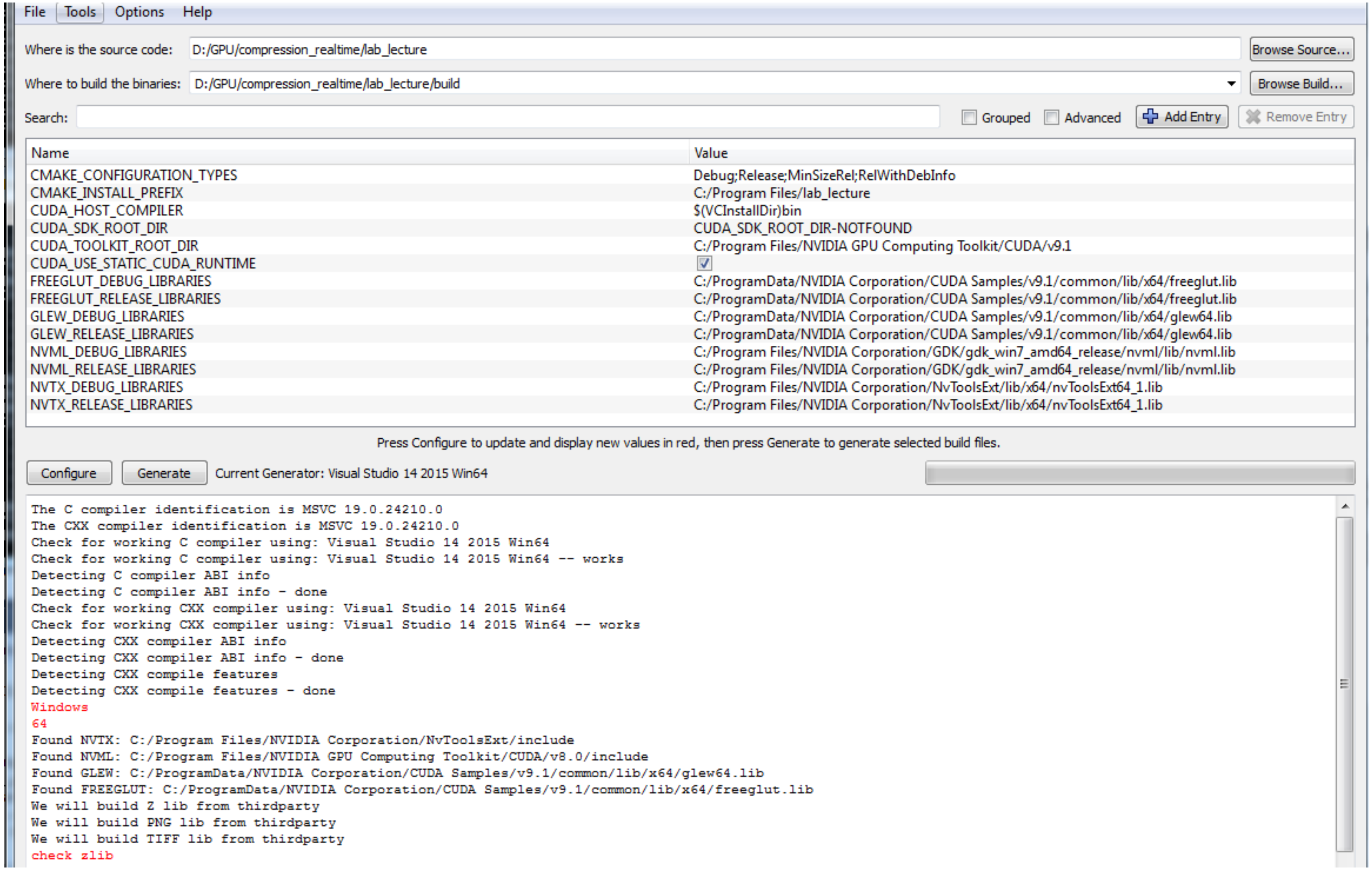
Lab Lecture

ECE 277

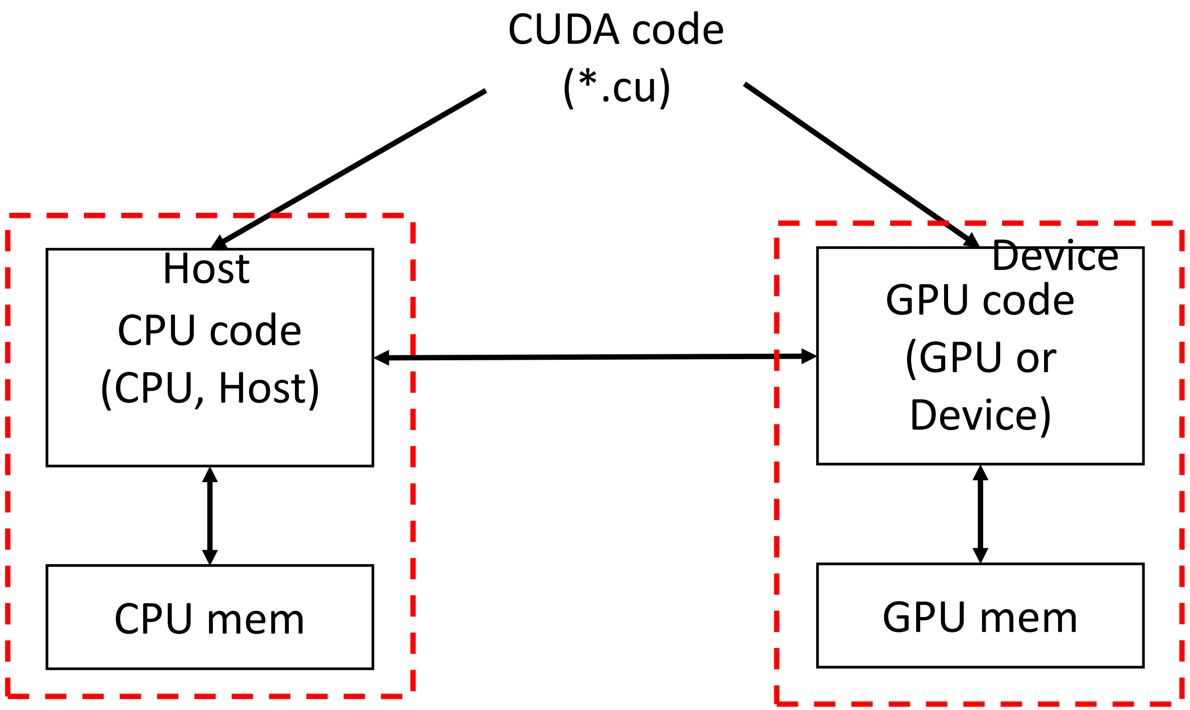
Cheolhong An

CMake setup: Create a VS project solution



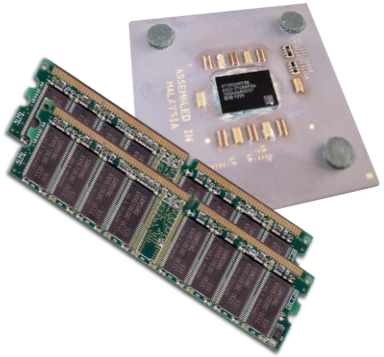
Heterogeneous Computing

- A single code can run both CPU and GPU
- CUDA (*.cu) is a heterogenous programming language



Heterogeneous Computing

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



Host (CPU)

Device (GPU)

