COMP 605: Introduction to Parallel Computing Lecture : Compute Unified Device Architecture (CUDA) Overview

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 04/20/17
Last Update: 04/20/17

Table of Contents

Compute Unified Device Architecture (CUDA) Overview

Introduction to Compute Unified Device Architecture (CUDA, K&W Ch3; S&K, Ch3)

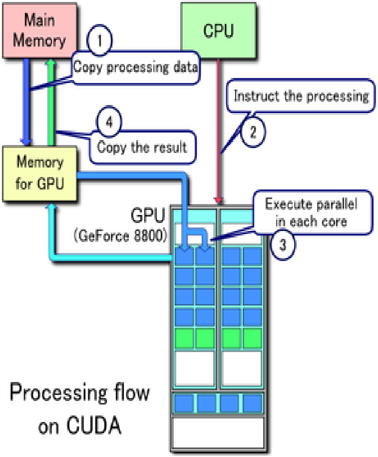
Outline:

- Basic Program Example
- The CUDA Kernel
- Passing Parameters
- Memory Management

CUDA (Compute Unified Device Architecture)

Example of CUDA processing flow:

1. CPU initializes, allocates, copies data from main memory to GPU memory
2. CPU sends instructions to GPU
3. GPU executes parallel code in each core
4. GPU Copies the result from GPU mem to main mem



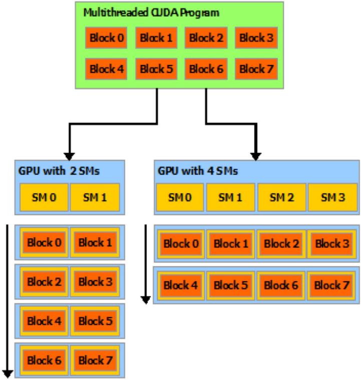
Source: http://en.wikipedia.org/wiki/CUDA

CUDA API (1)

- CUDA C is a variant of C with extensions to define:
  - where a function executes (host CPU or the GPU)
  - where a variable is located in the CPU or GPU address space
  - execution parallelism of kernel function distributed in terms of grids and blocks
  - defines variables for grid, block dimensions, indices for blocks and threads
- Requires the *nvcc* 64-bit compiler and the CUDA driver outputs PTX (Parallel Thread eXecution, NVIDIA pseudo-assembly language) , CUDA, standard C binaries
- CUDA run-time JIT compiler (optional);
  compiles PTX code into native operations
- math libraries, cuFFT, cuBLAS and cuDPP (optional)

CUDA Programming Model

- Mainstream processor chips are parallel systems: multicore CPUs and many core GPUs
- CUDA/GPU provides three key abstractions:
  - hierarchy of thread groups
  - shared memory
  - barrier synchronization
- fine-grained data & thread parallelism, nested within coarse-grained data & task parallelism
- partitions problem into coarse sub-probs solved with parallel independent blocks of threads
- sub-problems divided into finer pieces solved in parallel by all threads in block
- GPU has array of Streaming Multiprocs (SMs)
- Multithreaded program partitioned into blocks of threads that execute independently from each other
- Scales: GPU (more MPs) executes in less time than GPU (fewer MPs).



Source: NVIDIA cuda-c-programming-guide

CUDA Kernel Basics

CUDA Code Example: simple_hello.cu (K&S Ch3)

```
[mthomas@tuckoo hello]$ cat simple_hello.cu
/*
 * Copyright 1993-2010 NVIDIA
 *     Corporation.
 *     All rights reserved.
  */
#include <stdio.h>

__global__ void mykernel( void ) {
}

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

CUDA code highlights:

- *mykernel* <<< 1, 1 >>> () directs the function to be run on the device
- *mykernel*() is an empty function
- *__global__* is a CUDA directive that tells system to run this function on the GPU device

CUDA API: Kernel

In its simplest form it looks like:

kernelRoutine <<< *gridDim*, *blockDim* >>> (*args*)

Kernel runs on the device. It is executed by threads, each of which knows about:

- variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
  - *gridDim*: size (or dimensions) of grid of blocks
  - *blockIdx* : index (or 2D/3D indices)of block
  - *blockDim*: size (or dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread

Function Type Qualifiers

Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device:

- __device__
  - Executed on GPU
  - Launched on GPU
- __global__
  - Executed on device
  - Callable from host
  - Callable from the device for devices of compute capability 3.x
- __host__ (optional)
  - Executed on host
  - Callable from host only

Source:

`http://docs.nvidia.com/cuda/cuda-c-programming-guide/#function-type-qualifiers`

Grids and Blocks

- A *Grid* is a collection of blocks:
  - *gridDim*: size (dimensions) of grid of blocks
  - *blockIdx* : index (2D/3D indices) of block
- A *Block* is a collection of threads (columns):
  - *blockDim*: size (dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread
- *Threads* execute the *kernel* code on *device*:

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Source: *Cuda By Example (Ch 5)*

Two types of parallelism:

Block Parallelism

Launch N blocks with 1 thread each:

$$add <<< N, 1 >>> (dev\_a, \quad dev\_b, \quad dev\_c) >>>$$

Thread Parallelism

Launch 1 block with N threads:

$$add <<< 1, N >>> (dev\_a, \quad dev\_b, \quad dev\_c) >>>$$

We will look at examples for each type of parallel mechanisms.

Memory Allocation

- CPU: malloc, calloc, free, cudaMallocHost, cudaFreeHost
- GPU: cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray

Passing Parameters to the Kernel

simple_kernel_params.cu (part 1)

```
#include <iostream>
#include "book.h"

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

 /* allocate memory on the device for the variable */
 HANDLE_ERROR(
   cudaMalloc((void**)&dev_c, sizeof(int) ) );

 /* nothing to copy -- no call to cudaMemcpy */

 /* launch the kernel */
 add<<<1,1>>>( 2, 7, dev_c );

 /* copy results back from device to the host */
 HANDLE_ERROR(
   cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost)
 );

 printf( "2 + 7 = %d\n", c );

 cudaFree( dev_c );

 return 0;
}
```

- The Kernel: $add <<< 1, 1 >>> (2, 7, dev\_c)$ runs on the device.
- *__global__* is a CUDA directive that tells system to run this function on the GPU device
- Kernel passing variables that are modified on the device.
- using 1 block with 1 thread
- Result passed from the device back to the host
- Must use pointers

simple_kernel_params.cu (part 1)

```
[cuda_by_example/chapter03] nvcc -o simple_add   simple_add.cu

[cuda_by_example/chapter03] qsub simple_add.bat
7987.tuckoo.sdsu.edu
[cuda_by_example/chapter03]$ cat simple_device_call.o7987

simple_device_call using 1 cores...
2 + 7 = 9
```

```
#!/bin/bash
#
#
#PBS -V
#PBS -l nodes=node9:ppn=1
#PBS -N simple_add
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR

echo "Running simple_add."
./simple_add
```