

CS 380 - GPU and GPGPU Programming

Lecture 5: GPU Architecture, Pt. 3

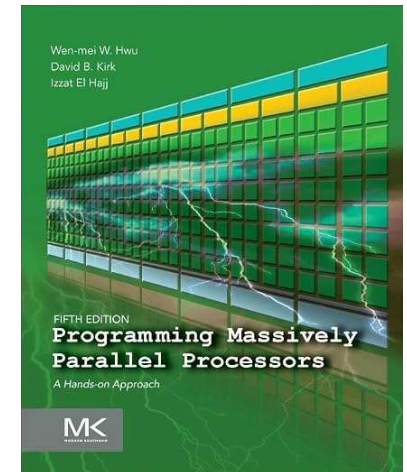
Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 21)



Read (required):

- Programming Massively Parallel Processors book, 5th edition, **Chapter 2** (*Heterogeneous data-parallel computing*)
- Programming Massively Parallel Processors book, 5th edition, **Chapter 4** (*Compute architecture and scheduling*) **until 4.3**
- NVIDIA CUDA C++ Programming Guide
(current: v13.4, Sep 9, 2026)



[https://docs.nvidia.com/cuda/cuda-programming-guide/pdf/
cuda-programming-guide.pdf](https://docs.nvidia.com/cuda/cuda-programming-guide/pdf/cuda-programming-guide.pdf)

Read **Chapter 1.3.1** (*Compute Capability and Streaming Multiprocessor Versions*),
Chapter 5.1 (Compute Capabilities)

- Look at NVIDIA CUDA C++ Best Practices Guide
(current: v13.4, Sep 9, 2026)

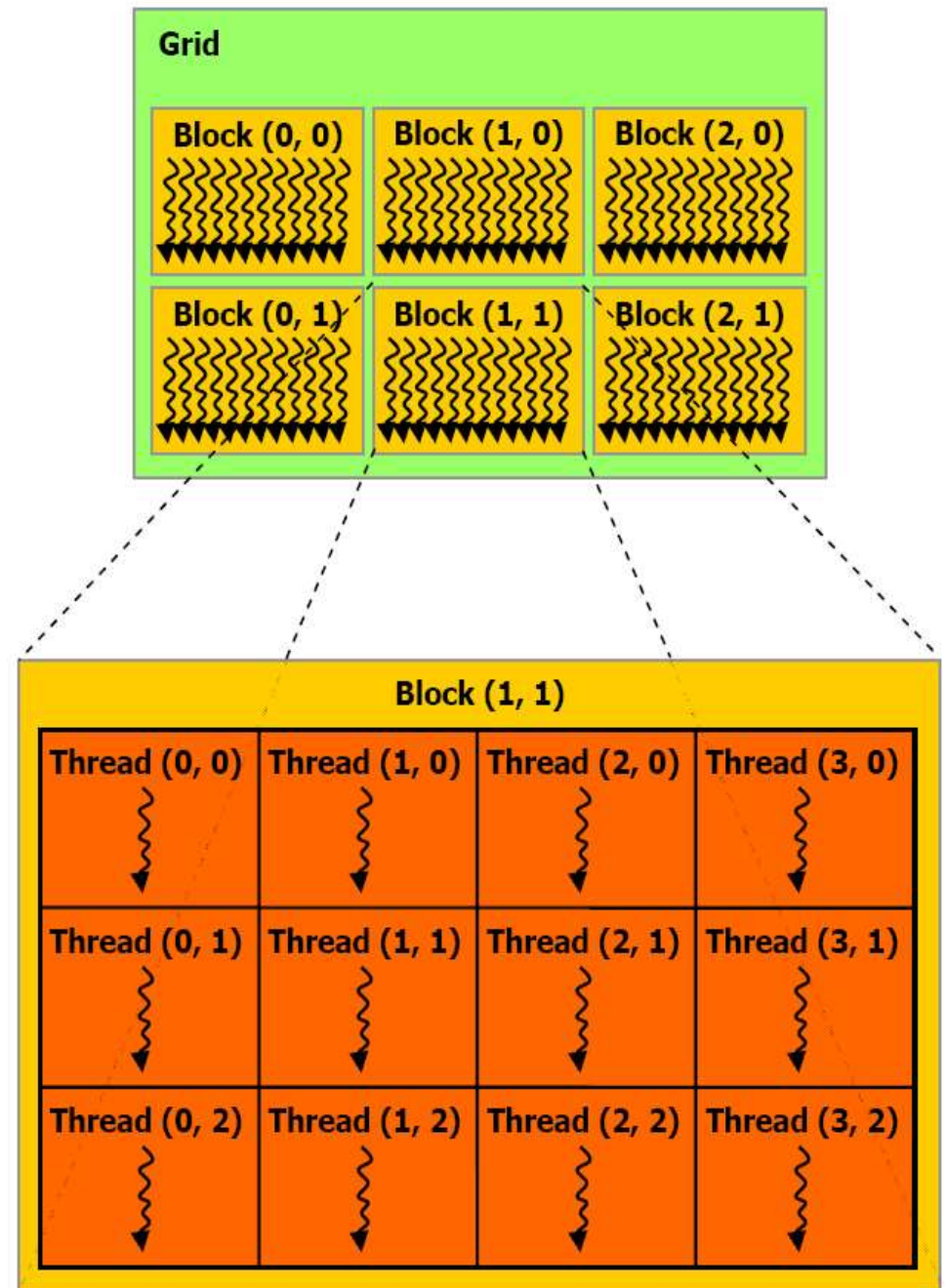
https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

