UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

High Performance Computing

(6CS005)

Portfolio Report

Student Id : [2040364]

Student Name : Poojan Shrestha

Student ID : NP03A170005

Cohort/Batch : 4

Submitted on : 26th December 2020

Table of Contents

Task 1: Parallel and Distributed Systems

1. **What are threads and what they are designed to solve?**
   Ans: A thread is a small collection of instructions programmed independently of the parent to be scheduled and executed by the CPU. For instance, a program might have an open thread waiting for a certain occurrence to occur or running a different job, freeing other tasks to be executed by the main program.
   In a modern programming language, threads are designed because whenever a process has several tasks to execute independently of others.

2. **Name and describe two process scheduling policies. Which one is preferable and does the choice of policies have any influence on the behavior of Java threads?**
   Ans: The two process scheduling policies are as follows:
   - **Pre-emptive Scheduling**
     The tasks are often allocated to their priorities in Preemptive Scheduling. In this scheduling tasks are often prioritized according to their importance such as the higher priority task are executed even though the lower priority task is still running. The lower task is paused for a while when the higher priority task completes its execution.
   - **Co-operative Scheduling**
     The Processor has been assigned to a particular process in this form of a scheduling system. Either by swapping the background or terminating, the mechanism that holds the CPU busy will free the CPU. It is the only strategy for different hardware platforms that can be used. That is because, like preemptive scheduling, it does not require special hardware (for example, a timer).
     Pre-emptive is more preferred and the thread scheduling algorithm of the Java runtime framework is also pre-emptive.

3. **Distinguish between Centralized and Distributed systems?**
   Ans: In Centralized System, all the computing or tasks are performed on a single computer. It is a system that uses terminals temporarily connected to the central computer to compute at a central location.
   In Distributed System all the computing or tasks are distributed to multiple computers. It is a system that has a set of independent computers interconnected where tasks are distributed for concurrent processing without a server.

4. **Explain transparency in DS?**
   Ans: Transparency is an essential aspect of distributed systems, as it makes running them more user-friendly, simpler, or visible in the eyes of the user. Users should be ignorant of the position of the services and should be clear about the switch from a local computer to a remote machine.

5. **The following three statements contain a flow dependency, an antidependency and an output dependency. Can you identify each?**

   Given that you are allowed to reorder the statements, can you find a permutation that produces the same values for the variables C and B as the original given statements?

   Show how you can reduce the dependencies by combining or rearranging calculations and using temporary variables.

   Note: Show all the works in your report and produce a simple C code simulate the process of producing the C and B values. (2marks for solving dependencies and 2marks for the code)

   B=A+C
   B=C+D
   C=B+D

   Ans: B = A + C is a flow dependency, C = B + D is an anti-dependency, B = C + D is an output dependency.

6. **What output do the following 2 programs produce and why?**

#include <pthread.h> #include <stdio.h>	#include <pthread.h> #include <stdio.h>

int counter;
static void * thread_func(void * _tn) {
int i;
 	
for (i = 0; i < 100000; i++)
 	
counter++;
 	
return NULL;
 	
}
	int counter;
	static void * thread_func(void * _tn) {
	int i;
	
	for (i = 0; i < 100000; i++)
	
	counter++;
	
	return NULL;
	
	}

int main() {
 	
int i, N = 5;
 	
pthread_t t[N];
 	
for (i = 0; i < N; i++)
 	
pthread_create(&t[i], NULL, thread_func, NULL);
 	
for (i = 0; i < N; i++) pthread_join(t[i], NULL);
 	
printf("%d\n", counter);
 	
return 0;
 	
}
	int main() {
	
	int i, N = 5;
	
	pthread_t t[N];
	
	for (i = 0; i < N; i++) {
	
	pthread_create(&t[i], NULL, thread_func, NULL);
	
	pthread_join(t[i], NULL);
	
	}
	
	printf("%d\n", counter);
	
	return 0;
	
	}

Output: 349151	**Output: 500000**

Because both programs use a thread function that performs a thread count for the provided unassigned integer [N]. The first program runs an extra loop.

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

Part A: Single Thread Matrix Multiplication

Study the following algorithm that is written for multiplying two matrices A and B and storing the result in C.

