Content-based Image Search

Parallelization using the CUDA library

Group members:

Faridreza Momtaz Zandi

Mahya EhsaniMehr

Ali Mojahed

Spring 2023

::General description::

As we have seen in the previous phase, by converting the main project and rewriting it in CPP language, we were able to implement the image search program based on the content and run the program in serial and parallel form with a different number of threads and get a full report of it. Now, we will rewrite the same CPP program to CUDA that we chose CPP CUDA and implement all the features mentioned in the previous phase in this language.

There are many differences between the final program in Coda language, which has the .cu format, and the CPP file, despite the same functionality, which we will find out further by examining this file. The first problem we faced in doing this project was Coda's monopoly on NVIDIA graphics cards it forced us to use a new system and install all the necessary libraries from the beginning, but fortunately there were no problems during the installation process, and after installation, All the libraries were able to use both OpenMP and Coda, both of which use OpenCV, a special CPP image processing library.

In this phase, to better show the power difference in Cuda, we will use the complete dataset that contains 5000 images and the difference between the results obtained in the serial section and OpenMP with eight threads in the section The results we see will be because of this.

::Code Review::

Together, we will review the program code and the use of each section:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <dirent.h>
#include <opencv2/opencv.hpp>
#include <chrono>
```

This code includes several libraries for I/O operations, image processing and timing. Considering that we are writing the program in Coda language, there is no need to add a special library.

```
__global__
void computeDistanceKernel(double* datasetMean, double* queryMean, double* datasetMedian, double* queryMedian, double* datasetStdDev,
    double* queryStdDev, double* datasetHuMoments, double* queryHuMoments, double* datasetHistogram, double* queryHistogram,
    double* distances, int datasetSize)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < datasetSize)
    {
        double distance = 0.0;

        distance += std::pow(datasetMean[index] - queryMean[0], 2);
        distance += std::pow(datasetMedian[index] - queryMedian[0], 2);
        distance += std::pow(datasetStdDev[index] - queryStdDev[0], 2);

        for (int i = 0; i < 7; ++i)
        {
            distance += std::pow(datasetHuMoments[index * 7 + i] - queryHuMoments[i], 2);
        }

        for (int i = 0; i < 256; ++i)
        {
            distance += std::pow(datasetHistogram[index * 256 + i] - queryHistogram[i], 2);
        }

        distances[index] = std::sqrt(distance);
    }
}
```

This section of code defines a CUDA kernel function computeDistanceKernel that runs on the GPU. Calculates the distance between the input image and each image in the dataset using various image features.

The kernel function takes several arrays representing the features of the query dataset and image, along with an array to store the calculated distances.

Each thread in the GPU is responsible for calculating the distance of an image from the dataset.

The calculated distances are stored in the distances array.

```
// Function to resize an image to a given size
cv::Mat resizeImage(const cv::Mat& image, int width, int height)
{
    cv::Mat resizedImage;
    cv::resize(image, resizedImage, cv::Size(width, height));
    return resizedImage;
}
```

resizeImage: resizes an image to the specified width and length.

```
// Function to compute the mean value of an image
double computeMean(const cv::Mat& image)
{
    cv::Scalar meanVal = cv::mean(image);
    return meanVal[0];
}

```

computeMean: calculates the mean value of an image.

```
// Function to compute the median value of an image
double computeMedian(const cv::Mat& image)
{
    cv::Mat sortedImage;
    cv::sort(image, sortedImage, cv::SORT_EVERY_COLUMN | cv::SORT_ASCENDING);
    int totalPixels = image.rows * image.cols;

    double medianValue;
    if (totalPixels % 2 == 0)
    {
        int index1 = (totalPixels / 2) - 1;
        int index2 = index1 + 1;
        medianValue = (sortedImage.at<uchar>(index1) + sortedImage.at<uchar>(index2)) / 2.0;
    }
    else
    {
        int index = (totalPixels / 2);
        medianValue = sortedImage.at<uchar>(index);
    }

    return medianValue;
}
```

computeMedian: calculates the median value of an image.

```cpp
// Function to compute the histogram of an image
cv::Mat computeHistogram(const cv::Mat& image)
{
    int numBins = 256; // Number of bins for the histogram
    int histSize[] = { numBins };
    float range[] = { 0, 256 };
    const float* ranges[] = { range };
    int channels[] = { 0 }; // Compute histogram only for the first channel (grayscale image)

    cv::Mat histogram;
    cv::calcHist(&image, 1, channels, cv::Mat(), histogram, 1, histSize, ranges);

    return histogram;
}

```

computeHistogram: calculates the histogram of an image.

