

# CUDA C/C++ BASICS

NVIDIA Corporation  
Modified by Dov Kruger

# What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
  - Proprietary NVIDIA Language :-(
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

# Introduction to CUDA C/C++

- Overview of GPU Architecture
  - Documentation
  - Performance Bottlenecks
  - Getting information about the GPU
  - Thread programming model

# CUDA Parallel Computing Platform

[www.nvidia.com/getcuda](http://www.nvidia.com/getcuda)

Programming  
Approaches

Libraries

“Drop-in”  
Acceleration

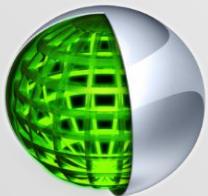
OpenACC  
Directives

Easily Accelerate  
Apps

Programming  
Languages

Maximum Flexibility

Development  
Environment



Nsight IDE  
Linux, Mac and Windows  
GPU Debugging and  
Profiling

CUDA-GDB  
debugger  
NVIDIA Visual  
Profiler

Open Compiler  
Tool Chain

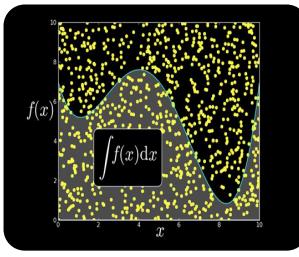


Enables compiling new languages to CUDA  
platform, and CUDA languages to other  
architectures

# Some GPU-accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



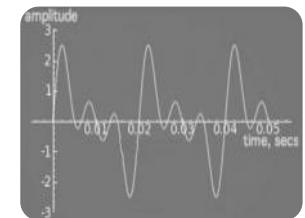
Vector Signal  
Image Processing



GPU Accelerated  
Linear Algebra



Matrix Algebra  
on GPU and  
Multicore



NVIDIA cuFFT



ROGUE WAVE  
SOFTWARE  
IMSL Library



ArrayFire Matrix  
Computations

C U S P

Sparse Linear  
Algebra



C++ STL  
Features for  
CUDA



# Documentation

[https://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf)

<https://github.com/RU-ECE/ECE451-Parallel/tree/main/ref>

# Installing CUDA

Download

<https://developer.nvidia.com/cuda-downloads>

CUDA discovers capabilities dynamically

Not all features will work on all cards

# NVIDIA History

Board	Clock Speed	Cores	transistor s (B)	RAM GB/s	RAM size	NVIDIA
GeForce GTX 1050 [	1350	640	3.3		2GB	6.1
GeForce RTX 2060	1470	3800	10.8	336GB/s GDDR6	6GB	7.5
GeForce RTX 3060		10k?		360 GB/s GDDR6	12GB	
RTX 4090	2500	16384		1TB/s GDDR6X	24GB	8.9

