King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 11: GPU Compute APIs, Pt. 1

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 11)

Read (required):

- CUDA NVCC doc (`https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf`)
  Read Chapters 1 – 3; Chapter 5; get an overview of the rest
- Programming Massively Parallel Processors book,
  3rd edition: Chapter 4 (*Memory and Data Locality*), **OR**
  2nd edition: Chapter 5 (*CUDA Memories*)
- Look at the "Tuning Guides" for different architectures in the CUDA SDK

Read (optional):

- PTX Instruction Set Architecture 7.4 (`https://docs.nvidia.com/cuda/pdf/ptx_isa_7.4.pdf`)
  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, Chapter 4: `https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf`

GPU Compute APIs

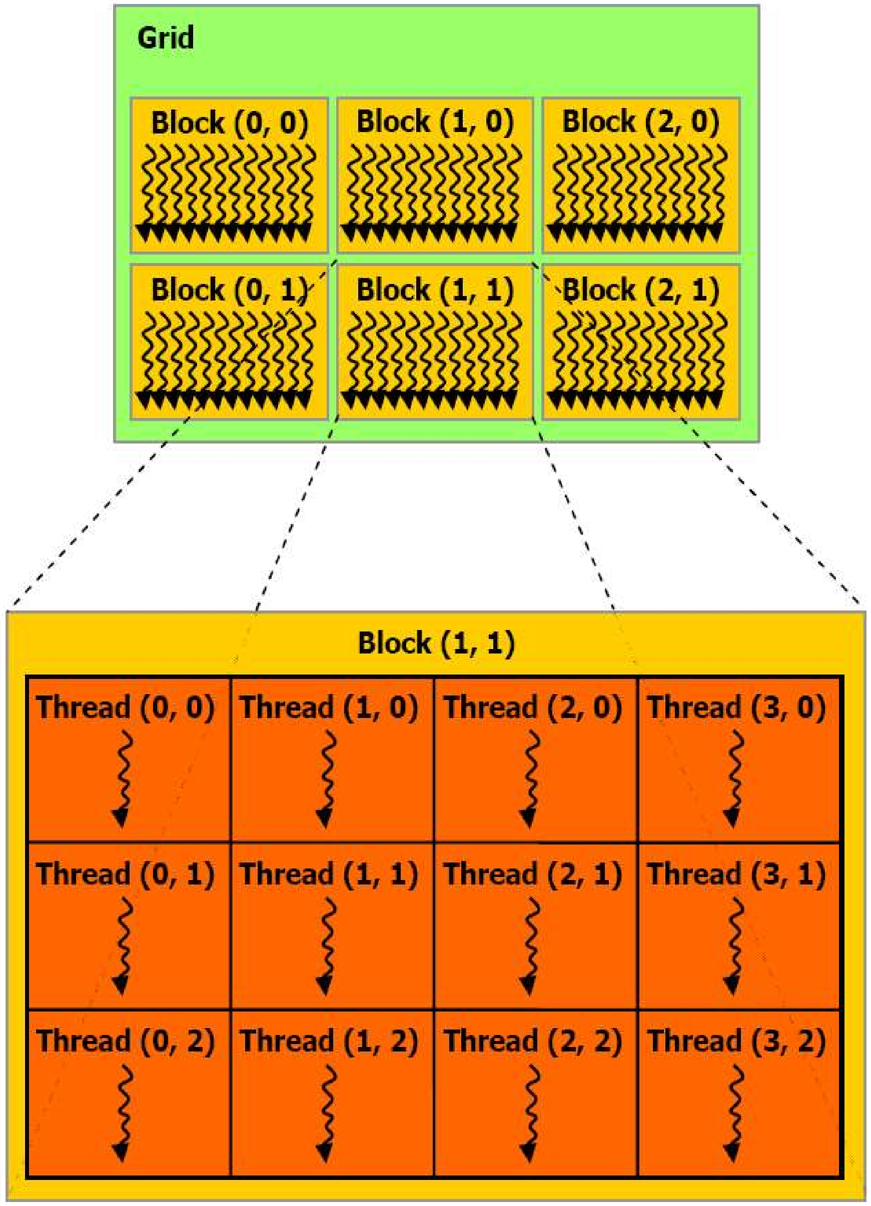
NVIDIA CUDA

- 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`)

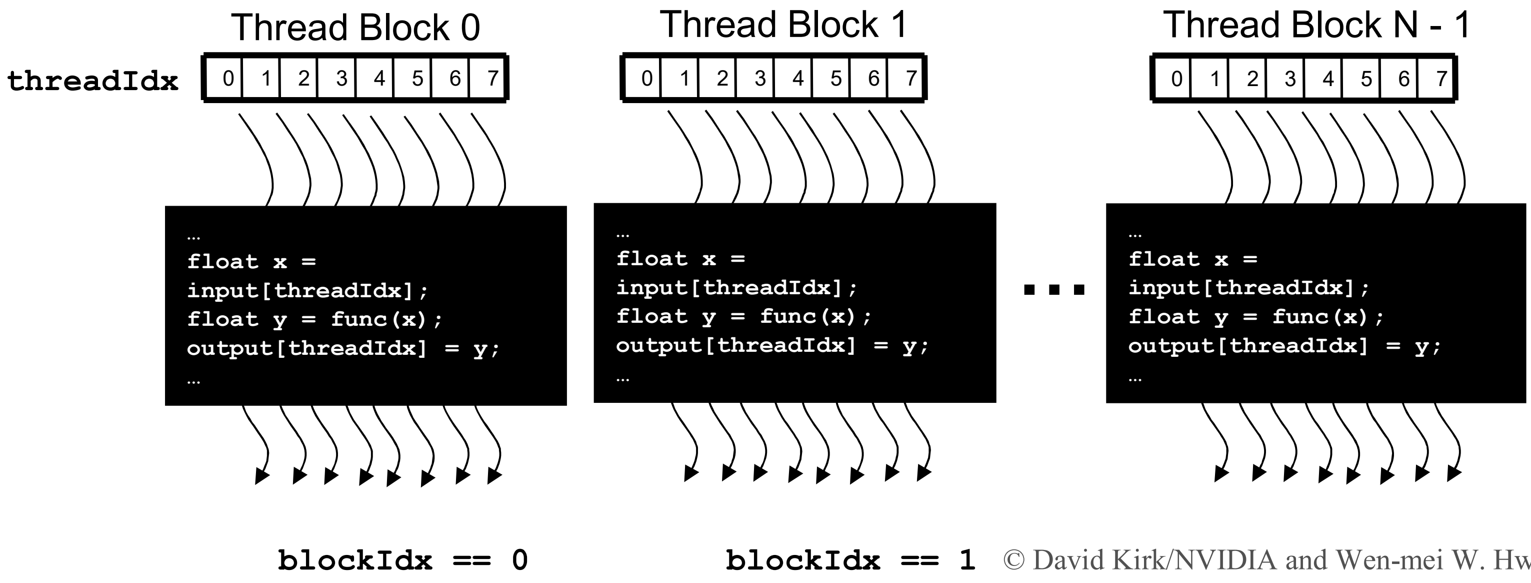
CUDA Multi-Threading

- CUDA model groups threads into blocks; blocks into grid
- Execution on actual hardware:
  - Block assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
  - 32 threads grouped into warp



