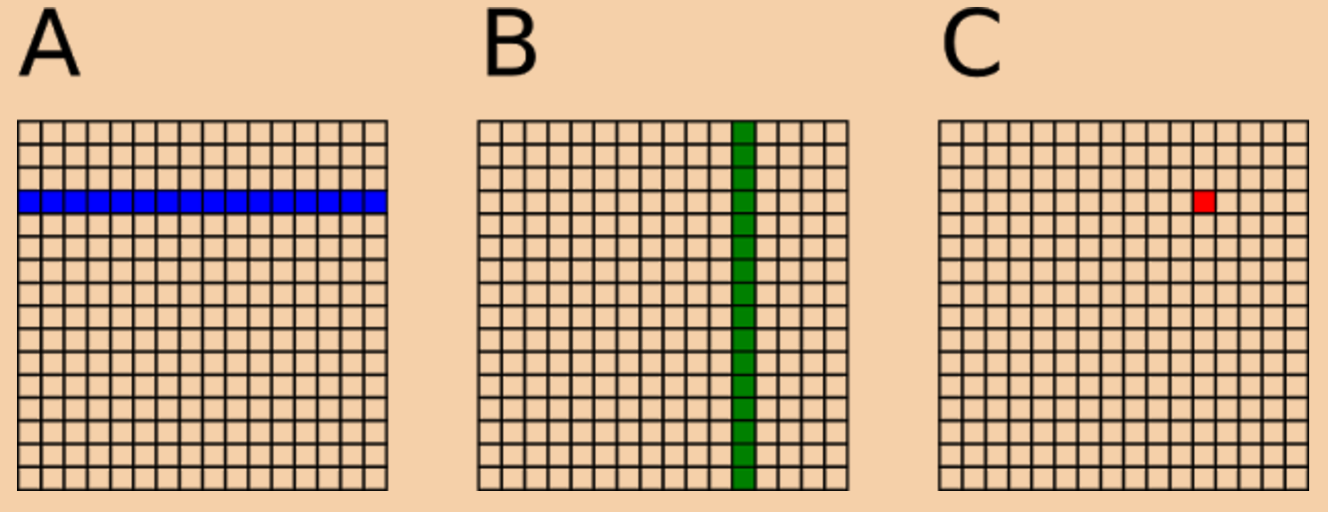
1. **Objective:**

Main objective of this program is to understand the hardware of the GPU machines in the lab and understand the performance improvement after parallelization through various versions of matrix-matrix multiplication.

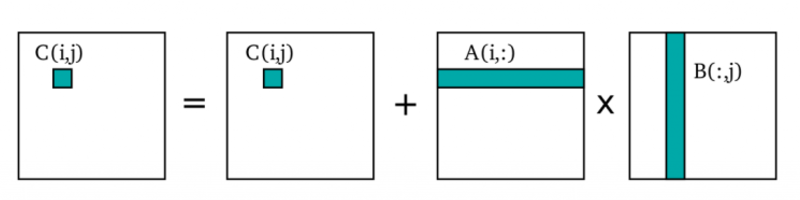
1. **Implementation:**

In this matrix-matrix multiplication, I have considered two square matrices of same sizes. I have implemented it various ways to understand parallelism and optimize the matrix-matrix multiplications.

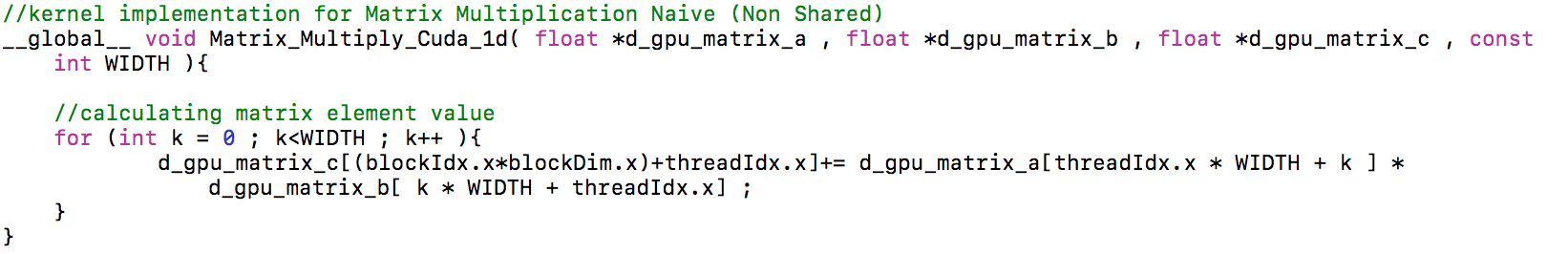


* 1. **Naïve Implementation using 1D blocks and 1D threads:**

In this approach, I have used 1D blocks and 1D threads. On GPU, one thread is assigned to compute one element of the resulting matrix. Each thread loads one row of first matrix and one column of second matrix from global memory, apply inner product and the store the result to resulting matrix in global memory.

