

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.

<http://hdl.handle.net/2066/204499>

Please be advised that this information was generated on 2021-06-18 and may be subject to change.

Article 25fa pilot End User Agreement

This publication is distributed under the terms of Article 25fa of the Dutch Copyright Act (Auteurswet) with explicit consent by the author. Dutch law entitles the maker of a short scientific work funded either wholly or partially by Dutch public funds to make that work publicly available for no consideration following a reasonable period of time after the work was first published, provided that clear reference is made to the source of the first publication of the work.

This publication is distributed under The Association of Universities in the Netherlands (VSNU) 'Article 25fa implementation' pilot project. In this pilot research outputs of researchers employed by Dutch Universities that comply with the legal requirements of Article 25fa of the Dutch Copyright Act are distributed online and free of cost or other barriers in institutional repositories. Research outputs are distributed six months after their first online publication in the original published version and with proper attribution to the source of the original publication.

You are permitted to download and use the publication for personal purposes. All rights remain with the author(s) and/or copyrights owner(s) of this work. Any use of the publication other than authorised under this licence or copyright law is prohibited.

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please contact the Library through email: copyright@ubn.ru.nl, or send a letter to:

University Library
Radboud University
Copyright Information Point
PO Box 9100
6500 HA Nijmegen

You will be contacted as soon as possible.



Bigger Buffer k -d Trees on Multi-Many-Core Systems

Fabian Gieseke¹(✉), Cosmin Eugen Oancea¹, Ashish Mahabal²,
Christian Igel¹, and Tom Heskes³

¹ Department of Computer Science, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen, Denmark

{fabian.gieseke,cosmin.oancea,igel}@di.ku.dk

² Caltech Astronomy, Caltech, 1200 East California Blvd, Pasadena, CA 91125, USA
aam@astro.caltech.edu

³ Institute for Computing and Information Sciences, Radboud University Nijmegen,
Toernooiveld 212, 6525 EC Nijmegen, The Netherlands
t.heskes@cs.ru.nl

Abstract. A buffer k -d tree is a k -d tree variant for massively-parallel nearest neighbor search. While providing valuable speed-ups on modern many-core devices in case both a large number of reference and query points are given, buffer k -d trees are limited by the amount of points that can fit on a single device. In this work, we show how to modify the original data structure and the associated workflow to make the overall approach capable of dealing with massive data sets. We further provide a simple yet efficient way of using multiple devices given in a single workstation. The applicability of the modified framework is demonstrated in the context of astronomy, a field that is faced with huge amounts of data.

1 Motivation

Nearest neighbor search is a fundamental problem and an ingredient of many state-of-the-art data analysis techniques. While being a conceptually very simple task, the induced computations can quickly become a major bottleneck in the overall workflow when both a large reference and a large query set are given. In the literature, many techniques have been proposed that aim at accelerating the search. Typical ones are the use of spatial search structures, approximation schemes, and parallel implementations [1, 7, 11]. A recent trend in the field of big data analytics is the application of massively-parallel devices such as *graphics processing units* (GPUs) to speed up the involved computations. While such modern many-core devices can significantly reduce the practical runtime, obtaining speed-ups over standard CPU-based execution is often not straightforward and usually requires a careful adaptation of the sequential implementations.

Spatial search structures such as k -d trees are an established way to reduce the computational requirements induced by nearest neighbor search for spaces of moderate dimensionality (e.g., up to $d = 30$). A typical parallel k -d tree

based search assigns one thread to each query and all threads process the same tree *simultaneously*. Such an approach, however, is not suited for GPUs since each thread might induce a completely different tree traversal, which results in massive branch divergence and irregular accesses to the device’s memory.

Recently, we have proposed a modification of the classical k -d tree data structure, called *buffer k -d tree*, which aims at combining the benefits of both spatial search structures and massively-parallel devices [8]. The key idea is to assign an additional buffer to each leaf of the tree and to *delay* the processing of the queries reaching a leaf until enough work has been gathered. In that case, all queries stored in all buffers are processed together in a brute-force manner via the many-core device. While the framework achieves significant speed-ups on modern many-core devices over both a massively-parallel brute-force execution on GPUs as well as over a multi-threaded k -d tree based search running on multi-core systems, it is limited by the amount of reference and query points that fit on a GPU. In this work, we show how to remove this limitation by modifying the induced workflow to efficiently support huge reference and query point sets that are too large to be completely stored on the devices. This crucial modification renders buffer k -d trees capable of dealing with huge data sets.

