CUDA-Based Bag-Of-Words Scene Recognition

Hanyu Chen
Carnegie Mellon University
hanyuche@andrew.cmu.edu

1. Summary

In this project, I implemented a parallel bag-of-words scene recognition algorithm using CUDA. The algorithm is based on 16-385 *Computer Vision* assignment 5, which was originally written as a sequential algorithm in Python [1]. Three stages of the algorithm—image filtering, k-means clustering, and feature creation—are parallelized separately. The programs runs on a single RTX 2070 Super GPU paired with an Intel i5-10600 6-core CPU. In the three stages of the algorithm, the CUDA implementation achieves speedups of 55.8x, 42.9x, and 48.1x respectively compared to the baseline sequential algorithm, and achieves speedups of 8.0x, 6.0x, and 6.3x respectively compared to an OpenMP implementation.

2. Background

Image classification is one of the most common tasks in the computer vision field. The task requires a computer program to identify the class that a given image belongs to from a set of predetermined classes. In scene recognition in particular, the program is expected to identify the scene that the given photo is taken in.

The bag-of-words model is a representation that was initially used for natural language processing tasks. The model essentially represents a text as a histogram of frequencies that certain words appear in the text, which captures high-level information helpful for classification tasks. The bag-of-words model is later adapted in computer vision and used in a similar manner. However, due to differences in the nature of computer vision and natural language processing tasks, certain aspects of the model are changed to suit computer vision tasks better.

2.1. Image Filtering

Image filtering is a common technique used in image processing to extract local features in images. The technique involves convolving an image filter, which is typically a smaller image, with the image being processed, and produces a response map. In my implementation, the response map is ensured to have the same dimensions as the original image by padding values around the original image depending on the size of the filter.

2.2. Visual Words

In the image filtering process, a filter bank of M filters are convolved with each image, producing M response values for every pixel of the original image. A visual word is defined as a vector of response values that corresponds to a particular pixel. For a grayscale image of shape $H \times W \times 1$, the image filtering process produces HW visual words of length M in total.

2.3. Dictionary

In natural language processing, a dictionary is usually constructed as a set of most frequently appearing words. Similarly, in this task, a dictionary is defined to be a set of visual words. However, since visual words are continuous unlike words consisting of of discrete letters, the dictionary is not constructed based on the frequency of visual words. Instead, a k-means clustering algorithm is run on a large subset of randomly chosen visual words from the entire set of training images. The cluster centers are used to constructed the dictionary.

2.4. Image Features

In natural language processing, bag-of-words features of a text is created by counting the number of times each word in the dictionary appears in the text. In this task, the features of each image is computed by considering which cluster the visual word corresponding to each pixel belongs to. Then, counting the number of pixels that belong to each cluster gives a vector of length C, which is normalized by dividing the total number of pixels HW to give a probability distribution over the clusters. The probability distribution vector of length C is defined to be the feature vector of an image.

2.5. k-Nearest Neighbors

k-nearest neighbors is a very basic classification algorithm that classifies a point based on majority vote of training points nearest to it (as the name suggests). In this particular case, the training points are the image features of train-

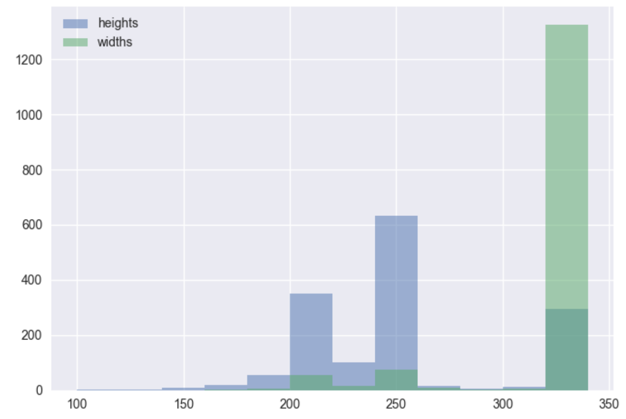

Figure 1: Distribution of image heights & widths

ing images, and a new image is classified based on training points nearest to its image features.

