Porting the K-means algorithm on GPU

Da Roit Stefano, Pasquali Thomas

Abstract

K-means is a popular unsupervised machine learning algorithm used for clustering. Its goal is to partition data into clusters in such a way that data points within the same cluster are more similar to each other than to those in other clusters.

The goal of the project is the design, development and testing of a K-means algorithm CUDA implementation for Nvidia GPU architectures. First of all, we decomposed the algorithm in different steps to choose which part were suitable for parallel computation and then we tried to implement the related kernels. After performing some local tests on the correctness of the implementation, we proceeded with a test campaign on a HPC cluster to compare our solution with other competitors, among which a very common used python implementation executed on CPU and another CUDA solution. The results highlighted that our solution performs better than others when the number of clusters requested is low; when the value increases the program decrease the performance.

The code is available in a GitHub repository. [1]

1 Clustering problem

The K-means algorithm is a local search optimization method. It seeks to find a partition $V_1, ..., V_K$ of k circular sets with centroids $c_1, ..., c_K$ 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:

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

```
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

[1] `https://github.com/ThomasPasquali/GPU_Kmeans`

2 State of the art

K-means has been around for a long time, for this reason there are many implementations and enhancements available both for CPU and GPU. Many of them are included in modern machine learning libraries. Nowadays the implementations of K-means algorithm are slightly different from the simplified version described by algorithm 1. In particular, the main improvements are related to the choice of the initial centroids, for example through the innovative technique provided by *K-means++* which yields better cluster quality and faster convergence.

One of the most common implementation for CPU is scikit-learn, a popular machine learning library in Python, which provides a robust implementation of K-means clustering. It offers flexibility in terms of customization and parameter tuning. Also several libraries and frameworks have been developed to perform K-means clustering on GPUs efficiently. CUDA, cuML (part of NVIDIA Rapids), and TensorFlow GPU support are some examples. These libraries provide optimized GPU implementations of the K-means algorithm, making use of parallel processing capabilities.

3 Solution

The implementation is based on algorithm 1. 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.

Each iteration consists of the following tasks:

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.

3.1 Details on implementation

In the following section is provided a description for each main task of the implementation. The choice of centroids and the convergence check are performed on the host; the others operations are executed on device.

1. 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. 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.

2. Computation of distances

The goal of this step is to compute the distance of points from each centroid. For this purpose, we exploited the notion of squared Euclidean distance metric which makes use of the same equation as the Euclidean distance metric, but it does not take the square root. This allows to perform clustering at a faster pace. The task performed is the following computation:

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

We implemented three different kernels to compute distances in order to reach a better optimization of this operation.

2.a Warp approach

The kernel *compute_distances_one_point_per_warp* uses a warp to compute the distance between a point and a centroid. The reduction is performed at warp level by 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. To overcome the limit of the warp size, in case $d > 32$, the kernel is invoked multiple times, and each one reduces a 32 size block of dimensions.

The main problem of this kernel is that for small values of d, the number of active threads is very low.

2.b Warp approach optimized

This kernel is based on the principles of the previous one, but it solves the problem of low active threads. The main difference is that this kernel try to fit as many points into a warp as possible, the maximum number of points per warp is: $mpp = \frac{warpSize}{nextpow2(d)}$. Grids are always scheduled in 2D: the x index of the block matched with the index of the thread identifies the point, and y index of the block, the cluster. The thread index is also related to the dimensionality.

The main drawback of this solution is that the kernel is bounded to $d \leq 32$.