# PCIe Bus Speed

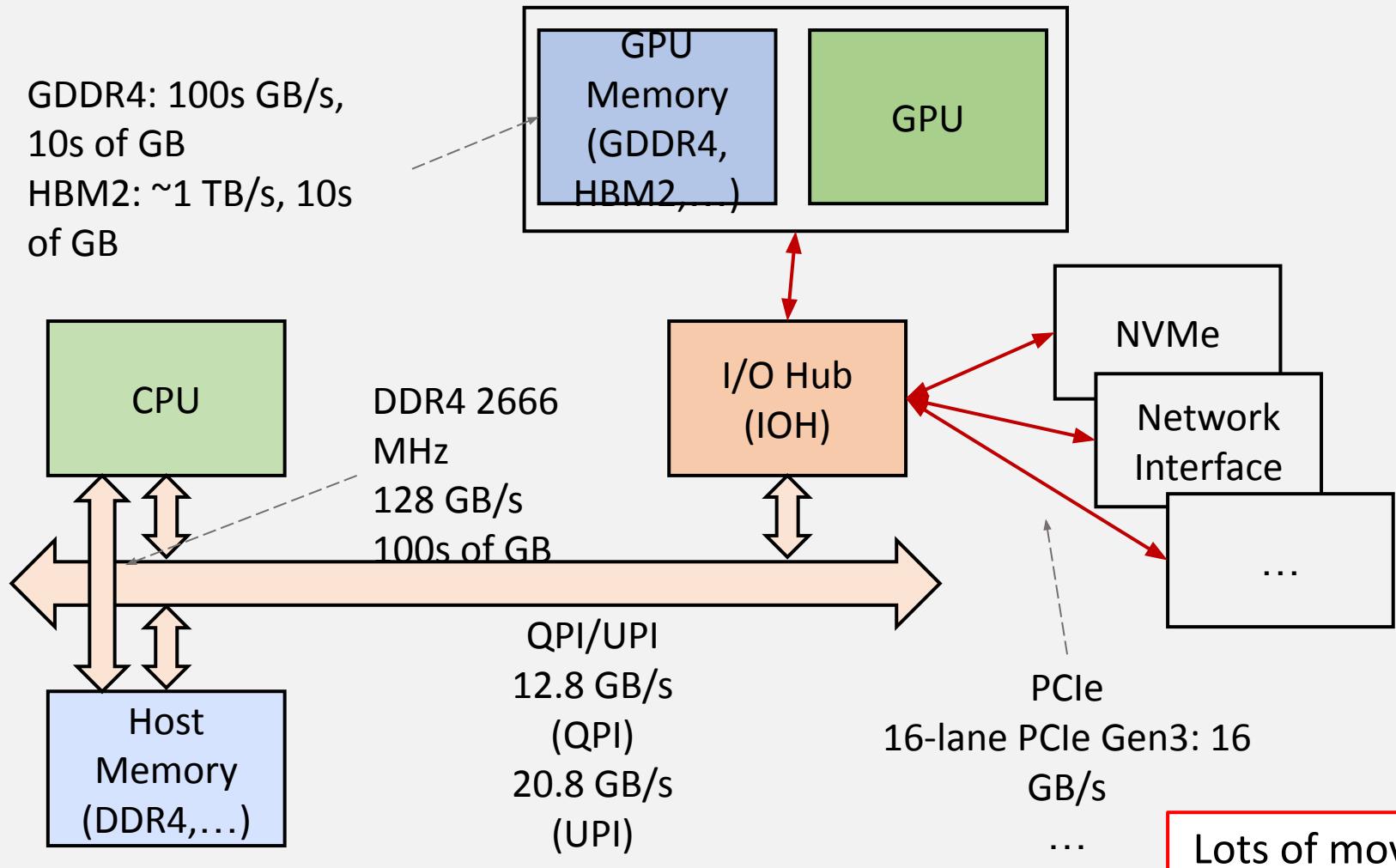
Gen 3: 16GB/s bidirectional

Gen 4: 32GB/s bidirectional

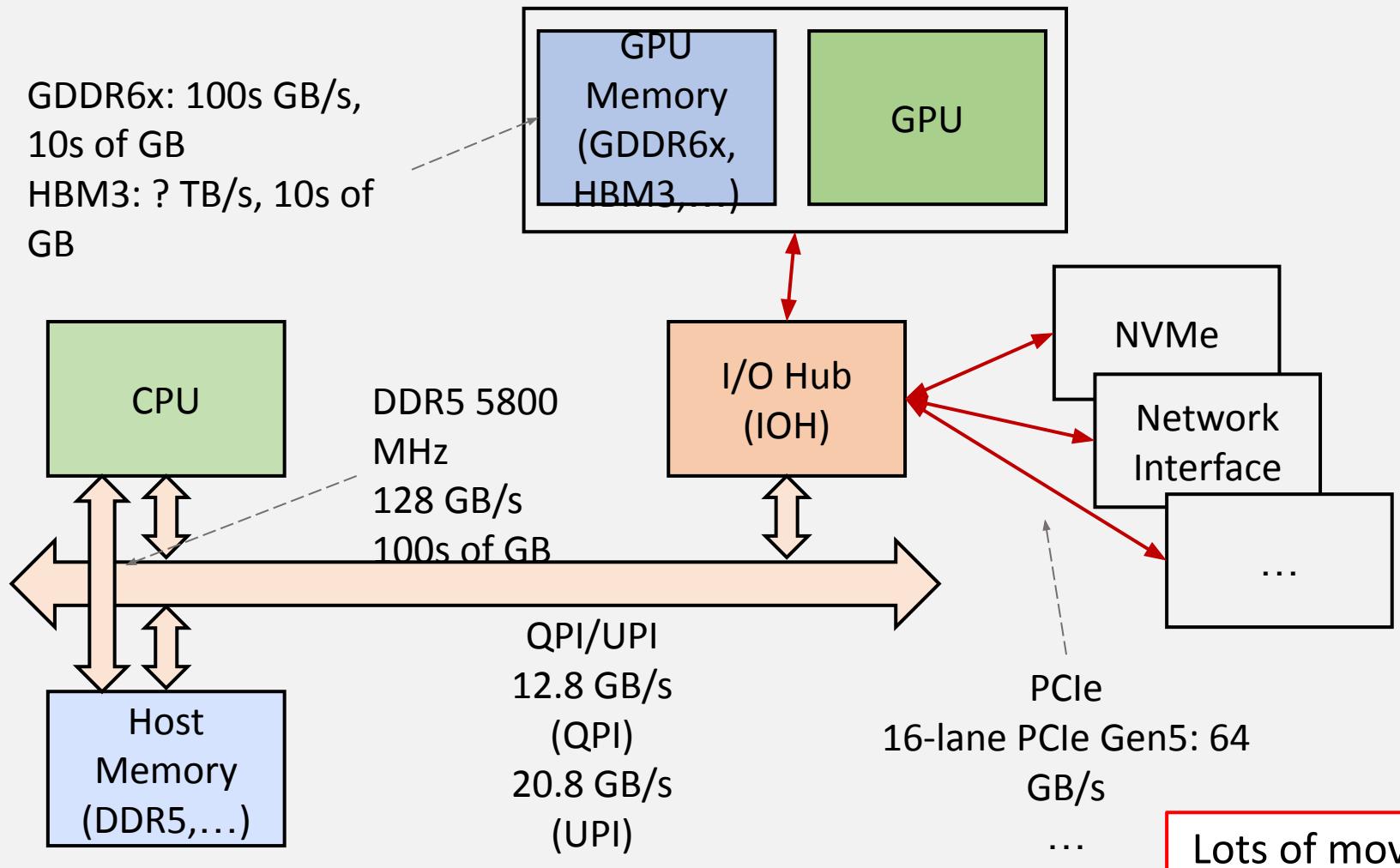
Gen 5: 64GB/s bidirectional

This may seem fast, but it's quite slow compared with memory bandwidth, for example!

# System Architecture Snapshot With a GPU (2019)



# System Architecture Snapshot With a GPU (2024)



# Themes

Memory bandwidth is faster (but more CPUs)

We are still memory limited!

PC RAM: 64-bit bus

Video RAM: 392-bit bus

PCIe bus is quite slow by comparison

Try to minimize data transfers

# CUDA Tools

nvcc file.cu # usual compile options apply

nvcc is a wrapper on c++, probably clang?

cuda-gdb

# CONCEPTS

---

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# HELLO WORLD!

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host



Device

# 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 index = threadIdx.x + RADIUS;

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

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

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

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

void fill_ints(int *x, int n) {
    fill_(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(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);

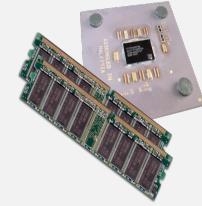
    // Clean up
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