3. Approach

In my CUDA implementation, I aim to parallelize three stages of the algorithm separately, since intermediate results need to be saved for easier debugging and visualization. Loading and saving images, response maps, and visual words are not considered as any part of the three stages, so they are not accounted for in the speedup calculations. This is a reasonable assumption to make since in most applications, the data preprocessing, model creation, and inference steps are often carried out separately.

3.1. Convolution

In my implementation, the filter bank consists of 8 sets of Gaussian, Laplacian of Gaussian, horizontal Sobel, and vertical Sobel filters of varying sizes to account for features of different scales. The dataset consists of $N = 1{,}491$ images from $C = 8$ classes. All images are in grayscale and the average image dimension is 246×310 while the image heights and widths follow the distribution shown in Figure 1.

The baseline sequential solution to image filtering is simple: for each image of shape $H \times W$ in the dataset and filter of shape $S \times S$ in the filter bank, loop over all HW image blocks of shape $S \times S$ in the image, and convolve the filter with the image block sequentially (i.e. take the element-wise product over all pixels and compute the sum) to compute the response map of shape $H \times W$. The convolution operation is visualized in Figure 2.

The naive method to parallelize image filtering is to parallelize over all possible image blocks for a given (image, filter) pair. In other words, each CUDA thread is responsible for convolving a image block with a filter and computes the value of a single corresponding pixel in the response map.

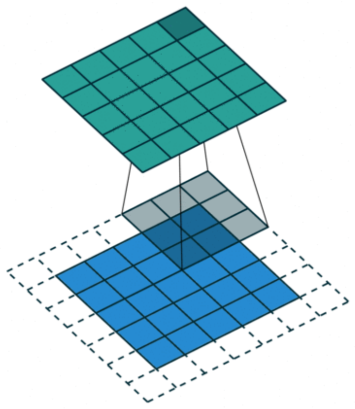

Figure 2: Visualization of convolution operation [2]

Specifically, the CUDA kernel is launched with a block size of $B \times B$ and a grid size of $\lceil H/B \rceil \times \lceil W/B \rceil$. Iterating through all images and all filters is still done sequentially, and they are sent to GPU memory separately on-demand. This method achieves a noticeable improvement over the sequential version. The relative speedups with varying block sizes are shown in Figure 3 (naive). The highest speedup of 23.8x is achieved with a block size of 8×8.

The second method I tried is to exploit as much parallelization as possible. Instead of having each CUDA thread being responsible for convolving a single image block with a filter (which is sequential in nature), each thread block is now responsible for this task. Each thread is then responsible for a single multiplication between a pixel in the image block and a pixel in the filter. The sum is accumulated (atomically) in a shared variable among the thread block. Specifically, the CUDA kernel is launched with a block size of $S \times S$ and a grid size of $H \times W$. Note that this method only works with filters of at most 32×32 in size. However, the results are not promising. This method requires no

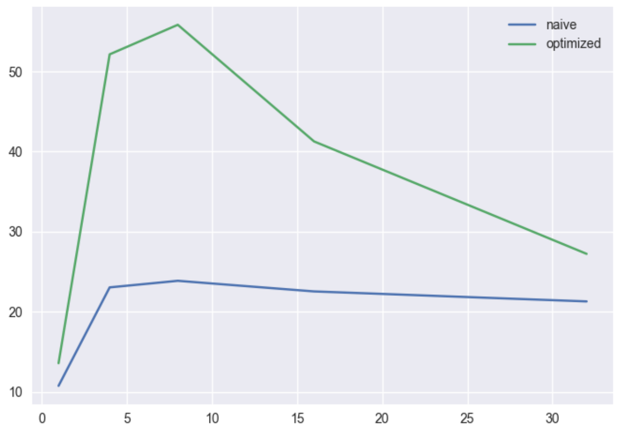

Figure 3: Speedup for image filtering

parameter tuning and only achieves a speedup of 1.5x.

