Cairo University
Faculty of Engineering

Department of Computer Engineering

Parallel Computing

KNN & K-means

Team: #11

Name	Sec.	BN	ID	Email
Adham Ali	1	11	9202226	adham.sayed01@eng-st.cu.edu.eg
Abdelrahman Mohamed Hamza	1	36	9202793	abdelrahman.othman01@eng-st.cu.edu.eg

GPU Implementation:

K-means

Accelerating k-means clustering algorithm by applying triangle inequalities in the labeling step is based on the idea that many distance calculations in the labeling step may be redundant. For example, given a data point P and two clusters with centroids Ci and Cj . When labeling P, in the standard algorithm, two distance calculations are needed to compute d(P, Ci) and d(P, Cj). However, we can first compute d(P, Ci), and if the inequality d(Cj, Ci) > 2d(P, Ci) holds, then by triangle inequality, we can infer immediately that d(P, Cj) > d(P, Ci) without computing the d(P, Cj).

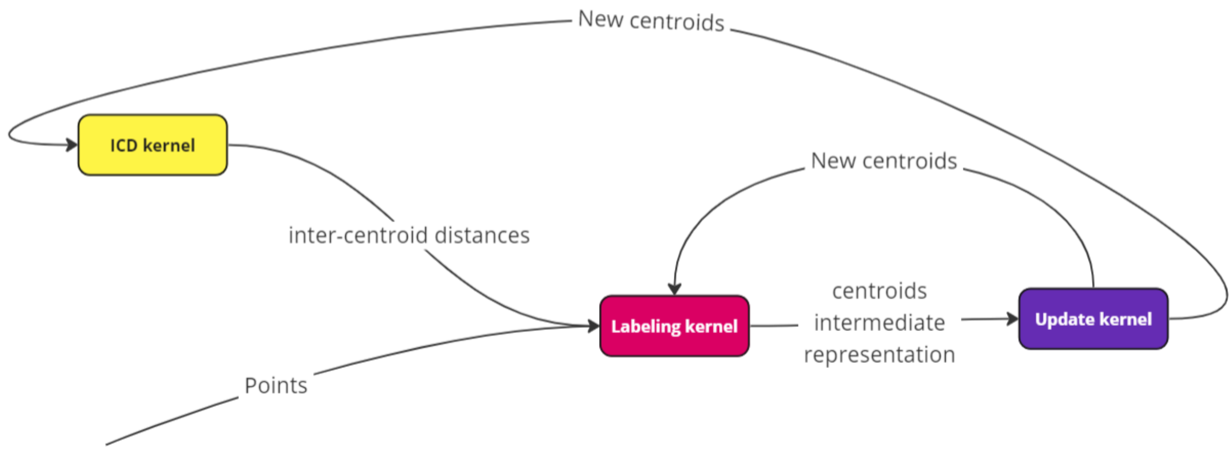
Here is the pseudocode of K-means labeling kernel:

$oldCnt \leftarrow L[i]$
$oldDist \leftarrow dist(P[i], C[oldCnt])$
$newCnt \leftarrow oldCnt$
for $j = 2$ to k **do**
    $curCnt \leftarrow RID[oldCnt][j]$
    if $ICD[oldCnt][curCnt] > 2 \times oldDist$ **then**
        break
    end if
    $curDist \leftarrow dist(P[i], C[curCnt])$
    if $curDist < newDist$ **then**
        $newDist \leftarrow curDist$
        $newCnt \leftarrow curCnt$
    end if
end for
$L[i] \leftarrow newCnt$

Kernels:

1. ICD kernel:
    a. Use shared memory to cache centroids(tiling).
    b. calculates inter-centroid distances matrix.
2. Labeling kernel:
    a. assign each point to the nearest centroid, accumulate the points assigned to the same centroid and count the number of points assigned to each cluster.
    b. Use shared memory to cache centroids as all threads in the same block use them also cache the ICD matrix for the same reason (tiling).
    c. Use privatization to accumulate points in the same cluster (histogram problem).
3. Update kernel:
    a. divide every element in centroids array by the corresponding count.

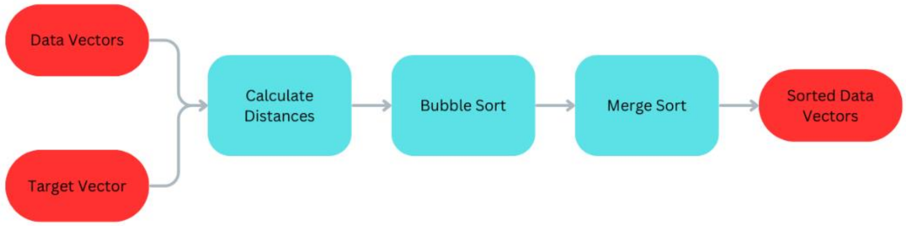
   b. Get the error of each element as the absolute difference between old and current value.
   c. Use reduction to accumulate errors in the same block (reduction problem).
   d. Use privatization to add all blocks error to the global memory location of error.
4. Repeat steps till convergence.



Constraints:

- The centroids matrix should be fitted in shared memory.
- The number of clusters must not exceed 32 because the ICD kernel is one block with dimension $k \times k$

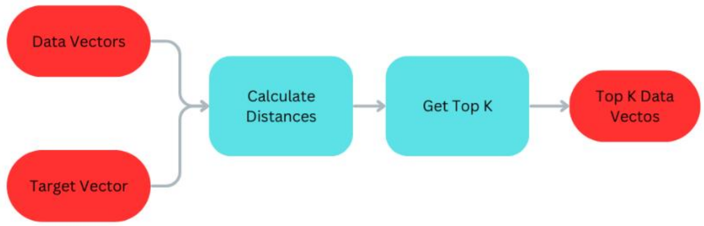
KNN Method 1



Explanation

- First, I calculate the distance between each vector and the target vector so that I don't need to calculate it every time.
- Then I perform a partial sorting using bubble sort, so the array is transformed into an array of sorted segments.
- Then I merge these segments iteratively using the merge sort algorithm in which each iteration i merge two sorted segments.
- How is the merging done?
    - Each block can merge only two sorted segments or a part of the first sorted segment and a part of the second sorted segment, it cannot merge more than that.
    - Each block must calculate the start index to merge the results K, the start index to start merge in the first array I, and the start index to start merge in the second array J.
        - $iOffeset < I < iOffset + n$
        - $jOffeset < J < jOffset + m$
        - $k \;=\; I - J + jOffset$
        - $0 < J - jOffset < m$
        - $0 < K - I + jOffset - jOffset < m$
        - $0 < K - I < m$
        - $-K < -I < m \;-\; K$
        - $K > I > K \;-\; m$
        - $iHigh \;=\; \min(K, iOffset + n)$
        - $iLow \;=\; \max(K - m, iOffset)$
    - Then I calculate using binary search, and compute J from it.

KNN Method 2



Explanation

- First, I calculate the distance between each vector and the target vector so that I don't need to calculate it every time.
- Then I launch threads so that each thread is responsible for a segment of data, the segment size should be more than K.
- Each thread gets the top K elements in its segment and then adds it to the output array.
- Then in the next iteration I make the input array to be the output one and the output array to be the input one.

Performance analysis:

How much is the speedup of the GPU over the CPU?

K-means

Time complexity on CPU:

Assume that our parameters are

- n: the number of data points.
- k: the number of clusters.
- d: which is the dimension of the data.

Labeling

- Time O($n\,dk$)
- Memory O(n)

Update centroids

- Time O($n\,d + k\,d$) for n>>d O($n\,d$)
- Memory O(k)

Time complexity on GPU:

Assume that our parameters are

- n: the number of data points.
- k: the number of clusters.
- d: which is the dimension of the data.
- blockDim.x is the number of threads per block in x axis.
- Nb is the number of blocks.

ICD kernel:

- Time: O($d + \frac{d}{k}$)
- Memory: O($k \times k$) (shared memory)
- Note: this kernel always runs in 1 block mode with $k \times k$ block dimension.

Labeling kernel:

- Time: O(dk)
- Memory: O($Nb(k \times k + k \times d)$) (shared memory for ICD matrix and centroids)
- Note: the number of threads per block can be changed to match the host device specifications, the labeling kernel here does atomic addition as pre-step to update centroids.

Update kernel:

- Time: O($\log_2 blockDim.x$)

- Memory: O($Nb(k \times k + k \times d)$) (shared memory for ICD matrix and centroids)
- Note: the reduction step runs to calculate error to check convergence.

Theoretical Speedup

$$theoretical\ speedup = \frac{n\,dk + n\,d + k\,d}{d + \frac{d}{k} + dk + \log_2 blockDim.x} = \frac{1M \times 16 \times 32 + 1M \times 16 + 32 \times 16}{16 + \frac{16}{32} + 16 \times 32 + \log_2 256}$$
$$= 984157.5247$$

Actual Speedup

CUDA K-means for 300 iterations:

```
Time taken by CUDA K-means: 2.435352 seconds
```

CPU K-means for 300 iterations:

```
Time taken by CPU K-means: 657 seconds
```

$$actual\ speedup = \frac{CPU\ time}{GPU\ time} = \frac{657}{2.435352} = 269.776$$

KNN

Time complexity on CPU:

Assume that our parameters are

- n: the number of data points.
- k: the number of nearest Neighbours
- d: which is the dimension of the data

Brute force method

The time complexity of KNN on CPU can be **O(nkd)** or **O(nd + nk)**

1. For O(nd+nk):
   a. We first calculate the distance between the target vector and all dataset which is O(nd)
   b. Then we loop over these distances k times to select the minimum value.
2. For O(nkd):
   a. We loop over the whole dataset k times and in each time we calculate the distance with the target vector so it is O(nkd)

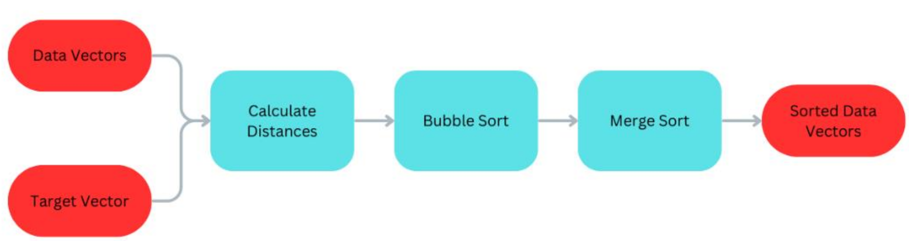
The difference is that we have a memory complexity in the first approach with O(n) for storing the distances.

Time complexity on GPU:

Assume that our parameters are

- n: the number of data points.
- k: the number of nearest Neighbours
- d: which is the dimension of the data
- t: number of threads
- Bt: number of threads per block
- S: min sorted segment

Method 1



Calculate Distances

- Time: O($\frac{n}{t} \times d$)

- Memory: O (1)

If we assume that $t = \infty$

- Time: O(d)
- Memory: O (1)

Bubble Sort

- Time: O(s^2)
- Memory: O (1)

If we assume that $t = \infty$

- Time: O (s^2)
- Memory: O (1)

Merge Sort

- Time: O$\left(\log_2(n) \cdot \left(\log_2\left(\frac{n}{t}\right) + \frac{n}{t}\right)\right)$
- Memory: O (Bt)

If we assume that $t = \infty$

- Time: O $(\log_2(n))$
- Memory: O (Bt)

Total Time Complexity

- O$\left(\log_2(n) \cdot \left(\log_2\left(\frac{n}{t}\right) + \frac{n}{t}\right) + s^2\right)$
- For $t = \infty$: O$(\log_2 ⬚ n + d)$

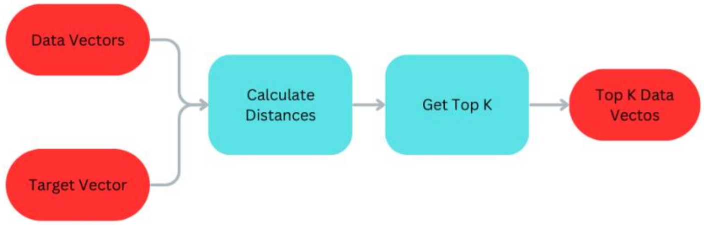
Memory Complexity

- O(Bt)
- For $t = \infty$: O(Bt)

Theoretical Speedup

- $\frac{nd+nk}{\left(\log_2(n)\cdot\left(\log_2\left(\frac{n}{t}\right)+\frac{n}{t}\right)\right)}$

Method 2



Calculate Distances

- Time: O($\frac{n}{t} \times d$)
- Memory: O (1)

Get Top K

- Time: O($\log_{\frac{s}{k}}(n) \cdot s$)
- Memory: O (n)

Total Time Complexity

- O$\left(\left(\log_{\frac{s}{k}}(n) + d\right) \cdot s\right)$

Total Memory Complexity

- O(n)

Theoretical Speed Up

- $\frac{nd+nk}{\left(\log_{\frac{s}{k}}(n)+d\right)\cdot s}$
- $\frac{10^7\cdot(10+5)}{\left(\log_{\frac{2\cdot 5}{5}}(10^7)+10\right)\cdot 2\cdot 5}$ = 451080.38 For d = 10, k = 5 n= 10^7

How do your GPU results compare to open-source peers of the same features?

Our K-means at 1 000,000 points:

Clusters \ Features	2	16	256
4	0.353514 s	1.115973 s	20.048042 s
16	0.272245 s	1.490182 s	46.201309 s
32	0.450959 s	1.943135 s	74.722580 s

kmeans_cuda from libKMCUDA at 1 000,000 points:

Clusters \ Features	2	16	256
4	3.77 s	1.96 s	26.2 s
16	3.67 s	5.62 s	46 s
32	3.4 s	14.9 s	61 s

Our CPU K-means at 1 000,000 points:

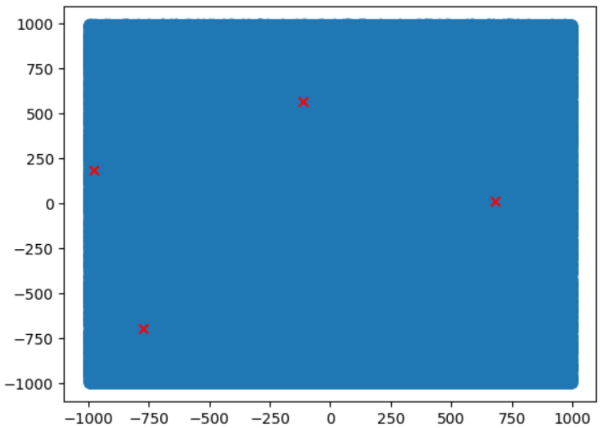
K = 32, d = 16 ➔ 657 s

K = 4, d = 2 ➔ 4 s

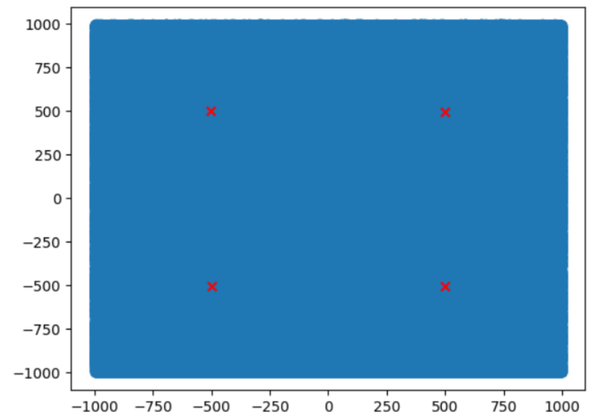
K = 16, d = 2 ➔ 65 s

Notes:

- This library works better when the dimensions are very large which exceeds Google Collaboratory memory so in dimensions (32,256) the library is better than my implementation but for other dimensions my implementation is almost the best.
- When the dimensions are very small there is overhead in my implementation and library so in that case the CPU is the best choice over those.



Before



After

Profiling of two large dimensions.

```
==2189== NVPROF is profiling process 2189, command: ./main /content/kmcuda/testcases/testcase01.txt /content/kmcuda/testcases/result.txt
Done
Time taken by CUDA K-means: 74.722580 seconds
==2189== Profiling application: ./main /content/kmcuda/testcases/testcase01.txt /content/kmcuda/testcases/result.txt
==2189== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.56%  74.0452s       300  246.82ms  234.48ms  332.35ms  labelingKernel(float*, float*, float*, int*, int*, float*, int, int, int)
                    0.34%  251.49ms         3  83.829ms  4.9920us  251.48ms  [CUDA memcpy HtoD]
                    0.09%  68.333ms       300  227.78us  203.71us  469.59us  ICDKernel(float*, float*, int, int)
                    0.00%  2.4063ms       302  7.9670us  1.3120us  1.9629ms  [CUDA memcpy DtoH]
                    0.00%  1.2487ms       300  4.1620us  3.7760us  4.6400us  updateKernel(float*, int*, float*, float*, int, int)
                    0.00%  898.74us       900     998ns     416ns  3.5200us  [CUDA memset]
      API calls:   99.06%  74.1575s      1200  61.798ms  1.2100us  332.37ms  cudaDeviceSynchronize
                    0.37%  276.62ms       900  307.35us  5.4300us  258.50ms  cudaLaunchKernel
                    0.36%  269.00ms       305  881.95us  26.812us  255.01ms  cudaMemcpy
                    0.19%  144.04ms         2  72.022ms  1.4830us  144.04ms  cudaEventCreate
                    0.01%  7.7679ms       900  8.6300us  2.7440us  45.960us  cudaMemset
                    0.00%  2.9405ms         7  420.07us  4.1400us  2.6692ms  cudaMalloc
                    0.00%  2.1556ms         6  359.26us  3.9410us  1.0742ms  cudaFree
                    0.00%  791.11us         2  395.56us  12.133us  778.98us  cudaEventRecord
                    0.00%  212.67us       114  1.8650us     187ns  82.182us  cuDeviceGetAttribute
                    0.00%  12.616us         1  12.616us  12.616us  12.616us  cuDeviceGetName
                    0.00%  8.8940us         1  8.8940us  8.8940us  8.8940us  cuDeviceGetPCIBusId
                    0.00%  7.6940us         1  7.6940us  7.6940us  7.6940us  cudaEventSynchronize
                    0.00%  5.1770us         1  5.1770us  5.1770us  5.1770us  cuDeviceTotalMem
                    0.00%  2.9530us         1  2.9530us  2.9530us  2.9530us  cudaEventElapsedTime
```

```
==5478== NVPROF is profiling process 5478, command: ./main /content/kmcuda/testcases/testcase01.txt /content/kmcuda/testcases/result.txt
Done
Time taken by CUDA K-means: 46.201309 seconds
==5478== Profiling application: ./main /content/kmcuda/testcases/testcase01.txt /content/kmcuda/testcases/result.txt
==5478== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.48%  45.9123s       300  153.04ms  145.62ms  225.93ms  labelingKernel(float*, float*, float*, int*, int*, float*, int, int, int)
                    0.49%  227.40ms         3  75.801ms  3.0400us  227.40ms  [CUDA memcpy HtoD]
                    0.02%  9.8547ms       300  32.848us  28.895us  72.735us  ICDKernel(float*, float*, int, int)
                    0.00%  2.2093ms       302  7.3150us  1.2480us  1.7898ms  [CUDA memcpy DtoH]
                    0.00%  1.0526ms       300  3.5080us  3.1680us  5.1200us  updateKernel(float*, int*, float*, float*, int, int)
                    0.00%  823.35us       900     914ns     384ns  3.4560us  [CUDA memset]
      API calls:   99.21%  45.9339s      1200  38.278ms  1.1700us  225.94ms  cudaDeviceSynchronize
                    0.52%  240.99ms       305  790.13us  16.092us  227.64ms  cudaMemcpy
                    0.22%  100.12ms         2  50.058ms  1.3480us  100.11ms  cudaEventCreate
                    0.03%  12.236ms       900  13.595us  4.4640us  471.11us  cudaLaunchKernel
                    0.01%  6.1547ms       900  6.8380us  2.5360us  23.653us  cudaMemset
                    0.00%  2.0480ms         6  341.33us  3.4070us  1.0672ms  cudaFree
                    0.00%  1.9148ms         7  273.54us  4.5230us  1.5865ms  cudaMalloc
                    0.00%  139.89us       114  1.2270us     134ns  55.958us  cuDeviceGetAttribute
                    0.00%  27.440us         2  13.720us  10.281us  17.159us  cudaEventRecord
                    0.00%  11.138us         1  11.138us  11.138us  11.138us  cuDeviceGetName
                    0.00%  6.9030us         1  6.9030us  6.9030us  6.9030us  cudaEventSynchronize
                    0.00%  4.7830us         1  4.7830us  4.7830us  4.7830us  cuDeviceGetPCIBusId
                    0.00%  4.4550us         1  4.4550us  4.4550us  4.4550us  cuDeviceTotalMem
```

KNN

Kernel 1

	N=10k	N=100k	N= 1M	N=10M
D=10	10ms	88ms	8821ms	4.5s
D=5	10ms	84ms	755ms	4.4s
D=2	10ms	80ms	690ms	4.2s

Kernel 2

	N=10k	N=100k	N=1M	N= 10M
D=10	6.1399ms	16.474ms	146.65ms	1.1s
D=5	6ms	15ms	194ms	870ms
D=2	5ms	15ms	139ms	790ms

NumPy

	N=10k	N=100k	N=1M	N= 10M
D=10	61.2 ms	0.025412s	5.86 s	2.961s
D=5	54 ms	923 ms	7.59 s	1min 26s
D=2	54.5 ms	566 ms	6.82 s	1min 23s

Pytorch

	N=10k	N=100k	N=1M	N= 10M
D=10	1.62 ms	434 ms	79 ms	87.3ms
D=5	726 µs	4.58 ms	88.3 ms	576 ms
D=2	1.28 ms	4.83 ms	49.2 ms	482 ms

CPU

	N=10k	N=100k	N=1M	10M
D=10	1e-06s	10.2 ms	0.246787s	3.1s
D=5	0.001525s	0.025672s	0.158101s	1.59311s
D=2	0.000888s	0.009767s	0.070262s	0.701493s

References

1. https://github.com/chrisjmccormick/brute_knn_benchmarks
2. https://docs.dgl.ai/en/latest/api/python/knn_benchmark.html
3. https://www.kernel-operations.io/keops/_auto_benchmarks/benchmark_KNN.html
4. https://www.diva-portal.org/smash/get/diva2:861804/FULLTEXT01.pdf
5. https://ieeexplore.ieee.org/document/6009040
6. https://github.com/src-d/kmcuda