```cpp

// Function to compute the standard deviation of an image
double computeStandardDeviation(const cv::Mat& image)
{
    double mean = computeMean(image);
    cv::Scalar stddev;
    cv::meanStdDev(image, cv::noArray(), stddev);
    return stddev[0];
}

```

computeStandardDeviation: Calculates the standard deviation of an image.

```cpp
// Function to compute the Hu moments of an image
cv::Mat computeHuMoments(const cv::Mat& image)
{
    cv::Mat moments;
    cv::HuMoments(cv::moments(image), moments);
    return moments;
}

```

computeHuMoments: Computes the Hu moments of an image.

```cpp
// Function to load the dataset of images
void loadDataset(const std::string& datasetPath, std::vector<cv::Mat>& datasetImages)
{
    datasetImages.clear();

    // Open the directory
    DIR* dir = opendir(datasetPath.c_str());
    if (dir == nullptr)
    {
        std::cerr << "Error opening directory: " << datasetPath << std::endl;
        return;
    }

    // Read the directory entries
    struct dirent* entry;
    while ((entry = readdir(dir)) != nullptr)
    {
        std::string filename = entry->d_name;

        // Skip directories and hidden files
        if (entry->d_type == DT_DIR || filename[0] == '.')
            continue;

        std::string imagePath = datasetPath + "/" + filename;

        // Load the image and add it to the dataset
        cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR);
        if (image.empty())
        {
            std::cerr << "Error loading image: " << imagePath << std::endl;
            continue;
        }

        datasetImages.push_back(image);
    }

    // Close the directory
    closedir(dir);
}
```

loadDataset: Loads a dataset of images from a specified directory.

```cpp
int main()
{
    auto start = high_resolution_clock::now();
    std::string datasetPath = "./dataset/images/";
    std::string queryImagePath = "000000124442.jpg";

    // Load the dataset images
    std::vector<cv::Mat> datasetImages;
    loadDataset(datasetPath, datasetImages);

    // Load the query image
    cv::Mat queryImage = cv::imread(queryImagePath, cv::IMREAD_COLOR);
    if (queryImage.empty())
    {
        std::cerr << "Error loading query image: " << queryImagePath << std::endl;
        return 1;
    }

    // Convert the query image to grayscale
    cv::Mat queryImageGray;
    cv::cvtColor(queryImage, queryImageGray, cv::COLOR_BGR2GRAY);

    // Resize the query image
    cv::Mat resizedQueryImage = resizeImage(queryImageGray, 128, 128);

    // Compute the features for the query image
    double queryMean = computeMean(resizedQueryImage);
    double queryMedian = computeMedian(resizedQueryImage);
    double queryStdDev = computeStandardDeviation(resizedQueryImage);
    cv::Mat queryHuMoments = computeHuMoments(resizedQueryImage);
    cv::Mat queryHistogram = computeHistogram(resizedQueryImage);
```

The program execution starts in the main function.

It measures execution time using high_resolution_clock from the std::chrono library.

Defines the directory path of the dataset and the path of the input image.

Calls the loadDataset function to load the dataset images into a vector.

Reads and loads the input image.

Converts the input image to grayscale and resizes it to a fixed size (128x128).

Calculates various features for the query image, such as mean, median, standard deviation, Hu modes and histogram.

```
// Allocate GPU memory for dataset features and distances
int datasetSize = datasetImages.size();
double* deviceDatasetMean;
double* deviceQueryMean;
double* deviceDatasetMedian;
double* deviceQueryMedian;
double* deviceDatasetStdDev;
double* deviceQueryStdDev;
double* deviceDatasetHuMoments;
double* deviceQueryHuMoments;
double* deviceDatasetHistogram;
double* deviceQueryHistogram;
double* deviceDistances;
cudaMalloc(&deviceDatasetMean, datasetSize * sizeof(double));
cudaMalloc(&deviceQueryMean, sizeof(double));
cudaMalloc(&deviceDatasetMedian, datasetSize * sizeof(double));
cudaMalloc(&deviceQueryMedian, sizeof(double));
cudaMalloc(&deviceDatasetStdDev, datasetSize * sizeof(double));
cudaMalloc(&deviceQueryStdDev, sizeof(double));
cudaMalloc(&deviceDatasetHuMoments, datasetSize * 7 * sizeof(double));
cudaMalloc(&deviceQueryHuMoments, 7 * sizeof(double));
cudaMalloc(&deviceDatasetHistogram, datasetSize * 256 * sizeof(double));
cudaMalloc(&deviceQueryHistogram, 256 * sizeof(double));
cudaMalloc(&deviceDistances, datasetSize * sizeof(double));
auto stop = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(stop - start);
std::cout << "GPU Memory allocation time: " << duration.count() << " milliseconds" << std::endl;

auto start1 = high_resolution_clock::now();
```

Allocates GPU memory to store dataset features, query features, and intervals.

It calculates and prints the time to put the dataset on the GPU.