Mark Harris, **Introduction to CUDA C,** NVIDIA Corporation

Heterogeneous Computing

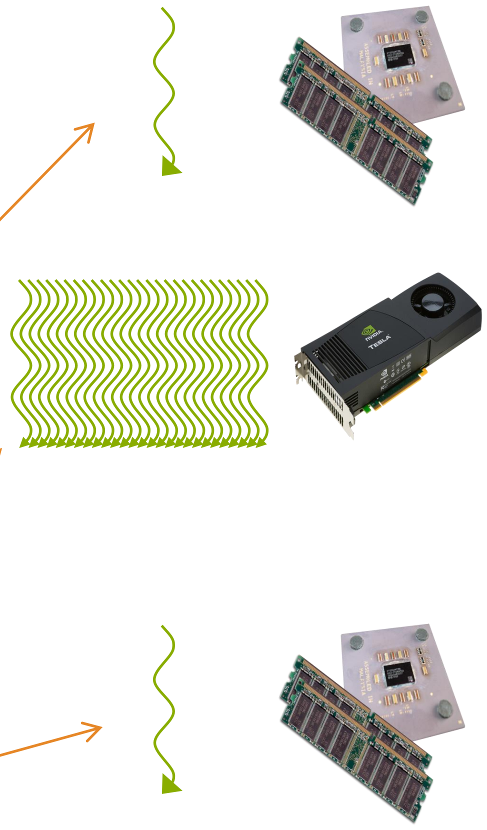
```
#include <iostream>
#include <algorithm>

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;
}
```

parallel fn

serial code

parallel code

serial code



Mark Harris, **Introduction to CUDA C,** NVIDIA Corporation

Hello.cu

- Create your own project (Your_name)
  1) Create one directory (any name is fine, e.g. “test”) under “Src” directory
  2) Add “add_subdirectory(your directory name)” in CmakeLists.txt under “Src” directory
     e.g. add_subdirectory(test)
  3) Copy CMakeLists.txt under “SimpleGL” into your directory (“test”)
  4) Replace project name (“check_system”) with your name, which is the project name in CMakeList.txt
  5) Create Hello.cu file in your directory (“test”)
  6) Replace simpleGL.cu with Hello.cu in CMakeList.txt
  7) Edit Hello.cu with the next slide functions
  8) Configure CMake using CMake-gui or rebuild solution
     If you encounter any error, you should use CMake-gui to check the error message

- Create multiple threads in a thread block
- Create multiple threadblocks in a grid