Threads in Block, Blocks in Grid

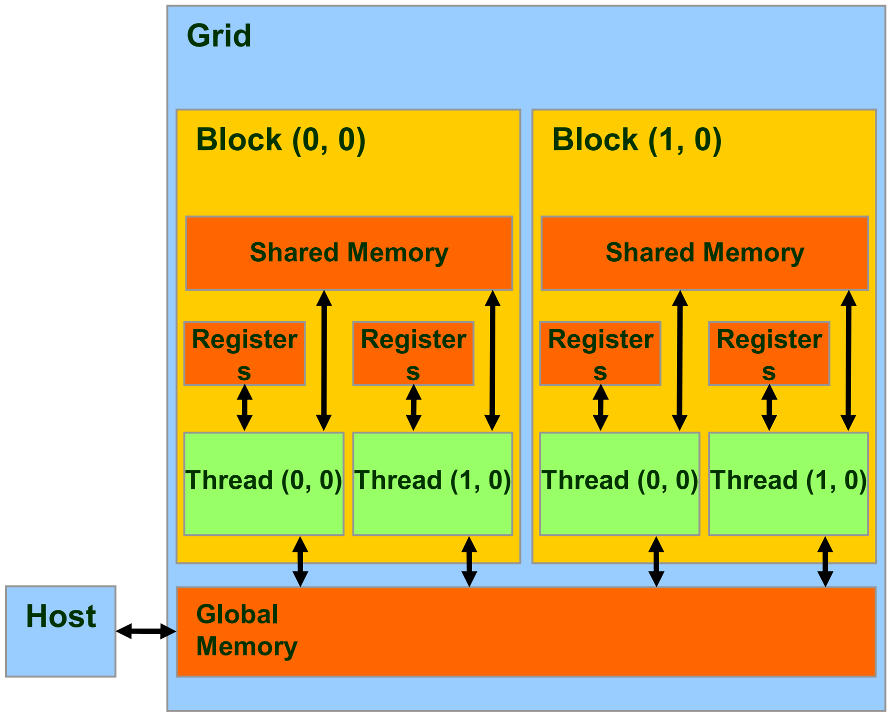
- Identify work of thread via
  - **threadIdx**
  - **blockIdx**



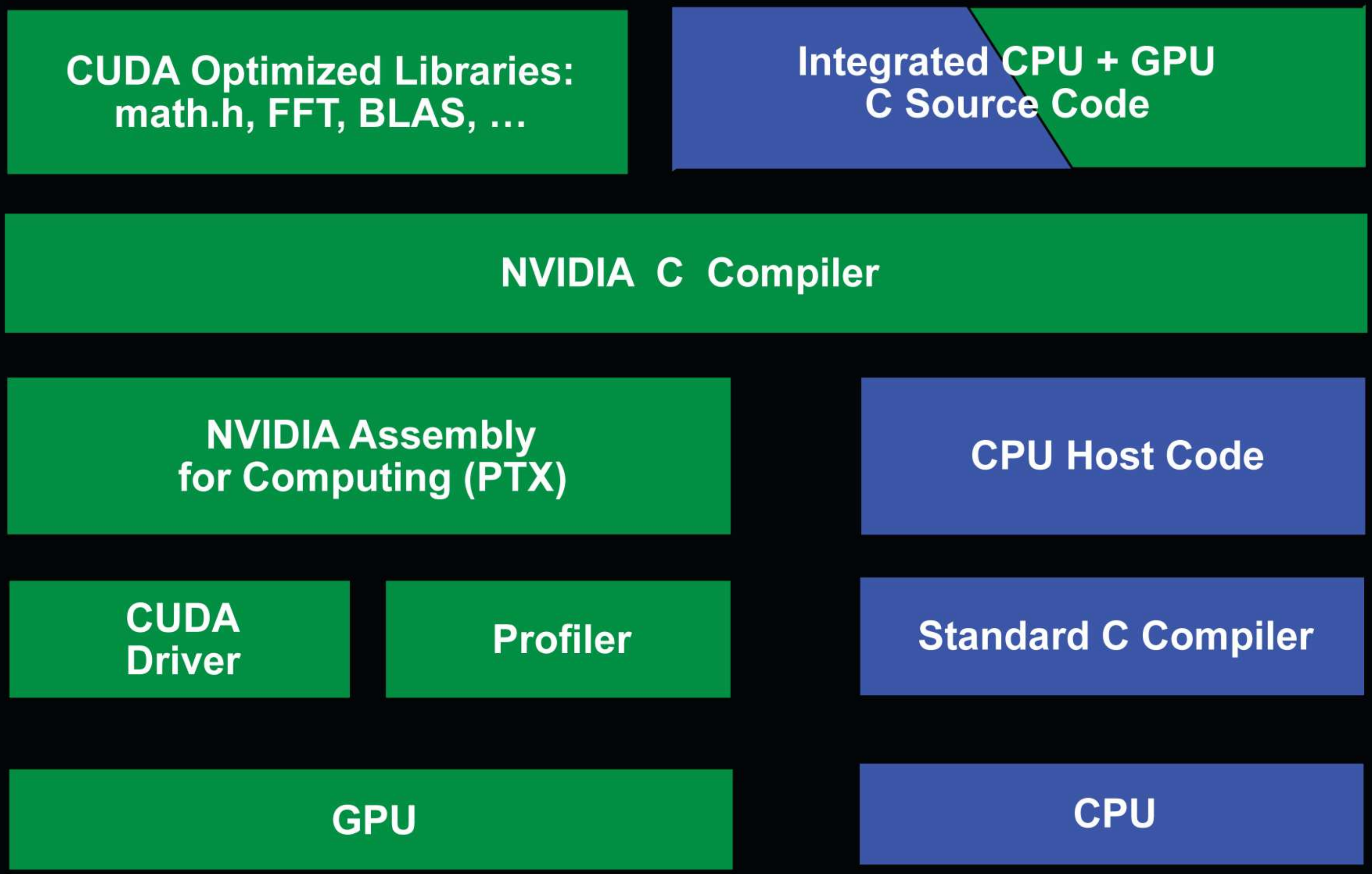
```
…
float x =
input[threadIdx];
float y = func(x);
output[threadIdx] = y;
…
```

