Applied GPU Programming - Assignment IV

DD2360 HT20

Pontus Asp

December 13, 2020

1 Git repository

I uploaded my git repository to GitHub. I use the same git repository for the entire course but the folder structure requested is still followed under the root folder. I also have 2 extra directories, one for this report and one where I have code from lectures. I also was not sure if Exercise 4 should have been separate or included in this report so I did both, the standalone Exercise_4.pdf can be found under the ex_4 folder.

Here is the link to my git repository:
`https://github.com/pontusasp/kth-dd2360/tree/master/Assignment_4`

2 Exercise 1

The first thing I did when extending the template was writing a `helloworld` kernel in the char array `mykernel`. The reason for why I wrote the kernel is probably pretty self explanatory, but why it is in a char array might be more confusing. The kernel is written in a string since it will (can) be compiled at runtime by OpenCL, so that the kernel is compiled to be able to be used on the devices on the host system. Therefore it is in a char array so that the source code can be passed to the OpenCL API to be compiled and set up.

After writing the kernel I started writing code in the `main` function between the comments specifying where to insert my code. The first thing I added here was a call to the `clCreateProgramWithSource` which I passed some arguments to, and the two most significant ones where my context and a pointer to my source code. What happens next is that `clCreateProgramWithSource` loads the program source code and stores it in a program object, which also gets associated with the OpenCL context I gave it, and then returns it.

However the program is not ready to be used yet, so what I did next was calling `clBuildProgram` which I passed the program object to. This function will then compile (and link) my program and update my program object.

After this point you might think that your program is ready to be executed, but one more API call is needed. To be honest, I could not find information of why exactly this step is needed but I am assuming that `clBuildProgram` simply compiles my executable program and the next step I did, `clCreateKernel` will take this program and prepare it to be launched on my OpenCL device, and also tell OpenCL the starting point of my program.

Now when my program, or should I say kernel, is compiled and loaded we need to launch it on the device. When launching the kernel we need to specify the size of our working groups and the number of work items

we want to use (on a GPU a work item is a thread). We do all this with `clEnqueueNDRangeKernel` which we also will give our command queue and kernel to, along with the number of work groups and items, and also in what dimension we want to compute with. In our case, we where expected to use three dimensions, and decide our own group and item sizes. I went with groups of dimension `1x1x1` and set my working items to 4 in all 3 axis, so `4x4x4`. This should yield a result of $4^3 = 64$ prints, which is it.

When running a kernel on OpenCL it is running asynchronously from our code so the next thing I did was call `clFinish` and give it our command queue. This function simply waits for OpenCL to finish all the commands that we have queued and then returns control to our code. We do this so that we will not exit our program before OpenCL is finished.

3 Exercise 2

In this exercise we implemented a SAXPY program in OpenCL, I also implemented the optional time measurements. I solved the issue of `ARRAY_SIZE` (which in my code is called `VSIZE`, but I will be calling it `ARRAY_SIZE` here for simplicity) potentially not being a multiple of the block size by first getting how many block sizes could cover the entire array and then multiplying this with the block size again to get a number that is equal to, or larger than the `ARRAY_SIZE` that will be divisible by `BLOCK_SIZE`. Here is the formula: $\frac{ARRAY\_SIZE+BLOCK\_SIZE-1}{BLOCK\_SIZE} \cdot BLOCK\_SIZE$. Take into account that this works thanks to integer division, which gets rid of all decimals after the first division.

Since the device I use with OpenCL is a GPU, this is what I will the OpenCL device as. When comparing the execution time on the CPU vs the GPU with varying array sizes I found something interesting on this particular exercise. The total performance did not vary much with GPU vs CPU. When looking closer I however found that the reason was that by a large majority the time spent on the GPU was transferring data to and from the device memory. When measuring time a bit more in-depth I could see that the actual kernel execution by far outperformed the CPU execution on big array sizes. On small array sizes the kernel did however not do as well as the CPU, and also considering that the GPU has the penalty of memory transfer it was much slower than the CPU variant on small sizes. On very large arrays the GPU variant did start outperforming the CPU version but to my surprise not by much.

Here is a partial output from one run with an array size of 80000000 (80M) and one run with the size of 10000 (10k):

3.1 80M Run:

```
Computing SAXPY on the GPU...
    Computation done in   4.487000 ms
    Memory transfer:
        To Device:        55.451000 ms
        From Device:      87.244000 ms
    Total time:           147.182000 ms

Computing SAXPY on the CPU...
    Computation done in   155.476000 ms
```

3.2 10k Run:

```
Computing SAXPY on the GPU...
    Computation done in   0.014000 ms
    Memory transfer:
        To Device:        0.121000 ms
        From Device:      0.022000 ms
    Total time:           0.157000 ms

Computing SAXPY on the CPU...
    Computation done in   0.019000 ms
```

4 Bonus Exercise

5 Exercise 4