CUDA Programming model

CUDA

CUDA -Compute Unified DeviceArchitecture

- is a parallel computing platform and programming model created by NVIDIA
- Implemented by the GPUs
- CUDA gives developers access to the instruction set and memory of the parallel computational elements in CUDA GPUs.
- Using CUDA, GPUs become accessible for computation like CPUs.

CUDA

- functions for the GPU (device) and functions for the system processor (host),
- CUDA uses _device_ or _global_ for GPU and -host- for the CPU.
- CUDA variables declared in the device or global functions are allocated to the GPU Memory

CUDA Philosophy

SIMT philosophy

- Single Instruction Multiple Threads

Computationally intensive

- The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

Massively parallel

- The computations can be broken down into hundreds or thousands of independent units of work

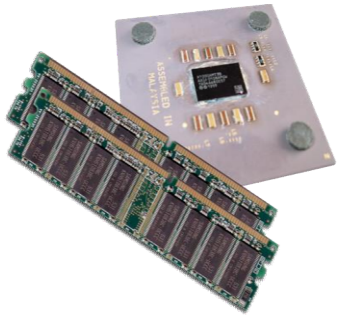
Definitions

- dimGrid - dimensions of the code (in blocks)
- dimBlock- dimensions of a block (in threads)
- *Blockldx*- identifier for blocks
- *threadldx*- identifier for threads per block
- *blockDim*- number of threads per block

Heterogeneous Computing

Host

- CPU and its memory (host memory)



Device

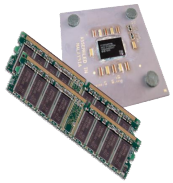
- GPU and its memory (device memory)

SSN

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;
}
```

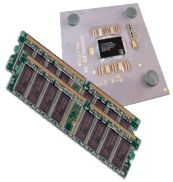
parallel fn



serial code

parallel code

serial code



Heterogeneous Computing

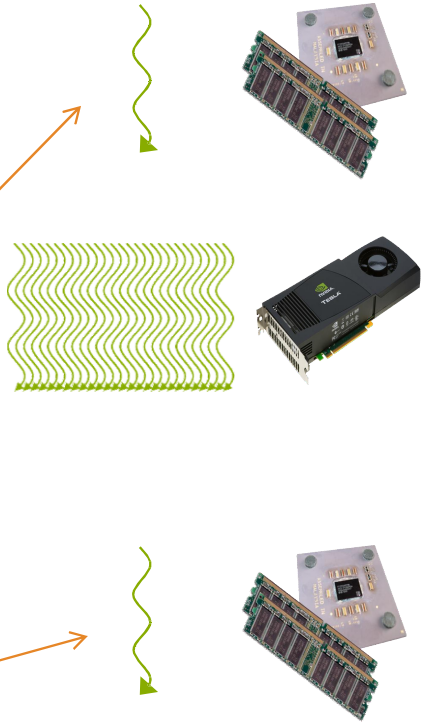
```
#include <iostream>
#include <algorithm>

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



Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu

$./a.out

$
```

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
printf("Hello world!\n");
}

int main() {
 hwkernel<<<1, 1>>>();
 cudaDeviceSynchronize()
}
```

```
$ nvcc hello-world.cu

$./a.out
Hello world!

$
```

Program returns immediately after launching the kernel. To prevent program to finish before kernel is completed, we call **`cudaDeviceSynchronize().`**

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
printf("Hello world!\n");
}

