
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 Cuda 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 Cuda'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 Cuda, 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:

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <dirent.h>
5  #include <opencv2/opencv.hpp>
6  #include <chrono>
```

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

```
12  __global__
13  void computeDistanceKernel(double* datasetMean, double* queryMean, double* datasetMedian, double* queryMedian, double* datasetStdDev,
14  double* queryStdDev, double* datasetHuMoments, double* queryHuMoments, double* datasetHistogram, double* queryHistogram,
15  double* distances, int datasetSize)
16  {
17  int index = blockIdx.x * blockDim.x + threadIdx.x;
18
19  if (index < datasetSize)
20  {
21  double distance = 0.0;
22
23  distance += std::pow(datasetMean[index] - queryMean[0], 2);
24  distance += std::pow(datasetMedian[index] - queryMedian[0], 2);
25  distance += std::pow(datasetStdDev[index] - queryStdDev[0], 2);
26
27  for (int i = 0; i < 7; ++i)
28  {
29  distance += std::pow(datasetHuMoments[index * 7 + i] - queryHuMoments[i], 2);
30  }
31
32  for (int i = 0; i < 256; ++i)
33  {
34  distance += std::pow(datasetHistogram[index * 256 + i] - queryHistogram[i], 2);
35  }
36
37  distances[index] = std::sqrt(distance);
38  }
39  }
```

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.

```
11 // Function to resize an image to a given size
12 cv::Mat resizeImage(const cv::Mat& image, int width, int height)
13 {
14     cv::Mat resizedImage;
15     cv::resize(image, resizedImage, cv::Size(width, height));
16     return resizedImage;
17 }
```

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

```
19 // Function to compute the mean value of an image
20 double computeMean(const cv::Mat& image)
21 {
22     cv::Scalar meanVal = cv::mean(image);
23     return meanVal[0];
24 }
25
```

computeMean: calculates the mean value of an image.

```
26 // Function to compute the median value of an image
27 double computeMedian(const cv::Mat& image)
28 {
29     cv::Mat sortedImage;
30     cv::sort(image, sortedImage, cv::SORT_EVERY_COLUMN | cv::SORT_ASCENDING);
31     int totalPixels = image.rows * image.cols;
32
33     double medianValue;
34     if (totalPixels % 2 == 0)
35     {
36         int index1 = (totalPixels / 2) - 1;
37         int index2 = index1 + 1;
38         medianValue = (sortedImage.at<uchar>(index1) + sortedImage.at<uchar>(index2)) / 2.0;
39     }
40     else
41     {
42         int index = (totalPixels / 2);
43         medianValue = sortedImage.at<uchar>(index);
44     }
45
46     return medianValue;
47 }
48
```

computeMedian: calculates the median value of an image.

```
49 // Function to compute the histogram of an image
50 cv::Mat computeHistogram(const cv::Mat& image)
51 {
52     int numBins = 256; // Number of bins for the histogram
53     int histSize[] = { numBins };
54     float range[] = { 0, 256 };
55     const float* ranges[] = { range };
56     int channels[] = { 0 }; // Compute histogram only for the first channel (grayscale image)
57
58     cv::Mat histogram;
59     cv::calcHist(&image, 1, channels, cv::Mat(), histogram, 1, histSize, ranges);
60
61     return histogram;
62 }
63
```

computeHistogram: calculates the histogram of an image.

```
93
94 // Function to compute the standard deviation of an image
95 double computeStandardDeviation(const cv::Mat& image)
96 {
97     double mean = computeMean(image);
98     cv::Scalar stddev;
99     cv::meanStdDev(image, cv::noArray(), stddev);
100     return stddev[0];
101 }
102
```

computeStandardDeviation: Calculates the standard deviation of an image.

```
103 // Function to compute the Hu moments of an image
104 cv::Mat computeHuMoments(const cv::Mat& image)
105 {
106     cv::Mat moments;
107     cv::HuMoments(cv::moments(image), moments);
108     return moments;
109 }
110
```

computeHuMoments: Computes the Hu moments of an image.

