Parallel Implementations of Computer Vision Algorithms using a GPU

Andrew Kim
Computer Science
California Polytechnic University, San Luis Obispo
Email: akim73@calpoly.edu

Joseph Angeja
Computer Science
California Polytechnic University, San Luis Obispo
Email: jangeja@calpoly.edu

***Abstract*—The amount of performance increase seen in Central Processing Units (CPUs) seems to be plateauing within the recent years, as both speed and thermal limits are being reached. However, Graphics Processing Units (GPUs) have seen exponential growth in performance metrics within the past few years. Recent development has allowed GPUs to be used, not only for rendering complex and high-resolution graphics programs, but also for general purpose programming. This has allowed the GPU to act as a co-processor to the CPU resulting in high throughput. This harmonization between the CPU and GPU can be used for computationally-expensive algorithms where numerous independent calculations must be done to determine the final result of the program. Computer vision is a field that has developed within recent years to make sense of data within images. Algorithms within this fields are computationally expensive due to the fact that every pixel within the image in question must be analyzed. This paper explores the affects of using a GPU to create parallel implementations of both simple and complex computer vision algorithms.**

I. Introduction

The Graphics Processing Unit (GPU) was originally designed to render images and videos that the CPU would have trouble performing linearly. Computer vision, a field that has been of great interest of recent years, aims to solve the inverse problem of graphics processing. The goal of graphics processing is to render images from scenes. The GPU is used to calculate and update pixel values of an image faster than the CPU. However, a computer vision application tries to understand scenes from an image by analyzing the different pixel values and recognizing patterns generated by those values. Computer vision is heavily being utilized within various different industries. Cell phones use it to identify the user's face for security purposes. Photo storage applications such as Google Photos also utilize computer vision to identify people in images and automatically group photos based on the people in them or the surroundings identified in the images. However, as datasets for computer vision applications are continuously increasing in size, the amount of time to execute these programs linearly on the CPU is also increasing. The computationally-expensive aspect of some of the algorithms in computer vision can take advantage of the parallel processing that GPUs offer.

Some of the more simple operations in computer vision include various types of thresholding. Thresholding is the simplest form of image segmentation where pixels are turned black or white based on some comparison defined by the user. The basic form of thresholding turns a pixel white if the pixel value is greater than that of a defined constant, or black if it is less. This is done by comparing the value of every pixel within the image with the defined constant value. There are various types of thresholding that are used for different circumstances including binary thresholding ,adapative thresholding, RGB thresholding, etc. These operations are not as computationally expensive as more complex algorithms, but can still be parallelized.

More complex operations in computer vision are clustering and classification algorithms. Classification is an important data mining technique whereby a model is trained using a data set with class labels and is used to predict the class label of an unknown object [1]. One of the more widely used classification algorithms is k-nearest neighbors (kNN) [2]. kNN is used in computer vision algorithms to classify pixels as part of different objects. This is done by calculating the distance between every pixel in the input image and every pixel in a training set and using the k pixels with the shortest distances to classify the pixel in question. However, this algorithm is computationally expensive, especially for high-resolution images and large training sets.

In this paper, we explore how performance in both simple and complex computer vision algorithms can be improved using a GPU. We implement binary thresholding, RGB thresholding, and k-nearest neighbor algorithms and evaluate execution performance on a set of strawberry images. The OpenCV library is used for basic computer vision functionality such as reading in images to a matrix object. However, none of the OpenCV algorithms are used for benchmarking and analysis. The OpenCL library is used for GPU programming and running the parallel components of the algorithms.

The rest of this paper will proceed as follows. Sections II and III contain detailed explanations on how the thresholding and kNN algorithms work respectively and how they would run on the CPU. In Section IV we present our parallel algorithms for each of the algorithms. Our experimental details and results are presented in Section V, and our work is summarized in Section VI.

II. Thresholding

Thresholding is one of the simplest and most basic operations in computer vision. It is the simplest segmentation method and is used to separate out regions of an image corresponding to objects which we want to analyze. This separation is based on the variation between the pixels of the object and the pixels of the background [4]. To further understand how this process works, consider the source image with the intensity values src(x, y). The plot in Figure 1 depicts the threshold, which is represented by the horizontal line. The values above the line are accepted values that are part of the image to be analyzed, while values that fall below the line are blacked out. There are various types of thresholding, and this paper analyzes two: (1) binary and (2) RGB thresholding.

