Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs

Fabian Gieseke FABIAN.GIESEKE@DIKU.DK
Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark

Justin Heinermann JUSTIN.HEINERMANN@INFORMATIK.UNI-OLDENBURG.DE
Department of Computing Science, University of Oldenburg, Uhlhornsweg 84, 26111 Oldenburg, Germany

Cosmin Oancea COSMIN.OANCEA@DIKU.DK
Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark

Christian Igel IGEL@DIKU.DK
Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark

Abstract

We present a new approach for combining k-d trees and graphics processing units for nearest neighbor search. It is well known that a direct combination of these tools leads to a non-satisfying performance due to conditional computations and suboptimal memory accesses. To alleviate these problems, we propose a variant of the classical k-d tree data structure, called buffer k-d tree, which can be used to reorganize the search. Our experiments show that we can take advantage of both the hierarchical subdivision induced by k-d trees and the huge computational resources provided by today's many-core devices. We demonstrate the potential of our approach in astronomy, where hundreds of million nearest neighbor queries have to be processed.

1. Introduction

Finding the nearest neighbors for a query object is fundamental in data analytics. Given both a large reference and query set, however, the involved nearest neighbor computations can quickly become a bottleneck. Depending on the particular learning task at hand (e.g., the size of the reference/query set or the dimensionality of the input space), various techniques can be used to accelerate the search. Prominent examples are spatial data structures such as *k-d* and *cover trees* (Bentley, 1975; Beygelzimer et al., 2006) or *locality-sensitive hashing* (Indyk & Motwani, 1998).

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

A recent trend is to resort to *graphics processing units* (GPUs) for accelerating the search (Cayton, 2012; Garcia et al., 2010; Pan & Manocha, 2011). One way to utilize such devices is to parallelize the search over the query points in a brute-force manner, which can lead to significant speed-ups. In low-dimensional feature spaces, however, the performance gain over spatial search structures vanishes with increasing data set sizes. Therefore, a desirable goal is to combine the benefits of both worlds.

A typical parallel k-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously. While such a scheme performs well on multi-core machines, it is ill-suited for GPU execution since each query may induce a completely different tree traversal. This results in both branch divergence and irregular, prohibitively expensive accesses to global memory (negating much of the potential benefits of using a k-d tree).

This work presents a *general purpose computation on graphics processing units* (GPGPU) solution to *exact* nearest neighbor search in Euclidean spaces. Our approach relies on a *memory-centric* (Oancea et al., 2009; Shuf et al., 2002) refinement of a classical k-d tree, which we name *buffer k-d tree,* and enables an effective, massively-parallel processing of huge amounts of queries. Each leaf of the new tree structure corresponds to a set of reference patterns and uses a buffer to delay the processing of the queries reaching that leaf until enough work has been gathered. Processing all buffered queries is performed in a brute-force manner. Here, we take advantage of the fact that the queries collected in the same buffer are compared with the same pattern in the same block-wide SIMD (single instruction, multiple data) instruction, and such memory accesses are effectively supported by today's GPUs via hardware caches (Jia et al., 2012).

Our approach is memory-centric in the sense that it reorganizes the program's control flow to improve locality of reference.[1] The experimental evaluation, conducted on a commodity CPU+GPU system, demonstrates a significant performance gain of our approach over a brute-force GPU scheme (2–55$\times$) and a multi-threaded k-d tree implementation (5–34$\times$), given manually tuned CPU k-d tree heights (and using all four cores). Our approach is suitable for applications that permit "lazy querying" (i.e., in cases where an increase of response latency for single test patterns is not harmful), for example when a large batch of test queries needs to be processed.

2. Background

We briefly sketch k-d tree-based and parallel nearest neighbor search (see Andoni & Indyk, 2008, for other schemes).

2.1. Revisited: Nearest Neighbors via k-d-Trees

A *k-d tree* (Bentley, 1975; Friedman et al., 1977) for a set $P = \{\mathbf{x}, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ of reference points is a balanced binary tree. The root of the tree corresponds to all points and its two children represent disjoint (almost) equal-sized subsets of P. Splitting up the points into subsets is done in a level-wise manner, starting from the root (at level 0) down to the leaves. For a given node v at level i, the points associated with v are split into two halves by resorting to the median in dimension $i \bmod d$ (other splitting rules may also be applied). The recursive construction ends as soon as a node v corresponds to a singleton or to a set of predefined size. A k-d tree can be constructed in $\mathcal{O}(n \log n)$ time via linear-time median-finding and occupies linear space.

The nearest neighbor search makes use of the hierarchical subdivision induced by the tree: Let $\mathbf{q} \in \mathbb{R}^d$ be a query point. To find its nearest neighbor in P (the generalization to $k > 1$ neighbors is straightforward), one traverses the tree in two phases. In the first phase, one navigates down from top to bottom to find the d-dimensional box that contains $\mathbf{q}$ (using the median values stored in the nodes). By computing the distances between $\mathbf{q}$ and all points stored in the associated leaf, one can identify an initial nearest neighbor candidate. In the second phase, one traverses the tree from bottom to top, and on the way back to the root, one checks if neighboring boxes potentially contain points that are closer to $\mathbf{q}$ as the current best candidate (using the median values). In case a point might be closer, one recurses to the subtree that has not yet been visited.

