CUDA Programming

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
Department of Biological Sciences
University of Southern California

Email: anakano@usc.edu

Goal: Multithreading on graphics processing units (GPUs)
Graphics Processing Unit (GPU)

- **GPU**: A specialized processor that offloads 3D graphics rendering from the central processing unit (CPU).

- **GPGPU**: General-purpose computing on GPU, by using a GPU to perform computation traditionally handled by the CPU; GPU is considered as a multithreaded, massively data parallel co-processor (accelerator).

- **NVIDIA Quadro & Tesla GPUs** are capable of general-purpose computing with the use of Compute Unified Device Architecture (CUDA).

Tesla K20 (2496 cores)
• **CUDA**

- **Compute Unified Device Architecture**
- **Integrated host (CPU) + device (GPU) application programming interface based on C language developed at NVIDIA**
- **CUDA homepage**
  
  http://www.nvidia.com/object/cuda_home.html
Using CUDA on HPC

- **Set an environment on the front-end** *(ssh to hpc-login3.usc.edu)*
  
  ```bash
  source /usr/usd/cuda/default/setup.csh (if tcsh)
  or
  source /usr/usd/cuda/default/setup.sh (if bash)
  ```

- **Compilation**
  
  ```bash
  nvcc -o pi pi.cu
  ```

- **Submit a PBS script using the qsub command**
  
  ```bash
  #!/bin/bash
  #PBS -l nodes=1:ppn=1:gpus=1
  #PBS -l walltime=00:00:59
  #PBS -o pi.out
  #PBS -j oe
  #PBS -N pi
  #PBS -A lc_an1
  source /usr/usd/cuda/default/setup.sh
  cd /home/rcf-proj/anl/your_folder
  ./pi
  ```
NVIDIA Tesla at HPC

- **Host (CPU)**
  - Dual octacore \((2 \times 8 = 16)\) Intel Xeon
  - Clock rate: 2.4 GHz
  - Memory: 64 GB

- **Device (GPU): Dual NVIDIA Tesla K20m**
  - Number of streaming multiprocessors (SMs) per GPU: 13
  - Number of cores (or streaming processors, SPs) per SM: 192
  - Total number of cores: \(13 \times 192 = 2496\)
  - Clock rate: 706 MHz
  - Global memory: 5 GB
  - Shared memory per SM: 48 KB
Grid, Blocks, & Threads

- **Computational grid** = a 1 or 2D grid of thread blocks (cf. SMs); each block = a 1, 2 or 3D array of ≤ 512 threads (cf. SPs); the application specifies the grid & block dimensions
  - **gridDim** provides dimension of grid; 1 or 2 element struct: “.x” & “.y”
  - **blockDim** provides dimension of block; 1, 2 or 3 element struct: “.x”, “.y” & “.z”

- All threads within a block execute the same kernel (SPMD) & cooperate via shared memory, atomic operations & barrier synchronization

- Each block has an unique block ID
  - **blockIdx** is 1 or 2 element struct

- Each thread has a unique ID within the block
  - **threadIdx** is a struct with up to 3 elements: “.x”, “.y” (in 2 or 3D) & “.z” (in 3D) for the innermost, intermediated & outermost index

- Each thread uses the block & thread IDs to decide what data to work on
Each thread can:
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory

Host code can:
- Read/write per-grid global memory
- Read/write per-grid constant memory
Device Memory Allocation

cudaMalloc()

- Allocates object in the device global memory
- Requires two parameters:
  - Address of a pointer to the allocated object
  - Size of allocated object

cudaMalloc((void **) &sumDev, size);

cudaFree()

- Frees object from device global memory
- Parameter: Pointer to freed object

cudaFree(sumDev);
cudaMemCpy(dest, src, size, cmd)

- **Arguments**
  - `dest` = pointer to array to receive data
  - `src` = pointer to array to source data
  - `size` = # of bytes to transfer
  - `cmd` = transfer direction
    
    > `cudaMemcpypyHostToDevice`
    > `cudaMemcpypyDeviceToHost`

- **Transfer specified # of bytes from one memory to the other in direction specified**

```c
cudaMemcpypy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
```
Kernel Program for Device

- Set of threads triggered by invocation of a single kernel
- **Definition**
  ```
  __global__ void kernel_fun(argument_list)
  ``
  - **Kernel that can be called from a host function**
- **Invocation**
  ```
  kernel_fun <<<execution configuration>>>(operands)
  ```
  - Range specifies set of values for thread grid

```c
host_fun() {
  ...
  dim3 dimGrid(4,2,1)
  dim3 dimBlock(2,2,2)
  kernel_fun <<<dimGrid, dimBlock>>>(operands)
  ...
}
```

- **4×2 grid (3rd dimension not used)**
- **2×2×2 block**

3-element struct accessed by `dimGrid.x, dimGrid.y, dimGrid.z`
**Built-in Variables**

- `dim3 gridDim;`
  
  **Dimensions of the grid in blocks** *(gridDim.z unused)*

- `dim3 blockDim;`
  
  **Dimensions of the block in threads**

- `dim3 blockIdx;`
  
  **Block index within the grid**

- `dim3 threadIdx;`
  
  **Thread index within the block**
Calculate Pi with CUDA: pi.cu (1)

```c
#include <stdio.h>
#include <cuda.h>

#define NBIN 10000000 // Number of bins
#define NUM_BLOCK 13 // Number of thread blocks
#define NUM_THREAD 192 // Number of threads per block

int tid;
float pi = 0;

__global__ void cal_pi(float *sum, int nbin, float step, int nthreads, int nblocks) {
    int i;
    float x;
    int idx = blockIdx.x*blockDim.x+threadIdx.x;  // Sequential thread index across blocks
    for (i=idx; i< nbin; i+=nthreads*nblocks) {
        x = (i+0.5)*step;
        sum[idx] += 4.0/(1.0+x*x);
    }
}
```

**Grid Dimensions:**
- `gridDim.x` = 13
- `gridDim.y` = 1
- `blockDim.x` = 192
- `blockDim.y` = 1
- `blockDim.z` = 1

**Total number of threads:**
- `NBIN` = 10,000,000
- `NUM_THREAD` = 192
- `NUM_BLOCK` = 13
- Total threads = 13 * 192 = 2,496
Calculate Pi with CUDA: \texttt{pi.cu} (2)

```c
// Main routine that executes on the host
int main(void) {
    dim3 dimGrid(NUM_BLOCK, 1, 1); // Grid dimensions
dim3 dimBlock(NUM_THREAD, 1, 1); // Block dimensions
float *sumHost, *sumDev; // Pointer to host & device arrays

float step = 1.0/NBIN; // Step size
size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float); // Array memory size
sumHost = (float *)malloc(size); // Allocate array on host
cudaMalloc((void **) &sumDev, size); // Allocate array on device
// Initialize array in device to 0
cudaMemset(sumDev, 0, size);
// Do calculation on device by calling CUDA kernel
cal_pi <<<dimGrid, dimBlock>>> (sumDev, NBIN, step, NUM_THREAD, NUM_BLOCK);
// Retrieve result from device and store it in host array
cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++)
    pi += sumHost[tid];
pi *= step;

// Print results
printf("PI = %.16f\n",pi);

// Cleanup
free(sumHost);
cudaFree(sumDev);
return 0;
}
```