SP And Comm Matrix Multiplication Report

The algorithms that are used in this project are as follows:

Basic Matrix Multiplication: Here we are multiplying the two matrices by passing two global ids ([L,N] will be the global size) ie. get\_global\_id(0) and get\_global\_id(1). The number of work items is L x N (final matrix Y).

Optimization 1: We are multiplying the matrices by passing one global ID (number of rows is the global size). To optimize the speed of multiplication, we are iterating through Y’s row of items (M) in each work item.

Optimization 2: We are multiplying the matrices by passing one global ID (number of rows). To optimize the speed of multiplication, we are iterating through Y’s row of items (M) in each work item and also storing X’s rows within each item by storing it in private memory.

The graphs for the Various operations are given below:

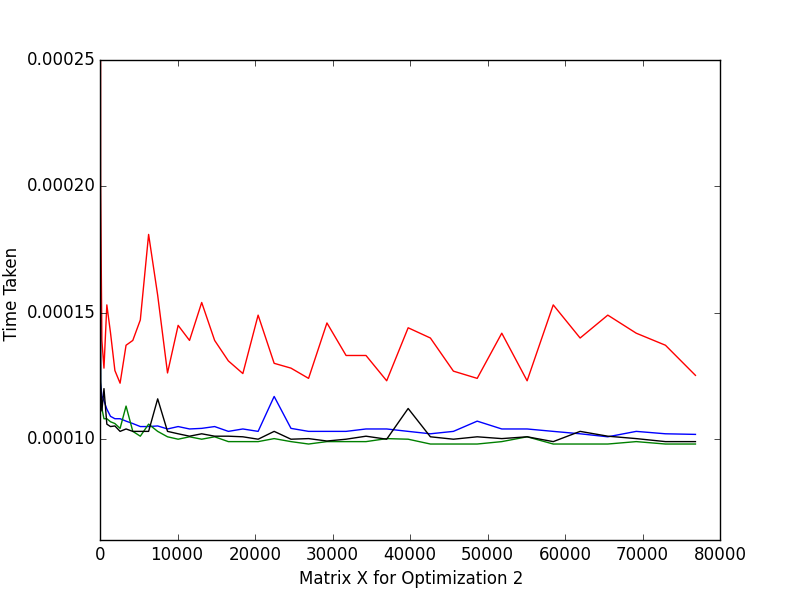
Red - Python Execution Time

Blue - Basic Matrix Multiplication

Green – Optimization 1

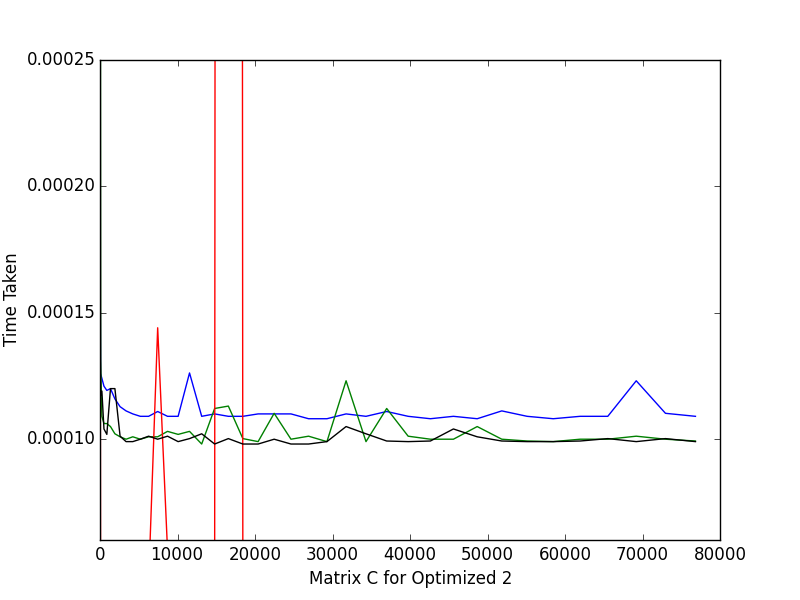
Black – Optimization 2

These are plotted against the size of the input matrices X(L\*M) and C(M\*N)

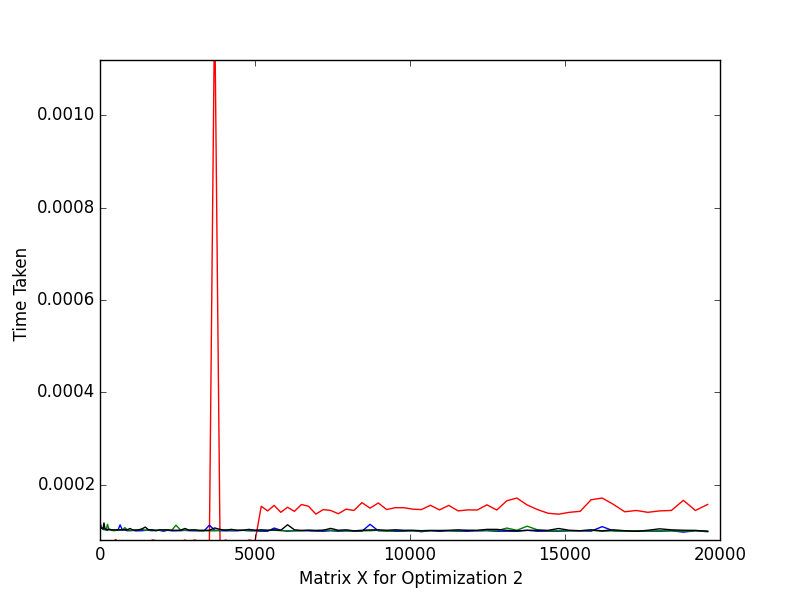


Here the Python time is clearly much slower than that of OpenCL. For this step size(5) the OpenCL speed is clearly almost the same for larger matrices.

This maybe due to the fact that the private memory is being used up and this overflowing into the global memory causing slower than expected performance.



In this case we clearly notice that the Optimization 2 speed is much higher than that of basic multiplication. It is still quite close to that of Optimization 1 speed.



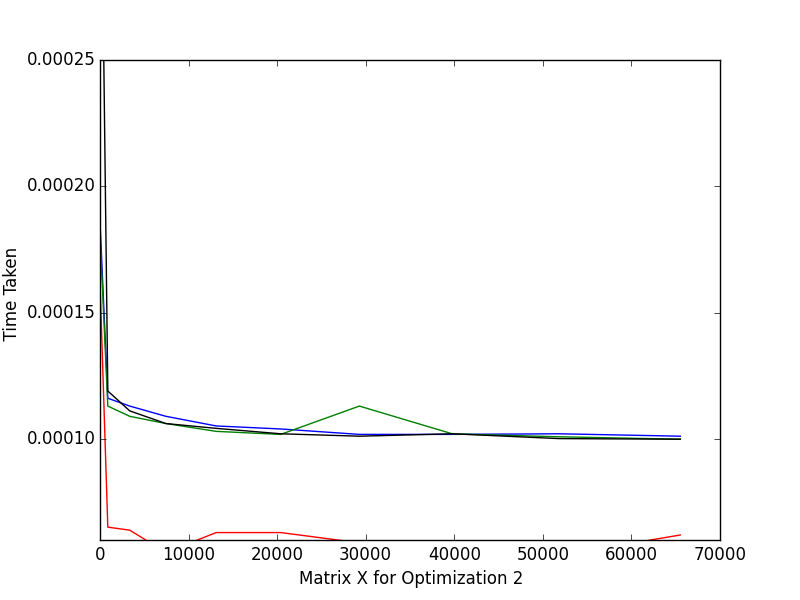
If the code is run independently The graph Is as shown abhove.

Note:

If the step size is changed then the algorithms produce dissimilar values:

For example considering a step size of 20 we get,

Here the python time for execution is much faster than that of OpenCL.



Though once again if the code is run for a different step size we get:

