Introduction to CUDA Programming

Steve Lantz
Cornell University Center for Advanced Computing

October 30, 2013

Based on materials developed by CAC and TACC
Outline

• Motivation for GPUs and CUDA
• Overview of Heterogeneous Parallel Computing
• TACC Facts: the NVIDIA Tesla K20 GPUs on Stampede
• Structure of CUDA Programs
• Threading and the Thread Hierarchy
• Memory Model
• Advanced Performance Tips
Motivation

Why Use GPUs?

• Parallel and multithreaded hardware design
• Floating-point computations
  – Graphics rendering
  – General-purpose computing as well
• Energy efficiency
  – More FLOP/s per watt than CPUs

• MIC vs. GPU
  – Comparable performance
  – Different programming models
### Motivation

- **Compute Unified Device Architecture**
  - Many-core, shared-memory, multithreaded programming model
  - An Application Programming Interface (API)
  - General-purpose computing on GPUs (GPGPU)

- **Multi-core vs. Many-core**
  - Multi-core – Small number of sophisticated cores (=CPUs)
  - Many-core – Large number of weaker cores (=GPU)
Motivation | Why CUDA?

• Advantages
  – High-level, C/C++/Fortran language extensions
  – Scalability
  – Thread-level abstractions
  – Runtime library
  – **Thrust** parallel algorithm library

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

This presentation will be in C
Overview

Heterogeneous Parallel Computing

- CPU: Fast serial processing
  - Large on-chip caches
  - Minimal read/write latency
  - Sophisticated logic control

- GPU: High parallel throughput
  - Large numbers of cores
  - High memory bandwidth
### Overview

#### Different Designs, Different Purposes

<table>
<thead>
<tr>
<th></th>
<th>Intel Sandy Bridge E5 - 2680</th>
<th>NVIDIA Tesla K20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Units</td>
<td>8</td>
<td>13 SMs, 192 cores each, 2496 cores total</td>
</tr>
<tr>
<td>Clock Speed (GHz)</td>
<td>2.7</td>
<td>0.706</td>
</tr>
<tr>
<td>Maximum Hardware Threads</td>
<td>8 cores, 1 thread each (not 2: hyperthreading is off) = 8 threads with <strong>SIMD</strong> units</td>
<td>13 SMs, 192 cores each, all with <strong>32-way SIMT</strong> = 79872 threads</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>51.6 GB/s</td>
<td>205 GB/s</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>64 KB/core</td>
<td>64 KB/SMs</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>256 KB/core</td>
<td>768 KB, shared</td>
</tr>
<tr>
<td>L3 Cache Size</td>
<td>20MB</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**SM = Stream Multiprocessor**
<table>
<thead>
<tr>
<th>Overview</th>
<th>Alphabet Soup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>- Graphics Processing Unit</td>
</tr>
<tr>
<td><strong>GPGPU</strong></td>
<td>- General-Purpose computing on <strong>GPUs</strong></td>
</tr>
<tr>
<td><strong>CUDA</strong></td>
<td>- Compute <strong>Unified Device Architecture</strong> (NVIDIA)</td>
</tr>
<tr>
<td><strong>Multi-core</strong></td>
<td>- A processor chip with 2 or more CPUs</td>
</tr>
<tr>
<td><strong>Many-core</strong></td>
<td>- A processor chip with 10s to 100s of “CPUs”</td>
</tr>
<tr>
<td><strong>SM</strong></td>
<td>- Stream Multiprocessor</td>
</tr>
<tr>
<td><strong>SIMD</strong></td>
<td>- Single Instruction Multiple Data</td>
</tr>
</tbody>
</table>
| **SIMT** | - Single Instruction Multiple Threads  
= **SIMD-style multithreading on the GPU** |
Overview

SIMD

• SISD: Single Instruction Single Data
• SIMD: Single Instruction Multiple Data
  – Example: a vector instruction performs the same operation on multiple data simultaneously
  – Intel and AMD extended their instruction sets to provide operations on vector registers
• Intel’s SIMD extensions
  – MMX
    Multimedia eXtensions
  – SSE
    Streaming SIMD Extensions
  – AVX
    Advanced Vector Extensions
• SIMD matters in CPUs
• It also matters in GPUs
• 6400+ compute nodes, each with:
  – 2 Intel Sandy Bridge processors (E5-2680)
  – 1 Intel Xeon Phi coprocessor (MIC)

• 128 GPU nodes, each augmented with 1 NVIDIA Tesla K20 GPU

• Login nodes do not have GPU cards installed!
To run your CUDA application on one or more Stampede GPUs:

- 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 your 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`

