

UNIVERSITY PARTNER



# High Performance Computing

(6CS014)

Final Portfolio Report

Student Id : 2227097

Student Name : Radhika Neupane

Group: L6CG10

Lecturer : Yamu Poudel

Tutor : Yamu Poudel

Cohort: 8

Submitted on: 27th Jan 2024

## **Abstract**

This report is a portfolio report which consists of four different tasks done for the High Performance Computing(6CS014). Each of the tasks focus on the parallel processing using multithreading and CUDA. For the first task we have done password cracking using multithreading where a library called 'crypt' is used to decrypt the password. In this program different possible password combinations were explored using multithreading. For the second task matrix multiplication is done using multithreading where matrices are read from a particular file and if the condition is matched multiplication is performed. It also shows how the dynamic memory is allocated in matrices. For the third task, similar to task one password was cracked but by using CUDA. The passwords were encrypted by parallel processing on the GPU(Graphics Processing Unit). CUDA Kernel Function which is designed to run on a GPU is used to generate the password and compare the password and encrypted password. For the last task a box blur filter is applied to a png image using CUDA. Those images are decoded into an array after that with the CUDA Kernel function box blur filter was executed. Multithreading is a technique where multiple threads run simultaneously within the same program. On GPU 'cudaMalloc' is used to allocate the memory whereas 'cudaMemcpy' is used to transfer data from the CPU to GPU. The dynamic memory allocation is done which involves allocating memory for the program during its execution.

## **Acknowledgement**

I would like to thank and express my gratitude to lecturer/tutor Mr. Yamu Poudel for his guidance for this entire semester on High Performance Computing. His guidance and patients were so incredible for all of the four tasks. Along with this I would like to thank each and every one presented in the High-Performance Computing course.

## Table of contents

1.Password cracking using multithreading.....	1
2. Matrix Multiplication using multithreading.....	3
3. Password Cracking using CUDA.....	7
4. Box Blur using CUDA.....	10

## 1.Password cracking using multithreading

In this task, you will be asked to use the “crypt” library to decrypt a password using multithreading. You will be provided with two programs. The first program is called “EncryptSHA512.c” which allows you to encrypt a password. For this assessment, you will be required to decrypt a 4-character password consisting of 2 capital letters, and 2 numbers. The format of the password should be “LetterLetterNumberNumber.” For example, “HP93.” Once you have generated your password, this should then be entered into your program to decrypt the password. The method of input for the encrypted password is up to you. The second program is a skeleton code to crack the password in regular C without any multithreading syntax. Your task is to use multithreading to split the workload over many threads and find the password. Once the password has been found, the program should finish, meaning not all combinations of 2 letters and 2 numbers should be explored unless it’s ZZ99 and the last thread happens to finish last.

**Cracks a password using multithreading and dynamic slicing based on thread count (75 marks)**

**Program finishes appropriately when password has been found (25 marks)**

This program cracks the password of two letters and two digits by using multiple threads. The password which needs to be cracked is defined in the ‘salt\_and\_encrypted’ global variable.

```
printf("\n\n |Creating threads and checking for a matching hash| \n");

// Defining the encrypted password to be cracked inside ""
salt_and_encrypted = "$6$AS$9IwGTn5WbH$alUs4ba3Jb0f0UX/vlyD71Z4M2F6Yusz5k2WQE0FxqLIY80tudGtcFttqr/Zq6RIPjHkl/t2Pp1";

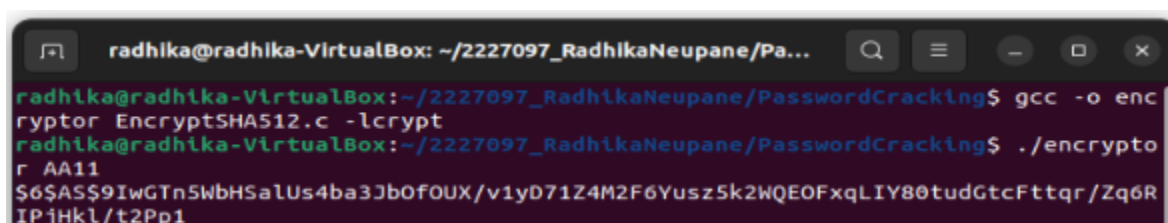
printf("salt_and_encrypted: %s\n", salt_and_encrypted);

//For thread execution
```