Hello World! with Device Code

```
__global__ void mykernel(void) {
        // Print your name, site number
}

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

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 (CUDA compiler) separates source code into host and device components
  - Device functions (e.g. hello**()**) processed by NVIDIA compiler
  - Host functions (e.g. **main()**) processed by standard host compiler
    - **gcc, cl.exe**

Common CUDA code structure

- Ex) vectorAdd

CPU memory allocation (malloc) <- Host
GPU memory allocation (cudaMalloc) <- Device

↓

Transfer data from Host to Device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice)

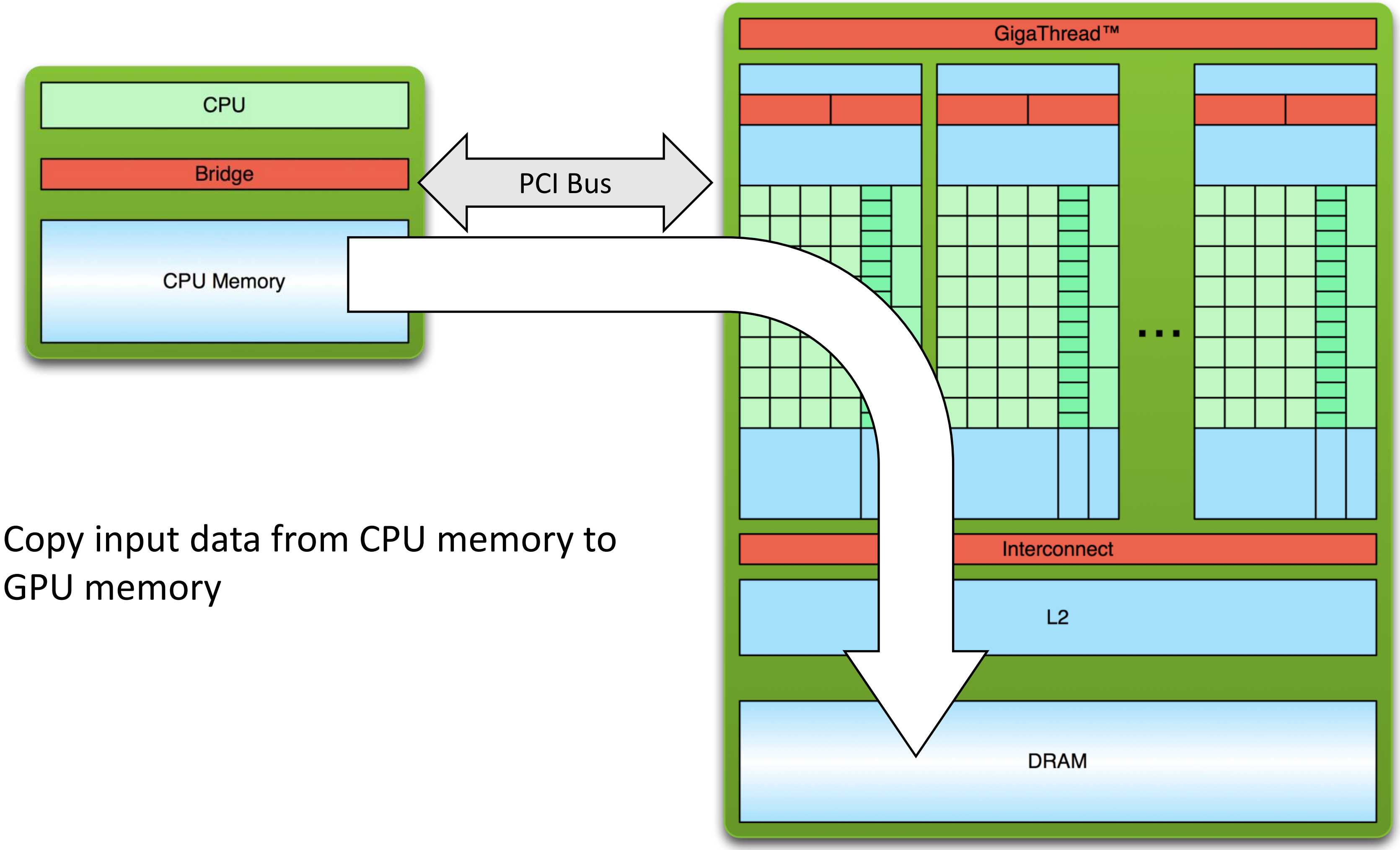
