Assignment 3

Question 1

Efficiency of the approaches of a b and c are interesting. I have setup everything locally on my computer so timings are going to be different than if it were on the WLU servers.

When N is 65536 for my serial CPU calculation, it took 44.4s to complete its execution of finding the minimum distance between points. When N is 2048, execution time is 47ms. So we can see that when N increases, execution time is exponentially slower. (N increased by x32, execution time increased by ~940%).

When N is 65536 for part b, execution time is 12 ms, which is dramatically better than CPU serial algorithm. When N is 2048, execution time is 0.31ms which is still dramatically better than the CPU. In order to get the execution time to 30s, N had to be of 2048 x 2048 size, and it took 36.35s to compute.

When N is 65536 for part c, I actually run out of memory, so I can only base it off of when N is 16384, which takes 7.6ms of execution. Surprisingly, the b approach is faster at this N value than the c approach. Approach b was able to execute its calculation in 2.2ms when N is 16384, which is 5ms faster. Approach c runs out of memory because every combination is added to cudaMem, which is N x N – 1. However, when dealing with lower N numbers, such as 4096, approach c does compute faster than approach b. In this instance, approach c was 0.1ms faster on average.

I noticed when executing part c, the result isn’t accurate. I’m using an atomicMin reduction approach, but I think there are too many atomic collisions that change the result, as I noticed the global min result does report back within an iteration (with console logging), but is then overwritten by a larger result unfortunately.

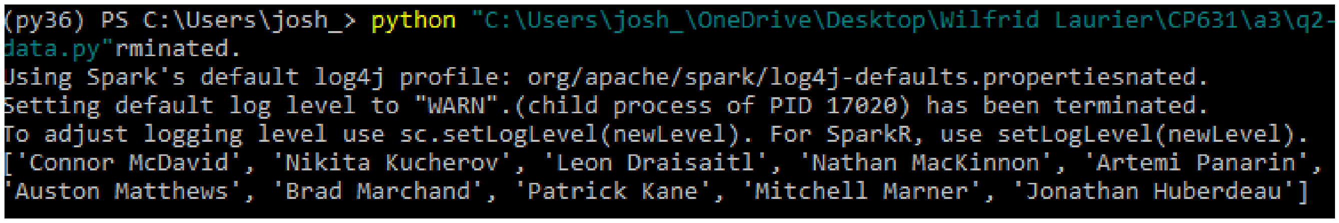
Considering the limitations of my personal computer, I believe approach c is supposed to be the fastest, but the resources needed are way more than approach b requires and I believe it hinders approach c’s performance.

To speed things up, there are libraries out there such as CUB, which help pick the best algorithm for reductions. However there are different techniques on how to reduce the execution time without the need of an independent library, or by using atomic operations as I did. There is a shuffle warp reduce approach, which shifts down on memory values with a comparison on each thread within a warp. It is a common and very effective approach, which honestly could solve my accuracy issue with approach c. This approach is notably faster as there isn’t any blocking, as a common approach is to use mutexes. Using mutexes to lock threads can hinder performance. How the algorithm is written also matters. Math operations are very slow, and short cuts can be taken to help speed things up. For example, squaring and using the power of 2 on numbers is expensive for the point to point distance algorithm. On approach b, I adjusted the distance algorithm to (x_dist * x_dist) + (y_dist * y_dist) and it saved 5ms on execution time when N was 65536, averaging ~7ms on execution.

Question 2

I chose to grab NHL statistics. The data available from the api endpoints is from the years 2018 – 2021. Each year has its own api response page and has all the player stats associated with its corresponding year.

I decided to accumulate all points for all seasons for each player. Then I averaged out each players points per game. I then printed the top ten performers over the years 2018 – 2021. This can be seen in Figure 1.



```
(py36) PS C:\Users\josh_> python "C:\Users\josh_\OneDrive\Desktop\Wilfrid Laurier\CP631\a3\q2-
data.py"rminated.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.propertiesnated.
Setting default log level to "WARN".(child process of PID 17020) has been terminated.
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
['Connor McDavid', 'Nikita Kucherov', 'Leon Draisaitl', 'Nathan MacKinnon', 'Artemi Panarin',
'Auston Matthews', 'Brad Marchand', 'Patrick Kane', 'Mitchell Marner', 'Jonathan Huberdeau']
```

Figure 1. The top 10 performers in the NHL over the 2018-2021 span are listed. This was done by grabbing the average points per game per player, with the player playing more than 50 games.

I also was curious about how many goals per hockey team was scored since 2018. I accumulated all goals per player and associated that value to each designated team since 2018. This can be seen in Figure 2.

