PDC – ASSIGNMENT # 03

OPEN-CL

ABSTRACT

This Assignment Focusses on the usage of Open CL to execute Code on CPUs and GPU. Sequential codes have also been written in order to compare and visualize the execution times of same code with different modes.

Muhammad Huzaifa

20I-0604

Contents

OpenCL-Kernel-Execution 2
Setting up OpenCL on Docker 2
    Task # 01 Matrix Multiplication Using CPU 2
        Graphic Visualization: 7
    Task # 02 Merge Sort Using CPU 8
    Visualizing the Results 9

OpenCL-Kernel-Execution

This repository contains multiple codes executing using OpenCl on CPU & GPU. Sequential Codes for each program executed on CPU and GPU has also been written to compare the execution time of all from the same problem category.

Setting up OpenCL on Docker

```
- docker-compose up -d master --build
- docker exec -it dockercl-master-1 bash
```

- or just run `script.bat` in terminal and get started

After getting a nice bash shell to the container traverse into the folder you need to check for executon and follow the following commands.

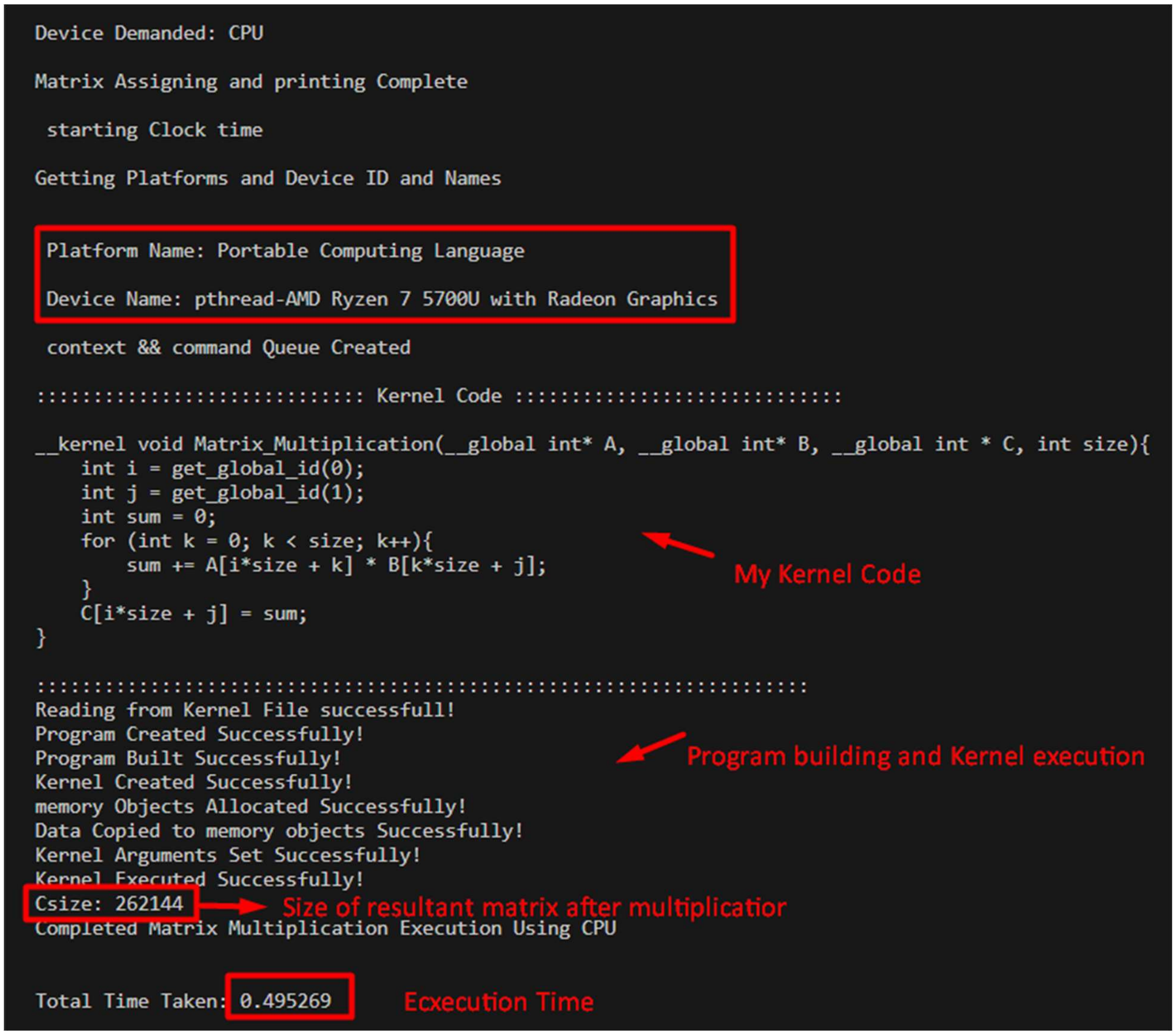
Task # 01 Matrix Multiplication Using CPU

Assuming Matrix Multiplication is the desired folder

```
cd Matrix Multiplication
gcc Matrix_Mul_CPU.c -o matmul -lOenCL
./matmul <mode of execution> i.e., ./matmul CPU
```

After running the executable via accurate command line arguments you will get something like this:

CPU based Kernel Execution



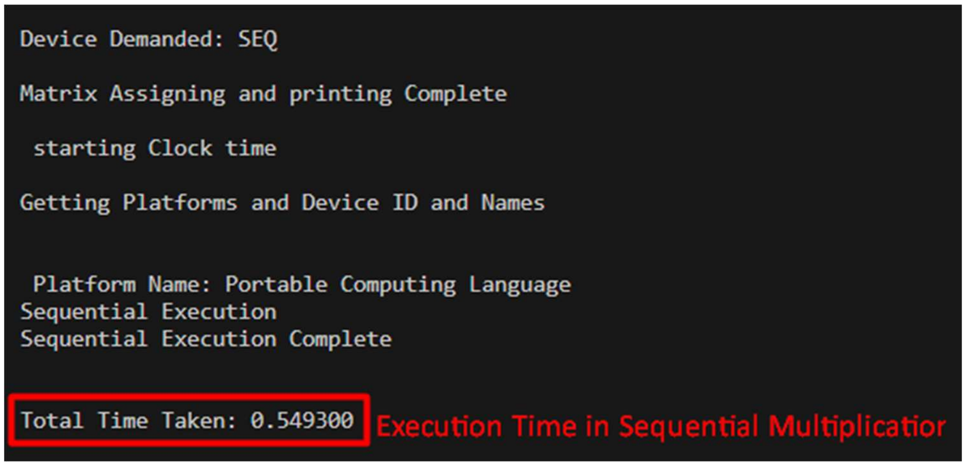
```
Device Demanded: CPU

Matrix Assigning and printing Complete

 starting Clock time

Getting Platforms and Device ID and Names

 Platform Name: Portable Computing Language

 Device Name: pthread-AMD Ryzen 7 5700U with Radeon Graphics

 context && command Queue Created

:::::::::::::::::::::::::::::: Kernel Code :::::::::::::::::::::::::::::

__kernel void Matrix_Multiplication(__global int* A, __global int* B, __global int * C, int size){
    int i = get_global_id(0);
    int j = get_global_id(1);
    int sum = 0;
    for (int k = 0; k < size; k++){
        sum += A[i*size + k] * B[k*size + j];
    }
    C[i*size + j] = sum;
}

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Reading from Kernel File successfull!
Program Created Successfully!
Program Built Successfully!
Kernel Created Successfully!
memory Objects Allocated Successfully!
Data Copied to memory objects Successfully!
Kernel Arguments Set Successfully!
Kernel Executed Successfully!
Csize: 262144
Completed Matrix Multiplication Execution Using CPU


Total Time Taken: 0.495269
```

My Kernel Code
Program building and Kernel execution
Size of resultant matrix after multiplicatior
Ecxecution Time

As you can see in the screenshot above, the name of the platform and the name of the CPU device used for the execution of kernel code responsible for the multiplication of 2 matrices.

Sequential Code Execution

Similarly, Let's try and run treh sequential code and check for its execution time.



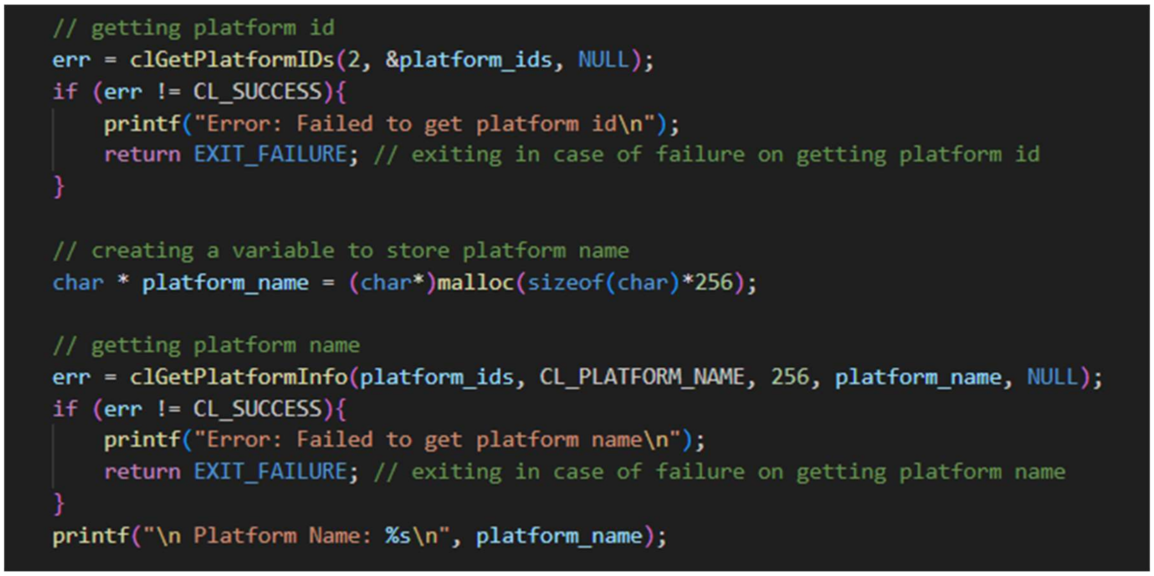
```
Device Demanded: SEQ

Matrix Assigning and printing Complete

 starting Clock time

Getting Platforms and Device ID and Names


 Platform Name: Portable Computing Language
Sequential Execution
Sequential Execution Complete

Total Time Taken: 0.549300
```

As you would have already notice that the execution time taken by a sequential code is rather high then the execution time taken by the CPU kernel code.