Talking about the ‘CRackAZ99.c’ we have included the header files for the input/output, string manipulation, memory allocation, string manipulation ,semaphore handling and multithreading. The global variables are declared as ‘count’ to track the number of

password combinations. We have 'totalThread' which will store the number of threads and 'threadCount' is set to 26 which is the number of letters in the English alphabet. The structure called 'threadInfo' is defined which holds the values of start and end for each thread. The function called 'substr' is declared which takes a substring of the specific length and copies it to the destination string from the source string. Next the function 'crack' is declared which has the core of the password-cracking logic. It also takes the information related to thread and also extracts the salt from 'salt\_and\_encrypted' and then iterates through the combinations of ASCII values for two characters and two digit numbers. The function 'crypt' is used to encrypt the password combination which was generated and also compares the targeted encrypted password. In case if the match is found after comparison the result is printed with its details before exiting. The function is then set up and then parallel cracking is executed. In this function, the password combination is determined for each thread along with the calculation of start and end ASCII values for each thread. At the end of the program, a function is declared to validate the command line arguments and the result is printed through the driver code. Moving towards a file called 'EncryptedSHA512.C', it just takes the string and then encrypts it using 'SHA-512' with the predefined salt i.e., 'SALT' and then prints the encrypted password.

### In terminal:



```
radhika@radhika-VirtualBox: ~/2227097_RadhikaNeupane/Pa...
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$ gcc -o encryptor EncryptSHA512.c -lcrypt
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$ ./encryptor AA11
$6$AS$9IwGTn5WbH$alUs4ba3Jb0fOUX/v1yD71Z4M2F6Yusz5k2WQEOFxqLIY80tudGtcFttqr/Zq6R
IPjHkl/t2Pp1
```

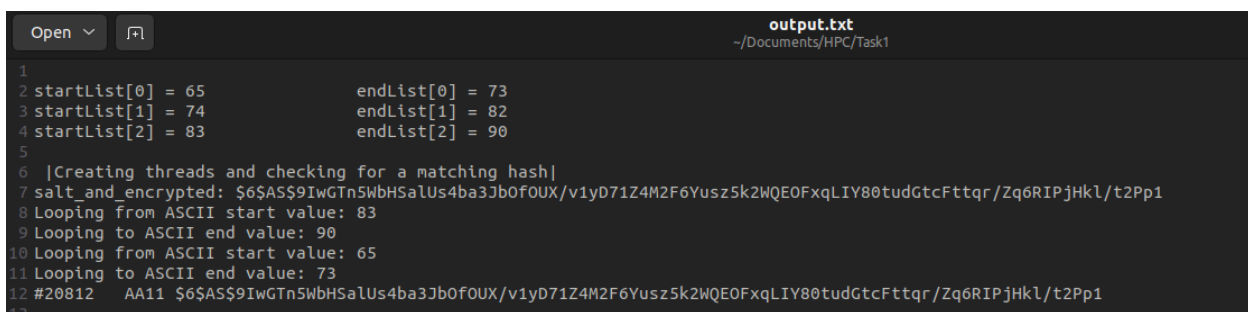
```

radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$ gcc -o CrackAZ99 CrackAZ99.c -lcrypt
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$ ./CrackAZ99 > output.txt 3
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$ cat output.txt

startList[0] = 65          endList[0] = 73
startList[1] = 74          endList[1] = 82
startList[2] = 83          endList[2] = 90

[Creating threads and checking for a matching hash]
salt_and_encrypted: $6$AS$9IwGTn5WbHSaUs4ba3Jb0f0UX/v1yD71Z4M2F6Yusz5k2WQE0FqxLIY80tudGtcFttqr/Zq6RIPjHkl/t2Pp1
Looping from ASCII start value: 83
Looping to ASCII end value: 90
Looping from ASCII start value: 65
Looping to ASCII end value: 73
#20812 AA11 $6$AS$9IwGTn5WbHSaUs4ba3Jb0f0UX/v1yD71Z4M2F6Yusz5k2WQE0FqxLIY80tudGtcFttqr/Zq6RIPjHkl/t2Pp1
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/PasswordCracking$