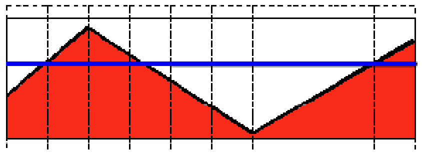

Fig. 1. Threshold Plot Diagram

A. Binary Thresholding

Binary thresholding uses a grayscale image to segment the image into black and white pixels. This is done by first grayscaling the input image to reduce the image to a single channel, reducing the number values to consider from three (R, G, and B) to a single value between 0 and 255. Once the image is converted to grayscale, each pixel value is compared to the constant threshold value. Any of the pixels in which their color value is less than or equal to the threshold value are turned black, while pixels with values higher than the threshold are turned white. Figure 3 shows the result of a binary threshold on the original image shown in Figure 2.

B. RGB Thresholding

RGB thresholding is very similar to the binary threshold algorithm described above, the only difference being that all three color channels are considered, as opposed to the one in binary threshold. Due to the fact that all three (red, green, and blue) channels are taken into account, there is no need to grayscale the image before performing the threshold. However, the channels do need to be split and iterated through separately. There are now three threshold values, one per channel. Each pixel is iterated through, comparing the values for the r, g, and b channels to the respective threshold constants. If the values in all three channels for the pixel exceed their respective thresholds, then the pixel is turned white, otherwise it is turned black. This is much more computationally expensive compared to the grayscale binary thresholding as it has to make three times the comparisons per pixel. The results of an RGB threshold on the input image in Figure 2 is shown in Figure 4.

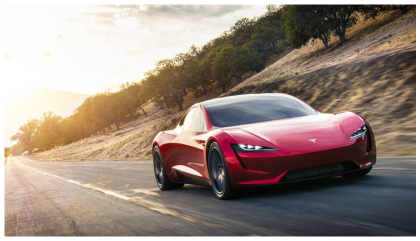

Fig. 2. Original Image

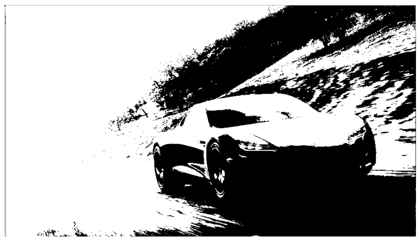

Fig. 3. Binary Threshold Results

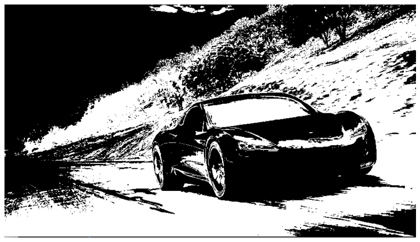

Fig. 4. RGB Threshold Results

III. K-Nearest Neighbors

The kNN classification algorithm is used to predict the class of an unknown object, or pixel, by the majority of the class labels of its k nearest neighbors in the reference objects [1]. We are interested in the brute force kNN algorithm, which is an exhaustive search which computes the distances between the pixel in question and every pixel in the training set [3]. This classification method requires a training set with each data point in the training set classified into a cluster. For every pixel in a given unknown image, a comparison is done between every labeled pixel in the training set. At every comparison, a distance metric (Euclidean distance in this case) is calculated and stored. Once all comparisons are made, the pixels with the k shortest distances are chosen, and a majority vote is taken in order to determine the class prediction of the unknown pixel in question. Figure 5 shows an illustration of the kNN search problem with k = 3 using the Euclidean distance.

IV. Parallel GPU Implementations

Before the GPU is used, initialization and preparation must be done to ensure the right compute device is selected, that there is sufficient memory available, and that the correct kernel is executing on the compute units. Before executing each GPU algorithm, the following steps are taken: select the correct

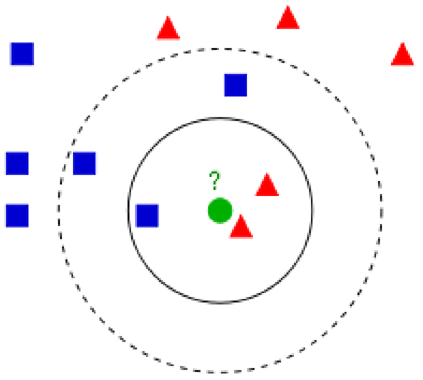

Fig. 5. RGB Threshold Results

GPU, create a context to allow communication between the GPU and CPU, read and build the OpenCL kernel code into the CPU, initialize the command queue to issue commands to the GPU, and allocate memory on the GPU for the image that will be run through the computer vision algorithms. The computer vision algorithms reference the command queue and image buffer, so storing those are necessary. We do so by storing all relevant GPU data in a GPU data structure that we create. All of the above steps are performed within the C++ code running on the systems CPU.

