# CS410 Assignment 3

## Report

## 210010014

## Part 1 : Matrix Multiplication on GPU

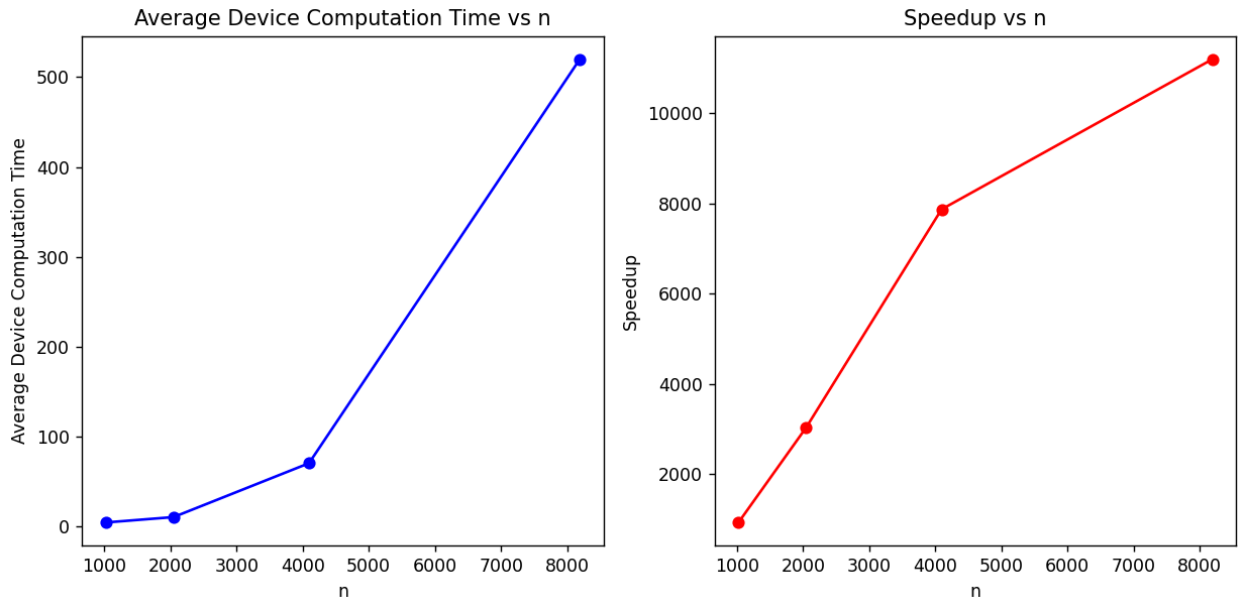
**How does my program work?**

- create and allocate matrices on the host.
- allocate memory on device (GPU) and use **cudaMemcpy** to copy allocated matrices on host to device.
- Dimensions :
    a. **dim3 threadsPerBlock(32, 32):** it specifies that each block will contain 32x32 = 1024 threads.
    **b. dim3 blocksPerGrid(ceil(n/32.0), ceil(n/32.0)):** allocate ceil(n/32) blocks along both x and direction in a grid.
- Launch CUDA kernel function matrixMul with the specified grid and block dimensions
- kernel function, calculates dot product of ith row of devA and jth column of devB and stores it in devC.
- At last, copy the data from device to host using **cudaMemcpy** and cudaMemcpyDeviceToHost flag , and free memory on host's heap and device memory.

**How does my program exploit parallelism?**

- CUDA allows us to launch multiple threads in parallel, each responsible for computing a portion of the output matrix. In our implementation, each thread computes a single element of the output matrix.

