**COMPUTER ARCHITECTURE ASSIGNMENT 2**

**Name:** Chinthakommadinne Vinay Kumar

**Roll No.:** B210554CS

**Section:** CS03

CUDA is a programming language that uses the Graphical Processing Unit (GPU). It is a parallel computing platform and programming. This allows computations to be performed in parallel while providing well-formed speed.

* GPUs run one kernel (a group of tasks) at a time.
* Each kernel consists of blocks, which are independent groups of ALUs.
* Each block contains threads, which are levels of computation.
* The threads in each block typically work together to calculate a value.
* Threads in the same block can share memory.

In our code we use \_\_global\_\_ for GPU computing tasks. That is

* A \_\_global\_\_ function is a kernel function that runs on the GPU but is invoked from the CPU (host).
* These functions execute in parallel using multiple threads.

In our assignment we have to solve system of linear equations AX=B using LU decomposition method.

In linear algebra, **LU decomposition** is a fundamental algorithm used to solve systems of linear equations by breaking a matrix A into the product of a **lower triangular matrix L** and an **upper triangular matrix U**. When the matrix size N is large, the computational workload increases significantly. In such cases, parallelization becomes essential to achieve faster execution.  
So, we use support of GPU by parallelizing the computations using CUDA programming.

For LU decomposition, I used gaussian elimination method to get L and U matrices. After getting L and U matrices, X can be obtained from backward substitution of **UX=Y,** Y can be obtained from forward substitution of **LY=B.**

**For Input file – input10.txt**

We have following timing information:

* Time to read A and B matrices: 0.000214 seconds
* Time to compute L matrix and U matrix: 0.000689 seconds
* Time to compute Y matrix and X(solution) matrix: 0.000106 seconds
* Total time to solve system of equations: 0.001009 seconds

**For Input file – indata50.txt**

We have following timing information:

* Time to read A and B matrices: 0.003155 seconds
* Time to compute L matrix and U matrix: 0.004575 seconds
* Time to compute Y matrix and X(solution) matrix: 0.000498 seconds
* Total time to solve system of equations: 0.008229 seconds

**For Input file – indata80.txt**

We have following timing information:

* Time to read A and B matrices: 0.006765 seconds
* Time to compute L matrix and U matrix: 0.015587 seconds
* Time to compute Y matrix and X(solution) matrix: 0.000937 seconds
* Total time to solve system of equations: 0.023289 seconds

**For Input file – indata100.txt**

We have following timing information:

* Time to read A and B matrices: 0.010309 seconds
* Time to compute L matrix and U matrix: 0.031160 seconds
* Time to compute Y matrix and X(solution) matrix: 0.001289 seconds
* Total time to solve system of equations: 0.042758 seconds

**For Input file – indata200.txt**

We have following timing information:

* Time to read A and B matrices: 0.040118 seconds
* Time to compute L matrix and U matrix: 0.264530 seconds
* Time to compute Y matrix and X(solution) matrix: 0.003334 seconds
* Total time to solve system of equations: 0.307981 seconds

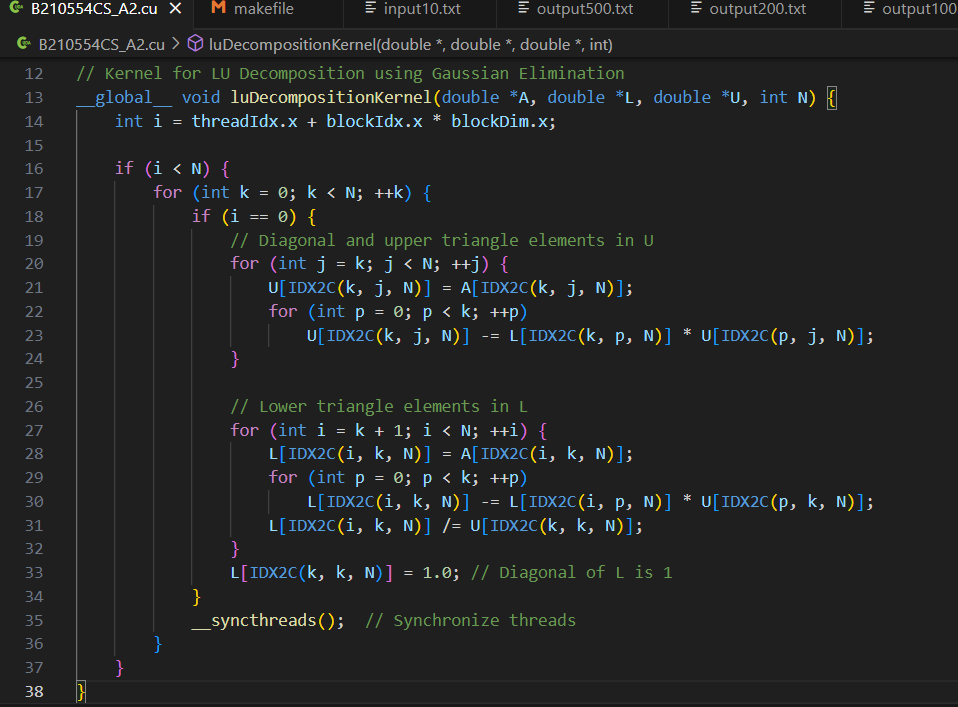
**For Input file – indata500.txt**

We have following timing information:

* Time to read A and B matrices: 0.247719 seconds
* Time to compute L matrix and U matrix: 4.503295 seconds
* Time to compute Y matrix and X(solution) matrix: 0.029358 seconds
* Total time to solve system of equations: 4.780372 seconds

**a. Parallel algorithm used to implement the solution**

To implement the solution we have used LU decomposition as parallel algorithm where multiple threads cooperate to compute the elements of the lower triangular matrix (L) and upper triangular matrix (U).



This code basically performs LU decomposition That is we can write

A = L\*U

***int i = threadIdx.x + blockIdx.x \* blockDim.x;***

This index i corresponds to a specific row in the matrix, allowing the parallel processing of rows.

* After that we can see that L and U matrices are computing parallelly   
  The algorithm proceeds column by column (k), calculating the elements of U (upper triangular) and L (lower triangular) for each column.
* Threads computes the elements in the k th row and all columns j >= k of the upper triangular matrix U
* Threads also computes the elements below the diagonal for the k th column
* The diagonal element of L is explicitly set to 1

After each column's elements are computed, the kernel calls ***\_\_syncthreads()*** to synchronize all threads.

**b. Kernel configuration (grid and block sizes) for each of the kernels called.**

Here I have used 3 Kernels :

1. **\_\_global\_\_ void luDecompositionKernel()**

threadsPerBlock = 256

blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock; d\_U, N)

*luDecompositionKernel<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_L, d\_U, N);*

* Grid size (blocksPerGrid): It depends on the size of the matrix N. The formula ensures that we have enough blocks to cover all the rows, with each block handling threadsPerBlock (256) rows. It divides the problem into smaller chunks of 256 threads.
* Block size (threadsPerBlock): 256 threads are used per block. Each thread will handle one row of the matrix (depending on the matrix size N).

1. **\_\_global\_\_ void CalculateYKernel()**

*forwardSubstitutionKernel<<<1, 1>>>(d\_L, d\_B, d\_Y, N);*

* Grid size: 1 block is used, which means the computation is handled by a single block.
* Block size: 1 thread is used in this kernel. This is a sequential process where one thread will handle the forward substitution for the entire matrix L. Since forward substitution has dependencies (each row depends on the result of the previous one), it is implemented sequentially and doesn't benefit from parallelization.

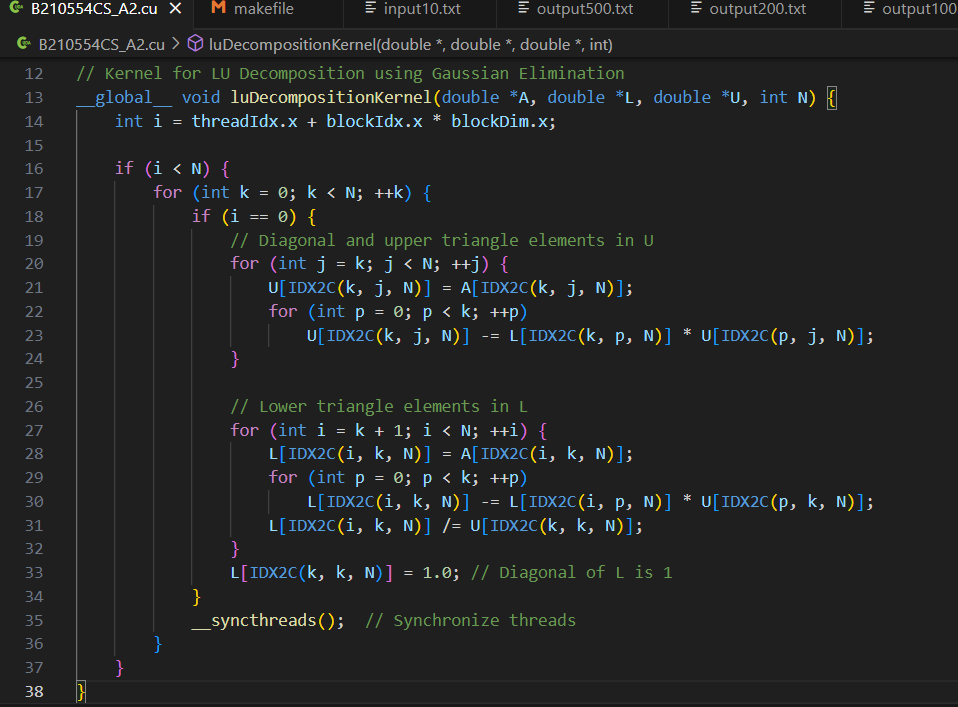
1. **\_\_global\_\_ void CalculateXKernel()**

*backwardSubstitutionKernel<<<1, 1>>>(d\_U, d\_Y, d\_X, N);*

* Grid size: Again, 1 block is used.
* Block size: 1 thread is used, making the backward substitution sequential like forward substitution. Each row depends on the next, so only one thread is employed to handle the entire matrix U.