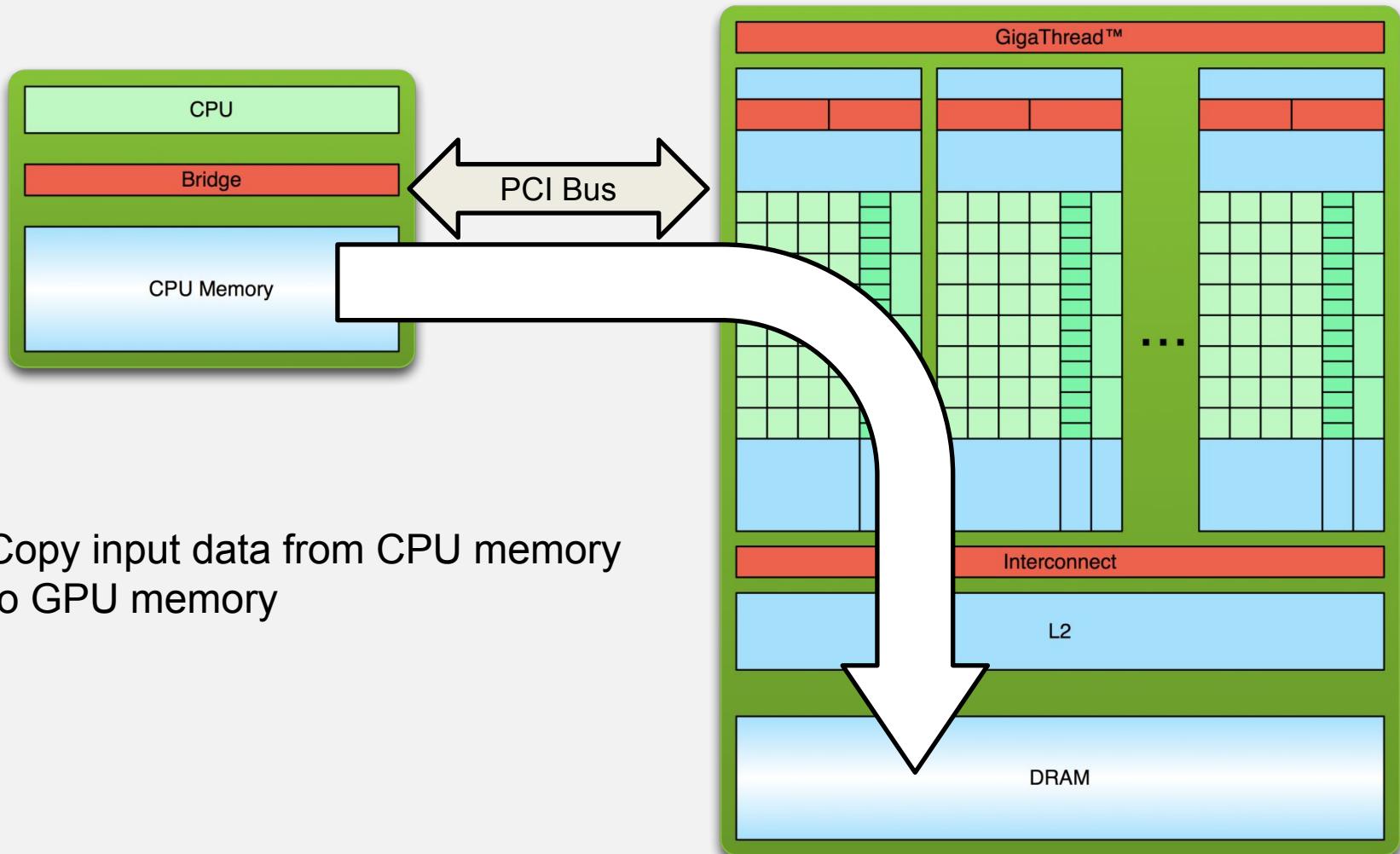
serial code

parallel code

serial code

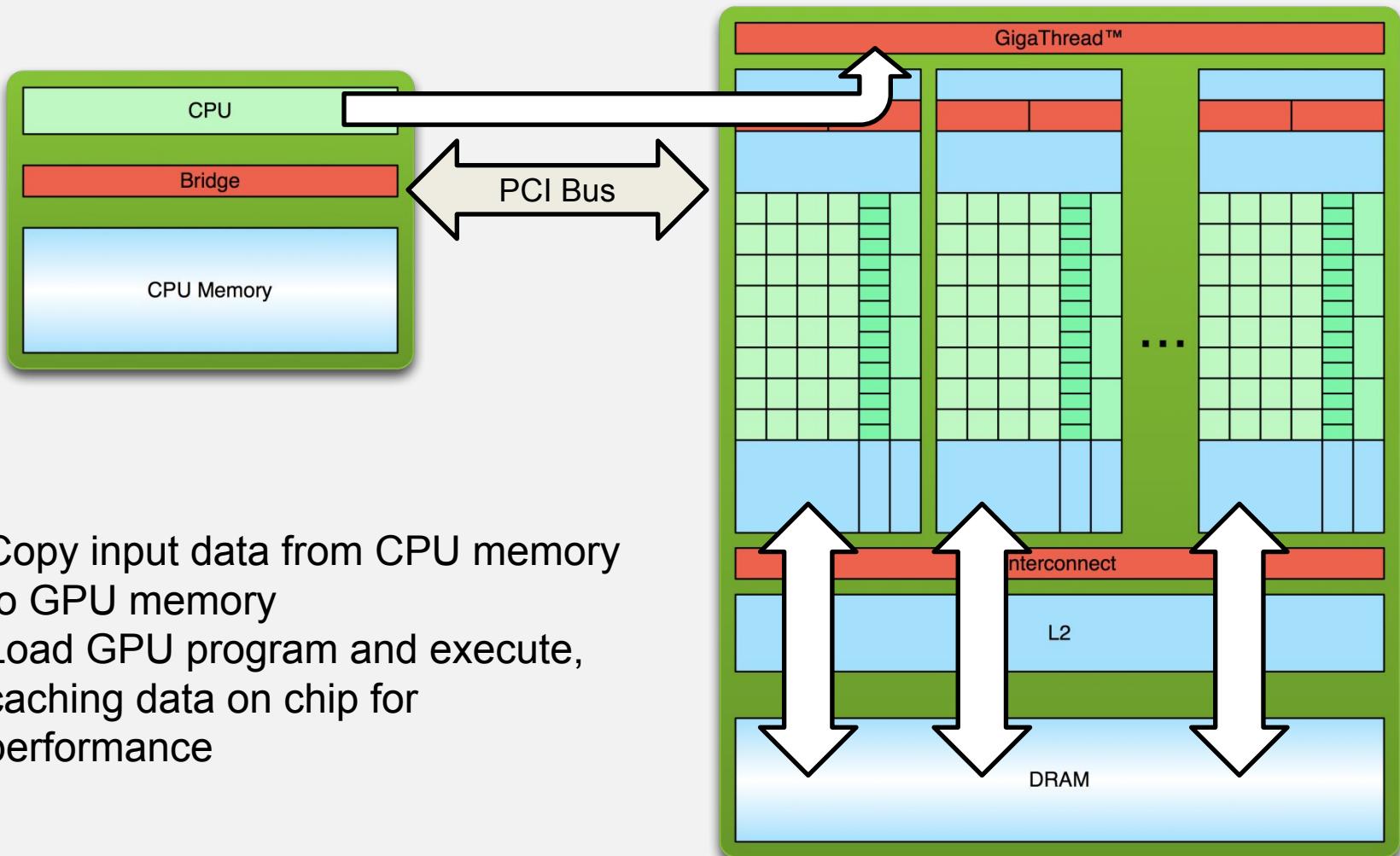


# Simple Processing Flow

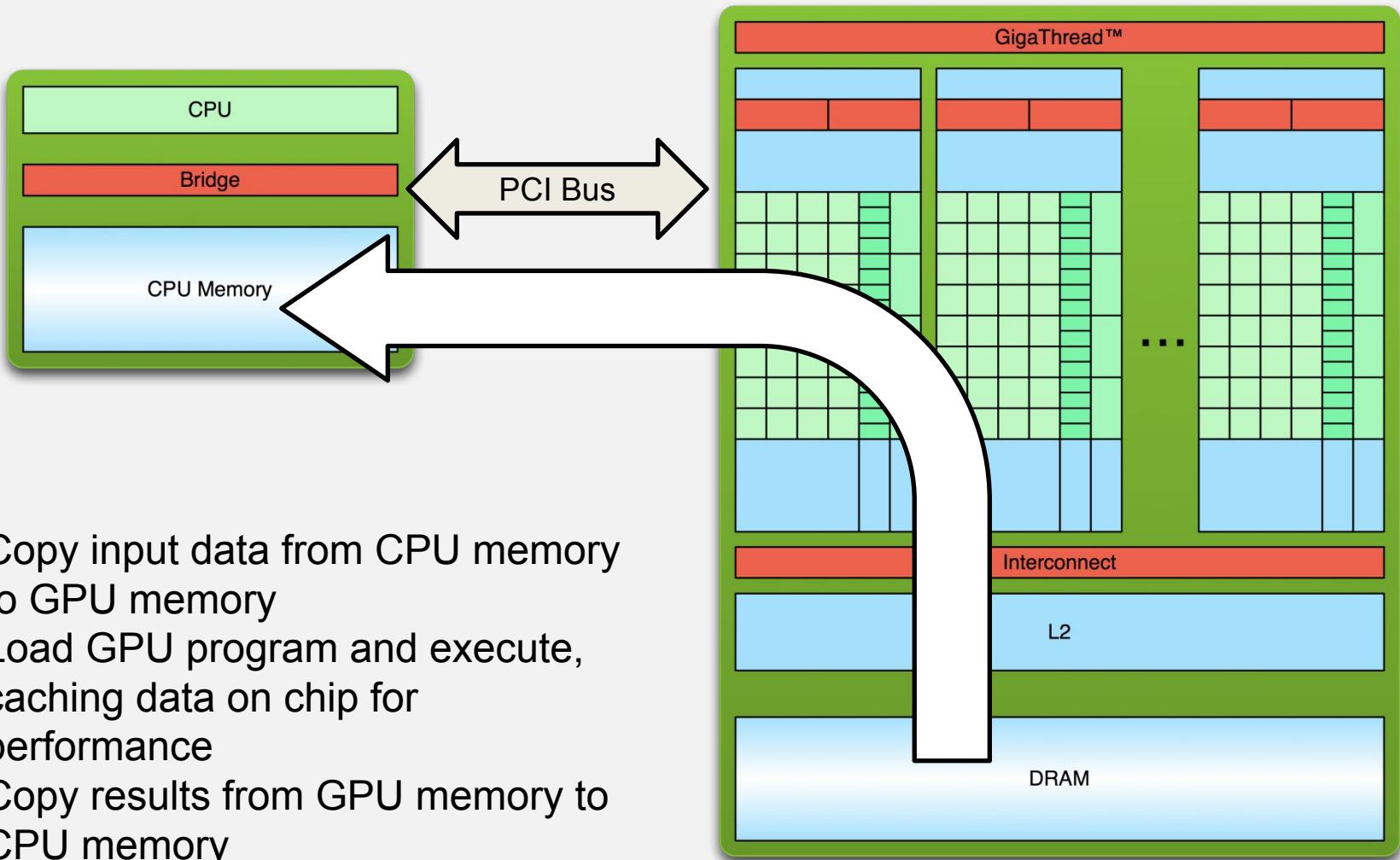


1. Copy input data from CPU memory to GPU memory

# Simple Processing Flow



# Simple Processing Flow



# Note: This is not the Only Model!

The model you have just seen is for NVIDIA's flagship boards

They have their own (massive) processors

They have their own high-performance RAM

But for lower-end computers, typically laptops the GPU can share RAM with the CPU.