```

111 // Function to load the dataset of images
112 void loadDataset(const std::string& datasetPath, std::vector<cv::Mat>& datasetImages)
113 {
114     datasetImages.clear();
115
116     // Open the directory
117     DIR* dir = opendir(datasetPath.c_str());
118     if (dir == nullptr)
119     {
120         std::cerr << "Error opening directory: " << datasetPath << std::endl;
121         return;
122     }
123
124     // Read the directory entries
125     struct dirent* entry;
126     while ((entry = readdir(dir)) != nullptr)
127     {
128         std::string filename = entry->d_name;
129

```

```

129
130         // Skip directories and hidden files
131         if (entry->d_type == DT_DIR || filename[0] == '.')
132             continue;
133
134         std::string imagePath = datasetPath + "/" + filename;
135
136         // Load the image and add it to the dataset
137         cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR);
138         if (image.empty())
139         {
140             std::cerr << "Error loading image: " << imagePath << std::endl;
141             continue;
142         }
143
144         datasetImages.push_back(image);
145     }
146
147     // Close the directory
148     closedir(dir);
149 }
150

```

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

```

156 int main()
157 {
158     auto start = high_resolution_clock::now();
159     std::string datasetPath = "./dataset/images/";
160     std::string queryImagePath = "000000124442.jpg";
161
162     // Load the dataset images
163     std::vector<cv::Mat> datasetImages;
164     loadDataset(datasetPath, datasetImages);
165
166     // Load the query image
167     cv::Mat queryImage = cv::imread(queryImagePath, cv::IMREAD_COLOR);
168     if (queryImage.empty())
169     {
170         std::cerr << "Error loading query image: " << queryImagePath << std::endl;
171         return 1;
172     }
173
174     // Convert the query image to grayscale
175     cv::Mat queryImageGray;
176     cv::cvtColor(queryImage, queryImageGray, cv::COLOR_BGR2GRAY);
177
178     // Resize the query image
179     cv::Mat resizedQueryImage = resizeImage(queryImageGray, 128, 128);
180
181     // Compute the features for the query image
182     double queryMean = computeMean(resizedQueryImage);
183     double queryMedian = computeMedian(resizedQueryImage);
184     double queryStdDev = computeStandardDeviation(resizedQueryImage);
185     cv::Mat queryHuMoments = computeHuMoments(resizedQueryImage);
186     cv::Mat queryHistogram = computeHistogram(resizedQueryImage);
187

```

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.

