GPU Basics

Isaac Ye, High Performance Technical Consultant

SHARCNET, York University

CUDA Example

- Hello, CUDA!

CUDA Basics

- Introduction to CUDA
- CUDA Programming

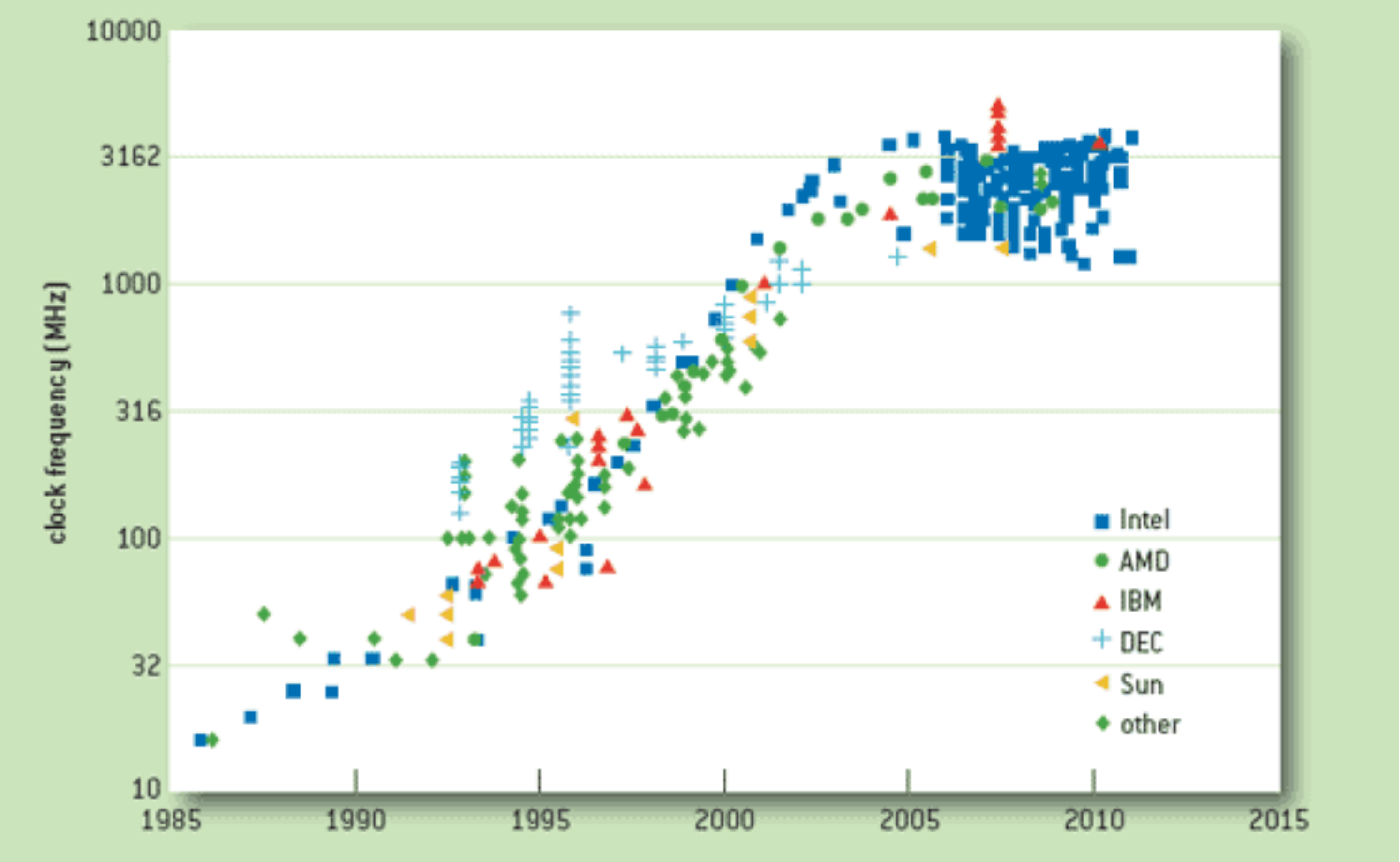
CPU vs GPU

- GPU computing
- GPGPU
- GPU systems in SHARCNET

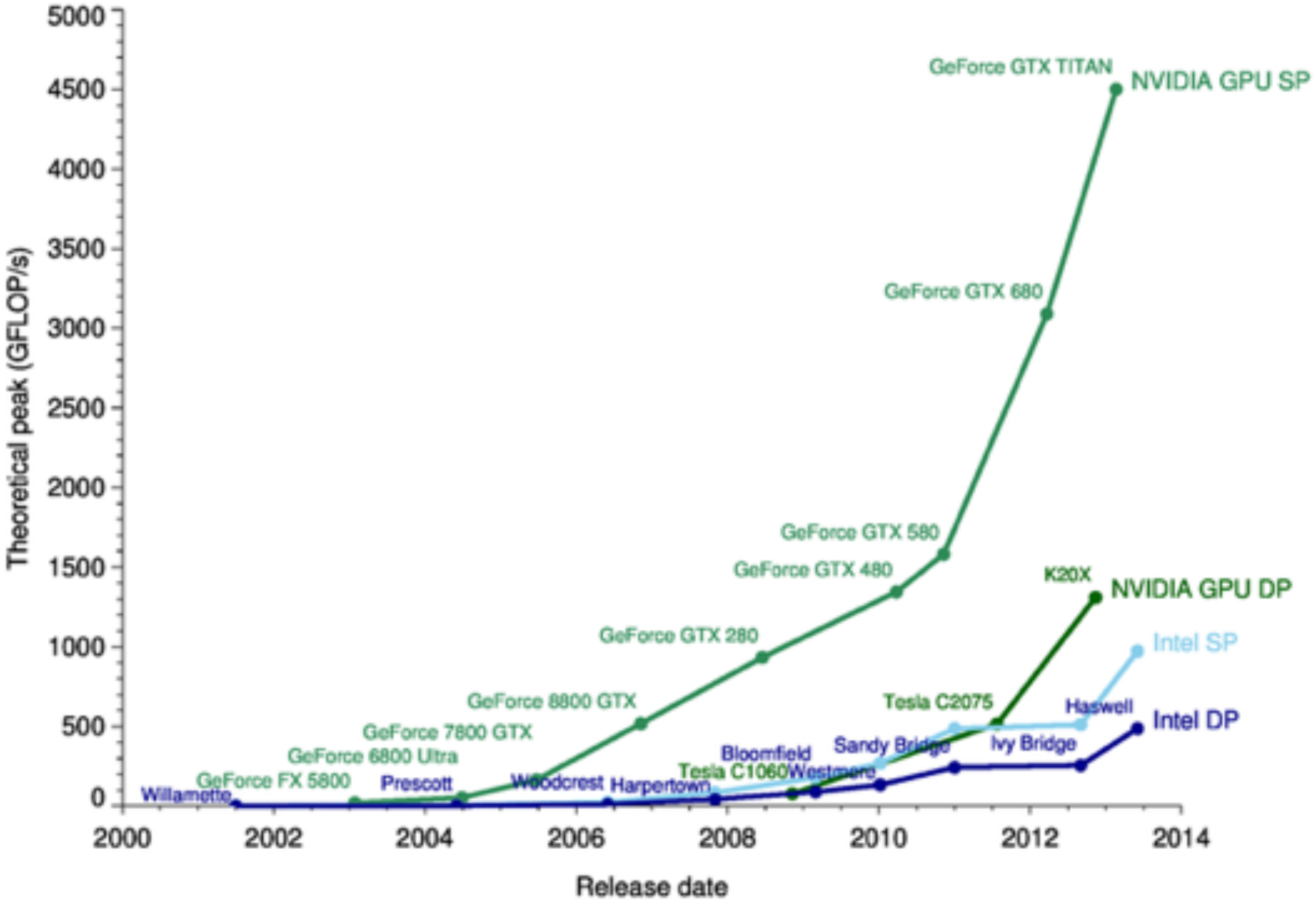
CPU vs GPU

- GPU COMPUTING
- GPGPU
- CPU VS. GPU

What happens to CPU?



GPU computing timeline



General computing APIs for GPUs

- NVIDIA offers ***CUDA*** while AMD has moved toward ***OpenCL*** (also supported by NVIDIA)

- These computing platforms bypass the graphics pipeline and expose the raw computational capabilities of the hardware. Programmer needs to know nothing about graphics programming.

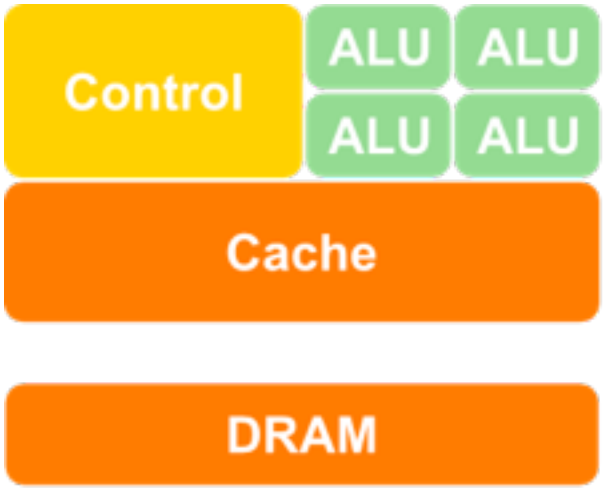
- **OpenACC** compiler directive approach is emerging as an alternative (works somewhat like OpenMP)

- More recent and less developed alternative to CUDA: ***OpenCL***
  - a vendor-agnostic computing platform
  - supports vendor-specific extensions akin to OpenGL
  - goal is to support a range of hardware architectures including GPUs, CPUs, Cell processors, Larrabee and DSPs using a standard low-level API

The appeal of GPGPU

- "Supercomputing for the masses"
  - significant computational horsepower at an attractive price point
