ES215

Assignment 4

Name: Prakram Rathore

Roll No.: 20110141

Q1.

Given that we have 5 stage pipelines with each stage taking one cycle. It means without any dependencies; each instruction will be executed in one cycle.

Every RAW dependency will have 3 Stalls without considering any optimizations. It means every instruction having RAW dependency will take 4 cycles to execute.

Every branch dependency will have 2 stalls without considering any optimizations like using extra hardware on instruction decode stage. It means every instruction having branch dependency will take 3 cycles to execute.

$Speedup = \frac{CPI(initial)}{CPI(calculated)}$, where initial CPI is the CPI when there is no dependency, and calculated CPI is when we are considering given dependencies. This speedup is with respect to ideal case.

We can beforehand say that the speedup will be less than 1 because when there is dependency than CPI will increase. It means time of execution will increase because of Stalls.

1. **Without branch predictor:**
   a. **30% RAW and 20% branch dependency**

CPI = 30% * Cycle for RAW dependency + 20% * Cycle for branch dependency + 50% * Cycle for no dependency instruction

$$CPI = \frac{30}{100}(4) + \frac{20}{100}(3) + \frac{50}{100}(1)$$

$$CPI = 2.3$$

$$Speedup = \frac{1}{2.3} = 0.43478$$

b. **40% branch dependency**

CPI = 40% * Cycle for branch dependency + 60% * Cycle for no dependency instruction

$$CPI = \frac{40}{100}(3) + \frac{60}{100}(1)$$

$$CPI = 1.8$$

$$Speedup = \frac{1}{1.8} = 0.5556$$

Comparing part, A and B:

$Speedup\ of\ A\ with\ respect\ to\ B = \frac{1.8}{2.3} = 0.783$, It means time of execution is more when there are 30% Raw and 20% branch dependency as compared to that when 40% branch dependency. It's obvious because number of stalls in B is less than that in A.

2. **With branch predictor:**

a. **30% RAW and 20% branch dependency**

Prediction will help in optimization because if we correctly guess than CPI or Cycles for execution will be 1 and if we incorrectly predict then the CPI or Cycles for execution will be 3 for branch dependency.
There will be no effect on RAW dependency.
Given that our prediction is 80% accurate.

CPI = 30% * Cycle for RAW dependency + 80%*20% * Cycle for branch dependency and Correct guess + 200%*20% * Cycle for branch dependency and Incorrect guess + 50% * Cycle for no dependency instruction

$$CPI = \frac{30}{100}(4) + \frac{80}{100} \times \frac{20}{100}(1) + \frac{20}{100} \times \frac{20}{100}(3) + \frac{50}{100}(1)$$

$$CPI = 1.98$$

$\boldsymbol{Speedup} = \frac{1}{1.98} = 0.505$, This speedup is with respect to ideal case where there is no dependency.

Comparison in part A, with and without prediction:

$\boldsymbol{Speedup}\left(\frac{\boldsymbol{with\ prediction}}{\boldsymbol{without\ prediction}}\right) = \frac{2.3}{1.98} = 1.1616$ **,** It means with prediction system is faster than without prediction.

b. 40% branch dependency

Prediction will help in optimization because if we correctly guess than CPI or Cycles for execution will be 1 and if we incorrectly predict then the CPI or Cycles for execution will be 3 for branch dependency.
There will be no effect on RAW dependency.
Given that our prediction is 80% accurate.
CPI = 80%*40% * Cycle for branch dependency and Correct guess + 20%*40% * Cycle for branch dependency and Incorrect guess + 60% * Cycle for no dependency instruction

$$CPI = \frac{80}{100}\frac{40}{100}(1) + \frac{20}{100}\frac{40}{100}(3) + \frac{60}{100}(1)$$

$$CPI = 1.16$$

$\boldsymbol{Speedup} = \frac{1}{1.16} = 0.8620$, This speedup is with respect to ideal case where there is no dependency.

Comparison in part B, with and without prediction:

$\boldsymbol{Speedup}\left(\frac{\boldsymbol{with\ prediction}}{\boldsymbol{without\ prediction}}\right) = \frac{1.8}{1.16} = 1.552$ **,** It means with prediction system is faster than without prediction.

