CIS-750 Advanced Computer Architecture- Final Project Report

Matrix Multiplication Using CUDA C

a) Introduction and motivation

Graphics processing unit is a specialized processor designed to accelerate computation for drawing and manipulating images, and perform high computational research. In a multi-core processor performance, Acceleration is achieved by integrated circuit. Modern GPUs have tera floating point operations per second processing power and manufactures are designing them for general purpose. GPU have shown highest increase in chip level parallelism in recent years and Cuda has been one of the programming approaches has given us the opportunity to understand the power across many computational domains. A GPU has up to m streaming processor and organized as n streaming multiprocessor. Each streaming multiprocessor is an independent processing unit. Most part of the executable source code is performed inside the GPU. High computations involving linear algebra algorithms come out as an organic fit for CUDA and GPU's. Most of the GPU'S bring their computing power from accelerators; hence it is important for any efficient and scalable algorithm to extract the most performance out of the accelerators to achieve highest efficiency in terms of time and memory usage. Several projects aim at benchmarking dense and highly calculation oriented kernels.

Thus, it is important to do critical analysis on one of the basic operations in linear algebra: The matrix multiplication which has efficient implementation for input sizes ranging from 32 to 4096. So, in this project we discuss deeply about efficient ways to solve the problems involving high computational approach: Matrix- multiplication. As studies show that, it is not easy to achieve high performance using sequential algorithms for matrix multiplication. Matrix multiplication has been naturally chosen as a benchmark for performance of standard GPU or any kind of parallel machine. Matrix multiplication performed in this methodology addresses various approaches in performing multiplication operations and efficient way of performing those multiplications. This methodology also addresses performance comparison between Open MP and MPI approaches. In this implementation GeForce GTX 980 Ti which has 2861 CUDA cores and 22 streaming multiprocessors has been used for all computation approaches and CUDA C for implementation.

(b) Related research

The Application of matrix multiplication is increasing day by day. Few research fields which use matrix multiplication are (i) To analyze weather patterns, Linear algebra applications; recognize human faces, neural networks, self-learning cars and many machine learning algorithms [1]. An extensive research is going on to find out the interconnectedness of graph theory, matrix and probability. In many applications, the shape of the matrix can significantly impact the research field. Research related to improving parallel matrix multiplication involving SRUMMA matrix multiplication to improve performance of transpose matrix and rectangular matrices [5]. This research exploits performance characteristics of clustered and shared memory systems. It differs from the other parallel matrix multiplication algorithms by the explicit use of shared

memory and remote memory access (RMA) communication rather than message passing [2]. Many researchers are working to provide an optimum solution over exponential number of candidate solutions this technique is called dynamic programming. Dynamic programming can be applied to matrix multiplication [3]

(c) Methodology or approach

CUDA programming interface:

GPUs such as GeForce GTX 980 Ti are very powerful computing platforms which use Compute unified device as their general purpose computing. At any given cycle, each core executes the same instruction on different data (SIMD), and communication between multiprocessors is performed through global memory
As a programming interface, CUDA consists of C language library functions. CUDA compiler generates an executable from the source code. A program in CUDA c consists of following elements
Block: A block is group of threads that are handled by a single processor. CUDA program is partitioned into blocks that run in parallel.

Thread: A threads are the components of block where group of threads constitute a block. Threads share resources equally among themselves in single instruction Multiple data (SIMD). This decomposition of resources can be managed programmatically.

Warp: Warp is a collection of threads that run simultaneously on all the multiprocessors

Kernel: It is the code that needs to be executed in each thread. Kernel execution can be managed programmatically by using unique ID for each thread.

