GPU K-means

University of Trento

Pasquali Thomas, Stefano Da Roit

Department of Computer Science

September, 2023

K-means

The K-means algorithm is a local search optimization method. It seeks to find a partition $V_1, \ldots, V_K$ of k circular sets with centroids $c_1, \ldots, c_K$ which minimizes the sum of the squared euclidean distance between $p_i \in \mathbb{R}^d$ and the centroid of the set to which it is assigned:

$$\min \left(\sum_{j=1}^{k} \sum_{p_i \in V_k} \|p_i - c_j\|^2 \right) \tag{1}$$

K-means algorithm

```
Data: Points $P = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^d$, an integer $k < n$, an integer maxiter $(m)$
Result: A partition $\{V_1, ... V_k\}$ with $V_i \subset P$ of disjoint nonempty clusters
Initialize $k$ centroids $c_j$ by sampling random points from $P$ while $c_j$ is changing and
 $iter < maxiter$ do
    $V_i = \emptyset \quad \forall 1 \leq i \leq k$ for $p_i \in P$ do
        $k = \arg\min_{1 \leq j \leq k} \|p_i - c_j\|^2$ add $p_i$ to $V_j$
    end
    compute new centroids: $(c_j)_i = \frac{1}{V_j} \sum_{p \in V_j} (p)_i \quad \forall 1 \leq i \leq d$
end
```

Algorithm 1: K-means pseudo algorithm

Algorithm overview

The program receives as input a set of points $P = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^d$, a number of requested clusters K, and an integer representing the maximum number of iterations *maxiter*. After the initialization of the centroids, the execution will loop until: (a) the algorithm converges (i.e. the centroids do not change any more) or (b) *maxiter* is reached.

Algorithm steps

1. Computation of the distances between the points and each centroid;
2. Identification of the centroid with the lowest distance for each point. Every point is matched to the cluster of the closest centroid (i.e. updating the partition $V_1, ..., V_K$);
3. Computation of the new centroids;
4. Checking if centroids have changed or *maxiter* is reached.

Centroids initialization

Before starting the main loop, a set of k distinct points is **randomly** sampled from the input P. The set represents the initial location of the centroids.

This step is critical

The approach of choosing the initial centroids randomly may lead the algorithm to converge at different iterations (or not converge at all) in executions with the same input.

Distances goal

The goal of this step is to compute the distance of points from each centroid.

$$d^2(x_i, c_j) = \|x_i - c_j\|^2 = (x_{i_1} - c_{j_1})^2 + \dots + (x_{i_d} - c_{j_d})^2 \qquad (2)$$

Single-warp approach

This kernel uses a warp to compute the distance between a point and a centroid.

- The reduction is performed using the shuffle primitive;
- Grids are scheduled in 2D; the x index of the block identifies the index of the point, while y the index of the cluster. The thread index is related to the dimensionality.

Issues

- For small values of d, the number of active threads is very low;
- Cannot automatically handle $d > warpSize$.

Single-warp approach optimization

Idea

Try to fit as many points into a warp as possible.

- The maximum number of points per warp is: $mpp = \frac{warpSize}{nextpow2(d)}$;
- This increases the number of active thread;
- Still does not handle $d > warpSize$.

Handle $d > warpSize$

To overcome the limit of the warp size, the kernel is invoked multiple times, and each time reducing a *warpSize* size block of dimensions.

Nearest centroids identification goal

The goal of this step is to find the closest centroid for each point by the comparison of the just computed distances.

$$\forall p_i \in P \qquad k = \underset{1 \leq j \leq k}{\arg\min} \left\| p_i - c_j \right\|^2 \tag{3}$$

Nearest centroids identification

The kernel:

1. Allocates a block for each point;
2. Each block reduces its warps using the shuffle primitive;
3. The result of each reduction is a *Pair*: a simple *struct* which contains the index of a centroid and its distance to the point;
4. All these *Pairs* are stored in shared memory to be then reduced to a single one;
5. The resulting *Pair* indicates to which cluster the point must be associated.

New centroids goal

To compute the new centroid c_j for a cluster V_j, all the points p_i assigned to it are summed up and then scaled by the cardinality of points assigned to that cluster V_j.

$$(c_j)_i = \frac{1}{V_j} \sum_{p \in V_j} (p)_i \quad \forall i,\ 1 \leq i \leq d \tag{4}$$

New centroids computation

The kernel responsible for this task:

1. Is scheduled in 2D for both grid and block;
2. Each grid is responsible of computing the new centroid of a single cluster;
3. Each block performs the reduction on a subset of points.

