6CS005 Learning Journal - Semester 1 2020/21

Subin Shrestha, 2039281

Table of Contents

1 Parallel and Distributed Systems

1.1 Answer of First Question

Threads are the smallest unit of computational process executed by the CPU.

Computing a single task at a time takes a long time to compute. Threads are designed to compute independent processes. This enables CPU to execute multiple independent threads simultaneously saving time. Thus, CPU were designed so that CPUs could perform computation parallelly.

1.2 Answer of Second Question

The two types of Process scheduling policies are Pre-emptive and Co-operative.

Pre-emptive: In pre-emptive process scheduling the scheduler decides how long each process runs for and allocates time for each process. If a process exceeds its allocated time, it is stopped by the scheduler.

Co-operative: In Co-operative process scheduling the process decides how long it runs for. Only when process has completed its execution, it stops and allows other process to execute.

Pre-emptive is preferred as if there is a flaw in design of a process and it does not stop, the process will take valuable computational/CPU resources. In pre-emptive scheduling all the process will have access to CPU resources and will result in high CPU utilization.

Java does not have any built-in thread scheduler and java does not force the machine to schedule the threads in any specific manner. This implies the scheduling of java threads depends on the platform it is running on. So, if the execution of file depends on how the threads are scheduled java is not preferred as it would not produce consistent output across multiple platforms. So, yes the choice of thread scheduler in the platform used to run java has influence on the behaviour of Java Threads.

1.3 Answer of Third Question

Centralized System: Centralized System are systems where all the programs are executed at a same computer. All the users always share the same resources and has single point of control and failure and security is easier to maintain.

Distributed System: Distributed systems are systems where all the programs are executed on multiple independent autonomous interconnected computers. Resources are spread across multiple computers and has multiple point of failure. Distributed System has high fault tolerance as even if some part of the system fails the whole system continues to function. Resources in distributed systems are effectively utilized and has significantly more computational resources than centralized systems. Maintaining security and difficulty in developing software for distributed systems are a major problem in distributed systems.

1.4 Answer of Fourth Question

Transparency in Distributed systems means to hide from the users that their processes are running on multiple independent computers. Users perceive that computation is being done on a single centralized system rather than on multiple interconnected systems. This is important in Distributed Systems as it is easier and intuitive for users to use single processor systems.

Types of Transparency in Distributed Systems:

Access Transparency: Local and Remote resources are accessed using identical operations.

Local Transparency: Resources are accessed without knowing their physical location.

Migration Transparency: Location of resources can be moved without any effect in the process.

Replication Transparency: Users do not know how many copies/backups of data exit.

Concurrency Transparency: Multiple users can use same resource concurrently without interference between them.

Failure Transparency: Users do not know if the failure of resource occurs.

Performance Transparency: Resources are allocated and utilized to improve performance.

Scaling Transparency: System can scale be scaled up or down without changing system structure or application programs.

1.5 Answer of Fifth Question

B = A + C ------ 1

B = C + D ------ 2

C = B + D ------ 3

Identification of dependency

3 has flow dependency with 2 and 1 as value of B must be calculate before 3 can be executed.

1, 2 and 3 have anti dependency as 1 and 2 requires value of C is calculated which is at 3rd step. So, 1 and 2 must wait for 3 to be computed to calculate the value of C and use it in 1 and 2.

1 and 2 have output dependency as the value of B changes after each step. So, changing the order of 1 and 2 will affect the final output value of B. This will also affect the value of C as the value of C depends on value of B.

Only reordering the statements does not produce same values for B and C as in original statements as the value of B truly depends on C and the value of C truly depends on B.

C = B_old + D --- 1

B_new = A + C --- 2

B = C + D --- 3

This solves arrangement of statements removes anti dependencies and output dependency.

This Also enables statement 2 and 3 to be executed in parallel.

Anti-dependency is solved as no value is updated after it is required. Output dependency is solved as no value is updated once it is computed.

