Fundamentals of Accelerated Computing C/C++

Notes inspired by NVIDIA DLI

April 6, 2021

1 Accelerating Applications with CUDA C/C++

1.1 Writing Application Code for GPU

```
void CPUFunction()
{
        printf("This function is defined to on on the CPU.\n");
}
```

```
!nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run
```

nvcc
arch defines the architecture, in this case it is sm_70

1.2 Writing Application Code for the GPU

```
void CPUFunction()
{
  printf("This function is defined to run on the CPU.\n");
}

__global__ void GPUFunction()
{
  printf("This function is defined to run on the GPU.\n");
}

int main()
{
  CPUFunction();

  GPUFunction<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

`__global__`

indicates that that this function will run on the GPU, and must return the type **void()**

```
GPUFunction<<<1, 1>>>();
GPUFunction<<<#ofBlocks,#ofThreads>>>();
```

This function is a kernel, using the ¡¡¡...¿¿¿ syntax allows us to specify the thread hierarchy and the number of thread groupings (called blocks). In this kernel, it is being lanched with 1 block that contains 1 thread.

1.3 CUDA Thread Hierarchy

1.4 Launching Parallel Kernels

The execution configuration allows programmers to specify details about launching the kernel to run in parallel on multiple GPU threads. More precisely, the execution configuration allows programmers to specifiy how many groups of threads - called thread blocks, or just blocks - and how many threads they would like each thread block to contain. The syntax for this is:

```
<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK>>>
```

1.5 Thread and Block Indices

Each thread is given an index within its thread block, starting at 0. Additionally, each block is given an index, starting at 0. Just as threads are grouped into thread blocks, blocks are grouped into a grid, which is the highest entity in the CUDA thread hierarchy. In summary, CUDA kernels are executed in a grid of 1 or more blocks, with each block containing the same number of 1 or more threads.

CUDA kernels have access to special variables identifying both the index of the thread (within the block) that is executing the kernel, and, the index of the block (within the grid) that the thread is within. These variables are **threadIdx.x** and textbfblockIdx.x respectively.

1.6 Accelerating For Loops

```
#include <stdio.h>


/*
 * Notice the absence of the previously expected argument `N`.
 */

__global__ void loop()
{
  /*
   * This kernel does the work of only 1 iteration
   * of the original for loop. Indication of which
   * "iteration" is being executed by this kernel is
   * still available via `threadIdx.x`.
   */

  printf("This is iteration number %d\n", threadIdx.x);
}

int main()
{
  /*
   * It is the execution context that sets how many "iterations"
   * of the "loop" will be done.
   */

  loop<<<1, 10>>>();
  cudaDeviceSynchronize();
}
```

The formula:

```
threadIdx.x + blockIdx.x * blockDim.x
```

will map each thread to one element in the vector

2 Unified Memory

2.1 Iterative Optimizations with the NVIDIA Command Line Profiler

Profile an Application with nsys

Optimize and Profile

Optimize Iterative