

## Assignment 2 Report

In this assignment, I developed a C and OpenCL implementation for 100x100 matrix addition. The report's folder structure is shown in Figure 1. Both of the two different implementations have their own dedicated section in this report. More detailed information on the functions discussed in this report can be seen in the extensive commenting on the code files. The codes were created on Windows 10 with Visual Studio 2019. A NVIDIA GPU was used with OpenCL. Sources for different code snippets are displayed in the code files.

```
--CMatrixAddition
|
|   |-->CMatrixAddition.c
|   |-->MatrixFunctions.c
|   |-->MatrixFunctions.h
|
|--OpenCLMatrixAddition
|
|   |-->OpenCLMatrixAddition.c
|   |-->add_matrix.cl
|   |-->OpenCLFunctions.c
|   |-->OpenCLFunctions.h
|   |-->MatrixFunctions.c
|   |-->MatrixFunctions.h
```

### C Matrix Addition

The code for the C Matrix addition consists of three different files. *CMatrixAddition.c* defines the required *Add\_Matrix* function and *main()*. *MatrixFunctions.c* and its header file *MatrixFunctions.h* define the common matrix related functions *createMatrix()*, *printMatrix()*, and *freeMatrix()*.

The function names are very self explanatory, *createMatrix* creates a matrix of the specified size, *printMatrix* outputs the given matrix to the console, and *freeMatrix* frees the memory assigned to the given matrix. *Add\_Matrix* performs element-wise addition of two given matrices, and returns the resulting matrix.

Figure 1: Folder structure

It also prints the time elapsed during the addition to console. The elapsed time is obtained by utilizing WINAPI.

I designed the *CreateMatrix* function in a way that allows the user create matrices of any size, excluding 0. The matrices are created with the following command:

```
int* matrix;
matrix = malloc(matrix_size * matrix_size * sizeof(int));
They are then populated with random values with this piece of code:
for (i = 0; i < matrix_size; i++) {
    for (j = 0; j < matrix_size; j++) {
        matrix[i * matrix_size + j] = (rand() % (max - min + 1)) + min;
    }
}
```

Additionally, I tried two other ways of creating the matrices. They were:

```
int** matrix = malloc(matrix_size * sizeof(int));
for(i = 0; i < matrix_size; i++) matrix[i] = malloc(matrix_size * sizeof(int), sizeof(int));
and:
```

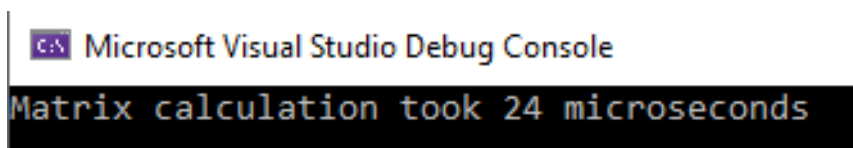
```
int (*matrix)[matrix_size] = malloc(matrix_size * matrix_size * sizeof(int));
```

However, these styles of matrix creation created problems in different parts of the implementations. The first one does not work with OpenCL Kernels, since they don't accept pointers-to-pointers (e.g. *int\*\**) as params. The second one works with OpenCL, but due to my limited experience with C, I was not able to get it to work with the pure C implementation of *Add\_Matrix()*.

The element wise addition of two matrices is performed like this:

```
for (i = 0; i < matrix_size; i++) {
    for (j = 0; j < matrix_size; j++) {
        result[i * matrix_size + j] = m1[i * matrix_size + j] + m2[i * matrix_size + j];
    }
}
```

When ran, the *Add\_Matrix()* also outputs the execution time. This is illustrated in Figure 2.



```
Microsoft Visual Studio Debug Console
Matrix calculation took 24 microseconds
```

Figure 2: C Implementation output

## OpenCL Implementation

The OpenCL code consists of six files. *MatrixFunctions* was explained in the previous section, *OpenCLFunctions.c* and *.h* define the OpenCL related functions *errorCheck()*, *loadKernel()*, and *printDeviceInfo()*. *OpenCLMatrixAddition.c* performs most of the OpenCL related actions in *main()*, and *add\_matrix.cl* defines the Kernel code for matrix addition.

I wanted that loading the Kernel source from a file would not clutter *main()*. Due to this, I created the function *loadKernel()* and the structure *kernel\_source*. When *loadKernel* is called with the Kernel files name, the file's contents is read and stored to the *kernel\_source* structure. The structure is then returned.

*errorCheck* is a collection of different OpenCL errors I ran into. The function takes the error number obtained from OpenCL functions and outputs the error codes name.

*PrintDeviceInfo()* takes a *cl\_device\_id* and outputs the devices name, type (GPU or CPU), and its OpenCL version.

