**Documentation – Roee Elkoubi**

**Introduction:** The project is designed to calculate the matching between Objects and Pictures. It deals with a set of Pictures and Objects of different sizes. Each member of the matrix represents a "color," and the range of possible colors is [1, 100]. For each pair of overlapping members, the difference is calculated, and the total difference is defined as an average of all relative differences for all overlapping members. We will call it Matching (I, J).

**Inputs:** The input for the project is a set of Pictures and Objects of different sizes. As input, we got – Matching value, number of pictures, picture id, picture dimension, and N line of members of the picture row by row. We got the same things for the objects.

**Output:** The output of the project is the Matching (I, J) for each Position (I, J) of the Object in the Picture. This project finds if the given picture contains at least three objects from the given object set and writes the result to the output.txt file.

**Something Important:**

My project folder includes 4 folders, CUDA which is the final result of the project, MPIOpenMP (implementation with both things), MPIOpenMP with task construct implementation, and sequential implementation. All those implementations help me to provide the final result as implement MPI + OpenMP + CUDA. All those folders include all the files they need to run, such as input file, output file, c files, h files, run on 2 computers txt file and make file that help us to build run and compile all those files.

In this documentation, I will describe the final result implement MPI + OpenMP + CUDA. My documentation describes the final result of the project as implementing MPI + OpenMP + CUDA but it was important for me to describe the whole process on the way to the final result, before I found a way to implement CUDA also I was implemented MPI + OpenMP version, and MPI + OpenMP with task Construct and it's important for me to include those things also.

**General Description and Algorithms:**

**Files –**

**input.txt -** This is an input file that serves as input for the program.

**makefile -** This is a configuration file for build and run.

**cFunction.c -** This is a C source code file that contains the implementation of the general function on c that the whole program uses.

**cudaFunction.cu –** the file is a CUDA file that contains code for performing computations on a GPU using the CUDA platform.

**output.txt -** This is an output file that contains the program's output and the results of the matching.

**run2computers -** This is the file that allows the program to be run on two different computers.

**Prototype.h –** This file contains function prototypes, data structures, macros, and other declarations that are needed in multiple source code files, including libraries and different definitions.

**Main.c** – This file manages the main flow of the program, initializing variables and data structures Reading input from the user or a file Calling functions from other source code files to perform specific tasks Performing calculations or processing data Outputting results or displaying information to the user Handling errors or exceptions that occur during the program.

**Algorithms Description:**

**cFunction.c:**

1) The function 'readMatchingValue' reads the first value from a file, which should be a matching value, and stores it in the variable pointed to by 'matchingValue'. The function returns an integer indicating whether the read was successful or not.

2) The function 'read element' reads an element from a file and creates an array of structures that describe the element. The function takes a file pointer and a pointer to an integer that will be used to store the number of elements read. The function returns a pointer to the array of structures.

3) The function 'SendingStructData' takes a structure and an integer representing the destination process and sends the structure to that process via MPI.

4) The function 'RecievingStructData' takes an integer representing the sourcing process and a pointer to an MPI\_Status struct and receives a structure from that process. The function returns a pointer to the received structure.

5) The function 'FreeElementFromFile' takes a pointer to an array of structures and an integer representing the number of elements in the array, and frees the memory allocated to it.

6) The function 'resultMatrixToFile' takes an integer array representing the result matrix, a file pointer, an integer 'pictureId', and an integer 'ObjectCounter', and writes the result matrix to the specified file.

**cudaFunction.cu:**

**1) handleCudaError:** A function that helps to handle errors that can occur during the execution of CUDA code. It takes two parameters, the error code and the line number where the error occurred and provides an error message that can help identify and fix the issue.

**2) matrix2DTo1D:** A function that takes a 2D matrix and converts it to a 1D array

**3) matchingCuda:** A CUDA kernel function designed to perform matching between a picture and an object using parallelism to improve performance. It takes in several parameters and uses the GPU threads to iterate over possible matching locations and calculate the matching between the picture and object at each location.