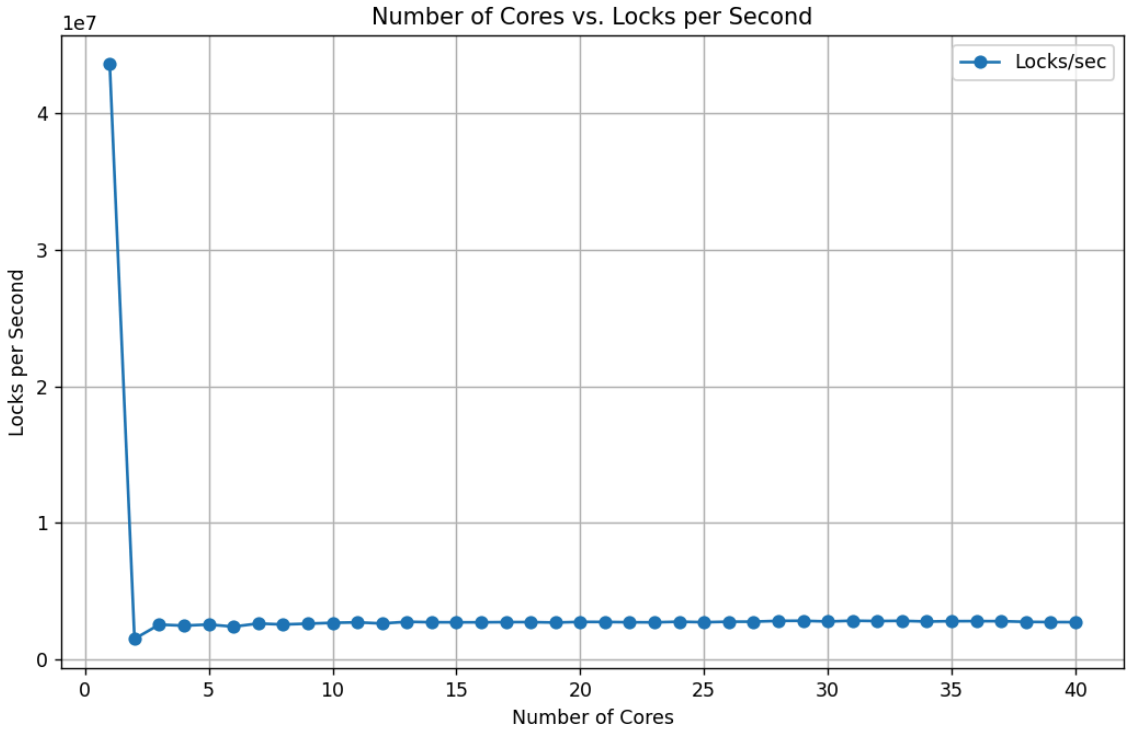
| n | avg. device computation time | avg. device computation time including data transfer time | avg.Computation time on host | Speedup |
|---|---|---|---|---|
| 1024 | 4 | 7 | 3708 | 927 |
| 2048 | 10 | 19 | 30318 | 3032 |
| 4096 | 70 | 106 | 550677 | 7866 |
| 8192 | 520 | 628 | 5819870 | 11192 |




**Why does speedup increase as n increases?**

There is no control divergence in the kernel code when n is a multiple of 32. So all threads in the same warp get executed in one **SIMD** instruction. While the serial code executes one instruction at a time.

# Part 2 : K42 MCS Lock




On a single core, lock performs excellently. When we execute it on multiple cores, locks per second doesn't vary much, which is a good indication as its performance is not decreasing when we increase threads from 2 to 40.

**leanings from implementing the K42 lock:**

1. Atomic operations (Compare and store in our case) make sure that several threads can safely update the same piece of data without introducing race conditions or corrupting the data. Atomic operations are used in the K42 lock implementation to manipulate pointers and boolean flags in an atomic manner, guaranteeing proper synchronization.
2. The K42 lock controls access to the crucial area through a queue-based mechanism. When a thread tries to obtain the lock, it queues up and the lock is passed from one thread to another in a first-in-first-out (FIFO) order. This method guarantees equitable thread scheduling while assisting in the reduction of contention.

I tried, but failed to **properly** understand the idea behind the k42 algorithm.