# Hello World!

```
int main() {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

## Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

# Hello World! with Device Code

```
__global__ void mykernel() {  
}  
  
int main() {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- nvcc separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc, cl.exe`

# Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

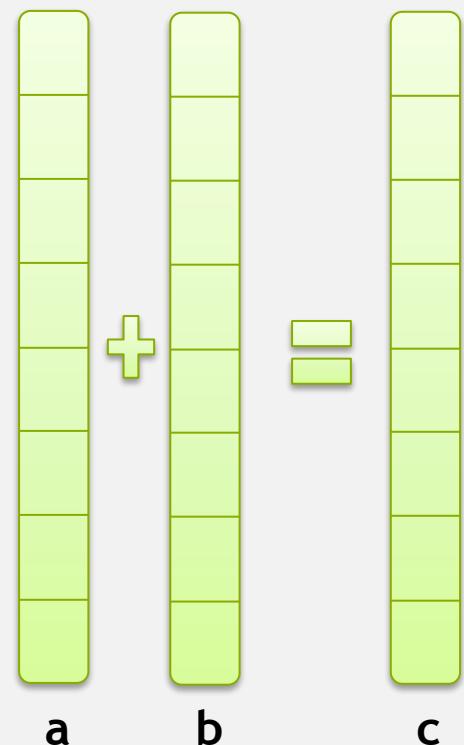
- **mykernel()** does nothing,  
somewhat anticlimactic!

Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

# Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



# Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before **\_\_global\_\_** is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

# Addition on the Device

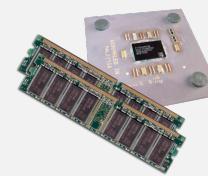
- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

# Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



# Addition on the Device: add()

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at `main()`...

# Addition on the Device: main()

```
int main(void) {
    int a, b, c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                      // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size); //equates to d_a=alloc(size)
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

# Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# RUNNING IN PARALLEL

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();  
      ^  
add<<< N, 1 >>>();
```

- Instead of executing add ( ) once, execute N times in parallel

# Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`
- By using `blockIdx.x` to index into the array, each block handles a different index

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

# Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at `main()`...

# Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;      // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

# Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(...);`
  - Use `blockIdx.x` to access block index

# CUDA Fundamentals

Thread has

- Registers (255?) (fastest)
- \_\_shared\_\_ memory (on chip, shared with block)  
global memory on GPU
- Ability to communicate between threads

# Shared Memory Banks

Capability 1.0: warp size 32, 16 banks

Capability 2.0: Multiple requests to same memory can be *multicast*

Capability 3.0: Bank size can be set (support for 64-bit numbers (double))

If more than 1 threads are accessing same block they share sequentially

# Configuring Shared Memory

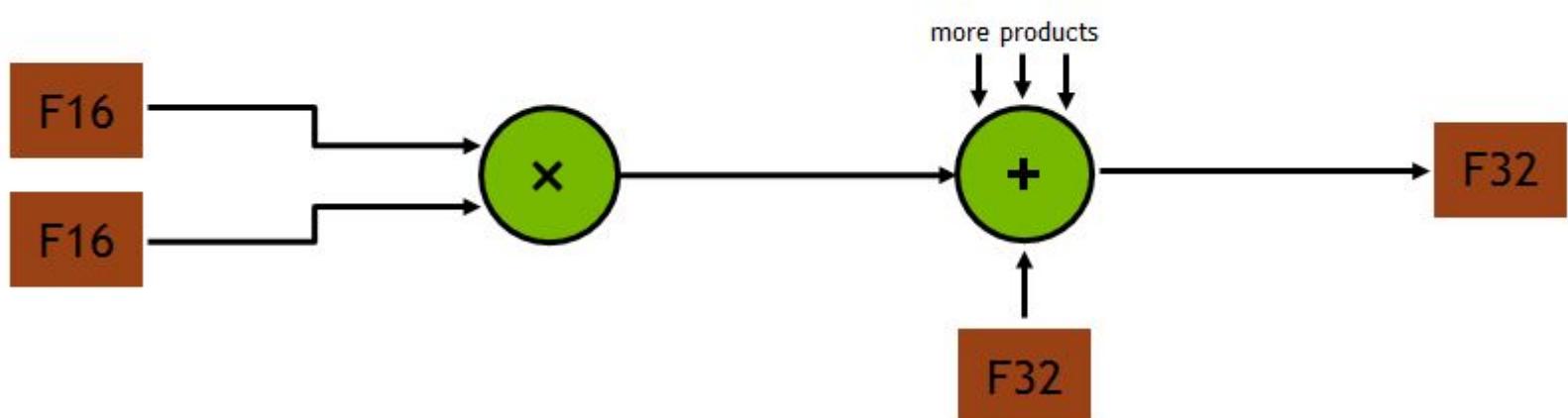
Cache can be repurposed: 64kB partitioned

- L1 cache
- shared memory

# Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

FP16 storage/input      Full precision product      Sum with FP32 accumulator      Convert to FP32 result



# Tensor Cores Speed Libraries

cuBLAS and cuDNN

Optimize: convolutions, RNN

```
// First, create a cuBLAS handle:  
cublasStatus_t cublasStat = cublasCreate(&handle);  
  
// Set the math mode to allow cuBLAS to use Tensor Cores:  
cublasStat = cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);  
  
// Allocate and initialize your matrices (only the A matrix is shown):  
size_t matrixSizeA = (size_t)rowsA * colsA;  
T_ELEM_IN **devPtrA = 0;  
  
cudaMalloc((void**)&devPtrA[0], matrixSizeA * sizeof(devPtrA[0][0]));  
T_ELEM_IN A = (T_ELEM_IN *)malloc(matrixSizeA * sizeof(A[0]));  
  
memset( A, 0xFF, matrixSizeA* sizeof(A[0]));  
status1 = cublasSetMatrix(rowsA, colsA, sizeof(A[0]), A, rowsA, devPtrA[i], rowsA);  
  
// ... allocate and initialize B and C matrices (not shown) ...  
  
// Invoke the GEMM, ensuring k, lda, ldb, and ldc are all multiples of 8,  
// and m is a multiple of 4:  
cublasStat = cublasGemmEx(handle, transa, transb, m, n, k, alpha,  
                           A, CUDA_R_16F, lda,  
                           B, CUDA_R_16F, ldb,  
                           beta, C, CUDA_R_16F, ldc, CUDA_R_32F, algo);
```

# cuBLAS Mixed-Precision GEMM (FP16 Input, FP32 Compute)



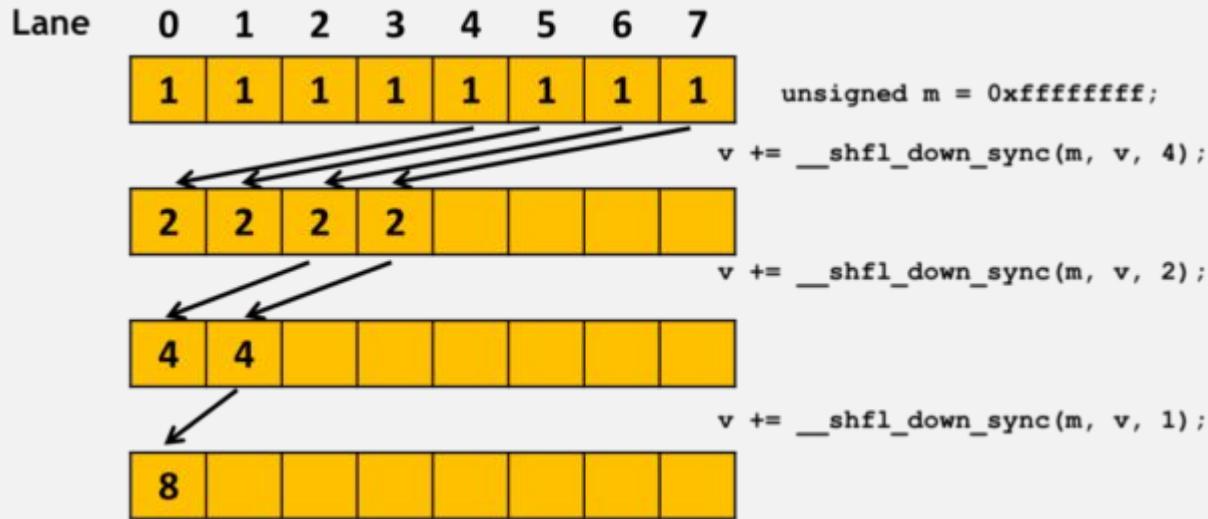
# Warp Level Programming

You can communicate between threads out of band!