↓

Run GPU codes (Call kernel functions)
Kernel_fun<<<,>>> ()

↓

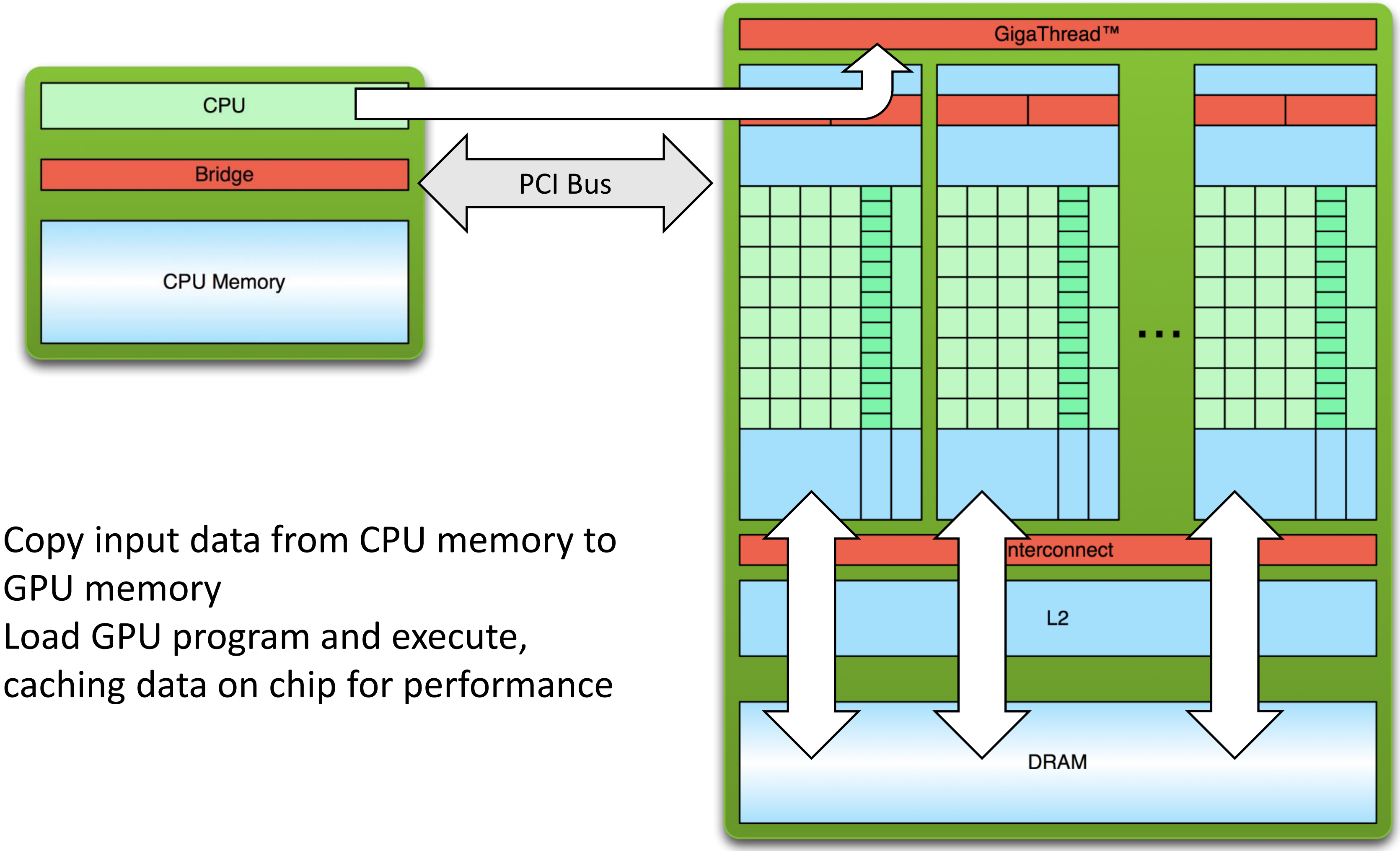
Transfer processed data from Device to Host
cudaMemcpy(a, d_a, size, cudaMemcpyDeviceToHost)

Transfer data from Host to Device



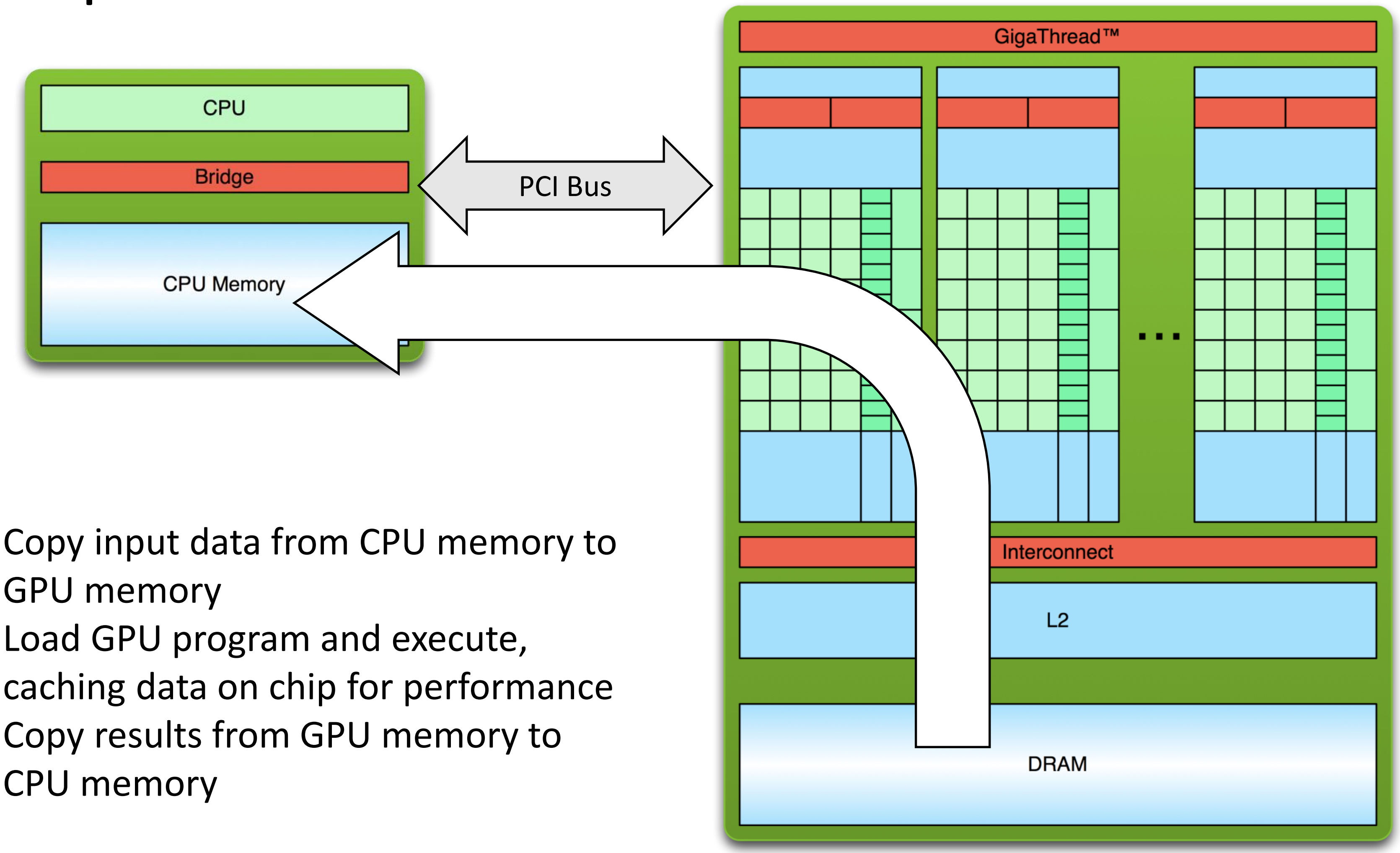
1. Copy input data from CPU memory to GPU memory

Run GPU codes



