6CS005 Learning Journal - Semester 1 2020/21

Samir Singh Chhetri

UNID-2041374

Table of Contents

1 Parallel and Distributed Systems

1.1 Answer of question number 1

Thread is a lightweight and independent instruction sequence that the os can schedule to run as such, but independently of its main program.

They are built to solve the things that follow.

- For the enhancement of program efficiency.
- With less machine resources, it can be run.
- To build it takes very few overheads.

1.2 Answer of question number 2

Two policies for process scheduling are:—

i. Preemptive: It is the duty of the scheduler to decide the time taken by each thread or procedure in this policy. If a process reaches its allotted time, it is stopped by the scheduler.

ii. Cooperative: Each mechanism is accountable for deciding the time taken by them in this policy. Only if it feels like cooperating can a method reno unce execution.

Pre-emptive policy is superior between two policies because, unlike cooperative policy, the scheduler decides the process time for each thread because each process will get equal processing time and the thread's total execution time will be set and concurrent.

Yes, the strategy of process scheduling influences the behavior of threads. Threads would get the same amount of processing time in Preemptive p oicy regardless of which time is set for the completion of the program. But one thread will get more time in Cooperative Policy, while others will get l ess time because the time of program completion is not set.

1.3 Answer of Question number 3

S. N.	Centralized system	Distributed system
1.	This is a client/server architecture that links multiple clients to a single server.	Many independent computers are connected and generated as a single coherent device in this scheme.
2.	If a server fails, the entire device or network is disabled.	When a server crashes, it will be replaced by another server to solve a device failure.
3.	Cheaper and faster, but more risky to implement.	Expensive and hard to execute, but safer.
4.	The same form of technology is used to construct them, such as the network, operating system, programming language, etc. (Homogeneity).	Different types of technology may be used to build them (Heterogeneity).

1.4 Answer of question number 4

Although they are composed of multiple systems with different networks, operating systems, programming languages, etc., distributed systems look like a single machine. Such a concealment of a user's distinct components to display them as if the system is single or centralized is called distributed system transparency.

1.5 Answer of question number 5

Let s1: B=A+C, s2:B=C+D, s3:C=B+D

Flow dependency (previous step's output is next step's input): s3 is flow dependent on s2

B=C+D

C=B+D

Anti-dependency (previous step's input is next step's output): s3 is anti-dependent on s2.

B=C+D

C=B+D

Output dependency (previous step's output is next step's output): s2 is output dependent on s1.

B=A+C

B=C+D

Flow dependency is impossible to solve in this function while anti-dependency and output dependency can be solved by introducing two temporary variable.

```
bTemp=B
cTemp=C
bTemp=A+C
B=cTemp+D
C=B+D
```

Code:

```c
#include <stdio.h>
//compile with "gcc Dependency.c -o dependency"
int main() {
//Assigning values to variables
   int A=10;
   int B=20;
   int C=30;
   int D=40;

//assigning temporary variable to B and C
   int cTemp=C;
   int bTemp=B;

//Executing function
    bTemp=A+C;
    B=cTemp+D;
    C=B+D;

//Printing output
    printf("Result after removing dependencies:\n");
    printf("A=%d\n",A);
    printf("B=%d\n",B);
    printf("C= %d\n", C);
    printf("D= %d\n", D);

    return 0;
}
```

From above permutation, output dependency and anti-dependency has been solved. This permutation will also produce same output as original one.

1.6 Answer of question number 6

Output of first program: 181729

Output of second program: 500000

The output is different since it runs sequentially rather than parallel to the second program. This is because within the same loop, two pthread-create and pthread-join operations are carried out. It can be inferred from this that the second program does not obey multithreading, but multi-threading, unlike the second program, follows the first program. So, without multithreading, the value created by the first multithreaded program is lo wer than the second program.

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)
  Complexity of this 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)
  Ways to speed up the matrix multiplication algorithm are as follows:
  i. Using GPU or CUDA.
  ii. Using multiple posix threads.
  iii. Using integer k in middle loop and j in last and removing C[i][j]=0.