Flow Dependency cannot be solved as flow dependencies cannot be removed without Changing the purpose of the program.

```
#include <stdio.h>
int main(int argc, char const * argv[]) {
  int A = 5;
  int B_old = 4;
  int D = 4;
  int C = B_old + D;
  int B_new = A + C;
  int B = C + D;

  printf("Final Values are: \n");
  printf("A = %d \n", A);
  printf("B_old = %d \n", B_old);
  printf("B_new = %d \n", B_new);
  printf("B = %d \n", B);
  printf("C = %d \n", C);
  printf("D = %d \n", D);
  return 0;
}
```

Code 1 Task 1 Question 5 code

- Code is included in the file: 2039281_Task1_5.c

1.6 Answer of Sixth Question

Code on left (first code) gives random values as output each time it is run.

Outputs on running the code 5 times are:

188444, 208898 ,193965, 217172, 242647

Counter is a global variable and thread_func updates it. The thread uses the current value of counter in the thread when the thread is created. Since the threads are created while other threads are running the value of counter is changing. So, when the threads are joined the value of counter is different.

Code on the right (second code) gives a constant output of 500000 each time it is run.

Counter is a global variable and thread_func updates it. The thread uses the current value of counter in the thread when the thread is created. The threads are created only when the previous thread is joined, meaning new thread is created the previous thread is already completed. Since the thread_func adds 100000 to counter, after running for 5 times the value is 500000

- The codes are included as task1_6A.c and task1_6B.c inside Supporting Files folder.

2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system

2.1 Single Thread Matrix Multiplication

- The analysis of the algorithm's complexity. (1 mark)

  Algorithm as 3 loops each running N, M and P times. Considering each calculation takes unit time:

  The complexity of Algorithm is O(N*M*P).

  If the matrices are square matrices with size n. Then,

  The complexity of Algorithm is $O(n^3)$

- Suggest at least three different ways to speed up the matrix multiplication algorithm given here. (Pay special attention to the utilisation of cache memory to achieve the intended speed up). (1 marks)
  - The Three ways to improve are:
    - Using cache memory to prevent the access of C[i][j] value at every step. This saves access time.
    - Regarding n x n matrix as 2d array of a x a where each block is a sub matrix of n of size (n/a) x (n/a). Then a^3 matrix multiplication is done for each involving (n/a) x (n/a)
    - Using multithreading to compute multiple results at once.

- Write your improved algorithms as pseudo-codes using any editor. Also, provide reasoning as to why you think the suggested algorithm is an improvement over the given algorithm. (1 marks)

```
Int A[N][P], B[P][M], C[N][M];

int cache;

for (int i = 0; i < N; i++) {

  for (int j = 0; j < M; j++) {

    cache = 0;

    for (int k = 0; k < P; k++) {

      cache = cache + A[i][k] * B[k][j];

    }

    c[i][j] = cache;

  }

}
```

- Write a C program that implements matrix multiplication using both the loop as given above and the improved versions that you have written. (1marks)
    - Full Code is included in the file: 2039281_Task2_A.c

- Measure the timing performance of these implemented algorithms. Record your observations. (Remember to use large values of N, M and P – the matrix dimensions when doing this task). (1 marks)

  Observations when values of N, M and P, are 800, 750 and 850 respectively:

  original_time = [3.54136, 3.27827, 3.72438, 3.34107, 3.61651, 3.66029, 3.73083, 4.07393, 3.64318, 4.92446]

  improved_time = [2.63679, 2.31044, 2.36693, 2.67324, 2.51085, 2.50984, 2.66308, 2.69766, 2.71722, 2.54553]