2 Background

Modern many-core devices such as GPUs offer massive parallelism and can nowadays also be used for so-called *general-purpose computations* such as matrix-matrix multiplication. In contrast to standard CPU-based systems, GPUs rely on simplified control units and on a memory subsystem that does not attempt to provide the illusion of a uniform access cost to memory.

The main ingredients for an efficient many-core implementation are (1) exposing sufficient parallelism to fully utilize the device and (2) accessing the memory in an efficient way. The latter one includes techniques that

- (a) hide the latency of the memory transfer between host and GPU, for example by overlapping the kernel execution with the memory transfer, and
- (b) restructure the program in order to improve both spatial and temporal locality of reference to the global memory of the GPU.

Our implementation makes effective use of both ingredients.

2.1 Massively-Parallel Nearest Neighbor Computations

We address the problem of computing the $k \geq 1$ nearest neighbors of all points given in a query set $Q = \{\mathbf{q}_1, \dots, \mathbf{q}_m\} \subset \mathbb{R}^d$ w.r.t. all points provided in a reference set $P = \{\mathbf{x}_1, \dots, \mathbf{x}_n\} \subset \mathbb{R}^d$. Usually, the “closeness” between two points is defined via the Euclidean distance (which we will use), but other distance measures can be applied as well. Such nearest neighbor computations form the basis for a variety of methods both in data mining and machine learning including proximity-based outlier detection, classification, regression, density estimation,

and dimensionality reduction. The task of computing the induced distances (and keeping track of the list of neighbors per object) can be addressed naively in a brute-force manner spending $\mathcal{O}(nm \cdot (d + \log k))$ time, which quickly becomes computationally very demanding. Massively-parallel computing can significantly reduce the runtime in this case as shown by Garcia *et al.* [7]. Still, for large query and reference sets, the computational requirements can become very large.

Various other approaches have been proposed in the literature that aim at taking advantage of the computational resources provided by GPUs in combination with other techniques [11, 14]. The focus of this work is on massively-parallel processing of k -d trees. While several implementations have been proposed that address such traversals from a more general perspective (e.g., in the context of *ray tracing*) [10], these approaches are not suited for nearest neighbor search in moderate-sized feature spaces (i.e., $d > 3$), except for the recently proposed buffer k -d tree extension [8].

2.2 Nearest Neighbor Search via k -d Trees

Spatial search structures such as k -d trees can be used to speed up nearest neighbor search. K -d trees can be constructed as follows [1, 6]: For a given point set P , a k -d tree is a binary tree with the root corresponding to P . The children of the root are obtained by splitting the point set into two (almost) equal-sized subsets, which are processed recursively. In their original form, k -d trees are obtained by resorting to the median values in dimension $i \bmod d$ to split a point set corresponding to a node v at level i (starting with the root at level 0).¹ The recursive process stops as soon as a predefined number of points are left in a subset. The k -d tree stores the splitting values in its internal nodes; the points corresponding to the remaining sets are stored in the leaves.

The tree structure can be used to accelerate nearest neighbor search: Let $\mathbf{q} \in \mathbb{R}^d$ be a query point. For the sake of exposition, we focus on $k = 1$ (the case of $k > 1$ neighbors works similarly). The overall search takes place in two phases. In the first one, the tree is traversed from top to bottom to find the d -dimensional cell (induced by the tree/splitting process) that contains the query point \mathbf{q} . Going down the tree can be conducted efficiently using the median values stored in the internal nodes of the k -d tree. In the second phase, the tree is traversed bottom-up, and on the way back to the root, subtrees are checked in case the query point is close to the corresponding splitting hyperplane. If the distance of the query point \mathbf{q} to the hyperplane is less than the distance to the current nearest neighbor candidate, then the subtree is checked for better candidates (recursively). Otherwise, the whole subtree can be safely pruned (no recursion). Once the root is reached twice, the overall process stops and the final nearest neighbor candidate is returned.

Given a low-dimensional search space, it is usually sufficient to process a relatively small number of leaves, which results in a logarithmic runtime behavior (i.e., $\mathcal{O}(\log n)$ time per query in practice).

¹ Other splitting rules might be applied (e.g., according to the “longest” side).

Algorithm 1. LAZYSEARCH [8]

Require: A chunk $Q = \{\mathbf{q}_1, \dots, \mathbf{q}_m\} \subset \mathbb{R}^d$ of query points.