- 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];

for (int i = 0; i < N; i++)
 {
        for (int k = 0; k < M; k++)
         {
                for (int j = 0; j < P; j++)
                        {
                                C[i][j] = C[i][j] + A[i][k] * B[k][j];
                        }
                }
}
```

This algorithm is improvement over original algorithm because both C and B are continuously accessed. This is because cache miss will occur on B in every iteration as k is swapped to middle loop.

- Write a C program that implements matrix multiplication using both the loop as given above and the improved versions that you have written. (1marks)

  Include your code using a text file in the submitted zipped file under name Task2.1

- 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)
  Insert a paragraph that hypothesises how long it would take to run the original and improved algorithms. Include your calculations. Explain your results of running time.

For original algorithm, it may take 20 seconds to run for 1024*1024 sized matrix while it may take 15 seconds for improved one. This is because although the time complexity for both algorithm is $O(n^3)$ however the previous algorithm has to perform certain operation after second loop and cache miss will occur in B after every iteration in improved one.

2.2 Multithreaded Matrix Multiplication

```c
/*********************************************************************
    This program demonstrates matrix multiplication with threads.
    Compile this program with cc -o Matrix  Matrix.c  -pthread

*********************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>


#define MAT_SIZE 1024
int MAX_THREADS;


int N,M,P;          //Parameters For Rows And Columns
int matrix1[MAT_SIZE][MAT_SIZE]; //First Matrix
int matrix2[MAT_SIZE][MAT_SIZE]; //Second Matrix
int result [MAT_SIZE][MAT_SIZE]; //Multiplied Matrix
int step_i = 0;

//Type Defining For Passing Function Argumnents
//typedef struct parameters {
```

```
//    int x,y;
//}args;

//Function For Calculate Each Element in Result Matrix Used By Threads - - -//
void* mult(void* arg){


    int core = step_i++;
    //Calculating Each Element in Result Matrix Using Passed Arguments
    for (int i = core * N / MAX_THREADS; i < (core + 1) * N / MAX_THREADS; i++)
        for (int k = 0; k< P; k++)
            for (int j = 0; j< M; j++)
                result[i][j] += matrix1[i][k] * matrix2[k][j];
    //sleep(3);

    //End Of Thread
    pthread_exit(0);
}
int time_difference(struct timespec *start, struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;

  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}
```

```c
int main(){
    struct timespec start, finish;
    long long int time_elapsed;
    // Getting Row And Column(Same As Row In Matrix2) Number For Matrix1
    printf("Enter number of rows for matrix 1: ");
    scanf("%d",&N);
    printf("Enter number of columns for matrix 1: ");
    scanf("%d",&M);
    printf("Enter number of columns for matrix 2: ");
    scanf("%d",&P);
    printf("Enter the number for threads you want to create  \n");
    scanf("%d",&MAX_THREADS);


    for(int x=0;x<N;x++){
        for(int y=0;y<M;y++){
            matrix1[x][y]=rand()%50;
        }
    }

    for(int x=0;x<M;x++){
        for(int y=0;y<P;y++){
            matrix2[x][y]=rand()%50;
        }
    }

    pthread_t *t = malloc(sizeof(pthread_t) * MAX_THREADS);

    //Defining Threads
    pthread_t thread[MAX_THREADS];

    //Counter For Thread Index
    int thread_number = 0;