Now answer each of the following questions:

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

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

    {
     for (int j = 0; j < M; j++)
         {
     C[i][j] = 0;

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

            {
              C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
         }
     }
```

- **Analyze the above algorithm for its complexity.**
  Ans: O(n^3) is the complexity of the given program.

- **Suggest at least three different ways to speed up the matrix multiplication algorithm given here. (Pay special attention to the utilization of cache memory to achieve the intended speed up).**
  Ans: Different ways to speed up the matrix multiplication would be Divide and Conquer and Strassen's Matrix Multiplication.

- **Write your improved algorithms as pseudo-codes using any editor. Also, provide a reasoning as to why you think the suggested algorithm is an improvement over the given algorithm.**
  Ans: In the algorithm given above, 8 multiplications are executed for the matrices of the size n/2 * n/2 and 4 additions but the improved algorithm performs 7 multiplications hence, the improved algorithm is faster.

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

- **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).**

Code:

```
if n = threshold then compute
        Z = x + y is a conventional matrix.
    Else
        Partition x into four sub matrices x11, x12, x21, x22.
        Partition y into four sub matrices y11, y12, y21, y22.
        Strassen ( n/2, x11 + x22, y11 + y22, w1)
        Strassen ( n/2, x21 + x22, y11, w2)
        Strassen ( n/2, x11, x12 - y22, w3)
        Strassen ( n/2, x22, y21 - y11, w4)
        Strassen ( n/2, x11 + x12, y22, w5)
        Strassen ( n/2, x21 - x11, y11 + y22, w6)
        Strassen ( n/2, x12 - x22, y21 + y22, w7)

        Z = w1 + w4 - w5 + w7       w3 + w5
            w2 + w4                 w1 + w3 - w2 - w6

    end if

    return(W)

end
```

Part B: Write a code to implement matrix multiplication using multithreading

- **Write a C program to implement matrix multiplication using multithreading. The number of threads should be configurable at run time, for example, read via an external file.**

- **The code should print the time it takes to do the matrix multiplication using the given number of threads by averaging it over multiple runs.**

- **Plot the time it takes to complete the matrix multiplication against the number of threads and identify a sweet spot in terms of the optimal number of threads needed to do the matrix multiplication.**

(In doing this exercise, you should use matrices of large sizes e.g.,1024 * 1024 sized matrices)

Code:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define MAT_SIZE 1024
#define MAX_THREADS 100

/********
 Compile with cc -o task2b 2040364_Task2_B.c -pthread

 ./task2b

********/

int i,j,k;            //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

//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){

    args* p = arg;

    //Calculating Each Element in Result Matrix Using Passed Arguments
    for(int a=0;a<j;a++){
        result[p->x][p->y] += matrix1[p->x][a]*matrix2[a][p->y];
    }
    sleep(3);

    //End Of Thread
    pthread_exit(0);
}


int main(){

    //Initializing All Defined Matrices By Zero - - - - - - - - - - - - - - -//
    for(int x=0;x<10;x++){
        for(int y=0;y<10;y++){
            matrix1[x][y] = 0;
            matrix2[x][y] = 0;
            result[x][y] = 0;
        }
    }


    //Getting Matrix1 And Matrix2 Info from User - - - - - - - - - - - - - - -//

    printf(" --- Defining Matrix 1 ---\n\n");

    // Getting Row And Column(Same As Row In Matrix2) Number For Matrix1
    printf("Enter number of rows for matrix 1: ");
    scanf("%d",&i);
    printf("Enter number of columns for matrix 1: ");
    scanf("%d",&j);
```

```
    printf("\n --- Initializing Matrix 1 ---\n\n");
    for(int x=0;x<i;x++){
        for(int y=0;y<j;y++){
            printf("Enter variable [%d,%d]: ",x+1,y+1);
            scanf("%d",&matrix1[x][y]);
        }
    }

    printf("\n --- Defining Matrix 2 ---\n\n");

    // Getting Column Number For Matrix2
    printf("Number of rows for matrix 2 : %d\n",j);
    printf("Enter number of columns for matrix 2: ");
    scanf("%d",&k);

    printf("\n --- Initializing Matrix 2 ---\n\n");
    for(int x=0;x<j;x++){
        for(int y=0;y<k;y++){
            printf("Enter variable [%d,%d]: ",x+1,y+1);
            scanf("%d",&matrix2[x][y]);
        }
    }++

    //Printing Matrices - - - - - - - - - - - - - - - - - - - - - - - - - - -//

    printf("\n --- Matrix 1 ---\n\n");
    for(int x=0;x<i;x++){
        for(int y=0;y<j;y++){
            printf("%5d",matrix1[x][y]);
        }
        printf("\n\n");
    }

    printf(" --- Matrix 2 ---\n\n");
    for(int x=0;x<j;x++){
        for(int y=0;y<k;y++){
            printf("%5d",matrix2[x][y]);
        }
        printf("\n\n");
    }


    //Multiply Matrices Using 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
    args p[i*k];

    //Start Timer
    time_t start = time(NULL);

    for(int x=0;x<i;x++){
        for(int y=0;y<k;y++){

            //Initializing Parameters For Passing To Function
            p[thread_number].x=x;
            p[thread_number].y=y;

            //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]);
```

