GPU Parallel Programming: CUDA Summary

National Tsing-Hua University

2017, Summer Semester

Outline

- Execution
- Memory
- Synchronization
- Stream

CUDA program framework

GPU code (parallel)

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

```
#include <cuda_runtime.h>
__global__ void my_kernel(…) {
   …
}
int main() {
   …
   cudaMalloc(…)
   cudaMemcpy(…)
   …
      my_kernel<<<nblock,blocksize>>>(…)
   …
   cudaMemcpy(…)
   …
}
```

Function Qualifiers

Function qualifiers	limitations
`__device__` function	
`__global__` function	
`__host__` function	
Functions without qualifiers	
`__host__` `__device__` function	

Function Qualifiers

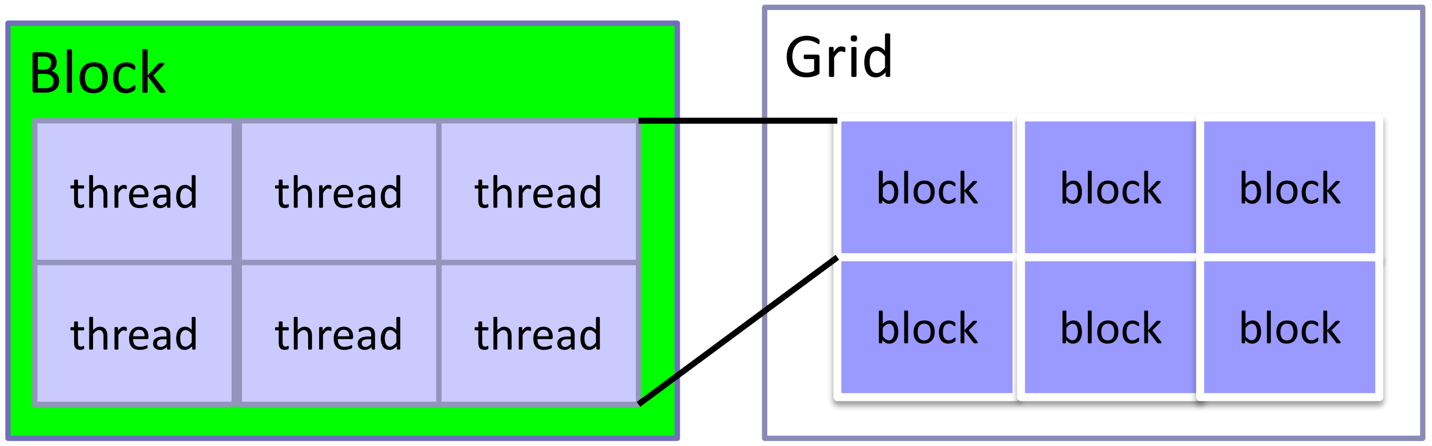
Function qualifiers	limitations
`__device__` function	Executed on the device Callable from the device only
`__global__` function	Executed on the device Callable from the host only (must have **void** return type!)
`__host__` function	Executed on the host Callable from the host only
Functions without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

CUDA Programming Terminology

- Host?
- Device?
- Kernel?
- Thread?
- Block is a group of what?
- Grid is a group of what?

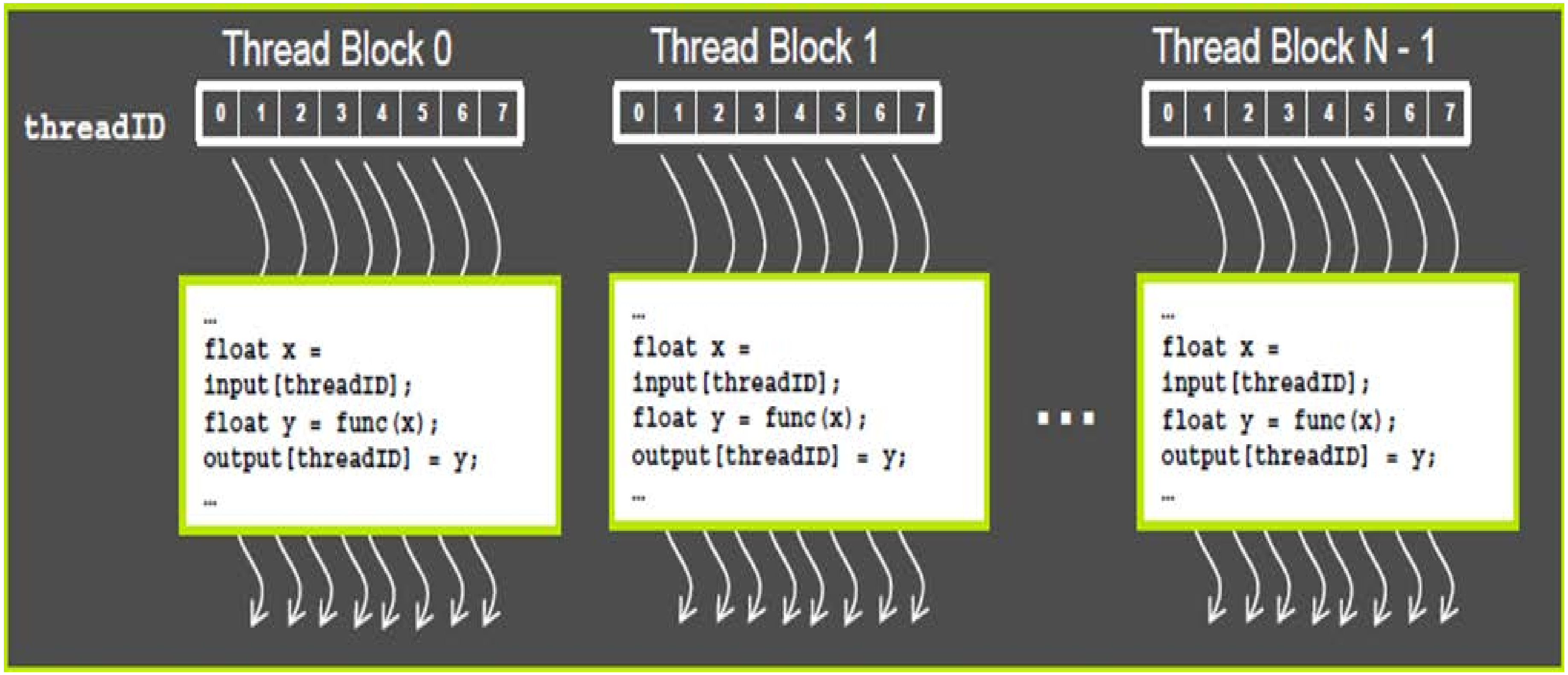
CUDA Programming Terminology

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



Hierarchy of Concurrent Threads

- Threads are grouped into thread blocks
  - Kernel = gird of thread blocks
  - Each thread has a unique **threadIdx** and **blockIdx** identifier

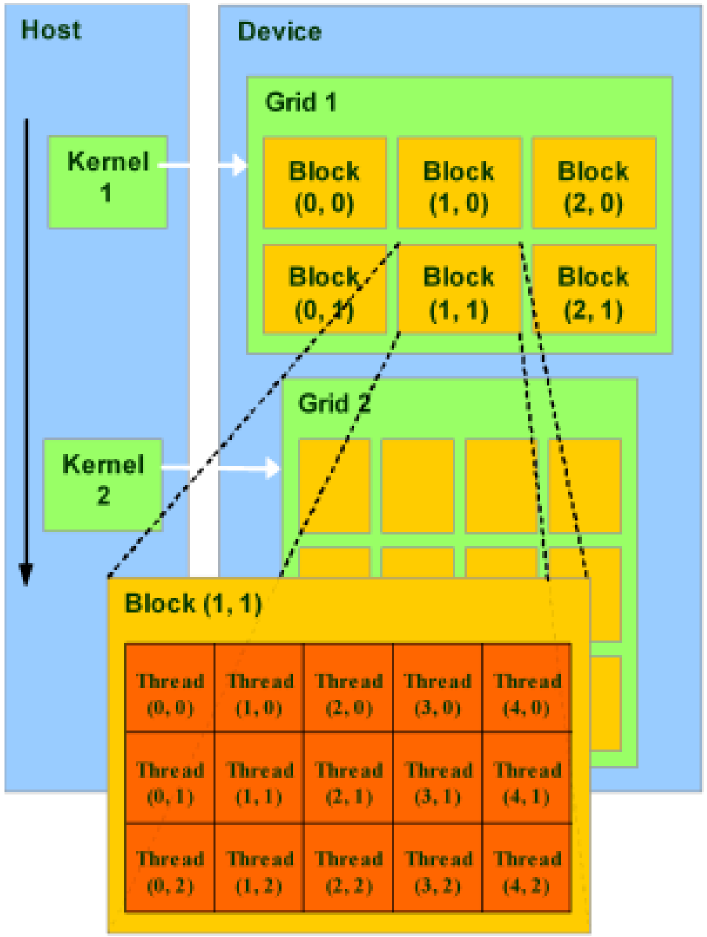


```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```

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?

Thread and Block IDs

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

```
// Kernel definition
__global__ void VecAdd(float* A)
{
      int i = 
      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);
```

Thread and Block IDs

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

```
// Kernel definition
__global__ void VecAdd(float* A)
{
     int i = 
     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);`

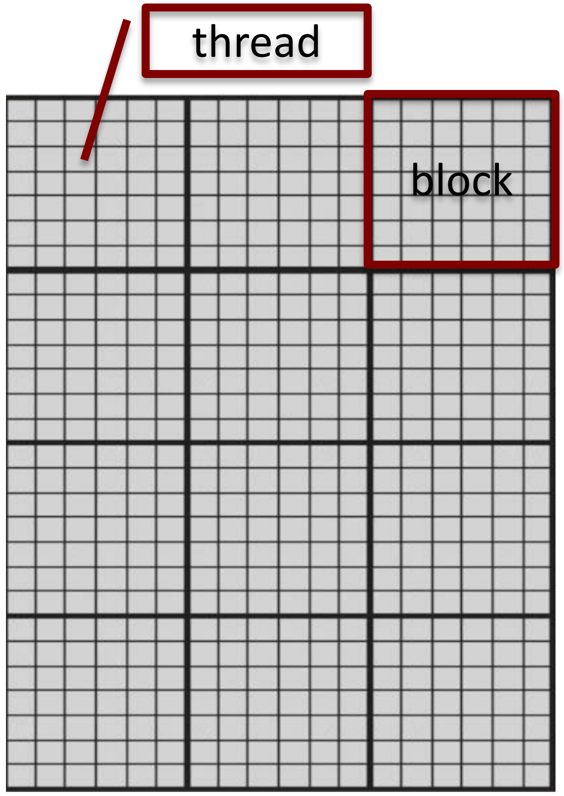
   ➔ int i=blockIdx*blockDim+threadIdx;
4. `size=10; dim3 blk(size, size);`