    //Defining p For Passing Parameters To Function As Struct
```

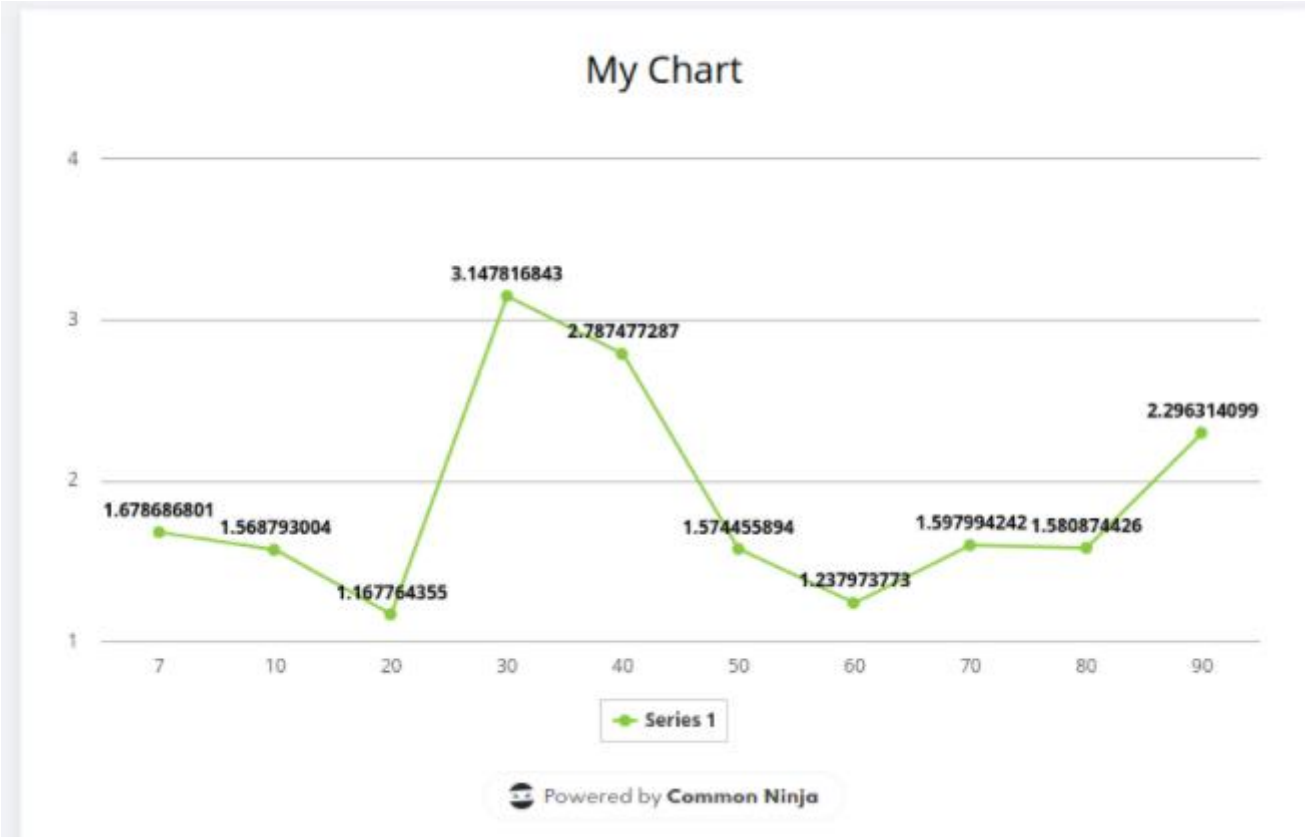
```
 //Start Timer
 clock_gettime(CLOCK_MONOTONIC, &start);


 for (int x = 0; x < MAX_THREADS; x++)
 {
     int *p;
         //Status For Checking Errors
         int status;

         //Create Specific Thread For Each Element In Result Matrix
         status = pthread_create(&thread[thread_number], NULL, mult, (void *) &p[thread_number]);

         //Check For Error
         if(status!=0){
             printf("Error In Threads");
             exit(0);
         }

         thread_number++;
     }




 //Wait For All Threads Done - - - - - - - - - - - - - - - - - - - - - - - //

for (int z = 0; z < MAX_THREADS; z++)
     pthread_join(thread[z], NULL);


 //Print Multiplied Matrix (Result) - - - - - - - - - - - - - - - - - - - -//
```

```
    printf(" --- Multiplied Matrix ---\n\n");
    for(int x=0;x<N;x++){
        for(int y=0;y<P;y++){
            printf("%5d",result[x][y]);
        }
        printf("\n\n");
    }


    //Calculate Total Time Including 3 Soconds Sleep In Each Thread - - - -//

        clock_gettime(CLOCK_MONOTONIC, &finish);
        time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,(time_elapsed/1.0e9));


