# University of Vienna
# Assignment 3: PAP OpenCL
# Report

**Name:** Zoheir El Houari
**Matriculation number**: a12044027

## *Introduction:*

The goal of the assignment is to transform our provided sequential version into an OpenCL parallel version and trying different optimization techniques.

## *1 - Code Structure:*

- **a3-parallel.cpp:** In this file, both the sequential and the OpenCL version are implemented in order to see the difference in the performance between the two version . Some helper functions were needed, ***InitializeMatrix()*** which initialize our matrices entries with the required values for both versions. ***Version_comparison()*** which verifies if the output results of the sequential and OpenCL are the same or not, with a margin error of (= 0.99). I execute the sequential version first and calculate it's runtime. The OpenCL host code is written directly in the main function in order to create clean code.
- **kernel.cl:** In this file the OpenCL kernel is written. The main parallelism happens here. The file has one kernel called "**MatriceTransformations**". The kernel takes 3 arguments, they are the matrix A which serve as our M and the matrix B that serve as our tmp_M and the SIZE. These parameters are accessed through the global memory. My constant is the matrix *A, which is why it's type is set to **CL_MEM_READ_ONLY** during the memory buffer creation, because we don't manipulate the entries inside the kernel, I just update it inside the iteration for loop and after that I pass it as a

parameter to the kernel for the next iteration. On the other hand the matrix *B has a type **CL_MEM_READ_WRITE** during the memory buffer creation, since it is required to updated it's entries inside the kernel to get the required results. The global id was accesses with the help of the function **"get_global_id(0)",** this allow us to access the array's indices to apply the described formula.

## *2 – Results table:*

| Matrix size | Sequential Time(s) | Best OpenCL time (s) | Speedup |
|---|---|---|---|
| **2048** | 3.53793 | 2.37702 | 1.4888 x |
| **4096** | 13.6594 | 2.83987 | 4.776 x |
| **8192** | 53.1358 | 4.82107 | 11.0206 x |