  - readily accessible hardware
- Scalability
  - programs can execute without modification on a run-of-the-mill PC with a $150 graphics card or a dedicated multi-card supercomputer worth thousands of dollars
- Bright future – the computational capability of GPUs doubles each year
  - more thread processors, faster clocks, faster DRAM, …
  - "GPUs are getting faster, faster"

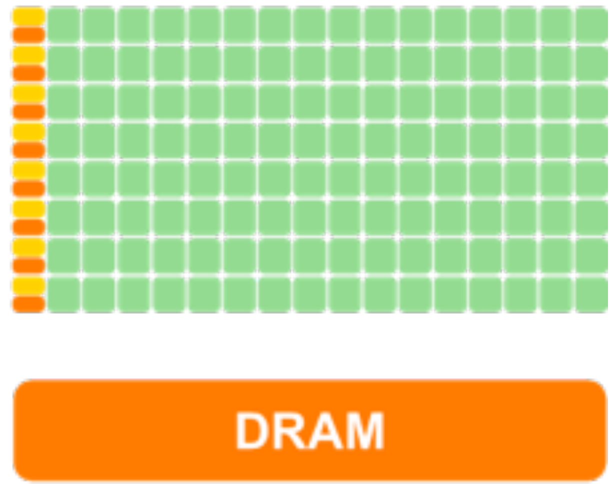
Comparing GPUs and CPUs



CPU

- Task parallelism
- Minimize latency
- Multithreaded
- Some SIMD

Latency-optimized cores
(Fast serial processing)



GPU

- excel at number crunching
- data parallelism (single task)
- maximize throughput
- super-threaded
- large-scale SIMD

Throughput-optimized cores
(Scalable parallel processing)

CUDA Basics

- INTRODUCTION TO CUDA
- CUDA PROGRAMMING

CUDA

- “Compute Unified Device Architecture”
- A platform that exposes NVIDIA GPUs as general purpose *compute devices*
- Is CUDA considered GPGPU?
  - yes and no
    - CUDA can execute on devices with no graphics output capabilities (the NVIDIA Tesla product line) – these are not “GPUs”, per se
    - however, if you are using CUDA to run some generic algorithms on your graphics card, you are indeed performing some ***G***eneral ***P***urpose computation on your ***G***raphics ***P***rocessing ***U***nit…