```

188 // Allocate GPU memory for dataset features and distances
189 int datasetSize = datasetImages.size();
190 double* deviceDatasetMean;
191 double* deviceQueryMean;
192 double* deviceDatasetMedian;
193 double* deviceQueryMedian;
194 double* deviceDatasetStdDev;
195 double* deviceQueryStdDev;
196 double* deviceDatasetHuMoments;
197 double* deviceQueryHuMoments;
198 double* deviceDatasetHistogram;
199 double* deviceQueryHistogram;
200 double* deviceDistances;
201 cudaMalloc(&deviceDatasetMean, datasetSize * sizeof(double));
202 cudaMalloc(&deviceQueryMean, sizeof(double));
203 cudaMalloc(&deviceDatasetMedian, datasetSize * sizeof(double));
204 cudaMalloc(&deviceQueryMedian, sizeof(double));
205 cudaMalloc(&deviceDatasetStdDev, datasetSize * sizeof(double));
206 cudaMalloc(&deviceQueryStdDev, sizeof(double));
207 cudaMalloc(&deviceDatasetHuMoments, datasetSize * 7 * sizeof(double));
208 cudaMalloc(&deviceQueryHuMoments, 7 * sizeof(double));
209 cudaMalloc(&deviceDatasetHistogram, datasetSize * 256 * sizeof(double));
210 cudaMalloc(&deviceQueryHistogram, 256 * sizeof(double));
211 cudaMalloc(&deviceDistances, datasetSize * sizeof(double));
212 auto stop = high_resolution_clock::now();
213 auto duration = duration_cast<milliseconds>(stop - start);
214 std::cout << "GPU Memory allocation time: " << duration.count() << " milliseconds" << std::endl;
215
216 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.


```

218 // Copy dataset features to GPU memory
219 for (int i = 0; i < datasetSize; ++i)
220 {
221     cv::Mat resizedImage;
222     cv::Mat imageGray;
223     cv::cvtColor(datasetImages[i], imageGray, cv::COLOR_BGR2GRAY);
224     resizedImage = resizeImage(imageGray, 128, 128);
225
226     double imageMean = computeMean(resizedImage);
227     double imageMedian = computeMedian(resizedImage);
228     double imageStdDev = computeStandardDeviation(resizedImage);
229     cv::Mat imageHuMoments = computeHuMoments(resizedImage);
230     cv::Mat imageHistogram = computeHistogram(resizedImage);
231
232     cudaMemcpy(deviceDatasetMean + i, &imageMean, sizeof(double), cudaMemcpyHostToDevice);
233     cudaMemcpy(deviceDatasetMedian + i, &imageMedian, sizeof(double), cudaMemcpyHostToDevice);
234     cudaMemcpy(deviceDatasetStdDev + i, &imageStdDev, sizeof(double), cudaMemcpyHostToDevice);
235     cudaMemcpy(deviceDatasetHuMoments + (i * 7), imageHuMoments.ptr<double>(), 7 * sizeof(double), cudaMemcpyHostToDevice);
236     cudaMemcpy(deviceDatasetHistogram + (i * 256), imageHistogram.ptr<double>(), 256 * sizeof(double), cudaMemcpyHostToDevice);
237 }
238
239 // Copy query features to GPU memory
240 cudaMemcpy(deviceQueryMean, &queryMean, sizeof(double), cudaMemcpyHostToDevice);
241 cudaMemcpy(deviceQueryMedian, &queryMedian, sizeof(double), cudaMemcpyHostToDevice);
242 cudaMemcpy(deviceQueryStdDev, &queryStdDev, sizeof(double), cudaMemcpyHostToDevice);
243 cudaMemcpy(deviceQueryHuMoments, queryHuMoments.ptr<double>(), 7 * sizeof(double), cudaMemcpyHostToDevice);
244 cudaMemcpy(deviceQueryHistogram, queryHistogram.ptr<double>(), 256 * sizeof(double), cudaMemcpyHostToDevice);
245

```

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.

```

246 // Set the number of threads per block and the number of blocks
247 int threadsPerBlock = 256;
248 int numBlocks = (datasetSize + threadsPerBlock - 1) / threadsPerBlock;
249
250 // Launch the kernel to compute distances
251 computeDistanceKernel<<<numBlocks, threadsPerBlock>>>>(deviceDatasetMean, deviceQueryMean, deviceDatasetMedian,
252 deviceQueryMedian, deviceDatasetStdDev, deviceQueryStdDev, deviceDatasetHuMoments, deviceQueryHuMoments,
253 deviceDatasetHistogram, deviceQueryHistogram, deviceDistances, datasetSize);
254
255 // Copy the distances from GPU memory to the host
256 double* distances = new double[datasetSize];
257 cudaMemcpy(distances, deviceDistances, datasetSize * sizeof(double), cudaMemcpyDeviceToHost);
258
259 std::vector<DistanceIndexPair> distanceIndexPairs;
260 for (int i = 0; i < datasetSize; ++i) {
261     DistanceIndexPair pair;
262     pair.distance = distances[i];
263     pair.index = i;
264     distanceIndexPairs.push_back(pair);
265 }
266
267 std::sort(distanceIndexPairs.begin(), distanceIndexPairs.end(), [](const DistanceIndexPair& a, const DistanceIndexPair& b) {
268     return a.distance < b.distance;
269 });
270
271 std::cout << "20 lowest distances:" << std::endl;
272 for (int i = 0; i < 20; ++i) {
273     std::cout << "Index: " << distanceIndexPairs[i].index << ", Distance: " << distanceIndexPairs[i].distance << std::endl;
274 }
275

