1. Password cracking using multithreading (15% - 100 marks)
    a. **Cracks a password using multithreading and dynamic slicing based on thread count (75 marks)**

Answer:

Implemented Multithreading:

- The program successfully implements multithreading using the pthread library. Each thread is assigned a portion of the search space based on the total number of threads specified by the user.

Dynamic Slicing of Workload:

The workload is divided among the threads dynamically using the following approach:

- The number of uppercase letters (26) is split among the specified number of threads.
- Each thread gets an equal portion of the letter range (NUM_LETTERS / thread_count).
- Any remainder is distributed to the first few threads to ensure a fair workload distribution.
- The first character is assigned to a specific thread, while the remaining characters (letters and digits) are explored exhaustively.

Synchronization and Thread Safety:

- Mutex (pthread_mutex_t) is used to ensure safe access to shared resources (e.g., checking and updating the password_found flag, printing messages).
- The password_found flag is checked within loops to allow early termination.

Efficient Execution & Debugging Support:

- Debugging print statements show the assigned workload ranges and progress.
- Threads only continue running until a password is found, ensuring efficiency.

Overall, the program correctly cracks a password using multithreading and dynamic slicing.

b. Program finishes appropriately when password has been found (25 marks)
Answer:

Proper Early Termination:

- The password_found flag is set immediately when a thread finds the password, preventing unnecessary processing.
- Each loop checks password_found, so threads terminate early rather than continuing unnecessary computations.
- This ensures that once the correct password is found, no additional computations waste CPU time.

Proper Cleanup & Execution Flow:

- Threads are properly joined using pthread_join(), ensuring no orphaned threads.
- Mutex and condition variables are destroyed properly after execution.
- The program prints the final found password and execution time before exiting.

Correct Output and Handling:

- If the password is found, it is displayed, and the program exits successfully.
- If not found, it informs the user accordingly.
- Time taken for execution is measured and printed, confirming efficiency.

2. Matrix Multiplication using multithreading (25% - 100 marks)

a. Read data from file appropriately (20 marks)

Answer:

The program reads matrices from an input file by first extracting the matrix dimensions from the first line of each matrix block. It ensures that the data is properly formatted and handles missing or incorrect values gracefully. Empty lines are skipped, and if the file ends unexpectedly, appropriate error messages are displayed. The implementation makes use of fgets and sscanf to parse matrix dimensions and elements, ensuring robustness in handling real-world input files.

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

Answer:

To efficiently handle matrices of varying sizes, the program dynamically allocates memory using malloc. A two-dimensional array is implemented using an array of pointers, where each row is separately allocated. This approach optimizes memory usage and allows for flexibility in dealing with different matrix sizes. The allocate_matrix function ensures proper memory allocation and includes error handling to prevent memory leaks.

c. Creating an algorithm to multiply matrices correctly (20 marks)

Answer:

The multiplication algorithm follows the standard matrix multiplication rule. Each element C[i][j] in the result matrix is computed. This ensures that each element in the resulting matrix is derived correctly. The program checks for valid dimensions before performing multiplication, ensuring compatibility (A_cols == B_rows).

d. Using multithreading with equal computations (30 marks)

Answer:

The program optimally distributes the computational workload using multiple threads. Each thread is responsible for computing one row of the resulting matrix. Threads are created and managed dynamically based on the number of available threads and the number of rows to process. The use of pthread_create and pthread_join ensures efficient parallel execution while avoiding race conditions or excessive thread creation overhead.

e. Storing the correct output matrices in the correct format to a file (20 marks)
Answer:

The computed result is written to output.txt in a structured format. Each matrix is prefixed with its dimensions to facilitate easy reading. The output format mirrors the input format, ensuring that the result can be used for further computation if needed. The file writing is performed using fprintf, maintaining two decimal precision for readability.

3. Password Cracking using CUDA (30% - 100 marks)

a. Generate encrypted password in the kernel function (using CudaCrypt function) to be compared to original encrypted password (25 marks)
Answer:

The CudaCrypt function is implemented as a device function (__device__ void CudaCrypt(...)) and is used within the crack kernel to encrypt generated passwords. Each thread generates a possible password combination, encrypts it using CudaCrypt, and then compares it to the target encrypted password. If a match is found, the result is stored in result and a flag is set to indicate that the password was found.

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

The program correctly allocates memory on the GPU using cudaMalloc for:

- gpuAlphabet (size ALPHABET_SIZE)
- gpuNumbers (size NUMBER_SIZE)
- gpuTargetPassword (size ENCRYPTED_LENGTH)
- gpuResult (size 4)
- gpuFound (size bool)

After kernel execution, memory is freed using cudaFree, ensuring efficient resource management.

