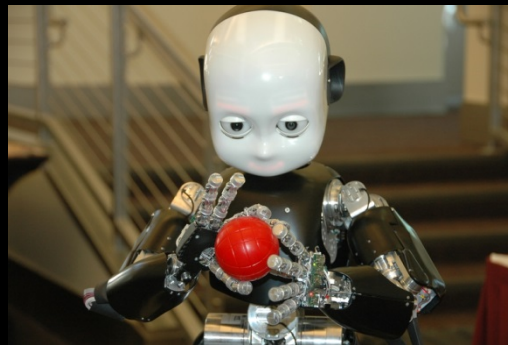


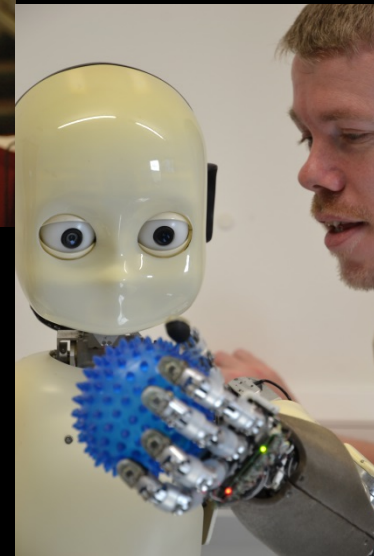
CUDA C BASIC INTRODUCTION

Who we are.....

- Dr Anthony Morse
- Plymouth University
- CUDA Teaching & Research Centre



Artificial bio-inspired
Cognitive systems



Neural networks
Evolutionary algorithms

Structure

Day 1:

Basics of CUDA C
CUDA memory model

Lectures and
Hands-on exercises

Day 2:

Error handling, Shared Memory
Optimisations
OpenACC

Material

- Practicals and slides

<https://sites.google.com/site/cudatraining>

- <https://developer.nvidia.com/cuda-education-training>

Books

- Kirk & Hwu, Programming Massively Parallel Processors, Second Edition: A Hands-on Approach
- Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs

NVIDIA offer



50% Discount on Tesla

- Complete the online survey
- Complete the attendance form

Prerequisites

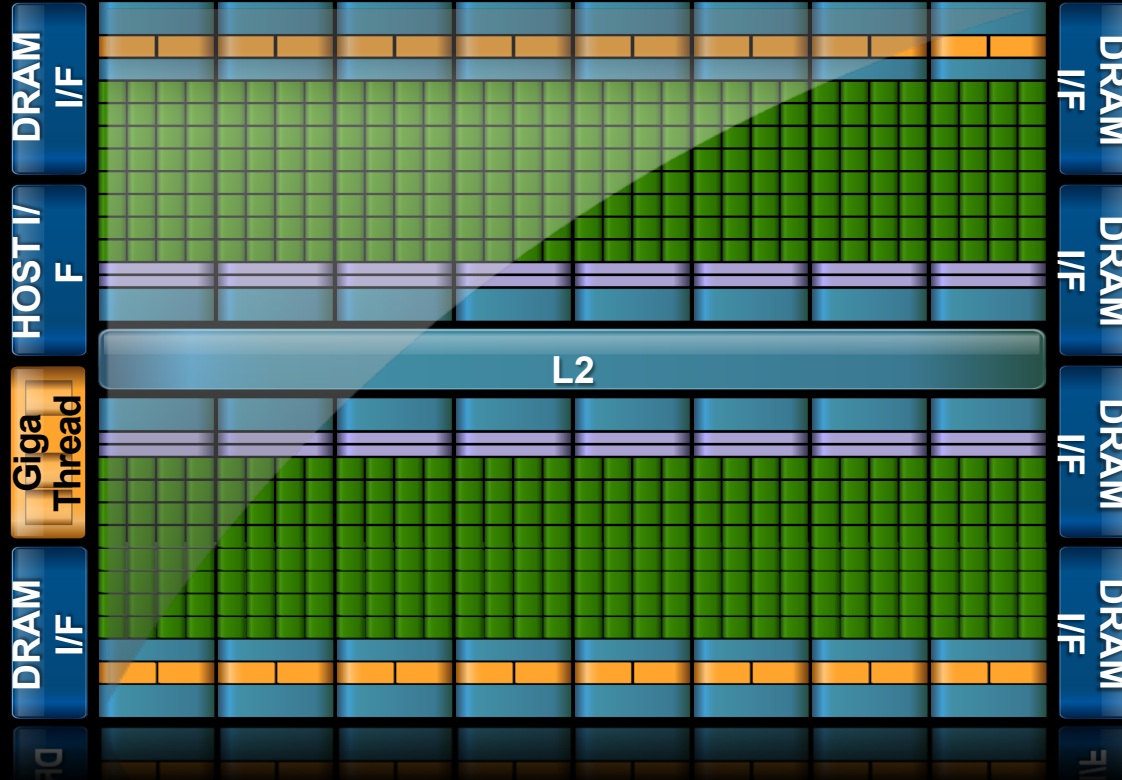
- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

What is CUDA?

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

NVIDIA GPU Architecture

Fermi GF100 (2010)

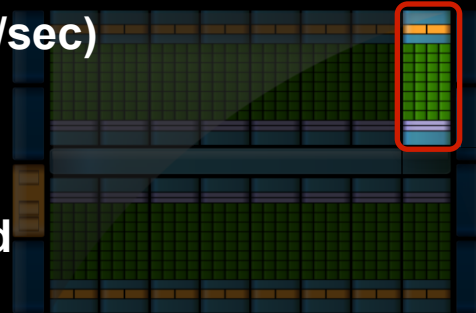


- **32 CUDA Cores per SM (512 total)**

- **Direct load/store to memory**
 - Usual linear sequence of bytes
 - High bandwidth (Hundreds GB/sec)

- **64KB of fast, on-chip RAM**
 - Software or hardware-managed
 - Shared amongst CUDA cores
 - Enables thread communication