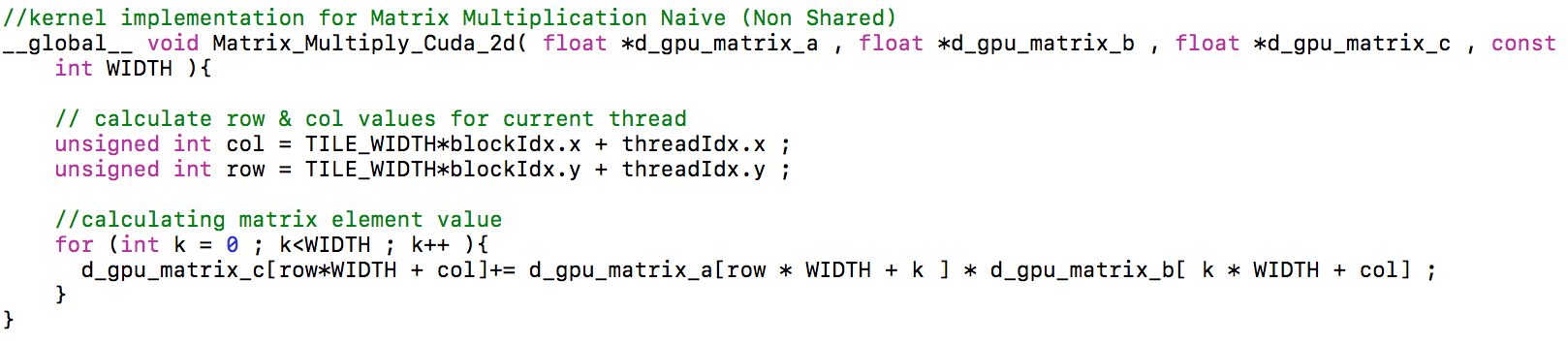


Please see the code snippet for more clear understanding:



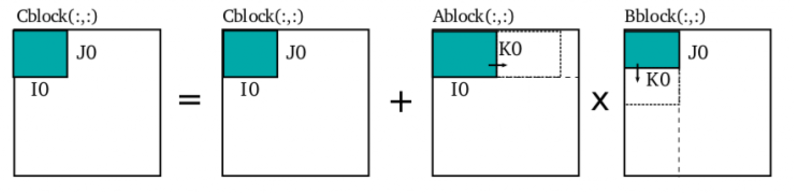
* 1. **Implementation using 2D blocks and 2D threads:**

In this approach, threads and blocks are arranged in a two-dimensional structure so that there x and y co-ordinates can be calculated. Please see code snippet for more clear understanding:

