Behrad Hemati
11849389

# Assignment 3: OpenCL

In this assignment, we want to preferably use GPUs to accelerate our task, although OpenCL can run both on GPUs and CPUs but since we are dealing with floating point arithmetic, it is much preferable if we could potentialize the GPUs.
The code on alma is in ~/pap/ex3
Compile with:
**g++ -O2 -I/usr/local/cuda-11/targets/x86_64-linux/include/ -L/usr/local/cuda-11/targets/x86_64-linux/lib/ -o ocl_simple ocl_simple.cpp -lOpenCL**

Or using the **makefile**.
There also exists another version **ocl_interactive.cpp** which allows user input from the terminal to change the matrix size, as will be explained this version has a different logic and uses a 1D matrix to control the indices.

## Code Explanation

Initially I had written the code with user input, but after it was clarified in the forums that there does not have to be a user input, I have made two different versions, and I will explain the normal version here which has no user input.
It should be noted the two different versions have a slightly different logic, since the indexing gets convoluted for the user input version and as a result I had to make the matrix that is used in the OpenCL part a 1D matrix, so I have full control on where in memory the data gets stored and be retrieved, this was necessary to compare the final results to make sure that they were similar (note that they are not exactly similar, and I have provided a counter which shows how many were not exactly matching).
Also of note is that the OpenCL program, kernel, device etc. are also not explained here since they are almost identical in between programs, and I have simply used the version that was mentioned in the course already, so I do not believe that it would need more explanation.
Now I will proceed to explain how the code for the simple version works.

## The Kernel

```
__kernel void iterVectors(__global float *B) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    int z = get_global_id(2);
    int szx = get_global_size(0);
    int szy = get_global_size(1);
    int szz = get_global_size(2);
    // const int idx = x + y * szx + z * szx * szy;
```

Behrad Hemati
11849389

```
    // x * szy * szz + y * szz + z
    const int idx = x * szy * szz + y * szz + 1;
    const int idx1 = (x+1) * szy * szz + y * szz + 0;
    const int idx2 = (x-1) * szy * szz + y * szz + 2;
    float res = B[idx];
    // if (z == 1 )
    if( x > 0 && x < szx-1)
    {
        if (z == 1)
        {
            for (int t=0;t<24;t++)
            {
                res += rsqrt(B[idx1]+B[idx2]);
                // res += 1 / sqrt(B[idx1]+B[idx2]);
            }

        }
        // barrier(CLK_GLOBAL_MEM_FENCE);
        B[idx] = res;
    }
```

In this kernel, x, y and z represent the width, height and depth of the matrix as indicated in the host program. In addition, I have also used get_global_size() in order to catch the length of each of those dimensions, and so it is not actually necessary to pass those as extra variables to the kernel.
Idx represents the current index we plan to write to and idx1 and idx2 are the indices that we need to access. For the reverse square  I am using the rsqrt() function that is provided by the OpenCL library.

## The Host

```
        // ############## ITER PART ##################
        size_t globalWorkSize2[3] = { SIZE, SIZE, DEPTH};
        size_t localWorkSize2[3] = {32,32,1};
        ret = clEnqueueWriteBuffer(commandQueue, bMat, CL_TRUE, 0, SZ, B,
0, NULL, NULL);
        cl_kernel kernel = clCreateKernel(program, "iterVectors", &ret);
        ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&bMat);
        ret = clEnqueueNDRangeKernel(commandQueue, kernel, 3, NULL,
globalWorkSize2, localWorkSize2, 0, NULL, NULL);
```

Behrad Hemati
11849389

```
        ret = clEnqueueReadBuffer(commandQueue, bMat, CL_TRUE, 0, SZ, B,
0, NULL, NULL);
```

As stated before, the initialization part of OpenCL is not included in the report, as it is commented from the original source, and is basically the same in every OpenCL Implementation. Here I will try to implement a 3D execution model for greater flexibility in local work sizes. Each global work size needs to be a multiple of local work size. Given the exercise, it seems that the best values for local work size would be {32,32,1} but to find out the best ones, we need to do some experiments.

## Results

| **Type** | Sequential | {1,1,1} | {16,16,3} | {16,16,1} | {32,32,1} |
|---|---|---|---|---|---|
| Time | 16.35 | 1.41 | 0.33 | 0.44 | 0.42 |

As can be seen from this table, the worst case when running on GPU is indeed {1,1,1} as each work.-group compromises of only one work -item in each dimension which predictably is the slowest of the bunch. My main curiosity was to compare 16 and 32, but as can be seen there is actually not a big difference between them for this particular code and device. In general with very little optimization we could reach from around **~11.6x** speed-up (which is already a very nice gain) to almost a factor of **~50**. Unfortunately I was unable to run it with {32,32,3} and so {16,16,3} remains the best performing. This shows that further optimization must be possible, e.g. using local memory. But with the current indexing that is a rather difficult task to do. Furthermore the code also outputs how many values were not an exact match with the sequential version. This mismatch is usually **<5%** which is a good amount since it's calculating up to very high decimal points and it is expected that the values do no exactly match.