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

            thread_number++;
        }
    }


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

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


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

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


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

    printf(" ---> Time Elapsed : %.2f Sec\n\n", (double)(time(NULL) - start));


    //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]);

    return 0;
}
```

```
--- Matrix 1 ---

   1    3    5

   4    3    2

   1    6    8

--- Matrix 2 ---

   1    3    5

   2    6    8

   1    2    3

--- Multiplied Matrix ---

  12   31   44

  12   34   50

  21   55   77

---> Time Elapsed : 3.00 Sec

---> Used Threads : 9

- Thread 1 ID : 809846528
- Thread 2 ID : 801453824
- Thread 3 ID : 793061120
- Thread 4 ID : 784668416
- Thread 5 ID : 776275712
- Thread 6 ID : 767883008
- Thread 7 ID : 759490304
- Thread 8 ID : 751097600
- Thread 9 ID : 742704896
poozan@Poozan-A7:~/Herald/HPC/Assessment/6cs005_PortfolioS1_19_20_2040364_Poojan_Shrestha$
```

Part C: Password cracking using POSIX Threads

You will be provided with a template code of encrypting a password using SHA-512 algorithm. You are asked to run a code using the given template to encrypt a password contains TWO uppercase letters and TWO integer numbers (total 4 digits). Afterwards, you need to run the given code to crack the password, i.e. find the plain-text equivalents. An example password is AS12.

1. **Run the given cracking program 10 times and calculate the mean running time (Using 2 uppercase letters and 2 integer numbers).**
   Ans:

Number of Programs	Time in nanosecond	Time in seconds
1	148831480552.00	148.8314806
2	145810733719.00	145.8107337
3	145903921347.00	145.9039213
4	145833303542.00	145.8333035
5	145674117202.00	145.6741172
6	145592184620.00	145.5921846
7	145611231362.00	145.6112314
8	145605059608.00	145.6050596
9	145936199061.00	145.9361991
10	145927396998.00	145.927397
Average	**146072562801.10**	**146.0725628**

2. **In your learning journal make an estimate of how long it would take to run on the same computer if the number of initials were increased to 3. Include your working in your answer.**
   Ans: For the three initials a loop is added on the two initials code, so the loop goes through another 26 characters. So,
   Estimated Time = Original Time * 26 (Here, 26 is the number of alphabets)
   = 146.0725628 * 26
   = 3,797.8866328 seconds
   Now,
   Seconds into minutes = 3,797.8866328 / 60
   = 63.29811054666667 minutes

   Hence, the estimated time for three initials is 63 minutes.

3. **Modify the program to crack the three-initials-two-digits password given in the three_initials variable. An example password is HPC19.**

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

/***********************************************************************
*******

  Compile with:
    cc -o task2c3 2040364_Task2_C_3.c -lcrypt

    ./task2c3 > task2c3.txt
***********************************************************************
******/


int n_passwords = 1;

char *encrypted_passwords[] = {
 "$6$AS$OQ5qXRODiyBaHo4mVhIaWOm1Xd/eLiR8Egz38LxdzvRsZK8YDZSGU4856f2i/.i4IknRiEPs5lDb7JuwKSJmp."
};


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


void crack(char *salt_and_encrypted){
  int p, o, z, n;     // Loop counters
  char salt[7];    // String used in hashing the password. Need space for \0
  char plain[7];   // The combination of letters currently being checked
  char *enc;       // Pointer to the encrypted password
  int count = 0;   // The number of combinations explored so far

  substr(salt, salt_and_encrypted, 0, 6);

  for(p='A'; p<='Z'; p++){
    for(o='A'; o<='Z'; o++){
     for(z='A'; z<='Z'; z++){
      for(n=0; n<=99; n++){
        sprintf(plain, "%c%c%c%02d", p, o, z, n);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
          printf("#%-8d%s %s\n", count, plain, enc);
        } else {
          printf(" %-8d%s %s\n", count, plain, enc);
        }
       }
      }
    }
  }
  printf("%d solutions explored\n", count);
}
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[]){
  int i;
struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);


  for(i=0;i<n_passwords;i<i++) {
    crack(encrypted_passwords[i]);
  }

 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;
}
```