No more memory limitations

Only communicate between threads in a warp

# Communication



```
define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

# \_\_shfl\_down\_sync

A warp comprises 32 *lanes*

Each thread occupies 1 lane

For a thread at lane x in the warp

`__shfl_down_sync(FULL_MASK, var, offset)`

gets the value of var from the thread at lane x + offset

# Communication Between Threads

CUDA 9 introduced three categories of new or updated warp-level primitives.

1. Synchronized data exchange: exchange data between threads in warp.
  - `--all_sync`, `--any_sync`, `--uni_sync`, `--ballot_sync`
  - `--shfl_sync`, `--shfl_up_sync`, `--shfl_down_sync`,  
`--shfl_xor_sync`
  - `--match_any_sync`, `--match_all_sync`
2. Active mask query: returns a 32-bit mask indicating which threads in a warp are active with the current executing thread.
  - `--activemask`
3. Thread synchronization: synchronize threads in a warp and provide a memory fence.
  - `--syncwarp`

# INTRODUCING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# CUDA Threads

- Terminology: a block can be split into parallel **threads**
- Let's change add () to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```
- We use **threadIdx.x** instead of **blockIdx.x**
- Need to make one change in **main()** ...

# Vector Addition Using Threads: main()

```
#define N 512

int main(void) {
    int *a, *b, *c;                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition Using Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

## CONCEPTS

# COMBINING THREADS AND BLOCKS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

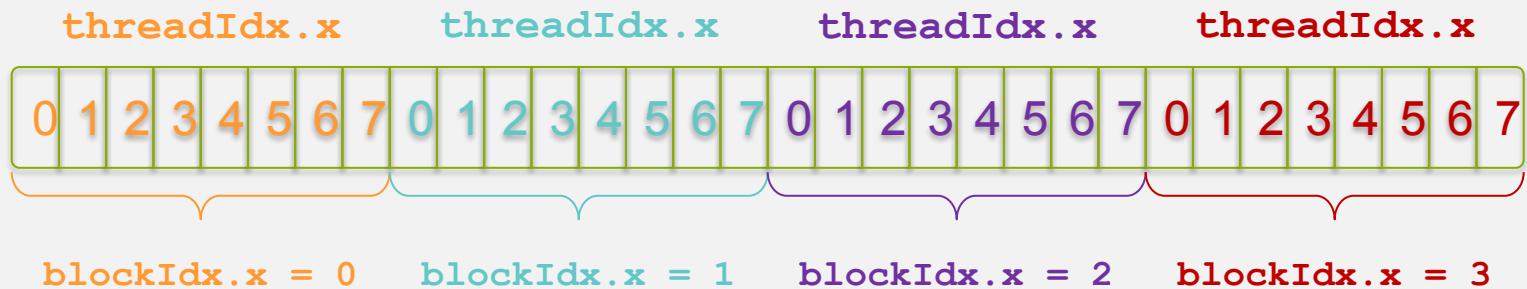
Managing devices

# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

# Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

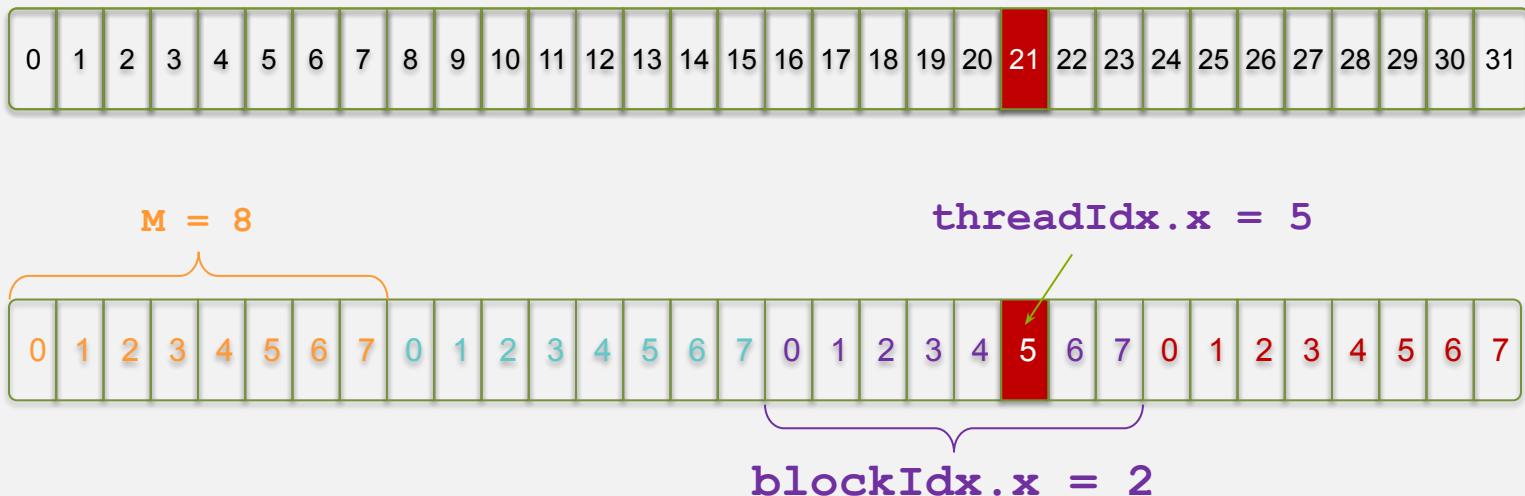


- With  $M$  threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

# Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

# Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```
- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- What changes need to be made in `main()`?