2.c Matrix multiplication approach

An alternative approach to previous two is computing the Euclidean square distances as a product of matrices. 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:

$$\begin{aligned}(x - x_c)^2 + ... + (d - d_c)^2 = \\ = x^2 - 2xx_c + x_c^2 + ... + d^2 - 2dd_c + d_c^2\end{aligned} \quad (3)$$

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:

$$d^2(p, c) = q \cdot C \cdot q^T \quad (4)$$

2.c (bis) Formula extension

In (4), q is a vector that represents a single point. This equation can be extended to compute multiple distances in one operation:

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

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

By applying (4) we obtain:

$$P \cdot C \cdot P^T = \begin{bmatrix} d^2(p_1, c) & & & \\ & d^2(p_2, c) & & \\ & & \ddots & \\ & & & d^2(p_n, c) \end{bmatrix} \quad (5)$$

This idea has been implemented using cuBLAS library. Before starting the algorithm main loop, the associated matrices for all points are computed using the kernel *compute_point_associated_matrices* (shuffle reduction); furthermore the centroids are augmented with a column of $1s$. The function *compute_gemm_distances* will then compute the distances as described above.

We have chosen to compute the associated matrices for points instead of centroids because points are fixed for the whole execution, while centroids change after every iteration.

3. Nearest centroids identification (*argmin*)

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

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

The output of this kernel is an array of size n which stores the association point-cluster and an array of size k which stores the size of each cluster.

4. New centroids computation

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.

$$(c_j)_i = \frac{1}{V_j} \sum_{p \in V_j} (p)_i \quad \forall i,\ 1 \le i \le d \tag{6}$$

The kernel responsible for this task is *compute_centroids_shfl* which performs a warp-level reduction for each dimensionality component of the points. For each warp the first thread is responsible to scale and write the sum in global memory. The kernel is scheduled in 2D for both grid and block; each grid is responsible of computing the new centroid of a single cluster, while 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.

To 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; to overcome the limit of 32 threads in a warp, which affects the dimensionality, the kernel is launched many times; each invocation performs the reduction on a subset of 32 of the dimensionality: $n\_invoc = \frac{d-1}{warpSize} + 1$.

4. (bis) Matrix multiplication approach (not implemented)

An alternative approach to compute new centroids could be to express it as a matrix multiplication:
Let $pc = [x_1, x_2, \ldots, x_n]$ with $0 < x_i \le k$ (points-clusters), $pc\_len = [l_1, l_2, \ldots, l_k]$ with $0 < l_i < (n-k)$ (cluster-lengths) be the results of the previous step (argmin). Let M be a $k \times n$ matrix built as follows:

$$\begin{cases} m_{ij} = 1 & \text{if } i = x_j,\ 0 < j \le n \\ m_{ij} = 0 & \text{otherwise} \end{cases} \quad \text{for } 0 < i \le k \tag{7}$$

For example:

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

The new centroids C are computed as follows:

$$C^{tot} = M \cdot \begin{bmatrix} c_{1x} & c_{1y} & \cdots & c_{1d} \\ c_{2x} & c_{2y} & \cdots & c_{2d} \\ & & \cdots & \\ c_{kx} & c_{ky} & \cdots & c_{kd} \end{bmatrix} \tag{8}$$

$$C = \begin{bmatrix} c^{tot}_{1x}/l_1 & c^{tot}_{1y}/l_1 & \cdots & c^{tot}_{1d}/l_1 \\ c^{tot}_{2x}/l_2 & c^{tot}_{2y}/l_2 & \cdots & c^{tot}_{2d}/l_2 \\ & & \cdots & \\ c^{tot}_{kx}/l_k & c^{tot}_{ky}/l_k & \cdots & c^{tot}_{kd}/l_k \end{bmatrix} \tag{9}$$

This idea has **NOT been implemented** because the matrix M is sparse: the operation would not be suitable without specific techniques to perform sparse matrix multiplication.

5. 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. For each pair (old, new) of centroids the euclidean distance is computed; if for each cluster the distance is lower than a set tolerance parameter, the program will exit.

3.2 Limitations/Considerations

The provided implementation has some limitations. In particular, the kernel *clusters_argmin_shfl* is bounded to $k \le 1024$ (maximum block x dimension).