```
    // Copy dataset features to GPU memory
    for (int i = 0; i < datasetSize; ++i)
    {
        cv::Mat resizedImage;
        cv::Mat imageGray;
        cv::cvtColor(datasetImages[i], imageGray, cv::COLOR_BGR2GRAY);
        resizedImage = resizeImage(imageGray, 128, 128);

        double imageMean = computeMean(resizedImage);
        double imageMedian = computeMedian(resizedImage);
        double imageStdDev = computeStandardDeviation(resizedImage);
        cv::Mat imageHuMoments = computeHuMoments(resizedImage);
        cv::Mat imageHistogram = computeHistogram(resizedImage);

        cudaMemcpy(deviceDatasetMean + i, &imageMean, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(deviceDatasetMedian + i, &imageMedian, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(deviceDatasetStdDev + i, &imageStdDev, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(deviceDatasetHuMoments + (i * 7), imageHuMoments.ptr<double>(), 7 * sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(deviceDatasetHistogram + (i * 256), imageHistogram.ptr<double>(), 256 * sizeof(double), cudaMemcpyHostToDevice);
    }

    // Copy query features to GPU memory
    cudaMemcpy(deviceQueryMean, &queryMean, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceQueryMedian, &queryMedian, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceQueryStdDev, &queryStdDev, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceQueryHuMoments, queryHuMoments.ptr<double>(), 7 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceQueryHistogram, queryHistogram.ptr<double>(), 256 * sizeof(double), cudaMemcpyHostToDevice);
```

Copies the query dataset and features from the host to the GPU memory.

We transfer the dataset to the GPU in the form of chunks.

```
    // Set the number of threads per block and the number of blocks
    int threadsPerBlock = 256;
    int numBlocks = (datasetSize + threadsPerBlock - 1) / threadsPerBlock;

    // Launch the kernel to compute distances
    computeDistanceKernel<<<numBlocks, threadsPerBlock>>>(deviceDatasetMean, deviceQueryMean, deviceDatasetMedian,
        deviceQueryMedian, deviceDatasetStdDev, deviceQueryStdDev, deviceDatasetHuMoments, deviceQueryHuMoments,
        deviceDatasetHistogram, deviceQueryHistogram, deviceDistances, datasetSize);

    // Copy the distances from GPU memory to the host
    double* distances = new double[datasetSize];
    cudaMemcpy(distances, deviceDistances, datasetSize * sizeof(double), cudaMemcpyDeviceToHost);

    std::vector<DistanceIndexPair> distanceIndexPairs;
    for (int i = 0; i < datasetSize; ++i) {
        DistanceIndexPair pair;
        pair.distance = distances[i];
        pair.index = i;
        distanceIndexPairs.push_back(pair);
    }

    std::sort(distanceIndexPairs.begin(), distanceIndexPairs.end(), [](const DistanceIndexPair& a, const DistanceIndexPair& b) {
        return a.distance < b.distance;
    });

