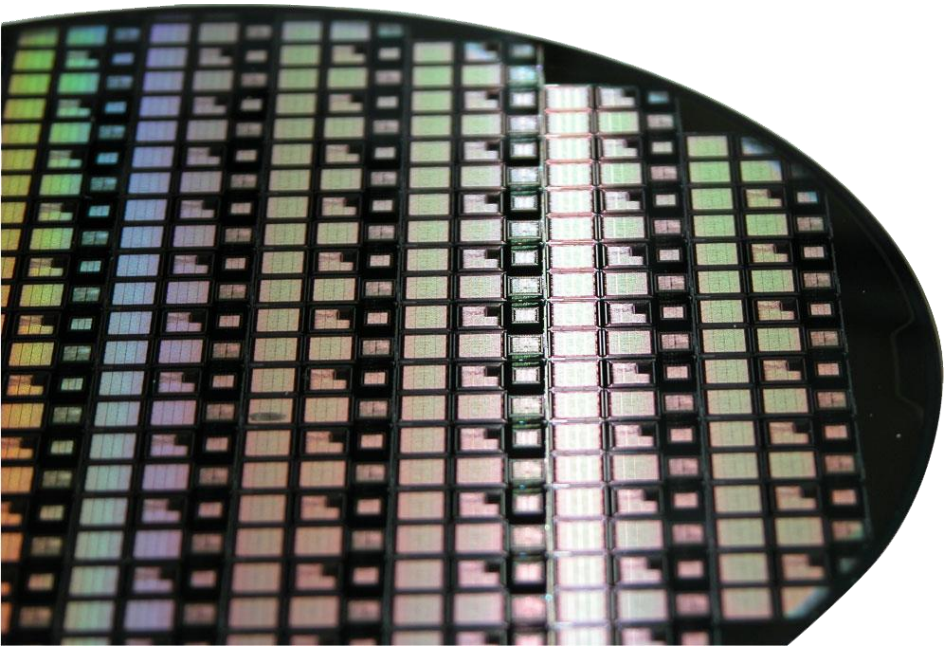
ETH zürich

GPU Programming

CUDA Programming Model

10.05.2016

Lukas Cavigelli
Daniele Palossi



ETH zürich
Integrated Systems Laboratory

Motivation

- Powerful
- Cheap
- Simple
- Efficient

	spGFLOPS	Price	Power	release
NV Titan X	7000	1150$	230 W	15Q1
NV GTX 780	3977	330$	230 W	13Q2
Intel E5-1620v2	118	280$	100 W	13Q3
NV Tegra X1	512	?	10 W	15Q2
NV Tegra K1	326	?	10 W	14Q3

- Size
- Embeddability
- Availability

Outline

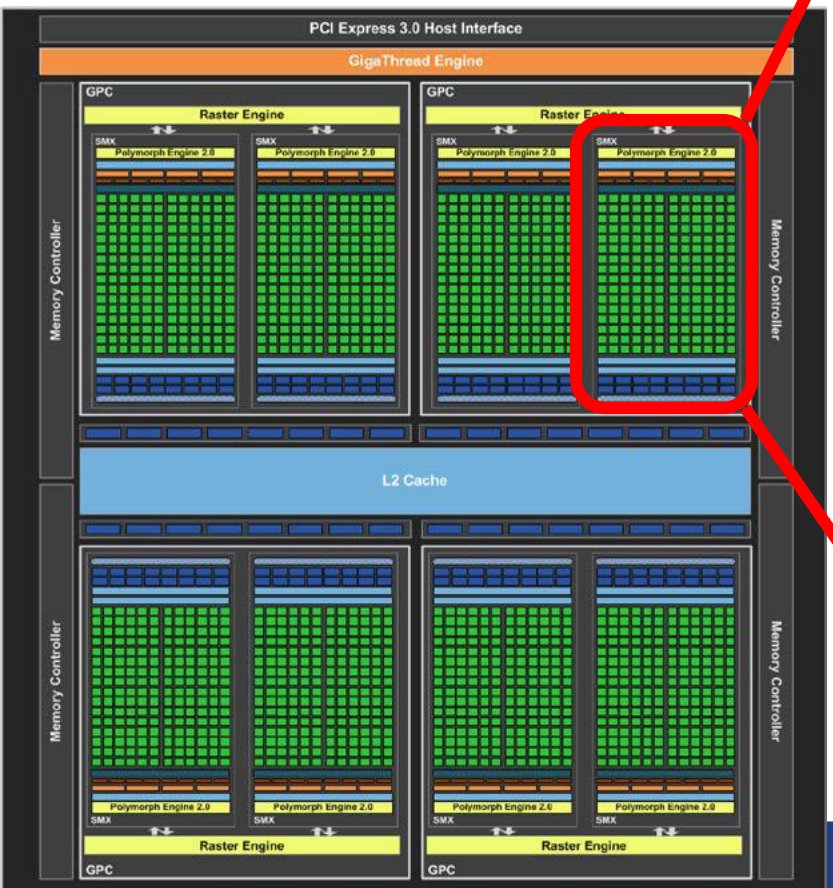
- GPU architecture recap
  - Computation model
  - Memory hierarchy
- CUDA basics
  - Threads, blocks, grids
  - Memory model
  - Synchronization
- Laboratory exercises
  - GPU info
  - Hello World!
  - Vector addition
  - Matrix Multiplication
  - …

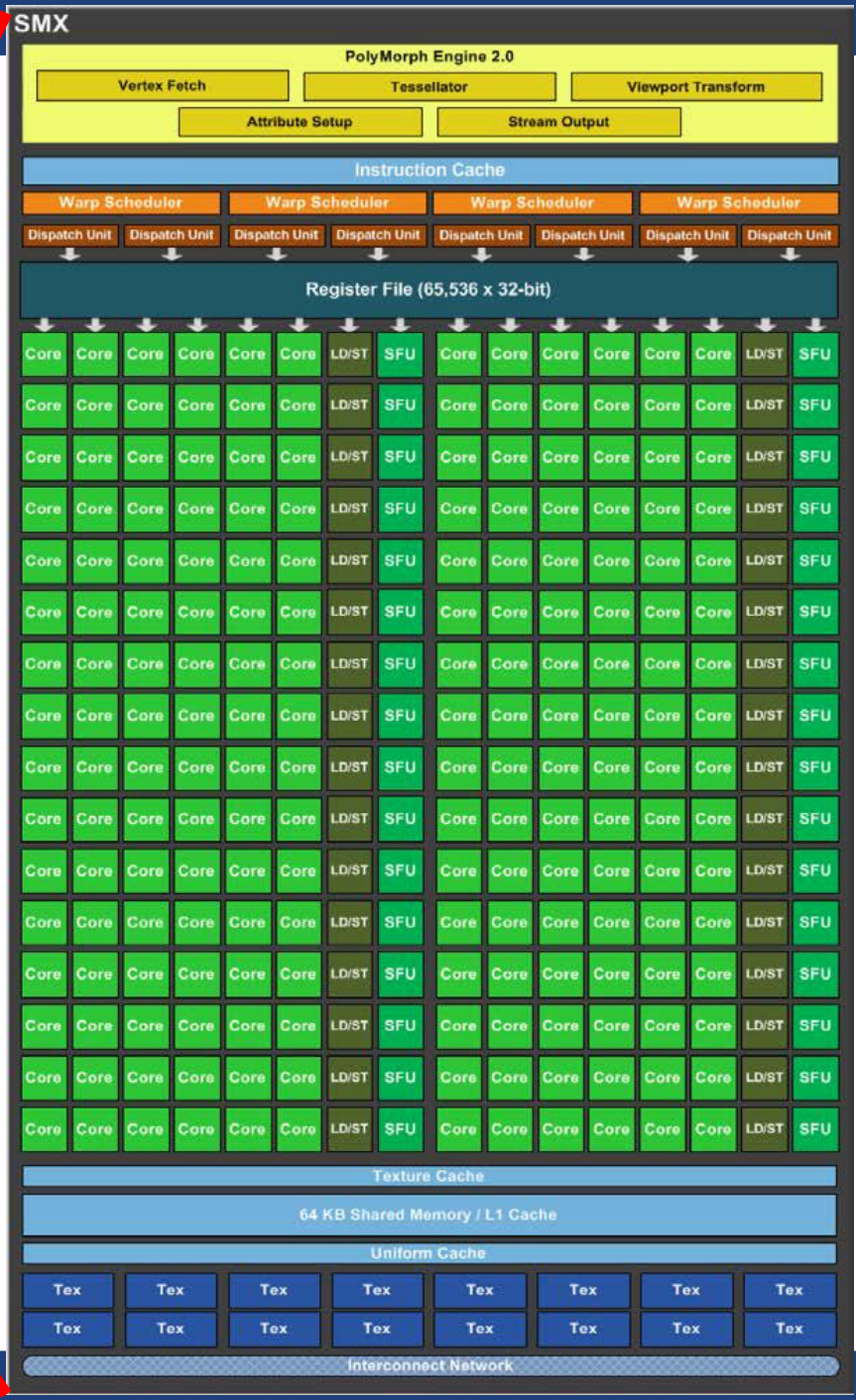
Outline

- **GPU architecture recap**
  - **Computation model**
  - **Memory hierarchy**
- CUDA basics
  - Threads, blocks, grids
  - Memory model
  - Synchronization
- Laboratory exercises
  - GPU info
  - Hello World!
  - Vector addition
  - Matrix Multiplication
  - …

NVIDIA Kepler

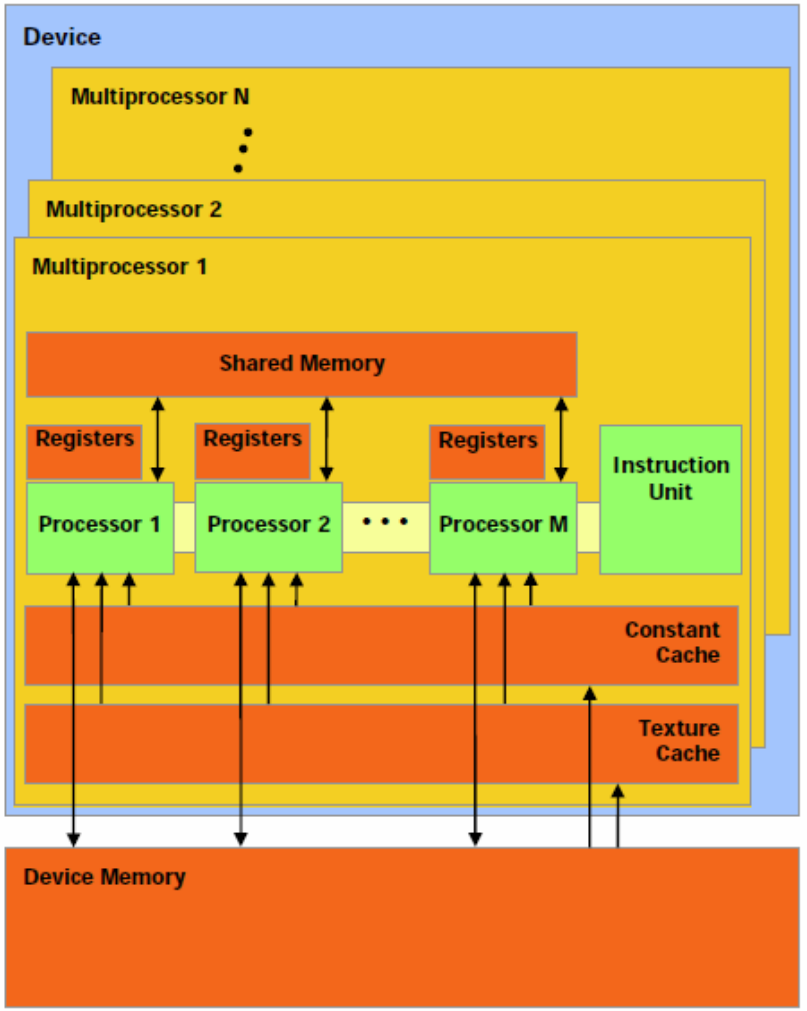
- Three key ideas
  - Use many "slimmed down cores" to run in parallel
  - Pack cores full of ALUs (by sharing instruction stream across groups of work items)
  - Avoid latency stalls by interleaving execution of many groups of work-items/ threads/
    - When one group stalls, work on another group