For low-dimensional spaces, a small number of leaf visits is often enough (yielding a logarithmic runtime). The performance can, however, significantly decrease with increasing d due to the *curse of dimensionality* (Hastie et al., 2009).

2.2. Nearest Neighbor Search on GPUs

Nowadays GPUs offer massive parallelism (e.g., 2048 cores). They rely on a simplified control unit and an explicitly programmable memory hierarchy, in which global memory is up to 100$\times$ slower than, e.g., local memory. Two memory access patterns can lead to smaller latencies: The first one corresponds to *coalesced accesses*, in which, informally, groups of threads access consecutive memory locations. The second one is *caching*, which is given when a group of consecutive threads (repeatedly) accesses nearby memory locations; such memory operations can effectively be accelerated using hardware caches (Jia et al., 2012).

Most of the work on computing nearest neighbors using GPUs so far has focused on the domain of computer graphics (e.g., on *ray tracing*) with data structures that are adapted to the specific needs of such tasks (Popov et al., 2007; Zhou et al., 2008; Wald & Havran, 2006; Horn et al., 2007). For the more general task of nearest neighbor search in higher dimensions (e.g., $\mathbb{R}^{10}$), only few schemes have been proposed: A direct way is to parallelize the search over the query points. This approach, followed by Garcia et al. (2010), offers a tremendous speed-up given medium-sized data sets. However, it fails for large-scale settings with both many reference and query points. Pan & Manocha (2011) propose an efficient GPU implementation of locality-sensitive hashing, which is also applicable to large-scale settings, but possibly yields inexact answers (as its sequential analog). Other schemes are based on the efficient use of texture memory or on adapted sorting schemes (Bustos et al., 2006; Sismanis et al., 2012).

Some k-d tree-based methods have been proposed for, e.g., the computation of forces between particles (Qiu et al., 2009; Wang & Cao, 2010; Heinermann et al., 2013; Nakasato, 2012). However, these schemes are designed for—and limited to—very low-dimensional feature spaces such as $\mathbb{R}^3$ or $\mathbb{R}^4$ and are conceptually very different from our framework. The approach most related to our work is given by Cayton (2012) and is based on a *random ball cover* data structure. Similarly to k-d trees, this data structure induces a subdivision of the search space (into d-dimensional balls), which can be used to accelerate the search. In contrast to our work, however, no associated tree structure is employed for guiding the pruning process.

Hence, except for the tree-based schemes mentioned above, all approaches either provide efficient but possibly inexact answers or address special instances (e.g., small d or n). Note that other parallel schemes (not using GPUs) have been proposed such as "distributed" k-d trees, which can be queried in a map-reduce manner (Aly et al., 2011).

[1]This is also related to *I/O-efficient algorithms* (Aggarwal & Vitter, 1988; Brodal & Fagerberg, 2002), which gather large amounts of data items before invoking a memory transfer.

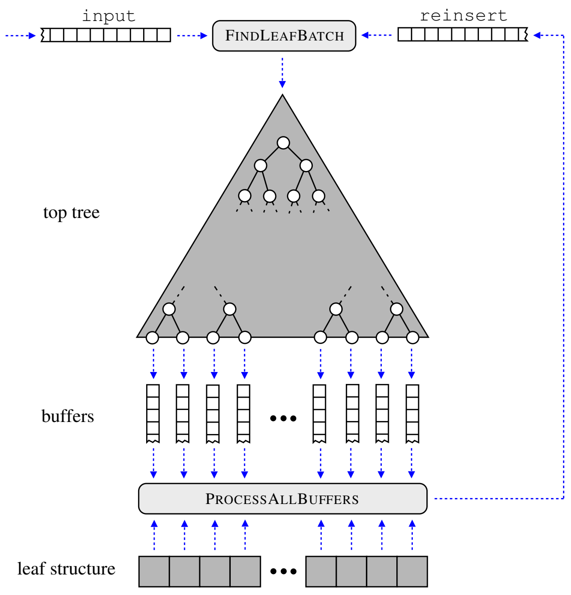


Figure 1. A buffer k-d tree: In each iteration, the procedure FindLeafBatch removes query indices from both queues and distributes them to the buffers (or removes them if no further processing is needed). In case enough work has been gathered, the procedure ProcessAllBuffers is invoked, which updates the nearest neighbors and reinserts all query indices into `reinsert`. The process stops as soon as both queues and all buffers are empty.

Algorithm 1 LazyParallelNN

```
Require: A chunk $Q = \{\mathbf{q}_1, \ldots, \mathbf{q}_m\} \subset \mathbb{R}^d$ of query points.
Ensure: The $k \geq 1$ nearest neighbors for each query point.
1:  Construct buffer $k$-d tree $\mathcal{T}$ for $P = \{\mathbf{x}, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$.
2:  Initialize queue input with all $m$ query indices.
3:  while either input or reinsert is non-empty do
4:      Fetch $M$ indices $i_1, \ldots, i_M$ from reinsert and input.
5:      $r_1, \ldots, r_M$ = FindLeafBatch($i_1, \ldots, i_M$)
6:      for $j = 1, \ldots, M$ do
7:          if $r_j \neq -1$ then
8:              Insert index $i_j$ in buffer associated with leaf $r_j$.
9:          end if
10:     end for
11:     if at least one buffer is half-full (or queues empty) then
12:         $l_1, \ldots, l_N$ = ProcessAllBuffers()
13:         Insert $l_1, \ldots, l_N$ into reinsert.
14:     end if
15: end while
16: return list of $k$ nearest neighbors for each query point.
```

3. Algorithmic Framework

The basis for the reorganized querying process is the concept of buffer k-d trees, which we describe next.