SIMT (Single Instruction Multiple Thread) execution



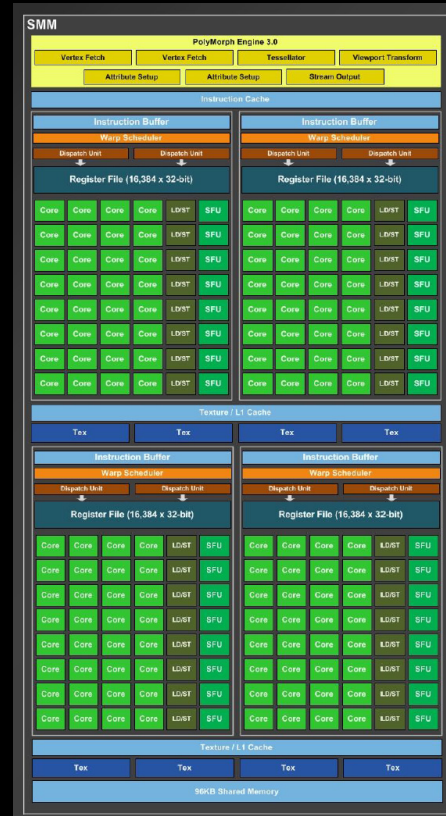
NVIDIA GPU Architecture

Kepler GK110 (2012)



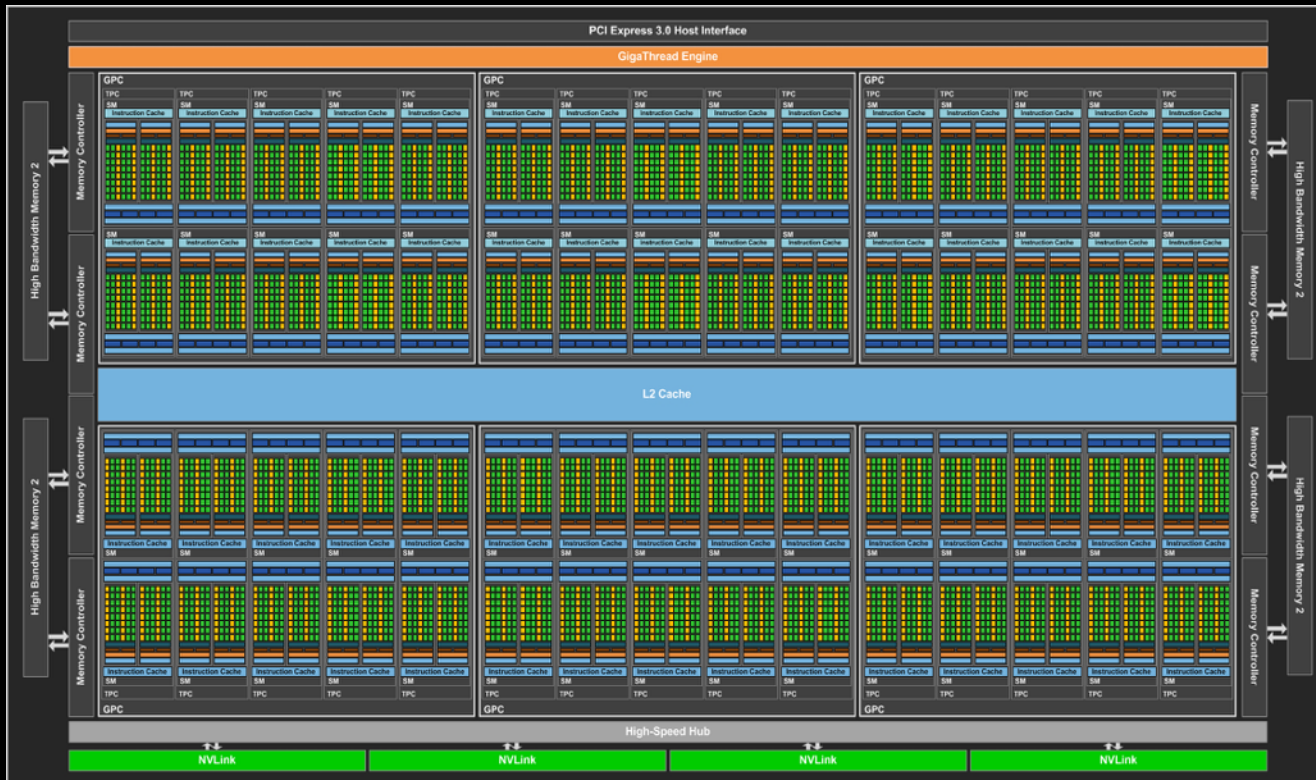
NVIDIA GPU Architecture

Maxwell GM204 (2014)



NVIDIA GPU Architecture

Pascal P100 (2016)



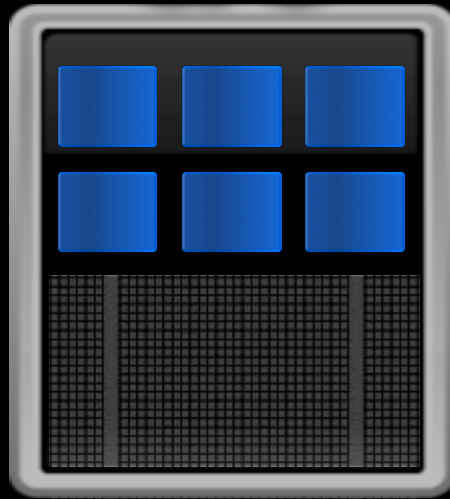
Compute Capability

- The compute capability of a device describes its architecture, e.g.
 - Number of SMs and 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, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series
3.5	Dynamic parallelism, HyperQ	K-series
5.2	C11 support	GTX980
6.0	???	P100