```



The screenshot shows a text editor window titled "output.txt" with the file path "~/Documents/HPC/Task1". The editor displays the same output as the terminal window above, with line numbers 1 through 13 on the left margin.

```

1
2 startList[0] = 65          endList[0] = 73
3 startList[1] = 74          endList[1] = 82
4 startList[2] = 83          endList[2] = 90
5
6 [Creating threads and checking for a matching hash]
7 salt_and_encrypted: $6$AS$9IwGTn5WbHSaUs4ba3Jb0f0UX/v1yD71Z4M2F6Yusz5k2WQE0FqxLIY80tudGtcFttqr/Zq6RIPjHkl/t2Pp1
8 Looping from ASCII start value: 83
9 Looping to ASCII end value: 90
10 Looping from ASCII start value: 65
11 Looping to ASCII end value: 73
12 #20812 AA11 $6$AS$9IwGTn5WbHSaUs4ba3Jb0f0UX/v1yD71Z4M2F6Yusz5k2WQE0FqxLIY80tudGtcFttqr/Zq6RIPjHkl/t2Pp1
13

```

## 2. Matrix Multiplication using multithreading

You will create a matrix multiplication program which uses multithreading. Matrices are often two-dimensional arrays varying in sizes, for your application, you will only need to multiply two-dimensional ones. Your program will read in the matrices from a supplied file (txt), store them appropriately using dynamic memory allocation features and multiply them by splitting the tasks across “n” threads (any amount of threads). You should use command line arguments to specify which file to read from (argv[1]) and the number of threads to use (argv[2]). If the number of threads requested by the user is greater than the biggest dimension of the matrices to be multiplied, the actual number of threads used in the calculation should be limited to the maximum dimension of the

matrices. Your program should be able to take “any” size matrices and multiply them depending on the data found within the file. Some sizes of matrices cannot be multiplied together, for example, if Matrix A is 3x3 and Matrix B is 2x2, you cannot multiply them. If Matrix A is 2x3 and Matrix B is 3x2, then this can be multiplied. You will need to research how to multiply matrices, this will also be covered in the lectures (briefly). If the matrices cannot be multiplied, your program should output an error message notifying the user and move on to the next pair of matrices. Your program should store the results of your successful matrix multiplications in a text file – for every 2 matrices, there should be one resulting matrix. As a minimum, you are expected to use the standard C file handling functions: `fopen()`, `fclose()`, `fscanf()`, and `fprintf()`, to read and to write your files. `stdin` and `stdout` redirection will not be acceptable.