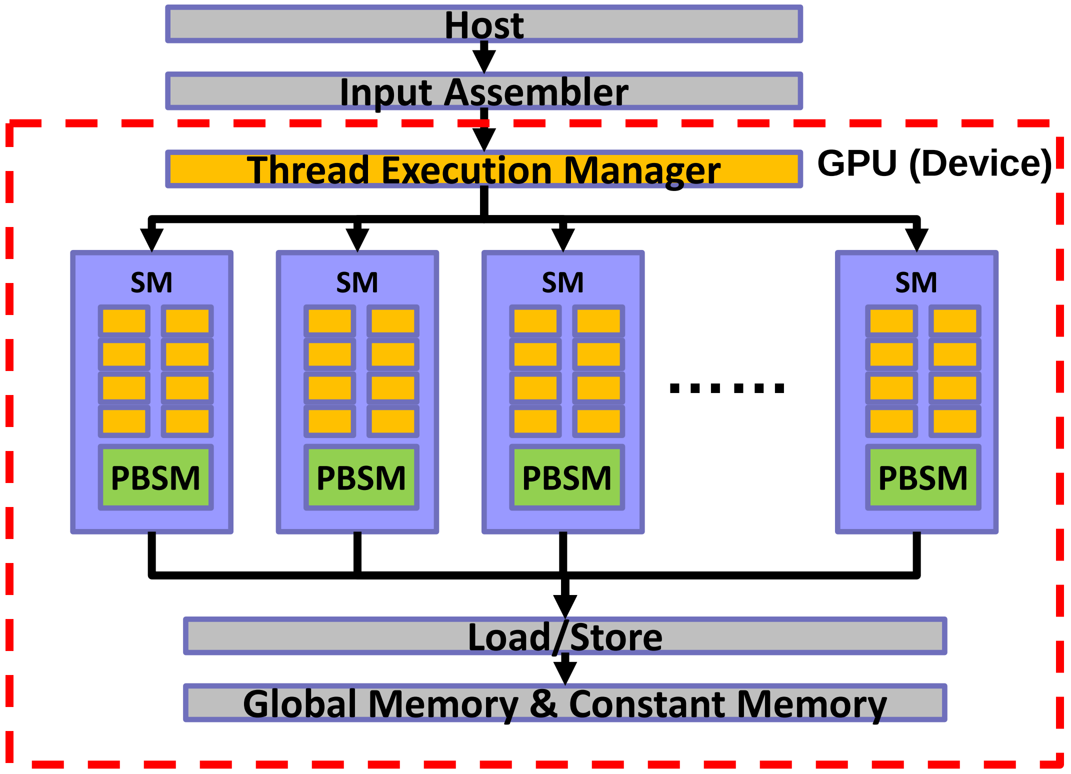
   `my_kernel<<< 1, blk >>>(A, size);`

   ➔ int i = threadIdx.x * blockDim.x + threadIdx.y;

Software to Hardware Mapping

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





Software to Hardware Mapping

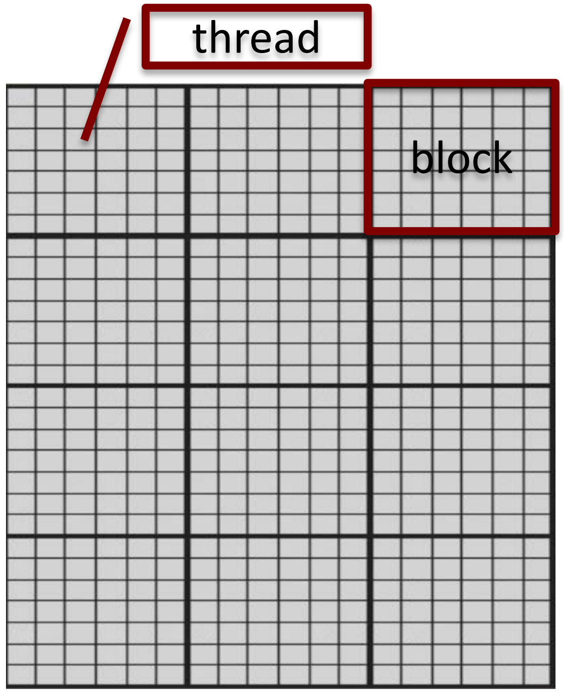
- Can a kernel run on two different devices at the same time?
- Can a kernel run across multiple SM processors?
- Can the threads from a same block run across multiple SM processors?
- What is the difference between the two kernals below?
  1. `my_kernel<<< 1, 100 >>>(A);`
  2. `my_kernel<<< 100, 1 >>>(A);`
- Wshared block memory can only be accessed by the threads in the same blocks?
- Why __syncthreads() is not supported across blocks?

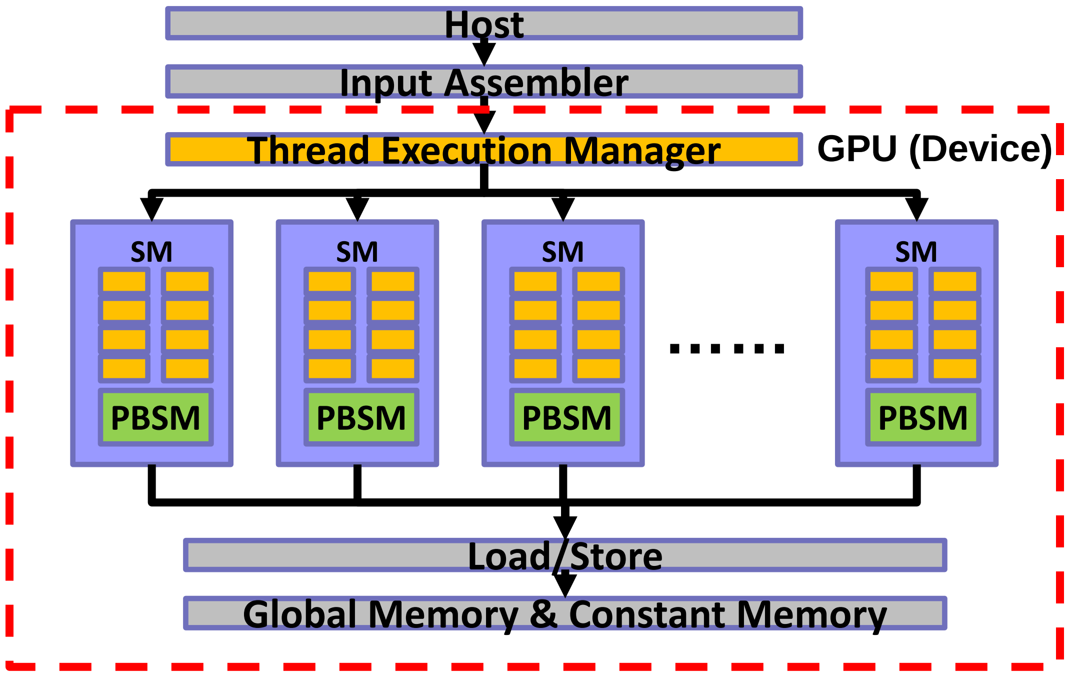
Software to Hardware Mapping

- Can a kernel run on two different devices at the same time? No
- Can a kernel run across multiple SM processors? Yes
- Can the threads from a same block run across multiple SM processors? No
- What is the difference between the two kernals 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 __syncthreads() is not supported across blocks?

Software to Hardware Mapping

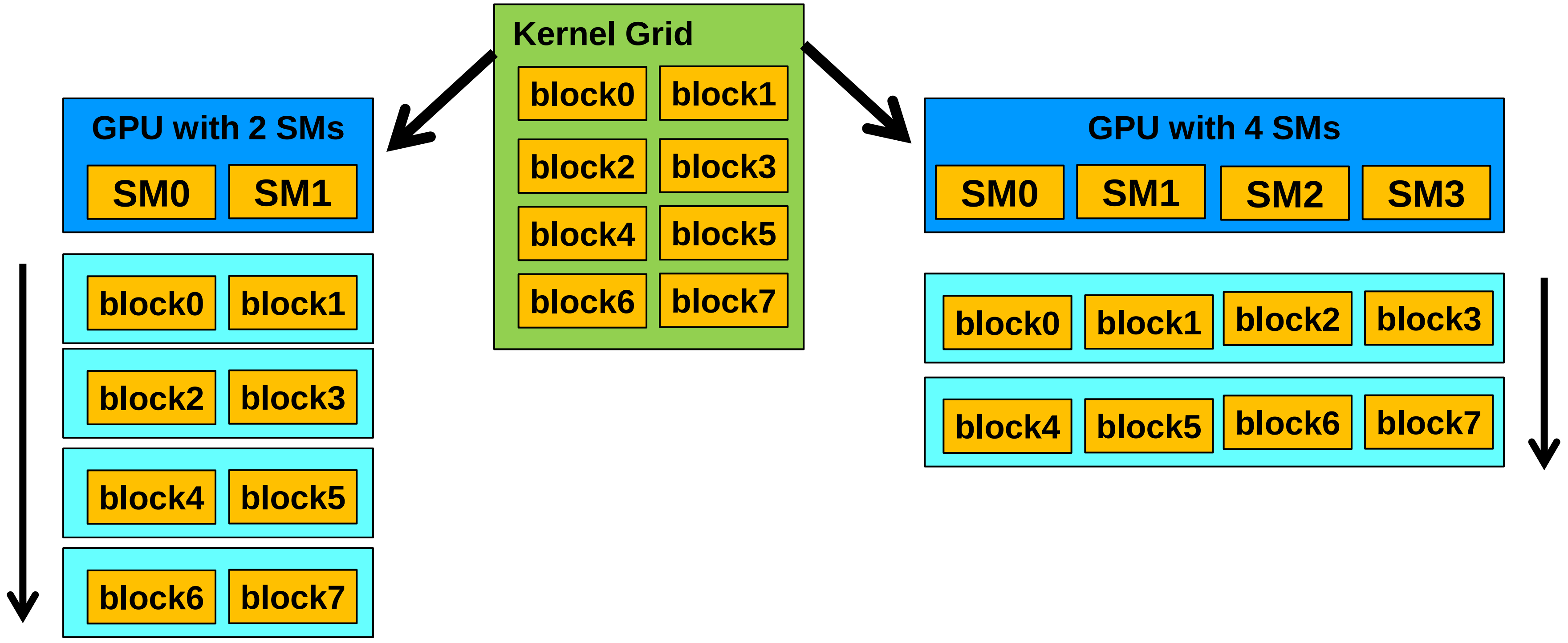
- The actual **software running instances** to **hardware computing unit** mapping is done by the hardware thread execution manager **at runtime**