CUDA Memory Model and Usage

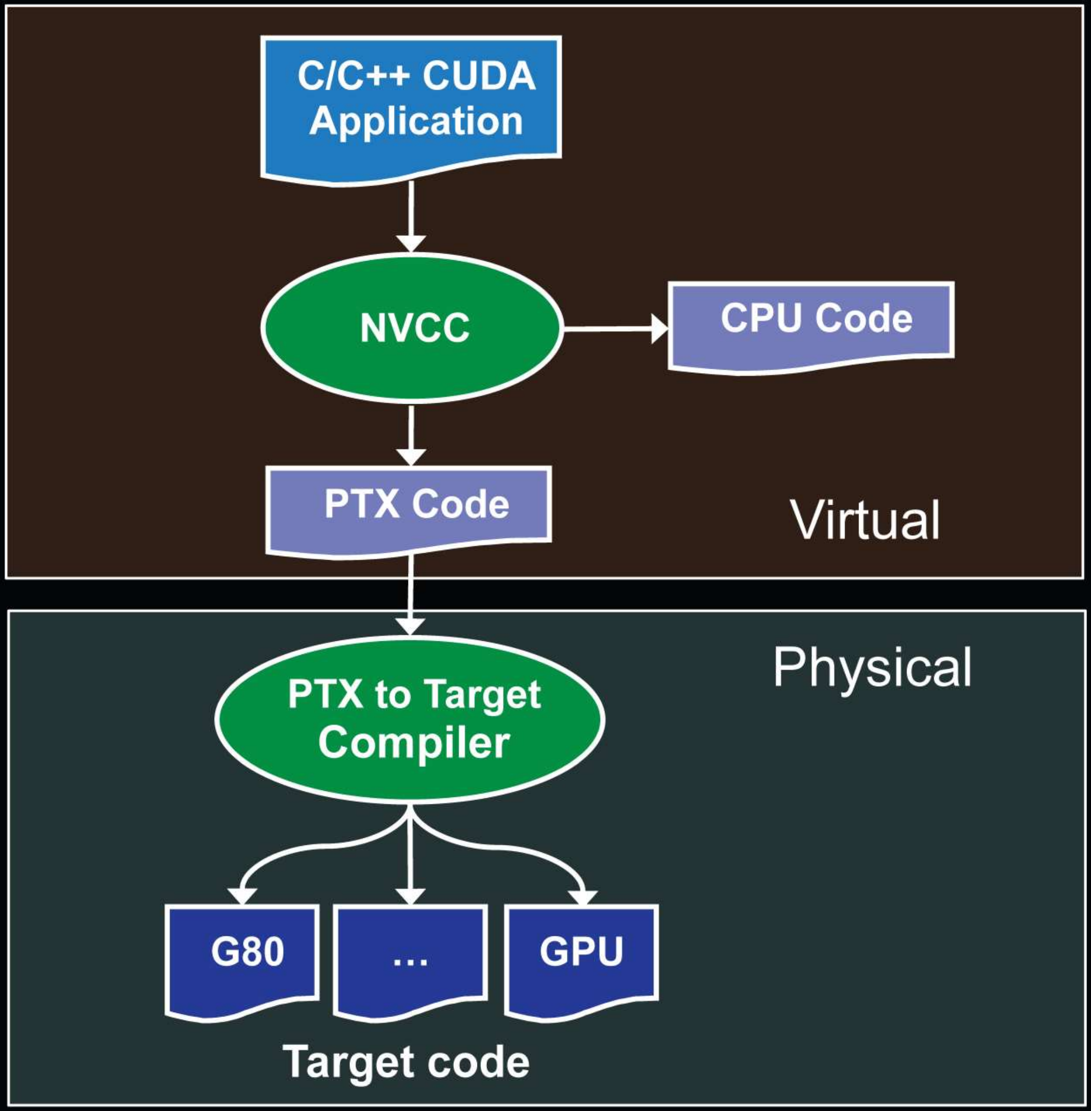
- **`cudaMalloc(), cudaFree()`**
- **`cudaMallocArray(), cudaMalloc2DArray(), cudaMalloc3DArray()`**
- **`cudaMemcpy()`**
- **`cudaMemcpyArray()`**
- Host ↔ host
  Host ↔ device
  Device ↔ device
- Asynchronous transfers possible (DMA)



