bufferkdtree Documentation

Release 1.3

Fabian Gieseke

November 11, 2016

Contents

The bufferkdtree library is a Python library that aims at accelerating nearest neighbor computations using both k-d trees and graphics processing cards (GPUs) via OpenCL. The source code is published under the GNU General Public License Version 2 (GPLv2).

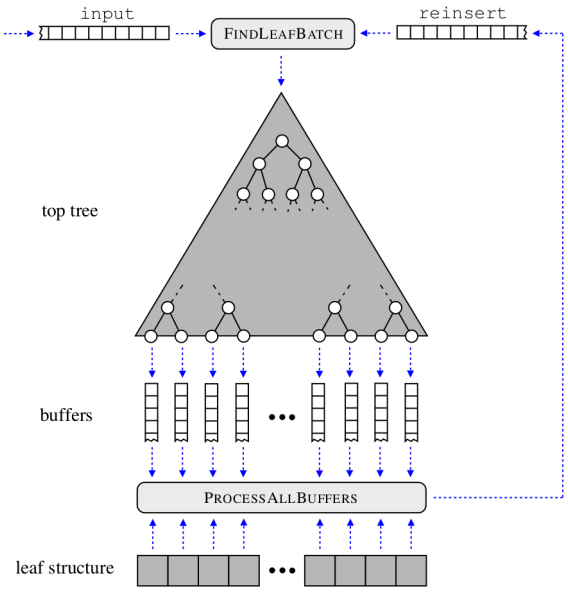
CHAPTER 1

Contents

1.1 Quick Overview

The main approach provided by the bufferkdtree package is an efficient many-core implementation (suitable for, e.g., GPUs) for processing huge amounts of nearest neighor queries by means of so-called *buffer k-d trees*. Such trees depict modifications of standard k-d trees that can be used to make use of the massive parallelism provided by today's many-core devices (such as GPUs).

Buffer k-d trees aim at scenarios, where you are given both a large reference (e.g., 1,000,000 points) and a huge query set (e.g., 10,000,000 points or more) with an input space of moderate dimensionality (e.g., from d=5 to d=25 dimensions). The general workflow is sketched below; the key idea is to "delay" the processing of nearest neighbor queries until enough work is gathered that can be processed by the many-core device.



Workflow: Initially, all queries are given in the input queue. The computation of nearest neighbors takes place in iterations. 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. For each query (index), an associated stack is stored, which is used to traverse the overall tree.

Note that each query is traversed in the same manner as for a standard k-d tree traversal (given the same tree depth). However, in contrast to the original traversal, queries are now grouped together before the nearest neighbors are

updated in each leaf. This greatly improves the performance on today's many-core devices, since similar memory regions are processed by neighbored threads.

A detailed description of the techniques used and an experimental evaluation of the implementation using massive astronomical data sets are provided in:

Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. *Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs*. In: Proceedings of the 31st International Conference on Machine Learning (ICML) 32(1), 2014, 172-180. [pdf] [bibtex]

Implicit Hardware Caches

The brute-force step that takes place to empty the leaves via the many-core device makes use of implicit hardware caches. To achieve satisfying speed-ups, this feature has to be supported by the device (see, e.g., the Kepler GK110 Whitepaper)

1.2 Installation

Warning: The authors are not responsible for any implications that stem from the use of this software.

1.2.1 Quick Installation

The package is available on PyPI, but can also be installed from the sources. We recommend to use virtualenv to install the package and all dependencies (see below). The installation process has been tested on various Linux-based machines: To install the package via PyPI on Linux systems, type:

```
$ sudo pip install bufferkdtree
```

To install the package from the sources, first get the current stable release via:

```
$ git clone https://github.com/gieseke/bufferkdtree.git
```

Subsequently, install the package locally via:

```
$ cd bufferkdtree
$ python setup.py install --user
```

or, globally for all users, via:

```
$ sudo python setup.py build
$ sudo python setup.py install
```

1.2.2 Dependencies

The bufferkdtree package has been tested under various Linux-based systems such as Ubuntu and OpenSUSE and requires Python 2.6/2.7/3.*. Below, some installation instructions are given for Linux-based systems; similar steps have to be conducted on other systems.