Ensure: The $k \geq 1$ nearest neighbors for each query point.

1: Construct buffer k -d tree \mathcal{T} for $P = \{\mathbf{x}_1, \dots, \mathbf{x}_n\} \subset \mathbb{R}^d$.

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, \dots, i_M from **reinsert** and **input**.

5: $r_1, \dots, r_M = \text{FINDLEAFBATCH}(i_1, \dots, i_M)$

6: **for** $j = 1, \dots, M$ **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: $l_1, \dots, l_N = \text{PROCESSALLBUFFERS}()$

13: Insert l_1, \dots, l_N into **reinsert**.

14: **end if**

15: **end while**

16: **return** list of k nearest neighbors for each query point.

2.3 Revisited: Buffer k -d Trees

A standard multi-threaded kd-tree based traversal assigns one thread to each query. For GPUs, such an approach is not suited since each thread might induce a completely different path, which significantly shortens their computational benefits. The main idea of the buffer k -d tree extension is to delay the processing of the queries by buffering similar patterns prior to their common processing [8]. The reorganized workflow is based on buffer k -d trees, which are sketched next, followed by a description of the buffered nearest neighbor search.

A buffer k -d tree consists of (1) a top tree, (2) a leaf structure, (3) a set of buffers, and (4) two input queues that store the queries, see Fig. 1. The top tree corresponds to a classical k -d tree with its splitting values (e.g., medians) laid out in memory in a pointer-less manner. The leaf structure stores the point sets that stem from the splitting process in a

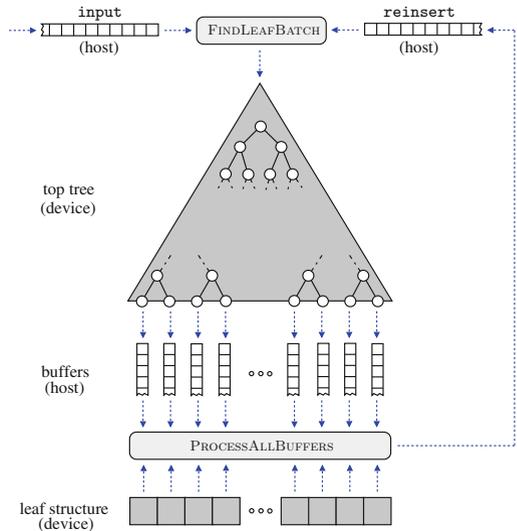


Fig. 1. A buffer k -d tree [8]: The gray elements are stored on the GPU.

the leaf structure stores the point sets that stem from the splitting process in a

consecutive manner. In addition, a buffer is attached to each leaf of the top tree that can store B query indices (e.g., $B = 1024$). The input queues are used to store the input query indices and the query indices that need further processing after a `ProcessAllBuffers` call.

A buffer k -d tree can be used to delay the processing of the queries by performing several iterations, see Algorithm 1: In each iteration, the procedure `FINDLEAFBATCH` retrieves indices from both the `input` and `reinsert` queue and propagates them through the top tree. The indices, which are stored in the corresponding buffers, are processed in chunks via the procedure `ProcessAllBuffers` once the buffers get full. All indices that need further processing (i.e., their implicit tree traversal has not reached the root twice) are inserted into `reinsert` again. Thus, in each iteration, one (1) finds the leaves that need to be processed next and (2) updates the queries' nearest neighbors.

While the first phase is not well-suited for massively-parallel processing, the second one is and, since it constitutes the most significant part of the runtime, yields valuable overall speed-ups. The main advantage of the reorganized workflow is that all queries are processed in the same block-wide SIMD instruction and exhibit either good spatial or temporal locality of reference, i.e., coalesced or cached global memory accesses. For details of the many-core implementation of the procedure `PROCESSALLBUFFERS`, we refer to Gieseke *et al.* [8].

3 Processing Bigger Trees

Both the top tree and the leaf structure need to be stored on the GPU—this limits the amount of reference patterns that can be processed. We now describe modifications that allow buffer k -d trees to scale to massive data sets.

3.1 Construction Phase

As shown below, one can basically process arbitrarily large query sets by considering chunks of data points. Dealing with huge reference sets, however, is more difficult: Since the top tree and the full leaf structure (that stores the rearranged reference points) have to be made available to all threads during the execution of `PROCESSALLBUFFERS`, one cannot directly split up the leaf structure. However, as explained next, one can avoid storing the leaf structure in its full entirety on the many-core device without significantly increasing the overall runtime.

We start by focusing on the space needed for the top tree: From a practical perspective, a small top tree is usually advantageous compared to the full tree used by a classical k -d tree-based search, since the efficiency gain on GPUs stems from processing “big” leaves. For instance, given a reference set $P \subset \mathbb{R}^{10}$ with two million points, top trees of height $h = 8$ or $h = 9$ are usually optimal [8]. Further, since only median values are stored in the top tree, the space consumption is negligible even given much bigger top trees. In addition, the top tree can be built efficiently via linear-time median finding [2], which results in $\mathcal{O}(h \log n)$ time for

the whole construction phase.² Hence, one can (1) build the top tree efficiently on the host system and (2) store it in its full entirety on the GPU.

The main space bottleneck stems from the leaf structure. For instance, one billion points in \mathbb{R}^{15} occupy about 60 GB of space—too much for a modern many-core device. For this reason, we do *not* copy the leaf structure from the host to the device after the construction of the top tree. Instead, we allocate space for two chunk buffers of fixed size on the many-core device. These chunk buffers will be used to overlap the execution of the PROCESSALLBUFFERS procedure with the host-to-device memory transfer for the next chunk.³ Note that we also allocate two associated memory buffers on the host (*pinned memory* [13]) to achieve efficient concurrent compute and copy operations. The overall memory layout is shown in Fig. 2, where the memory that is actually allocated on the device is sketched via gray rectangles.

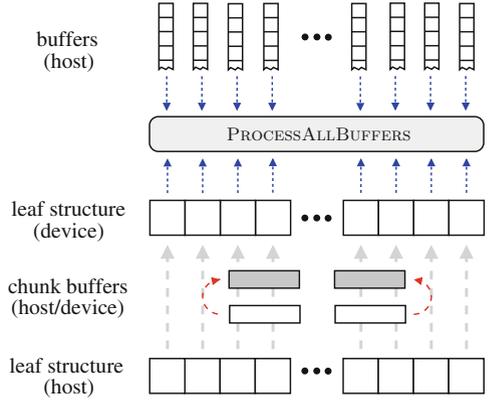


Fig. 2. Adapted memory layout

3.2 Query Phase

We now describe the details of the modified querying process.

Processing the Leaf Structure: In each iteration of Algorithm 1, the procedure PROCESSALLBUFFERS is invoked to retrieve all query indices from the leaf buffers. The queries are processed in a massively-parallel fashion, where each thread compares a particular query with all reference points stored in the associated leaf. Given the modified memory layout, one can now process the leaves in chunks in the following way: The leaf structure containing all n rearranged reference points is split into $1 < N \ll n$ chunks C_1, \dots, C_N (e.g., $N = 10$). Each chunk C_j contains the points of the leaf structure at positions $k = C_j^L, \dots, C_j^R$, where $C_j^L = \lceil \frac{(j-1) \cdot n}{N} \rceil$ and $C_j^R = \lceil \frac{j \cdot n}{N} \rceil$. A buffer attached to the top tree corresponds to a leaf in the leaf structure with leaf bounds $0 \leq l_i < r_i \leq n - 1$. All queries removed from the buffers are then processed in N iterations and a query i with leaf bounds l_i and r_i is processed in iteration $j \in \{1, \dots, N\}$ if $[l_i, r_i] \cap [C_j^L, C_j^R] \neq \emptyset$, i.e., if the leaf bounds overlap with chunk C_j .

² For a tree of height $h = 20$, suitable for more than one billion points, less than ten megabytes are needed. Note that the space for the buffers (e.g., of size 128 each) stored on the host does usually not cause any problems (e.g., less than a gigabyte).

³ Note that we allocate space for two chunk *buffers* per device; the actual number N of chunks being considered is usually larger (e.g., $N = 10$).

The chunks are processed sequentially C_1, C_2, \dots, C_N . The data needed for each chunk is copied from the host to one of the two chunk buffers allocated on the device prior to conducting the brute-force computations, see Fig. 2. To hide the overhead induced for these copy operations, the copy process for the next chunk is started as soon as the computations for its predecessor have been invoked. In particular, the processing of a chunk C_j takes place in three phases:

- (1) *Brute*: First, the massively-parallel brute-force nearest neighbor computations are invoked (non-blocking kernel call). The data needed for these computations (chunk C_j) have been copied in round $j - 1$ (for C_0 , the data is either available from an initial copy operation or from the previous round).
- (2) *Copy*: While the brute-force computations are conducted by the GPU, the data for the next chunk are copied from host to the currently unused device buffer.⁴ Note that copy operations on the host system have to be conducted as well to ensure that the correct part of the leaf structure is moved to the appropriate pinned memory buffer (which is then copied to the device).
- (3) *Wait*: In the final phase, one simply waits for the kernel invoked in the first phase to finish its computations (blocking call).

