Tutorial 02: CUDA in Actions

Introduction

In tutorial 01, we implemented vector addition in CUDA using only one GPU thread. However, the strength of GPU lies in its massive parallelism. In this tutorial, we will explore how to exploit GPU parallelism.

Going parallel

CUDA use a kernel execution configuration `<<<...>>>` to tell CUDA runtime how many threads to launch on GPU. CUDA organizes threads into a group called "***thread block***". Kernel can launch multiple thread blocks, organized into a "***grid***" structure.

The syntax of kernel execution configuration is as follows

```
<<< M , T >>>
```

Which indicate that a kernel launches with a grid of `M` thread blocks. Each thread block has `T` parallel threads.

Exercise 1: Parallelizing vector addition using multithread

In this exercise, we will parallelize vector addition from tutorial 01 (`vector_add.cu`) using a thread block with 256 threads. The new kernel execution configuration is shown below.

```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

CUDA provides built-in variables for accessing thread information. In this exercise, we will use two of them: `threadIdx.x` and `blockIdx.x`.

- `threadIdx.x` contains the index of the thread within the block
- `blockDim.x` contains the size of thread block (number of threads in the thread block).

For the `vector_add()` configuration, the value of `threadIdx.x` ranges from 0 to 255 and value of `blockDim.x` is 256.

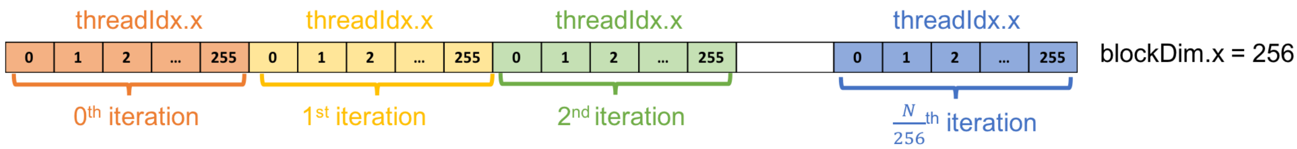
Parallelizing idea

Recalls the kernel of single thread version in `vector_add.cu`. Notes that we modified the `vector_add()` kernel a bit to make the explanation easier.

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = 0;
    int stride = 1
    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
    }
}
```

In this implementation, only one thread computes vector addition by iterating through the whole arrays. With 256 threads, the addition can be spread across threads and computed simultaneously.

For the `k`-th thread, the loop starts from `k`-th element and iterates through the array with a loop `stride` of 256. For example, in the 0-th iteration, the `k`-th thread computes the addition of `k`-th element. In the next iteration, the `k`-th thread computes the addition of `(k+256)`-th element, and so on. Following figure shows an illustration of the idea.



EXERCISE: Try to implement this in `vector_add_thread.cu`

1. Copy `vector_add.cu` to `vector_add_thread.cu`

```
$> cp vector_add.cu vector_add_thread.cu
```

1. Parallelize `vector_add()` using a thread block with 256 threads.
2. Compile and profile the program

```
$> nvcc vector_add_thread.cu -o vector_add_thread
$> nvprof ./vector_add_thread
```

See the solution in `solutions/vector_add_thread.cu`

Performance

Following is the profiling result on Tesla M2050

```
==6430== Profiling application: ./vector_add_thread
==6430== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 39.18%  22.780ms         1  22.780ms  22.780ms  22.780ms  vector_add(float*, float*, float*, int
 34.93%  20.310ms         2  10.155ms  10.137ms  10.173ms  [CUDA memcpy HtoD]
 25.89%  15.055ms         1  15.055ms  15.055ms  15.055ms  [CUDA memcpy DtoH]
```

Exercise 2: Adding more thread blocks

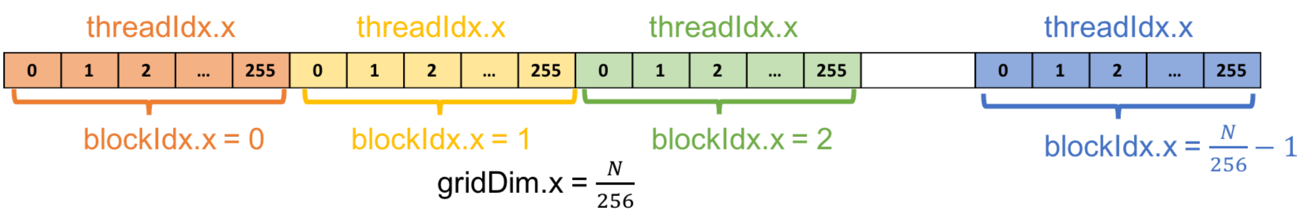
CUDA GPUs have several parallel processors called ***Streaming Multiprocessors* or *SMs***. Each SM consists of multiple parallel processors and can run multiple concurrent thread blocks. To take advantage of CUDA GPUs, kernel should be launched with multiple thread blocks. This exercise will expand the vector addition from exercise 1 to uses multiple thread blocks.

Similar to thread information, CUDA provides built-in variables for accessing block information. In this exercise, we will use two of them: `blockIdx.x` and `gridDim.x`.

- `blockIdx.x` contains the index of the block with in the grid
- `gridDim.x` contains the size of the grid

Parallelizing idea

Instead of using a thread block to iterate over the arrays, we will use multiple thread blocks to create `N` threads; each thread processes an element of the arrays. Following is an illustration of the parallelization idea.



With 256 threads per thread block, we need at least `N/256` thread blocks to have a total of `N` threads. To assign a thread to a specific element, we need to know a unique index for each thread. Such index can be computed as follow

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

EXERCISE: Try to implement this in `vector_add_grid.cu`

1. Copy `vector_add.cu` to `vector_add_grid.cu`

```
$> cp vector_add.cu vector_add_thread.cu
```


1. Parallelize `vector_add()` using multiple thread blocks.

2. Handle case when `N` is an arbitrary number.
3. HINT: Add a condition to check that the thread work within the acceptable array index range.
4. Compile and profile the program

```
$> nvcc vector_add_grid.cu -o vector_add_grid
$> nvprof ./vector_add_grid
```

See the solution in `solutions/vector_add_grid.cu`

Performance

Following is the profiling result on Tesla M2050

```
==6564== Profiling application: ./vector_add_grid
==6564== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 55.65%  20.312ms         2  10.156ms  10.150ms  10.162ms  [CUDA memcpy HtoD]
 41.24%  15.050ms         1  15.050ms  15.050ms  15.050ms  [CUDA memcpy DtoH]
  3.11%  1.1347ms         1  1.1347ms  1.1347ms  1.1347ms  vector_add(float*, float*, float*, int
```

Comparing Performance

Version	Execution Time (ms)	Speedup
1 thread	1425.29	1.00x
1 block	22.78	62.56x
Multiple blocks	1.13	1261.32x

Wrap up

This tutorial gives you some rough idea of how to parallelize program on GPUs. So far, we learned about GPU threads, thread blocks, and grid. We parallized vector addition using multiple threads and multiple thread blocks.

Acknowledgments

- Contents are adopted from An Even Easier Introduction to CUDA by Mark Harris, NVIDIA and CUDA C/C++ Basics by Cyril Zeller, NVIDIA.