**4) matchingOnGPU:** A function that performs matching between a picture and multiple objects using GPU parallelism. It first converts the picture and all objects to 1D arrays and copies them to the GPU memory. It then launches the matchingCuda kernel function with the appropriate parameters and waits for it to complete. Finally, it copies the output arrays from the GPU memory back to the host memory.

**Prototype.h –**

The first structure is named ElementFromFile and represents an element from a file, which could be either a picture or an object, along with its attributes. This structure has three members:

**Id:** an integer that represents the ID of the element.

**Dim:** an integer that represents the dimension of the element.

**matrixElement:** a pointer to a two-dimensional integer array that represents the matrix of the element.

The second structure is named tags and represents different tags that can be associated with the elements. This structure has eight members:

**tagNumberOfObject:** an integer that represents the number of objects message.

**tagMatchingValue:** an integer that represents the matching value message.

**tagObject:** an integer that represents the type of object message.

**tagPicture:** an integer that represents the type of picture message.

**tagResultArray:** an integer that represents the result matrix message.

**tagPictureId:** an integer that represents the ID of the picture message.

**tagNumberOfObjectResult:** an integer that represents the number of objects matching in the result message.

**tagNumberOfPictures:** an integer that represents the number of pictures message.

**Explanation of what and how the problem was parallelized:**

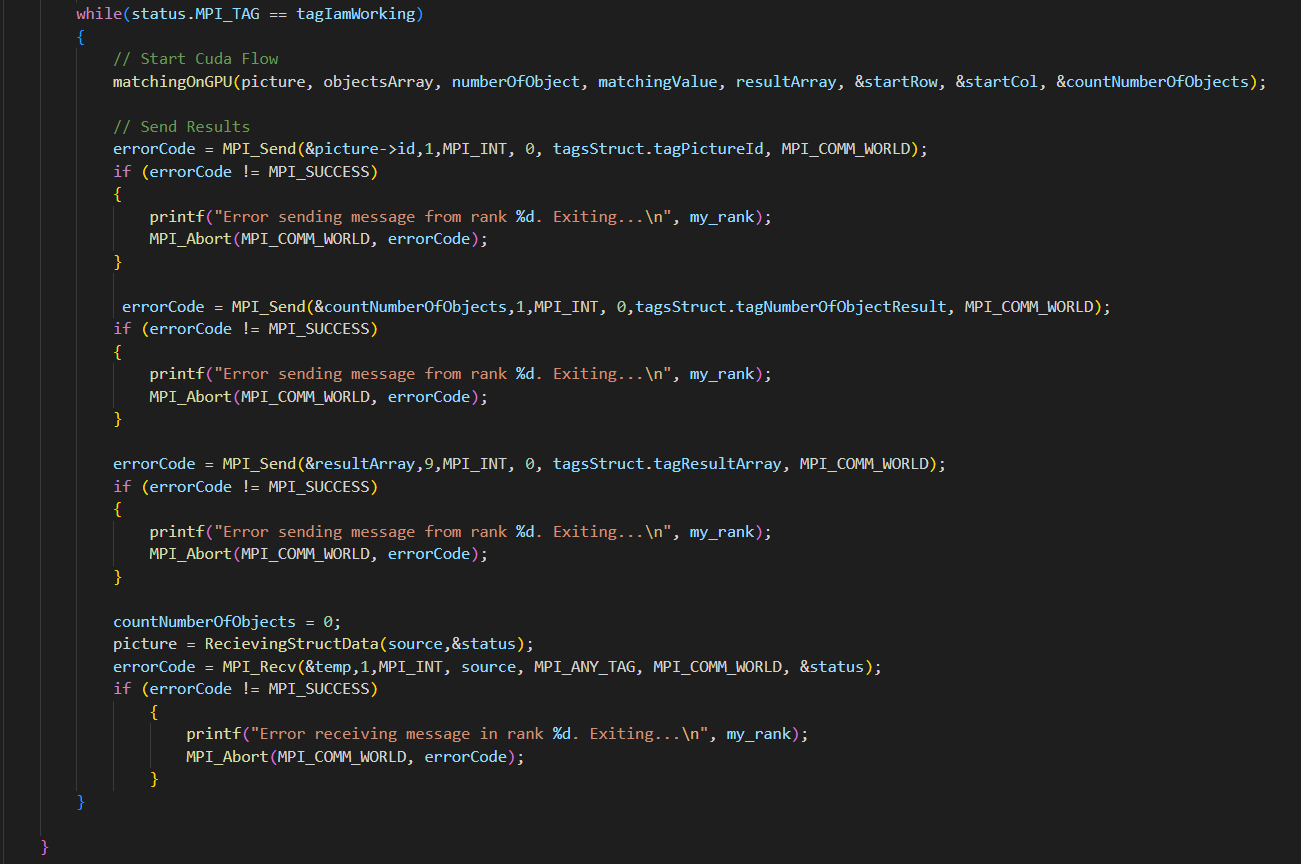
In general, I decided to choose work with a dynamic work method to parallel the program. In the dynamic work method, the master process initially distributes a small set of tasks to all available worker processes. As each worker process completes a task, it requests the master process for a new task. The master process then assigns a new task to the requesting worker process from a global list of unassigned tasks. This process continues until all tasks are completed.