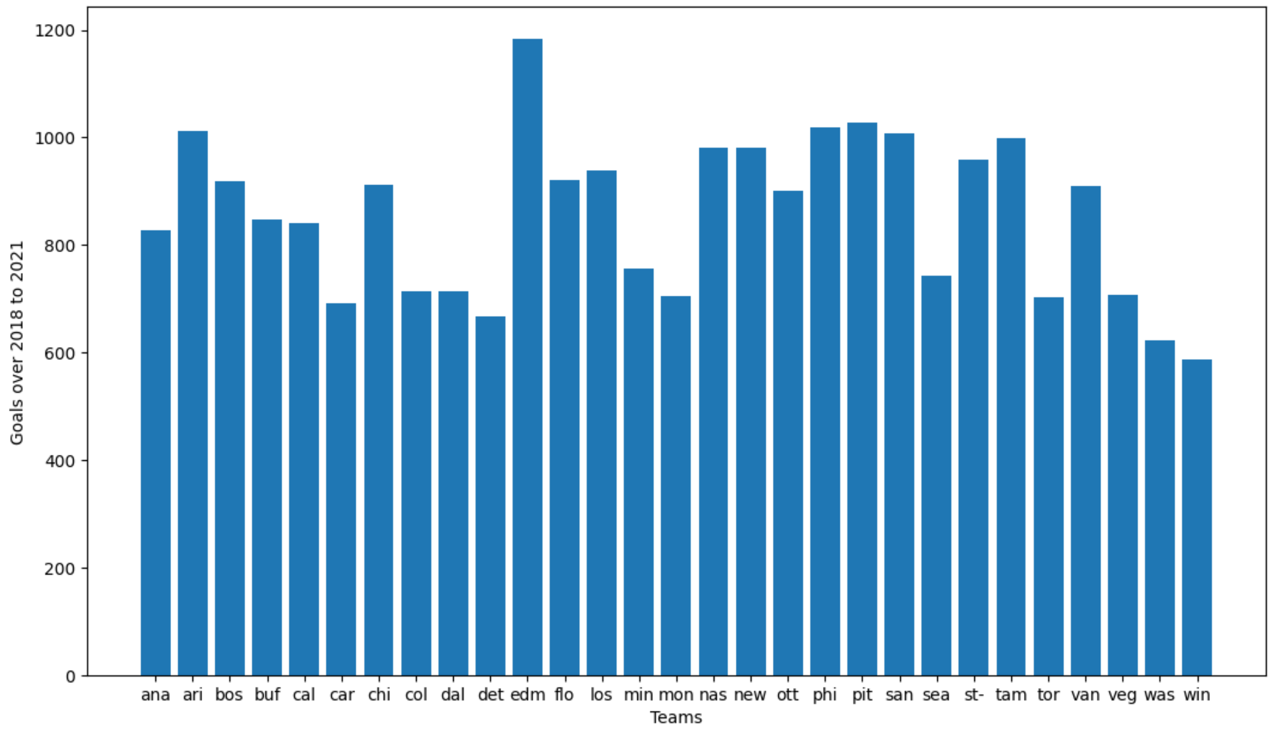


Figure 2. Goals per team over the span of 2018 – 2021.

I was then also curious about how many goals were scored per season. So I have a scatter plot of goals per year, and possible correlations including penalty minutes and power play goals. As seen in Figure 3.

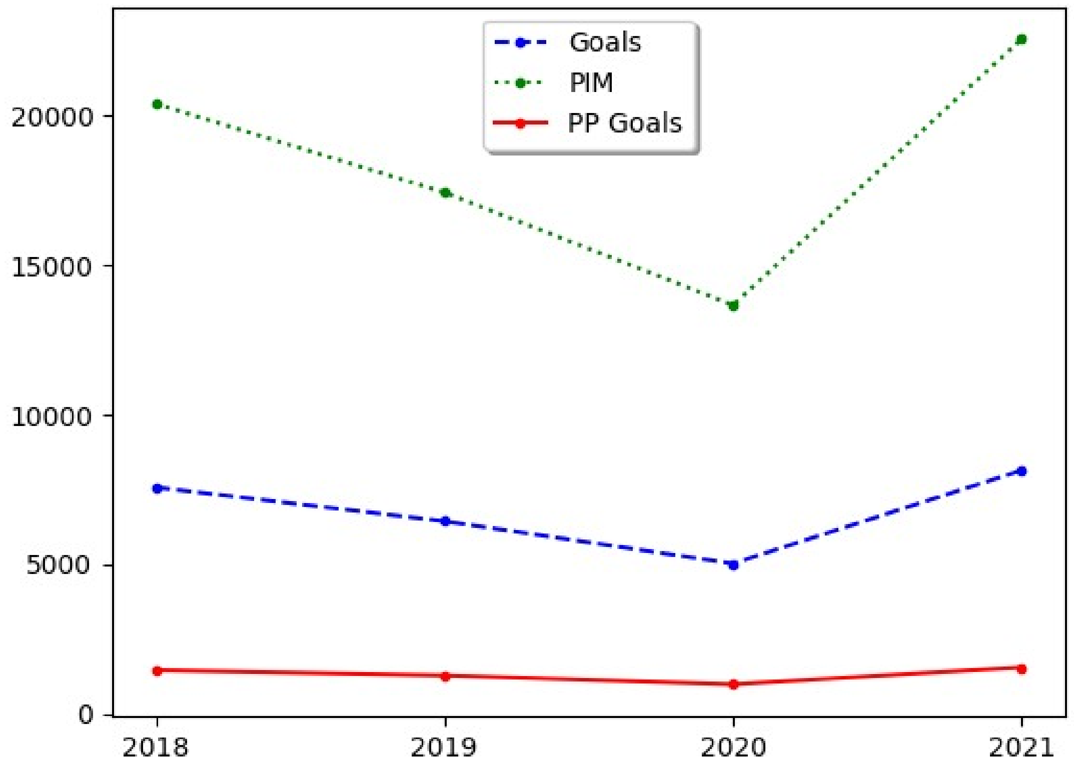


Figure 3. Season totals of goals scored in blue, penalties in minutes in green, and power play goals in red, over the span of 2018 – 2021.