```

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.

```

278
279 // Free GPU memory
280 cudaFree(deviceDatasetMean);
281 cudaFree(deviceQueryMean);
282 cudaFree(deviceDatasetMedian);
283 cudaFree(deviceQueryMedian);
284 cudaFree(deviceDatasetStdDev);
285 cudaFree(deviceQueryStdDev);
286 cudaFree(deviceDatasetHuMoments);
287 cudaFree(deviceQueryHuMoments);
288 cudaFree(deviceDatasetHistogram);
289 cudaFree(deviceQueryHistogram);
290 cudaFree(deviceDistances);
291
292 // Free host memory
293 delete[] distances;
294
295 auto stop1 = high_resolution_clock::now();
296 auto duration1 = duration_cast<milliseconds>(stop1 - start1);
297 std::cout << "Execution time: " << duration1.count() << " milliseconds" << std::endl;
298
299
300 return 0;
301 }

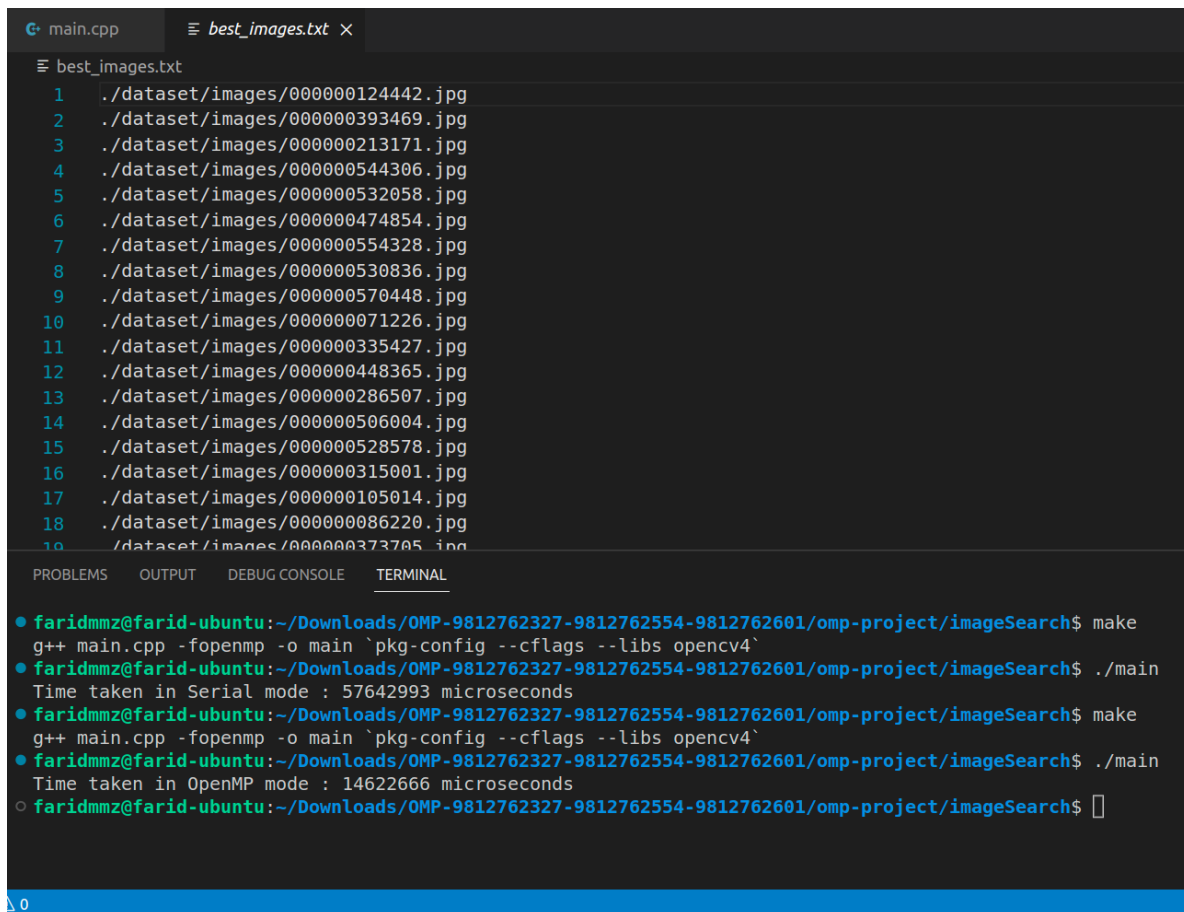
```

