Abstract—The first step of the K-nearest neighbor classification is to find the K-nearest neighbors of the query. A basic operation in calculating Jaccard distance is to count the number of ones in a binary vector – population count. This article focuses on finding the K-nearest neighbors in a high-dimensional Jaccard space. There are two main contributions. First, software methods for population count are improved and implemented in Advanced Vector Extensions. They are faster than the hardware instruction. Second, a new method for KNN is proposed. Jaccard distance is a metric. After the K-nearest neighbors of a vector are found, their K-nearest neighbors must be in a hypersphere with a radius determined by the triangle inequality. Sequential computation and parallelization by OpenMP and POSIX Threads are implemented. The new method is up to six to 11 times faster than the classic method.

I. INTRODUCTION

The Hamming weight of a binary vector is its number of 1’s. It is also known as population count if the bits represent the presence or absence of the elements of a totally ordered set. When the vector is stored as bits in computer memory, calculating its Hamming weight is sideways addition. Let A and B be two sets. Their Jaccard index \[ J(A, B) = \frac{|A \cap B|}{|A \cup B|} \] is between 0 and 1 – if both A and B are empty, let \( J(A, B) = 1 \). The Jaccard distance between A and B is 1 minus their Jaccard index:

\[ \delta(A, B) = 1 - J(A, B). \]

Jaccard distance is a metric [13]. In some fields, Jaccard distance is known as Tanimoto distance [8].

Jaccard index is widely used in data analysis. In bioinformatics, nucleotides are coded in bits, and short pieces of DNA as binary vectors; here, Jaccard index measures the similarity of two DNA sequences [20]. In chemoinformatics, a compound is coded as a binary vector, where 1 means the presence of a molecular property; here, Jaccard index captures compound similarity [8]. Population count is also used in astronomical data analysis [15]. The vectors in these applications are hundreds to thousands of bits long.

The K-nearest neighbor method (KNN) is an algorithm for classification. K is a small constant, often between ten and twenty. To predict the class of a query object, the algorithm looks at its K-nearest neighbors whose classes are known, and assigns the query to the most populous class in this neighborhood. There are many variations of this general description. This work focuses on the task of finding the K-nearest neighbors in a high-dimensional Jaccard space.

Let S be a set of \( n \) binary vectors of length \( d \). For each vector in S, the goal is to find its K-nearest neighbors in S. A straightforward method uses \( O(dn^2) \) time to compute all pairwise Jaccard distances, and \( O(n^2) \) time to find the K-nearest neighbors. In some applications, the query objects are not elements in S. Then space partitioning schemes, such as k-d trees [4], may use sublinear time for KNN queries in a low-dimensional space by preprocessing S into appropriate data structures. The advantage of space partitioning, however, disappears when \( d \) is about ten or higher [22], because the time or space requirements of these schemes are exponential in \( d \). One solution to ever increasing dimensions is dimension reduction, such as locality-sensitive hashing [7]. This solution, however, leads to approximate nearest neighbor search.

Eschewing approximation, this work explores three directions to speed up exact KNN computation.

First, Fast Population Count. Many modern CPUs have an instruction called \texttt{POPCNT} that calculates the population count of a 32 or 64-bit word. Warren’s book [21] has the definitive account of population count by software. This work improves upon Warren’s plan for population count of a long vector, and implements it in Advanced Vector Extensions (AVX) – SIMD instructions that operate on registers of 256-bit width. The code is 14% faster than hardware \texttt{POPCNT} for the 4,096-dimensional space. Section II describes these and related results.

Second, Fast Selection. Many researchers use GPUs for computing KNN in the Euclidean space, and use matrix multiplication for calculating pairwise distances. Given the distances, their methods of finding K-nearest neighbors include, in chronological order, modified insertion sort [6], priority queue [2], probabilistic selection [16], truncated bitonic sort [19], quick select [11], and truncated merge sort [14]. For CPU computation, the methods that come to mind are priority queue and quick select. Results in Section III, however, show that they are slower than the brute force method – repeatedly finding the minimum in the remaining objects – when K is eight or smaller. Instead, a good pivot for partitioning, identified by truncated min-reduction, leads to the fastest selection.

