In Proceedings of the 24th IEEE International Conference on Data Engineering, pp. 1111–1120, Cancun, Mexico, April 2008

A Fast Similarity Join Algorithm Using Graphics Processing Units

Michael D. Lieberman Jagan Sankaranarayanan Hanan Samet

Department of Computer Science
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742 USA
{codepoet,jagan,hjs}@cs.umd.edu

***Abstract*— A similarity join operation $A \bowtie_\epsilon B$ takes two sets of points A, B and a value $\epsilon \in \mathbb{R}$, and outputs pairs of points $p \in A, q \in B$, such that the distance $D(p,q) \leq \epsilon$. Similarity joins find use in a variety of fields, such as clustering, text mining, and multimedia databases. A novel similarity join algorithm called LSS is presented that executes on a Graphics Processing Unit (GPU), exploiting its parallelism and high data throughput. As GPUs only allow simple data operations such as the sorting and searching of arrays, LSS uses these two operations to cast a similarity join operation as a GPU sort-and-search problem. It first creates, on the fly, a set of space-filling curves on one of its input datasets, using a parallel GPU sort routine. Next, LSS processes each point p of the other dataset in parallel. For each p, it searches an interval of one of the space-filling curves guaranteed to contain all the pairs in which p participates.**

Using extensive theoretical and experimental analysis, LSS is shown to offer a good balance between time and work efficiency. Experimental results demonstrate that LSS is suitable for similarity joins in large high-dimensional datasets, and that it performs well when compared against two existing prominent similarity join methods.

I. Introduction

A similarity join $A \bowtie_\epsilon B$ takes two sets of objects A, B and a value $\epsilon \in \mathbb{R}$, and outputs pairs of points $(p,q), p \in A, q \in B$, such that $D(p,q) \leq \epsilon$, where $D(\cdot,\cdot)$ is an arbitrary distance measure. The similarity join has important applications in knowledge discovery [1], including clustering, text mining, image and multimedia databases, and geographic information systems (GIS). In these domains, a similarity join is performed by transforming the set of objects to be searched into a high-dimensional feature vector representation via the application of a suitable technique, such as feature vector extraction [2], K-L transformation [3], or embedding [4]. The similarity join is then performed on the vector representation of the objects; if two objects are within distance ϵ of each other, they are considered *similar* in the original domain. Unfortunately, the vector representation is usually high-dimensional, so similarity join processing suffers from the *curse of dimensionality* [5]. In this paper we present a novel algorithm, named LSS, that performs fast similarity join operations on high-dimensional datasets using a Graphics Processing Unit (GPU). LSS takes advantage of a modern GPU's ability to perform simple arithmetic operations at a high throughput. As GPUs can only perform simple data operations, we have cast the similarity join operation as a *sort-and-search* problem. That is, our algorithm only requires simple sort and search routines, which have been implemented on the GPU. LSS creates several space-filling curves, built over one of its input datasets. It then performs interval searches for each point in the other dataset. By generating multiple space-filling curves, we are able to reduce the size of the interval searches, thereby producing significant speedups. We have applied LSS to large high-dimensional datasets, and have shown at least an order of magnitude improvement when compared against two prominent techniques in the literature.

This work was supported in part by the U.S. National Science Foundation under Grants EIA-00-91474, CCF-05-15241, and IIS-0713501, as well as NVIDIA Corporation and Microsoft Research.

In this paper, we focus on a variant of the similarity join problem, where $A, B \in \mathbb{R}^d$ are both point datasets drawn from a high-dimensional vector space of dimension $d \in \mathbb{N}$, and $D(\cdot,\cdot)$ is a *Minkowski*, or *normed* metric. We also assume that both A and B are initially unsorted point sets. If both A and B refer to the same dataset, the problem is known as a *self-join* operation. An alternate formulation of the similarity searching problem [6], [7] requires the computation of the approximate or exact k-nearest neighbors in B to each point in A. LSS can be easily modified to accomodate this and other formulations of the similarity join problem.

The rest of the paper is organized as follows. In Section II, we present an overview of related similarity join techniques. We also briefly describe several other database operations that have been implemented on a GPU. Next, in Section III, we provide a brief overview of a programming paradigm that is suitable for GPUs. In Section IV, we describe a quadtree-based data structure for the GPU, which is used in Section V to describe our similarity search algorithm and its variations. Experimental results are discussed in Section VI. Finally, in Section VII, we offer concluding remarks and possible avenues for future research.

II. Related Work

Similarity join operations can be classified based on the availability of a multidimensional indexing structure on both

A and B [8] or on neither [9], [10]. If a multidimensional index is not present, it is usually constructed on the fly during the algorithm's execution, although if the indexing structure is dependent on ϵ, as in [9], [10], [11], [12], it may not be possible to reuse the index for future similarity join operations on the dataset. A variety of data structures have been used for similarity join processing, including quadtrees [10], [13], R-trees [8], hash functions [6], space-filling curves [10], and ϵ-kbd trees [12].

Two recent methods are based on the application of grids to multidimensional point datasets. The *Epsilon Grid Order (EGO)* [9] method works by first partitioning the initially unsorted data into an ϵ-sized grid G. The join algorithm then processes each cell c in the grid, and examines every cell in G within a distance ϵ from c. The EGO join method's novelty is a heuristic, termed *crab stepping*, that minimizes the number of times a neighboring cell is read by the algorithm. The *Generic External Space Sweep (GESS)* method of Dittrich and Seeger [10] is an adaptation of the *Multidimensional Spatial Join (MSJ)* algorithm of Koudas and Sevcik [13]. A hypercube with side length ϵ is constructed around each multidimensional input point. The similarity join problem is thus transformed to a *spatial intersection* [13] problem.

LSS maps one of its input datasets to a set of space-filling curves [14], [15], [16]. Bern [15] used a set of 2^d space-filling curves to compute $O(\sqrt{d})$ approximate nearest neighbors to a query point. Chan [16] improved upon Bern's original formulation by reducing the number of space-filling curves to $O(d)$, to obtain $O(d^{\frac{3}{2}})$ approximate nearest neighbors. Liao *et al.* [7] applied this method to perform similarity searching in high-dimensional datasets, though their formulation of similarity is based on finding an approximate nearest neighbor to a query point. In this paper, we adapt the set of space-filling curves to be created on the fly on a GPU from an input dataset. This data structure reduces the similarity join to $O(d)$ parallel sorts on one of the input datasets, and $O(\text{dataset size})$ binary searches and interval searches.

We use a GPU-based *bitonic sort* [17] algorithm, similar to that of the GPUSort toolkit of Govindaraju *et al.* [18]. Govindaraju *et al.* [19], [18] show that a GPU can speed up sorting and other database predicate operations. In addition, Sun *et al.* [20] propose techniques for spatial selection and join operations aided by the GPU. Their method relies on GPU algorithms to compute fast 2D and 3D polygon intersections. Unfortunately, their work is limited to low-dimensional datasets. Bandi *et al.* [21] adapt the work of Sun *et al.* to a commercial database system – in their case, *Oracle Spatial*.