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.

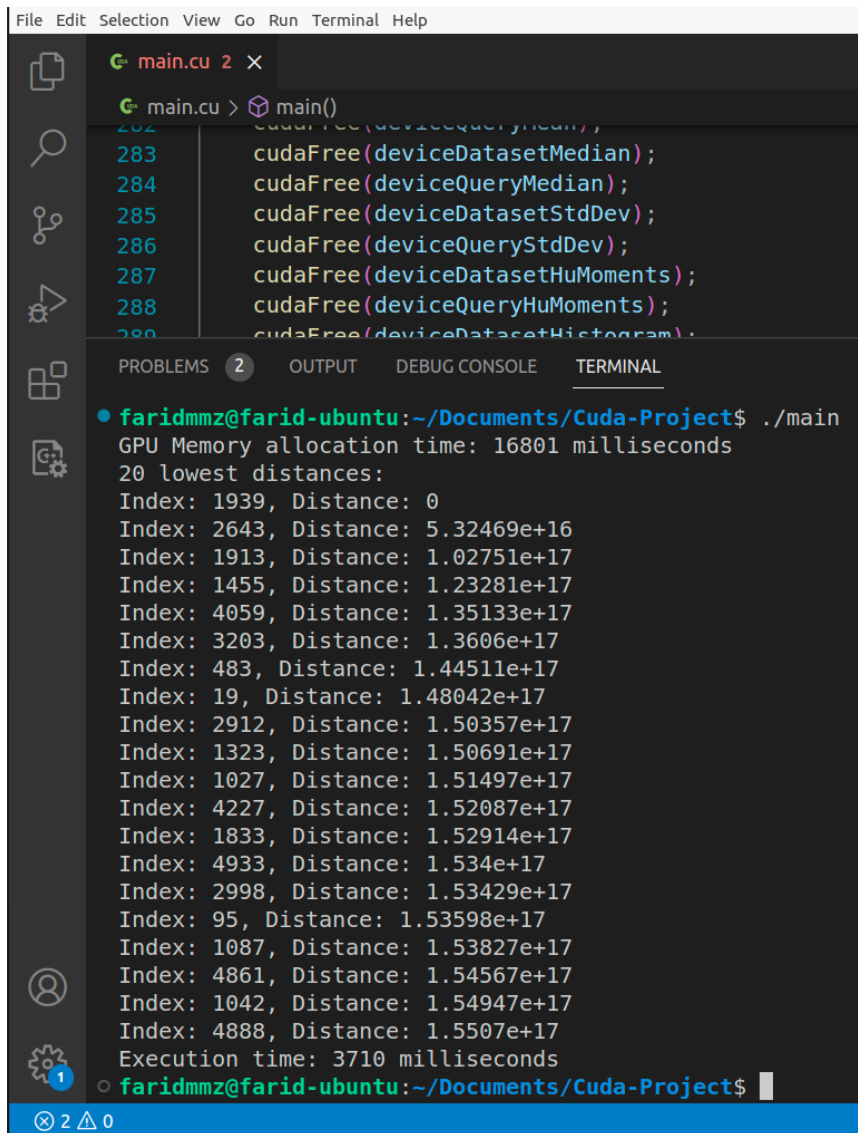


The screenshot shows a code editor with a file named `best_images.txt` containing 19 image paths. Below the editor is a terminal window showing the execution of a C++ program. The terminal output shows the time taken in serial mode (57642993 microseconds) and in OpenMP mode (14622666 microseconds).

```
main.cpp best_images.txt x
best_images.txt
1 ./dataset/images/000000124442.jpg
2 ./dataset/images/000000393469.jpg
3 ./dataset/images/000000213171.jpg
4 ./dataset/images/000000544306.jpg
5 ./dataset/images/000000532058.jpg
6 ./dataset/images/000000474854.jpg
7 ./dataset/images/000000554328.jpg
8 ./dataset/images/000000530836.jpg
9 ./dataset/images/000000570448.jpg
10 ./dataset/images/00000071226.jpg
11 ./dataset/images/000000335427.jpg
12 ./dataset/images/000000448365.jpg
13 ./dataset/images/000000286507.jpg
14 ./dataset/images/000000506004.jpg
15 ./dataset/images/000000528578.jpg
16 ./dataset/images/000000315001.jpg
17 ./dataset/images/000000105014.jpg
18 ./dataset/images/000000086220.jpg
19 ./dataset/images/000000373705.jpg

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• faridmmz@farid-ubuntu:~/Downloads/OMP-9812762327-9812762554-9812762601/omp-project/imageSearch$ make
g++ main.cpp -fopenmp -o main `pkg-config --cflags --libs opencv4`
• faridmmz@farid-ubuntu:~/Downloads/OMP-9812762327-9812762554-9812762601/omp-project/imageSearch$ ./main
Time taken in Serial mode : 57642993 microseconds
• faridmmz@farid-ubuntu:~/Downloads/OMP-9812762327-9812762554-9812762601/omp-project/imageSearch$ make
g++ main.cpp -fopenmp -o main `pkg-config --cflags --libs opencv4`
• faridmmz@farid-ubuntu:~/Downloads/OMP-9812762327-9812762554-9812762601/omp-project/imageSearch$ ./main
