Parallelization of K-means Clustering Model for chest X-ray images using CUDA and OpenMP

by
Juhi Checker - 1605378
Rohnie Baskar - 1629049
Piyush Kulkarni

Introduction

Motivation

A global population of 7% is affected by pneumonia, which kills around billions of people a year. For the diagnosis of this disease, chest X-rays are commonly used. However, even a trained radiologist can find examining chest X-rays challenging. In order to improve the accuracy and speed of the diagnostic process, computer technology has been used to improve human expertise. Some limitations of human-based diagnostics have motivated the use of artificial intelligence techniques. Therefore, in this project we use parallelizing strategies to K-means clustering algorithms to train and test X-rays of patients at a faster rate with maximum accuracy. Parallelizing frameworks such as OpenMP and CUDA are analysed in this project.

Objective

The following were the objectives of the project:

1. Developing K-means clustering algorithm from scratch
2. Classifying X-ray into Pneumonia or normal
3. Parallelizing the serial K-means algorithm using the OpenMP framework
4. Learning the CUDA framework
5. Parallelizing the serial K-means algorithm using CUDA framework
6. Increasing the speedup for testing X-rays
7. Analysing both the frameworks

System Design & Implementation

Algorithm

For this project, we have used the K-means Clustering algorithm. K-means algorithm is an unsupervised learning algorithm which classifies every pixel of the given image into a given cluster.
The K-means algorithm works as follows:

1. Set K as the number of clusters.
2. The centroids are initialized by taking random data points of pneumonia and non-pneumonia.
3. Continue to iterate until there are no differences between the centroids i.e. the assignment of data points to clusters isn't changing. Here the distance is calculated using the Euclidean distance.
4. Calculate the squared distance between all data points and all centroids.
5. Match each data point with its nearest cluster (centroid).
6. Using the average of all points belonging to each cluster, determine the centroids for the clusters.

The reason behind selecting this algorithm was that the K-means algorithm is performed serially. In other words, each iteration is dependent on the previous iteration for its calculated centroid i.e. loop dependent. With a huge amount of data this algorithm's execution time would increase therefore, we choose to parallelize it. Thus, this loop dependent algorithm was challenging to parallelize.

Frameworks Used

1. OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. The runtime environment allocates threads to processors depending on usage, machine load and other factors.

We were most comfortable with OpenMP since we had hands-on practice with it. Therefore, we chose to go ahead with it.

2. **CUDA**

CUDA is a parallel computing platform and application programming interface (API) that allows software to use certain types of GPU for general purpose processing. It is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives such as OpenACC, and extensions to industry-standard programming languages including C, C++ and Fortran.

CUDA is a very popular framework since it provides the best performance for parallel programming. Therefore, we decided to go ahead with it.

Parallelizing Details

OpenMP

- Take training/testing input images as a text file with all the pixels of the images in a list.
- Equally divide the data points array amongst each of the threads available.
- Centroids are initialized and shared among all the threads.
- For each iteration, each thread calculates the euclidean distance between all the data points they have in their assigned chunk of the array and the two centroids.
- Next, with the distances it will classify the datapoint into one of the two clusters whose centroid is the closest.
- All the assigned datapoint values will then be added to a local sum variable for each cluster.
- Each thread will compute average global centroids for the two clusters. This particular section needs to be done one at a time hence using the critical construct.
- Only after every thread computes the above step, one thread will find the distance between the old centroid and new centroid. If the distance computed is less than the threshold defined, it will break the loop and give the two cluster's centroid points, else it will train again.
- Only after the single thread computes the above step, each thread will increment its local iterator.
- This process is repeated until the maximum number of iterations is reached or finds the proper centroid.

CUDA

The parallelization strategy for CUDA is similar to OpenMP, in terms of tasks assigned to each thread during parallel execution. However, it differs in some manner:

- Parallel execution of the code is done on GPU cores.
- Instead of parallelizing a section of code, CUDA requires a complete function to be declared as __global__ or __device__.
- Data lies on unified memory, thus being shared between CPU and GPU.

OpenMP uses only CPU threads to execute in parallel, whereas the case with CUDA is different, in the way that it runs the complete parallel code on GPU cores. To do this, it needs data to be present on the GPU(Device) memory itself, instead of the host's (CPU's) main memory.