Implementation

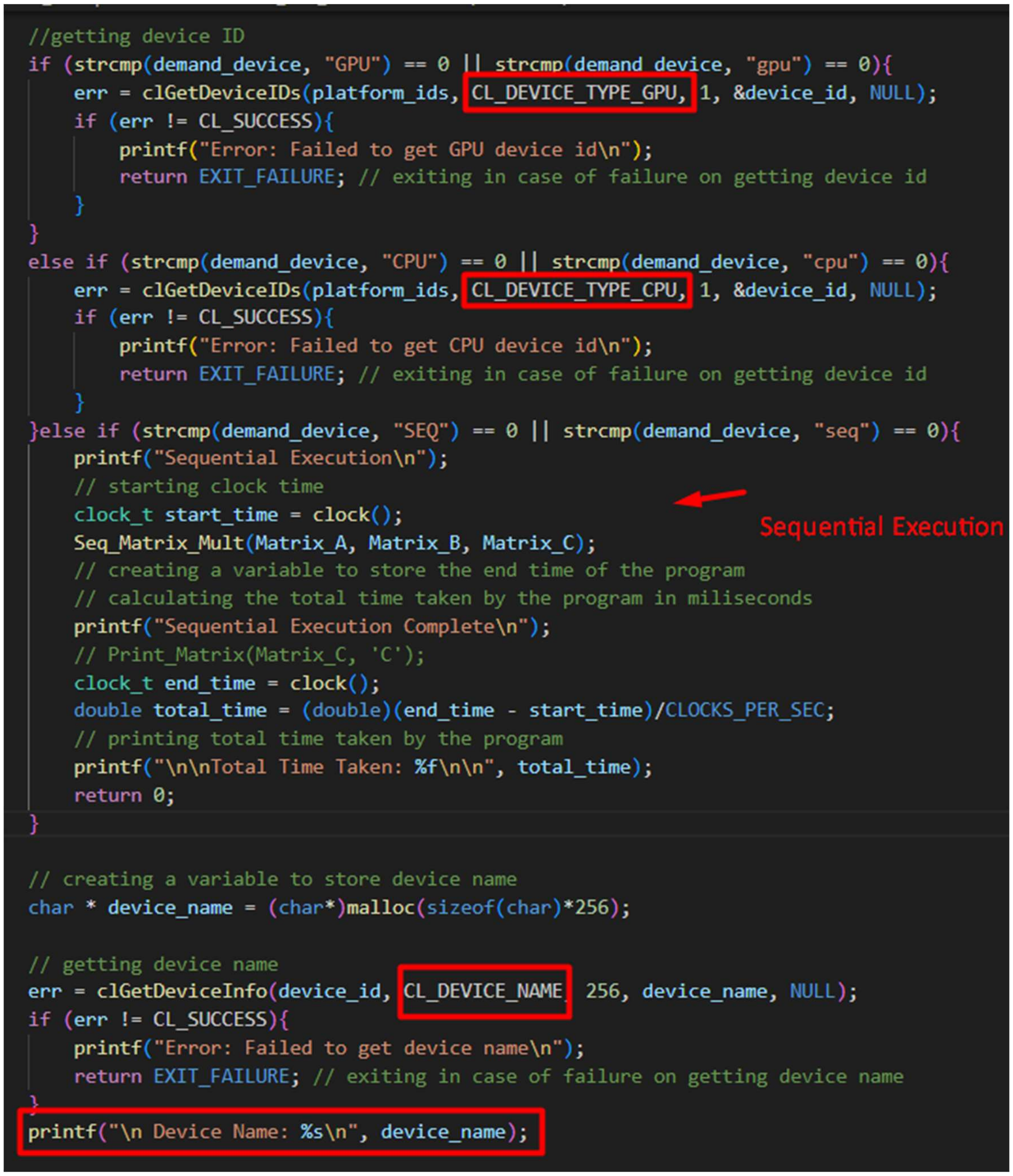
Foe the implementation of the CPU based kernel execution, thorough steps have been taken to ensure proper and smooth execution. Here are the highlights of the code:

- Getting Platform ID and Platform name



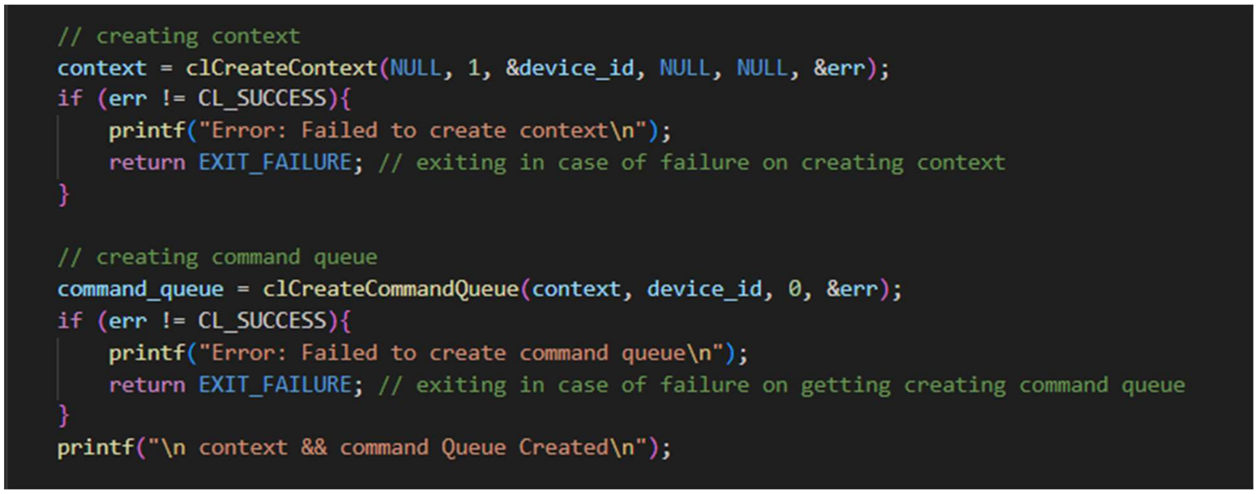
```
// getting platform id
err = clGetPlatformIDs(2, &platform_ids, NULL);
if (err != CL_SUCCESS){
    printf("Error: Failed to get platform id\n");
    return EXIT_FAILURE; // exiting in case of failure on getting platform id
}

// creating a variable to store platform name
char * platform_name = (char*)malloc(sizeof(char)*256);

// getting platform name
err = clGetPlatformInfo(platform_ids, CL_PLATFORM_NAME, 256, platform_name, NULL);
if (err != CL_SUCCESS){
    printf("Error: Failed to get platform name\n");
    return EXIT_FAILURE; // exiting in case of failure on getting platform name
}
printf("\n Platform Name: %s\n", platform_name);
```