III. The GPU as a Parallel Machine

A GPU can be viewed as a parallel SIMD machine with a limited instruction set. Modern GPUs allow for user-programmable *pixel* and *vertex* shaders, thereby transforming the GPU into a general-purpose parallel computing device of limited functionality. Early programmable shaders featured an assembly-like language for manipulating data as it moved through the graphics pipeline [22]. These languages had a small instruction set, which consisted mostly of register manipulation operators. Later, higher-level languages [23] such as Cg and GLSL made shader programming more accessible to programmers, as they required a less intimate knowledge of the workings of the graphics hardware. With the introduction of NVIDIA's CUDA [24], the implementation-specific details of shader programming are almost fully hidden behind a high-level abstraction. In this paradigm, GPU programs, called *kernels*, are written in an extended form of C or C++, and are executed in parallel by many *threads* running on the GPU.

Even though CUDA provides a good measure of abstraction to programmers, we point out that serial and parallel programming paradigms are different in their focus, which makes efficient serial algorithms unattractive candidates for parallel machines. In a parallel paradigm, the fastest or most *time-efficient* algorithm is not necessarily the most *work-efficient*. For example, suppose we are given datasets A, B, each containing n points, and a distance ϵ, and we wish to compute $A \bowtie_\epsilon B$. In a parallel machine with $O(n^2)$ processors, the join can be executed in $O(1)$ time by naively examining each of the n^2 pairs. This naive algorithm is time-efficient, even though it is not work-efficient. Our simple example illustrates that in a parallel machine with enough processors, time and work efficiencies are *decoupled*. Serial algorithms almost exclusively focus on making algorithms work-efficient, as in a serial algorithm, work efficiency equates to time efficiency. This makes them unsuitable for execution on parallel machines like GPUs.

As GPUs have much fewer processors (than $O(n^2)$), we see an interesting tradeoff between time and work efficiencies. Our proposed algorithm creates a set of space-filling curves on one of its input point sets, which makes our algorithm more work-efficient. However, we did not use a multidimensional indexing structure [4] like an R-tree or quadtree, which would make our algorithm even more work-efficient at the expense of time efficiency. Our algorithm strikes a balance between time and work efficiencies and is well-suited for the GPU.

IV. The Set of Z-Lists Data Structure

We now describe the Set of Z-Lists (SZL) data structure and mention some of its interesting properties, which makes it suitable for similarity join operations on the GPU. Let $S : \mathbb{R}^d$ be a set of n points in a d dimensional space. Without loss of generality, we assume that S is contained in a d-dimensional hypercube of unit side length. We define $Z(p) : \mathbb{R}^d \rightarrow \mathbb{N}$ as a mapping of a point p to its location on a *Z-order*, or *Morton order* [25], [14], [4], space-filling curve on S. The mapping $Z(p) \in \mathbb{N}$ is a *bit-interleaved* representation of the d coordinate values of p. Furthermore, given two points p and q, we define a *Z-ordering relation*, $p \preceq_Z q$, based on the relative positions of p and q on the Z-order space-filling curve: $p \preceq_Z q \iff Z(p) \leq Z(q)$. The following lemma states a property of $\preceq_Z$, which will be used later to show our algorithm's correctness.

Lemma 4.1: Let $p, q \in \mathbb{N}^d$ be points such that $p \neq q$ and q dominates p. That is, $\forall_{i=1}^d p_i \leq q_i$, where p_i, q_i are the ith dimensions of p, q, respectively. We have that $p \preceq_Z q$.

Proof: Without loss of generality, we will prove the above lemma for $d = 2$. An integer n can be represented as $n = \Sigma_i d_i 2^i$, where d_i is the ith bit in the binary notation of n. We define an *integer dilation* function E, such that $E(n) = \Sigma_i d_i 4^i$. That is, $E(n)$ spreads the bits of n apart with zeros. For example: $E(\texttt{111}) = \texttt{10101}$, where `111` and `10101` are binary numbers.

Let $q = (x, y) \in \mathbb{N}^2$. Note that $Z(q)$ is the mapping of q to a Z-order space filling curve defined by $E(x) + 2 \cdot E(y)$. For every point $p = (x', y')$, where $x' < x$ and $y' < y$, we can see that $Z(p) < Z(q)$ because $E(x') + 2 \cdot E(y') < E(x) + 2 \cdot E(y)$. Note that the above condition still holds for the case when $x' = x$ (or $y' = y$), because $E(y') < E(y)$ (or $E(x') < E(x)$). ■

We assume that the Z-ordering relation between p and q can be determined in $O(d)$ time. Finally, we define $Z(S) : S \subset \mathbb{R}^d \rightarrow \mathbb{N}$ as a *total ordering* of the points in S, based on the Z-ordering relation. We refer to $Z(S)$ as a *Z-list* Z_S of S, where $Z_S[i]$ is the ith point in Z_S.

A set of objects can be sorted on the GPU in $O(n \log^2 n)$ time using a parallel sorting network [19]. We use such a network, the *bitonic sort* [17], to produce Z-lists on the GPU. Our version of the bitonic sort algorithm orders points according to the Z-ordering relation, and produces a Z_S from S in $O(d^2 n \log^2 n)$ time. The resulting representation Z_S is similar to a *linear quadtree* [26], although in contrast to linear quadtrees, Z_S's search hierarchy is implicit.

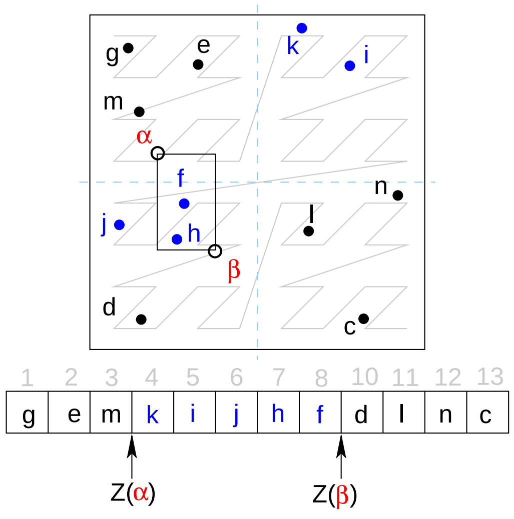

Figure 1. A set of points c–n, sorted based on their positions on a Z-order space-filling curve

It is possible to use Z_S to perform region searches in $\mathbb{R}^d$ on S. Let $p^{T(\alpha)}$ refer to the translation of a point p by α units across all dimensions; $T(\alpha)$ is referred to as a *translation vector*. $p^{T(-\epsilon)}$ and $p^{T(\epsilon)}$ are two *antipodal* points of a d-dimensional hypercube R centered at p, where $p^{T(-\epsilon)}$ is the closest point in R to the origin. The following lemma shows that the points in S contained in R can be obtained by examining only those points contained in a particular *search interval* of Z_S. That is, searching region R on Z_S reduces to examining all points in Z_S spanned by the search interval $[Z(p^{T(-\epsilon)}), Z(p^{T(\epsilon)})]$ for containment in R.