Therefore, for the final method I reverted to an optimized version of the first method. Firstly, since the number of filters is relatively small and the filters are fixed, it is feasible to parallelize not just over the image pixels but also across different filters. Each CUDA thread will be responsible for convolving a single image block with a filter while multiple filters are being processed at the same time. In other words, a block size of $B \times B \times 1$ and a grid size of $\lceil H/B \rceil \times \lceil W/B \rceil \times M$ are used. Secondly, since the images and filters are all fixed, there is no need to send them to GPU memory separately. Therefore, in my final method, the images and filters are reshaped into 1D float arrays in the preprocessing step and are sent to GPU memory as a whole. Also, note that parallelizing over images is less feasible since image dimensions vary and it is difficult to determine a single grid and block size that works for all images at the same time. The relative speedups with varying block sizes are shown in Figure 3 (optimized). The highest speedup of 55.8x is achieved with a block size of 8×8.

3.2. Clustering

After image filtering, each pixel will have a corresponding visual word of length M. Ideally, the clusters centers should be determined from all of these vectors, but running k-means on such a large number of points is infeasible. Therefore, p pixels (and their corresponding vectors) are random selected from each image in the dataset, giving pN points in total, and k-means is performed on the points. In my final implementation, values of $p = 200, k = 500$ are used.

The k-means algorithm consists of two steps in each iteration: assigning every point to its nearest cluster and computing new cluster centers from the mean of all points assigned to each cluster. Note that when assigning points to clusters in the first step, the sum of all points in each cluster can be accumulated at the same time, which makes the second step of finding the mean rather trivial.

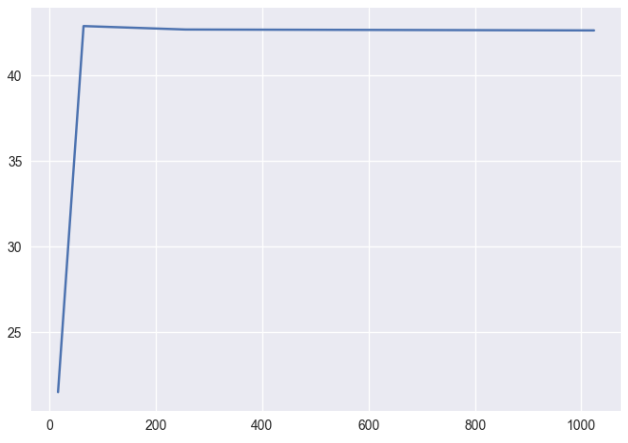

Figure 4: Speedup for k-means clustering

For my first parallel implementation, I still tried to maximize parallelism by making the entire loop run in a single CUDA kernel. This reduces the need to constantly synchronize variables between host memory and GPU memory in each iteration. However, the main issue with this approach is that all threads must have completed the first step before any thread moves on to the second step. This requires synchronization between all threads (not just within each block) in every iteration, which introduces a non-trivial overhead. Therefore, I decided to move the main loop back to host code and instead create two CUDA kernels that are called sequentially for the two steps in the k-means algorithm.

In the first kernel, each thread is responsible for computing the distances to all cluster centers for one single point. After determining the nearest cluster, it updates three global variables: one that keeps track of which cluster each point belongs to, one that counts the number of points in each cluster, and one that accumulates the sum of all points (each a vector of length M) in each cluster. Note that the latter two operations need to be atomic since multiple threads may be updating the same value at the same time.

In the second kernel, each thread is responsible for both checking if the cluster assignment of a particular point has changed, which is used for checking convergence, and computing the new cluster center of each cluster, by dividing the accumulated sum by the number of points. For the first task, a single global variable is set to 0 initially, while every thread whose cluster assignment has changed will try to set the variable to 1. The loop in the host code terminates when it finds that the variable is set to 0 after an iteration completes.

When testing with 200 images, the algorithm takes between 96 to 166 iterations to converge, so the performance is evaluated on the average time that each iteration takes. The speedup over the sequential implementation is shown in Figure 4. Note that since this step only requires parallelization in 1D, the block has size B and the grid has size $\lceil pN/B \rceil$. The highest speedup of 42.9x is achieved with a block size of 64.

3.3. Features

Assigning features to images follows a similar process as dictionary creation. In the first step, each pixel in an image is assigned to the nearest cluster of its corresponding visual word. At the same time, the number of pixels that are assigned to each cluster is counted. This produces a pixel count vector of length k. Then, each pixel count is divided by HW, which is the total number of pixels to produce a distribution that sums up to 1, which is used as the feature vector of the corresponding image.

