Speeding up K-Means Algorithm with CPU Parallelism

1st Haoxiang Xu
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, United States
haoxianx@andrew.cmu.edu

2nd Heng Wang
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, United States
hengw@andrew.cmu.edu

3rd Ge Song
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, United States
gesong@andrew.cmu.edu

***Abstract*—This report describes the details of our semester-long project in CMU graduate course *How to Write Fast Code (18-645)*. By analyzing the independent operations in typical K-means algorithm, we design some SIMD kernels that can make full use of the function units of the machine. Then, we organize the input data layout to achieve better cache locality, and explore how OpenMP share memory parallelism can boost the performance further. Finally, we compare our implementation with baseline library and theoretical peak on specific hardware.**

***Index Terms*—High Performance Computing, Unsupervised Machine Learning, CPU Parallelism, Cache-Aware Design**

I. Introduction

K-means is a very popular algorithm. In machine learning area, this algorithm can find cluster centers that minimize the intra-class variance, i.e. the sum of squared distances from each data point being clustered to its cluster center (the center that is closest to it). It can be applied in many areas, such as evaluating the performance of a certain basketball team, dividing the followers to a certain cluster, and quantifying the signal.

This algorithm can be briefly summarized with the following steps:

- Choose the initial values (or ”seeds”) for the k-means clustering algorithm.
- Calculate the distance between every node and each center of the cluster.
- Divide each node into the closest cluster.
- Repeat the above steps until it reaches the specific condition.

CPU parallelism is a parallelism method which breaks large problems into smaller ones, which can then be solved by multiple hardware threads of CPU at the same time. We utilized OpenMP to achieve CPU parallelism in this project. We also leverage hardware SIMD function units that can process multiple data simultaneous to exploit lower level parallelism.

Some operations in k-means, such as calculating the distance between points and clusters, dividing nodes clusters, could be paralleled to speed up the execution dramatically as long as planning the parallel region properly.

CMU Graduate Course: 18-645 How to Write Fast Code

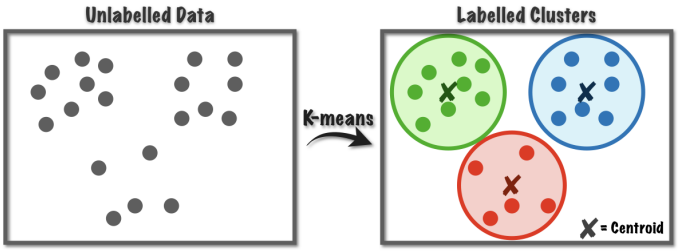

Fig. 1. K-means Algorithm [1]

II. K-mean algorithms

There are many different kinds of k-means algorithms.

In scikit-learn library, one of the k-means method is Mini-Batch k-means. Mini Batch k-means algorithm‘s main idea is to use small random batches of data of a fixed size, so they can be stored in memory. Each iteration a new random sample from the dataset is obtained and used to update the clusters and this is repeated until convergence. Each mini batch updates the clusters using a convex combination of the values of the prototypes and the data, applying a learning rate that decreases with the number of iterations. This learning rate is the inverse of the number of data assigned to a cluster during the process. As the number of iterations increases, the effect of new data is reduced, so convergence can be detected when no changes in the clusters occur in several consecutive iterations. [2] However, this method is not a good baseline because Mini-Batch k-means deals with large data but our design didn’t and it is hard to ensure the correctness.

One k-means variant is k-means++, which chooses initial centers in a way that gives a provable upper bound on the WCSS objective. The advantage of k-means++ is keeping different initial centers not so close, which may help users to get better results. However, we only focus on the performance not the better clustering results, those operations in choosing initial centers could be an overhead.

The baseline we used to compare against correctness and performance is the k-means algorithms in scikit-learn library, the parameter: algorithm is set to default, auto. This algorithm

is the classical k-means algorithm but its behavior is different when copes with the different density of data.

III. Kernel Design

A. Overall Loop Structure

Pseudocode. 1 shows that there are four main loops in typical k-means algorithm. The first loop is the maximum iteration for the algorithm to run. In practice, k-means algorithm may ends earlier when the clusters between two iterations converge, but this feature is not implemented, since complete control of the number of iterations make performance evaluation more convenient. In second and third loops, all centers are enumerated for every points, while the fourth loop iterate through all dimensions to compute distance between points and centers.

