6CS005 Learning Journal - Semester 1 2020/2021

Name: Pranish Acharya

UV\_ID: 2038527

# Table of Contents

[Table of Contents 1](#_Toc20306348)

[1 Parallel and Distributed Systems 2](#_Toc20306349)

[1.1 Answer of First Question 2](#_Toc20306350)

[1.2 Answer of Second Question 2](#_Toc20306351)

[1.3 Answer of Third Question 2](#_Toc20306352)

[1.4 Answer of Fourth Question 2](#_Toc20306353)

[1.5 Answer of Fifth Question 2](#_Toc20306354)

[1.6 Answer of Sixth Question 2](#_Toc20306355)

[2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system 2](#_Toc20306356)

[2.1 Single Thread Matrix Multiplication 2](#_Toc20306357)

[2.2 Multithreaded Matrix Multiplication 3](#_Toc20306358)

[2.3 Password cracking using POSIX Threads 3](#_Toc20306359)

[3 Applications of Password Cracking and Image Blurring using HPC-based CUDA System 3](#_Toc20306360)

[3.1 Password Cracking using CUDA 3](#_Toc20306361)

[3.2 Image blur using multi dimension Gaussian matrices 3](#_Toc20306362)

# Parallel and Distributed Systems

## Answer of First Question

Threads are an independent instruction stream that can be scheduled by the OS to run. Threads allow applications to simultaneously perform multiple tasks. Threads allow multiple computations to be performed in parallel.

## Answer of Second Question

The two process scheduling policies are :

1. Pre-emptive: The schedular is in charge of how long a process runs for. If a process exceeds its time slice, it is stopped by the schedular.
2. Co-operative: Each process is in-charge of how long it runs for. When a process feels like co-operating, it will surrender execution.

As every process gets its equal execution time, pre-emptive is preferable. The execution time period of the C threads which change the choice of policies..

## Answer of Third Question

|  |  |
| --- | --- |
| Centralized Systems | Distributed Systems |
| 1. Centralized Systems uses the architecture of the client-server where one or more client nodes are connected directly to the central server. | 1. Distributed networks use a peer-to-peer architecture in which each node makes its own choice. The system's final action is the sum of all nodes' decisions. |
| 1. The system is composed of one component with non-autonomous components. | 1. This system is made up of various autonomous components. |
| 1. All the services are available in this system. | 1. Resources may not be available in this system. |
| 1. All the time, components are shared by users. | 1. There is no sharing of components among all users. |

## Answer of Fourth Question

Transparency in Distributed System is defined to hide something. In order to realize the single system image, transparency is an essential issue that makes systems as simple to use as a single processor system.

Classification of the Transparency:

1. Access Transparency: Data and resources can be used in a consistent way.
2. Location Transparency: A user cannot tell where resources are located.
3. Migration Transparency: Resources can move at will without changing their names.
4. Concurrency Transparency: Multiple users can share resource automatically.
5. Failure Transparency: A user does not notice resource failure.
6. Performance Transparency: Systems are reconfigured to improve performance as loads vary.
7. Scaling Transparency: Systems can expand in size without changing the system structure and the application programs.

## Answer of Fifth Question

Include your code using a text file in the submitted zipped file under name Task1.5

## Answer of Sixth Question

The output of the given programs are 100000 and 500000 respectively. In these two programs pthread\_join() function is call which terminate thread after executing target thread but in first program thread is call before reaching to the target thread. So, the result is different although the two programs are almost same.

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

## Single Thread Matrix Multiplication

* The analysis of the algorithm’s complexity. (1 mark)

The time complexity of the given program 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)

Ans.

There are different approaches present to speed up of the given program. Some of them are:

1. Using multithreading.

In multi-threading, instead of using a single processor core, all or more core is used to solve the problem. The problem is divided into different blocks.

1. Using cache memory

By utilizing the cache memory, the program took reference from previous executed program which may have stored in cache memory.

1. Strassen algorithm

Strassen algorithm can be used to improve the speed of above and its time complexity is O(n^(2.8074)).