- Setting Device Type, (In case of Sequential) & Getting Device Name



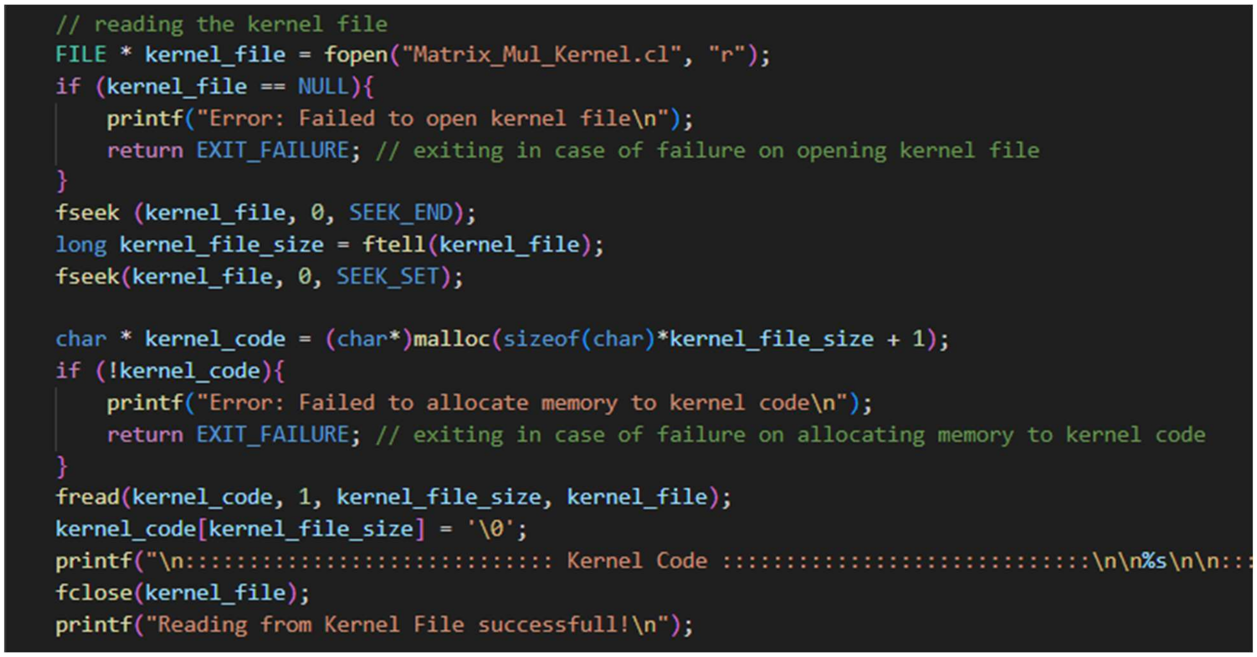
```
//getting device ID
if (strcmp(demand_device, "GPU") == 0 || strcmp(demand_device, "gpu") == 0){
    err = clGetDeviceIDs(platform_ids, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    if (err != CL_SUCCESS){
        printf("Error: Failed to get GPU device id\n");
        return EXIT_FAILURE; // exiting in case of failure on getting device id
    }
}
else if (strcmp(demand_device, "CPU") == 0 || strcmp(demand_device, "cpu") == 0){
    err = clGetDeviceIDs(platform_ids, CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
    if (err != CL_SUCCESS){
        printf("Error: Failed to get CPU device id\n");
        return EXIT_FAILURE; // exiting in case of failure on getting device id
    }
}else if (strcmp(demand_device, "SEQ") == 0 || strcmp(demand_device, "seq") == 0){
    printf("Sequential Execution\n");
    // starting clock time
    clock_t start_time = clock();
    Seq_Matrix_Mult(Matrix_A, Matrix_B, Matrix_C);
    // creating a variable to store the end time of the program
    // calculating the total time taken by the program in miliseconds
    printf("Sequential Execution Complete\n");
    // Print_Matrix(Matrix_C, 'C');
    clock_t end_time = clock();
    double total_time = (double)(end_time - start_time)/CLOCKS_PER_SEC;
    // printing total time taken by the program
    printf("\n\nTotal Time Taken: %f\n\n", total_time);
    return 0;
}

// creating a variable to store device name
char * device_name = (char*)malloc(sizeof(char)*256);

// getting device name
err = clGetDeviceInfo(device_id, CL_DEVICE_NAME, 256, device_name, NULL);
if (err != CL_SUCCESS){
    printf("Error: Failed to get device name\n");
    return EXIT_FAILURE; // exiting in case of failure on getting device name
}
printf("\n Device Name: %s\n", device_name);
```

- Creating Context and Command Queues



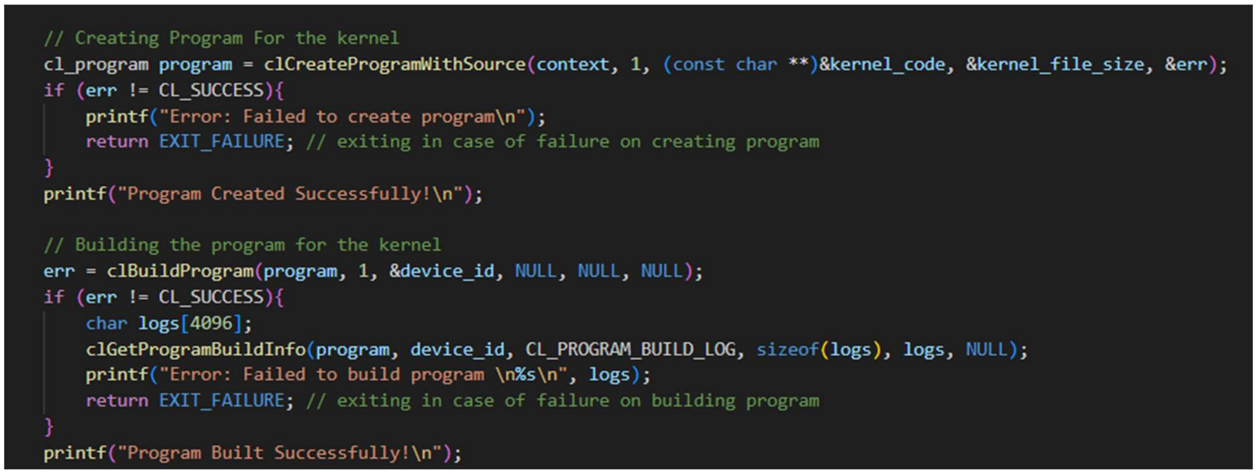
```
// creating context
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
if (err != CL_SUCCESS){
    printf("Error: Failed to create context\n");
    return EXIT_FAILURE; // exiting in case of failure on creating context
}

// creating command queue
command_queue = clCreateCommandQueue(context, device_id, 0, &err);
if (err != CL_SUCCESS){
    printf("Error: Failed to create command queue\n");
    return EXIT_FAILURE; // exiting in case of failure on getting creating command queue
}
printf("\n context && command Queue Created\n");
```

- Reading Kernel Code from Kernel File This has two ways, you can either use a const char * string to store the code that kernel is going to execute and pass this to create kernel and program. Or you can create a saperate kernel file and read it to pass kernel code to create kernel and program. In my opinion the later is a better option.



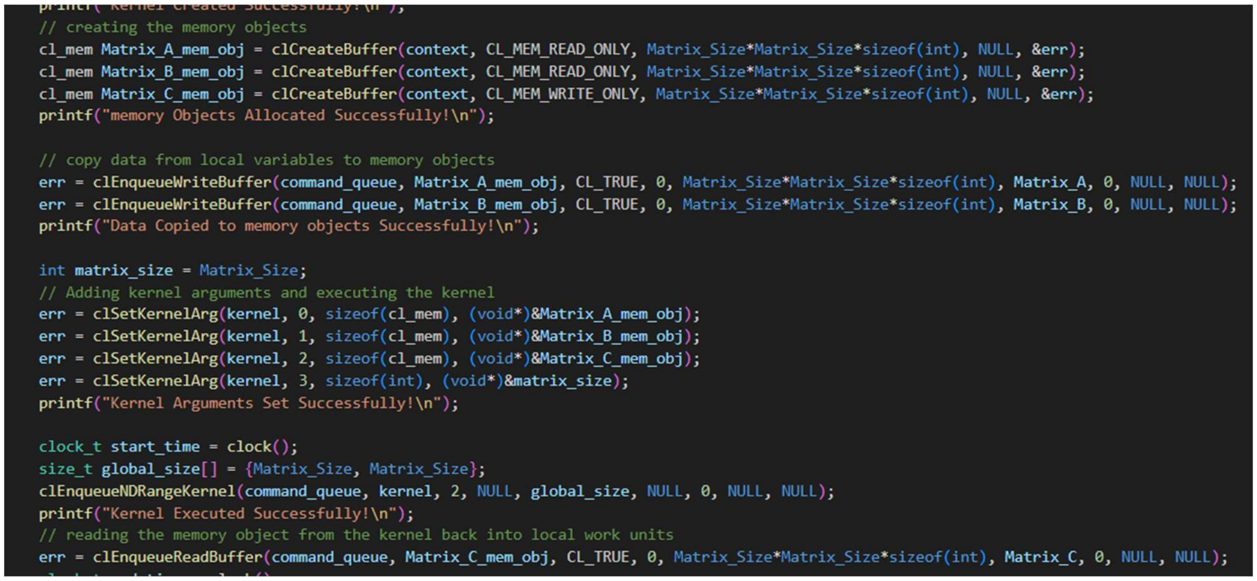
```
// reading the kernel file
FILE * kernel_file = fopen("Matrix_Mul_Kernel.cl", "r");
if (kernel_file == NULL){
    printf("Error: Failed to open kernel file\n");
    return EXIT_FAILURE; // exiting in case of failure on opening kernel file
}
fseek (kernel_file, 0, SEEK_END);
long kernel_file_size = ftell(kernel_file);
fseek(kernel_file, 0, SEEK_SET);

char * kernel_code = (char*)malloc(sizeof(char)*kernel_file_size + 1);
if (!kernel_code){
    printf("Error: Failed to allocate memory to kernel code\n");
    return EXIT_FAILURE; // exiting in case of failure on allocating memory to kernel code
}
fread(kernel_code, 1, kernel_file_size, kernel_file);
kernel_code[kernel_file_size] = '\0';
printf("\n:::::::::::::::::::::::::::: Kernel Code ::::::::::::::::::::::::::::\n\n%s\n\n::
fclose(kernel_file);
printf("Reading from Kernel File successfull!\n");
```