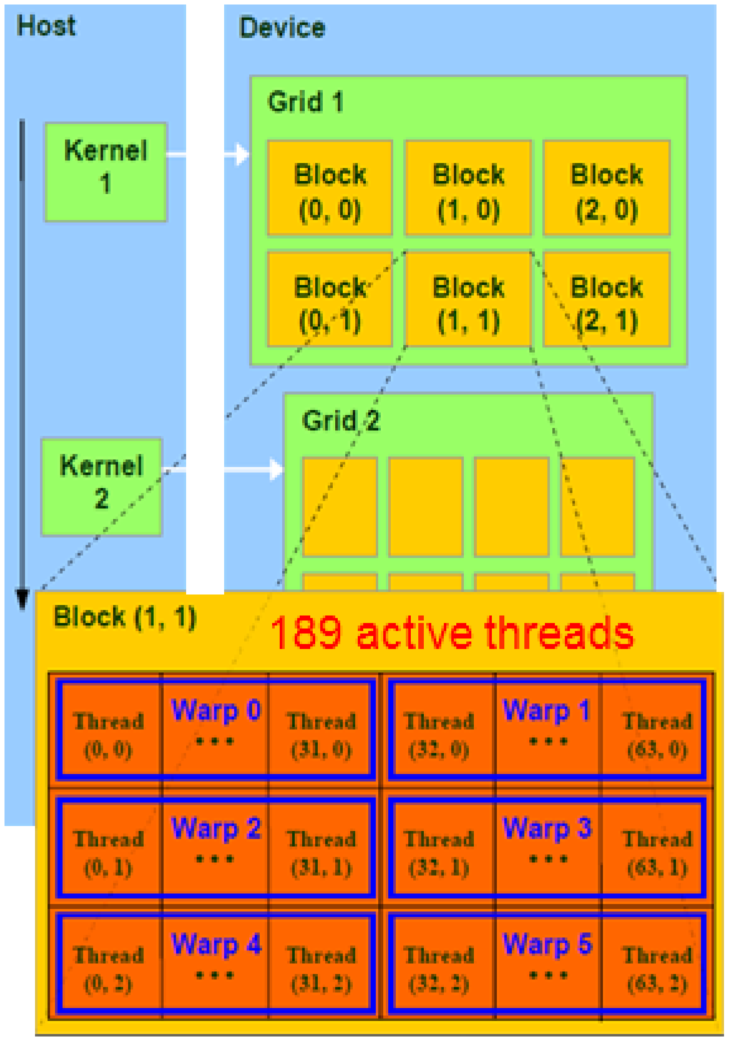
Block Level Scheduling

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



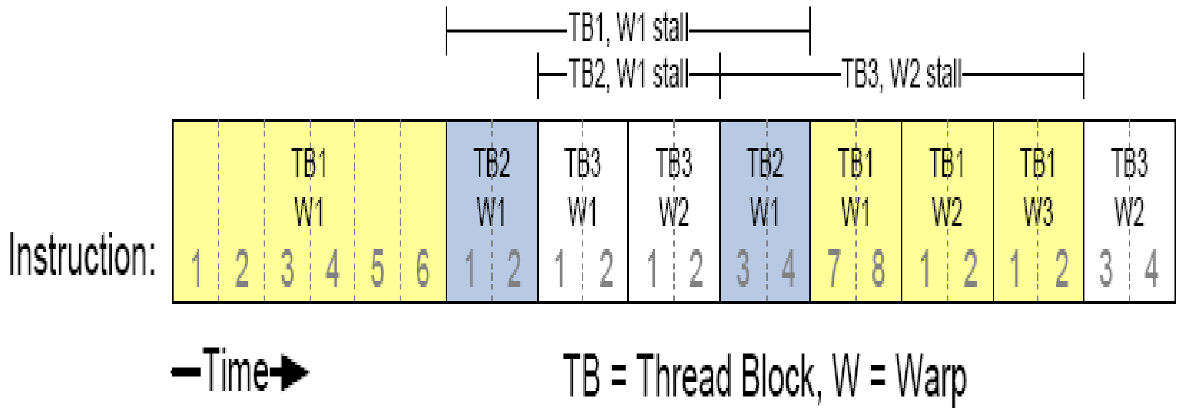
Thread Level Scheduling - Warp

- **Inside the SM**, threads are launched in groups of 32, called warps
  - Warps share the control part (warp scheduler)
  - At any time, **only one warp is executed per SM**
  - Threads in a warp will be executing the same instruction (SIMD)
- In other words …
  - Threads in a wrap execute **physically** in parallel
  - Warps and blocks execute **logically** in parallel



Thread Level Scheduling - Warp

- Hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Wraps are switched when memory stalls
- There are limited number of active blocks & warps
  - Maximum number of active blocks per SM is 8 (Fermi)
  - Maximum number of active warps per SM is 48 (Fermi)
  - Maximum number of active threads per SM is 48*32=1,536

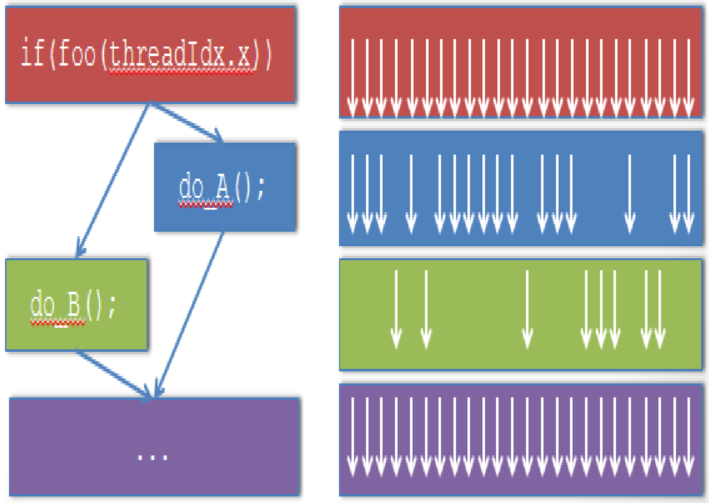


Software to Hardware Mapping

- Does the kernel below have to run on 10 different SM processors?
  - `my_kernel<<< 10, 10 >>>();`
- Can multiple blocks run on a single SM processors?
- If a SM only has 128 cores, does it mean there can only be 128 active threads on the SM?
- Why we have to call __syncthreads() within a block if there are data dependency between statements
  - A[threadIdx] = threadIdx; A[threadIdx] += A[threadIdx/2];
- Why branch will cause divergence?
  - If (A[threadIdx] > 0) A[threadIdx]=1; else A[threadIdx]=-1;

Software to Hardware Mapping

- Does the kernel below have to run on 10 different SM processors? No
  - `my_kernel<<< 10, 10 >>>();`
- Can multiple blocks run on a single SM processors? Yes
- If a SM only has 128 cores, does it mean there can only be 128 active threads on the SM? No
- Why we have to call __syncthreads() within a block if there are data dependency between statements
  - A[threadIdx] = threadIdx;
  - A[threadIdx] += A[threadIdx/2];
- Why branch will cause divergence?
  - If (A[threadIdx] > 0) A[threadIdx]=1; else A[threadIdx]=-1;



Outline

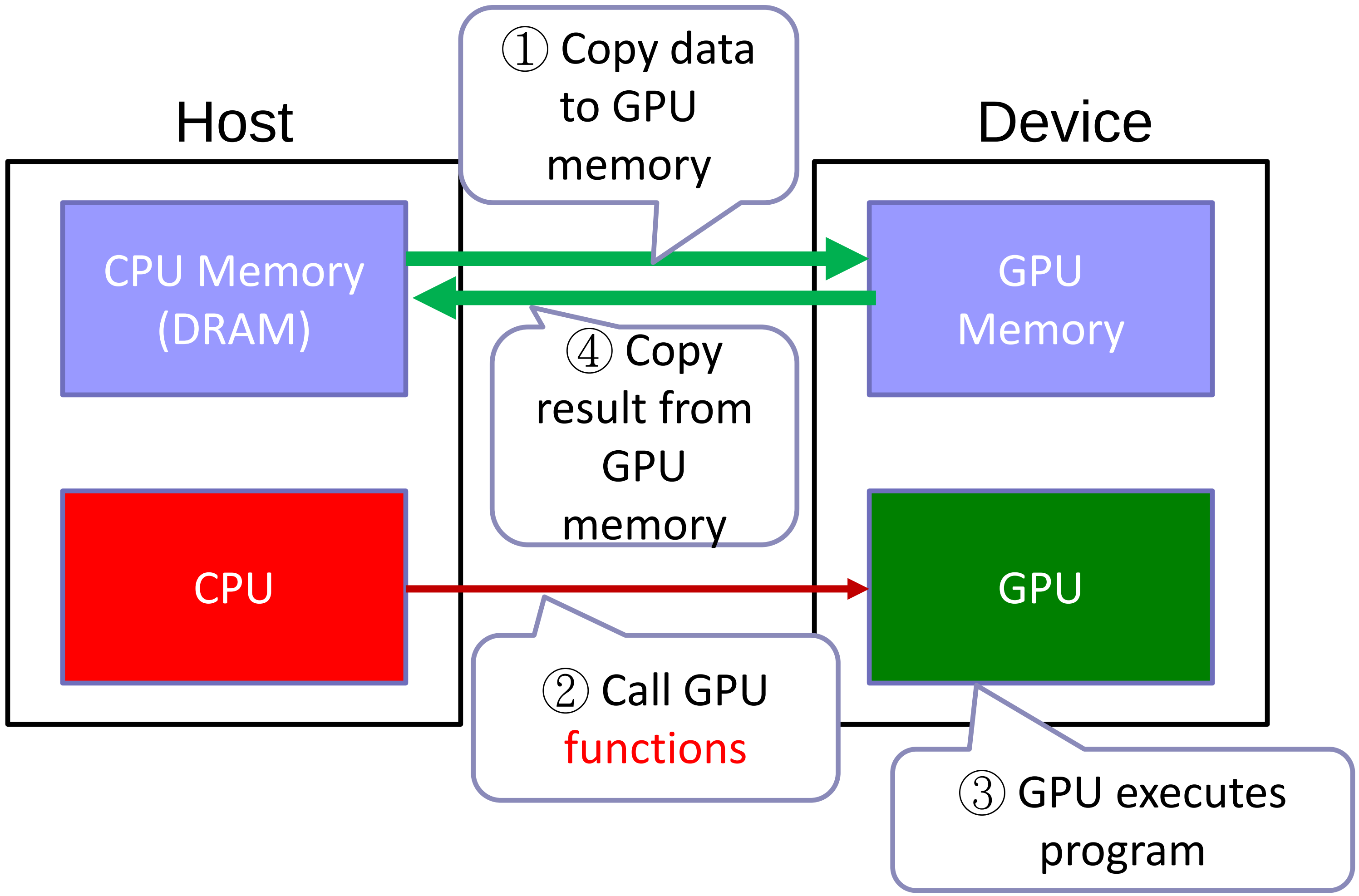
- Execution
- Memory
- Synchronization
- Stream

Outline

- Execution
- Memory
- Synchronization
- Stream

CUDA program flow

Q: Why we have to copy the memory content?



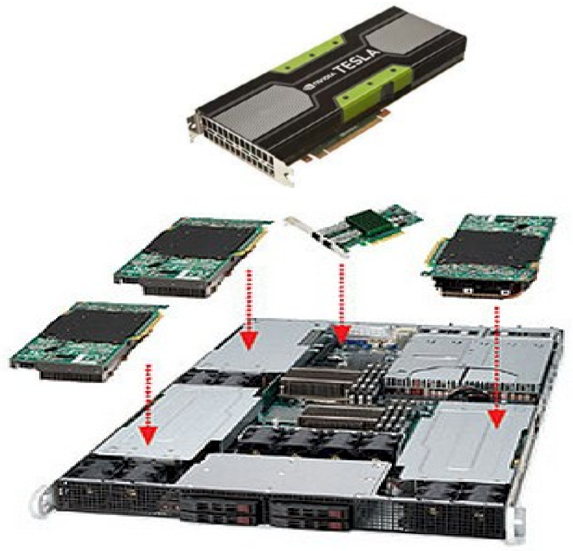
GPU Servers

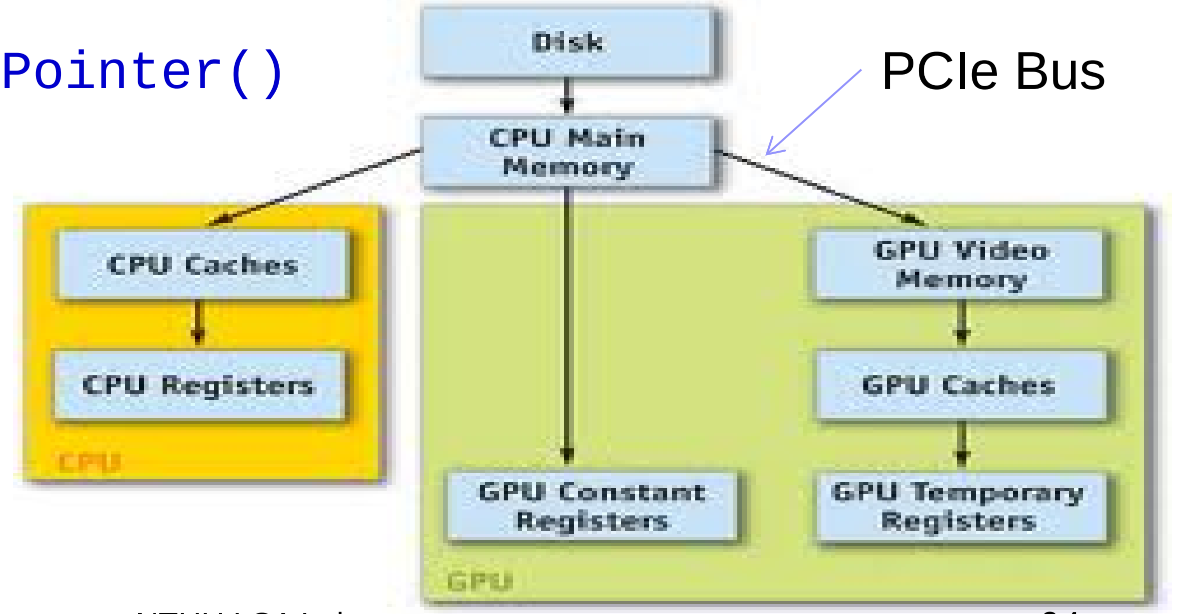
Device pointer & host pointer are in two different address space!