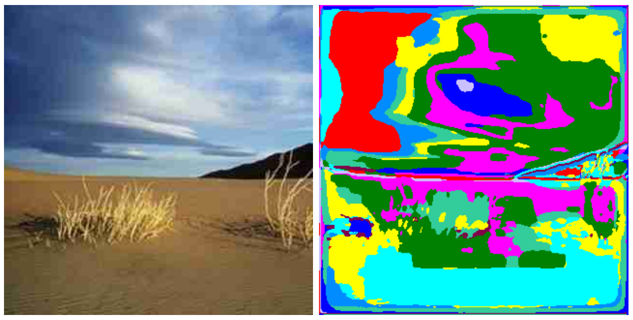
Figure 5: Original image & cluster assignments

This step is easier than dictionary creation in the sense that the feature vector of each image can be computed independently with fixed cluster centers. However, it is also more computationally expensive since it requires computing the nearest cluster center for every pixel in an image instead of just a subset of p pixels.

My implementation still focuses on parallelizing over pixels instead of across images due to the same reason as in image filtering. Each CUDA thread is responsible for a single pixel. It iterates over all cluster centers and computes the distances from its corresponding visual word. The pixel is then assigned to the nearest cluster and the counter for that cluster is incremented. Note that this operation needs to be atomic. After all threads have finished, each element in the counter vector is divided by HW in the host code to give the final feature vector. The highest speedup of 42.9x is achieved with a block size of 64.

The pixel assignment to clusters can be visualized in Figure 5, with different colors corresponding to different clusters. It is worth noting how connected regions in the original image are often assigned to the same cluster. This provides some evidence that the algorithm is capable of recognizing some image semantics, although a detailed discussion is beyond the scope of this project.

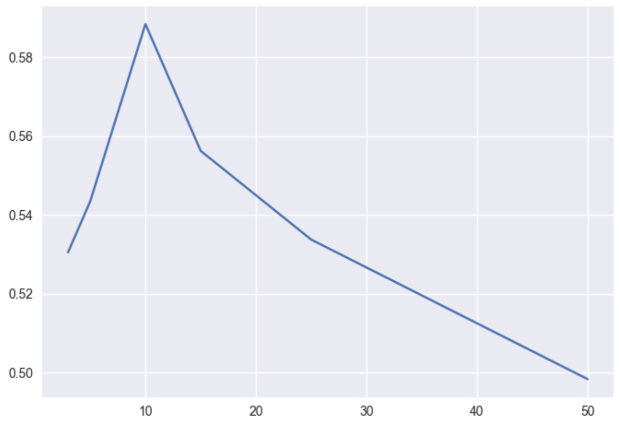

Figure 6: Classification accuracy for K-NN

3.4. Classification

For evaluation, I implemented a simple K-nearest neighbors classifier (capital K used to avoid confusion with lower case k in k-means. The dataset of N images is divided into a training set of size $0.8N$ and a validation set of size $0.2N$. For each image in the validation set, the algorithm simply finds the K nearest images in the feature space of training images, and it uses majority vote to determine the class of the image.

It is worth noting that since the image features are probability distributions, the chi-squared distance metric is more suitable than the standard Euclidean distance metric, and it gives noticeably better prediction accuracy. The classification accuracy with varying K values are shown in Figure 6, and the highest accuracy of 58.8% is achieved with $K = 10$. The confusion matrix is also shown in Figure 7.

The sequential classifier runs in less than 1 second on the dataset of 1,491 images, so I did not parallelize it since the speedup would not be very significant anyways. However, if I were to implement a more advanced classifier that relies on, for example, matrix multiplications, then parallelizing the classifier will have a more significant impact on performance.

4. Analysis

Since results have been mostly presented along with the approaches, in this part, I will focus on analyzing the impact that different design choices in parallelization have on the speedups and also briefly discuss limiting factors and potential improvements for my current implementation.

