Introduction to CUDA Programming

Philip Nee
Cornell Center for Advanced Computing

June 2013

Based on materials developed by CAC and TACC
<table>
<thead>
<tr>
<th>Overview</th>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Heterogeneous Parallel Computing</td>
</tr>
<tr>
<td></td>
<td>• Stampede and NVIDIA K20 GPU</td>
</tr>
<tr>
<td></td>
<td>• Programming Structure</td>
</tr>
<tr>
<td></td>
<td>• Thread Hierarchy</td>
</tr>
<tr>
<td></td>
<td>• Memory Model</td>
</tr>
<tr>
<td></td>
<td>• Performance Topics</td>
</tr>
<tr>
<td>Overview</td>
<td>Terminology</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>• GPU</td>
<td>– Graphics Processing Unit</td>
</tr>
<tr>
<td>• CUDA</td>
<td>– Compute Unified Device Architecture</td>
</tr>
<tr>
<td>• Manycore</td>
<td></td>
</tr>
<tr>
<td>• Multicore</td>
<td></td>
</tr>
<tr>
<td>• SM</td>
<td>– Stream Multiprocessor</td>
</tr>
<tr>
<td>• SIMD</td>
<td>– Single Instruction Multiple Data</td>
</tr>
<tr>
<td>• SIMT</td>
<td>– Single Instruction Multiple Threads</td>
</tr>
</tbody>
</table>
Overview

What is CUDA?

- Compute Unified Device Architecture
  - Manycore and shared-memory programming model
  - An Application Programming Interface (API)
  - General-purpose computing on GPU (GPGPU)

- Multicore vs Manycore
  - Multicore – Small number of sophisticated cores
  - Manycore – Large number of weaker cores
Overview

• Why CUDA?
  – High level
    • C/C++/Fortran language extensions
  – Scalability
  – Thread-level abstraction
  – Runtime library
  – **Thrust** parallel algorithm library

• Limitations
  – Not vendor neutral: NVIDIA CUDA-enabled GPUs only
    • Alternative: **OpenCL**

This course will be in C
Overview

Why are we using GPU?

- Parallel and multithread hardware design
- Floating point computation
  - Graphic rendering
  - General-purpose computing
- Energy Efficiency
  - More FLOPS per Watt than CPU

- MIC vs GPU
  - Comparable performance
  - Different programming models
Different designs for different purposes

- **CPU**: Fast serial processing
  - Large on-chip cache, to minimize read/write latency
  - Sophisticated logic control
- **GPU**: High computational throughputs
  - Large number of cores
  - High memory bandwidth
<table>
<thead>
<tr>
<th>Overview</th>
<th>Heterogeneous Parallel Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Sandy Bridge E5 - 2680</strong></td>
</tr>
<tr>
<td>Processing Units</td>
<td>8</td>
</tr>
<tr>
<td>Clock Speed (GHz)</td>
<td>2.7</td>
</tr>
<tr>
<td>Maximum Threads</td>
<td>8 cores, 2 threads each = 16 threads</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>51.6 GB/s</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>64 KB/core</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>256 KB/core</td>
</tr>
<tr>
<td>L3 Cache Size</td>
<td>20MB</td>
</tr>
</tbody>
</table>

*SM = Stream Multiprocessor*
Overview

SIMD

• SISD: Single Instruction Single Data
• SIMD: Single Instruction Multiple Data
  – A vector instruction that perform the same operation on multiple data simultaneously

• SIMD Instruction Sets:
  – MMX
    • Multimedia eXtension
  – SSE
    • Streaming SIMD Extensions
  – AVX
    • Advanced Vector Extensions
• 6400+ compute nodes, each has:
  – 2 Sandy Bridge processors (E5-2680)
  – 1 Xeon Phi Coprocessor (MIC)

• There are 128 GPU nodes, each is augmented with 1 NVIDIA K20 GPU

• Login nodes do not have GPU cards installed!
Running your GPU application on Stampede:

- Load CUDA software using the `module` utility

- Compile your code using the NVIDIA `nvcc compiler`
  - Acts like a wrapper, hiding the intrinsic compilation details for GPU code

- Submit your job to a `GPU queue`
1. Extract the lab files to the home directory

   $ cd $HOME
   $ tar xvf ~tg459572/LABS/Intro_CUDA.tar

2. Load the CUDA software

   $ module load cuda
3. Go to lab 1 directory, `devicequery`

```bash
$ cd Intro_CUDA/devicequery
```

- There are 2 files:
  - Source code: `devicequery.cu`
  - Batch script: `batch.sh`