The iterative compute-and-copy processing of the chunks can be implemented via two (OpenCL) command queues [13]: For the first chunk, phase (1) and (3) are instantiated via command queue A, whereas the copy process (2) is instantiated via command queue B (non-blocking for both (1) and (2)). For the second chunk, phases (1) and (3) are instantiated via command queue B and (2) via command queue A. This process continues until all chunks have been processed. In essence, the use of two command queues allows the copy phase (2) to run in parallel with the brute-force computation phases (1) and (3).

Multi-Many-Core Querying: Assuming that a fixed amount of memory is available on the many-core device to store the query patterns and the results at all times (e.g., one gigabyte), one can process an arbitrarily large query set by removing fully processed indices and by adding new indices on-the-fly, see Algorithm 1. An even simpler approach is to split up the queries into chunks and to handle these chunks independently. One drawback of the latter approach could be the overhead induced by applying the procedure `ProcessAllBuffers` in case the buffers are not sufficiently filled (which usually takes place as soon as no queries are available anymore). However, given relatively large chunks, the induced overhead is very small as shown in our experimental evaluation.

In a similar fashion, one can make use of multiple many-core devices by splitting all queries into “big” chunks according to the devices that are available. These chunks, which might have to be split into smaller chunks as described above, can be processed independently from each other.

⁴ For $j = N$, the data for chunk 0 are copied from host to the corresponding buffer on the device for next round (i.e., next call of `PROCESSALLBUFFERS`).

4 Experiments

The purpose of the experiments provided below is to analyze the efficiency of the modified workflow and to sketch the potential of the overall approach in the context of large-scale scenarios. For a detailed experimental comparison including an analysis of the different processing phases and the influence of parameters related to the buffer k -d tree framework, we refer to our previous work [8].

4.1 Experimental Setup

All runtime experiments were conducted on a standard desktop computer with an Intel(R) Core(TM) i7-4790K CPU running at 4.00 GHz (4 cores; 8 hardware threads), 32 GB RAM, and two Nvidia GeForce Titan Z GPUs (each consisting of two devices with 2880 shader units and 6 GB main memory) using single precision. The operating system was Ubuntu 14.4.3 LTS (64 Bit) with kernel 3.13.0-52, CUDA 7.0.65 (graphics driver 340.76), and OpenCL 1.2. All algorithms were implemented in C and OpenCL, where Swig was used to obtain appropriate Python interfaces.⁵ The code was compiled using gcc-4.8.4 at optimization level -O3.

For the experimental evaluation, we report runtimes for both the construction and the query phase (referred to as “train” and “test” phases), where the focus is on the latter one (that makes use of the GPUs). We consider the following three implementations:

- (1) `bufferkdtree(i)`: The adapted buffer k -d tree implementation with both `FindLeafBatch` and `ProcessAllBuffers` being conducted on i GPUs.
- (2) `kdtree(i)`: A multi-core implementation of a k -d tree-based search, which runs i threads in parallel on the CPU (each handling a single query).
- (3) `brute(i)`: A brute-force implementation that makes use of i GPUs to process the queries in a massively-parallel manner.

The parameters for the buffer k -d tree implementation were fixed to appropriate values.⁶ Note that both competitors of `bufferkdtree` have been evaluated extensively in the literature; the reported runtimes and speed-ups can thus be put in a broad context. For simplicity, we fix the number k of nearest neighbors to $k = 10$ for all experiments.

We focus on several data-intensive tasks from the field of astronomy. Note that a similar runtime behavior can be observed on data sets from other domains as well as long as the dimensionality of the search space is moderate (e.g., from $d = 5$ to $d = 30$). We follow our previous work and consider the `psf_mag`, `psf_model_mag`, and `all_mag` data sets of dimensionality $d = 5$, $d = 10$, and $d = 15$, respectively; for a description, we refer to Gieseke *et al.* [8]. In addition, we consider a new dataset derived from the *Catalina Realtime Transient Survey*

⁵ The code is publicly available under <https://github.com/gieseke/bufferkdtree>.