To demonstrate the efficient use of CUDA threads and blocks this project follows 4 different types of algorithms. In general, every proposed approach in this solution contains three functions main, CPU initialization function and GPU kernel function. Main function is responsible for running the program and calling CPU matrix multiplication function with integer arguments like 32, 64, 128, 256,…4096. Inside the CPU initialization function memory allocation and de allocation of both device and host takes place. CPU initialization function is also responsible for copying data from host to device and vice versa. CPU initialization function is also responsible for invoking the kernel and randomly initializing the elements in the matrix and calculating the time taken for kernel operation. Finally, this function computes the number of threads and blocks required for the CUDA kernel to start the operation by calling kernel by Matrix_GPU<<<n_blocks, block_size>>> (device_matrix_A, device_matrix_B ….,….,…)

Methodologies used in CPU initialization function:

Allocating memory to matrix A in the host is done by using malloc with the size m*n where m and n are number of rows and columns of the matrix A respectively. mat_A = (double*)malloc(m*n* sizeof(double));. The de-allocation of memory is done by free(mat_A)

Similarly, allocating memory to the device matrix A is done by using cudaMalloc with the size m*n where m and n are number of rows and columns of the matrix A respectively. cudaMalloc (&device_matrix_A, m*n*sizeof (double)); allocates memory to the matrix A. The de allocation of memory is done by cudaFree (device_matrix_A);

Initialization of matrices:
As memory is allocated to matrices in dynamic and it is of size m*n the matrices are randomly initialized using rand() for simple pointer arithmetic with two for loops ranging from 0 to m and 0 to n. The elements are initialized by matrix_A [i *n + j] = rand () % 10;

Copying data host memory to device memory and vice versa:

Data initialized to matrices must be copied from host to device this is done by the cudaMemcpy. DataProgrammatically written as cudaMemcpy (device_matrix_A, matrix_A, m*n*sizeof (double), cudaMemcpyHostToDevice). Similarly, the data from the device kernel to host is done by
cudaMemcpy(matrix_C, device_mat_D, m*l*sizeof(double), cudaMemcpyDeviceToHost);

CUDA Kernels: Each algorithm mention in this project has unique kernels with main function similar with few changes i.e calculating number of blocks in x and y direction.

Algorithm 1:

It is the unsophisticated way of using CUDA blocks and threads. It uses one dimensional blocks in calculating matrix multiplication. Every thread is responsible for calculating the one element is the resultant matrix. Block size in this algorithm is 256. Number of blocks for the CUDA kernel in calculated by the

```
numberOfBlocks = (m*n + blockSize - 1)/blockSize);
blockSize = 256;
```

Step 1: launching the kernel
```
GPU_MATRIX_1<<<num_blocks, BLOCK_SIZE_1>>>(d_mat_A,  d_mat_B,
d_mat_C,d_mat_D,
```

Step2: Matrix multiplication inside the kernel.
Calculate unique thread ID by using built in CUDA functions
```
int threadID = threadIdx.x + blockDim.x*blockIdx.x;
```

Allocate a row and column for each thread by which matrix multiplication can be computed.
```
int column = threadID %n;
int row = threadID /n;
```

compute the multiplication and summation.
```
for(int k = 0; k< n; k++){
     sum += d_mat_A[k*m + row]*d_mat_B[k*n + column];
}
```

Write matrix C and pass it to the main function

```
Mat_D_ID = column*m + row
d_mat_D[idx] = sum;
```

Algorithm 2:

This algorithm is a bit sophisticated compared to the previous algorithm. This uses two Dimensional blocks instead of one dimensional block. This is still an unsophisticated way of using cuda blocks and threads but better than the previous algorithm. In this algorithm the block size is defined as block_size_x=32 in x-dimension and BLOCK_SIZE_y = 16 in Y dimension. Number of block in x dimension = (m + threadsInXdirection - 1)/ threadsInXdirection. This calculation follows to the number of blocks in y dimension.

Calculating number of blocks and launching the kernel:

```
nthreads.x = BLOCK_SIZE_x;
nthreads.y = BLOCK_SIZE_y;
nblocks.x = (m + nthreads.x - 1)/nthreads.x;
nblocks.y = (l + nthreads.y - 1)/nthreads.y;
device_GEMM_2 <<<nblocks, nthreads>>>(d_mat_A, d_mat_B,
d_mat_C, m, n);
```