**Read data from file appropriately (20 marks)**

**Using dynamic memory (malloc) for matrix A and matrix B (10 marks)**

**Creating an algorithm to multiply matrices correctly (20 marks)**

**Using multithreading with equal computations (30 marks)**

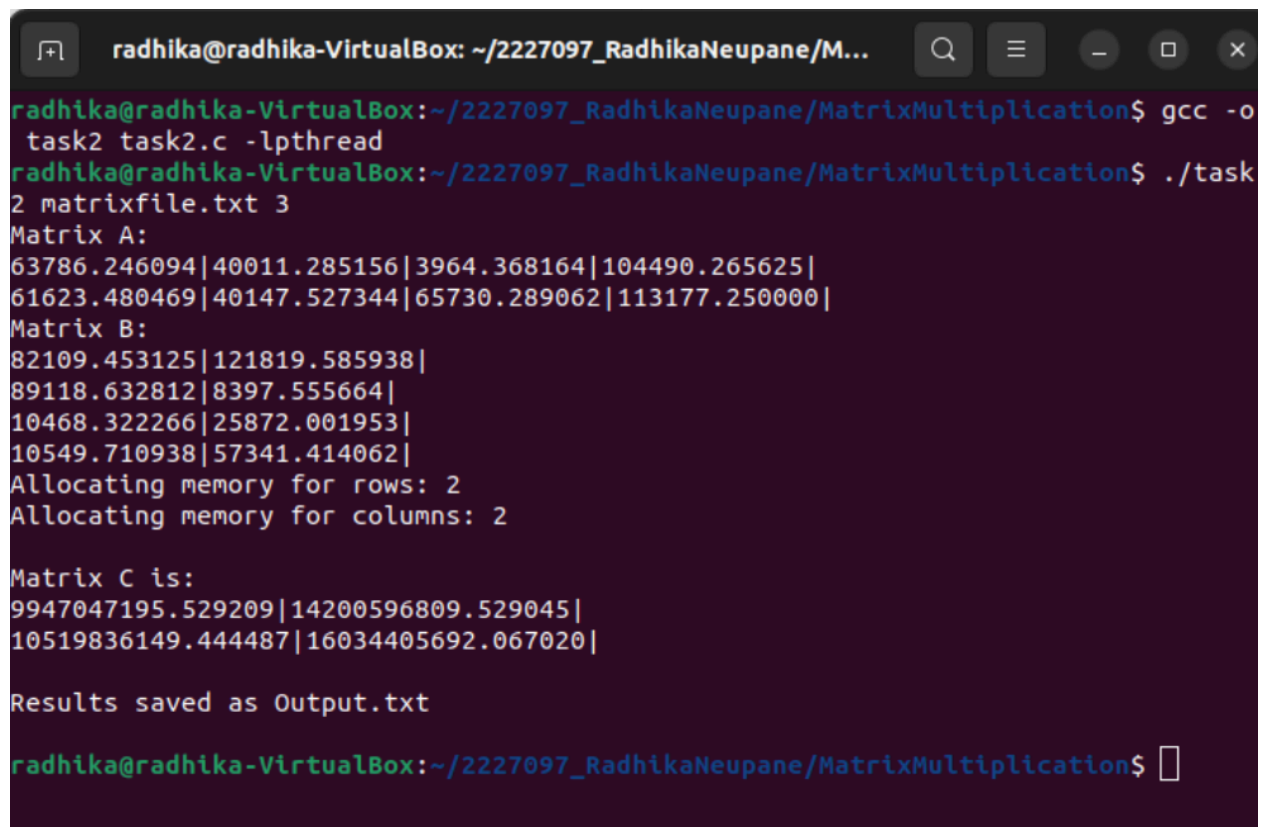
**Storing the correct output matrices in the correct format to a file (20 marks)**

This program performs the matrix multiplication using the multiple threads. It reads the matrices from the file and checks the condition and see's if multiplication is possible, then those works are distributed among the threads and at last the result is printed and finally the result is saved. As in the above program, the necessary header files were included for standard input/output, string manipulation, memory allocation and multi-threading. Two structures 'Matrix' and 'ThreadInfo' are declared which represent the matrix with the information about number of rows, columns and 2D array for storing value and information about a thread. The global variables are made to store the input and result matrices and store the information about each thread. The various functions are declared for reading the matrices from a file, printing the matrices, multiplying matrices, saving matrices, checking if those matrices can be multiplied or not along with others. A function 'allocateMatrix' is also created where memory is allocated for the matrix on the basis of its rows and columns number. 'readMatricesFromFile' is used to read out the matrices from the provided files and assign those values respectively to the



corresponding matrix. The function 'readFile' and 'getSize' are used in order to read the matrices from a file. allocateMatrix. For the condition where two matrices are checked either it can be multiplied or not 'canMultiply' is simply used and 'printMatrix' is used to print the values of a matrix. Coming towards the driver code it reads the matrices from A and B and displays those matrices. Then it checks whether it can be multiplied or not. The memory is allocated for the result matrix along with the thread information. At last the works are distributed among the threads and the multiplication is done. The result is displayed and saved after which allocated memory is set free.