⁶ For a tree of height h , we fixed $B = 2^{24-h}$ and the number M of indices fetched from `input` and `reinsert` in each iteration of Algorithm 1 to $M = 10 \cdot B$.

(*crts*) [4,9]. This survey contains tens to hundreds of observations for more than 500 million sources over a large part of the sky. The resulting light-curves (time-series of light received as a function of time) are used to derive several statistical features.⁷ We use ten such features on a set of 30 million light-curves for the experiments described here.

4.2 Modified Workflow

While the modified workflow permits the use of buffer k -d trees for massive data sets that do not fit in the memories of the many-core devices, it might also induce a certain overhead compared to its original version due to the induced copy operations and reduced workload per kernel call.

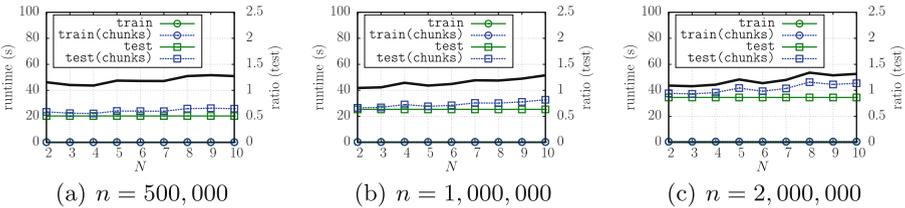


Fig. 3. Comparison of the training and testing times between the original workflow (*train* and *test*) and the modified workflow that processes the leaves in chunks (*train(chunks)* and *test(chunks)*). A varying number N of chunks and training points n are considered using the *psf_model_mag* data set. The number m of test patterns and the tree height h are fixed to $m = 10,000,000$ and $h = 9$, respectively. The ratio between *test* and *test(chunks)* is shown as black, thick line (a value close to 1 indicates a small overhead caused by the chunked processing).

Processing Leaves in Chunks: The main modification is the different processing of the leaf structure in case it does not fit in the device’s memory. To evaluate the potential overhead caused by the additional copy operations (between host and device) during the execution of *PROCESSALLBUFFERS*, we consider data set instances that still fit in memory and compare the runtimes of (1) the original workflow with (2) the workflow that is based on multiple chunks.

In Fig. 3, the outcome of this comparison is shown for a varying number N of chunks and a varying number n of training points. In all three cases, the number of test patterns is fixed to $m = 10,000,000$. Further, the tree height is set to $h = 9$ and a single GPU is used (i.e., *bufferkdtree(1)*). Two observations can be made: First, the training time is very small compared to the test time for all cases (even though the tree is constructed sequentially on the host). Second, the performance loss induced by the chunked processing is very small for almost any number N of chunks (i.e., the ratio shown as black, thick line is close to 1). In

⁷ In particular, we make use of the *amplitude*, *Stetson_j*, *Stetson_k*, *Skew*, *fpr_{mid35}*, *fpr_{mid50}*, *fpr_{mid65}*, *fpr_{mid80}*, *shov*, *maxdiff* [5].

particular, this is the case for smaller values of N ; using more chunks naturally yields more overhead due to, e.g., smaller leaf blocks being processed per kernel call. This overhead, however, decreases again if one increases the number of training and test patterns.

Thus, the runtimes of the new, chunked workflow are close to the one of the original approach—indicating that the overlapping compute-and-copy process successfully hides the additional overhead for the copy operations.

Multi-Many-Core Processing: As outlined above, one can simply distribute the test queries to multiple devices to take advantage of the additional computational resources. Again, this can lead to a certain overhead, since invoking the procedure `PROCESSALLBUFFERS` becomes less efficient at the end of the overall processing (and this happens earlier in case the test queries are split into chunks). Similarly to the experiment provided above, we compare the efficiency of `bufferkdtree(4)` with the one of `bufferkdtree(1)`, a standard single-device processing with all test patterns fitting on the GPU. For this sake, we consider $N = 1$ leaf chunks (i.e., no modified processing of the leaves), $n = 2 \cdot 10^6$ training patterns, and vary the number m of test patterns.

