

Programming General Purpose Graphic Processing Unit using CUDA (basics)

Erik Saule

esaule@uncc.edu

Parallel Computing

11/18/2018

Learning Outcomes

At the end of this lecture, you will be able to

- Differentiate cases where GPU are useful and when they are not.
- Explain the processing flow of accelerated applications.
- Decompose computation in blocks and threads.
- Write trivial kernels.

The assignment will ask you to

- Transfer data back and forth from a GPU.
- Implement a kernel.
- Relate data transfer overhead to computational volume.

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation
- 4 Memory management
- 5 Writing kernels : vector addition
- 6 Error Management
- 7 Resources

Why is GPGPU so popular?

Availability

Most discrete graphic cards produced by NVIDIA, AMD, or Intel can be programmed to do general purpose computation. Most machines have them.

Why is GPGPU so popular?

Availability

Most discrete graphic cards produced by NVIDIA, AMD, or Intel can be programmed to do general purpose computation. Most machines have them.

A good bang for buck ratio (numbers from 2017)

A modern CPU: Intel Xeon Processor E5-4610 v2 (8 cores, 16M Cache, 2.30 GHz, AVX): 294Gflop/s, 51.2 GB/s, \$1220.

A modern GPU: NVIDIA K20, 3.52Tflop/s, 208GB/s, \$3500

Roughly, 10 times the flops, 4 times the bandwidth, 3 times the price.

Why is GPGPU so popular?

Availability

Most discrete graphic cards produced by NVIDIA, AMD, or Intel can be programmed to do general purpose computation. Most machines have them.

A good bang for buck ratio (numbers from 2017)

A modern CPU: Intel Xeon Processor E5-4610 v2 (8 cores, 16M Cache, 2.30 GHz, AVX): 294Gflop/s, 51.2 GB/s, \$1220.

A modern GPU: NVIDIA K20, 3.52Tflop/s, 208GB/s, \$3500

Roughly, 10 times the flops, 4 times the bandwidth, 3 times the price.

So what is the catch?

The catch compared to CPUs

- **Less memory:** A GPU typically has only a limited amount of memory. A Tesla K40 has 12GB. You can build a 2TB machine out of CPUs.
- **Alien to program:** Code can not be just recompiled. It needs to be ported.
- **Complex architecture:** Complex architectural details must be accounted for to get performance.
- **Computation must be massively parallel:** Lots of threads (10,000+) on many (1000s) dumb cores.
- Overall, that is what is referred to as "low latency" (CPUs) and "High throughput" (GPUs)

Options to program them

- **Libraries.** No need to worry about anything, the library does everything for you.
- **OpenCL.** Some generic programming model for “heterogeneous systems” that maps to various architectures. Originally targeted for GPUs, but also works on multicore processors
- **Pragmas.** OpenACC is similar to OpenMP for accelerators. Support in OpenMP is coming.
- **CUDA.** “Native” programming model for NVIDIA GPUs.

Many of the following slides are from Sarah Tariq's slides. "An Introduction to GPU Computing and CUDA Architecture".

Almost everything you need to know about GPUs are in "NVIDIA CUDA C Programming Guide" published by NVIDIA.

Outline

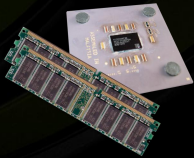
- 1 Why GPGPU?
- 2 Principle**
- 3 Compilation
- 4 Memory management
- 5 Writing kernels : vector addition
- 6 Error Management
- 7 Resources

Heterogeneous Computing



- Terminology:

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



Host



Device

© NVIDIA Corporation 2011

Heterogeneous Computing



```

#include <cuda_runtime.h>
#include <device_launch_parameters.h>

using namespace std;

#define N 1024
#define RAD2US 2
#define BLOCK_SIZE 16

__global__ void serial_cpu(float *a, float *b) {
    int i;
    for (i = 0; i < N; i++) {
        a[i] = b[i] * RAD2US;
    }
}

// Host code
int main() {
    float *a, *b;
    cudaMalloc((void**)&a, N * sizeof(float));
    cudaMalloc((void**)&b, N * sizeof(float));
    // Launch serial CPU kernel
    serial_cpu(a, b);
    // Copy result back to host
    cudaMemcpy(a, b, N * sizeof(float), cudaMemcpyDeviceToHost);
    return 0;
}
    
```

parallel fn

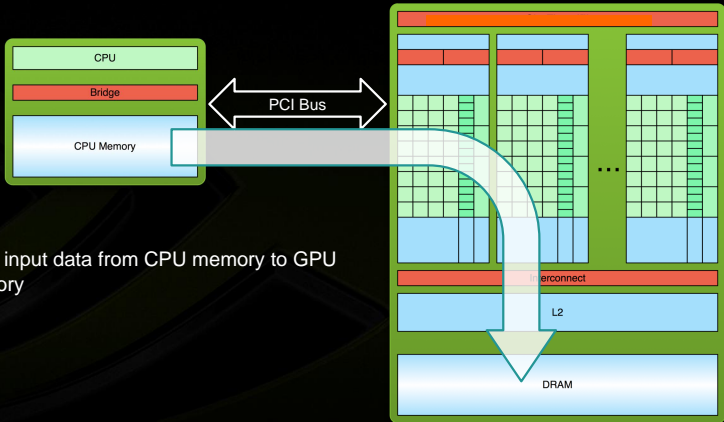
serial code

parallel code

serial code



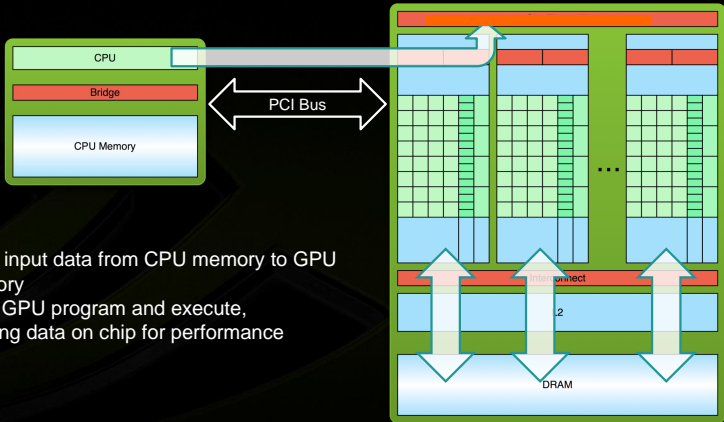
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

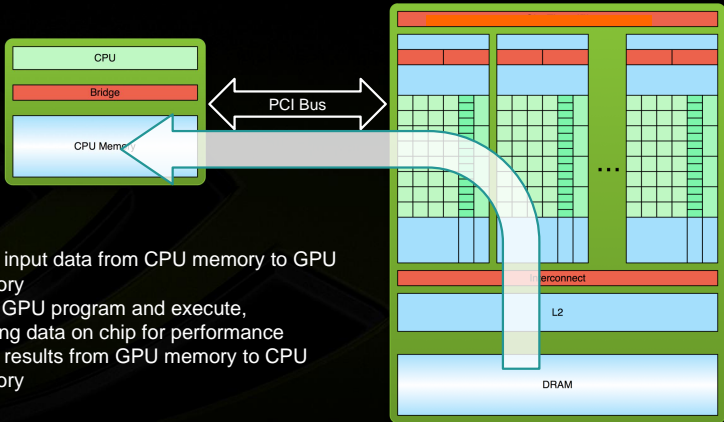
© NVIDIA Corporation 2011

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

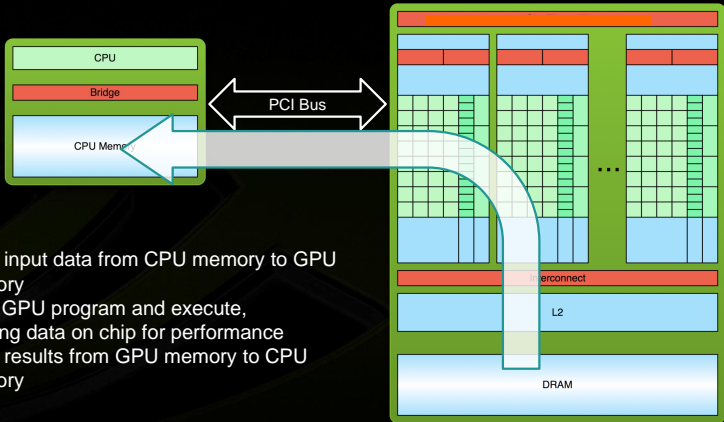


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Outline

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation**
- 4 Memory management
- 5 Writing kernels : vector addition
- 6 Error Management
- 7 Resources

