CM IMPLEMENTATION OF K-MEANS

[Document subtitle]

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you're ready to add your content, just click here and start typing.]

Lueh, Guei-Yuan
[Email address]

k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. *k*-means clustering aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

https://en.wikipedia.org/wiki/K-means_clustering

http://stanford.edu/~cpiech/cs221/handouts/kmeans.html

High-level algorithm

```
typedef struct {
    float x;
    float y;
    int cluster;
} Point;
```

The input is an array of Point. Each Point has three fields. “x” and “y” are the point’s coordinate in x and y dimensions respectively. “cluster” indicates to which cluster the point belong. The total number of input points is `NUM_POINTS`.

```
typedef struct {
    float x;
    float y;
    int num_points;
} Centroid;
```

Centroid has 3 fields. “x” and “y” are the centroid’s coordinate in x and y dimensions respectively. num_points is the total points that has been clustered to this centroid.

For Gen9 and Gen11, the K-means process is comprised of two passes. The first phase, `cmk_kmeans`, divides input data into chunks with chunk size (`POINTS_PER_THREAD`). Each HW thread processes clustering for each chunk. The current CM implementation assumes that is `POINTS` divisible by `POINTS_PER_THREAD`. `cmk_kmeans` computes the minimum distance to determine to which cluster (centroid) a point belong. To facilitate the final computation of new centroid positions, each HW thread accumulates x and y coordinate of all points of its dedicated chunk for each cluster. The accumulated results are saved in an auxiliary data structure, accum. Each thread has its own local accum so that all threads can update their own local copy without global atomic update. The data structure used by a HW thread is depicted in Figure 1.

```
typedef struct {
  float x_sum;
  float y_sum;
  int num_points;
} Accum;
```

The second phase, `cmk_compute_centroid_position`, sums up accum_x and accum_y, num_points of each cluster and computes the new centroid positions.K-means process is invoked multiple iterations, `NUM_ITERATIONS`, for centroids to converge.

For Gen12+, we don’t need the second pass. With one global Accum with NUM_CENTROIDS entries, after the computation of accumulated x,y coordinates and number of points for each chunk is done, the results can be globally updated to Accum via FADD atomic (for x and y coordinate) and ADD atomic (for number of points). The final calculation of new centroid position can be done by CPU as the number of centroid is small. Gen12 HW is not yet available at the time this documentation is written. We are able to evaluate the performance of the two approaches.