- Device memory is controlled by GPU
  - Must use device pointer
- Host memory is controlled by CPU
  - Must use host pointer
- **Zero copy** can allow GPU directly access host memory, but GPU still **use the device pointer not host pointer**

  `cudaHostAlloc()`

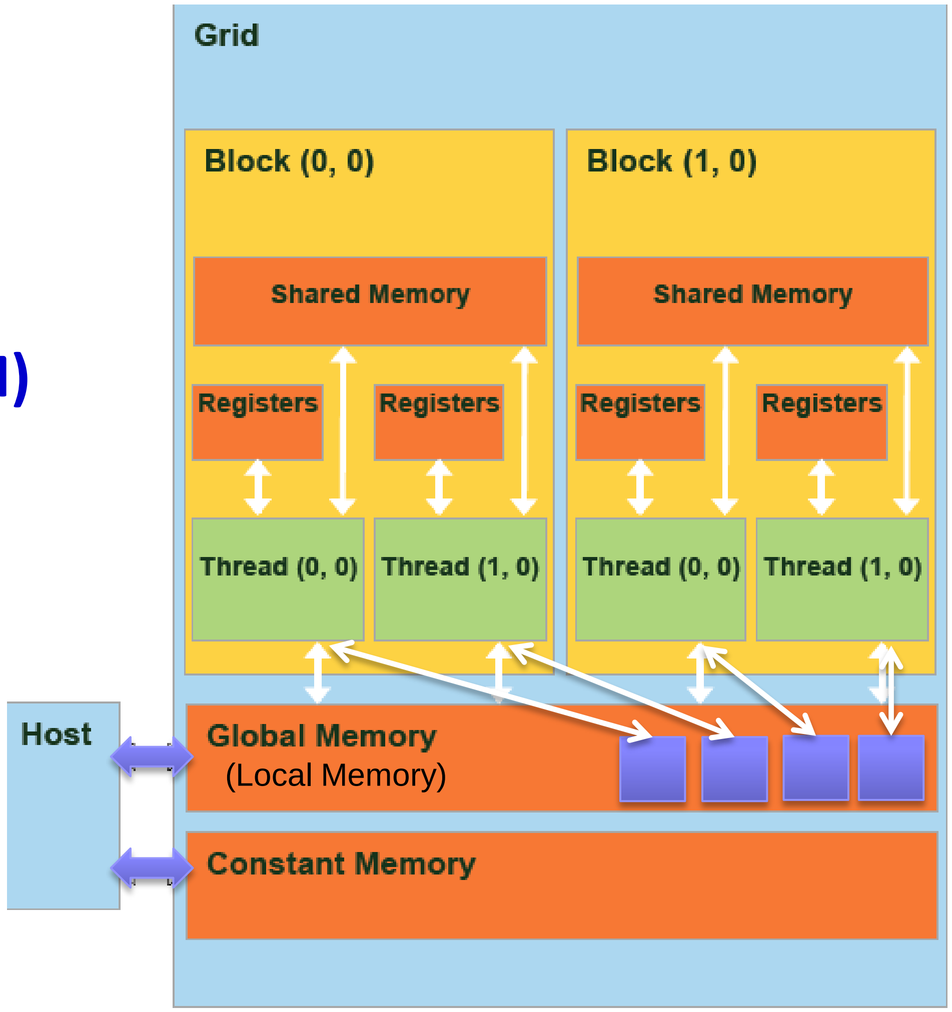
  `cudaHostGetDevicePointer()`





GPU Memory Hierarchy

- **Registers**
- **Shared memory**
- **Global/Local memory (DRAM)**
- **Constant memory**



Software to Hardware Mapping

CUDA Variables within a Kernel

Local variable

Shared variable

Global variable

Register

Local Memory

Shared Memory

Global Memory

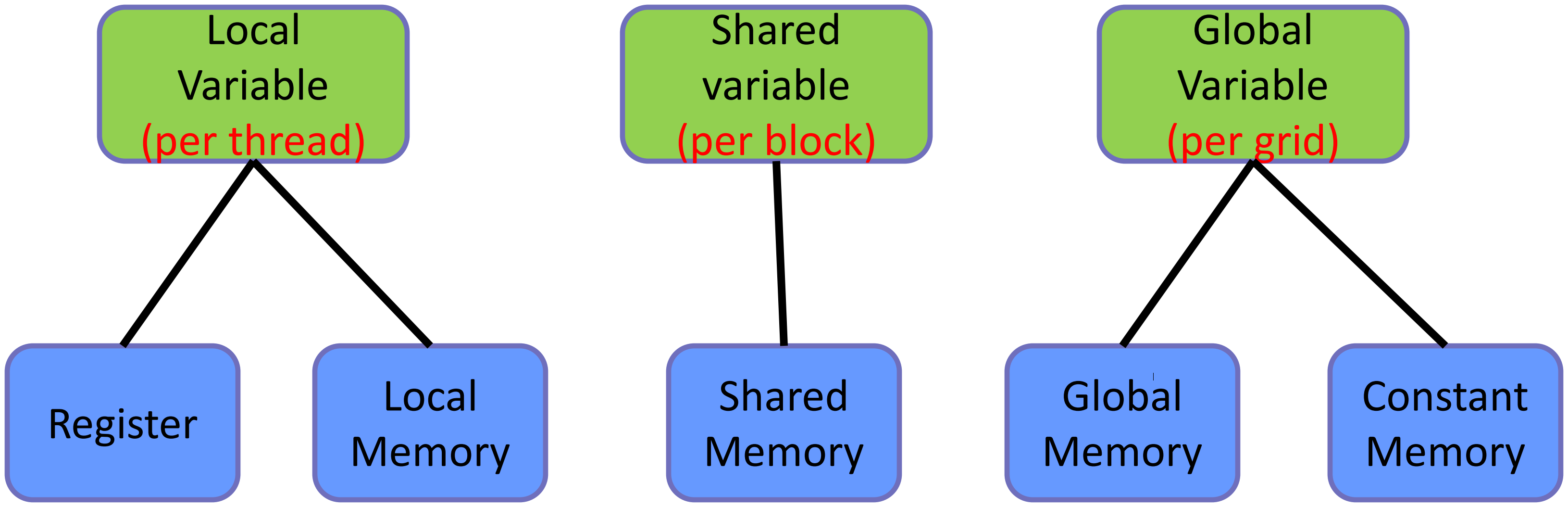
Constant Memory

GPU Memory Hierarchy

Q: What is the data scope of each variable, and what is the mapping between software and hardware?

Software to Hardware Mapping

CUDA Variables within a Kernel



GPU Memory Hierarchy

Q: What is the data scope of each variable, and what is the mapping between software and hardware?

Outline

- Execution
- Memory
  - Local variable
  - Global variable
  - Shared variable
- Synchronization
- Stream

Local Variable: Register vs Local Memory

- A local variable declared in a kernel function can be reside in **register** (on chip memory) or **local memory** (off chip memory). What are the differences between them?
  - When will a local variable be stored in local memory? (hint: only in 2 situations)

```
__global__ void distance(int *m, int *n, int *V){
    int i, j, k;
    int a[10], b[10], c[10];
}
```

Q: Where are the variables (i, j, k, a, b, c, m, n, V) stored (register, local memory, global memory, shared memory)?

Local Memory

Q: What is register pressure?

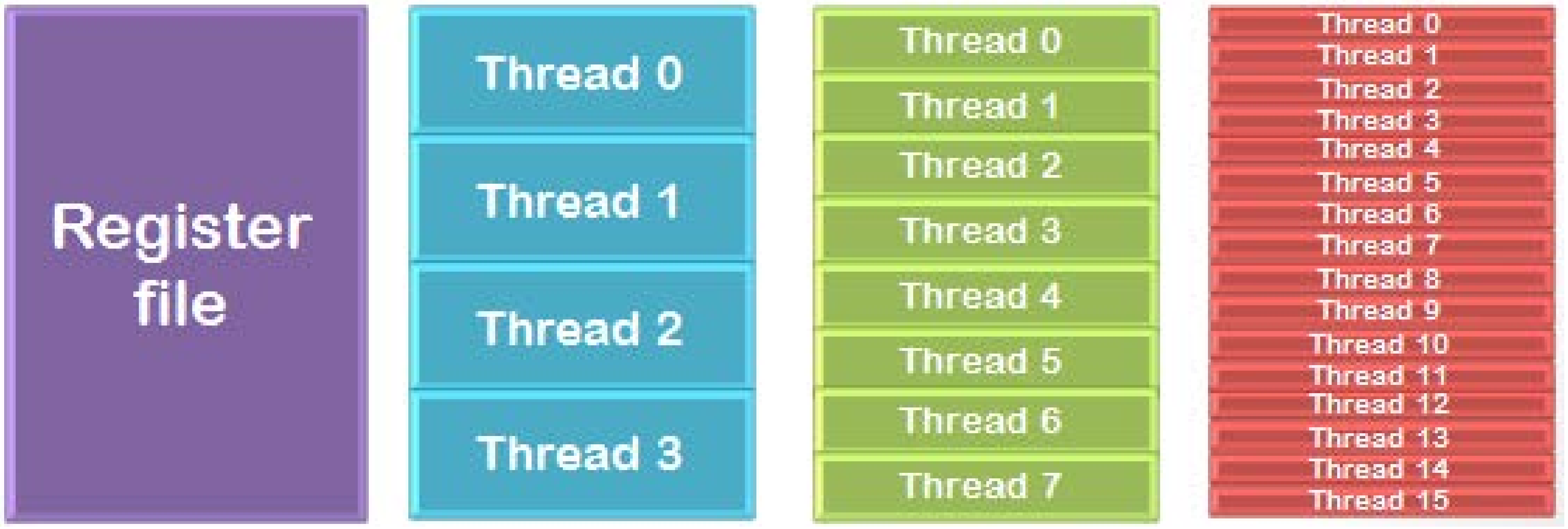
- Name refers to memory where registers and other thread-data (array) is spilled
  - Runs out of SM resources due to register pressure
  - Registers aren't indexable, so arrays have to be placed in local memory
  - "Local" because each thread has its own **private access scope**
- Details:
  - Not really a "memory" – bytes are stored in **global memory (DRAM)**

Q: Where is L1 cache?