- Create & Build Program



```c
// Creating Program For the kernel
cl_program program = clCreateProgramWithSource(context, 1, (const char **)&kernel_code, &kernel_file_size, &err);
if (err != CL_SUCCESS){
    printf("Error: Failed to create program\n");
    return EXIT_FAILURE; // exiting in case of failure on creating program
}
printf("Program Created Successfully!\n");

// Building the program for the kernel
err = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
if (err != CL_SUCCESS){
    char logs[4096];
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(logs), logs, NULL);
    printf("Error: Failed to build program \n%s\n", logs);
    return EXIT_FAILURE; // exiting in case of failure on building program
}
printf("Program Built Successfully!\n");
```

- Finally Load Memory arguments, Execute Kernel and Store the Returned global memory object for resultant into your work unit’s local memory object.



```c
printf("Kernel Created Successfully!\n");
// creating the memory objects
cl_mem Matrix_A_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, Matrix_Size*Matrix_Size*sizeof(int), NULL, &err);
cl_mem Matrix_B_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, Matrix_Size*Matrix_Size*sizeof(int), NULL, &err);
cl_mem Matrix_C_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, Matrix_Size*Matrix_Size*sizeof(int), NULL, &err);
printf("memory Objects Allocated Successfully!\n");

// copy data from local variables to memory objects
err = clEnqueueWriteBuffer(command_queue, Matrix_A_mem_obj, CL_TRUE, 0, Matrix_Size*Matrix_Size*sizeof(int), Matrix_A, 0, NULL, NULL);
err = clEnqueueWriteBuffer(command_queue, Matrix_B_mem_obj, CL_TRUE, 0, Matrix_Size*Matrix_Size*sizeof(int), Matrix_B, 0, NULL, NULL);
printf("Data Copied to memory objects Successfully!\n");

int matrix_size = Matrix_Size;
// Adding kernel arguments and executing the kernel
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&Matrix_A_mem_obj);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&Matrix_B_mem_obj);
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&Matrix_C_mem_obj);
err = clSetKernelArg(kernel, 3, sizeof(int), (void*)&matrix_size);
printf("Kernel Arguments Set Successfully!\n");

clock_t start_time = clock();
size_t global_size[] = {Matrix_Size, Matrix_Size};
clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global_size, NULL, 0, NULL, NULL);
printf("Kernel Executed Successfully!\n");
// reading the memory object from the kernel back into local work units
err = clEnqueueReadBuffer(command_queue, Matrix_C_mem_obj, CL_TRUE, 0, Matrix_Size*Matrix_Size*sizeof(int), Matrix_C, 0, NULL, NULL);
```

Note

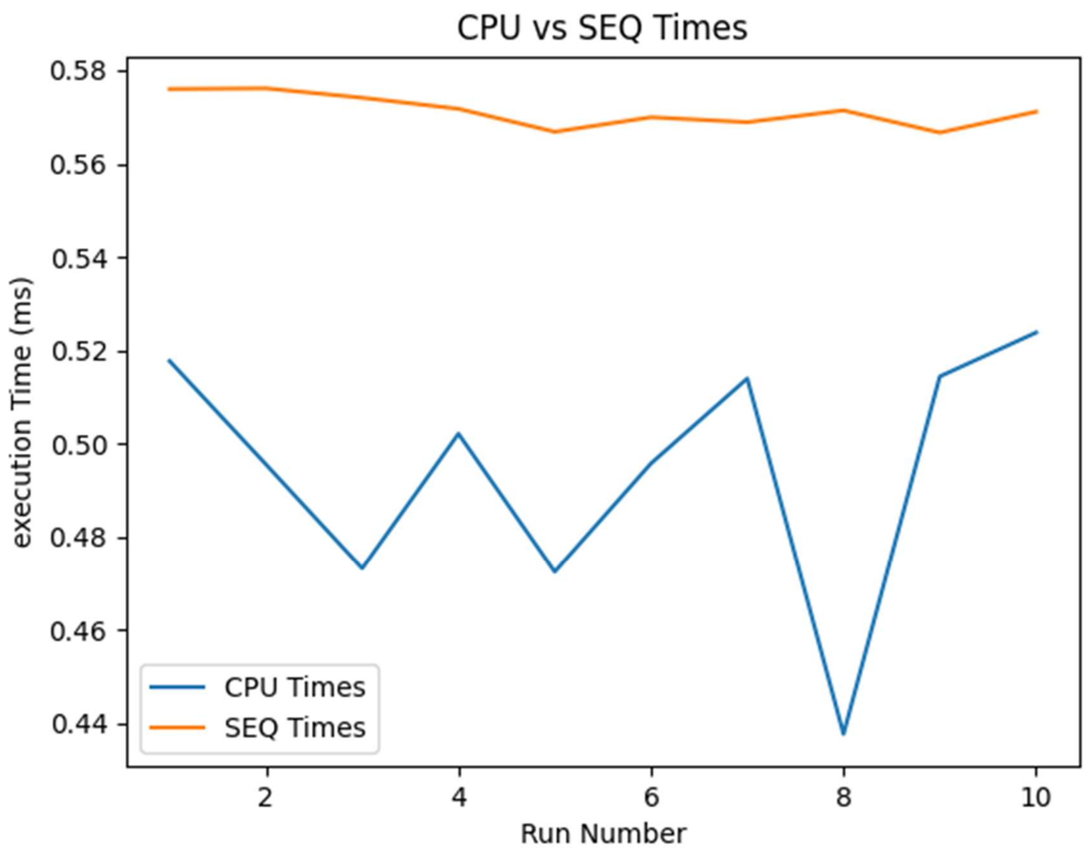
- you can uncomment the matrix printing codes on line # 82, 83, 272 to print the input and resultant matrices to observe the output better.