    std::cout << "20 lowest distances:" << std::endl;
    for (int i = 0; i < 20; ++i) {
        std::cout << "Index: " << distanceIndexPairs[i].index << ", Distance: " << distanceIndexPairs[i].distance << std::endl;
    }
```

Specifies the number of threads per block and the number of blocks that should execute the CUDA kernel.

Runs computeDistanceKernel on GPU to calculate distances.

Copies the GPU memory spaces to the host.

Sorts the intervals along with their respective indices to find the closest ones.

Prints the 20 shortest distances and their corresponding indices.

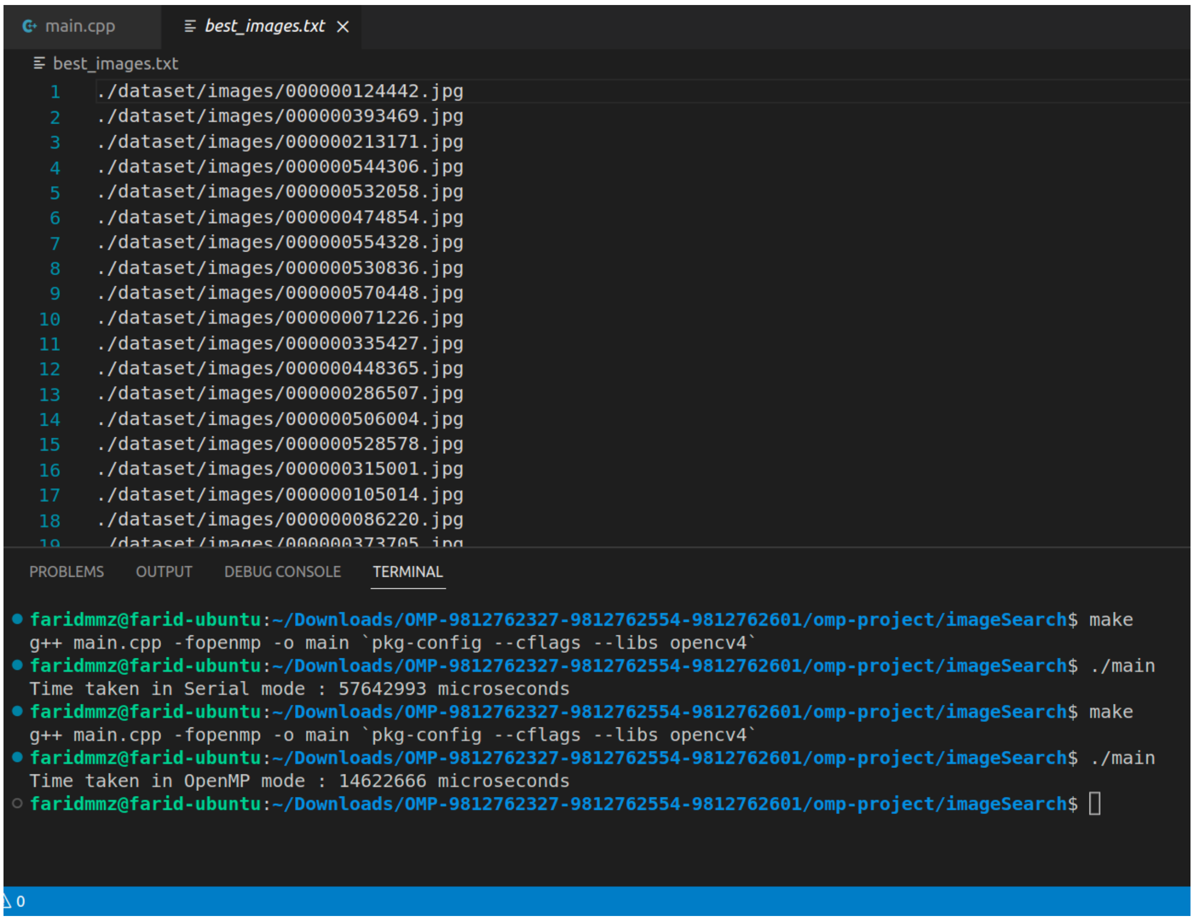
```

    // Free GPU memory
    cudaFree(deviceDatasetMean);
    cudaFree(deviceQueryMean);
    cudaFree(deviceDatasetMedian);
    cudaFree(deviceQueryMedian);
    cudaFree(deviceDatasetStdDev);
    cudaFree(deviceQueryStdDev);
    cudaFree(deviceDatasetHuMoments);
    cudaFree(deviceQueryHuMoments);
    cudaFree(deviceDatasetHistogram);
    cudaFree(deviceQueryHistogram);
    cudaFree(deviceDistances);