One way in which this is achieved is by using the cudaMemCpy() function, which is capable of transferring specified data between host and device memories, as per instructions. However, since our code has a large number of arrays and different variables to be copied to and fro, this didn't seem an optimal way to proceed with; since we need to transfer data everytime a new variable/memory space is introduced, or is modified.

To overcome this issue, we came across the concept of Unified memory. Unified Memory creates a pool of managed memory, where each allocation from this memory pool is accessible on both the host and device with the same memory address. The underlying system automatically migrates data in the unified memory space between the host and device. This data movement is transparent to the application, simplifying the application code. Initially, we declared all the memory pointers using the __managed__keyword, which made them available to both the host and the device. Global memory spaces such which were declared exclusively to be used by threads, were stored on the device, and made common between the threads using the __shared__ keyword.

Later, in the host functions, we initialized these memory spaces dynamically using the cudaMallocManaged() function, which is similar to the malloc() function in conventional C language. This function allocates size bytes of managed memory and returns a pointer. The pointer is now valid on all devices and the host. A program that uses managed memory can take advantage of automatic data migration and duplicate pointer elimination.

This way, the code turned out more optimal in a way that the number of memory transfers during execution is reduced, thus bringing data locality into the picture.

Experiments

Dataset

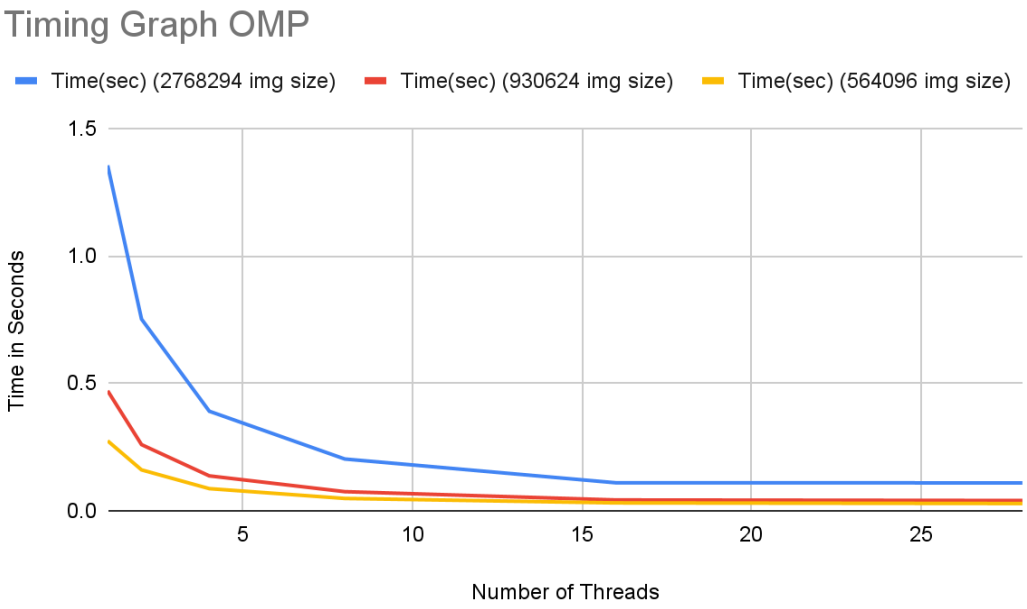
Name: Chest X-Ray Images (Pneumonia)
Source: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia
Type of data: Dataset consists of images in Jpeg format
Size of data: The dataset consists of 31.6M pixels from all the training images.
Training/Testing: For training the model using K-means we took 10 images from normal X-ray and X-ray with pneumonia. There were a total of 31.6M data points. Whereas, for testing the model with a new dataset we took 3 images.
Pre-processing: In order to use the images for training, first they need to be read into an array. Now, to read it into an array we have to convert a jpeg image into a text file. This procedure is done using Python. Once we have the text file for training and testing, we need to read input from the text file and store it in a 1-D array.