4. Use NVIDIA `nvcc` compiler, to compile the source code

```bash
$ nvcc -arch=sm_30 devicequery.cu -o devicequery
```
5. Job submission:
   – Running 1 task on 1 node: `#SBATCH -n 1`
   – GPU development queue: `#SBATCH -p gpudev`

$ sbatch batch.sh
$ more gpu_query.o

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Time Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu</td>
<td>24 hrs</td>
<td>GPU queue</td>
</tr>
<tr>
<td>gpudev</td>
<td>4 hrs</td>
<td>GPU development node</td>
</tr>
<tr>
<td>vis</td>
<td>8 hrs</td>
<td>GPU nodes + VNC service</td>
</tr>
</tbody>
</table>
CUDA Device Query...
There are 1 CUDA devices.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Device #0</td>
<td></td>
</tr>
<tr>
<td>Major revision number</td>
<td>3</td>
</tr>
<tr>
<td>Minor revision number</td>
<td>5</td>
</tr>
<tr>
<td>Name</td>
<td>Tesla K20m</td>
</tr>
<tr>
<td>Total global memory</td>
<td>5032706048</td>
</tr>
<tr>
<td>Total shared memory per block</td>
<td>49152</td>
</tr>
<tr>
<td>Total registers per block</td>
<td>65536</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Maximum memory pitch</td>
<td>2147483647</td>
</tr>
<tr>
<td>Maximum threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum dimension 0 of block</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum dimension 1 of block</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum dimension 2 of block</td>
<td>64</td>
</tr>
<tr>
<td>Maximum dimension 0 of grid</td>
<td>2147483647</td>
</tr>
<tr>
<td>Maximum dimension 1 of grid</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimension 2 of grid</td>
<td>65535</td>
</tr>
<tr>
<td>Clock rate</td>
<td>705500</td>
</tr>
<tr>
<td>Total constant memory</td>
<td>65536</td>
</tr>
<tr>
<td>Texture alignment</td>
<td>512</td>
</tr>
<tr>
<td>Concurrent copy and execution</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of multiprocessors</td>
<td>13</td>
</tr>
<tr>
<td>Kernel execution timeout</td>
<td>No</td>
</tr>
</tbody>
</table>
• Host Code
  – Your CPU codes
  – Takes care of:
    • Device memory
    • Kernel invocation

• Kernel Code
  – Your GPU code
  – Executed on the device
  – __global__ qualifier
    • Must have return type void
• Function Type Qualifiers in CUDA
  __global__
  • Callable from the host only
  • Executed on the device
  • void return type
  __device__
  • Executed on the device only
  • Callable from the device only
  __host__
  • Executed on the host only
  • Callable from the host only
  • Equivalent to declare a function without any qualifier

• There are **variable type qualifiers** available
• Visit the [NVIDIA documentation](https://developer.nvidia.com/) for detail information
• Kernel is invoked from the host

```c
int main() {
    ...
    //Kernel Invocation
gpufunc<<<gridConfig,
blkConfig>>>(arguments…)
    ...
```

• Calling a kernel uses familiar syntax (function/subroutine call) augmented by Chevron syntax

• The Chevron syntax (<<<…>>>) configures the kernel
  – First argument: How many blocks in a grid
  – Second argument: How many threads in a block
Thread Hierarchies

- **Thread**
- **Block**
  - Assigned to a SM
  - Independent
  - Threads within a block can:
    - Synchronize
    - Share data
    - Communicate
  - On K20: 1024 threads per block (max)
- **Grid**
  - Invoked kernel
<table>
<thead>
<tr>
<th>Thread</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads and blocks have its unique ID</td>
<td></td>
</tr>
<tr>
<td>- Thread: <code>threadIdx</code></td>
<td></td>
</tr>
<tr>
<td>- Block: <code>blockIdx</code></td>
<td></td>
</tr>
<tr>
<td><code>threadIdx</code> can have maximum 3 dimensions</td>
<td></td>
</tr>
<tr>
<td>- <code>threadIdx.x</code>, <code>threadIdx.y</code>, and <code>threadIdx.z</code></td>
<td></td>
</tr>
<tr>
<td><code>blockIdx</code> can have maximum 2 dimensions</td>
<td></td>
</tr>
<tr>
<td>- <code>blockIdx.x</code> and <code>blockIdx.y</code></td>
<td></td>
</tr>
<tr>
<td>Why multiple dimensions?</td>
<td></td>
</tr>
<tr>
<td>- Programmer’s convenience</td>
<td></td>
</tr>
<tr>
<td>- Think about working with a 2D array</td>
<td></td>
</tr>
</tbody>
</table>
### Thread Parallelism