CUDA Multi-Threading

CUDA model groups threads into **thread blocks**; blocks into **grid**

Execution on actual hardware:

- Thread blocks assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
- 32 threads grouped into a **warp** (on all compute capabilities)

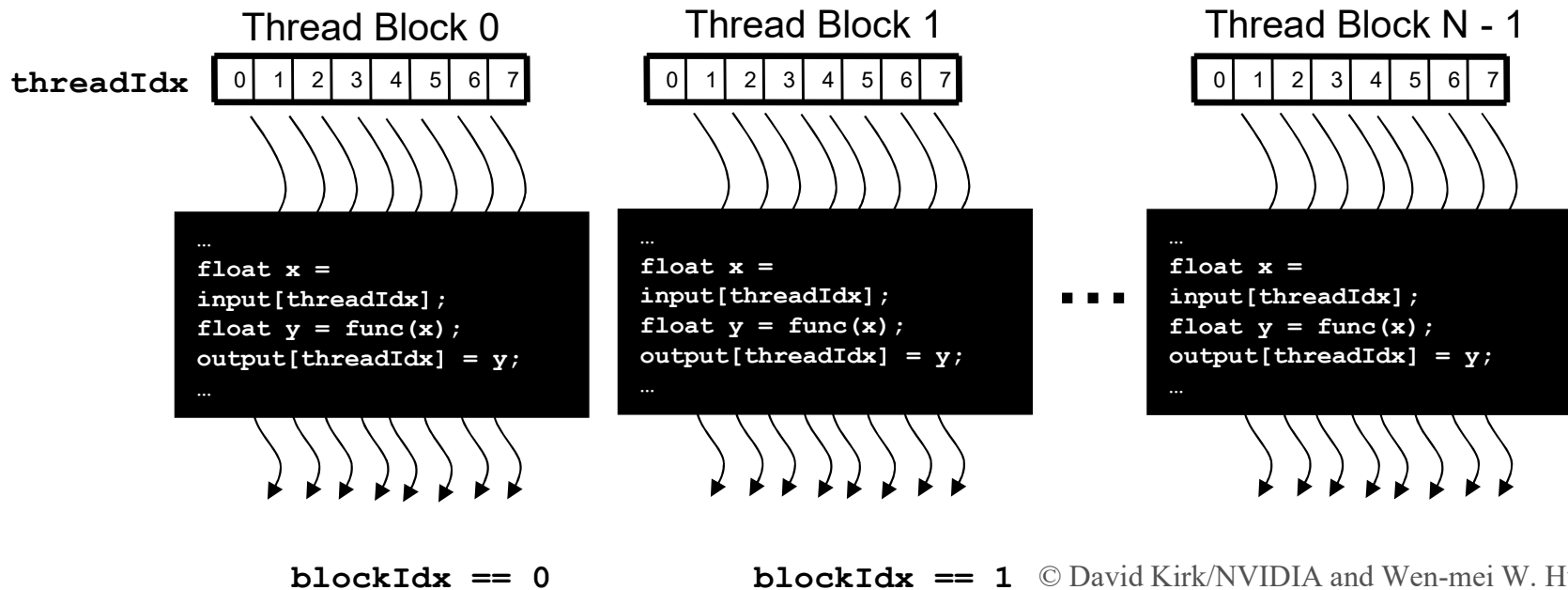


Threads in Block, Blocks in Grid



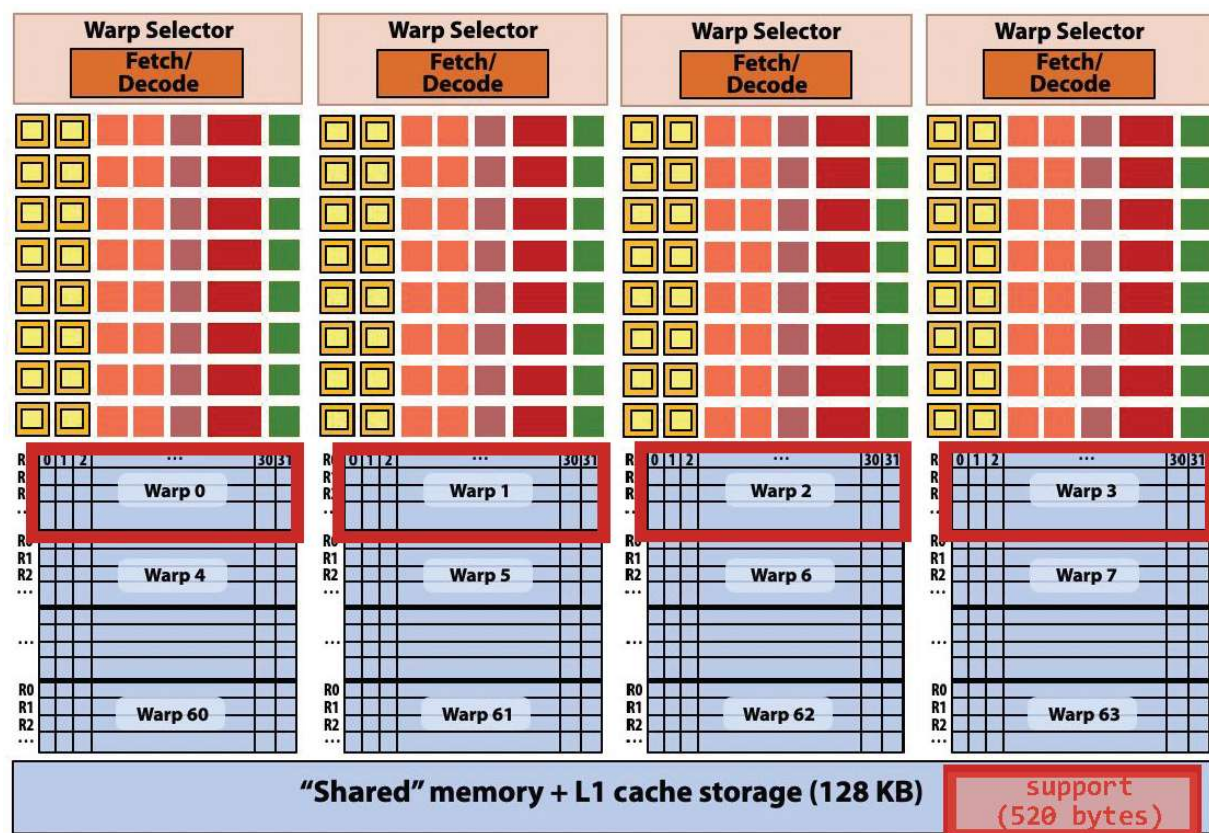
- Identify work of thread via

- `threadIdx`
- `blockIdx`



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

Teaser: Typical CUDA Kernel (SM Perspective)



A convolve thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

SM core operation each clock:

- Each sub-core selects one runnable warp (from the 16 warps in its partition)
- Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                        float* output)
{
    __shared__ float support[THREADS_PER_BLK+2];
    int index = blockIdx.x * blockDim.x +
                threadIdx.x;

    support[threadIdx.x] = input[index];
    if (threadIdx.x < 2) {
        support[THREADS_PER_BLK+threadIdx.x]
            = input[index+THREADS_PER_BLK];
    }

    __syncthreads();

    float result = 0.0f; // thread-local
    for (int i=0; i<3; i++)
        result += support[threadIdx.x + i];

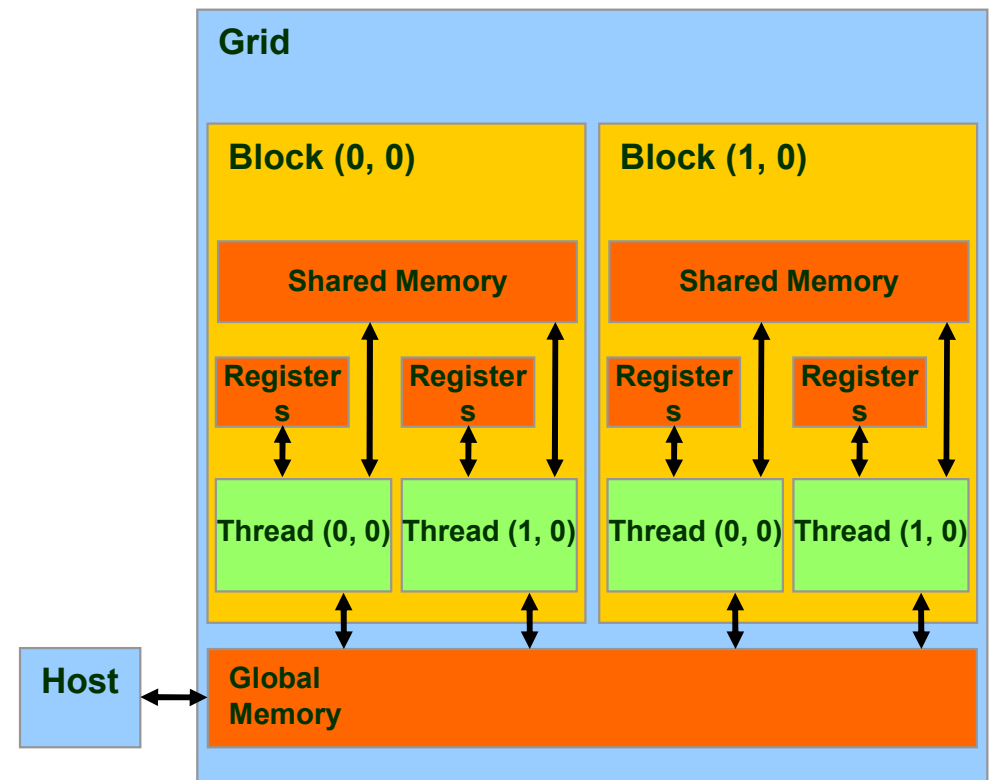
    output[index] = result / 3.f;
}
```

(sub-core == SM partition)

CUDA Memory Model and Usage



- `cudaMalloc()` , `cudaFree()`
- `cudaMallocArray()` ,
`cudaMalloc2DArray()` ,
`cudaMalloc3DArray()`
- `cudaMemcpy()`
- `cudaMemcpyArray()`
- Host ↔ host
Host ↔ device
Device ↔ device
- Asynchronous transfers possible (DMA)



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as **kernels**
 - One **kernel** is executed at a time
 - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
 - CUDA threads are extremely lightweight
 - Very little creation overhead
 - Instant switching
 - CUDA uses 1000s of threads to achieve efficiency
 - Multi-core CPUs can use only a few

Definitions

Device = GPU

Host = CPU

Kernel = function that runs on the device

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
 - All threads run the same code
 - Each thread has an ID that it uses to compute memory addresses and make control decisions

threadID

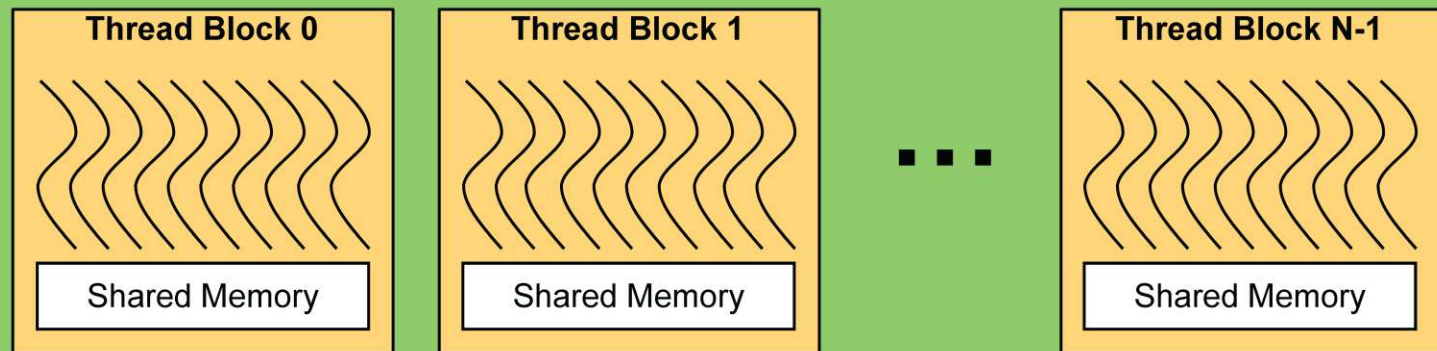
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

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

Thread Batching

- Kernel launches a **grid** of **thread blocks**
 - Threads within a block cooperate via shared memory
 - Threads within a block can synchronize
 - Threads in different blocks cannot cooperate*
- Allows programs to *transparently scale* to different GPUs

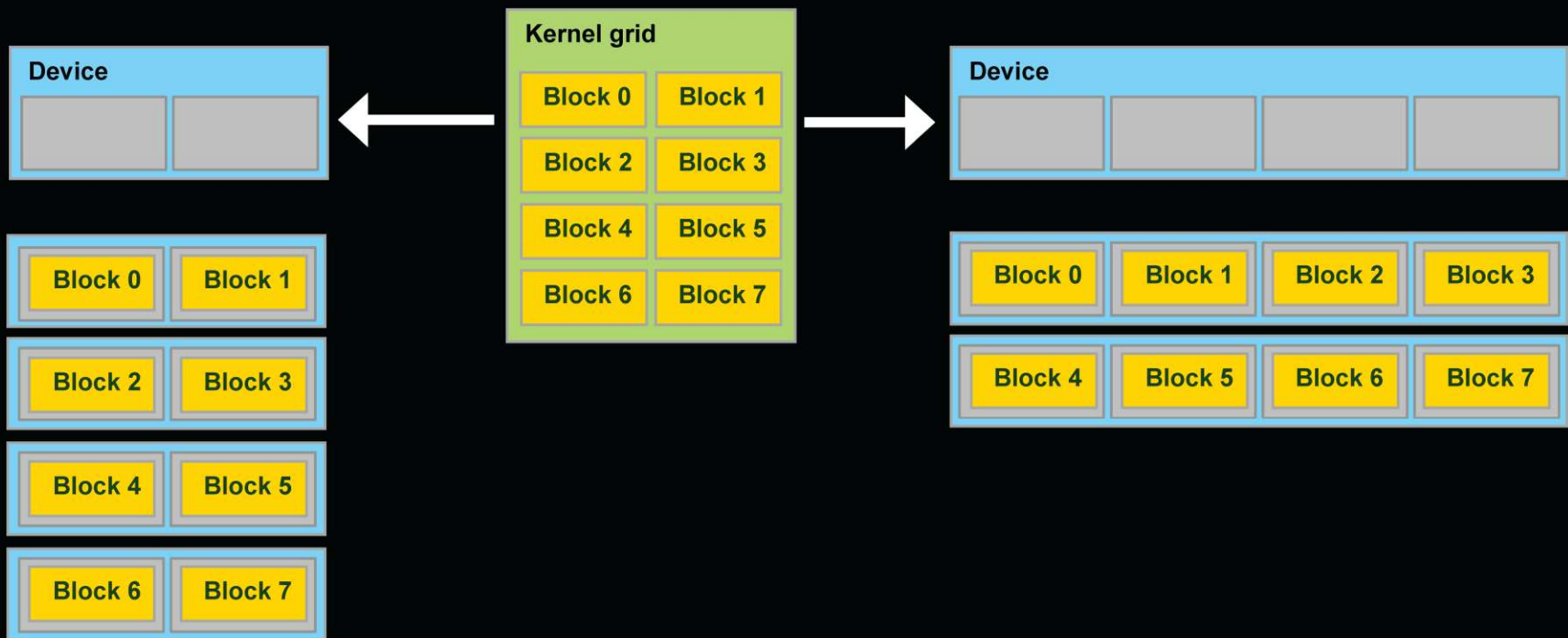
Grid



* brand new on Hopper: thread block clusters

Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
- A kernel scales across parallel multiprocessors



Execution Model

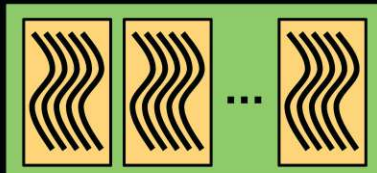
Software



Thread



Thread Block



Grid

