

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Introduction CUDA C

Instructor: Haidar M. Harmanani

Spring 2017

CONCEPTS

Heterogeneous Computing

Hello World!

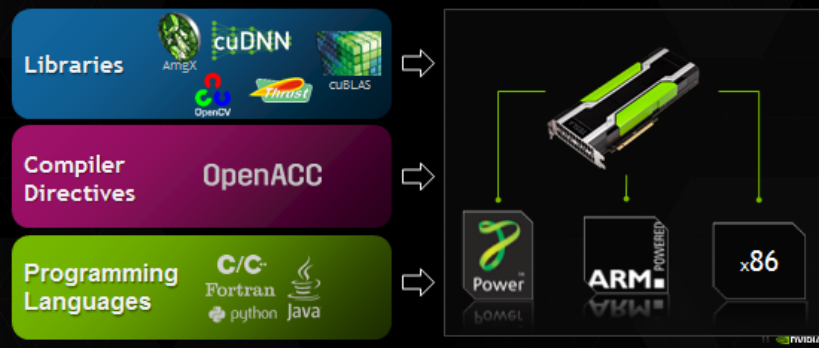
Compiling



- CUDA is a parallel computing platform and programming model invented by NVIDIA
- Enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)

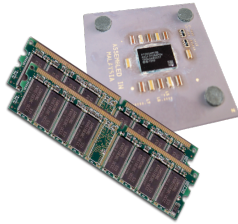
COMMON PROGRAMMING APPROACHES

Across a Variety of Heterogeneous Systems



Heterogeneous Computing

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



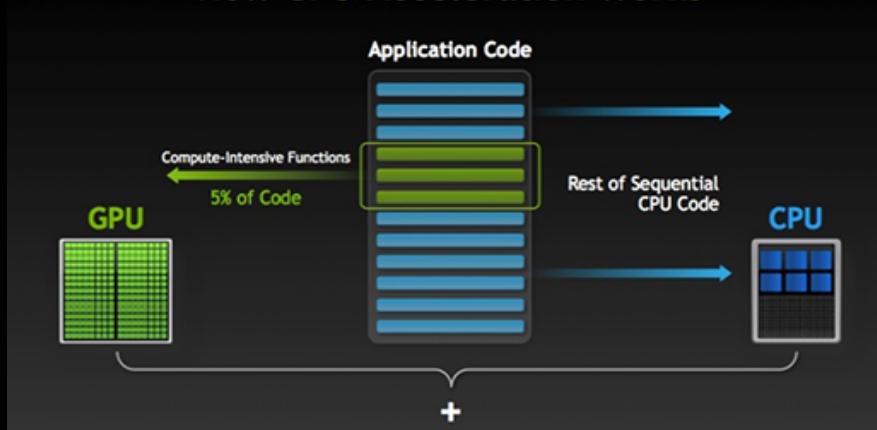
Host



Device

Heterogeneous Parallel Computing

How GPU Acceleration Works



Heterogeneous Computing

```

#include <iostream>
#include <algorithm>

using namespace std;

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

__global__ void stencil_1d(int *a, int *out) {
    // Read input elements into shared memory
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < N; i += stride) {
        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS; offset <= RADIUS; offset++)
            result += a[i + offset];

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

int main(int argc, char **argv) {
    // Host copies of a, b, c
    int *a, *b, *c;
    int *out;
    int n = N;
    int r = RADIUS;
    int bsize = BLOCK_SIZE;

    // Allocate space for host copies and setup vectors
    a = (int *) malloc(n * sizeof(int));
    b = (int *) malloc(n * sizeof(int));
    c = (int *) malloc(n * sizeof(int));
    out = (int *) malloc(n * sizeof(int));

    // Copy to device
    cudaMalloc((void **) &a_dev, n * sizeof(int));
    cudaMalloc((void **) &b_dev, n * sizeof(int));
    cudaMalloc((void **) &c_dev, n * sizeof(int));
    cudaMalloc((void **) &out_dev, n * sizeof(int));

    // Launch stencil_1d kernel on GPU
    stencil_1d(<math>a\_dev</math>, <math>b\_dev</math>, <math>c\_dev</math>, <math>out\_dev</math>);

    // Copy result back to host
    cudaMemcpy(out, out_dev, n * sizeof(int), cudaMemcpyDeviceToHost);

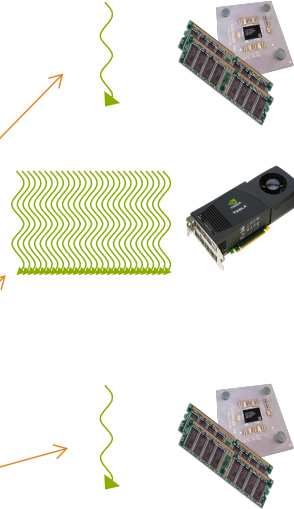
    // Cleanup
    free(a); free(b); free(c); free(out);
    return 0;
}
    
```