Lemma 4.2: Let S be a set of points $\in \mathbb{R}^d$, and let Z_S be a Z-list on S. Let p and q refer to the antipodal points of a hypercube $R \in \mathbb{R}^d$, such that p is the closest point in R to the origin. The interval spanned by $[Z(p)$,$Z(q)]$ in Z_S encompasses all the points from S contained in R.

Proof: From Lemma 4.1. ■

Note that the search interval may contain many points from S that are not contained in R. For example, Figure 1 shows a region R, with antipodal points α and β, that crosses the bounding box's *medial axis*. The search interval spanned by $Z(\alpha)$ and $Z(\beta)$ encompasses points k, i, j, f, and h, even though only points f and h are contained in R.

As every point in the search interval will be examined, a configuration like that in Figure 1 will cause the search interval to be very large, resulting in too much wasted work. We therefore require a method of reducing the span of the search intervals for arbitrary region searches on Z_S, without sacrificing time efficiency. Tropf and Herzog [27] show that the search interval spanned by $[Z(p^{T(-\epsilon)}), Z(p^{T(\epsilon)})]$ can be broken into a set R^* of *sub-intervals*, such that R^* only encompasses those points that are contained in R. Although this technique leads to a more work-efficient algorithm, it requires a binary search tree on Z_S and the repeated computation of two tree operations, termed LITMAX and BIGMIN. It is therefore not suitable for the GPU.

Instead, we use a quadtree-like data structure called the *Set of Z-Lists (SZL)* [15], [16], [7]. To describe the utility of the SZL data structure, we must first define *r-regions*, the regions corresponding to blocks produced by a quadtree or any other *regular space decomposition* [4].

Definition 4.3 ([7], [28]): An r-region is an open-ended hypercube in R^d with side length $r = 2^{1-m}$, with sides $[a_1 r, (a_1+1)r) \times \ldots \times [a_d r, (a_d+1)r)$, where $a_1 \ldots a_d, m \in \mathbb{N}$.

As seen in Figure 1, an interval $[Z(\alpha), Z(\beta)]$ may encompass many points that are not contained in R. However, if R is an r-region, that interval will contain only (and all) those points that are contained in R. The key idea behind the SZL data structure is to produce multiple Z-lists, each slightly shifted, so that in at least one of the Z-lists, the smallest r-region containing R is not much larger than R.

We now describe the Set of Z-Lists data structure. Let $v(j) : \mathbb{N} \rightarrow \mathbb{R}^d$ be a function that produces a translation vector $T(\frac{j}{d+1})$ for $0 \leq j \leq d$, and recall that $S^{T(\alpha)}$ refers to the translation of all points in S by $T(\alpha)$. The Set of Z-Lists is a collection of $d+1$ Z-lists on S, such that the points in list j have been translated by $v(j)$ and then sorted. The SZL data structure has the following important property.

Lemma 4.4 ([16]): Let $R_\epsilon(p)$ refer to a d-dimensional hypercube with side length $2 \cdot \epsilon$, centered at p, and suppose d is even (for odd d, replace d by $d+1$). Given $p \in S$ and ϵ, such that $0 < \epsilon < \frac{1}{2d+2}$, there exists a j, $0 \leq j \leq d$, such that $R_\epsilon(p^{v(j)})$ is contained in an r-region satisfying $\frac{r}{(4d+4)} \leq \epsilon < \frac{r}{(2d+2)}$.

That is, for reasonable values of ϵ, there exists a Z-list in the SZL of S such that the side length of the smallest r-region R^+ containing $R_\epsilon(p)$ is no more than $O(d)$ times bigger than ϵ. Note that R^+ in turn bounds the search interval spanned by $R_\epsilon(p)$, as the search interval of $R_\epsilon(p)$ can be no larger than the search interval of R^+. Hence, we are assured that the search interval of $R_\epsilon(p)$ is likewise of a reasonable size, so examining the search interval spanned by $R_\epsilon(p)$ for points closer than ϵ will not entail too much wasted work. This leads to an efficient similarity join algorithm.

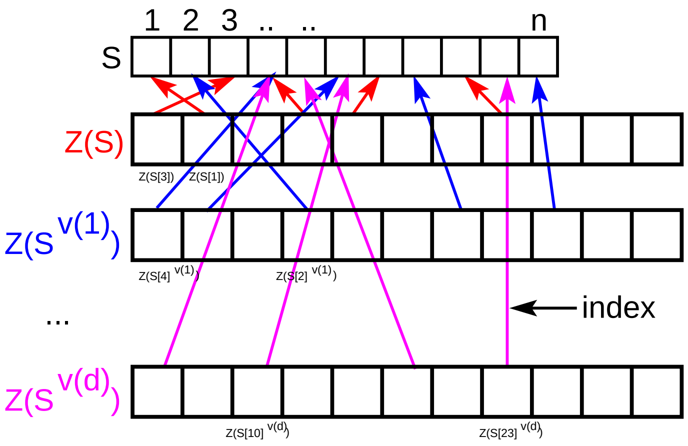

Figure 2. Organization of SZL on S, on the GPU

Figure 2 shows the SZL's organization as it is stored on the GPU. All the data structures are stored as simple arrays. The SZL on S is stored as a set of arrays, such that $Z(S^{v(j)})$ is the j^{th} Z-list in the SZL. Each Z-list in the SZL stores the index position of its corresponding point in S.

V. LSS: A GPU-based Similarity Join

The pseudocode of our similarity join algorithm is listed as Algorithm 1. The inputs to our algorithm are the point sets A and B, and the value ϵ. It returns a list of pairs, (p, q), where $p \in A, q \in B, D(p, q) \leq \epsilon$. $D(p, q)$ refers to any normed distance measure, such as the L_2 (Euclidean) or L_∞ (Chessboard) metric.

Our algorithm begins by building the set of $(d + 1)$ Z-lists on A using procedure ParallelSort, the parallel bitonic sort described in Section IV (lines 1 – 3). Note that in contrast to [10], [9], the SZL on A does not depend on ϵ. Once created, it can used with any other dataset B and any value of ϵ. After the SZL on A has been constructed, LSS then performs a region search for each point of B, in parallel (lines 4 – 11). For each point $p \in B$, we first find the offset positions of $Z(p^{v(j)-T(\epsilon)})$ and $Z(p^{v(j)+T(\epsilon)})$ in each of the $d+1$ curves, as shown in lines 6– 8. These positions are obtained using the GetInterval procedure, which finds the search interval $[m^j_\alpha, m^j_\beta]$ by invoking two binary searches on $Z_S[j]$, using $Z(p^{v(j)-T(\epsilon)})$ and $Z(p^{v(j)+T(\epsilon)})$ as search keys. After all $d+1$ search intervals of p have been computed, the algorithm uses procedure MinInterval to find the interval $[m_\alpha, m_\beta]$ that spans the fewest number of points, called the *minimum search interval* of p (line 9). The minimum search interval satisfies the following property: $\forall^d_{j=0}(m_\beta - m_\alpha) \leq (m^j_\beta - m^j_\alpha)$. MinInterval also returns the *curve identifier* of the Z-list that provided the minimum search interval of p. The minimum search interval's starting and ending offsets (m_α and m_β), as well as its curve identifier, are stored in the M_α, M_β, and R arrays respectively, in the offset position i corresponding to p.