    //Total Threads Used In Process - - - - - - - - - - - - - - - - - - - -//

    printf(" ---> Used Threads : %d \n\n",thread_number);
    for(int z=0;z<thread_number;z++)
        printf(" - Thread %d ID : %d\n",z+1,(int)thread[z]);
    free(t);
    return 0;
}
```

- **Plot the time it takes to complete the matrix multiplication against the number of threads**

The graph representing the time taken by the matrix multiplication to complete against the number of threads used is shown below:

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



Since time elapsed is less when 60 thread were used which is best.

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

- Original “two alphabet two digit (GR58)” password cracking code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <time.h>



void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}

int count=0;

void crack(char *salt_and_encrypted){
  int x, y, z;     // Loop counters
  char salt[7];    // String used in hashing the password. Need space for \0 // incase you have modified the salt value, the
n should modifiy the number accordingly
  char plain[7];   // The combination of letters currently being checked // Please modifiy the number when you enlarge the e
ncrypted password.
  char *enc;       // Pointer to the encrypted password

  substr(salt, salt_and_encrypted, 0, 6);

  for(x='A'; x<='Z'; x++){
    for(y='A'; y<='Z'; y++){
      for(z=0; z<=99; z++){
        sprintf(plain, "%c%c%02d", x, y, z);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
```

```
        printf("#%-8d%s %s\n", count, plain, enc);
      //return; //uncomment this line if you want to speed-
  up the running time, program will find you the cracked password only without exploring all possibilites
          }
        }
      }
    }
  }

  int time_difference(struct timespec *start, struct timespec *finish,
                      long long int *difference) {
    long long int ds =  finish->tv_sec - start->tv_sec;
    long long int dn =  finish->tv_nsec - start->tv_nsec;

    if(dn < 0 ) {
      ds--;
      dn += 1000000000;
    }
    *difference = ds * 1000000000 + dn;
    return !(*difference > 0);
  }

  int main() {
  int i;
    struct timespec start, finish;
    long long int time_elapsed;

    clock_gettime(CLOCK_MONOTONIC, &start);

         crack("$6$AS$WD5QAO45ZEbw8S.ds92cL0ipdqmHCf1I2VF4t4OTAHP/Qfo6MEvxgSNdPSlhwPw8ncg1HiWK5dFKldaGqSHTj1");

    printf("%d solutions explored\n", count);

    clock_gettime(CLOCK_MONOTONIC, &finish);
```

```
    time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
           (time_elapsed/1.0e9));

    return 0;
}
```

- Insert a table of 10 running times and the mean running time.

	Time (In second)
Time elapsed	126.233844879
Time elapsed	119.113784190
Time elapsed	188.307031907
Time elapsed	193.083221601
Time elapsed	193.778107099
Time elapsed	210.900023293
Time elapsed	197.794822583
Time elapsed	201.071428790
Time elapsed	204.591146124
Time elapsed	229.514197990
Mean Time	**186.438760846**

- 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.

  If the number of initials is to be raised to 3, an additional loop in crack function will be added to the loop, which will increase time complexity from O(n3) to O(n4), raising processing time by 100 times as there are 26 alphabets. As the mean time for cracking 4 digit passwords was 186.438760846 seconds, we will get 4827.4077 seconds if we multiply it by 26, which is 80.79 minutes. It would then take 80.79 minutes or 4827.4077 seconds for the password to crack if the initials were to be increased to 3

- Original “three alphabet two digit (GAR58)” password cracking code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <time.h>



void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}

int count=0;

void crack(char *salt_and_encrypted){
  int w,x, y, z;     // Loop counters
  char salt[7];    // String used in hashing the password. Need space for \0 // incase you have modified the salt value, then should modifiy the number accordingly
  char plain[7];   // The combination of letters currently being checked // Please modifiy the number when you enlarge the en crypted password.
  char *enc;       // Pointer to the encrypted password
```

```
    substr(salt, salt_and_encrypted, 0, 6);

    for(w='A'; w<='Z'; w++){
      for(x='A'; x<='Z'; x++){
      for(y='A'; y<='Z'; y++){
        for(z=0; z<=99; z++){
          sprintf(plain, "%c%c%c%02d",w, x, y, z);
          enc = (char *) crypt(plain, salt);
          count++;
          if(strcmp(salt_and_encrypted, enc) == 0){
        printf("#%-8d%s %s\n", count, plain, enc);
      //uncomment this line if you want to speed-
  up the running time, program will find you the cracked password only without exploring all possibilites
          }
        }
      }
      }
    }
  }

  int time_difference(struct timespec *start, struct timespec *finish,
                      long long int *difference) {
    long long int ds =  finish->tv_sec - start->tv_sec;
    long long int dn =  finish->tv_nsec - start->tv_nsec;

    if(dn < 0 ) {
      ds--;
      dn += 1000000000;
    }
    *difference = ds * 1000000000 + dn;
    return !(*difference > 0);
  }
```

```
int main() {
int i;
  struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);

       crack("$6$AS$1m8hTZ4U4lOpgB24Sd7yFKvgDVy.eoMeO2TWah6Bm/owOy80Nw5fZW33Jj91ZloviMSfol9PyOgfZSPOEbv7k0");

  printf("%d solutions explored\n", count);