4. **Write a short paragraph to compare the running time (average of 10 times) of your three_initials program with your earlier estimate. If your estimate was wrong explain why you think that is.**

Ans:

Number of Programs	Time in nanosecond	Time in seconds
1	5299177490263.00	5299.17749
2	5280573811093.00	5280.573811
3	5289536270541.00	5289.536271
4	5281426879345.00	5281.426879
5	5291635214781.00	5291.635215
6	5298254194655.00	5298.254195
7	5299132456847.00	5299.132457
8	5298324569158.00	5298.324569
9	5298789654126.00	5298.789654
10	5297874998463.00	5297.874998
Average	**5293472553927.20**	**5293.472554**

So, the average running time for three initials is 5293.472554 seconds which is 88.224542566667 minutes i.e. Approximately 1.47 hours. The estimated time for three initials was 63.29811054666667 minutes but the original running time was a lot more than estimated which is a huge difference. For such a difference in the running time, the reason could be due to the background process during the execution of three initials.

5. Modify the original version of the program to run on 2 threads. It does not need to do anything fancy, just follow the following algorithm.

- **Record the system time a nanosecond timer**
- **Launch thread_1 that calls kernel_function_1 that can search for passwords starting from A-M**
- **Launch thread_2 that calls kernel_function_2 that can search for passwords starting from N-Z**
- **Wait for thread_1 to finish**
- **Wait for thread_2 to finish**
- **Record the system time using a nanosecond time and print the elapsed time**

Code:

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

/*************************************************************************
*******

  Compile with:
    cc -o task2c5 2040364_Task2_C_5.c -lcrypt -lrt -pthread
or  gcc -pthread -o task2c5 2040364_Task2_C_5
or  gcc -o task2c5 2040364_Task2_C_5 -lpthread
or  gcc -o task2c5 2040364_Task2_C_5 -pthread

    ./task2c5 > task2c5.txt
*************************************************************************
******/


int n_passwords = 1;

char *encrypted_passwords[] = {
"$6$AS$OQ5qXRODiyBaHo4mVhIaWOm1Xd/eLiR8Egz38LxdzvRsZK8YDZSGU4856f2i/.i4IknRiEPs5lDb7JuwKSJmp."
};



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


void run()
{
  int i;
pthread_t thread_1, thread_2;

    void *kernel_function_1();
    void *kernel_function_2();
for(i=0;i<n_passwords;i<i++) {


    pthread_create(&thread_1, NULL,kernel_function_1, encrypted_passwords[i]);
    pthread_create(&thread_2, NULL,kernel_function_2, encrypted_passwords[i]);