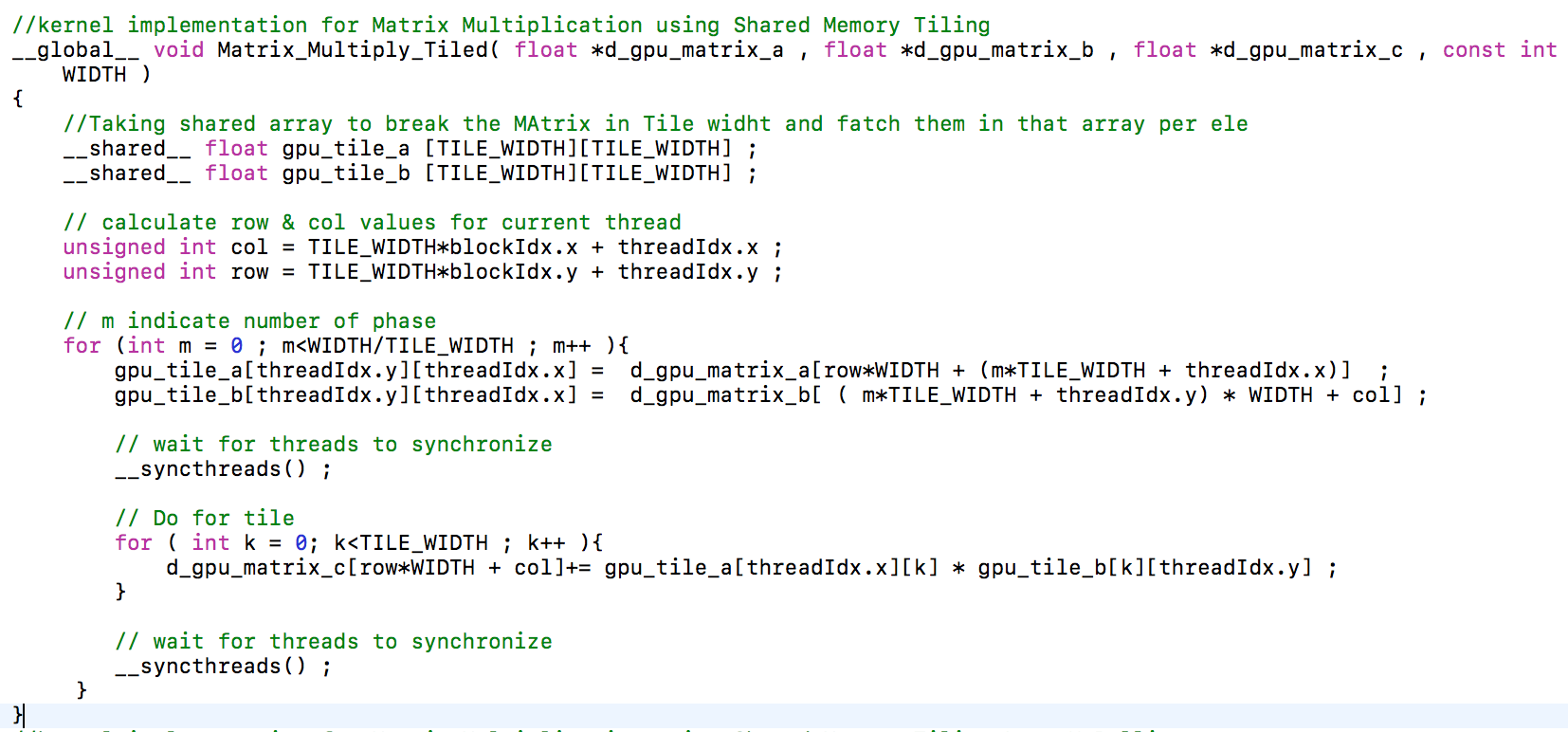


* 1. **Implementation using Tiling approach:**

This is an optimization method to increase the “computation to memory” ratio. This method achieves better locality through blocking. Basic idea is to rearrange the threads for smaller working sets in shared memory so that we don’t have to use global memory again and again. It ensures that data used in a loop stays in a cache until it is reused. I have specified different sizes of tiles per the size of the matrix.

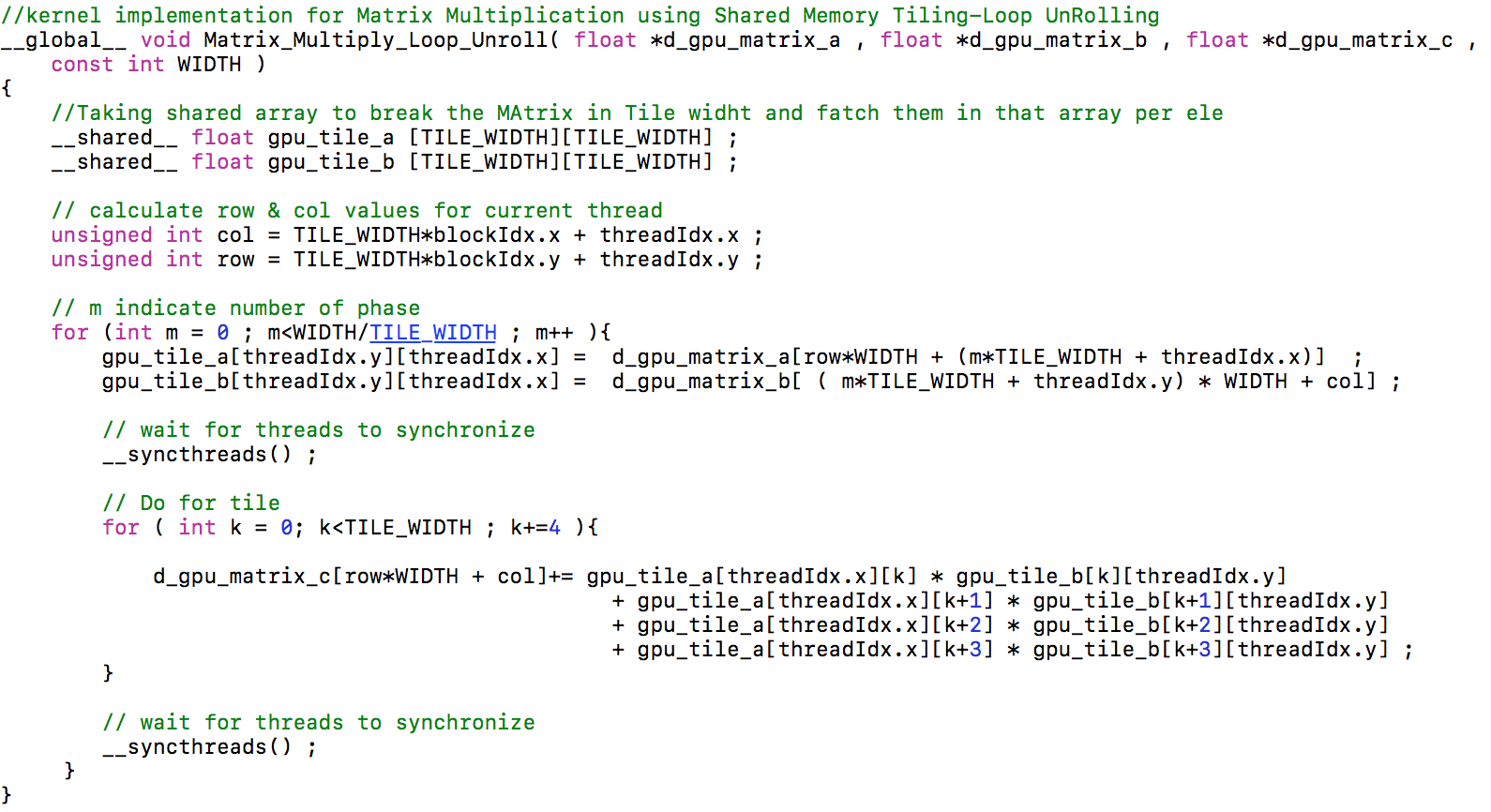


Please see code snippet for this approach:



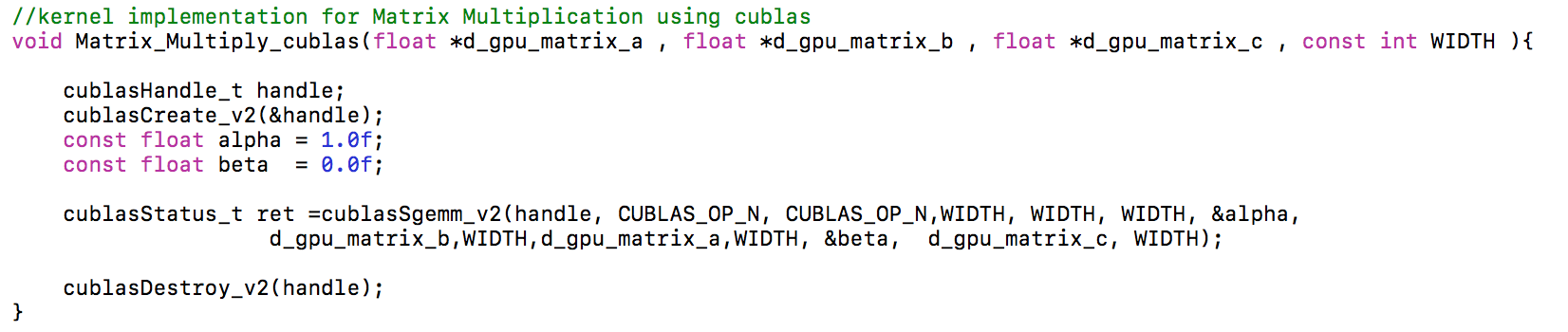
* 1. **Implementation using Tiling with Loop Unrolling approach:**

Loop unrolling is a technique in which the body of suitable loop is replaced with multiple copies of itself and the computational code is updated accordingly. I have unrolled the loop by factor of 4. Loop unrolling provides opportunity to exploit the registers and memory hierarchy locality when outer loops and unrolled and inner loops are used. Please see code snippet for this approach:



* 1. **Implementation using cuBLAS:**

This cuBLAS is a library already installed with nvidia toolkit. I have used this to measure my results against cuBLAS.



1. **Output Screenshots:**

**System Specifications:**

GPU Specifications: Tesla M2090: 2.0

Global memory: 5331mb

Shared memory: 48kb

Constant memory: 64kb

Block registers: 32768

Warp size: 32

Threads per block: 1024

Max block dimensions: [ 1024, 1024, 64 ]

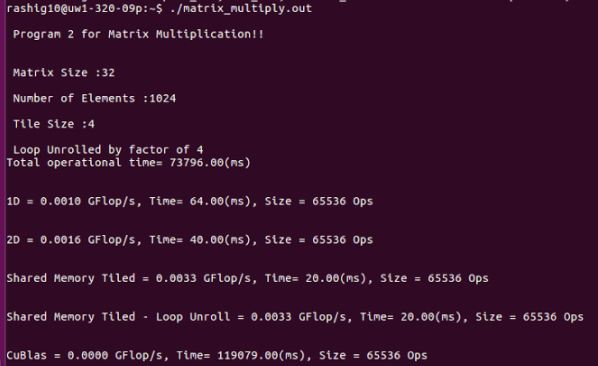
Max grid dimensions: [ 65535, 65535, 65535 ]

Number of multiprocessors: 16  
Number of CUDA cores: 512

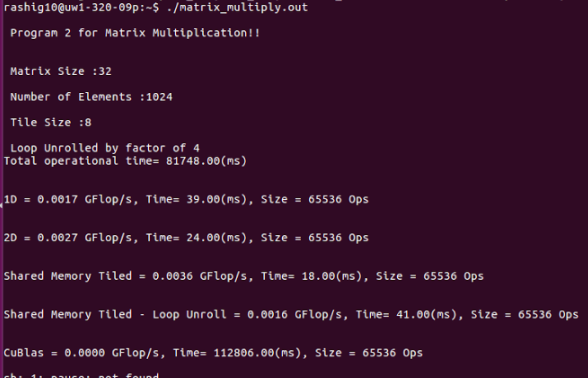
Max mem pitch: [2147483647](tel:(214)%20748-3647)  
Memory Clock Rate (GHz): 1.848000  
Memory Bus Width (bits): 384  
Peak Memory Bandwidth (GB/s): 177.408000