The block's x, y indexes are used to identify respectively the index of the point and the index of the dimensionality.

Overcome the limit of 1024 threads per block

The 2nd dimension of the grid is incremented by powers of 2 to manage large number of points.

Overcome the limit of 32 threads in a warp

The kernel is launched many times; each invocation performs the reduction on a subset of 32 of the dimensionality: $n\_invoc = \frac{d-1}{warpSize} + 1$.

Convergence check

At the end of each iteration the convergence of the algorithm is checked by the comparison between the new computed centroids and the ones used in the iteration for the clusters assignment.

1. For each pair (old, new) of centroids the euclidean distance is computed;
2. **If** for each cluster the distance is lower than a set tolerance parameter $\Rightarrow$ *exit*;
3. **Else** move one with the next iteration.

Distances as a matrix multiplication

Consider a given centroid $c = (x_c, y_c, z_c, ..., d_c)$ and a generic point $p = (x, y, z, ..., d)$, the euclidean distance $d^2(p, c)$ is equal to:

$$\begin{aligned} (x - x_c)^2 + ... + (d - d_c)^2 = \\ = x^2 - 2xx_c + x_c^2 + ... + d^2 - 2dd_c + d_c^2 \end{aligned} \tag{5}$$

We can build the centroid associated matrix for the formula:

$$C = \begin{bmatrix} x_c^2 + ... + d_c^2 & -x_c & -y_c & -z_c & ... & -d_c \\ -x_c & 1 & 0 & 0 & 0 & 0 \\ -y_c & 0 & 1 & 0 & 0 & 0 \\ -z_c & 0 & 0 & 1 & 0 & 0 \\ ... & 0 & 0 & 0 & 1 & 0 \\ -d_c & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Let $q = \begin{bmatrix} 1 & x & y & z & ... & d \end{bmatrix}$, then we have:

$$d^2(p, c) = q \cdot C \cdot q^T \tag{6}$$

Formula extension

Eq. (6) can be extended to compute multiple distances in one operation:

Final formula

Let C be the associated matrix for a given centroid c, $Points = [p_1, p_2, \dots, p_n]$ a set of points.
Let:

$$P = \begin{bmatrix} 1 & p_{1x} & p_{1y} & \dots & p_{1d} \\ 1 & p_{2x} & p_{2y} & \dots & p_{2d} \\ & & \dots & & \\ 1 & p_{nx} & p_{ny} & \dots & p_{nd} \end{bmatrix}$$

By applying (6) we obtain:

$$P \cdot C \cdot P^T = \begin{bmatrix} d^2(p_1, c) & & & \\ & d^2(p_2, c) & & \\ & & \ddots & \\ & & & d^2(p_n, c) \end{bmatrix} \tag{7}$$

Compute new centroids as a matrix multiplication

Let's define the results of *nearest centroids identification* as follows:

$$pc = [x_1, x_2, \ldots, x_n] \qquad 0 < x_i \leq k \text{ (points-clusters)}$$
$$pc\_len = [l_1, l_2, \ldots, l_k] \qquad 0 < l_i < (n - k) \text{ (cluster-lengths)}$$

Let M be a $k \times n$ matrix built as follows:

$$\begin{cases} m_{ij} = 1 & \text{if } i = x_j,\ 0 < j \leq n \\ m_{ij} = 0 & \text{otherwise} \end{cases} \quad \text{for } 0 < i \leq k \tag{8}$$

Compute new centroids as a matrix multiplication [cont.]

The new centroids C are computed as follows:

$$C^{tot} = M \cdot \begin{bmatrix} c_{1x} & c_{1y} & \dots & c_{1d} \\ c_{2x} & c_{2y} & \dots & c_{2d} \\ & & \dots & \\ c_{kx} & c_{ky} & \dots & c_{kd} \end{bmatrix} \tag{9}$$

$$C = \begin{bmatrix} c^{tot}_{1x}/l_1 & c^{tot}_{1y}/l_1 & \dots & c^{tot}_{1d}/l_1 \\ c^{tot}_{2x}/l_2 & c^{tot}_{2y}/l_2 & \dots & c^{tot}_{2d}/l_2 \\ & & \dots & \\ c^{tot}_{kx}/l_k & c^{tot}_{ky}/l_k & \dots & c^{tot}_{kd}/l_k \end{bmatrix} \tag{10}$$