Step2: Inside the kernel

Calculate uique thread IDs

```
const int threadid_x = threadIdx.x + blockDim.x*blockIdx.x;
const int threadid_y = threadIdx.y + blockDim.y*blockIdx.y;
```

calculate multiplication and summation of the matrices:

```
for (int k = 0; k < n; k++){
     sum  +=  d_mat_A[k*m  +  threadid_x;]*d_mat_B[threadid_y*n  +
k];
}
```

Transfer sum to matric C and pass it to main function:

```
Int Cid = threadid_y*m + threadid_x;
d_mat_C[Cid] = sum;
```

Algorithm 3:

This algorithm has better performance than algorithm 2. This uses two dimensional blocks with shared memory. In this algorithm each block has 32 *32 threads which computers smaller matrices 32 *32. After each computation the values calculated are loaded in the shared memory. Blocks in x and y direction are calculated as nblocks.x = (m + nthreads.x - 1)nthreads.x; nblocks.y = (l+nthreads.y - 1)/nthreads.y; where nthreads.x = nthreads_y = 32 and as mentioned in previous cases m *n is the size of the matrix.

Step 1: Launching the kernel.

```
nthreads.x =32;
nthreads.y = 32;
nblocks.x = (m + nthreads.x - 1)/nthreads.x;
nblocks.y = (l + nthreads.y - 1)/nthreads.y;
device_GEMM_3 <BLOCK_SIZE_3><<<nblocks, nthreads>>> (
device_mat_A, device_mat_B, device_mat_C, device_mat_D, m, n )
```

step 2:
Inside the kernel
//calculating global thread, block ids

```
int global_threadid_x = threadIdx.x + blockDim.x*blockIdx.x;
int global_threadid_y = threadIdx.y + blockDim.y*blockIdx.y;
int threadid_x = threadIdx.x;
int threadid_y = threadIdx.y;
int blockid_x = blockIdx.x;
int blockid_y = blockIdx.y;
```

- Loop over the matrices to be multiplied
- Use __syncthreads(); to synchronize that matrices are loaded in the shared memory.
- Calculate the matrix sum

//calculate ID of Matric C:

```
int ID_C = m*block_size*blockid_y + block_size*blockid_x +
threadid_x + m*threadid_y;
```

Load the sum into Device_Matrix_C by looping over shared matrix A and B with K from 0 to block size.

```
sum += mat_A_shared[threadid_x][k]*mat_B_shared[k][threadid_y];
d_mat_D[ID_C] = sum;
```

Algorithms 4:

This algorithm is best of 4 algorithms which use 2 dimensional blocks that use 16* 4 threads per block. Each thread calculates sub matrix of size 64*16. Blocks in x direction and y direction are calculated as shown.

Step 1: Launching the Kernel

```
nthreads.x = 16;
nthreads.y = 4;
nblocks.x = (m + nthreads.x*nthreads.y -
1)/(nthreads.x*n_threads.y);
nblocks.y = (m + nthreads.x - 1)/nthreads.x;
```

```
device_GEMM_4<BLOCK_SIZE_x_4, BLOCK_SIZE_y_4> <<<nblocks,
nthreads>>> (device_mat_A, device_mat_B, device_mat_C,
device_mat_D, m, n)
```

Step2:Matrix multiplication inside the kernel.

Calculate block IDs and thread IDs.

```
int threadid_x = threadIdx.x;
int threadid_y = threadIdx.y;
int blockid_x = blockIdx.x;
int blockid_y = blockIdx.y;
```

create a shared memory:

```
__shared__ double matrix_B_shared[block_size_x][block_size_x+1];
```

syncthreads(); here

Caclculate element id of Matrix A, Matrix B,

```
int idx_A = blockid_x*block_size_x*block_size_y + threadid_x
+threadid_y*block_size_x;
int idx_B = threadid_x + (bid_y*block_size_x + threadid_y)*n;
```

Load matrix B into shared memory

```
mat_B_shared[threadid_x][threadid_y + i] = d_mat_B[idx_B + i*n];
```

calculate multiplication and summation between Shared Matrix B and matrix A

```
c[id] += d_mat_A[idx_A]*mat_B_shared[i][j];
```

(d) Experiments: number of elements versus number of threads:

To know how the proposed algorithms work and to measure how fast the algorithms work experiments are done with two matrices each of size raging from (32 to 4096). The kernel is launched by calculating number of threads and number of blocks dynamically based on the number of elements of each matrix. So, the total number of threads launched can be calculated as writing a block of code inside the kernel as shown below. atomicAdd increases the value of total_threads whenever the kernel is executed by each individual thread.

```
__device__ unsigned long long total_Threads = 0;
__global__ void Marix_Mult(double argument1,double argument2){
atomicAdd(&total_threads, 1);
}
```

Inside the main function after the launching the kernel the total number of threads used can be printed as shown below

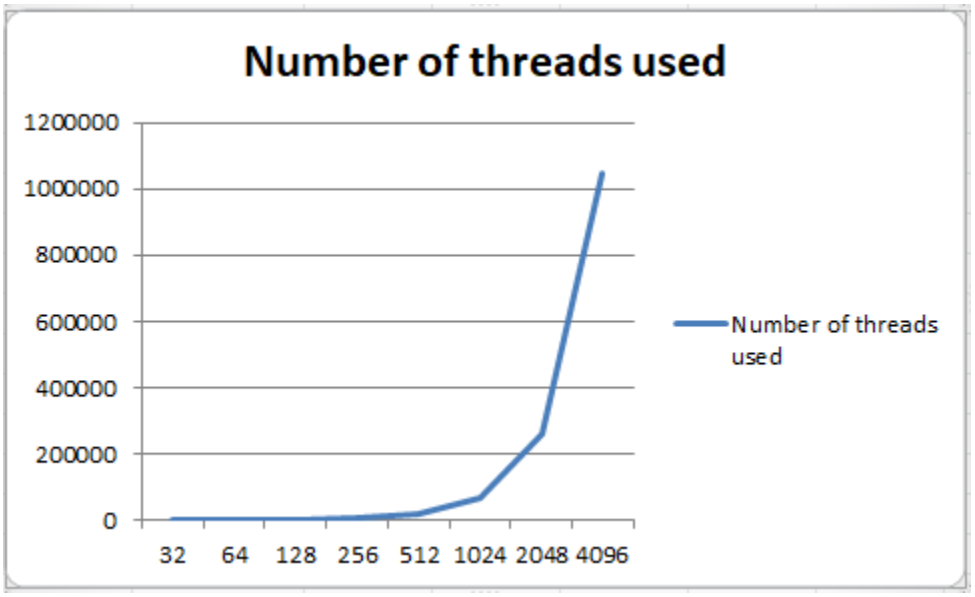
```
unsigned long long total;
cudaMemcpyFromSymbol(&total,totThr,sizeof(unsigned long long));
printf("Total threads counted: %lu for %d \n", total, m);
```

Number of elements	Number of threads used
32	128
64	256
128	1024
256	4096
512	16384
1024	65536
2048	262144
4096	1048576

Table 1.

As tabulated in the above Table 1. Number of threads with kernel has been launch increases rapidly as the number of elements of the matrix is increased. With highest number of threads 1048576 launched for the matrix of size 4096.

figure 1.



Results: Performance comparison of 4 algorithms

This section explains the performance results of 4 CUDA C algorithms for matrix multiplication. In this case metrics considered to measure the performance is milliseconds Performance analysis for all the four algorithms are done using cudaEventRecord (start) and cudaEventRecord (stop) and the time is measured in milliseconds. Time taken (in milliseconds) to perform matrix multiplication of matrices of sizes 32,64, 128, 256, 512, 1024, 2048, 4096 are tabulated as shown in the table 1. As it can be observed that the table illustrates interesting facts of CUDA matrix multiplication. It can be understood that the algorithm 4 completes its execution for 4096 elements took only 736 milliseconds. It achieved a speedup of 18.xx times

compared to naïve algorithm 1 which uses one dimensional block to compute matrix multiplication. Algorithm 4 achieved a speed up of 9.xx times compared to second matrix multiplication algorithm and achieved a speed up of 1.2x times for third matrix multiplication algorithm.