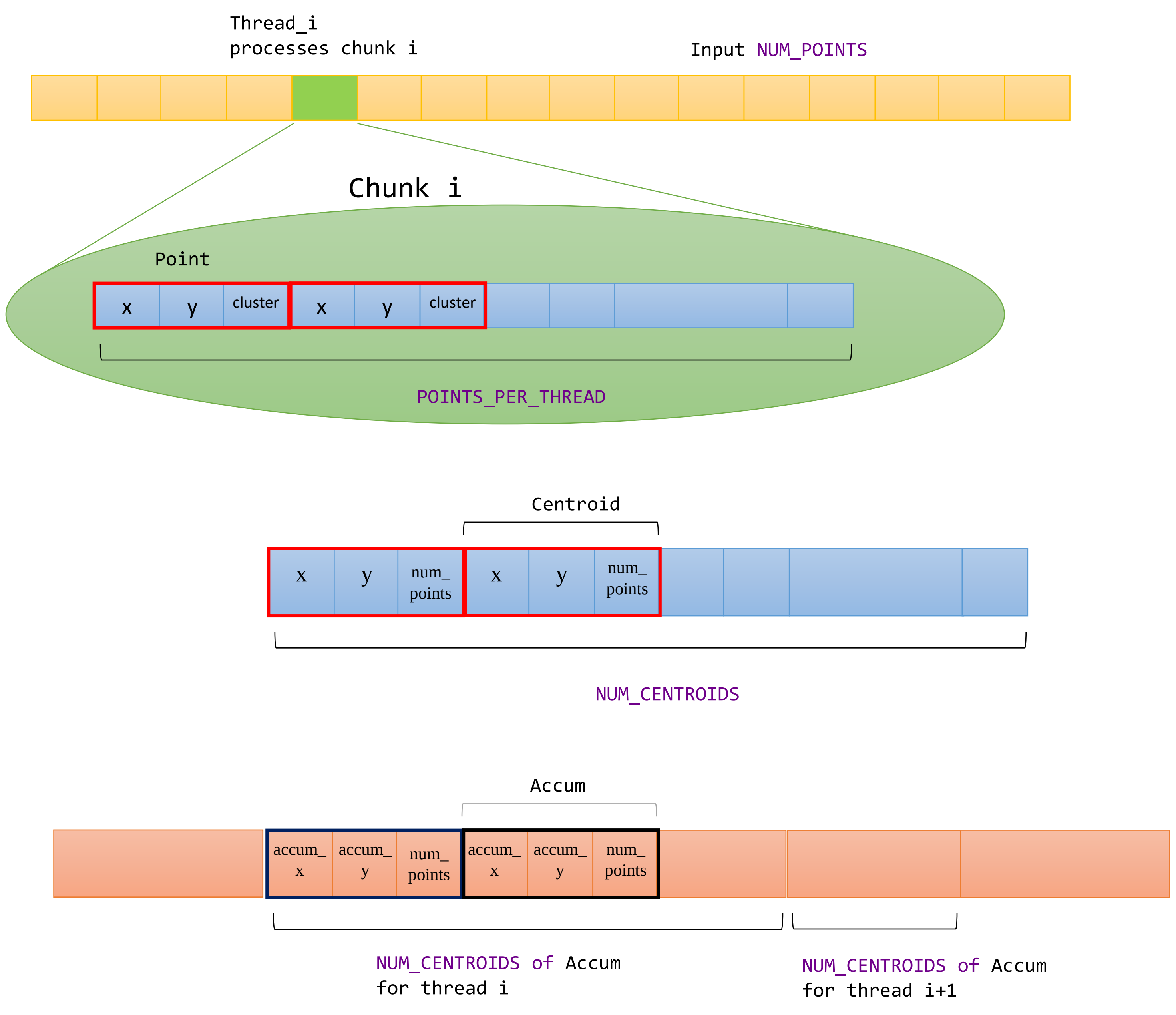

Figure 1. data structure

Pass 1: cmk_kmeans

Each HW thread computes POINTS_PER_THREAD points. From the experiments, we have tried 128 and 512 points per thread. Depending on the input size and system configuration, using a big chunk size may not necessary yield optimal performance because NUM_POINTS/POINTS_PER_THREAD threads may not be enough to saturate the system. Users need to profile to choose the right chunk size.

```
_GENX_MAIN_ void cmk_kmeans(
       SurfaceIndex pts_si,     // binding table index of points
       SurfaceIndex cen_si,     // binding table index of centroids
       SurfaceIndex acc_si,     // binding table index of accum for computing new centroids
       unsigned num_points,     // number of points
       bool final_iteration)    // if this is the final iteration of kmeans
{
```

k-mean process is iterated multiple times until centroids converge. There is no need to write out the intermediate clustering results. final_iteration is a flag passed in to tell this invocation is the final iteration and cluster result of each point needs to be stored.

```
       // read in all centroids
       // this version we can only handle number of centroids no more than 64
       // We don't need cluster field so we read only the first two field
       matrix<float, 2, ROUND_TO_16_NUM_CENTROIDS> centroids;
       vector<unsigned int, 16> offsets(init16);
       offsets = offsets*DWORD_PER_POINT;
```

We are about to perform untyped read from 16 locations. “offsets” is to compute the location of read data (shown in Figure 2)

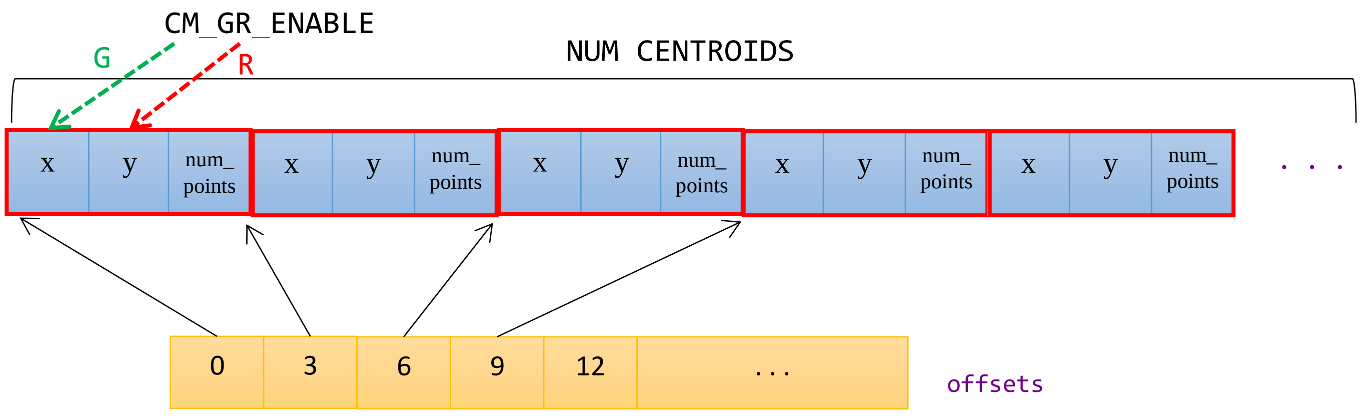

Figure 2. offsets for untyped read

```
#pragma unroll
       for (unsigned i = 0; i < ROUND_TO_16_NUM_CENTROIDS; i+=16) // round up to next 16
       {
              // untyped read will pack R, G into SOA
              // matrix.row(0): x x x x . . . . . // 16 position x
              // matrix.row(1): y y y y . . . . . // 16 position y
              read_untyped(cen_si, CM_GR_ENABLE, centroids.select<2,1,16,1>(0,i), offsets +
i*DWORD_PER_CENTROID);
       }
```

The first step reads in all centroids. To generate concise code sequence of subsequent SIMD computation of minimum distance, centroids are read in and kept in SOA format. Untyped read performs scatter read of 16 offsets. CM_GR_ENABLE flag dictates the read to read in Red and Green channels that offsets point to. Since all centroids are read in and saved in registers, there is a limit on NUM_CENTROIDS. The maximum number of centroids this kernel can process is 64. To handle NUM_CENTROIDS >= 64, we need to implement a different algorithm, e.g., accum and centroids resides in shared local memory.

```
        matrix<float, 3, ROUND_TO_16_NUM_CENTROIDS> accum = 0;
        matrix_ref<unsigned, 1, ROUND_TO_16_NUM_CENTROIDS> num = accum.row(2).format<unsigned, 1,
ROUND_TO_16_NUM_CENTROIDS>();
```

for efficient SIMD computation, accum is laid out in SOA as shown in Figure 3. NUM_CENTOIDS may not be divisible by 16. Due to matrix/vector layout in register file, the column size of a matrix is rounded to multiple of 16 so that each row of the matrix is aligned on GRF boundary. The declaration of matrix/vector takes only uniform type. Accum matrix is declared as float type. The third field, num_points, is unsigned. The third row is formatted from float to unsigned type.

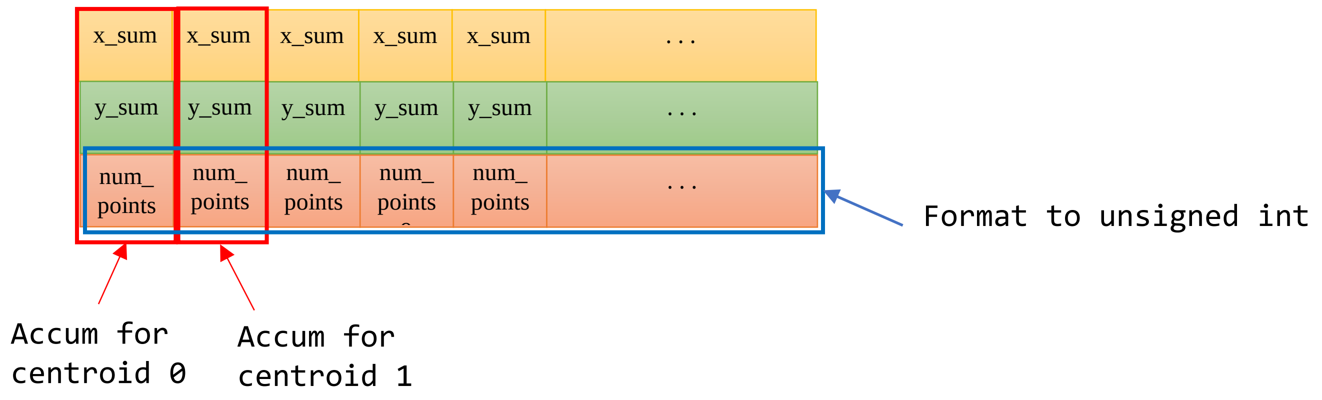

Figure 3. matrix<float, 3, ROUND_TO_16_NUM_CENTROIDS> accum

```
        uint linear_tid = cm_linear_group_id();
        // each thread handles POINTS_PER_THREAD points
        unsigned start = linear_tid * POINTS_PER_THREAD * DWORD_PER_POINT;  // each point has 3 DWORD
```

Each thread processes a chunk of POINTS_PER_THREAD points. Each points have 3 fields. Start is the chunk starting offset that the thread processes.

```
        // use untyped read to read in points.
        // Point is x, y, c
        // the returned result will be shuffled. x, y, c will be packed nicely
        for (unsigned i = 0; i < POINTS_PER_THREAD; i += 16)
        {
                matrix<float, 2, 16> pos;
                vector<unsigned, 16> cluster = 0;
                read_untyped(pts_si, CM_GR_ENABLE, pos, start+ offsets+i*DWORD_PER_POINT);
                vector<float, 16> dx = pos.row(0) - centroids(0,0);
                vector<float, 16> dy = pos.row(1) - centroids(1,0);
                vector<float, 16> min_dist = dx * dx + dy * dy;
#pragma unroll
                for (unsigned j = 1; j < NUM_CENTROIDS; j++)
                {
                        // compute distance
                        dx = pos.row(0) - centroids(0, j);
                        dy = pos.row(1) - centroids(1, j);
                        vector<float, 16> dist = dx * dx + dy * dy;
                         // track minimum distance and clustering index
                        cluster.merge(j, dist < min_dist);
                        min_dist.merge(dist, dist < min_dist);
                }
```

```
            // if this is the final invocation of kmeans, write back clustering
            // result
            if (final_iteration)
            {
                  // point: x, y, cluster
                  // i * DWORD_PER_POINT + 2 to write to cluster field
                  write(pts_si, start, offsets + i * DWORD_PER_POINT + 2, cluster);
            }
```

The maximum of untyped read is SIMD16, i.e., 16 points per read. For 16 points read in, the inner loop goes over each centroid to compute the distance and tracks the minimum distance and the clustering index. If the current k-means process is the final iteration, the clustering result is written back via scatter write. "start, offsets + i * DWORD_PER_POINT" is the location offsets for points. Adding 2 pointing to the 3rd DWORD (3rd fields) of point struct which is cluster field.

```
            // go over each point and according to their classified cluster update accum

#pragma unroll
            for (unsigned k = 0; k < 16; k++)
            {
                  unsigned c = cluster(k);
                  accum(0, c) += pos(0, k);
            }
#pragma unroll
            for (unsigned k = 0; k < 16; k++)
            {
                  unsigned c = cluster(k);
                  accum(1, c) += pos(1, k);

            }
#pragma unroll
            for (unsigned k = 0; k < 16; k++)
            {
                  unsigned c = cluster(k);
                  num(0,c)++;
            }

      }
```

After finding out to which clusters points belong, we accumulate points' coordinate to their corresponding accum of classified clusters. Likewise, the number of points in each cluster is updated as well.

```
      unsigned startoff = linear_tid * DWORD_PER_ACCUM * NUM_CENTROIDS;
#pragma unroll
      for (unsigned i = 0; i < ROUND_TO_16_NUM_CENTROIDS; i += 16) // round up to next 16
      {
            matrix<float, 3, 16> a16 = accum.select<3, 1, 16, 1>(0, i);
            SIMD_IF_BEGIN(offsets + i * DWORD_PER_ACCUM < NUM_CENTROIDS * DWORD_PER_ACCUM) {
                  write_untyped(acc_si, CM_BGR_ENABLE, a16, startoff + offsets + i * DWORD_PER_ACCUM);
            } SIMD_IF_END;
      }

}
```

Each HW thread writes out its own accum (NUM_CENTRIODS entries). The accum starting offset that the thread is writing out is "linear_tid * DWORD_PER_ACCUM * NUM_CENTROIDS". Predication (SIMD IF) ensures the only NUM_CENTROIDS accum are written out.

Pass 2: cmk_compute_centroid_position

In Pass1, each thread writes out NUM_CENTROIDS of Accum locally, one Accum for each centroid. This pass launches NUM_CENTROIDS threads, one for each centroid. Thread i sums up all Accum of centroid i by written by NUM_POINTS/POINTS_PER_THREAD threads in Pass1. As depicted in Figure 4. Thread 0 highlighted in yellow sums up centroid0's accum records pointed by light blue arrows. Thread 1 sums up all accum pointed by red arrows for centroid1.

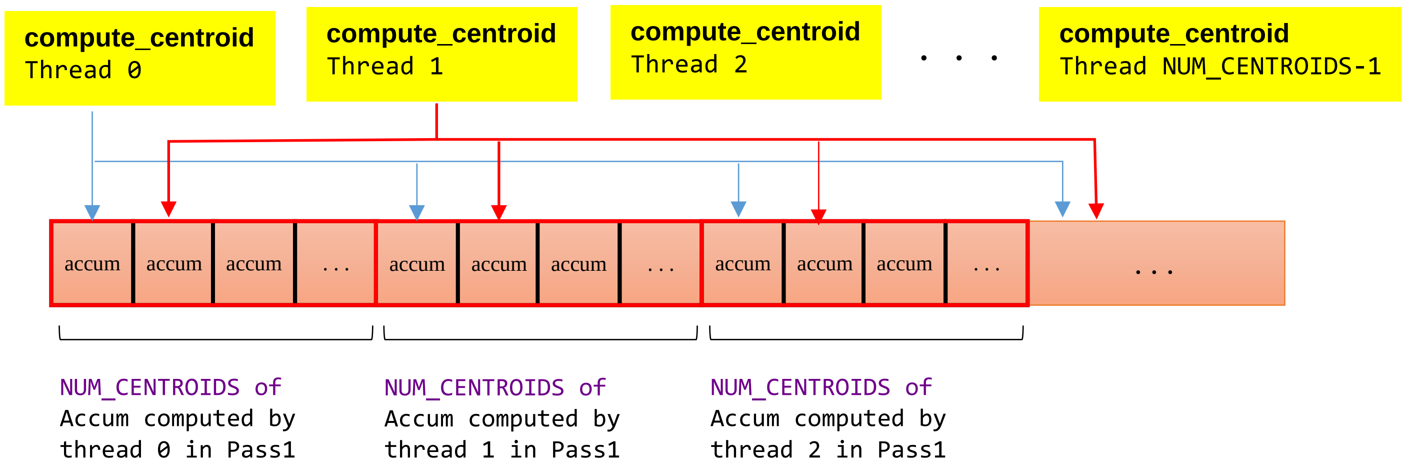
compute_centroid Thread 0
compute_centroid Thread 1
compute_centroid Thread 2
. . .
compute_centroid Thread NUM_CENTROIDS-1
accum
accum
accum
. . .
NUM_CENTROIDS of Accum computed by thread 0 in Pass1
NUM_CENTROIDS of Accum computed by thread 1 in Pass1
NUM_CENTROIDS of Accum computed by thread 2 in Pass1

Figure 4. computation of centroid

```
_GENX_MAIN_ void cmk_compute_centroid_position(
       SurfaceIndex cen_si,      // binding table index of centroids
       SurfaceIndex acc_si) {    // binding table index of accum for computing new centroids

       vector<unsigned int, 16> offsets(init16);
       const unsigned stride = NUM_CENTROIDS * ACCUM_REDUCTION_RATIO;
       offsets = offsets * DWORD_PER_ACCUM * stride;
```

stride distance is byte offset distance between the two consecutive Accum that belong to one centroid. One SIMD16 untyped read can read 16 Accum. Here we pre-compute stride distances of 16 consecutive Accum for one centroid.

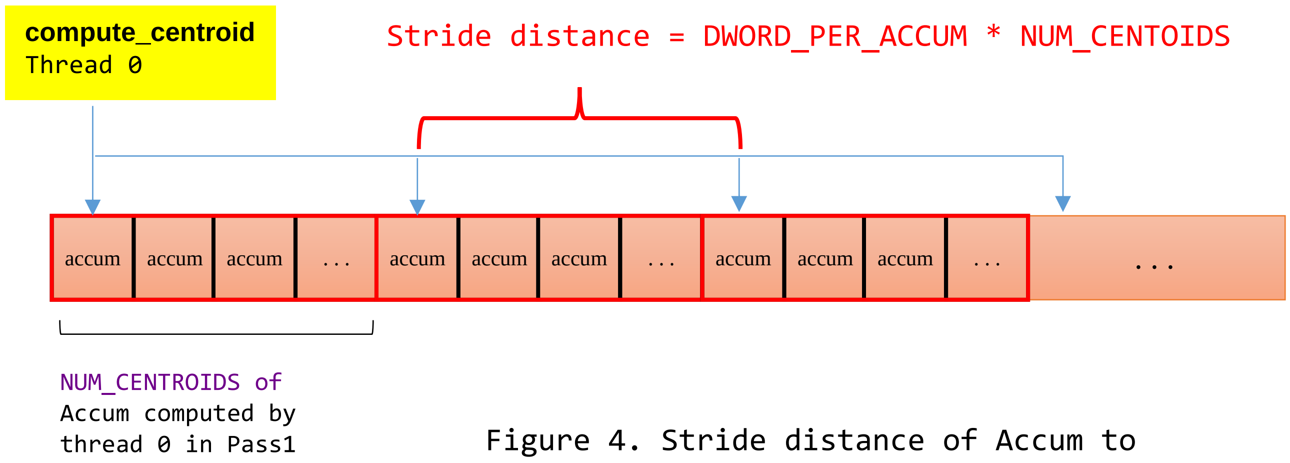
compute_centroid Thread 0
Stride distance = DWORD_PER_ACCUM * NUM_CENTOIDS
accum
accum
accum
. . .
NUM_CENTROIDS of Accum computed by thread 0 in Pass1

Figure 4. Stride distance of Accum to be read by each thread

```
      // each thread computes one single centriod
       uint linear_tid = cm_linear_group_id();

       vector<float, 16> X = 0;
       vector<float, 16> Y= 0;
       vector<unsigned, 16> N = 0;

       unsigned num_accum_record = (NUM_POINTS / (POINTS_PER_THREAD*ACCUM_REDUCTION_RATIO));
```

vector X, Y and N are used to sum up accum_x, accum_y and num_points for all Accum records of centroid linear_tid. num_accum_record is the total number of accum a centroid has.

```
       // process 4 reads per iterations to hide latency
#pragma unroll
       for (unsigned i = 0; i < (num_accum_record >> 6) << 6; i += 64)
       {
              // untyped read will pack R, G, B into SOA
              // matrix.row(0): x x x x . . . . // 16 position x
              // matrix.row(1): y y y y . . . . // 16 position y
              // matrix.row(1): n n n n . . . . // 16 position num of points
              matrix<float, 3, 16> accum0;
              read_untyped(acc_si, CM_BGR_ENABLE, accum0, offsets + i* DWORD_PER_ACCUM * stride +
linear_tid * DWORD_PER_ACCUM);
              matrix<float, 3, 16> accum1;
              read_untyped(acc_si, CM_BGR_ENABLE, accum1, offsets + (i+16) * DWORD_PER_ACCUM * stride +
linear_tid * DWORD_PER_ACCUM);
              matrix<float, 3, 16> accum2;
              read_untyped(acc_si, CM_BGR_ENABLE, accum2, offsets + (i+32) * DWORD_PER_ACCUM * stride +
linear_tid * DWORD_PER_ACCUM);
              matrix<float, 3, 16> accum3;
              read_untyped(acc_si, CM_BGR_ENABLE, accum3, offsets + (i+48) * DWORD_PER_ACCUM * stride +
linear_tid * DWORD_PER_ACCUM);
              X += accum0.row(0) + accum1.row(0) + accum2.row(0) + accum3.row(0);
              Y += accum0.row(1) + accum1.row(1) + accum2.row(1) + accum3.row(1);
              N += accum0.row(2).format<unsigned, 1, 16>() + accum1.row(2).format<unsigned, 1, 16>() +
                     accum2.row(2).format<unsigned, 1, 16>() + accum3.row(2).format<unsigned, 1, 16>();
       }
       // process remaining loop iterations
#pragma unroll
       for (unsigned i = (num_accum_record >> 6) << 6; i < num_accum_record; i += 16)
       {
              matrix<float, 3, 16> accum0;
              read_untyped(acc_si, CM_BGR_ENABLE, accum0, offsets + i * DWORD_PER_ACCUM * stride +
linear_tid * DWORD_PER_ACCUM);
              X += accum0.row(0);
              Y += accum0.row(1);
              N += accum0.row(2).format<unsigned, 1, 16>();
       }
```

The code above simply sums up all Accum records. Since num_accum_record can be large and the latency of a read is high, serializing all reads incurs serious long latency. The first loop do 4 parallel reads per iteration to hide read latency.

```
        vector<float, 8> centroid = 0;
        unsigned num = cm_sum<unsigned>(N);
        centroid(0) = cm_sum<float>(X)/num;
        centroid(1) = cm_sum<float>(Y)/num;
        centroid(2) = *((float*)&num);

        // update centroid(linear_tid)

        vector<ushort, 8> mask(initmask);
        vector<uint, 8> offs(init8);
        SIMD_IF_BEGIN(mask) {
                write(cen_si, linear_tid * DWORD_PER_CENTROID, offs, centroid);
        } SIMD_IF_END;

}
```

Each read is SIMD16. The summation is per lane. The final step performs reduction to sum up 16 lanes. The sum of X (Y) coordinate divided by total points of this cluster is the new X (Y) coordinate position.

If num_accum_record is too large, the latency of long sequence of reads may start to occupy a big portion of total execution time. When this scenario happens, we can perform a parallel reduction pass with reduction ratio ACCUM_REDUCTION_RATIO between Pass1 and Pass2. The explanation of parallel reduction is omitted from this Documentation. The parallel reduction code is listed in Appendix.

Performance:

Target system configuration:

- CPU: i7-6770HQ
- Memory: DDR4-2400 32GB
- GPU: SKL GT4e
- OS: Ubuntu 16.04.2

CM test configuration: NUM_POINTS = 786432, NUM_CENTROIDS = 20, NUM_ITERATIONS = 400

- Kernel execution time with POINTS_PER_THREAD = 128

	cmk_kmeans	cmk_accum_reduction	cmk_compute_centroid_position	All kernels
REDUCTION_RATIO = 1	0.360563	0.000000	0.098218	0.458781
REDUCTION_RATIO = 2	0.364309	0.056209	0.052639	0.473156
REDUCTION_RATIO = 4	0.360302	0.042735	0.030134	0.433171
REDUCTION_RATIO = 8	0.360690	0.039421	0.019669	0.419780
REDUCTION_RATIO = 16	0.360544	0.042464	0.014677	0.417685

- Kernel execution time with POINTS_PER_THREAD = 256

	cmk_kmeans	cmk_accum_reduction	cmk_compute_centroid_position	All kernels
REDUCTION_RATIO = 1	0.365182	0.000000	0.051595	0.416777
REDUCTION_RATIO = 2	0.364423	0.030370	0.030270	0.425062
REDUCTION_RATIO = 4	0.369720	0.025597	0.020433	0.415750
REDUCTION_RATIO = 8	0.370241	0.023717	0.015219	0.409177
REDUCTION_RATIO = 16	0.364325	0.023927	0.011871	0.400123

- Overall execution time (including kernel execution and host enqueue)

```
NUM_POINTS = 786432
NUM_CENTROIDS = 20
NUM_ITERATIONS = 400
POINTS_PER_THREAD = 256
ACCUM_REDUCTION_RATIO = 1

// Timing stats for 3 consecutive runs

Average wall-clock time: 0.739535 ms
Total wall-clock time: 295.814026 ms


Average wall-clock time: 0.722952 ms
Total wall-clock time: 289.181000 ms


Average wall-clock time: 0.717570 ms
Total wall-clock time: 287.027832 ms
```

Appendix: Accum reduction pass:

```
_GENX_MAIN_ void cmk_accum_reduction(SurfaceIndex acc_si) {

       // each thread computes one single centriod
       uint linear_tid = cm_linear_group_id();
       matrix<float, 3, ROUND_TO_16_NUM_CENTROIDS> accum;
       matrix<float, 3, ROUND_TO_16_NUM_CENTROIDS> sum = 0;
       matrix_ref<unsigned, 1, ROUND_TO_16_NUM_CENTROIDS> num = sum.row(2).format<unsigned, 1,
ROUND_TO_16_NUM_CENTROIDS>();
       vector<unsigned int, 16> offsets(init16);
       offsets = offsets * DWORD_PER_POINT;
       unsigned start = linear_tid * ACCUM_REDUCTION_RATIO*NUM_CENTROIDS*DWORD_PER_ACCUM;
#pragma unroll
       for (unsigned i = 0; i < ACCUM_REDUCTION_RATIO; i++)
       {
              unsigned next = start + i * NUM_CENTROIDS*DWORD_PER_ACCUM;
#pragma unroll
              for (unsigned j = 0; j < ROUND_TO_16_NUM_CENTROIDS; j += 16) // round up to next 16
              {
                     read_untyped(acc_si, CM_BGR_ENABLE, accum.select<3, 1, 16, 1>(0, j), next + offsets
+ j * DWORD_PER_ACCUM);
              }
              sum.row(0) += accum.row(0);
              sum.row(1) += accum.row(1);
              num += accum.row(2).format<unsigned, 1, ROUND_TO_16_NUM_CENTROIDS>();
       }


#pragma unroll
       for (unsigned i = 0; i < ROUND_TO_16_NUM_CENTROIDS; i += 16) // round up to next 16
       {
              matrix<float, 3, 16> a16 = sum.select<3, 1, 16, 1>(0, i);
              SIMD_IF_BEGIN(offsets + i * DWORD_PER_ACCUM < NUM_CENTROIDS * DWORD_PER_ACCUM) {
                     write_untyped(acc_si, CM_BGR_ENABLE, a16, start + offsets + i * DWORD_PER_ACCUM);
              } SIMD_IF_END;
       }
}
```