Example

$$pc = [0,0,1,2], pc\_len = [2,1,1] \implies M = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Limitations/Considerations

- The provided implementation has some limitations. In particular, the kernel *clusters_argmin_shfl* is bounded to $k \leq 1024$ (maximum block x dimension).
- The approach of computing distances as a matrix multiplication showed poor performances.
- By default, the GPU performs floating point computations using the Fused Multiply-Add operator since it has high performance and increases the accuracy of computations. This operator is not available on the CPU, so there can be differences comparing the numerical results.

References

To validate our solution we compared it with some other implementations, in particular:

- KMeans (scikit-learn) [1] $\longrightarrow$ one of the most common sequential python implementations of the algorithm provided by *scikit-learn* library.
- kcuda [2] $\longrightarrow$ CUDA parallel implementation of k-means based on *"Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup"* [3]
- cpukmeans $\longrightarrow$ "silly" sequential implementation of k-means algorithm created specifically for these tests. It does not provide any optimization in order to give an idea about the efficiency of the other competitors.

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`
[2] `https://github.com/src-d/kmcuda`
[3] `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ding15.pdf`

Sperimental setup

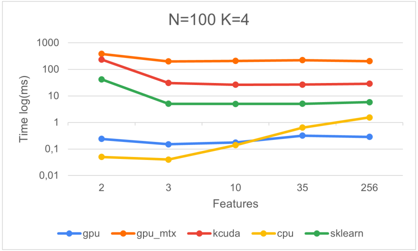
To fairly compare the different implementations, we needed to run all of them with the same input parameters and conditions:

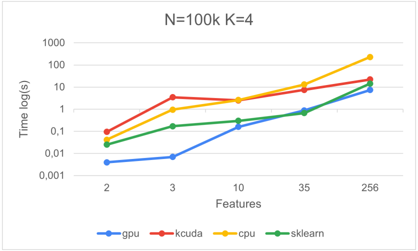
- Random sampling of observations for initial centroids;
- Same seed for the random initialization of centroids;
- Set a common tolerance value of $1 \cdot 10^{-5}$;
- Times are obtained with the average of 5 consecutive runs.

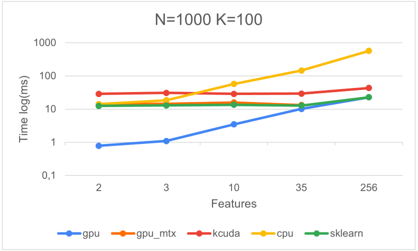
Datasets

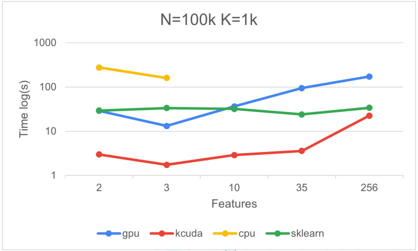
The datasets generated have $n \in \{10^2, 10^3, 10^5\}$ number of observations and $d \in \{2, 3, 10, 35, 256\}$ number of features. The goal is to compare the behavior of our solution with different input sizes: $n \times d$.

Results


N=100 K=4
Time log(ms)
1000
100
10
1
0,1
0,01
2
3
10
35
256
Features
gpu
gpu_mtx
kcuda
cpu
sklearn


N=100k K=4
Time log(s)
1000
100
10
1
0,1
0,01
0,001
2
3
10
35
256
Features
gpu
kcuda
cpu
sklearn


N=1000 K=100
Time log(ms)
1000
100
10
1
0,1
2
3
10
35
256
Features
gpu
gpu_mtx
kcuda
cpu
sklearn


N=100k K=1k
Time log(s)
1000
100
10
1
2
3
10
35
256
Features
gpu
kcuda
cpu
sklearn

Conclusions

Considerations

- Great performances with low clusters and features;
- Poor performances with high values of k;
- The gemm distances function is not optimized because the implementation has been implemented when the project structure was already defined.

Future work

- Avoid to copy the centroids on the host at each iteration, and perform the comparison directly on the device with a kernel (for small $k \cdot d$ the comparison can be carried out still on the host);
- Optimize and remove limitations from argmin kernel;
- Re-implement the algorithm using the alternative strategies (considering to exploit sparse matrix optimizations);