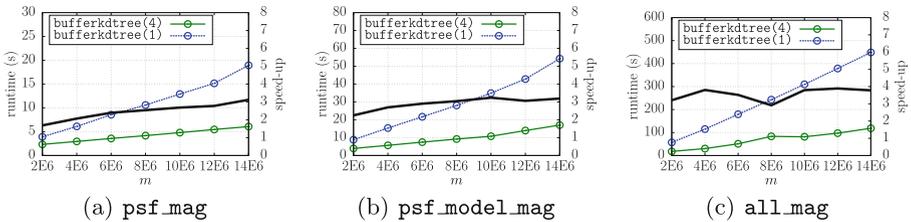


Fig. 4. Runtime comparison for the test phase between `bufferkdtree(1)` and `bufferkdtree(4)`, where the test queries are distributed uniformly among the devices for the latter one. The speed-up is shown as black, thick line (maximum 4).

The outcome of this experiment for three different data sets is shown in Fig. 4: It can be seen that a suboptimal speed-up of about 2 is achieved in case a relatively small amount of test patterns is processed. However, as soon as the number of test patterns increases, the speed-up gets closer to 4, which depicts the maximum that can be achieved. Hence, the naive way of using all devices in a given workstation does not yield significant drawbacks as soon as large-scale scenarios are considered, which is the scope of this work.

4.3 Large-Scale Applications

To demonstrate the potential of the modified framework, we consider two large-scale tasks: (1) the application of nearest neighbor models that are based on very large training sets and (2) large-scale density-based outlier detection.

Huge Nearest Neighbor Models: The first scenario addresses nearest neighbor models that are based on very large training sets. Such models have been successfully applied for various tasks in astronomy including the detection of distant galaxies or the estimation of physical parameters [12].

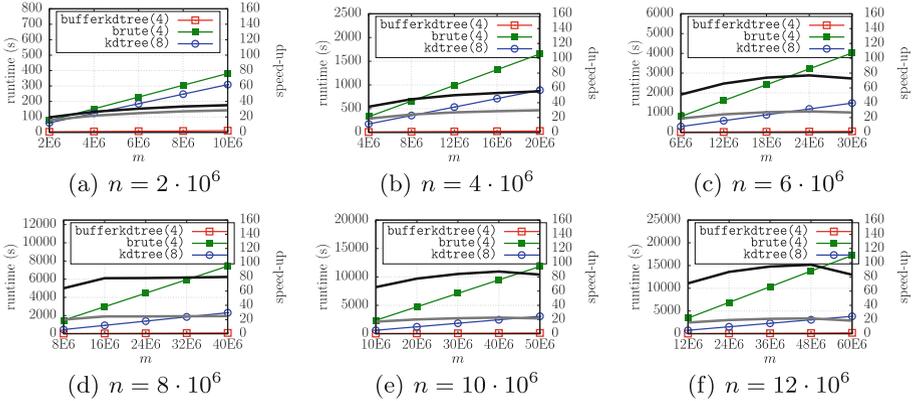


Fig. 5. Runtime comparison for a varying number n of training patterns given the `psf_model_mag` data set. The speed-ups of `bufferkdtree` over `brute` and `kdtree` are shown as thick black and gray lines, respectively. In each case, $m = n, \dots, 5n$ test patterns are considered for the `bufferkdtree` implementation; for both `kdtree` and `brute`, only $m = 10^6$ were processed to obtain runtime estimates (which are plotted).

For the experimental comparison we consider scenarios with both a large amount of training and test patterns. More precisely, we consider up to $n = 12 \cdot 10^6$ training points and up to $m = 5 \cdot n$ test points given the `psf_model_mag` data set. For both tree-based methods, appropriate tree depths are set beforehand (i.e., optimal ones w.r.t. the runtime needed in the test phase). The outcome of this comparison is shown in Fig. 5: It can be seen that valuable speed-ups can be achieved over both competitors. Further, the speed-ups generally become more significant the more patterns are processed. Note that for Fig. (e) and (f), the number N of chunks is (automatically) set to $N = 3$ for the `bufferkdtree` implementation due to the training patterns exceeding the space reserved for them on the device. This results in a slightly worse performance for $m > 30 \cdot 10^6$.

Large-Scale Proximity-Based Outlier Detection: As final use case, we consider large-scale proximity-based outlier detection. Various outlier scores have been proposed that are based on the computation of nearest neighbors. A typical one is to rank the points according to their average distance according to their k nearest neighbors, see, e.g., Tan *et al.* [15]. Such techniques depict very promising tools in case many reference points are given in a moderate-sized feature space, which is precisely the case for many tasks in astronomy. Typically, these scores

require the computation of the nearest neighbors for each of the reference points, which can quickly become very time-consuming.