Comparing part, A and B with prediction:

$Speedup\ of\ A\ with\ respect\ to\ B = \frac{1.16}{1.98} = 0.585$, It means time of execution is more when there are 30% Raw and 20% branch dependency as compared to that when 40% branch dependency. It’s obvious because number of stalls in B is less than that in A. Also, this time A is taking more time with resect to B then it was taking without prediction, it means prediction improves the branching dependency.

Q2.

Given that 20% of the total instructions are branch instructions and we are going to use branching with one delay slot by using extra hardware in decode stage. It means every time there will be a branch instruction there will be 1 delay slot not 2 and it will be NOP instruction if there is no instruction available for execution in delay slot, it means 1 stall otherwise it will execute other instruction in that delay slot. It's given that 85% of the delay slot is fill. Also, given that base CPI = 1.5, where base case is when we do not use any delay slot and we put a stall for every delay slot.

Note: We need to calculate the average time of execution of an instruction, we cannot simply take it as 1 because it's not given. For that we'll use the base case.

Now, if there are 100 instructions than 20 instructions will create 1 stall.

It means, total number of stalls per instruction =0.8*0+0.2*1= 0.2 stall/instruction

It means average total time of execution of every instruction = x + 0.2 Cycle, where x is the average time per instruction was taking to complete without any stall. We are calculating this for base case.

Given that,

$$base\ case\ CPI = 1.5 = x + 0.2$$

$$x = 1.3$$

Now we need to estimate the CPI if compiler is able to fill 85% of the delay slots, it means 15% as stalls.

So, total number stalls per instruction = 0.2*0.85*0+0.2*0.15*1 = 0.03

It means average total time of execution if every instruction = x + 0.03 = 1.3+0.03 = 1.33

Q3.

Note: Time for C++ is in nanoseconds whereas for pythons its in seconds.