The algorithm then invokes procedure WalkIntervals, which computes, for each point p in B, the distances to the points spanned by the minimum search interval of p (line 11). WalkIntervals is listed as Algorithm 2. The minimum search intervals of all points $p \in B$ are examined in parallel (lines 2 – 11). For each p, the procedure iterates over each point $q \in A$ between the minimum search interval starting and ending offsets $M_\alpha[i]$ and $M_\beta[i]$, in curve $R[i]$ (lines 5 – 10). If $D(p, q) \leq \epsilon$, the pair (p, q) is added to the result list L. Finally, once the minimum search intervals of all points in B have been examined, the result L is reported (line 12).

Our algorithm's correctness is ensured by Lemma 4.2. Furthermore, as discussed in Section IV, Lemma 4.4 provides a bound on the size of the minimum search interval of p, which in turn bounds the work done by the algorithm. The algorithm's time complexity is the sum of the time taken to build an SZL on A, obtain the minimum search interval for each point in B, and compute the actual distances to the points contained in the minimum search intervals. For the sake of simplicity, we will assume that A and B are both drawn from a uniform distribution. Furthermore, without loss of generality, we will assume that the GPU can execute k threads in parallel. Let $m = |A|$ and $n = |B|$, and let $f(d, \epsilon)$ denote the *average* minimum search interval for each point in B, where ϵ is small (given by Lemma 4.4). The total time complexity is

$$O(d^2 \frac{m}{k} \log^2 m) + O(d \frac{n}{k} \log m) + O(\frac{n}{k} f(d, \epsilon)), \quad (1)$$

which arises from $O(d)$ parallel bitonic sorts of A, $O(d)$ binary searches for each point in B, and $O(f(d, \epsilon))$ distance computations for each point of B.

Algorithm 1
Procedure SimilarityJoin[A, B, ϵ]
Input: $A, B \in [0, 1]^d$, $\epsilon \in [0, 1]$
Output: A list of pairs (p, q),
where $p \in A$, $q \in B$, $D(p, q) \leq \epsilon$

```
1.  for $j \in 0 \ldots d$ do
2.      $Z_A[j] \leftarrow$ ParallelSort($A^{v(j)}$)
3.  end-for
4.  for $i \in 1 \ldots |B|$ pardo
5.      $p \leftarrow B[i]$
6.      for $j \in 0 \ldots d$ do
7.          $[m^j_\alpha, m^j_\beta] \leftarrow$
                GetInterval($Z_A[j]$, $p^{v(j)-T(\epsilon)}$, $p^{v(j)+T(\epsilon)}$)
8.      end-for
9.      $(M_\alpha[i], M_\beta[i], R[i]) \leftarrow$
            MinInterval($[m^0_\alpha, m^0_\beta]$, ..., $[m^d_\alpha, m^d_\beta]$)
10. end-for
```

```
11.  WalkIntervals($Z_A$, $M_\alpha$, $M_\beta$, $R$, $|B|$)
12.  return
```

Algorithm 2

```
Procedure WalkIntervals[$Z_A$, $M_\alpha$, $M_\beta$, $R$, $n$]
Input: $Z_A$ – SZL on $A$
Input: $M_\alpha, M_\beta$ – minimum search interval offset arrays
Input: $R$ – curve identifier array
Input: $n$ – number of points to process
1.   $L \leftarrow \{\}$ ($*$ GPU-based buffer $*$)
2.   for $i \in 1 \ldots n$ pardo
3.       $p \leftarrow B[i]$
4.       $r \leftarrow R[i]$
5.       for $j \in M_\alpha[i] \ldots M_\beta[i]$ do
6.           $q \leftarrow A[Z_A[r][j]$.index]
7.           if $D(p,q) \leq \epsilon$ then
8.               Add $(p,q)$ to $L$
9.           end-if
10.      end-for
11.  end-for
12.  Report $L$
13.  return
```

A. Output Size Constraints

In general, GPU programming architectures do not allow asynchronous data transfer between the CPU and GPU; data transfer is blocking and costly. Therefore, the similarity join algorithm must store its results in a GPU-based buffer and report them in *chunks*. These chunks must be reasonably large, in order to offset the overhead involved in the data transfer, but small enough to fit in the GPU's memory. To determine appropriate sizes for these chunks, several key problems must be addressed. First, given an instance of a similarity join problem $A \bowtie_\epsilon B$, we cannot easily estimate the number of pairs in the result, so we do not know if the entire result will fit in the GPU's memory. Second, each point of B may have different numbers of points from A in its result, so the output buffer must accommodate the varying result size requirements of each point. Finally, we need to ensure that no two GPU threads write their results to the same location in the output buffer, which would lead to race conditions.

We address the aforementioned problems by assigning a large enough output buffer space to each point $p \in B$, such that it cannot *overflow*. Specifically, p is assigned a buffer with size equal to that of its minimum search interval, as the total number of output pairs in which p participates cannot exceed that size. Also, we avoid race conditions by assigning each thread an non-overlapping output buffer space, which is done using an *exclusive prefix-sum* operation [29], [30].

To implement chunked output, we replace line 11 of Algorithm 1 with an invocation of Procedure WalkIntervalsChunked, which is listed as Algorithm 3. We set the constant BufferSize to the largest number of output pairs that can be stored in the GPU's memory. BufferSize is calculated on the fly by accounting for the total physical memory available on the GPU, the size of the Z-list on A, and the size of B. Then, for each $p \in B$, the algorithm estimates the size of p's output buffer as equal to the size of its minimum search interval, which is then stored in the PS array (lines 1 – 3). Next, an in-place parallel exclusive prefix-sum procedure [29] computes for each $j \in 1 \ldots n$, $PS[j] = ((\sum_{i=1}^{j} PS[i]) - PS[j])$ (line 4). $PS[j]$ now contains point $B[j]$'s starting offset in the output buffer.

The algorithm now processes a chunk of the output, such that the size of the chunk does not exceed BufferSize (lines 6 – 11). In the while loop, i refers to the index of the first point in B whose output is to be computed. The algorithm now obtains the offset j of the last point in B whose output will fit in the chunk by performing a binary search on PS with the search key $k = i$ + BufferSize (lines 7 – 8). Procedure BinarySearch returns the offset position of the first point whose output buffer offset is greater or equal to k. At this point, the offset interval $[i, j-1]$ refers to the range of offsets in B that would be processed in this iteration of the while loop. Line 9 invokes Procedure WalkIntervals to report all results in the chunk, except that it now uses the PS array to determine the proper offset where the result of a point would be written. Note that the result buffer may contain a number of empty positions that do not contain an output pair. To remove these empty positions, we can apply a parallel *compaction* algorithm, described in [30].