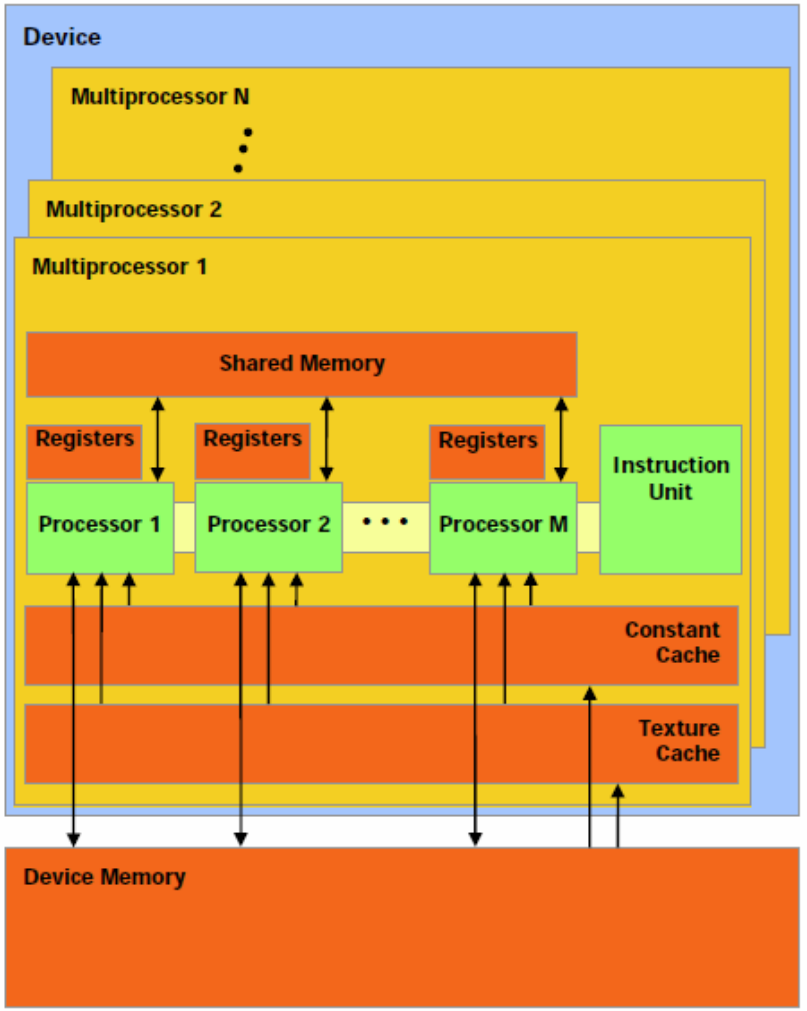
On-chip memory

- Each multiprocessor has on-chip memory of the four following types:
  - One set of local 32-bit ***registers per processor***,
  - A parallel ***shared memory*** *shared by all scalar processor cores*,
  - A **read-only *constant cache*** *shared by all cores:*
    - *speeds* up reads from the constant memory space, which is a read-only region of device memory,
  - A **read-only *texture cache*** *shared by all cores:*
    - *speeds up* reads from the texture memory space, which is a read-only region of device memory;
    - Accessable directly or through texture units u*nit that* implement the various addressing modes and data filtering.
- The local and global memory spaces are read-write regions of device memory and are not cached.



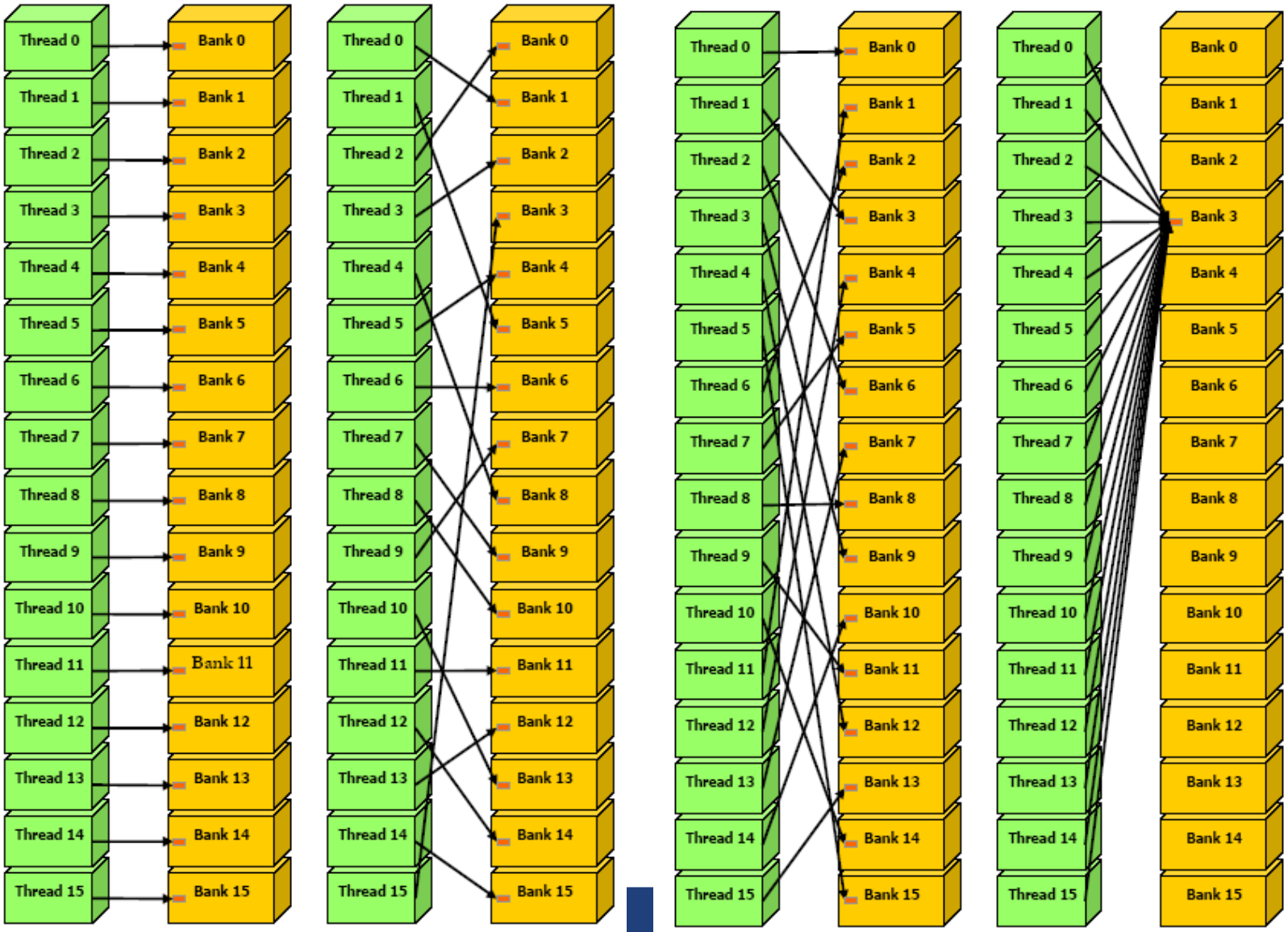
Shared Memory

- Is on-chip:
  - much faster than the global memory
  - divided into equally-sized memory banks
  - as fast as a register when no bank conflicts
- Successive 32-bit words are assigned to successive banks
- Each bank has a bandwidth of 32 bits per clock cycle.