$ 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

$ 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

$$\text{sbatch batch.sh}$$
$$\text{\$ more gpu_query.o[job ID]}$$

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Time Limit</th>
<th>Max Nodes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu</td>
<td>24 hrs</td>
<td>32</td>
<td>GPU main queue</td>
</tr>
<tr>
<td>gpudev</td>
<td>4 hrs</td>
<td>4</td>
<td>GPU development nodes</td>
</tr>
<tr>
<td>vis</td>
<td>8 hrs</td>
<td>32</td>
<td>GPU nodes + VNC service</td>
</tr>
<tr>
<td>visdev</td>
<td>4 hrs</td>
<td>4</td>
<td>GPU + VNC development</td>
</tr>
</tbody>
</table>
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number: 3
Minor revision number: 5
Name: Tesla K20m
Total global memory: 5032706048
Total shared memory per block: 49152
Total registers per block: 65536
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Clock rate: 705500
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 13
Kernel execution timeout: No
Structure

CPU Host

- CPU Code

Kernel Invocation

- (continue) CPU Code

GPU Device

Grid

- Block (0, 0)  
- Block (0, 1)
- Block (1, 0)  
- Block (1, 1)
- Block (2, 0)  
- Block (2, 1)
- Block (3, 0)  
- Block (3, 1)
- Block (4, 0)  
- Block (4, 1)
- Block (5, 0)  
- Block (5, 1)
Host Code
- Your CPU code
- Takes care of:
  - Device memory
  - Kernel invocation

Kernel Code
- Your GPU code
- Executed on the device
- __global__ qualifier
  - Must have return type void

```c
int main() {
    ...
    //CPU code
    [Invoke GPU functions]
    ...
}

__global__
void gpufunc(arg1, arg2, ...) {
    ...
    //GPU code
    ...
}
```
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 declaring a function without any qualifier

There are variable type qualifiers available as well.

Refer to the NVIDIA documentation for details.
• 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
**Threading**

- **Thread**
  - Basic execution unit

- **Block**
  - Thread group assigned to an SM*
  - Independent
  - Threads within a block can:
    - Synchronize
    - Share data
    - Communicate

- **Grid**
  - All the blocks invoked by a kernel

*Max 1024 threads per block (K20)
Threading

• Threads and blocks have unique IDs
  – Thread: threadIdx
  – Block: blockIdx

• threadIdx can have maximum 3 dimensions
  – threadIdx.x, threadIdx.y, and threadIdx.z

• blockIdx can have maximum 2 dimensions
  – blockIdx.x and blockIdx.y

• Why multiple dimensions?
  – Programmer’s convenience
  – Helps to think about working with a 2D array
Types of parallelism:

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

- **Block-level Task Parallelism**
  - Different blocks perform different tasks
  - Multiple kernels are invoked to start the tasks

- **Data Parallelism**
  - Memory is shared across threads and blocks
Threads in a block are bundled into small groups of *warps*

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

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

- Therefore, threads in a warp cannot diverge
  - Execution is serialized to prevent divergence
  - For example, in an *if-then-else* construct:
    1. All threads execute *then* – affects only threads where condition is true
    2. All threads execute *else* – affects only threads where condition is false
Memory Model

- **Kernel**
  - Per-device global memory
- **Block**
  - Per-block shared memory
- **Thread**
  - Per-thread local memory
  - Per-thread register

- CPU and GPU do not share memory
Memory

- Per-thread local memory
  - Private storage for local variables
  - Fastest
  - Lifetime: thread

- Per-block shared memory
  - Shared within a block
  - 48kB, fast
  - Lifetime: kernel
  - __shared__ qualifier

- Per-device global memory
  - Shared
  - 5GB, Slowest
  - Lifetime: application
Memory Transfer

- Allocate device memory
  - `cudaMalloc()`

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

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

    //Allocate device memory
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    //Move data from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);

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

    //Deallocate the memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);
}

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 a different vector length
  – Time the performance using a different block size

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

• 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 Coalesced Memory Access

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

• What are the requirements?
  – Memory alignment
  – Sequential memory access
  – Dense memory access
Advanced

Memory Alignment

1 Transaction:
Sequential,
In-order,
Aligned

1 Transaction:
Sequential,
Reordered,
Aligned

2 Transactions:
Sequential,
In-order,
Misaligned
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 massively parallel jobs
  – CPU is very good at moderately parallel jobs and serial processing
• GPU threads and memory are linked in a hierarchy
  – A block of threads shares local memory (on the SM)
  – A grid of blocks shares global memory (on the device)
• CUDA provides high-level syntax for assigning work
  – The kernel is the function to be executed on the GPU
  – Thread count and distribution are specified when a kernel is invoked
  – cudaMemcpy commands move data between host and device
• Programming rules must be followed to get the best performance
  – Move data between host and device as little as possible
  – Avoid thread divergence within a warp of threads (32)
  – Preferentially use on-chip (local block) memory
  – Try to perform coalesced memory accesses with warps of threads
$ 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 its performance? Hints:
    • Use on-chip memory
    • Use page-locked memory, see cudaMallocHost()
Recommended Reading:

- CUDA Documentation