* 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)

Paste your algorithm’s pseudo code here

* 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.

## Multithreaded Matrix Multiplication

Multithread Matrix Multiplication

#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#define dim 1024

int thread\_count;

struct param {

int start\_line, end\_line;

};

int a[dim][dim], b[dim][dim], c[dim][dim];

void init\_matrix(int m[dim][dim]) {

int i, j;

for (i = 0; i < dim; i++)

for (j = 0; j < dim; j++)

m[i][j] = rand() % 5;

}

void\* thread\_function(void \*v) {

struct param \*p = (struct param \*) v;

int i, j, k;

for (i = p->start\_line; i < p->end\_line; i++)

for (j = 0; j < dim; j++) {

int s = 0;

for (k = 0; k < dim; k++)

s += a[i][k] \* b[k][j];

c[i][j] = s;

}

}

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() {

struct timespec start, finish;

long long int time\_elapsed;

int i;

int start\_time, end\_time;

printf("Enter the number of thread: ");

scanf("%d",&thread\_count);

struct param params[thread\_count];

pthread\_t t[thread\_count];

clock\_gettime(CLOCK\_MONOTONIC, &start);

init\_matrix(a);

init\_matrix(b);

for (i = 0; i < thread\_count; i++) {

int code;

params[i].start\_line = i \* (dim / thread\_count);

params[i].end\_line = (i + 1) \* (dim / thread\_count);

code = pthread\_create(&t[i], NULL, thread\_function, &params[i]);

if (code != 0)

printf("Error starting thread %d\n", i);

}

for (i = 0; i < thread\_count; i++)

pthread\_join(t[i], NULL);

printf("\nThe results is…\n");

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

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

printf("%d ", c[i][j]);

}

printf("\n");

}

clock\_gettime(CLOCK\_MONOTONIC, &finish);

time\_difference(&start, &finish, &time\_elapsed);

printf("Time taken was %lldns or %0.9lfs\n", time\_elapsed,

(time\_elapsed/1.0e9));

}

* 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.

Table

Description automatically generated

A picture containing chart

Description automatically generated

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

In the above program the time was decreasing by increasing the number of threads. At certain point matrix take same time to

## 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

Text

Description automatically generated  
Text

Description automatically generated

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

Graphical user interface, application, table, Excel

Description automatically generated

* 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.

Original password cracking program took 164.42 seconds (2.74 minutes) mean runtime, if we add 1 more initial i.e. one more for-loop starting from A-Z in program, it will take 4274.92 seconds (71.25 minutes) to run and create all three possible passwords of three alphabets and two numbers for a whole program. This is because it would take an additional 26 more time than the original program due to extra for loop. Therefore, the cost of time would rise by 26times.

Password Crack Using three Initial in Original Program

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <crypt.h>

#include <unistd.h>

#include <time.h>

int counter=0;

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, r, a, n;

char salt[7];

char plain[7];

char \*enc;

substr(salt, salt\_and\_encrypted, 0, 6);

for(p='A'; p<='Z'; p++){

for(r='A'; r<='Z'; r++){

for(a='A'; a<='Z'; a++){

for(n=0; n<=99; n++){

sprintf(plain, "%c%c%c%02d",p, r, a, n);

enc = (char \*) crypt(plain, salt);

counter++;

if(strcmp(salt\_and\_encrypted, enc) == 0){

printf("#%-8d%s %s\n", counter, plain, enc);

}else{

printf("%-8d%s %s\n", counter, plain, enc);

}

}

}

}

}

}

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);

crack("$6$AS$1mYI1AoWnumQr1TU5VoPouGGT0IGe4jKnKYY6SS7o1pbKWAJz.19AXQqRQHgH9Hwp3Zgy.MsRuZj/bHdvcNS41");

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

clock\_gettime(CLOCK\_MONOTONIC, &finish);

time\_difference(&start, &finish, &time\_elapsed);

printf("Time taken was %lldns or %0.9lfs\n", time\_elapsed,

(time\_elapsed/1.0e9));

return 0;

}

**Code Description:**

The program above has a code with 3 alphabets and 2 numbers. In the original program, one additional for-loop was introduced to render the third for-loop for the alphabet, starting from A-Z. In the four encrypted passwords issued, the first six digits are referred to as salt. $6$KB$ is the salt displayed in the encrypted password. The variable ‘’i was declared in the main function with the data type int, in order to measure the program's elapsed time, the start time and the end time were declared.