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

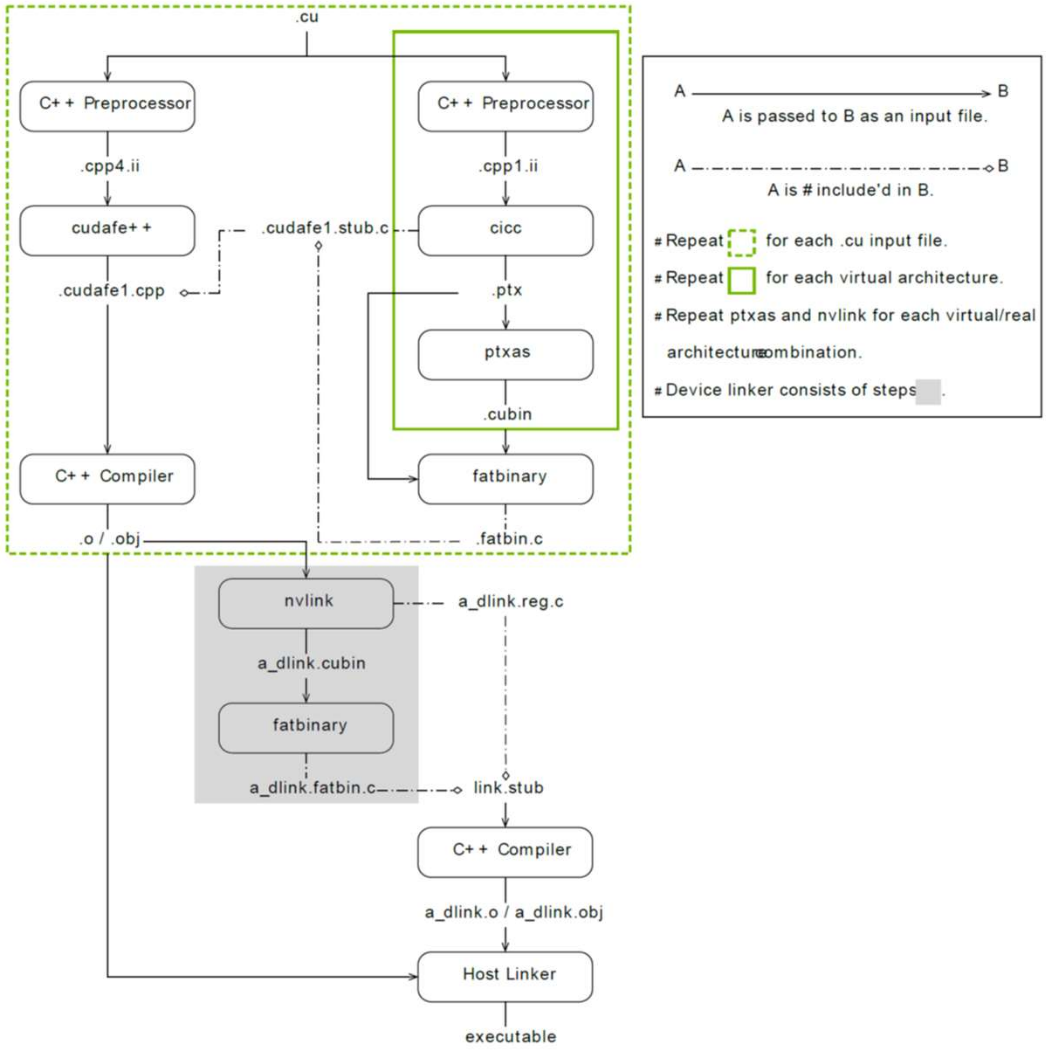
Compiling CUDA Code



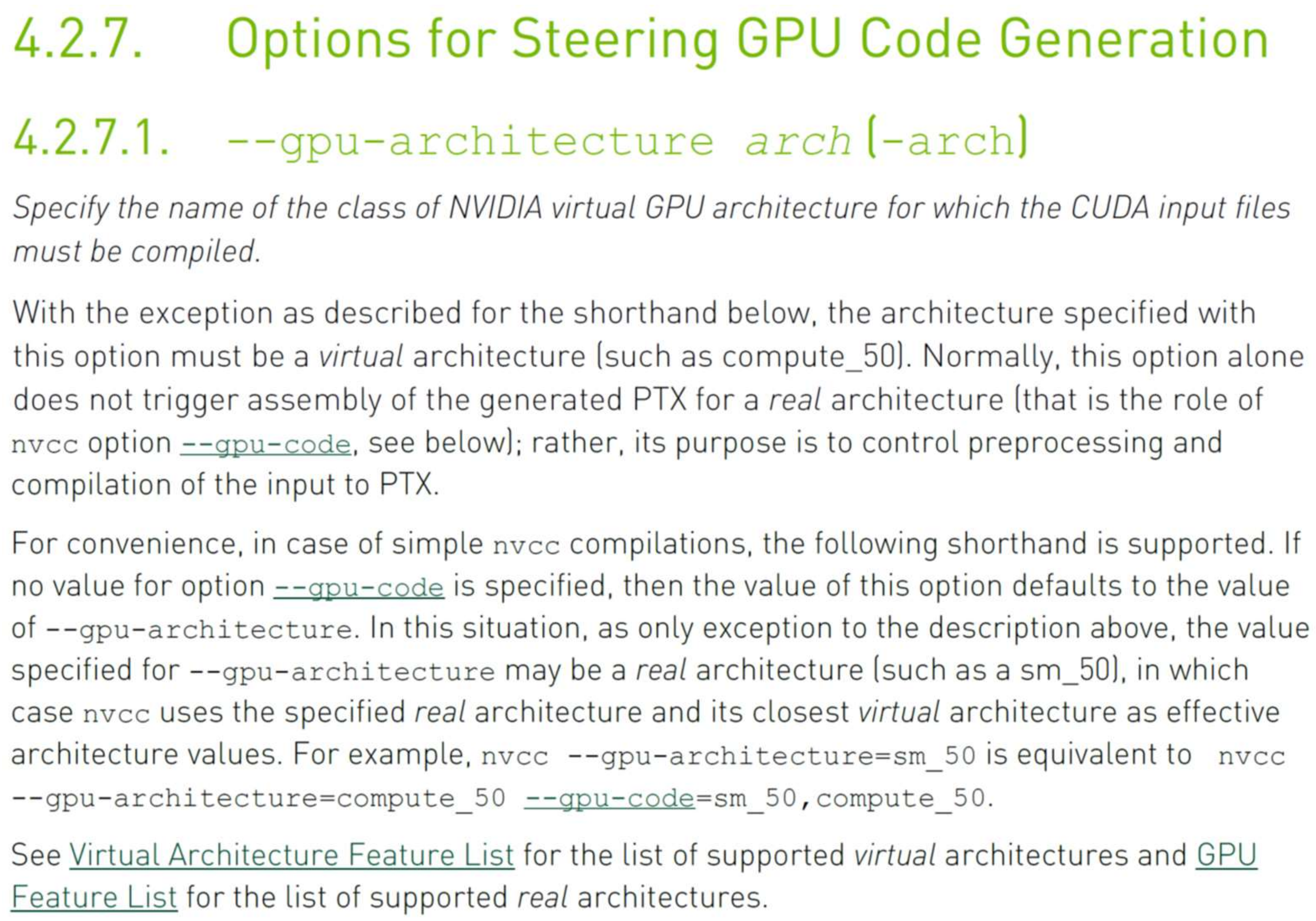
CUDA Compilation Trajectory

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`)

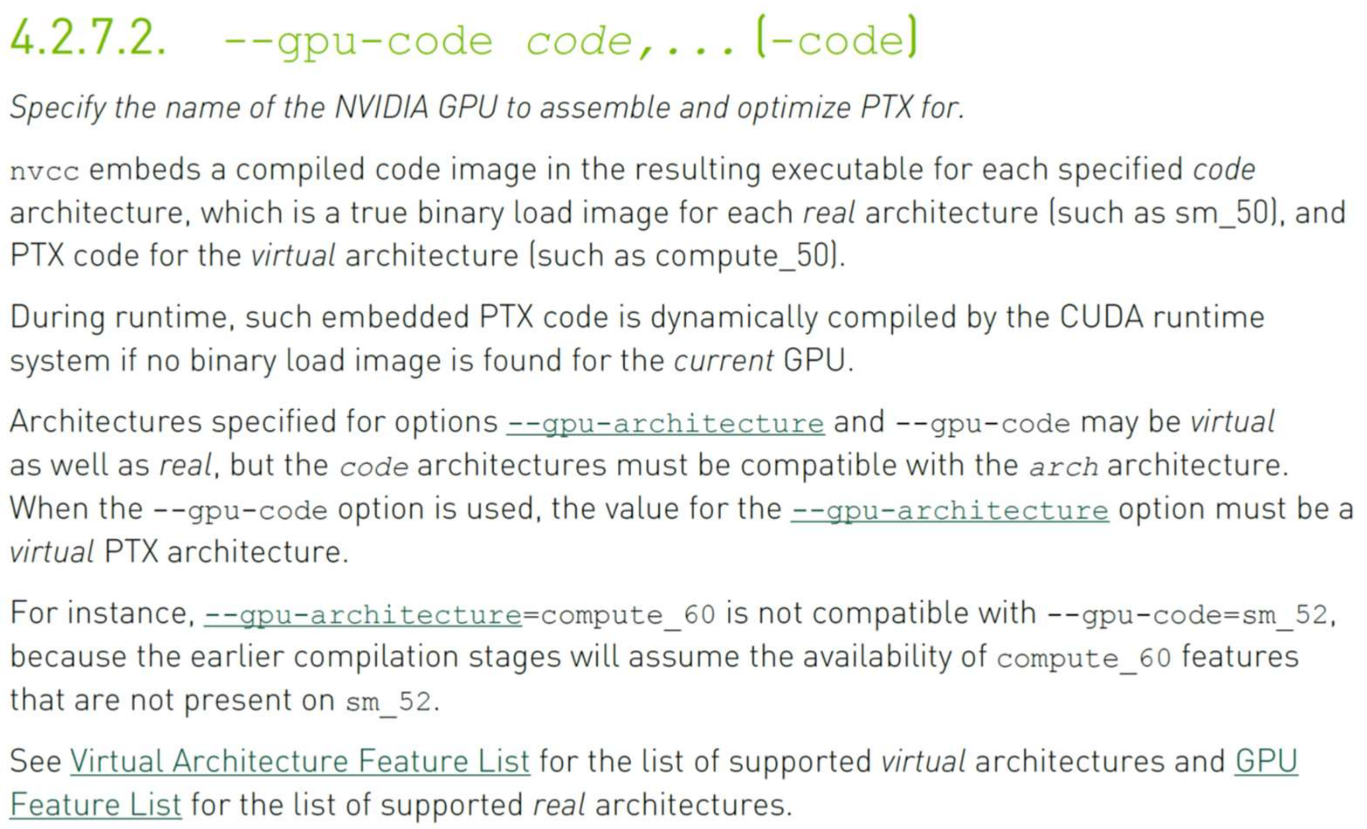
Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

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.