Third, Triangle Inequality. Being close to a vector \( A \), the K-nearest neighbors of \( A \) are not far from each other. How far can their K-nearest neighbors be? Let us write \( k \) instead of K.
#define MASK_55 0x55555555
inline unsigned popcnt32(unsigned u){
  u = u - ((u >> 1) & MASK_55);
  // [0, 2] in 2-bit subwords
  u = (u & MASK_33) + ((u >> 2) & MASK_33);
  // [0, 4] in 4-bit subwords
  u = (u + (u >> 4)) & MASK_0F;
  // [0, 8] in 8-bit subwords
  u = u + (u >> 8);
  // [0, 16] in the lower 8 bits
  u = u + (u >> 16);
  // [0, 32] in the lowest 8 bits
  return u & 0x000000FF;
}

Listing 1. Fastest population count for vector length 32 using 15 instructions

in formulas. Let $B_1, \ldots, B_k$ be the K-nearest neighbors of $A$, sorted by their distances to $A$. The triangle inequality gives an upper bound on the distances among them: for $1 \leq i < j \leq k$,

$$
\delta(B_i, B_j) \leq \delta(A, B_i) + \delta(A, B_j) \\
\leq \delta(A, B_{k-1}) + \delta(A, B_k).
$$

(3)

If $B_x$ is far from $A$ such that $\delta(A, B_x) > \delta(A, B_{k-1}) + 2\delta(A, B_k)$, $B_x$ cannot be one of the K-nearest neighbors of $B_i$:

$$
\delta(B_x, B_i) \geq \delta(A, B_x) - \delta(A, B_i) \\
\geq \delta(A, B_x) - \delta(A, B_k) \\
> \delta(A, B_{k-1}) + \delta(A, B_k).
$$

(4)

The $B_i$’s already have at least K neighbors – themselves and $A$ – within this distance. This observation leads to a new method for KNN computation. After finding the K-nearest neighbors of $A$, the method finds the subset $S'$ of $S$ that contains vectors no farther than $\delta(A, B_{k-1}) + 2\delta(A, B_k)$ to $A$. If $|S'|$ is less than $|S|/2$, it proceeds to finding the K-nearest neighbors for $B_1, \ldots, B_k$ with respect to $S'$. Section IV reports computational experiments with this new method. Results show that it is faster than the classic KNN method. At its best, it is 11 times faster by sequential computation, and six times faster by parallel computation.

II. FAST POPULATION COUNT

This section studies software techniques for population count, and compares their performance to hardware POPCNT. Many people have studied population count by software. Warren’s book [21] is the best source of information. HAKMEM [3] and Anderson’s webpage [1] are also instructive. The Intel C compiler (ICC) exposes assembly instructions to C programs through intrinsics [9]. There are two POPCNT intrinsics, for 32 and 64 bits. In the computational experiments, the input data are $2^{28}$ random binary vectors for each of these lengths: 32, 64, 256, 1,024, and 4,096. The reported wall clock time is the average of ten runs.

All experiments are conducted on a Linux machine with the following specifications: Two Intel Xeon E5-2643 V3 hexa-core 20MB-cache 3.4 GHz CPUs; 256 GB DDR4 RAM connected to the CPUs through four channels of 64-bit width at 1,866 MHz for a peak transfer rate of 14.6 GB/sec per channel, and 58.3 GB/sec in total; 256 GB SATA III SSD boot drive; CentOS 6.7; ICC 16.0.2. A single thread running on one CPU core is used in experiments, unless otherwise stated. The compilation flags -03 -xHost -ipo lead to the fastest executables for the task at hand on this machine.

A. Vector Length 32

Using 15 instructions, Listing 1 is the fastest C code for vector length 32. Its computation fits the concept of SIMD within a register, a term that first appeared in [5] for an existing practice. Line 1 defines the bit mask MASK_55 in hexadecimal. MASK_33 and MASK_0F are similarly defined but not shown. The input $u$ starts as 32 individual bits. At Line 4, pairs of adjacent bits are summed and stored in 2-bit subwords; at Line 6, pairs of adjacent 2-bit subwords are summed and stored in 4-bit subwords; and so on. On input of $2^{28}$ binary vectors of length 32, Listing 1 takes 0.30 seconds, and POPCNT takes 0.37 seconds. The 15 instructions of Listing 1 cannot be faster than one instruction of POPCNT. Inspection of the compiled executable reveals that ICC has automatically vectorized Listing 1. Instead of working on 32-bit unsigned integers one at a time, the compiler generates AVX instructions that use 256-bit registers to process eight 32-bit integers simultaneously. Excluding the instructions for loop control, there are 17 AVX instructions (1 load, 15 compute, and 1 store) for every eight 32-bit integers, for an average of 2.125 AVX instructions per population count.

B. Vector Length 64

There are two software methods for length 64. First, Listing 1 is written in the 64-bit operations that are introduced in C99. Second, Lauradoux [12] has proposed a merging technique. Observe that, at Line 7 of Listing 1, a 4-bit subword can hold values up to 15. Thus at Line 8 in Listing 2, $v$ can be added to $u$ without overflowing the 4-bit subwords. Then $v$ may retire, and only $u$ needs to go through the rest of the calculation. There is one complication – in contrast to Line 8

Listing 2. Merging two population counts using 24 instructions
inline unsigned lookup256(_m256i u) {
  _m256i low, high;
  _m256i mask_0F = _mm256_set1_epi8(0x0F);
  _m256i table = _mm256_setr_epi8(
      0, 1, 1, 2, 1, 2, 3, 1, 2, 1, 2, 3, 1, 2, 3, 3,
      0, 1, 2, 1, 2, 3, 1, 2, 3, 3, 1, 2, 3, 3);
  low = _mm256_and_si256(mask_0F, u);
  high = _mm256_and_si256(mask_0F, _mm256_srli_epi8(u, 4));
  u = _mm256_add_epi8(_mm256_shuffle_epi8(table, low),
                     _mm256_shuffle_epi8(table, high));
  return _mm256_extract_epi64(u, 4) + _mm256_extract_epi64(u, 1) +
        _mm256_extract_epi64(u, 2) + _mm256_extract_epi64(u, 3);
}

Listing 3. Lookup table for vector length 256

Listing 4. C code using CSA for population count of eight words

#define FULLADDER(carryOut, sum, carryIn, w1, w2) {
  sum = sum ^ w2;
  carryIn = carryIn & w1;
  carryOut = sum & w2;
  sum = sum ^ w2;
  carryOut = carryOut | carryIn;
}

// Stage 1
FULLADDER(carry1, tmp1, word1, word2, word3);
FULLADDER(carry2, tmp2, word4, word5);
FULLADDER(carry3, sum1, tmp2, word6, word7);

// Stage 2
FULLADDER(sum3, sum2, carry3, carry2, carry1);
popcnt256 = popcnt32(sum1) + 2 * popcnt32(sum2) +
            4 * popcnt32(sum3) + popcnt32(word8);

Fourth, carry-save adder (CSA) can be used for population count. The idea is attributed to Harley and Seal (see [21]).
On input of two bits and a carry-in bit, a full adder calculates two bits: sum and carry-out. Figure 1 shows that a CSA adds seven 8-bit words into three words. The C code in Listing 4 implements this seven-word CSA, and, at Line 18, the eighth word is processed separately by the code in Listing 1. When each word is a 32-bit integer, Listing 1 performs population count of a 256-bit vector. The macro FULLADDER emulates a full adder with five instructions. As shown, the code uses 85 instructions – five instructions per full adder, and 15 instructions per popcnt32. The number of instructions can be reduced by merging the counting of sum1 and word8 as well as that of sum2 and sum3 (Lines 17 and 18). After merging is applied, the compacted code uses 75 instructions. In contrast, for the same task, the code in Figure 5-9 of [21] adds up all eight words using seven full adders, expending 101 instructions.

In general, to count a long vector of $2^m$ words, the best plan is to process one word separately, and sum up $2^m - 1$ words by CSA in $m - 1$ stages. Stage 1 uses $2^{m-1} - 1$ full adders, Stage 2 uses $2^{m-2} - 1$ full adders, and so on, and Stage $m - 1$ uses 1 full adder. This plan uses $2^m - m - 1$ full adders and $5 \cdot 2^m + 12m + 9$ instructions in total. Warren's book [21] gives an outline of this plan for $2^m - 1$ words, but does not consider using it for $2^m$ words by processing one word separately. The plan in the book uses $2^m - 1$ full adders and $5 \cdot 2^m + 17m + 10$ instructions in total. Regardless which of the improved plan or Warren's plan is implemented, patient programmers will trim off a few instructions by the merging technique at places such as Lines 17 and 18 in Listing 4.

On input of $2^{28}$ vectors of 256 bits, execution times are:

- 1.17 seconds, hardware POPCNT.
- 1.33 seconds, AVX translation of Listing 1.
- 1.08 seconds, three-way merging of 64-bit words plus processing one 64-bit word separately.
- 1.11 seconds, lookup table.
- 0.96 seconds, CSA adding seven 32-bit words plus pro-

Fig. 1. Carry-save adder with input of seven words and output of three words

of Listing 1, at Line 10 of Listing 2, the mask MASK_0F must be applied to both $u$ and $u >> 4$ before they are summed.

In experiments, the 64-bit version of Listing 1 takes 0.46 seconds, and merging takes 0.43 seconds. The hardware POPCNT is the fastest, 0.42 seconds. Sixty-four is the only length that hardware is faster than software.

### C. Vector Length 256

Four methods in ten variations are implemented and tested. First, Listing 1 is written in 256-bit AVX instructions. Second, two pairs of 64-bit vectors are merged, or three 64-bit vectors are merged and the fourth one is processed by merging two 32-bit vectors.

Third, lookup table is a popular method for population count. Listing 3 shows the implementation with AVX intrinsics. Its advantage is that the table resides in a 256-bit register (Lines 4, 5, and 6), and thus it is readily available. The register holds thirty-two 8-bit integers, and these integers are the population counts of 0, 1, ..., 15 stored twice in the low and high 128-bit subwords. The population counts of thirty-two 4-bit subwords are looked up by the shuffle instructions at Lines 11 and 12. It takes eight AVX instructions to do population count of a 256-bit vector.
TABLE I

BEST RUNTIME OF POPULATION COUNT

<table>
<thead>
<tr>
<th>Vector length</th>
<th>Fastest method</th>
<th>Time (sec)</th>
<th>Throughput (GB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Listing 1</td>
<td>0.30</td>
<td>1.67</td>
</tr>
<tr>
<td>64</td>
<td>hardware POPCNT</td>
<td>0.42</td>
<td>4.76</td>
</tr>
<tr>
<td>256</td>
<td>CSA (32-bit words)</td>
<td>0.96</td>
<td>8.33</td>
</tr>
<tr>
<td>1,024</td>
<td>lookup table</td>
<td>5.09</td>
<td>6.29</td>
</tr>
<tr>
<td>4,096</td>
<td>CSA (256-bit words)</td>
<td>26.0</td>
<td>4.92</td>
</tr>
</tbody>
</table>

cessing one 32-bit word separately.

D. Vector Length 1,024

On input of $2^{28}$ vectors of 1,024 bits, execution times are:

- 6.22 seconds, hardware POPCNT.
- 5.74 seconds, using AVX instructions to do three-way merging of 256-bit words plus processing one 256-bit word separately.
- 5.09 seconds, lookup table.
- 5.47 seconds, CSA adding fifteen 64-bit words plus processing one 64-bit word separately.
- 5.68 seconds, CSA adding seven 128-bit words plus processing one 128-bit word separately by SSE instructions – SIMD instructions that operate on registers of 128-bit width.

E. Vector Length 4,096

On input of $2^{28}$ vectors of 4,096 bits, execution times are:

- 30.1 seconds, hardware POPCNT.
- 26.8 seconds, lookup table.
- 26.0 seconds, CSA adding fifteen 256-bit words plus processing one 256-bit word separately by AVX instructions.

F. Remarks

The fastest methods for different lengths are summarized in Table I. For lengths 32 and 256, the fastest methods are standard C programs using 32-bit operations, but the Intel C compiler automatically generates 256-bit AVX assembly to parallelize them. For lengths 1,024 and 4,096, the fastest methods are C code with AVX intrinsics (such as Listing 3). Because there are $2^{28}$ vectors for each length, throughput can be derived from runtime.

Computation slows down beyond length 256. There is a five-fold increase in computing time for a quadruple increase in data, leading to decreased throughput. This is caused by the limited bandwidth of a single CPU core and non-uniform memory access. At the largest, the input data is 128 GB, occupying half of the available RAM, and this amount of data is delivered to one CPU core through different memory channels and interconnects. Figure 2 plots runtime, in seconds, using multiple OpenMP [17] threads for the 4,096-bit case.

All three bit-counting methods have similar runtime in the multithreading mode, and the minimum, 3.28 sec, is reached by 12 threads – additional threads gain no advantage. At 3.28 sec, throughput is 39 GB/sec, or 67% of the theoretical peak transfer rate of the machine. Among the hundreds of test runs, the fastest time is 2.55 sec, for a throughput of 86% of the theoretical peak. That population count is inherently memory-bound has been observed in [8], where Morton order is used to improve cache usage. From this perspective, any method of population count described in this section is good enough to saturate data transfer. To attain higher performance, however, a method must do fewer population counts for the same amount of KNN computation, using the observation based on the triangle inequality. Thus fast population count still makes a difference because the new KNN method transforms KNN computation from memory-bound to compute-bound.

III. FAST SELECTION

The task is to find the indices of the K smallest values in a long array. Four methods are considered. The first method works by brute force. It repeatedly finds the minimum in the array and moves it to the front – this is insertion sort that terminates after K iterations. The second method is priority queue. It incurs overhead at initialization and during restoration. The third method is quick select – there are different strategies for choosing the pivot, such as random pivoting, and the specific quick select tested in this work uses the median of the first, the middle, and the last elements in the array as the pivot for partitioning.

The fourth method finds a good pivot for partitioning. It works like truncated min-reduction. It repeatedly keeps the minimums of pairs of elements, and thus reduces the number of elements by half in each iteration. It stops the reduction when the next iteration will leave fewer than K elements. The pivot for partitioning is the maximum among the remaining elements. One partitioning is enough when K is small. The partition to the left of the pivot is simply sorted so that the K smallest distances and their indices are in the right places.

Implementation is straightforward. The test data is a 32,768 by 32,768 ($2^{15} \times 2^{15}$) matrix of random floating-point numbers.
between 0 and 1, and the task is to find the K smallest elements \( \delta \) in each row. Runtime, in seconds, of these four methods using one CPU thread is listed in Table II. Pivoting by truncated min-reduction is the best for the values of K considered in this work, although runtime is rising gradually. If K is larger than 128, a second partitioning will keep the runtime low. Let us call this method \textit{reduce-select} and use it in the next section.

### IV. Triangle Inequality

Almost all papers on KNN adhere to the separation of distance calculation and neighbor selection. This makes sense when the metric is Euclidean distance. People use matrix multiplication in highly optimized libraries for wholesale distance calculation.

Jaccard distance is different. There is currently no method for combining multiple distance calculations in a single set of operations. Rewrite Equation (1) into the following form:

\[
J(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \tag{5}
\]

This form is conducive to faster computation than Equation (1) – the method pre-calculates the population counts of the sets, and then those of their intersections.

Consider a vector \( A \). After calculating its distances to all the other vectors, a program should proceed to finding its K-nearest neighbors, because these distances may still be in the cache. \textit{Reduce-select} in Section III can be used for that. Let \( B_1, \ldots, B_k \) be the K-nearest neighbors sorted by their distances to \( A \). Based on the triangle inequality, Equations (3) and (4) near the end of Section I restrict the K-nearest neighbors of \( B_1, \ldots, B_k \) to a hypersphere centered at \( A \) with a radius of \( \delta(A, B_k-1) + 2\delta(A, B_k) \). Using this radius as the pivot, the new KNN method partitions the vectors by their distances to \( A \), and obtains those residing in the hypersphere. Here the new method faces a choice. If the number of vectors in the hypersphere is small, it will calculate the K-nearest neighbors of \( B_1, \ldots, B_k \) at little cost. If the number is large, it should pass up this hypersphere, given that there is overhead in this additional computation. The current implementation uses \( n/2 \) as the threshold – the method passes up this chance if the hypersphere contains more than half of the data set. Listing 5 is the pseudocode of the new method.

To test the performance of the new method, real data must be used. If uniform pseudorandom numbers are used, the expected Jaccard distance between two such vectors is 0.5, and the hyperspheres almost always include the whole data set. Chromosome 1 of the reference human genome is used to generate the data. Three nucleotides form a codon, and there are 64 codons. A 64-bit vector is generated as the fingerprint of a sequence of 32 nucleotides. For each of the 30 frames of three nucleotides in the sequence, the corresponding bit in the 64-bit vector is set to 1. The population counts of these vectors are in the range of 1 to 30. For the 4,096-dimensional space, a frame of six nucleotides is used. There are 2,025 arithmetic progressions of length six in the range of 1 to 140. Thus from a sequence of 140 nucleotides, the 2,025 “double codons” are extracted, and the corresponding bits in the 4,096-bit vector are set to 1.

Table III reports the computation time in seconds. “Classic KNN” calculates, for each vector, its Jaccard distances to the other vectors in the data set. After distance calculation, the method immediately uses reduce-select to find the K-nearest neighbors. Classic KNN readily submits to parallelization by OpenMP [17], which spawns 24 threads to run on 12 CPU cores. Parallelization gains 16 times or higher speedup.

To parallelize the new method, the race condition in accessing the Boolean array \texttt{processed} must be prevented. POSIX Threads [18] provides finer control than OpenMP in this regard. The data is divided into blocks of 64 vectors, and one mutex is used to coordinate the access to the 64 elements of \texttt{processed} of the block. Each of the 24 Pthreads works on one block at a time. Before a thread reads or writes \texttt{processed[i]} (Listing 5: Lines 6 and 12), it first locks the mutex of the block \texttt{mutex[i / 64]}, and it unlocks it afterwards. When a thread is in the loop of Lines 19 to 23
of Listing 5, however, it enters blocks under other threads' care, and locks and unlocks their mutexes \texttt{mutex[j / 64]}. This arrangement works well. Parallelization gains ten times or higher speedup, in spite of the overhead in using mutexes.

The new method exhibits an anomalous pattern in Table III. In both the top and bottom parts of the table, runtime is faster for \( K \) being 16 than being 8. The explanation is that the hyperspheres for \( K \) being 16 are not much more populated than for \( K \) being 8, and thus the former yields the 16-nearest neighbors of 16 vectors while the latter yields the 8-nearest neighbors of 8 vectors – twice the amount of progress for a comparable amount of work.

V. DISCUSSION

This work has two main contributions. First, in Section II, various methods for population count are studied. A more efficient method of using carry-save adder than what is in [21] is described. The fastest methods, implemented in AVX instructions, are faster than hardware \texttt{POPCNT} for vectors of 256, 1,024, and 4,096 bits.

The second main contribution is a new method for KNN computation. Both sequential and parallel versions are implemented and compared to the classic method. After it finds the K-nearest neighbors of a vector, it proceeds to finding their K-nearest neighbors in a hypersphere with a radius derived from the triangle inequality. Because the hypersphere has fewer vectors than the whole data set, the new method calculates fewer Jaccard distances and runs faster than the classic method.

The new method has an algorithmic parameter: the number of vectors in the hypersphere. The current implementation uses half of the size of the data as the cutoff. When the hypersphere contains more than half of the data, the new method ignores this hypersphere and moves on, with the hope that more compact clusters of vectors will be discovered later on. This is merely a heuristic. A good choice for this parameter may only be determined by examining the data and doing test runs. Another possible improvement is to use a priority queue of up to \( K \) elements to store the \( K \)-nearest neighbors while computing the distances.

There is a sweet spot for the new method in the bottom part of Table III. For a large data set (\( n \) being 131,072 or \( 2^{17} \)) of long vectors (4,096 bits), when \( K \) is 16, the new method is 14 times faster than the classic method by sequential computation, and six times faster by parallel computation. Runtime for the new method increases rapidly when \( K \) is 32 or larger, because the hyperspheres encompass large portions of the data set. From this perspective, runtime of the new method corresponds to population sparsity in the neighborhood – when the chosen \( K \) is too large for the neighborhood, the \( K \)-th neighbor is so far afield that the new method faces a large hypersphere containing many vectors. A future direction is to use the new method to choose different values of \( K \) dynamically by the population density or the lack thereof in the neighborhood.

Jaccard spaces of up to 4,096-dimensions are considered in this work. Intel Xeon Phi coprocessors have many cores, and each core has registers of 512 bits, which are helpful for computation in high-dimensional Jaccard spaces.

The new method relies heavily on branched execution. Thus it is difficult to parallelize it in its entirety for GPU computation. It is conceivable, however, that heterogeneous computation is useful for very large data sets. The CPU calculates K-nearest neighbors for one vector, and finds the vectors in the hypersphere. Then this subspace of data is sent to the GPU for branchless computation. This approach can be construed as online space partitioning.

The new method for KNN can be adapted to any metric space.

REFERENCES