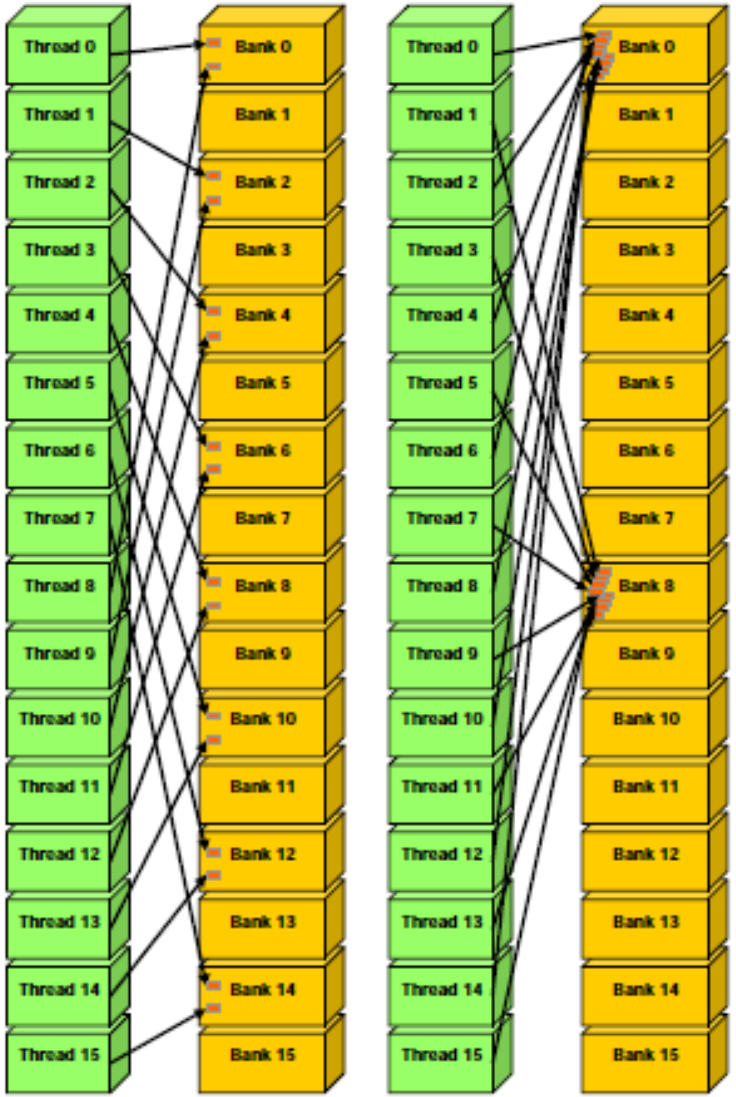
Shared Memory

Examples of Shared Memory Access Patterns
without Bank Conflicts



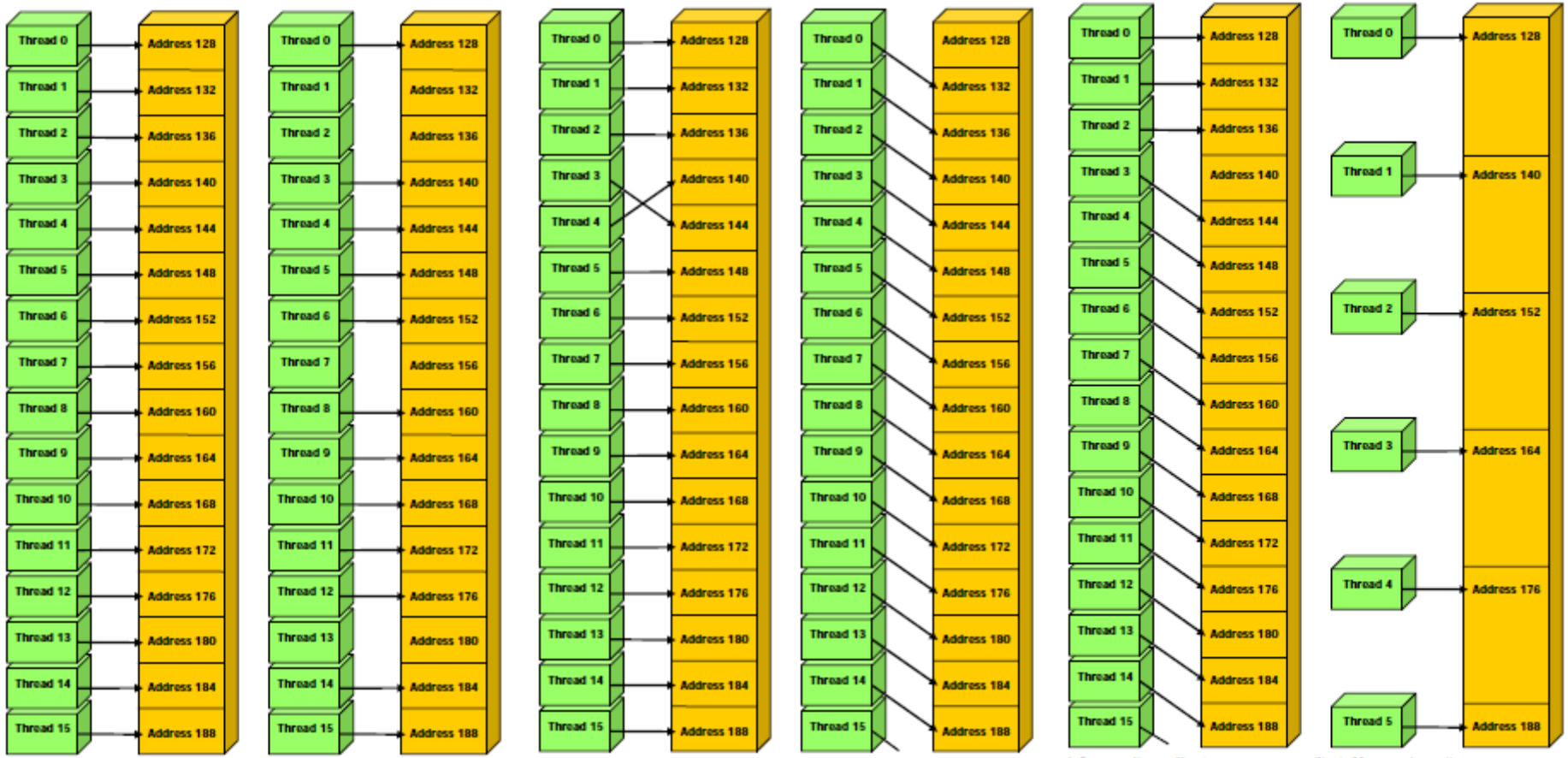
Shared Memory

Examples of Shared Memory Access Patterns with Bank Conflicts



Global Memory: Coalescing

- The device is capable of reading 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction.
- Global memory bandwidth is used most efficiently when the simultaneous memory accesses can be *coalesced into a single memory transaction of 32, 64, or 128* bytes.



Left: coalesced `float` memory access, resulting in a single memory transaction.
Right: coalesced `float` memory access (divergent warp), resulting in a single memory transaction.

Left: non-sequential `float` memory access, resulting in 16 memory transactions.
Right: access with a misaligned starting address, resulting in 16 memory transactions.

Left: non-contiguous `float` memory access, resulting in 16 memory transactions.
Right: non-coalesced `float3` memory access, resulting in 16 memory transactions.

Outline

- GPU architecture recap
  - Computation model
  - Memory hierarchy
- **CUDA basics**
  - **Threads, blocks, grids**
  - **Memory model**
  - **Synchronization**
- Laboratory exercises
  - GPU info
  - Hello World!
  - Vector addition
  - Matrix Multiplication
  - …

What is CUDA?

- CUDA is a **scalable** parallel programming model and a software environment for parallel computing
  - Minimal **extensions** to familiar **C/C++** environment
  - **Heterogeneous** serial-parallel programming model
  - C++ templates for GPU code

- CUDA Architecture
  - Expose GPU computing for general purpose
  - Retain performance

- CUDA defines:
  - Programming model
  - Memory model

Some Design Goals

- Scale up to 100's of cores, 1000's of parallel threads
- Let programmers focus on parallel algorithms
  - Not on the mechanics of a parallel programming language
- Enable heterogeneous systems (i.e. CPU + GPU)
  - CPU and GPU are separate devices with separate DRAMs

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N       1024
#define RADIUS    3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