3.1. Buffer k-d Trees

A buffer k-d tree is composed of four parts: (1) a top tree, (2) a leaf structure, (3) a set of buffers (one buffer per leaf of the top tree), and (4) two input queues, see Figure 1.

The top tree consists of the first h levels of a standard k-d tree (i.e., its median values), laid out in memory in a pointer-less manner (the root is stored at index 0 and the children of a node v with index i are stored at indices $2i$ and $2i+1$, respectively). During the construction of the top tree, all patterns are ordered *in-place* w.r.t. the median values such that all points of a leaf are stored *consecutively* in memory.[2] The leaf structure consists of blocks and stores all rearranged patterns. The blocks are in a one-to-one correspondence with the leaves of the top tree. Again, no pointers are needed to link the blocks with their associated leaves (the block of leaf i is stored at index $i - (2^h - 1)$). In addition to the rearranged patterns, two integer variables are stored in each block to indicate the leaf/block bounds.

[2]We keep track of the original order (needed for reporting the nearest neighbors) by using an additional array of original indices that is sorted simultaneously in each step.

The third component consists of buffers, one buffer for each leaf of the top tree. These buffers will be used to store query indices and can accommodate a predefined number $B > 1$ of integers each. In addition, a variable that determines the filling status is stored for each buffer. Finally, we allocate space for two (first-in-first-out) queues of size m. The *height* of the buffer k-d tree is given by the height of its top tree ($h = 0, 1, \ldots$) and B denotes its *buffer size*.

Proposition 1. *A buffer k-d tree of height h for a set $P = \{\mathbf{x}, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ of reference and a chunk $Q = \{\mathbf{q}, \ldots, \mathbf{q}_m\} \subset \mathbb{R}^d$ of query points uses $\mathcal{O}(2^{h+1})$ additional floats and $\mathcal{O}(n + m + 2^h B)$ additional integers. It can be constructed in $\mathcal{O}(hn) \subset \mathcal{O}(n \log n)$ time.*

Since the height of the buffer k-d tree will be reasonably small (e.g., $h = 8$), the space overhead is negligible (and mainly dominated by the space for the buffers, which can further be reduced via dynamic memory allocation). Buffer k-d trees can be adapted to support the insertion and deletion of reference patterns (as k-d trees by, e.g., reserving more space as initially needed for the leaf structure).

3.2. Lazy Parallel Nearest Neighbor Search

We are now ready to describe the reformulation of the tree-based search: The main idea is to “delay” the querying process by performing several iterations. In each iteration, query indices are propagated through the top tree and are stored in the corresponding buffers. As soon as the buffers get full, all collected nearest neighbor queries are processed at a single blow. This essentially leads to a separation of the two main phases of the classical k-d tree-based search: (1) finding the leaf that needs to be processed next and (2) updating the nearest neighbors. The first phase cannot be parallelized easily on GPUs. However, the second one is

Algorithm 2 FINDLEAFBATCH

```
Require: A sequence $i_1, \ldots, i_M \in \{1, \ldots, m\}$ of query indices.
Ensure: A sequence $r_1, \ldots, r_M$ of leaf indices.
1: for all $i_1, \ldots, i_M$ process $k$-d tree in parallel
2:     Initialize stack for $i_j$ (that stems from the previous call)
3:     Find next leaf index $r_j$ for $i_j$ (-1 if root is reached twice).
4: end for
5: return $r_1, \ldots, r_M$
```

much more amenable to such devices—and constitutes by far the most significant part of the overall runtime.

3.2.1. WORKFLOW: LAZY QUERYING

The modified workflow is shown in Algorithm 1: In the preprocessing phase, a buffer k-d tree $\mathcal{T}$ is constructed for the set P of reference points (Step 1). After initializing the queue `input` with all query indices (Step 2), the iterative process is started. In the first phase of each iteration, a large (user-defined) number M of indices is removed from both queues, where indices are only removed from `input` if `reinsert` is empty (Step 4). Afterwards, one invokes the procedure FINDLEAFBATCH (Algorithm 2) to obtain, for each of these query indices, the corresponding leaf that needs to be processed next (Step 5). This procedure essentially simulates the (original) recursive tree traversal by keeping explicitly track of a recursion stack for each query index.[3] In case such an associated tree traversal has reached the root (and both subtrees have been visited), the query index has been processed completely and can be removed from the overall process. All other indices are moved into the appropriate buffers (Steps 6–10).

In the second phase, all buffers are processed. In case one of the buffers has reached a certain filling status (e.g., half-full), the procedure PROCESSALLBUFFERS (Algorithm 3) is invoked, which empties all buffers and updates, for each query index being removed, the associated list of nearest neighbors found so far (Step 12). This is accomplished in a brute-force manner by comparing the query with all reference patterns of the associated block of the leaf structure. Afterwards, all indices $l_1, \ldots, l_N$ taken from the buffers are inserted into `reinsert` (Step 13). The overall process stops as soon as no indices are left in *both* queues *and* the buffers (in practice, a brute-force step is applied as soon as the total number of remaining indices is small enough).