Look at compatibility guides:

`https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf`

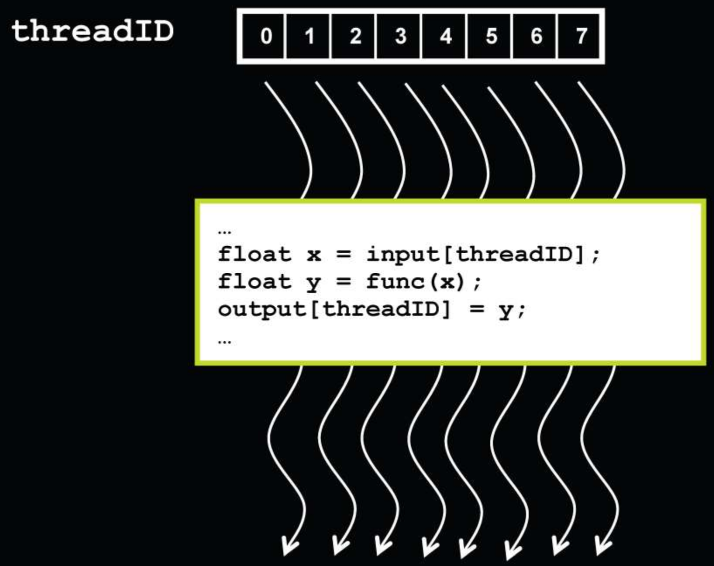
CUDA Kernels and Threads

- 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

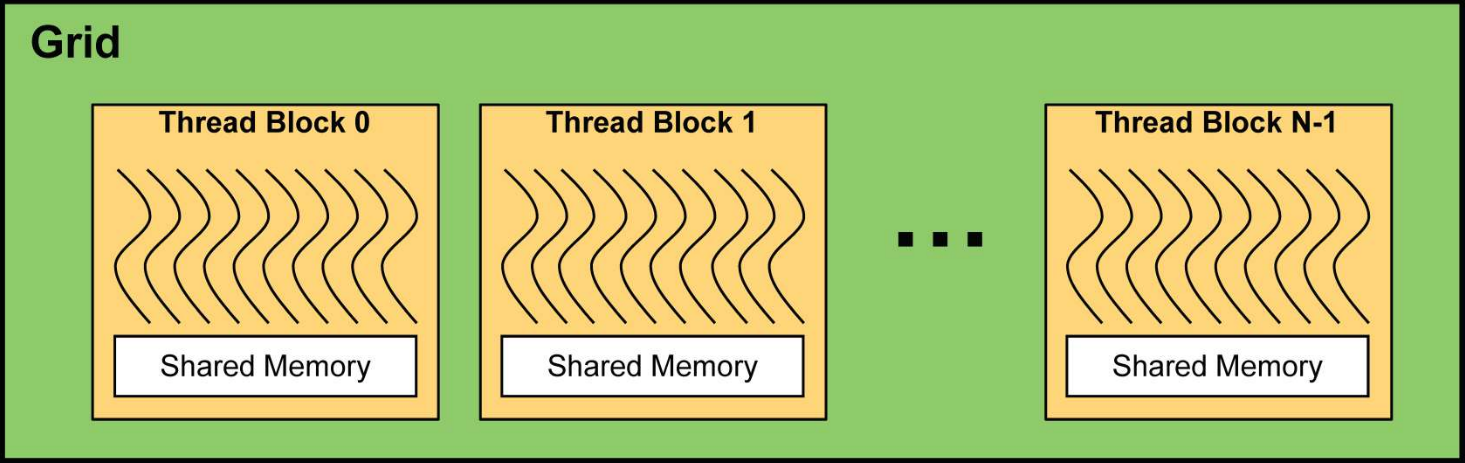
- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



