Software Studio - Project 1: Matrix Multiplication

Anastasia Riana 邱慧莉 - 109006234

Platform Name

```
se22p26@SE22:~$ clinfo
Number of platforms                               1
  Platform Name                                   NVIDIA CUDA
  Platform Vendor                                 NVIDIA Corporation
  Platform Version                                OpenCL 3.0 CUDA 11.3.55
  Platform Profile                                FULL_PROFILE
  Platform Extensions                             cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64 cl_khr_3d_image_writes cl_khr_byte_addre
ssable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll cl_nv_
copy_opts cl_nv_create_buffer cl_khr_int64_base_atomics cl_khr_int64_extended_atomics cl_khr_device_uuid
  Platform Host timer resolution                  0ns
  Platform Extensions function suffix             NV
```

Device Name

```
Number of devices                                 1
  Device Name                                     NVIDIA GeForce GTX 1080 Ti
  Device Vendor                                   NVIDIA Corporation
  Device Vendor ID                                0x10de
  Device Version                                  OpenCL 3.0 CUDA
  Driver Version                                  465.19.01
  Device OpenCL C Version                         OpenCL C 1.2
  Device Type                                     GPU
  Device Topology (NV)                            PCI-E, 01:00.0
  Device Profile                                  FULL_PROFILE
  Device Available                                Yes
  Compiler Available                              Yes
  Linker Available                                Yes
  Max compute units                               28
  Max clock frequency                             1683MHz
  Compute Capability (NV)                         6.1
  Device Partition                                (core)
    Max number of sub-devices                     1
    Supported partition types                     None
    Supported affinity domains                    (n/a)
  Max work item dimensions                        3
  Max work item sizes                             1024x1024x64
  Max work group size                             1024
  Preferred work group size multiple              32
  Warp size (NV)                                  32
```

Implementation of Matrix Multiplication (Host Code)

I did the assignment based on the vecAdd.cpp, provided by the TAs and from that example, I learned that there are several steps to do kernel programming. In order to do that, to make things easier to understand, we are going to divide it into multiple steps.

Step 1: Declare All Needed Variables

Before we start the steps, we need to declare a few variables that we need, such as ***MatA, MatB, MatAns, countDevice, error, platformID, device, context, commands, program,*** and ***kernel.*** We need to declare all these variables so that it will be compatible with OpenCL.

Step 2: Identifying Platform and Devices

Functions of OpenCl returns an error code, hence we create an error variable to store those values. First, using ***clGetPlatformID***, we need to list the platform IDs and the number of different host

systems. If it fails to get any platform, it will print an error message: 'No platform found!' Next, we choose one of the platforms and proceed with it. Then, we collect the devices list, using ***clGetDevices,*** so we can check and select what devices can run on that selected platform. If it fails to get any devices, it will print an error message: 'No device found!'.

Step 3: Creating a Context and a Command Queue

In OpenCl, we need to create a context with ***clCreateContext***, which is a group of one or more devices that will be executing commands. We need to group devices so that it would be easier for each device to receive its own command queue, created with ***clCreateCommandQueueWithProperties*** and it will be able to share the same memory. If a device has different contexts, it will not have this ability.

Step 4: Opening the OpenCl Kernel

We first create the variable ***myFile*** and we use another command called ***.open(),*** to open our kernel file . If it cannot read the file, we should display an error message. If it succeeds, then we close the file using ***.close()***. If it fails to open the .cl file, it will print an error message: 'Cannot open the file!'.

Step 5: Creating, Building a Program, and Kernel

Due to the context consisting of unique devices, then a program needs to be built, with ***clCreateProgramWithSource,*** on the specific context mentioned in Part 2. Then, after that program is built, we also need to make sure that all the lists of devices are compatible, hence the program that we built with ***clCreateProgramWithSource,*** must be built for each device as it may have different architectures. If it fails to create a program, it will print an error message: . In this case, we use the function ***clBuildProgram.*** If it fails to do so, it will print an error message: 'Error building program!'. By then, we can start to create the kernel, using the ***clCreateKernel*** command. If it fails to create a kernel, then it will print an error message: 'No kernel found!'.

Step 6: Creating a Buffer and Passing It to the Kernel

Next, we need to create a buffer, with ***clCreateBuffer***, as it is an array that the kernel will be returning and the size of that array that we are going to return is specified by computing the memory size of each matrix. Then, we set all the arguments that the kernel is going to take, with ***clSetKernelArg***, which passes it from the .cl file to the .cpp file.

Step 7: Run the kernel and Reading the Result

Then, we will try to run uif the kernel is working correctly or not. We use the command ***clEnqueueNDRangeKernel***, to enqueue a command to execute the program on the selected program and device we already chose in Part 1. If it doesn't successfully run, it will print an error message: . Then, we use ***clEnqueueReadBuffer***, to receive the result from the kernel. If it doesn't receive the final result, it will print an error message: .

Step 8: Check Correctness

Lastly, we need to check the correctness of the code. Using a for-loop, we check if the result of sequential matrix multiplication values, stored in the **gt** array, is equal to the **ans** array. Using the sequential matrix multiplication, we store the result in the **gt** array and using parallelization, the

answer is stored in the **ans** array. If it's equal, we will see a message 'AC' to indicate that the program is correct. Else, it will print 'Failed.'

Implementation of Matrix Multiplication (Kernel Code)

We name the kernel as **matrixMul** and it will take the global matrices A, B, ans alongside X, Y, and Z as the kernel function parameter. Inside the body of the function, we need to take coordinates and we do that by using **get_global_id()** function. We assign the **x_axis** with the value 0 and **y_axis** with the value 1.

Next, we create a new variable called **value**, where we can store the result of the multiplication for each value of matrix A and B. Then, inside a for-loop, we append the value of the dot product to the variable value, like: ***value += A[y_axis*Y+i] * B[i*Z+x_axis]***. The way the indices work is that we want to store the 2D input matrix to a single dimension array because in a physical memory, everything is linear. Then, we can assign the value variable to matrix ans with indices in linear fashion, like such: ***ans[y_axis*Y + x_axis] = value***.