UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

6CS005 High Performance Computing

Portfolio Task

Student Name: Pratik Dhimal

University ID: 2407779

Group: L6CG1

Tutor: Yamu Poudel

Submission Date: 29th January, 2026

Table of Contents

1. Introduction

This project will examine the use of parallel and heterogeneous computing methods to address performance-sensitive problems that prove to be inefficient when run in single thread execution. With growing data-sets and computational complexity, standard sequential programs do not utilizes the the newer multi-core CPUs and GPUs. To overcome this weakness, the coursework applies several high-performance solutions based on **POSIX threads, OpenMP, and CUDA**, showing how concurrency, synchronization, and appropriate memory management can contribute significantly to the speed of execution without reducing correctness and reliability.

The project has four tasks that are major. The former one is an implementation of a parallel word frequency counter in C with POSIX threads, in which a large input file is counted by a number of threads carefully using shared memory, synchronization, and boundary conditions. The second assignment deals with the operations of matrices on OpenMP such as arithmetic, transposition, and matrix multiplication, and the importance of scalability of the work, its resistance to the issue of input corruption, and dynamical allocation of the memory. The third assignment presents the idea of password cracking on the GPUs, and the specifics of CUDA include the execution of a kernel, transfers between memory, and the communication between the device and the host. The last task uses CUDA to edge detection with Sobel showing parallel applications to the images on the GPU. Taken together, these tasks demonstrate fundamental concepts of parallel computing both on the CPU and the GPU architectures such as optimization of performance, synchronization in parallel computers, and proper use of memory in computing high-performance computer systems.

2. Word Occurance Counter Using Posix

2.1. Input Validiation

This programm expects 2 additional argument which are the fiel containing the word and another is the number of threads to use. In this very first steps we check weather or there are right number of arguments which is 3 in this case, and also we check and notify the user if the number of thread is a positve integer.

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <filename> <thread_count>\n", argv[0]);
        return 1;
    }

    char *endptr;
    long thread_count = strtol(argv[2], &endptr, 10);
    if (*endptr != '\0' || thread_count < 1) {
        printf("Please enter a valid positive integer for the thread count.\n");
        return 1;
    }
```

Figure 1 Input Validiation

2.2. File handeling

In this process we read the file in the memory. If the file doesn't exist than a error message is shown to the user. Initially the file pointer as shown in figure below is pointing to the start of the file than we use the **fseek()** function with **SEEK_END** argument to change the pointer to the end and then read the current pointer value using **ftell()** which return the **file size** in bytes. Next we use **rewind()** with the file pointer argument to repoint the pointer to the start of the file. Now we check weather or not the file is empty using the **file_size,** if it is zero than the file is empty. If the file is empty we close the programm. If file content exist than we use **malloc()** to allocate memory. We allocate one additioal byte of memory which is later filled with end terminator '\0' to indicate the end of cotnet. Lastly we fill the memory with the **file data** usign the **fread()** function and close the file using **fclose()** function.

```
FILE *file = fopen(argv[1], "r");
if (file == NULL) {
    perror("Error opening input file");
    return 1;
}

fseek(file, 0, SEEK_END);
long file_size = ftell(file);
rewind(file);

if (file_size == 0) {
    printf("Error: File is empty\n");
    fclose(file);
    return 1;
}

char *buffer = malloc(file_size + 1);
if (buffer == NULL) {
    perror("Error allocating memory");
    fclose(file);
    return 1;
}

size_t bytes_read = fread(buffer, 1, file_size, file);
if (bytes_read == 0) {
    perror("Error reading file");
    free(buffer);
    fclose(file);
    return 1;
}
buffer[bytes_read] = '\0';
fclose(file);
```

Figure 2 File handeling

2.3. Buffer slicing and thread creation

Here the thread id is initialized and the thread argument is also declared using malloc. Next a loop is runned to the thread count so that we can populate the thread arguments with necessary value. Now we fill each thread arguments with the start(size from where to start in bytes), end(size where to end in bytes), buffer(the actual file buffer), and the total buffer size(bytes). Next we run the thread function using the **pthread_create().** In this case we run the **process_slice()** function with the thread arguments creted previously.Lastly we joint all the threads using the **pthred_join()** function.

```
pthread_t *threads = malloc(thread_count * sizeof(pthread_t));
ThreadArgs *thread_args = malloc(thread_count * sizeof(ThreadArgs));

if (threads == NULL || thread_args == NULL) {
    perror("Error allocating memory for threads");
    free(buffer);
    free(threads);
    free(thread_args);
    return 1;
}

long slice_size = actual_size / thread_count;
```

Figure 3 threads argument creation