Note that Algorithm 2 assigns a thread to each point in B. An alternate approach more suited for GPUs would be to assign a thread to each of pair in the result set. Using this approach, we now briefly describe the steps performed by the kth thread executing on the GPU. It first performs a binary search on the PS array, with k as the search key, in order to obtain the identity of a pair (p,q), where $p \in A, q \in B$. If $D(p,q) \leq \epsilon$, the thread stores the pair in the kth index position of the result buffer, and terminates.

Algorithm 3

```
Procedure WalkIntervalsChunked[$Z_A$, $M_\alpha$, $M_\beta$, $R$, $n$]
Input: $Z_A$ – SZL on $A$
Input: $M_\alpha, M_\beta$ – minimum search interval offset arrays
Input: $R$ – curve identifier array
Input: $n$ – number of points to process
1.   for $i \in 1 \ldots n$ pardo
2.       $PS[i] \leftarrow M_\beta[i] - M_\alpha[i]$
3.   end-for
4.   $PS \leftarrow$ ParallelPrefixSum($PS$)
5.   $i \leftarrow 0$
6.   while $i < n$ do  ($*$ process chunk $*$)
7.       $k \leftarrow PS[i] +$ BufferSize
8.       $j \leftarrow$ BinarySearch($PS$, $k$)

9.       WalkIntervals($Z_A$, $M_\alpha[i \ldots j-1]$,
             $M_\beta[i \ldots j-1]$, $R[i \ldots j-1]$, $j - i$)
10.      $i \leftarrow j$
11.  end-while
12.  return
```

B. Input Size Constraints

Our algorithm assumes that both A and B can fit in the GPU's memory. If this is not the case, we can apply a *data partitioning* technique on both A and B. We split $A = A_1 \cup A_2 \cup \ldots \cup A_j$ into j arbitrary chunks, and $B = B_1 \cup B_2 \cup \ldots \cup B_k$ into k arbitrary chunks, such that a pair of chunks from A and B can fit in the GPU's memory. We then process pairs of chunks independently, effectively reducing the original join $A \bowtie_\epsilon B$ into $j \cdot k$ subproblems, $A_1 \bowtie_\epsilon B_1 \cup A_1 \bowtie_\epsilon B_2 \cup \ldots A_2 \bowtie_\epsilon B_1 \cup \ldots A_j \bowtie_\epsilon B_k$. This data partitioning scheme has its drawbacks: if j and k are large, too many subproblems will bog down the execution time.

However, we can improve the above scheme's work efficiency by creating partitions in such a way as to avoid unnecessary work. For example, suppose that all the points of a particular chunk A_h are at least ϵ distance from all points of another chunk B_i. We know that $A_h \bowtie_\epsilon B_i$ would yield no results, and need not be processed. In fact, any data partitioning technique, such as [9], [10], [12], can be used in conjunction with the LSS algorithm. These data partitioning techniques reduce the original similarity join problem into several subproblems of suitable sizes, which can then be processed on the GPU using the LSS algorithm. Moreover, each chunk can be several hundreds of megabytes in size, to take advantage of the parallelism and high computational throughput afforded by the GPU.

C. Building an SZL on A versus B

Our join algorithm's time complexity is given by Equation (1). However, consider what would happen if the roles of A and B were reversed; that is, the algorithm constructs an SZL on B, and executes parallel interval searches for points in A. The resulting time complexity is given by

$$O(d^2 \frac{n}{k} \log^2 n) + O(d \frac{m}{k} \log n) + O(\frac{m}{k} f(d, \epsilon)). \qquad (2)$$

Our algorithm therefore has two strategies to process a similarity join: build the SZL on A, or build the SZL on B. These strategies have time complexities given by Equations (1) and (2). Observe that for both (1) and (2), the cost of creating the SZL dominates the other two terms. Therefore, it is advantageous to compute the SZL on the smaller of A and B. If $|A| < |B|$, strategy (1) is the best choice; otherwise, strategy (2) is preferred. Moreover, if $|A|$ and $|B|$ are nearly equal, our experimental results show that it is still best to create the SZL on the smaller of the two datasets (see Section VI).

D. Changing SZL Size

Our algorithm constructs the SZL on A, which consists of $(d + 1)$ Z-lists on A. The SZL provides an upper bound on the size of the search interval for each point in B. However, creating the SZL requires $d + 1$ invocations of the parallel sorting algorithm. If we reduce the size of the SZL, we might be able to reduce the algorithm's total running time.

First of all, we point out that the correctness of our algorithm does not depend on the SZL's size. That is, the algorithm would still produce the correct output with one or any subset of the Z-lists on A. Furthermore, in some cases, the cost of creating a Z-list on A will outweigh its benefit in terms of reducing the sizes of minimum search intervals. Most of the time, it is enough to only create a subset of the $d+1$ space filling curves. In Section VI, we describe alternate formulations of the LSS algorithm, where the SZL of A may contain fewer than $(d + 1)$ Z-lists. Using extensive empirical analysis, we provide heuristics for choosing an appropriate SZL size that results in improved time efficiency.

E. LSS Variants

In this section, we discuss several variations of the similarity join operation, and show how the LSS algorithm can be easily modified to support these alternate formulations.

Computing k-nearest neighbors: With small changes, the LSS algorithm can compute the k-nearest neighbors in A to each point in B. Our approach is similar to the method used in [7], [28] in order to compute k exact and approximate nearest neighbors to a query point. We exploit the *locality-preserving* property of space-filling curves, which states that points that are close on a space-filling curve are usually close in the original $\mathbb{R}^d$ space. As before, the algorithm computes the SZL Z_A on A. For each $p \in B$, the algorithm then computes a set K_p of k approximate nearest neighbors to p, by examining all the points within k index positions of p in any of the $(d+1)$ Z-lists in Z_A.

Let p_k be the $k\text{th}$ approximate nearest neighbor to p in K_p. We point out that the actual k closest point to p can be no farther than $\epsilon_p = D(p, p_k)$ from p. The algorithm therefore searches a hypercube of side length $2 \cdot \epsilon_p$ centered at p, which is guaranteed to contain the actual k nearest neighbors to p. It proceeds as before, except that every p has a different search region. The algorithm reports the k closest points contained in the minimum search interval in Z_A, as defined by ϵ_p.

Computing a self-join: If A and B point to the same dataset S, our algorithm will report each pair twice – two points $p, q \in S$ within distance ϵ of each other would produce (p, q) and (q, p) in the result set. Fortunately, we can make a simple modification to the LSS algorithm to eliminate duplicate pairs from the result set, and at the same time speed up the algorithm's execution. For each $p \in B$, Algorithm 1 performs a region search corresponding to a hypercube R of side length $2 \cdot \epsilon$ centered at p, such that $p^{T(-\epsilon)}$ and $p^{T(\epsilon)}$ are the two antipodal points of R. We modify the algorithm to instead search a hypercube R' of side length ϵ, such that p and $p^{T(\epsilon)}$ form the two antipodal points of R'. The algorithm also ignores any pairs (p, q) where $p = q$. To ensure the modified algorithm's correctness, note that if $p \preceq_Z q$, the pair (p, q) would be in the result set; otherwise, (q, p) would be reported. Moreover, as R' is smaller than R, this simple modification results in smaller minimum search intervals, which translates to a more time-efficient algorithm.