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

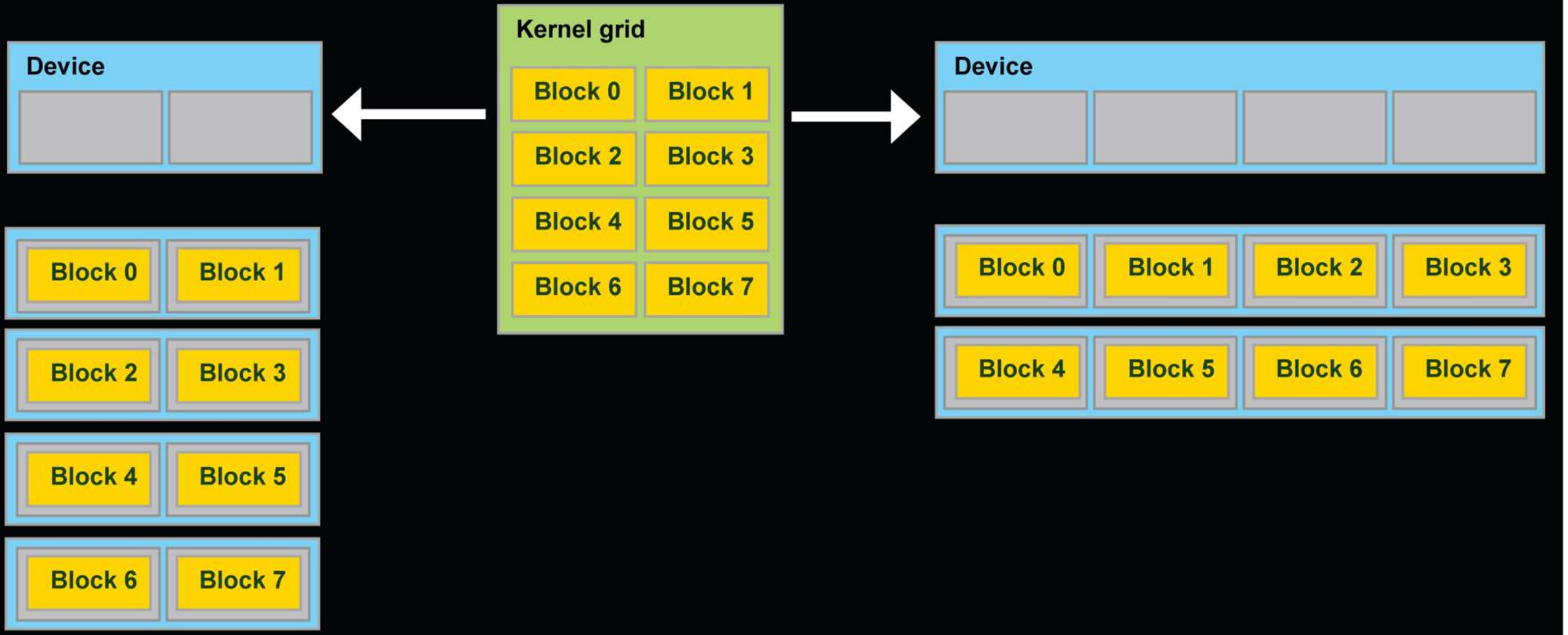
Thread Batching

- 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



Transparent Scalability

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

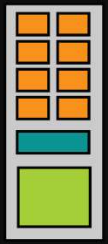


Execution Model

Software **Hardware**

Threads are executed by thread processors

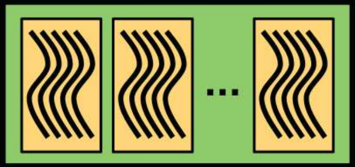
Thread Block

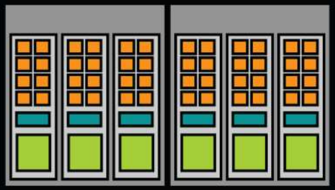


Thread blocks are executed on multiprocessors

Thread blocks do not migrate

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



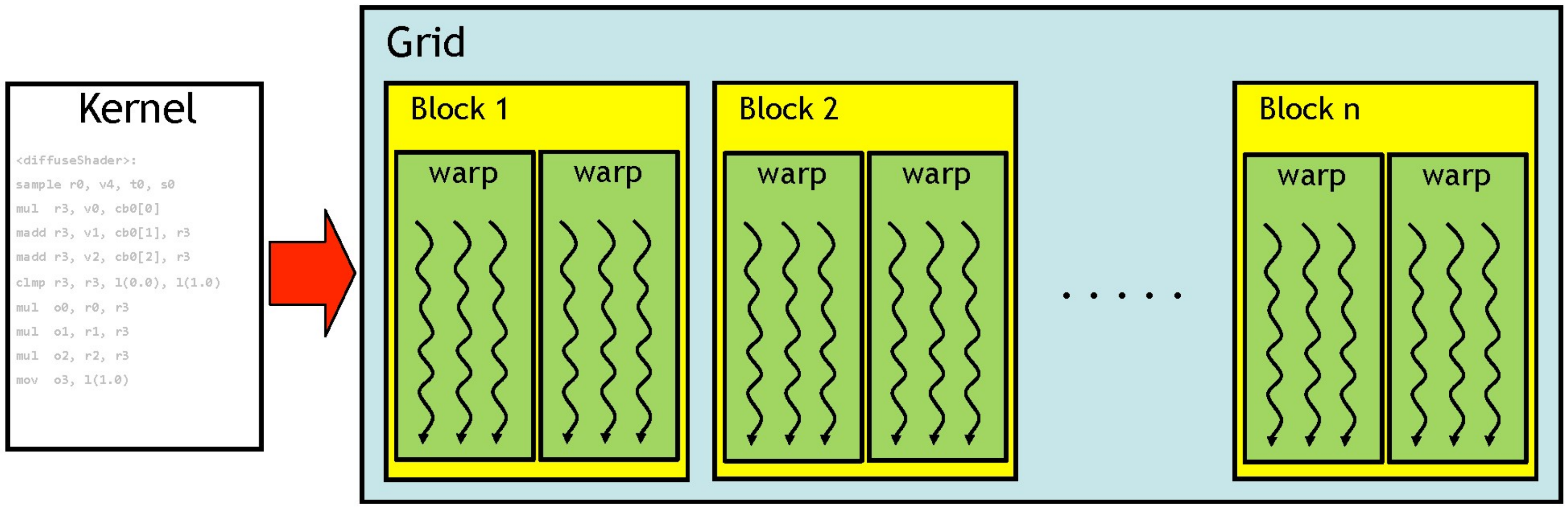


A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

CUDA Programming Model

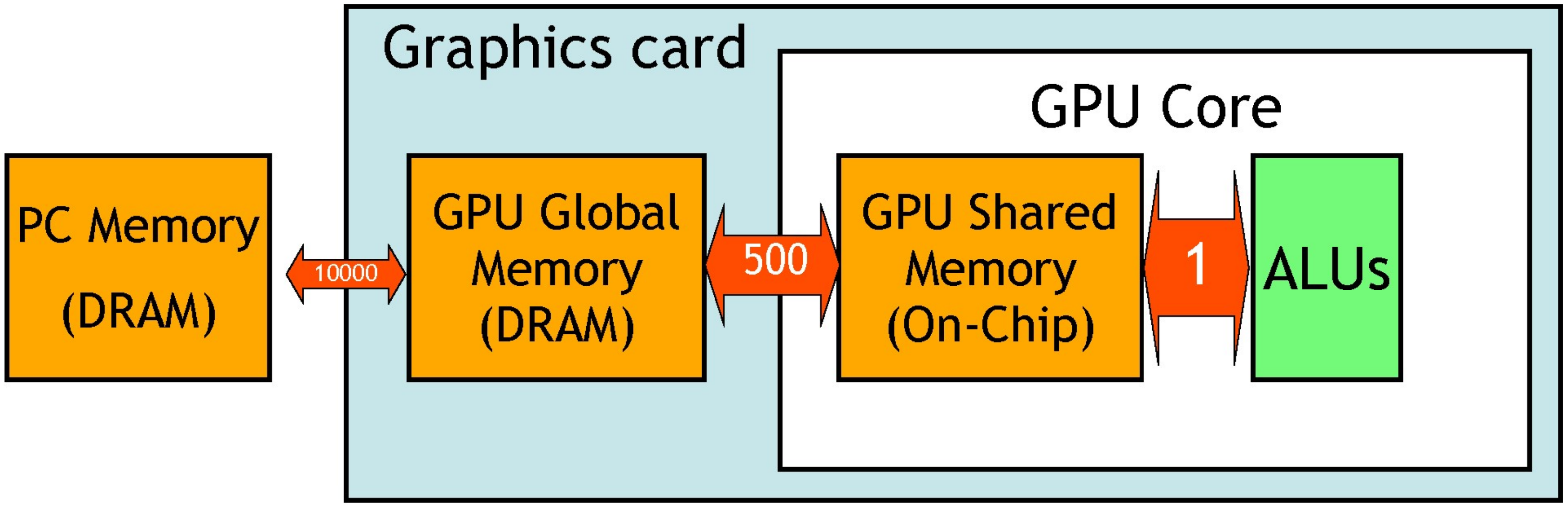
- Kernel
  - GPU program that runs on a thread grid
- Thread hierarchy
  - Grid : a set of blocks
  - Block : a set of warps
  - Warp : a SIMD group of 32 threads
  - Grid size * block size = total # of threads



