K-means Design

Hongbo Sun, Qingxuan Kuang, Rongjian Xiao

1. Algorithm Description and Design of implementation

The way K-means algorithm works is as follows:

1. Specify the number of clusters *K*.
2. Initialize centroids by first shuffling the dataset and then randomly selecting *K* data points for the centroids without replacement.
3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.

- Compute the sum of the square distance between data points and all centroids(Kernel 1.1).
- Assign each data point to the closest cluster centroid (Kernel 1.2).
- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster. (Kernel 2)

There are 3 kernels in our K-means design. The first kernel is for calculating distance from each point to each centroid, the second kernel is for selecting the nearest centroid, and the last kernel is for re-position the centroids.

Assume we have n points with d dimensions each, and we want to have k clusters.

1. Calculating Distance

The first kernel is to calculate each point-to-centroid distance. Here we use Euclidian distance but omitting the square root since it does not affect our distance comparing. This is Similar to matrix multiplication, but with an extra operation of squaring. We are using FMADD and MUL instructions.

Input: n * d matrix for n points and k * d matrix for k centroids.

Output: n * k matrix for each point-to-centroid distance.

independent operation: Calculate the distance between a point and a centroid.

Dependent instruction:

- total distance between a point and a centroid += distance of each dimension
- SUB

Function unit: MUL, SUB, ADD

2. Selecting Nearest Centroid

After having n * k distance matrix, we want to first find the minimum distance to centroids for each point. This would be a n dimensional vector, then we broadcast this vector to match back to the n * k distance matrix to generate a n * k mask of 0 or 1 representing if this this the nearest centroid for corresponding point. This n * k mask could also be viewed as n k-dimensional one-hot vectors.

Input: n * k distance matrix

Output: n * k mask

Independent operation: Compare distances

Dependent instruction: Compare -> Assign data

Function unit: CMP

3. Moving centroids

With the n * k mask we can do a reduce by column to generate a k-dimensional vector representing the count of points that belongs to each centroid. Then, we perform an matrix multiplication on the n * k mask and n * d original matrix to generate the summation of the coordinates that belongs to each cluster. Then we simply divide this by the count vector to generate the average of each cluster.

Input: n * k mask and n * d matrix for n points

Output: k * d matrix for adjusted centroids.

Independent operations:

- compute the different clusters centroid's position (eg. Computing cluster centroid A and Computing cluster centroid B is independent)
- Calculate the average value of different dimensions of a cluster centroid

Dependent instruction: Sums all values of a dimension(ADD) -> Get average value(DIV)

Function unit: ADD, DIV

2. Performance Peak. Based on your description of the algorithm(s) and the kernel(s), answer the following questions:

(a) Describe the machine of your choice (model & architecture)

Model: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz

Architecture: haswell

cat /proc/cpuinfo

cat /sys/devices/cpu/caps/pmu_name

(b) What are the latency and throughput of the instructions identified in the previous question?

	Latency	**Throughput**
`__m256i _mm256_abs_epi32 (__m256i a)` no 256d	3	1
`__m256d _mm256_sub_pd (__m256d a, __m256d b)`	3	1
`__m256d _mm256_add_pd (__m256d a, __m256d b)`	3	1
`__m256d _mm256_div_pd (__m256d a, __m256d b)`	17-21	13
`__m256d _mm256_cmp_pd (__m256d a, __m256d b, const int imm8)`	3	1
`__m256d _mm256_mul_pd (__m256d a, __m256d b)`	5	0.5

(c) Do you have the appropriate SIMD instructions on your machine?

`__m256i _mm256_abs_epi32 (__m256i a)` no 256d
`__m256d _mm256_sub_pd (__m256d a, __m256d b)`

`__m256d _mm256_add_pd (__m256d a, __m256d b)`
`__m256d _mm256_div_pd (__m256d a, __m256d b)`
`__m256d _mm256_cmp_pd (__m256d a, __m256d b, const int imm8)`
`__m256d _mm256_mul_pd (__m256d a, __m256d b)`

(d) Are there specialized units that can be used your machine?

No

(e) Explain how you computed the theoretical peak of your kernel(s)?

We can split it into 3 kernels and calculate the theoretical peak for each of them.

Let's say

d = data dimension,

n = the number of data points,

k = the number of cluster centroid

freq = Base freq = 2.4 GHz

Theoretical peak = flops per cycle * freq

Kernel 1.1:

Throughput of "SUB" is 1.

Throughput of "Mul" is 0.5.

Throughput of "ADD" is 1

Throughput of Adding the distance of all dimension = min(1, 0.5, 1) = 0.5

Theoretical peak = ops/cyc * freq = 0.5 * 2.4G = 1.2G

Kernel 1.2:

We need to do CMP for each data point for each cluster centroid.

Throughput = 1

Theoretical peak = 1*2.4G = 2.4G

Kernel 2:

Throughput of Summing the value of one dimension = 1

Latency of computing average value(DIV): 13

Theoretical peak = 1 * 2.4 G = 2.4G

3. Performance Baseline.

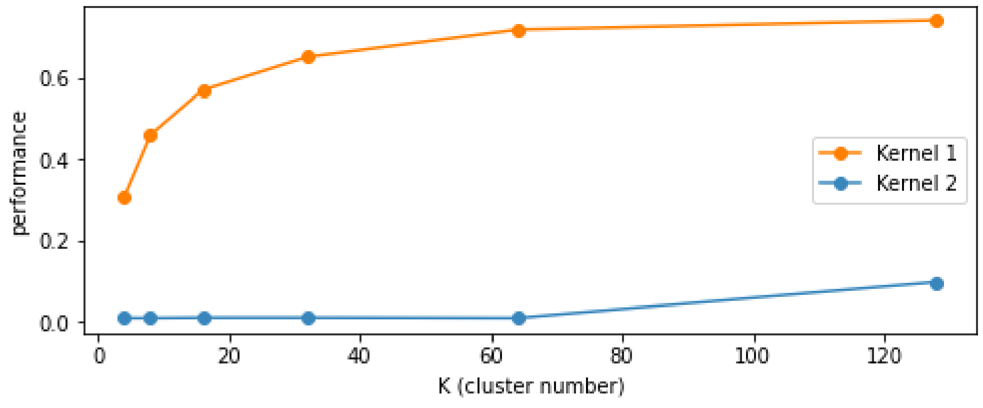
I see Kernel 1.1 with Kernel 1.2 as a whole. For the baseline code, the formula of performance is as follows:

Kernel 1's performance = (2 * n * d * K + n * K) / (duration / iterations);

Kernel 2's performance = (n * d) / (duration / iterations);

For total points (n) = 1024, dimensions (d) = 16:

K	Kernel 1	Kernel 2
4	0.307	0.010
8	0.460	0.010
16	0.570	0.011
32	0.651	0.011
64	0.718	0.010
128	0.741	0.098



This is very far away from the theoretical peak mentioned above, because the baseline code used function calls, for and while loops and classes.