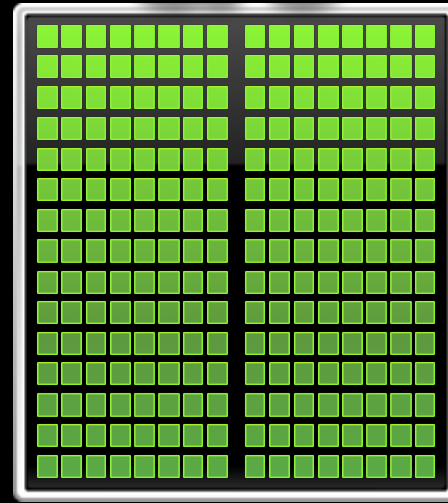
Heterogeneous computing

CPU

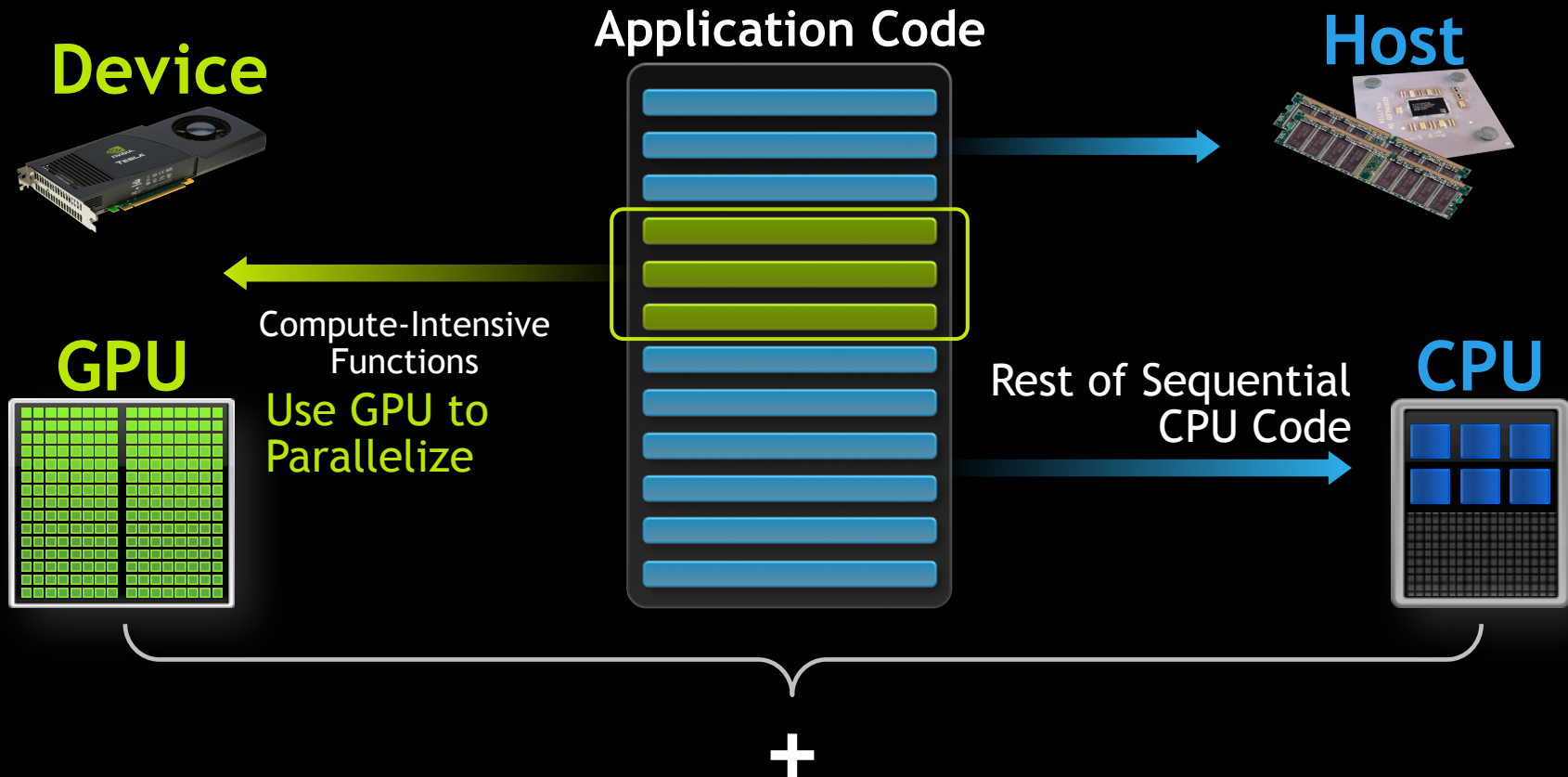


+

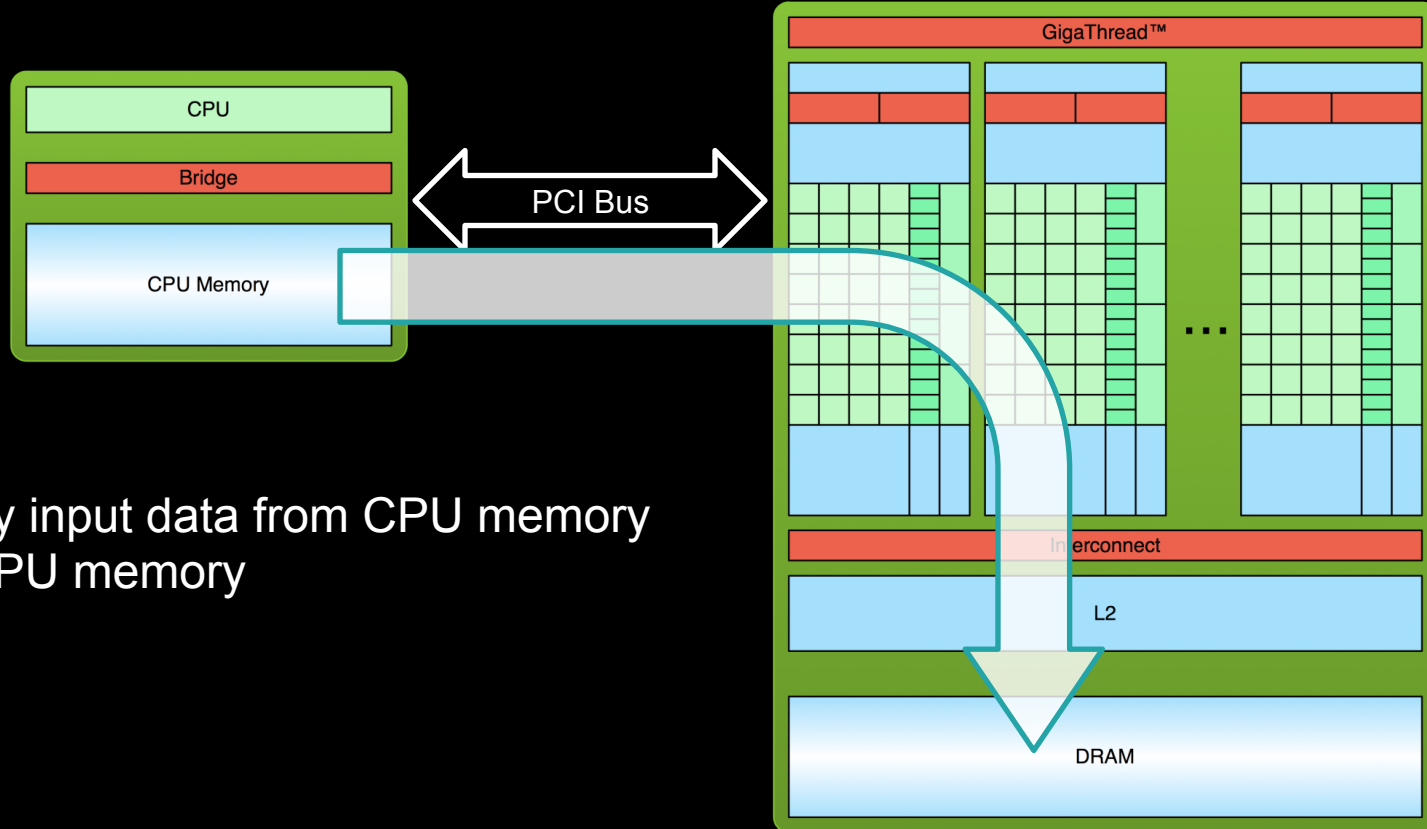
GPU



Heterogeneous computing

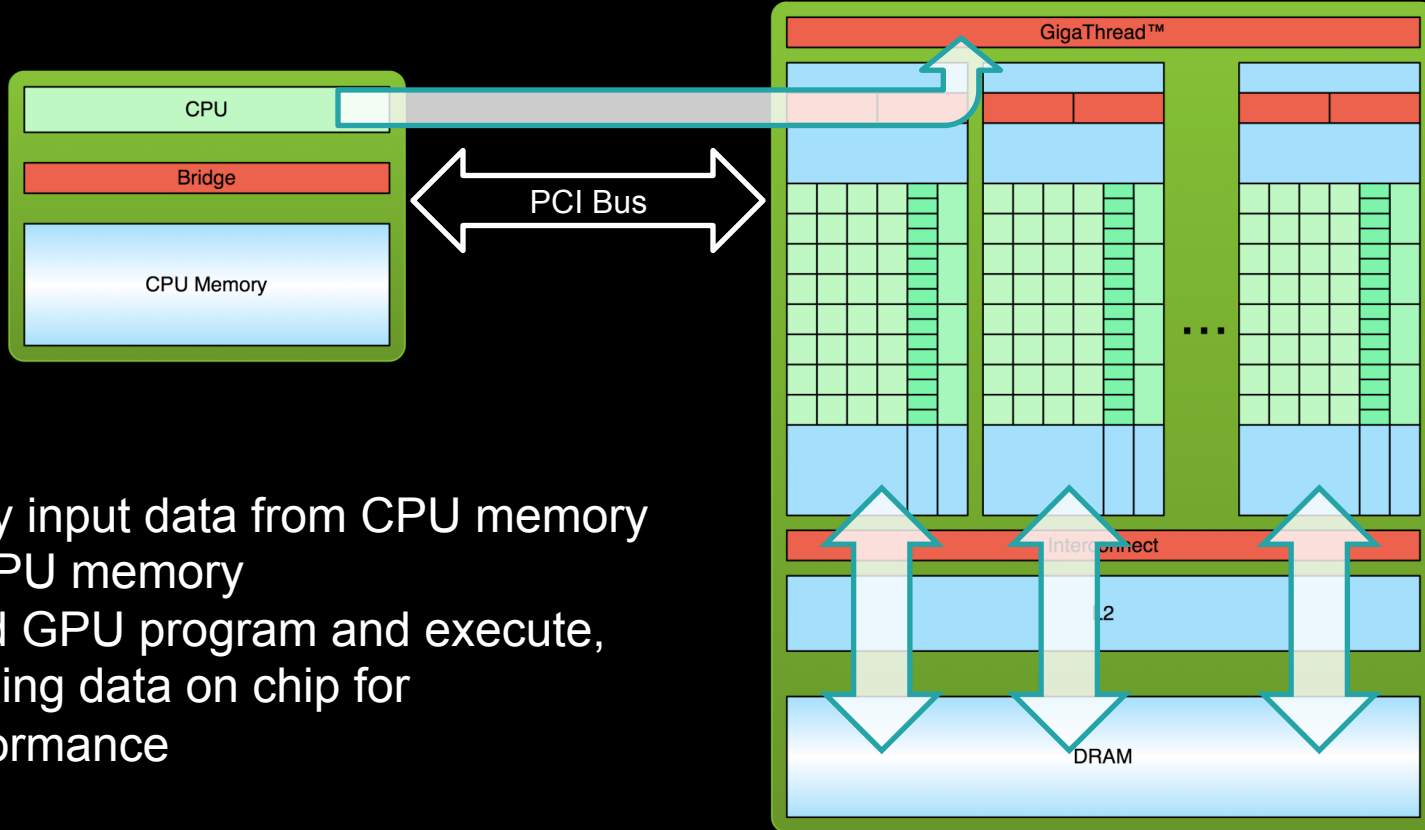


Simple Processing Flow



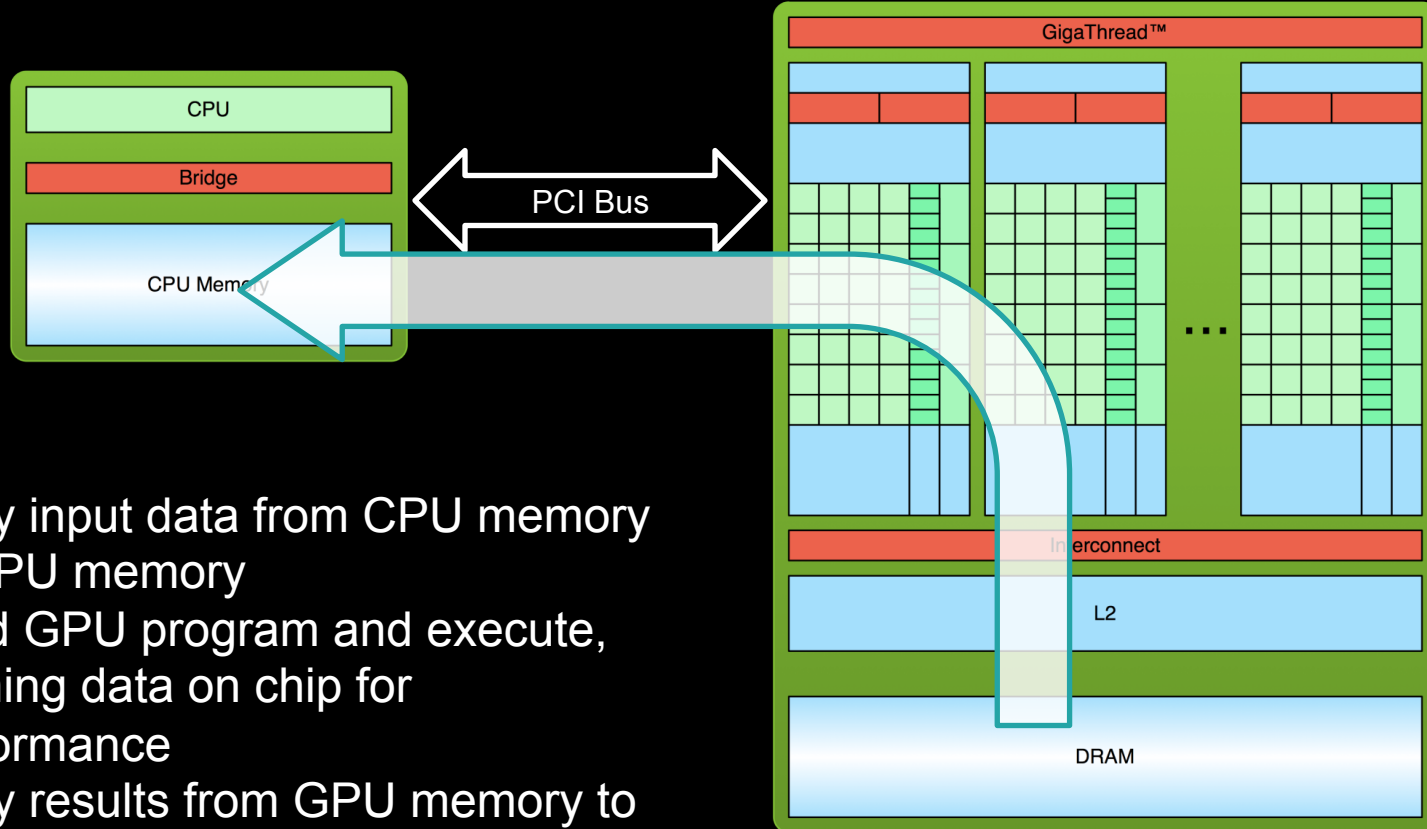
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing Flow



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

Hello World!

```
int main(void) {  
    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(void) {  
  
    }  
  
int main(void) {  
    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”
- 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;  
}
```

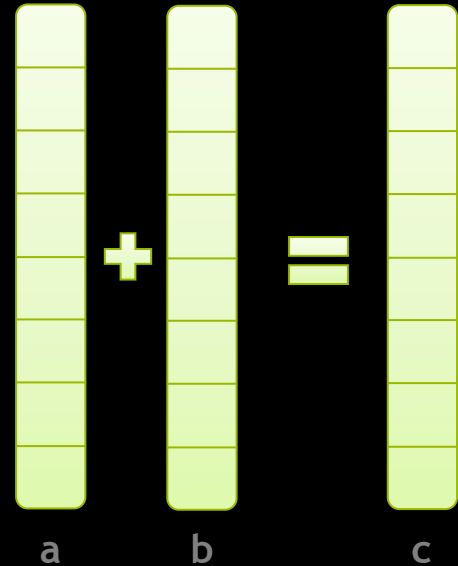
- mykernel() does nothing....

Output:

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

Parallel Programming in CUDA C/C++

- 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**

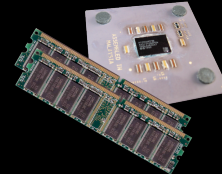
- May be passed to/from host code

- 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);  
    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;