The approach of computing distances as a matrix multiplication showed poor performances for the following reasons:

- cuBLAS requires matrices to be stored in column-major format. This problem could be avoided by transposing the matrix when calling the *gemm* function, but the centroids' matrix needs to be first padded with a column of 1s. This is not easily achievable with row-major format without breaking the logic of other kernels/functions;
- Each point needs 2 matrix multiplications to compute the distances from each centroid; the distances consist in the main diagonal of the result matrix. All the diagonals need to be merged into one array and it is quite costly in therms of memory operations.

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.

4 Performances

To have a preliminar idea about the performance of the developed solution , we include the system traces of GPU_Kmeans executions. They can be useful to compare the behavior alternatively using shuffle distances kernel or matrix multiplication kernel as the first step on the k-means iterations.

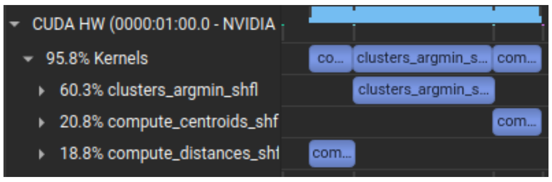
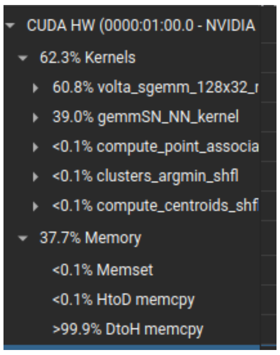

CUDA HW (0000:01:00.0 - NVIDIA
95.8% Kernels
60.3% clusters_argmin_shfl
20.8% compute_centroids_shf
18.8% compute_distances_shl
co...
clusters_argmin_s...
com...
clusters_argmin_s...
com...
com...

Figure 1



```
▾ CUDA HW (0000:01:00.0 - NVIDIA
  ▾ 62.3% Kernels
    ▸ 60.8% volta_sgemm_128x32_r
    ▸ 39.0% gemmSN_NN_kernel
    ▸ <0.1% compute_point_associa
    ▸ <0.1% clusters_argmin_shfl
    ▸ <0.1% compute_centroids_shf
  ▾ 37.7% Memory
      <0.1% Memset
      <0.1% HtoD memcpy
      >99.9% DtoH memcpy
```

Figure 2

Figure 1 highlights that the slowest kernel is the one responsible to compute the *argmin*. This may be due to the reduction performed with *structs* and the iteration to perform the block reduction.

Figure 2 gives some hints about the poor performances of the matrix multiplication distances function. Excluding the many (necessary) *sgemm* invocations, the main issue lies in the memory: this is because results are stored on matrices diagonals and need to be "extracted" at each step (centroids matrix $\cdot$ point associated matrix $\cdot$ centroids matrix^T).

4.1 Competitors

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

- KMeans (scikit-learn) [2] $\longrightarrow$ one of the most common sequential python implementations of the algorithm provided by *scikit-learn* library.
- kcuda [3] $\longrightarrow$ CUDA parallel implementation of k-means based on *"Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup"* [4]
- 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.

4.2 Sperimental setup

To fairly compare the different implementations, we needed to run all of them with the same input parameters and conditions. The attributes which affect the speed of execution are the choice of the initial centroids and the tolerance to declare convergence. Since our algorithm provides a random sampling of observations for initial centroids, we needed to have the same approach on all the competitors. The same goes for the tolerance; we set a common value of $1 \cdot 10^{-5}$ on the difference in the cluster centers of two consecutive iterations to declare convergence. The speed has been measured putting a timer around the k-means function invocations; most of the values has been computed as the average of 5 consecutive runs with the same seed ($s = 0$) for the random initialization of centroids in order to have deterministic behavior during the iterations.

The datasets used have been generated by a custom python script *data_generator.py* which exploits *torch* library to create random tensors with uniform distribution. The script provides also the possibility to output datasets with well-defined clusters; this feature has been used to generate 2D 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: $n \times d$.