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

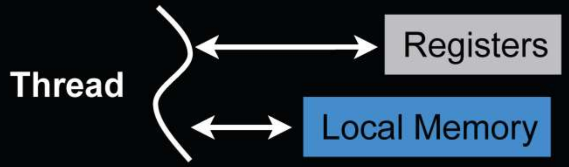
CUDA Memory Structure

- 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

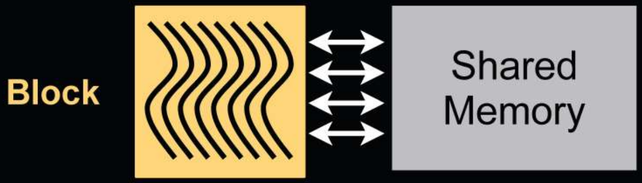


On-chip

Off-chip, uncached

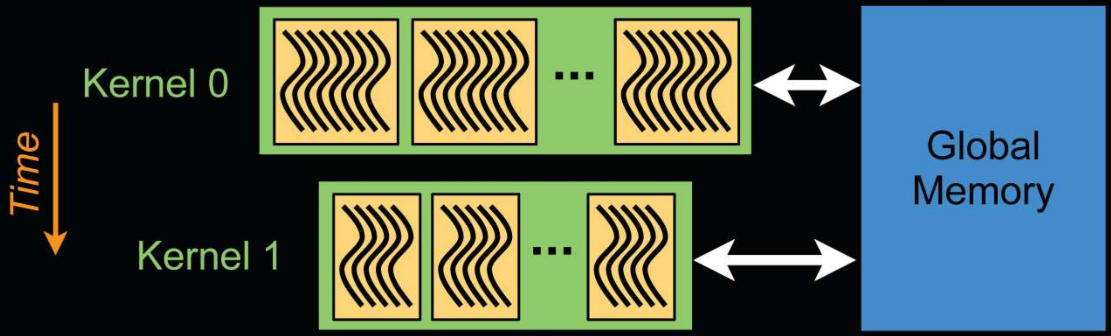
cached on Fermi or newer

- Per-block



- On-chip, small
- Fast

- Per-device



- Off-chip, large
- Uncached
- Persistent across kernel launches
- Kernel I/O

cached on Fermi or newer

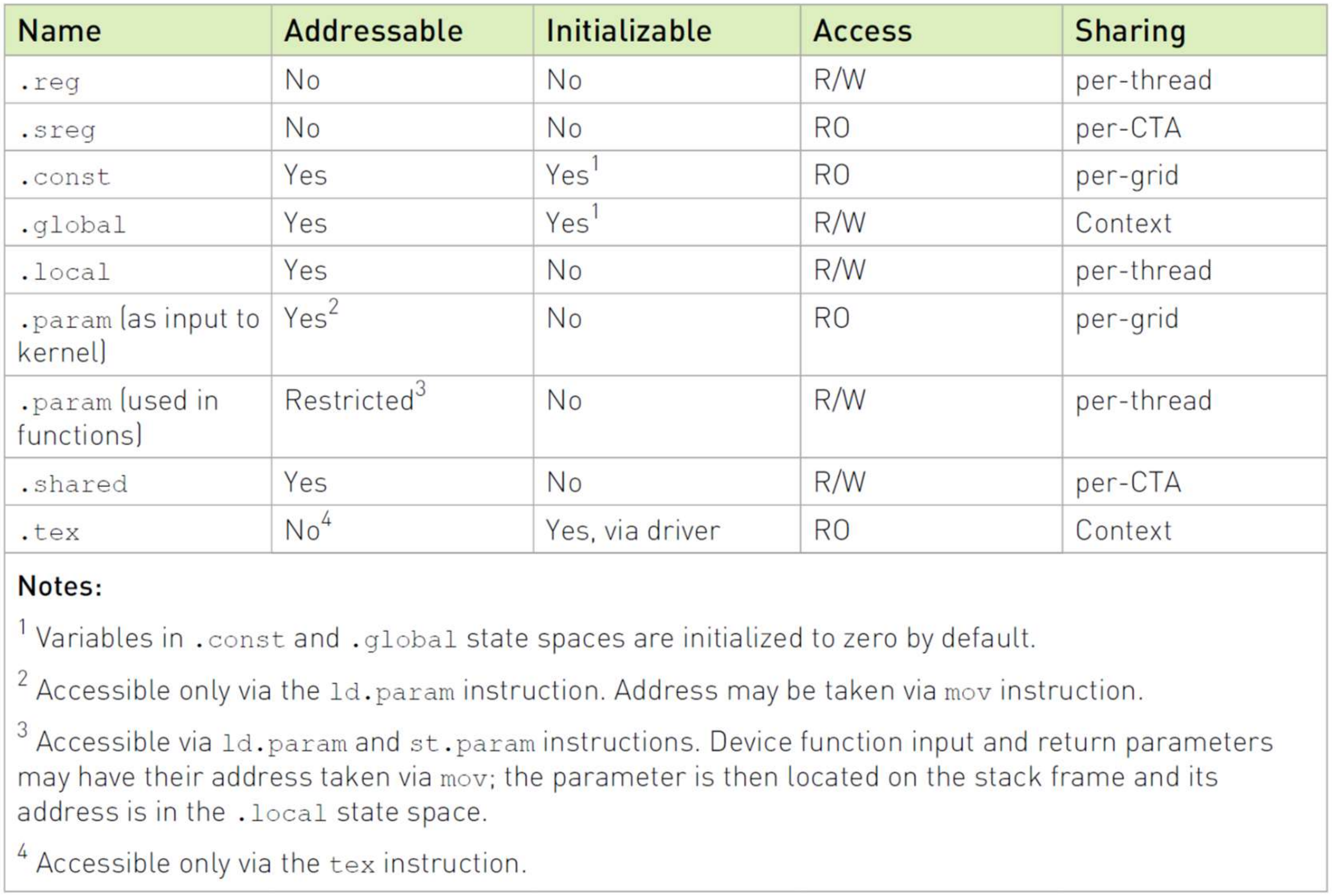
Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No*	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No*	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 7.4 (Chapter 5)



Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-CTA
`.tex`	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.

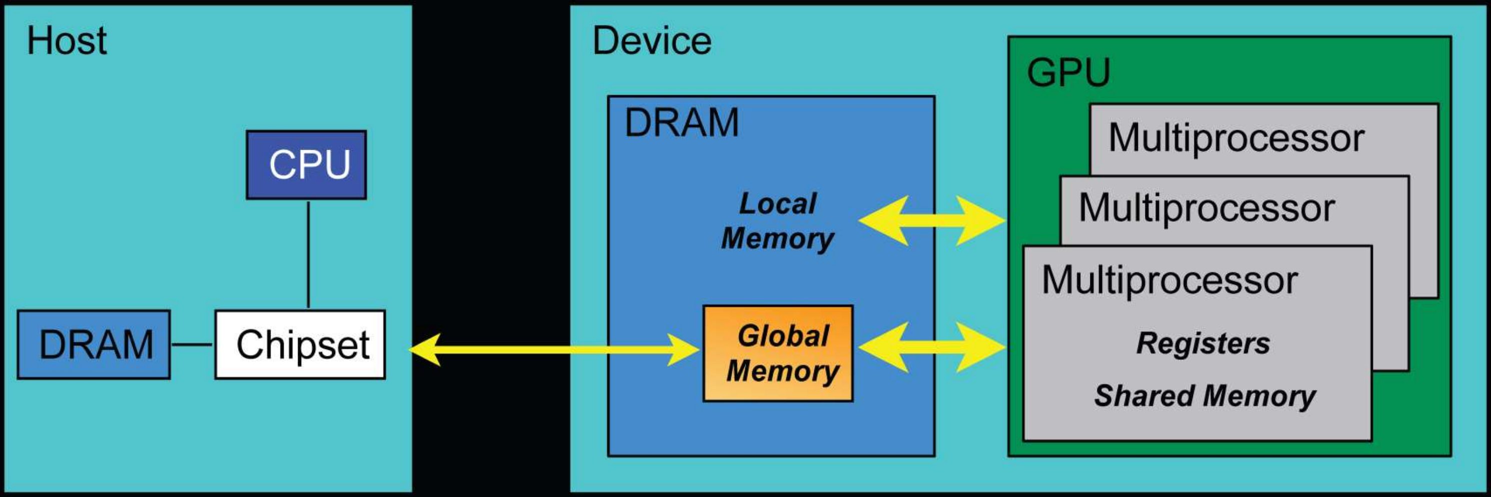
[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

[4] Accessible only via the `tex` instruction.

