Fast K-Means Clustering using AVX2 SIMD

Yu-Chen Huang
yuchenhu

Qixuan Feng
qixuanf

Michael Li
mcli

***Abstract*—K-Means Clustering is an algorithm which partitions a dataset of points into k clusters such that the distance between each point and its assigned cluster is minimized. The algorithm will iteratively find clusters that converge to some local minimum by repeatedly finding out which cluster each point belongs to, and then re-assigning each cluster to the mean of the points assigned to it.**

By limiting the dimensions of the input dataset, k-means can be sped up greatly with the use of Intel AVX2 SIMD.

I. Separating K-Means into Kernels

A. *K-Means Structure*

K-Means can be generally implemented in this fashion:

```
func K-Means(data, k):
   1. Initialize k points in the
       dimensionality of the dataset as the
       initial guesses for the clusters,
       either randomly or with some heuristic
       to improve correctness

   2. Loop until convergence:
      3. Assign each point in the data to its
          closest cluster:
         3a. Loop through each point in the
             dataset and get its squared
             distance from each cluster, for
             computational efficiency.

         3b. Loop through each point, find the
              closest (minimum distance)
             cluster, and assign it to that
             point. Here, we can also count
             the number of points assigned to
             each cluster.

      4. Loop through each cluster and assign
          it to be the average of all points
          assigned to it
         4a. Loop through each cluster and get
              the sum of the points assigned
             to it

         4b. For each of the point sums,
             divide by the number of points
             assigned to each cluster to get
             the average of the points. Assign
              the cluster to its average.

At the end of 2, K-Means will return the
    converged cluster means.
```

B. *Kernels*

There are parallelizable operations in parts 3 and 4:

- Part 3 is independent between each point: the closest cluster for each point is independent from the closest cluster for all other points, so finding it is independent between points.
  - Part 3b is dependent on 3a: to get the minimum distance cluster for a point, we need to find the squared distance to each cluster first.
- Part 4 is independent between clusters: finding and assigning the average of the points assigned to a cluster is independent from the other clusters, since each cluster already has its own assignment of points.
  - Part 4b is dependent on 4a: to get the average of the points assigned to a cluster, we need to get the sum of the points and the number of points assigned to the cluster first.
- Part 4 is dependent on part 3: finding the new cluster means is dependent on finding the points assigned to the cluster first.

Because part 3 is independent between points and part 4 is independent between clusters, we can parallelize the computation in these parts.

Our kernels are parts 3a, 3b, 4a and 4b.

II. Hardware Requirements, Input Restrictions and Memory Layout for the Kernels

A. *Functional units used in each kernel*

We implemented and tested our kernels on ECE Machine 10, which has a Intel(R) Xeon(R) CPUE5-2640v4 processor that runs at a base frequency of 2.40GHz.

- Part 3a requires using FMAs and subtracts, as the fundamental operation to get the squared distance to each cluster is $\delta(point, cluster) = \sum_i (point_i - cluster_i)^2$, where each i is a dimension in the data.
- Part 3b requires using at least mins and compares to get the min of an array of distances.
- Part 4a requires using adds, as the fundamental operation is summing up the points in each cluster, i.e. sums[cluster][i] += point[i] .
- Part 4b requires using divides, as the fundamental operation is dividing each cluster's sum by the number of points assigned to the cluster.

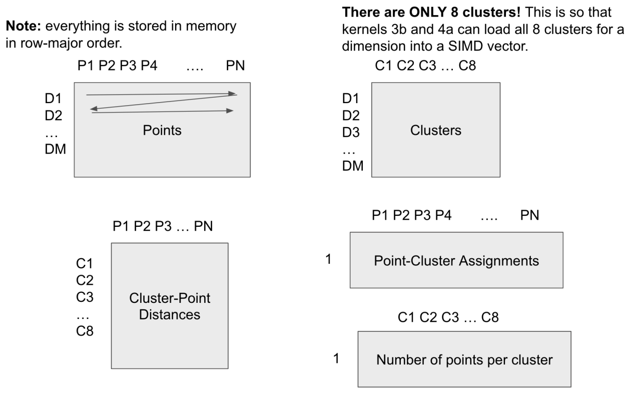
We have all the SIMD versions of the instructions needed to write and run our kernels on this machine.

B. Memory Layout Design

Here are the memory layout and definitions for each of the matrices that our k-means kernels use. Assume that there are n points, m dimensions, and exactly 8 clusters.

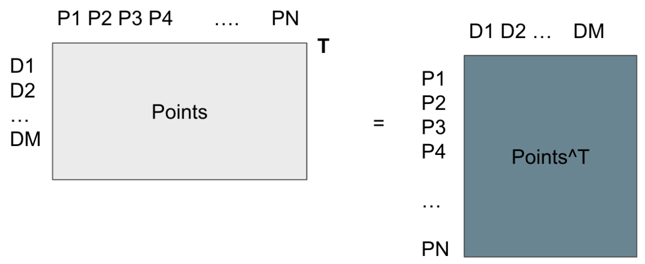
Arrays used by the kernels:

1) "Points" is the input to the function. It's used by kernels 3a and 4a. Its rows are dimensions for each point, and its columns are each point. Each column represents the coordinates of a point in the dataset.
2) "Clusters" is the output of the function. It's calculated by kernel 4a and 4b and used by kernel 3a. Its rows are dimensions for each point, and its columns are each cluster. Each column represents the coordinates of a cluster mean.
3) "Point-Cluster Distances" is calculated by kernel 3a and used by 3b. Its rows represent each point and its columns represent each cluster. Each entry is the distance between each point and cluster.
4) "Point-Cluster Assignments" is calculated by kernel 3b and used by kernel 4a. It only has one row, and each column is for a point. Each entry represents the cluster that a point is assigned to.
5) "Number of points per cluster" is calculated by kernel 3b and used by kernel 4b. It only has one row, and each column is for a cluster. Each entry represents the number of points assigned to each cluster.