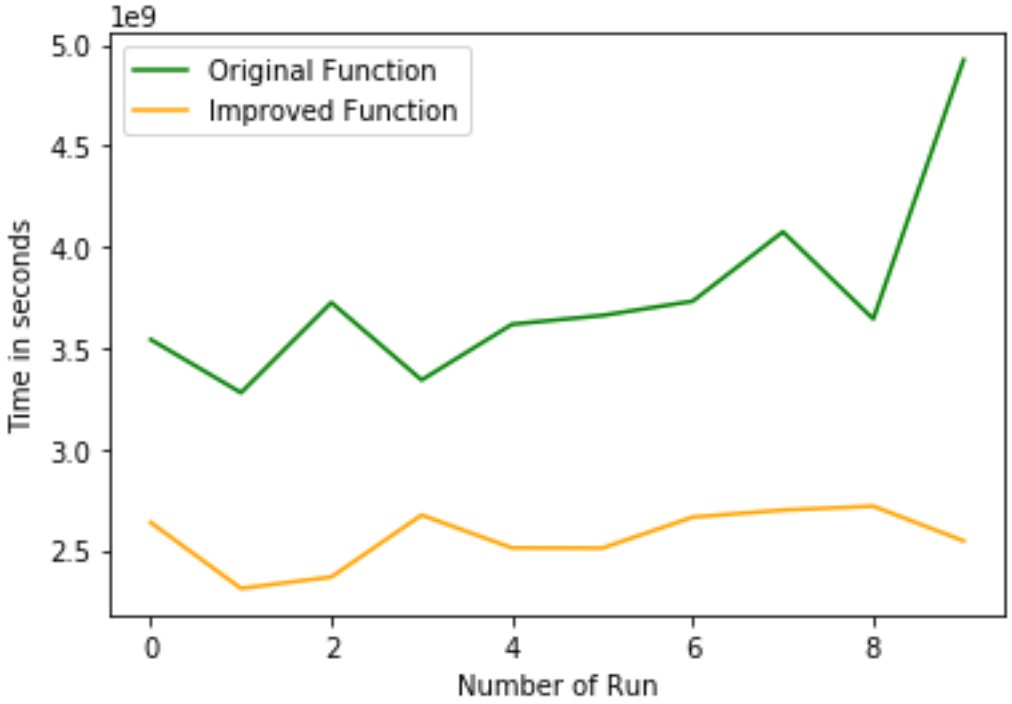

Figure 1: Comparison of Improved and Original Algorithm

The comparison shows that improved version is better than the original version. The original version takes an average of 3.5 seconds to complete where as improved version takes an average of 2.5 seconds.

The major computation line is c[i][j] = c[i][j] + a[i][k] * b[k][j] in the original code. Assuming it takes 2 nanoseconds to compute this line. The total time taken to compete the entire program is 2 x N x M x P nanoseconds. Using cache reduces the access time of c[i][j] at step. Assuming it reduces access time by .5 nanosecond the time taken to complete the entire program in improved code will be: 1.5 x N x M x P.

Using the values of N, M and P we get:

For Original Code:

Time = 2 x 800 x 750 x 850

Time = 1020000000 ns

Time = 1.02 Seconds.

For Improved Code

Time = 1.5 x 800 x 750 x 850

Time = 765000000 ns

Time = 0.765 Seconds.

The results show that the time shown by my hypothesis is less than the actual results. It may be because it takes longer than 2 ns to compute the line and the time may change with change in numbers.

2.2 Multithreaded Matrix Multiplication

- Include your code using a text file in the submitted zipped file under name Task2.2
    - Code included in file: 2039281_Task2_B.c
- Insert a table that has columns containing running times for the original program and your multithread version. Mean running times should be included at the bottom of the columns.

Original Program **(Output Matrix Size: 800 x 750)**	**Multithread Version** **(Output Matrix Size: 800 x 800)**	**Multithread Version** **(Output Matrix Size: 1500 x 1500)**
3.54136	1.32874	14.69905
3.27827	1.28103	14.74464
3.72438	1.25867	16.05465
3.34107	1.24007	13.68569
3.61651	1.24630	13.49028
3.66029	1.24933	13.51400
3.73083	1.27338	16.39930
4.07393	1.28390	16.23843
3.64318	1.25072	13.92603
4.92446	1.25832	15.46304
Average: 3.753428 s	Average: 1.26705 s	Average: 14.82151 s

- Insert an explanation of the results presented in the above table.

  The above table shows that for a similarly sized matrices, using multithreading shows significant improvement over normal methods.

- Finding Optimal Number of Threads

  Threads are implemented in such away that each thread is used to compute one row of the output matrix. Example. Thread 1 computes all the values for row 1, Thread 2 computes all the values for row 2 and so on. If the number of threads is less than the required, the program ends in error. The matrix used is of size 1500 x 1500. So minimum number of threads required is 1500. The program multiplies the matrix 10 times and gives the average time taken. The number of threads is increased by 20, 6 times and the results are shown in the graph below.

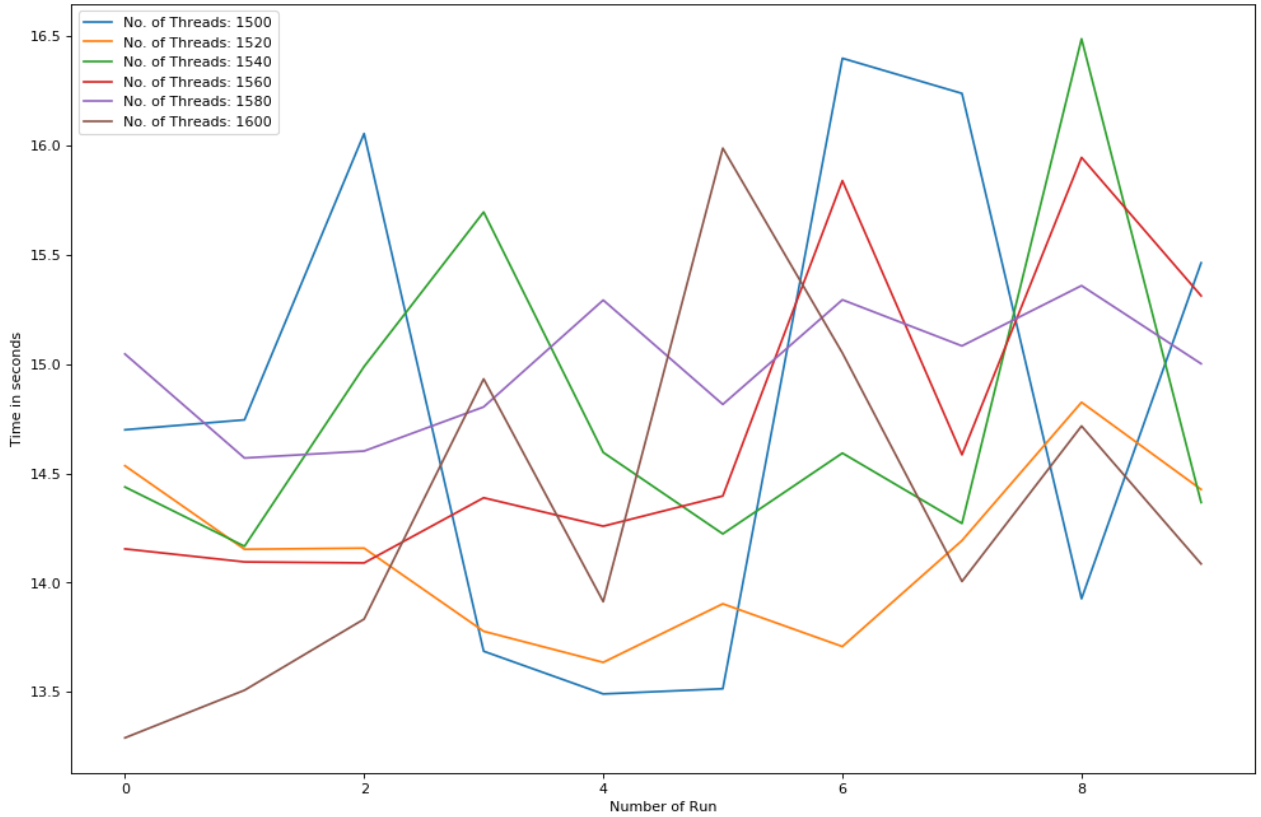

Figure 2: Time taken for multiple number of threads

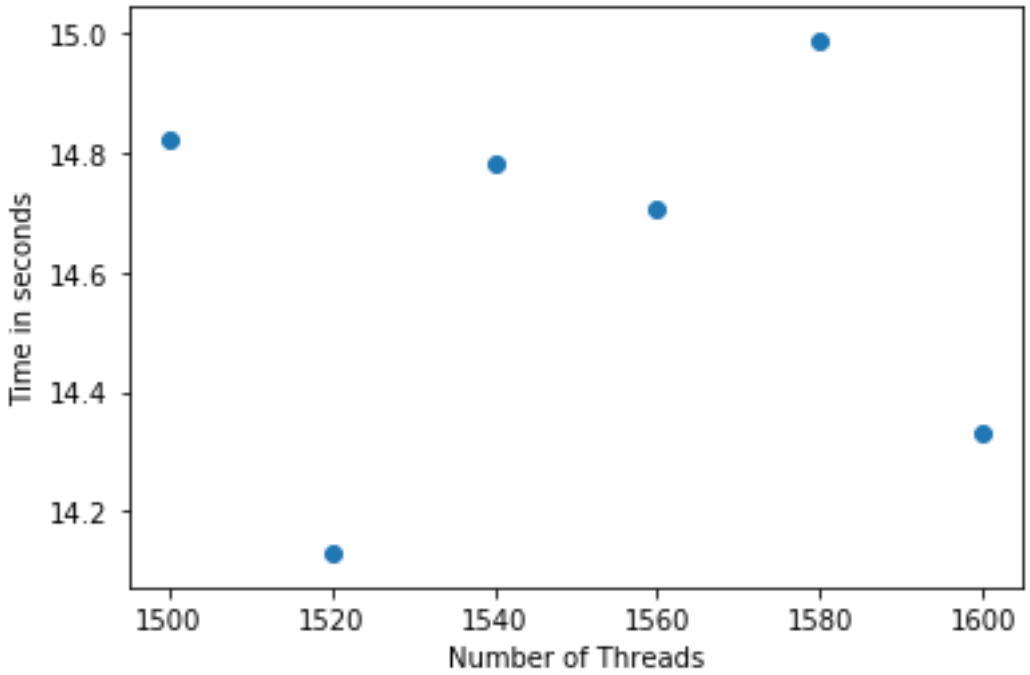

Figure 3: Average Time taken for different number of threads.

Figure 2 and 3 shows that there are some fluctuations for execution time but nothing significant is seen. The average time taken does not deviate more than 1 second. This is because the time taken depends on the utilization of threads by the program. One thread produces all the results for 1 row. For the matrix of size 1500 it requires at least 1500 threads. If the number of threads is less than that the code gives segmentation fault. Increases the number of threads above 1500 does not decrease execution time as the threads would not be utilized during computation. Some fluctuations are seen due to uncertainty and the numbers involved in the computation.

So, the execution time depends on the implementation and utilization of threads and not on the number of threads.

2.3 Password cracking using POSIX Threads

- Include your code using a text file in the submitted zipped file under name Task2.3.1, Task2.3.3, Task2.3.5
    - Code For decrypting 2 letters and 2 numbers is named as CrackAZ99.c inside Supporting Files Folder. This is the original file provided to us which has been modified to run the code 10 times and record running time.
    - Code included in file: 2039281_Task2_C_3.c