- Differences from global memory:
  - Addressing is resolved by the compiler
  - Stores are cached in **L1**

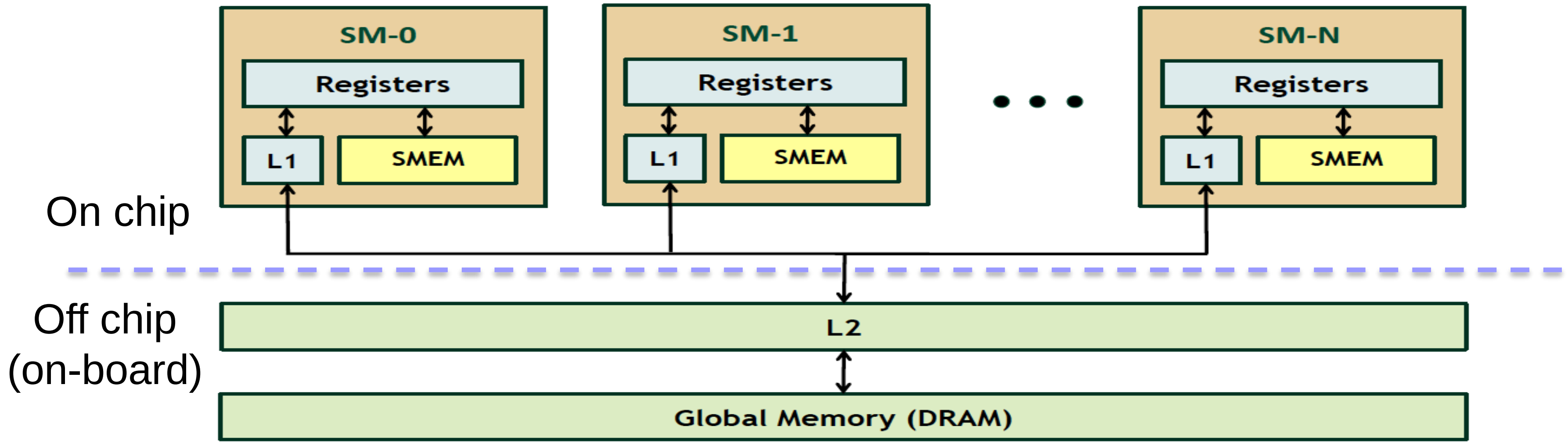
Register Pressure

- Too few threads
  - can't hide pipeline / memory access latency
- Too many threads
  - register pressure
  - Limited number of registers among concurrent threads
  - Limited shared memory among concurrent blocks



Memory Cache

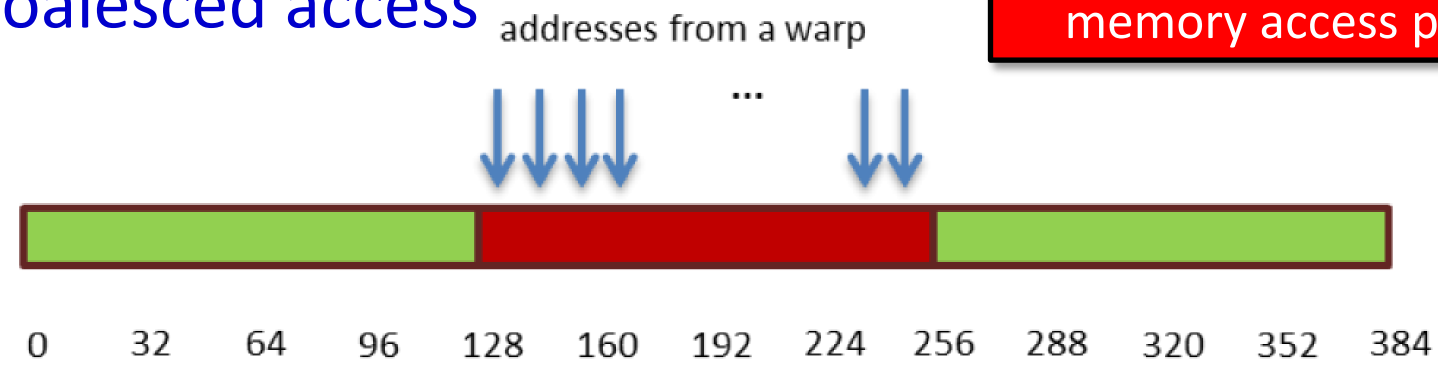
- L1 & L2 are used to cache local memory contents
  - L1: On chip memory. Same as share memory
    - Programmers can decide the ratio of shared memory and L1 cache
  - L2: Off chip memory Cache. Same as global memory



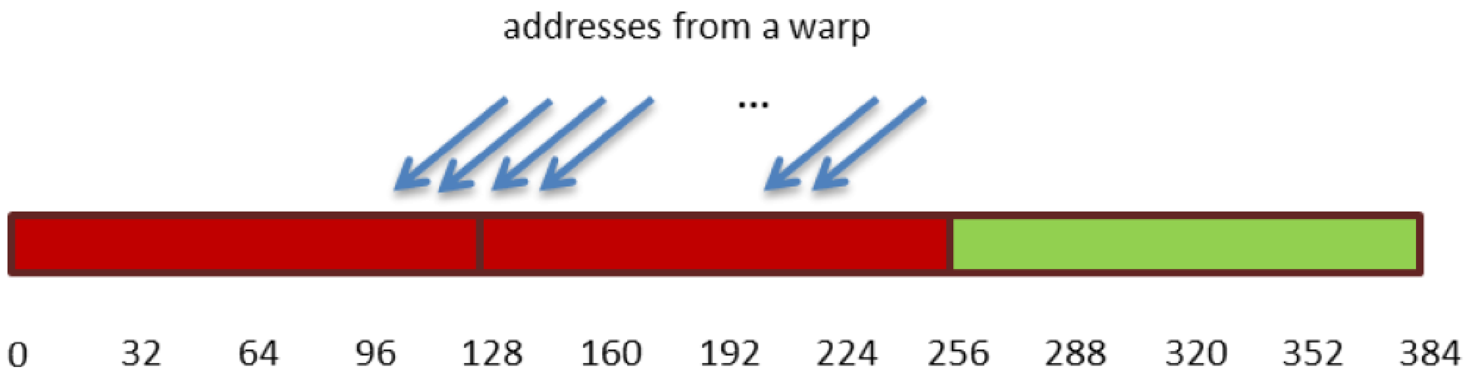
Coalesced Memory Access Problem

Prefer linear and aligned memory access pattern

- Coalesced access



- **Unaligned** sequential addresses that fit into **two** 128-byte **L1-cache lines**



Outline

- Execution
- Memory
  - Local variable
  - Global variable
  - Shared variable
- Synchronization
- Stream

Global Variable: Global Memory vs Constant Memory

- What are the two key differences between them?

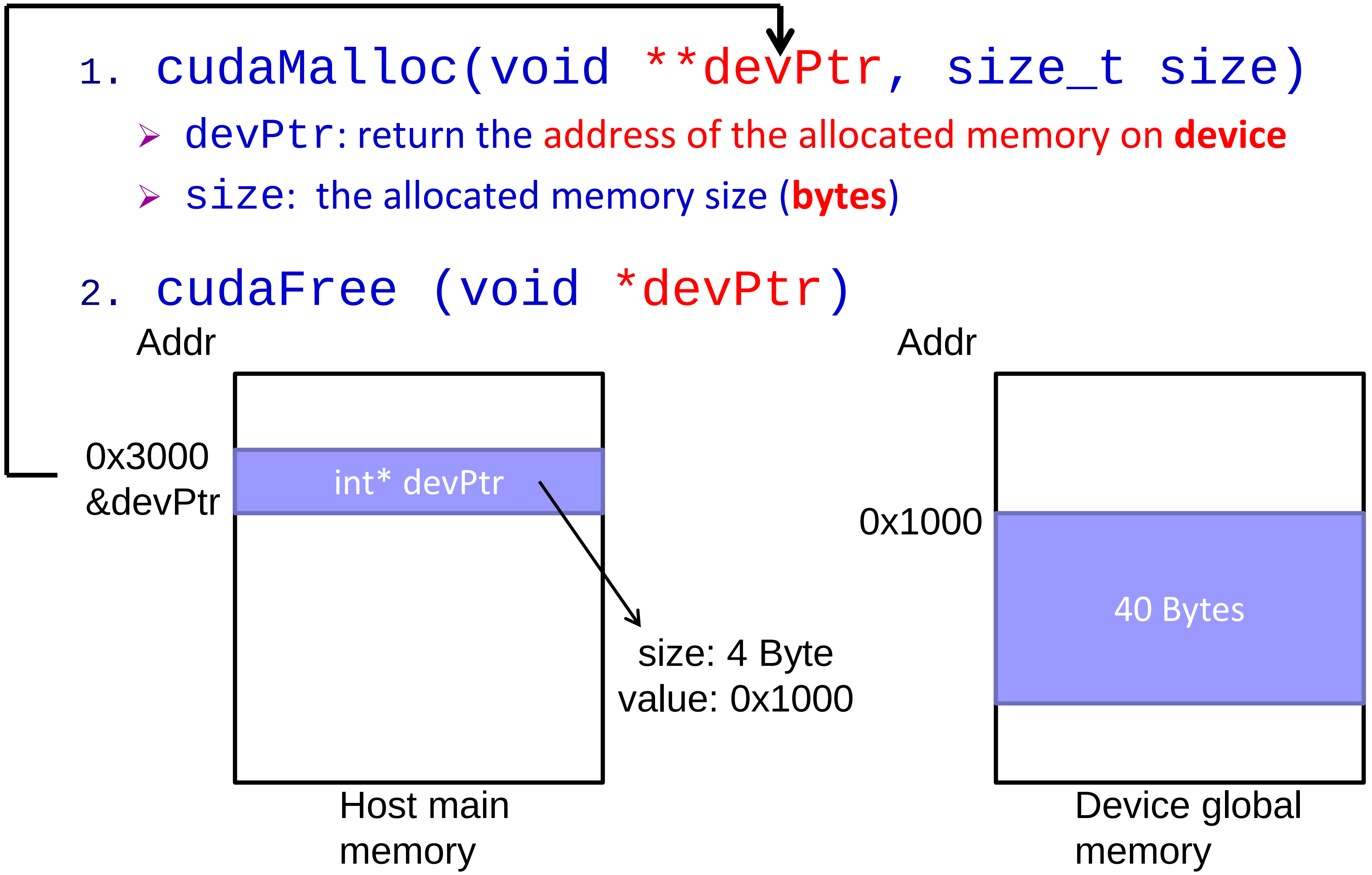
Global Variable: Global Memory vs Constant Memory

- What are the two key differences between them?
  - Constant memory is read only
  - Constant memory can be cached

How to Allocate Device (Global) Memory

1. `cudaMalloc(void **devPtr, size_t size)`
   - `devPtr`: return the address of the allocated memory on **device**
   - `size`: the allocated memory size (**bytes**)

2. `cudaFree (void *devPtr)`



Synchronous Global Memory Copy

- `cudaMemcpy( void *dst, const void *src, size_t count, enum` **`cudaMemcpyKind`** `kind)`
  - `count`: size in **bytes** to copy

`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()

Complier does not distinguish between the device pointer & host pointer.
You have to check by yourself very carefully.