3.2.2. PROCESSING TIME

To sum up, we do not process all queries separately, but split the search process into two phases each handling large chunks of indices. Keeping track of the current "tree traversals" is achieved by explicitly managing the induced recursion stacks. Note that exactly the same leaves are visited for each query index as for the classical k-d tree traversal.

[3]A stack of size h is associated with each query index and is maintained for the duration of that index being alive.

Algorithm 3 PROCESSALLBUFFERS

```
Ensure: A sequence $i_1, \ldots, i_N \in \{1, \ldots, m\}$ of query indices.
1: $I = \varnothing$
2: for $j = 1, \ldots, 2^h$ do
3:     Remove all query indices $i_1, \ldots, i_{N(b_j)}$ from buffer $b_j$.
4:     for all $i_1, \ldots, i_{N(b_j)}$ do in parallel
5:         Update nearest neighbors w.r.t. all points in the leaf
           associated with the buffer $b_j$.
6:     end for
7:     $I = I \oplus i_1, \ldots, i_{N(b_j)}$ (concatenate indices)
8: end for
9: return $I$
```

Proposition 2. *Processing $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^d$ queries via Algorithm 1 takes the same (asymptotic) runtime as the original k-d tree-based search (given same tree heights).*

Thus, except for a small overhead caused by keeping track of the query indices, the runtime is the same as for the original k-d tree traversal. However, the *order* of the query indices is changed, i.e., indices belonging to the same leaf are now processed in chunks, providing data locality.

3.3. GPGPU Implementation

We now sketch our GPGPU implementation and describe, in detail, the kernel implementation for the procedure PROCESSALLBUFFERS, which usually takes more than 99% of the total sequential runtime (see Section 4).

3.3.1. MEMORY LAYOUT AND FINDING LEAVES

The buffer k-d tree is built on the host system (CPU), since the construction time is negligible for processing huge query sets, and all relevant information is copied to the GPU. Other than this, the host system is only used (1) to ensure the flow of query indices between leaf buffers and the queues `input` and `reinsert`, (2) to spawn the GPU kernels, and (3) to transfer arrays of indices to the GPU that associate a given query to the training patterns of its corresponding leaf. It has to be stressed that only indices are moved between the host and the GPU during the iterative process (the d-dimensional training and test patterns need to be copied only *once*). This leads to a negligible overall cost for memory transfer, see Section 4.

The procedure FINDLEAFBATCH operates on the top tree to determine, for each query index, the next leaf that may contain closer neighbors (if such a leaf is found, then the leaf index is returned, otherwise -1 to signal that the stack traversal has reached the root twice). Executing the tree traversal on the GPU exhibits massive flow divergence and irregular, hence expensive, accesses to memory. To minimize these inefficiencies, we use a relatively small top tree

Algorithm 4 `LeafNearestNeighbor`

```
Require: float tests[d,N], float trains[n,d],
         int lbs[N], int ubs[N]
Ensure: float nn_dists[k,N],float nn_inds[k,N]
 1: int tid=get_global_id(0);
 2: float test_patt[d];
 3: float nnds[k]={inf};int nnis[k]={0};
 4: int LB=lbs[tid]; int UB=ubs[tid];
 5: for i=0...d-1 do
 6:   test_patt[i]=tests[i,tid];
 7: end for
 8: for t=LB...UB do
 9:   float dist=0.0;
10:   for j=0...d-1 do
11:     dist+=(trains[t,j]-test_patt[j]);
12:   end for
13:   updateNearestNeighb(dist,t,nnds,nnis);
14: end for
15: for i=0...k-1 do
16:   nn_dists[i,tid]=nnds[i];
17:   nn_inds[i,tid]=nnis[i];
18: end for
```

(e.g., $h = 8$), which results in a suboptimal but still significant speed-up over the sequential execution.

3.3.2. Cached Processing of Buffers

The efficient execution of ProcessAllBuffers is crucial for our approach and implemented via three kernels:

(1) The first kernel (`TestSubset`) takes the query indices from the buffers to fill (on the GPU) an array (`tests`) of consecutive test patterns. The buffers ensure that the `tests` array is inherently sorted w.r.t. the queries' associated leaves.[4] In addition, arrays that associate each query to the lower and upper indices of the corresponding leaf training patterns are computed (named `lbs` and `ubs`). The query indices and these bounds can be loaded in a coalesced way.

(2) The second kernel (`LeafNearestNeighbor`) computes, for each query index, the local k nearest neighbors within its associated leaf. This kernel accounts for more than 90% of the total runtime of ProcessAllBuffers and is examined in detail below.

(3) The third kernel (`Update`) merges, for each query index, the local with the previously known nearest neighbors (coalesced global memory accesses).

The implementation for the `LeafNearestNeighbor` kernel is sketched in Algorithm 4: The arrays `tests` and `trains` hold the test and training patterns in global memory arrays, respectively. Further, the arrays `nn_dists` and `nn_inds` will be filled with the distances and indices of the k nearest neighbors for all test indices. Note that `tests`, `nn_inds`, and `nn_dists` are maintained in transposed form to achieve coalesced access. Finally, the global arrays `lbs` and `ubs` associate the queries with the leaves (i.e., query `tid` $\in \{0, \ldots, N-1\}$ has to be compared with the training patterns from `lbs[tid]` to `ubs[tid]`).