}
```

Memory Management & Unified Memory

- From **compute capacity 3.0+** and **CUDA 6.0+**
 - Unified Memory creates a pool of managed memory that is shared between the CPU and GPU and accessible to both using a single pointer.
 - The system automatically *migrates* data allocated in Unified Memory between host and device.

```
int main() {  
    float *data;  
    cudaMallocManaged (&data, dataSize * sizeof(float));  
    ...  
    cudaFree(data);  
}
```

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

CUDA logical architecture

- A kernel is launched as a **grid** of **blocks** of **threads**
 - `blockIdx` and `threadIdx` can represent up to 3 dimensions
- Built-in variables:
 - `threadIdx`
 - `blockIdx`
 - `blockDim`
 - `gridDim`

Device

Grid 1

Block
(0,0,0)

Block
(1,0,0)

Block
(2,0,0)

Block
(0,1,0)

Block
(1,1,0)

Block
(2,1,0)

Block (1,1,0)

Thread
(0,0,0)

Thread
(1,0,0)

Thread
(2,0,0)

Thread
(3,0,0)

Thread
(4,0,0)

Thread
(0,1,0)

Thread
(1,1,0)

Thread
(2,1,0)

Thread
(3,1,0)

Thread
(4,1,0)

Thread
(0,2,0)

Thread
(1,2,0)

Thread
(2,2,0)

Thread
(3,2,0)

Thread