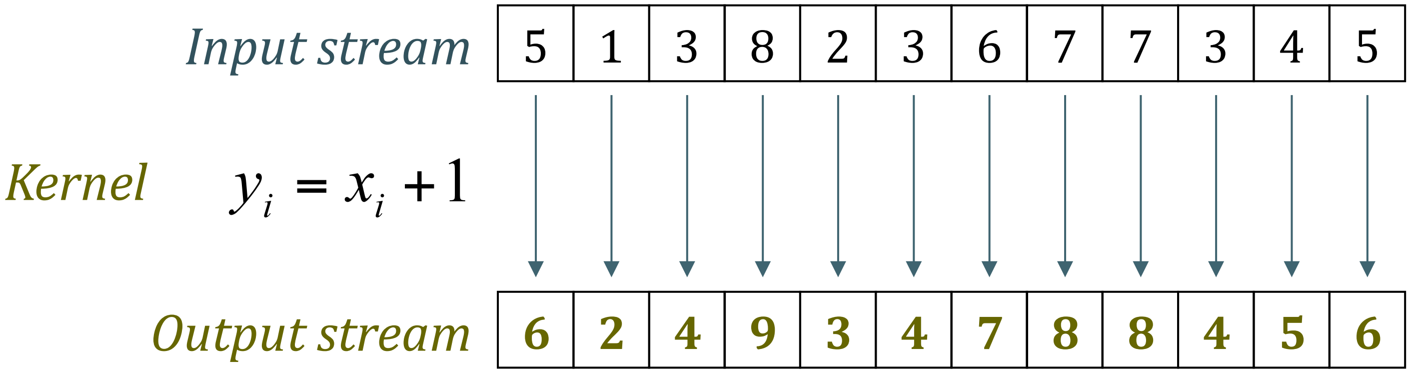
Speedup

- What kind of speedup can I expect?
  - 0x – 2000x reported
  - 10x – considered typical (vs. multi-CPU machines)
  - >= 30x considered worthwhile

- Speedup depends on
  - problem structure
    - need many identical independent calculations
    - preferably sequential memory access
  - level of intimacy with hardware
  - time investment

Stream computing

- A parallel processing model where a computational ***kernel*** is applied to a set of data (a ***stream***)
  – the kernel is applied to stream elements in parallel



- GPUs excel at this thanks to a large number of processing units and a parallel architecture

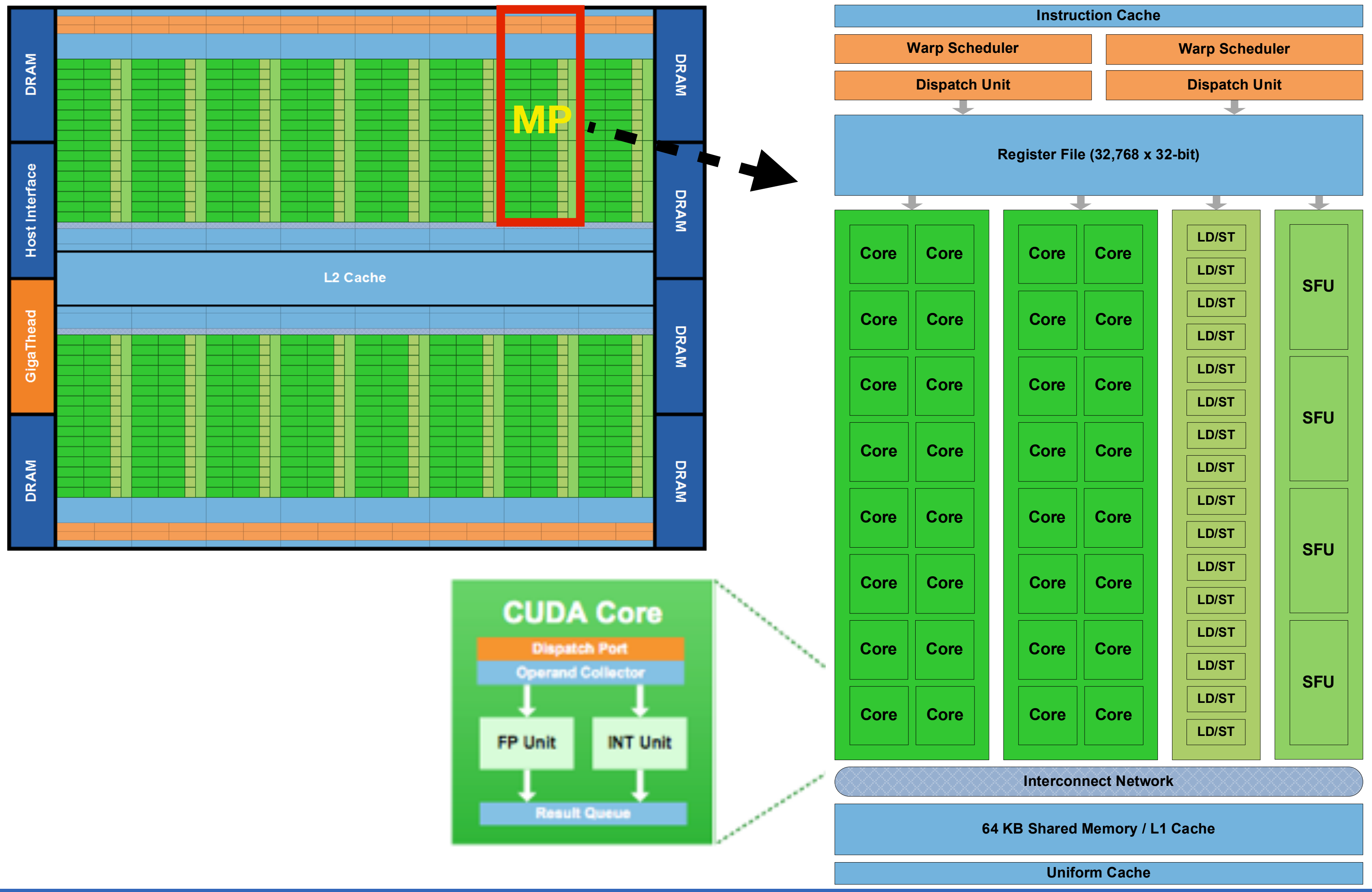
Beyond stream computing

- Current GPUs offer functionality that goes beyond mere stream computing
- Shared memory and thread synchronization primitives eliminate the need for data independence
- Gather and scatter operations allow kernels to read and write data at arbitrary locations

CUDA programming model

- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor*** capable of executing a large number of lightweight threads in parallel
- Computation on the GPU device is performed by ***kernels***, functions executed in parallel on each data element
- Both the host and the device have their own ***memory***
  - the host and device cannot directly access each other's memory, but data can be transferred using the runtime API
- The host manages all memory allocations on the device.