In Steps 5–7, the test pattern is loaded from global to private memory. This access is coalesced since the last index is the global thread id, hence, consecutive threads necessarily access consecutive memory locations. Further uses of `test_patt`, stored in private thread memory, exhibit efficient access. Steps 8–14 compute the distances between all training patterns and the test pattern, and, if necessary, update the nearest neighbors. Finally, Steps 15–18 commit the computed nearest neighbors to global memory and exhibit coalesced accesses.

The access to `trains[t,j]` in Step 11 provides *the main motivation* for buffer k-d trees: First, the SIMD execution ensures that threads within a warp have the same value for `j` and `t`. Second, since many consecutive threads operate on the same leaf, all such threads access nearby memory locations. Hence, this access pattern exhibits both *within-warp* and *within-block data locality* (Jia et al., 2012) yielding latencies comparable to those of constant memory.[5]

4. Experiments

Our method speeds up nearest neighbor search in scenarios with a large amount of training and a huge amount of test patterns, given input spaces with moderate dimensionality. This is precisely the setting faced in astronomy, which we use to demonstrate the applicability of our approach.

4.1. Data Mining in Astronomy

Current projects such as the *Sloan Digital Sky Survey* (SDSS) (Ahn et al., 2013) have gathered terabytes of data for hundreds of million astronomical objects. Upcoming projects will produce these amounts *per night* (Ivezic et al., 2011), with final data volumes in the petabyte range. Machine learning techniques have been identified as "increasingly essential in the era of data-intensive astronomy" (Borne, 2008) and already led to new discoveries (Mortlock et al., 2011). In astronomy, one is often given a huge amount of patterns (e.g., two billion) in a low-dimensional feature space (e.g., $\mathbb{R}^{10}$). For this reason, nearest neighbor models (Polsterer et al., 2013; Stensbo-Smidt et al., 2013) are often among the state-of-the-art models.

4.1.1. Large-Scale Photometric Catalogs

The two most common types of data are *photometric* and *spectroscopic data* (Ahn et al., 2013): The former one es-

[4]The alternative would be explicit (bitonic) sorting.

[5]Demand-fetched caches have been recently introduced by several GPU vendors (e.g., `Nvidia GeForce GTX 770`).

sentially corresponds to images obtained at different wavelength ranges. For a small subset of potentially "interesting" objects (say, a thousandth), time-consuming follow-up observations in terms of spectra are made. One of the main challenges is to identify promising photometric targets for such follow-up observations (needed to verify an object's nature). In a typical application, almost *all* of the spectroscopically confirmed data are used to build models, which are then applied to all remaining objects.[6] Furthermore, computing *exact* answers is important since minor differences w.r.t. the features are crucial to capture the characteristics of astronomical objects.

While spatial search techniques are well-suited for this task, the application of the final model can still easily take hours (or even days) on today's multi-core desktop machines. In contrast, *scanning* large amounts of query patterns (i.e., transferring them between disk and main memory) can be done efficiently due to the data being stored *consecutively* (e.g., in minutes only). Thus, reducing the testing to scanning time is desirable for today's catalogs, and will play a crucial role for upcoming catalogs.

4.1.2. Feature Space Dimensionality

Given photometric data, one usually extracts a small set of expressive features, called *magnitudes*. The basis for the extraction are five grayscale images obtained through five filters (u,g,r,i,z) for each observed region (SDSS). The most established extraction schemes are the *point-spread-function* (psf), the *Model* (model), and the *Petrosian* (pet) approaches (Ahn et al., 2013), and the induced features often form the basis for data mining models. Typical features are the *colors*, which are differences of magnitudes. Below, we consider various combinations of features, see Table 1 (all denotes psf,model,pet).

4.2. Experimental Setup

All experiments were conducted on a standard PC with an Intel(R) Core(TM) i7-3770 CPU at 3.40GHz (4 cores, 8 hardware threads), 16GB RAM, and a GeForce GTX 770 GPU with 1536 shader units (4GB RAM). The operating system was Ubuntu 12.04 (64 Bit). We report runtimes for the query phase (all tree construction times are very small and irrelevant in the above scenario).

All algorithms were implemented in C and OpenCL compiled using Swig with gcc-4.6.3 and -fopenmp as additional compiler option; Python was used to set up the experiments.[7] For a buffer k-d tree of height h, we fixed $B = 2^{22-h}$. In each iteration of Algorithm 1, $M = 5B$ indices were fetched from input and reinsert.[8] We compared the following implementations:

Table 1. Astronomical Data Sets

Data set	d	features
psf_colors	4	psf_{u-g,g-r,r-i,i-z}
psf_mag	5	psf_{u,g,r,i,z}
psf_model_mag	10	psf,model_{u,g,r,i,z}
all_mag	15	all_{u,g,r,i,z}
all_colors	12	all_{u-g,g-r,r-i,i-z}
all	27	all_mag,all_colors

(1) bufferkdtree(gpu): our approach with FindLeafBatch and ProcessAllBuffers on GPU.
(2) bufferkdtree(cpu): a corresponding sequential CPU variant that only operates on the host system.
(3) kdtree(cpu,i): a multi-core implementation of a k-d tree-based search, running i threads in parallel (each handling a single query).
(4) kdtree(gpu): a naïve GPU implementation, which traverses an appropriately built k-d tree in parallel (one thread per query).
(5) bruteforce(gpu): a brute-force GPU implementation that processes all test queries in parallel.