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”
 - We’ll return to the parameters (1,1) in a moment
- 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;  
}
```

Output:

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

- `mykernel()` does nothing, somewhat anticlimactic!

Outline

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation
- 4 Memory management**
- 5 Writing kernels : vector addition
- 6 Error Management
- 7 Resources

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;
}
```

Other kind of memory management techniques: later

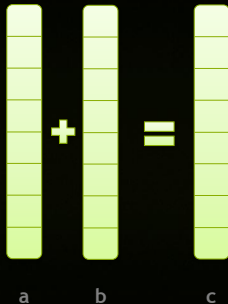
Outline

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation
- 4 Memory management
- 5 Writing kernels : vector addition**
- 6 Error Management
- 7 Resources

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

So `a`, `b`, `c` must be on the device.

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

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];`

The word is “can” not “will”. There is no certainty: maybe it will, maybe it won't.

CUDA Threads



- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

Using blocks:

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}  
add<<<N,1>>>>(d_a, d_b, d_c);
```

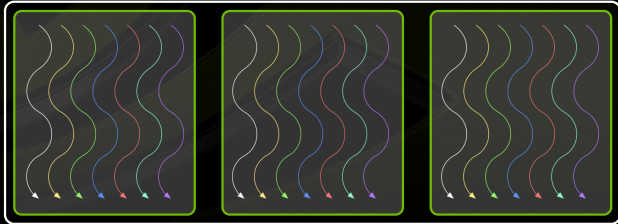
Using threads:

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}  
add<<<1,N>>>>(d_a, d_b, d_c);
```

© NVIDIA Corporation 2011

Combining Blocks and Threads

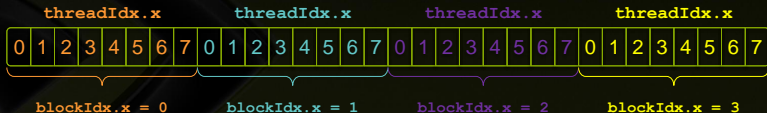
- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both *blocks* and *threads*



© NVIDIA Corporation 2011

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

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;
}
```

© NVIDIA Corporation 2011

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);
```

Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
 - Communicate
 - Synchronize
- To look closer, we need a new example...

We'll talk about why use thread later. For now, just trust that's useful to have both.

Outline

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation
- 4 Memory management
- 5 Writing kernels : vector addition
- 6 Error Management**
- 7 Resources

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
 - Error in the API call itself
 - OR
 - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

In other words, all errors that happen on the GPU are usually **silent**.
Something bad happened and you are not notified through segfault or anything.

Outline

- 1 Why GPGPU?
- 2 Principle
- 3 Compilation
- 4 Memory management
- 5 Writing kernels : vector addition
- 6 Error Management
- 7 Resources**

- OpenACC: <http://www.openacc-standard.org/>
- OpenCL: <https://www.khronos.org/opencvl/>
- CUDA homepage: <http://docs.nvidia.com/cuda/index.html>
- CUDA programming guide: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- CUDA runtime API: <http://docs.nvidia.com/cuda/cuda-runtime-api/>
- CUDA best practice: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>
- Sarah Tariq's slides:
http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC_Express_Sarah_Tariq_June2011.pdf
- Jason Sanders and Edward Kandort. CUDA by example. Addison Wesley. 2011.

I found a very helpful tool in the cuda toolkit called cuda Memcheck.
compile with -lineinfo to get linenumbers in the output.

To run: `cuda-memcheck ./binary options`

It's that simple. It will detect memory leaks and is similar to valgrind in output.

A link to the documentation:

<http://docs.nvidia.com/cuda/cuda-memcheck/#axzz4fZdek1Vh>