    // Free host memory
    delete[] distances;

    auto stop1 = high_resolution_clock::now();
    auto duration1 = duration_cast<milliseconds>(stop1 - start1);
    std::cout << "Execution time: " << duration1.count() << " milliseconds" << std::endl;


    return 0;
}
```

Frees GPU and host memory.

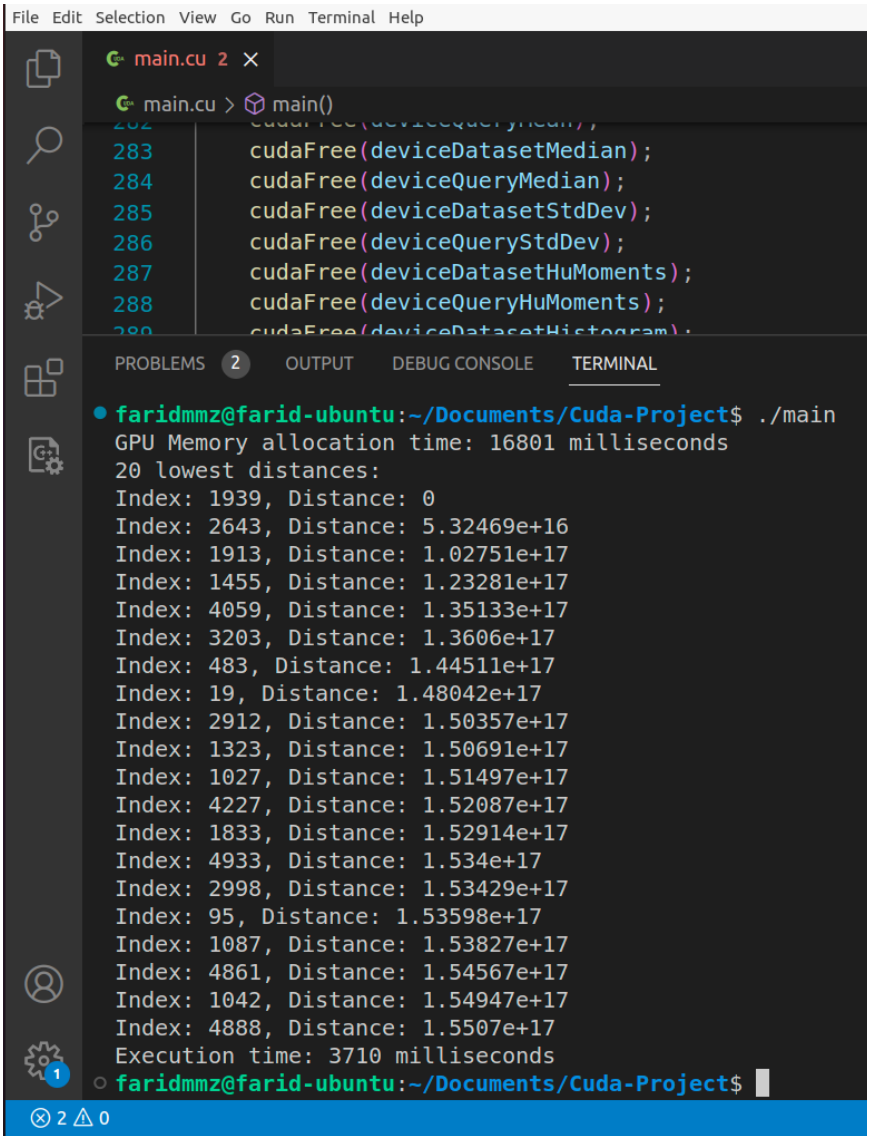
It measures the execution time of the calculation and prints it.

::Results::

First, we see the running time of the program with the complete dataset and including 5000 photos in serial mode and using OpenMP at its best, i.e. eight threads below. In serial mode, the program takes about 57 seconds, and in OpenMP mode, using eight threads, only 14 seconds.



Now, we run the new program on the GPU and check the total time, calculation time, and data set time on the GPU:



As we can see, although the total execution time of the program is about 20 seconds, a large part of this time was spent on placing our large dataset on the GPU memory, and the calculations took very little time, i.e. only 3.7 seconds. This time is the best and if we send the data in other ways, the result would be weaker.

Below we see a table comparing these times:

modes	Serial	OpenMP 8Thread	CUDA memaloc	CUDA Execution	CUDA Total
records	57.6	14.6	16.8	3.7	**20.5**

By examining these numbers, we can conclude that due to the increase in speed below in Coda, it is very economical to use it, of course, on the condition that we want to do a lot of calculations, otherwise, if you have a large dataset, but you don't have that many calculations, OpenMP may be a better and more suitable tool.

::Hardware specifications::

The machine on which this code was executed and the results obtained on Ubuntu 20.04 operating system and hardware are as follows:

Machine model: MSI GE62 6QD

Processor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2601 Mhz, 4 Core(s), 8 Logical Processor(s)

Physical cores: 4 pcs

Supported threads: 8 pcs

RAM memory: 8 GB

Graphic Card: GTX 960M 4 / 2GB GDDR5