The main baselines are bruteforce(gpu) and kdtree(cpu,i). Both approaches have been evaluated extensively in the literature, which allows to place the runtimes reported in a broader context.

4.3. Results

4.3.1. Tree-Related Experiments

If not stated otherwise, we used psf_colors and psf_model_mag as input data sets and searched for $k = 10$ nearest neighbors given $n = 10^6$ training patterns.

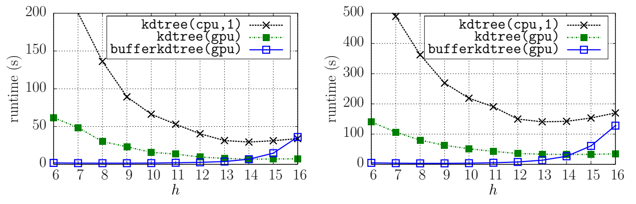
Influence of Tree Heights. For all k-d tree-based approaches, the height h can significantly influence the running time. In Figure 2, the runtime behaviors for varying tree heights are given. For bufferkdtree(gpu), a smaller tree height was favorable (to be explained below), whereas larger assignments for h led to better runtimes for the other schemes. For classical k-d trees, this is a well-known behavior since h determines the trade-off between pruning capability and the overhead for visiting too many leaves (i.e., many leaf visits vs. the overhead for

[6]For the SDSS (DR 9), this amounts to building models based on about two million objects, which have then to be applied to all remaining objects in the catalog (about one billion).

[7]The code is made publicly available on the authors' websites.

[8]The particular assignment for both parameters did not have a significant influence as long as reasonable values were chosen. In rare cases, too many indices are assigned to a single buffer (and must be reinserted into reinsert without any processing).

traversing the tree); the naïve many-core implementation `kdtree(gpu)` exhibited a similar behavior. For the following experiments, we fixed the optimal tree height for each method.

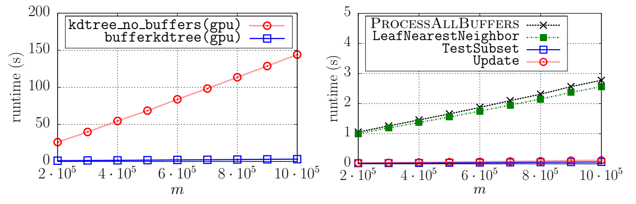


(a) (b)

Figure 2. Influence of the tree height h on the runtime for (a) `psf_colors` and (b) `psf_model_mag` ($n, m = 10^6$, $k = 10$).

Buffering → Caching. The *main benefit* of the delayed querying via a buffer k-d tree is the induced locality of the test and training patterns on the GPU. Note that a similar "delayed" traversal can also be achieved without any buffers (by processing M indices in each step being stored in the same order as they enter the top tree). However, this way, the data locality is lost, i.e., two consecutive query indices will, in general, belong to different leaves. This, in turn, leads to arbitrary global memory accesses for the training patterns (i.e., no caching).

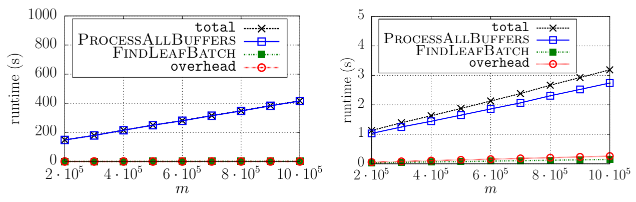
To investigate the importance of buffering, we compared the runtimes obtained *with* and *without* buffering, see Figure 3 (a). Obviously, buffering plays a crucial role in this context: Given arbitrary accesses to global memory of the GPU led to a significant drop in performance.



(a) (b)

Figure 3. The importance of buffering test queries is shown in (a). The runtimes for the different phases of ProcessAllBuffers are given in (b) (`psf_model_mag`, $n = 10^6$, $k = 10$, $h = 8$).

Overhead & Speed-Ups. The use of buffer k-d trees provides a separation of the overall workflow into two phases: (1) finding the leaves that need to be processed next and (2) updating the nearest neighbor candidates. In Figure 4, a comparison between `bufferkdtree(cpu)` and `bufferkdtree(gpu)` is given. Most of the runtime of `bufferkdtree(cpu)` was spent for processing the buffers (>99%), and `bufferkdtree(gpu)` yielded a significant speed-up (about 130) over its CPU analog. The speed-up was even larger for the ProcessAllBuffers phase (about 150), which demonstrates the efficiency of the corresponding kernel implementation. A deeper insight into the runtimes for this phase is provided in Figure 3 (b). It can be seen that the nearest neighbor search (Algorithm 4) took most of the time; all remaining steps (e.g., rearranging the test patterns in global memory, see Section 3.3.2) consumed less than 10% of the time (in particular, all memory operations between the host and the GPU took less than 1% of the overall execution).