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
         (time_elapsed/1.0e9));

  return 0;
}

```

- Insert a table of 10 running times and the mean running time.

S.N	Running time for 5 digit Password cracking (in seconds)	Running time for 5 digit Password cracking (in nano seconds)
1.	5536.627454821	5536627454821
2.	5537.527354920	5537527354920
3.	5511.674458601	5511674458601
4.	5526.627454821	5526627454821
5.	5548.627490786	5548627490786
6.	5539.124364821	5539124364821
7.	5536.135754821	5536135754821
8.	5536.123454821	5536123454821
9.	5556.627123210	5556627123210
10.	5536.627454821	5536627454821
	Mean=5536.572236644	Mean=5536572236644

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

  The real time of the initial password cracker running 3 was greater than the previous hypothesis. It may be because of the CPU, since many other functions other than password cracking must be performed. As the time elapsed increases, some software or device implementation may also take Processor time to raise the elapsed time of 3 digit password cracker in return.

- Original “two alphabet two digit (GA58)” password cracking code with posix_threads

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <time.h>
#include <pthread.h>



char *passwords_enc ="$6$AS$WD5QAO45ZEbw8S.ds92cL0ipdqmHCf1I2VF4t4OTAHP/Qfo6MEvxgSNdPSlhwPw8ncg1HiWK5dFKldaGqSHTj1";

void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}


void shreethread(){
int i;
pthread_t SS1, SS2;

void *kernel_function_1();
void *kernel_function_2();

pthread_create(&SS1, NULL,kernel_function_1,passwords_enc);
pthread_create(&SS2, NULL,kernel_function_2,passwords_enc);

pthread_join(SS1, NULL);
```

```
  pthread_join(SS2,NULL);
  }
  void *kernel_function_1(char *salt_and_encrypted){
    int S,H,R;
    char salt[7];
    char plain[7];
    char *enc;
    int count = 0;
    substr(salt, salt_and_encrypted, 0, 6);

    for(S='A';S<='M'; S++){
      for(H='A'; H<='Z'; H++){
        for(R=0; R<=99; R++){
          sprintf(plain, "%c%c%02d", S,H,R);
          enc = (char *) crypt(plain, salt);
          count++;
          if(strcmp(salt_and_encrypted, enc) == 0){
        printf("#%-8d%s %s\n", count, plain, enc);
      //uncomment this line if you want to speed-
  up the running time, program will find you the cracked password only without exploring all possibilites
          }
        }
      }
    }
  }

  void *kernel_function_2(char *salt_and_encrypted){
    int E,J,A;
    char salt[7];
    char plain[7];
    char *enc;
    int count = 0;

    substr(salt, salt_and_encrypted, 0, 6);
```

```
    for(E='N';E<='Z'; E++){
      for(J='A'; J<='Z'; J++){
        for(A=0; A<=99; A++){
          sprintf(plain, "%c%c%02d", E,J,A);
          enc = (char *) crypt(plain, salt);
          count++;
          if(strcmp(salt_and_encrypted, enc) == 0){
        printf("#%-8d%s %s\n", count, plain, enc);
      //uncomment this line if you want to speed-
  up the running time, program will find you the cracked password only without exploring all possibilites
          }
        }
      }
    }
  }

  int time_difference(struct timespec *start, struct timespec *finish,
                      long long int *difference) {
    long long int ds =  finish->tv_sec - start->tv_sec;
    long long int dn =  finish->tv_nsec - start->tv_nsec;

    if(dn < 0 ) {
      ds--;
      dn += 1000000000;
    }
    *difference = ds * 1000000000 + dn;
    return !(*difference > 0);
  }

  int main() {
  int i;
    struct timespec start, finish;
    long long int time_elapsed;
```

```
    clock_gettime(CLOCK_MONOTONIC, &start);


      shreethread();


    clock_gettime(CLOCK_MONOTONIC, &finish);
    time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
           (time_elapsed/1.0e9));

    return 0;
  }
```

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

	Time (In second)	Time (In nanosecond)
Time elapsed	126.233844879	126233844879
Time elapsed	119.113784190	119113784190
Time elapsed	188.307031907	188307031907
Time elapsed	193.083221601	193083221601
Time elapsed	193.778107099	193778107099
Time elapsed	210.900023293	2120900023293
Time elapsed	197.794822583	197794822583
Time elapsed	201.071428790	201071142879
Time elapsed	204.591146124	204591146124
Time elapsed	229.514197990	229514197990
Mean Time	**186.438760846**	**377438732255**