Device Code

Host Code

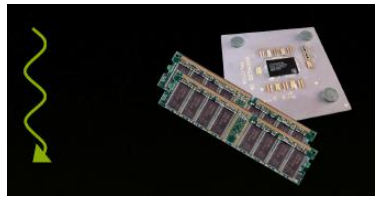
Parallel Function

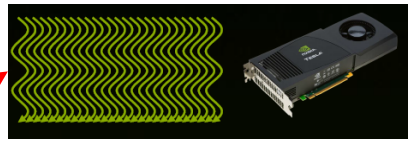
Serial Function

Serial Code

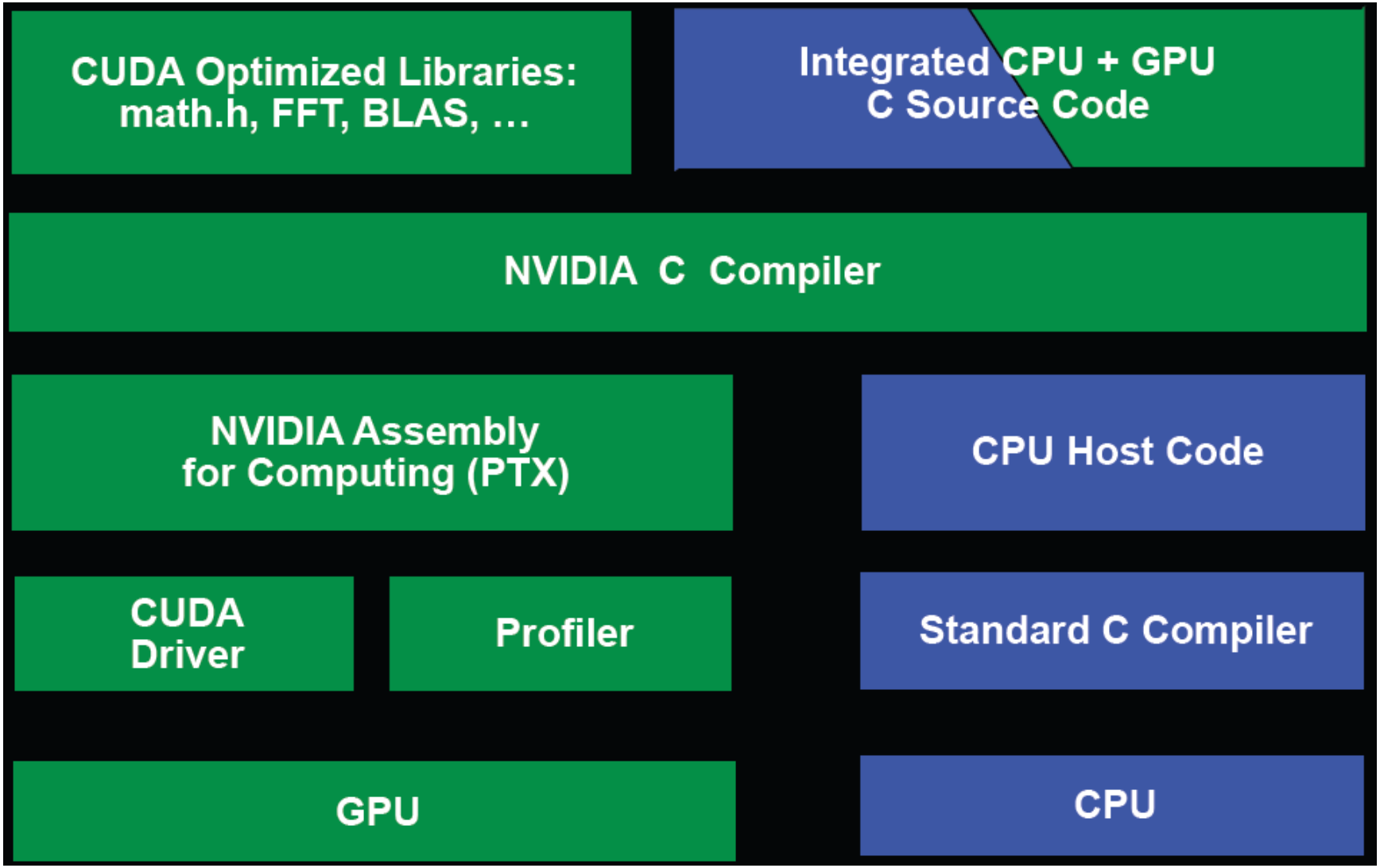
Parallel Code

Serial Code

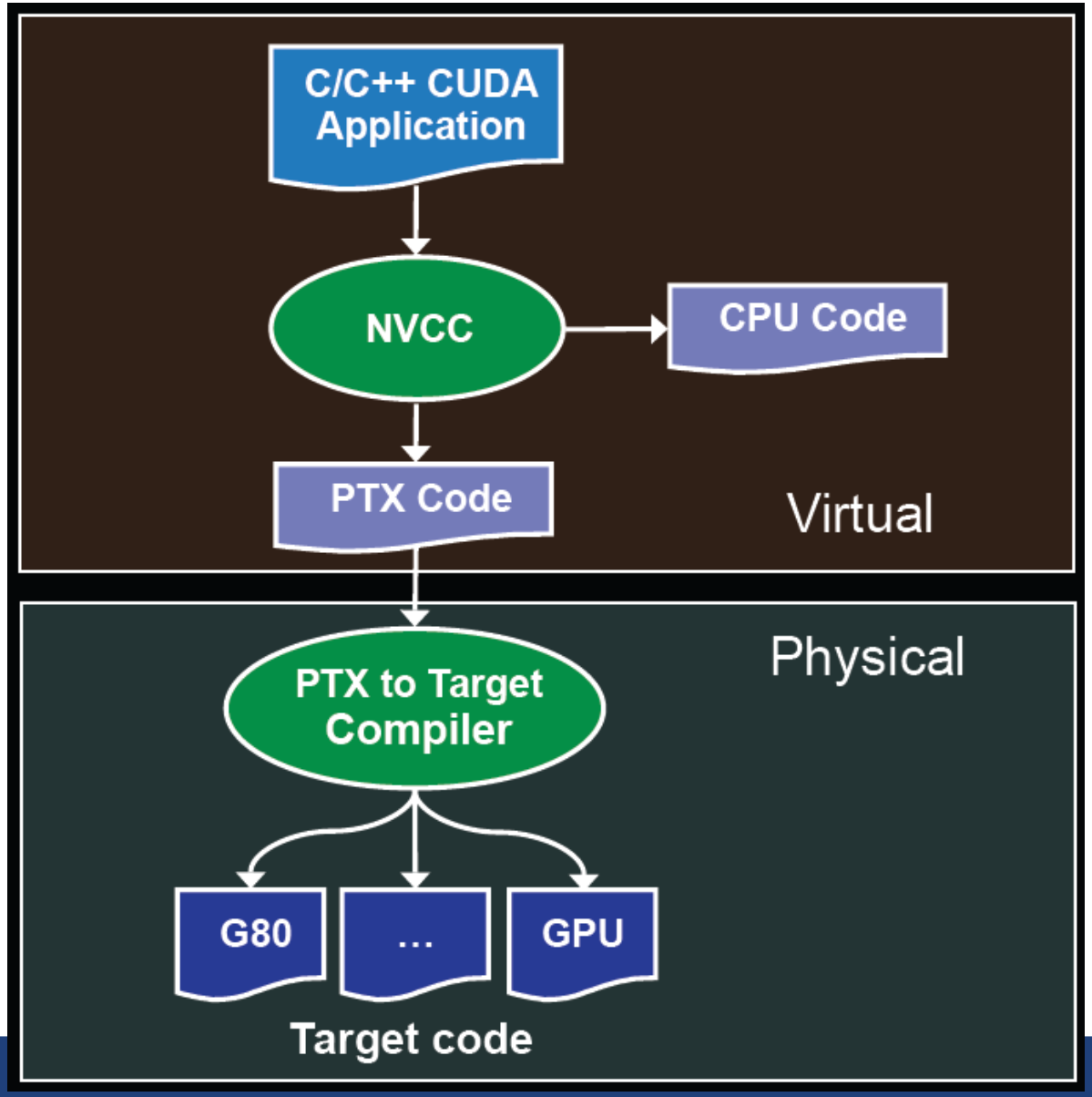




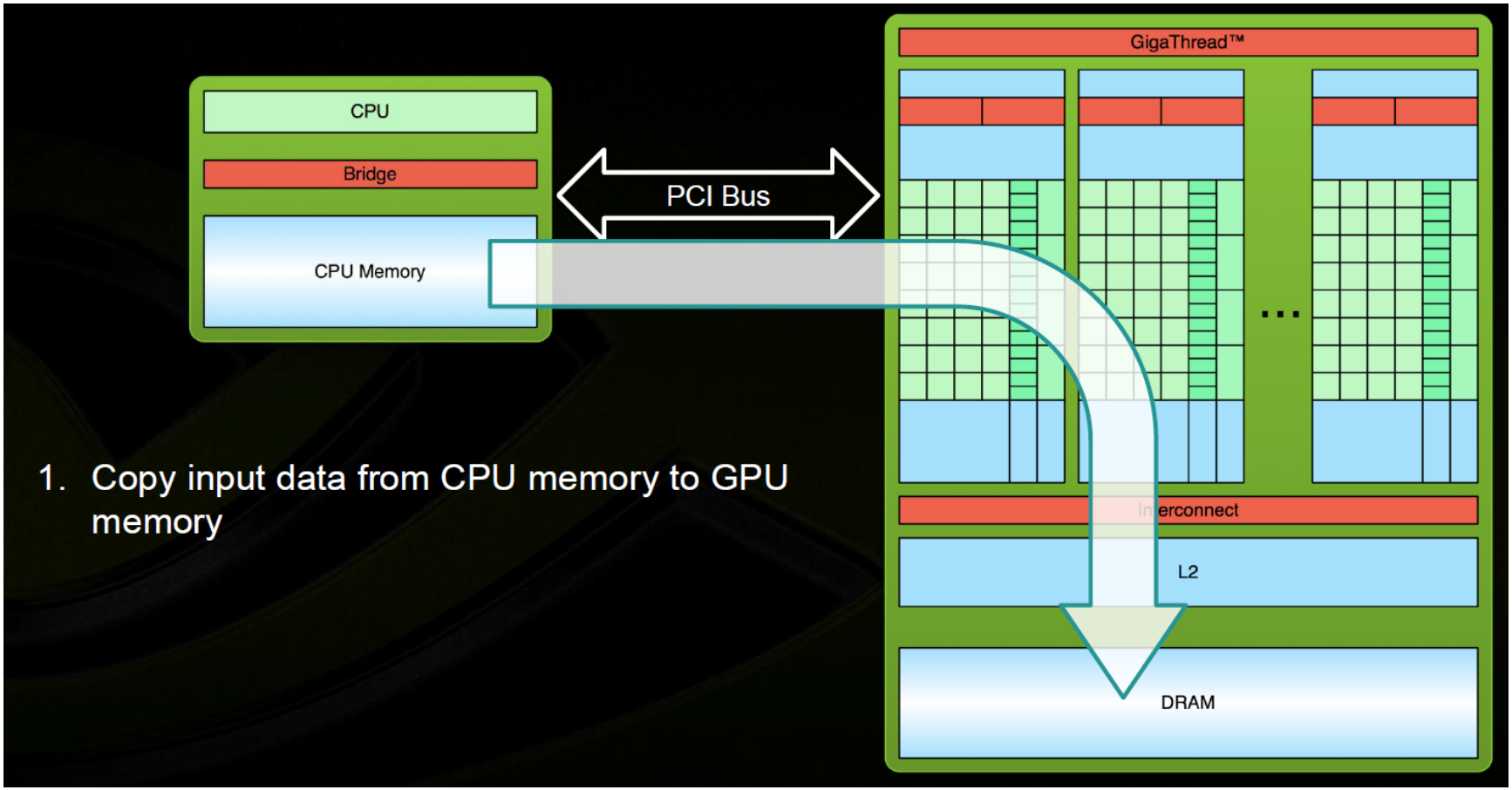
CUDA Software Development



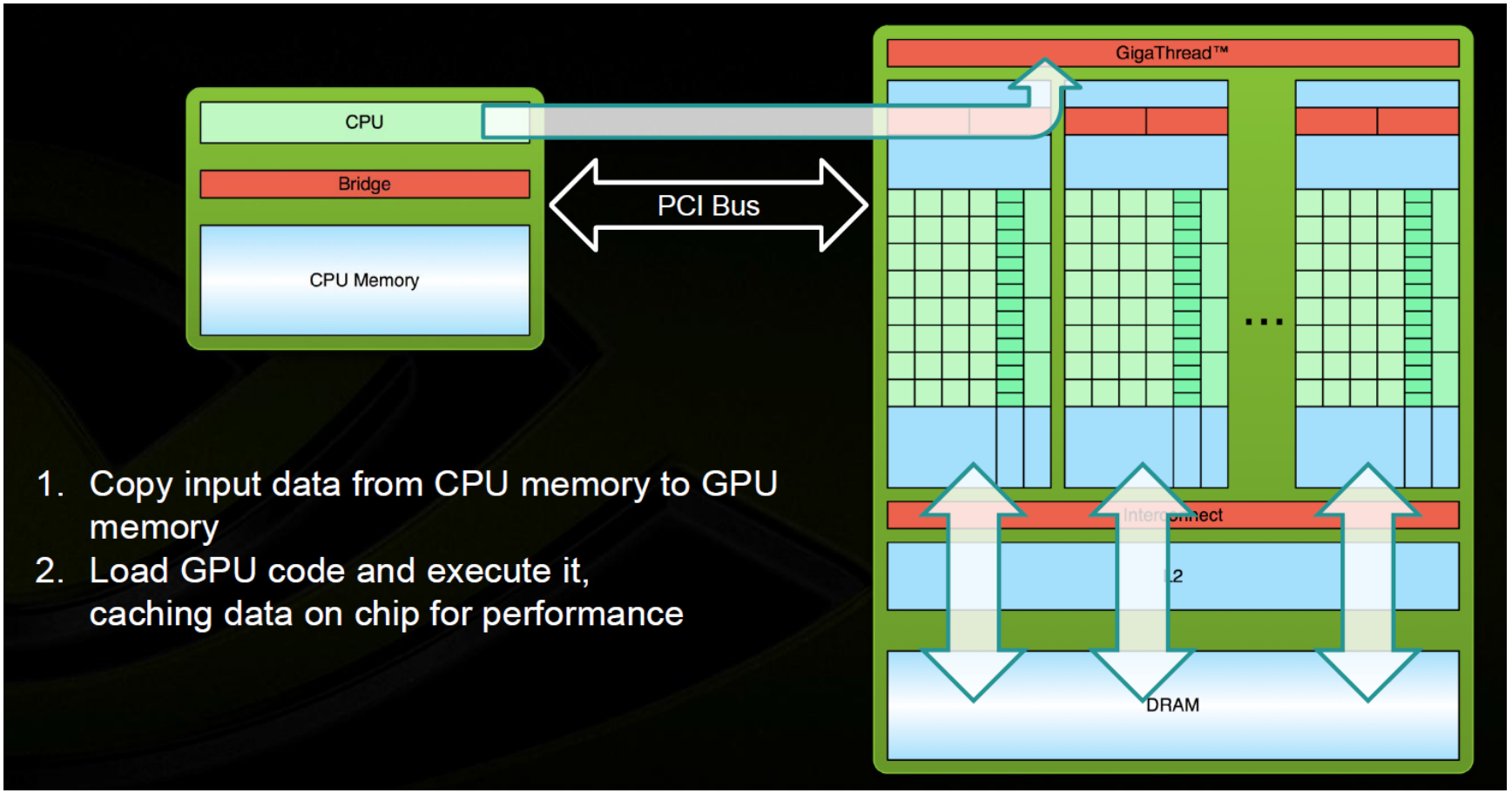
Compiling CUDA Code



Simple Processing Flow

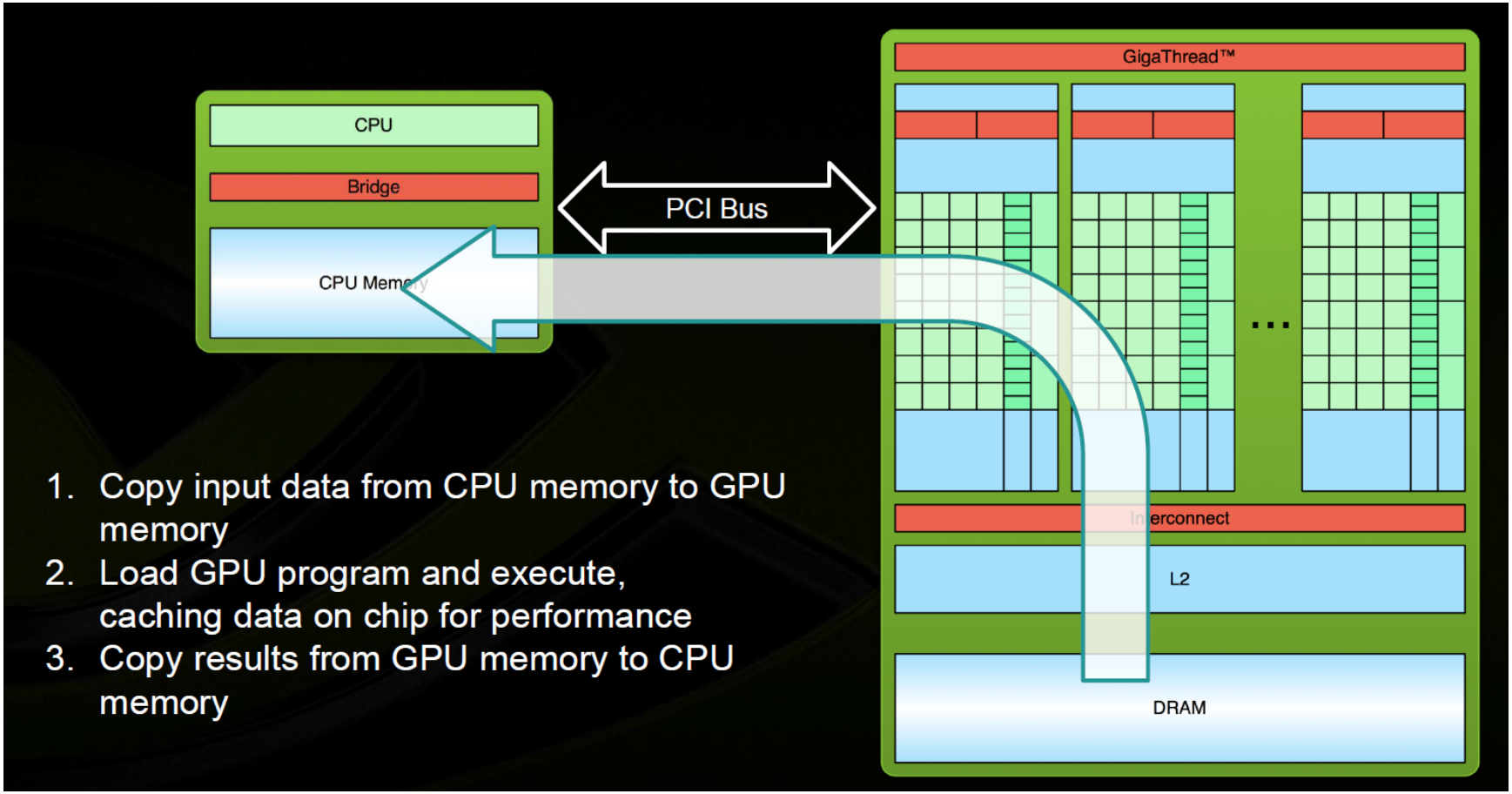


Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

Simple Processing Flow



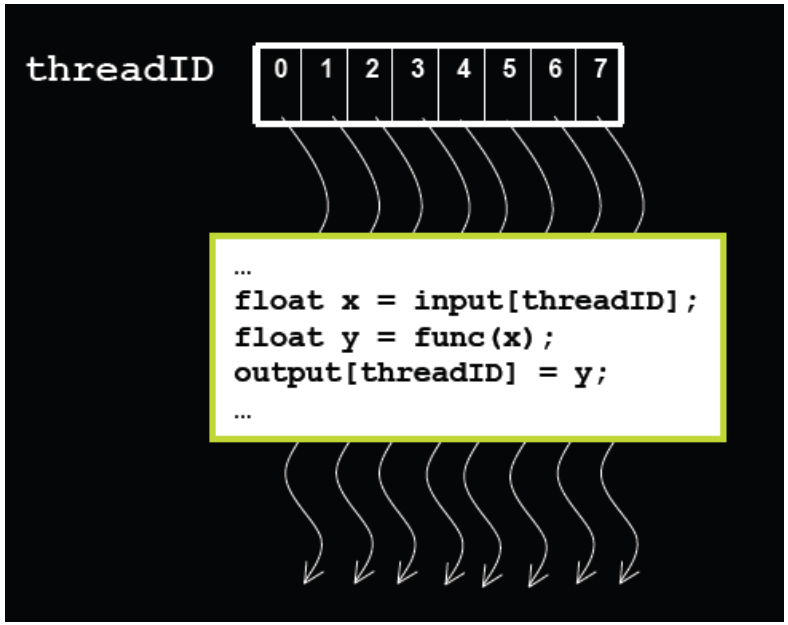
CUDA Kernels and Threads

- 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
  - Fast switching