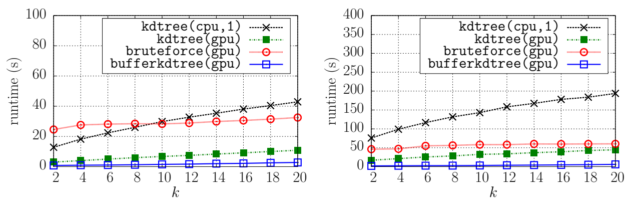


(a) (b)

Figure 4. Runtimes needed in the different processing phases for (a) `bufferkdtree(cpu)` and (b) `bufferkdtree(gpu)` given `psf_model_mag` ($n = 10^6$, $k = 10$, $h = 8$).

4.3.2. Problem-Dependent Parameters: k & n

Figure 5 shows runtimes for different k values: Increasing k led to more leaf visits/larger runtimes for all tree-based schemes; the decrease of performance, however, was similar (optimal tree heights were selected). The performance of `bruteforce(gpu)` was not significantly affected.



(a) (b)

Figure 5. Runtimes given varying k for (a) `psf_colors` and (b) `psf_model_mag` ($n, m = 10^6$, optimal tree heights h).

In Figure 6, runtime results for varying training set sizes n are shown. To compensate the increase of training patterns, we adapted the tree heights dynamically.[9] The performance of the brute-force scheme decreased significantly for increasing n, whereas all tree-based methods could compensate the increase of n much better.

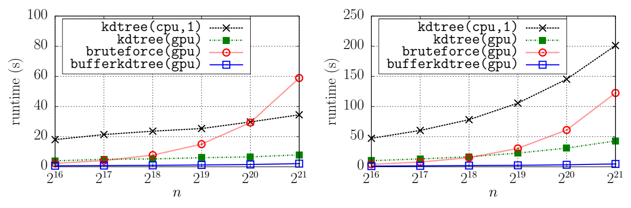
4.3.3. Competition: Multi-Core vs. GPU

A detailed runtime comparison is given in Table 2 for $m = 10^7$ test patterns (for `all`, two chunks of size $5 \cdot 10^6$ were

[9] Given $n = 2^i$ patterns, we used $h = 2^{i-12}$ and $h = 2^{i-6}$ for `bufferkdtree(gpu)` and the other schemes, respectively.

Table 2. Runtime comparison (in seconds) given $n = 2 \cdot 10^6$ training and $m = 10^7$ test patterns ($k = 10$; a $-$ indicates that the runtime was larger than 12 hours; the speed-up of `bufferkdtree(gpu)` over its competitors is given in brackets). For all tree-based methods, the *optimal tree heights* h were chosen in a preprocessing phase. The highlighted columns indicate feature space dimensions for which our approach has been developed (for $d < 5$ specialized solutions exist, and for $d \gg 20$, the tree-based approaches break down).

	psf_colors	psf_mag	psf_model_mag	all_mag	all_colors	all
	($d = 4$)	($d = 5$)	($d = 10$)	($d = 15$)	($d = 12$)	($d = 27$)
kdtree(cpu,1)	338 (×24)	259 (×22)	1965 (×55)	12459 (×59)	42314 (×89)	-
kdtree(cpu,2)	173 (×12)	130 (×11)	1012 (×28)	6478 (×31)	22108 (×46)	-
kdtree(cpu,3)	117 (× 8)	91 (× 8)	707 (×20)	5059 (×24)	17724 (×37)	-
kdtree(cpu,4)	92 (× 7)	70 (× 6)	584 (×16)	4639 (×22)	16770 (×35)	-
kdtree(cpu,8)	71 (× 5)	57 (× 5)	527 (×15)	4616 (×22)	16394 (×34)	-
bruteforce(gpu)	552 (×39)	664 (×55)	1160 (×32)	1595 (× 8)	1586 (× 3)	3047 (×2)
kdtree(gpu)	73 (× 5)	67 (× 6)	445 (×12)	3050 (×15)	10373 (×22)	-
bufferkdtree(gpu)	14	12	36	210	478	1717



(a) (b)

Figure 6. Increasing the number n of training patterns for (a) `psf_colors` and (b) `psf_model_mag` ($m = 10^6$, $k = 10$).

processed). Two main observations can be made: First, our `bufferkdtree(gpu)` scheme yielded a valuable speed-up for all data sets. This was in particular the case for `psf_model_mag` and `all_mag`. Here, speed-ups between 15 and 22 could be obtained compared to the corresponding multi-core implementation (`kdtree(cpu,8)`). Second, our approach was always superior to the (known to be a strong) `bruteforce(gpu)` scheme. Even in higher dimensional feature spaces, our approach still yielded practical benefits (where all other tree-based approaches did not yield any performance gain). Hence, our `bufferkdtree(gpu)` implementation always yielded significant performance gains over the multi-core k-d tree scheme and still provided benefits over `bruteforce(gpu)` for feature spaces up to $d = 27$.

4.3.4. Large-Scale Application

Our framework can basically handle an arbitrary amount of test patterns by processing chunks of queries.[10] As a final experiment, we applied a nearest neighbor model with $n = 2 \cdot 10^6$ training patterns to the whole SDSS catalog (DR 9) with a total amount of $m \approx 1178$ million test patterns (using `psf_model_mag` and $k = 10$). One core of the CPU was used to parse the data, while a second one and the GPU were used for `bufferkdtree(gpu)`. The test patterns were processed in chunks of size 10^7. The overall runtime needed to apply the model was 4639 seconds (about 39 seconds per chunk on average), which is in line with the results reported in Table 2. The memory consumption was dominated by the space needed to accommodate the training and test patterns as well as to keep track of the nearest neighbors (distances and indices). On the GPU, a maximum amount of 3GB was allocated during the execution of each chunk for this experiment (all resources were released after the completion of a single chunk).