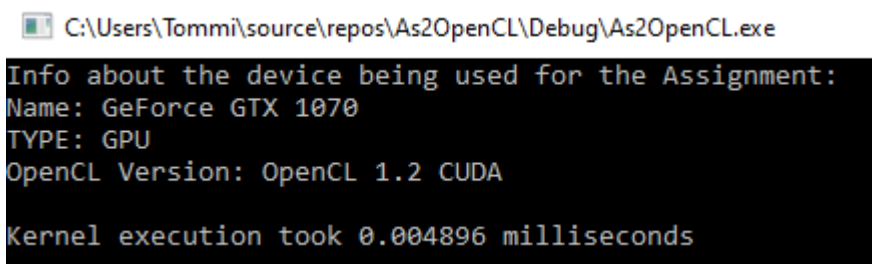
In *main()* the execution goes as follows:

1. Load Kernel file into memory
2. Obtain the first OpenCL platform
3. Obtain the first OpenCL device on the platform
4. Print the device's information
5. Create a command queue with profiling enabled
6. Create three matrices with *createMatrix()*
7. Create memory buffers for the matrices
8. Create and build the Kernel program
9. Set the matrices as arguments for the Kernel function
10. Execute the Kernel with *clEnqueueNDRangeKernel*
11. Obtain Kernel execution time with profiling
12. Read resulting matrix
13. Output elapsed time and finalize by freeing memory

The kernel code I used is:

```
__kernel void add_matrix(__global const int* m1, __global const int* m2, __global int* result) {  
    int id = get_global_id(0);  
    result[id] = m1[id] + m2[id];  
}
```

The output of the OpenCL implementation can be seen in Figure 3.



```
C:\Users\Tommi\source\repos\As2OpenCL\Debug\As2OpenCL.exe  
Info about the device being used for the Assignment:  
Name: GeForce GTX 1070  
TYPE: GPU  
OpenCL Version: OpenCL 1.2 CUDA  
Kernel execution took 0.004896 milliseconds
```

Figure 3: OpenCL implementation output

When calling the `clEnqueueNDRangeKernel` command, I set the global work size to `matrix_size * matrix_size` and the local work size to `matrix_size`. So if two 3x3 matrices are summed, something along the lines shown in Figure 4 will happen. In this case the matrices' values are divided into groups of three that are summed simultaneously, making execution much faster.

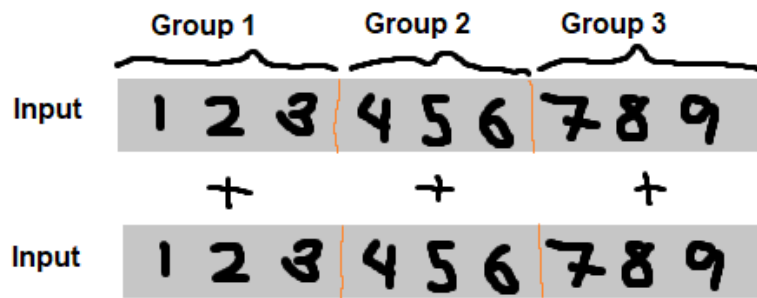


Figure 4: Kernel summing two 3x3 matrices

### Comparison between C and OpenCL

For matrices with the size 100x100, the OpenCL implementation was approximately 5 times faster. I decided to do a bit more comparison between the two. Thus, Figure 5 contains a graph showing the difference in elapsed time for `Add_Matrix()` for both implementations.

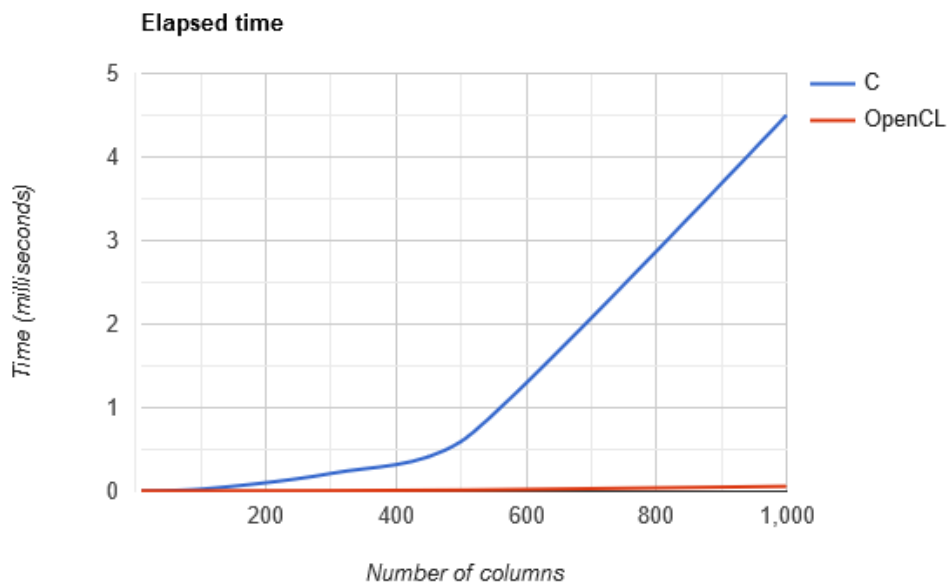


Figure 5: Elapsed time for both implementations