GPU Hardware architecture – NVIDIA Fermi



Hardware basics

- The compute device is composed of a number of ***multiprocessors***, each of which contains a number of SIMD processors
  - Tesla M2070 has 14 multiprocessors (each with 32 CUDA cores)
- A multiprocessor can execute K ***threads*** in parallel physically, where K is called the ***warp size***
  - ***thread*** = instance of kernel
  - warp size on current hardware is 32 threads
- Each multiprocessor contains a large number of 32-bit ***registers*** which are divided among the active threads

Output of device diagnostic program

```
[isaac@mon241:~] ssh monk-dev1
[isaac@mon54:~/GI_seminar/device_diagnostic] ./device_diagnostic.x
found 2 CUDA devices
   --- General Information for device 0 ---
Name:  Tesla M2070
Compute capability:  2.0
Clock rate:  1147000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  5636554752
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  14
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)

   --- General Information for device 1 ---
Name:  Tesla M2070
...
```

CUDA versions installed (SHARCNET)

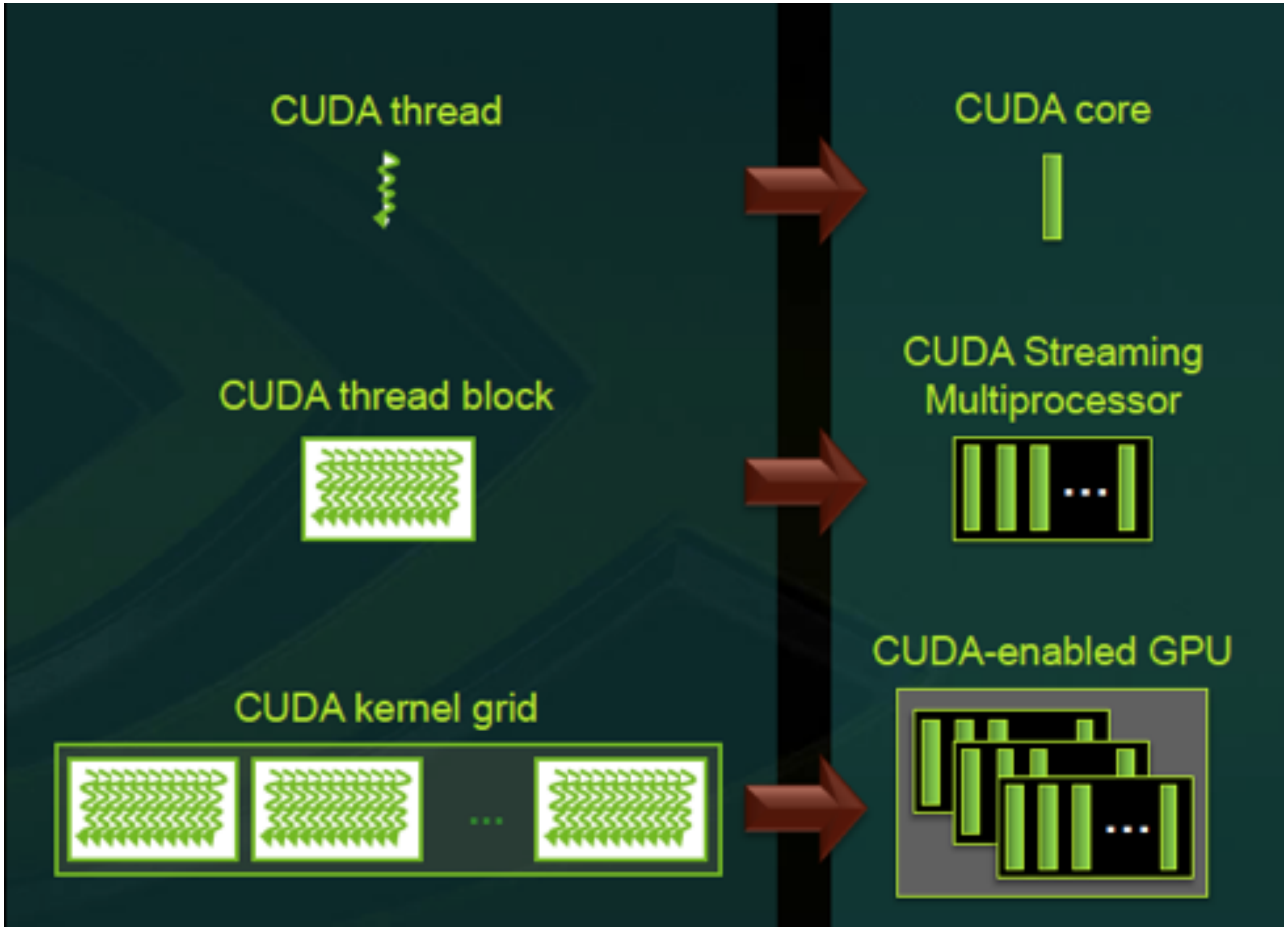
- Different versions of CUDA available - choose one via modules

– on monk latest CUDA installed in /opt/sharcnet/cuda/6.0.37/

```
[isaac@mon241:~] module list
Currently Loaded Modulefiles:
  1) torque/2.5.13          6) openmpi/intel/1.6.2
  2) moab/7.0.0             7) ldwrapper/1.1
  3) sq-tm/2.5              8) cuda/6.0.37
  4) mkl/10.3.9             9) user-environment/2.0.1
  5) intel/12.1.3
```