It is clearly seen from the table above that cracking passwords with threads takes more time than cracking passwords without threads. This is because the multithreaded program has two functions that make it simple to merge loops from A-M and N-Z. As a multi-threaded program, more threads are run in a multi-core CPU than a single threaded program, resulting in a faster password combination.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//__global__ --> GPU function which can be launched by many blocks and threads
//__device__ --> GPU function or variables
//__host__ --> CPU function or variables

// Compile this program with --->  nvcc -o PasswordCrack PasswordCrack.cu



//This function encrypt the plain provided values using CudaCrypt method and compares the result with encrypted password and finds the password


__device__ char* CudaCrypt(char* rawPassword){

  char * newPassword = (char *) malloc(sizeof(char) * 11);
```

```
  newPassword[0] = rawPassword[0] + 2;
  newPassword[1] = rawPassword[0] - 2;
  newPassword[2] = rawPassword[0] + 1;
  newPassword[3] = rawPassword[1] + 3;
  newPassword[4] = rawPassword[1] - 3;
  newPassword[5] = rawPassword[1] - 1;
  newPassword[6] = rawPassword[2] + 2;
  newPassword[7] = rawPassword[2] - 2;
  newPassword[8] = rawPassword[3] + 4;
  newPassword[9] = rawPassword[3] - 4;
  newPassword[10] = '\0';

  for(int i =0; i<10; i++){
          if(i >= 0 && i < 6){ //checking all lower case letter limits
                  if(newPassword[i] > 122){
                          newPassword[i] = (newPassword[i] - 122) + 97;
                  }else if(newPassword[i] < 97){
                          newPassword[i] = (97 - newPassword[i]) + 97;
                  }
          }else{ //checking number section
                  if(newPassword[i] > 57){
                          newPassword[i] = (newPassword[i] - 57) + 48;
                  }else if(newPassword[i] < 48){
                          newPassword[i] = (48 - newPassword[i]) + 48;
                  }
          }
  }
  return newPassword;
}
__device__ int is_match(char* attempt){
      char password[]="sc55";
```

```
    char *a=attempt;

    char *p=CudaCrypt(password);
    //printf("Encrypted Password: %s\n",a);
    // printf("Plain Password: %s\n",p);
  while (*a == *p){
  //printf("possible Plain Passwords: %s\n",a);
  if (*a == '\0')
  {
  printf("Encrypted Password: %s\n",attempt);
  printf("found password: %s\n",password);
  break;
  }
  a++;
  p++;
  }
  return 0;
}

__global__ void crack(char * alphabet, char * numbers){
  char genRawPass[4];

  genRawPass[0] = alphabet[blockIdx.x];
  genRawPass[1] = alphabet[blockIdx.y];

  genRawPass[2] = numbers[threadIdx.x];
  genRawPass[3] = numbers[threadIdx.y];


  char *generated=CudaCrypt(genRawPass);
```

```
  //firstLetter - 'a' - 'z' (26 characters)
      //secondLetter - 'a' - 'z' (26 characters)
      //firstNum - '0' - '9' (10 characters)
      //secondNum - '0' - '9' (10 characters)
  is_match(generated);



}

int time_difference(struct timespec *start, struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;

  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}

int main(int argc, char ** argv){
  struct timespec start, finish;
            long long int time_elapsed;
             //Start Timer
            clock_gettime(CLOCK_MONOTONIC, &start);

  char cpuAlphabet[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
  char cpuNumbers[26] = {'0','1','2','3','4','5','6','7','8','9'};
```