Each thread executes the same kernel code, but to divide the work, the thread `id` is used. Threads are given a global `id` and a local `id`. The former refers to the `id` of the thread in relation to all the other threads. The latter refers to the `id` of the thread within a work group. Our GPU implementations did not make use of the local `id`. However, the global `id` played a major role in all of our kernels.

A. *Background*

OpenCV is an open source computer vision library, mainly for real-time applications. The library is written in C++, but it includes APIs for other languages such as Python, Java, and Matlab. It contains useful functions and algorithms for computer vision and machine learning applications including implementations of thresholding, clustering, and classification algorithms. Some of OpenCVs implementations have the ability to take advantage of multi-core systems and GPUs [5].

OpenCV has implemented some OpenCL aware functions that require the use of a different data structure for the image, training set, etc [5]. However, the OpenCL implementations are hidden behind the normal OpenCV functions which makes it difficult to determine if OpenCL is actually being utilized.

OpenCL is an open source platform for parallel computing for a wide range of systems. Its platform is abstracted in a way that permits the use of a variety of hardware accelerators such as Graphics Processing Units (GPUs), digital signal processors (DSPs), and field-programmable gate arrays (FPGAs) [7]. This abstraction is done by viewing all devices on a particular system as compute devices, each of which contain a number for compute units. An OpenCL kernel is executed on multiple compute units in parallel. The division of the workload depends entirely on the programmer due to the fact that the number of threads, work groups, and how the compute device is divided is fully configurable.

The generic workflow for an OpenCL assisted program consists of the following steps: (1) build the kernel for the compute device on the CPU, (2) initialize data structures and allocate compute device memory, (3) write data to compute device buffer, (4) start kernel on compute device and let compute units on the compute device execute the kernel in parallel, (5) read the data from the compute devices memory back into the CPU. This generic workflow is accomplished by a set of OpenCL APIs built for C and C++. The OpenCL kernels are written in a C-like language called OpenCL C that is built upon C99. They are compiled at runtime using the above described OpenCL APIs.

B. *Binary Thresholding*

Our implementation of binary thresholding consists of a minimum and a maximum value that pixel values must fall between to be classified as a 1. We choose this method as opposed to the more widely used single value binary thresholding because the number of comparisons are doubled and allows for more computations to be executed on the GPU.

Algorithm 1 Kernel for Binary Thresholding
Input: img, min, max, numPixels, numThreads

```
1: for $id = global.id()$ to $numPixels$ do
2:     pixel = image[id]
3:     if ($pixel >= min\ and\ pixel <= max$) then
4:         pixel = 1
5:     else
6:         pixel = 0
7:     end if
8:     id += numThreads
9: end for
```

The Algorithm 1 kernel inputs five arguments: image, min, max, number of threads, and number of pixels within the image. The number of threads and number of pixels are the most important components of the kernel. It determines how much work each thread or work item will be doing on the image. The number of pixels a single thread will compute can be determined by dividing the number of pixels in the image by the number of existing threads in the kernel. Some threads will inherently execute one more pixel than others because some threads will have to process the remainder pixels. A particular thread will determine which pixel locations it is working on by first saving the `thread_id` to a variable `id`. If the `id` is less than the number of pixels in the image, then we want to do the work on the pixel specified by the `id` and increment the `id` by the number of threads. These steps are repeated until `id` is greater than the number of pixels. the work done for each pixel is defined as checking if the pixel grayscale value falls within the maximum and minimum thresholds. If the pixel lies within the range, then the pixel is set to 1 (white), otherwise it is set to 0 (black).

The CPU code performs the initialization steps and hands off the computational workload to be executed in parallel to the GPU. First, the image is written to the GPU buffer and a binary threshold kernel object is created, with the arguments set accordingly. The kernel is then enqueued to run, allowing the threads of the GPU to begin computing the values of different pixels in parallel. The CPU waits until all threads are complete to read the GPU image buffer back.