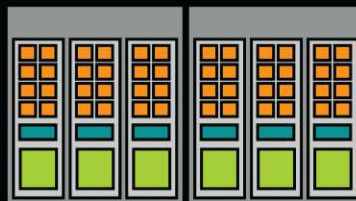
Hardware



Thread Processor



Multiprocessor



Device

Threads are executed by thread processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

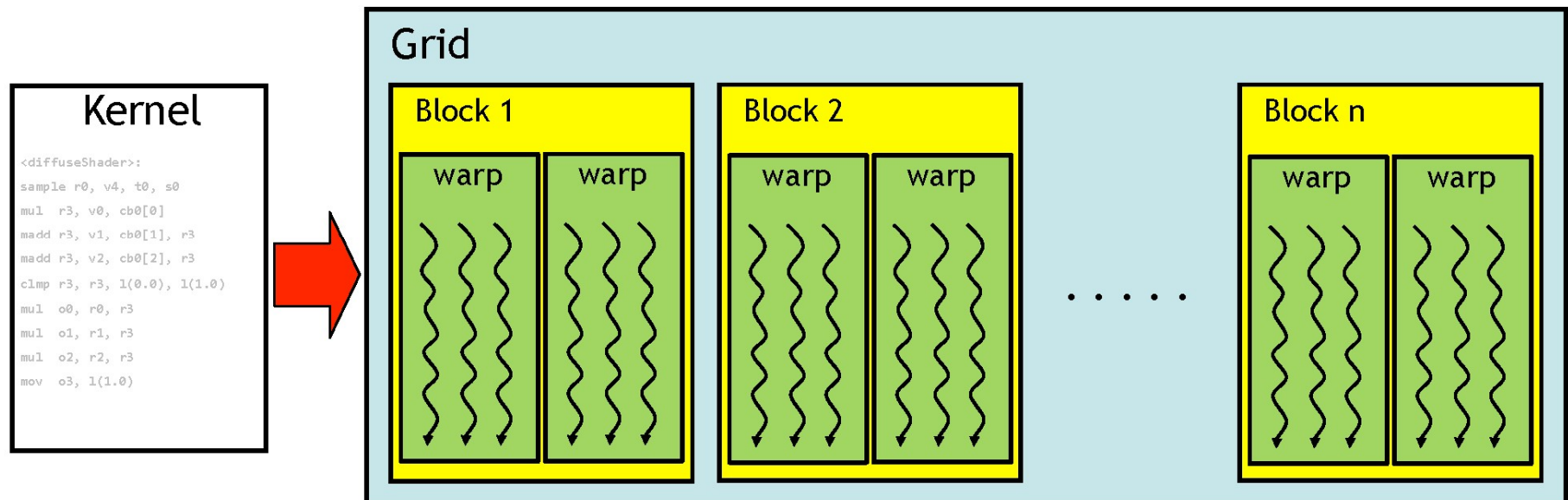
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

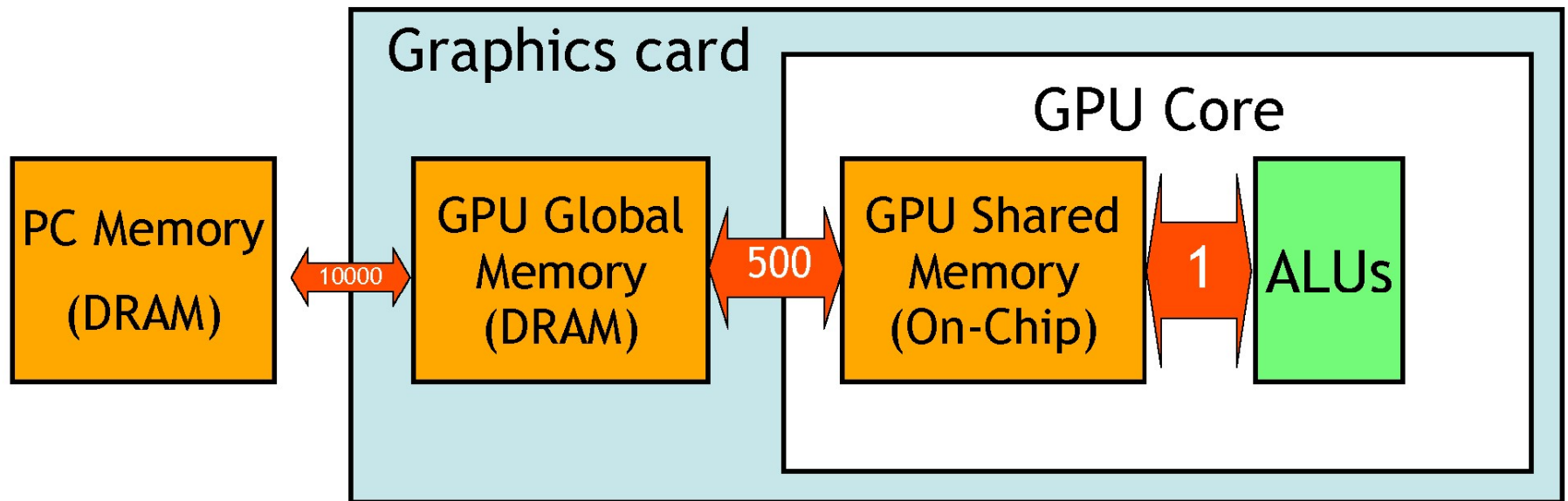
CUDA Programming Model

- Kernel
 - GPU program that runs on a thread grid
- Thread hierarchy
 - Grid : a set of blocks
 - Block : a set of warps
 - Warp : a SIMD group of 32 threads
 - Grid size * block size = total # of threads



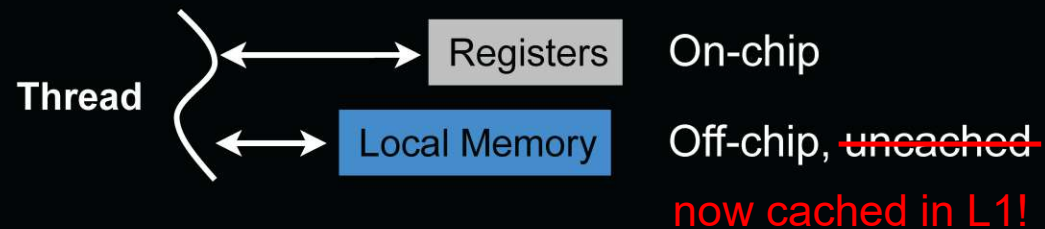
CUDA Memory Structure

- Memory hierarchy
 - PC memory : off-card
 - GPU global : off-chip / on-card
 - GPU shared/register/cache : on-chip
- The host can read/write global memory
- Each thread communicates using shared memory

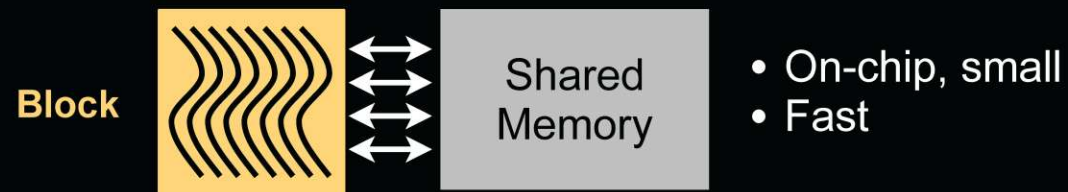


Kernel Memory Access

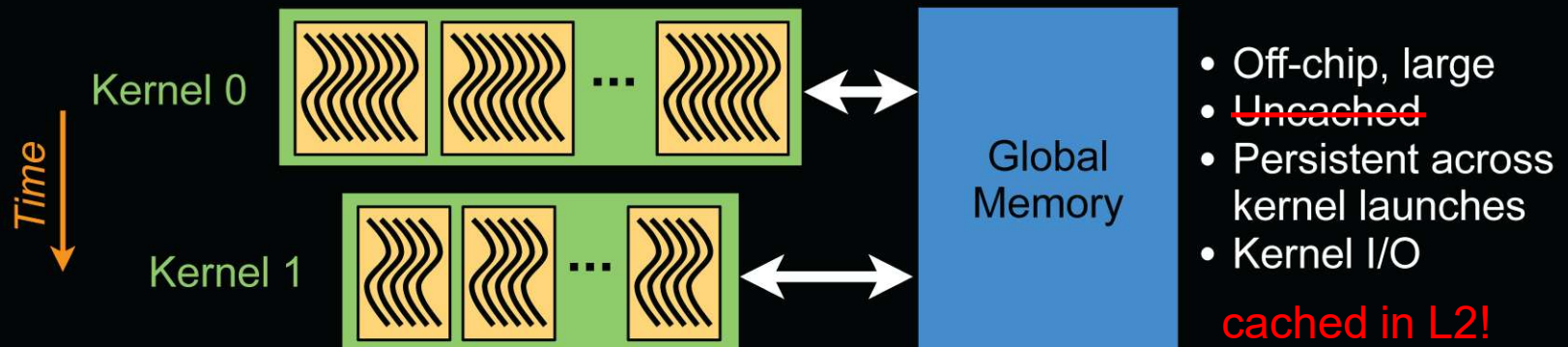
● Per-thread



● Per-block



● Per-device



Memory Architecture



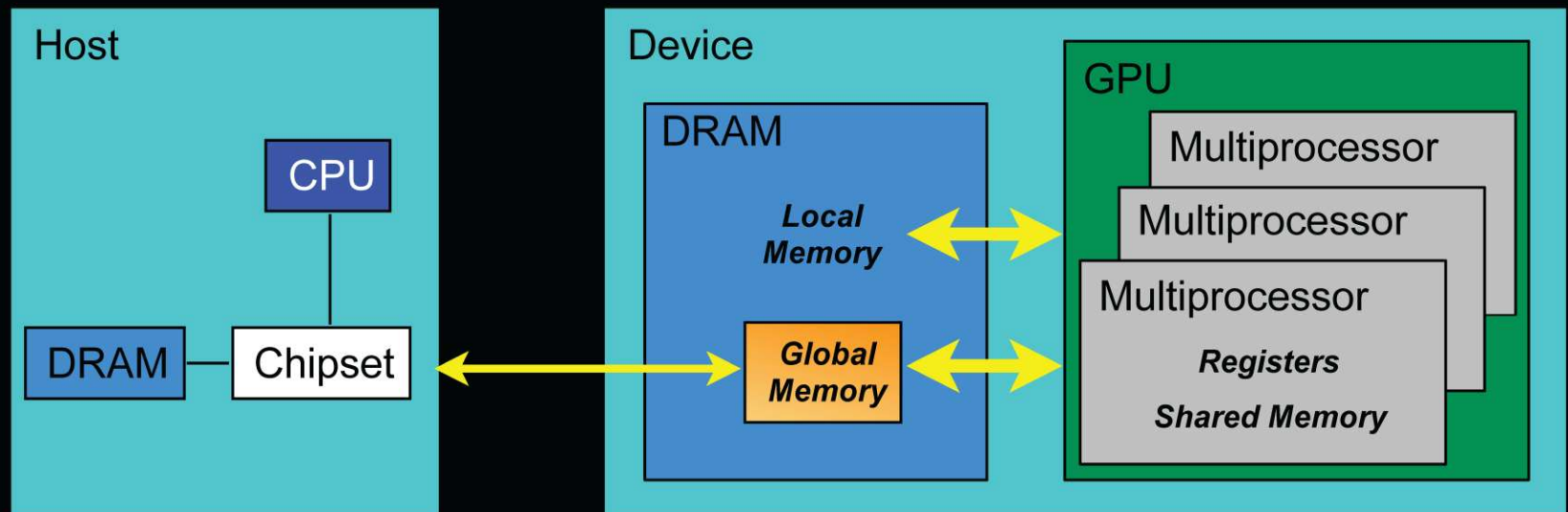
Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No * YES	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No * YES	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

* cached on Fermi or newer!

Managing Memory

Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- **CPU and GPU have separate memory spaces**
- **Host (CPU) code manages device (GPU) memory:**
 - Allocate / free
 - Copy data to and from device
 - Applies to *global* device memory (DRAM)



GPU Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```
int n = 1024;  
int nbytes = 1024*sizeof(int);  
int *a_d = 0;  
cudaMalloc( (void**) &a_d,  nbytes );  
cudaMemset( a_d, 0, nbytes);  
cudaFree(a_d);
```