The MPI library helps us to provide a dynamic work method that enables us to communicate with our processes in a variety of ways. This method allows us to easily send and receive messages, as well as perform other actions. With this feature, we can talk to our processes, exchange information, and accomplish a multitude of tasks.

Using the MPI library in my project has allowed for efficient parallel computing by breaking down complex tasks into smaller pieces (each process get a picture) and enabling seamless communication between processes. The library has significantly improved the efficiency and performance of my project.

In my program, the master processes manage the dynamic work method. First, he read the whole data from the input.txt file and allocated and organized the whole data before its sent. The master processes send all the necessary general data that all processes needed such as matching value, number of pictures, number of objects the whole objects and etc.

Something important that the master sends is the working tag - we have 2 options working tag / no more work tag, these 2 tags help us control the workflow until all the works are done (in the beginning all processes that get a picture get also working tag to start work). After that, we made the first division depending on 2 situations that the master processes deal with if the number of running processes is larger than the number of pictures and when the number of running processes is smaller than the number of pictures. now each process has the general data from the file, a picture, and all the objects that the master processes read from the file. After each process receives the whole data, he is ready to start work and look for the matching. To keep parallel the program we use the OpenMP library, we use that on 2 function SendingStructData, we use the #pragma omp for statement to parallelize the inner loop that copies the values of the 2D matrix element.matrixElement into a 1D array matrixFrom2Dto1D. By using OpenMP to parallelize this loop, the iterations can be executed in parallel on multiple threads, potentially reducing the overall execution time of the program. The second function recievingStructData using OpenMP to parallelize the loop that copies the values of a 1D array into a 2D matrix.



The while loop is checking the status of an MPI tag, which is a allow us to control the workflow and keep sending tasks or not. The loop continues as long as the tag value is equal to "tagIamWorking". Within the loop, each process calls to the matching function, a function that performs a matching algorithm between an input picture and multiple objects using CUDA. The input parameters are:

**picture:** a pointer to an ElementFromFile struct, which contains the input picture.

**objectsArr:** an array of pointers to ElementFromFile structs, which represent the objects to match against the input picture.

**numOfObjects:** an integer that represents the number of objects in objectsArr. matchingValue: a double value that represents the threshold for matching between the input picture and objects.

