Assignment IV: OpenCL

Group 12 - Chao Xiong, Chang Fu

Repository link: https://github.com/Kevin-XiongC/DD2360

Exercise 1 - Hello World!

List and explain your extensions to the basic template provided by the course.

The very first thing is to create a kernel which is represented in a constant string. We name the kernel **HelloWorld**, and append keyword **__kernel** before it. No arguments are needed for this simple kernel. In this kernel we explore 3 functions that are used to get the id of the current thread: **get_group_id(*dim*)**, **get_local_id(*dim*)** and **get_global_id(*dim*)**, where ***dim*** indicates in which dimension the thread lies. Another thing to notice is that for some special characters such as quotation marks and backslash to be used in a constant string, we should add backslash before them.

The next step is to build the program and invoke it. To create an OpenCL program, we use **clCreateProgramWithSource()** to create an OpenCL program object, and then we use **clBuildProgram()** to compile the program. To create an OpenCL kernel, we use **clCreateKernel()**, where we pass our constant string as one of parameters. To invoke the kernel, we use **clEnqueueNDRangeKernel()**, where we need to specify the number of work items and work groups.

To use 1-dimensional work items, we simply specify **n_workitem = 256** and **workgroup_size = 256**, which means in total we have 256 work items, and we group them in one work group. For a 2D distribution of work items, we need to specify **n_workitem[2] = {16, 16}** and **workgroup_size[2] = {4, 4}**, which means in x-axis we have 16 work items and we group them into 4 work groups and so as the y-axis. For a 3D version, we specify **n_workitem[3] = {4, 8, 8}** and **workgroup_size[3] = {2, 4, 4}**, where we assign 4 work items in x-axis, 8 work items in y and z axis, and we group each dimension into 2 work groups.

Exercise 2 - Performing SAXPY using OpenCL

Explain how you solved the issue when the **ARRAY_SIZE** is not a multiple of the block size. If you implemented timing in the code, vary **ARRAY_SIZE** from small to large, and explain how the execution time changes between GPU and CPU.

First we set our block size as 256, which means in each work group we have 256 work items in total. The grid size and number of total work items are set as follows to ensure that we round up when the **ARRAY_SIZE** is not a multiple of the block size.

```
size_t workgroup_size = BLOCK_SIZE;
size_t n_workitem = (int)(ARRAY_SIZE / BLOCK_SIZE + 1) * BLOCK_SIZE;
```

We test the performance of the program varying the number of **ARRAY_SIZE** from 1,000 to 500,000, and the result shows as follows.

ARRAY_SIZE	Execution Time (CPU) / s	Execution Time (GPU) / s
1,000	0.000004	0.000025
10,000	0.000048	0.000023
100,000	0.000430	0.000025
500,000	0.000918	0.000026

** The program is tested on Google Colab with GPU Tesla K80.*

The execution time of the CPU version is almost linear with **ARRAY_SIZE**, while the GPU version remains roughly the same when **ARRAY_SIZE** changes, since each work item is responsible for one element of calculation. We also notice that when **ARRAY_SIZE** is rather small, the GPU version is slower, and we assume this is due to the overhead of invoking the kernel.

Exercise Bonus - OpenCL Particle Simulations

#Iteration = 100 with Telsa K80 .

Particles Block Size	10k	1M	10M
16	0.001760s	0.133916s	1.099921s
32	0.001749s	0.104260s	0.890304s
64	0.001733s	0.105564s	0.886555s
128	0.001143s	0.104932s	0.899111s
256	0.001990s	0.105107s	0.886717s
CPU	0.028031s	2.916675s	28.605453s

And the block_size 32 seems optimal as 64+requires more resources and overheads with little improvement. With the increasing number of particles, the GPU code gains a huge advantage in terms of performance because in the kernel particles can be updated in parallel by each thread while in the CPU code, we have to traverse each particle and update them one by one.

From assignment 3, we've already known that proper asynchronous data transfer can improve the parallelism as well as but I don't know if that is supported in OpenCL for it must handle those devices without such built-in features. But here, since there is no requirement for transfer back to the host every iteration, if added some constraints between the GPU and CPU execution, it surely would become slower.

And compared to the performance with pure Cuda, this result is a little worse as we expected. Because OpenCL should be some combination of calls of Cuda when running on an Nvidia GPU and the driver itself has overhead, it is likely to have some efficiency loss during execution.