5. Conclusions

We proposed the concept of a buffer k-d tree for efficient k-d tree-based nearest neighbor search on GPUs. The method is designed for processing huge amounts of queries in $\mathbb{R}^d$ with d larger than 4 and up to ≈ 25. The key idea is to reorganize the querying process such that queries belonging to the same leaf are processed in batches. As both the training and the query patterns reside consecutively in memory, the processing is much more amenable to an efficient GPU implementation. The principle of achieving "data locality" can be found in other fields such as memory-centric or I/O-efficient algorithms, but is new for GPU-based nearest neighbor search and may be also useful for other tree structures. The experimental results obtained on various astronomical data sets clearly demonstrate a significant performance gain over competing methods.

Acknowledgements. FG acknowledges support from the *German Academic Exchange Service* and CI from *The Danish Council for Independent Research* (*SkyML* project).

[10] Another way is to fill the `input` queue "on the fly" with new queries (potentially leading to even better runtimes at the cost of increased latency for certain queries). In this case, completely processed query patterns need to be removed from the GPU's global memory to release sufficient resources.

References

Aggarwal, A. and Vitter, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31 (9):1116–1127, 1988.

Ahn, C. P. et al. The tenth data release of the Sloan Digital Sky Survey: First spectroscopic data from the SDSS-III Apache Point Observatory galactic evolution experiment. *eprint arXiv:1307.7735*, 2013.

Aly, M., Munich, M., and Perona, P. Distributed kd-trees for retrieval from very large image collections. In *Proceedings of the 22nd British Machine Vision Conference*. BMVA Press, 2011.

Andoni, A. and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Beygelzimer, A., Kakade, S., and Langford, J. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 97–104. ACM, 2006.

Borne, K. D. Scientific data mining in astronomy. In *Next Generation of Data Mining*, pp. 91–114. Chapman and Hall/CRC, 2008.

Brodal, G. S. and Fagerberg, R. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pp. 426–438. Springer, 2002.

Bustos, B., Deussen, O., Hiller, S., and Keim, D. A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pp. 196–199. Springer, 2006.

Cayton, L. Accelerating nearest neighbor search on manycore systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 402–413. IEEE, 2012.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

Garcia, V., Debreuve, E., Nielsen, F., and Barlaud, M. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proceedings of the 17th IEEE International Conference on Image Processing*, pp. 3757–3760. IEEE, 2010.

Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning*. Springer, 2 edition, 2009.

Heinermann, J., Kramer, O., Polsterer, K. L., and Gieseke, F. On GPU-based nearest neighbor queries for large-scale photometric catalogs in astronomy. In *KI 2013: Advances in Artificial Intelligence*, volume 8077 of *Lecture Notes in Computer Science*, pp. 86–97. Springer, 2013.

Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. Interactive k-d tree GPU raytracing. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pp. 167–174. ACM, 2007.

Indyk, P. and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 604–613. ACM, 1998.

Ivezic, Z. et al. LSST: From science drivers to reference design and anticipated data products. *eprint arXiv:0805.2366*, 2011.

Jia, W., Shaw, K. A., and Martonosi, M. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 15–24. ACM, 2012.

Mortlock, D. J. et al. A luminous quasar at a redshift of z = 7.085. *Nature*, 474(7353):616–619, 2011.

Nakasato, N. Implementation of a parallel tree method on a GPU. *Journal of Computational Science*, 3(3):132–141, 2012.

Oancea, C. E., Mycroft, A., and Watt, S. M. A new approach to parallelising tracing algorithms. In *Proceedings of the 2009 International Symposium on Memory Management*, pp. 10–19. ACM, 2009.

Pan, J. and Manocha, D. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 211–220. ACM, 2011.

Polsterer, K. L., Zinn, P. C., and Gieseke, F. Finding new high-redshift quasars by asking the neighbours. *Monthly Notices of the Royal Astronomical Society*, 428(1):226–235, 2013.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.

Qiu, D., May, S., and Nüchter, A. GPU-accelerated nearest neighbor search for 3D registration. In *Proceedings of the 7th International Conference on Computer Vision Systems*, pp. 194–203. Springer, 2009.

Shuf, Y., Gupta, M., Franke, H., Appel, A., and Singh, J. Pal. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 13–25. ACM, 2002.

Sismanis, N., Pitsianis, N., and Sun, X. Parallel search of k-nearest neighbors with synchronous operations. In *IEEE Conference on High Performance Extreme Computing*, pp. 1–6. IEEE, 2012.

Stensbo-Smidt, K., Igel, C., Zirm, A., and Steenstrup Pedersen, K. Nearest neighbour regression outperforms model-based prediction of specific star formation rate. In *IEEE International Conference on Big Data 2013*, pp. 141–144. IEEE, 2013.

Wald, I. and Havran, V. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing*, pp. 61–69. IEEE, 2006.

Wang, W. and Cao, L. Parallel k-nearest neighbor search on graphics hardware. In *Proceedings of the 2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, pp. 291–294. IEEE, 2010.

Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, 2008.