Time taken in OpenMP mode : 14622666 microseconds
• faridmmz@farid-ubuntu:~/Downloads/OMP-9812762327-9812762554-9812762601/omp-project/imageSearch$
```

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



The screenshot shows the Visual Studio Code interface. The editor window displays the file `main.cu` with the following code:

```
main.cu > main()
282 cudaFree(deviceQueryMedian);
283 cudaFree(deviceDatasetMedian);
284 cudaFree(deviceQueryMedian);
285 cudaFree(deviceDatasetStdDev);
286 cudaFree(deviceQueryStdDev);
287 cudaFree(deviceDatasetHuMoments);
288 cudaFree(deviceQueryHuMoments);
289 cudaFree(deviceDatasetHistogram);
```

The terminal window shows the output of running the program:

```
faridmmz@farid-ubuntu:~/Documents/Cuda-Project$ ./main
GPU Memory allocation time: 16801 milliseconds
20 lowest distances:
Index: 1939, Distance: 0
Index: 2643, Distance: 5.32469e+16
Index: 1913, Distance: 1.02751e+17
Index: 1455, Distance: 1.23281e+17
Index: 4059, Distance: 1.35133e+17
Index: 3203, Distance: 1.3606e+17
Index: 483, Distance: 1.44511e+17
Index: 19, Distance: 1.48042e+17
Index: 2912, Distance: 1.50357e+17
Index: 1323, Distance: 1.50691e+17
Index: 1027, Distance: 1.51497e+17
Index: 4227, Distance: 1.52087e+17
Index: 1833, Distance: 1.52914e+17
Index: 4933, Distance: 1.534e+17
Index: 2998, Distance: 1.53429e+17
Index: 95, Distance: 1.53598e+17
Index: 1087, Distance: 1.53827e+17
Index: 4861, Distance: 1.54567e+17
Index: 1042, Distance: 1.54947e+17
Index: 4888, Distance: 1.5507e+17
Execution time: 3710 milliseconds
faridmmz@farid-ubuntu:~/Documents/Cuda-Project$
```

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 memalloc	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 Cuda, 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