Introduction to Programming in CUDA

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

Some materials/slides are adapted from:
Andreas Moshovos’ Course at the University of Toronto
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas

CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU

- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Explicit GPU memory management
CPU-GPU Interaction

- **GPU:**
  - Runs a number of compute-intensive *Kernels*

- **CPU:**
  - Provides data for GPU and manages it
  - Runs the rest of the computation

**Diagram:**
- **CPU:**
  - 1. Copy Input Data
  - 2. Call Kernel
  - 3. Return
  - 4. Copy Result
- **GPU:**
  - 1. Copy Input Data
  - 2. Call Kernel
  - 3. Return
  - 4. Copy Result
CPU-GPU Communication Path

- ~1000 cycles latency
- ~10GB/s bandwidth limitation
- Only coarse-grained offloading

CUDA Devices and Threads

- A Compute device
  - A coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device

- Data-parallel portions of an application are expressed as device kernels which run on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
A Sample CUDA Device

CUDA Language: Extended C

- **Type Qualifiers**
  - `global`, `device`, `shared`, `local`, `constant`

- **Keywords**
  - `threadIdx`, `blockIdx`

- **Intrinsics**
  - `__syncthreads`

- **Runtime API**
  - `Memory Management`
  - `Execution Management`

- **Function launch**

```c
__device__ float filter[N];

__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx] = image[i];
    __syncthreads();
    ...
    image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```
Parallel Threads

/* N is large */
for (i = 0; i < N; i++)
{
    a[i] = a[i] * c;
}

CUDA Kernel

- A piece of code executed on GPU (device)
- A CUDA kernel is executed by an array of parallel threads
- All threads run the same code (SPMD model)
- Each thread has a unique ID
Thread Blocks: Scalable Cooperation

Divide monolithic thread array into multiple blocks

- Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
- Threads in different blocks cannot cooperate

Threads within a block cooperate via shared memory, atomic operations and barrier synchronization.

**Threads in different blocks cannot cooperate**

Grids of Blocks of Threads

Why? Realities of integrated circuits: need to cluster computation and storage to achieve high speeds

**Source:** NVIDIA
Dimension Limits

- Grid of Blocks 1D or 2D
  - Max x: 65535
  - Max y: 65535

- Block of Threads: 1D, 2D, or 3D
  - Max number of threads: 512
  - Max x: 512
  - Max y: 512
  - Max z: 64

- Limits apply to Compute Capability 1.0, 1.1, 1.2, and 1.3
  - GTX280 = 1.3

---

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data

- IDs and dimensions are easily accessible through predefined “variables”, e.g., `blockDim.x` and `threadIdx.x`

source: NVIDIA
Memory Model

- Shared Memory
  - Shared among all threads within the same thread block
  - Very fast
  - Small in size (16 Kbytes)

- Global Memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access (hundreds of CPU cycles)
  - Large in size (1 GBytes)

- Texture and Constant Memory
  - Constants initialized by host
  - Contents visible to all threads
### Memory Model Summary

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>off-chip</td>
<td>No</td>
<td>R/W</td>
<td>thread</td>
</tr>
<tr>
<td>Shared</td>
<td>on-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>all threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>off-chip</td>
<td>No</td>
<td>R/W</td>
<td>all threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>off-chip</td>
<td>Yes</td>
<td>RO</td>
<td>all threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>off-chip</td>
<td>Yes</td>
<td>RO</td>
<td>all threads + host</td>
</tr>
</tbody>
</table>

### CUDA Execution Model

- All blocks are identical
  - Same structure, same number of threads

- Block execution order is undefined
  - Thread block 1 can execute **before** thread block 0 or vice versa

- Synchronization
  - Threads within the same block can synchronize through shared memory
  - Threads from different blocks cannot communicate => cannot synchronize => cannot cooperate

- Block-to-Processor assignment
  - Multiple thread blocks can be assigned to the same SM
  - Thread blocks do not migrate from one SM to another during kernel execution
CUDA Example

Kernel Candidate:

```c
/* N is large */
for (i = 0; i < N; i++)
{
    a[i] = a[i] * c;
}
```

Steps:
1. Memory Allocation on CPU and GPU
2. Initialize Data on CPU
3. Copy Data from CPU to GPU
4. Define Execution Configuration
5. Run Kernel
6. CPU synchronizes with GPU
7. Copy Data from GPU to CPU
8. Deallocate GPU and CPU memory

---

Step 1: Memory Allocation

- Allocating host memory
  ```c
  float *host_array;
  host_array = (float *)malloc(sizeof(float) * N);
  if (host_array == NULL) {
      printf("ERROR: cannot allocate memory \n");
      exit(-1);
  }
  ```

- Allocating device (GPU) memory
  ```c
  cudaError_t error;
  float *device_array;
  error = cudaMalloc((void *)&device_array,
                      sizeof(float) * N);
  if (error != cudaSuccess) {
      printf("ERROR: cannot allocate memory \n");
      ...
  }
  ```
Step 2: Initialization