- CUDA uses 1000s of threads to achieve efficiency
  - Multi-core CPUs can only use 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



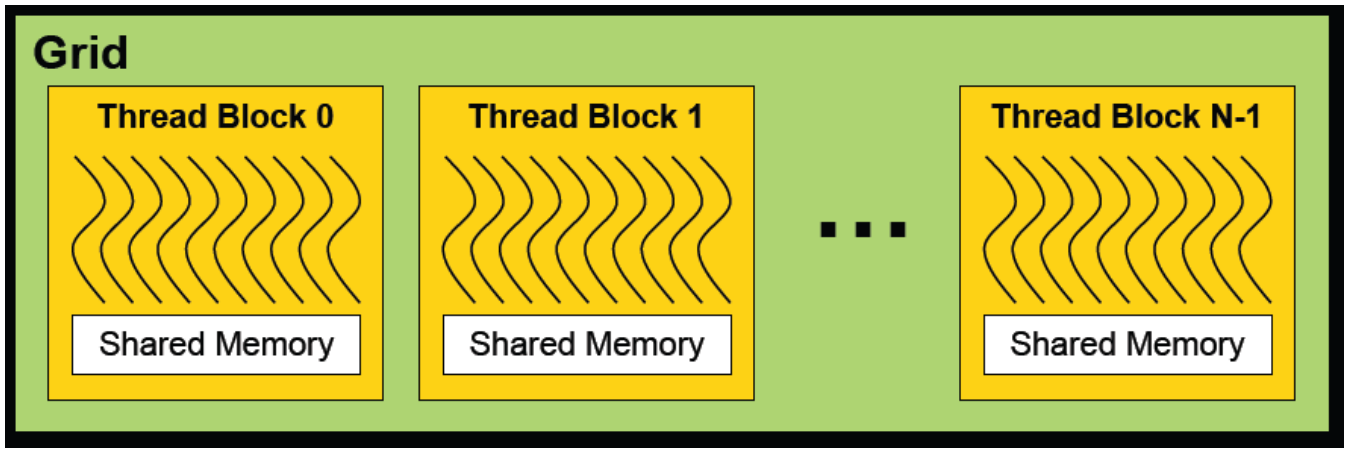
Thread Cooperation

- Threads may need or want to cooperate
- Thread cooperation is a powerful feature of CUDA
- Thread cooperation is valuable because threads can
  - Cooperate on memory accesses
    - Bandwidth reduction for some applications
  - Share results to avoid redundant computation

- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller **batches** of threads is scalable

Thread Batching

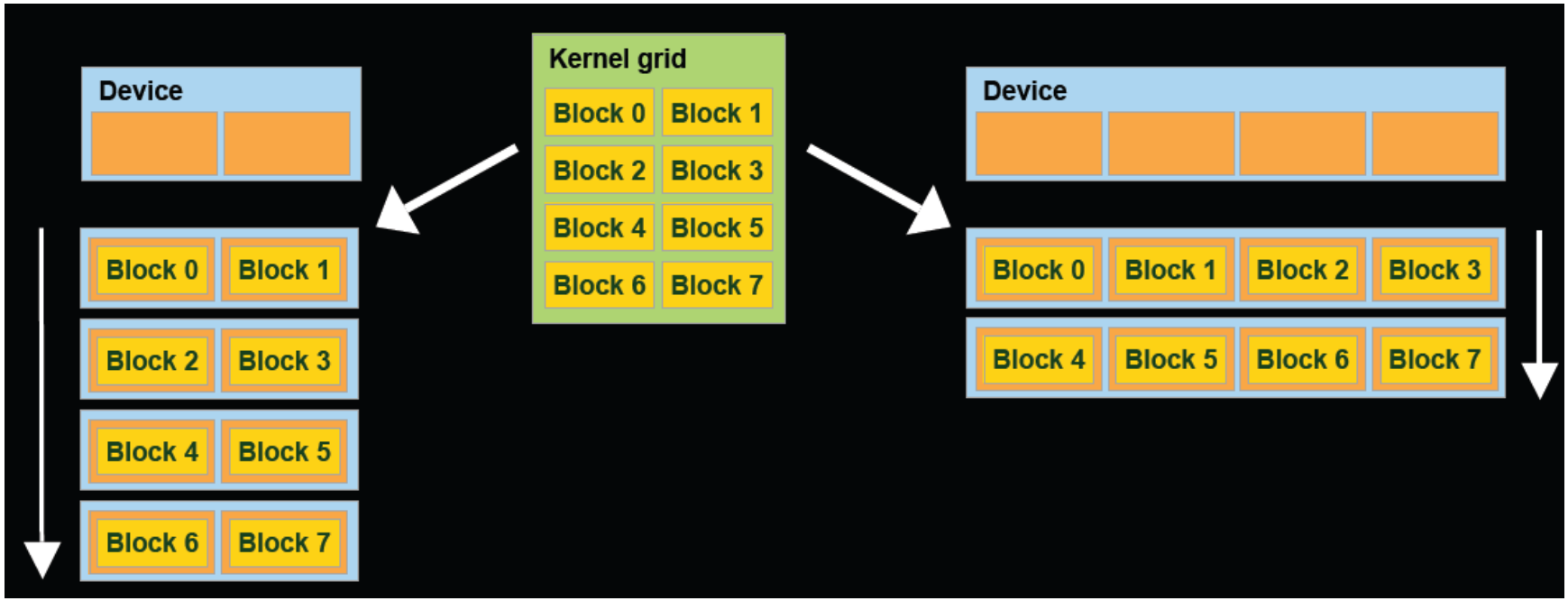
- Kernel launches a **grid** of **thread blocks**
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate



- Allows programs to **transparently scale** to different GPUs

Transparent Scalability

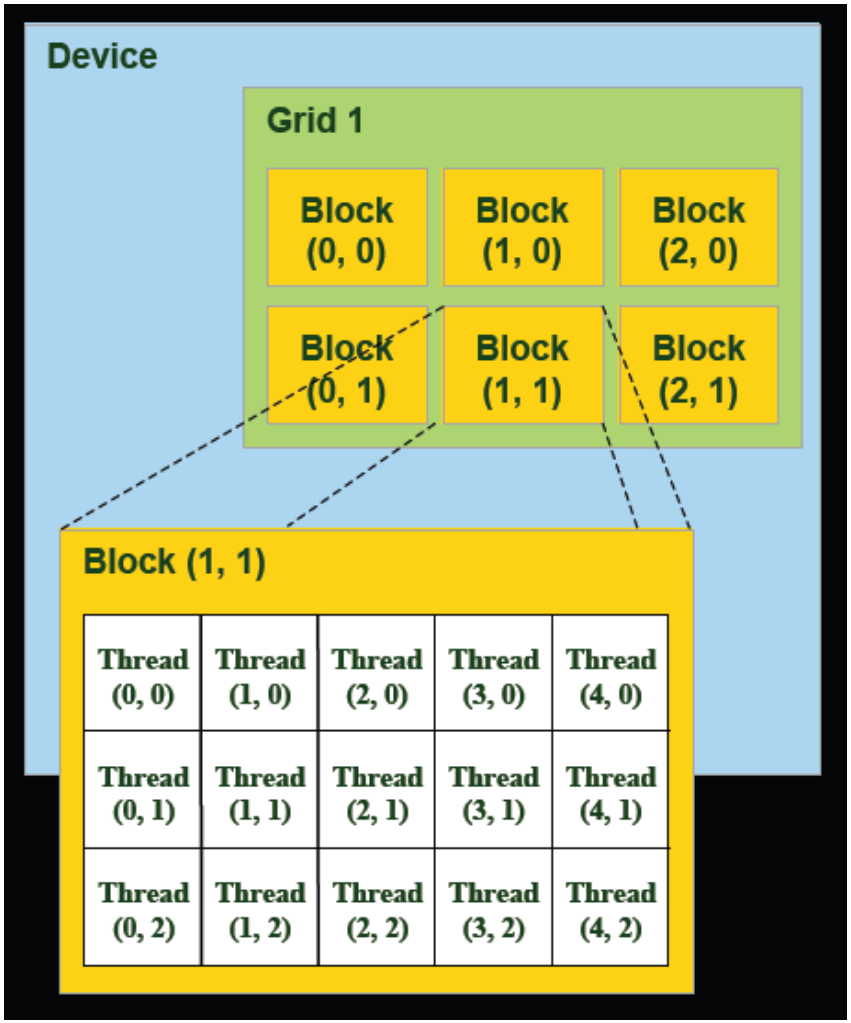
- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors



Multidimensional IDs

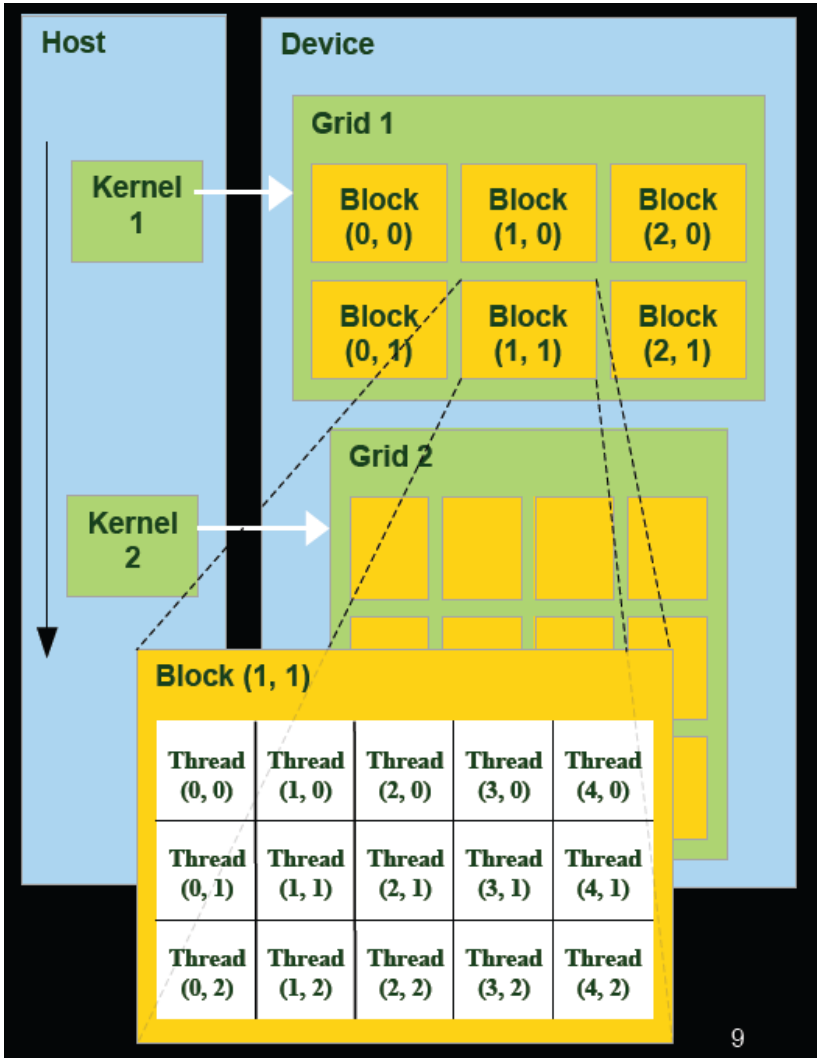
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Video processing
  - Solving PDEs on volumes



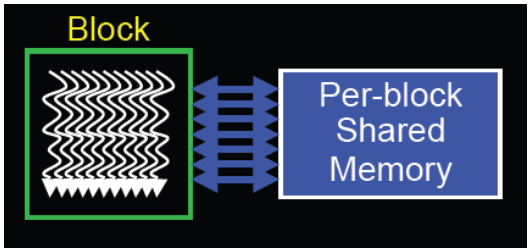
CUDA Programming Model

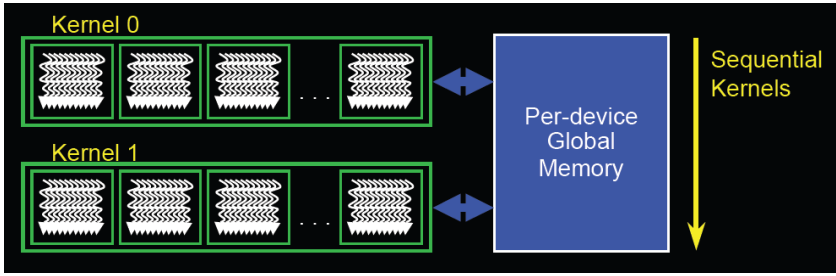
- A kernel is executed by a **grid** of **thread blocks**
- A **thread block** is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate

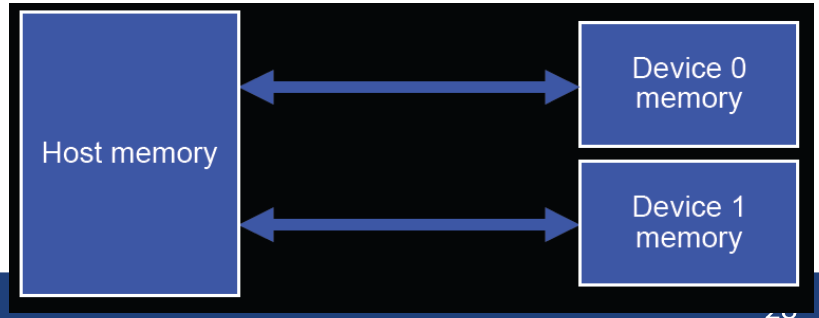


Memory Model

- Registers
  - Per thread
  - Data lifetime = thread lifetime
  - On-chip
- Local memory
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- Shared memory
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to de-allocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