int main()
{  hwkernel<<<1, 32>>>();
 cudaThreadSynchronize();
}
```

```
$ nvcc hello-world.cu

$./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
…
…
$
```

How NVCC works?

- Nvcc is a driver program
  - Compiles and links all input files
  - Requires a general-purpose C/C++ host compiler
    - Uses gcc and g++ by default on Linux platforms
  - `nvcc --version`

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

SSN

CUDA Compilation Trajectory

- Conceptually, the flow is as follows
  - Input program is preprocessed for device compilation
  - It is compiled to a CUDA binary and/or PTX (Parallel Thread Execution) intermediate code which are encoded in a fatbinary
- Input program is processed for compilation of the host code
  - CUDA-specific C++ constructs are transformed to standard C++ code
  - Synthesized host code and the embedded fatbinary are linked together to generate the executable

Function Declarations in CUDA

	Executed on	Callable from
`__device__float deviceFunc()`	Device	Device
`__global__void kernelFunc()`	Device	Host
`__host__float hostFunc()`	Host	Host

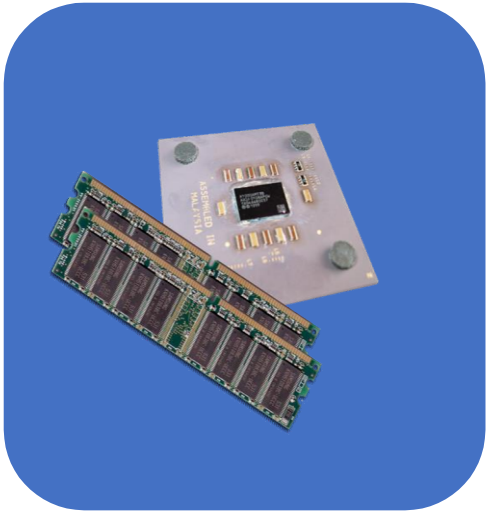
- __global__define a kernel function, must return void
- __device__functions can have return values
- __host__is default, and can be omitted
- Prepending__host__device__causes the system to compile separate host and device versions of the function

Dynamic Parallelism

- It is possible to launch kernels from other kernels
- Calling `__global__` functions from the device is referred to as dynamic parallelism
  - Requires CUDA devices of compute capability 3.5 and CUDA 5.0 or higher