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

Transfer processed data from Device to Host



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

Mark Harris, **Introduction to CUDA C,** NVIDIA Corporation

Pytorch High-level Interface

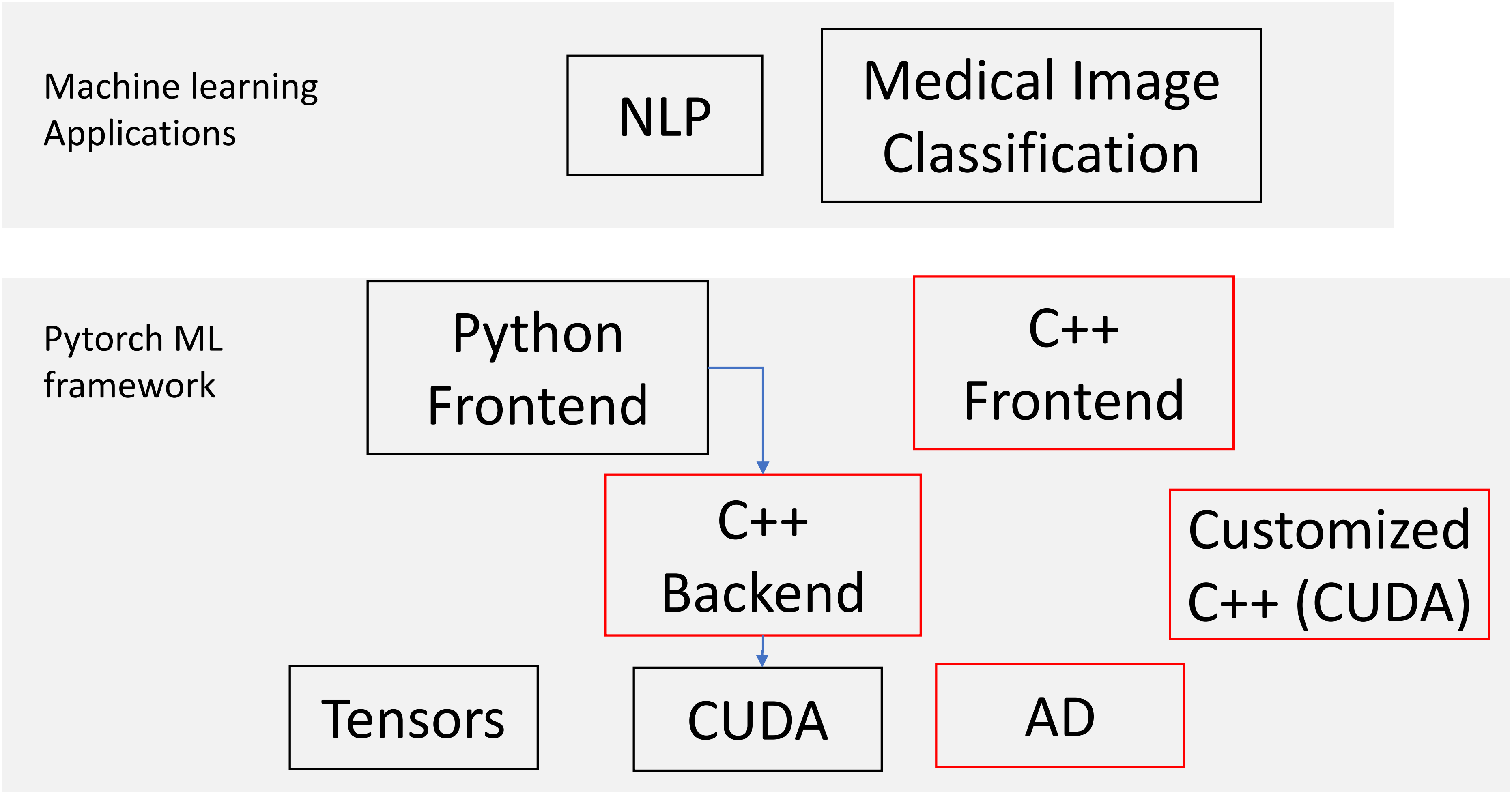
```
device = torch.device("cuda" if use_cuda else "cpu")
model = Net().to(device)