– sample projects in /opt/sharcnet/cuda/6.0.37/sample

Execution model

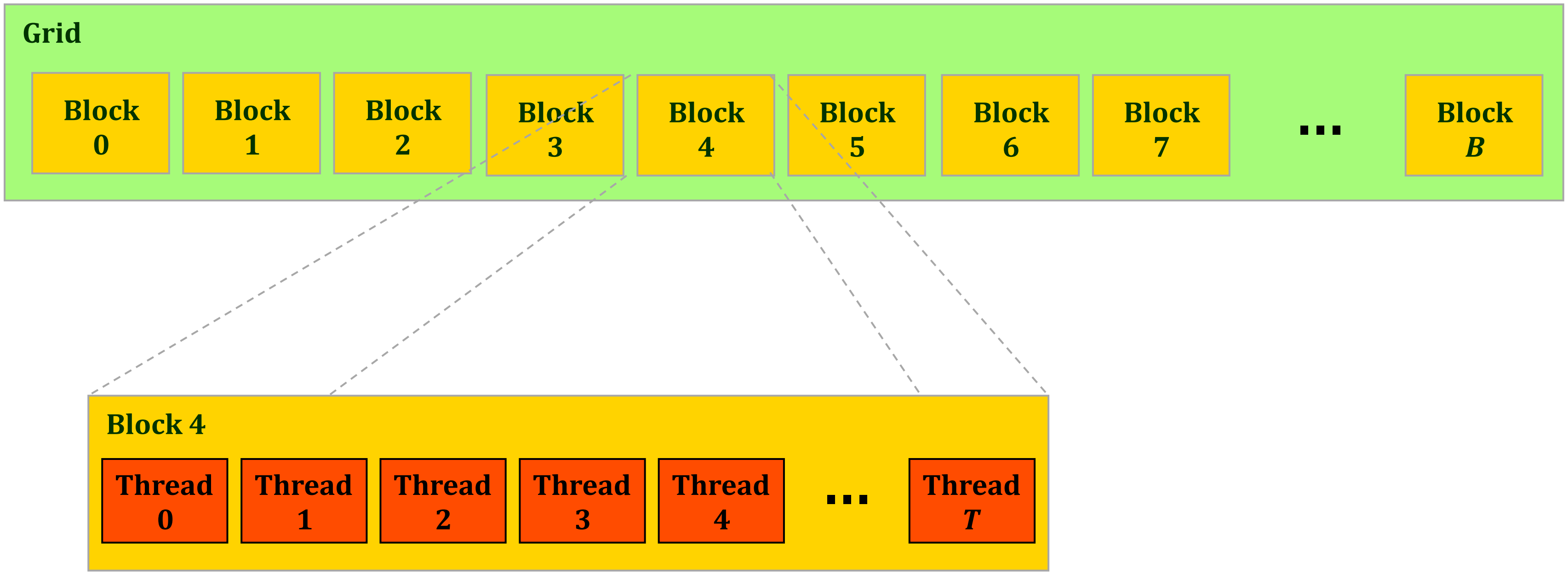


- **Each thread is executed in a core**
- **Each block is executed by one MP**
- **Each kernel is executed on one device**

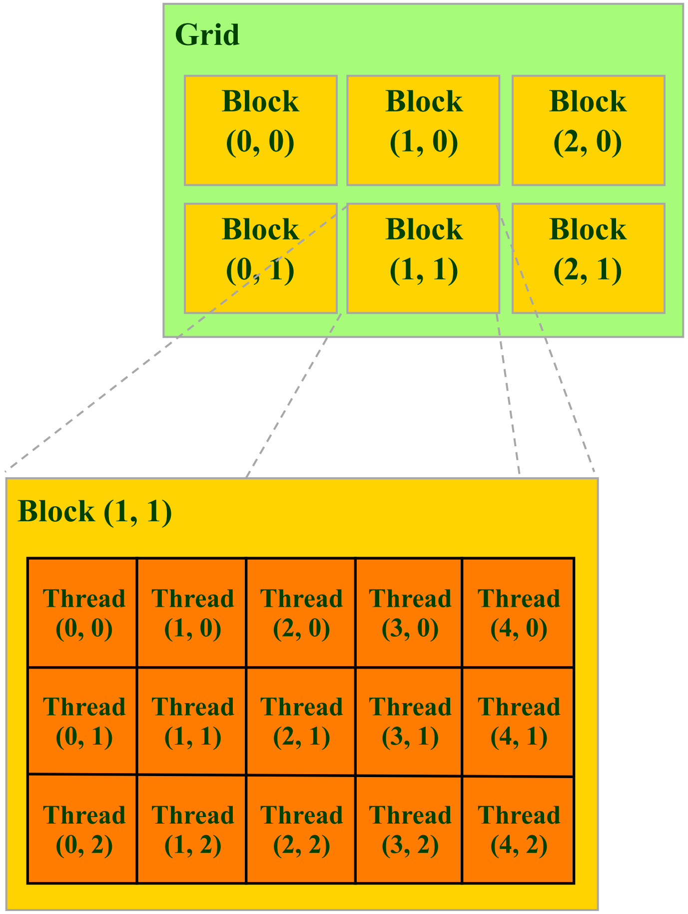
Thread batching

- To take advantage of the multiple multiprocessors, kernels are executed as a ***grid*** of ***threaded blocks***
- All threads in a thread block are executed by a single multiprocessor
- The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.)
  – this has several important implications that will be discussed later