After executing clock\_gettime (CLOCK\_MONOTONIC, &start), it will note the start-time from where it started to crack the password. For-loop has been used that will run from 0 to n<passwords, which is 0 to 3. In crack function, nested for-loop has been used in which first three for-loop will calculate alphabet and last for-loop will calculate numbers. At last clock\_gettime (CLOCK\_MANOTONIC, &finish) will be called to note a time when program has stopped.

**time\_difference ()** function has been called to calculate the total time taken by the whole program to generate all possible passwords.

**Crypt ()** helps to convert salt and plan passwords into encrypted password.

**Substr ()** extracts salt from encrypted passwords.

**Strcmp ()** has been used to compare whether the encrypted password and password generated by the combination matches or not. If the password matches, # will be used at the front of the password to identify easily.

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

After executing three initials, the program took 4247.02 seconds (70.78 minutes). Comparing the exact time after the code with above hypothesises, the exact time the program took was 4247.023895071 seconds where estimated time was 4274.92 seconds. As we can see that the difference between two is just 27.894879026 seconds. This difference may have arises due multiple process running in background.

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

Password Crack Using multithread

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <crypt.h>

#include <time.h>

#include <pthread.h>

int n\_passwords = 1;

int counter = 0;

char \*encrypted\_password[]={"$6$AS$5MUX6oD4KOSmVBi5yCmu.8FTv1EgYaX09sO49BVt7CutV9Y1KWJVmjqSHoeub/AnfQU4eEV6NeIaDUf.OG7XK/" };

void substr(char \*dest, char \*src, int start, int length){

memcpy(dest, src + start, length);

\*(dest + length) = '\0';

}

void runThread(){

int i;

pthread\_t PA1, PA2;

void \*kernel\_function\_1();

void \*kernel\_function\_2();

for(i=0; i<n\_passwords;i<i++){

pthread\_create(&PA1, NULL, kernel\_function\_1, encrypted\_password[0]);

pthread\_create(&PA2, NULL, kernel\_function\_2, encrypted\_password[0]);

pthread\_join(PA1, NULL);

pthread\_join(PA2, NULL);

}

}

void \*kernel\_function\_1(char \*salt\_and\_encrypted){

int p, r, a;

char salt[7];

char plain[7];

char \*enc;

int counter=0;

substr(salt, salt\_and\_encrypted, 0, 6);

for(p='A'; p<='M' ; p++){

for(r='A'; r<='Z'; r++){

for(a=0; a<=99; a++){

sprintf(plain, "%c%c%02d", p, r, a);

enc = (char \*) crypt(plain, salt);

counter++;

if(strcmp(salt\_and\_encrypted, enc) == 0){

printf("#%-8d%s %s\n", counter, plain, enc);

}

else{

printf("#%-8d%s %s\n", counter, plain, enc);

}

}

}

}

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

}

void \*kernel\_function\_2(char \*salt\_and\_encrypted){

int p, r, a;

char salt[7];

char plain[7];

char \*enc;

int counter=0;

substr(salt, salt\_and\_encrypted, 0, 6);

for(p='N'; p<='Z' ; p++){

for(r='A'; r<='Z'; r++){

for(a=0; a<=99; a++){

sprintf(plain, "%c%c%02d", p, r, a);

enc = (char \*) crypt(plain, salt);

counter++;

if(strcmp(salt\_and\_encrypted, enc) == 0){

printf("#%-8d%s %s\n", counter, plain, enc);