Theoretical Peak Performance: 665 GFlops

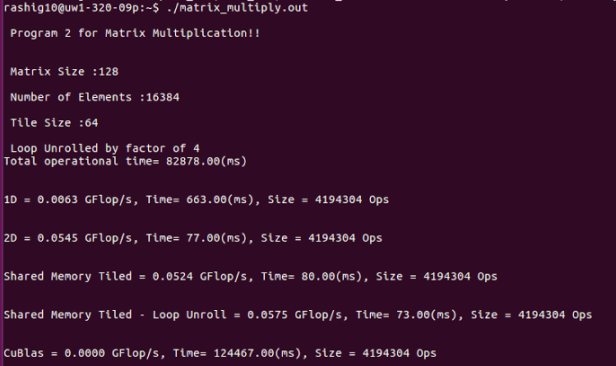
**3.1 Output for Matrix Size = 32, Tile Size = 4**



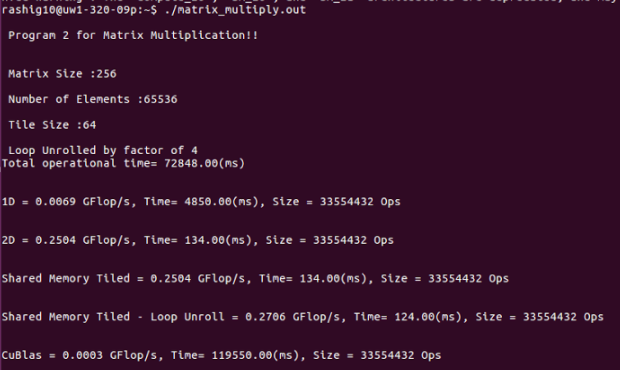
**3.2 Output for Matrix Size = 32, Tile Size = 8**



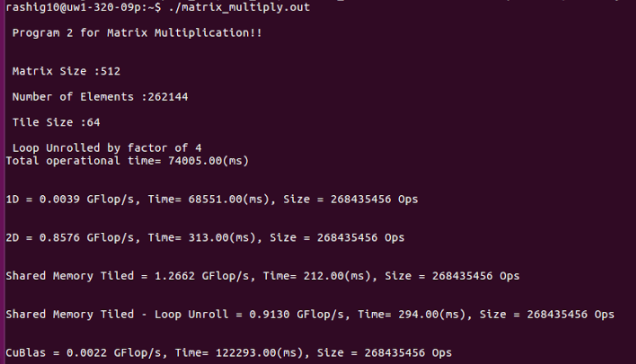
**3.3 Output for Matrix Size = 128, Tile Size = 64**