Thread batching: 1D example



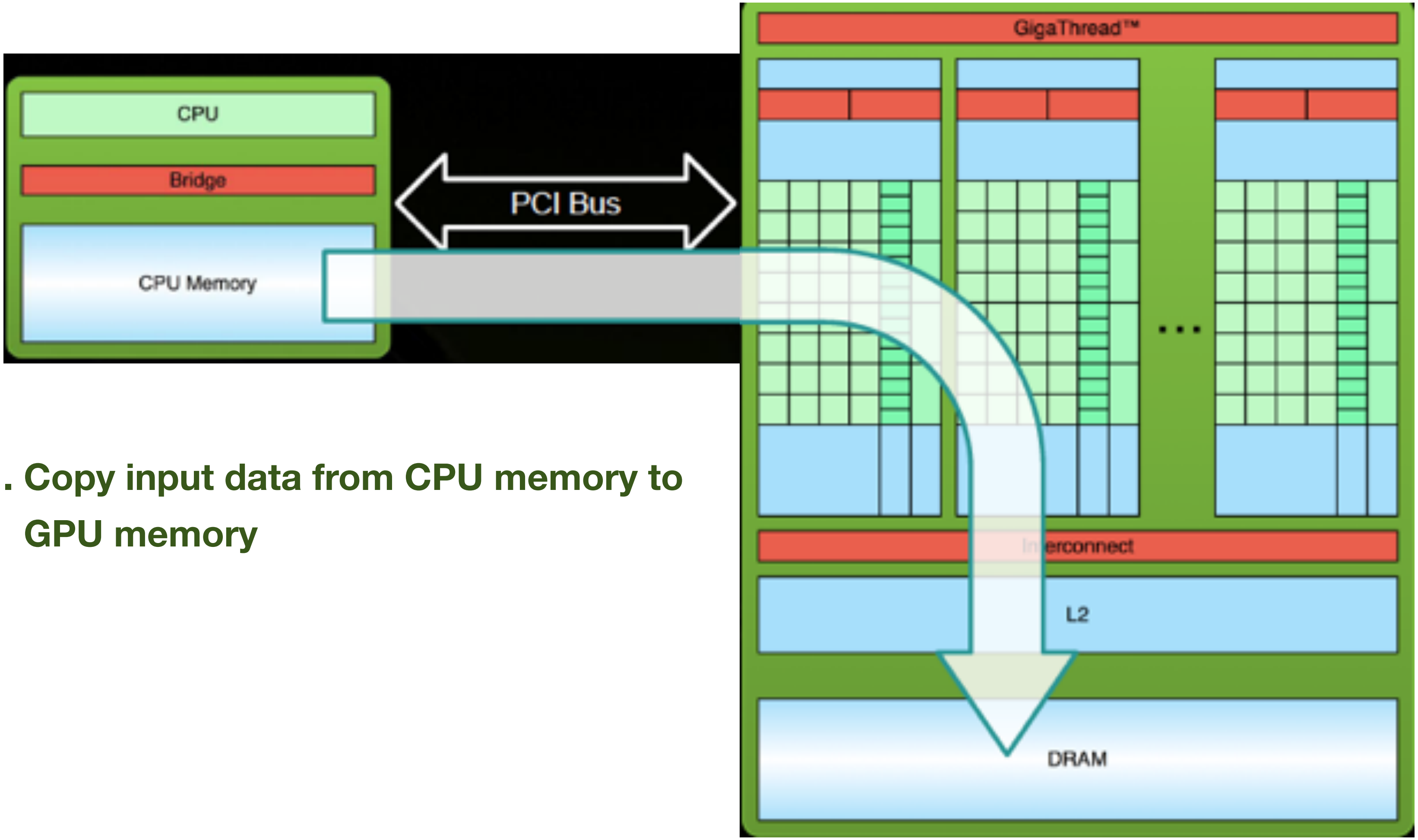
Thread batching: 2D example



CUDA Hands-on

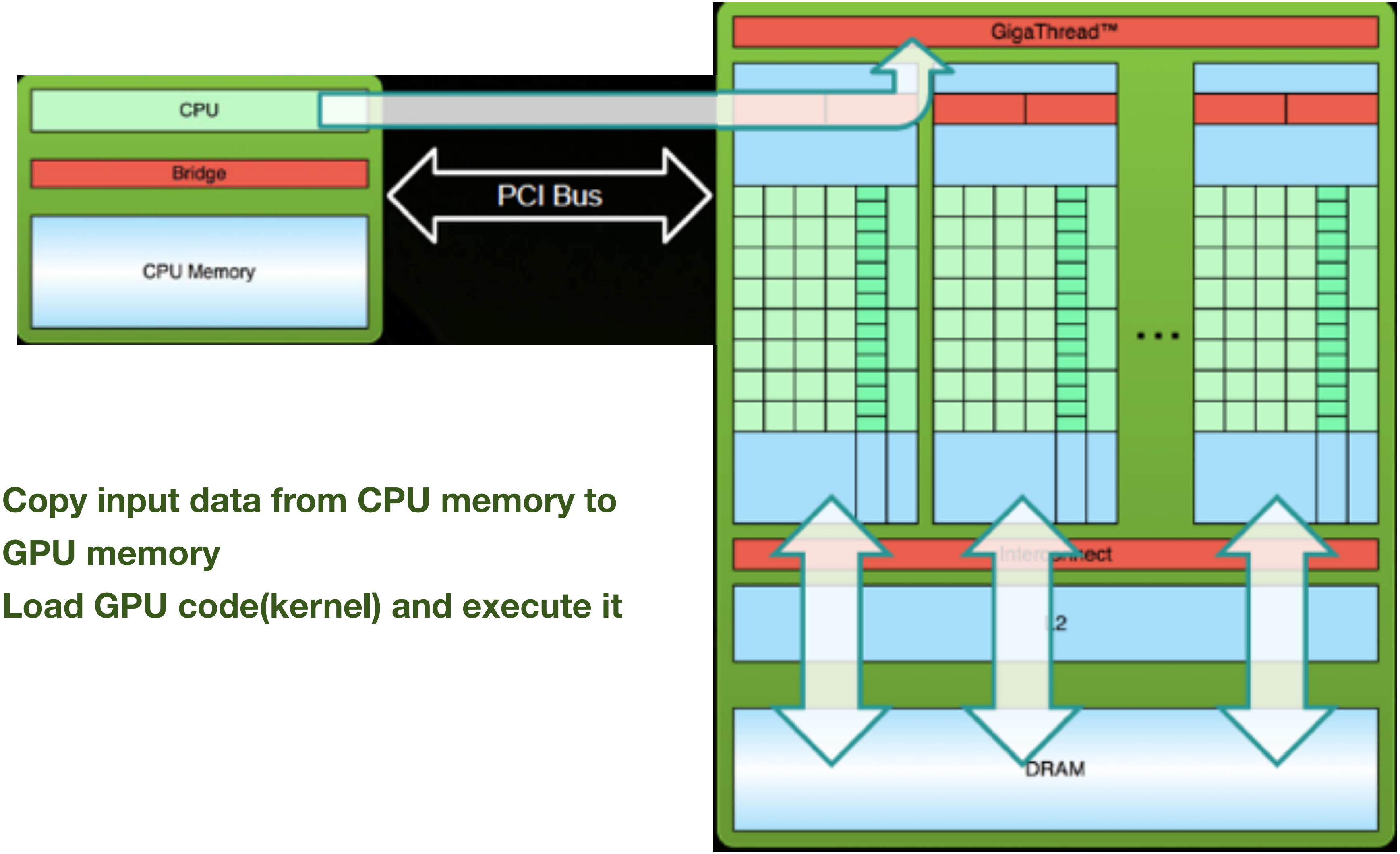
- HELLO, CUDA!
- SAXPY CUDA, SAXPY CUBLAS
- DOT PRODUCT