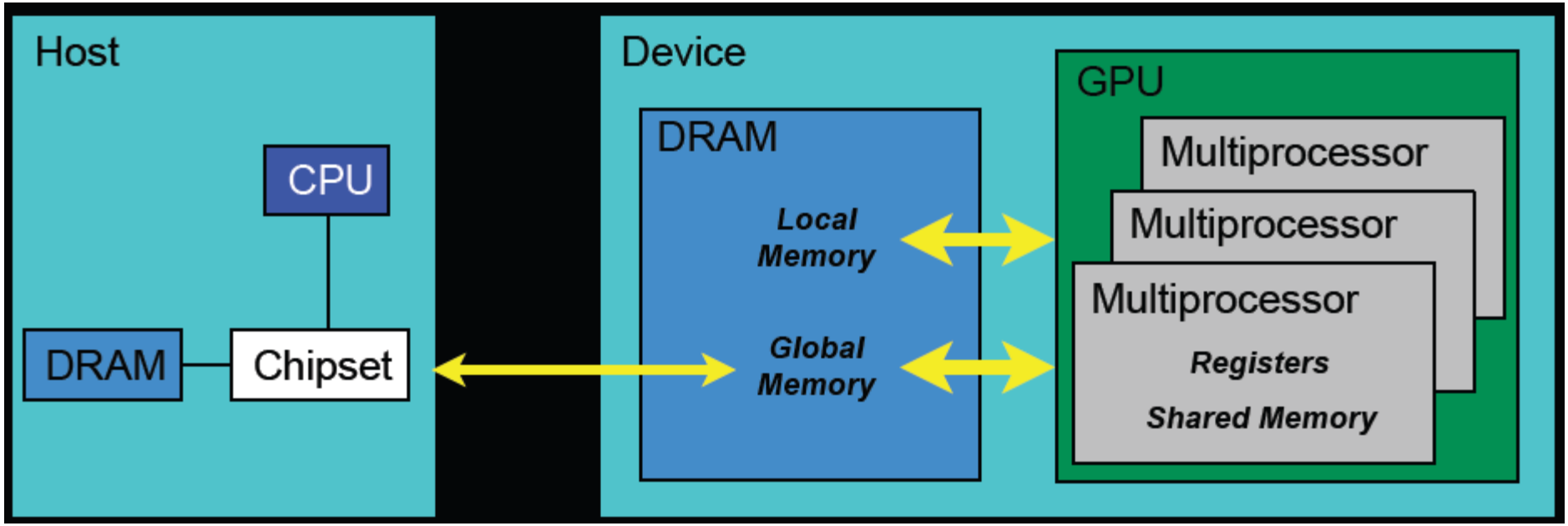




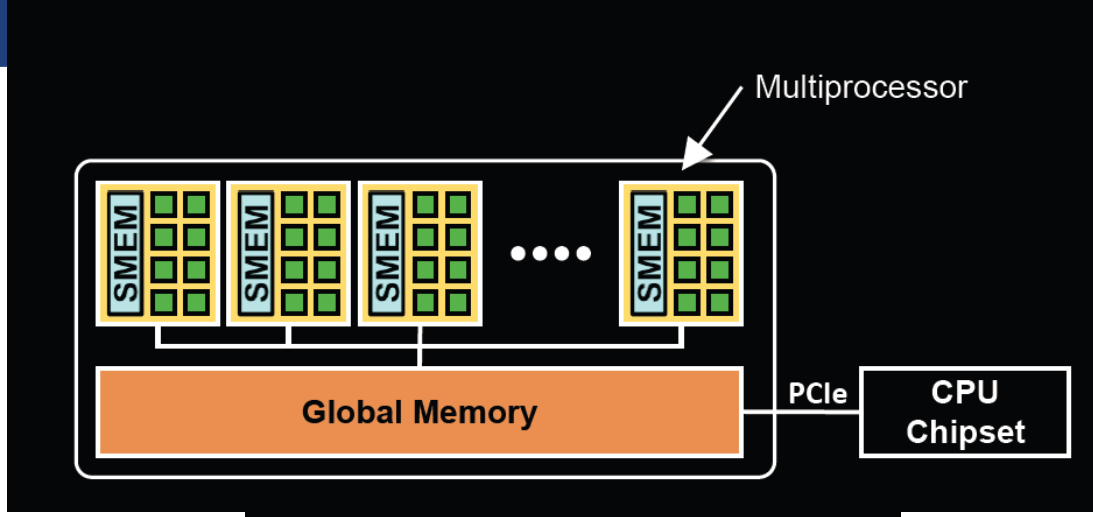


Physical Memory Layout

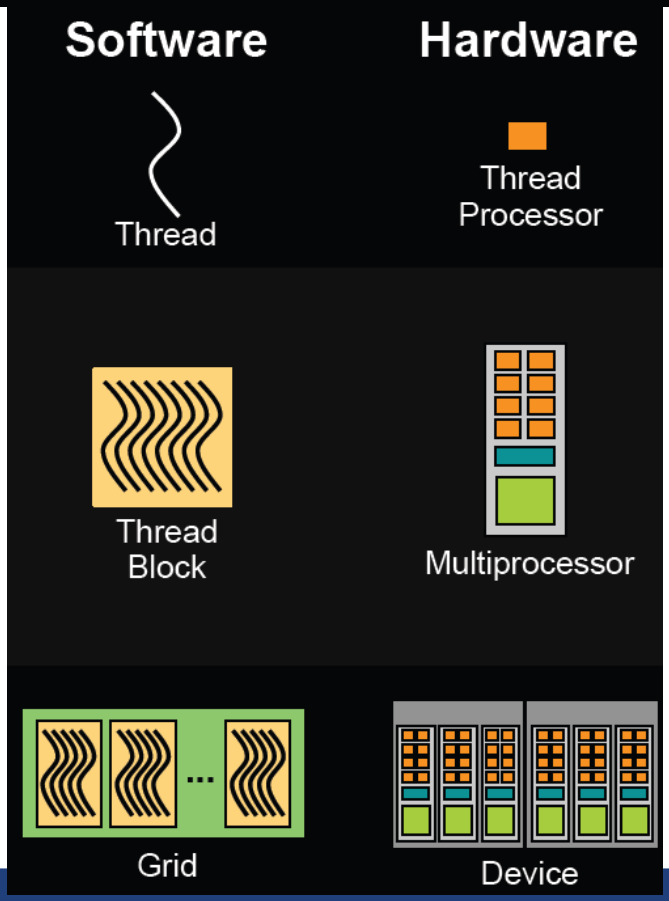
- "Local" memory resides in device DRAM
  - Use registers and shared memory to minimize local memory use
- Host can read and write global memory but not shared memory



Execution Model



- **Kernels are launched in grids**
  - One kernel executes at a time
- **A thread block executes on one multiprocessor**
  - Does not migrate
- **Several blocks can reside concurrently on one multiprocessor**
  - Number is limited by multiprocessor resources
    - **Registers** are partitioned among all resident threads
    - **Shared memory** is partitioned among all resident thread blocks



Key Parallel Abstractions in CUDA

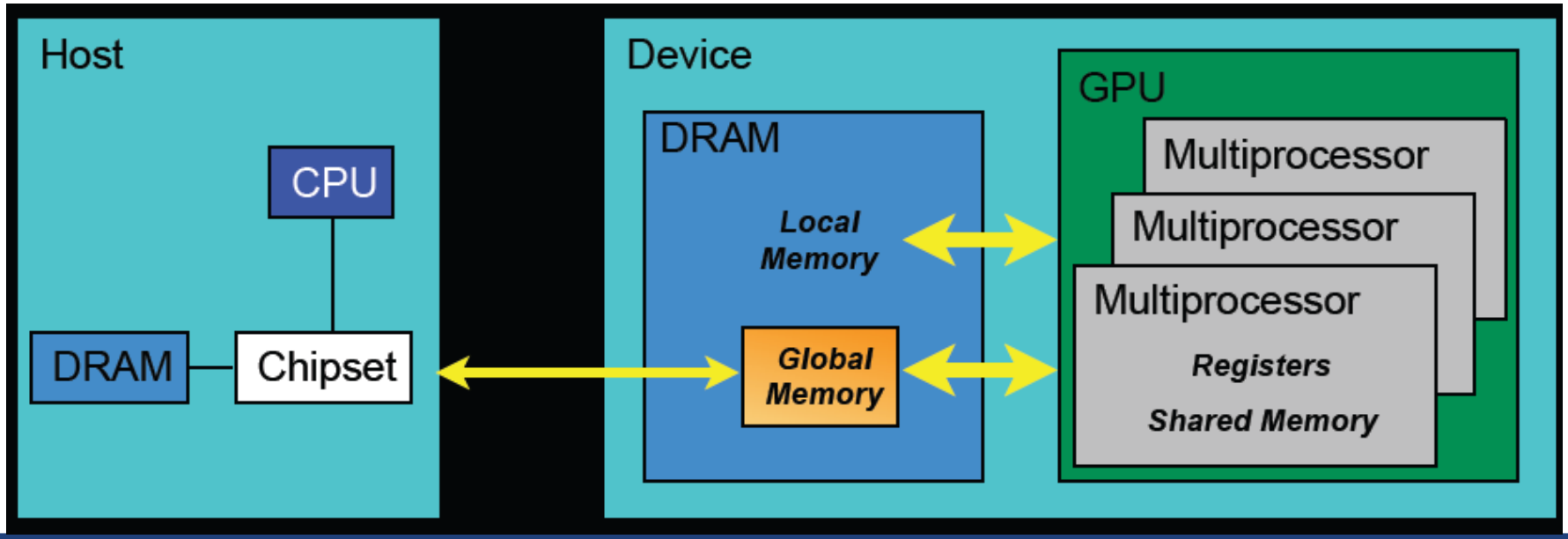
- Trillions of lightweight thread
  - Simple decomposition model
- Hierarchy of concurrent threads
  - Simple execution model
- Lightweight synchronization of primitives
  - Simple synchronization model
- Shared memory model for thread cooperation
  - Simple communication model

Outline of CUDA Basics

- Basics to set up and execute GPU code:
  - GPU memory management
  - GPU kernel launches
  - Some specifics of GPU code
- Some additional features:
  - Vector types
  - Synchronization
  - Checking CUDA errors
- Note: only the basic features are covered
  - See the Programming Guide and Reference Manual for more information

Managing Memory

- 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 *d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

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

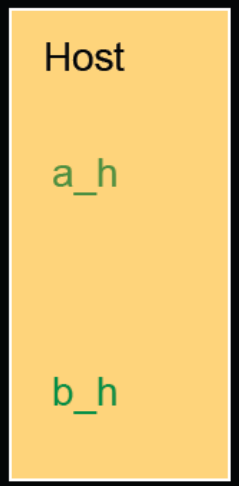
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

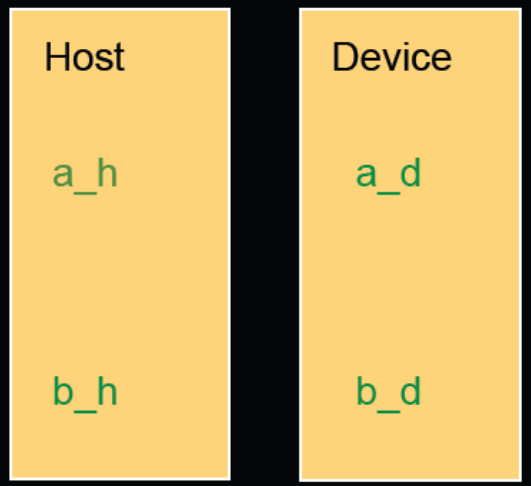
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

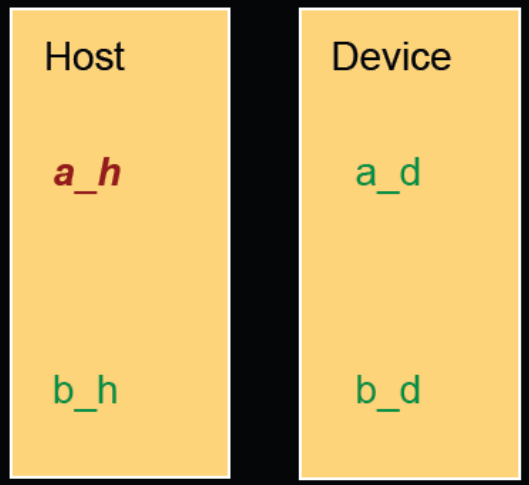
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