def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data = data.to(device)
        target = target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

CPU to GPU transfer

CUDA variables

CUDA operation

Pytorch framework with CUDA



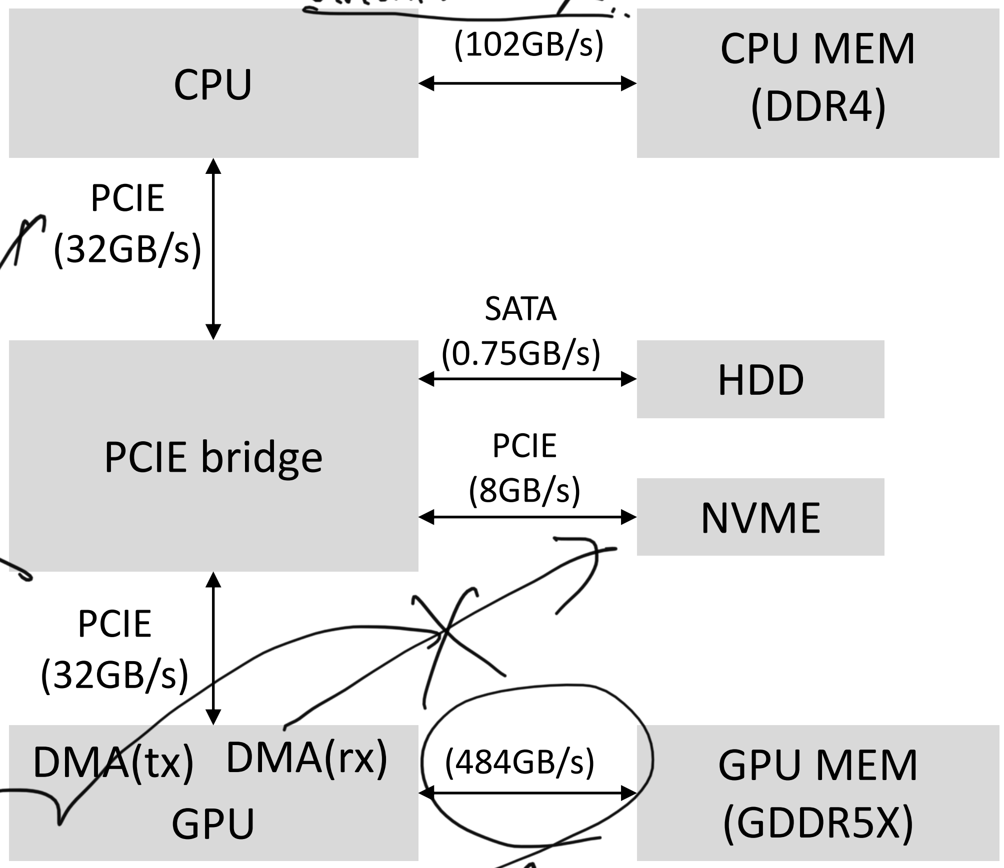
AD: Automatic differentiation

Check bandwidth

- Memory bandwidth
  - Host to Device
  - Device to Host
  - Device to Device
- Run “bandwidth Test”

malloc ⇒ ≥ physical memory

virtual memory.



low !!

only CPU part.

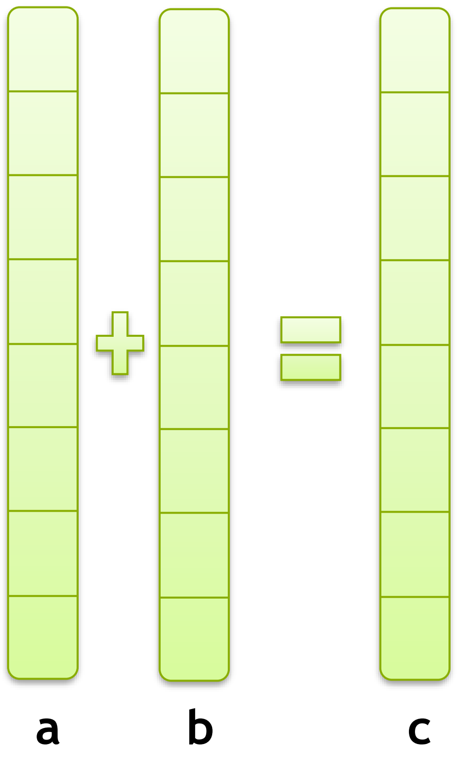
virtual memory X.

VMEM addressing
하지 않음.

high.

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