Simple processing flow



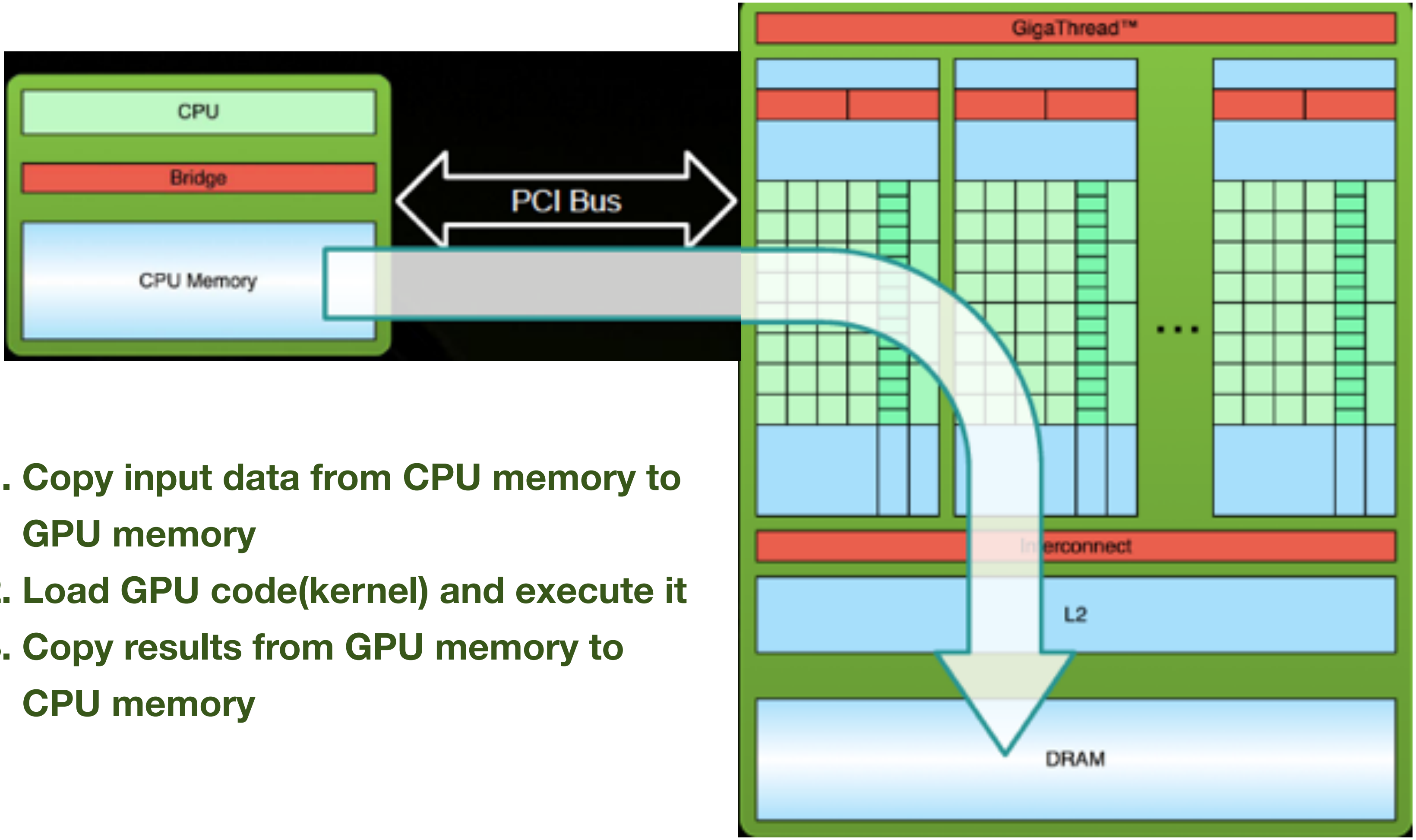
1. Copy input data from CPU memory to GPU memory

Simple processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code(kernel) and execute it

Simple processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code(kernel) and execute it
3. Copy results from GPU memory to CPU memory

Template for CUDA

```
#include <stdio.h>

main(){

   Initialize the GPU
   Memory allocation

   Memory copy

   FunctionG << N, M >> (Parameters)

   Memory copy

   }

   void __global__ functionG(parameters){

      functionA();
      functinB ();
   }

   cudafree();

   }
```

Memory control

Mem copy to GPU

Execute kernel

Mem copy from GPU

CUDA kernel (global)

CUDA cleanup

GPU Programming: Hands-on #1

HELLO, CUDA!

Example: Hello, CUDA!

- Basic example: hello_cuda.c

```
#include <stdio.h>

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

```
[isaac@mon54:~/hpcs14/hellocuda] ./a.out
Hello, CUDA!
```

Hello CUDA Kernel

- CUDA language closely follows C/C++ syntax with minimum set of extension

```
#include <stdio.h>

__global__ void cudakernel(void){
  printf("Hello, I am CUDA kernel ! Nice to meet you!\n");
}
```

- The **__global__** qualifier identifies this function as a kernel that executes on the device

Qualifiers

Functions

`__global__`	Device kernels callable from host
`__device__`	Device functions (only callable from device)
`__host__`	Host functions (only callable from host)

Data

`__shared__`	Memory shared by a block of threads executing on a multiprocessor.
`__constant__`	Special memory for constants (cached)

CUDA data types

- C primitives:
  - `char`, `int`, `float`, `double`, ...
- Short vectors:
  - `int2`, `int3`, `int4`, `uchar2`, `uchar4`, `float2`, `float3`, `float4`, ...
- Special type used to represent dimensions
  - `dim3`
- Support for user-defined structures, e.g.:

```
struct particle
{
    float3 position, velocity, acceleration;
    float mass;
};
```

Library functions available to kernels

- Math library functions:
  - sin, cos, tan, sqrt, pow, log, ...
  - sinf, cosf, tanf, sqrtf, powf, logf, ...
- ISA intrinsics
  - __sinf, __cosf, __tanf, __powf, __logf, ...
  - __mul24, __umul24, ...
- Intrinsic versions of math functions are faster but less precise

Hello CUDA code

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

```
int main(void){

   printf("Hello, Cuda! \n");

   cudakernel<<<1,1>>>();
   cudaDeviceSynchronize();

   printf("Nice to meet you too! Bye, CUDA\n");

   return(0);
}

__global__ void cudakernel(void){
  printf("Hello, I am CUDA kernel ! Nice to meet you!\n");
}
```

HOW TO COMPILE AND RUN

SHARCNET GPU systems

- Always check our software page for latest info! See also:
  https://www.sharcnet.ca/help/index.php/GPU_Accelerated_Computing

- angel.sharcnet.ca

  11 NVIDIA Tesla S1070 GPU servers

  each with 4 GPUs + 16GB of global memory

  each GPU server connected to **two** compute nodes (2 4-core Xeon CPUs + 8GB RAM each)

  1 GPU per quad-core CPU; 1:1 memory ratio between GPUs/CPUs

- visualization workstations

  Some old and don't support CUDA, but some have up to date cards, check list at:

  https://www.sharcnet.ca/my/systems/index

“monk” cluster

- 54 nodes, InfiniBand interconnect, 80 Tb storage
- Node:
  8 x CPU cores (Intel Xeon 2.26 GHz)
  48 GB memory
  2 x M2070 GPU cards
- Nvidia Tesla M2070 GPU
  “Fermi” architecture
  ECC memory protection
  L1 and L2 caches
  2.0 Compute Capability
  448 CUDA cores
  515 Gigaflops (DP)

Language and compiler

- CUDA provides a set of extensions to the C programming language
  – new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in .cu files
  – host and device code and coexist in the same file
  – storage qualifiers determine type of code
- Compiled to object files using nvcc compiler
  – object files contain executable host and device code
- Can be linked with object files generated by other C/C++ compilers

Compiling

- nvcc -arch=sm_20 -O2 program.cu -o program.x
- -arch=sm_20 means code is targeted at Compute Capability 2.0 architecture (what monk has)
- -O2 optimizes the CPU portion of the program (needs to be off for debugging/profiling)
- There are no flags to optimize CUDA code
- Various fine tuning switches possible
- SHARCNET has a CUDA environment module preloaded. See what it does by executing: module show cuda
- add -lcublas to link with CUBLAS libraries

Hello CUDA code with built-in variable

- Basic example: hello_cuda.cu

```
#include <stdio.h>

__global__ void cudakernel(void){
  printf("Hello, I am CUDA block %d! Nice to meet you!\n", blockIdx);
}

int main(void){

   printf("Hello, Cuda! \n");

   cudakernel<<<16,1>>>();
   cudaDeviceSynchronize();

   printf("Nice to meet you too! Bye, CUDA\n");

   return(0);
}
```

cudakernel<<<16,1>>>();

Block 0	`Hello, I am CUDA block 0! Nice to meet you!`
Block 1	`Hello, I am CUDA block 1! Nice to meet you!`
Block 2	`Hello, I am CUDA block 2! Nice to meet you!`
:	
:	
Block 15	`Hello, I am CUDA block 15! Nice to meet you!`

Hello CUDA result with BlockIdx value

```
[isaac@mon54:~/GI_seminar/hellocuda] ./a.out
Hello, Cuda!
Hello, I am CUDA block 4 ! Nice to meet you!
Hello, I am CUDA block 11 ! Nice to meet you!
Hello, I am CUDA block 15 ! Nice to meet you!
Hello, I am CUDA block 5 ! Nice to meet you!
Hello, I am CUDA block 7 ! Nice to meet you!
Hello, I am CUDA block 14 ! Nice to meet you!
Hello, I am CUDA block 3 ! Nice to meet you!
Hello, I am CUDA block 9 ! Nice to meet you!
Hello, I am CUDA block 13 ! Nice to meet you!
Hello, I am CUDA block 6 ! Nice to meet you!
Hello, I am CUDA block 2 ! Nice to meet you!
Hello, I am CUDA block 12 ! Nice to meet you!
Hello, I am CUDA block 8 ! Nice to meet you!
Hello, I am CUDA block 0 ! Nice to meet you!
Hello, I am CUDA block 1 ! Nice to meet you!
Hello, I am CUDA block 10 ! Nice to meet you!
Nice to meet you too! Bye, CUDA
```

C Language extensions

- Basic example: hello_cuda_thread.cu

```
#include <stdio.h>

__global__ void cudakernel(void){
  printf("Hello, I am CUDA thread %d! Nice to meet you!\n", threadIdx.x);
}

int main(void){

...
   cudakernel<<<1,16>>>();
   cudaDeviceSynchronize();

...
}
```

```
cudakernel<<<1,16>>>();
```

Block 0

Thread 0 Hello, I am CUDA thread 0! Nice to meet you!

Thread 1 Hello, I am CUDA thread 1! Nice to meet you!

Thread 2 Hello, I am CUDA thread 2! Nice to meet you!

:

:

Thread 15 Hello, I am CUDA thread 15! Nice to meet you!

C Language extensions

```
[isaac@mon54:~/GI_seminar/hellocuda] ./a.out
Hello, Cuda!
Hello, I am CUDA thread 0! Nice to meet you!
Hello, I am CUDA thread 1! Nice to meet you!
Hello, I am CUDA thread 2! Nice to meet you!
Hello, I am CUDA thread 3! Nice to meet you!
Hello, I am CUDA thread 4! Nice to meet you!
Hello, I am CUDA thread 5! Nice to meet you!
Hello, I am CUDA thread 6! Nice to meet you!
Hello, I am CUDA thread 7! Nice to meet you!
Hello, I am CUDA thread 8! Nice to meet you!
Hello, I am CUDA thread 9! Nice to meet you!
Hello, I am CUDA thread 10! Nice to meet you!
Hello, I am CUDA thread 11! Nice to meet you!
Hello, I am CUDA thread 12! Nice to meet you!
Hello, I am CUDA thread 13! Nice to meet you!
Hello, I am CUDA thread 14! Nice to meet you!
Hello, I am CUDA thread 15! Nice to meet you!
Nice to meet you too! Bye, CUDA
```