

# CS 380 - GPU and GPGPU Programming

## Lecture 12: GPU Compute APIs, Pt. 1

Markus Hadwiger, KAUST



# Reading Assignment #6 (until Oct 9)

## Read (required):

- Programming Massively Parallel Processors book (4th edition),  
**Chapter 2** (*Heterogeneous data parallel computing*),  
**Chapter 3** (*Multidimensional grids and data*)
- CUDA NVCC documentation: [https://docs.nvidia.com/cuda/pdf/CUDA\\_Compiler\\_Driver\\_NVCC.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf)  
Read Chapters 1 – 3; Chapter 5; get an overview of the rest

## Read (optional):

- Look at the “Tuning Guides“ for different architectures in the CUDA SDK
- PTX Instruction Set Architecture 7.8: <https://docs.nvidia.com/cuda/parallel-thread-execution/>  
Read Chapters 1 – 3; get an overview of Chapter 12;  
browse through the other chapters to get a feeling for what PTX looks like
- CUDA SASS ISA, Chapter 4: [https://docs.nvidia.com/cuda/pdf/CUDA\\_Binary\\_Utils.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utils.pdf)



# CUDA Update (11.8)

CUDA SDK 11.8 and documentation now online

## CUDA C Programming Guide

- New compute capability 9.0 (Hopper GPUs)
- Specific info for compute capability 8.9 (Ada Lovelace GPUs) missing

## CUDA Binary Utilities

- New Hopper SASS (Ada Lovelace SASS is the same as Ampere)

## CUDA NVCC Compiler Driver

- Support for cc 8.9 and 9.0 (PTX & cubin: sm\_89, sm\_90, compute\_89, compute\_90)

## PTX ISA 7.8

- Support for cc 8.9 and 9.0 (sm\_89 and sm\_90) target architectures

## Hopper Compatibility Guide, Hopper Tuning Guide

<https://developer.nvidia.com/blog/cuda-toolkit-11-8-new-features-revealed/>



# Ada Lovelace Architecture Whitepaper

<https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf>

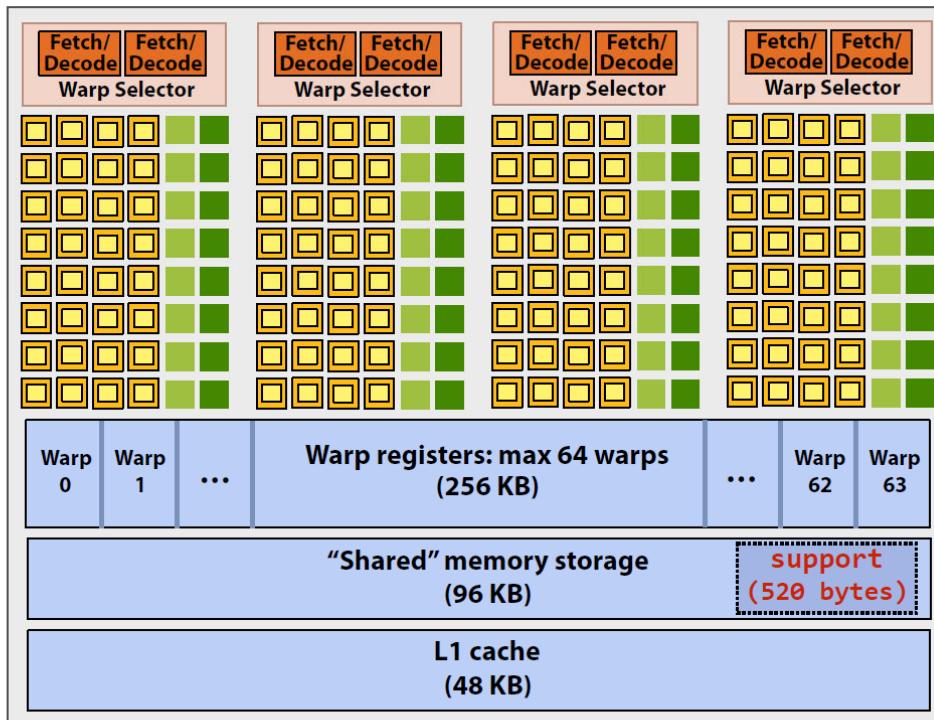
Graphics Card	GeForce RTX 2080 Ti	GeForce RTX 3090 Ti	GeForce RTX 4090
GPU Codename	TU102	GA102	AD102
GPU Architecture	NVIDIA Turing	NVIDIA Ampere	NVIDIA Ada Lovelace
GPCs	6	7	11
TPCs	34	42	64
SMs	68	84	128
CUDA Cores / SM	64	128	128
CUDA Cores / GPU	4352	10752	16384
Tensor Cores / SM	8 (2nd Gen)	4 (3rd Gen)	4 (4th Gen)
Tensor Cores / GPU	544	336 (3rd Gen)	512 (4th Gen)
RT Cores	68 (1st Gen)	84 (2nd Gen)	128 (3rd Gen)
GPU Boost Clock (MHz)	1635	1860	2520
Peak FP32 TFLOPS (non-Tensor) <sup>1</sup>	14.2	40	82.6
Peak FP16 TFLOPS (non-Tensor) <sup>1</sup>	28.5	40	82.6