Using other space-filling curves: Instead of using the Z-order for building curves in the SZL, other space-filling curves could be used, such as Hilbert curves [4]. We can

TABLE I
PROPERTIES OF THE COREL IMAGE FEATURES DATASETS

Dataset	Size	Dimensions
ColorHistogram	68,040	32
ColorMoments	68,040	9
CoocTexture	68,040	16
LayoutHistogram	66,616	32

define the *Hilbert-ordering relation* analogously to the Z-ordering relation (Section IV): $p \preceq_H q \iff H(p) \leq H(q)$. Unfortunately, the Hilbert curve does not satisfy Lemma 4.1, as adapted to the Hilbert-ordering relation. That is, if a point q dominates another point p, it is not always true that $p \preceq_H q$. As a result, it would not be sufficient to only examine the minimum search interval spanned by the search region $R_\epsilon(p)$. All the points spanned by the minimum search interval of the smallest r-region R^+ containing $R_\epsilon(p)$ would need to be examined, leading to more distance computations than the original algorithm using the Z-ordering relation. In addition, computing $H(p)$ takes more computation time than $Z(p)$, which would further slow the similarity join.

However, as Hilbert curves have better locality-preserving behavior, using Hilbert curves for the SZL would improve the approximate k-nearest neighbors variant discussed earlier. Each K_p computed by the algorithm would be more accurate, and thus improve each ϵ_p estimate.

Producing sorted output: Our join algorithm can be used in complex query processing scenarios (*e.g.*, *cascading join* operations [31]), where it is beneficial for the similarity join output to be sorted, or in some cases, to preserve the ordering of one of the input datasets. Our algorithm uses the prefix-sum operation to allocate output buffer space, which preserves the input ordering of B in the result set. Furthermore, an additional sort of the result set can preserve the original ordering of A. Note that the minimum search interval for a point in B contains the index positions of points in A, some of which are added to the result set if they are within a distance ϵ. The result set therefore contains pairs (p, q), such that p is the index position of a point in A. By applying a stable sort to the result set, where pairs (p, q) are arranged in increasing order of p, we restore the original ordering of points in A. The result set now has a primary ordering based on A, and a secondary ordering on B.

VI. EXPERIMENTAL ANALYSIS

We performed experiments on both synthetic and real-world datasets. For our synthetic data, we generated datasets drawn from an uniform distribution of up to 4M points and 1024 dimensions. Our real-world data consists of the four datasets from the Corel Image Features collection, available from the UC Irvine KDD repository [32]. These datasets contain image features extracted from a collection of over 60,000 photo images. Table I lists the properties of the Corel datasets used in our evaluation.

All our experiments were performed on a quad-processor Intel Xeon 2.4GHz with 1GB main memory, running Red Hat Enterprise Linux 3; one processor was dedicated to running the experiments. We executed LSS on a NVIDIA GeForce 8800 GTX, with 128 stream processors and 768MB of device memory. It was implemented using the C++ interface of the NVIDIA CUDA Toolkit version 0.9, and was executed with the NVIDIA Display Driver version 100.14.10. The EGO [9], [11] and GESS [10] algorithms were implemented in C++, and were allowed buffer sizes large enough to fit the entire dataset in main memory. We incorporated some of the improvements to the EGO algorithm suggested by Kalashnikov and Prabhakar [11]. The GESS algorithm was implemented using a Hilbert space-filling curve [4], and the size of the algorithm's sort codes were limited to 256 bytes. Moreover, we adjusted the value of k in the GESS algorithm in order to ensure good performance. The times reported in our experiments is the sum total of the index construction time, I/O time, and join processing time. In practice, we found I/O times to be insignificant when compared to the actual join processing times, so they are not shown separately in our experimental results. Furthermore, all our experiments used the Euclidean (L_2) distance metric.

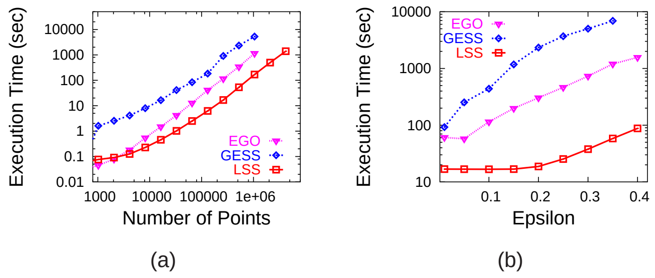

Figure 3. Execution times on uniformly-distributed data, for (a) varying n up to 4M points, ϵ=0.1, d=16, and (b) n=256k, varying ϵ, d=16

Figure 3 shows the execution times for EGO, GESS, and LSS when processing uniformly-distributed random data. Figure 3a shows that both EGO and LSS outperform GESS for all tested values of n. For $1\text{k} \leq n \leq 1\text{M}$, LSS is faster than GESS by factors of 21.2-44.1. EGO is slightly faster than LSS for $n < 2\text{k}$, but LSS performs better for larger n, with an improvement factor of 6.8 at $n = 1\text{M}$ points. LSS's speedup is more apparent as ϵ increases, as shown in Figure 3b. For $0.01 \leq \epsilon \leq 0.35$, LSS outperforms GESS by factors of 5.5-118.3, and its speedups over EGO are 3.6-26.9.

Figure 4 shows the results for the EGO, GESS and LSS methods when applied to our four real-world datasets, for varying values of ϵ. We can see that LSS shows dramatic performance improvements over GESS, with improvement factors of 1.6–117.2 over the values of ϵ examined. Furthermore, on average, LSS is 10.2 times faster than EGO.