Taking the least number of elements into consideration the four algorithms does not show any show any significant differences in their execution times. All the algorithms show almost equal timings when the number of elements is in the interval 32 -256. Actual optimization and performance can be compared when size of matrix grows linearly from 256 elements. The speed up achieve by algorithm 4 for the matrix of size 32 compared to algorithm 1, algorithm 2, algorithm 3 is 1.75, 1.37, 1.38 respectively. So, the efficiency and performance of any algorithm can be seen as computing elements grow in size.

The tabular form shows the time taken for matrix multiplication against number of elements with all the algorithms running on same number of threads for their respective matrix size.

Number of elements	CUDA 1(ms)	CUDA 2(ms)	CUDA 3(ms)	CUDA 4(ms)
32	0.046688	0.0568	0.056384	0.078176
64	0.05376	0.063232	0.05264	0.085408
128	0.174208	0.167744	0.100288	0.16384
256	1.16944	1.152256	0.42704	0.45008
512	6.972352	7.20752	2.754432	2.201408
1024	69.119682	54.033314	18.630655	17.062817
2048	804.886475	384.229889	142.047897	115.546722
4096	13542.08398	6679.260742	945.32782	736.899902

Comparing Best performing MPI, openMP and CUDA algorithms in terms of execution times: This section explains CUDA C's high performance algorithm compared to openMP, MPI and sequential algorithms. Metrics considered in comparing is these algorithms are similar to previous section i.e milliseconds. To overcome the drawbacks of sequential algorithm CUDA based matrix multiplication algorithm is proposed and runtimes are calculated to check how fast the parallel algorithm works compared to sequential. From the tabulated data, runtimes of each algorithm can be analyzed. The performance of CUDA C outweighs sequential algorithm as well as other two parallel algorithms.

CUDA C algorithm achieves a speed up of 878.xx times compared to sequential algorithm. As discussed the speedup achieved for matrices of small sizes is quite negligible, CUDA C's performance compared to OpenMP for matrix of size 1024 achieved a speedup of 238.xx times. Speed up is quite negligible for matrices of smaller sizes.

Comparing MPI standard timings proposed best performing CUDA C algorithm. It is observed that CUDA achieved a speed up of 201.xx for matrix of size 512. And relatively high speed of 25.xx even for smaller size matrices. Same number of threads has been used to multiply between same number of matrices input sizes for OpenMp, CUDA and MPI program being on 8 processes.

number of elements	Sequential(milliseconds)	MPI(milliseconds)	OPEN MP(milliseconds)	CUDA(milliseconds)
32	0.09657	1.814	0.26499	0.078176
64	1.384032	17.725	1.359649	0.085408
128	44.49878	33.68	17.18784	0.16384
256	600.4988	156.2248	34.17784	0.45008
512	1391.243	408	387.5622	2.201408
1024	14934.15	4368.661	4013.944	17.06282

Analysis:

Speedup due to Coalescing accesses to device memory:
Coalescing enabled the data accesses us to recover simultaneously from the device memory. So, all the threads within warp compute in parallel

Speedup due to use of shared memory and tiled approach:
In order to increase the computation speed, algorithms reuse data located in the lower level of memory hierarchy by tiled approach. To perform tiled approach, first we need to copy the device to shared memory used by the threads in a single block. These threads cooperate with each other and load data efficiently from the shared memory. By which coalescing can be achieved. Then perform the actual computations in the shared memory by proper usage of memory banks. Finally copy back the shared memory results to the device memory.

These two approaches have resulted reduction of kernel time from algorithm1 to algorithm4 by running on same number threads and there by achieving better performance and speedup compared to previous ones.

Comparing four algorithms with line graph with time in milliseconds on x-axis and number of elements of each matrix on y-axis. Time measured in milliseconds. As discussed in previous section the continuous plot timings of all algorithms is show in the below figure 2.

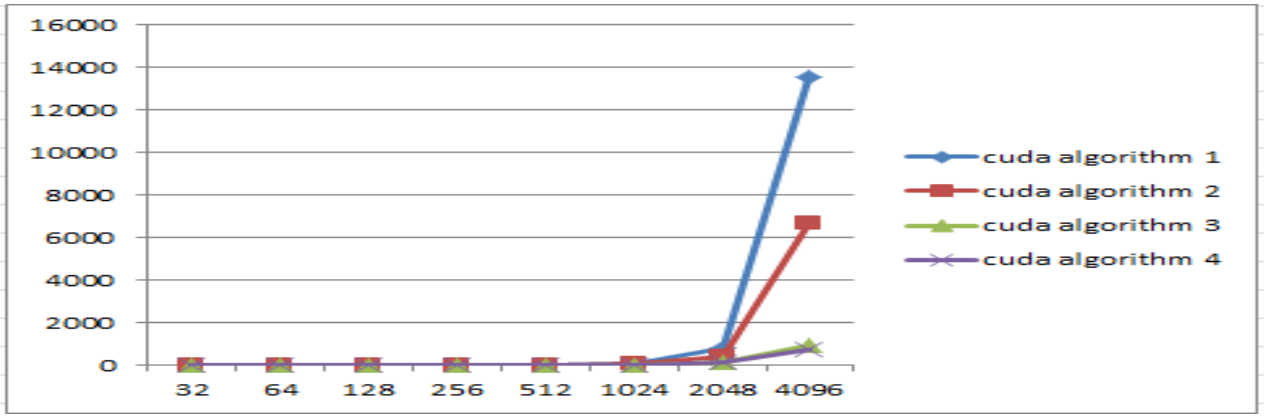
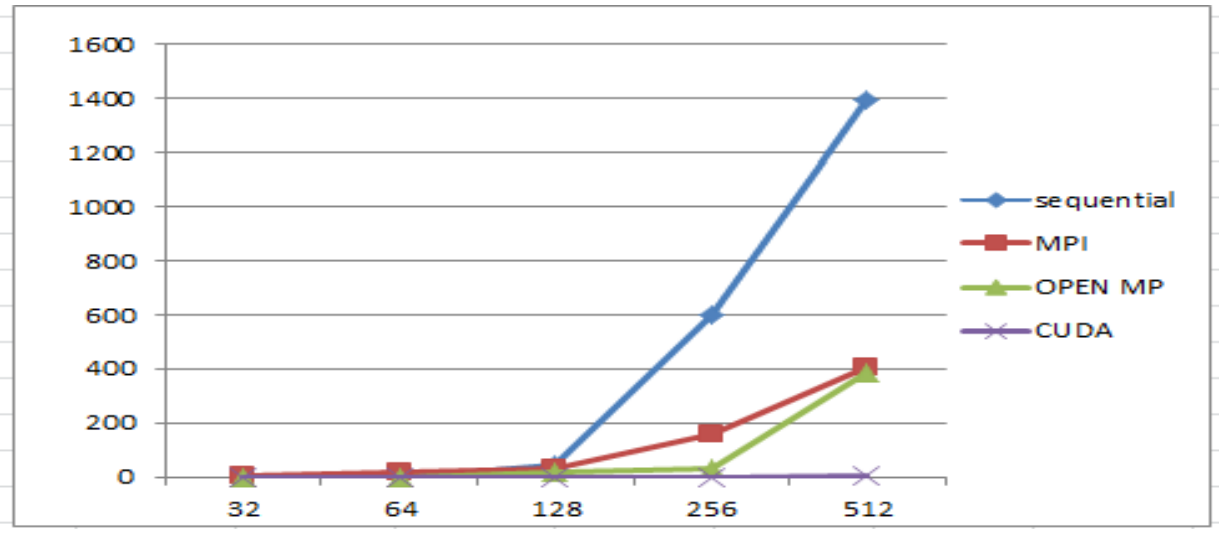


Figure (2)

CUDA vs OpenMp vs sequential: As explained in the previous section with tabular form the continuous plot for the matrix multiplication with comparison to open Mp, Mpi,sequential versus CUDA C is show in the plot below. Time measured in milliseconds on Y-axis. Number of elements used on x-axis.



