

CS 380 - GPU and GPGPU Programming

Lecture 11: GPU Compute APIs, Pt. 1

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 13)



Read (required):

- Programming Massively Parallel Processors book (4th edition),
Chapter 3 (*Multidimensional grids and data*)

Read (optional):

- Programming Massively Parallel Processors book (4th edition),
Chapter 20 (*An introduction to CUDA streams*)
- Programming Massively Parallel Processors book (4th edition),
Chapter 21 (*CUDA dynamic parallelism*)

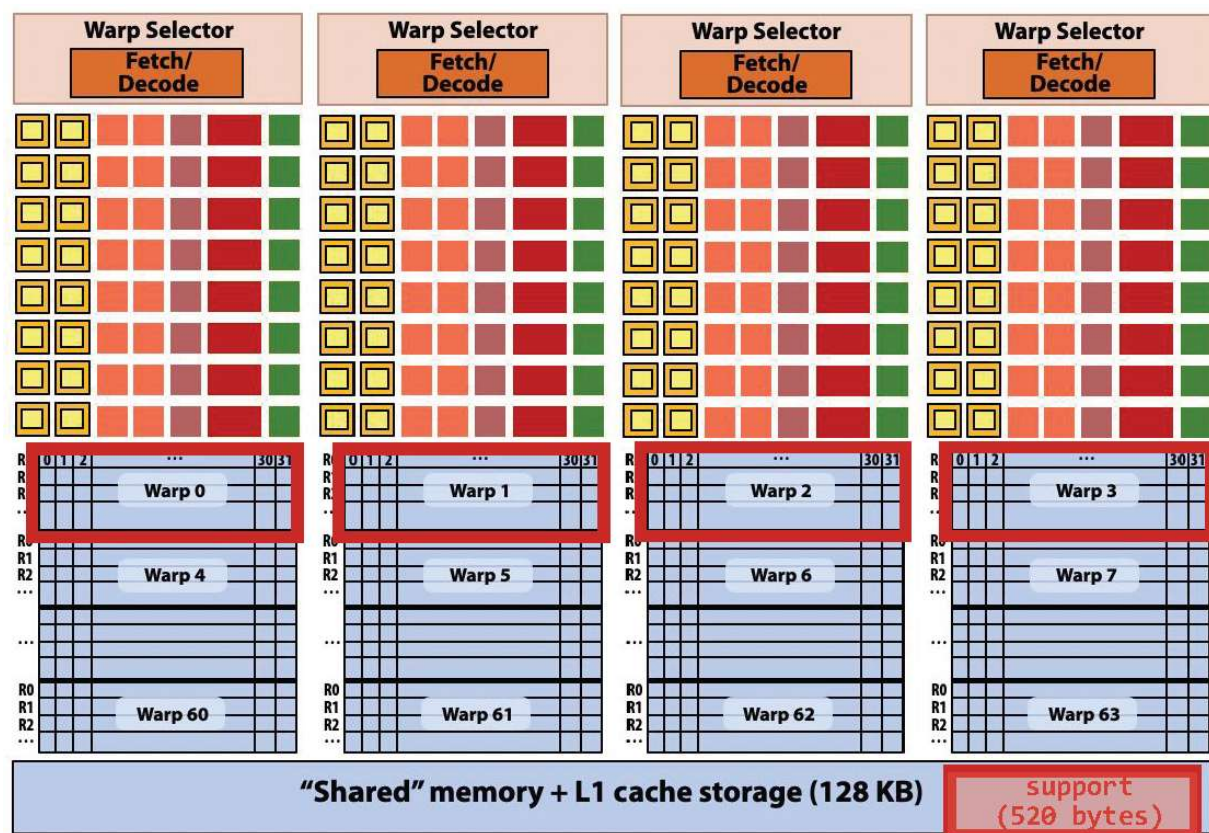
GPU Compute APIs

NVIDIA CUDA



- Old acronym: “Compute Unified Device Architecture”
- Extensions to C(++) programming language
 - `__host__`, `__global__`, and `__device__` functions
 - Heavily multi-threaded
 - Synchronize threads with `__syncthreads()`, ...
 - Atomic functions
(before compute capability 2.0 only integer, from 2.0 on also float)
- Compile `.cu` files with NVCC
- Uses general C compiler (Visual C, gcc, ...)
- Link with CUDA run-time (`cudart.lib`) and cuda core (`cuda.lib`)