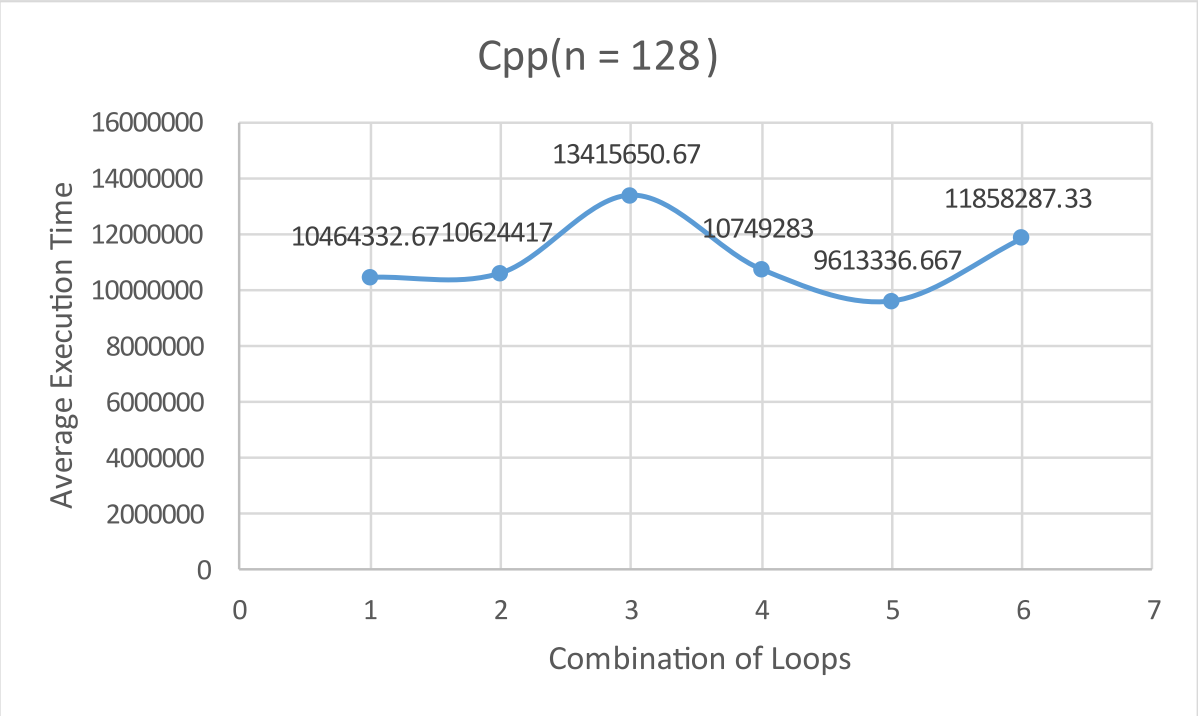
Bucket 1 => C++

a) and b)

1. For n = 128

Cpp (n = 128)

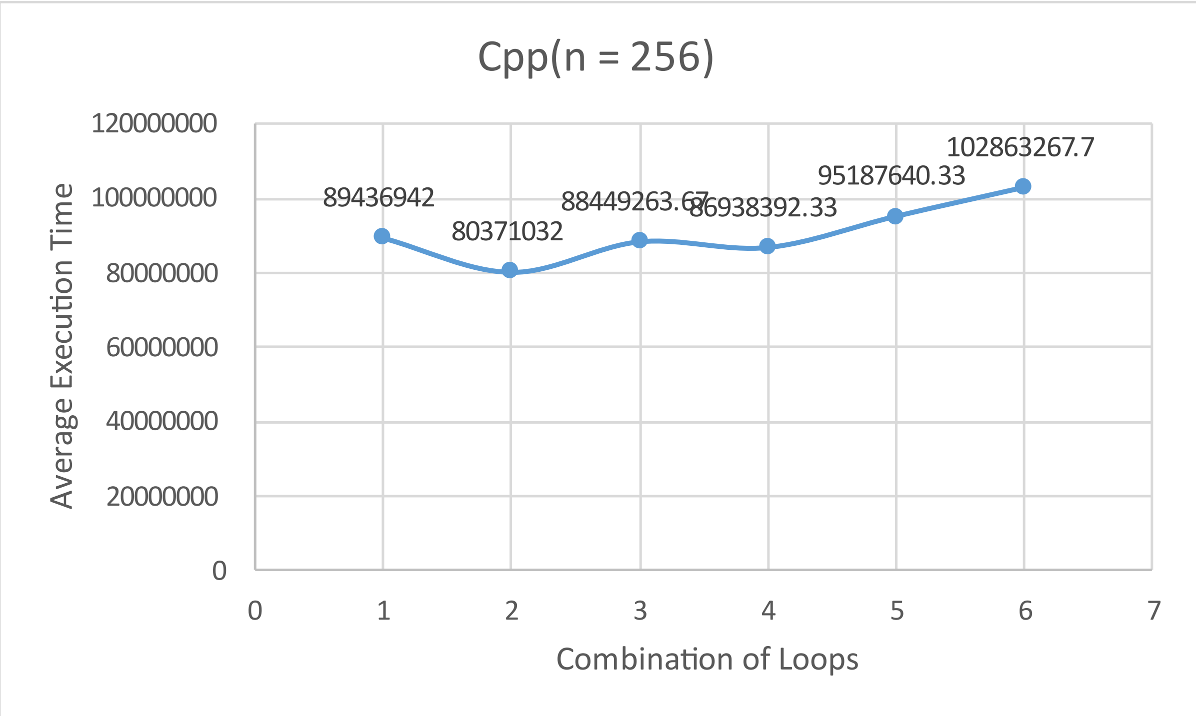
Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	10893892	12035757	8463349	10464332.67
2	7950643	13391582	10531026	10624417
3	13137783	11721167	15388002	13415650.67
4	8529314	12242989	11475546	10749283
5	7812729	13628168	7399113	9613336.667
6	12152340	11621200	11801322	11858287.33



2. For n = 256

Cpp(n = 256)