..Figure (3)

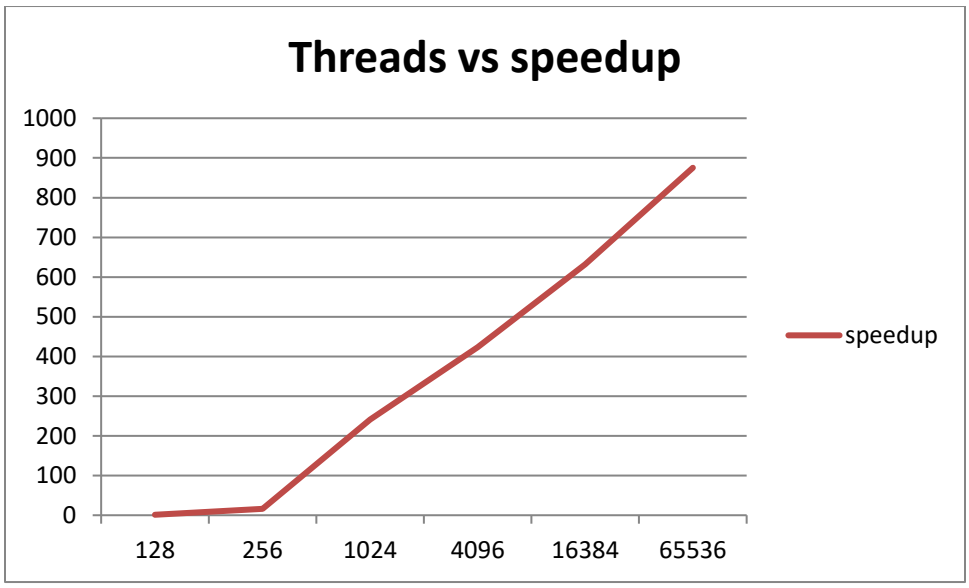
(f) Conclusions:

The work in this project represents a reference for efficient use of CUDA implementation of a typical real world problem: matrix multiplication and efficient use threads and blocks in CUDA kernels. These algorithms can be used in any types of platforms. Works similar to performance measured using GeForce GTX 980 Ti. The optimizations done in this project can be guidance for future implementation of CUDA kernels. To summarize the results from this project, the performance analysis showed that a speed up of 18.33 can be achieved by the sophisticated use of CUDA blocks and threads for a matrix of size 4096compared to naïve approach. By comparing CUDA C with open MP and sequential, MPI: we observed a speed up of 875.xx can be achieved by CUDA C compared to sequential algorithm and a speedup of 236.xx compared to Open MP.

The analysis show below is to demonstrate speedup produced by the usage of threads in matrix multiplication best performing CUDA algorithm out of 4 proposed algorithms in this project.

Number of threads used Vs speed up obtained:

Threads Used	speedup
128	1.235284486
256	16.20494567
1024	271.5989716
4096	423.2553668
16384	631.9785024
65536	875.2455529



(g) Future research work directions:

Matrix multiplication performed in this project achieved optimum levels of performance in terms of execution time and memory usage. Further research in this project leads to extending the algorithm to compute matrix multiplication with irregular tiles. We have to multiply matrices with different tile sizes across both rows and columns. This develops understanding of refined allocation techniques and load balancing techniques. Use matrices of different order such as Matrix 1 with order m*n and matrix 2 with order n*o. In this matrix multiplication algorithms in complicated machine learning algorithms like neural networks for example convoluted neural networks, back-propagation neural networks etc.,

(h) References

[1] Yegnanarayanan, V. (2013) “An Application of Matrix Multiplication.” Resonance 18: 368-377.

[2] Krishnan, Krishnan, and Jarek Nieplocha. (2004) “Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices”, in Proceedings, Tenth International Conference on Parallel and Distributed Systems, ICPADS 2004. pp. 257-266

[3] https://pdfs.semanticscholar.org/0d72/c1fe8df3b80aed5b067c05c35c1fd2863036.pdf

[4] https://hal.inria.fr/hal-02282529s

[5] Manojkumar Krishnan and J. Nieplocha, "SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. Santa Fe, NM, USA, 2004, pp. 70-, doi: 10.1109/IPDPS.2004.1303000.

[6] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.484.2206&rep=rep1&type=pdf