#### Types of parallelism:

- **Thread-level Task Parallelism**
  - Every thread or group of threads, executes a different instruction
  - Not ideal because of thread divergence

- **Task Parallelism**
  - Different blocks perform different tasks
  - Invoke multiple kernels to perform different tasks

- **Data Parallelism**
  - Shared memory across threads and blocks
Threads in a block are bundled into small groups of *warps*

- 1 warp = 32 Threads with consecutive *threadIdx* values
  - Example: [0..31] from the first warp, [32…63] from the second warp

- A full warp is mapped to the SIMD unit (Single Instruction Multiple Threads, *SIMT*)

- Threads in a warp cannot diverge
  - Divergence serializes the execution

- Example: In an *if-then-else* construct:
  1. All threads will execute *then*
  2. then execute *else*. 
Memory Model

- **Kernel**
  - per-device Global Memory
- **Block**
  - per-block shared memory
- **Thread**
  - per-thread register
- **CPU and GPU do not share memory**

Two-way arrows indicate read/write capability
Memory Model

- **per-thread register**
  - Private, storage for local variables
  - Fastest
  - Life time: thread

- **per-block shared memory**
  - Shared within a block
  - 48k, fast
  - Life time: Kernel
  - `__shared__` qualifier

- **per-device global memory**
  - Shared
  - 5G, Slowest
  - Life time: Application
Memory Transfer

- Allocate device memory
  - `cudaMalloc()`

- Memory transfer between host and device
  - `cudaMemcpy()`

- Deallocate memory
  - `cudaFree()`
int main()
{
    //Host memory allocation
    h_A=(float *)malloc(size);
    h_B=(float *)malloc(size);
    h_C=(float *)malloc(size);

    //Device memory allocation
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    //Memory transfer, kernel invocation
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);

    vec_add<<<<<N/512, 512>>>(d_A, d_B, d_C);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);
}

Allocate host memory

Allocate device memory

Transfer memory from host to device

Invoke the kernel

Transfer memory from device to host

Deallocate the memory

h_varname : host memory
d_varname : device memory
Memory

Lab 2: Vector Add

//Vector Size
#define N 5120000

//Kernel function
__global__
void vec_add(float *d_A, float *d_B, float *d_C)
{
    //Define Index
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    //Vector Add
    d_C[i] = d_A[i] + d_B[i];
}

int main()
{
    ...
    vec_add<<<N/512, 512>>>(d_A, d_B, d_C);
    ...
}
$ cd $HOME/Intro_CUDA/vectoradd
$ nvcc -arch=sm_30 vectoradd.cu -o vectoradd
$ sbatch batch.sh

• Things to try on your own (After the talk):
  – Time the performance using different vector length
  – Time the performance using different block size

• Timing tool:
  – /usr/bin/time –p <executable>
  – CUDA also provides a better timing tool, see NVIDIA Documentation
Advanced Performance Topics

• Minimize execution divergence
  – Thread divergence serializes the execution

• Maximize on-chip memory (per-block shared, and per-thread)
  – Global memory is slow (~200GB/s)

• Optimize memory access
  – Coalesced memory access
Advanced Performance Topics

• **What is coalesced memory access?**
  – Combine all memory transactions into a single warp access
  – K20: 32 threads * 4-byte word = 128 bytes

• **What are the requirements?**
  – Memory alignment
  – Sequential memory access
  – Dense memory access
1 Transaction
Sequential and aligned (Stride 1)

1 Transaction
Non-sequential and Aligned (Stride 1)

2 Transactions
Sequential and Misaligned (Stride 2)
Consider the following code:

- Is memory access aligned?
- Is memory access sequential?

//The variable, offset, is a constant
int i=blockDim.x * blockIdx.x + threadIdx.x;
int j=blockDim.x * blockIdx.x + threadIdx.x + offset;
d_B2[i]=d_A2[j];
Summary

• GPU is very good at massive parallel jobs, and CPU is very good at serial processing

• Avoid thread divergence

• Use on-chip memory

• Always try to perform coalesced memory access
$ cd $HOME/Intro_CUDA/matrix_mul
$ nvcc -arch=sm_30 matrix_mul.cu -o matmul
$ sbatch batch.sh

- Things to try on your own (After the talk):
  - Compare the performance to the CUDA BLAS matrix multiplication routine
  - Can you improve the performance of it?
    - Hints:
      - Use on-chip memory
      - Use page-locked memory (see cudaMallocHost())
Recommended Reading:

- CUDA Documentation