Copy 3 Int between Device & Host

```
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, sizeof(int), cudaMemcpyDeviceToHost);
   // Cleanup
   cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
   return 0;
}
```

Asynchronous Global Memory Copy

- cudaMemcpyAsync: the **host** memory must be pinned using `cudaMallocHost()` or `cudaHostAlloc()` with the flag **`cudaHostAllocMapped`**

```
int main(void) {
    int *in, out;
    int *d_in, *d_out;
    cudaMallocHost(&in, sizeof(int));
    int* = 1;
    cudaMalloc((void **)&d_in, sizeof(int));
    cudaMalloc((void **)&d_out, sizeof(int));
    cudaMemcpyAsync(d_in,in,sizeof(int), cudaMemcpyHostToDevice);
    add<<<1,1>>>(d_in, d_out);
    cudaMemcpyAsync(&out, d_out, sizeof(int),
cudaMemcpyDeviceToHost);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

If host memory is not pinned, error may occur at runtime when the memory is swapped by OS

Host Memory ←→ Constant Memory

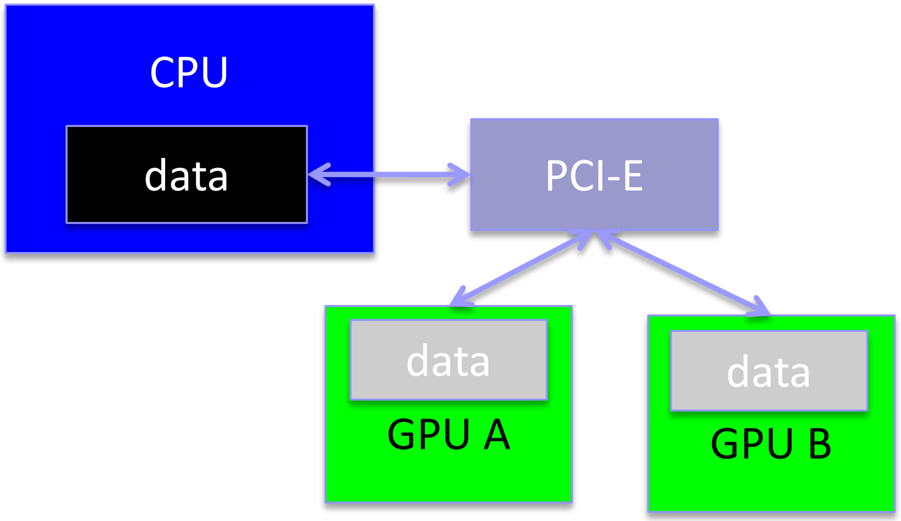
- Use cudaMemcpyToSymbol() & cudaMemcpyFromSymbol()

```
__constant__ int constData[100];
int main(void) {
   int A[100];
   cudaMemcpyToSymbol(constData, A, sizeof(A));
   add<<<grid_size,blk_size>>>();
   cudaMemcpyFromSymbol(A, constData, sizeof(A));
}
__global__ kernel() {
   int v = constData[threadIdx];
}
```

Page-Locked Data Transfers (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

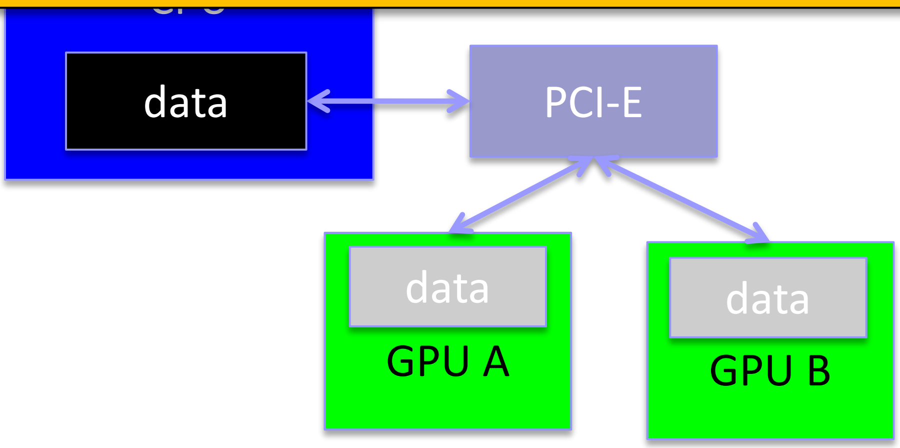
Q: What are the two purposes of using zero-copy?



Page-Locked Data Transfers (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

Q: What are the two purposes of using zero-copy?
Ans: (1) avoid the overhead of memory copy
(2) used as a shared memory between 2 GPU devices



Page-Locked Data Transfers (Zero-copy)

- 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);
```

cudaHostAlloc()vs cudaMallocHost()

- Both allocate memory on **host**
  - cudaMalloc() allocate memory on device
- Page-locked
  - cudaMallocHost()pins memory by default
  - cudaHostAlloc()has to use the flag cudaHostAllocMapped to pin the memory
- Page-shared
  - cudaMallocHost() doesn't support
  - cudaHostAlloc()has the flag cudaHostAllocPortable

*cudaHostAlloc()is only supported after CUDA2.2

Outline

- Execution
- Memory
  - Local variable
  - Global variable
  - Shared variable
- Synchronization
- Stream

Shared Variable: Shared Memory

- What is the difference between shared memory and global memory?
  - Data scope?
  - Speed?
  - Size?

Using Shared Memory for Optimization

```
__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];
}
```

Q: How many global memory accesses?

```
extern __shared__ int S[][];
__global__ void FW_APSP(int k, int D[n][n]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    S[i][j]=D[i][j]; // move data to shared memory
    __syncthreads();
    // do computation
    if (S[i][j]>S[i][k]+S[k][j])
       D[i][j]= S[i][k]+S[k][j];
}
```

Using Shared Memory for Optimization

```
__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];
}
```

Q: Why only move 1 element, and need sync?

```
extern __shared__ int S[][];
__global__ void FW_APSP(int k, int D[n][n]) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    S[i][j]=D[i][j]; // move data to shared memory
    __syncthreads();
    // do computation
     if (S[i][j]>S[i][k]+S[k][j])
        D[i][j]= S[i][k]+S[k][j];
}
```

When to Synchronize between Threads

- When there is data dependency between threads, __syncthreads() is required

```
__global__ void FW_APSP(int* A, int* B) {
    int i = threadIdx;
    __shared__ S[10];
    S[i] = A[i];
    // do computation
    int V = S[i] + S[i+1];
    S[10-i] = V;
    B[i]=V;
}
```

Q: Where to add __syncthreads()?

When to Synchronize between Threads

- When there is data dependency between threads, __syncthreads() is required

```
__global__ void FW_APSP(int* A, int* B) {
    int i = threadIdx;
    __shared__ S[10];
    S[i] = A[i];
    __syncthreads();
   // do computation
   int V = S[i] + S[i+1];
    __syncthreads();
    S[10-i] = V;
    B[i]=V;
}
```

Using Shared Memory for Optimization

```
__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];
}
```

Q: Why declare shared variable using extern?

```
extern __shared__ int S[][];
__global__ void FW_APSP(int k, int D[n][n]) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    S[i][j]=D[i][j]; // move data to shared memory
    __syncthreads();
    // do computation
     if (S[i][j]>S[i][k]+S[k][j])
        D[i][j]= S[i][k]+S[k][j];
}
```

Dynamic Allocate Shared Memory

- Use extern variable, and 3rd kernel launch argument
- Limitation: All shared variables point to the same memory buffer address

```
extern __shared__ int S[];
__global__ void FW_APSP(int* k, int* D, int* n) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    S[i*(*n)+j]=D[i*(*n)+j]; //move data to shared memory
    __syncthreads();
    // do computation
     if (S[i*(*n)+j]>S[i*(*n)+k]+S[k*(*n)+j])
         D[i*(*n)+j]= S[i*(*n)+k]+S[k*(*n)+j];
}
void main(){
    FW_APSP<<<1,n*n, n*n*sizeof(int)>>>(…);
}
```

Q: Why the massy index for shared variable?

Address Issue of Dynamic Allocation

- If you have multiple extern declaration of shared:

```
extern __shared__ float As[];
extern __shared__ float Bs[];
```

  this will lead to As pointing to the same address as Bs.

- Solution: keep `As` and `Bs` inside the 1D-array.

```
extern __shared__ float smem[];
```

- Need to do the memory management yourself
  - When calling kernel, launch it with size of `sAs+sBs`, where `sAs` and `sBs` are the size of `As` and `Bs` respectively.
  - When indexing elements in `As`, use `smem[0:sAs-1];` when indexing elements in `Bs`, use `smem[sAs:sAs+sBs]`.

Shared memory address translation

- `Mul<<<1, N, N>>>(A, B, C, N); //C=A*B`
  - The third parameter is the size of shared memory.

```
extern __shared__ int S[];
inline int Addr(int matrixIdx, int i, int N)  {
    return (N*matrixIdx + i);
}
__global__ void Mul(int* A, int* B,int* C, int* N) {
    int i = threadIdx.x;
    //move data to shared memory
    S[Addr(0, i)]=A[i];
    S[Addr(1, i)]=B[i];
    __syncthreads();
    // do computation
    C[i]=S[Addr(0, i)]*S[Addr(1, i)];
}
```

Static Shared Memory Allocation

- Mul<<<1, N>>>(A, B, C); //C=A*B
  - The third parameter is the size of shared memory.

```
__global__ void Mul(int* A, int* B,int* C) {
    int N=10;
    int i = threadIdx.x;
    __static__ int SA[10]; //static allocation
    __static__ int SB[10]; //static allocation
    //move data to shared memory
    SA[i]=A[i];
    SB[i]=B[i];
    __syncthreads();
    // do computation
     C[i]=SA[i]*SB[i];
}
```

Summary of CUDA Variables in Kernel

Static Variable declaration	Memory	Scope	Lifetime	Speed	Total # of variables	Visible by # of threads
`int var`	Register	Thread	Thread	1x	100,000	1
`int array_var[10]`	Local	Thread	Thread	100x	100,000	1

`__shared__`
`int shared_var`
	Shared	Block	Block	1x	100	100

`__device__`
`int global_var`
	Global	Grid	App	100x	1	100,000

`__constant__`
`int constant_var`
	Constant	Grid	App	1x	1	100,000

Outline

- Execution
- Memory
- Synchronization
- Stream

Asynchronous Functions

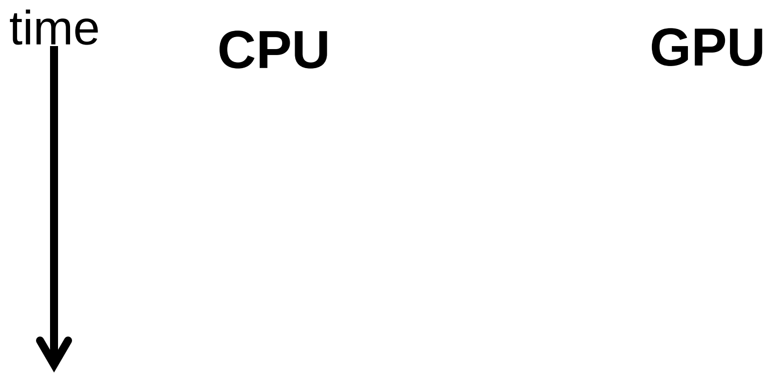
- What does asynchronous function mean in CUDA programming?

Asynchronous Functions

- What does asynchronous function mean in CUDA programming?
  - Functions that can facilitate concurrent execution between **host** and **device**
  - Control is returned to the host thread **before the device has completed the requested task**

Prog.

```
void main()  {
   kernel1<<1,1>>(); // need  10s
   kerne2<<1,1>>(); // need 10s
   cpu_method();  // need 20s
}
```

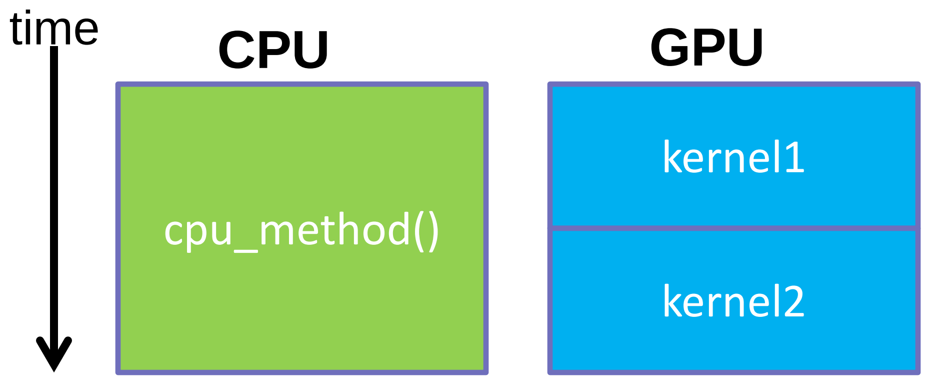


Asynchronous Functions

- What does asynchronous function mean in CUDA programming?
  - Functions that can facilitate concurrent execution between **host** and **device**
  - Control is returned to the host thread **before the device has completed the requested task**

Prog.

```
void main()  {
    kernel1<<1,1>>(); // need  10s
    kerne2<<1,1>>(); // need 10s
    cpu_method();  // need 20s
}
```



Asynchronous Functions

- What are the asynchronous functions?
  - kernel launch?
  - cudaMemcpy?
  - cudaMemcpyAsync?
  - cudaMalloc?
  - cudaFree?
  - cudaEventRecord?
  - cudaEventElapsedTime?