        - This is for decrypting 3 Letters and 2 numbers
    - Code included in file: 2039281_Task2_C_5.c
        - This is for decrypting 2 letters and 2 numbers using 2 threads.
- Insert a table of 10 running times and the mean running time.

Cracking 2 Letters 2 Numbers (Single Thread) **Task 2.C.1 (Password used: CD20)**	**Cracking 3 Letters 2 Numbers (Single Thread)** **Task 2.C.3 (Password used: BCD20)**	**Cracking 2Letters 2Numbers (Multithreading)** **Task 2.C.5 (Password used: CD20)**
15.24570	136.59915	16.47418
14.95672	136.51136	16.33655
14.66667	136.50965	16.52581
14.72703	136.56938	16.47705
14.89622	136.48027	16.32894
14.61538	136.45425	16.29129
14.60511	136.47424	16.45596
14.56440	136.46502	16.52852

14.56669	136.46192	16.17783
14.57694	136.46606	16.27570
Average: 14.742086 s	Average: 136.49913 s	Average: 16.387183 s

- Insert a paragraph that hypothesises how long it would take to run if the number of initials were to be increased to 3. Include your calculations.

Number of possible combinations for 3 Uppercase Letters and 2 numbers are = 26 x 26 x 26 x 100

= 1,757,600

1,757,600 are the total number of possible passwords with 3 letters and 2 numbers. Depending on how long it will take to compute each combination that involves encrypting the current password and comparing the encrypted password with the given test.