Graphic Visualization:

How to Run:

```
cd Matrix Multiplication
pip3 install -r requirements.txt (U can search for system specific issues on
your own)
python3 graph.py
```

To visualize the difference in execution times of both CPU and SEQ matrix multiplication I executed the Matric_Mul_CPU.c file with both CPU and SEQ modes using python's subprocess. Then I extracted the execution times from teh output of both mode executions and stored in 2 different Arrays. Then I used matplotlib to plot the graph using both mode's execution time and here is teh final result.

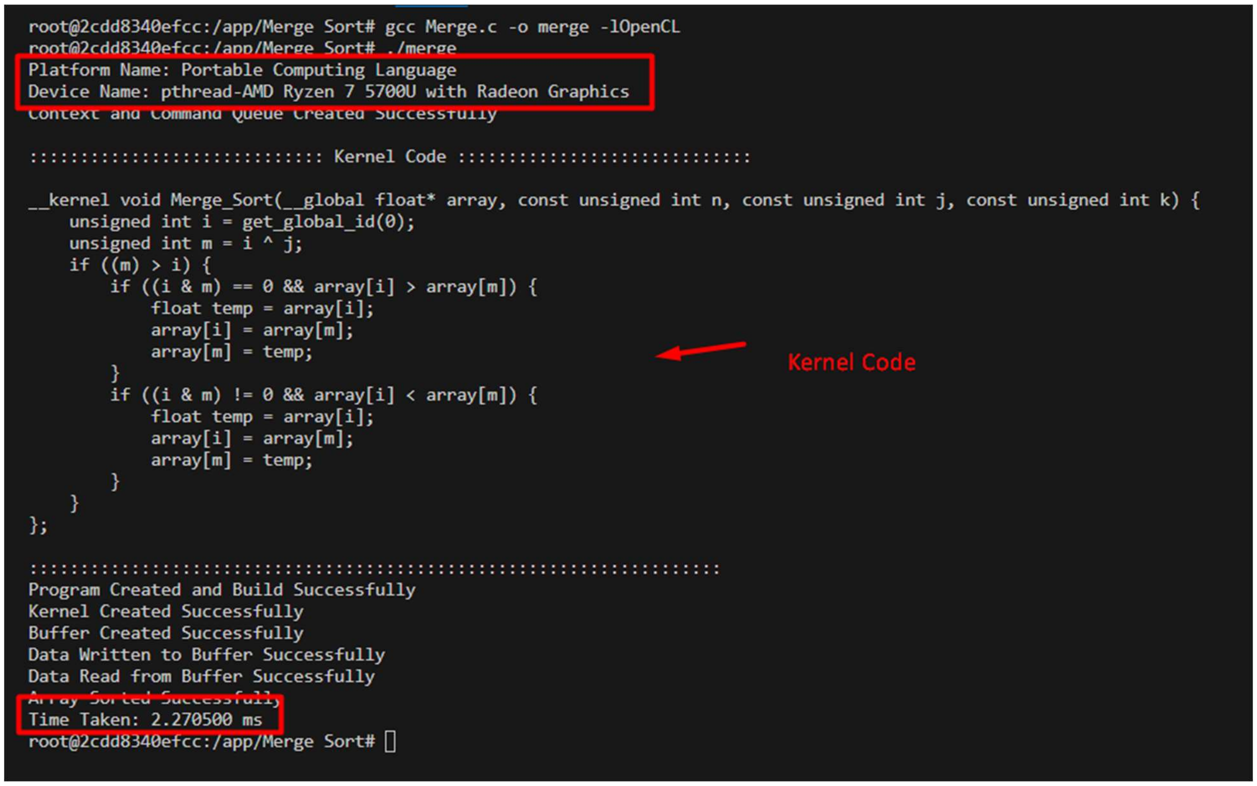


Task # 02 Merge Sort Using CPU

```
cd Merge Sort
gcc Merge.c -o merge -lOenCL
./merge
gcc SEQ_Merge_Sort.c -o sort
./sort
```

After running the executable you will get something like this:

CPU based Kernel Execution

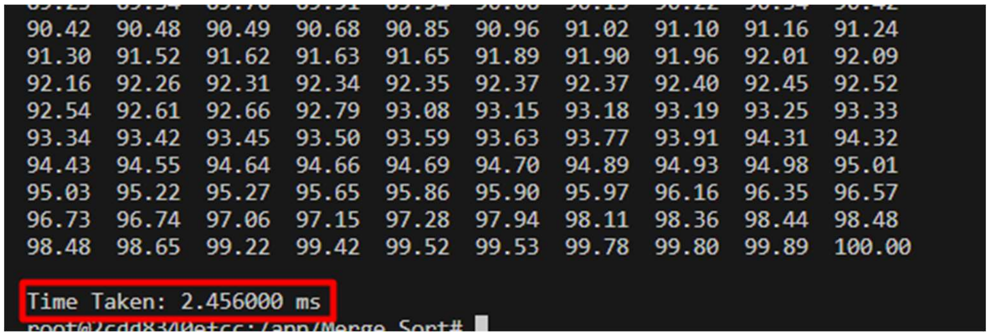


```
root@2cdd8340efcc:/app/Merge Sort# gcc Merge.c -o merge -lOpenCL
root@2cdd8340efcc:/app/Merge Sort# ./merge
Platform Name: Portable Computing Language
Device Name: pthread-AMD Ryzen 7 5700U with Radeon Graphics
Context and Command Queue Created Successfully

:::::::::::::::::::::::::::: Kernel Code ::::::::::::::::::::::::::::

__kernel void Merge_Sort(__global float* array, const unsigned int n, const unsigned int j, const unsigned int k) {
    unsigned int i = get_global_id(0);
    unsigned int m = i ^ j;
    if ((m) > i) {
        if ((i & m) == 0 && array[i] > array[m]) {
            float temp = array[i];
            array[i] = array[m];
            array[m] = temp;
        }
        if ((i & m) != 0 && array[i] < array[m]) {
            float temp = array[i];
            array[i] = array[m];
            array[m] = temp;
        }
    }
};

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Program Created and Build Successfully
Kernel Created Successfully
Buffer Created Successfully
Data Written to Buffer Successfully
Data Read from Buffer Successfully
Array Sorted Successfully
Time Taken: 2.270500 ms
root@2cdd8340efcc:/app/Merge Sort# 
```