    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
 }
}

void *kernel_function_1(void *salt_and_encrypted){
  int p, o, z;     // Loop counters
  char salt[7];    // String used in hahttps://www.youtube.com/watch?v=L8yJjIGleMwshing the password. Need space
  char plain[7];   // The combination of letters currently being checked
  char *enc;       // Pointer to the encrypted password
  int count = 0;   // The number of combinations explored so far

  substr(salt, salt_and_encrypted, 0, 6);

  for(p='A'; p<='M'; p++){
    for(o='A'; o<='Z'; o++){
      for(z=0; z<=99; z++){
        sprintf(plain, "%c%c%02d", p,o,z);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
          printf("#%-8d%s %s\n", count, plain, enc);
```

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


void *kernel_function_2(void *salt_and_encrypted){
  int p, o, z;     // Loop counters
  char salt[7];    // String used in hahttps://www.youtube.com/watch?v=L8yJjIGleMwshing the
  password. Need space
  char plain[7];   // The combination of letters currently being checked
  char *enc;       // Pointer to the encrypted password
  int count = 0;   // The number of combinations explored so far

  substr(salt, salt_and_encrypted, 0, 6);

  for(p='N'; p<='Z'; p++){
    for(o='A'; o<='Z'; o++){
      for(z=0; z<=99; z++){
        sprintf(plain, "%c%c%02d", p,o,z);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
          printf("#%-8d%s %s\n", count, plain, enc);
        }
      }
    }
  }
  printf("%d solutions explored\n", count);
}

//Calculating time

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;

    clock_gettime(CLOCK_MONOTONIC, &start);



            run();

    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;
}
```

Output:

```
poozan@Poozan-A7:~/Herald/HPC/Assessment/6cs005_PortfolioS1_19_20_2040364_Poojan_Shrestha$ cc -o task2c5
poozan@Poozan-A7:~/Herald/HPC/Assessment/6cs005_PortfolioS1_19_20_2040364_Poojan_Shrestha$ chmod a+x mr.
poozan@Poozan-A7:~/Herald/HPC/Assessment/6cs005_PortfolioS1_19_20_2040364_Poojan_Shrestha$ ./mr.py ./tas
Time elapsed was 141122062844ns or 141.122062844s
Time elapsed was 145910649716ns or 145.910649716s
Time elapsed was 148002100925ns or 148.002100925s
Time elapsed was 143460225450ns or 143.460225450s
Time elapsed was 144553274685ns or 144.553274685s
Time elapsed was 150270064771ns or 150.270064771s
Time elapsed was 142057749906ns or 142.057749906s
Time elapsed was 148280849281ns or 148.280849281s
Time elapsed was 146781496012ns or 146.781496012s
Time elapsed was 140445026602ns or 140.445026602s
poozan@Poozan-A7:~/Herald/HPC/Assessment/6cs005_PortfolioS1_19_20_2040364_Poojan_Shrestha$
```

6. **Compare the results of the mean running time of the original program, (obtained by step no. 1), with the mean running time of the multithread version (10 running times).**
   Ans:

Number of Time Programs ran	Time taken by Original Program	Time taken by Multithread Program
1	148.831480552	142.051412568
2	145.810733719	138.248301761
3	145.903921347	135.939179243
4	145.833303542	139.407503836
5	145.674117202	141.657376717
6	145.592184620	142.386132090
7	145.611231362	137.325570259
8	145.605059608	139.540573378
9	145.936199061	136.086533270
10	145.927396998	143.868812312
Average (seconds)	**146.072562801**	**139.651139543**

So, the results were not too different, but the multithread ran a bit faster that took 139.6511395 seconds than the original program that took about 146.0725628 seconds because in original program only single thread was used but in Multithread program two threads were used.

Task 3: Applications of Password Cracking and Image Blurring using HPC based CUDA system

Part A: Password cracking using CUDA

Using the same concept as before, you will now crack passwords using CUDA. Your program will take in an encrypted password and decrypt using many threads. CUDA allows multidimensional thread configurations so your kernel function (which runs on the GPU) will need to be modified according to how you configure the execution command.