Note: everything is stored in memory in row-major order.
P1 P2 P3 P4 …. PN
D1
D2
…
DM
Points
There are ONLY 8 clusters! This is so that kernels 3b and 4a can load all 8 clusters for a dimension into a SIMD vector.
C1 C2 C3 … C8
D1
D2
D3
…
DM
Clusters
P1 P2 P3 … PN
C1
C2
C3
…
C8
Cluster-Point Distances
P1 P2 P3 P4 …. PN
1
Point-Cluster Assignments
C1 C2 C3 … C8
1
Number of points per cluster

C. Packing the Points Matrix

To speed up kernel 4a, we need to pack the points matrix into an easier format for the kernel to be able to get the points' coordinates into a SIMD vector. At the beginning of the kernel, we do a "packing" routine where we transpose the points matrix.



D. Input Restrictions

These are the input restrictions we have on the dataset. They will be explained in detail in the designs of the individual kernels.

Input Restrictions based on each kernel:

- For kernel 3a to work, the number of points must be a multiple of 7 and 8, i.e. a multiple of $LCM(7,8) = 56$.
- For kernel 3b to work, the number of clusters must be exactly 8.
- For kernel 4a to work, the number of clusters must be exactly 8 and the number of points must be a multiple of 8.
- For kernel 4b to work, the number of clusters must be exactly 8.

III. Updated Kernel Designs

A. Kernel 3a

To measure the distance between each point and each cluster, we need to calculate the sum of the squares of difference between each dimension of two different points. To accelerate the calculation process, we use SIMD instruction to do the calculation in parallel and then get the result of all the distance between each point and each cluster.

To calculate the distance between each point and each cluster, we need to loop through each point. Inside the loop, we loop through each cluster. For each point and each cluster, we need to calculate the difference between each dimension using SIMD SUB instruction. Then after that, we need to take the square of each difference and then add the squares together by using SIMD FMA instructions.

To calculate the distance between each point and each cluster, the steps are:

```
Loop through each cluster
   Loop through each point
      1. Initialize distance variable by
          _mm256_setzero_ps
      Loop through each dimension
         2. Package this dimension of 8 points
              into one SIMD __m256 register by
              using _mm256_loadu_ps
         3. Broadcast this dimension of the
             cluster to __m256 by
             _mm256_broadcast_ss
         4. Using SIMD SUB instruction
             _mm256_sub_ps to subtract 8-D
             packages of a point and a cluster
```

```
5. Using SIMD FMA instruction
    _mm256_fmadd_ps to calculate the
    result and store it in the
    distance variable
6. Store the result distance of each
    eight-package points to this
    cluster to memory by
    _mm256_storeu_ps
```

1) Input Restrictions: Since we use SIMD 256-bits single-precision floating-point instructions, the size of the data, which is an array of points, must be a multiple of eight. Therefore, the number of points in the data must be a multiple of eight.

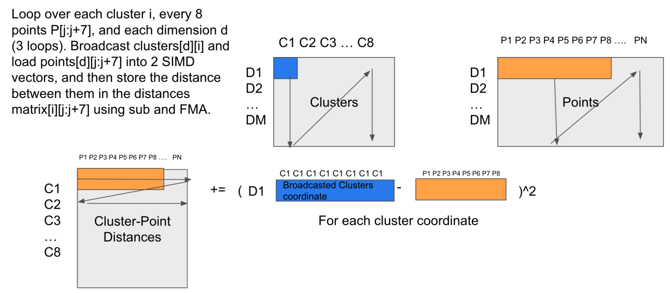
Additionally, to fill up the register limit, the inner loop can process seven points at a time, so the number of points must be a multiple of 8 and a multiple of 7, i.e. a multiple of $LCM(7, 8) = 56$.

2) Implementation of the kernel under register limitation: On the same dimension, we need one register to load 8 points, and one register to load 1 cluster, and one register to store the result. After using one register to broadcast 1 dimension of cluster into 8 float simd, calculating distance between 8 points and 1 cluster of the same dimension needs only 2 registers:

1) 1 register for load 8 points of cluster's dimension, and then use it as input of SIMD SUB and then reuse it to store the SUB result.
2) 1 register to use as SIMD FMA's input, initialized as zeros, and do FMA: store the result of each dimension of (8 point - 1 cluster) squared, to sum up and get the result of the distance, and then reuse it to store the result into memory.

Since we only have 16 SIMD 256-bit registers, we can only maximize the usage of SIMD registers by using 16-1 = 15 registers to calculate multiple points at a time. 15/2 = 7.5. Since the number of clusters is set to be 8, the number of clusters going through in the kernel must be a number that divides 8 evenly, so the number of points must be a multiple of 8 and a multiple of 7, i.e. a multiple of $LCM(7, 8) = 56$.

This diagram details the operation of broadcasting one dimension of one cluster into a SIMD vector and computing the distances between eight points and the cluster in that dimension:


Loop over each cluster i, every 8 points P[j:j+7], and each dimension d (3 loops). Broadcast clusters[d][i] and load points[d][j:j+7] into 2 SIMD vectors, and then store the distance between them in the distances matrix[i][j:j+7] using sub and FMA.
C1 C2 C3 … C8
D1
D2
…
DM
Clusters
P1 P2 P3 P4 P5 P6 P7 P8 …. PN
Points
C1
C2
C3
…
C8
Cluster-Point Distances
+=
(D1
C1 C1 C1 C1 C1 C1 C1 C1
Broadcasted Clusters coordinate
-
P1 P2 P3 P4 P5 P6 P7 P8
)^2
For each cluster coordinate

The pseudo code for this kernel is:

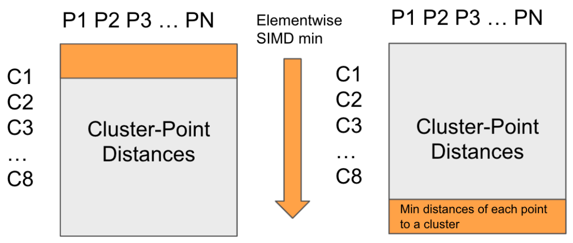
```
For each each clusters C_i in C: {
   For each seven points [P_j,...,P_j+6] in P:
        {
      #initialize distance = zeros
            YMM0 = SETZERO()
      YMM1= SETZERO()
      YMM2 = SETZERO()
      YMM3 = SETZERO()
      YMM4 = SETZERO()
      YMM5 = SETZERO()
      YMM6 = SETZERO()

            For dimension k of these 56 points
                 (seven times eight points [
               P_j,...,P_j+7] package) and 1
               clusters C_i: {
              # broadcast 1 dimension of C_i
                YMM7 = BCAST(C_i_k)

         # load 56 points of same dimension
         YMM8 = LOAD([P_j..j+7]_k)
         YMM9 = LOAD([P_j+8..j+15]_k)
         YMM10 = LOAD([P_j+16..j+23]_k)
         YMM11 = LOAD([P_j+24..j+31]_k)
         YMM12 = LOAD([P_j+32..j+39]_k)
         YMM13 = LOAD([P_j+40..j+47]_k)
         YMM14 = LOAD([P_j+48..j+55]_k)

         # calculate distance by SUB and then
             FMA
         YMM8 = SUB(YMM7, YMM8)
         YMM9 = SUB(YMM7, YMM9)
         YMM10 = SUB(YMM7, YMM10)
         YMM11 = SUB(YMM7, YMM11)
         YMM12 = SUB(YMM7, YMM12)
         YMM13 = SUB(YMM7, YMM13)
         YMM14 = SUB(YMM7, YMM14)

         YMM0 = FMADD(YMM8, YMM8, YMM0)
         YMM1 = FMADD(YMM9, YMM9, YMM1)
         YMM2 = FMADD(YMM10, YMM10, YMM2)
         YMM3 = FMADD(YMM11, YMM11, YMM3)
         YMM4 = FMADD(YMM12, YMM12, YMM4)
         YMM5 = FMADD(YMM13, YMM13, YMM5)
         YMM6 = FMADD(YMM14, YMM14, YMM6)
      }

      # store the result
      STORE([P_j..j+7]_k, YMM0)
      STORE(P_j+8..j+15]_k, YMM1)
      STORE([P_j+16..j+23]_k, YMM2)
      STORE([P_j+24..j+31]_k, YMM3)
      STORE([P_j+32..j+39]_k, YMM4)
      STORE([P_j+40..j+47]_k, YMM5)
      STORE([P_j+48..j+55]_k, YMM6)
   }
}
```

B. Kernel 3b

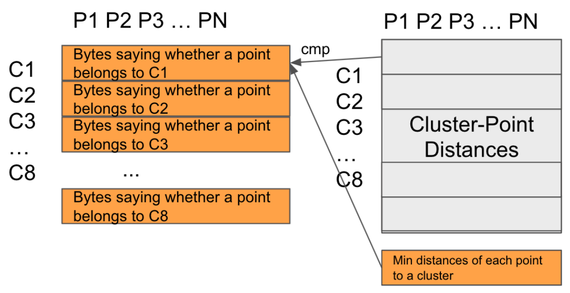
1) Design: Given the cluster-point distances, stored in row-major order with each cluster as a row and each point as a column, we know that each column represents each cluster's distance from a point. We need to find the minimum cluster for each point, which is the argmin of each column.

To do this, we go in three steps: we find the min of each column, mask the columns of the distances matrix to become zeros and a single one at the index of the argmin, and finally use intrinsics to get the index of the argmin for each of the points. We can do this process with eight columns/points at a time, since a SIMD vector holds 8 floats at a time.

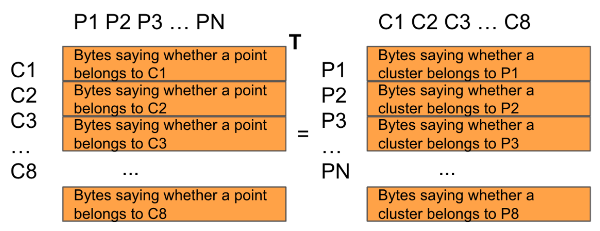
1) Our first step is to use SIMD to get the min of each column. We do this by loading the first row of the matrix into a SIMD vector, and then using *_mm256_min_ps* to min the vector each row of the matrix. The resulting vector has the minimum values for each column. Since each column has a point's distances to the clusters, this means that the vector has the minimum distance to a cluster for each point.



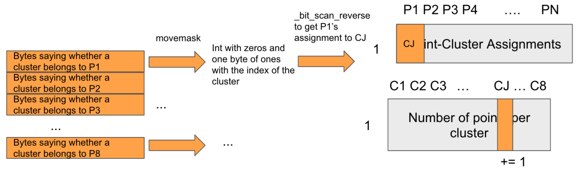
2) Now that we have the minimum distances for each point in a cluster, we can use *_mm256_cmp_ps* to find the index within each column of that column's minimum value. Since we only have eight clusters, we can put the results of *_mm256_cmp_ps* and each row of the distances matrix into eight SIMD registers. This allows us to do the next steps using a matrix of clusters and the points belonging to the cluster stored solely in registers. After doing *_mm256_cmp_ps* between each row of the matrix and the minimum distances vector, each of the eight SIMD vectors/rows has bytes of all zeros or all ones saying whether each point belongs to a certain cluster (as a one will only occur if the value in the distances matrix for that point and cluster are the actual minimum distance).


P1 P2 P3 … PN
C1
C2
C3
…
C8
Bytes saying whether a point belongs to C1
Bytes saying whether a point belongs to C2
Bytes saying whether a point belongs to C3
…
Bytes saying whether a point belongs to C8
cmp
P1 P2 P3 … PN
C1
C2
C3
…
C8
Cluster-Point Distances
Min distances of each point to a cluster

3) Now we transpose the eight SIMD vectors in the registers to get a matrix with each row representing each point, and each column representing whether the point belongs to a cluster. This format allows us to figure out which cluster each point belongs to, since it's now a row in the matrix.


P1 P2 P3 … PN
C1
C2
C3
…
C8
Bytes saying whether a point belongs to C1
Bytes saying whether a point belongs to C2
Bytes saying whether a point belongs to C3
…
Bytes saying whether a point belongs to C8
T
=
C1 C2 C3 … C8
P1
P2
P3
…
PN
Bytes saying whether a cluster belongs to P1
Bytes saying whether a cluster belongs to P2
Bytes saying whether a cluster belongs to P3
…
Bytes saying whether a cluster belongs to P8

4) Using this transposed matrix, we can use *_mm256_movemask_ps* on each SIMD vector into an integer representing the same zeros and ones, and use *_bit_scan_reverse* to find the largest index that has a one for the point, which will be the index of the cluster that the point belongs to.
From here, we can store the assignment into an array of point-cluster assignments and increment the number of points in the assigned cluster.

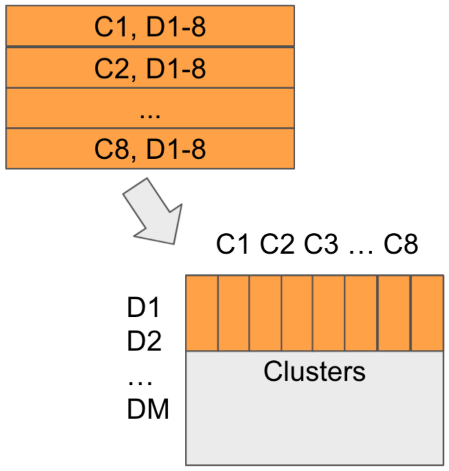

Bytes saying whether a cluster belongs to P1
Bytes saying whether a cluster belongs to P2
Bytes saying whether a cluster belongs to P3
…
Bytes saying whether a cluster belongs to P8
movemask
…
Int with zeros and one byte of ones with the index of the cluster
…
_bit_scan_reverse to get P1's assignment to CJ
P1 P2 P3 P4 …. PN
1
CJ
int-Cluster Assignments
C1 C2 C3 … CJ … C8
1
Number of poin per cluster
+= 1

2) *Restrictions to the input:* This section requires that there are exactly eight clusters, as the clusters will need to fit into eight SIMD registers. It also requires that the number of points in the dataset is a multiple of eight, as the kernel will find cluster assignments for eight points at a time (the points are stored in a SIMD vector of size eight).

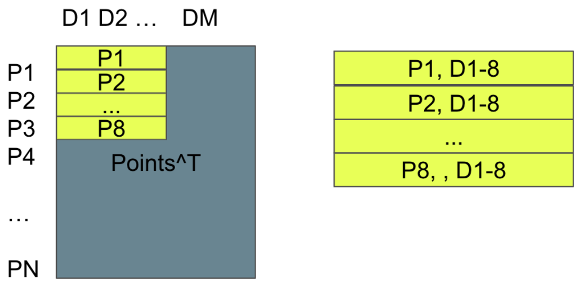
C. Kernel 4a

1) *Design:* In this kernel, we need to sum up all of the points assigned to each cluster so that we can calculate the mean of each cluster. Since we transpose the points matrix before the kernels start via our packing routine at the start of the program, we can use it to get the coordinates of multiple points in parallel and calculate cluster sums using the coordinates. We'll use eight SIMD vectors, one for each cluster, to store the sums of the points assigned to each cluster. The registers act as a cluster matrix, where each row is the sum of points in 8 dimensions for a cluster. Then, we can get eight points at a time, store eight of their dimensions into SIMD vectors, and add their points to their assigned cluster's SIMD vector in parallel. After the sums are calculated, to store the cluster's SIMD vectors into the real cluster matrix, which has clusters as columns and not rows, we transpose the registers' cluster matrix and store it into the actual cluster matrix in memory.

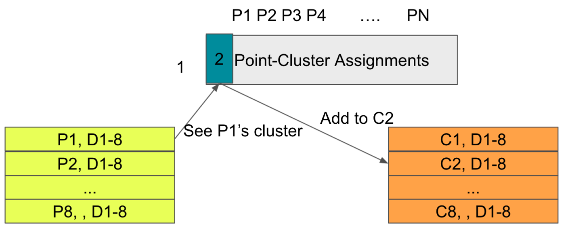
1) The first step is to zero out eight SIMD vectors. Each SIMD vector will hold the sums for a cluster. Note that in the actual cluster matrix in memory, each cluster is represented by a column in the matrix. In the registers, the cluster coordinates are stored in a row, which is why we can use SIMD adds.



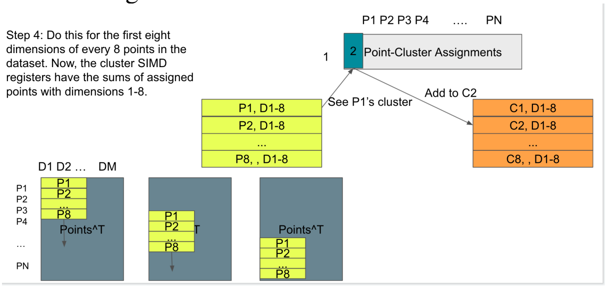
2) We load eight points' coordinates at a time from the transposed points matrix into eight SIMD vectors.



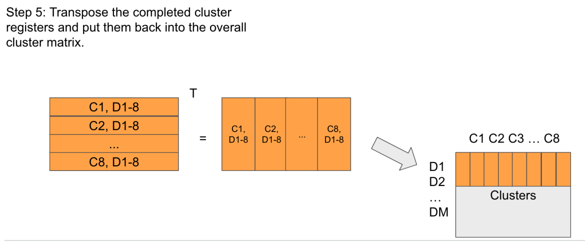
3) Based on the point-cluster assignments, each point's SIMD vector can be added using $\_mm256\_add\_ps$ to the assigned cluster's SIMD vector.



4) Repeat steps 2 and 3 for every eight points, so that we add every point's eight dimensions to the eight clusters. The clusters now have the sum of their assigned points for these eight dimensions.


Step 4: Do this for the first eight dimensions of every 8 points in the dataset. Now, the cluster SIMD registers have the sums of assigned points with dimensions 1-8.
P1 P2 P3 P4 …. PN
1
2
Point-Cluster Assignments
See P1's cluster
Add to C2
P1, D1-8
P2, D1-8
...
P8, , D1-8
C1, D1-8
C2, D1-8
...
C8, , D1-8
D1 D2 … DM
P1
P2
P3
P4
…
PN
Points^T

5) To store the cluster's sums back in memory, we transpose the registers with the clusters so that they match the memory layout of the cluster matrix. Then, we store the registers back in memory with the cluster sums.


Step 5: Transpose the completed cluster registers and put them back into the overall cluster matrix.
C1, D1-8
C2, D1-8
...
C8, D1-8
T
=
C1, D1-8
C2, D1-8
...
C8, D1-8
C1 C2 C3 … C8
D1
D2
…
DM
Clusters

6) Do steps 1-5 (summing up eight dimensions of point coordinates for the clusters) for every eight dimensions. Since our SIMD vector size is eight, we can only get eight dimensions at a time, so the number of dimensions must be a multiple of eight.

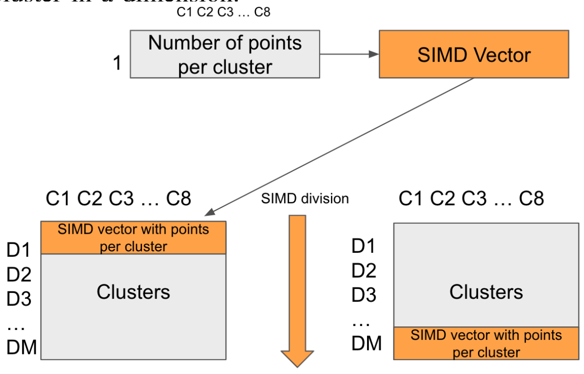
2) Restrictions to the input: This section requires that there are exactly eight clusters, and that the number of points in the dataset is a multiple of eight.

The reason there are exactly eight clusters is because the clusters need to be able to fit in the SIMD registers, since we don't know which cluster a point is assigned to- if there's too many clusters, then we can't store the clusters in SIMD registers, and have no efficient way to add points to their assigned clusters.

D. Kernel 4b

1) Design: To compute the mean of each cluster for updating each cluster's centroid , we just need to divide the sum of each cluster's coordinates by the number of points in each cluster. For the sum of each cluster, we can use SIMD to parallelize the division of different dimensions.

1) We load the number of points per cluster into a SIMD register. This way, we can divide each of the rows of the cluster matrix by the number of points in each cluster, which gets us the average coordinate in each dimension for all the clusters.
2) We use $\_mm256\_div\_ps$ to load in each row of the cluster matrix (representing the cluster's coordinates in one dimension) into a SIMD vector, and elementwise-divide it by the SIMD vector with the number of points per cluster. Storing this back into the cluster matrix, each row of the matrix has the average coordinate of each cluster in a dimension.


C1 C2 C3 … C8
1
Number of points per cluster
SIMD Vector
C1 C2 C3 … C8
SIMD division
C1 C2 C3 … C8
SIMD vector with points per cluster
D1
D2
D3
…
DM
Clusters
D1
D2
D3
…
DM
Clusters
SIMD vector with points per cluster

2) Restrictions to the input: Since all of the SIMD operations in the kernel involve loading in all the clusters into a

SIMD vector and dividing another SIMD vector which has the number of points in each cluster, and the length of a SIMD vector is eight, then the number of clusters must again be exactly eight for this kernel to work.

IV. Parallelization

A. Kernel 3a

In this kernel, the outer loop is parallelizable. Since each of the outer loop iterations in kernel 3a will write to disjoint sections of the $cluster\_point\_distances$ matrix while sharing reads from the constant $points$ matrix, the outer loop iterations do not share volatile memory. Then, the outer loop is parallelizable with an OpenMP parallel for loop and no critical sections.

B. Kernel 3b

For this kernel, the outer loop which iterates over eight points at a time is *mostly* parallelizable, since each point's assigned cluster is independent from any other point's assigned cluster in memory. Each thread accesses a different part of memory when storing the assigned cluster for each point, so there are no races with respect to the point-cluster assignments.

However, the matrix with the number of points in each cluster is shared memory among the concurrent threads of execution. In order to prevent races, the increments to the number of points matrix must be declared as atomic/a critical section. This serializes the concurrent operations and dramatically slows down performance, but is necessary to maintain correctness.

For parallelization with OpenMP, the kernel is declared as a parallel region, and the loop iterations are split mostly evenly among the number of threads available for OpenMP's use. The increments to the number of points per cluster are all declared as atomic. We found that this approach yielded significantly worse throughput than just having a single thread, likely due to thread creation/deletion overhead and contention overhead at the critical section.

C. Kernel 4a

In this kernel, the outer loop is parallelizable. Since each of the outer loop iterations in kernel 4a will write to disjoint sections of the $clusters$ matrix while sharing reads from the constant $points\_transpose$ matrix, the outer loop iterations do not share volatile memory. Then, the outer loop is parallelizable with an OpenMP parallel for loop and no critical sections.

D. Kernel 4b

In this kernel, the loop that divides each row of the $clusters$ matrix by the number of points in each cluster is parallelizable. Since each of those loop iterations will write to a disjoint chunk of 8 contiguous rows in the $clusters$ matrix while sharing reads from the constant $num\_points\_in\_cluster$ matrix, the loop iterations do not share volatile memory. Then, the loop that divides the rows of the $clusters$ matrix is parallelizable with an OpenMP parallel for loop and no critical sections.

V. Performance Plots

A. Theoretical Peak Calculations

1) Kernel 3a: To compute the "euclidean distance" by using SIMD instruction, it's obvious to see that we can use SIMD SUB instruction and SIMD FMA instruction. To make more data packed in one package and increase the effect of SIMD parallel computation, let's set the type of data of all kinds of points (both data point and cluster) to be 4-Bytes floating point. Thus we can use single-precision floating-point SIMD instruction. The only SIMD single-precision floating-point SUB instruction is $\_mm256\_sub\_ps$, which has latency 3 and throughput 1(CPI)=1(IPC) on our ECE machine (Broadwell), using pipeline p1. The only SIMD single-precision floating-point FMA instruction is $\_mm256\_fmadd\_ps$, which has latency 5 and throughput 0.5(CPI)=2(IPC) on our ECE machine (Broadwell), using pipeline p01. Since they both use p1, best case is all FMA instructions go through in pipeline 0 and all SUB instructions run in pipeline 1, and hence one FMA instruction and one SUB instruction come out at each cycle.

Since calculating each point's distance to each cluster is independent, and to calculate a distance we need to use both two SUB and 1 the theoretical peak Thus, the theoretical peak of this kernel is (1 FMA + 1 SUB) = (2 SIMD/cycle + 1 SIMD/cycle) = 3 SIMD/cycle = 24 FLOP/cycle.

However, kernel 3a has a tighter bound on the throughput than the theoretical peak. The latency of SIMD FMA is 5, and the best case is that SIMD SUB uses pipeline 1 and SIMD FMA uses pipeline 0. Using only one pipeline, to reach the theoretical peak, we must avoid the bubbles in between the stages in the pipeline. Hence, the least number of independent FMA instructions in the kernel is 5.

From kernel 3a's kernel size calculation in the "kernel design" section, we know that the kernel implementation has 4 independent FMA instructions per loop when fitting the requirements as best as possible given the register limits. We cannot do more than 4 cluster broadcasts at a time without using more than 16 registers, and therefore cannot avoid bubbles in the pipeline. **Kernel 3a's highest possible throughput in practice will be lower than the calculated theoretical peak for this reason.**

2) Kernel 3b: Since this kernel does each of the distinct SIMD instructions/sections dependently, the theoretical peak will be the "bottleneck" instruction between $\_mm256\_min\_ps$, $\_mm256\_cmp\_ps$, $\_mm256\_movemask\_ps$, and $bit\_scan\_reverse$. Each of these instructions has throughput 1 (per the Agner document and Intel docs). Since the SIMD vector size is 8 floats per vector for all of the instructions, the theoretical peak for this kernel is:

1 SIMD/cycle = 8 FLOPs/cycle.

However, note that we must store the cluster assignments and the number of points per cluster into memory using the result of $bit\_scan\_reverse$ as an index before the next kernel can start. Additionally, we must perform the summation computation for clusters in registers, transpose the registers with SIMD permutes and shuffles, and store it back in memory due to the memory layout of the input and output. In this case, **kernel 3b's throughput is bounded by the machine's memory bandwidth and not the theoretical peak**. The layout of the problem necessitates that we perform the computation in this way, since we have to have the number of points assigned to each cluster to get the new cluster means, so this is an unavoidable property of the kernel's peak.

3) Kernel 4a: When we loop through all the points, since the sums of different dimensions of points are independent, we can use SIMD instruction $\_mm256\_add\_ps$ with latency 3 and throughput 1 on our ECE machine (Broadwell) to speed up these operations. To use $\_mm256\_add\_ps$, we pack the points' coordinates to be 8-Bytes floating point, then summing all points per cluster needs nm/8 times instruction. Thus, the theoretical peak for this kernel is:

1 SIMD/cycle = 8 FLOP/cycle

However, note that based on our design for this kernel, we must look up the cluster assignment for each point before adding its coordinates to the respective SIMD register. Additionally, we must perform the summation computation for clusters in registers, transpose the registers with SIMD permutes and shuffles, and store it back in memory due to the memory layout of the input and output. In this case, **kernel 4a's throughput is bounded by the machine's memory bandwidth and not the throughput of** $\_mm256\_add\_ps$. The requirement to find each point's assigned cluster necessitates that we perform the computation in this way, so this is an unavoidable property of the kernel's peak.

4) Kernel 4b: For each cluster, if there is any point in this cluster, we will average out each coordinate of points assigned to this cluster, otherwise we will initialize it randomly again. For the first case, when we loop through all the clusters, the average of different dimensions' coordinates are independent and the SIMD instruction we can use to speed up is $\_mm256\_div\_ps$ which has the latency of 13-17 and throughput of 1/16. Thus, the theoretical peak for this kernel is:

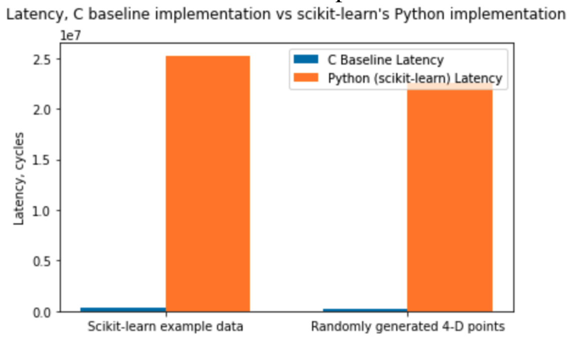
1/16 SIMD/cycle = 0.5 FLOP/cycle

B. Performance Plots

1) C Baseline Performance, compared to Scikit-Learn's Performance (Python): We created a naive C K-Means implementation which doesn't use any SIMD as both a starting point for our project and also a performance baseline.

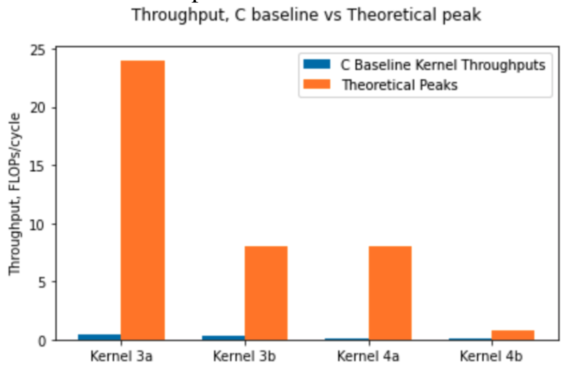
Comparing its performance to the scikit-learn Python implementation of K-Means, we find that the C baseline is much faster. This is due to C's significantly higher performance as a low-level, compiled language, compared to Python's lower performance due to overhead as a high-level, interpreted language.

We timed the cycle latency of entire runs of the C baseline and scikit-learn Python baseline on two datasets: the first is the scikit-learn example dataset, which is in 2-D and has two clusters. The second one is randomly generated 4-D points around two cluster means. The plot below shows how much faster the C baseline is when compared to scikit-learn:

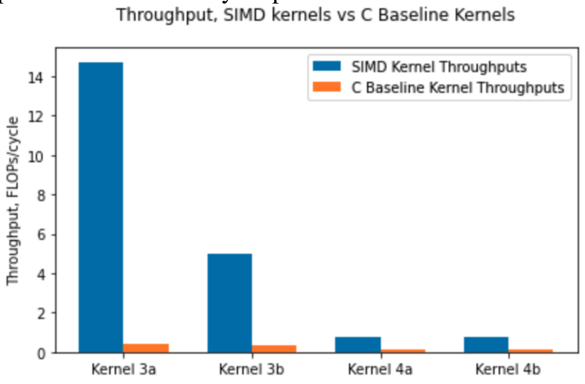


2) C Baseline Performance relative to the Theoretical Peaks: While the C baseline is much faster than the Python baseline, it is still not close to the theoretical peaks. The C baseline is also split into the four kernels written in section 1 to facilitate the timing of each baseline kernel.

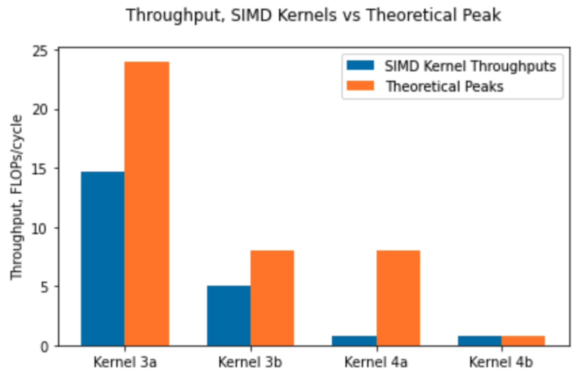
We timed the throughputs of each baseline kernel to the theoretical peak on the scikit-learn documentation example data. As shown in the plot below, the baseline is very far from the theoretical peaks we calculated in section 2:



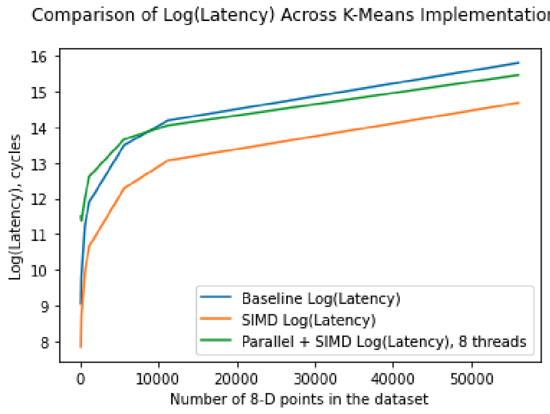
3) SIMD Kernels' Performance Relative to the Baseline Kernels': We timed the throughput of each kernel in both the C baseline implementation and the SIMD implementation, by calculating the total number of FLOPs in each kernel per iteration and dividing by the average latency per iteration of the kernel. As shown in the plot below, the new SIMD implementation is vastly improved on the C baseline:



4) SIMD Kernels' Performance relative to the Theoretical Peaks: The SIMD kernels have a throughput significantly closer to the theoretical peaks than the baseline kernels. As reflected in the theoretical peak calculations and discussion above, kernels 3b and 4a are further from the calculated theoretical peak because they require accessing memory, bounding their throughput by the memory access performance. Kernel 3a's throughput is also a bit further from the calculated theoretical peak because of unavoidable bubbles in the FMA functional unit's pipeline that bound its throughput.

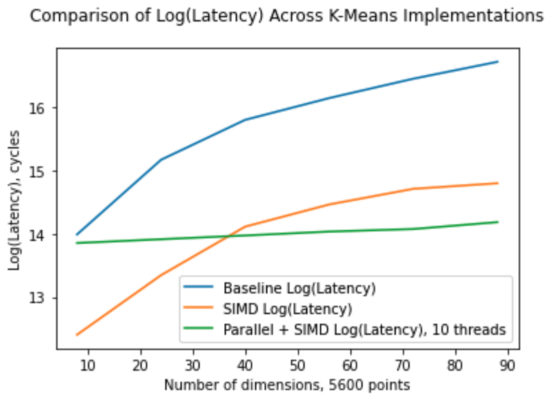


5) A Comparison of Latencies between the C Baseline Implementation, SIMD Implementation, and SIMD Implementation with OpenMP: The following plot compares the latency between the three implementations across the number of points in the input data, where the input dataset is randomly generated 8-D points. To better plot the latencies, this plot shows the log_e of the recorded latencies, or the "log latency":



The single-threaded SIMD implementation is noticeably faster than both the parallel SIMD and baseline implementations when comparing latency across the number of points in the input dataset. The parallel SIMD implementation is slower than the single-threaded SIMD implementation in this visualization, since this plot only compares growing the number of points, and kernels 4a and 4b are parallelized across the number of dimensions and not the number of points.

Because of this parallelization over the number of dimensions, OpenMP's speedup becomes clear when comparing latencies between the three implementations across the number of dimensions in the input dataset:



Since the OpenMP implementation was benchmarked with 10 threads, and kernels 4a and 4b spread the work over 8 dimensions at a time to each thread in parallel, the OpenMP implementation should stay around a constant latency up to 10 threads * 8 dimensions per thread = 80 dimensions. The plot reflects that claim: there is very little latency increase in dimensions until the dimensions exceed 80 dimensions. At the same time, the SIMD implementation and baseline implementation increase linearly with the number of dimensions, causing them to be significantly slower than the OpenMP implementation at higher dimensions.

Thus, the SIMD implementation provides a major speedup when compared to the baseline implementation in all cases, and OpenMP provides another major speedup when added to the SIMD implementation as the input dataset's dimensions increase.

VI. Future Directions

If the semester were much longer, future directions to further increase performance in the SIMD and SIMD + OpenMP implementations could be:

1) A big optimization would be to combine the four kernels into one monolithic kernel to minimize loads and stores. Although there are many necessary loads and stores that cannot be optimized out, there is some room for memory optimization.
2) Exploring the OpenMP parallelization of each of the kernels in deeper detail with different scheduling schemes and possibly different parallelization approaches might yield even better parallel performance.