### In Terminal:

A terminal window titled 'radhika@radhika-VirtualBox: ~/2227097\_RadhikaNeupane/M...' shows the execution of a C program. The user runs 'gcc -o task2 task2.c -lpthread' and then './task2 matrixfile.txt 3'. The program outputs 'Matrix A:' followed by two rows of floating-point numbers. Then it outputs 'Matrix B:' followed by two rows of floating-point numbers. It then says 'Allocating memory for rows: 2' and 'Allocating memory for columns: 2'. Next, it outputs 'Matrix C is:' followed by two rows of floating-point numbers. Finally, it says 'Results saved as Output.txt' and returns to the prompt.

```
radhika@radhika-VirtualBox: ~/2227097_RadhikaNeupane/M...  
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$ gcc -o  
task2 task2.c -lpthread  
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$ ./task  
2 matrixfile.txt 3  
Matrix A:  
63786.246094|40011.285156|3964.368164|104490.265625|  
61623.480469|40147.527344|65730.289062|113177.250000|  
Matrix B:  
82109.453125|121819.585938|  
89118.632812|8397.555664|  
10468.322266|25872.001953|  
10549.710938|57341.414062|  
Allocating memory for rows: 2  
Allocating memory for columns: 2  
  
Matrix C is:  
9947047195.529209|14200596809.529045|  
10519836149.444487|16034405692.067020|  
  
Results saved as Output.txt  
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$
```

**Input File :**

The screenshot shows a text editor window titled 'matrixfile.txt' with the path '~/2227097\_RadhikaNeupane/MatrixMultiplication'. The editor contains the following text:

```

1 2,4
2 63786.246094,40011.285156,3964.368164,104490.265625
3 61623.480469,40147.527344,65730.289062,113177.250000
4
5 4,2
6 82109.453125,121819.585938
7 89118.632812,8397.555664
8 10468.322266,25872.001953
9 10549.710938,57341.414062

```

**Output File :**

The screenshot shows a text editor window titled 'Output.txt' with the path '~/2227097\_RadhikaNeupane/MatrixMultiplication'. The editor contains the following text:

```

1 9947047195.529209,14200596809.529045
2 10519836149.444487,16034405692.067020

```

**Checking if Matrix A 3x3 and Matrix B 2x2 can be multiplied or not:**

**Input File:**

The screenshot shows a text editor window titled 'matrixfile.txt' with the path '~/2227097\_RadhikaNeupane/MatrixMultiplication'. The editor contains the following text:

```

1 3,3
2 82109.453125,121819.585938,3964.368164
3 89118.632812,8397.555664,104490.265625
4 10549.710938,57341.414062,113177.250000
5
6 2,2
7 63786.246094,40011.285156
8 61623.480469,40147.527344

```

**In Terminal:**

```

radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$ gcc -o task2 task2.c -lpthread
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$ ./task2 matrixfile.txt 3
Matrix A:
82109.453125|121819.585938|3964.368164|
89118.632812|8397.555664|104490.265625|
10549.710938|57341.414062|113177.250000|
Matrix B:
63786.246094|40011.285156|
61623.480469|40147.527344|
Matrices cannot be multiplied.
radhika@radhika-VirtualBox:~/2227097_RadhikaNeupane/MatrixMultiplication$ █

```

**3. Password Cracking using CUDA**

Using a similar concept as question 2, you will now crack passwords using CUDA. As a kernel function cannot use the crypt library, you will be given an encryption function instead which will generate a password for you. Your program will take in an encrypted password and decrypt it using many threads on the GPU. CUDA allows multidimensional thread configurations so your kernel function (which runs on the GPU) will need to be modified according to how you call your function.

**Generate encrypted password in the kernel function (using CudaCrypt function) to be compared to original encrypted password (25 marks)**