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;
}
```

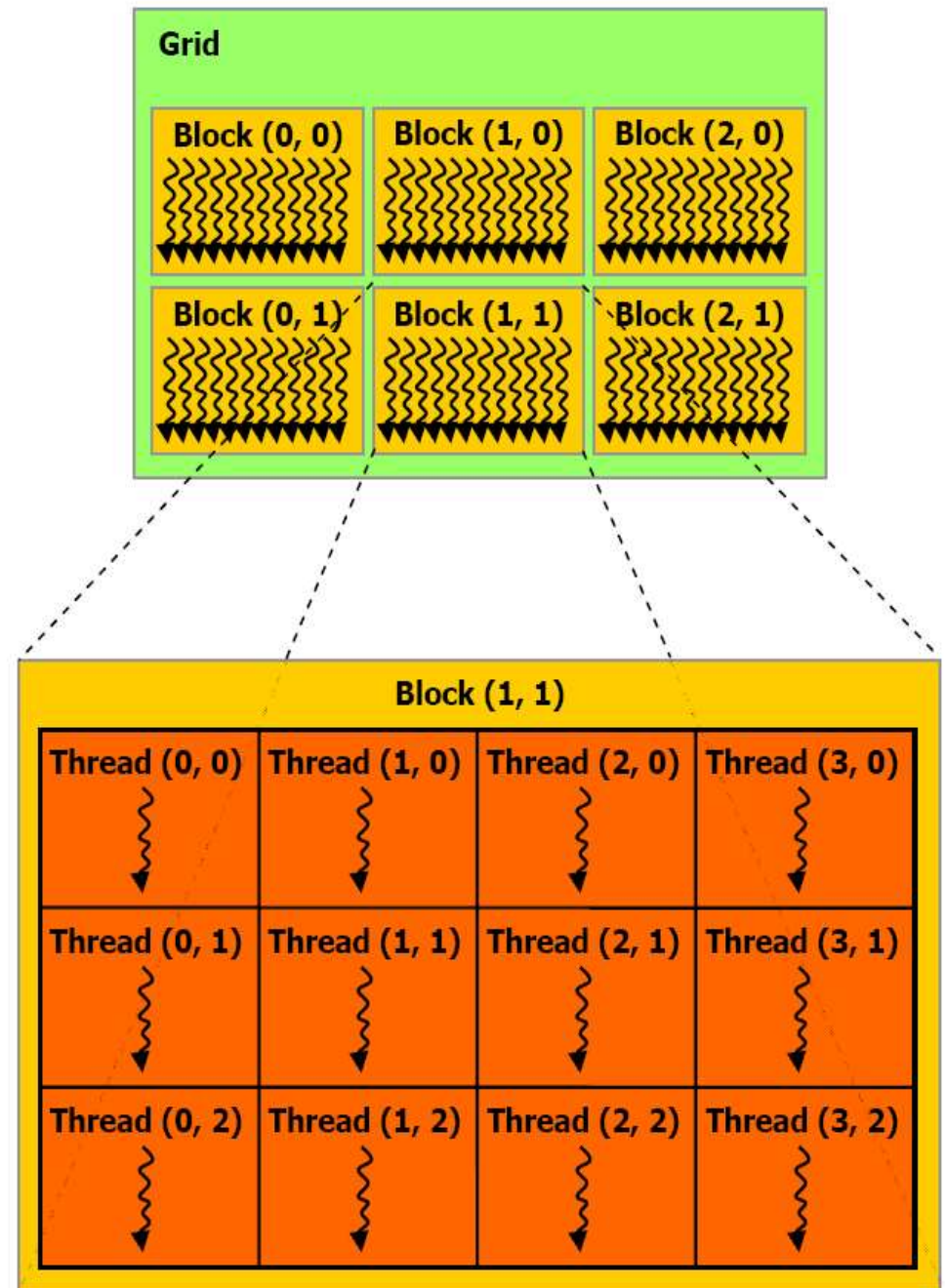
(sub-core == SM partition)

CUDA 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)

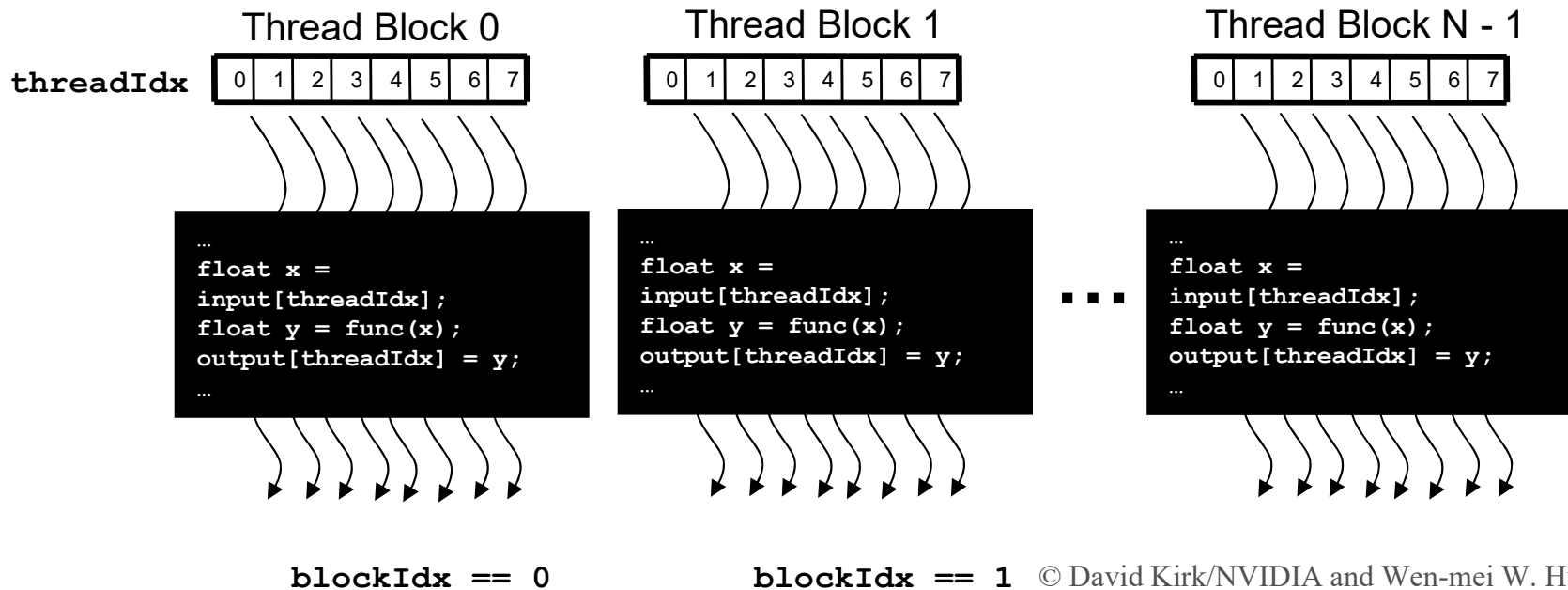


Threads in Block, Blocks in Grid