Data Copies

- **cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);**
 - **direction** specifies locations (host or device) of **src** and **dst**
 - Blocks CPU thread: returns after the copy is complete
 - Doesn't start copying until previous CUDA calls complete
- **enum cudaMemcpyKind**
 - **cudaMemcpyHostToDevice**
 - **cudaMemcpyDeviceToHost**
 - **cudaMemcpyDeviceToDevice**

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Device

a_d

b_d

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Device

a_d

b_d

Data Movement Example

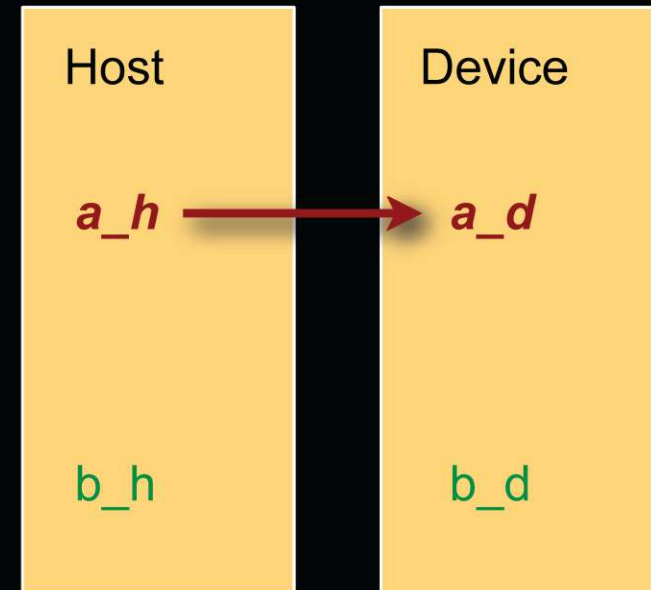
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Device

a_d

b_d



Data Movement Example

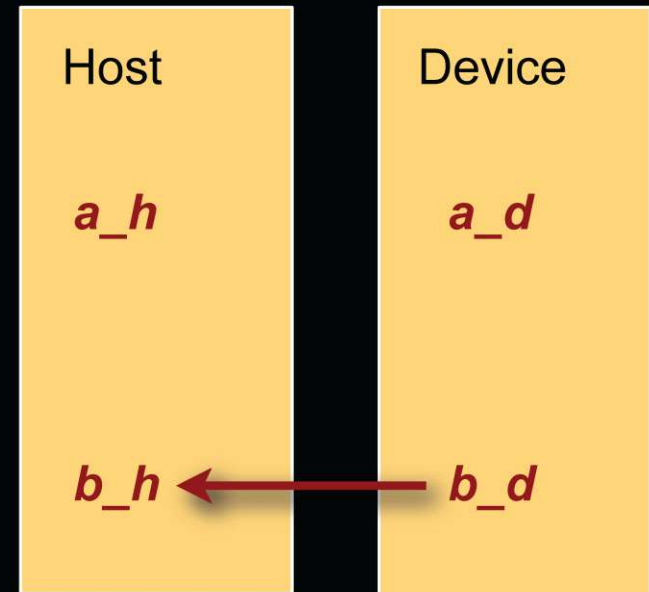
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

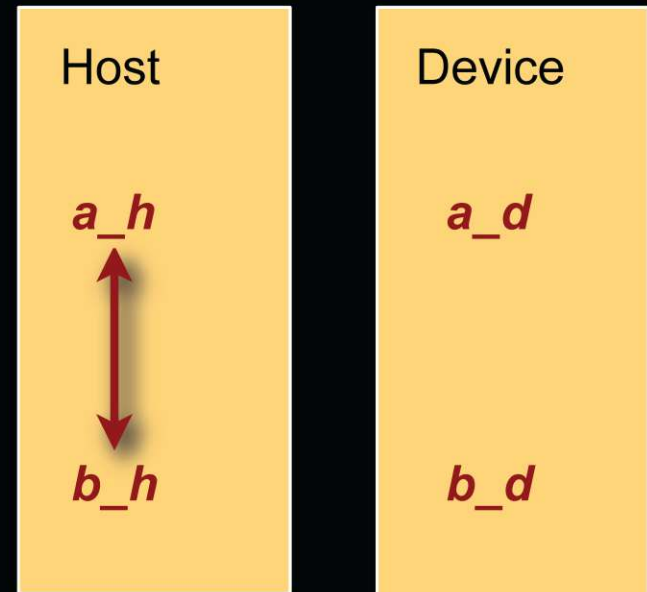
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

Device

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...)
```

- Execution Configuration (“<<< >>>”)

- **dG** - dimension and size of grid in blocks

- Two-dimensional: **x** and **y**
- Blocks launched in the grid: **dG.x * dG.y**

- **dB** - dimension and size of blocks in threads:

- Three-dimensional: **x**, **y**, and **z**
- Threads per block: **dB.x * dB.y * dB.z**

- Unspecified **dim3** fields initialize to 1

Shared Memory Allocation

- 2 modes
- **Static size within kernel**

```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

```
// in main
```

```
int VecSize = MAX_THREADS * sizeof(float4);
```

```
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, ...);
```

```
// declare as extern within kernel
```

```
extern __shared__ float vec[];
```

Thank you.