(4,2,0)

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`

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

- By using `blockIdx.x` to index into the array, each block handles a different index

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;
}
```

CUDA logical architecture

- A kernel is launched as a grid of blocks of threads
 - `blockIdx` and `threadIdx` can represent up to 3 dimensions
- Built-in variables:
 - `threadIdx`
 - `blockIdx`
 - `blockDim`
 - `gridDim`

Device

Grid 1



Block (1,1,0)

Thread (0,0,0)	Thread (1,0,0)	Thread (2,0,0)	Thread (3,0,0)	Thread (4,0,0)
Thread (0,1,0)	Thread (1,1,0)	Thread (2,1,0)	Thread (3,1,0)	Thread (4,1,0)
Thread (0,2,0)	Thread (1,2,0)	Thread (2,2,0)	Thread (3,2,0)	Thread (4,2,0)

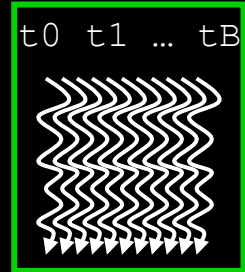
Threads Hierarchy

- **A block can be split into parallel threads**
all threads execute the same sequential program
- **Thread block is a group of threads that can:**
 - Synchronize their execution
 - Communicate via shared memory

Thread t



Block b



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);