C. RGB Thresholding

Our RGB thresholding method is very similar to the binary thresholding algorithm we implemented. We use a minimum and a maximum threshold values for each component in RGB which doubles the number of comparisons as compared to the standard RGB threshold algorithm. The kernel differs from the binary threshold kernel by the number of arguments received and the extra comparisons included to account for r, g, and b values. The CPU code is the same as the binary thresholding, except for the increase in number of bytes that are written to and read from the GPU because of the extra two components in RGB.

D. kNN

The kernel code for kNN is more complex. In binary and RGB thresholding all the work is done by the GPU while the CPU acts only as the director of traffic. In kNN, the GPUs role is not as clear. We tried multiple different methods, but only one provided accurate results. Our default training set was relatively small at around 50,000 pixels to save time on the test runs.

The first working method of kNN defined the work of a thread differently than the thresholding algorithms. In the earlier algorithms, a thread worked on one or more pixels on the test image. However, in this kNN method, a thread works on one or more pixels on the training set, with all threads working on the same test image pixel. Our training set is an array of Pixel structs that we defined which contain a distance field to store the calculation of the Euclidean distance between the training set pixel and the test image pixel in question. Because classifying pixels in parallel is desired, a single training set shared by all threads is insufficient. This is because each pixel needs to calculate the distance between itself and each training set pixel and store that distance until all the distances are calculated. Because of this, it makes sense to have a copy of the training set for each pixel within the test image, but even with a small training set as ours, it becomes quiet memory intensive to do so. We combat this by only allowing a `MAX_TRAININGSETS` number of pixels to be classified in parallel. This limits the number of training set copies we need to allocate on the GPU and CPU.

Algorithm 2 Kernel for Method 1

```
Input: imgPixel, trainingIndex, trainingSets[], numTrainingPixels, numThreads
1: train = trainingSets[index * numTrainingPixels]
2: for $id = global.id()$ to $numTrainingPixels$ do
3:     train[offset].dist = Dist(imgPixel, train[offset])
4:     id += numThreads
5: end for
```

Algorithm 2, takes in five arguments: the test image pixel, the index into the array of training sets, the array of training sets, the number of pixels in one training set, and the number of threads. The threads' first step is to store the global thread id into a variable called `id`. The kernel grabs one training set by using the passed in variables `index` and `length` of the training set. The indexing into the array is done by multiplying the index by the training set length. Similar to the thresholding algorithms, the thread executes until the `id` variable is greater than the number of training set pixels. For each training set pixel, the distance to the input image pixel is calculated and the result is written back to the training set associated with that input image pixel.

As expected, the CPU plays a bigger role in this algorithm. On top of the initial setup of the binary thresholding algorithm, the CPU must create `MAX_TRAININGSETS` buffer on the CPU and the GPU. Since the algorithm can only classify `MAX_TRAININGSETS` test image pixels at a time, there must be some syncing of the CPU and GPU after all `MAX_TRAININGSETS` test image pixels have had their distances completely calculated. This is shown in Algorithm 3.

Algorithm 3 CPU for Method 1

```
Input: img, k, numThreads
1: trainSize = gpuTrainSets.size()
2: for $i = 0$ to $numPixels$ do
3:     pixel = image[id]
4:     if ($i\% \ MAX\_TRAININGSETS == 0$) then
5:         trainSets[] = readBuffer(gpuTrainSets)
6:         for $j = 0$ to $MAX\_TRAININGSETS$ do
7:             sort(trainSets[j])
8:             class1 = SUM1(trainSets[j][0]...trainSets[j][k])
9:             class2 = SUM2(trainSets[j][0]...trainSets[j][k])
10:            if $class1 > class2$ then
11:                $img[j + i - MAX\_TRAININGSETS] = 0$
12:            end if
13:        end for
14:    end if
15:    spawnKernel(img[i], i)
16: end for
```

V. Experiments

After implementing binary thresholding, RGB thresholing, and kNN in both linear CPU and parallel GPU environments, each of the algorithms are compared for speed performance. As for the hardware utilized, we have an Intel Core i7 processor running at 2.7GHz as well as an AMD Radeon Pro 560 dedicated graphics card clocked at 907MHz.

A. Threshold Benchmark Process

The thresholding benchmarks are used to compare the performance between the linear (CPU) and parallel (GPU) implementations of the operations. Embedded timing callsed withing the code provide the time that it takes the entire program to run, as well as the time it takes to run the isolated thresholding functions (excludes other operations such as grayscaling). The benchmarks were run 10 times on a 1024x768 image, and the average of the results were recorded.