To show the potential of our many-core implementation, we consider the `crt`s data set described above (with $d = 10$ features) and vary the number n of reference points (here, we have $n = m$ for the full data set). We again compare the performances of all three competitors, where we consider both the runtime for the construction and the one for the query phase. The outcome is shown in Fig. 6. Note that the runtimes for both `kdtree(8)` and `brute(4)` depict estimates based on a reduced query due to the computational complexity (i.e., up to $n = 30 \cdot 10^6$ reference points and a fixed query set of size $\hat{m} = 1,000,000$ are considered; the runtime estimates w.r.t. the full data set instances are plotted). For the `bufferkdtree(4)` implementation, we fix $N = 3$. It can be seen that the buffer k -d tree implementation yields valuable speed-ups and can successfully process the whole data set in a reasonable amount of time.

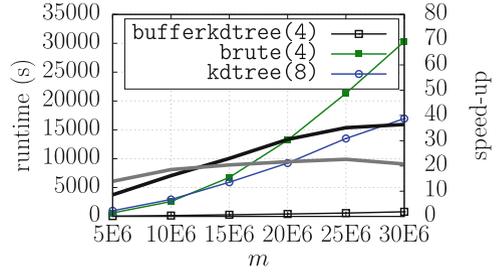


Fig. 6. Large-scale outlier detection

5 Conclusions and Outlook

We showed how to modify and extend buffer k -d trees such that the overall workflow is capable of dealing with both massive training and massive testing tests. In addition, we provided a simple way of making use of multiple many-core devices that are often given in modern workstations. We further demonstrated the applicability of our modifications in the context of astronomy.

In future, it would be interesting to extend our results to other spatial search structures such as random projection trees [3] and distributed computing environments. We also expect our implementation to be applicable and useful for a variety of other application domains as well. We plan to extend our work along these lines of research.

Acknowledgements. The authors would like to thank the *Radboud Excellence Initiative* of the Radboud University Nijmegen (FG), *NVIDIA* for generous hardware donations (FG), the *Danish Industry Foundation* through the *Industrial Data Analysis Service* (FG, CI, CO), the *The Danish Council for Independent Research | Natural Sciences* through the project *Surveying the sky using machine learning* (CI), and ACP, IUCAA, IUSSTF, and NSF (AM).

References

1. Bentley, J.: Multidimensional binary search trees used for associative searching. *Commun. ACM* **18**(9), 509–517 (1975)
2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. *J. Comput. Syst. Sci.* **7**(4), 448–461 (1973)
3. Dasgupta, S., Freund, Y.: Random projection trees and low dimensional manifolds. In: *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, pp. 537–546. ACM, New York (2008)
4. Djorgovski, S.G., et al.: The catalina real-time transient survey. In: *Monitoring Variable X-Ray Sources, The First Year of MAXI* (2011)
5. Faraway, J., Mahabal, A., Sun, J., Wang, X., Yi, W., Zhang, L.: Modeling light curves for improved classification. eprint [arXiv:1401.3211](https://arxiv.org/abs/1401.3211) (2014)
6. Friedman, J., Bentley, J., Finkel, R.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.* **3**(3), 209–226 (1977)
7. Garcia, V., Debreuve, E., Nielsen, F., 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)
8. Gieseke, F., Heinermann, J., Oancea, C., Igel, C.: Buffer k-d trees: processing massive nearest neighbor queries on GPUs. In: *Proceedings of the 31st International Conference on Machine Learning. JMLR W&CP*, vol. 32, pp. 172–180 (2014)
9. Mahabal, A.A., et al.: Discovery, classification, and scientific exploration of transient events from the catalina real-time transient survey. *Bull. Astronomical Soc. India* **39**(3), 387–408 (2011)
10. Nakasato, N.: Implementation of a parallel tree method on a GPU. *J. Comput. Sci.* **3**(3), 132–141 (2012)
11. Pan, J., 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)
12. Polsterer, K., Zinn, P., Gieseke, F.: Finding new high-redshift quasars by asking the neighbours. *Monthly Not. R. Astron. Soc.* **428**(1), 226–235 (2013)
13. Scarpino, M.: *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning, New York (2012)
14. Sismanis, N., Pitsianis, N., Sun, X.: Parallel search of k-nearest neighbors with synchronous operations. In: *IEEE Conference on High Performance Extreme Computing*, pp. 1–6. IEEE (2012)
15. Tan, P.N., Steinbach, M., Kumar, V.: *Introduction to Data Mining*. Addison-Wesley, Boston (2005)