# GPU Compute APIs



- 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: Simple Typical CUDA Kernel (SM Perspective)



```
#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;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called “warps”.

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

(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SM core operation each clock:

- Select up to four runnable warps from 64 resident on SM core (thread-level parallelism)
- Select up to two runnable instructions per warp (instruction-level parallelism) \*

# CUDA Software Development

CUDA Optimized Libraries:  
math.h, FFT, BLAS, ...

Integrated CPU + GPU  
C Source Code

NVIDIA C Compiler

NVIDIA Assembly  
for Computing (PTX)

CPU Host Code

CUDA  
Driver

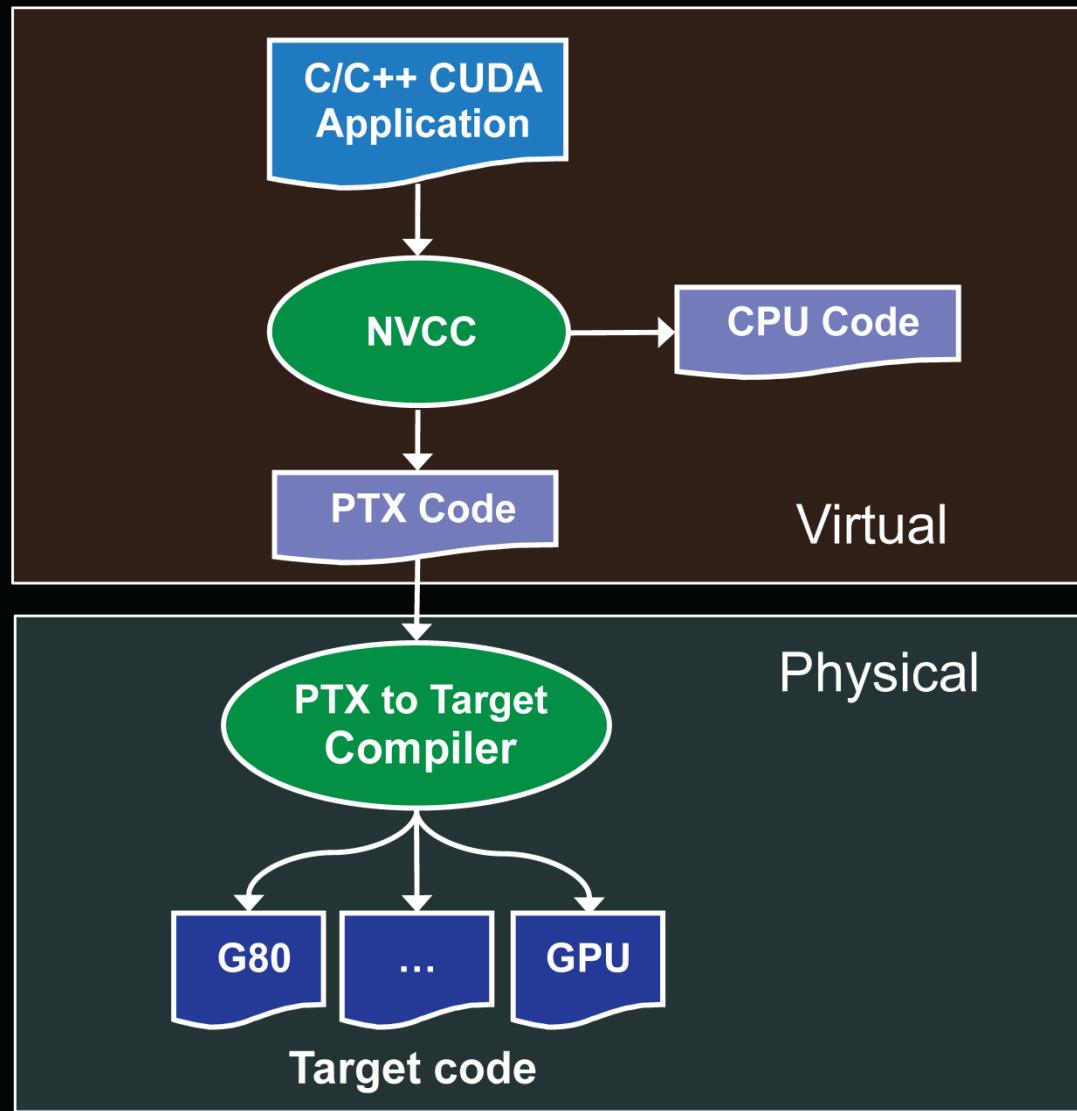
Profiler

Standard C Compiler

GPU

CPU

# Compiling CUDA Code

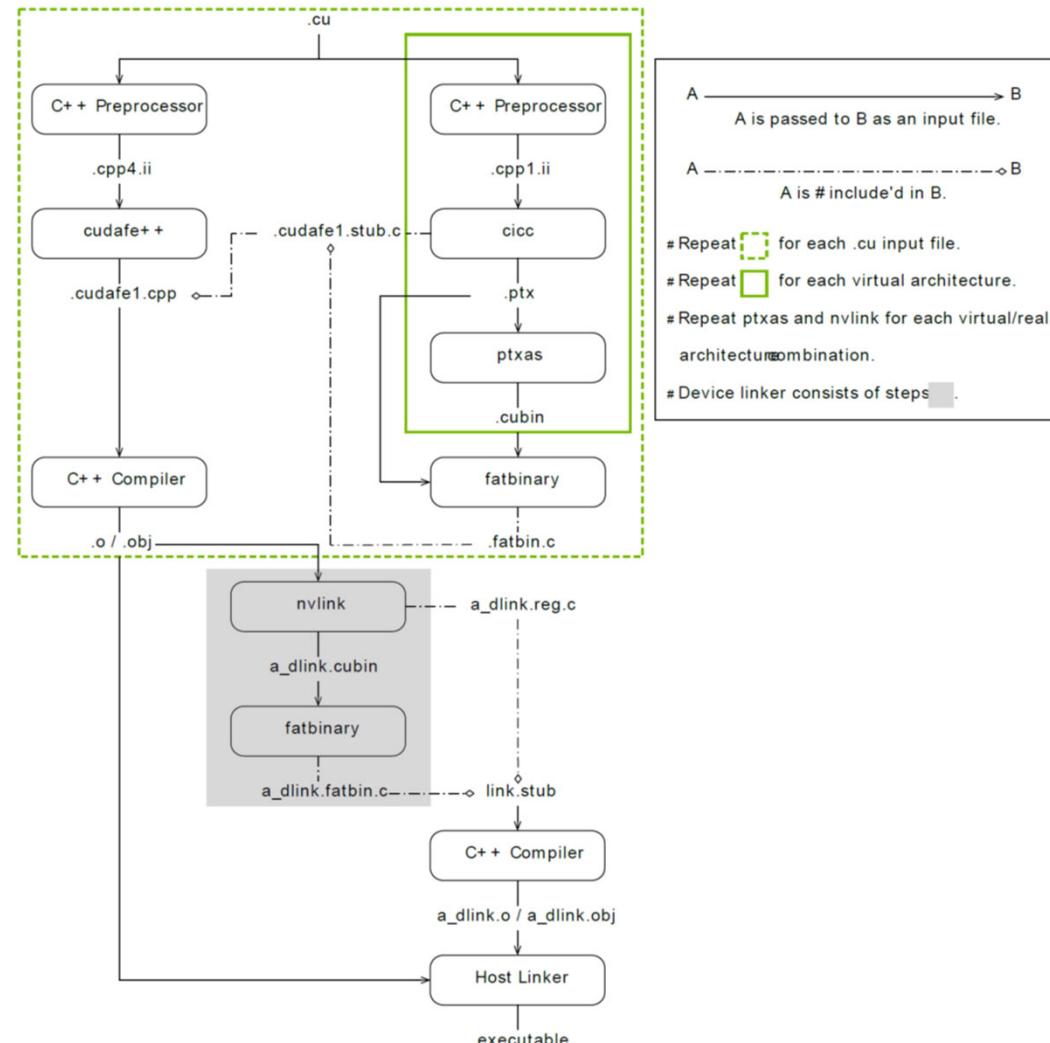




# CUDA Compilation Trajectory

## CUDA Compiler Driver (NVCC) docs:

[CUDA\\_Compiler\\_Driver\\_NVCC.pdf](#)



# CUDA Compilation Trajectory / Code Gen



## 4.2.7. Options for Steering GPU Code Generation

### 4.2.7.1. `--gpu-architecture arch` (`-arch`)

*Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.*

With the exception as described for the shorthand below, the architecture specified with this option must be a *virtual* architecture (such as `compute_50`). Normally, this option alone does not trigger assembly of the generated PTX for a *real* architecture (that is the role of `nvcc` option `--gpu-code`, see below); rather, its purpose is to control preprocessing and compilation of the input to PTX.