As you can see in the screenshot above, the name of the platform and the name of the CPU device used for the execution of kernel code responsible for the multiplication of 2 matrices.

Sequential Code Execution

Similarly, Let's try and run treh sequential code and check for its execution time.

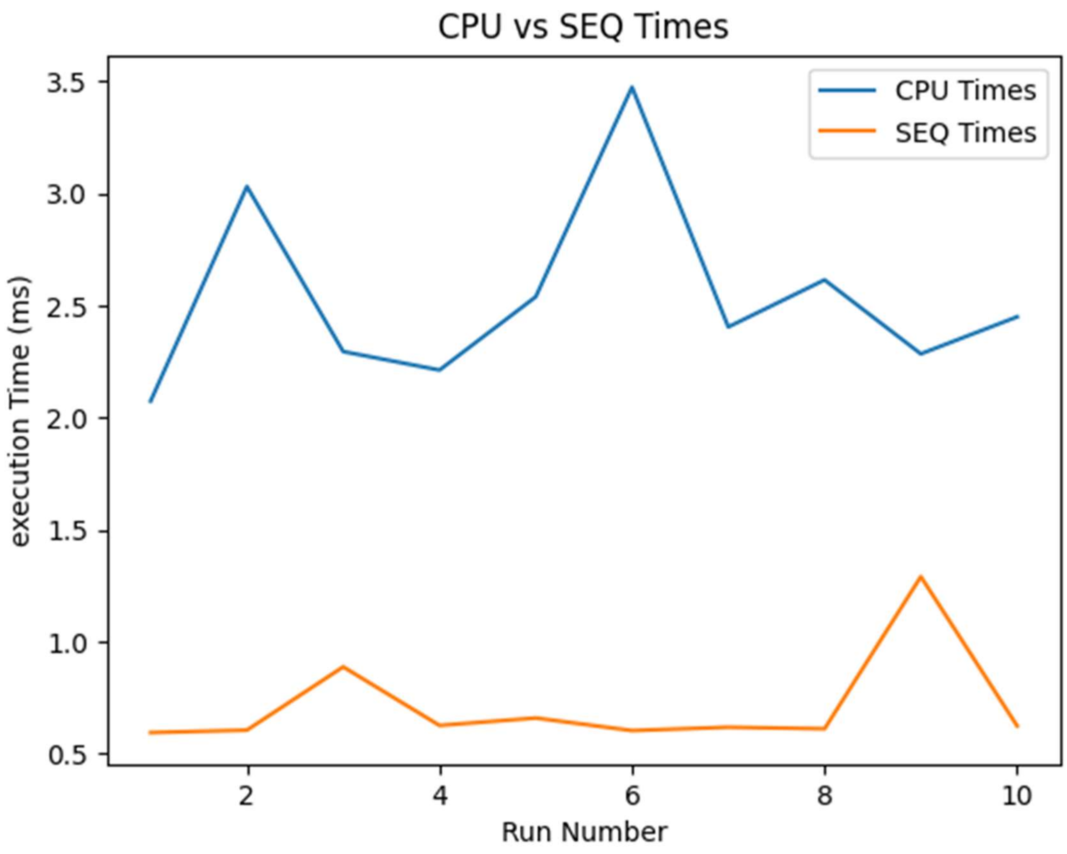


```
90.42  90.48  90.49  90.68  90.85  90.96  91.02  91.10  91.16  91.24
91.30  91.52  91.62  91.63  91.65  91.89  91.90  91.96  92.01  92.09
92.16  92.26  92.31  92.34  92.35  92.37  92.37  92.40  92.45  92.52
92.54  92.61  92.66  92.79  93.08  93.15  93.18  93.19  93.25  93.33
93.34  93.42  93.45  93.50  93.59  93.63  93.77  93.91  94.31  94.32
94.43  94.55  94.64  94.66  94.69  94.70  94.89  94.93  94.98  95.01
95.03  95.22  95.27  95.65  95.86  95.90  95.97  96.16  96.35  96.57
96.73  96.74  97.06  97.15  97.28  97.94  98.11  98.36  98.44  98.48
98.48  98.65  99.22  99.42  99.52  99.53  99.78  99.80  99.89  100.00

Time Taken: 2.456000 ms
root@2cdd8340efcc:/app/Merge Sort# 
```

As you would have already notice that the execution time taken by a sequential code is rather high then the execution time taken by the CPU kernel code.

Visualizing the Results

Following are teh visualized results of the execution times in CPU and Sequential code executions:


CPU vs SEQ Times
CPU Times
SEQ Times
execution Time (ms)
Run Number
3.5
3.0
2.5
2.0
1.5
1.0
0.5
2
4
6
8
10