## *3 - Speedup graphs:*

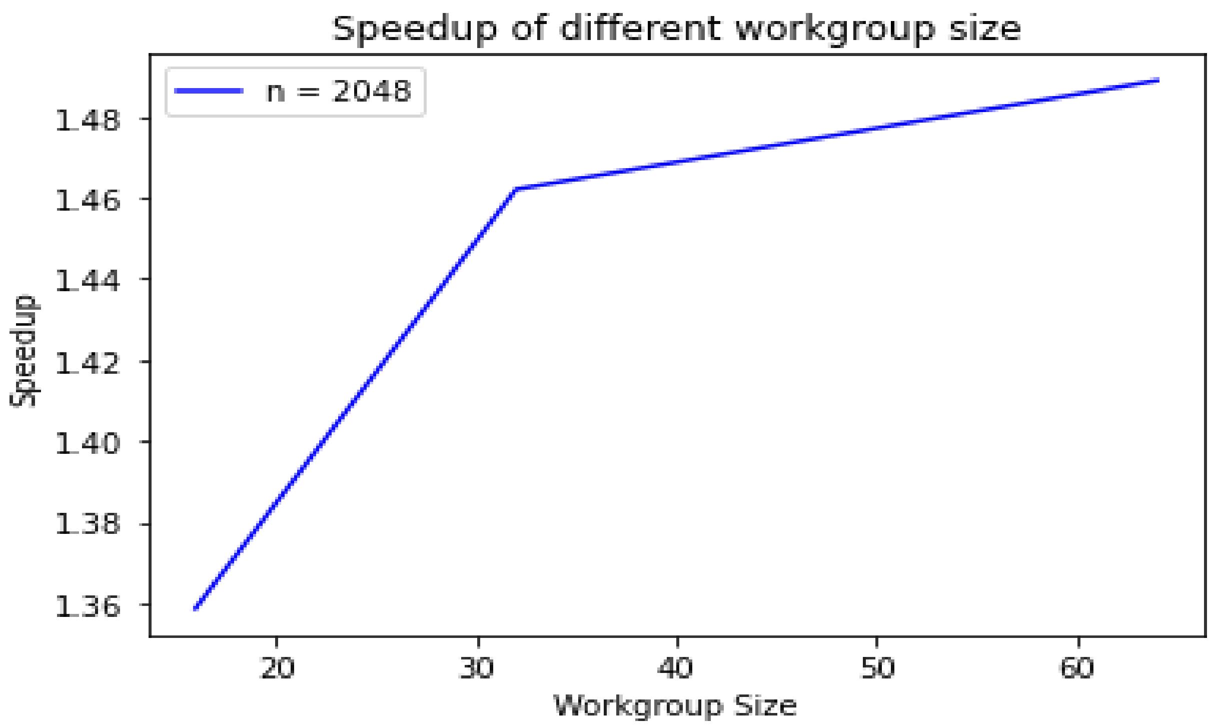

**Fig-1:** speedup of OpenCL comparing to sequential version with different workgroup sizes for n = 2048

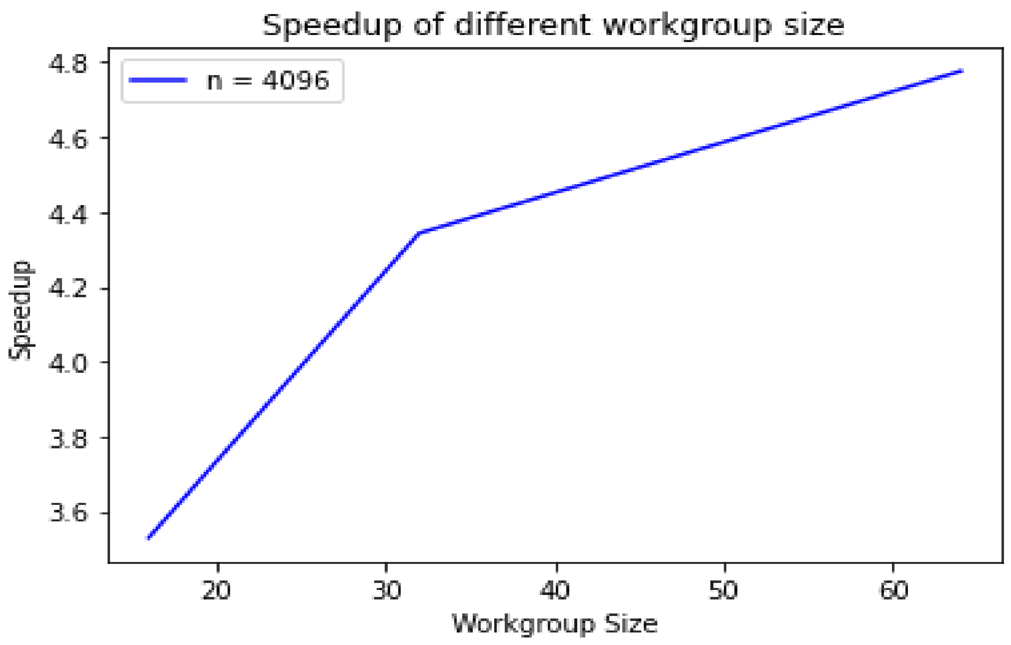

**Fig-2:** speedup of OpenCL comparing to sequential version with different workgroup sizes for n = 4096

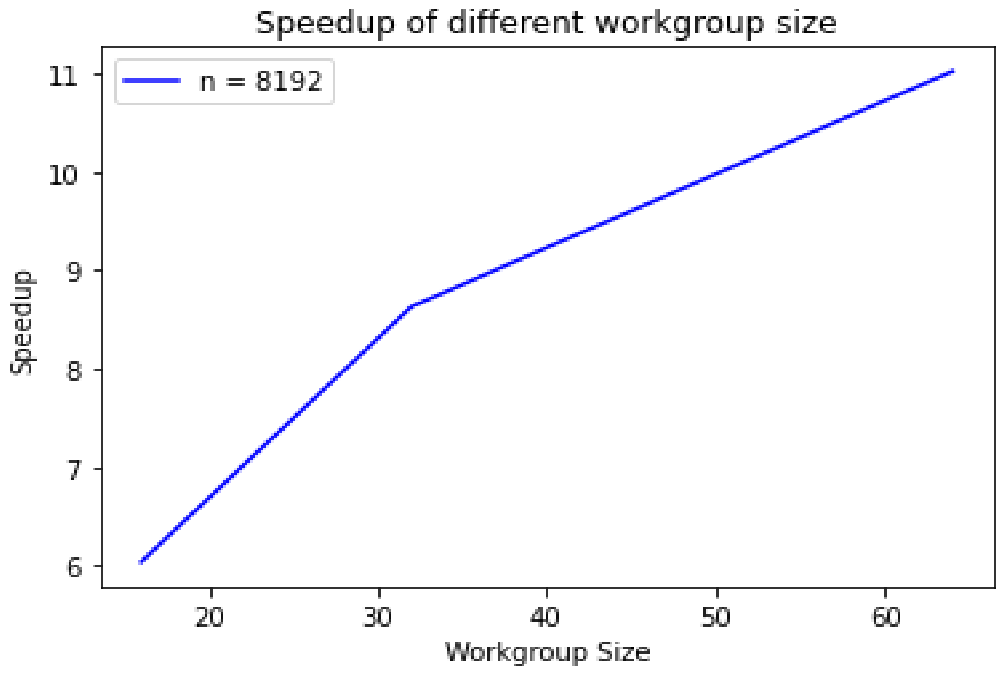

**Fig-3**: speedup of OpenCL comparing to sequential version with different workgroup sizes for n = 8192

## *4 - Graphs analysis:*

The code was compiled and run on ALMA server multiple times, with matrix sizes 2048, 4096, 8192 and workgroup sizes 16, 32, 64, these sizes are multiple of 16. I tested with these size in order to see how well OpenCL version would outperform the sequential version. We can clearly observe from Fig-1, Fig-2 and Fig-3 that the bigger the matrix size, the better speedup we achieve, with the best speedup of 11x in the case of n = 8192 and workgroup size = 64. I expected the speedup to get better when we increase the workgroup size as a speedup optimization technique, and the results confirmed my initial assumption.  We can see clearly that tuning the workgroup size optimize the runtime.

## *5 - Parallelization:*

We must first comprehend how OpenCL works in order to understand how the code has been parallelized. In essence, OpenCL is GPU programming. It uses both parallel and sequential code. The host code and kernel code are the two components of OpenCL code.

A. **Host Code:** This piece of the code was written in a c++ file. Here some code needs to write to set up the GPU for parallel programming. First, the kernel code has to be read. After reading the kernel file we have to get the platform and device information using predefined OpenCL functions. Platform implies checking in the event that the system has OpenCL framework or not and the device implies checking if the system has GPU for running OpenCL code. We select here only the GPU. In the event that the platform and devices are found, the next task is making a