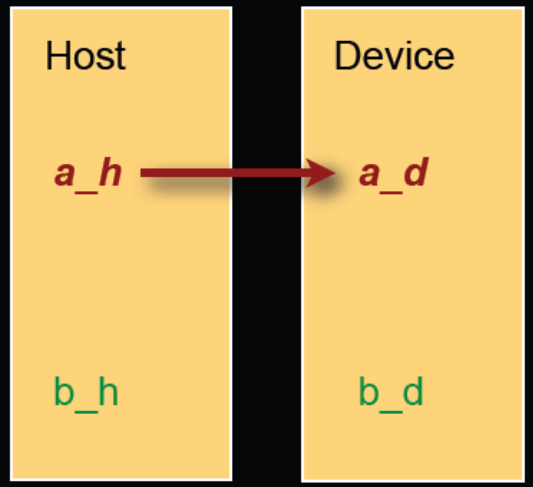
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

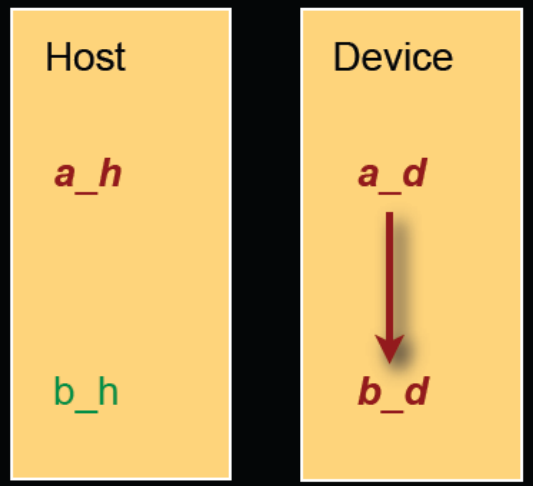
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

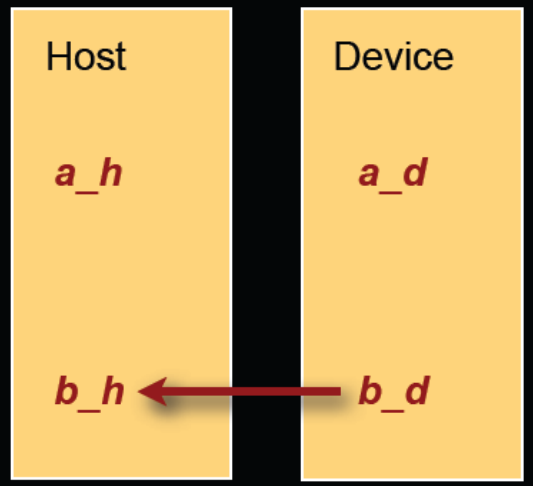
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

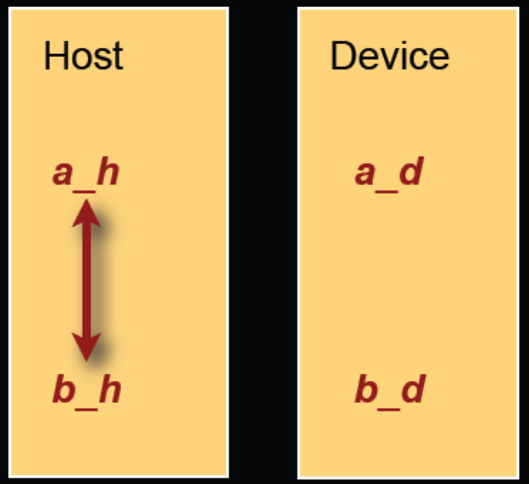
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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

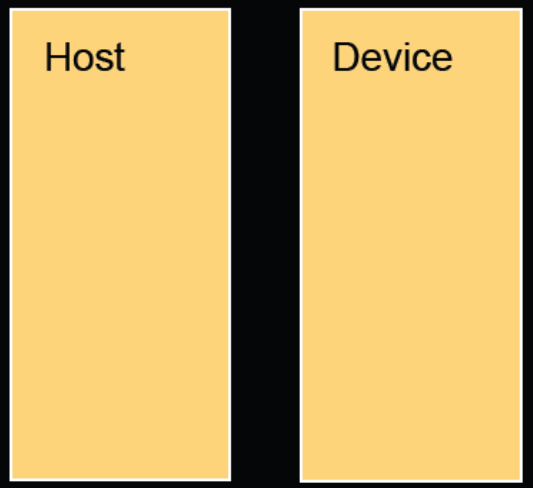
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  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;
}
```



Executing Code on the GPU

- **Kernels are C functions with some restrictions**
  - Can only access GPU memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- **Function arguments** automatically copied from CPU to GPU memory

Function Qualifiers

- Kernels designated by function qualifiers
  - **__global__**
    - Invoked from within host (CPU) code, cannot be called from device (GPU) code must return void
- Other CUDA function qualifiers
  - **__device__**
    - Called from other GPU functions, cannot be called from host (CPU) code
  - **__host__**
    - Can only be executed by CPU, called from host
- **__host__** and **__device__** qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

Launching kernels

- Modified C function call syntax:
  - kernel<<<dim3 grid, dim3 block>>>(…)
- Execution Configuration ("<<< >>>"):
  - grid dimensions: **x** and **y**
  - thread-block dimensions: **x**, **y**, and **z**

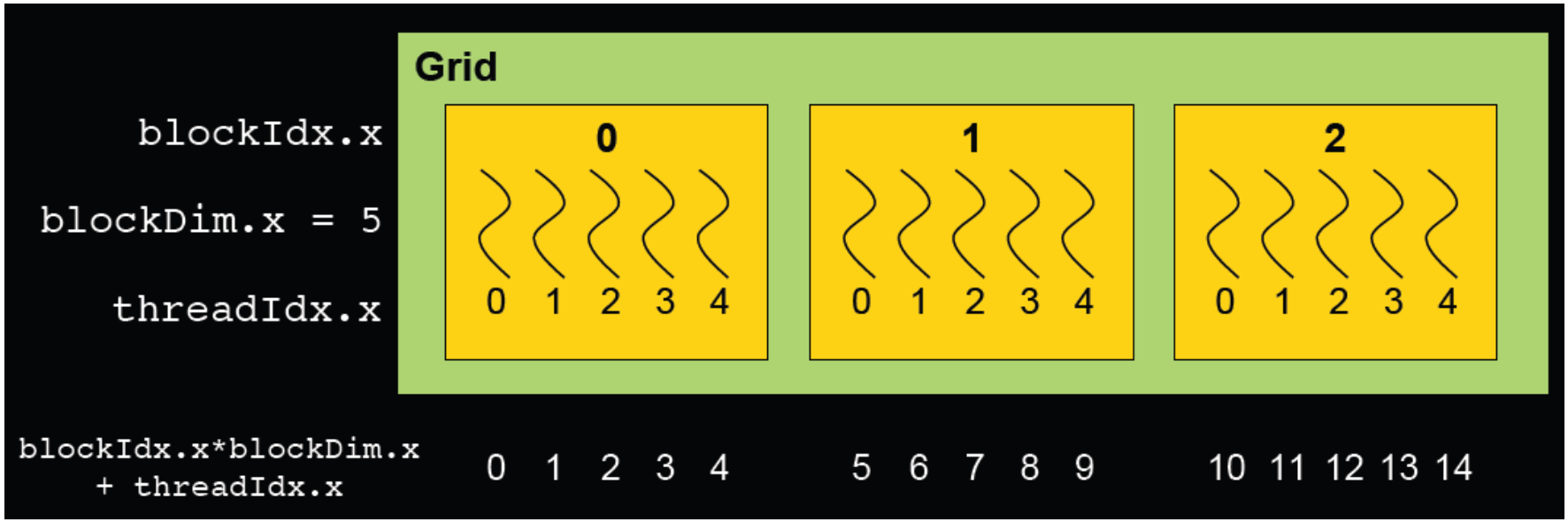
```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

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

Data Decomposition

- Often want each thread in kernel to access a different element of an array



Minimal Kernels

```
__global__ void minimal( int* d_a)
{
  *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

- Increment N-element vector a by scalar b

- Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{

        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
        increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
        dim3 dimBlock (blocksize);
        dim3 dimGrid( ceil( N / (float)blocksize)  );
        increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Minimal Kernel for 2D data

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
        int iy = blockDim.y * blockIdx.y + threadIdx.y;
        int ix = blockDim.x * blockIdx.x + threadIdx.x;
        int idx = iy * w + ix;
        d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

Example: Host Code

```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Variable Qualifiers (GPU code)

- **__device__**
  - Stored in device memory (large capacity, high latency, uncached)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- **__shared__**
  - Stored in on-chip shared memory (SRAM, low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: duration of thread block
- **Unqualified variables**
  - Scalars and built-in vector types are stored in registers
  - Arrays may be in registers or local memory (registers are not addressable)

Using shared memory

Size known at compile time

```
__global__ void kernel(…)
{
  …
  __shared__ float sData[256];
  …
}

int main(void)
{
  …
  kernel<<<nBlocks,blockSize>>>(…);
  …
}
```

Size known at kernel launch

```
__global__ void kernel(…)
{
  …
  extern __shared__ float sData[];
  …
}

int main(void)
{
  …
  smBytes = blockSize*sizeof(float);
  kernel<<<nBlocks, blockSize,
     smBytes>>>(…);
  …
}
```

Built-in Vector Types

- Can be used in GPU and CPU code
  - [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]
  - Structures accessed with x, y, z, w fields:
    - uint4 param;
    - int y = param.y;
- dim3
  - Based on uint3
  - Used to specify dimensions
  - Default value (1,1,1)

GPU Thread Synchronization

- *void __syncthreads();*
- **Synchronizes all threads in a block**
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Textures in CUDA

- Texture is an object for reading data
- Benefits:
  - Data is cached (optimized for 2D locality)
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear
    - dedicated hardware
  - Wrap modes (for "out-of-bounds" addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates
- Usage:
  - CPU code binds data to a texture object
  - Kernel reads data by calling a fetch function

GPU Atomic Integer Operations

- Requires hardware with compute capability >= 1.1
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3
- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: **OK**
  - shared lock: **BAD** … can easily deadlock

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop, int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control one GPU
    - multiple CPU threads can control the same GPU
    - calls are serialized by the driver

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

```
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```

Outline

- GPU architecture recap
  - Computation model
  - Memory hierarchy
- CUDA basics
  - Threads, blocks, grids
  - Memory model
  - Synchronization
- **Laboratory exercises**
  - **GPU info**
  - **Hello World!**
  - **Vector addition**
  - **Matrix Multiplication**
  - **…**

CUDA Exercises

- We have provided skeletons and solutions for several hands-on CUDA exercises
- In each exercise, you have to implement the missing portions of the code
- Finished when you compile and run the program and get the output "Correct!"
- Solutions will be provided after the lab

Exercises

1. Enumerate GPUs
2. Hello World
3. Vectors Addition
   A. Using only threads
   B. Using only blocks
   C. Using threads and blocks
4. Matrix Multiplication
5. Matrix Transpose

Q&A