B. kNN Benchmark Process

To benchmark the performance of our kNN implementations, the algorithm is put to the test of identifying strawberry pixels in a given image. A training set is created using a different image of a strawberry. A mask is created and overlapped onto the image in order to classify strawberry and non-strawberry pixels for the training set. After the training process is complete, the kNN algorithm is passed in a foreign image to identify the strawberry pixels. Timing statements in the source code provided accurate time measurements of the runtime of each algorithm and critical parts of the program such as the classification process.

In order to measure how performance scales with the dataset size, the kNN algorithm is run against a combination of different sized training sets and input images. The baseline test took a 424x283 input image to classify, which was run against a 50,246 pixel count training set. We then increase the training set size to 270,800 pixels while keeping the input image size the same. This results in many more comparisons as every pixel in the input image now has to be compared to a greater amount of pixels in the training set. The final test we do is keeping the training set at the baseline value, and increasing the input image size to 1024x768. This also drastically increases the amount of computations as there are more pixels to classify. The benchmarks are intended to test in which situations does the GPU's computational throughput create the most performance increase, if any. Each of the benchmarks were run ten times with a constant value for k, and the average of the results were recorded for comparison.

C. Results

Our GPU algorithms were run with multiple different number of threads to determine the optimal number. When the test image pixel to thread ratio approached one, we saw that the GPU thresholding algorithms actually performed slightly worse than the respective CPU versions. The reason for the performance decrease as the ratio approaches one is that the overhead of creating threads to analyze each pixel is greater than having pixels analyze multiple pixels. In order to fully utilize the GPU efficiently, a threads creation time should be less than the time it spends doing work. We theorize that the degrading performance is due to the fact that when the ratio approaches 1, many threads are being created, but they are doing the work of only one or two pixels which takes less time than it takes to create the thread. We found the optimal pixel to thread ratio to be somewhere between 100 and 200. The tests shown below were run with a pixel to thread ratio of 100. We saw an improvement in both thresholding algorithms and this is mainly due to the GPUs ability to analyze multiple pixels at once, whereas the CPU version must operate serially.

The GPU binary thresholding algorithm performed roughly 60% better than the CPU algorithm when the number of threads are significantly lower than the number of pixels in the image, while the GPU RGB thresholding algorithm exhibited on average a 77% decrease in execution time. The higher speedup with the GPU RGB algorithm compared to the GPU binary algorithm is attributed to the more comparisons being done in RGB thresholding. The graphical results are shown in Figure 6.

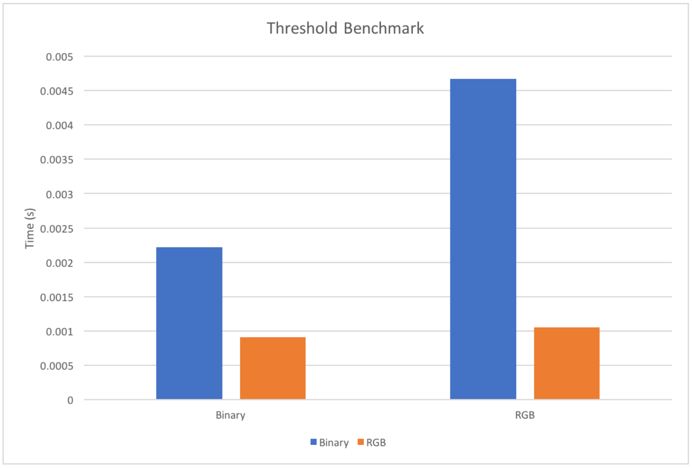

Fig. 6. Benchmark for Threshold Algorithms

The baseline tests show a decrease in classification time by about 10%. The varying of the value k did not exhibit any change in the difference of overall program performance, nor did it seem to have affected the time it took to classify each pixel. We suspect this is because k plays such a small role in the actual number of computations the algorithm performs. For both k = 3 and k = 15, the algorithm still needs to calculate the distance for every pixel in the training set for every pixel in the test image. The k value only comes into play when the distance calculations are done and the distances are sorted. The algorithm then looks at the k minimum distances and runs through the voting algorithm. This small part of the kNN has magnitudes less calculations than the distance calculating and sorting. This is why there is no change in the magnitude of performance increase when changing k as shown in Figure 7.