All the tests have been carried out on a HPC cluster, a collection of nodes connected by a fast network and equipped with GPUs. The login node available at **marzola.disi.unitn.it** provides a SLURM workload manager to which submit the job script describing the tasks to execute on the nodes. Every job performed the measurements for a single implementation on all the different datasets with a fixed number of clusters k. Because of a time limit of 5 minutes per job in the partition **edu5** on the HPC cluster, the longest executions have been limited to 1-2 iterations. We set one node for each job, with one processor and GPU per task in order to equally compare the implementations.

4.3 Results

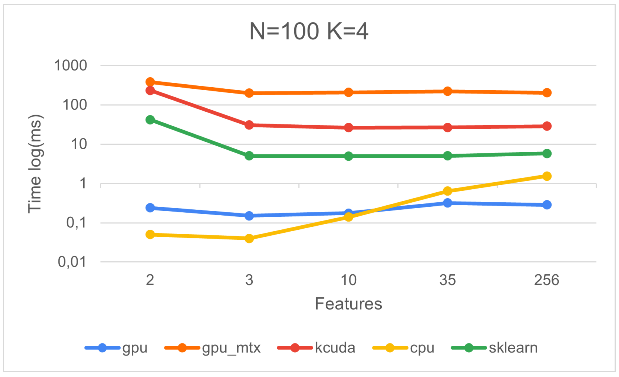
The most relevant results are showed in Figure 3 where there are four line charts, each with different combinations of number of samples and number of clusters. The x axes display the number of features of the datasets, while y axes represent the time in logarithmic scale.

Subfigure 3a displays tests performed with 100 samples and 4 clusters; for very small datasets the "silly" CPU implementation is faster than our GPU solution since the latency of device's memory highly affects the performances. The first iterations for *kcuda* and *sklearn* are much slower than the others; their average speed in these test cases is affected by those. Subfigure 3b shows that with a big dataset of 10^5 samples our solution performs better compared to the others with low features. Greater the features, much our solution has the same speed of the competitors. Same holds for Subfigure 3c where the number of clusters is quite high in relation to the number of samples. Performances are very good for low features and became worst when dimensions increase. The last Subfigure 3d highlights the fact that when the number of clusters and samples is high our solution become much slower compared to *kcuda*, which has the best performances. We can observe that our solution which uses the matrix multiplication approach to compute the distances, *gpu_mtx*, has poor performance in every test case except for *low samples - high clusters.*

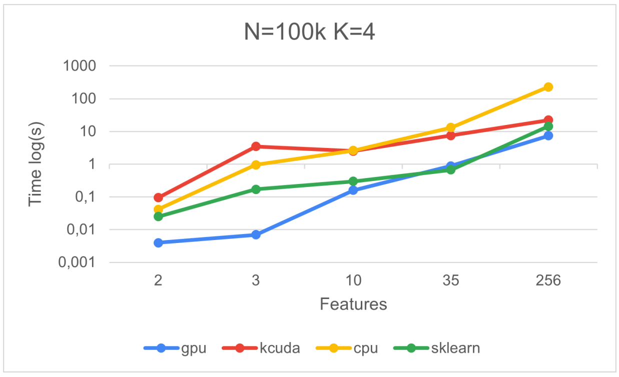
We can compute the speed up of our solution with respect to the competitors; for example for $n = 10^5; d = 256; k = 4$ we have:

$$S_1 = \frac{T_{sklearn}}{T_{gpu}} = \frac{14,5s}{7,4s} = 1,95; \; S_2 = \frac{T_{kcuda}}{T_{gpu}} = \frac{22,2s}{7,4s} = 3$$

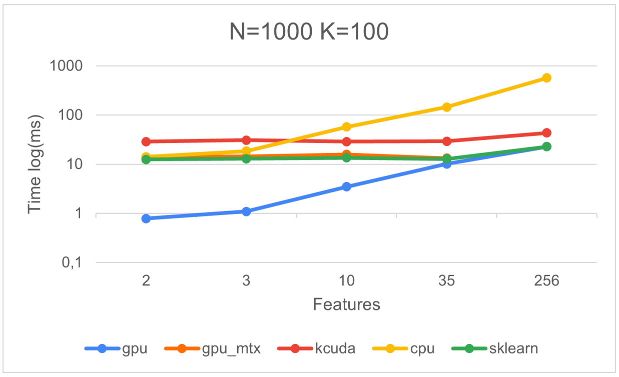
