- `cudaError_t cudaEventSynchronize(cudaEvent_t event)`
 - Wait until the completion of **all device work preceding the most recent call to `cudaEventRecord()`**

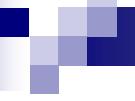
Event Synchronization Example

```
void main()  {  
    cudaSetDevice(0);  
    kernel1 <<< grid, block>>> () ;  
    cudaEventRecord(event)  
    kernel2 <<< grid, block>>> () ;  
    cudaSetDevice(1);  
    kernel3 <<< grid, block>>> () ;  
    cudaEventSynchronize (event)  
    cpu_method();  
}
```



Kernel Time Measurement Example

```
1. cudaEvent_t start, stop;
2. cudaEventCreate(&start);                                Create event
3. cudaEventCreate(&stop);                               Record event
4. cudaEventRecord(start);
5. kernel<<<block, thread>>>();
6. cudaEventRecord(stop);                                Record event and
7. cudaEventSynchronize(stop);                            synchronize
8. float time;
9. cudaEventElapsedTime(&time, start, stop);          Compute the event duration
```



Outline

- Programming Model
- CUDA Language
- Example Code Study
- CPU & GPU Synchronization
- Multi-GPU

Multi-GPUs

■ Multi-GPUs within a node

- A single CPU thread, multiple GPU
- Multiple CPU threads belonging to the **same process**, such as **pthread** or **openMP**

■ Multiple GPUs on multiple nodes

- Need to go through network API, such as **MPI**

Single thread multi-GPUs

- All CUDA calls are issued to the current GPU

- `cudaSetDevice()` sets the current GPU

```
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetDeviceCount(&numDevs);
for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    kernel<<<blocks, threads>>>(args);
}
```

- Asynchronous calls (kernels, memcopies) don't block switching the GPU

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```

Using CUDA with OpenMP

- Put CUDA functions inside the parallel region
- General setting:
 - The number of CPU threads is the same as the number of CUDA devices. Each CPU thread controls a different device, processing its portion of the data.
- It's possible to use more CPU threads than there are CUDA devices, BUT...
 - Several CPU threads will be allocating resources and launching kernel on the same device, which will slow down the performance.
 - Kernel launched on a single GPU could not operate concurrently (unless using different **stream**)

Example: cudaOMP.cu

```
...
cudaGetDeviceCount (&num_gpus);
...
omp_set_num_threads(num_gpus);
// create as many CPU threads as there are CUDA devices
#pragma omp parallel
{
    unsigned int cpu_thread_id = omp_get_thread_num();
    unsigned int num_cpu_threads = omp_get_num_threads();
    cudaSetDevice(cpu_thread_id);
    int gpu_id = -1;
    cudaGetDevice(&gpu_id);
    printf("CPU thread %d (of %d) uses CUDA device %d\n",
           cpu_thread_id, num_cpu_threads, gpu_id);
...
}
```

Using CUDA with MPI

```
int main(int argc, char* argv[]) {
    int rank, size;
    int A[32];
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    for(i = 0; i< 32; i++) A[i] = rank+1;
launch(A); // a call to launch CUDA kernel
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Example: launch (A)

```
extern "C"  
  
void launch(int *A) {  
    int *dA;  
  
    cudaMalloc( (void**) &dA, sizeof(int)*32);  
    cudaMemcpy(dA, A, sizeof(int)*32,  
              cudaMemcpyHostToDevice);  
    kernel<<<1, 32>>>(dA);  
    cudaMemcpy(A, dA, sizeof(int)*32,  
              cudaMemcpyDeviceToHost);  
    cudaFree(dA);  
}
```

Sharing data between GPUs

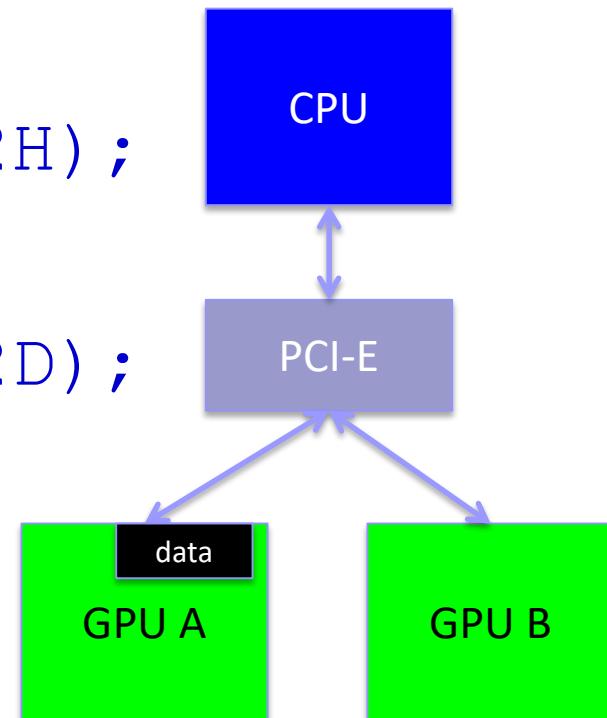
■ Options

1. Explicit copies via host
2. Zero-copy shared host array
3. Peer-to-peer memory copy

1. Explicit copies via host

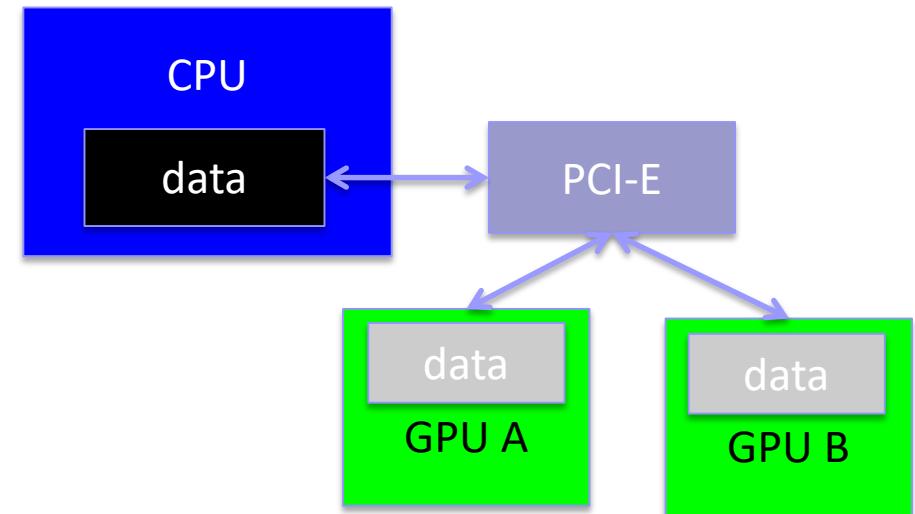
- CPU explicitly copies data from device A to device B
- Example:

```
cudaSetDevice(0);  
cudaMemcpy(DM1, HM, n, D2H);  
cudaSetDevice(1);  
cudaMemcpy(HM, DM2, n, H2D);
```



2. Using zero-copy

- “Zero-copy” refers to direct device access to host memory
- Device threads can read directly from host memory over PCI-e without using cudaMemcpy H2D or D2H
- The host memory must be pinned (page-locked)
 - Pageable memory cannot be directly accessed by the GPU because of the OS virtual memory mechanism



Host Memory Allocation

■ `malloc()`

- Regular C library memory allocation
- Managed by host
- Cannot be directly accessed by GPUs
- Must be copied to device memory via `cudaMemcpy()`

■ `cudaMallocHost(void ** hostPtr, size_t size)`

- Allocate pinned (page-locked) host memory for higher `cudaMemcpy` performance
- Used with `cudaMemAsync()` for **async memory copy** or CUDA stream

■ `cudaHostAlloc(void ** hostPtr, size_t size, unsigned int flags)`

- Add the flag “`cudaHostAllocMapped`” to allocate pinned host memory for higher `cudaMemcpy` performance
- Add the flag “`cudaHostAllocPortable`” to allocate shared host memory for “Zero copy”

➔ All these functions return **host memory pointer**

Zero Copy via cudaHostAlloc()

- Allocate **host memory** using cudaHostAlloc()
 - It returns a **host pointer**. It can enable faster host memory access.
 - **Must** add flag cudaHostAllocMapped for **page-locked**
 - Add flag cudaHostAllocPortable for **sharing among all devices**
- Bind host pointer to device pointer using
cudaHostGetDevicePointer(void**, void*, 0)

```
int *hostPtr, *dev0Ptr, *dev1Ptr;  
cudaHostAlloc(&hostPtr, 10, cudaHostAllocMapped |  
            cudaHostAllocPortable);  
cudaSetDevice(0);  
cudaHostGetDevicePointer(&devPtr0, hostPtr, 0);  
kernel<<1,10>>(devPtr0);  
cudaSetDevice(1);  
cudaHostGetDevicePointer(&dev1Ptr, hostPtr, 0);  
kernel<<1,10>>(devPtr1);
```

Pitfalls of using zero-copy

- PCI-e is lower in bandwidth and higher in latency than **GPU global memory**
 - ➔ access speed is slower than global memory
- Use zero-copy if:
 - The data is **only accessed once** or few times
 - Generate data on the device and copy back to host **without reuse**
 - Kernel(s) that access the memory are **compute bound**

3. Peer-to-Peer Memcpy

- Direct copy from pointer on GPU A to pointer on GPU B
- Using two functions

```
cudaError_t cudaMemcpyPeer(void *dst, int dstDevice, const void* src, int srcDevice, size_t count)
```

```
cudaError_t cudaMemcpyPeerAsync(void *dst, int dstDevice, const void* src, int srcDevice, size_t count, cuda_stream_t stream=0)
```

Example: P2P memcpy

```
cudaSetDevice(0);      //set device 0 as current
float *p0;
size_t size = 1024*sizeof(float);
cudaMalloc(&p0, size); //allocate mem on device 0
cudaSetDevice(1);      //set device 1 as current
float *p1;
cudaMalloc(&p1, size); //allocate mem on device 1

cudaSetDevice(0);      //set device 0 as current
Kernel1<<<1000,128>>>(p0); //launch on dev 0
cudaSetDevice(1);      //set device 1 as current
cudaMemcpyPeer(p1,1,p0,0,size); //copy p0 to p1
Kernel1<<<1000,128>>>(p1); //launch on dev 1
```

Reference

- [NVIDIA CUDA Library Documentation](#)
- [NVIDIA, Introduction to Dynamic Parallelism](#)
- [NVIDIA, CUDA C++ Best Practices Guide](#)
- [CUDA Tutorial](#)