In the next test, we increased the training set size expecting to see a larger increase in performance due to the fact that the total number of distance calculations is over 32 billion (size of test image * size of training set) which is far more than the 6 billion pixels from the baseline test. Also, the number of test image pixels this algorithm can classify in parallel

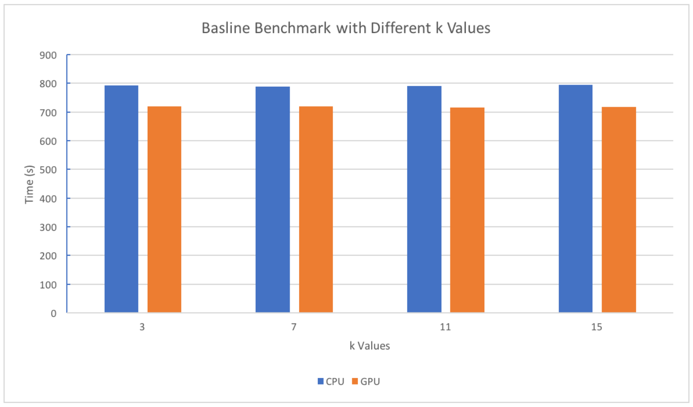

Fig. 7. Baseline kNN with Different k Values

is bottlenecked by `MAX_TRAININGSETS`. The training set distance calculation for one pixel is fully parallelized, so the larger the training set, the more of an increase in performance for the GPU algorithm. With this test we see about a decrease in classification time by 15% in Figure 8.

The final test kept the same training set size and increase the test image size so we should not see the increase in performance that we saw in the second test, but we should still see something comparable to the baseline test. We saw an improvement of about 13%.

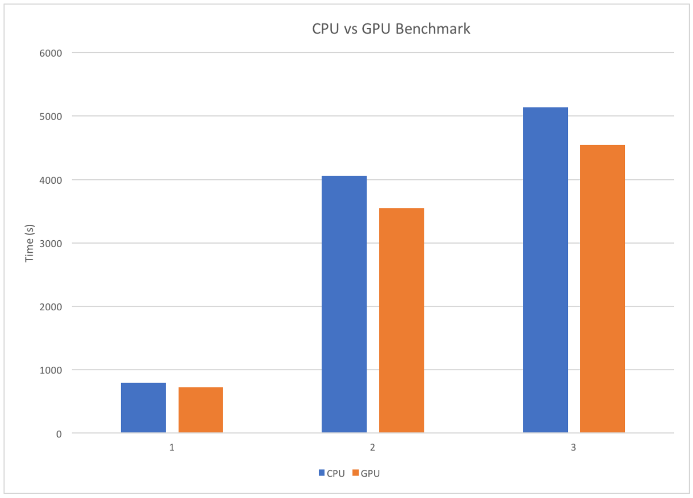

Fig. 8. Benchmark kNN with Different Image Size

VI. Conclusion

In this paper, we explored the affects of parallelizing simple and complex computer vision algorithms and examined how it affected performance. We implemented two different thresholding algorithms: binary and RGB, as well as the k-nearest neighbors classification algorithm for both the CPU and the GPU. In both cases we saw at least a 10% increase in performance when run on the GPU as compared to the CPU.

However, we believe that we could achieve more drastic results by using non-mobile computing hardware. The mobile GPU was not clocked nearly as fast as those meant for dedicated desktops or servers. The CPU in our system is very fast and efficient, and it does a fairly good job at keeping up execution speeds linearly on a single thread. This experiment shows that GPU-based parallel computing can contribute greatly to computer vision applications.

Source Code: https://github.com/ak2795/CSC_515.git

Acknowledgment

The authors would like to thank Professor John Seng for instructing the CSC 515 Graduate Computer Architecture course.

References

[1] Shenshen Liang, Ying Liu, Cheng Wang, Liheng Jian, *Design and Evaluation of a Parallel K-Nearest Neighbor Algorithm on CUDA-enabled GPU*, IEEE 2010.
[2] M. Kamber, J. Han, *Data Mining: Concepts and Techniques*, 2nd Edition, Morgan Kaufmann, 2005.
[3] Vincent Garcia, Eric Debreuve, Frank Nielsen, Michel Barlaud, *K-Nearest Neighbor Search: Fast GPU-Based Implementations and Application to High-Dimensional Feature Matching*, 2010 IEEE 17th International Conference on Image Processing.
[4] *Basic Thresholding Operations*, https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html.
[5] *OpenCL*, https://opencv.org/platforms/opencl.html
[6] *OpenCV*, https://opencv.org
[7] *The open standard for parallel programming of heterogeneous systems*, https://www.khronos.org/opencl/