[2] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`
[3] `https://github.com/src-d/kmcuda`
[4] `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ding15.pdf`



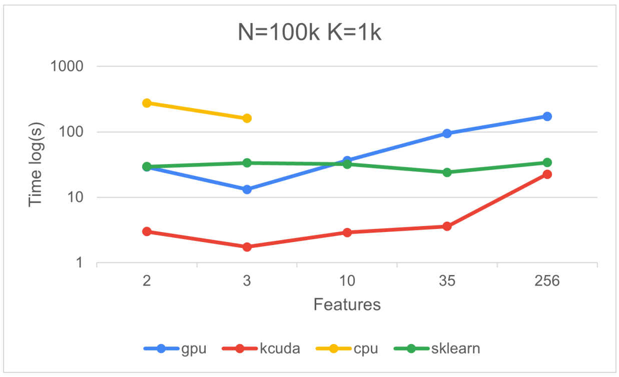
(a) Low samples - low clusters



(b) High samples - low clusters



(c) Low samples - high clusters



(d) High samples - high clusters

Figure 3: Speed comparison between different k-means implementations

However, for $n = 10^5; d = 2; k = 10^3$:

$$S_1 = \frac{T_{sklearn}}{T_{gpu}} = \frac{29,4s}{29s} = 1; \; S_2 = \frac{T_{kcuda}}{T_{gpu}} = \frac{3s}{29s} = 0,1$$

The reason of the performances worsening with number of clusters increasing is very probably caused by the fact that at the end of each iteration, the program compare the new centroids and the old ones on the host. To do it, we need to copy from the device the new centroids at each iteration. The centroids' matrix has $k \cdot d$ size, so greater the number of clusters and features, worse the performance will be since it is needed to copy a bigger set of data on the host.

To have a detailed vision about the tests illustrated above and also more, we invite to examine the data in the attachments.

5 Conclusions

Summing up the results of the tests, we noticed that our device implementation of k-means performs well with low clusters and features; the number of sampling does not affect significantly the performances. However, in real world it is quite difficult that input datasets have a big number of features d, since a dimensionality reduction is usually performed before applying a clusterization algorithm. Also the number of clusters is usually quite limited to assure a better clusterization. Anyway, to solve the issue we should avoid to copy the centroids on the host at each iteration, and perform the comparison directly on the device with a kernel. The call of the kernel to check convergence worth it only if the centroids matrix is quite big; for small $k \cdot d$ the comparison can be carried out still on the host.

The poor performances observed in the matrix multiplication approach for distances are caused by its integration in a second time and not in an optimal way. In the current version, even for the most optimized implementation would be really hard to be compared with competitors' performances. Probably it would have been necessary to fully revise also the other kernels involved in the algorithm implementation before the integration of the new approach, considering also to exploit sparse matrix optimizations.