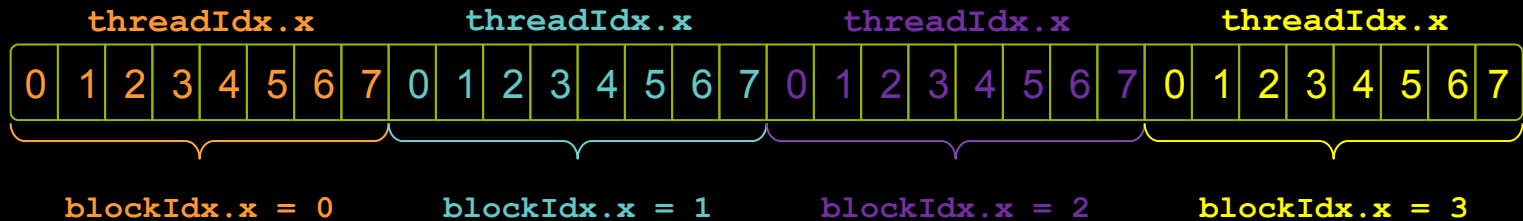
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Adapting vector addition to use both *blocks* and *threads*
- 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)

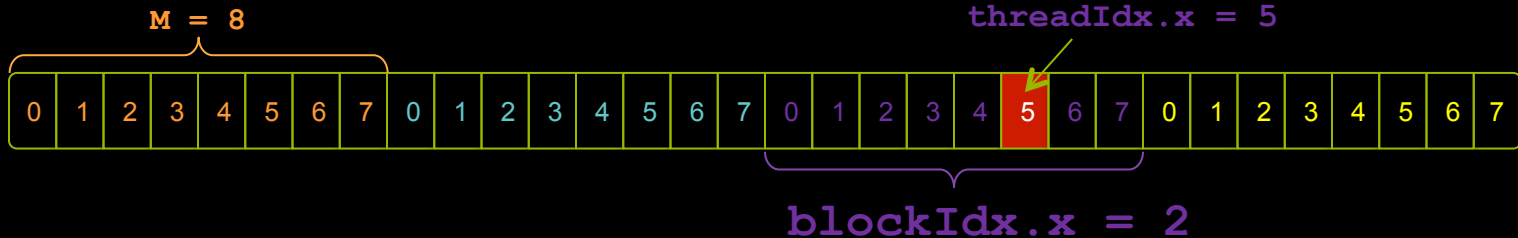


- With M threads/block a unique index for each thread is given by:

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

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);
```

Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )  
{  
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;  
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;  
    int idx = iy*dimx + ix;  
  
    a[idx] = a[idx]+1;  
}
```

Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}
```

```
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

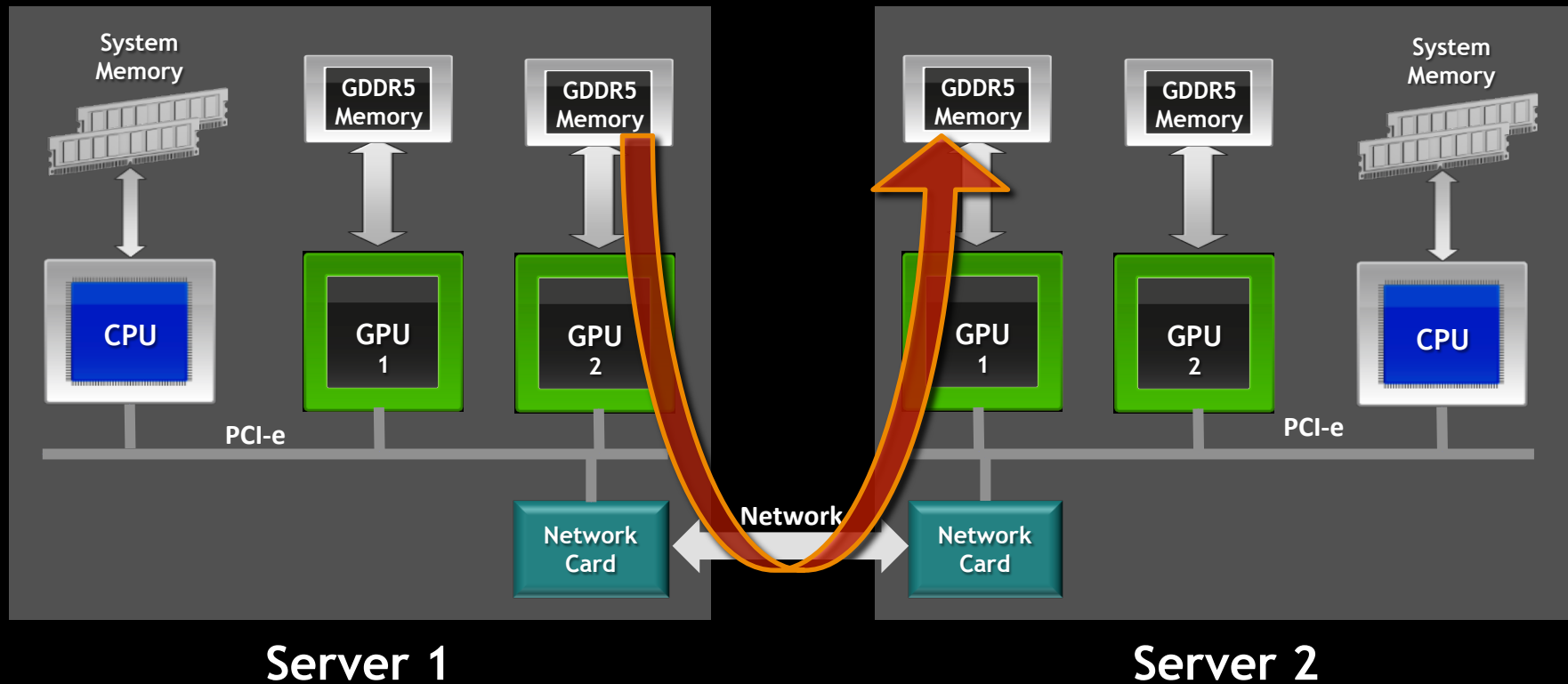
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    .....
}
```

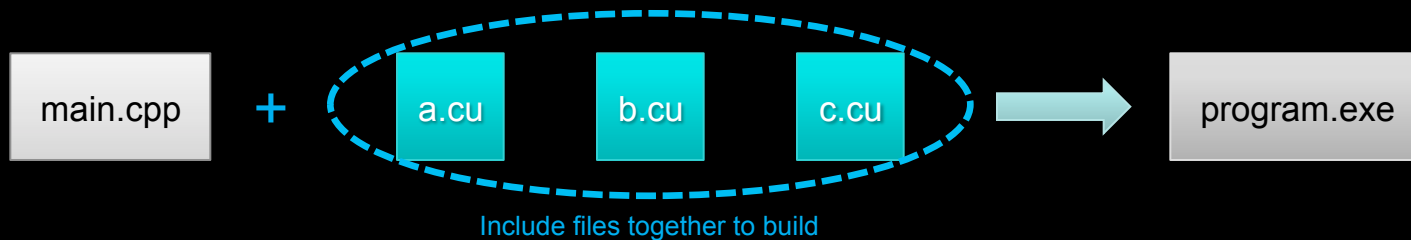
End

NVIDIA GPUDirect™ Now Supports RDMA

GPU COMPUTING
WITH
PLYMOUTH
UNIVERSITY

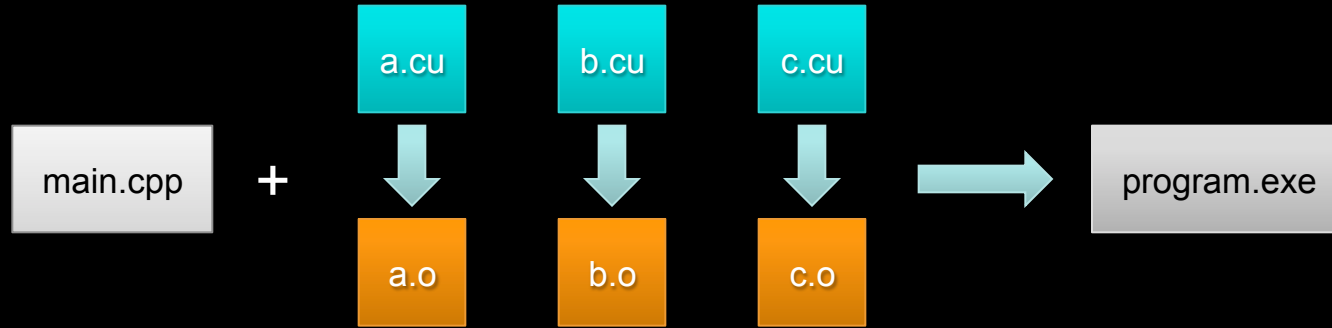


CUDA 4: Whole-Program Compilation, Linking



CUDA 4 required all GPU kernel source included into one file
No linking external device code

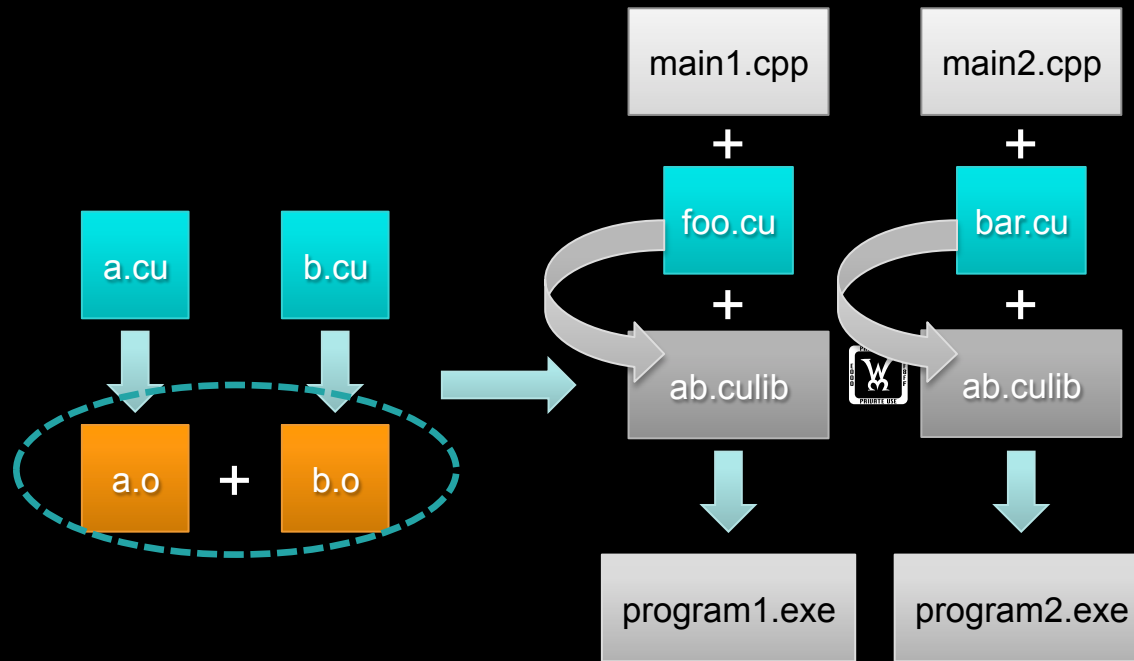
CUDA 5: GPU Library Object Linking



Separate compilation allows building independent object files

CUDA 5 can link multiple object files into one program

CUDA 5: GPU Library Object Linking



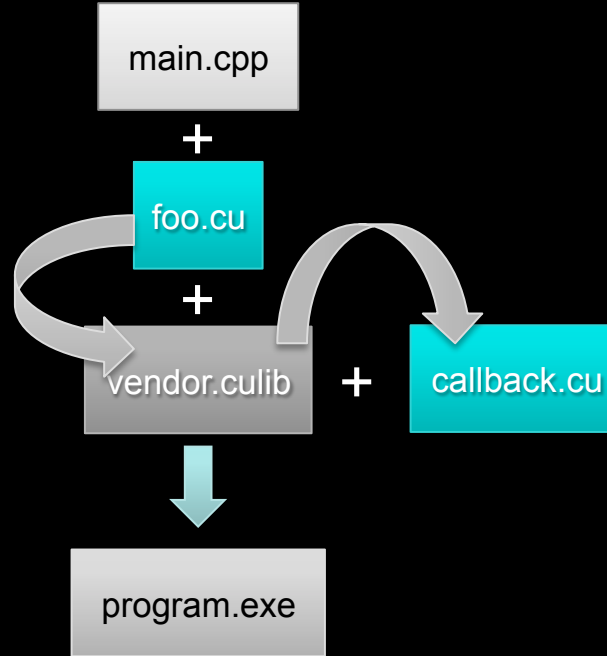
Can also combine object files into static libraries

Link and externally call device code

Facilitates code reuse, reduces compile time

CUDA 5: GPU Library Object Linking

Enables 3rd party
closed-source
device libraries



User-defined
device callback
functions

INTRODUCING: DYNAMIC PARALLELISM

Dynamic Parallelism