For convenience, in case of simple `nvcc` compilations, the following shorthand is supported. If no value for option `--gpu-code` is specified, then the value of this option defaults to the value of `--gpu-architecture`. In this situation, as only exception to the description above, the value specified for `--gpu-architecture` may be a *real* architecture (such as a `sm_50`), in which case `nvcc` uses the specified *real* architecture and its closest *virtual* architecture as effective architecture values. For example, `nvcc --gpu-architecture=sm_50` is equivalent to `nvcc --gpu-architecture=compute_50 --gpu-code=sm_50,compute_50`.

See [Virtual Architecture Feature List](#) for the list of supported *virtual* architectures and [GPU Feature List](#) for the list of supported *real* architectures.

# CUDA Compilation Trajectory / Code Gen



## 4.2.7.2. --gpu-code *code*,... (-code)

*Specify the name of the NVIDIA GPU to assemble and optimize PTX for.*

nvcc embeds a compiled code image in the resulting executable for each specified *code* architecture, which is a true binary load image for each *real* architecture (such as sm\_50), and PTX code for the *virtual* architecture (such as compute\_50).

During runtime, such embedded PTX code is dynamically compiled by the CUDA runtime system if no binary load image is found for the *current* GPU.

Architectures specified for options [--gpu-architecture](#) and --gpu-code may be *virtual* as well as *real*, but the *code* architectures must be compatible with the *arch* architecture. When the --gpu-code option is used, the value for the [--gpu-architecture](#) option must be a *virtual* PTX architecture.

For instance, `--gpu-architecture=compute_60` is not compatible with `--gpu-code=sm_52`, because the earlier compilation stages will assume the availability of `compute_60` features that are not present on `sm_52`.

See [Virtual Architecture Feature List](#) for the list of supported *virtual* architectures and [GPU Feature List](#) for the list of supported *real* architectures.

Also look at compatibility guides:

[https://docs.nvidia.com/cuda/pdf/NVIDIA\\_Ampere\\_GPU\\_Architecture\\_Compatibility\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf)

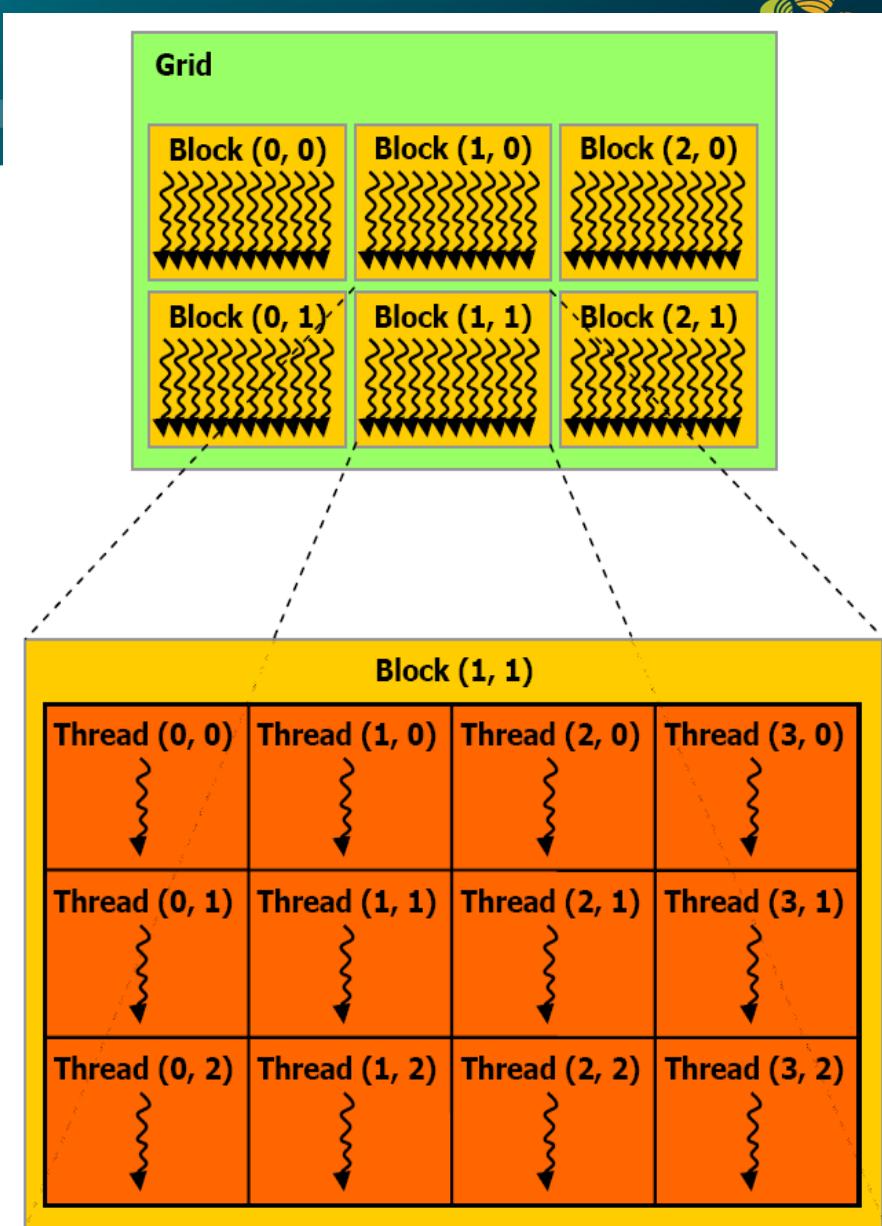
[https://docs.nvidia.com/cuda/pdf/Hopper\\_Compatibility\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/Hopper_Compatibility_Guide.pdf)

# CUDA Multi-Threading

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

Execution on actual hardware:

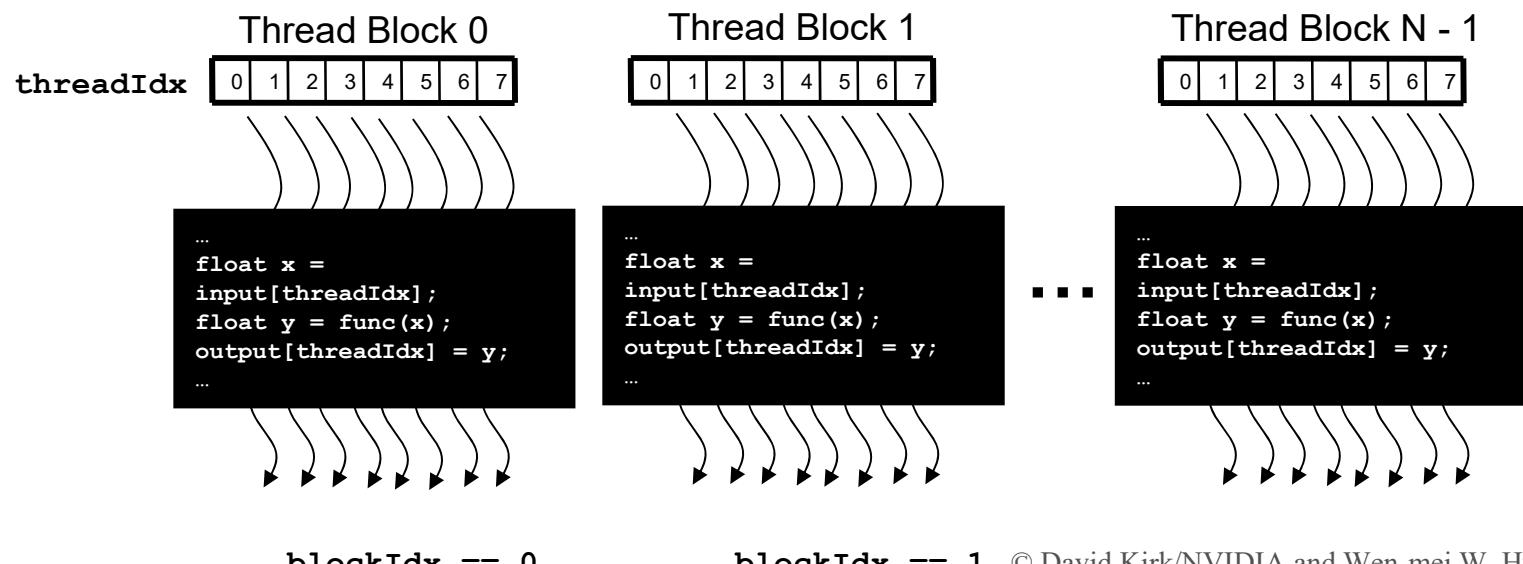
- Thread blocks assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
- 32 threads grouped into a **warp** (on all compute capabilities)





# Threads in Block, Blocks in Grid

- Identify work of thread via
  - **threadIdx**