```c
for (j = 0 ; j < N; j++) {
    host_array[j] = j;
}
```

- No direct access to the GPU memory
  - Distributed Memory (as opposed to shared memory)
  - Access only through `cudaxxxx` functions

---

Step 3: Copy Data From CPU to GPU

```c
float *host_array;
float *device_array;
...

cudaMemcpy((void *)device_array, // Destination
            (void *)host_array, // Source
            sizeof(float) * N, // Size
            cudaMemcpyHostToDevice); // Direction
```
Host/Device Data Transfers

- The host initiates all transfers:
  ```c
  cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  ```
- Asynchronous from the CPU’s perspective
  - CPU thread continues
- In-order processing with other CUDA requests

```c
enum cudaMemcpyKind {
  cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice,
};
```

Step 4: Define Execution Configuration

```c
// we’ll see how to find a 
// good value for this later
int blk_size = 64;

int num_blks = N / blk_size;

// adding one more block if N is 
// not a multiple of thread_block
if (num_blks % N != 0)
    num_blks++;
```
Step 5: Run the Kernel

\[
d_{\text{add}} \langle\langle \text{num\_blks, blk\_size} \rangle\rangle (\text{device\_array, 3.85, N});
\]
\[
cudaThreadSynchronize();
\]
- Kernel Name: \texttt{d\_add}
- Execution Configuration:
  \[
  \langle\langle \text{num\_blks, blk\_size} \rangle\rangle
  \]
  - We’ll talk more about this later
- Arguments: \texttt{(device\_array, 3.85, N)}

Step 6: CPU-GPU Synchronization

- CPU does not block on \texttt{cuda...()} calls
  - Kernel/requests are queued and processed in-order
  - Control returns to CPU immediately
- Good if there is other work to be done
  - e.g., preparing for the next kernel invocation
- Eventually, CPU must know when GPU is done
- Then it can safely copy the GPU results
- \texttt{cudaThreadSynchronize()}:
  - Block CPU until all preceding cuda...() and kernel requests have completed
Step 7: Copy Results from GPU to CPU

```c
float *host_array;
float *device_array;
...

cudaMemcpy((void *)host_array, // Destination
device_array,             // Source
sizeof(float) * N,         // Size
cudaMemcpyDeviceToHost);   // Direction
```

Step 8: Deallocate Memories

```c
free(host_array);
cudaFree(device_array);
```
The GPU Kernel

```c
__global__ d_add (float *da, float x, int N)
{
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
        device_array[i] = device_array[i] * x;
}
```

- **BlockIdx**: unique block ID (0,1,...)
- **ThreadId**: per block thread ID (0,1,...)
- **BlockDim**: Dimensions of the blocks
  - `BlockDim.x, BlockDim.y, BlockDim.z`
  - Unused dimensions default to 0

---

Array Index Calculation Example

```c
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

---

Source: A. Moshovos
Thread and Block Index Calculations #1

- **1D Grid / 1D Blocks:**
  
  UniqueBlockIndex = blockIdx.x;
  
  UniqueThreadIndex = blockIdx.x * blockDim.x + threadIdx.x;

- **1D Grid / 2D Blocks:**
  
  UniqueBlockIndex = blockIdx.x;
  
  UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;

- **1D Grid / 3D Blocks:**
  
  UniqueBlockIndex = blockIdx.x;
  
  UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;

Source: A. Moshovos

---

Thread and Block Index Calculations #2

- **2D Grid / 1D Blocks:**
  
  UniqueBlockIndex = blockIdx.y * blockDim.x + blockIdx.x;
  
  UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;

- **2D Grid / 2D Blocks:**
  
  UniqueBlockIndex = blockIdx.y * blockDim.x + blockIdx.x;
  
  UniqueThreadIndex = UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;

- **2D Grid / 3D Blocks:**
  
  UniqueBlockIndex = blockIdx.y * blockDim.x + blockIdx.x;
  
  UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x + threadIdx.z * blockDim.y * blockDim.x + threadIdx.x * blockDim.x + threadIdx.x;

Source: A. Moshovos
**CUDA Function Declarations**

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__</code> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><code>__global__</code> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><code>__host__</code> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- `__global__` defines a kernel function
  - Must return void
  - Can only call `__device__` functions
- `__device__` and `__host__` can be used together

---

**Restrictions on GPU Functions**

- No recursion
  ```c
  d_add (...)  
  {
    d_add (...)  
  }
  ```

- No static variable declarations inside the function
  ```c
  d_add (...)  
  {
    static int var;  // illegal declaration  
  }
  ```

- No variable number of arguments
  - e.g., something like `printf (....)`