# Addition with Blocks and Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Addition with Blocks and Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

# Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example...

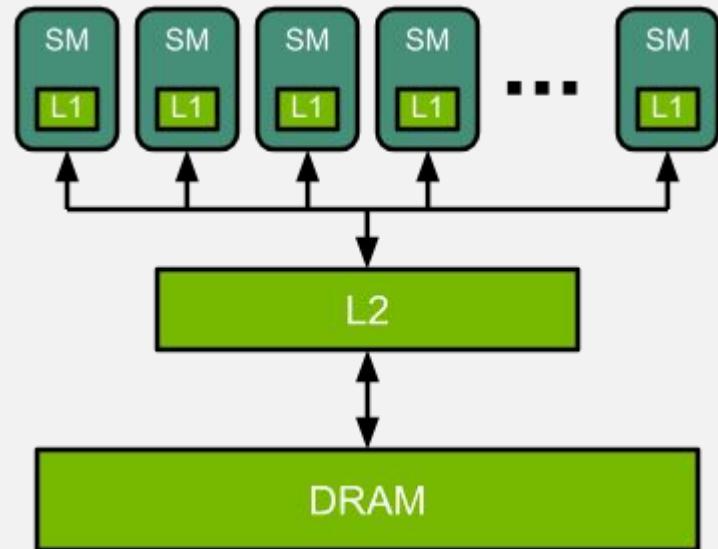
# Review

- Launching parallel kernels
  - Launch  $N$  copies of `add()` with `add<<<N/M, M>>>(...);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

# GPU Structure

The GPU has  
Streaming Multiprocessors  
(SM)  
Thread Block  
All threads execute same  
code (SIMD, like AVX2)



# Memory Hierarchy

Registers: 64k?

Limits 255 per thread? Fastest

Shared memory: Shared between threads in a block (99k) `__shared__`

Global memory: GB, much slower

- Many banks
- Very wide (384-bit)

shared

# COOPERATING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

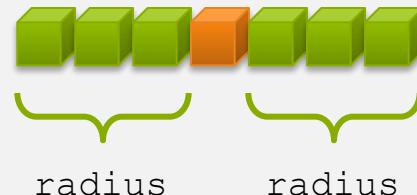
Asynchronous operation

Handling errors

Managing devices

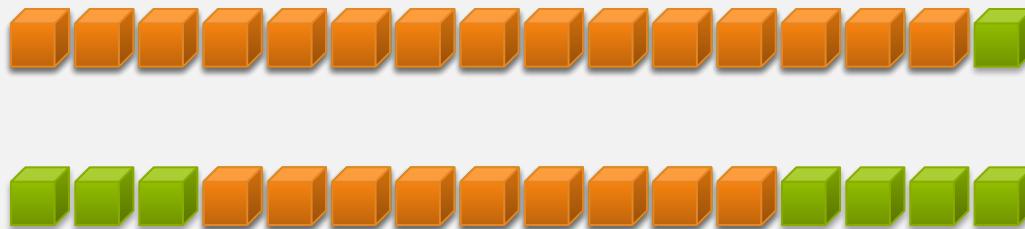
# 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



# Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times

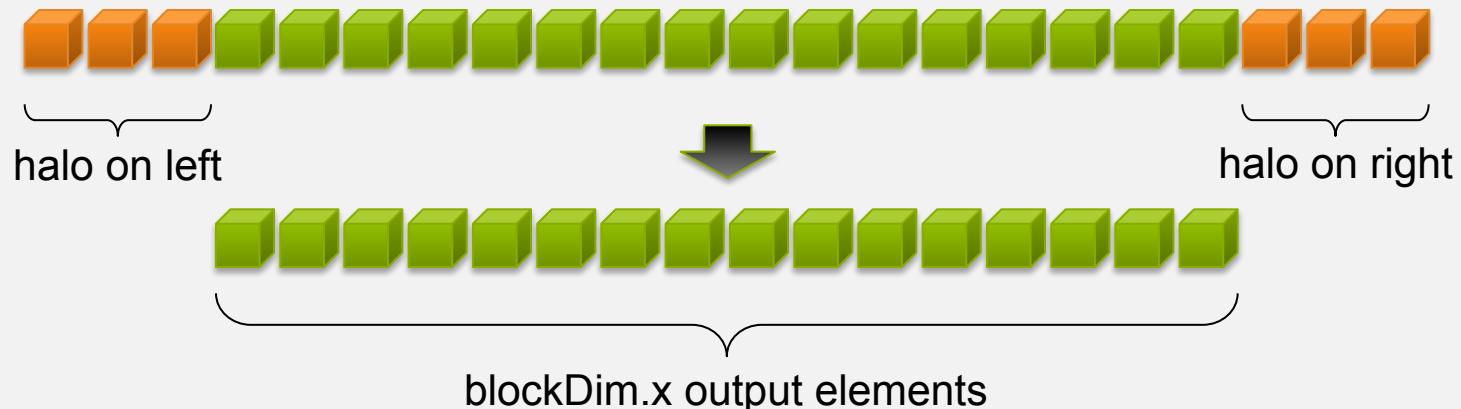


# Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

# Implementing With Shared Memory

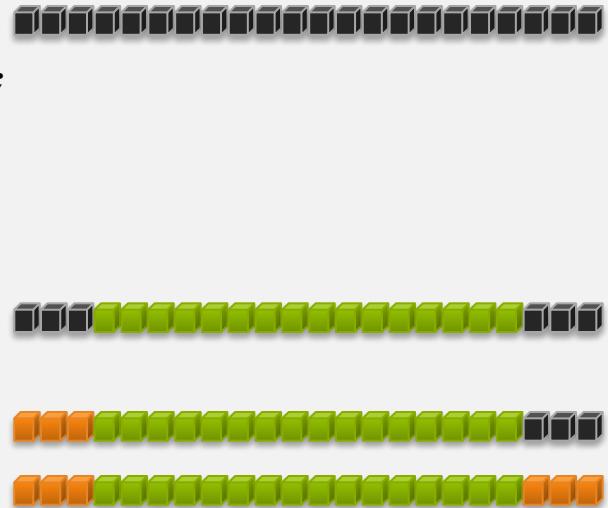
- Cache data in shared memory
  - Read  $(\text{blockDim.x} + 2 * \text{radius})$  input elements from global memory to shared memory
  - Compute  $\text{blockDim.x}$  output elements
  - Write  $\text{blockDim.x}$  output elements to global memory
  - Each block needs a **halo** of radius elements at each boundary