Attachments

Distances matrix multiplication examples

1D example

$$c = (7), \quad p_1 = (3), p_2 = (1)$$

Using directly (3) we have:

$$\begin{aligned} dist^2(p_1, c) &= (3-7)^2 = \mathbf{16} \\ dist^2(p_2, c) &= (1-7)^2 = \mathbf{36} \end{aligned}$$

We now use the new approach:

$$C = \begin{bmatrix} 47 & -7 \\ -7 & 1 \end{bmatrix}, \quad q_1 = \begin{bmatrix} 1 & 3 \end{bmatrix}, q_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

$$\begin{aligned} dist^2(p_1, c) &= q_1 \cdot C \cdot q_1^T \\ &= \begin{bmatrix} 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 49 & -7 \\ -7 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 28 & -4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} \\ &= 28 - 12 = \mathbf{16} \end{aligned}$$

$$\begin{aligned} dist^2(p_2, c) &= q_2 \cdot C \cdot q_2^T \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 49 & -7 \\ -7 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 42 & -6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= 42 - 6 = \mathbf{36} \end{aligned}$$

2D example

$$c = (1, 4), \quad p = (5, 2)$$

Using directly (3) we have:

$$dist^2(p, c) = (5-1)^2 + (2-4)^2 = \mathbf{20}$$

We now use the new approach:

$$C = \begin{bmatrix} 17 & -1 & -4 \\ -1 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix}, \quad q = \begin{bmatrix} 1 & 5 & 2 \end{bmatrix}$$

$$\begin{aligned} dist^2(p_1, c) &= q_1 \cdot C \cdot q_1^T \\ &= \begin{bmatrix} 1 & 5 & 2 \end{bmatrix} \cdot \begin{bmatrix} 17 & -1 & -4 \\ -1 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 4 & -2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \\ &= 4 + 20 - 4 = \mathbf{20} \end{aligned}$$

3D example

$$c = (5, 2, 3), \quad p = (4, 3, 2)$$

Using directly (3) we have:

$$dist^2(p, c) = (4-5)^2 + (3-2)^2 + (2-3)^2 = \mathbf{3}$$

We now use the new approach:

$$C = \begin{bmatrix} 38 & -5 & -2 & -3 \\ -5 & 1 & 0 & 0 \\ -2 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix}, \quad q = \begin{bmatrix} 1 & 4 & 3 & 2 \end{bmatrix}$$

$$\begin{aligned} dist^2(p_1, c) &= q_1 \cdot C \cdot q_1^T \\ &= \begin{bmatrix} 1 & 4 & 3 & 2 \end{bmatrix} \cdot \begin{bmatrix} 38 & -5 & -2 & -3 \\ -5 & 1 & 0 & 0 \\ -2 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 4 \\ 3 \\ 2 \end{bmatrix} \\ &= \begin{bmatrix} 6 & -1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 4 \\ 3 \\ 2 \end{bmatrix} \\ &= 6 - 4 + 3 - 2 = \mathbf{3} \end{aligned}$$

Split matrix multiplications in δ dim chunks

If instead of using libraries like cuBLAS a direct approach is preferred, follows how the multiplication can be divided in sub-operations of size $\delta \times \delta$.

Let $C \in \mathbb{R}^{r\times s}$ ($r = s$ by definition of C, we will use only r) and $P \in \mathbb{R}^{q\times r}$.

Consider the multiplication $P \cdot C \cdot P^T$, we can develop this multiplication reducing the dimensionality of each sub-operation to a maximum size δ. Let's set the chunk size to δ, then the matrix C is padded left and bottom with ϵ rows/columns of 0s s.t. $(r+\epsilon)\ mod\ \delta = 0$, set $r := r + \epsilon$.