**3.4 Output for Matrix Size = 256, Tile Size = 64**

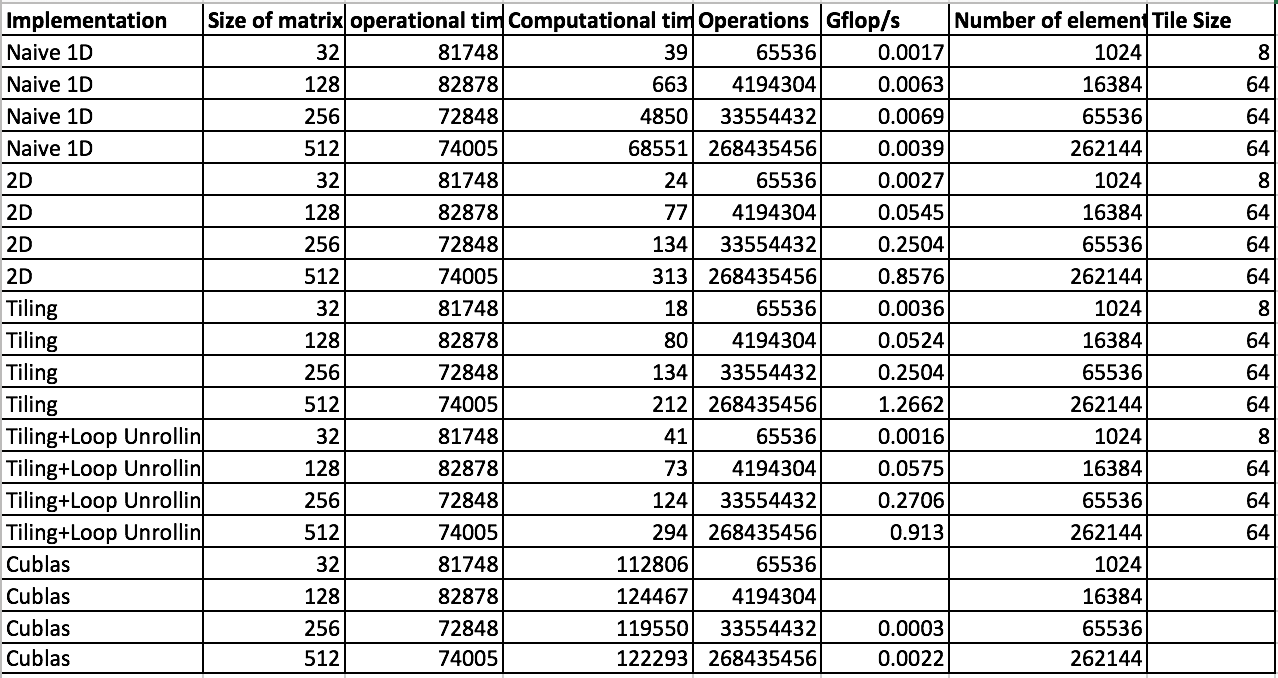


* 1. **Output for Matrix Size = 512, Tile Size = 64**



1. **Benchmarking:**

**Some result values for different sizes and their total operational and computational time**



**Discussions using Graphs:**

**GFLOP**S = NUMBER OF FLOATING POINT OPERATIONS

EXECUTION TIME TO MULTIPLY MATRIX (seconds)

**FLOPS** = 2 \* (Rows of Matrix A \* Cols of Matrix B \* Cols of Matrix C)

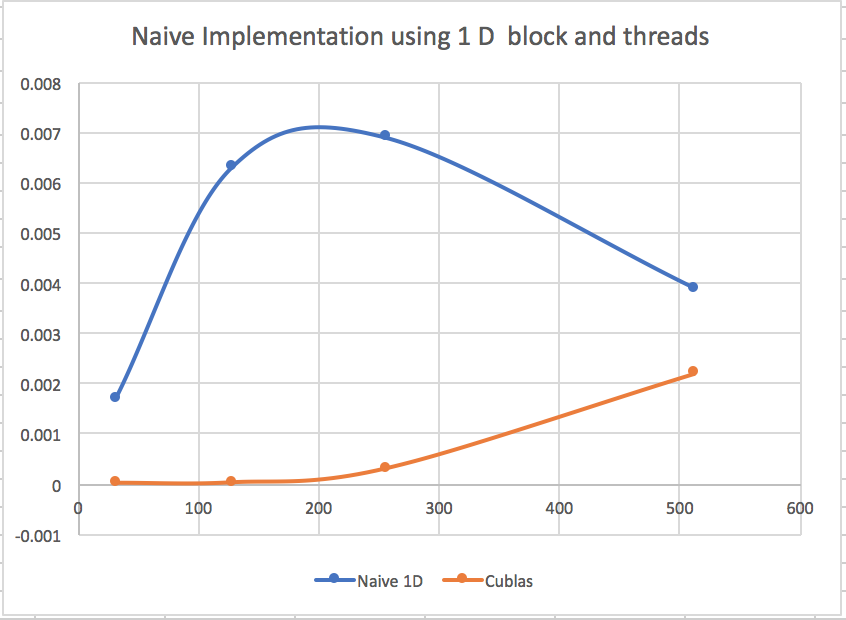
**OR** 2N3 for square Matrix

The N3 is multiplied by 2 for Multiplication and Addition operations performed.

* 1. **Naïve 1D:**

x- axis represents matrix sizes

y-axis represents Gflop/s

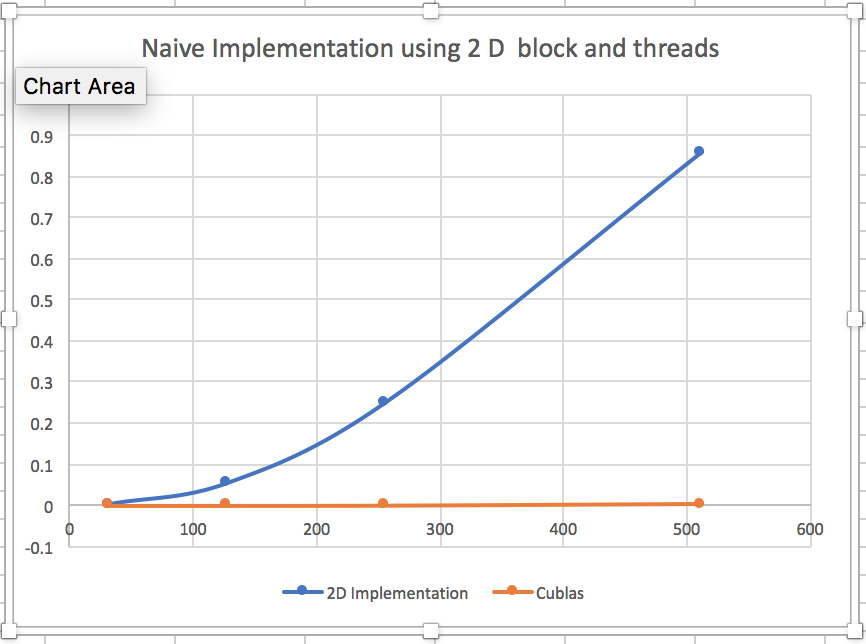
****

In the graph shown below, GFLOPS values are plotted against increasing matrix sizes using Naïve 1D Threading approach. As the size of the matrix increased, performance decreased because threads have to access global memory more and more with increasing time which increases computational time causing latency. The cuBLAS shows improvement in performance.

