

**Report**

**Parallel Computing Assignment 2**

|  |  |  |
| --- | --- | --- |
| Names | BN | Sec |
| Abdelaziz Salah Mohammed | 3 | 2 |
| Ahmed Hosny Abdelrazik | 2 | 1 |

**Delivered to:**

Eng/ Mohammed Abdullah

**Introduction:**

We have created many different test cases to be able to analyze the performance in a better way, so we used the provided test cases, and then we have created another 3 test cases each of sizes: 10x10, 100x100, 1000x1000 respectively, and then we ran them and got the results which will be shown in the following screenshots.

Also, we have 3 main kernels as required in the requirement, but we have a fixed window of 16 x 16 threads per block, and we used this because we have found out that the best window size should be 256 and it is a common choice also.

However, the number of used blocks varies depending on the number of rows and columns using the equation used in the lecture by dividing the rows / 16 then getting the ceiling, and same operation is done on the columns, to have 2 dimensions block.

Moreover, we have made 2 assumptions:

1. size of vector is number of columns to apply multiplication (Matrix (n,m) \*vector (m,1) the result is (n,1) )
2. input files have numbers only (no comments)

**Analysis:**

Here we will show the total time taken by executing the kernals only, then at the end of the report we will provide screenshots for time taken for allocating memory on the GPU, copying elements and so on.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **K1** | **K2** | **K3** |
| **10x10** | 3520 ns | 4448 ns | 7584 ns |
| **100x100** | 3168 ns | 22529 ns | 42593 ns |
| **1000x1000** | 3392 ns | 16928 ns | 37921 ns |

As we expected, K1 is the fastest kernal, and this is obvious, because we have a lot of threads which are running concurrently, so all of them will calculate the addition of two corresponding cells in matrix A, and B, then evaluate the result in the matrix C.

What is interesting now is why does kernal 3 slower than kernal 2?

If we remember how elements are sorted in the memory, we will notice that the matrix is linearized in the memory in format similar to this: A close-up of a graph

Description automatically generated

So, this implies that elements in the same row are inserted next to each other, that indicates that threads could get the whole row inside its cache, and get the data closer to it, then apply the computation.

So, this allows it to reduce the number of IOs which is the bottle neck usually and decrease latency.

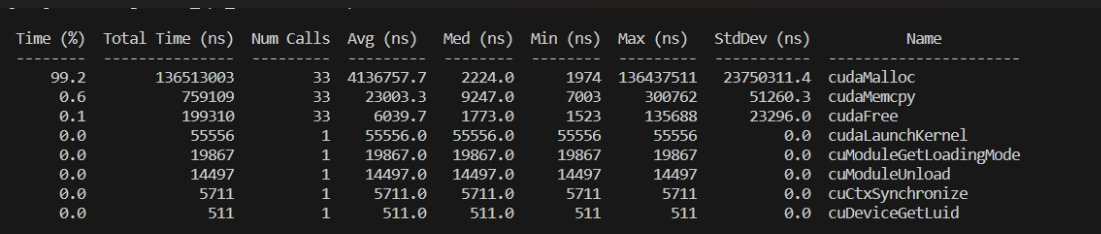
However, in kernel 3, we fix the column, and iterate over rows, so we stride with huge step in each iteration to get the col index in each row, so it does not use the benefit of having all elements next to each other.

**Other Factors:**

Here is an analysis of other factors such as time taken for copying data to GPU, and allocating space there and so on.

Usually transferring data, and allocating space, takes a huge amount of time relative to processing the data on the GPU.

**K1:**

**10x10: **

**100x100:**

**A black background with numbers

Description automatically generated**

**1000x1000:**

**A screenshot of a computer screen

Description automatically generated**

**K2:**

**10x10:**

**A screenshot of a computer

Description automatically generated**

**100x100:**

A screenshot of a computer

Description automatically generated

**1000x1000:**

A screenshot of a computer

Description automatically generated

**K3:**

**10x10:**

**A screenshot of a computer screen

Description automatically generated**

**100x100:**

A screenshot of a computer screen

Description automatically generated

**1000x1000:**

A black screen with white text

Description automatically generated