**Allocating the correct amount of memory on the GPU based on input data. Memory is freed once used (15 marks)**

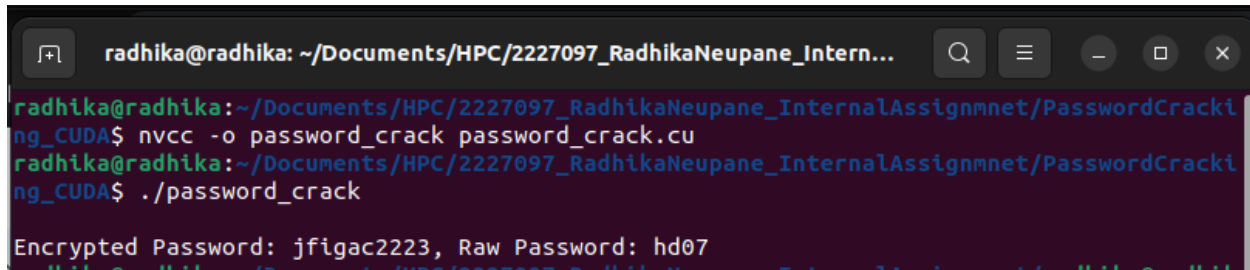
**Program works with multiple blocks and threads – the number of blocks and threads will depend on your kernel function. You will not be penalized if your program only works with a set number of blocks and threads however, your program must use more than one block (axis is up to you) and more than one thread (axis is up to you) (40 marks)**

**Decrypted password sent back to the CPU and printed (20 marks)**

This program is a CUDA program which cracks the encrypted password using the parallel processing on GPU. In this program an encryption function called 'CodeCrypt' is used which generates the password on the basis of the password which has been input by the users. It also has the CUDA kernel function to find out the combinations of letters

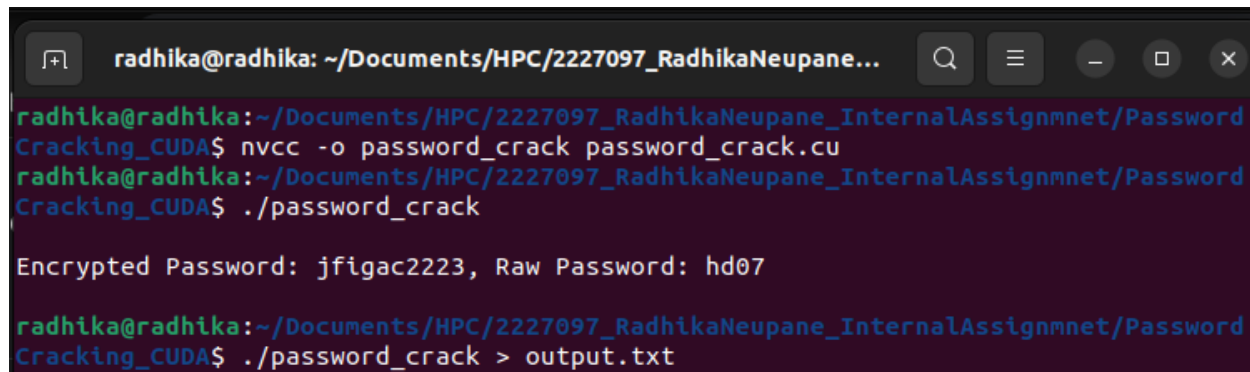
and numbers for the password. The necessary header file has been at first, then a device function called 'copy\_strings' is made which basically copies the strings from the source to the destination. A 'do-while' loop is implemented which copies the character from the source to destination until a null character comes across. The device function called 'compare\_strings' is made to compare two strings. Three variables i.e., 'match', 'i' and 'done' are initialized to zero(0). A while loop is initialized which keeps running as long as three conditions are true. The first condition is that the current index (i) should be less than the specified length (len). The second condition is that there shouldn't be mismatches between the characters in the two strings. The third condition is that neither of the strings reaches the null terminator. Inside the loop it checks if the string has reached the null terminator. This function simply returns the result of comparison. Again a device function named 'CudaCrypt' is made which performs the encryption based on the given specific logic. In this function 'newPassword' is dynamically allocated on a device with a size of 11 characters and this array stores the encrypted password. A specific arithmetic operation is performed which consists of the logic for encryption of password. A loop is initialized which iterates through the character of 'newPassword' and logic to adjust characters which stays within the valid ranges provided in ASCII. Next up, a global function is made, this CUDA kernel is named 'crack' which cracks the encrypted password using CUDA. In this function the kernel generates raw passwords on the basis of block and thread indices. It also checks if the encrypted password is matched to the raw password. Finally a driver code is made which initializes data, transfers those data to the GPU, launches the CUDA kernel to perform password cracking using parallelism and prints the results after making the allocated memory free. I have initialized the arrays for the lowercase alphabet, numbers and encrypted password where Pointers i.e., "gpuAlphabet, gpuNumbers, gpuPassword" are responsible for allocating the memory on GPU. 'CudaMalloc' allocates the memory on the GPU to store the 26 characters and 'sizeof(char) \*26' calculates the total size of memory needed for these characters. The CUDA Kernel is launched with 2D grid, 26\*26 blocks and a 2D blocks, 10\*10 threads per block. 'cudaMemcpy' function copies the data from GPU to CPU and both final results are printed. At the end, both the memory allocated in CPU and GPU are freed.