```
 for (int i = 0; i < thread_count; i++) {
    thread_args[i].text_buffer = buffer;
    thread_args[i].start_index = i * slice_size;
    thread_args[i].end_index = (i == thread_count - 1) ? actual_size : (i + 1) * slice_size;
    thread_args[i].buffer_size = actual_size;

    if (pthread_create(&threads[i], NULL, process_slice, &thread_args[i]) != 0) {
        perror("Failed to create thread");
        free(buffer);
        free(threads);
        free(thread_args);
        return 1;
    }
 }

 for (int i = 0; i < thread_count; i++) {
    pthread_join(threads[i], NULL);
 }
```

Figure 5 Pouplating Threads args and running thread

2.4. Result Generation and Output Handling

As shown in figure below, the final processing of results, reporting, and cleanup of all threads that have completed counting words are the responsibilities of this part of the program. To begin with, the global words array is sorted using **qsort()**in a descending order of frequency of the words in the array where the most frequently appearing words come at the beginning of the output. The program subsequently opens an output file with the error control so as to check if the file result.txt has been created successfully. A structured header is appended to the file alongside the number of unique words which are summed up and the column names that should be read. The program repeats the sorted word list reading and writing out each word and its frequency to the file and at the same time adds together the total number of words handled by adding individual counts together. The key statistics are also shown on the console afterward after printing a summary section (total words processed) so that the user can give feedback. The last thing that happens is the program closes the output file, the dynamically allocated memory (buffer, thread array, and thread argument array) are freed, and 0 is returned indicating successful execution.

```
// Sort words by frequency (descending order)
qsort(words, word_count, sizeof(WordEntry), compare_frequency);

FILE *out = fopen("result.txt", "w");
if (out == NULL) {
    perror("Error opening output file");
    free(buffer);
    free(threads);
    free(thread_args);
    return 1;
}

fprintf(out, "Word Frequency Count Results\n");
fprintf(out, "============================\n");
fprintf(out, "Total unique words: %d\n", word_count);
fprintf(out, "(Sorted by frequency - descending order)\n\n");
fprintf(out, "%-20s %s\n", "Word", "Count");
fprintf(out, "-------------------- ------\n");

int total_words = 0;
for (int i = 0; i < word_count; i++) {
    fprintf(out, "%-20s %6d\n", words[i].word, words[i].count);
    total_words += words[i].count;
}

fprintf(out, "\n============================\n");
fprintf(out, "Total words processed: %d\n", total_words);

printf("Processing complete!\n");
printf("Total unique words: %d\n", word_count);
printf("Total words processed: %d\n", total_words);
printf("Results saved to result.txt\n");

fclose(out);
free(buffer);
free(threads);
free(thread_args);

return 0;
}
```

Figure 6 Adding content to file and shwoign necessary logs

2.5. The process_slice() function

This is the worker logic that is the essence of every thread and that which is used to process a given segment of the input text buffer. Each thread is provided with a **ThreadArgs** structure that has the shared text buffer and start and end indices of it slice. The operation starts with the initialisation of a temporary **current_word buffer** and a position index with which words are constructed character by character. As a way of avoiding counting a partial word, the thread initially examines whether the index at which it currently is in an **alpha-numeric sequence** here **the non alpha-numberic sequenc is the "\n"** which indicated next line is in the middle of the sequence; in that case, the index is advanced until the index reaches a word boundary. This loop then reads characters in the slice assigned, adding alphanumeric characters to the current word, and changing them to lowercase to enforce the counting of cases being insensitive to the case. Upon coming to a non-alphanumeric character, the word that has been formed is terminated at the end and safely inserted into the global word list.

The function also takes precautions with the boundary conditions in the end of the slice where a word can be longer than the range assigned to the thread. Once the initial loop is done, another loop starts again reading characters beyond the slice point until a **delimiter** is located, which makes sure that a word cut between two slices is counted once by the current thread. Should there be a meaningful word in the buffer at the end of this process then it is completed and inserted into the world collection. This functionality ensures that word counts are made correctly without redundancy or data races by making sure that the index adjustment is done very carefully, word construction is done very carefully, and updates are synchronized even when multiple threads are using the same shared buffer.

```
void *process_slice(void *arg) {
    ThreadArgs *data = (ThreadArgs *)arg;
    if (data == NULL) return NULL;

    char current_word[50];
    int pos = 0;

    long i = data->start_index;

    // This line Skips partial word at the start (if we're in the middle of a word)
    if (i > 0 && isalnum(data->text_buffer[i-1])) {
        while (i < data->end_index && i < data->buffer_size && isalnum(data->text_buffer[i])) {
            i++;
        }
    }

    // Process words in this slice
    for (; i < data->end_index && i < data->buffer_size; i++) {
        char c = data->text_buffer[i];
        if (isalnum(c)) {
            if (pos < 49) {
                current_word[pos++] = tolower(c);
            }
        } else {
            if (pos > 0) {
                current_word[pos] = '\0';
                add_word(current_word);
                pos = 0;
            }
        }
    }

    // Complete the word that crosses the boundary
    while (i < data->buffer_size && isalnum(data->text_buffer[i])) {
        if (pos < 49) {
            current_word[pos++] = tolower(data->text_buffer[i]);
        }
        i++;
    }

    if (pos > 0) {
        current_word[pos] = '\0';
        add_word(current_word);
    }

    return NULL;
}
```

Figure 7 Process slice function

2.6. The add_word() function

This function performs the work of safely updating the data structure of words frequencies in the world in a multithreaded setting. When a thread recognizes a complete word it calls **add_word()**, which initially takes out a **mutex lock** to be able to exclusively access the shared words array and variable word count to avoid **race conditions**. The function then conducts a linear search of the available entries in order to determine whether the word already exists; in this case of a match the appropriate frequency count is increased and the mutex returned. When the word is not found and there was still space, the feature will add the new word to the array, set its count to one, and the total number of unique words. Lastly, **the mutex is released** and only a single thread is allowed to update the shared data at any point in time and keeps the concurrent execution **correct and consistent**.