$$\text{Let: } \Delta r := \left\lceil \frac{r}{\delta} \right\rceil, \Delta q := \left\lceil \frac{q}{\delta} \right\rceil \qquad C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1r} \\ c_{21} & \dots & & \\ \dots & & \dots & \\ c_{r1} & & & c_{rr} \end{bmatrix} \implies$$

$$\implies \left[\begin{array}{cccc|ccc|c|ccc}
c_{11} & c_{12} & \cdots & c_{1\delta} & c_{1(\delta+1)} & \cdots & c_{1(2\delta)} & \cdots & c_{1((\Delta r-1)\delta+1)} & \cdots & c_{1(\delta\Delta r)} \\
c_{21} & \ddots & & & & & & & & & \vdots \\
\vdots & & \ddots & & & & & & & & \\
c_{\delta 1} & & & c_{\delta\delta} & & & & & & & \\
\hline
c_{(\delta+1)1} & & & & & & & & & & \\
\vdots & & & & & \ddots & & & & & \vdots \\
c_{(2\delta)1} & & & & & & & & & & \\
\hline
\vdots & & & & & & & \ddots & & & \vdots \\
\hline
c_{((\Delta r-1)\delta+1)1} & & & & & & & & \ddots & & \\
\vdots & & & & & & & & & \ddots & \vdots \\
c_{(\delta\Delta r)1} & \cdots & & & & \cdots & & \cdots & & \cdots & c_{(\delta\Delta r)(\delta\Delta r)}
\end{array}\right] =$$

$$= \left[\begin{array}{c|c|c|c}
C_{11} & C_{12} & \dots & C_{1\Delta r} \\
\hline
C_{21} & \ddots & & \\
\hline
\vdots & & \ddots & \\
\hline
C_{\Delta r1} & & & C_{\Delta r\Delta r}
\end{array}\right] = C_{chunked} \tag{10}$$

Then apply the same procedure to P obtaining: $P_{chunked} = \left[\begin{array}{c|c|c|c}
P_{11} & P_{12} & \dots & P_{1\Delta r} \\
\hline
P_{21} & \ddots & & \\
\hline
\vdots & & \ddots & \\
\hline
P_{\Delta q1} & & & P_{\Delta q\Delta r}
\end{array}\right]$

Consider $P_{chunked} \cdot C_{chunked} \cdot P_{chunked}^T$[5], let i be a parameter that identifies a row, then:

$$\begin{aligned} D_i = [P_{i1}, P_{i2}, \dots, P_{i\Delta r}] \cdot C_{chunked} \cdot [P_{i1}, P_{i2}, \dots, P_{i\Delta r}]^T &= \\ = [P_{i1}C_{11} + P_{i2}C_{21} + \dots + P_{i\Delta r}C_{\Delta r1}, \quad P_{i1}C_{12} + P_{i2}C_{22} + \dots + P_{i\Delta r}C_{\Delta r2}, \dots \\ \dots, P_{i1}C_{1\Delta r} + P_{i2}C_{2\Delta r} + \dots + P_{i\Delta r}C_{\Delta r\Delta r}] \cdot [P_{i1}^T, P_{i2}^T, \dots, P_{i\Delta r}^T]^T \end{aligned} \tag{11}$$

Let $diag : \mathbb{R}^{r\times r} \to \mathbb{R}^r$ be the function that "extracts" the main diagonal of a matrix e.g. $diag(M)$ returns the main diagonal of the matrix M.

$$D = \begin{bmatrix} diag(D_1) & & & \\ & diag(D_2) & & \\ & & \ddots & \\ & & & diag(D_{\Delta q}) \end{bmatrix} \tag{12}$$

[5] $P_{chunked}^T$ transposes the whole original matrix P.

The vector $diag(D) \in \mathbb{R}^q$ contains the distances of the points in P from the center e.g. $d_{33} \equiv diag(D)_3$ is the distance of the point number 3 from the center.

Observation: many of the multiplications in (11) can be simplified: $P_{ab}C_{ij} = 0_{\delta\delta} \quad \text{if } i,j > 1 \wedge i \neq j \quad \forall a, b$[6]

Computational complexity

Let $\theta_{cnk} = max(\Delta r, \Delta q)$, the overall computational complexity of the procedure is:

$$O(2(\theta_{cnk}^2 - (3\theta_{cnk} + 2))) \text{ having } P_{ab}C_{ij} \text{ as } O(1) \tag{13}$$

Let $\theta = max(r, q)$ Without chunking the complexity is:

$$O(\theta^2) \text{ having } p_{ab}c_{ij} \text{ as } O(1) \tag{14}$$

Since $\theta_{cnk} := \lceil \frac{\theta}{\delta} \rceil$ and $P_{ab}C_{ij}$ costs $O(\delta^2)$, the two complexities are similar.