Execution Model

Host
(serial eexecutions)



Serial code on host

Parallel kernel on device

Serial code on host

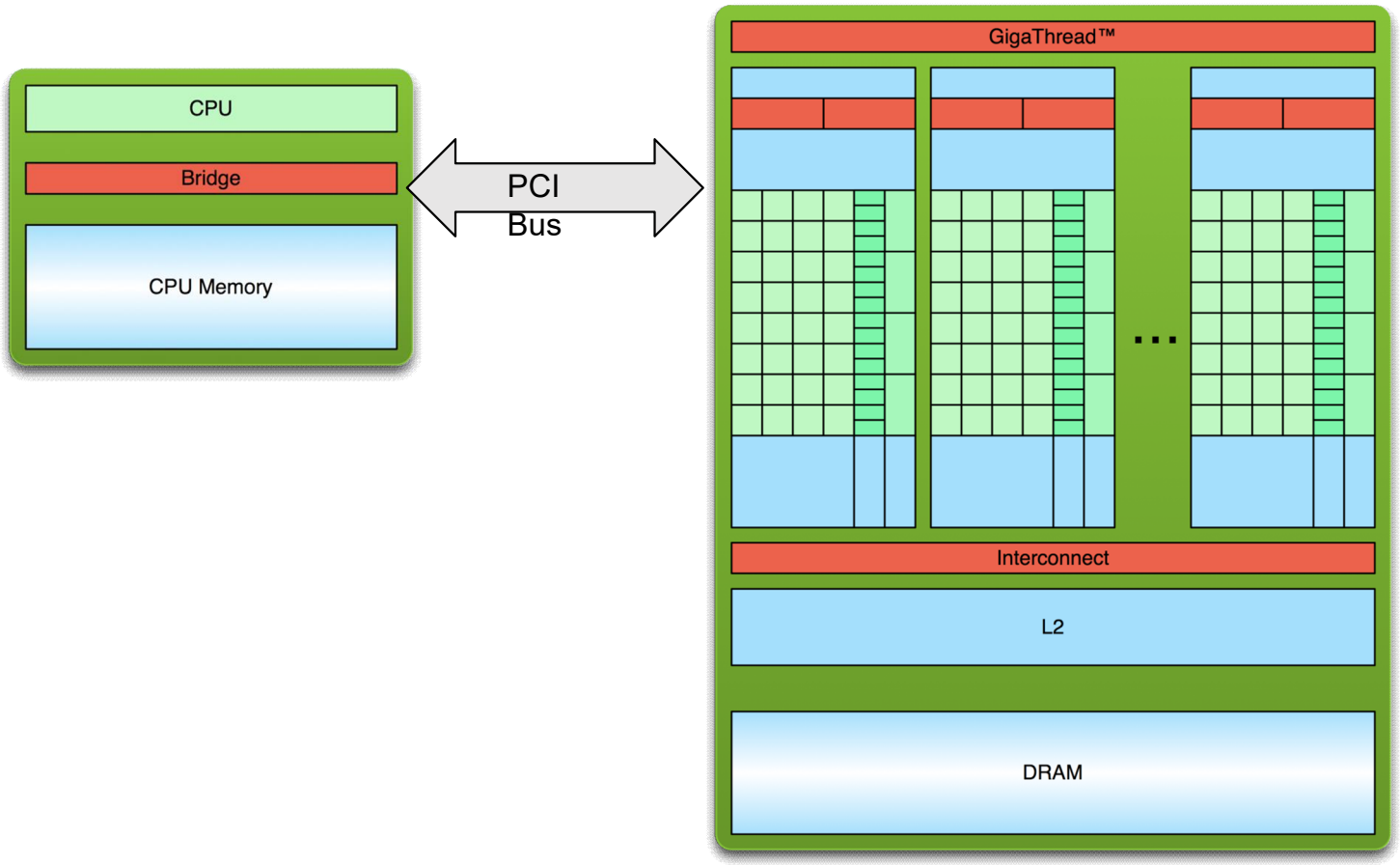
Parallel kernel on

device

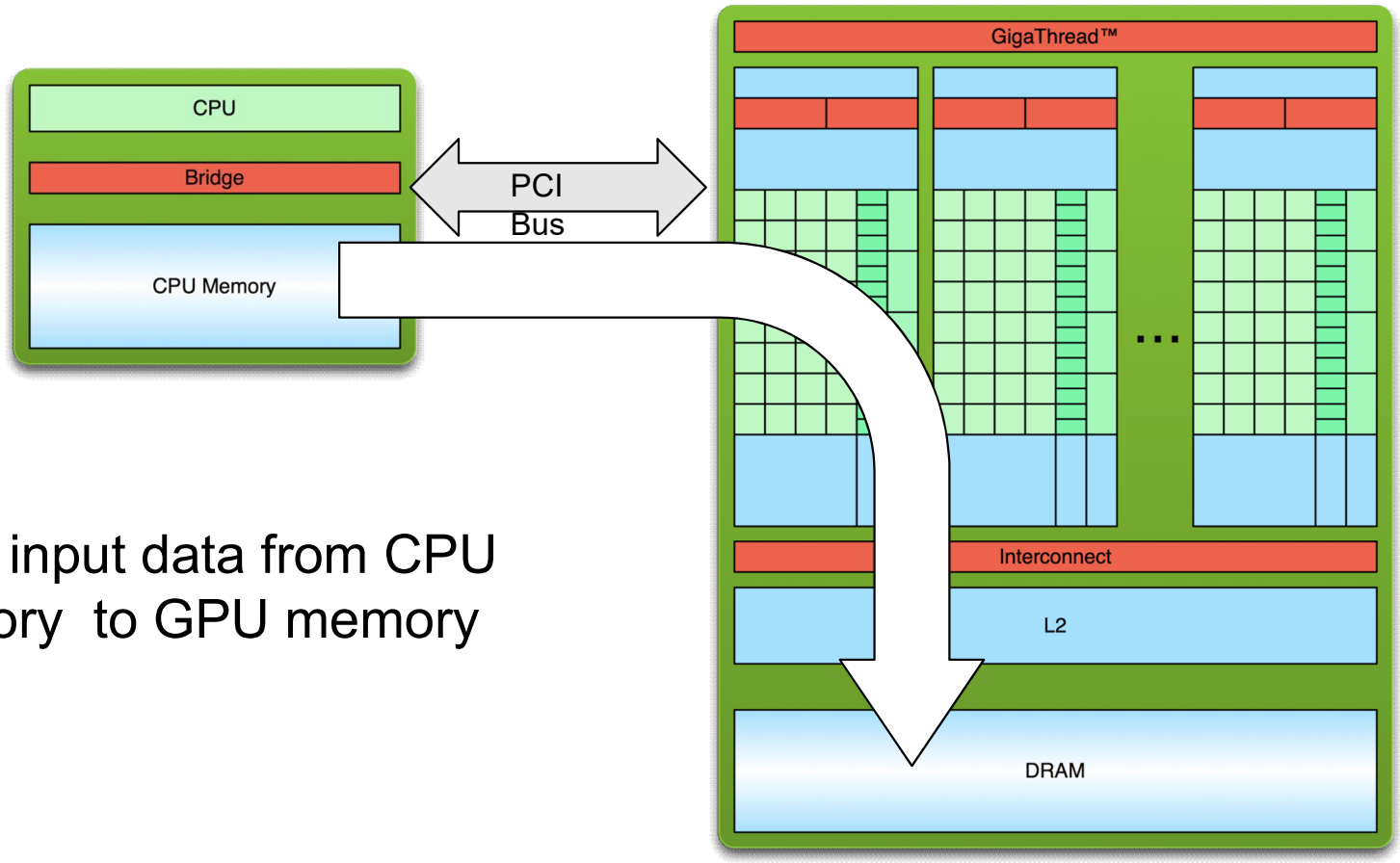
Device
(Parallel executionss)

SSN

Simple Processing Flow

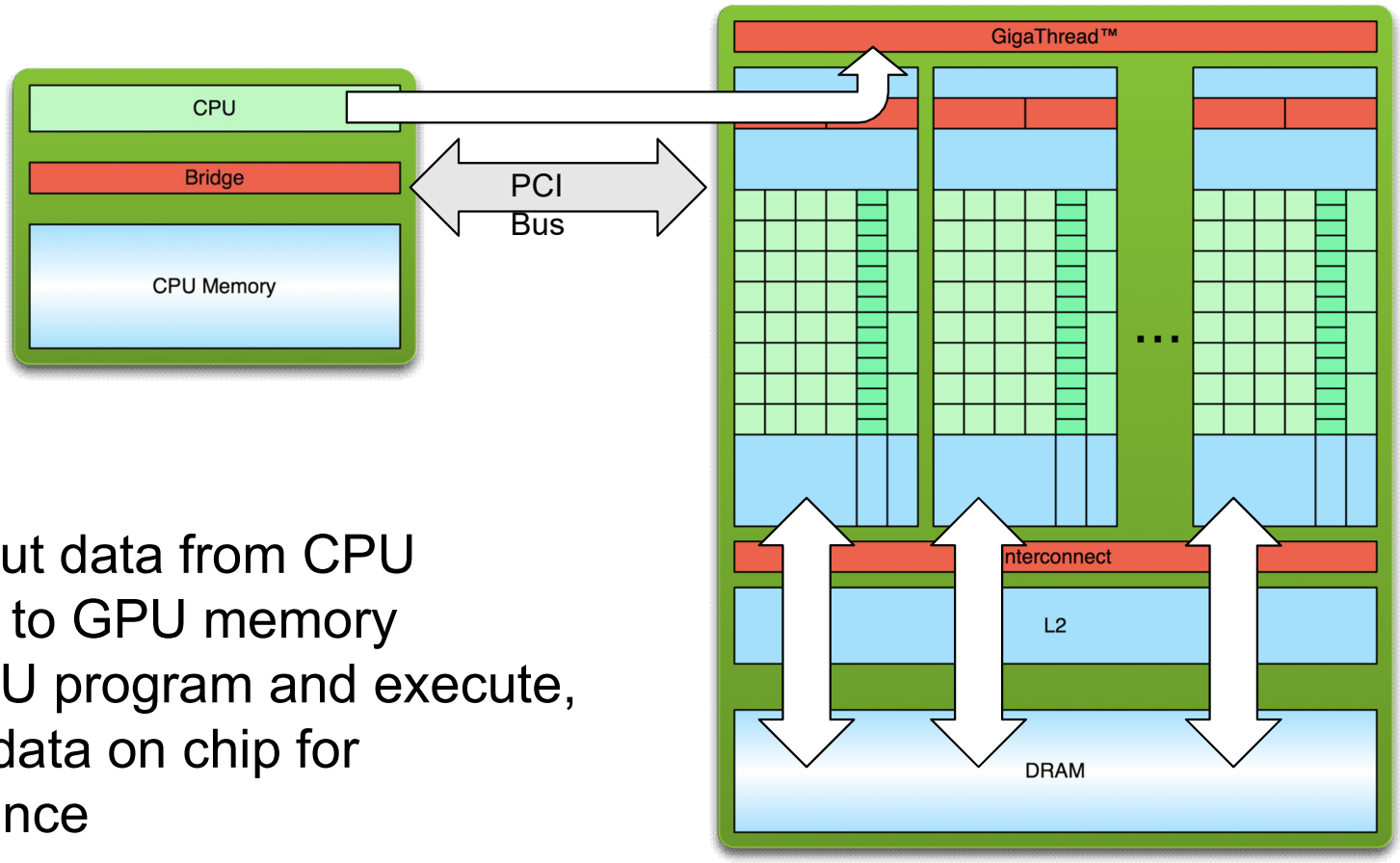


Simple Processing Flow



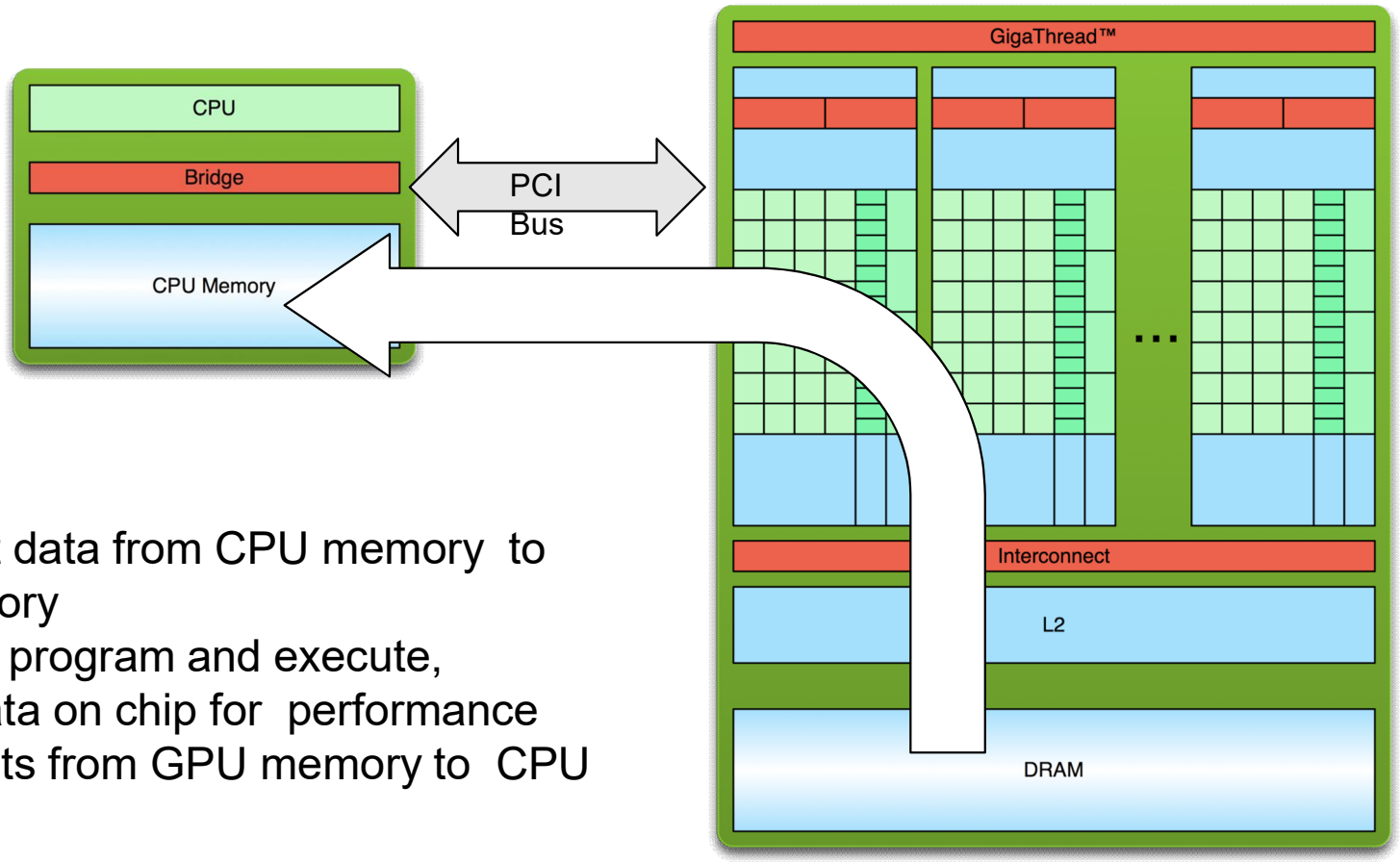
1. Copy input data from CPU memory to GPU memory

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