In Figure 5, we show LSS's execution time in terms of its two principal components: the time taken to construct the SZL on A (SZL), and the time taken to examine the minimum search intervals of all points in B (IS). We can see that only d affects SZL construction time, but the IS time depends on both d and ϵ. As shown in Figure 5a, for smaller

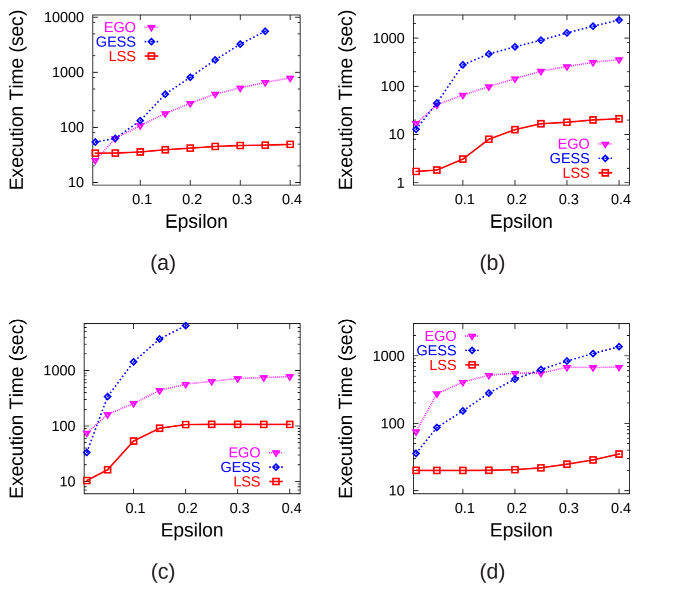

Figure 4. Execution times for varying ϵ on (a) ColorHistogram, (b) ColorMoments, (c) CoocTexture, and (d) LayoutHistogram

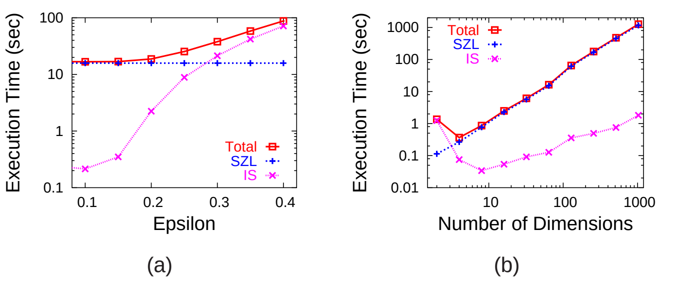

Figure 5. LSS execution time in terms of SZL construction and interval search, for (a) n=256k, varying ϵ, d=16, and (b) n=64k, $\epsilon = 0.1$, varying d

$\epsilon < 0.2$, the SZL construction dominates LSS's running time, as most points in B have small minimum search intervals due to the small ϵ. However, for $\epsilon > 0.25$, the interval search starts dominating the total running time. Figure 5b shows the effect of increasing dimensionality of the input datasets. For small d, IS time dominates LSS's total running time, but SZL construction time quickly overtakes the interval search time at $d = 4$. This observation can be explained as follows. First, every additional dimension adds another Z-list to the SZL, thereby increasing the SZL construction time. Second, for uniform data, increasing d while holding n and ϵ constant will produce fewer pairs of points in the result set, so the average minimum search interval will span fewer points.

In Section V-C, our analysis of the LSS algorithm showed that it is beneficial to build the SZL on the smaller of its two input datasets. Figure 6 shows our experimental results, which confirm our earlier hypothesis. Here, we will refer to the larger dataset as L, and the smaller dataset as S. The curve labeled "SZL-L" refers to an LSS variant where the SZL is constructed on L, while "SZL-S" corresponds to building the SZL on S. Furthermore, L's size was fixed at 4M points, while the size of S was varied. Figure 6a shows that the total execution time for SZL-L is large, since the SZL construction dominates the execution time. In contrast, SZL-S scales smoothly as $|S|$ increases, indicating that its execution time is more balanced between the SZL construction and interval search. Figure 6b shows the average minimum search interval (MSI) size per point, for both the SZL-L and SZL-S variants. As expected, SZL-L examines smaller minimum search intervals than SZL-S for smaller values of $|S|$. However, for $|S| > 10$k, the average minimum search intervals for SZL-L and SZL-S are almost equal, indicating that there is no benefit from building the SZL on L.

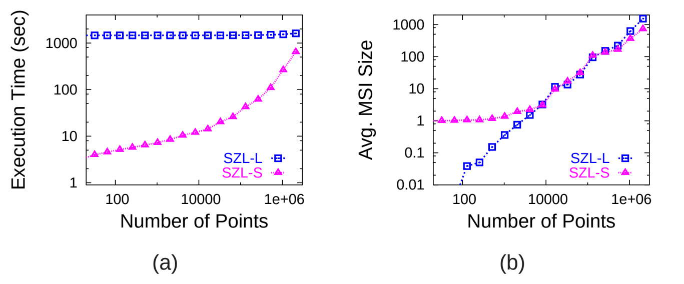

Figure 6. Effects of creating the SZL on the larger dataset (n=4M) versus the smaller, for ϵ=0.1, d=16

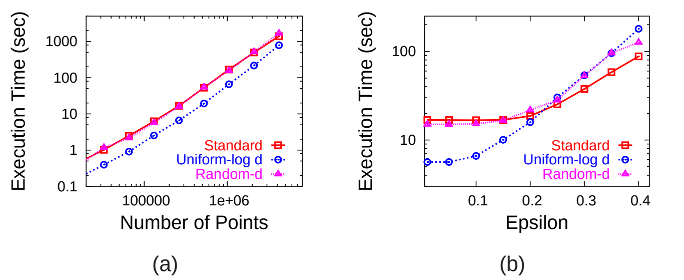

Figure 7. Execution times of LSS variants for (a) varying n, ϵ=0.1, d=16, and (b) n=256k, varying ϵ, d=16

We now examine the performance of the LSS algorithm when using a different set of translation vectors (described in Section IV). In Figure 7, we compare standard LSS's execution time to that of two of its variants: "*Uniform*-$\log d$" and "*Random*-d". The *Uniform*-$\log d$ variant creates $(\log d + 1)$ Z-lists, such that the translation vectors are uniformly chosen between $[0, 1]$, while the *Random*-d variant generates $(d+1)$ Z-lists, with randomly chosen translation vectors between $[0, 1]$. The expected performance of the latter variant is discussed in [15]. Figure 7a shows comparison results for varying n and ϵ fixed at 0.1. Uniform-$\log d$ performs better than standard LSS and Random-d, while Random-d has performance results similar to standard LSS. Figure 7b shows the effect of varying ϵ while keeping n constant. For $\epsilon < 0.2$, Uniform-$\log d$ outperforms both standard LSS and Random-d, by a factor of 3 at $\epsilon = 0.01$. However, for larger ϵ, LSS outperforms the Uniform-$\log d$, showing an improvement factor of 2 at $\epsilon = 0.4$. In general, as ϵ increases, it is worthwhile to generate more Z-lists for the SZL.