# Stencil Kernel

```
__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];
    }
}
```



# Stencil Kernel

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

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

# Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];   Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];          Load from temp[19] 
```

# \_\_syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

# Stencil Kernel

```
__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();
}
```

# Stencil Kernel

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

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

# Review (1 of 2)

- Launching parallel threads
  - Launch  $N$  blocks with  $M$  threads per block with  
`kernel<<<N,M>>>(...);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

# Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

# MANAGING THE DEVICE

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

**cudaMemcpy()**

Blocks the CPU until the copy is complete  
Copy begins when all preceding CUDA calls have completed

**cudaMemcpyAsync()**

Asynchronous, does not block the CPU

**cudaDeviceSynchronize()**

Blocks the CPU until all preceding CUDA calls have completed

# Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:  
`cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:  
`char *cudaGetString(cudaError_t)`  
  
`printf("%s\n", cudaGetString(cudaGetLastError()));`

# Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)  
cudaSetDevice(int device)  
cudaGetDevice(int *device)  
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

`cudaSetDevice(i)` to select current device  
`cudaMemcpy(...)` for peer-to-peer copies<sup>†</sup>

<sup>†</sup> requires OS and device support

# Introduction to CUDA C/C++

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<>>>`
  - Manage GPU memory
    - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
  - Manage communication and synchronization
    - `__shared__`, `__syncthreads()`
    - `cudaMemcpy()` VS `cudaMemcpyAsync()`,  
`cudaDeviceSynchronize()`

# Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

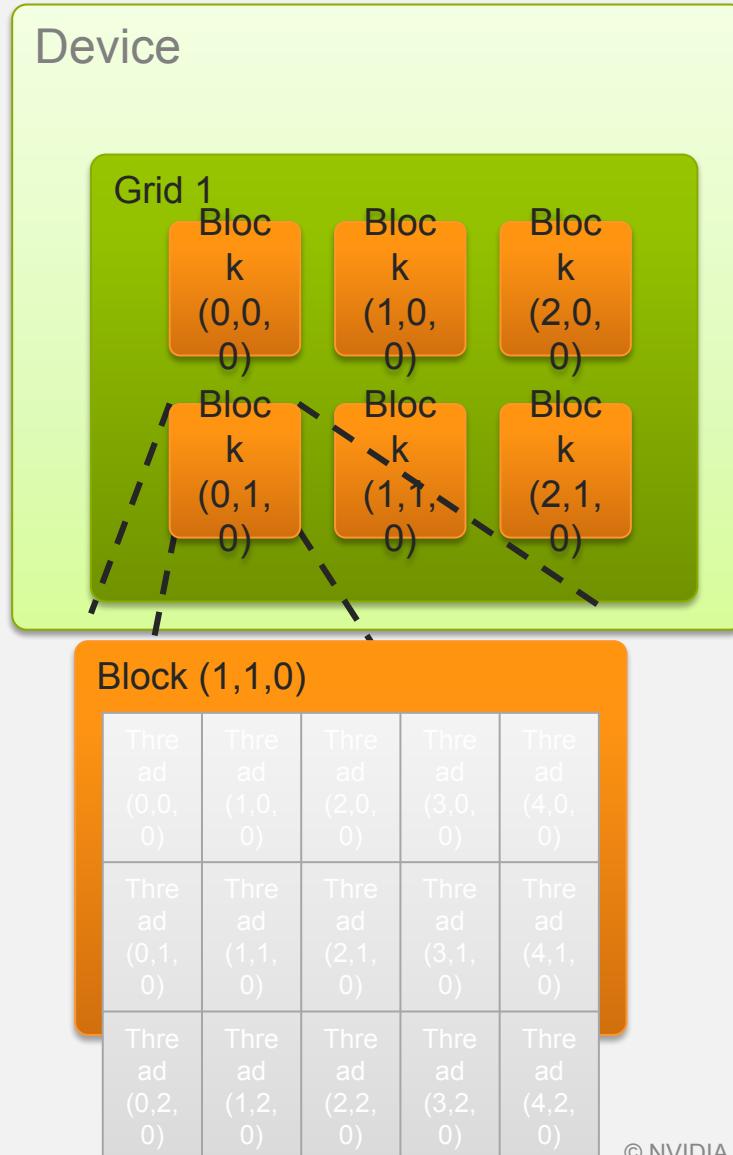
- The following presentations concentrate on Fermi devices
  - Compute Capability  $\geq 2.0$

# IDs and Dimensions

- A kernel is launched as a grid of blocks of threads

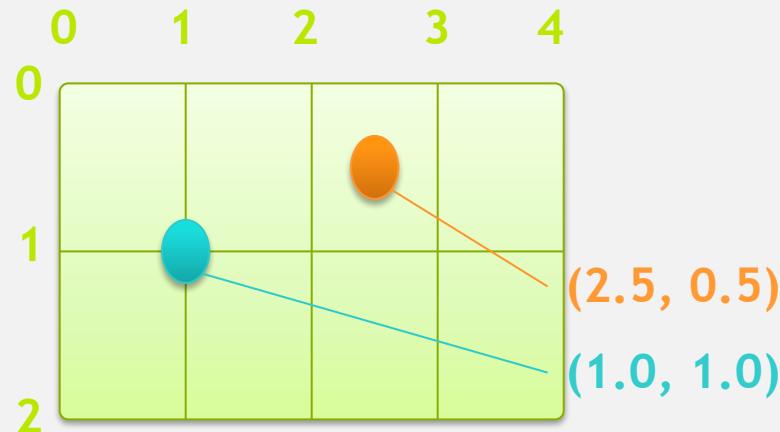
- `blockIdx` and `threadIdx` are 3D
- We showed only one dimension ( $x$ )

- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



# Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware  
(Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling  
(Wrap, clamp)



# Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more  
[developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)
- Need a quick primer for later:
  - Multi-dimensional indexing
  - Textures