```
  char * gpuAlphabet;
  cudaMalloc( (void**) &gpuAlphabet, sizeof(char) * 26);
  cudaMemcpy(gpuAlphabet, cpuAlphabet, sizeof(char) * 26, cudaMemcpyHostToDevice);

  char * gpuNumbers;
  cudaMalloc( (void**) &gpuNumbers, sizeof(char) * 26);
  cudaMemcpy(gpuNumbers, cpuNumbers, sizeof(char) * 26, cudaMemcpyHostToDevice);

  crack<<< dim3(26,26,1), dim3(10,10,1) >>>( gpuAlphabet, gpuNumbers);
  cudaThreadSynchronize();

  clock_gettime(CLOCK_MONOTONIC, &finish);
       time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,(time_elapsed/1.0e9));

return 0;
}
```

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

S.N	Running time for original Password cracking (in seconds)	Running time for original password cracking with p_thread (in seconds)	Running time for cuda password cracking (in seconds)
1	125.3042736	102.8409	0.00000015
2	124.6004254	103.4054	0.000000043
3	126.0476163	107.9834	0.000000043
4	124.2047332	112.8563	0.000000043
5	124.9973153	105.3332	0.000000043
6	125.1319259	114.0182	0.000000032
7	127.501112	104.3952	0.000000048
8	127.308642	102.8627	0.000000033
9	135.2202762	109.6538	0.000000031
10	131.5030401	108.4377	0.000000032
	Mean time=127.0818364	Mean time=107.0719	Mean time=0.0000000363

- Write a short analysis of the results

In the sense of password cracking, it is obvious from the table above that the GPU is much faster than the CPU. It is because the processing of password cracking is parallel in cuda cracking while password cracking is sequential in normal version because of which password combination in cuda version is faster. Likewise, posix threads are heavier and take more time to switch between threads than cuda threads, which is the explanation for slower than cuda password cracking as cuda thread runs in 'warps' synchronously. For another explanation, normal password cracking is also slow, i.e. original and p-thread version has encrypt password using sha-12 encryption that generates more character encrypted password and also salt is applied before encryption in normal password than normal 10-digit encryption used in CUDA because of which CPU decryption time also increases.

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

```
#include "lodepng.h"
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>

//compile with "nvcc GaussianBlur.cu loadpng.cpp -o gaussianBlur"
```

```
__global__ void blurImage(unsigned char *newImage, unsigned char *image, unsigned int width, unsigned int height)
{

    int r = 0;
    int g = 0;
    int b = 0;
    int t = 0;
    int row, col;
    int count = 0;

    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int pixel = idx * 4;

    for (row = (pixel - 4); row <= (pixel + 4); row += 4)
    {
        // Checking conditions so pixel is available at x
        if ((row > 0) && row < (height * width * 4) && ((row - 4) / (4 * width) == pixel / (4 * width)))
        {
            for (col = (row - (4 * width)); col <= (row + (4 * width)); col += (4 * width))
            {
                if (col > 0 && col < (height * width * 4))
                {
                    r += image[col];
                    g += image[1 + col];
                    b += image[2 + col];
                    count++;
                }
            }
        }
    }

    t = image[3 + pixel];

```

```
        newImage[pixel] = r / count;
        newImage[1 + pixel] = g / count;
        newImage[2 + pixel] = b / count;
        newImage[3 + pixel] = t;
    }
    int time_difference(struct timespec *start,
                        struct timespec *finish,
                        long long int *difference)
    {
        long long int ds = finish->tv_sec - start->tv_sec;
        long long int dn = finish->tv_nsec - start->tv_nsec;

        if (dn < 0)
        {
            ds--;
            dn += 1000000000;
        }
        *difference = ds * 1000000000 + dn;
        return !(*difference > 0);
    }
    int main(int argc, char **argv)
    {
        struct timespec start, finish;

        clock_gettime(CLOCK_MONOTONIC, &start);
        clock_gettime(CLOCK_MONOTONIC, &finish);

        long long int time_elapsed;
        time_difference(&start, &finish, &time_elapsed);
        printf("Time elapsed was %lldns or %0.9fs\n", time_elapsed, (time_elapsed / 1.0e9));

        unsigned char *image;
        unsigned int width;
        unsigned int height;
```

```
    const char *filename = "hck.png";
    const char *newFileName = "output.png";

    lodepng_decode32_file(&image, &width, &height, filename);

    printf("Image width = %d height = %d\n", width, height);
    const int ARRAY_SIZE = width * height * 4;
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(unsigned char);

    unsigned char host_imageInput[ARRAY_SIZE * 4];
    unsigned char host_imageOutput[ARRAY_SIZE * 4];

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        host_imageInput[i] = image[i];
    }

    // declare GPU memory pointers
    unsigned char *d_in;
    unsigned char *d_out;