  - **blockIdx**

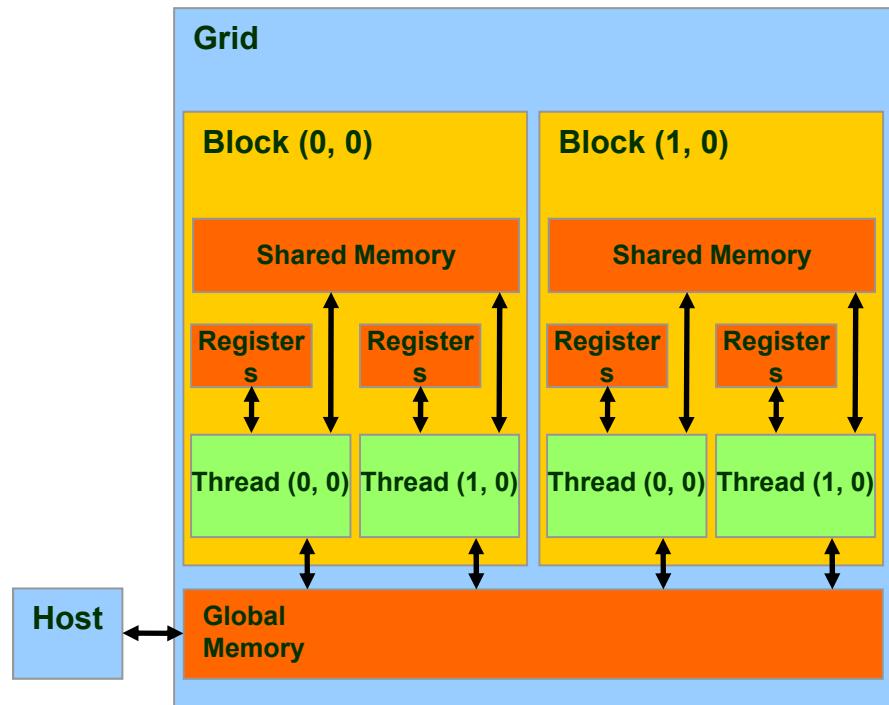


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009  
ECE 498AL, University of Illinois, Urbana-Champaign



# 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

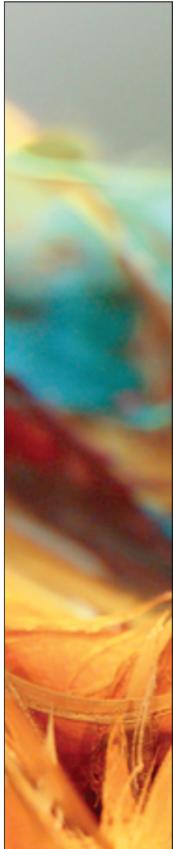
- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching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

## Definitions

*Device* = GPU

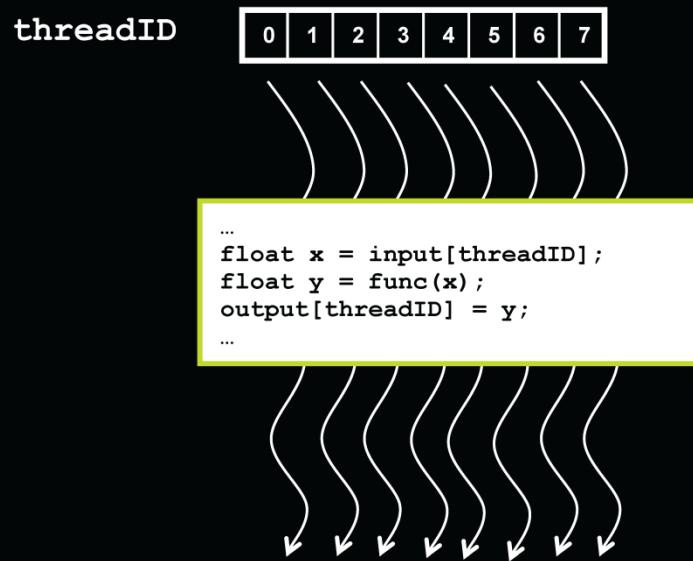
*Host* = CPU

*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

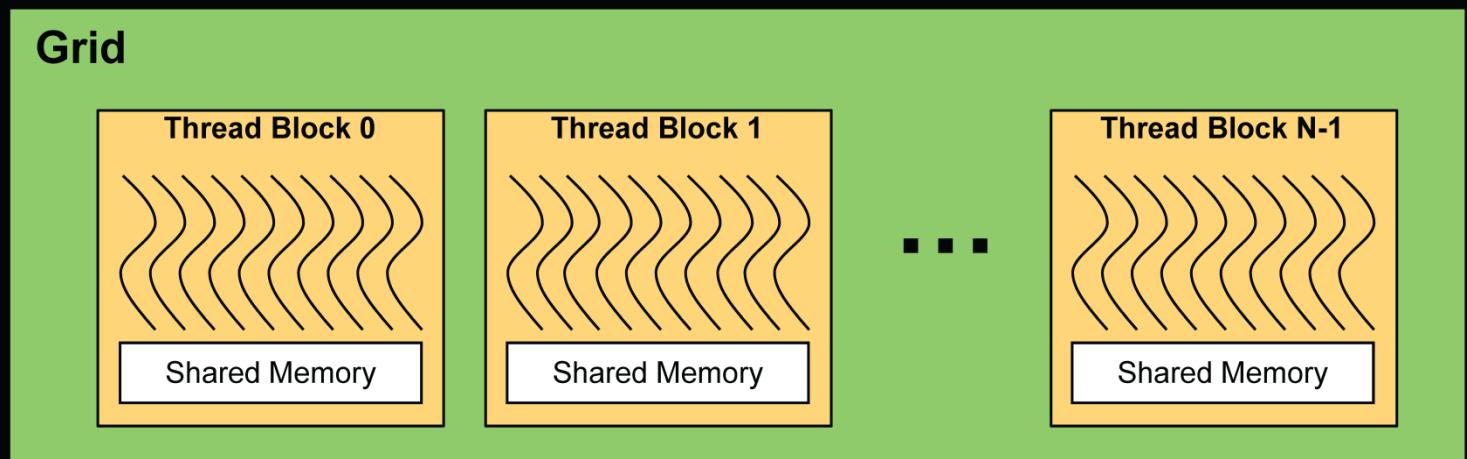


© 2008 NVIDIA Corporation.



# 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*



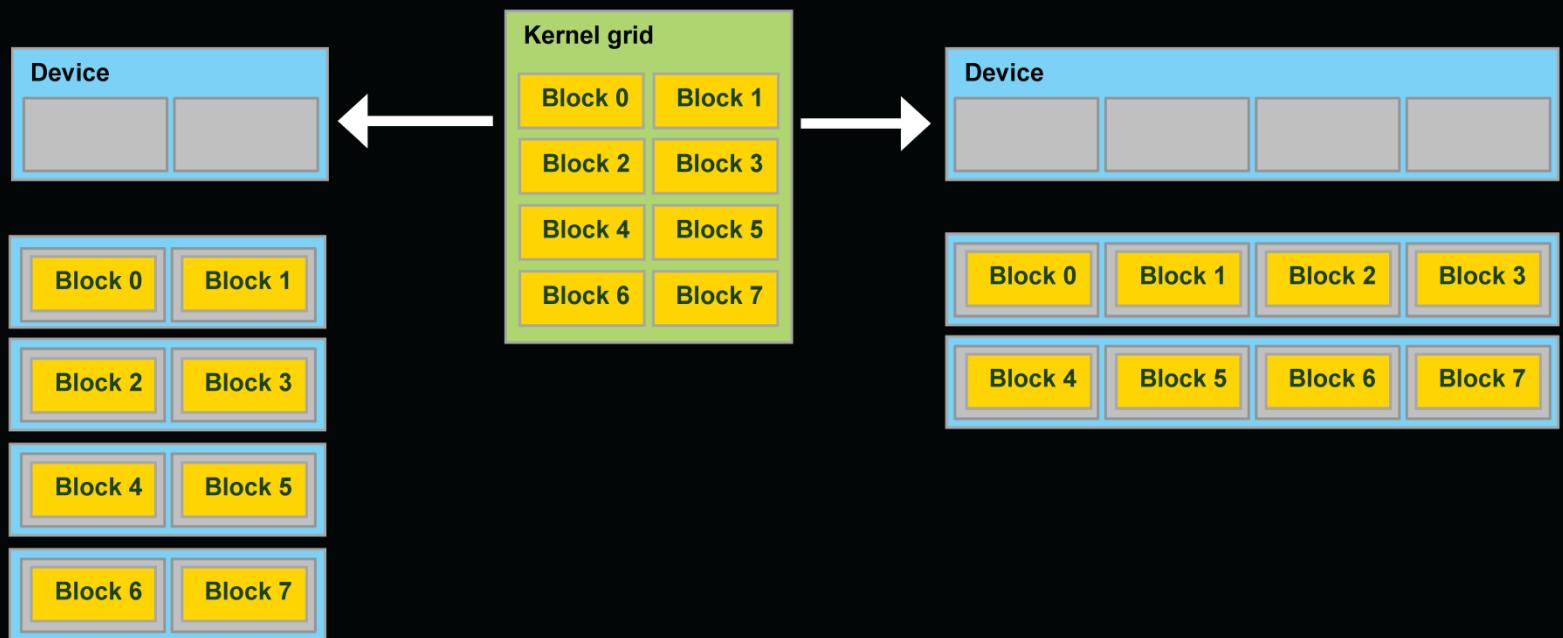
\* brand new on Hopper: thread block clusters

© 2008 NVIDIA Corporation.

**NVIDIA**

# Transparent Scalability

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



# Execution Model

## Software

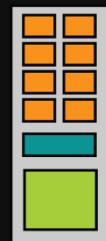


Thread Block



Grid

## Hardware



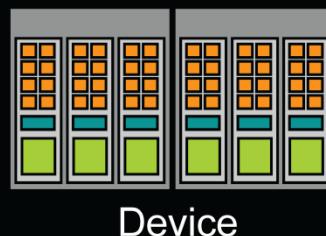
Multiprocessor

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)

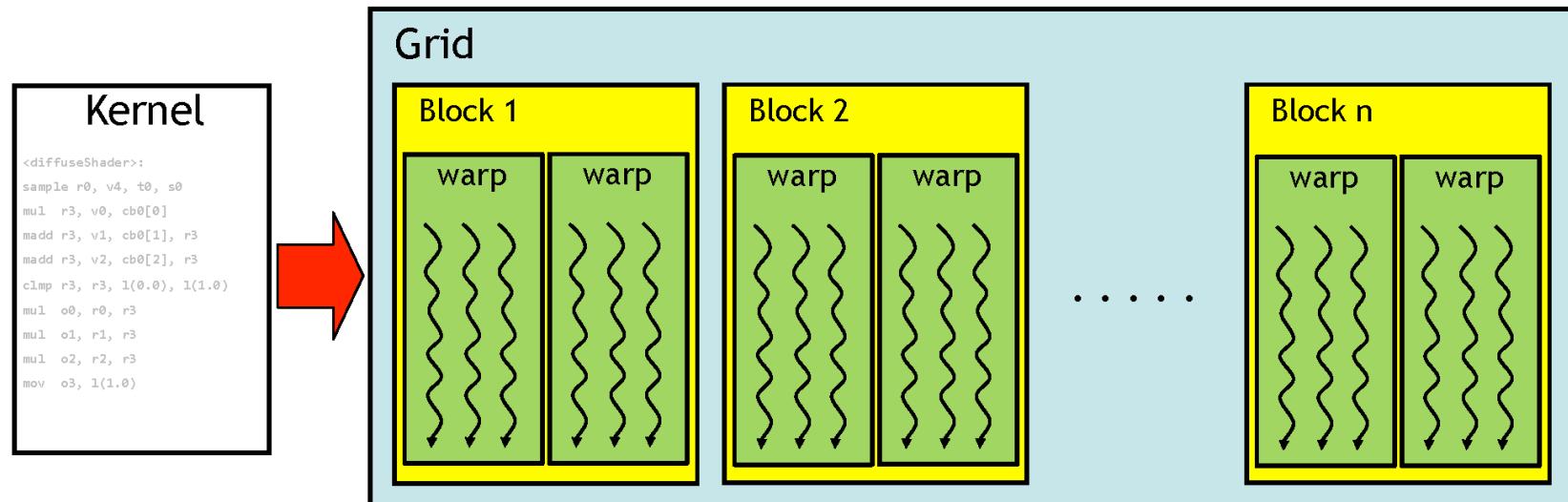


A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

# CUDA Programming Model

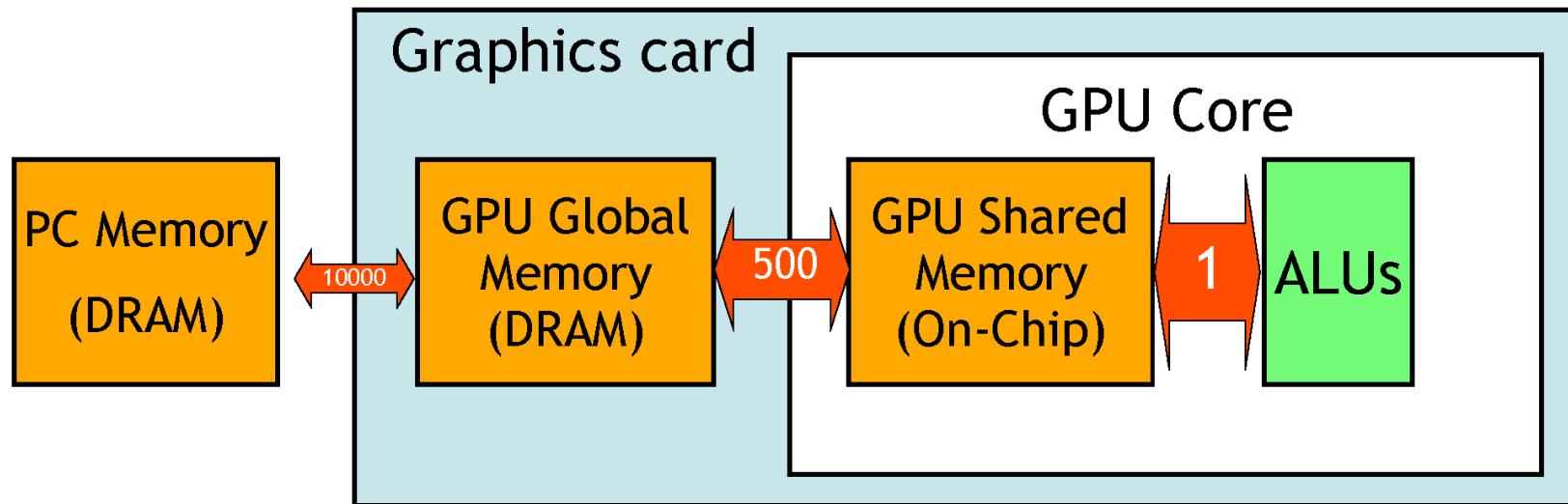
- Kernel
  - GPU program that runs on a thread grid
- Thread hierarchy
  - Grid : a set of blocks
  - Block : a set of warps
  - Warp : a SIMD group of 32 threads
  - Grid size \* block size = total # of threads



