Introduction to CUDA Programming

Jian Tao
jtao@tamu.edu
Fall 2017 HPRC Short Course
09/28/2017
Relevant Short Course and Workshop

Intermediate CUDA Programming
https://hprc.tamu.edu/training/intermediate_cuda.html
2:30pm - 3:55pm, Tuesday, October 31 2017

Bring-Your-Own-Code Workshop
https://coehpc.engr.tamu.edu/byoc/
Offered regularly
GPU as an Accelerator
CPU

GPU Accelerator
NVIDIA Tesla V100 with 21.1 Billion Transistors
Why Computing Perf/Watt Matters?

Traditional CPUs are not economically feasible

- 2.3 PFlops
- 7000 homes
- 7.0 Megawatts

GPU-accelerated computing started a new era

- CPU: Optimized for Serial Tasks
- GPU Accelerator: Optimized for Many Parallel Tasks

7000 homes

7.0 Megawatts
Add GPUs: Accelerate Science Applications

Application Code

GPU

CPU

Use GPU to Parallelize Compute-Intensive Functions

Rest of Sequential CPU Code
HPC - Distributed Heterogeneous System

Programming Models: MPI + (CUDA, OpenCL, OpenMP, OpenACC, etc.)
Amdahl's Law

\[ S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}} \]
CUDA Parallel Computing Platform


**Programming Approaches**
- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Apps
- Programming Languages
  - Maximum Flexibility

**Development Environment**
- Nsight IDE
  - Linux, Mac and Windows
  - GPU Debugging and Profiling
- CUDA-GDB debugger
- NVIDIA Visual Profiler

**Open Compiler Tool Chain**
- Enables compiling new languages to CUDA platform, and
  - CUDA languages to other architectures

**Hardware Capabilities**
- SMX
- Dynamic Parallelism
- HyperQ
- GPUDirect
3 Ways to Accelerate Applications

Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Flexibility
3 Ways to Accelerate Applications

- **Libraries**
  - “Drop-in” Acceleration

- **OpenACC Directives**
  - Easily Accelerate Applications

- **Programming Languages**
  - Maximum Flexibility
Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming.

- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes.

- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications.

- **Performance:** NVIDIA libraries are tuned by experts.
Some GPU-accelerated Libraries

CUDA-accelerated Application with Libraries

• **Step 1:** Substitute library calls with equivalent CUDA library calls
  
  \[
  \text{saxpy ( \ldots )} \quad \Rightarrow \quad \text{cublasSaxpy ( \ldots )}
  \]

• **Step 2:** Manage data locality
  
  - with CUDA: \text{cudaMalloc()}, \text{cudaMemcpy()}, etc.
  - with CUBLAS: \text{cublasAlloc()}, \text{cublasSetVector()}, etc.

• **Step 3:** Rebuild and link the CUDA-accelerated library
  
  \$\text{nvcc myobj.o -l cublas}
Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone.

3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Flexibility
OpenACC Directives

CPU

GPU

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Program myscience
... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
... End Program myscience
OpenACC

The Standard for GPU Directives

• **Easy:** Directives are the easy path to accelerate compute intensive applications

• **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

• **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU
Directives: Easy & Powerful

- Real-Time Object Detection
  Global Manufacturer of Navigation Systems
  - 5x in 40 Hours

- Valuation of Stock Portfolios using Monte Carlo
  Global Technology Consulting Company
  - 2x in 4 Hours

- Interaction of Solvents and Biomolecules
  University of Texas at San Antonio
  - 5x in 8 Hours
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility
## GPU Programming Languages

<table>
<thead>
<tr>
<th>Category</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical analytics</td>
<td>MATLAB, Mathematica, LabVIEW</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenACC, CUDA Fortran</td>
</tr>
<tr>
<td>C</td>
<td>OpenACC, CUDA C, OpenCL</td>
</tr>
<tr>
<td>C++</td>
<td>Thrust, CUDA C++, OpenCL</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA, PyOpenCL, Copperhead</td>
</tr>
<tr>
<td>Java / F#</td>
<td>JCuda / Alea GPU</td>
</tr>
</tbody>
</table>
Rapid Parallel C++ Development

• Resembles C++ STL
• High-level interface
  • Enhances developer productivity
  • Enables performance portability between GPUs and multicore CPUs
• Flexible
  • CUDA, OpenMP, and TBB backends
  • Extensible and customizable
  • Integrates with existing software
• Open source

```c++
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

[https://thrust.github.io/](https://thrust.github.io/)
Learn More

These languages are supported on all CUDA-capable GPUs. You might already have a CUDA-capable GPU in your laptop or desktop PC!

- **CUDA C/C++**

- **Thrust C++ Template Library**

- **CUDA Fortran**

- **Alea GPU**
  - [http://www.aleagpu.com](http://www.aleagpu.com)

- **PyCUDA (Python)**

- **MATLAB**

- **Mathematica**
CUDA C/C++ BASICS
What is CUDA?

- CUDA Architecture
  - Used to mean “Compute Unified Device Architecture”
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
A Brief History of CUDA

• Researchers used OpenGL APIs for general purpose computing on GPUs before CUDA.
• In 2007, NVIDIA released first generation of Tesla GPU for general computing together their proprietary CUDA development framework.
• Current stable version of CUDA is 8.0 (as of Sept. 2017).
• CUDA 9 Release Candidate is available.
Heterogeneous Computing

- **Terminology:**
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
```cpp
#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;
}
```

Heterogeneous Computing

parallel function

serial code

parallel code

serial code

parallel code

serial code
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
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
Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016
Hello World!