Asynchronous Functions

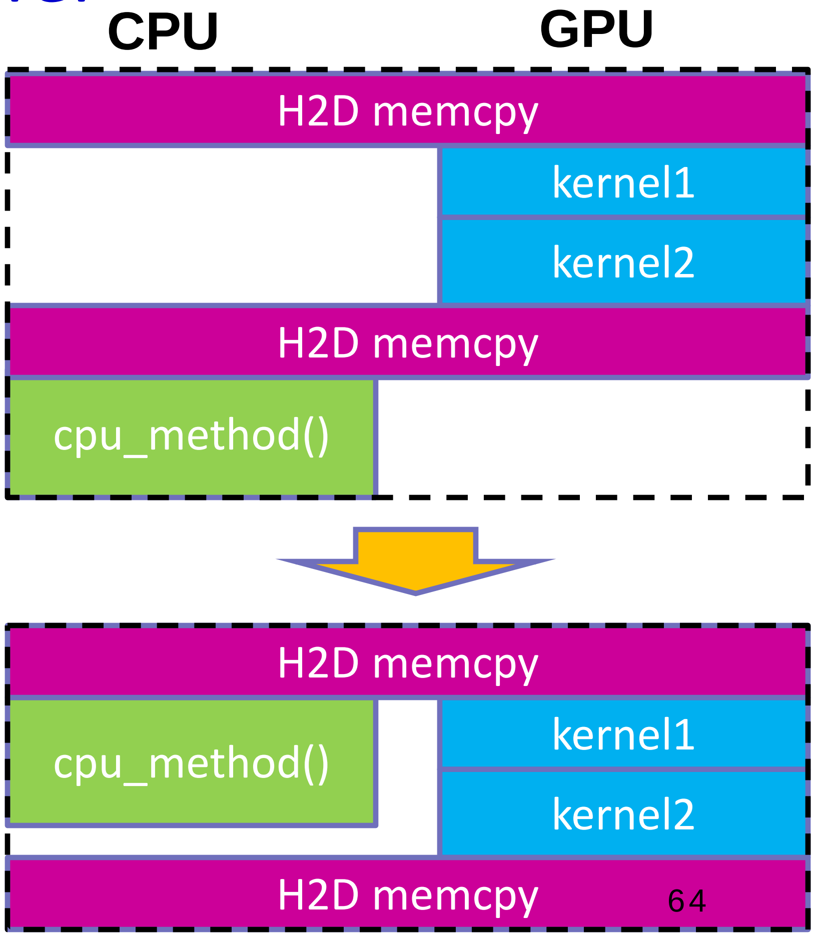
- What are the asynchronous functions?
  - kernel launch? Yes
  - cudaMemcpy? No
  - cudaMemcpyAsync? Yes
  - cudaMalloc? No
  - cudaFree? No
  - cudaEventRecord? Yes
  - cudaEventElapsedTime? Yes

- Check CUDA API reference.
- Think about whether the operations need to involve both CPU and GPU cooperation together.

Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

```
void main()  {
 cudaMemcpy ( /**/, H2D ) ;
 kernel2 <<< grid, block>>> () ;
 kernel3 <<< grid, block>>> () ;
 cudaMemcpy ( /**/, D2H ) ;
 cpu_method();
}
```

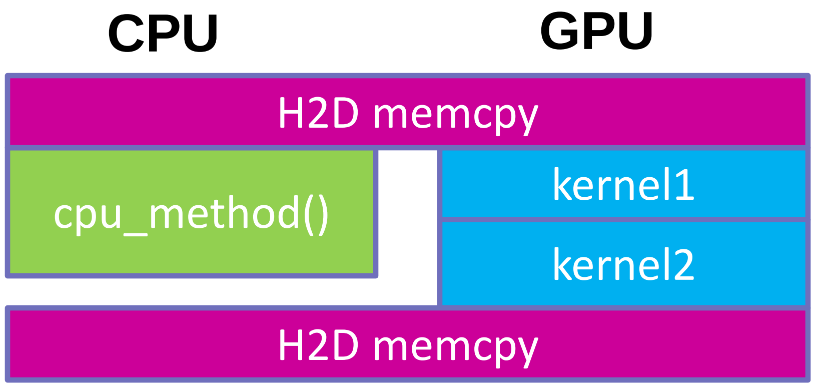


Q: How to overlap the time?

Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

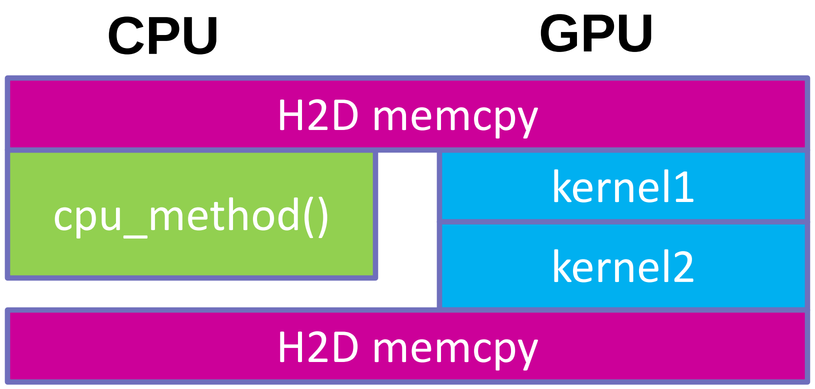
```
void main()  {
 cudaMemcpy ( /**/, H2D ) ;
 kernel2 <<< grid, block>>> () ;
 kernel3 <<< grid, block>>> () ;
 cudaMemcpyAsync ( /**/, D2H ) ;
 cpu_method();
}
```



Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

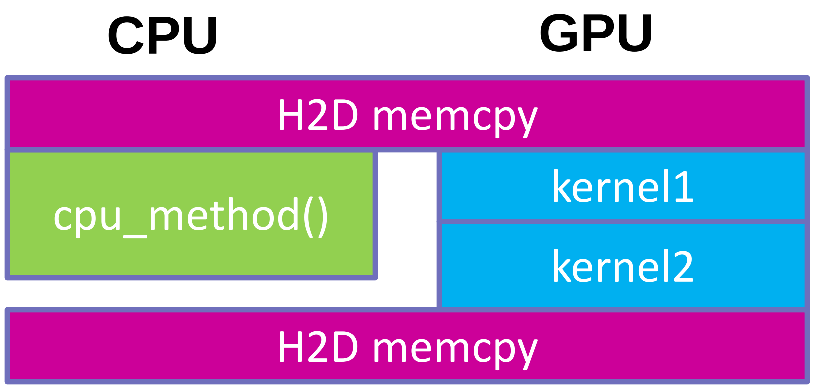
```
void main()  {
 cudaMemcpy ( /**/, H2D ) ;
 kernel2 <<< grid, block>>> () ;
 kernel3 <<< grid, block>>> () ;
 cpu_method();
 cudaMemcpy ( /**/, D2H ) ;
}
```



Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

```
void main()  {
 cudaMemcpy ( /**/, H2D ) ;
 kernel2 <<< grid, block>>> () ;
 cpu_method();
 kernel3 <<< grid, block>>> () ;
 cudaMemcpy ( /**/, D2H ) ;
}
```



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);
}
```

Q: Is it correct? What’s wrong?

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);
}
```

Q: Is it correct? What’s wrong?
Ans: “h_a” has data dependency

Risk of Using Asynchronous Functions

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

```
void main()  {
    cudaEventRecord(start);
    kernel<<<block, thread>>>();
    cudaEventRecord(stop);
    cudaEventElapsedTime(&time, start, stop);
}
```

Q: Is it correct? What's wrong?

Risk of Using Asynchronous Functions

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

```
void main()  {
    cudaEventRecord(start);
    kernel<<<block, thread>>>();
    cudaEventRecord(stop);
    cudaEventElapsedTime(&time, start, stop);
}
```

Q: Is it correct? What’s wrong?
Ans: “stop” has data dependency

How to Synchronize 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

How to Synchronize CPU & GPU?

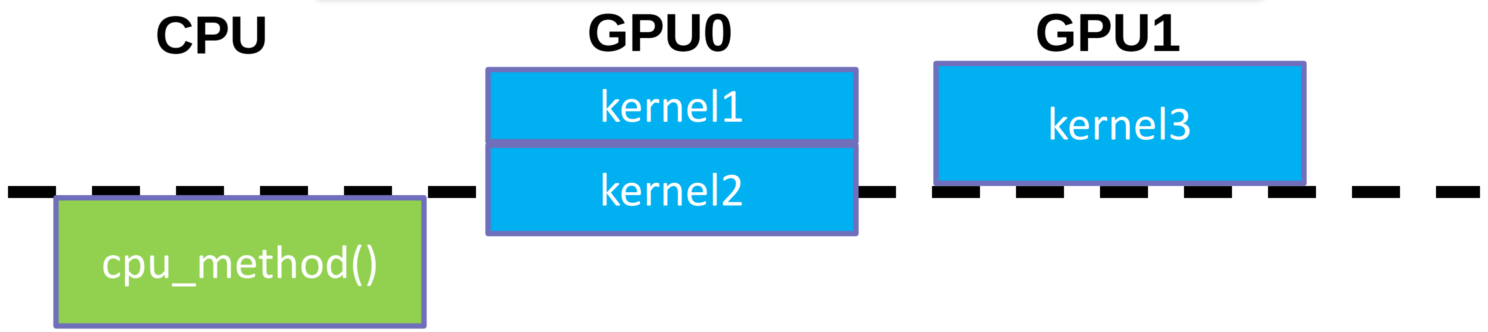
```
void main()  {
 cudaSetDevice(0);
 kernel1 <<< grid, block>>> () ;
 kernel2 <<< grid, block>>> () ;
 cudaSetDevice(1);
 kernel3 <<< grid, block>>> () ;
 cudaDeviceSynchronize()
 cpu_method();
}
```

Q: What happens?

How to Synchronize CPU & GPU?

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

Q: What happens?



How to Synchronize CPU & GPU?