**resultArray:** an array to hold the results of the matching process.

**rowIndex:** an array to hold the row index of the match for each object.

**colIndex:** an array to hold the column index of the match for each object.

**numOfMatchingInObject** a pointer to hold the number of matches found for each object. The function allocates memory for the input picture, object matrices, and result arrays on both the host and the device using CUDA memory allocation functions. It then iterates over each object, allocates memory for the row and column index arrays and the result value on the device using CUDA memory allocation functions. The function then copies the input picture and the current object from the host to the device using the cudaMemcpy() function. After that, the function launches the matchingCuda kernel, which compares the input picture and the current object to find the matching. If the match is found, the kernel sets the corresponding element in the resultOnDevice array to 1. The function then copies the result value from the device to the host using the cudaMemcpy() function. Finally, the function checks the result value to see if there was a match, and if so, copies the row and column indices from the device to the host using the cudaMemcpy() function. It then updates the resultArray, rowIndex, and colIndex arrays with the matching information for the current object and updates the numOfMatchingInObject array with the number of matches found for the current object. The function then frees the memory allocated on the host and the device using the free() and cudaFree() functions. Using CUDA here is good because it allows for parallel computation of the matching process, which can greatly reduce the time needed for the algorithm to run. The code above is a CUDA kernel that runs on the GPU, which means that each thread is assigned to compute a specific part, making it faster than the sequential processing on a CPU.

**Deep look on matching function:**

This is a CUDA kernel function that performs image matching between an object and a larger picture. The function takes as input the pictureArr and objectArr arrays, which represent the color values of the picture and the object, respectively, along with their respective dimensions pictureDim and objectDim. It also takes as input a matchingValue threshold, which determines whether a match is found, and pointers to the rowIndex, colIndex, and result output variables. The kernel function iterates over all possible positions of the object within the picture, calculating a matching score for each position. The matching score is computed as the average difference in color values between the object and the corresponding region of the picture. The function first computes the starting position i and j of the object in the picture based on the thread index threadIdx and block index blockIdx and the blockDim value. It then checks if the object fits within the picture boundaries at this position, and if so, iterates over all color in the object, comparing them to the corresponding colors in the picture. For each color, the function calculates the difference between the color values of the object and the corresponding color value in the picture. This difference is then normalized by dividing it by the value of the corresponding color in the picture, and the absolute value of this division is added to a running total of matching values. After computing the matching values for all colors in the object, the kernel function divides the sum of matching values by the total number of colors in the object to obtain the average matching value for the object. Finally, if the average matching value is less than the threshold matchingValue, the kernel function sets the output variables rowIndex and colIndex to the current row and column index of the object in the picture and sets result to 1 to indicate that a match has been found.

After we finally found matching results, the process sends the results back to the master process (process 0), and process 0 as a manager of the Dynamic method, decides if to keep sending tasks to the process.

All the other process waiting to get more tasks (if there is) and a tag if they need to keep working or there is no more to do. This workflow keeps going until all the tasks are done. Finally, the master process writes the results to the file.

**Rational of choosing the specific architecture:**

The choice of a specific architecture depends on several factors, such as the problem's characteristics, the available resources, and the performance requirements. In our case, using Open MP + MPI + CUDA with dynamic work distribution, meets the following requirements:

1**) Parallelization potential:** The task of calculating the Matching(I, J) values for all possible positions of an Object in a Picture involves many independent calculations. Therefore, it has a high potential for parallelization.

2) **Scalability:** The size of the Pictures and Objects can vary, and the number of calculations can increase significantly with the size. Using MPI + OpenMP + CUDA with dynamic work distribution allows me to distribute the work evenly among the available processes and each process distributes the work with strong GPU power compute, which can help achieve good scalability.

3) **Flexibility:** The dynamic work distribution method allows me to adjust the workload based on the availability of resources and the workload distribution efficiency. This approach can help me achieve optimal performance by adapting to the current state of the computing system and each process.

4) **Load Balancing -** The dynamic work distribution method can be good for load balancing because it allows me to distribute the workload among available processes as they become available. This approach ensures that all processes are utilized efficiently, which can help achieve better load balancing**.**

**Complexity evaluation – In General**

The time complexity of the matchingCuda kernel function: The kernel function iterates over each element in the picture array, so the time complexity is O(n^2) where n is the dimension of the picture array. Within each iteration, the kernel function also iterates over each element in the object array (which has dimension m), so the time complexity within each iteration is O(m^2). Therefore, the overall time complexity of the matchingCuda kernel function is O(n^2 \* m^2).

The time complexity of the matchingOnGPU function: The matchingOnGPU function performs a loop over the objects array (which has length k), and calls the matchingCuda kernel function once for each object. Therefore, the time complexity of the matchingOnGPU function is O(k \* n^2 \* m^2).