If it takes 0.0001 second to compute one combination, the maximum time to encrypt any text must be 175.76 or 176 seconds.

- Explain your results of running your 3 initial password crackers with relation to your earlier hypothesis.

The results of cracking password with 3 letters and 2 numbers took 137 seconds for BCD20. This is very close to my predicted time of 176 seconds for a combination that is close to the beginning as the first letter is B. I think that my hypothesis my idea maximum time to decrypt all the password is correct but my assumption that it takes 0.0001 second to compute one combination was wrong. The result showed that it will takes around 0.01 - 0.001 to compute each combination.

- Write a paragraph that compares the original results with those of your multithread password cracker.

  The results of original cracker and multithreaded cracker shows that multithreaded cracker takes longer than original cracker.

  The original cracker only took 14.74 seconds on average whereas multithreaded cracker took 16.38 seconds on average for password CD20.

  Even though multithreaded cracker took slightly longer than original cracker to compute the same password the multithreaded cracker has explored doubled the number of combinations than the original cracker.

  The multithreaded cracker computes 2 combinations at simultaneously. One starting from AA00 and other from NA00 but the original cracker only computes one combination at a time starting from AA00.

  Say the password we are trying to crack is the last combination in the loop i.e. ZZ99. Since the multithreaded cracker will only take half the time taken by original cracker.

  So, on average for random set of passwords the multithreaded cracker will be twice as fast as original cracker.

3 Applications of Password Cracking and Image Blurring using HPC-based CUDA System

3.1 Password Cracking using CUDA

- Include your code using a text file in the submitted zipped file under name Task3.1
    - Code included in the file: 2039281_Task3_A.cu

- Insert a table that shows running times for the original and CUDA versions.

Cracking 2 Letters 2 Numbers (Single Thread) **Task 2.C.1 (Password used: CD20)**	**Cracking 2Letters 2Numbers (Multithreading)** **Task 2.C.5 (Password used: CD20)**	**Cracking 2Letters 2Numbers (CUDA)** **Task 2.C.5 (Password used: cd20)**
15.24570	16.47418	0.21034
14.95672	16.33655	0.15946
14.66667	16.52581	0.16189
14.72703	16.47705	0.15951
14.89622	16.32894	0.16104
14.61538	16.29129	0.16045
14.60511	16.45596	0.16409
14.56440	16.52852	0.15833
14.56669	16.17783	0.15976
14.57694	16.27570	0.16426

Average: 14.742086 s Standard Deviation: 0.225	Average: 16.387183 s Standard Deviation: 0.121	Average: 0.165913s Standard Deviation: 0.0157s

- Write a short analysis of the results

To crack password using CUDA, a custom encryption function was made to run on device. The normal encryption function used to encrypt in SHA512 cannot be run on device. The custom encryption function encrypts the 2 letters and 2 numbers into 6 letters and 4 numbers. The letters used are lower case.

The above table shows very significant improvement over original single threaded cracking and multithreaded cracking. The CUDA program creates a grid of blocks of size 26 x 26 and each block in the grid has 10 x 10 threads. This allows the program to run each possible combination on its individual thread significantly decreasing the execution time. The total execution time is the average of time taken to complete individual thread. Each individual thread will encrypt its corresponding combination, compares it with the given encrypted text and prints the result if the encrypted texts match.

Analysis of running time of all three programs showed that CUDA produced consistent execution time compared to single threaded and multithreaded cracking. The execution time of CUDA program only had the standard deviation of 0.0175s whereas single threaded and multithreaded had the standard deviation of 0.225 and 0.121 respectively.

If a proper SHA512 encryption function is implemented on GPU, execution time may increase as it takes longer to encrypt each possible combination and it also takes longer time to compare the encrypted text. This will increase the execution time, but the total time taken to compute will still be significantly less than the single threaded and multithreaded program.