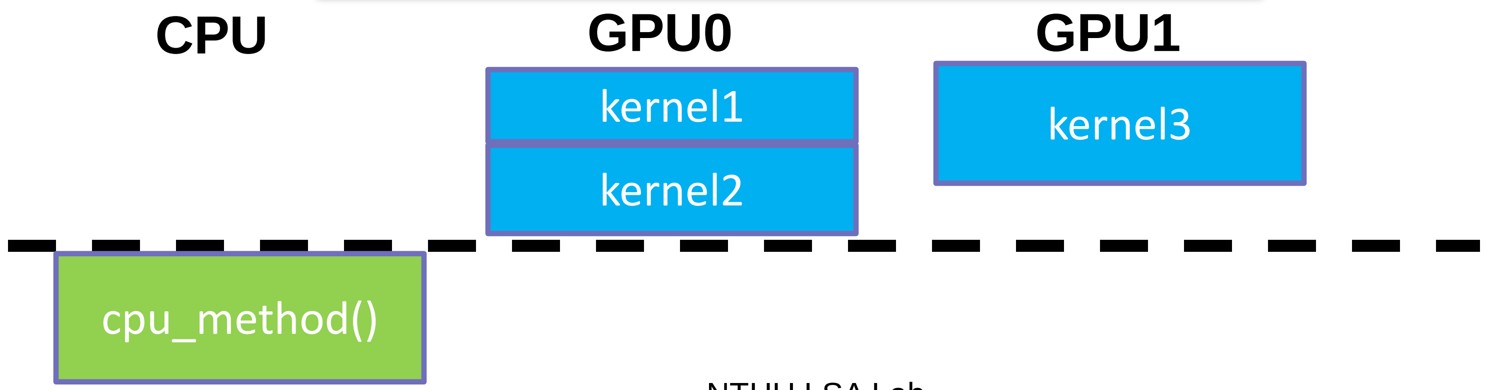
```
void main()  {
 cudaSetDevice(0);
 kernel1 <<< grid, block>>> () ;
 kernel2 <<< grid, block>>> () ;
 cudaSetDevice(1);
 kernel3 <<< grid, block>>> () ;
 cudaThreadSynchronize()
 cpu_method();
}
```

Q: What happens?

How to Synchronize CPU & GPU?

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

Q: What happens?



How to Synchronize CPU & GPU?

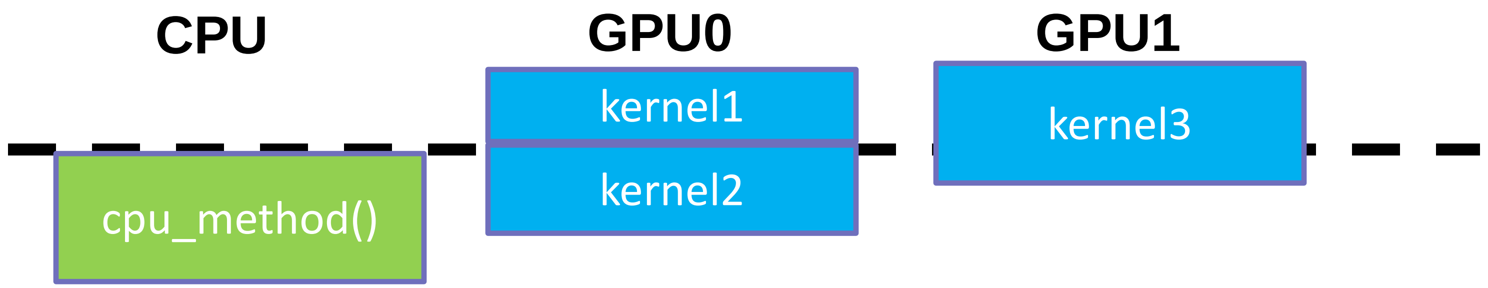
```
void main()  {
 cudaSetDevice(0);
 kernel1 <<< grid, block>>> () ;
 cudaEventRecrod(event)
 kernel2 <<< grid, block>>> () ;
 cudaSetDevice(1);
 kernel3 <<< grid, block>>> () ;
  cudaEventSynchronize (event)
 cpu_method();
}
```

Q: What happens?

How to Synchronize CPU & GPU?

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

Q: What happens?



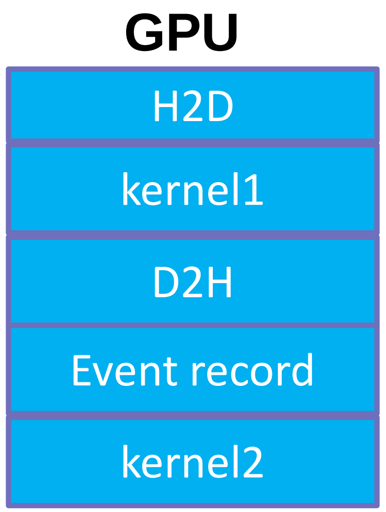
Outline

- Execution
- Memory
- Synchronization
- Stream

CUDA Function Execution on GPU

- All the CUDA functions (**regardless asynchronous or not**) issued from the **same CPU thread** on the **same device** is serialized and executed in order

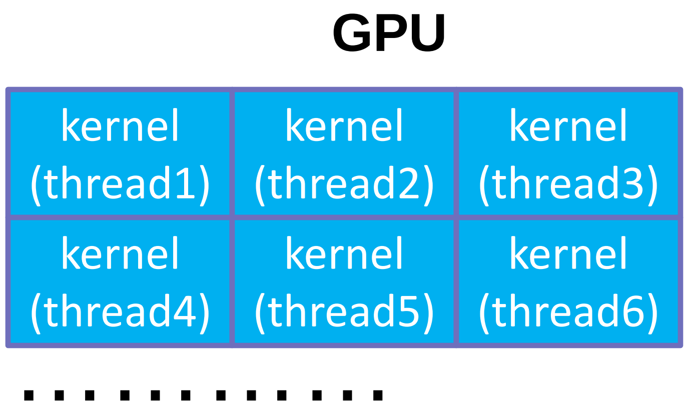
```
void main()  {
 cudaMemcpyAsync (/**/, H2D) ;
 kernel1 <<< grid, block>>> () ;
 cpu_method();
 cudaMemcpyAsync (/**/, D2H) ;
 cudaEventRecrod(event)
 kernel2 <<< grid, block>>> () ;
}
```



CUDA Function Execution on GPU

- CUDA functions from **different CPU threads/processes** can always be **executed in parallel as long as the device has sufficient resources**

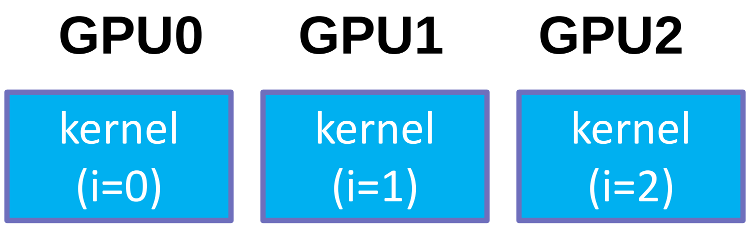
```
#pragma omp parallel
{
   kernel <<< grid, block>>> () ;
}
```



CUDA Function Execution on GPU

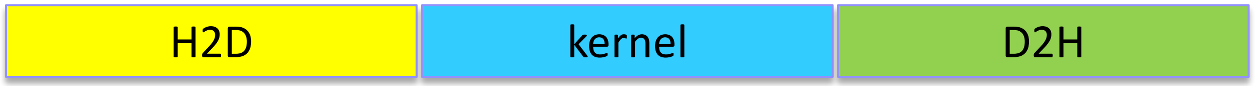
- CUDA functions issued from the **same CPU thread**, but on **different devices** can be executed at the same time

```
for(int i=0; i<3; i++) {
  cudaSetDevice(i);
  kernel <<< grid, block>>> ();
}
```

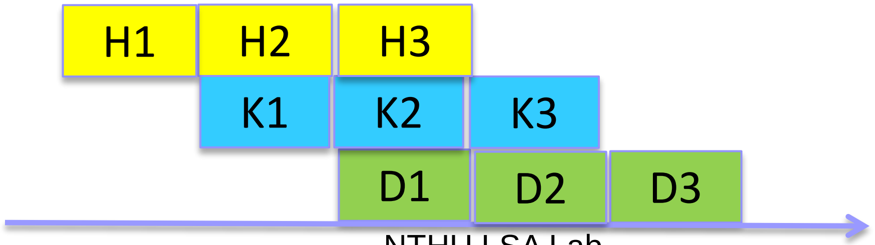


CUDA Streams

- CUDA Stream is a technique to **overlap the execution of a kernel**, and **hide data transfer delay from computations**
  - Operations in different streams can be interleaved and, when possible, they can even run concurrently
  - **Operations in the same stream are still serialized and executed in order**
- Consider a kernel process a huge dataset
  - Without stream, the kernel computation can only start after the dataset is transferred



  - With stream, we can partition the dataset, assign each partition to a stream, and execute them in a pipeline



CUDA Streams

- kernel launch
  - `kernel<<<grid,block,0,stream-id>>>(/*…*/);`
- Stream-id must be allocated and destroyed
  - `cudaStream_t *stream;`
  - `cudaStreamCreate(&stream);`
  - `cudaStreamDestroy(stream);`
- Memory copy can be either synchronous or asynchronous. **But synchronous memcpy prevents streams from running in parallel**
- If asynchronous copy is used, host memory must be pinned

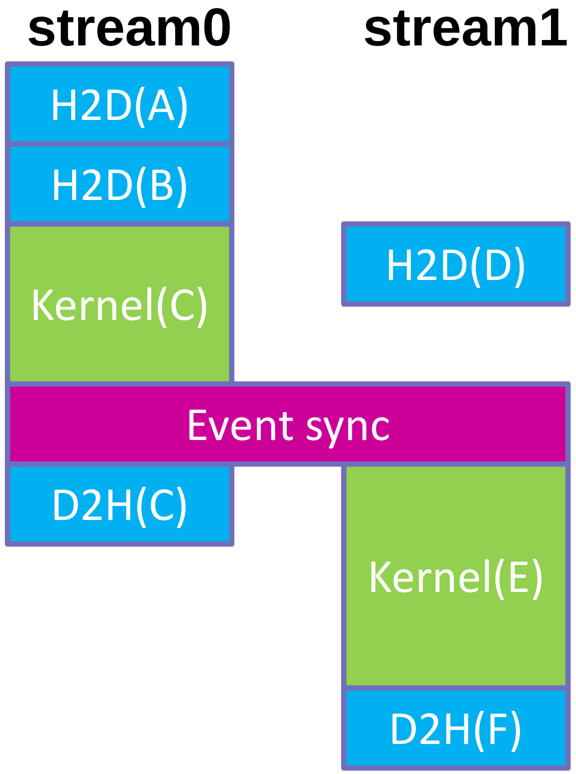
CUDA Streams

```
cudaStream_t stream[2];
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);
cudaMallocHost(&hostPtr, 2 * size); // pined(page
locked mem)
for (int i = 0; i < 2; ++i) {
   cudaMemcpyAsync(/*…*/,          // async memcpy
      cudaMemcpyHostToDevice, stream[i]);
   kernel<<<100,512,0,stream[i]>>>(/*…*/);
   cudaMemcpyAsync(/*…*/,
      cudaMemcpyDeviceToHost, stream[i]);
}
cudaStreamDestroy(stream[0]);
cudaStreamDestroy(stream[1]);
```

CUDA Stream Synchronization

- Operations in different streams can be interleaved and, when possible, they can even run concurrently
- **Operations in the same stream are still serialized and executed in order**

```
cudaMemcpyAsync(d_A, A, H2D, stream[0]);
cudaMemcpyAsync(d_B, B, H2D, stream[0]);
kernel<<<n,m,0,stream[0]>>>(d_A, d_B, d_C);
cudaEventRecord(event, stream[0]);
cudaMemcpyAsync(C, d_C, D2H, stream[0]);
cudaMemcpyAsync(d_D, D, H2D, stream[1]);
cudaStreamWaitEvent ( stream[1], event );
kernel<<<n,m,0,stream[1]>>>(d_C, d_D, d_E);
cudaMemcpyAsync(E, d_E, D2H, stream[1]);
cudaDeviceSynchronize()
print(E);
```


stream0
stream1
H2D(A)
H2D(B)
H2D(D)
Kernel(C)
Event sync
D2H(C)
Kernel(E)
D2H(F)