    // allocate GPU memory
    cudaMalloc((void **)&d_in, ARRAY_BYTES);
    cudaMalloc((void **)&d_out, ARRAY_BYTES);

    cudaMemcpy(d_in, host_imageInput, ARRAY_BYTES, cudaMemcpyHostToDevice);

    // launch the kernel
    blurImage<<<height, width>>>(d_out, d_in, width, height);

    // copy back the result array to the CPU
    cudaMemcpy(host_imageOutput, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

    lodepng_encode32_file(newFileName, host_imageOutput, width, height);
```

```
    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
#include "lodepng.h"
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>

//compile with "nvcc GaussianBlur.cu loadpng.cpp -o gaussianBlur"

__global__ void blurImage(unsigned char *newImage, unsigned char *image, unsigned int width, unsigned int height)
{

    int r = 0;
    int g = 0;
    int b = 0;
    int t = 0;
    int row, col;
    int count = 0;

    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int pixel = idx * 4;

    for (row = (pixel - 4); row <= (pixel + 4); row += 4)
    {
        // Checking conditions so pixel is available at x
        if ((row > 0) && row < (height * width * 4) && ((row - 4) / (4 * width) == pixel / (4 * width)))
        {
            for (col = (row - (4 * width)); col <= (row + (4 * width)); col += (4 * width))
            {
                if (col > 0 && col < (height * width * 4))
```

```
                {
                    r += image[col];
                    g += image[1 + col];
                    b += image[2 + col];
                    count++;
                }
            }
        }
    }

    t = image[3 + pixel];

    newImage[pixel] = r / count;
    newImage[1 + pixel] = g / count;
    newImage[2 + pixel] = b / count;
    newImage[3 + pixel] = t;
}
int time_difference(struct timespec *start,
                    struct timespec *finish,
                    long long int *difference)
{
    long long int ds = finish->tv_sec - start->tv_sec;
    long long int dn = finish->tv_nsec - start->tv_nsec;

    if (dn < 0)
    {
        ds--;
        dn += 1000000000;
    }
    *difference = ds * 1000000000 + dn;
    return !(*difference > 0);
}
int main(int argc, char **argv)
{
```

```
    struct timespec start, finish;

    clock_gettime(CLOCK_MONOTONIC, &start);
    clock_gettime(CLOCK_MONOTONIC, &finish);

    long long int time_elapsed;
    time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9fs\n", time_elapsed, (time_elapsed / 1.0e9));

    unsigned char *image;
    unsigned int width;
    unsigned int height;
    const char *filename = "hck.png";
    const char *newFileName = "output.png";

    lodepng_decode32_file(&image, &width, &height, filename);

    printf("Image width = %d height = %d\n", width, height);
    const int ARRAY_SIZE = width * height * 4;
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(unsigned char);

    unsigned char host_imageInput[ARRAY_SIZE * 4];
    unsigned char host_imageOutput[ARRAY_SIZE * 4];

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        host_imageInput[i] = image[i];
    }

    // declare GPU memory pointers
    unsigned char *d_in;
    unsigned char *d_out;

    // allocate GPU memory
```

```
    cudaMalloc((void **)&d_in, ARRAY_BYTES);
    cudaMalloc((void **)&d_out, ARRAY_BYTES);

    cudaMemcpy(d_in, host_imageInput, ARRAY_BYTES, cudaMemcpyHostToDevice);

    // launch the kernel
    blurImage<<<height, width>>>(d_out, d_in, width, height);

    // copy back the result array to the CPU
    cudaMemcpy(host_imageOutput, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

    lodepng_encode32_file(newFileName, host_imageOutput, width, height);

    cudaFree(d_in);
    cudaFree(d_out);
```

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

S.N.	**Running time for original image blur (in seconds)**	**Running time for cuda image blur (in seconds)**
1	0.000000333	0.000000075
2	0.000000389	0.000000082
3	0.000000399	0.000000086
4	0.000000355	0.000000085
5	0.000000383	0.000000089
6	0.000000354	0.000000089
7	0.000000354	0.000000079
8	0.000000361	0.000000098
9	0.000000361	0.000000077
10	0.000000354	0.000000073
	Mean=0.000000365	Mean=0.0000000822

- Write a short analysis of the results

  As expected, the CUDA version took less time to blur the image than the original version. This is because, unlike the original version, cuda threads are light weight and run parallel and synchronously in 'warps'. Gpu has many cuda cores that are also responsible for faster calculation, resulting in faster blurring of images.