```
void add_word(char *w) {
    pthread_mutex_lock(&lock);
    for (int i = 0; i < word_count; i++) {
        if (strcmp(words[i].word, w) == 0) {
            words[i].count++;
            pthread_mutex_unlock(&lock);
            return;
        }
    }
    if (word_count < 5000) {
        strcpy(words[word_count].word, w);
        words[word_count].count = 1;
        word_count++;
    }
    pthread_mutex_unlock(&lock);
}
```

Figure 8 The add word function

3. Matrices Arthemetics Using OpenMP

2.1. Input Validation

Endptr checks and strtod() function are utilized in the program. This will ensure that every letter in a token is a valid part of a number. The software identifies an error and removes the memory already in use and proceeds to the next pair of matrices when it comes across a non-numeric token. This will ensure integrity of output and prevent the system to process garbage data.

```
int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <input_file> <threads>\n", argv[0]);
        return 1;
    }

    char *endptr;
    long tcount = strtol(argv[2], &endptr, 10);
    if (*endptr != '\0' || tcount <= 0)
    {
        fprintf(stderr, "Invalid thread number.\n");
        return 1;
    }
    int threads = (int)tcount;

    FILE *fin = fopen(argv[1], "r");
    if (!fin)
    {
        fprintf(stderr, "Cannot open file %s\n", argv[1]);
        return 1;
    }

    FILE *fout = fopen("result.txt", "w");
    if (!fout)
    {
        fprintf(stderr, "Cannot create output file\n");
        fclose(fin);
        return 1;
    }

    int pair_idx = 0;
```

Figure 9 Input validiation and file handeling

2.2. Memory Management and Safety

This part is concerned with the safe creation, deallocation and access of matrices in memory, and the reading and writing of files. The **allocate_matrix** function is a dynamic means of constructing a 2D array of doubles and detects allocation failures and the **deallocate_matrix** is used to free all memory allocated to avoid leakage. In managing files, load matrix uses the read token to read out a complete matrix of the input file with the dimensions and contents being valid integers and read token reads numeric values in a file and ignores commas, spaces and new lines. Similarly, the **display_matrix** to file is a form of representation that numbers extensive numbers and elegantly receives NaNs when writing a matrix to a file. A combination of these processes ensures uniform file I/O and high memory usage within the application.

```
double **allocate_matrix(int rows, int cols)
{
    if (rows <= 0 || cols <= 0)
        return NULL;
    double **mat = (double **)malloc(rows * sizeof(double *));
    if (!mat)
        return NULL;

    for (int i = 0; i < rows; i++)
    {
        mat[i] = (double *)malloc(cols * sizeof(double));
        if (!mat[i])
        {
            for (int k = 0; k < i; k++)
                free(mat[k]);
            free(mat);
            return NULL;
        }
    }
    return mat;
}

void deallocate_matrix(double **mat, int rows)
{
    if (!mat)
        return;
    for (int i = 0; i < rows; i++)
        if (mat[i])
            free(mat[i]);
    free(mat);
}

int read_token(FILE *fp, char *buf)
{
    if (fscanf(fp, " %127[^, \t\n]", buf) != 1)
        return 0;
    int ch = fgetc(fp);
    if (ch != ',' && !isspace(ch) && ch != EOF)
        ungetc(ch, fp);
    return 1;
}
```

Figure 10 Memory allocation

```c
double **load_matrix(FILE *fp, int *rows_out, int *cols_out)
{
    char token[128];
    if (!read_token(fp, token))
        return NULL;
    if (!check_numeric(token))
    {
        fprintf(stderr, "Rows header invalid: '%s'\n", token);
        return NULL;
    }
    int r = atoi(token);

    if (!read_token(fp, token))
        return NULL;
    if (!check_numeric(token))
    {
        fprintf(stderr, "Cols header invalid: '%s'\n", token);
        return NULL;
    }
    int c = atoi(token);

    if (r <= 0 || c <= 0)
    {
        fprintf(stderr, "Invalid matrix size %dx%d\n", r, c);
        return NULL;
    }

    double **M = allocate_matrix(r, c);
    if (!M)
    {
        fprintf(stderr, "Failed memory allocation for %dx%d\n", r, c);
        return NULL;
    }

    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
        {
            if (!read_token(fp, token))
            {
                fprintf(stderr, "Unexpected EOF while reading matrix data\n");
                deallocate_matrix(M, r);
                return NULL;
            }
            if (!check_numeric(token))
            {
                fprintf(stderr, "Non-numeric value detected: '%s'\n", token);
                deallocate_matrix(M, r);
                return NULL;
            }
            M[i][j] = strtod(token, NULL);
        }

    *rows_out = r;
    *cols_out = c;
    return M;
}
```

Figure 11 Load matrix to memory

```
void display_matrix(FILE *fp, double **M, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            if (isnan(M[i][j]))
                fprintf(fp, "NaN");
            else
                fprintf(fp, "%0.6lf", M[i][j]);
            if (j < cols - 1)
                fprintf(fp, " | ");
        }
        fprintf(fp, "\n");
    }
}
```

Figure 12 Display Matric

2.3.OpenMP Parallelisation and Thread Capping

This program uses OpenMP to parallelize computationally intensive tasks across multiple threads, allowing for the simultaneous execution of loops for matrix arithmetic, multiplication, and transposition. Thread capping is used to prevent the creation of more threads than the relevant matrix dimension in order to reduce overhead and ensure efficient resource use. By **dynamically adjusting** the number of threads in accordance with matrix size, the approach optimizes performance without unnecessary context switching or memory contention, striking a balance between speed and system stability. This architecture scales effectively with matrix dimensions while ensuring safe and predictable parallel execution.

```
➜  Task2 git:(master) ✗ cat task2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <omp.h>
#include <errno.h>
#include <stdint.h>

int check_numeric(const char *str)
{
    if (!str || !*str)
        return 0;
    char *endptr;
    errno = 0;
    strtod(str, &endptr);
    if (errno != 0 || endptr == str)
        return 0;
    while (*endptr)
    {
        if (!isspace((unsigned char)*endptr) && *endptr != ',')
            return 0;
        endptr++;
    }
    return 1;
}

int limit_threads(int requested, int max_dim_val)
{
    return (max_dim_val < 1) ? 1 : ((requested > max_dim_val) ? max_dim_val : requested);
}

int maximum(int x, int y) { return (x > y) ? x : y; }
```

Figure 13 Utility Functions

2.4.Matrix Calculations

2.4.1. Addition

The **matrix addition** process involves the addition of two matrices of same dimensions element by element. The function calculates the sum of the matching elements through the iteration of the row and column using simultaneous threads, after memory allocation of the ultimate matrix. To avoid corrupt operations, addition is not performed when the matrices of the input are of dissimilar shapes. This technique involves the OpenMP threading that can be used to integrate secure memory handling with efficient computation.

2.4.2. Subtraction

Element-wise Matrix subtraction refers to the operation of subtraction between two matrices of equal size, similar to addition. The difference between the respective elements of the input matrices is calculated to obtain the elements of the output matrix, and a new result matrix is allotted. Whereas dimension checks prevent any not valid practices, parallelisation is employed to accelerate computation. This will ensure that accuracy and performance is maintained.

2.4.3. Element wise multiplication

Element-wise multiplication is the computation of the product of similar elements in two matrices of the same size. The process multiplies every element of the former matrix by the element of the latter one after assigning a new matrix to contain the outcomes. OpenMP parallel threads are used to improve performance and shape validation used to maintain stability in the program to ensure that only compatible matrices are multiplied.

2.4.4. Element wise division

The **Division** of corresponding elements of two matrices with the same dimensions is determined by element-wise division. Each element is divided into a result matrix, with a check to assign NaN in the event that the division is by zero. Performance can be increased by processing multiple rows at once thanks to parallelization, which also makes sure that invalid operations—like division by zero or mismatched shapes—are handled safely.

2.4.5. Transpose

The **Transpose** operation transforms rows into columns and vice versa. Each element is copied to its transposed position and a new matrix is assigned with the dimensions reversed. To increase processing speed, OpenMP parallelisation is used row-wise. This operation is essential for later operations like matrix multiplication and is always valid regardless of the original matrix structure.

```
double **matrix_add(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] + B[i][j];
    return R;
}

double **matrix_subtract(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] - B[i][j];
    return R;
}

double **matrix_elem_mul(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = A[i][j] * B[i][j];
    return R;
}

double **matrix_elem_div(double **A, double **B, int rows, int cols, int num_threads)
{
    double **R = allocate_matrix(rows, cols);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, maximum(rows, cols));
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            R[i][j] = (B[i][j] == 0.0) ? NAN : A[i][j] / B[i][j];
    return R;
}
```

Figure 14 Matrix calculation 1

2.4.6. Dot Product (Multiplication*)*

When the number of columns in the first matrix is equal to the number of rows in the second, **matrix multiplication** calculates the dot product of the two matrices. Each element is calculated as the sum of products of related elements in the row of the first matrix and the column of the second, and a result matrix with the proper output dimensions is assigned. While shape

validation guarantees that only compatible matrices are multiplied, preserving correctness, parallelisation is used across rows to speed up computation.

```
double **matrix_transpose(double **A, int rows, int cols, int num_threads)
{
    double **T = allocate_matrix(cols, rows);
    if (!T)
        return NULL;
    int t = limit_threads(num_threads, rows);
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            T[j][i] = A[i][j];
    return T;
}

double **matrix_multiply(double **A, double **B, int rA, int cA, int cB, int num_threads)
{
    double **R = allocate_matrix(rA, cB);
    if (!R)
        return NULL;
    int t = limit_threads(num_threads, rA);
#pragma omp parallel for num_threads(t)
    for (int i = 0; i < rA; i++)
        for (int j = 0; j < cB; j++)
        {
            double sum = 0.0;
            for (int k = 0; k < cA; k++)
                sum += A[i][k] * B[k][j];
            R[i][j] = sum;
        }
    return R;
}
```

Figure 15 Matrix Calculation 2

2.5.Handling Edge Cases and Division

Decoupling the operations was the last logic improvement to make sure that a failure in one would not halt the process as a whole. Mismatched dimensions for addition could completely halt the processing of that matrix pair in the base code.The updated program independently verifies each operation's requirements. Additionally, a particular check for "division by zero" was added for element-wise division. When a divisor is zero, the program specifically gives that cell a NaN (Not a Number) value. This keeps the program from crashing with a floating-point exception and guarantees that the output complies with standards for scientific computing.

4. Password Cracking Using CUDA

4.1. Generate encrypted password in the CPU

In this process the **PasswordGenerator.c** file generates password in **format aa11** where first two character is always lower cased alphabet and last 2 is always numeric valus. Then the 4 letter password is encrypted to a 10 letter password with simple asci conversion techniques. This password is than stored in a textfile.

```
⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3 cat PasswordGenerator.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void encrypt_ten_character(char* raw, char* out) {

    out[0] = raw[0] + 2;
    out[1] = raw[0] - 2;
    out[2] = raw[0] + 1;
    out[3] = raw[1] + 3;
    out[4] = raw[1] - 3;
    out[5] = raw[1] - 1;
    out[6] = raw[2] + 2;
    out[7] = raw[2] - 2;
    out[8] = raw[3] + 4;
    out[9] = raw[3] - 4;
    out[10] = '\0';

    // handle ascii wrapping
    for(int i=0; i<10; i++) {
        if(i < 6) {
            while(out[i] > 'z') out[i] -= 26;
            while(out[i] < 'a') out[i] += 26;
        } else {
            while(out[i] > '9') out[i] -= 10;
            while(out[i] < '0') out[i] += 10;
        }
    }
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        printf("Usage: %s <count (>= 10000)>\n", argv[0]);
        return 1;
    }

    int count = atoi(argv[1]);
    if(count < 10000) {
        printf("Error: count must be at least 10000\n");
        return 1;
    }

    FILE* fp = fopen("cpu_encrypted_password.txt", "w");

    // //just for checking purpose
    // FILE *ogPassword = fopen("originalPassword.txt", "w");

    if(!fp) {
        printf("Unable to open fiel for writing\n");
        return 1;
    }

    srand(time(NULL));
    char raw[5];
    char enc[12];

    for(int i=0; i<count; i++) {
        // generate raw passwords in format aa00, bb12, ce19
        raw[0] = 'a' + rand() % 26;
        raw[1] = 'a' + rand() % 26;
        raw[2] = '0' + rand() % 10;
        raw[3] = '0' + rand() % 10;
        raw[4] = '\0';

        // fprintf(ogPassword, "%s\n", raw);

        encrypt_ten_character(raw, enc);

        // save only the encrypted version
        fprintf(fp, "%s\n", enc);
    }

    fclose(fp);
    printf(" %d passwords generation succesfull to cpu_encrypted_password.txt \n", count);
    return 0;
}
```

Figure 16 PasswordGenerator.c

4.2. Hash the encrypted password using SHA512 Algorithm

In this process the build in function from **crypt()** is used to hash the password which wash encrypted in above step. This function comes from **crypt.h header file.** What happens here is the program goes through each line of the initially encrypted password and than hashes it and store to a output text file as specified in the comman line.

```
cat: Encrypt.c: No such file or directory
[⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3 cat EncryptSHA512.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crypt.h>

// standard sha512 salt
#define SALT "$6$AS$"

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf("usage: ./hasher <input_file> <output_file>\n");
        return 1;
    }

    FILE* fin = fopen(argv[1], "r");
    FILE* fout = fopen(argv[2], "w");

    if(!fin || !fout) { printf("file open error\n"); return 1; }

    char line[100];
    // read every line
    while(fgets(line, sizeof(line), fin)) {

        line[strcspn(line, "\r\n")] = 0;

        // hash it using  crypt lib
        char* hash = crypt(line, SALT);

        // save hash
        fprintf(fout, "%s\n", hash);
    }

    fclose(fin);
    fclose(fout);
    printf("hashing done from %s to %s\n", argv[1], argv[2]);
    return 0;
}
⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3
```

Figure 17 EncryptSHA512.c

4.3. Hash all the possible password into SHA512.

For the first time we utilized the power GPU to calcualte all the possible combination of our passwords and than encrypt it usign the same way like we did in the first step. So now we'll have all the possible combination of **encrypted but not hashed passwords** which is **67600** in our case. Than as shown in **Figure ,** the same **CPU function** is used to hash the 67600 passwords. Thus finally we'll have 67600 properly hashed passwords usign the same algorithm as the target file.

```
#define TOTAL_COMBOS 67600
#define ENC_LEN 11

__global__ void create_password(char* buffer) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= TOTAL_COMBOS) return;

    char raw[5];

    raw[0] = 'a' + (idx / 2600);
    idx %= 2600;
    raw[1] = 'a' + (idx / 100);
    idx %= 100;
    raw[2] = '0' + (idx / 10);
    raw[3] = '0' + (idx % 10);

    char* out = &buffer[(blockIdx.x * blockDim.x + threadIdx.x) * ENC_LEN];

    out[0]  = raw[0] + 2;
    out[1]  = raw[0] - 2;
    out[2]  = raw[0] + 1;
    out[3]  = raw[1] + 3;
    out[4]  = raw[1] - 3;
    out[5]  = raw[1] - 1;
    out[6]  = raw[2] + 2;
    out[7]  = raw[2] - 2;
    out[8]  = raw[3] + 4;
    out[9]  = raw[3] - 4;
    out[10] = '\0';

    for (int i = 0; i < 10; i++) {
        if (i < 6) {
            while (out[i] > 'z') out[i] -= 26;
            while (out[i] < 'a') out[i] += 26;
        } else {
            while (out[i] > '9') out[i] -= 10;
            while (out[i] < '0') out[i] += 10;
        }
    }
}
```

Figure 18 Password Generation Kernel

4.4. Crack the password and store to file

Brute-force search is initially performed on the GPU with all possible passwords generated correspondingly to the passwords by all possible password combinations and the encrypted/ hash values are stored in a large look-up table of 67,600 encrypted hash values. All the items in the table are distinct passwords based on a predetermined pattern and encrypted password. During the cracking, the CUDA kernel is executed to ensure that each thread in the GPUs has a single target hash. The thread then performs a comparison of its target hash against all the precalculated hashes in the table in sequence, the brute-force step of comparison. In case of a match, the matching table index is transformed back to the original plaintext password and the result is stored in the output; when no match is found after searching the entire table, a dummy(XXX) is stored instead. This scheme decouples the generation step of brute-force with the comparison step, thus enabling the costly password space to be calculable only once and then reused successfully to crack numerous target hashes at once.

```
__device__ int strings_equal(const char* a, const char* b) {
    for (int i = 0; i < HASH_LEN; i++) {
        if (a[i] != b[i]) return 0;
        if (a[i] == '\0') break;
    }
    return 1;
}

__device__ void index_to_password(int index, char* password) {
    password[0] = 'a' + (index / 2600);
    index %= 2600;
    password[1] = 'a' + (index / 100);
    index %= 100;
    password[2] = '0' + (index / 10);
    password[3] = '0' + (index % 10);
    password[4] = '\0';
}

__global__ void crack_passwords(
    char* target_hashes,
    char* precomputed_hashes,
    char* cracked_passwords,
    int target_count
) {
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id >= target_count) return;

    char* current_target = &target_hashes[thread_id * HASH_LEN];
    char* output_password = &cracked_passwords[thread_id * PASSWORD_LEN];

    int match_found = 0;

    for (int table_index = 0; table_index < HASH_TABLE_SIZE; table_index++) {
        char* table_hash = &precomputed_hashes[table_index * HASH_LEN];

        if (strings_equal(current_target, table_hash)) {
            index_to_password(table_index, output_password);
            match_found = 1;
            break;
        }
    }

    if (!match_found) {
        output_password[0] = '?';
        output_password[1] = '?';
        output_password[2] = '?';
        output_password[3] = '?';
        output_password[4] = '\0';
    }
}
```

Figure 19 Crack Password kernel

4.5. Memory deallocation – freeing memory appropriately

Lastly, all the GPU and CPU memory is freed to make sure memory leak doesn't happen.

```
FILE* output = fopen(output_file, "w");
if (!output) {
    printf("Failed to open output file\n");
    return 1;
}

for (int i = 0; i < target_count; i++) {
    fprintf(output, "%s\n", &host_results[i * PASSWORD_LEN]);
}

fclose(output);

printf("Password cracking completed successfully\n");

free(host_targets);
free(host_hash_table);
free(host_results);
cudaFree(device_targets);
cudaFree(device_hash_table);
cudaFree(device_results);

return 0;
```

Figure 20 Memory Deallocation

5. Sobel Edege Detection Using CUDA

5.1. Read image into CPU memory and store appropriately

In this step I have read the image file into CPU memory using then **lodepng_decode32_file()** function. This function decodes the image to its raw pixels form (RGBA for each pixels) and also gives the **width and heigth in pixels**. Later in the process, we'll need to convert back the pixels value to image so we've created a memory space named **h_out_rgba** using the maclloc() function which will be used later**.**

```
int main() {
    unsigned char* h_rgba = NULL;
    unsigned width, height;

    printf("Loading image from file...\n");
    unsigned error = lodepng_decode32_file(&h_rgba, &width, &height, "hck.png");
    if (error) {
        printf("ERROR: Failed to load image - %s\n", lodepng_error_text(error));
        return 1;
    }
    printf("SUCCESS: Image loaded - %u x %u pixels\n", width, height);

    size_t pixels = width * height;
    size_t rgba_size = pixels * 4;
    size_t gray_size = pixels;   // More specifically this coudl be called intensity value ( 1 value per pixel )


    unsigned char* h_out_rgba = (unsigned char*)malloc(rgba_size);
    if (!h_out_rgba) {
        printf("ERROR: Failed to allocate host memory\n");
        free(h_rgba);
        return 1;
    }
```

Figure 21 loadign image to cpu memory

5.2. Allocate GPU memory according to image size and transfer data to GPU

5.2.1. Allocating GPU Memory

Here **cudMalloc()** is used to allocate memory in the GPU, we have allocated memory for input rbga, gray converted, edeges detected, and output rgba for loding raw decoded rgba, storing converted gray values (one per pixel), detected edeges (one per pixel), output rgba to store the final manipulated rgba value respectively. In my case the **hck.png image is of 464 x 108 pixels** and as a singel **unsigned character holds 1 byte** of memory so the memory allocation looks soething like this as showin in table below:

Input RGBA	200448 bytes
GrayScale	50112 bytes
Edeges	50112 bytes
OutPut RGBA2	200448 bytes

5.2.2. Data Transfer to GPU

Now we have allocated sufficient memory in the GPU, we need to transfer data to it. We just transfer the **raw decoded pixel value** to the GPU as all other memory address will be filled by GPU itself by performing pixel manupulation. **CudeMemcpy()** function is used to transfer the data to the GPU we need to keep in mind the last argument is the direaction and it must be **cudaMemcpyHostToDevice** as we are copying from host(CPU) to device (GPU). How we have the required data in the device.

```
printf("\nAllocating GPU memory...\n");
unsigned char *d_rgba, *d_gray, *d_edges, *d_out_rgba; //memoryAllocation

cudaError_t err;

err = cudaMalloc(&d_rgba, rgba_size);
if (err != cudaSuccess) {
    printf("ERROR: cudaMalloc failed for d_rgba - %s\n", cudaGetErrorString(err));
    free(h_rgba);
    free(h_out_rgba);
    return 1;
}

err = cudaMalloc(&d_gray, gray_size);
if (err != cudaSuccess) {
    printf("ERROR: cudaMalloc failed for d_gray - %s\n", cudaGetErrorString(err));
    cudaFree(d_rgba);
    free(h_rgba);
    free(h_out_rgba);
    return 1;
}

err = cudaMalloc(&d_edges, gray_size);
if (err != cudaSuccess) {
    printf("ERROR: cudaMalloc failed for d_edges - %s\n", cudaGetErrorString(err));
    cudaFree(d_rgba);
    cudaFree(d_gray);
    free(h_rgba);
    free(h_out_rgba);
    return 1;
}

err = cudaMalloc(&d_out_rgba, rgba_size);
if (err != cudaSuccess) {
    printf("ERROR: cudaMalloc failed for d_out_rgba - %s\n", cudaGetErrorString(err));
    cudaFree(d_rgba);
    cudaFree(d_gray);
    cudaFree(d_edges);
    free(h_rgba);
    free(h_out_rgba);
    return 1;
}
```

Figure 22 Memory Allocation to GPU

5.3. Edges detected correctly via the CUDA kernel function

5.3.1. RBGA to GrayScale Kernel

In this kernel we pass all the raw rgba value and it converts to **grayscale intensity** value using the human **eye adaptive formula for grayscale based on luminiosity weights** rather than a normal average method we used to use in the past.

```
// CUDA KERNEL 1: RGB to Grayscale Conversion on GPU
__global__ void rgbToGrayscale(
    const unsigned char* rgba,
    unsigned char* gray,
    int width,
    int height
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    int rgba_idx = idx * 4;

    unsigned char r = rgba[rgba_idx + 0];
    unsigned char g = rgba[rgba_idx + 1];
    unsigned char b = rgba[rgba_idx + 2];

    // Grayscale conversion formula based on lumionicity of human eye(not the standard average method)
    gray[idx] = (unsigned char)(0.299f * r + 0.587f * g + 0.114f * b);
}
```

Figure 23 rgb to grayscale kernel

5.3.2. Edege Detection Kernel

This kernel performs the main logic of edege detection. Here we have two gradient **Gx and Gy** which is used to mease the rate of change of intensities in horizontal and vertical direction respectively.I t wokrs by multipling each value of the **gradient kernel** with its corrosponding grayscale value, and this step is repeated for each and every value of the grayscale intensity. So now we have both the gradiet in horizontal and vertical direction, then, **the formula** $\sqrt{Gx^2 + Gy^2}$ is used for the effective magnitude of intensity change. Lastly, the edeges values are populated to the **d_edges** variable.

```
// CUDA KERNEL 2: Sobel Edge Detection on GPU
__global__ void sobelEdgeDetection(
    const unsigned char* input,
    unsigned char* output,
    int width,
    int height
) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    int x = idx % width;
    int y = idx / width;


    int Gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };


    int Gy[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    int sumX = 0;
    int sumY = 0;


    for (int ky = -1; ky <= 1; ky++) {
        for (int kx = -1; kx <= 1; kx++) {
            // Calculate neighbor pixel coordinates
            int px = x + kx;
            int py = y + ky;

            // Zero padding: assume pixels outside image boundaries are 0
            unsigned char pixel = 0;
            if (px >= 0 && px < width && py >= 0 && py < height) {
                pixel = input[py * width + px];
            }

            //multiply pixel value by kernel value and accumulate
            sumX += pixel * Gx[ky + 1][kx + 1];
            sumY += pixel * Gy[ky + 1][kx + 1];
        }
    }


    int magnitude = (int)sqrtf((float)(sumX * sumX + sumY * sumY));

    // Clamp to valid pixel range [0, 255]
    if (magnitude > 255) magnitude = 255;

    output[idx] = (unsigned char)magnitude;
}
```

Figure 24 sobel edege detection kernel

5.3.3. Detected edeges to final RGBA kernel

We had one value per pexel in the gray scale intensity value now this kernel populates the value to all the RGB to make a perfect set of RGBA value, the A (transparency) is set to 255 for a perfectly opaque image. This makes sure we can later use the perfect set of RGBA to generatre output edege detected image.