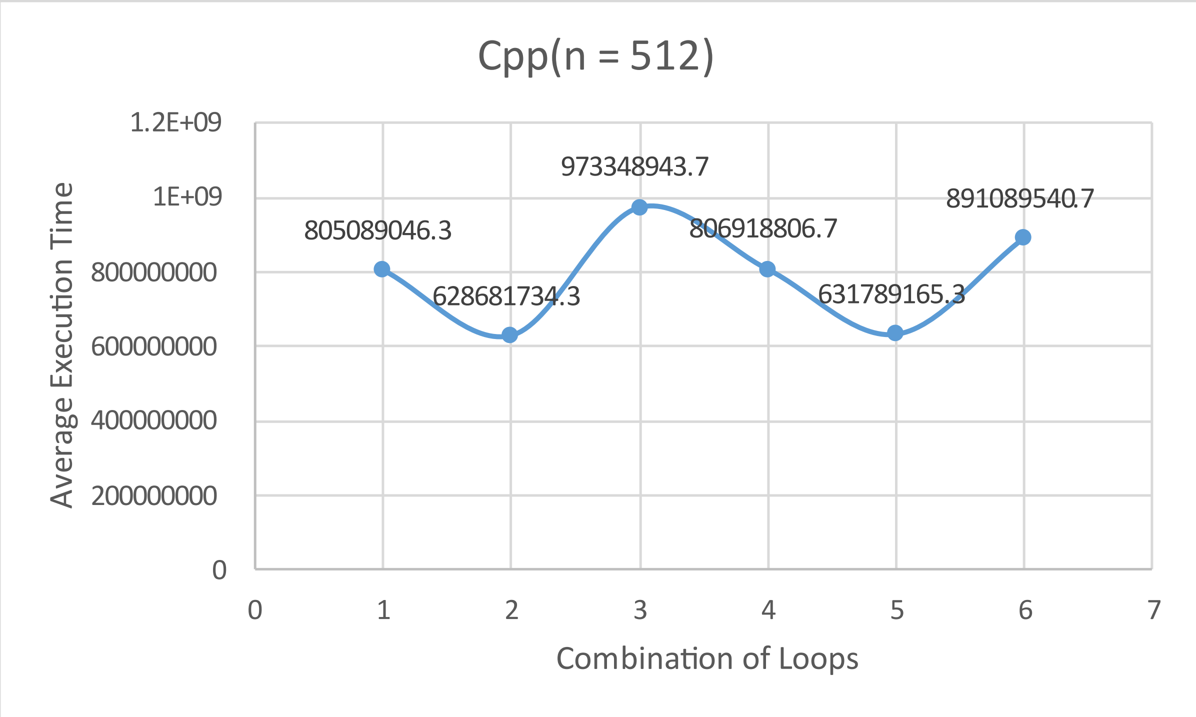
Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	97593757	95039831	75677238	89436942
2	63546782	104633077	72933237	80371032
3	91940482	88847002	84560307	88449263.67
4	98259640	80803206	81752331	86938392.33
5	98493920	94729967	92339034	95187640.33
6	102866231	101498210	104225362	102863267.7



3. For n = 512

Cpp(n = 512)

Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	844272900	849094277	721899962	805089046.3
2	616200808	674869230	594975165	628681734.3
3	1010269441	1023403795	886373595	973348943.7
4	769879617	829483766	821393037	806918806.7
5	564534230	654667855	676165411	631789165.3
6	847145264	981341641	844781717	891089540.7



Here we can see that for different positions of three for loops we are getting different answers. This might be due to cache memory and how it stores the data which is most likely to be used again. In general, if we are handling arrays then cache will contain the elements which are closer to the current index. So, if we are at index i=10 and we want to access element at i=11 then it will be faster than if we wanted to access i=30(say). Now considering a two-dimensional matrix which can be think of as collection of arrays. If order of matrix is said m*n (row*column), then the memory will be contiguous with respect to rows. For i=0 and j=0 to i=0 and j=n is a contiguous memory and after this the next element in memory is at i=1 and j=0. So, we move in the fashion above than it will be easier to access the elements because they would be more probably stored in the cache memory because we are moving in the direction on contiguous memory. But if try to move like, from i=0 and j=0 to i=m and j=0 then we are accessing the memory which is 'm' locations ahead from previous element which isn't contiguous so, probably it won't be stored in cache memory.

So, when we will be changing the position of the for loops, the result won't change but they way we are accessing the memory will change. This will affect the performance because if we are accessing values of elements from cache then it's pretty fast but if we are accessing them from far aways levels of cache of from main memory than it will take more time then what it would have taken when it would have been present at L1 cache.

The code in general looks like,

```
int resultant_matrix[n][n];
    long start = currentTime();
    for(int i = 0; i<n; i++){
        for(int k = 0; k<n; k++){
            for(int j = 0; j<n; j++){
                 if(k==0){
                    resultant_matrix[i][j] = 0;
                 }
                resultant_matrix[i][j] += m1[i][k]*m2[k][j];
            }
        }
    }
    long end = currentTime();
```

The performance depends upon the order of i,j,k in line (m1[i][k]*m2[k][j];) because this is the line where we are accessing the memory.

Note: if in (m1[i][k]*m2[k][j];) while iterating through it in third loop, if for m1[i][k] if either k is varying or the i and k are fixed then the memory access will be fast because it would be present in cache memory. Similarly, for m2[k][j] if either j is varying and k is fixed or both k and j is fixed then it would be fast because of the same reason.

Observations:

From the data and graph, it can be seen that for combination 2 it is working fastest in general because the code snippet present above is for combination 2 only. We can see that during the 3rd loop bith i and k are fixed only j is changing. So, m[i][k] is fixed and in m2[k][j] we are moving in contiguous direction of 2d matrix, so that's the reason why it's the fastest. Sometimes 4th combination is the fastest because it is nearly same as combination 2.

Q4.

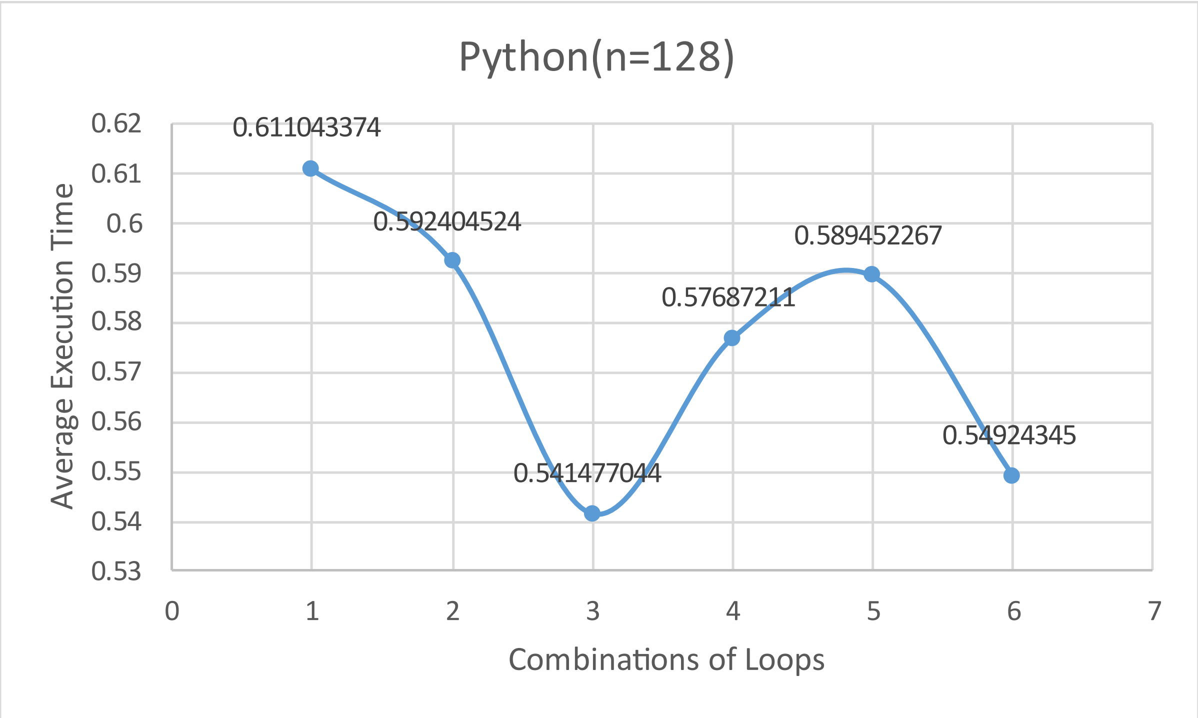
Bucket 2 => Python

a) And b)