}

else{

printf("#%-8d%s %s\n", counter, plain, enc);

}

}

}

}

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

}

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);

runThread();

clock\_gettime(CLOCK\_MONOTONIC, &finish);

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

time\_difference(&start, &finish, &time\_elapsed);

printf("Time elapsed was %lldns or %0.9lfs\n", time\_elapsed,(time\_elapsed/1.0e9));

return 0;

}

**Code Description**: #include <pthread.h> must be included every time, in order to execute a program in posix multithread. In main function, int i, start and finish and time\_elapsed has been done. After that runThread () function has been called. Inside runThread (), two thread PA1, PA2 and two kernel functions kernel\_function\_1 () and kernel\_function\_2() has been declared. pthread\_create () helps to create a new thread and makes it executable. It has 4 parameters.

|  |  |  |
| --- | --- | --- |
| First Parameter | &PA1/ &PA2 | Identifier for new thread. |
| Second parameter | NULL | Thread attribute object /argument |
| Third parameter | Kernel\_function\_1 /2 | start\_routine / function name |
| Fourth parameter | encrypted\_password[ ] | argument that may be passed to start\_routine or function. |

**pthread\_join ()** helps to join the two thread i.e. PA1 and PA2.

**time\_difference ()** function has been called to calculate the total time taken by the whole program to generate all possible passwords.

**Crypt ()** helps to convert salt and plan passwords into encrypted password.

**Substr ()** extracts salt from encrypted passwords.

**Strcmp ()** has been used to compare whether the encrypted password and password generated by the combination matches or not. If the password matches, # will be used at the front of the password to identify easily.

Table, Excel

Description automatically generated

Original password cracking program took 164.4199528499 seconds (2.74033254749833 minutes) mean running time to run while it took 129.8691925625 seconds (2.16448654270833 minutes) mean running time to run after inserting POSIX multithread in the same code. Using POSIX multithread, we can see that the cost of time from its initial program has been reduced, which is more effective. This is because it runs asynchronously as a thread.

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

## Password Cracking using CUDA

* Include your code using a text file in the submitted zipped file under name Task3.1

Password Crack Using CUDA version.

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <crypt.h>

#include <time.h>

#include <math.h>

#include <malloc.h>

#include <cuda\_runtime\_api.h>

#include <cuda\_runtime.h>

#include <cuda.h>

\_\_device\_\_ int is\_a\_match(char \*attempt){

char normal\_password1[]="HP21";

char \*a=attempt;

char \*np1= normal\_password1;

while (\*a==\*np1){

if(\*a=='\0'){

printf("password: %s\n", normal\_password1);

return 1;

}

a++;

np1++;

}

return 0;

}

\_global\_\_ void kernel(){

int p=0, q=0 ;

char password[4];

password[3]='\0';

int i= blockIdx.x+65;

int j= threadIdx.x+65;

char firstmatch=i;

char secondmatch=j;

password[0]= firstmatch;

password[1]= secondmatch;

for(p='0'; p<='9';p++){

for(q='0'; q<='9';q++){

password[2]=p;

password[3]=q;

if(is\_a\_match(password)){

}

}

}

}

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);

kernel <<<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;

}

**Code description:**

First, #include <cuda\_runtime\_api.h> must be included on top of the code. This will help to develop CPU and GPU connectivity. Inside the kernel function, defining and initializing an array as password [7] has be done. Index 6th of an array will have a value “\0”. Int i and int j will take the value from blockIdx.x+65 and threadIdx.x+65, value A and A respectively. After setting the value of firstMatch and secondMatch to I and j and executing four for loops, is\_a\_match function will be called.

MUTEX has been used by initializing mutex on top of the program.

CODE: Pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

After initializing mutex, it has been used inside main () function by locking and unlocking the mutex. Locking of the mutex has been included before calling the kernel function and unlocking has been done after cudaThreadSynchronize () has been called.