To install the package, a working C/C++ compiler, OpenCL (version 1.2 or higher), Swig, and the Python development files (headers) along with setuptools need to be available. Further, the NumPy package (>=1.11.0) is needed.

On Ubuntu 12.04/14.04/16.04, for instance, the following command can be used to install most dependencies (except for OpenCL):

```
$ sudo apt-get install python2.7 python-dev swig build-essential python-numpy python-setuptools
```

On an OpenSUSE system, the corresponding command is:

```
$ sudo zypper install python python-devel swig python-numpy python-setuptools
```

Compatibility

The implementation is based on the efficient use of implicit hardware caches. Thus, to obtain good speed-ups, the system's GPU has to support this feature! Current architectures such as Nvidia's Kepler architecture exhibit such caches, see, e.g., the Kepler GK110 Whitepaper.

1.2.3 OpenCL

OpenCL (version 1.2 or higher) needs to be installed correctly on the system. The installation of OpenCL depends on the particular host system and the devices used, see, e.g.,

- Intel
- Nvidia
- AMD

We refer to Andreas Klöckner's wiki page for an excellent description of the OpenCL installation process on Linux-based systems. OpenCL is installed on macOS. For Windows, we refer to this blog post.

Please make sure that the the OpenCL header files are available as well and accessible during the installation process, e.g., by setting the C_INCLUDE_PATH environment variable in the .bashrc file on Linux-based systems. For instance, given a Nvidia device alogn with CUDA and OpenCL being installed, the header files are probably located in `/usr/local/cuda/include`. Hence, the following command would update the environment variable accordingly (if needed):

```
export C_INCLUDE_PATH=$C_INCLUDE_PATH:/usr/local/cuda/include
```

Sometimes, one also needs to set the path to the OpenCL libraries (e.g., linking errors such as /usr/bin/ld: cannot find -lOpenCL). On Linux-based systems, adapting the library path usually fixes such problems:

```
export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/cuda/lib64
```

No OpenCL Support?

In case OpenCL is not supported on the system, one can still compile **standard k-d tree implementation** by setting the global variable *BUFFERKDTREE_KDTREE_ONLY* to *True*. On Linux-based systems, this can be achieved via

export BUFFERKDTREE_KDTREE_ONLY=True

1.2.4 Virtualenv & Pip

As for most Python packages, we recommend to make use of virtualenv to install the package. To install virtualenv on recent Debian/Ubuntu-based systems, the following commands can be used to install virtualenv and pip:

```
$ sudo apt-get install python-virtualenv python-pip
```

Afterwards, a new virtual environment can be created to install the Numpy and the bufferkdtree package:

```
$ mkdir ~/.virtualenvs
$ cd ~/.virtualenvs
$ virtualenv bufferkdtree
$ source bufferkdtree/bin/activate
$ pip install numpy==1.6.1
$ pip install bufferkdtree
```

1.3 Examples

The following two examples sketch the use of the different implementations and can both be found in the *examples* subdirectory of the bufferkdtree package.

1.3.1 Toy Example

```
#

"""
Nearest Neighbors and Artificial Data
=====================================

This example demonstrates the use of the different
implementations given on a small artifical data set.

Note: The platform and the device needs to be specified below
(via the parameter 'plat_dev_ids').
"""
print(__doc__)

import numpy
from bufferkdtree import NearestNeighbors

n_neighbors = 10
plat_dev_ids = {0:[0]}
n_jobs = 1
verbose = 1
```

All implementations are provided via the `NearestNeighbors` class, which exhibits a similar layout as the corresponding class of the scikit-learn package. The parameter `n_jobs` determines the number of threads that shall be used by the standard k-d tree implementation (CPU). The parameter `plat_dev_ids` determines the OpenCL devices that shall be used by the buffer k-d tree implementation (OpenCL): Each key of the dictionary corresponds to a OpenCL platform id and for each platform id, a list of associated device ids can be provided. For instance, the first platform (with id 0) and its first device (with id 0) is used for the current example.