4.1. Convolution

The lack of speedup from the second method (each thread responsible for a single multiplication) stands out the

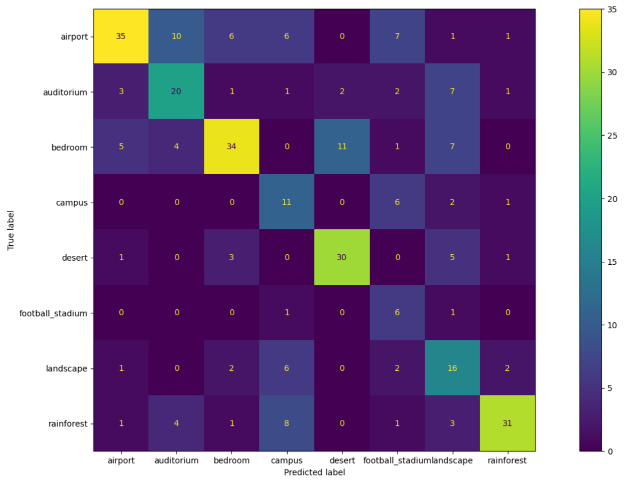

Figure 7: Confusion matrix for K-NN

most from the three methods. My rationale for implementing this method is that, each thread in the first method has to iterate over pixels in an sequential manner, so if this loop is parallelized, there might be a higher performance gain. However, this method is far from ideal, since every thread that is responsible for the same image block will be accumulating values at the same pixel location in the response map. Therefore, each update need to be atomic. This gives a very high overhead, while the actual computation of a single multiplication is too cheap to justify this.

The third method optimizes upon the first method and reduces communication costs significantly by passing all images and all filters to the GPU memory before any computation, instead of passing them $2MN$ times (two passes for each iteration of the nested loop) in total. Moreover, being able to parallelize across filters also improves performance. In particular, the additional dimension for M filters is added to the grid dimensions instead of the block dimensions because the filters vary in size, so having one thread warp process multiple filters might lead to divergence. In contrast, there is no synchronization across blocks, so having different blocks process different filters should give better performance.

4.2. Clustering & Feature Creation

The tradeoffs between running the main loop in a single CUDA kernel or in host code during k-means has been discussed briefly. The former requires several synchronization between all threads in each iteration, while the latter requires two kernel launches and some data copying in each iteration.

One other decision to make is whether each thread should be responsible for computing the distance of a visual word to every cluster center, or should the work be divided among multiple threads. However, the drawback of dividing the work is that, since the algorithm tries to find the nearest cluster to each visual word, a reduction needs to be done across all computed distances, and this increases the communication overhead.

It is worth noting that in general, there is also a tradeoff when choosing the block size. Ideally, we would like the block sizes to be 1 so that each thread can run completely independently. However, having a block size that is too small also increases the overhead for launching blocks significantly. Therefore, in general, there is usually a peak in speedup for intermediate block sizes (8×8 for image filtering and 64 for clustering).

4.3. Classification

As mentioned before, K-nearest neighbors is a very simple classification algorithm that does not achieve accuracy as high as many other more advanced algorithms like decision trees, logistic regression, and support vector machines.

Table 1: OpenMP & CUDA speedup comparison

	OpenMP	CUDA
Convolution	7.0x	55.8x
Clustering	7.2x	42.9x
Features	7.6x	48.1x

Although a discussion of different classification algorithms is beyond the scope of this project, it might be worth mentioning how matrix multiplication often plays an important row in these more advanced algorithms. These tasks are quite suitable for implementing in CUDA since they can be decomposed into smaller independent subtasks (such as dividing a matrix into smaller blocks).

4.4. OpenMP

Lastly, I would like to mention some parallelization results I obtained by adding simple OpenMP directives to my baseline sequential code. Considering how easy it is to implement such parallelization schemes, implementing more complicated CUDA kernels can only be justified if they achieve noticeably higher speedup compared to just using OpenMP, which is the case here.

Since I am running the program on a 6 core, 12 thread CPU, the ideal speedup from OpenMP should be around 12x. The actual OpenMP speedups and CUDA speedups of all three tasks are summarized in Table 1

References

[1] M. O'Toole. Programming assignment 5: Scene recognition with bag of words.

[2] I. Shafkat. Intuitively understanding convolutions for deep learning, 2018.