# CUDA Memory Structure

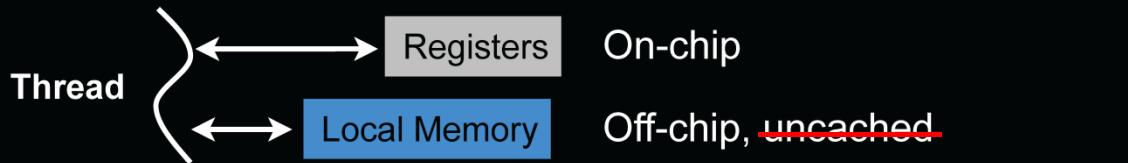
---

- 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

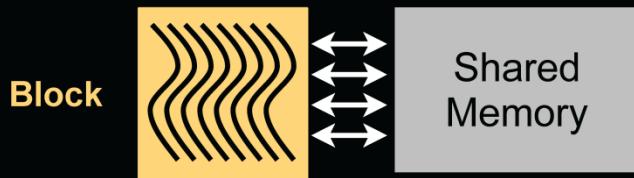


# Kernel Memory Access

- Per-thread

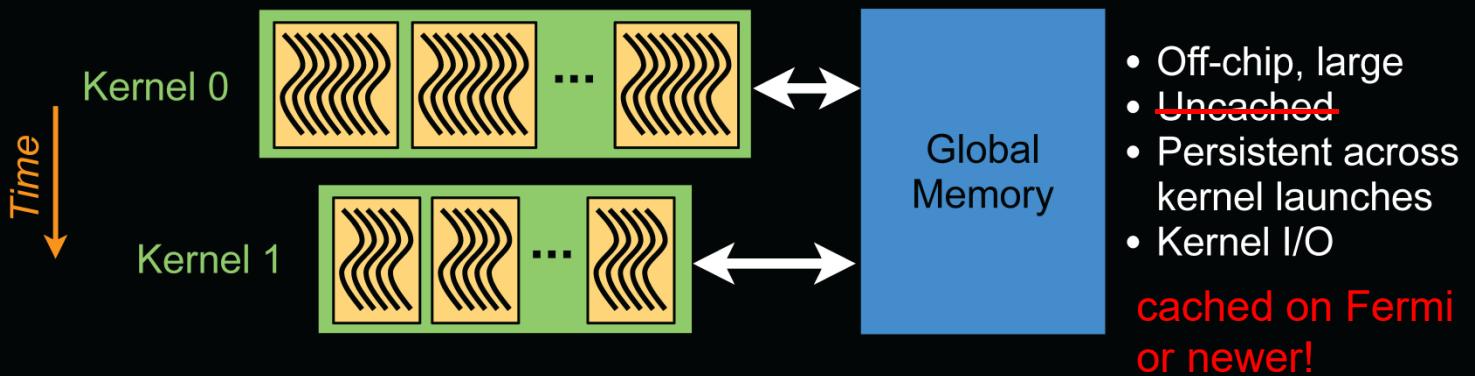


- Per-block



- On-chip, small
- Fast

- Per-device



# Memory Architecture



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	<del>No*</del> YES	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	<del>No*</del> 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!



# (Memory) State Spaces

## PTX ISA 7.8 (Chapter 5)

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

### Notes:

<sup>1</sup> Variables in .const and .global state spaces are initialized to zero by default.

<sup>2</sup> Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

<sup>3</sup> 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.

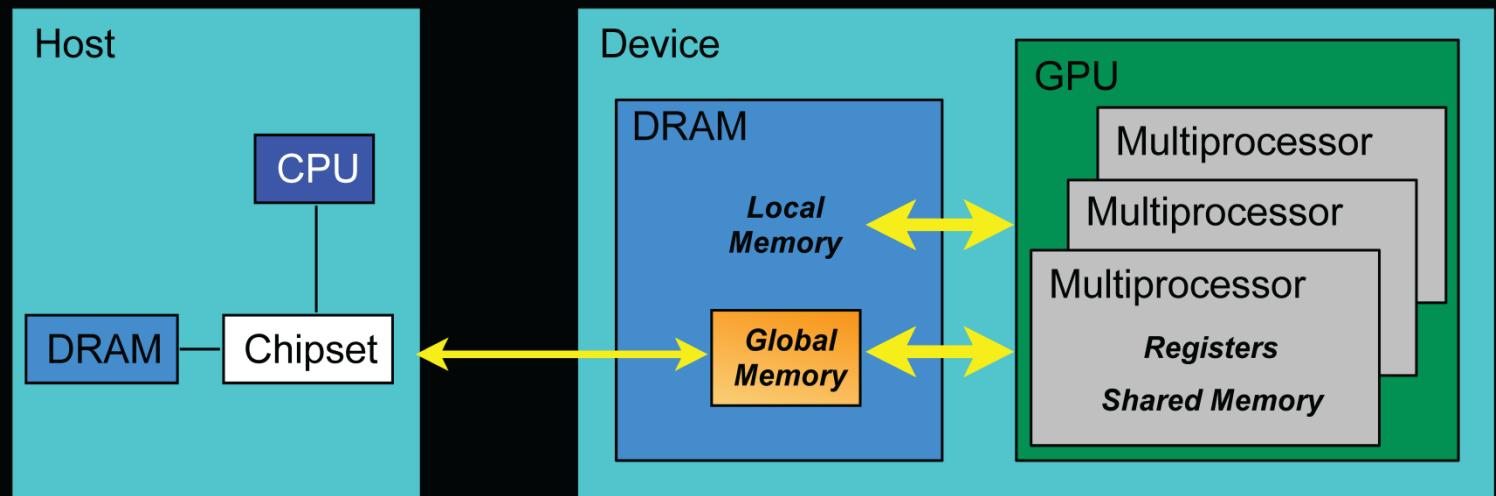
<sup>4</sup> Accessible only via the `tex` instruction.

<sup>5</sup> Visible to the owning CTA and other active CTAs in the cluster.

# Managing Memory

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

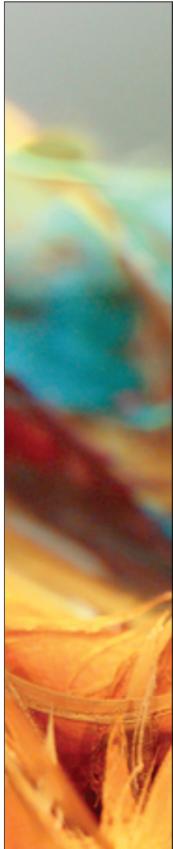
- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
  - Allocate / free
  - Copy data to and from device
  - Applies to *global device memory (DRAM)*



# 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**



# Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory except: (\*) and (\*\*)