CODE:

pthread\_mutex\_lock(&mutex);

kernel <<<26,26>>>();

cudaThreadSynchronize();

pthread\_mutex\_unlock(&mutex);

In is\_a\_match function (), four passwords, which we have set with 2 alphabets and 2 numeric will be stored in plain\_password1/. Passwords generated by kernel function will get stored in attempts and while loop will compare each password and if the password has ‘\0’ at its last index, it will indicate that the password has matched and printf statement will be called and break statement will be executed and while loop will get terminated.

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

Screen Shot of Password Cracking Using Original Code of 2 Initials

Graphical user interface, application, table, Excel

Description automatically generated  
Screen shot of Password Cracking using Thread

Table, Excel

Description automatically generated

Screen shot of password Cracking Using CUDA version.  
  
A picture containing table

Description automatically generated

* Write a short analysis of the results

The mean time of the program using CUDA version is 0.0042635307 second which is very less than the original one which is 164.4199528499 seconds (2.7 minutes) and thread version which is129.8691925625 second (2.1 minutes). It is because the CUDA in C programming is a more efficient package that interacts with GPC to execute the code. Compared to the Processor and GPU core, the CPU core is 42 times faster than the GPU core. Although the GPU core is slow, the task was performed simultaneously and much faster using the GPU core in the password cracker. Large numbers of cores will be split into 16 multicore units that will be able to perform eight operations in a single cycle. For this reason, if any multicore task is performed, 8 cores will run over 4 clock cycles every time. For this purpose, the GPU performs the tasks in a cost-effective manner than the CPU. CUDA is known to be a very strong core material and has the better potential to find productive results than others, such as POSIX. The CUDA processing speed is high, so that the performance is quickly taken out.

## 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 <stdlib.h>

#include <cuda\_runtime\_api.h>

#include <time.h>

// nvcc -o CudaImageBlur 2038527\_Task3\_B.cu lodepng.cpp

\_\_device\_\_ unsigned int deviceWidth;

\_\_device\_\_ unsigned char getRed(unsigned char \*image, unsigned int row, unsigned int col)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4);

return image[i];

}

\_\_device\_\_ unsigned char getGreen(unsigned char \*image, unsigned int row, unsigned int col)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 1;

return image[i];

}

\_\_device\_\_ unsigned char getBlue(unsigned char \*image, unsigned int row, unsigned int col)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 2;

return image[i];

}

\_\_device\_\_ unsigned char getAlpha(unsigned char \*image, unsigned int row, unsigned int col)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 3;

return image[i];

}

\_\_device\_\_ void setRed(unsigned char \*image, unsigned int row, unsigned int col, unsigned char red)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4);

image[i] = red;

}

\_\_device\_\_ void setGreen(unsigned char \*image, unsigned int row, unsigned int col, unsigned char green)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 1;

image[i] = green;

}

\_\_device\_\_ void setBlue(unsigned char \*image, unsigned int row, unsigned int col, unsigned char blue)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 2;

image[i] = blue;

}

\_\_device\_\_ void setAlpha(unsigned char \*image, unsigned int row, unsigned int col, unsigned char alpha)

{

unsigned int i = (row \* deviceWidth \* 4) + (col \* 4) + 3;

image[i] = alpha;

}

\_\_global\_\_ void changeImage(unsigned char\* image, unsigned char\* newImage, unsigned int \*width){

int row = blockIdx.x+1;

int col = threadIdx.x+1;

deviceWidth = \*width;

unsigned redTL, redTC, redTR;

unsigned redL, redC, redR;

unsigned redBL, redBC, redBR;

unsigned newRed;

unsigned greenTL, greenTC, greenTR;

unsigned greenL, greenC, greenR;

unsigned greenBL, greenBC, greenBR;

unsigned newGreen;

unsigned blueTL, blueTC, blueTR;

unsigned blueL, blueC, blueR;

unsigned blueBL, blueBC, blueBR;

unsigned newBlue;

setGreen(newImage, row, col, getGreen(image, row, col));

setBlue(newImage, row, col, getBlue(image, row, col));

setAlpha(newImage, row, col, 255);