Methodology

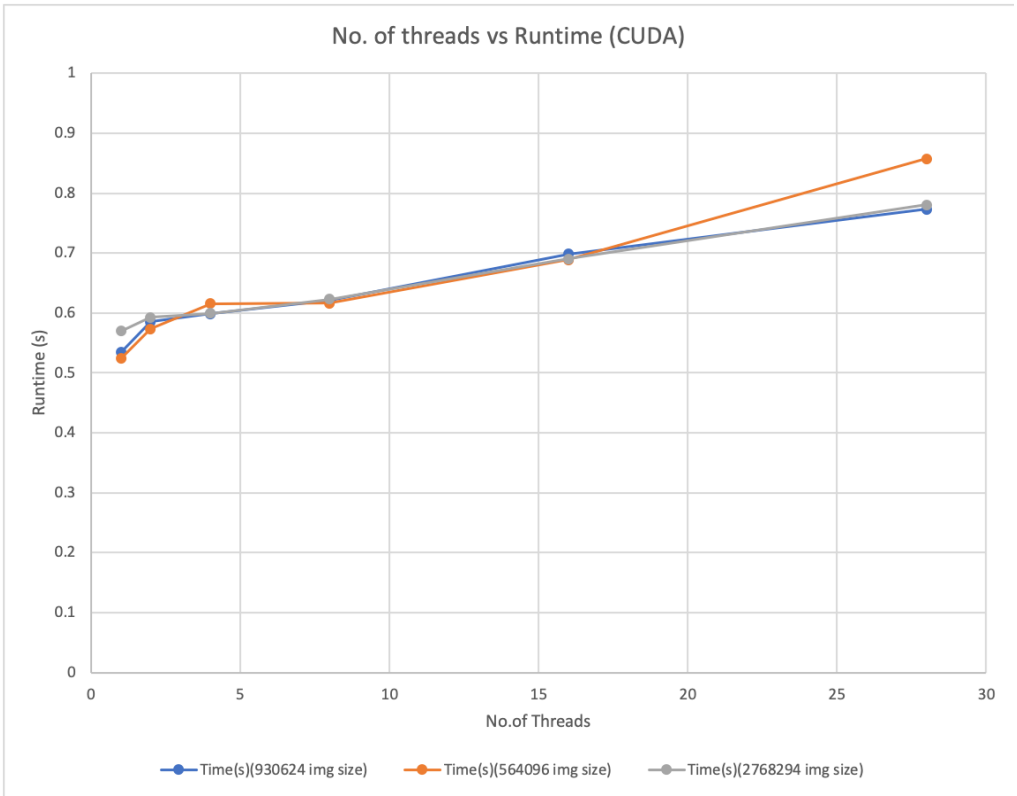
The experiment was done by requesting **56** cpus-per-task and **96GB** mem inside the Wave HPC system. The openMP code was tested on different numbers of threads. The number of threads considered for the experiment was 1, 2, 4, 8, 16, 28.

Measure of efficiency: The efficiency was measured by dividing the speedup obtained by the number of threads.

Graphs

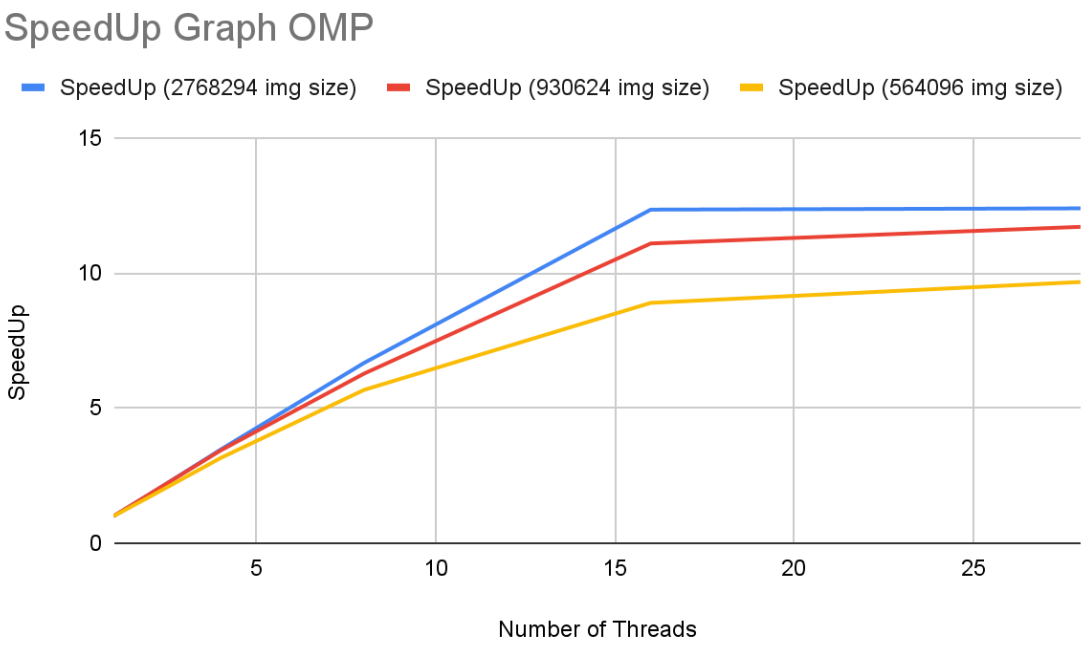


Graph 1: Time graph for OMP

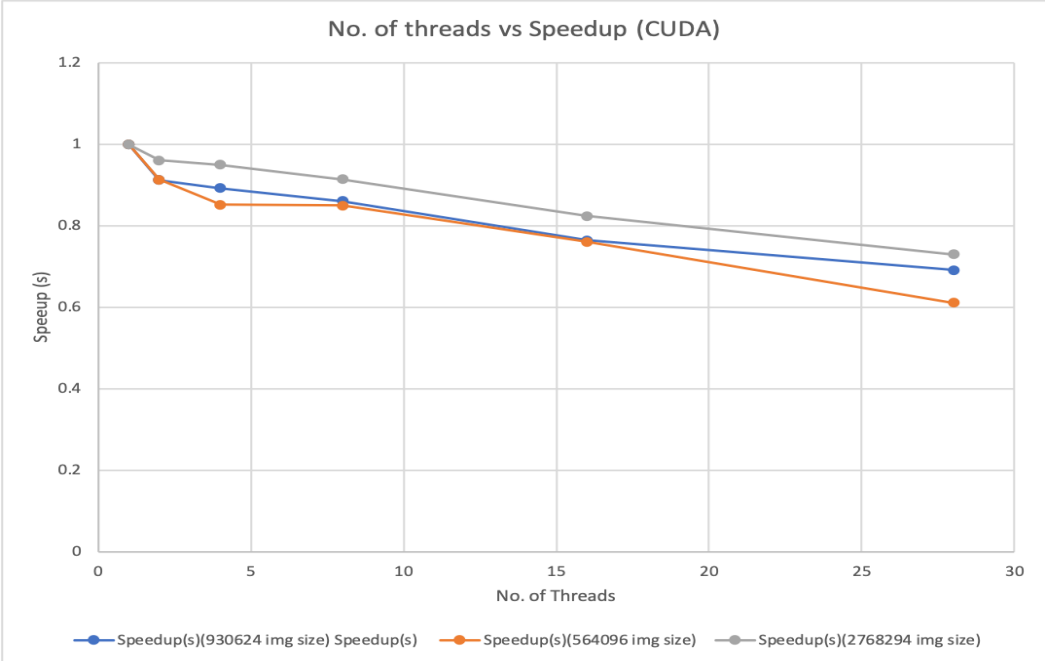


Graph 2: Time graph for CUDA

Efficiency/Speedup



Graph 3: Speedup graph for OMP



Graph 4: Speedup graph for CUDA

Analysis of results

In the OMP version, the testing part of the algorithm is parallelized and timed. Here as the number of threads increases the speedup increases and after a certain point it starts to decrease because of two scenarios. One is the work will be divided among too many threads and each thread has very little task to do then the process of creating and terminating those large blocks of threads swamps the useful work. Secondly as the number of threads increases, each thread may bring the entire row of its assigned chunk of the array to the cache for computation and there won't be enough space in cache memory which leads to an increase in cache miss rate thus increasing the run time.

Table 1: OMP timing table

Number of threads	Time(sec) (2768294 img size)	Time(sec) (930624 img size)	Time(sec) (564096 img size)
1	1.354276	0.47044	0.274192
2	0.751555	0.259453	0.160358
4	0.390319	0.13688	0.086833
8	0.202986	0.074964	0.04834
16	0.109754	0.042427	0.030842
28	0.109342	0.040201	0.028377

From the above table, as the size of the image increases, the runtime also increases because of the increase in number of iterations (refer Graph 1). This is because the total number of computations has increased.

From the strong scaling graph plotted above (Graph 3) it is seen that the program is not scalable (efficiency is not constant) for increasing number of threads and increasing test data size.

Each thread gets an equal chunk of the data points for computation except for the last thread which may be smaller than the other chunk sizes, thereby load balancing is achieved in this program.

In the CUDA version, the testing part of the algorithm is parallelized and timed. Here as the number of threads increases the speedup seems to decrease. This is because as the thread number goes up, fewer registers and also fewer shared memory per thread are available since we are using dynamically allocated memory.

Table 2: CUDA timing table

Number of threads	Time(sec) (2768294 img size)	Time(sec) (930624 img size)	Time(sec) (564096 img size)
1	0.53424	0.52413	0.56954
2	0.58518	0.57363	0.59261
4	0.59853	0.6153	0.59914
8	0.62067	0.61658	0.62278
16	0.69869	0.68861	0.69059
28	0.77306	0.8576	0.78026

From the above table, as the size of the image increases, the runtime decreases (refer Graph 2).

From the strong scaling graph plotted above (Graph 4) it is seen that the program is not scalable (efficiency is not constant) for increasing number of threads and increasing test data size.

Each thread gets an equal chunk of the data points for computation except for the last thread which may be smaller than the other chunk sizes, thereby load balancing is achieved in this program.

Discussion and Conclusion

Decisions made

1. In order to implement parallel programming for large datasets, we chose X-ray images to classify them into pneumonia or normal.
2. We were excited to implement CUDA framework because of its high performance therefore chose it as our second parallelising framework.

Difficulties faced

1. Segmentation fault because of global and local variables
2. Handling low accuracy for testing K-means algorithm
3. Taking input in C and preprocessing it
4. Handling global variables in CUDA
5. Handling global pointers in CUDA

Things that worked

1. The code runs in parallel using the OpenMP framework.
2. The code speeds up using the OpenMP framework.
3. Successful classification for testing was achieved.
4. CUDA : Successfully divided parallel blocks of code into global and device functions.
5. CUDA : Implementation of unified memory management for global variables, and shared memory for device exclusive variables.

Things that didn’t work

1. Taking images input parallelly in the same C code.
2. Improving the accuracy of the model.
3. Training with an even larger dataset.
4. CUDA : Handling unified memory (global) pointers
5. CUDA : Training

Conclusion

In this project, we learnt the implementation of two largely and most popularly used frameworks, OpenMP and CUDA. The OMP code could clearly classify each image into pneumonia and normal clusters. Hence, it was also able to classify test X-ray images into one of the two. By using the OMP framework, the code could run faster and could speed up to 12.3x. However, the same approach on CUDA gave better results for increasing problem size i.e larger images. But achieving speedup and accuracy in CUDA was difficult. Hence, we can conclude that the K-means algorithm when used with CUDA framework and OpenMP framework gives better performance and can make the model learn faster than the traditional approach.

Task Distribution

Break into components and explain

Components:

1. Pre-processing the dataset and forming a training and testing set.
2. Writing base code for K-means.
3. Parallelizing the base code using the OpenMP framework.
4. Analysing the results with different set sizes and number of threads for OpenMP.
5. Learning CUDA framework.
6. Implementation of CUDA on base code.
7. Analysing the results with different set sizes, number of threads and block size for CUDA.
8. Comparing results from different frameworks.
9. Writing a report.

10. Making a presentation.

Tasks Assigned and Completed:

1. Rohnie: Writing base code for K-means clustering, parallelising code using OMP, analysing the results and plotting the graph, learning CUDA, debugging the CUDA program, comparing the results of OMP and CUDA, PPT slides, report.
2. Juhi: Writing script for pre-processing, writing and debugging base line code, parallelising code using OMP, Analysing the results, learning CUDA, debugging the CUDA program, comparing the results of OMP and CUDA, PPT slides, report.
3. Piyush: Learning CUDA, Changing the code into CUDA, PPT slides, report.