Next, a small artificial data set is generated, where `X` contains the points, one row per point:

```
X = numpy.random.uniform(low=-1, high=1, size=(10000,10))
```

The package provides three implementations (`brute`, `kd_tree`, or `buffer_kd_tree`), which can be invoked via the `algorithm` keyword of the constructor:

```
# (1) apply buffer k-d tree implementation
nbrs_buffer_kd_tree = NearestNeighbors(algorithm="buffer_kd_tree",
                        tree_depth=9,
                        plat_dev_ids=plat_dev_ids,
                        verbose=verbose)
nbrs_buffer_kd_tree.fit(X)
dists, inds = nbrs_buffer_kd_tree.kneighbors(X, n_neighbors=n_neighbors)
print("\nbuffer_kd_tree output\n" + str(dists[0]))

# (2) apply brute-force implementation
nbrs_brute = NearestNeighbors(algorithm="brute",
                        plat_dev_ids=plat_dev_ids,
                        verbose=verbose)
nbrs_brute.fit(X)
dists, inds = nbrs_brute.kneighbors(X, n_neighbors=n_neighbors)
print("\nbrute output\n" + str(dists[0]))

# (3) apply k-d tree mplementation
nbrs_kd_tree = NearestNeighbors(algorithm="kd_tree",
                        n_jobs=n_jobs,
                        verbose=verbose)
nbrs_kd_tree.fit(X)
dists, inds = nbrs_kd_tree.kneighbors(X, n_neighbors=n_neighbors)
print("\nkd_tree output\n" + str(dists[0]) + "\n")
```

For a detailed description of the remaining keywords, see the description of the *documentation* of the NearestNeighbors class. The above steps yield the following output:

```
Nearest Neighbors
=================

This example demonstrates the use of the different
implementations given on a small artifical data set.


buffer_kd_tree output
[ 0.          1.0035212   1.09866345  1.11734533  1.13440645  1.17730558
  1.1844281   1.20736992  1.2085104   1.21593559]

brute output
[ 0.          1.0035212   1.09866357  1.11734521  1.13440645  1.17730546
  1.18442798  1.20736992  1.2085104   1.21593571]

kd_tree output
[ 0.          1.0035212   1.09866357  1.11734521  1.13440645  1.17730546
  1.18442798  1.20736992  1.2085104   1.21593571]
```

Brute-Force

Note that the brute-force implementatation is only used for comparison purposes given data sets in relatively low-dimensional search spaces. Its performance is suboptimal for high-dimensional feature spaces compared to matrix-based implementations that make use of e.g. CUBLAS (but superior to such implementations given low-dimensional search spaces).

1.3.2 Large-Scale Querying

The main purpose of the buffer k-d tree implementation is to speed up the querying phase given both a large number of reference and a huge number of query points. The next data example is based on astronomical data from the Sloan Digital Sky Survey (the data set will be downloaded automatically):

```
#

"""
Nearest Neighbors in Astronomy
==============================

This example demonstrates the use of both k-d tree-based
implementations on a large-scale astronomical data set that is
based on the Sloan Digitial Sky Survey (http://www.sdss.org).
The data set contains 2,000,000 training points and 10,000,000
test points in a 10-dimensional feature space.

Note: The platform and the device needs to be specified below
(via the parameter 'plat_dev_ids').
"""
print(__doc__)

import time
import generate
from bufferkdtree import NearestNeighbors

# parameters
plat_dev_ids = {0:[0,1,2,3]}
n_jobs = 8
verbose = 1
float_type = "float"
n_neighbors = 10
```

Note that four devices (with ids 0,1,2,3) of the first platform (with id 0) are used in this case.

Platform ID and Devices

Most likely, you will have to adapt these numbers! A common setting is *plat_dev_ids = {0:[0]}*, i.e., the first OpenCL platform and the first OpenCL device are used.