redTL = getRed(image, row - 1, col - 1);

redTC = getRed(image, row - 1, col);

redTR = getRed(image, row - 1, col + 1);

redL = getRed(image, row, col - 1);

redC = getRed(image, row, col);

redR = getRed(image, row, col + 1);

redBL = getRed(image, row + 1, col - 1);

redBC = getRed(image, row + 1, col);

redBR = getRed(image, row + 1, col + 1);

newRed = (redTL+redTC+redTR+redL+redC+redR+redBL+redBC+redBR)/9;

setRed(newImage, row, col, newRed);

greenTL = getGreen(image, row - 1, col - 1);

greenTC = getGreen(image, row - 1, col);

greenTR = getGreen(image, row - 1, col + 1);

greenL = getGreen(image, row, col - 1);

greenC = getGreen(image, row, col);

greenR = getGreen(image, row, col + 1);

greenBL = getGreen(image, row + 1, col - 1);

greenBC = getGreen(image, row + 1, col);

greenBR = getGreen(image, row + 1, col + 1);

newGreen = (greenTL+greenTC+greenTR+greenL+greenC+greenR+greenBL+greenBC+greenBR)/9;

setGreen(newImage, row, col, newGreen);

blueTL = getBlue(image, row - 1, col - 1);

blueTC = getBlue(image, row - 1, col);

blueTR = getBlue(image, row - 1, col + 1);

blueL = getBlue(image, row, col - 1);

blueC = getBlue(image, row, col);

blueR = getBlue(image, row, col + 1);

blueBL = getBlue(image, row + 1, col - 1);

blueBC = getBlue(image, row + 1, col);

blueBR = getBlue(image, row + 1, col + 1);

newBlue = (blueTL+blueTC+blueTR+blueL+blueC+blueR+blueBL+blueBC+blueBR)/9;

setBlue(newImage, row, col, newBlue);

}

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 char \*image;

const char \*filename = argv[1];

const char \*newFileName = "filtered.png";

unsigned char \*newImage;

unsigned int height = 0, width = 0;

lodepng\_decode32\_file(&image, &width, &height, filename);

newImage = (unsigned char \*)malloc(height \* width \* 4 \* sizeof(unsigned char));

unsigned char \* gpuImage;

cudaMalloc( (void\*\*) &gpuImage, sizeof(char) \* height\*width\*4);

cudaMemcpy(gpuImage, image, sizeof(char) \* height\*width\*4, cudaMemcpyHostToDevice);

unsigned char \* gpuNewImage;

cudaMalloc( (void\*\*) &gpuNewImage, sizeof(char) \* height\*width\*4);

unsigned int\* gpuWidth;

cudaMalloc( (void\*\*) &gpuWidth, sizeof(int));

cudaMemcpy(gpuWidth, &width, sizeof(int), cudaMemcpyHostToDevice);

printf("Image width = %d height = %d\n", width, height);

changeImage<<<height-1,width-1>>>(gpuImage, gpuNewImage, gpuWidth);

cudaThreadSynchronize();

cudaMemcpy(newImage, gpuNewImage, sizeof(char) \* height \* width \* 4, cudaMemcpyDeviceToHost);

lodepng\_encode32\_file(newFileName, newImage, width, height);

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.

Graphical user interface, application, website

Description automatically generated

Graphical user interface, application

Description automatically generated

* Write a short analysis of the results

Gaussian blurring process has been implemented in the logo of Herald College Kathmandu, in CUDA CORE GPU’s. The same blurring algorithm was implemented and run on CPU in order to compute the speed up based on execution times.

Following steps are used for calculating Gaussian blur

Step 1: Separating RGBA image to red, green and blue channels.

Step 2: Applying Gaussian blurring method for each time.

Step 3: Taking red, green and blue values into RGBA again

The method of Gaussian blurring can be done using Architecture of CUDA and support of multi-threads. The findings suggest that high performance processing is Usually obtained by costly hardware, it is possible to achieve by Using CUDA parallel shared-memory applications