The difference in time taken can clearly be seen in the graph below:

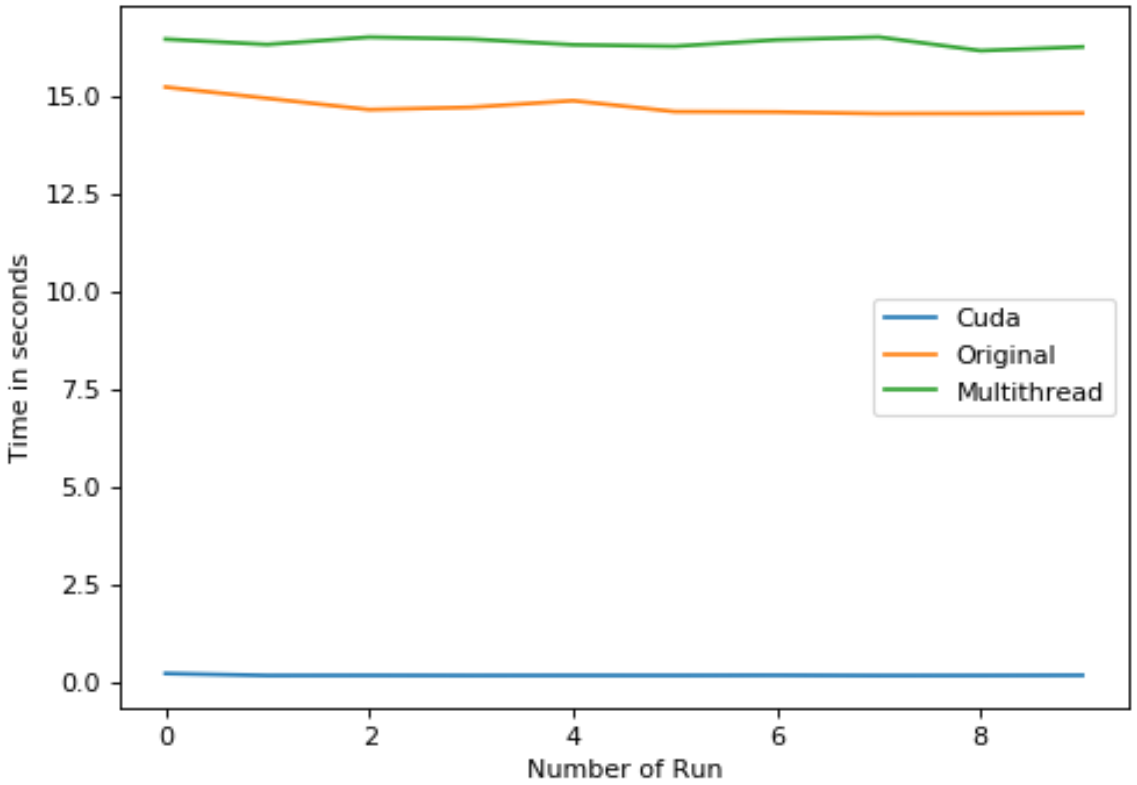

Figure 4: Comparison of CUDA Processing and Normal Processing

3.2 Image blur using multi dimension Gaussian matrices

- Include your code using a text file in the submitted zipped file under name Task3.2
    - Code Included in the file: 2039281_Task3_B.cu

Both original and CUDA used following image as input:

- Comparison of Output of Both Original and CUDA versions
    - The Original C Code is included in Supporting Files as: GaussianBlur.c

Figure 5: Input Image

Original C program produced the following output:

Figure 7:Output Image (Original)

CUDA Program produced the following output:

Figure 6: Output Image (CUDA)

- Insert a table that shows running times for the original and CUDA versions.

Original C Gaussian Blur Program	CUDA Gaussian Blur Program
0.07725s	0.16803s
0.05197s	0.02686s
0.05206s	0.02661s
0.04556s	0.02651s
0.04706s	0.02642s
0.04621s	0.02692s
0.04707s	0.02621s
0.04822s	0.02626s
0.04695s	0.02669s
0.04681s	0.02750s
Average: 0.050916s	Average: 0.040801s

- Write a short analysis of the results

On running both original and CUDA program, both produced similar and consistent output. It showed that CUDA version was slightly faster than original version. On Image of larger sizes, the difference may be apparent.

The CUDA program uses blocks of size(width-1) x (height-1) to compute each pixel. Similarly, original c program uses two nested loops where loop counters are of size (width-2) and (height-2).

Note:

- All files are stored in GitHub. Available at:

  https://github.com/sthasubin429/cuda-cw