The distance computation of each points is clearly independent, and thus we leverage the loop parallelism of the points loop. In the loop over center matrix, the nearest centers of points being enumerated and corresponding distance are updated. Thus, by the end of the points loop, these nearest centers indexes are used to form the new cluster for next iteration.

The steps of the last three loops are specified a way to fit the distance computation kernels. Inside this kernel, *kernel_p* points *kernel_c* centers and *kernel_d* dimensions are processed in batch with SIMD instructions. Since the distance computation works on every combination of points and centers, with time complexity *O(k*p*d)*, it should be the most important kernel to optimize in the whole project.

Algorithm 1: Loop Structure of K-means

```
1:  for i = 0, iterations do
2:    for p = 0, len(points); p += kernel_p do
3:      for c = 0, len(centers); c += kernel_c do
4:        for d = 0, dimensions; d += kernel_d do
5:          distance computation kernel
6:        end for
7:        update nearest center of points
8:      end for
9:      accumulate points to nearest cluster
10:   end for
11:   compute new cluster
12: end for
```

B. Independent Operations

Distance computation of different points between centers:

- To compute the distance between points and centers, like dist(P,C), if the points P is different, the distance are by no means dependent. Since those operations are independent, in the distance computation kernel, we can divide the input points into several groups. Points of each groups are independent.

Points labeling of different points between centers:

- Points labeling gives the result of which centers is closest to the a certain points. Different points would not intervene with each other.

New cluster computations between centers:

- For each new center with its size and the sum of points belong to them. The mean, also called new cluster, could be computed. For different centers, those operations are considered to be independent.

C. Dependent Operations

Distance computation on several dimensions:

- In different dimensions, distance computation kernel shares registers, which store the intermediate distance. As a result, they are dependent, the next 4 dimensions computation is rely on the previous result.

Distance computation of the same groups of points in the distance computation kernel:

- For those points in the same group, there is a dependent chain: Do _mm256_sub_pd first and do _mm256_fmadd_pd. Repeat those operations 4 times to update the intermediate distance between 4 points and one center. As a result, the chain size is 8. Since the latter operations are rely on the previous results, those operations are dependent.

Points labeling of one point between different centers:

- Given all the distance between one point and all centers dist(p1, c1), dist(p1, c2), dist(p1, c3), ... , the minimum distance should be figured out to get the label of this point. This operation is also dependent.

D. SIMD Intrinsics

The main Intel SIMD instructions used in the kernels and their corresponding benchmark on Broadwell architecture are listed below. In particular, _mm256_sub_pd, _mm256_add_pd, _mm256_fmadd_pd are used in distance and cluster computations, and they are the most frequent instructions in SIMD kernels, and our goal is to be close to the peaks of these instructions. Instructions _mm256_permute_pd and _mm256_blend_pd are used to compare the distance in batch when selecting the index of cluster for each points. The remaining instructions are used to load and store values into SIMD registers, and these overhead should be amortized.

TABLE I
Latency and Throughput of SIMD instructions

SIMD instruction	*Latency (cycles)*	*Throughput (IPC)*
_mm256_sub_pd	3	1
_mm256_add_pd	3	1
_mm256_fmadd_pd	5	2
_mm256_broadcast_pd	7	2
_mm256_load_pd	1	2
_mm256_store_pd	1	2
_mm256_permute_pd	1	1
_mm256_cmp_pd_mask	3	1
_mm256_blendv_pd	2	0.5

E. Distance Computation

The distance computation could be decomposed into the following steps:

- Load intermediate distant vectors, which has been computed in previous dimensions and should be updated in this kernel.
- Load points vectors. Points are column majored.
- Use _mm256_broadcast_sd to broadcast a single dimension of the center vector.
- Use _mm256_sub_pd to do the subtraction between points vectors and the vector after broadcast.
- FMA them together using _mm256_fmadd_pd to update the intermediate distance between 4 points and one center.

In order to use as many registers as possible, in the testing machine, there are 16 registers, the computation could be divided into several independent groups. Since 1 register is used for store the vectors of broadcasted one dimension of centers, there are 15 registers remained. For each group, need 4 registers to store intermediate distance, 1 register to store points and 1 register to store the result of subtraction between points vectors and the vector after broadcast. The group size should be 2.

As a result, the size of distance computation kernel is 4x8. The input of this kernel are 4 centers vectors, 8 points vectors, whose dimension size is 4. The output of this kernel are 8 distance vectors between 8 points and 4 centers in 4 dimensions.

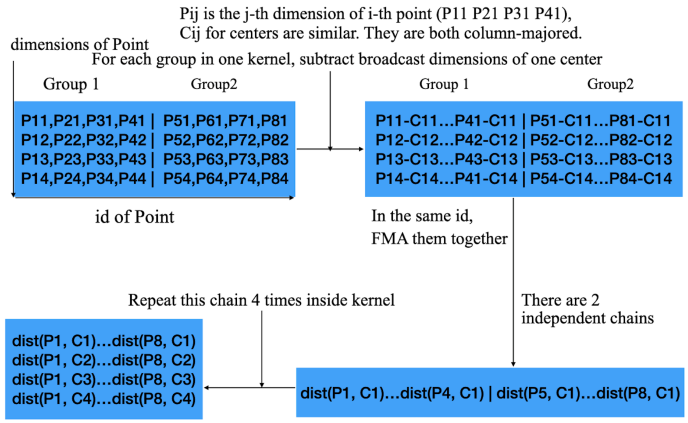

Fig. 2. Distance Computation Kernel

Evaluating the theoretical peak:

As the Fig. 2 shows, there are 2 independent chains. Those chains are subtract first then do a FMA and repeat those operations 4 times. Sub has 1 functional units and it deals with 4 doubles. FMA has 2 functional units and it deal with 8 doubles. The theoretical peak is, however; 4 + 8 * 1 = 12. In our testing architecture Intel, broadwell, FMA's execution pipes are 0 and 1; subtract's execution pipe is 1 [3], they shared the same pipe 1. Although we have 2 independent chains, FMA could not use 2 functional at the same time. As a result, the theoretical peak is 12 FLOPS / cycle.

F. Points Labeling

The points labeling could be decomposed into the following steps:

- Load ultimate distant vectors in this iteration.
- In each group, use _mm256_cmp_pd to get the mask indicating the smaller distance.
- In each group, use _mm256_blendv_pd to merge indexes into a single vector.

The input of this kernel is 8 distance vector from previous kernel. Able to fill up the pipeline. The output is minimum distance and labels of 8 points indicating which centers the point belongs to. In our implementation, the group size is 2, consisting with the distance computation kernel.

Evaluating the theoretical peak:

Throughput of _mm256_blendv_pd is 0.5, _mm256_cmp_pd is 1 and they are dependent operations in this kernel. _mm256_cmp_pd and _mm256_blendv_pd deal with 4 doubles at the same time. As a result, the theoretical peak is 4 * 0.5 = 2 FLOPS / cycle.

G. New Cluster Computation

New cluster computation could be decomposed into the following steps:

- Use _mm256_add_pd to add the new centers with its points. Record the sum of points belong to this center and the corresponding size of center.
- When all points are computed in this iteration, compute the new centers. Divide the sum of the points of its center by the size of center to get the new cluster.

Evaluating the theoretical peak:

Throughput of _mm256_add_pd is 1. _mm256_add_pd deal with 4 doubles at the same time. As a result, the theoretical peak is 4 * 1 = 4 FLOPS / cycle.

IV. Cache-Aware Design

A. Cache Layout of Machine

On the ECE machine we use, there is a 32 KB 8 ways set-associative L1 cache on each core. When all the data used by a thread routine can be fitted in L1 cache, the computation efficiency is expected to be maximized. In our design of kernels, one column-majored submatrix of points and one column-majored submatrix of centers are used in distance computation kernel, while the computation of new centers involve row-major points and centers matrices. Thus, these two centers and points submatrices in total handled in a single p loop. Since the distance computation kernel handle 8 points and 4 centers, the ratio of size of points matrices to these of centers matrix is 2:1. Every points and centers matrix should occupy whole ways of cache line, so that data in use in cache will not be evicted. Therefore, points matrices should ideally occupy 4 ways, while centers matrices occupy 2 ways. The remaining 2 ways should be reserved for other temporary allocated memory storing intermediate results.

Each cache way can store 32KB / 4 = 8KB data, that is 512 doubles (8 bytes). Since there are 8 points handled

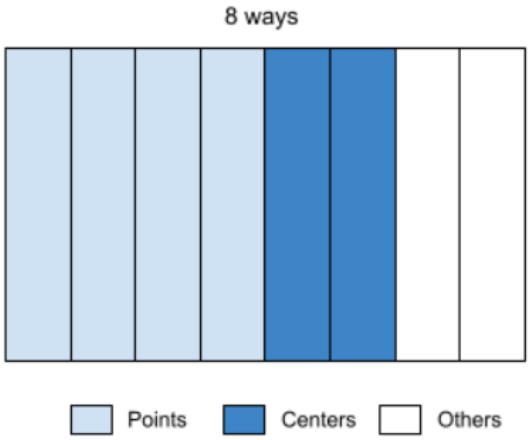

Fig. 3. Conceptual L1 cache assignment for k-means computation kernels in one p loop

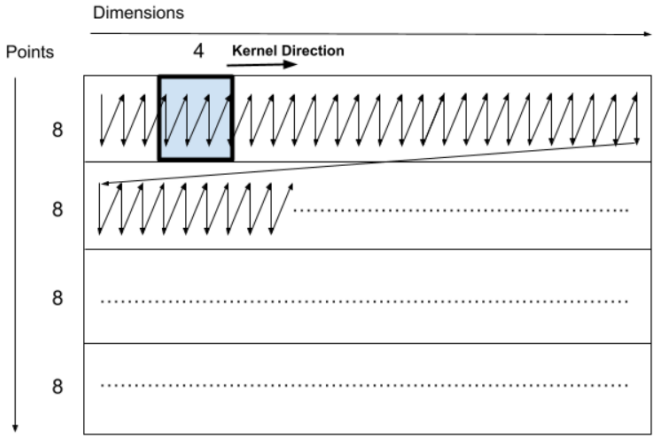

Fig. 4. The input data is in row-major order (each point in one row). Inside the kernel, data should be organized in column-major order to make use of SIMD instructions. Since the distance computation kernel handle 8 points in batch, the data should be organized into contiguous panels of 8*dimensions to maximize the locality.

in one p loop, and the points matrices are duplicated for row and column major order for different computation, the highest dimensions for points and centers matrices can be fitted into the layout of Fig. 3 is 512 / 8 * 2 = 128. However, we found that the performance of our single thread SIMD implementation's performance dropped when the number of dimensions reached 64, whereby one points metrix just occupied one whole cache way. This might be caused by the fact other unconsidered memory usage occupy the whole set, resulting in some cache lines with points and centers data being evicted earlier than expected. Despite less capacity to hold high dimensions data, we argue that the performance drop barely happen in practice, since our data layout and implementation is designed for memory-sufficient scenario, and high dimensions data always embrace other compressed data layout, like typical sparse matrix representation, which is out of the scope of this semester-long project.

B. Data Layout

In typical machine learning library, points are organized in a 2d matrix, whereby each row represent a single point. To make use of the SIMD instructions, our kernel handle the data in column-major order (each row represents one dimensions of all 8 points). When computing the distance, we always need to iterate the dimension of points center in the inner loop to get correct result. This means transpose the whole matrix is also not perfect, since this will cause the kernel to visit the matrix discontinuously, and therefore bring poor space locality.

To have a good utilization of L1 and L2 cache, the input points data should be organized into contiguous panels with height of 8 (Fig. 4). The height of panel is determined by the kernel size, to let kernel work on data which is previously loaded into cache and thus maximize cache hit ratio. This data layout design is also beneficial for OpenMP static chunk scheduling, since each thread will independently load one whole panel, or part of it, into the L1 cache of the hardware thread, and evict it until all data are computed.

V. CPU-based Parallelism

In previous sections, we have discussed how we designed the kernels with performance close to theoretical peak, and how we leverage cache to boost performance further. The next step is to explore the parallelism of multiple cores. In our project, there are several motivations for us to parallelize the implementations:

- The ECE machine used for our project has 10 cores and supports 40 hardware threads. Our serialize implementation of distance computation kernels only make use of the functional unites located on one single core
- Due to the number of registers reserved for intermediate distance result, the points labeling(Section. III-F) and computation of new cluster(Section. III-G) are limited to a relatively small kernel size
- There are dependencies cannot be eliminated in the new cluster computation, since each point cannot be added to corresponding cluster until the the label of it is computed, and several points inside the kernel can be grouped into one single cluster. Thus, the process of each points need a load-add-store pipeline, which cannot be optimized by SIMD instricution, but can benefit from higher level parallelism.

A. Parallelization Scheme

The OpenMP parallel programming library is chosen for our project. The other two candidates are MPI and CUDA, and the reasons why OpenMP surpass them in this project are similar. The Message Passing framework requires frequent communication between several distributed hosts, and in our project, the benchmark data set size is relatively small, which means the additional computational bonus brought by multiple processors might not make up the overhead of communication. The GPU architecture also requires data movement from host

to device, which is only cost effective on huge data set, where the parallel computation can amortize communication overhead.

B. Implementation

As shown in Pseudocode. 1, the *p* loop in *line 2* iterate all the points in the date set to compute the distance between centers, labeling the points, and adding points to corresponding new centers. Obviously, the distance computation and labeling are independent between points, and these two steps account for a huge proportion of calculations in the whole algorithm. Thus, we leveraged OpenMP *parallel for* on the *p* loop and construct critical section around new centers computation (Pseudocode. 1, line 9), since this step requires many concurrent updates on the shared data structure to figure out what the new centers for next iteration are. However, this simple implementation did not improve the performance but let the FLOPS per cycle drop when the number of threads increased.

The drop of the performance is mainly caused by three problems, and we optimized our implementation for them. The first optimization is to use static loop schedule with chunk size of *N over t*, where *N* is the total number of points and *t* is the number of threads. In this way, the OpenMP will divide the iterations into chunks of equal size that and distributes one chunk to each thread. For input data set with small number of dimensions, static schedule will utilize cache better, since each thread can handle contagious data that can be loaded into L1 cache and access efficiently in the whole computation.

The other two problems are related to new centers computation (Pseudocode. 1, line 9). First, this computation is obviously a bottleneck where only one one thread can access. Otherwise, there will be race condition and brings inconsistency in the result. Second, each thread accesses the shared memory storing updated centers, and they are loaded into the cache of thread. Thus, frequent updates of shared memory in threads will continuously invalidate the cache of other, causing low cache utilization. This is the so-called "cache ping pong" illustrated in Fig. 5. The optimization for these two problem is discussed in the next subsection.

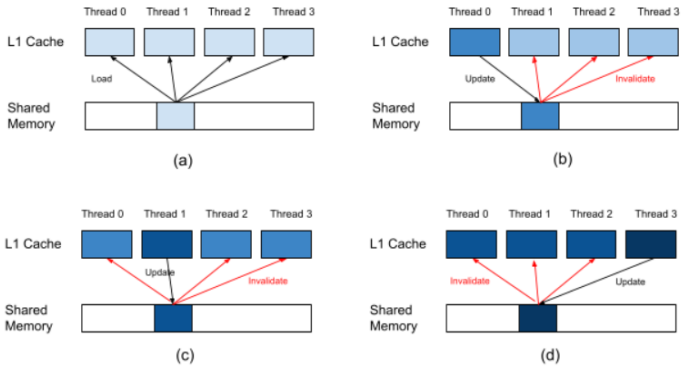

Fig. 5. Illustration of cache ping pong. (a) the shared memory is loaded into the cache lines of 4 threads; (b)(c)(d) show that thread 1, 2, 4 keeps updating on the shared memory alternately. The L1 caches of 4 threads are always being invalidated, and the probability of hitting the cache becomes low.

C. Optimization for Bottleneck and Cache Ping-Pong

The bottleneck and cache ping pong are caused by accessing and updating the shared area in memory, which requires mutual lock and results in frequent cache cross-thread invalidation. Thus, as shown in Fig. 6, we can generate a thread local copy of new centers for each thread before execution, and merge the result from each thread by summing them together. This method can get correct results since all the computation on new centers array are commutative and associative. The merging of results can also be parallelized after all the threads finish execution, but the performance improved by parallel computing on small centers array is negligible.

Typically, applying this kind of local copy method is a trade-off between memory usage and computation efficiency, but the only shared memory that involves concurrent updates in our design is the array maintaining the new centers for next iteration, which is often small in practice, since the number of centers (k) in k-means should not be too large in real life use case. Fig. 7 shows the significant performance improvement by using this method in our k-means implementation.

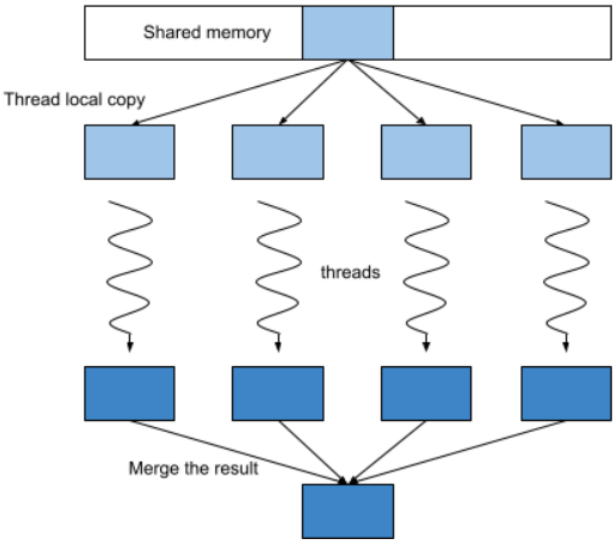

Fig. 6. Illustration of cache ping pong. (a) the shared memory is loaded into the cache lines of 4 threads; (b)(c)(d) show that thread 1, 2, 4 keeps updating on the shared memory alternately. The L1 caches of 4 threads are always being invalidated, and the probability of hitting the cache becomes low.

VI. Performance

A. Machine Info

- **Hostname of the machine:** ece008.ece.local.cmu.edu
- **CPU** [4] **model and manufacturer:** Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, Intel Corporation
- **Base and maximum frequencies**: 2.40 GHz and 3.40 GHz.
- **Number of physical cores:** 10
- **Number of hardware threads:** 20
- **Number of caches:** 4
- **Size of each cache: L1d Cache:** 32K, L1i Cache: 32K, L2 Cache: 256K, L3 Cache: 25600K

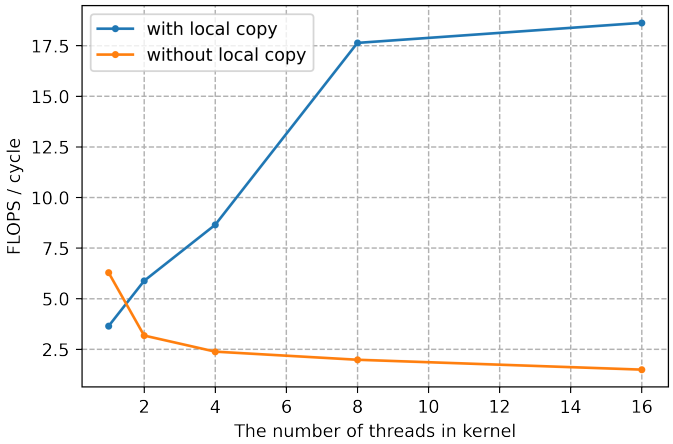

Fig. 7. We compared the performance of thread local copy implementation with a naive multi-threads implementation on 40,000 64-dimensions points data set. The plot shows that even through the single thread performance of thread local copy is lower than that of naive parallel due to the copy and merge overhead, its performance increases in path of the number of threads (<16), while the naive implementation suffers from the penalty of cache cross-thread invalidation and bottleneck of critical section.

- **Microarchitecture:** Broadwell

B. Computation of FLOPS

According to our kernel design, the total FLOPS can be calculated from three parts. If the input points size is **P**, the number of clusters is **C** and their dimensions are **D**, then the distance calculation kernel must be operated on each center and each dimension of each point, and each operation includes subtraction, multiplication and addition, so total FLOPS / cycle of computing distance is $\mathbf{3} \times \mathbf{P} \times \mathbf{C} \times \mathbf{D}$. Second, to find the index of the nearest cluster, it is required to compare the distance from each point to each cluster, so the FLOPS / cycle is $\mathbf{2} \times \mathbf{P} \times \mathbf{C}$. Finally, every point must be added to update the new cluster, therefore the FLOPS / cycle is $\mathbf{P} \times \mathbf{D}$. Besides, if the kernel runs **Iter** iterations, and after running the kernel **Runs** times, the calculated cycle result is **Cycle**, and the final result can be obtained by the following formula:

$$\frac{\mathbf{Runs} \times (\mathbf{Iter} \times (\mathbf{3} \times \mathbf{P} \times \mathbf{C} \times \mathbf{D} + \mathbf{2} \times \mathbf{P} \times \mathbf{C} + \mathbf{P} \times \mathbf{D}))}{\mathbf{Cycle}} \tag{1}$$

C. Dataset

The dataset we used for testing performance is generated by the make_blobs function in scikit-learn package [5] and it contains 400000 points with dimensions ranging from 4 to 320. In addition, we can also initialize our centers in the same way.

D. Baseline

Our baseline model is k-means function in scikit-learn package. There are several k-means implementations in this package, and we chose the classical EM-style algorithm [6], since this k-means algorithm is the closest to our project. To fairly eliminate the randomness and fairly compare the results, we force the baseline implementation to have same initial centers as our implementation, and run fixed number of iterations before converging.

E. Performance of Individual Kernels

From section.III-E, section.III-F and Section.III-G, we can obtain the theoretical peaks of these three kernels. From the test to each kernel, the comparison of the peak value and the result is shown in the table II. Our implementation of computing distance is extremely close to the theoretical peak, but new cluster computation result is much lower than peak value. This difference is caused by frequent SIMD load and store operations for each point and center in the kernel, and the entire calculation is dependent. This situation is almost inevitable in the model we designed, but compared with distance computation, this part of the calculation is very small, especially when the point dimension is large, therefore the slight performance decrease is acceptable.

TABLE II
THEORETICAL PEAK AND RESULT OF EACH KERNEL

Kernel	Theoretical peak	result
Distance Computation	12	11.747526
Points Labeling	2	1.678838
New Cluster Computation	4	1.287277

F. Performance Plots

To evaluate the performance of our code, we firstly measured the FLOPS/cycle of our single thread SIMD implementation. Then, we started to increase the threads in parallel computing to find the number of threads with the best performance. At the same time, we also tested our code in data sets of different size to ensure its correctness and scalability.

Figue.8 shows the FLOPS / cycle of our implementation and baseline. It can be clearly seen that the performance we achieved is 50 times or even more better than the baseline. And when the data set increases, it maintains the stable performance. Besides, the performance decrease when increasing the dimension of points from 64 to 128 can be explained by the cache layout in Section.IV-A.

After completing the single-threaded design, we started to use OpenMP to achieve multi-threaded application. We use threads ranging from 1 to 40 to run our program, and the result is presented in Fig.9. Since we need to make a local copy in every thread to solve the Cache Ping-Pong problem (Section.V-C), and the memory of local copies need to be cleared for each iteration, this part of the overhead cannot be ignored. And that's why when the number of threads increases to two, the FLOPS / cycle will decrease instead when the dimension of points is 4. However, when increasing data set and adding more threads in parallel computing, a huge performance improvements has been brought about. In addition, since the number of hardware threads in our machine is 40, when the number of threads reaches this value, not only the overhead of creating new threads, but also the cost of emptying a large block of memory for local copies, would

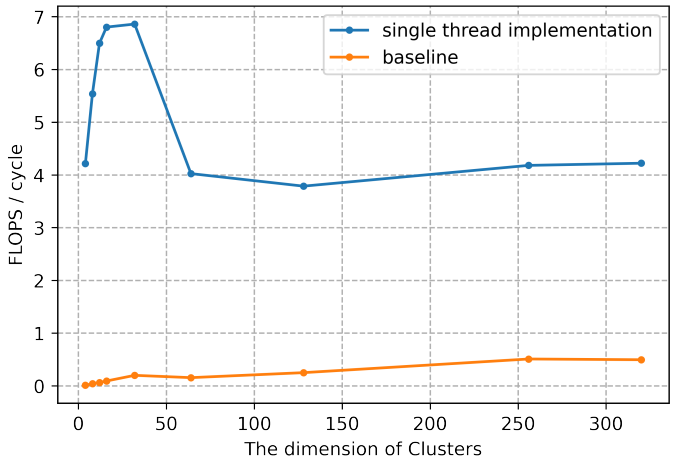

Fig. 8. The performance of our single thread implementation and baseline.

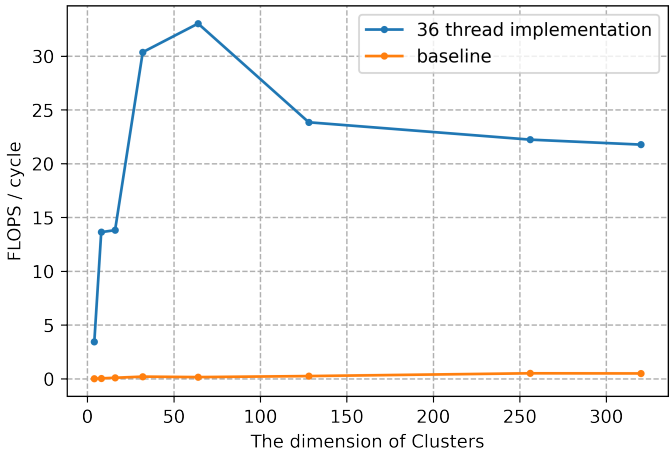

Fig. 10. The performance of baseline and our program running in 36 threads

reduce the performance of our model, which matches the result in the plot.

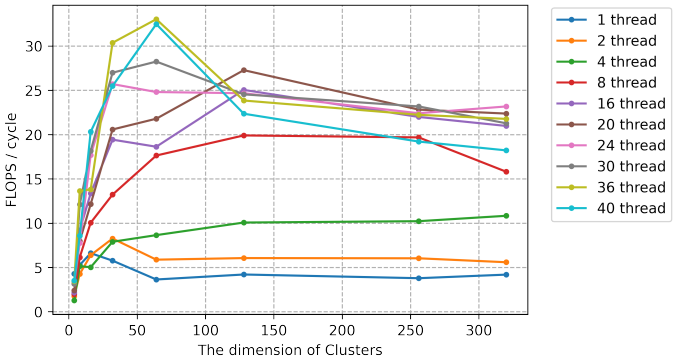

Fig. 9. The performance of our program running in different number of threads

Finally, as shown in Figue.10, if we compare the baseline with our best model (program runs in 36 threads), our performances has exceeded the baseline by hundreds of times.

VII. Conclusion and Future Work

The report introduces a thorough way to optimize k-means algorithm on CPU arcitecture form low level SIMD instructions and cache issues to higher level OpenMP parallelism. The proposed implementation has achieved good performance improvement on the given test data sets.

Even through our distance computation achieve more than 95% of the theoretical peak, the new clusters computation kernels inevitably lose some performance since the size of this kernel is limited to the registers and dependent on the output of previous kernel. Also, our k-means implementation is not memory efficient, in terms of the design decision we made to duplicate input in different layouts to utilize SIMD instructions and maximize locality for different computation. The other lesson we learnt from this project is that the cache utilization is really a tricky issue. Even through we have explored some methods to make use of cache (i.e. contiguous panels layout and thread local copy of shared memory), the performance drops earlier than we expected when the dimensions of data set increases. If we had more time in this semester, we would spend more time diving deep into these issues, and explore a more sophisticated data layout and kernel design with better performance.

Moreover, the implementation has been focused on relatively small data sets. If the data set becomes large enough, the overhead of moving data from host to GPU device will be amortized by the significant parallel computation power brought by GPU. Thus, the performance of a good implementation of k-means on GPU architecture may be better than ours on large data set.

References

[1] K-means image
https://www.analyticsvidhya.com/blog/2021/04/k-means-clustering-simplified-in-python/
[2] Mini-Batch k-means
https://www.geeksforgeeks.org/ml-mini-batch-k-means-clustering-algorithm/
[3] Broadwell Architecture info
https://www.agner.org/optimize/instruction_tables.pdf
[4] CPU_info
https://www.overleaf.com/project/61a84a7e54190185385cf07e
[5] sklearn_makeblobs
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
[6] sklearn_KMeans
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html