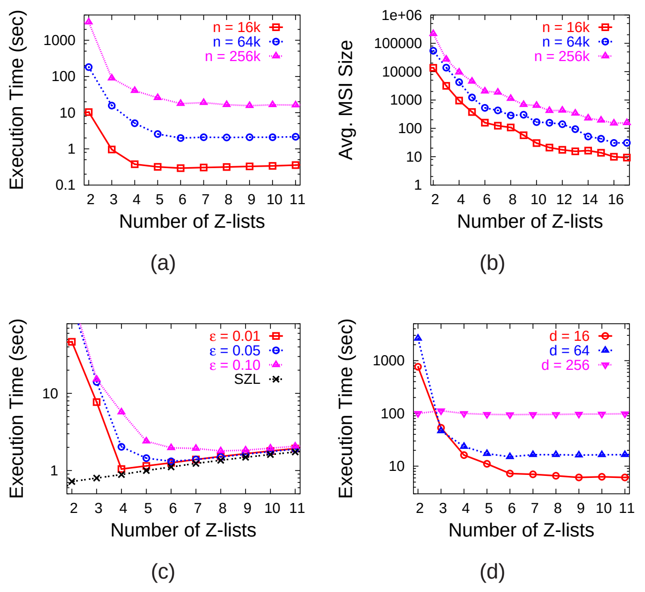

Figure 8. Effects of reducing SZL size for (a,b) ϵ=0.1, d=16, (c) n=64k, d=16, and (d) n=128k, ϵ=0.1

LSS's performance varies significantly when the SZL's size is altered, as discussed in Section V-D. We examined the effect of SZL size on execution time and average size of the minimum search interval, in order to determine the best SZL size for any similarity join scenario. Figure 8 shows our results for varying SZL sizes and uniformly spaced translation vectors depending on the SZL size. As is clear from Figures 8a and 8b, n does not play a role in determining the optimal SZL size. In contrast, Figure 8c shows that the best SZL size depends on ϵ; for $0.01 \leq \epsilon \leq 0.10$, the optimal SZL size is between 4-8 Z-lists. It also shows that as ϵ increases, it is worthwhile to create more Z-lists, as it improves both LSS's time and work efficiencies. Figure 8d shows that for $d \in \{16, 64, 256\}$, the best SZL sizes were 9, 6, and 5, respectively.

VII. Conclusions

In this paper, we introduced the LSS algorithm, a time-efficient algorithm to perform similarity join queries on the GPU. The algorithm uses the SZL data structure, which strikes a good balance between time and work efficiency. The SZL is a simple data structure that can be easily adapted to perform other parallel database operations on the GPU. In our future research, we will perform a more rigorous analysis of the SZL data structure, and investigate its applicability to other operations on high-dimensional databases. We will also further explore the effect of varying the SZL's size and using different sets of translation vectors.

GPUs represent a new generation of inexpensive, widely-available parallel machines, but CPUs will eventually catch up in terms of parallel processing power. Already, modern CPUs feature multiple cores on-chip, allowing several threads to execute in parallel. As CPUs continue to become more like parallel machines, we believe that algorithms like LSS will become increasingly relevant for mainstream database query processing.

VIII. Acknowledgements

The authors wish to thank Dr. Edwin Jacox for his assistance in preparing the GESS and EGO codebases, Dr. Amitabh Varshney for the generous use of his GPU computing cluster, and Dr. David Mount for pointing us to the work by Chan [16].

References

[1] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds., *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[2] D. Jurafsky and J. H. Martin, *Speech and language processing: An introduction to natural language processing, computational linguistics and speech recognition.* Upper Saddle River, NJ: Prentice Hall, Jan. 2000.

[3] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, 2nd ed. Boston: Academic Press, 1990.

[4] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA: Morgan-Kaufmann, 2006.

[5] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The pyramid-technique: towards breaking the curse of dimensionality," in *SIGMOD'98*, Seattle, WA, June 1998, pp. 142–153.

[6] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB'99*, Edinburgh, Scotland, Sept. 1999, pp. 518–529.

[7] S. Liao, M. A. Lopez, and S. T. Leutenegger, "High dimensional similarity search with space filling curves," in *ICDE'01*, Heidelberg, Germany, Apr. 2001, pp. 615–622.

[8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," in *SIGMOD'93*, Washington, DC, May 1993, pp. 237–246.

[9] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data," in *SIGMOD'01*, Santa Barbara, CA, May 2001, pp. 379–390.

[10] J.-P. Dittrich and B. Seeger, "GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces," in *KDD'01*, San Francisco, CA, Aug. 2001, pp. 47–56.

[11] D. V. Kalashnikov and S. Prabhakar, "Similarity join for low-and high-dimensional data," in *DASFAA '03*, Kyoto, Japan, Mar. 2003, pp. 7–16.

[12] K. Shim, R. Srikant, and R. Agrawal, "High-dimensional similarity joins," in *ICDE'97*, Birmingham, U.K., Apr. 1997, pp. 301–311.

[13] N. Koudas and K. C. Sevcik, "High dimensional similarity joins: algorithms and performance evaluation," *TKDE*, vol. 12, no. 1, pp. 3–18, January/February 2000.

[14] H. Sagan, *Space-Filling Curves.* New York City, NY: Springer-Verlag, 1994.

[15] M. Bern, "Approximate closest-point queries in high dimensions," *IPL*, vol. 45, no. 2, pp. 95–99, Feb. 1993.

[16] T. M. Chan, "Approximate nearest neighbor queries revisited," in *SCG'97*, Nice, France, June 1997, pp. 352–358.

[17] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computing Conf.*, Atlantic City, NJ, Apr. 1968, pp. 307–314.

[18] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *SIGMOD'05*, Baltimore, MD, June 2005, pp. 611–622.

[19] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD'04*, Paris, France, June 2004, pp. 215–226.

[20] C. Sun, D. Agrawal, and A. El-Abbadi, "Hardware acceleration for spatial selections and joins," in *SIGMOD'03*, San Diego, CA, June 2003, pp. 455–466.

[21] N. Bandi, C. Sun, A. El-Abbadi, and D. Agrawal, "Hardware acceleration in commercial databases: A case study of spatial operations." in *VLDB'04*, Toronto, Canada, Aug. 2004, pp. 1021–1032.

[22] R. Fosner, *Real-Time Shader Programming, Using DirectX 9.0.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[23] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] NVIDIA Corporation, “NVIDIA CUDA Compute Unified Device Architecture programming guide,” http://developer.nvidia.com/cuda.

[25] G. M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing,” IBM Ltd., Ottawa, Canada, Tech. Rep., 1966.

[26] I. Gargantini, “An effective way to represent quadtrees,” *CACM*, vol. 25, no. 12, pp. 905–910, Dec. 1982.

[27] H. Tropf and H. Herzog, “Multidimensional range search in dynamically balanced trees,” *Angewandte Informatik*, vol. 23, no. 2, pp. 71–77, Feb. 1981.

[28] M. A. Lopez and S. Liao, “Finding k-closest-pairs efficiently for high dimensional data,” in *CCCG’00*, Fredericton, Canada, Aug. 2000, eproceedings.

[29] G. E. Blelloch, “Prefix sums and their applications,” in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. San Francisco, CA: Morgan Kaufmann, 1990, pp. 35–60.

[30] U. Vishkin, “Thinking in parallel: Some basic data-parallel algorithms and techniques,” College Park, MD, 2007, http://www.umiacs.umd.edu/~vishkin/PUBLICATIONS/classnotes.pdf.

[31] W. G. Aref and H. Samet, “Cascaded spatial join algorithms with spatially sorted output,” in *ACM-GIS ’96*, Gaithersburg, MD, Nov. 1996, pp. 17–24.

[32] S. Hettich and S. D. Bay, “The UCI KDD Archive,” http://kdd.ics.uci.edu, University of California, Irvine, CA.