- Identify work of thread via

- `threadIdx`
- `blockIdx`

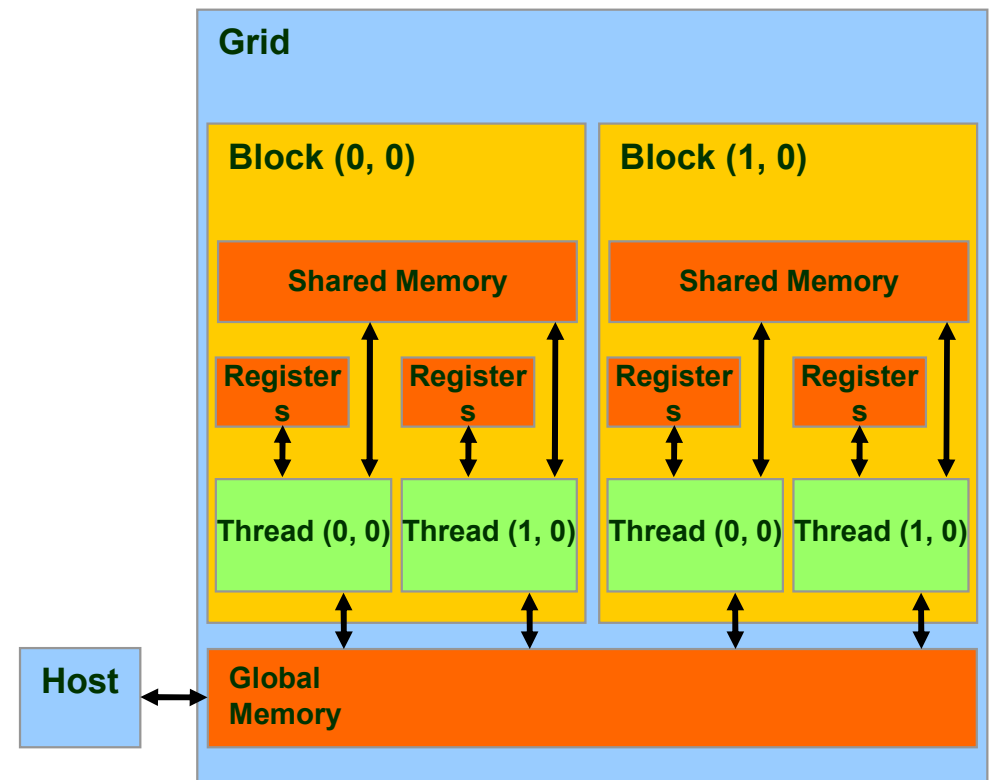


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

CUDA Memory Model and Usage



- `cudaMalloc()` , `cudaFree()`
- `cudaMallocArray()` ,
`cudaMalloc2DArray()` ,
`cudaMalloc3DArray()`
- `cudaMemcpy()`
- `cudaMemcpyArray()`
- Host \leftrightarrow host
Host \leftrightarrow device
Device \leftrightarrow 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

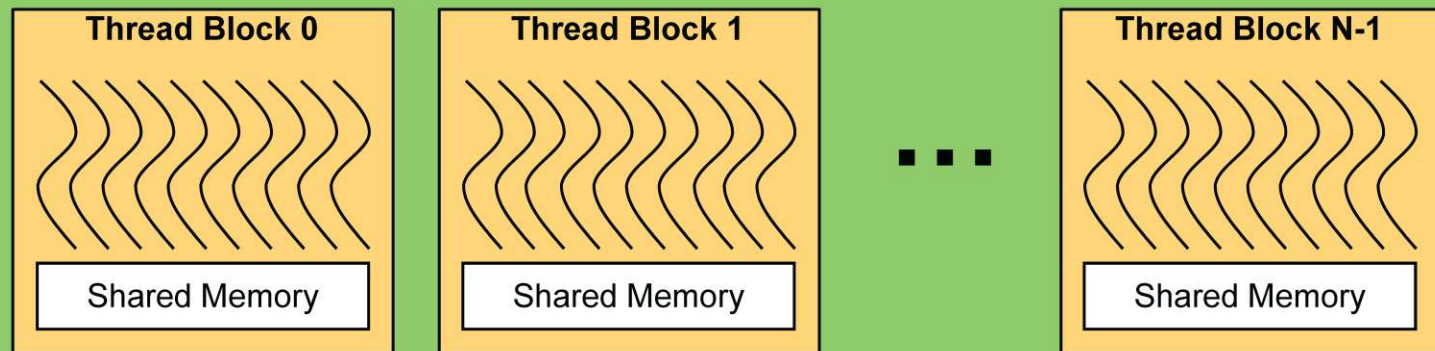
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

```
...  
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

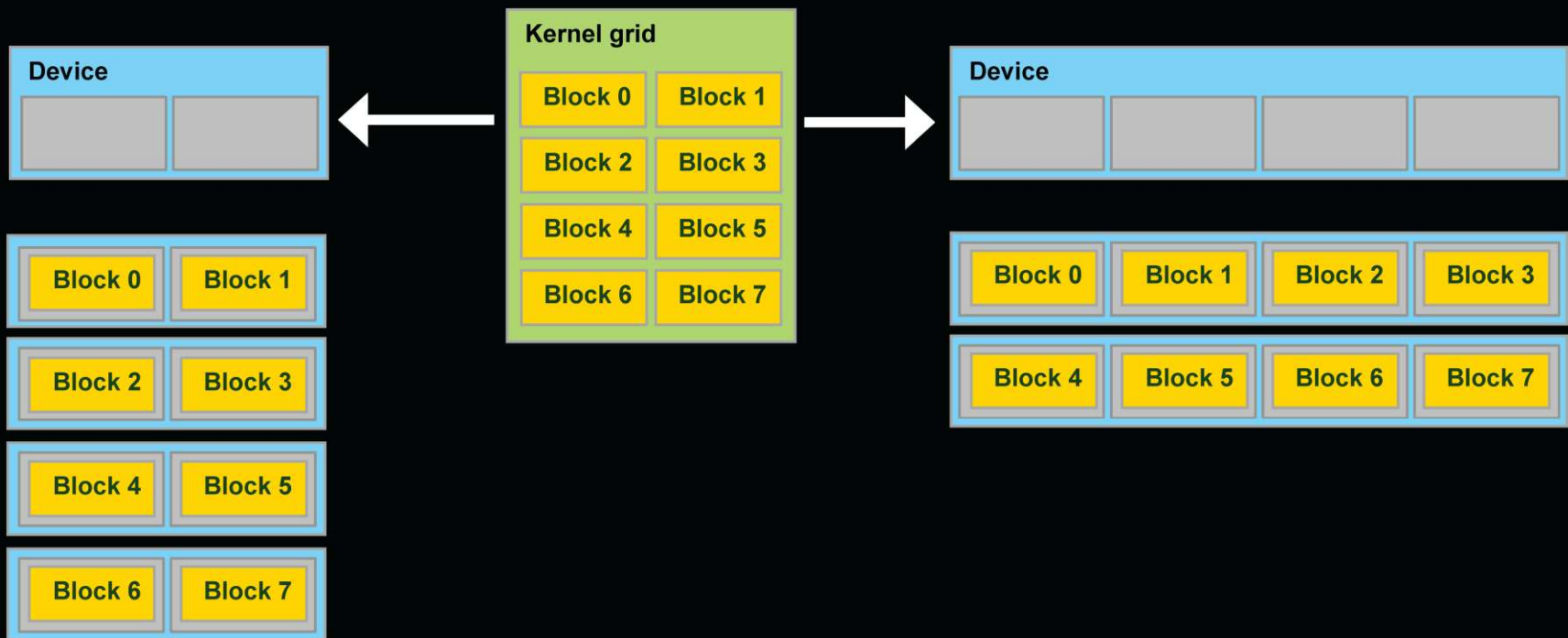
Grid



* 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

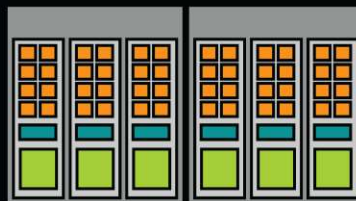


Execution Model

Software



Hardware



Threads are executed by thread processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

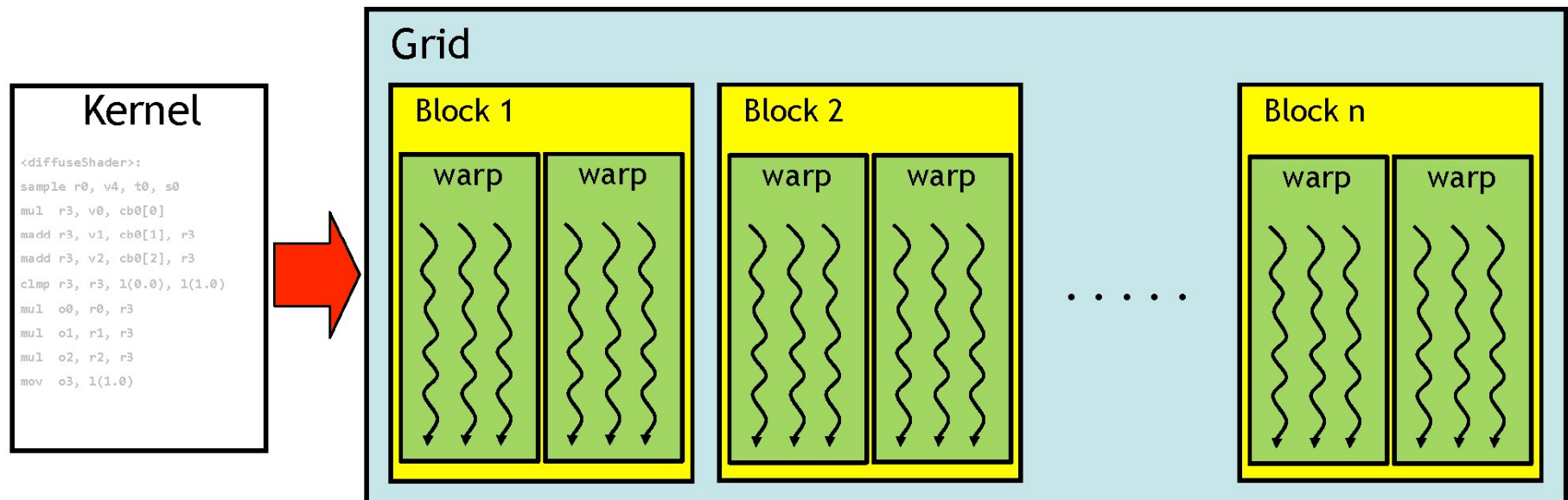
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

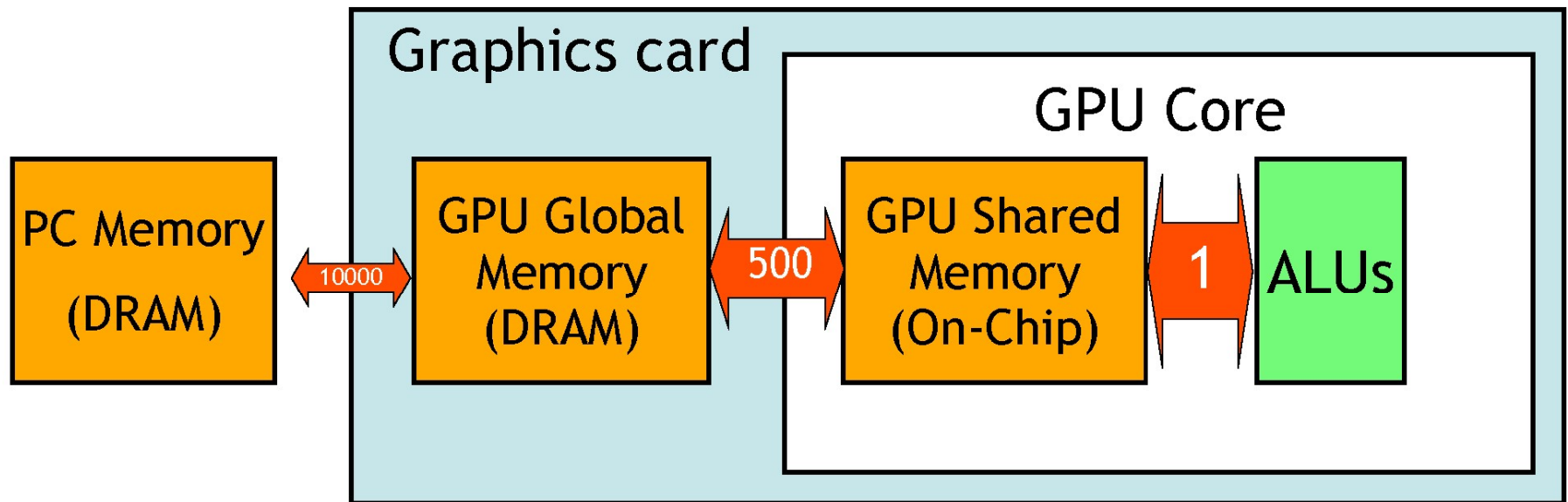
CUDA Programming Model

- 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



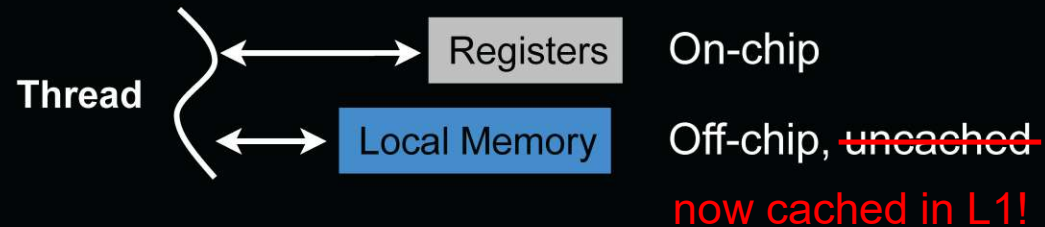
CUDA Memory Structure

- 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

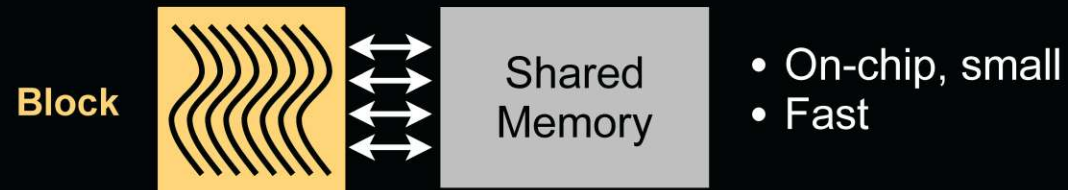


Kernel Memory Access

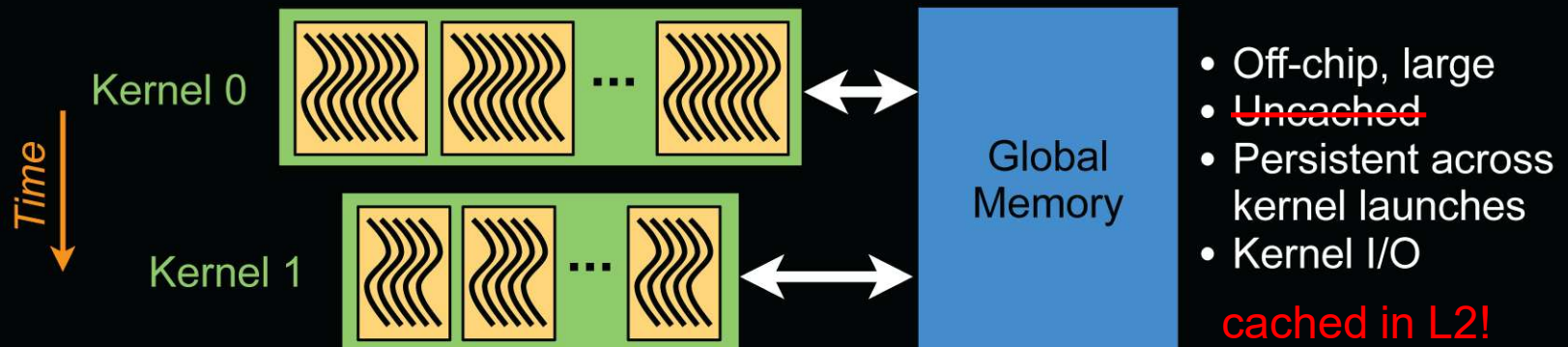
● Per-thread



● 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!

PTX (Memory) State Spaces



PTX ISA 9.0 (Chapter 5)

Name	Addressable	Initializable	Access	Sharing
<code>.reg</code>	No	No	R/W	per-thread
<code>.sreg</code>	No	No	RO	per-CTA
<code>.const</code>	Yes	Yes ¹	RO	per-grid
<code>.global</code>	Yes	Yes ¹	R/W	Context
<code>.local</code>	Yes	No	R/W	per-thread
<code>.param</code> (as input to kernel)	Yes ²	No	RO	per-grid
<code>.param</code> (used in functions)	Restricted ³	No	R/W	per-thread
<code>.shared</code>	Yes	No	R/W	per-cluster ⁵
<code>.tex</code>	No ⁴	Yes, via driver	RO	Context

Notes:

¹ Variables in `.const` and `.global` state spaces are initialized to zero by default.

² Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

³ Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

⁴ Accessible only via the `tex` instruction.

⁵ Visible to the owning CTA and other active CTAs in the cluster.

Thank you.