For N = 128,

Python(n=128)

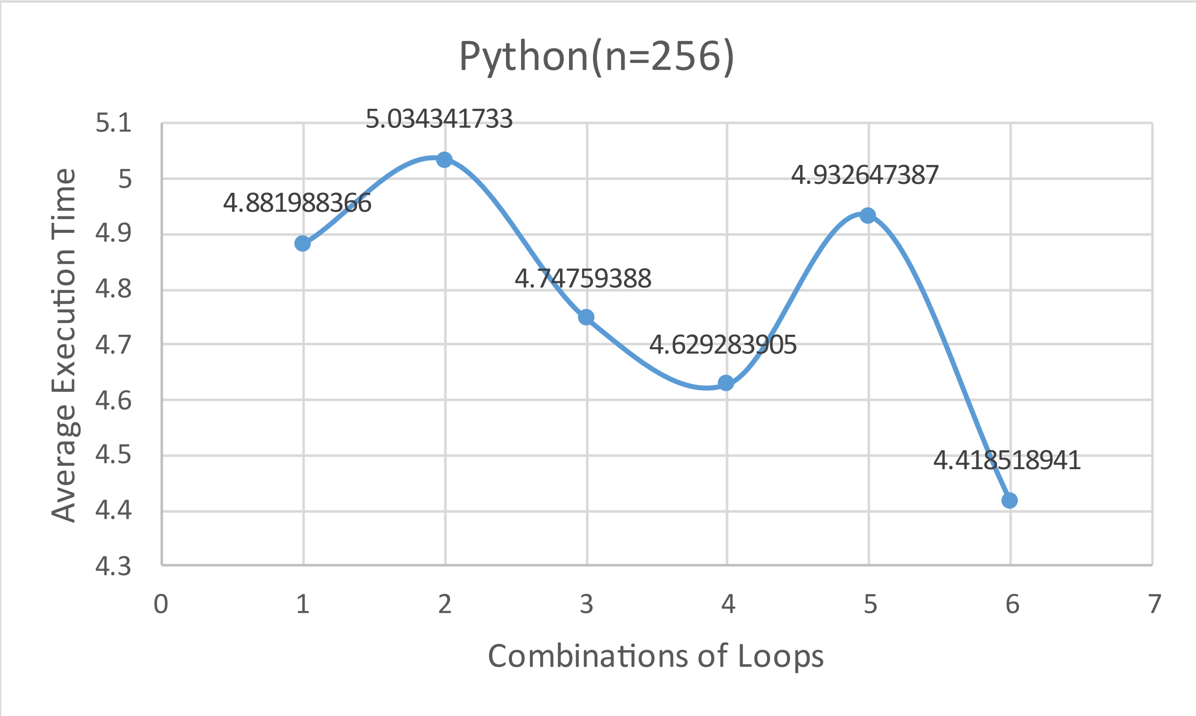
Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	0.694411755	0.601146936	0.53757143	0.611043374
2	0.565457106	0.59445405	0.617302418	0.592404524
3	0.567180872	0.521526098	0.535724163	0.541477044
4	0.676609039	0.507604361	0.546402931	0.57687211
5	0.551619053	0.568687916	0.648049831	0.589452267
6	0.523020983	0.638662577	0.486046791	0.54924345



For N = 256,

Python(n=256)

Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	4.254848003	5.122803926	5.268313169	4.881988366
2	4.971401453	4.897908926	5.233714819	5.034341733
3	5.057568073	4.722100019	4.463113546	4.74759388
4	4.959511757	4.094357491	4.833982468	4.629283905
5	5.007948637	5.191268206	4.598725319	4.932647387
6	4.428341627	4.501612902	4.325602293	4.418518941

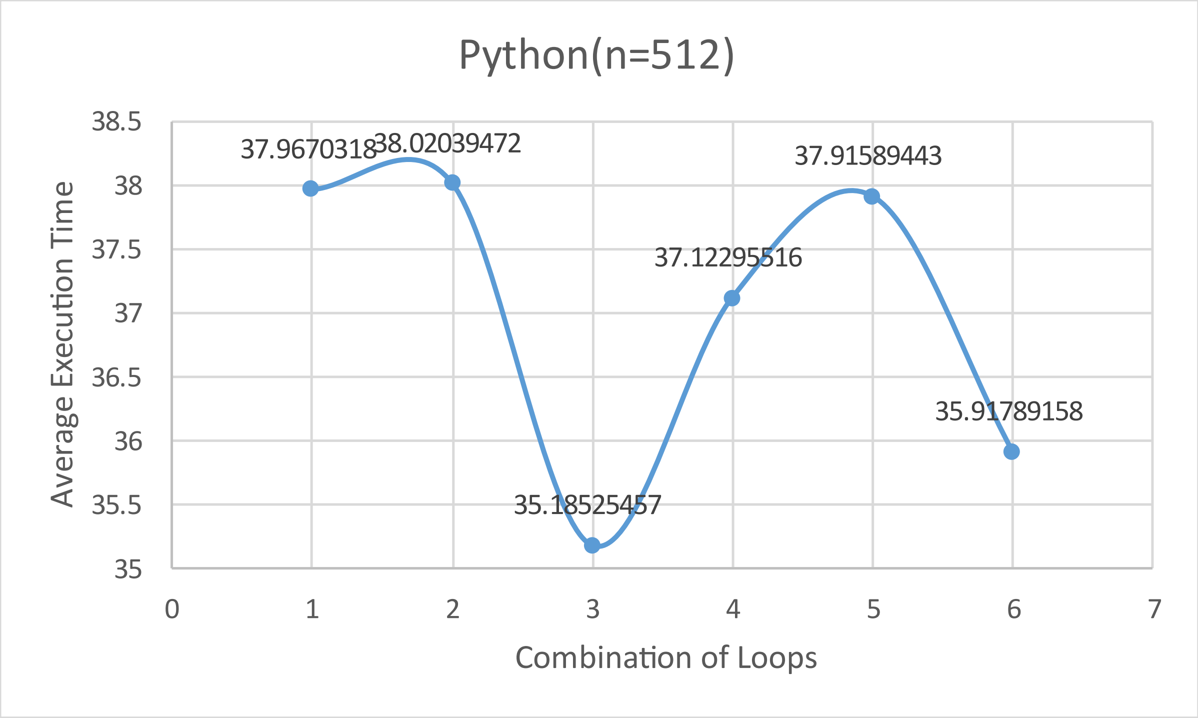


For N = 256,

Python(n=512)

Combo	1st Execution Time	2nd Execution Time	3rd Execution Time	Average Execution Time
1	37.16296554	38.34308434	38.39504552	37.9670318

2	39.11322331	35.72000551	39.22795534	38.02039472
3	34.38611531	34.50494885	36.66469955	35.18525457
4	41.3128531	33.31592035	36.74009204	37.12295516
5	39.38110781	36.37273097	37.99384451	37.91589443
6	36.4402864	35.71134019	35.60204816	35.91789158



Observations:

As we can see that the average execution time is very close for each iteration, so it's difficult to conclude about which combination is better because theoretically combination 5 should be the fastest which is same as no. 2 of C++ bucket. But as the average time is nearly same for each value of N the correct observation might not be visible or we might need to increase our number of runs and should remove other tasks from task manager and anti-virus so memory will've less load. Also, we know that python takes much more time than C++ takes for any N, because python use one of the most efficient way of memory access so, this might be the reason that for every combinations the average time of execution is same.