  - Must have **void** return type
  - No variable number of arguments (“varargs”)
  - ~~Not recursive~~ recursion supported on **\_\_device\_\_** functions from cc. 2.x (i.e., basically on *all* current GPUs)
  - No static variables

- Function arguments automatically copied from host to device

(\*) “unified memory programming” introduced with CUDA 6 (cc. 3.x +):  
allocate memory with **cudaMallocManaged()**; uses automatic migration

(\*\*) also: mapped pinned (page-locked) memory (“zero-copy memory”):  
allocate memory with **cudaMallocHost()**; beware of low performance!!

Note: UVA (“unified virtual addressing”; cc. 2.x +) is something different!!  
just pertains to unified pointers (see **cudaPointerGetAttributes()**, ...) 

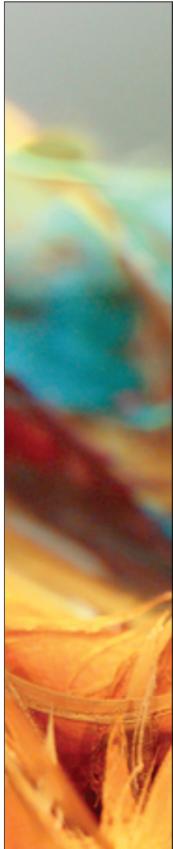
# Function Qualifiers

- **Kernels designated by function qualifier:**
  - **`__global__`**
    - Function called from host and executed on device
    - Must return void
- **Other CUDA function qualifiers**
  - **`__device__`**
    - Function called from device and run on device
    - Cannot be called from host code
  - **`__host__`**
    - Function called from host and executed on host (default)
    - `__host__` and `__device__` qualifiers can be combined to generate both CPU and GPU code

# Variable Qualifiers (GPU code)

- **\_\_device\_\_**
  - Stored in global memory (large, high latency, no cache)
  - Allocated with `cudaMalloc` (**\_\_device\_\_** qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- **\_\_shared\_\_**