* 1. **Matrix multiplication using 2D blocks and 2D threads:**

x- axis represents matrix sizes

y-axis represents Gflop/s

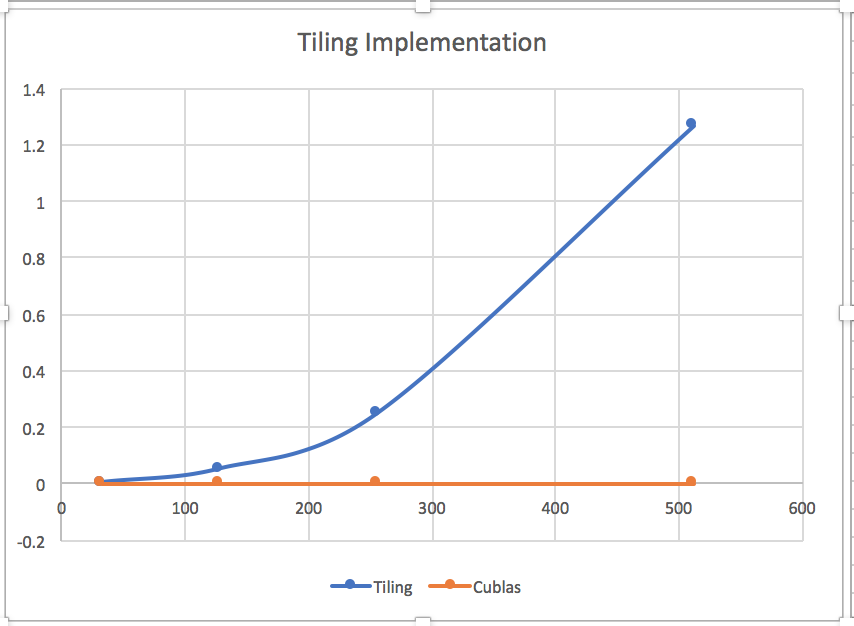
****

This graph shows performance of matrix-matrix multiplication using 2 dimensional blocks and threads. We can observe as compared to cuBLAS, the performance is increasing. It may be because using two-dimensional set of threads and blocks decreases latency. This is my observation. In my implementation, I have distributed computations using more blocks per grid while keeping the number of threads per block lower. The number of threads per block were 64 which twice the warp size.

* 1. **Matrix multiplication using Tiling:**

x- axis represents matrix sizes

y-axis represents Gflop/s

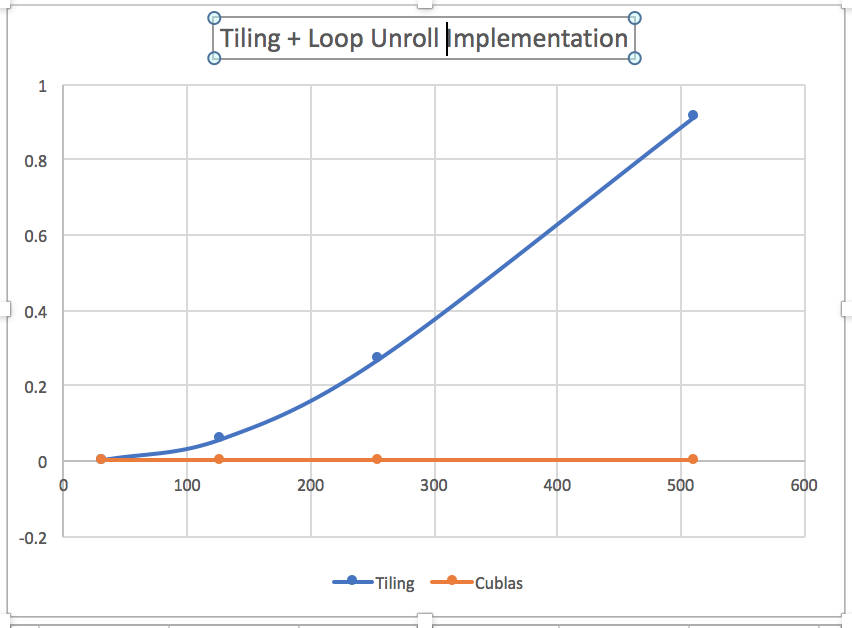
****

This graph shows performance of matrix-matrix multiplication using tiling method. Tiling method shows improvement as size of the matrix increases, performance increases because tiling is improving the locality of the program. Threads do not have to access global memory again and again. The multiplication of the tile happens in shared memory ensuring that data used in a loop stays in a cache until it is reused. I also observed an increase in Gflops values with increasing tile size for a fixed matrix.