In terminal:



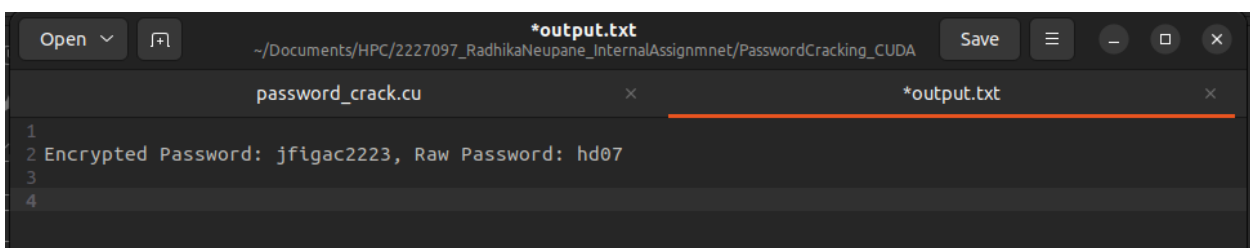
```
radhika@radhika: ~/Documents/HPC/2227097_RadhikaNeupane_Intern...  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/PasswordCracki  
ng_CUDA$ nvcc -o password_crack password_crack.cu  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/PasswordCracki  
ng_CUDA$ ./password_crack  
  
Encrypted Password: jfigac2223, Raw Password: hd07
```

Saving the output in text file :



```
radhika@radhika: ~/Documents/HPC/2227097_RadhikaNeupane...  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/Password  
Cracking_CUDA$ nvcc -o password_crack password_crack.cu  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/Password  
Cracking_CUDA$ ./password_crack  
  
Encrypted Password: jfigac2223, Raw Password: hd07  
  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/Password  
Cracking_CUDA$ ./password_crack > output.txt
```

Result of output.txt File :



```
*output.txt  
~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/PasswordCracking_CUDA  
password_crack.cu × *output.txt ×  
1  
2 Encrypted Password: jfigac2223, Raw Password: hd07  
3  
4
```

## 4. Box Blur using CUDA

Your program will decode a PNG file into an array and apply the box blur filter. Blurring an image reduces noise by taking the average RGB values around a specific pixel and setting its RGB to the mean values you've just calculated. This smoothens the colour across a matrix of pixels. For this assessment, you will use a 3x3 matrix. For example, if you have a 5x5 image such as the following (be aware that the coordinate values will depend on how you format your 2D array):

0,4	1,4	2,4	3,4	4,4
0,3	1,3	2,3	3,3	4,3
0,2	1,2	2,2	3,2	4,2
0,1	1,1	2,1	3,1	4,1
0,0	1,0	2,0	3,0	4,0

The shaded region above represents the pixel we want to blur, in this case, we are focusing on pixel 1,2 (x,y) (the center of the matrix). To apply the blur for this pixel, you would sum all the Red values from the surrounding coordinates including 1,2 (total of 9 R values) and find the average (divide by 9). This is now the new Red value for coordinate 1,2. You must then repeat this for Green and Blue values. This must be repeated throughout the image. If you are working on a pixel which is not fully surrounded by pixels (8 pixels), you must take the average of however many neighboring pixels there are.