```
// CUDA KERNEL 3: Convert Grayscale back to RGBA for output

__global__ void grayToRGBA(
    const unsigned char* gray,
    unsigned char* rgba,
    int width,
    int height
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    unsigned char v = gray[idx];
    rgba[idx * 4 + 0] = v;  // R
    rgba[idx * 4 + 1] = v;  // G
    rgba[idx * 4 + 2] = v;  // B
    rgba[idx * 4 + 3] = 255;  // A (fully opaque)
}
```

Figure 25 graysacel to rgba kernel

5.4. Transfer Edege detected value safely to the CPU

The **cudeMemcpy()** function is used with direction argument **cudaMemcpyDeviceToHost** to copy the final manipulated edege detected pixel value to the **h_out_rgba.** Now our Host(CPU) has the final sets of pixel value which could be easily encoded to a png image using the lodepng.

```
printf("\nTransferring result from device to host...\n");
err = cudaMemcpy(h_out_rgba, d_out_rgba, rgba_size, cudaMemcpyDeviceToHost);
if (err != cudaSuccess) {
    printf("ERROR: cudaMemcpy D2H failed - %s\n", cudaGetErrorString(err));
    cudaFree(d_rgba);
    cudaFree(d_gray);
    cudaFree(d_edges);
    cudaFree(d_out_rgba);
    free(h_rgba);
    free(h_out_rgba);
    return 1;
}
printf("SUCCESS: Result transferred to host\n");
```

Figure 26 Final RGBA value copy to host

5.5. Making final image and memory deallocation

Finally the **lodepng_encode32_file()** function is used to encode the pixel back to png image. This gives us a final png image named **sobel-final-image.png**. Lastly, we free all the memory used by the GPU as well as CPU to prevent memory leaking.

```
    printf("\nSaving output image...\n");
    error = lodepng_encode32_file("sobel-final-image.png", h_out_rgba, width, height);
    if (error) {
        printf("ERROR: Failed to save image - %s\n", lodepng_error_text(error));
    } else {
        printf("SUCCESS: Output saved to 'sobel-final-image.png'\n");
    }


    printf("\nCleaning up...\n");

    // Free GPU memory
    cudaFree(d_rgba);
    cudaFree(d_gray);
    cudaFree(d_edges);
    cudaFree(d_out_rgba);
    printf("  - GPU memory freed\n");

    // Free CPU memory
    free(h_rgba);
    free(h_out_rgba);
    printf("  - CPU memory freed\n");

    printf("\n=== Sobel Edge Detection Complete ===\n");
    return 0;
}
:wq
```

Figure 27 memory deallocation

```
bash: nvcc: command not found
pratik2407779@eb59e865dc53:/content$ ./task4
Loading image from file...
SUCCESS: Image loaded - 464 x 108 pixels

Allocating GPU memory...
SUCCESS: GPU memory allocated
  - Input RGBA: 200448 bytes
  - Grayscale: 50112 bytes
  - Edges: 50112 bytes
  - Output RGBA: 200448 bytes

Transferring data to GPU...
SUCCESS: Data transferred to GPU

Executing edge detection on GPU...
  Grid: 196 blocks
  Block: 256 threads

  Step 1: Converting to grayscale...
  SUCCESS: Grayscale conversion complete
  Step 2: Applying Sobel edge detection...
  SUCCESS: Edge detection complete
  Step 3: Converting to RGBA for output...
  SUCCESS: RGBA conversion complete
SUCCESS: All GPU processing complete

Transferring result from device to host...
SUCCESS: Result transferred to host

Saving output image...
ERROR: Failed to save image - failed to open file for writing

Cleaning up...
  - GPU memory freed
  - CPU memory freed

=== Sobel Edge Detection Complete ===
```

Figure 28 output commandline

Figure 29 output image

6. Conclusion

Lastly, it can be concluded that this portfolio has shown that parallel and heterogeneous computing methods can dramatically enhance performance, scalability and reliability in solving computational intensive problems. The project emphasizes thorough thread coordination, synchronization and boundary management to achieve the correctness in shared-memory parallelism through the POSIX-based word occurrence counter. The matrix arithmetic tasks with the help of OpenMP also note out the significance of safe memory management, input validation and adaptive thread usage to provide efficient and stable parallel execution by the CPU. CUDA password cracking task on the GPU side both demonstrates the power of massive parallelism to greatly reduce brute-force computation time by decoupling the generation and comparison steps, and the Sobel edge detector task demonstrates the successful application of both the GPU memory and kernel implementation as well as data transfers to a real-world image processing task. All these implementations collectively support the foundations of core concepts of high-performance computing, including concurrency, synchronization, memory efficiency and architecture-sensitivity, and show that modern multi-core CPU and graphic card hardware can be used to solve previously impractical or inefficient problems that are possible only in purely sequential programs.