UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

High Performance Computing
(6CS005)
Portfolio Report

Student Id : [2040368]

Student Name : [Akash Chanara]

Cohort/Batch : 4

Submitted on :26/12/2020

Table of Contents

Parallel and Distributed Systems

1. ?
   A thread is part of the process, working within its own execution space, and there can be multiple threads in one process. OS can perform several tasks in parallel with the aid of it (depending upon the number of processors the machine consists). Threads allow the CPU to perform several tasks at the same time in one operation.
2. ?
   Two policies for phase scheduling are: Pre-emptive: This scheduler tracks how long a device runs. The planner interrupts the procedure if a process crosses its time slice. Co-operative: The OS never interrupts each operation and the processes are responsible for how long they operate. If a method sounds like teamwork, implementation will be abandoned. Pre-emptive is preferred; the thread scheduling algorithm of the Java runtime framework is also pre-emptive.
3. ?
   Both measurements are performed on one computer in a consolidated scheme (system). The measurement is transmitted to several machines within the distributed machine. For example: If there is a huge volume of information, it can be separated and sent to each computer component to execute it.
4. ?
   Transparency of the distributed system ensures that, by concealing distribution from the customer and application programmer, one distributed system behaves like a single computer.

5. The following three statements contain a flow dependency, an anti-dependency 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.
   B=A+C
   B=C+D
   C=B+D
   B=A+C is a flow dependency
   C=B+D is an anti-dependency
   B=C+D is an output dependency

6. ?

   Program 1 answer: 349151
   Program 2 answer: 500000
   After the thread is created, program 1 runs an extra loop after passing the unsigned value to mark the number of threads or counter.

Matrix Multiplication and Password Cracking

Single Thread Matrix Multiplication

- The complexity of above program is $O(n^3)$.
- Divide and Conquer could also speed the multiplication process but the better option would be Strassen's matrix multiplication.
- Since the above program does 8 multiplications for matrices of size N/2 * N/2 and 4 additions, Strassen's multiplication performs it in 7 multiplications, so it is faster.

```
If n = threshold then compute
        C = a * b is a conventional matrix.
    Else
        Partition a into four sub matrices  a11, a12, a21, a22.
        Partition b into four sub matrices b11, b12, b21, b22.
        Strassen ( n/2, a11 + a22, b11 + b22, d1)
        Strassen ( n/2, a21 + a22, b11, d2)
        Strassen ( n/2, a11, b12 - b22, d3)
        Strassen ( n/2, a22, b21 - b11, d4)
        Strassen ( n/2, a11 + a12, b22, d5)
        Strassen (n/2, a21 - a11, b11 + b22, d6)
        Strassen (n/2, a12 - a22, b21 + b22, d7)

        C = d1+d4-d5+d7     d3+d5
        d2+d4           d1+d3-d2-d6

    end if

    return (C)
end.
```

- The output is:

```
Enter the 4 elements of first matrix: 5 4 8 6
Enter the 4 elements of second matrix: 8 5 4 5

The first matrix is

5       4
8       6
The second matrix is

8       5
4       5
After multiplication using

56      45
88      70
Process returned 0 (0x0)   execution time : 10.063 s
Press any key to continue.
```

Matrix Multiplication using Multithreading

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

#define MAT_SIZE 10
#define MAX_THREADS 100

/********
 Compile with   cc -o task2b 2040368_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]);

        //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;
```

Output:

```
--- Matrix 1 ---

   5    6    8

   1    3    0

   8    5    3

--- Matrix 2 ---

   8    5    6

   2    0    5

   6    7    9

--- Multiplied Matrix ---

 100   81  132

  14    5   21

  92   61  100

---> Time Elapsed : 3.00 Sec

---> Used Threads : 9

- Thread 1 ID : -633411840
- Thread 2 ID : -641804544
- Thread 3 ID : -650197248
- Thread 4 ID : -658589952
- Thread 5 ID : -666982656
- Thread 6 ID : -675375360
- Thread 7 ID : -683768064
- Thread 8 ID : -692160768
- Thread 9 ID : -700553472
com133@herald-OptiPlex-3050:~/6cs005 PortfolioS1 19 20 2040368 Akash Chanara$
```

The program uses each thread per element in resultant matrix. So, for a matrix of size 1024*1024, 1048576 threads are used.

Password Cracking using POSIX thread

1. .

Number of program run	Time in nano second	Time in seconds
1	126338942492.00	126.338942492
2	126653558762.00	126.653558762
3	126622886091.00	126.622886091
4	126230990570.00	126.230990570
5	126533256007.00	126.533256007
6	126301290324.00	126.301290324
7	127114770627.00	127.114770627
8	126889819803.00	126.889819803
9	126142905876.00	126.142905876
10	126315538400.00	126.315538400
Average	126514395895.20	126.514395895

2. .

Since two initials are cracked by the above program, three initials can be cracked using the same code but adding just one alphabet loop. Since alphabets consist of 26 characters and the loop passes through those 26 characters, the time estimated can be as follows:

Estimated Time = Original time * 26
= 126.514395895 * 26
= 3,289.37429327seconds

Converting to minutes = 3,289.37429327/ 60
= 54.82290488783333 minutes

Therefore, estimated time is 55 minutes.

3. .

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

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

  Compile with:
    cc -o task2c3 2040368_Task2_C_3.c -lcrypt

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


int n_passwords = 1;

char *encrypted_passwords[] = {
 "$6$AS$hD0mc7Et78KQYsDY6BoKa9kCo05lMD6.ONlTQQRwiikxe1sKfwOuC6BWONedu8x1yVcY4ebo0t9nILoucFMS21"
};


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 a, k, s, h;     // 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(a='A'; a<='Z'; a++){
    for(k='A'; k<='Z'; k++){
     for(s='A'; s<='Z'; s++){
      for(h=0; h<=99; h++){
        sprintf(plain, "%c%c%c%02d", a, k, s, h);
```

```
        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. .

Number of program run	Time in nano second	Time in seconds
1	3288222652622.00	3288.222652622
2	3280521542264.00	3280.521542264
3	3286626232322.00	3286.626232322
4	3282346742455.00	3282.346742455
5	3281556841522.00	3281.556841522
6	3281251521515.00	3281.251521515
7	3283562652352.00	3283.562652352
8	3288426284122.00	3288.426284122
9	3285586246265.00	3285.586246265
10	3289265662662.00	3289.265662662
Average	3284736637810.10	3284.736637810

The actual time is: 3284.736637810 seconds which converted is 54.74561063016667minutes. The estimated time was 54.82290488783333 which is which is a bit more than the estimated time.
This could be due to the background processes running while the two-initial program was running.

5. .

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

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

  Compile with:
    cc -o task2c5 2040368_Task2_C_5.c -lcrypt -lrt -pthread

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


int n_passwords = 1;

char *encrypted_passwords[] = {

"$6$AS$m5exzg8cZG1i2uh.7ohQlRxbutF91rihO63V.kpCF8WkS185/RSz1k/fEgmukhb1XKRyL7hqDR.q1.1GHfj2x0"
};



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 a, k, s;     // 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(a='A'; a<='M'; a++){
    for(k='A'; k<='Z'; k++){
      for(s=0; s<=99; s++){
        sprintf(plain, "%c%c%02d", a,k,s);
        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 a, k, s;     // 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(a='N'; a<='Z'; a++){
    for(k='A'; k<='Z'; k++){
      for(s=0; s<=99; s++){
```

```
        sprintf(plain, "%c%c%02d", a,k,s);
        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;
}
```

```
com133@herald-OptiPlex-3050:~/6cs005_PortfolioS1_19_20_2040368_Akash_Chanara$ ./mr.py ./task2c5 | grep Time
Time elapsed was 63074757786ns or 63.074757786s
Time elapsed was 63025944582ns or 63.025944582s
Time elapsed was 63217754109ns or 63.217754109s
Time elapsed was 63048749002ns or 63.048749002s
Time elapsed was 63070076230ns or 63.070076230s
Time elapsed was 63047567003ns or 63.047567003s
Time elapsed was 63190361978ns or 63.190361978s
Time elapsed was 63403759281ns or 63.403759281s
Time elapsed was 66169287699ns or 66.169287699s
Time elapsed was 69069176498ns or 69.069176498s
com133@herald-OptiPlex-3050:~/6cs005_PortfolioS1_19_20_2040368_Akash_Chanara$
```

6. .

Number of times Program ran	Time taken by original program	Time tke by multithread version
1	126.338942492	63.07447578
2	126.653558762	63.02594458
3	126.622886091	63.21775411
4	126.230990570	63.04287490
5	126.533256007	63.07007623
6	126.301290324	63.04756700
7	127.114770627	63.19036198
8	126.889819803	63.40375928
9	126.142905876	66.16176498
10	126.315538400	69.06917650
Average	126.514395895	64.03037553

Here the single thread version took 126.514395895 seconds while the multithread version took 64.03037553seconds. This is because in multithread, there are two threads while the other has one thread. That is why the multithread is faster than the single thread version.

CUDA

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

Password Cracking using CUDA

```
#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 = ccbnhj3162

  nvcc -o task3a 2040368_Task3_A.cu

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


char *encrypted_passwords[]{
        "ccbnhj3162"
};


__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 * 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;
}
```

Number of times program ran	Time taken by Original program	Time taken by Multithread version	Time taken by CUDA version
1	126.338942492	63.07447578	0.580939718
2	126.653558762	63.02594458	0.595920426
3	126.622886091	63.21775411	0.583720913
4	126.230990570	63.04287490	0.583794837
5	126.533256007	63.07007623	0.582852925
6	126.301290324	63.04756700	0.580954998
7	127.114770627	63.19036198	0.572143096
8	126.889819803	63.40375928	0.581389062
9	126.142905876	66.16176498	0.578276899
10	126.315538400	69.06917650	0.575746898
Average	126.514395895	64.03037553	0.581573977

As we can see, CUDA version of password cracking is very faster than the original and multithread version. The difference between those is that CUDA runs on GPU and POSIX runs on CPU. Since GPU has many CUDA cores, it will be faster than the original program. CUDA is also a parallel computing platform and can process huge number of data parallelly.

Gaussian Blur

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

#include "lodepng.h"

/********
Compile with nvcc 2040368_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;

    // 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];


}
```

```
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 = "image.png";
        const char* newFileName = "blur.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;
```

```
	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);

	//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;
}
```

Image

Result

GitHub: https://github.com/akash2040/HPC