Using tensor cores, we can approximate the cost of $P_{ab}C_{ij}$ as $O(1)$, obtaining a performance improvement of δ^2 w.r.t. a classical implementation of $P \cdot C \cdot P^T$.
Note: the matrix multiplication approach has a squared computational complexity w.r.t the warp-oriented one. Therefore same performances can be achieved only if the cost of $P_{ab}C_{ij}$ with tensor cores is similar to $p_{ab}c_{ij}$.

[6] $0_{\delta\delta}$ denotes a matrix in $\mathbb{R}^{\delta \times \delta}$ filled with 0s.

Tests results

Impl.	Features				
	2	3	10	35	256
gpu	242us	148us	175us	320us	285us
kcuda	239ms	31ms	26ms	27ms	29ms
cpu	47us	35us	136us	642us	1537us
sklearn	42ms	5ms	5ms	5ms	6ms
gpu_mtx	384ms	200ms	209ms	222ms	205ms

Table 1: $n = 10^2$, $k = 4$

Impl.	Features				
	2	3	10	35	256
gpu	225us	211us	170us	257us	284us
kcuda	49ms	27ms	26ms	26ms	28ms
cpu	73us	95us	198us	802us	2ms
sklearn	14ms	7ms	7ms	7ms	9ms
gpu_mtx	409ms	224ms	216ms	224ms	205ms

Table 2: $n = 10^2$, $k = 10$

Impl.	Features				
	2	3	10	35	256
gpu	160us	757us	1ms	2ms	4.5ms
kcuda	27ms	31ms	30ms	32ms	58ms
cpu	329us	2ms	8ms	23ms	93ms
sklearn	5ms	5ms	6ms	7ms	13ms
gpu_mtx	392ms	1.2s	1.7s	1.6s	1.4s

Table 3: $n = 10^3$, $k = 4$

Impl.	Features				
	2	3	10	35	256
gpu	793us	1ms	3,5ms	10ms	23ms
kcuda	29ms	31ms	29ms	29ms	43ms
cpu	14ms	18,5ms	58ms	146ms	571ms
sklearn	12,5ms	13ms	13,5ms	13ms	22ms
gpu_mtx	14s	14s	16s	13s	+5min

Table 4: $n = 10^3$, $k = 10^2$

Impl.	Features				
	2	3	10	35	256
gpu	2ms	3ms	9ms	71ms	245ms
kcuda	67ms	169ms	161ms	349ms	658ms
cpu	21ms	26ms	130ms	1,1s	3,4s
sklearn	11ms	14ms	40ms	74ms	348ms
gpu_mtx	14.4s	15.7s	28s	63s	69s

Table 5: $n = 10^4$, $k = 4$

Impl.	Features				
	2	3	10	35	256
gpu	77ms	104ms	404ms	1s	3,7s
kcuda	309ms	73ms	71,5ms	94ms	566ms
cpu	2,5s	2,6s	7,3s	24s	102,6s
sklearn	653ms	328ms	358ms	358ms	687ms
gpu_mtx	-	-	-	-	-

Table 6: $n = 10^4$, $k = 10^3$

Impl.	Features				
	2	3	10	35	256
gpu	3,9ms	68ms	160ms	879ms	7,4s
kcuda	95ms	3,5s	2,5s	7,6s	22,2s
cpu	42ms	961ms	2,6s	13,3s	229s
sklearn	25ms	166ms	312ms	666ms	14,5s
gpu_mtx	25,4s	+5min	+5min	+5min	+5min

Table 7: $n = 10^5$, $k = 4$

Impl.	Features				
	2	3	10	35	256
gpu	29s	13s	36s	94s	172s
kcuda	3s	1,8s	2,9s	3,6s	22,5s
cpu	276s	161s	+5min	+5min	+5min
sklearn	29s	34s	32s	24s	34s
gpu_mtx	-	-	-	-	-

Table 8: $n = 10^5$, $k = 10^3$