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- **Unqualified variables:**
  - Scalars and built-in vector types are stored in registers
  - What doesn't fit in registers spills to “local” memory

CUDA 6+: **\_\_managed\_\_** (with **\_\_device\_\_**) for managed memory (unified memory programming)



# Launching Kernels

- Modified C function call syntax:

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

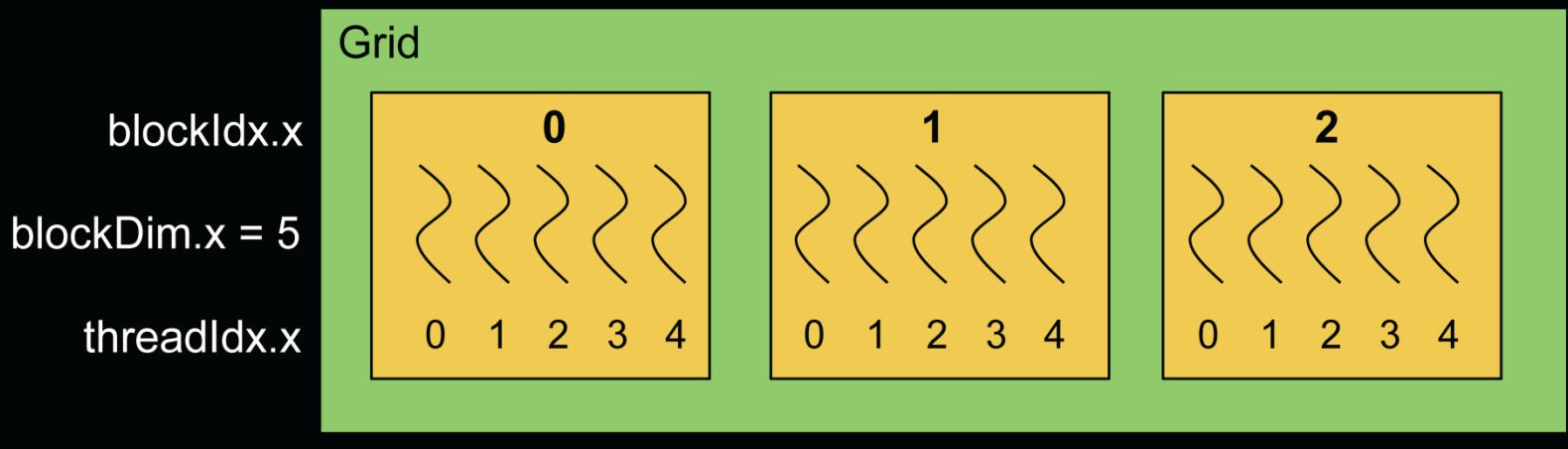
- Execution Configuration (“<<< >>>”)
  - **dG** - dimension and size of grid in blocks
    - Two-dimensional: **x** and **y**
    - Blocks launched in the grid: **dG.x \* dG.y**
  - **dB** - dimension and size of blocks in threads:
    - Three-dimensional: **x**, **y**, and **z**
    - Threads per block: **dB.x \* dB.y \* dB.z**
  - Unspecified **dim3** fields initialize to 1

# CUDA Built-in Device Variables

- All **`_global_`** and **`_device_`** functions have access to these automatically defined variables
  - **`dim3 gridDim;`**
    - Dimensions of the grid in blocks (at most 2D)
  - **`dim3 blockDim;`**
    - Dimensions of the block in threads
  - **`dim3 blockIdx;`**
    - Block index within the grid
  - **`dim3 threadIdx;`**
    - Thread index within the block

# Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (`threadIdx`) to a global ID which can be used as array indices



0 1 2 3 4      5 6 7 8 9      10 11 12 13 14

blockIdx.x\*blockDim.x  
+ threadIdx.x

Thank you.