**cl_context** utilizing the device IDs we saw before. From that point forward, I created a memory buffer by utilizing **clCreateBuffer()** capability. I involved **CL_MEM_READ_ONLY** for array *A, since we don't have to compose values inside this array. I likewise involved **CL_MEM_READ_WRITE** for array *B since we need to read values from array B and furthermore need to update it. After that I have done the followings:

- create a program from kernel source, build the program, create a kernel using the kernel code we read before, setting arguments into the kernel, declaring global item size, declaring local item size, execute kernel using **elEnqueueNDRangeKernel()** then read the array from device to host and at last free the memories.

A. **Kernel Code:** the kernel code is the parallel part of the program. The host code send the data here that we want to parallelize. Kernel.cl file contains our main kernel **MatriceTransformation()** that takes our 3 parameters, arrays A and B and the SIZE. With help of global id and the arrays size, I calculated the real indices for the row and column, which would be used in order to update the entries of the B array. The row and column borders were avoided from the update phase just like the sequential version. NDRange divide the work into multiple workgroups, which is the set of work-items where work-items are like a thread executing the kernel code. Prior to the parallel execution, the local work item size and global item size were set in the main function. I set the global item size to be equal to our arrays *A and *B SIZE, while I experimented with different local work item sizes as mentions above. The global item size should be a multiple of the local item size, any other way, our program wouldn't work as expected. The workgroups do their job separately, after finishing the data is read back from the device back to the host code.

## *6 – Optimization Ideas:*

- Using the most efficient data types

- Using local buffers
- Avoiding read/write buffer model on shared memory
- Using native function for math calculations