Crack passwords using CUDA

Code:

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


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



/***********************************************************************
*******
 pass = rnqdwy9523

  nvcc -o task3a 2040364_Task3_A.cu

  ./task3a >task3a.txt
************************************************************************
******/


char *encrypted_passwords[]{
    "rnqdwy9523"
};


__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;
}

__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];

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

//Idx --> gives current index of the block or thread

printf("%c %c %c %c = %s\n", genRawPass[0],genRawPass[1],genRawPass[2],genRawPass[3], CudaCrypt(genRawPass));




}

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;
    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 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();



    //crack <<<26,26>>>();
    //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;
}
```

Output:

```
# p z 7 7 = rnqdwy9523
```

Comparison, analysis, and evaluation of CUDA version for password cracking (compare with normal and "pthread" version)

Number of Time Programs ran	Time taken by Original Program	Time taken by Multithread Program	Time taken by CUDA Program
1	148.831480552	142.051412568	0.292592451
2	145.810733719	138.248301761	0.243579181
3	145.903921347	135.939179243	0.245739428
4	145.833303542	139.407503836	0.255705678
5	145.674117202	141.657376717	0.244173225
6	145.592184620	142.386132090	0.250527895
7	145.611231362	137.325570259	0.256544903
8	145.605059608	139.540573378	0.241956623
9	145.936199061	136.086533270	0.245235216
10	145.927396998	143.868812312	0.254830959
Average (seconds)	**146.072562801**	**139.651139543**	**0.253088556**

So, the time taken by the CUDA version of password cracking is about 0.25 seconds which is a lot faster than the other two i.e., Original program and Multithread Program. The reason for such significant difference is CUDA run on GPU and POSIX run CPU which make GPU substantially with more CUDA and with a capability of parallel computing which can large amount of data parallelly.

Part B: Image blur using multi dimension gaussian matrices (multi-pixel processing)
Applying gaussian blur with 3x3 matrix using CUDA

Applying gaussian blur with multiple dimension matrices using CUDA

Code:

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

#include "lodepng.h"

/********
    Compile with nvcc 2040364_Task3_B.cu lodepng.cpp -o task3b

    ./task3b
*********/

__global__ void blur_image(unsigned char * gpu_imageOuput, unsigned char * gpu_imageInput,int
width,int height){

    int counter=0;

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


    int i=blockIdx.x;
    int j=threadIdx.x;


    float t_r=0;
    float t_g=0;
    float t_b=0;
    float t_a=0;
    float s=1;

    if(i+1 && j-1){

        // int pos= idx/2-2;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x-1;
        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;



    }

    if(j+1){

        // int pos= idx/2-2;

        int pos=blockDim.x * (blockIdx.x) + threadIdx.x+1;

        int pixel = pos*4;
```

```
        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;
    }

    if(i+1 && j+1){

        // int pos= idx/2+1;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x+1;


        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;


    }

    if(i+1){
        // int pos= idx+1;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;



    }

    if(j-1){

        // int pos= idx*2-2;
        int pos=blockDim.x * (blockIdx.x) + threadIdx.x-1;

        int pixel = pos*4;
```

```
        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;




    }

    if(i-1){

        // int pos= idx-1;
        int pos=blockDim.x * (blockIdx.x-1) + threadIdx.x;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;


    }

    int current_pixel=idx*4;

    gpu_imageOuput[current_pixel]=(int)t_r/counter;
    gpu_imageOuput[1+current_pixel]=(int)t_g/counter;
    gpu_imageOuput[2+current_pixel]=(int)t_b/counter;
    gpu_imageOuput[3+current_pixel]=gpu_imageInput[3+current_pixel];


}

//Calculating Time

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;
    clock_gettime(CLOCK_MONOTONIC, &start);

    unsigned int error;
    unsigned int encError;
    unsigned char* image;
    unsigned int width;
    unsigned int height;
    const char* filename = "Normal_image.png";
    const char* newFileName = "Blurred_image.png";

    error = lodepng_decode32_file(&image, &width, &height, filename);
    if(error){
        printf("error %u: %s\n", error, lodepng_error_text(error));
    }

    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
    blur_image<<<height, width>>>(d_out, d_in,width,height);


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

    encError = lodepng_encode32_file(newFileName, host_imageOutput, width, height);
    if(encError){
        printf("error %u: %s\n", error, lodepng_error_text(encError));
    }

    //free(image);
    //free(host_imageInput);
    cudaFree(d_in);
    cudaFree(d_out);

    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;
}
```

Input Image:

Output Image: