Module Assessment Report

Matrix Multiplication using multithreading

Code: Matrix_Multiplication.ipynb

Test

Input:

2,4
63786.246094,40011.285156,3964.368164,104490.265625
61623.480469,40147.527344,65730.289062,113177.250000

4,2
82109.453125,121819.585938
89118.632812,8397.555664
10468.322266,25872.001953
10549.710938,57341.414062

Output:

2,2
9947047936.000000,14200596480.000000
10519836672.000000,16034406400.000000

Explanation

Reading Data from the File

```
FILE* input_fp = fopen(argv[1], "r");
```

This statement opens the input file for reading, specified via the command-line argument argv[1]. If the file is not found or an error occurs, an error message is printed, and the program exits.

Dynamic Memory Allocation

```
float** matrixA = allocate_matrix(rowsA, colsA);
```

```
float** matrixB = allocate_matrix(rowsB, colsB);
float** matrixC = allocate_matrix(rowsA, colsB);
```

Here, matrices A, B, and C are dynamically allocated using the allocate_matrix function. It allocates an array of pointers to rows and then allocates each row individually using malloc, allowing for matrices of varying dimensions to be stored and managed efficiently.

Matrix Multiplication Algorithm

```
for (int i = start_row; i < end_row; ++i) {
    for (int j = 0; j < colB; ++j) {
        thread_args->C[i][j] = 0;
        for (int k = 0; k < colA; ++k) {
            thread_args->C[i][j] += thread_args->A[i][k] *
thread_args->B[k][j];
        }
    }
}
```

This section performs the actual multiplication for a specific subset of rows. It initializes the result matrix elements to zero and iteratively computes each element using nested loops over rows, columns, and the shared dimension of the input matrices.

Multithreading for Equal Computations

```
pthread_create(&threads[t], NULL, multiply_rows, &args[t]);
pthread_join(threads[t], NULL);
```

The pthread_create function starts a new thread to execute the multiply_rows function. Each thread is given an appropriate subset of rows to process, ensuring an even distribution of work across threads. The pthread_join function ensures that the main thread waits until all threads have completed their computations before proceeding.

Storing Output in Correct Format

```
FILE* output_fp = fopen(output_filename, "w");
write_matrix(output_fp, matrixC, rowsA, colsB);
fclose(output_fp);
```

This block creates an output file and writes the resulting matrix using the write_matrix function, which ensures that the data is formatted consistently with the input file. The file is named uniquely for each matrix pair processed, and the file pointer is closed after writing.

Password cracking using multithreading

Code: Password cracking using multithreading.ipynb

Test

Input for Encryption:

AB12

Encrypted Hash Output:

6AS$EquwSMfZH6UigdniioE8VWG9qfQ/iburie8TclTB4HCYRomJtmDsM31EqQEbs5Zk09UzWMOtHoXFFmdKRKVoy/

Input for Password Crack:

6AS$EquwSMfZH6UigdniioE8VWG9qfQ/iburie8TclTB4HCYRomJtmDsM31EqQEbs5Zk09UzWMOtHoXFFmdKRKVoy/

Output of Password Crack:

Using Salt: 6AS$
#617 AB12
6AS$EquwSMfZH6UigdniioE8VWG9qfQ/iburie8TclTB4HCYRomJtmDsM31EqQEbs5Zk09UzWMOtHoXFFmdKRKVoy/
620 solutions explored

Explanation

Password Encryption

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

#define SALT "$6$AS$"

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

    printf("Password: %s\n", argv[1]);
```

```
    // Encrypt using the specified salt
    char *encrypted = crypt(argv[1], SALT);

    if (encrypted != NULL) {
        printf("Encrypted password: %s\n", encrypted);
    } else {
        printf("Error encrypting password.\n");
    }

    return 0;
}
```

This program uses the crypt library to generate a SHA-512 hashed password based on user input. The hash computation is salted with the constant SALT, which adds complexity to the hashing process, preventing the use of precomputed rainbow tables. The program reads the plain-text password from the command-line argument and calls crypt with the specified salt to produce the final hashed string. The encrypted password is displayed to the user for subsequent use.

Cracking a Password Using Multithreading with Dynamic Slicing

```
pthread_t threads[num_threads];
ThreadArgs args[num_threads];
```

This array of thread identifiers and ThreadArgs structures assigns data to each thread, including the range of letters, the salt, and the encrypted password to compare against.

```
int letters_per_thread = 26 / num_threads;
int remaining_letters = 26 % num_threads;
char current_letter = 'A';

for (int i = 0; i < num_threads; ++i) {
    args[i].start_letter = current_letter;
    args[i].end_letter = current_letter + letters_per_thread - 1;
    if (i < remaining_letters) {
        args[i].end_letter += 1;
    }
    current_letter = args[i].end_letter + 1;

    strcpy(args[i].salt, salt);
    args[i].salt_and_encrypted = salt_and_encrypted;

    pthread_create(&threads[i], NULL, crack, &args[i]);
}
```

This block divides the search space evenly among threads via dynamic slicing. Each thread receives a contiguous range of starting letters and other combinations to explore. Extra remaining letters are assigned to the initial threads to ensure balanced workloads. pthread_create launches each thread and passes it the ThreadArgs structure.

```
void* crack(void* args) {
    ThreadArgs* thread_args = (ThreadArgs*)args;
    char plain[7];
    char *enc;
    int x, y, z;

    for (x = thread_args->start_letter; x <=
thread_args->end_letter && !password_found; x++) {
        for (y = 'A'; y <= 'Z' && !password_found; y++) {
            for (z = 0; z <= 99 && !password_found; z++) {
                sprintf(plain, "%c%c%02d", x, y, z);

                pthread_mutex_lock(&crypt_mutex);
                enc = (char *)crypt(plain, thread_args->salt);
                pthread_mutex_unlock(&crypt_mutex);

                pthread_mutex_lock(&count_mutex);
                count++;
                pthread_mutex_unlock(&count_mutex);

                if (strcmp(thread_args->salt_and_encrypted, enc)
== 0) {
                    pthread_mutex_lock(&found_mutex);
                    if (!password_found) {
                        password_found = 1;
                        printf("#%-8d%s %s\n", count, plain, enc);
                    }
                    pthread_mutex_unlock(&found_mutex);
                    return NULL;
                }
            }
        }
    }
    return NULL;
}
```

The crack function iterates through combinations of two capital letters and two numbers using nested loops. The crypt function hashes each plain-text combination, and the resulting hash is compared with the input encrypted password. A mutex locks access to the crypt function and shared variables like count and password_found. Once a match is found, the function prints the successful combination and stops further searches by setting a flag.

Program Finishes Appropriately When Password Has Been Found

```
if (strcmp(thread_args->salt_and_encrypted, enc) == 0) {
    pthread_mutex_lock(&found_mutex);
    if (!password_found) {
        password_found = 1;
        printf("#%-8d%s %s\n", count, plain, enc);
    }
    pthread_mutex_unlock(&found_mutex);
    return NULL;
}
```

When a thread identifies the correct combination, it locks the found_mutex to safely mark the password as found, ensuring no further combinations are explored. The flag password_found is checked at every loop iteration, and if set, other threads immediately exit without further processing.

Password Cracking using CUDA

Code: Copy of Cuda_Password_crack

Test

Input for Encryption:

AB12

Encrypted Hash Output:

CCBECA3162

Input for Password Crack:

CCBECA3162

Output of Password Crack:

Decrypted Password: AB12

Explanation

Generating an Encrypted Password Using the Kernel Function

```
__device__ void CudaCrypt(char* rawPassword, char* newPassword) {
    // Encryption logic for each input character
    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';

    // Ensure uppercase letter and numeric limits
    for (int i = 0; i < 10; i++) {
        if (i < 6) {
            if (newPassword[i] > 'Z') {
                newPassword[i] = (newPassword[i] - 'Z') + 'A';
            } else if (newPassword[i] < 'A') {
                newPassword[i] = ('A' - newPassword[i]) + 'A';
            }
        } else {
            if (newPassword[i] > '9') {
                newPassword[i] = (newPassword[i] - '9') + '0';
            } else if (newPassword[i] < '0') {
                newPassword[i] = ('0' - newPassword[i]) + '0';
            }
        }
    }
}
```

The CudaCrypt function implements the encryption logic that transforms the input password (rawPassword) into a hashed string (newPassword). It ensures that uppercase letters wrap within the ASCII range for 'A' to 'Z', and numbers wrap from '0' to '9'. This function resides on the GPU and is called within the kernel function to encrypt passwords.

```
encryptKernel<<<1, 1>>>(d_rawPassword, d_encryptedPassword);
```

The kernel function encryptKernel encrypts the provided raw password on the GPU using a single thread-block and thread configuration, generating an encrypted password that will be compared against user input for verification.

Memory Allocation on the GPU

```
char *d_rawPassword, *d_encryptedPassword;
cudaMalloc((void**)&d_rawPassword, sizeof(rawPassword));
cudaMalloc((void**)&d_encryptedPassword,
sizeof(encryptedPassword));
```

This code allocates memory on the GPU for the raw and encrypted passwords using cudaMalloc, which ensures that the encryption process occurs entirely on the GPU. Once the kernel computation completes, the GPU memory is freed with cudaFree.

Program Working with Multiple Blocks and Threads

```
dim3 threadsPerBlock(6, 6, 6);
dim3 numBlocks(5, 5, 2);
crackKernel<<<numBlocks, threadsPerBlock>>>(d_target,
d_decryptedPassword, d_found);
```

The kernel configuration uses 3D grids and blocks to allow parallel processing of all combinations within each dimension (letters and numbers). The threadsPerBlock defines the grid's layout, while numBlocks divides the search space across multiple blocks. This configuration ensures that each block searches unique ranges of uppercase letters and numeric combinations.

Kernel Function

```
__global__ void crackKernel(char* d_target, char*
d_decryptedPassword, bool* d_found) {
    const char uppercase[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    const char digits[] = "0123456789";
    int idx1 = blockIdx.x * blockDim.x + threadIdx.x;
    int idx2 = blockIdx.y * blockDim.y + threadIdx.y;
    int idx3 = blockIdx.z * blockDim.z + threadIdx.z;

    if (idx1 < 26 && idx2 < 26 && idx3 < 10 && !(*d_found)) {
        for (int idx4 = 0; idx4 < 10; idx4++) {
            char rawPassword[5] = {uppercase[idx1],
uppercase[idx2], digits[idx3], digits[idx4], '\0'};
            char encryptedGuess[11] = {0};
            CudaCrypt(rawPassword, encryptedGuess);

            bool match = true;
            for (int i = 0; i < 10; i++) {
                if (encryptedGuess[i] != d_target[i]) {
```

```
                    match = false;
                    break;
                }
            }

            if (match) {
                for (int i = 0; i < 4; i++) {
                    d_decryptedPassword[i] = rawPassword[i];
                }
                d_decryptedPassword[4] = '\0';
                *d_found = true;
                break;
            }
        }
    }
}
```

This kernel function employs multiple threads and blocks to search through all possible password combinations. Each thread computes indices based on its block and thread IDs. The function ensures that combinations are explored efficiently, and once a match is found, further computations cease.

Decrypted Password Sent Back to CPU and Printed

```
cudaMemcpy(&h_found, d_found, sizeof(bool),
cudaMemcpyDeviceToHost);
if (h_found) {
    cudaMemcpy(decryptedPassword, d_decryptedPassword, 5,
cudaMemcpyDeviceToHost);
    printf("Decrypted Password: %s\n", decryptedPassword);
}
```

After the kernel completes its execution, the decrypted password and the found status are copied back from the GPU to the CPU using cudaMemcpy. If a match is found, the password is printed to the console.

Box Blur using CUDA

Code: Copy of 4.◦Box_Blur_using_CUDA

Test

Input:

Output:

Explanation

Reading an Image File into an Array

```
unsigned char* readPNG(const char* filename, int* width, int*
height, int* channels) {
    FILE* fp = fopen(filename, "rb");
    if (!fp) {
        perror("Error opening file");
        return NULL;
    }

    png_structp png =
png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png) {
        fclose(fp);
        return NULL;
    }

    png_infop info = png_create_info_struct(png);
    if (!info) {
        png_destroy_read_struct(&png, NULL, NULL);
        fclose(fp);
        return NULL;
    }

    if (setjmp(png_jmpbuf(png))) {
        png_destroy_read_struct(&png, &info, NULL);
        fclose(fp);
        return NULL;
    }

    png_init_io(png, fp);
    png_read_info(png, info);

    *width = png_get_image_width(png, info);
    *height = png_get_image_height(png, info);
    *channels = png_get_channels(png, info);

    png_bytep* row_pointers = (png_bytep*)malloc(sizeof(png_bytep)
* (*height));
    unsigned char* data = (unsigned char*)malloc((*width) *
(*height) * (*channels));

    for (int y = 0; y < *height; ++y) {
        row_pointers[y] = data + y * (*width) * (*channels);
    }
```

```
    png_read_image(png, row_pointers);
    free(row_pointers);
    png_destroy_read_struct(&png, &info, NULL);
    fclose(fp);

    return data;
}
```

The readPNG function uses the libpng library to read an image file into a 2D array. It extracts metadata such as width, height, and the number of channels, then allocates a contiguous memory block for the image data, which is read row-by-row.

Allocating Memory on the GPU and Freeing it Afterwards

```
unsigned char *d_in, *d_out;
cudaMalloc(&d_in, width * height * channels);
cudaMalloc(&d_out, width * height * channels);

// Copy input image to the GPU
cudaMemcpy(d_in, h_in, width * height * channels,
cudaMemcpyHostToDevice);

// Free memory after processing
cudaFree(d_in);
cudaFree(d_out);
```

GPU memory is allocated for both the input and output images using cudaMalloc, ensuring that the entire image can be processed on the GPU. The input data is copied from the host to the device via cudaMemcpy. Memory is freed after processing using cudaFree to prevent memory leaks.

Applying Box Filter in the Kernel Function