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

The program uses a 2D grid of blocks (dim3 blocks(ALPHABET_SIZE, ALPHABET_SIZE, 1)) and a 2D grid of threads (dim3 threads(NUMBER_SIZE, NUMBER_SIZE, 1)).

- Each block represents a unique combination of the first and second letters of the password.
- Each thread represents a unique combination of the two numeric digits.

This ensures parallelization across multiple blocks and threads, significantly speeding up the password cracking process.

d. **Decrypted password sent back to the CPU and printed (20 marks)**
Answer:

After the kernel completes execution, the decrypted password is copied from the GPU to the CPU using cudaMemcpy(result, gpuResult, sizeof(char) * 4, cudaMemcpyDeviceToHost). The program then checks if the password was found and prints it:

```
if (found) {
    printf("Password found! Raw: %c%c%c%c\n", result[0], result[1],
result[2], result[3]);
} else {
    printf("Password not found.\n");
}
```

4. Box Blur using CUDA (30% - 100 marks)
    a. **Reading in an image file into a single or 2D array (5 marks)**
    Answer:
    The image is read using the lodepng_decode32_file() function, which loads a PNG file into a **1D array** (unsigned char* image). Each pixel in the image is stored with 4 channels (RGBA). The dimensions of the image (width and height) are also extracted from the file.

```
unsigned char* image;
unsigned int width, height;
error = lodepng_decode32_file(&image, &width, &height, filename);
```

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

    Memory allocation for both input and output images is performed on the GPU using cudaMalloc(). The allocated memory size is calculated based on image dimensions (width * height * 4), considering that each pixel consists of 4 values (RGBA). After processing, memory is freed using cudaFree().

```
const int ARRAY_SIZE = width * height * 4;
cudaMalloc((void**)&d_in, ARRAY_SIZE);
cudaMalloc((void**)&d_out, ARRAY_SIZE);
...
cudaFree(d_in);
cudaFree(d_out);
```

c. Applying Box filter on image in the kernel function (30 marks)
Answer:

The applyBoxBlur kernel applies a 3x3 box filter to each pixel by computing the average of neighboring pixels' RGB values. It loops through a 3x3 grid, sums the RGB values, and divides by the total valid pixels. The alpha channel remains unchanged.

```
 for (int i = 0; i < 9; i++) {
    int newX = x + neighbors[i][0];
    int newY = y + neighbors[i][1];
    if (newX >= 0 && newX < width && newY >= 0 && newY < height) {
        int idx = getPixelIndex(newX, newY, width);
        sumR += input[idx];
        sumG += input[idx + 1];
        sumB += input[idx + 2];
        count++;
    }
}
output[outputIdx] = sumR / count;
output[outputIdx + 1] = sumG / count;
output[outputIdx + 2] = sumB / count;
output[outputIdx + 3] = input[outputIdx + 3]; // Preserve alpha
```

d. Return blurred image data from the GPU to the CPU (30 marks)
Answer:

Once the CUDA kernel applyBoxBlur finishes processing the image on the GPU, the blurred image is stored in d_out (the GPU memory buffer for output). However, since the image is required back on the CPU for further operations (such as saving it as a PNG), it must be transferred from GPU memory (device) to CPU memory (host).

CUDA provides a function cudaMemcpy() that handles this memory transfer. The process includes:

1. Allocating memory on the CPU (host_output) to store the blurred image.
2. Copying the image from d_out (GPU) to host_output (CPU) using cudaMemcpy().
3. Checking for any errors in the memory copy operation.
4. Using the copied data for saving the image.

```
// Allocate memory on CPU for the output image
unsigned char* host_output = (unsigned char*)malloc(ARRAY_BYTES);

// Copy the processed image back from GPU to CPU
cudaError_t cudaStatus = cudaMemcpy(host_output, d_out, ARRAY_BYTES,
cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    printf("CUDA error: %s\n", cudaGetErrorString(cudaStatus));
    return -1;
}
```

Breakdown of cudaMemcpy() Parameters:

```
cudaMemcpy(host_output, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

host_output → Destination (CPU memory).

d_out → Source (GPU memory containing blurred image).

ARRAY_BYTES → Size of data to be copied (width × height × 4 bytes per pixel).

cudaMemcpyDeviceToHost → Direction of transfer (from GPU to CPU).

e. Outputting the correct image with Box Blur applied as a file (20 marks)
Answer:

After retrieving the blurred image, it is saved as a PNG file using lodepng_encode32_file(). This function encodes the image in RGBA format and writes it to disk. The alpha channel remains unchanged, and error handling ensures successful file creation.

```
    lodepng_encode32_file("Output.png", host_output, width, height);
```