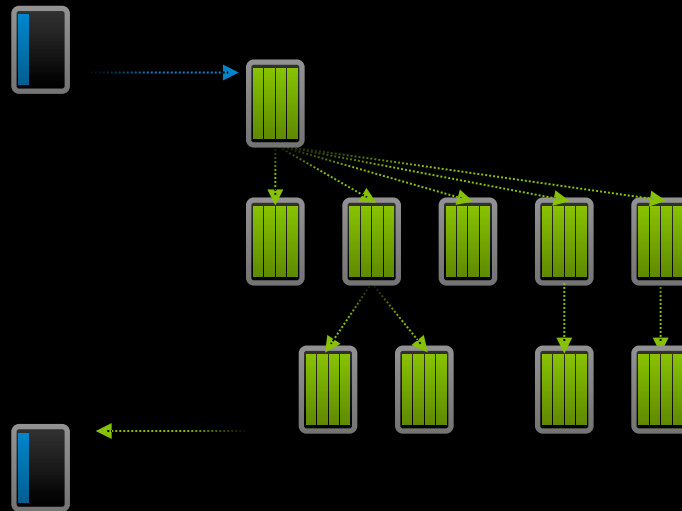
CPU

Fermi GPU



CPU

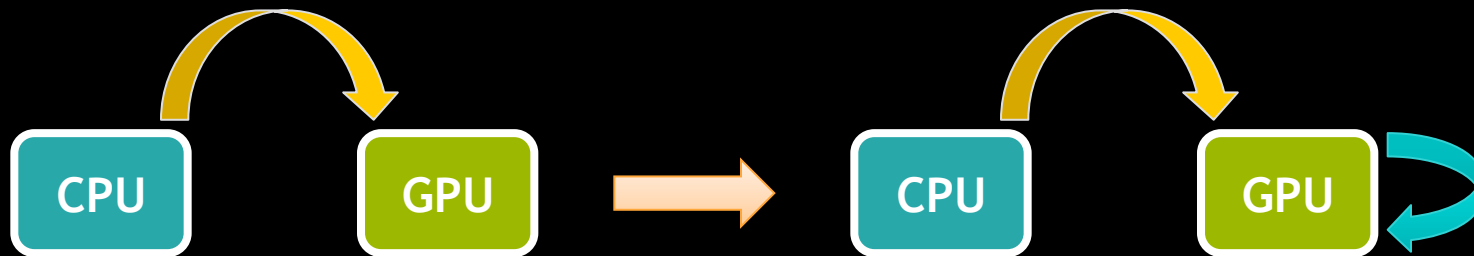
Kepler GPU



What is Dynamic Parallelism?

The ability to launch new kernels from the GPU

- Dynamically - based on run-time data
- Simultaneously - from multiple threads at once
- Independently - each thread can launch a different grid



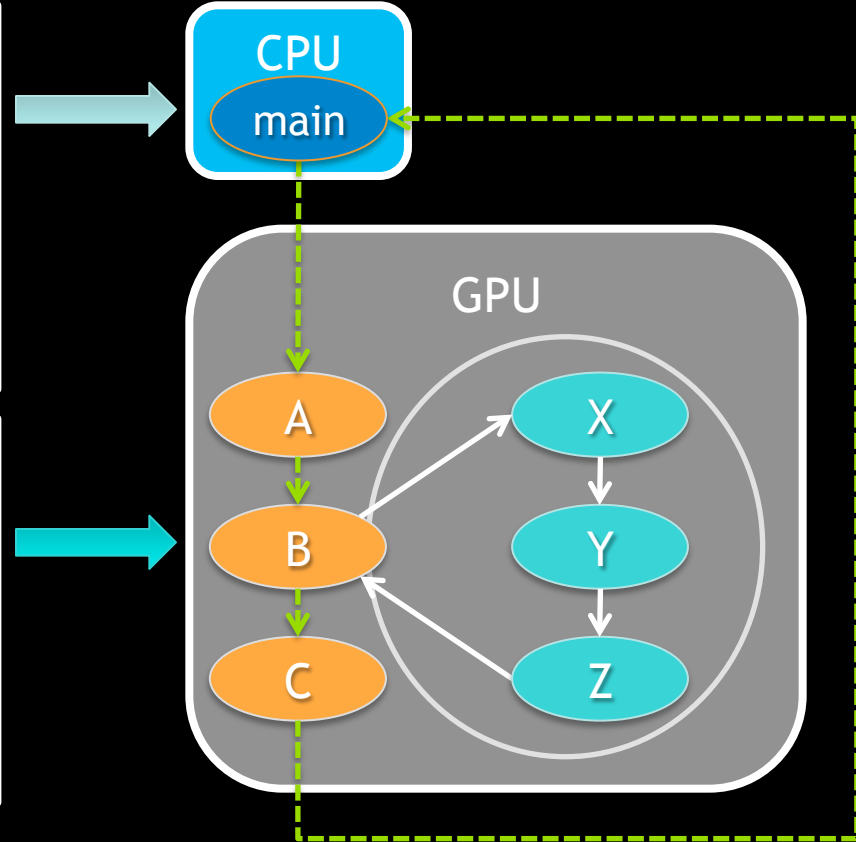
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

Familiar Syntax and Programming Model

```
int main() {  
    float *data;  
    setup(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
  
    cudaDeviceSynchronize();  
    return 0;  
}
```

```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



Simpler Code: LU Example

LU decomposition (Fermi)

```
dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>>
      memcpy
      dswap<<<>>>
      memcpy
      dscal<<<>>>
      dger<<<>>>
    next i

    memcpy
    dlaswap<<<>>>
    dtrsm<<<>>>
    dgemm<<<>>>
  next j
}
```

CPU Code

GPU Code

LU decomposition (Kepler)

```
dgetrf(N, N) {
  dgetrf<<<>>>
}

synchronize();
```

CPU Code

GPU Code

CPU is Free