```
__global__ void boxBlurKernelShared(unsigned char* d_in, unsigned
char* d_out, int width, int height, int channels) {
    extern __shared__ unsigned char shared_mem[];
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int tx = threadIdx.x + 1;
    int ty = threadIdx.y + 1;

    // Load the pixel into shared memory (center)
    for (int c = 0; c < channels; ++c) {
```

```
        int sharedIdx = (ty * (blockDim.x + 2) + tx) * channels +
c;
        int globalIdx = (y * width + x) * channels + c;
        shared_mem[sharedIdx] = (x < width && y < height) ?
d_in[globalIdx] : 0;
    }

    // Load borders (edge cases)
    if (threadIdx.x == 0 && x > 0) {
        for (int c = 0; c < channels; ++c) {
            shared_mem[(ty * (blockDim.x + 2)) * channels + c] =
(y < height) ? d_in[((y * width) + (x - 1)) * channels + c] : 0;
        }
    }

    if (threadIdx.x == blockDim.x - 1 && x < width - 1) {
        for (int c = 0; c < channels; ++c) {
            shared_mem[(ty * (blockDim.x + 2) + (blockDim.x + 1))
* channels + c] = (y < height) ? d_in[((y * width) + (x + 1)) *
channels + c] : 0;
        }
    }

    if (threadIdx.y == 0 && y > 0) {
        for (int c = 0; c < channels; ++c) {
            shared_mem[tx * channels + c] = (x < width) ? d_in[((y
- 1) * width + x) * channels + c] : 0;
        }
    }

    if (threadIdx.y == blockDim.y - 1 && y < height - 1) {
        for (int c = 0; c < channels; ++c) {
            shared_mem[((blockDim.y + 1) * (blockDim.x + 2) + tx)
* channels + c] = (x < width) ? d_in[((y + 1) * width + x) *
channels + c] : 0;
        }
    }

    __syncthreads();

    // Apply the box blur
    if (x < width && y < height) {
        for (int c = 0; c < channels; ++c) {
            int sum = 0;
            for (int dy = -1; dy <= 1; ++dy) {
                for (int dx = -1; dx <= 1; ++dx) {
                    int sharedIdx = ((ty + dy) * (blockDim.x + 2)
+ (tx + dx)) * channels + c;
```

```
                    sum += shared_mem[sharedIdx];
                }
            }
            int outputIdx = (y * width + x) * channels + c;
            d_out[outputIdx] = sum / 9;
        }
    }
}
```

This kernel function uses shared memory to apply a 3x3 box blur filter on the input image. It loads the central pixel and its neighboring pixels into a shared memory block and calculates the average RGB values to blur the pixel at (x, y). Edge cases are handled by checking bounds before accessing neighboring pixels.

Returning Blurred Image Data to the CPU

```
cudaMemcpy(h_out, d_out, width * height * channels,
cudaMemcpyDeviceToHost);
```

After processing, the blurred image data is copied back to the host using cudaMemcpy. The output data, now containing the blurred image, is stored in the host memory.

Outputting the Blurred Image as a File

```
void writePNG(const char* filename, unsigned char* data, int
width, int height, int channels) {
    FILE* fp = fopen(filename, "wb");
    if (!fp) {
        perror("Error opening file");
        return;
    }

    png_structp png =
png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png) {
        fclose(fp);
        return;
    }

    png_infop info = png_create_info_struct(png);
    if (!info) {
        png_destroy_write_struct(&png, NULL);
        fclose(fp);
        return;
    }
```

```
    if (setjmp(png_jmpbuf(png))) {
        png_destroy_write_struct(&png, &info);
        fclose(fp);
        return;
    }

    png_init_io(png, fp);

    png_set_IHDR(png, info, width, height, 8, PNG_COLOR_TYPE_RGB,
PNG_INTERLACE_NONE,
                 PNG_COMPRESSION_TYPE_DEFAULT,
PNG_FILTER_TYPE_DEFAULT);

    png_bytep* row_pointers = (png_bytep*)malloc(sizeof(png_bytep)
* height);
    for (int y = 0; y < height; ++y) {
        row_pointers[y] = data + y * width * channels;
    }

    png_set_rows(png, info, row_pointers);
    png_write_png(png, info, PNG_TRANSFORM_IDENTITY, NULL);

    free(row_pointers);
    png_destroy_write_struct(&png, &info);
    fclose(fp);
}
```

This function writes the final blurred image data back to a PNG file using the libpng library. It sets up the appropriate image properties such as width, height, and color channels, then writes the data row by row.