**c. CGMA value of each kernel called.**

CGMA stands for Compute to Global Memory Access and is defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

1. **For \_\_global\_\_ void luDecompositionKernel()**

For each itetration,

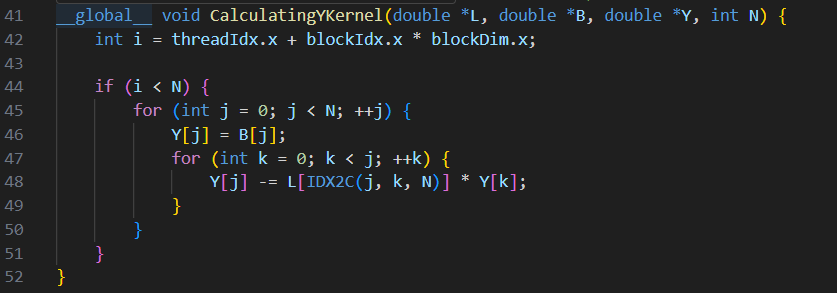
In line 23 we have 2 floating point calculations one is multiplication(\*) and subtraction(-), and in line 30 we have 2, and line 31 we have 1 calculation.

So, total of 2+2+1 = 5 floating point calculations.

Then memory access are 8.

Then CGMA value for this is 5/8 = 0.625

**b. \_\_global\_\_ void CalculateYKernel()**

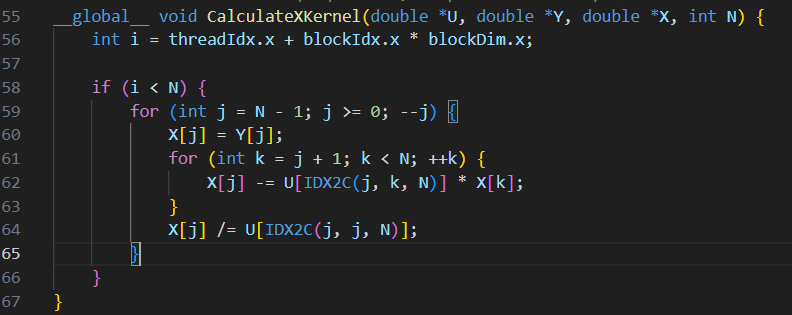
****

Here, number of floating point operations are = 2

Memory accesses are = 3

So, CGMA = 2/3 = 0.67

**c. \_\_global\_\_ void CalculateXKernel()**

****

Here, No of floating point operations are = 3

No. of memory accesses are = 4

So, CGMA = 3/4 = 0.75

**d. Discuss different types of synchronizations used in the solution and their impact on the performance.**

We have used many synchronizations in our code.

synchronization plays an essential role in ensuring that all threads operate correctly on the shared data during various stages of matrix operations.

1. **Implicit Synchronization in Kernel Launch**

Every time we launch a new kernel (e.g., the LU decomposition, forward substitution, or backward substitution kernels), there’s an implicit synchronization between kernel launches. This ensures that one kernel completes its entire execution before the next one starts. For instance, LU decomposition must finish before forward substitution begins, and forward substitution must complete before backward substitution can start.

The impact caused by this synchronization is as follows,

It is necessary for correctness, it introduces latency as no new computations begin until the previous kernel fully completes.

1. **Thread Synchronization with *\_\_syncthreads()***

Synchronizes threads within a block. It ensures that every thread has finished its current step before any thread proceeds to the next. This is crucial for ensuring that the upper (U) and lower (L) triangular matrices are fully updated by all threads at each iteration of the outer loop before moving to the next iteration.

Using *\_\_syncthreads()* ensures that all threads in a block are working with up-to-date data by pausing them until every thread reaches the same point. However, frequent use can slow down performance, as faster threads have to wait for slower ones to finish, reducing the overall efficiency in compute-heavy sections.

1. **Host-Device Synchronization with *cudaDeviceSynchronize()***

After launching each kernel, a *cudaDeviceSynchronize()* call is used to make sure the host waits until all threads on the device have finished their execution before moving on to the next stage. This ensures that the data processed by one kernel is completely written back to the global memory and is available for use by subsequent kernels.

This introduces a stall in the CPU execution as it waits for the GPU to finish its work. Frequent use of *cudaDeviceSynchronize()* adds to the overhead, which may affect performance, especially if there are many small kernel launches that could have been overlapped.