Your task is to use CUDA to blur an image. Your number of blocks and threads should in an ideal scenario reflect the dimension of the image however, there are limits to the amount of blocks and threads you can spawn in each dimension (regarding block and thread dimensions (x,y,z). You will not be penalized if you do not use different dimensions of blocks and threads, for this assessment, we will accept just one dimensional blocks and threads, e.g. function<<<blockNumber, threadNumber>>>

**Reading in an image file into a single or 2D array (5 marks)**

**Allocating the correct amount of memory on the GPU based on input data. Memory is freed once used (15 marks)**

**Applying Box filter on image in the kernel function (30 marks)**

**Return blurred image data from the GPU to the CPU (30 marks)**

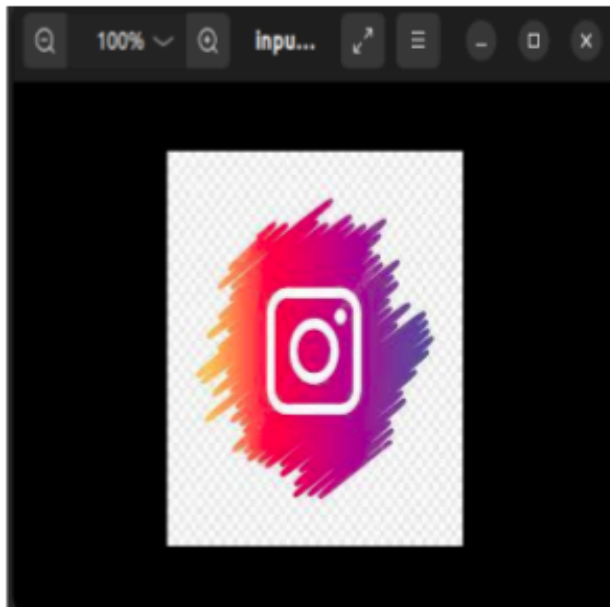
**Outputting the correct image with Box Blur applied as a file (20 marks)**

This program is a CUDA program which performs a box blur on a PNG image using parallel processing with CUDA. I have included all the necessary headers for standard library function and input/output, especially in this program 'lodepng' library is included since it helps with image encoding and decoding. The CUDA kernel function named as 'Imageblur' is launched which is responsible for applying the box blur to the image parallelly. In this function it takes the input array 'gpu\_implInput' and output array 'gpu\_impoutput'. Image width and height along with blur radius and parameters as arguments. The kernel calculates the average RGB values of neighboring pixels within blur radius and updates the pixels in the output image. In this program each thread is responsible for processing the specific pixels of the images. The nested loops iterate over each pixel, the outer loop looks at pixels from left and right i.e., horizontally and inner loops at pixels from up and down i.e., vertically. Now moving towards driver code, a common approach to obtain the blur radius from blur diameter is applied where blur parameters are set and blur is calculated as  $(\text{blurMD} - 1)/2$  and  $\text{blurMD}$  is set to 3. Few variables are declared where the error code returned by the decoding function will be stored in 'error', height and weight of the image will be stored in 'h' and 'w' respectively and 'filename' means the input image file name which is 'input.png'. The function called 'lodepng\_decode32\_file' is called in order to decode the input png file and in case of error, an error message will be printed. Host arrays are also declared in order to store the input and output of the image data. Using 'cudaMalloc' and 'cudaMemcpy' memory is allocated on the GPU and the image input data is copied from the host to the GPU. The kernel is called with the grid of 'w' threads per block and 'h' blocks. At last the blurred image will be copied back to the host from GPU and the function 'lodepng\_encode32\_file' is called to encode and save the blurred image as 'output.png'. In case of any error the error message is printed. The allocated memory is freed and the program ends.

In terminal:

```
radhika@radhika: ~/Documents/HPC/2227097_RadhikaNeupan...  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/BoxBlur_CUDA$ nvcc -o imageBlur imageBlur.cu lodepng.cpp  
radhika@radhika:~/Documents/HPC/2227097_RadhikaNeupane_InternalAssignmnet/BoxBlur_CUDA$ ./imageBlur input.png
```

Input Image:



Output Image:

