**Problem definition:**

This problem is matrix-matrix multiplication (i.e. multiplying 2 matrices). This is done by getting the dot product of i’th row of first matrix with j’th column of second matrix and saving the result to the (i,j) position of the resulting matrix. Furthermore the resulting matrix has the same number of rows as the first matrix and same number of columns as the second matrix.

**Algorithm Description:**

*(the algorithm checks for CUDA errors after each CUDA operation)*

*Kernel() algorithm:*

* Calculate a unique integer idx for each thread.
* If idx is less than maxIndex(=numberOfElements) (which equals to number of multiplications needed to be done) continue algorithm, else stop.
* Calculate the positions of 1 element on each input matrix: (assume i1 and i2 Is row count of matrix 1 and matrix 2, j1 and j2 is column count of matrix 1 and matrix 2)
  + The first position(on matrix 1) is at ***idx % j1 + j1\*(idx/(i2\*j2))***
  + The second position(on matrix 2) is ***at (idx%(i2\*j2))/i2 + (idx%i2)\*j2***
* Use atomicAdd function to add the multiplication of each element in respective matrices respective position to resulting matrices ***idx / i2*** position.

*Main() algorithm*:

* Set the matrix sizes, then create and allocate 2 matrices (input matrices)- these are allocated like an array-.
* Set the values of the matrices.
* Calculate numberOfElements then set blockSize and numberOfBlocks. numberOfElements is equal to total number of multiplications which is equal to number of rows of first matrix times number of elements of second matrix.
* Create 3 matrix pointers(on device). 2 of them are input matrices and last one is resulting matrix.
* Allocate memory for matrices on device.
* Set the resulting matrices values to 0 since I’ll be using atomicAdd function.
* Execute kernel(MatrixMultiplication )with variables of( in order) : pointer to first matrix(on device), pointer to second matrix(on device), pointer to resulting matrix(on device), number of rows of first matrix, number of columns of first matrix, number of rows of second matrix, number of columns of second matrix, numberOfElements.
* Copy the resulting matrix to a new matrix on host after allocating space.
* Print the resulting matrix.
* Free all pointers.

**Benchmarking:**

Different size of matrices would not increase or decrease performance in any unexpected way (they only increase decrease performance depending on their size) since I am using a different thread for each multiplication of elements of matrices.

**Pros-cons:**

I am not considering bank conflict which probably is decreasing performance.

I am freeing pointers at the end of main which unnecessarily hold memory for some pointers. Freeing them in appropriate positions would be better memory-wise.

**Discussion:**

I am using different threads for every single elements’ multiplication which greatly increases performance.

**Environment:**

*DeviceQueryResult is included as a text file*

NVIDIA GeForce GTX 1060 3GB (compute capability 6.1)

Intel Core-I3 7100 @3.9GHz \w 8GB DDR4 RAM

Windows 10 Education Version 1809 (OS Build 17763.55)

Nvcc V10.0.130 compiler / Visual studio 2017 Community

**NOTE:**

Since I am using an atomic function you compile with something like nvcc -arch compute\_11 cuda\_hw2.cu or in visual studio in project properties select appropriate Code Generation (mine is default at compute\_35, sm\_35 with visual studio 2017)

You can give me any unoffical comment/criticism via e-mail( [e223789@metu.edu.tr](mailto:e223789@metu.edu.tr) )