parallel fn

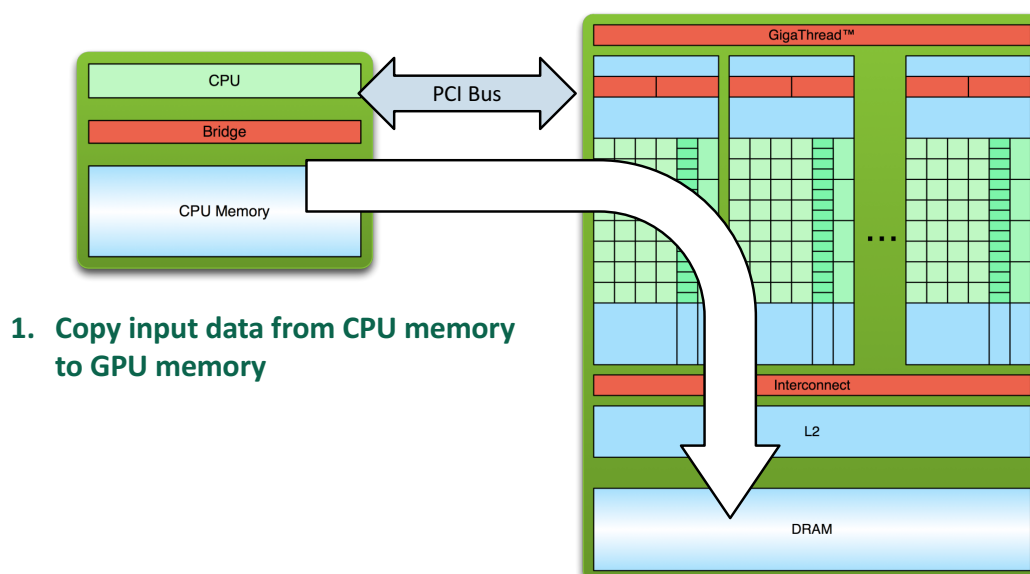
serial code

parallel code

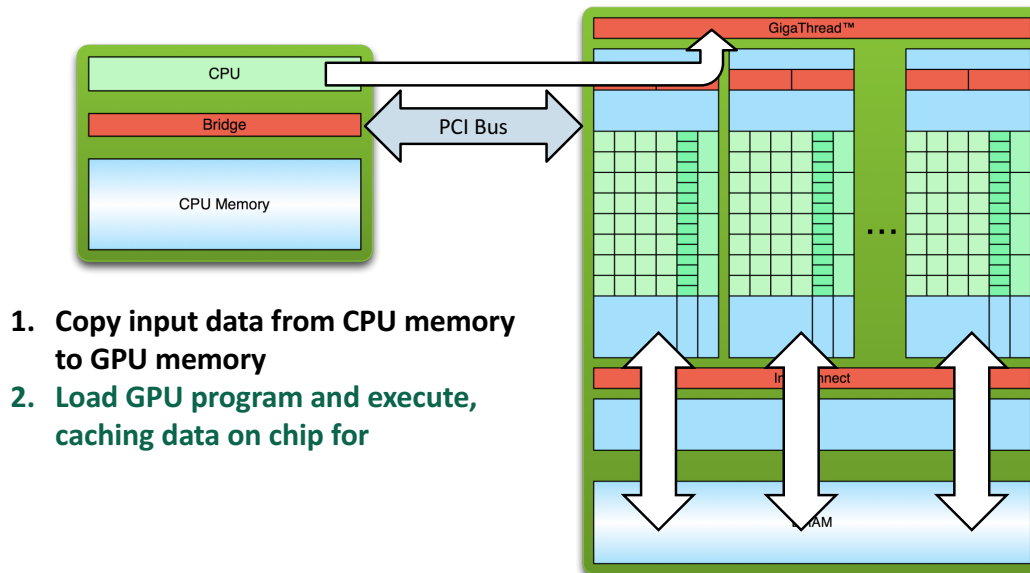
serial code



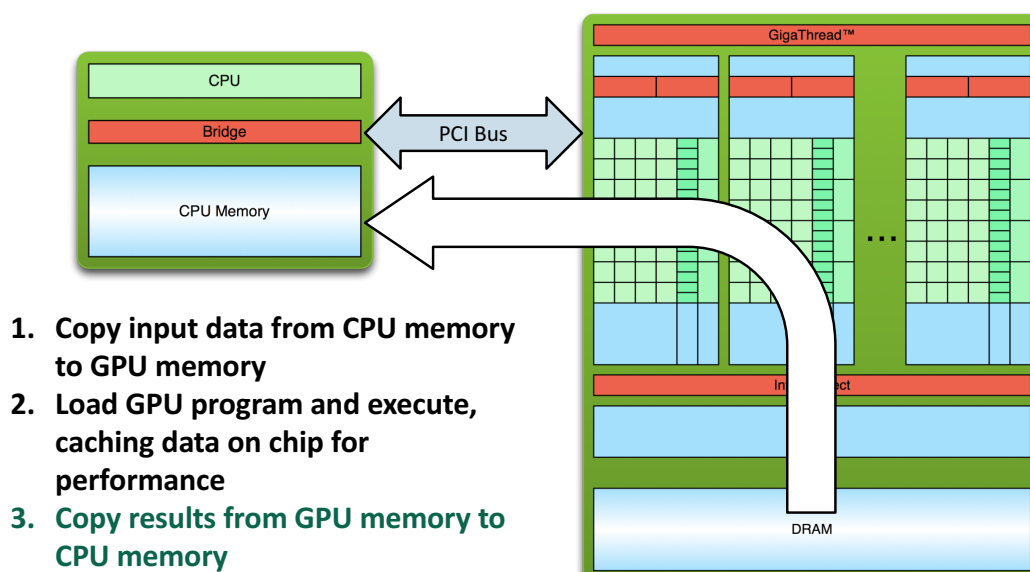
Simple Processing Flow



Simple Processing Flow



Simple Processing Flow



Hello World!

Heterogeneous Computing

Hello World!

Compiling

CUDA

- CUDA provides a simple grid model
 - Can decompose into CUDA blocks
- A linear distribution of work within a single block leads to an ideal decomposition into CUDA blocks.
 - Can assign up to sixteen blocks per SM and we can have up to 16 SMs (32 on some GPUs), any number of blocks of 256 or larger is fine.
 - In practice, limit the number of elements within the block to 128, 256, or 512

Blocks, Threads, and Grids

- CUDA C allows the definition of a group of blocks in two dimensions
- Use two-dimensional indexing for problems such as matrix math or image processing
 - Avoid annoying translations from linear to rectangular indices
- A collection of parallel blocks is called a grid
- No dimension of a launch of blocks may exceed 65,535

CUDA Hello World

- We will discuss over the following few slides a couple of simple examples
- We will start with the classical `hello_world` example

Hello World (Host Only)

- Recall
 - *Host*–The CPU and its memory (host memory)
 - *Device*–The GPU and its memory (device memory)

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

Hello World (Device Code)

- Recall
 - *Host*–The CPU and its memory (host memory)
 - *Device*–The GPU and its memory (device memory)

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

Output:

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


Hello World (Device Code)

- CUDA C keyword `__global__` indicates that a function
 - Runs on the device
 - Called from host code
- `kernel<<< 1, 1 >>>();`
 - Triple angle brackets mark a call from hostcode to devicecode
 - Sometimes called a “kernel launch”
- nvcc splits the source file into host and device components
 - NVIDIA’s compiler handles device functions like kernel()
 - Standard host compiler handles host functions like main()
 - gcc
 - Microsoft Visual C