```c
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

```c
__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

```c
__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`, `icc`, etc.
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

```c
__global__ void mykernel(void)
{
}

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

- `mykernel()` does nothing!

Output:

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

\[
\begin{align*}
a + b &= c
\end{align*}
\]
Addition on the Device

• A simple kernel to add two integers

```c
__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.
Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    May be passed to/from host code
    May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    May be passed to/from device code
    May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: `add()`

- Returning to our `add()` kernel

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- Let’s take a look at `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()`

```c
// 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;
```
Moving to Parallel

• GPU computing is about massive parallelism
  – So how do we run code in parallel on the device?
    \[
    \text{add}<<<1, 1 >>>();
    \]
    \[
    \text{add}<<<N, 1 >>>();
    \]

• Instead of executing \text{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`

```c
__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

```c
__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
\[
\begin{align*}
    c[0] &= a[0] + b[0]; \\
\end{align*}
\]

Block 1
\[
\begin{align*}
    c[1] &= a[1] + b[1]; \\
\end{align*}
\]

Block 2
\[
\begin{align*}
\end{align*}
\]

Block 3
\[
\begin{align*}
\end{align*}
\]
Vector Addition on the Device: \texttt{add()} 

• Returning to our parallelized \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

• Let’s take a look at \texttt{main()}...
Vector Addition on the Device: `main()`

```c
#define N 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 set up input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```
Vector Addition on the Device: `main()`

```c
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(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;
```
## Profiling with nvprof

### Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.45%</td>
<td>4.3520us</td>
<td>1</td>
<td>4.3520us</td>
<td>4.3520us</td>
<td>4.3520us</td>
<td>add(int*, int*, int*)</td>
</tr>
<tr>
<td>30.35%</td>
<td>3.0400us</td>
<td>2</td>
<td>1.5200us</td>
<td>1.3120us</td>
<td>1.7280us</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>26.20%</td>
<td>2.6240us</td>
<td>1</td>
<td>2.6240us</td>
<td>2.6240us</td>
<td>2.6240us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
</tbody>
</table>

### API calls:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.34%</td>
<td>231.73ms</td>
<td>3</td>
<td>77.242ms</td>
<td>6.1990us</td>
<td>231.71ms</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>0.33%</td>
<td>766.63us</td>
<td>182</td>
<td>4.2120us</td>
<td>171ns</td>
<td>143.74us</td>
<td>cuDeviceGetAttribute</td>
</tr>
<tr>
<td>0.15%</td>
<td>357.72us</td>
<td>2</td>
<td>178.86us</td>
<td>173.06us</td>
<td>184.67us</td>
<td>cuDeviceTotalMem</td>
</tr>
<tr>
<td>0.08%</td>
<td>175.05us</td>
<td>3</td>
<td>58.351us</td>
<td>6.6470us</td>
<td>147.94us</td>
<td>cudaFree</td>
</tr>
<tr>
<td>0.03%</td>
<td>75.722us</td>
<td>1</td>
<td>75.722us</td>
<td>75.722us</td>
<td>75.722us</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>0.03%</td>
<td>74.091us</td>
<td>3</td>
<td>24.697us</td>
<td>10.865us</td>
<td>35.014us</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>0.03%</td>
<td>65.073us</td>
<td>2</td>
<td>32.536us</td>
<td>30.391us</td>
<td>34.682us</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>0.00%</td>
<td>4.6390us</td>
<td>3</td>
<td>1.5460us</td>
<td>221ns</td>
<td>3.9590us</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>0.00%</td>
<td>4.4490us</td>
<td>3</td>
<td>1.4830us</td>
<td>434ns</td>
<td>3.3590us</td>
<td>cudaGetDeviceCount</td>
</tr>
<tr>
<td>0.00%</td>
<td>2.7070us</td>
<td>6</td>
<td>451ns</td>
<td>196ns</td>
<td>777ns</td>
<td>cudaDeviceGet</td>
</tr>
<tr>
<td>0.00%</td>
<td>1.9940us</td>
<td>1</td>
<td>1.9940us</td>
<td>1.9940us</td>
<td>1.9940us</td>
<td>cudaConfigureCall</td>
</tr>
</tbody>
</table>

$nvprof add_parallel
Review (1 of 2)

• Difference between *host* and *device*
  – *Host* CPU
  – *Device* GPU

• Using `__global__` to declare a function as device code
  – Executes on the device
  – Called from the host

• Passing parameters from host code to a device function
Review (2 of 2)

- Basic device memory management
  - cudaMemcpy()
  - cudaMemcpy()
  - cudaMemcpy()
- Launching parallel kernels
  - Launch N copies of add() with add<<<N,1>>>(...).
  - Use blockIdx.x to access block index.
  - Use nvprof for collecting & viewing profiling data.
More Resources

You can learn more about the details at

- CUDA Programming Guide (docs.nvidia.com/cuda)
- CUDA Zone – tools, training, etc. (developer.nvidia.com/cuda-zone)
- Download CUDA Toolkit & SDK (www.nvidia.com/getcuda)
- Nsight IDE (Eclipse or Visual Studio) (www.nvidia.com/nsight)

Intermediate CUDA Programming Short Course

- GPU memory management and unified memory
- Parallel kernels in CUDA C
- Parallel communication and synchronization
- Running a CUDA code on Ada
- Profiling and performance evaluation
Acknowledgements

- Educational materials from NVIDIA via its Academic Programs.
- Supports from Texas A&M Engineering Experiment Station (TEES) and High Performance Research Computing (HPRC).
# load CUDA module
$ml CUDA/8.0.44-GCC-5.4.0-2.26

# copy sample code to your scratch space
$cd $SCRATCH
$cp -r /scratch/training/CUDA .

# compile CUDA code
$cd CUDA
$nvcc hello_world_host.cu -o hello_world

# edit job script & submit your first GPU job
$bsub < cuda_run.sh

https://github.com/jtao/coehpc