* 1. **Matrix multiplication using Tiling + Loop unrolling:**

x- axis represents matrix sizes

y-axis represents Gflop/s

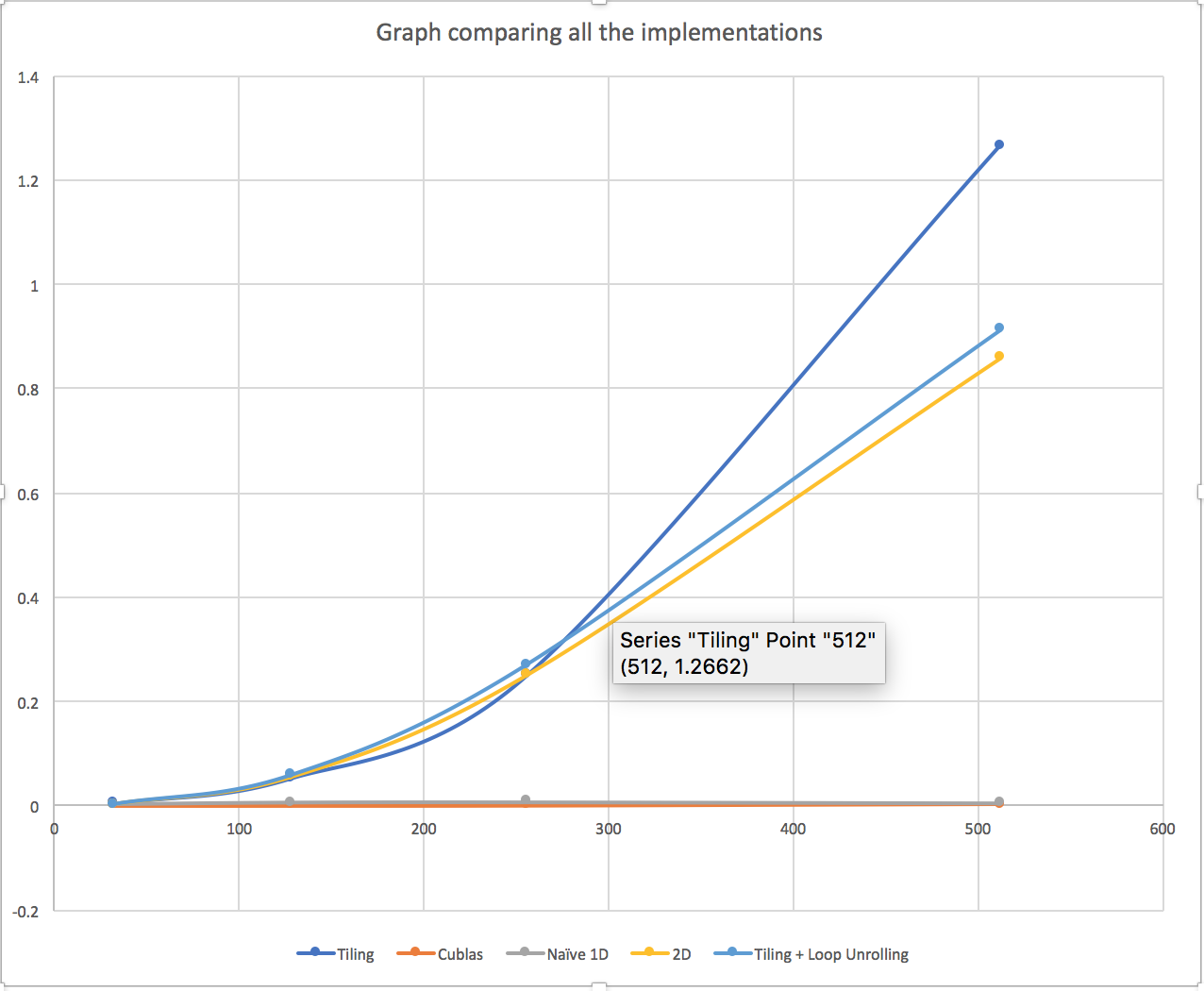
****

As discussed above, tiling improves the performance by decreasing the interaction with global memory and performing more work in shared memory which is accessible to all the threads in the block. Now, when we apply loop unrolling with tiling, it will replace the loop with multiple copies of itself and the computational code is updated accordingly. So together, this approach is using coalescing and more registers in the shared memory which is decreasing the execution time. Hence, the performance is increased.

* 1. **Graph comparing all the matrix multiplication implementations:**

x- axis represents matrix sizes

y-axis represents Gflop/s

****

This graph shows the performance comparison between of all the implementations we have tried. Clearly, Tiling and Tiling + Loop unrolling are better than Naïve 1D and 2D. cuBLAS is giving lowest values of all and I am not sure about the reason behind it. May be my configuration has beaten the cuBLAS.

1. **Steps to run this assignment:**
2. nvcc matrix\_multiply.cu -lcublas -o matrix\_multiply.out
3. ./ matrix\_multiply.out
4. In case you want to check out the result matrix values, Please uncomment the blocks printing matrix C.
5. **References**:

<http://www.academia.edu/12205718/Lecture02_memorycache>

<http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemm>

<http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-Naive-Implementation-On-GPUs>