Date of execution: 18/09/2020

Date of submission 25/09/2020

Assignment A2

→ Title: Vector and Matrix Operations.

→ Problem Statement: Design parallel algorithm to:

1. Add two large vectors.
2. Multiply Vector and Matrix.
3. Multiply two NxN arrays using n^2 processors.

→ Objectives:

- To design and implement parallel algorithm using CUDA programming in order to perform the given vector and matrix operation.

→ Outcome:

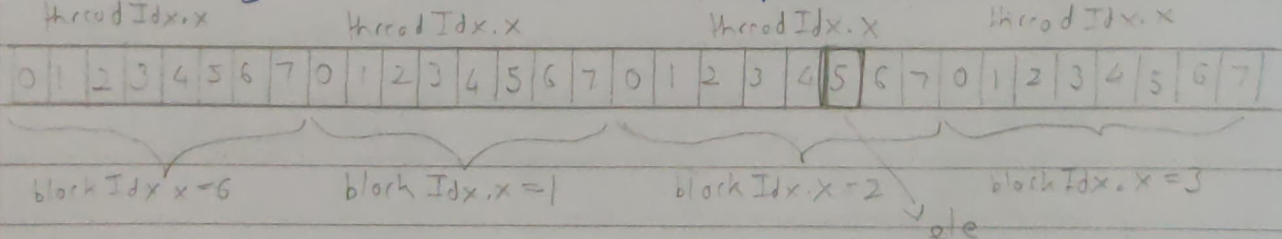
- Understood the concept of indexing arrays using blocks and threads.
- Implemented a CUDA program to perform the given vector and matrix operations.

→ Requirements: Google Colaboratory.

→ Theory:

Indexing Arrays with Blocks and Threads:

- Consider indexing an array with one element per thread (8 threads/block)



With M threads per block, a unique index of each thread is given by:

int index = threadIdx.x + blockIdx.x;

For the highlighted element (ele) index will be calculated as:

int index = threadIdx.x + blockIdx.x * M;

= 5 + 2*8;

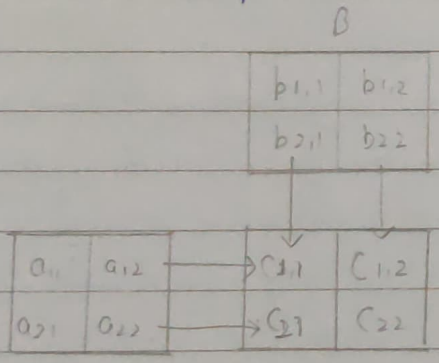
= 21

Here M can be replaced by built in variable blockDim.x for threads per block.

Matrix-Multiplication:

Consider two matrices ~~AXB~~ A and B. A is a nxm matrix, it has n rows and m columns. B is a mxw matrix. The result of multiplication C will be a nxw matrix.

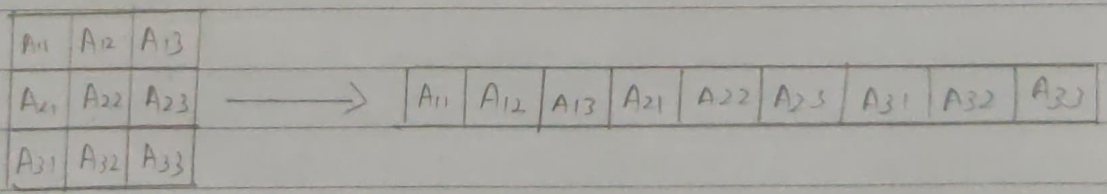
Normal Matrix Multiplication:



$AB = [c_{ij}] =$

$c_{ij} = a_{i1}b_{ij} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}$

Parallelization in matrix multiplication can be introduced by linearizing the multidimensional Array. The easiest way to do this is to stack each row length ways, from first to last.



As memory is contigous, this two dimensional array is really stored as one long one-dimensional array. It is stored in what's called row-order

meaning by row. In memory, the second row follows the first row and the third row follows the second row.

→ Algorithm:

① Sum of two vectors:

- Declare host and device identifiers
- initialize host variables
- copy values from host to device
- call device function with n blocks such that each block with threads_per_block = 25 and number of blocks = n/threads_per_block. such that the array will be devided into (n/ threads_per_block) blocks.

```
__global__ void add (int *a, *b, *c)
{
    index = blockIdx.x * blockDim.x + threadIdx.x
    if (index <= n) {
        result[index] = a[index] + b[index];
    }
```

② Matrix and Vector Multiplication: ~~& Matrix~~

- Declare host and device variables
- Initialize host variables
- Declare a grid corresponding to output matrix.
- Call device function with the following <<< blocksPerGrid, threadsPerBlock >>> where blocksPerGrid is a two dimensional collection of number of blocks in a grid. and threadsPerBlock is a 2d collection of number of threads per block.

```
__global__ void MatrixMultiplication(int *a, int *b, int *c , int m, int n
                                     int k)
{
  int row:= blockIdx.y * blockDim.y + threadIdx.y;
  int col:= blockIdx.x * blockDim.y + threadIdx.x
  int sum:=0
  if (col<k and row<m){
     for (int j:=0; j<n ;j++)
     {
        sum+:= a[row * n+j] * b[j*k +col]
     }
     c[k* row +col]:=sum
  }
}
```

3) Vector Multiplication:

- Declare host and device variables
- Initialize host variables
- ~~Declare a grid corresponding to output matrix.~~
- Call device function with the following configuration: << number_of_block, number_of_threads_per block>>

```
__global__ void matrixVector(int *vec, int*mat, int * result,int n, int m)
 {
    tid:= blockIdx.x * blockDim.x + threadIdx.x.
    sum:=0
    if (tid <=n) {
       for ( int i:=0 to n )
          sum+= vec[i] * mat[(i*m) +tid]
       }
    result[tid]=sum
  }
}
```

Test Cases:

	Operation	Input	Output (Actual)	Expected Output	Result
①	Vector Sum	① 84, 87, 78, 16, 94, ... ② 96, 71, 35, 79, 68, ...	180, 158, 113, 95, 162, ...	180, 158, 113, 95, 162, ...	Pass
②	Vector Matrix multiplication	Vector: 2, 2, 1 Matrix: 2, 3, 2, 2 1, 1, 2, 3 2, 3, 2, 3	8, 11, 10, 13	8, 11, 10, 13	Pass
③	Matrix Multi-plication	Mat 1: 4, 7, 8, 6 4, 6, 7, 3 10, 2, 3, 8 1, 10, 4, 7 1, 7, 3, 7 Mat 2: 2, 9, 8 10, 3, 1 3, 4, 8 6, 10, 3	Matrix: 138, 149, 121 107, 112, 103 97, 188, 130 156, 125, 71 123, 112, 60	Matrix: 138, 149, 121 107, 112, 103 97, 188, 130 156, 125, 71 123, 112, 60	Pass

Conclusion: Successfully implemented parallel algorithms for the given tasks using CUDA programming.