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

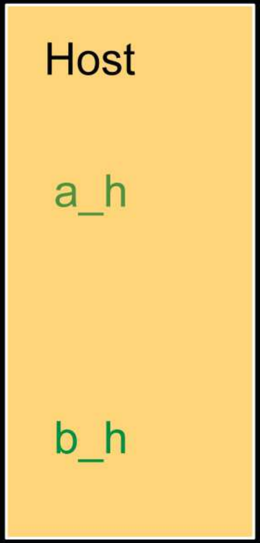
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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;
}
```



Data Movement Example

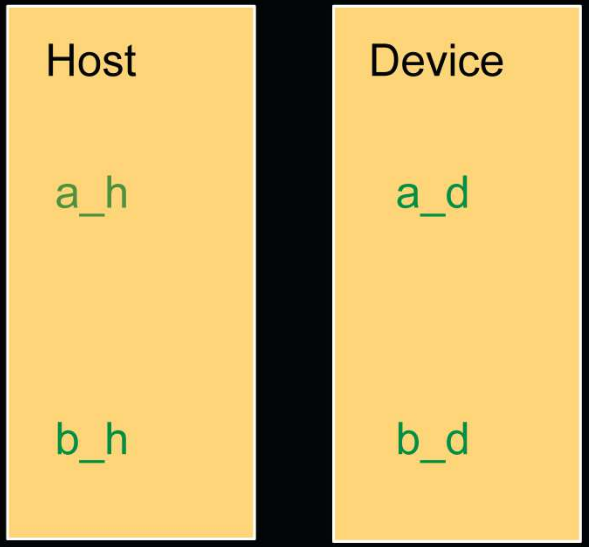
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

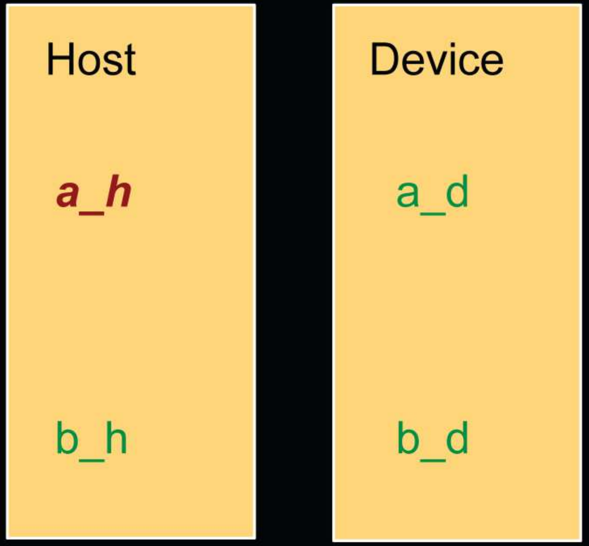
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

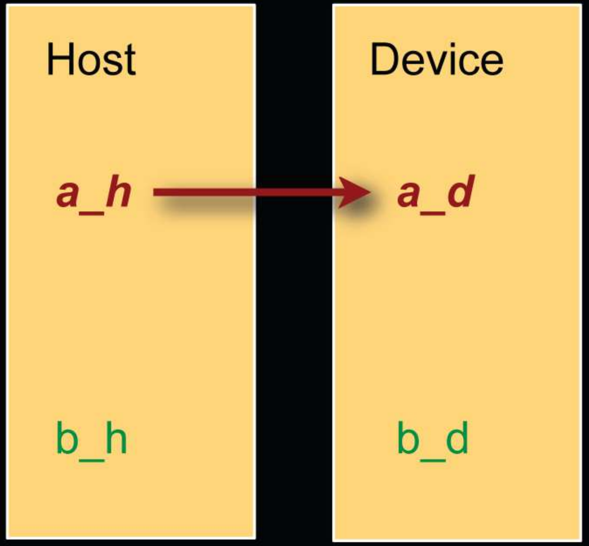
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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;
}
```



Data Movement Example

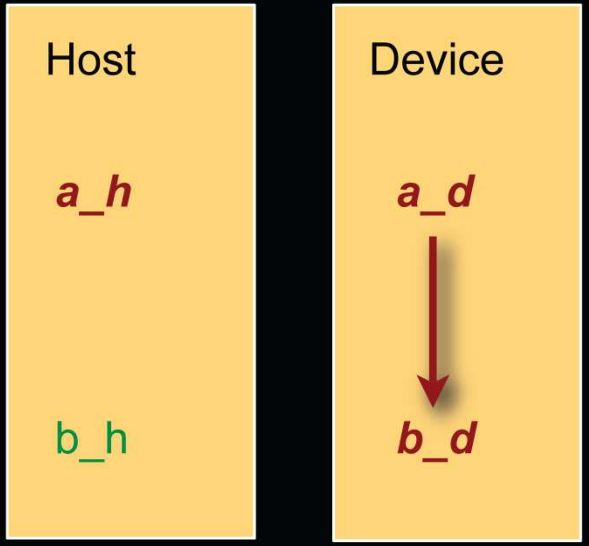
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

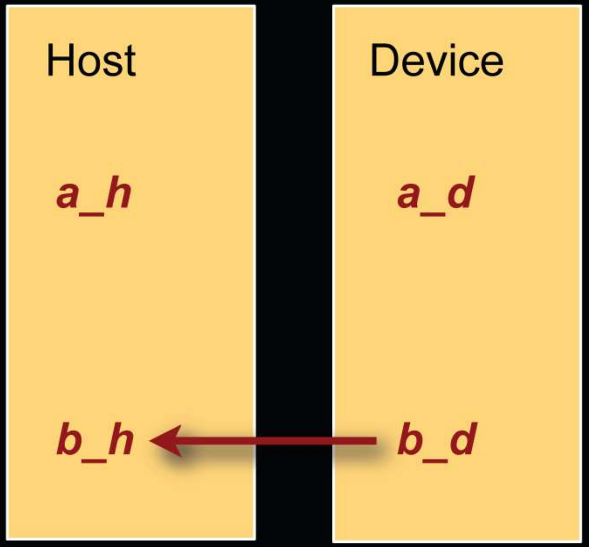
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

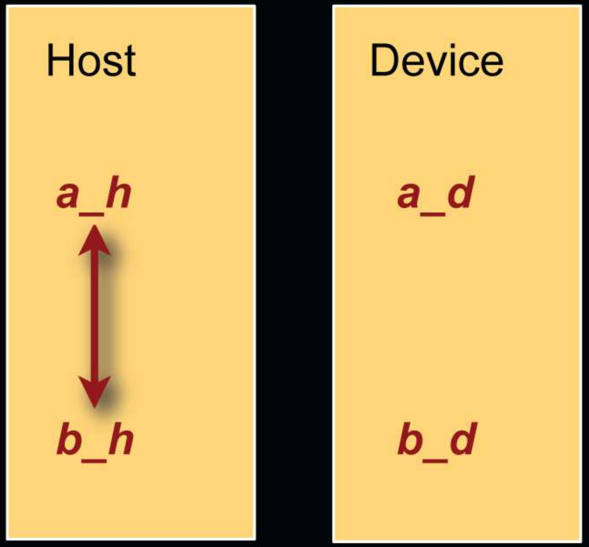
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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;
}
```



Data Movement Example

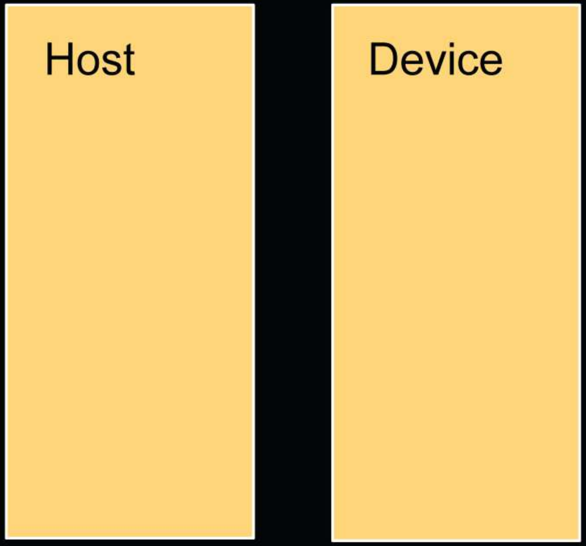
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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;
}
```



Thank you.