The helper function defined next is used to time the runtimes needed for the training and testing phases of each method:

```
def run_algorithm(algorithm="buffer_kd_tree", tree_depth=None, leaf_size=None):

    nbrs = NearestNeighbors(n_neighbors=n_neighbors,
                            algorithm=algorithm,
                            tree_depth=tree_depth,
                            leaf_size=leaf_size,
                            float_type=float_type,
                            n_jobs=n_jobs,
                            plat_dev_ids=plat_dev_ids,
                            verbose=verbose)

    start_time = time.time()
    nbrs.fit(Xtrain)
```

```
    end_time = time.time()
    print("Fitting time: %f" % (end_time - start_time))

    start_time = time.time()
    _, _ = nbrs.kneighbors(Xtest)
    end_time = time.time()
    print("Testing time: %f" % (end_time - start_time))
```

Note that either `tree_depth` or `leaf_size` is used to determine the final tree depth, see the *documentation*. For this example, large sets of reference (two million) and query points (ten million) are generated:

```
print("Parsing data ...")
Xtrain, Ytrain, Xtest = generate.get_data_set(NUM_TRAIN=2000000, NUM_TEST=10000000)
print("-------------------------------- DATA --------------------------------")
print("Number of training patterns:\t %i" % Xtrain.shape[0])
print("Number of test patterns:\t %i" % Xtest.shape[0])
print("Dimensionality of patterns:\t%i" % Xtrain.shape[1])
```

Loading the data this way should yield an output like:

```
Nearest Neighbors
=================

This example demonstrates the use of both tree-based
implementations on a large-scale data set.

-------------------------------- DATA --------------------------------
Number of training patterns: 2000000
Number of test patterns:     10000000
Dimensionality of patterns:  10
----------------------------------------------------------------------
```

Finally, both implementations are invoked to compute the 10 nearest neighbors for each query point:

```
print("\n\nRunning the GPU version ...")
run_algorithm(algorithm="buffer_kd_tree", tree_depth=9)

print("\n\nRunning the CPU version ...")
run_algorithm(algorithm="kd_tree", leaf_size=32)
```

The above code yields the folling output on an *Ubuntu 14.04* system (64 bit) with an *Intel(R) Core(TM) i7-4790K* running at 4.00GHz (4 cores, 8 hardware threads), 32GB RAM, two *Geforce Titan Z* GPUs (with two devices each), CUDA 6.5 and Nvidia driver version 340.76:

```
Running the GPU version ...
Fitting time: 1.394939
Testing time: 11.148126

Running the CPU version ...
Fitting time: 0.681938
Testing time: 314.787735
```

The parameters `tree_depth` and `leaf_size` play an important role: In case `tree_depth` is not `None`, then `leaf_size` is ignored. Otherwise, `leaf_size` is used to automatically determine the corresponding tree depth (such that at most `leaf_size` points are stored in a single leaf). For `kd_tree`, setting the leaf size to, e.g., 32 is usually a good choice. For `buffer_kd_tree`, a smaller tree depth is often needed to achieve a good performance (e.g., `tree_depth=9` for 1,000,000 reference points).

Performance

The performance might depend on the particular OpenCL version and on the particular Nvidia driver that are installed. For instance, we observed similar speed-ups (per device) with a *weaker* Gefore GTX 770 given CUDA 5.5, OpenCL 1.2, and Nvidia driver version 319.23.

Tree Construction

Both implementations are based on the standard rule for splitting nodes during the construction (cyclic, median based). Other splitting rules might be beneficial, but are, in general, data set dependent. Other construction schemes will be available in future for all tree-based schemes.

1.4 Reference

1.4.1 NearestNeighbors

All neighbor implementations can be invoked via the main `NearestNeighbors` class, which exhibits a similar structure as the corresponding class from the scikit-learn package:

class `bufferkdtree.neighbors.`**`NearestNeighbors`**(*n_neighbors=5*, *algorithm='buffer_kd_tree'*, *float_type='float'*, *tree_depth=None*, *leaf_size=30*, *splitting_type='cyclic'*, *n_train_chunks=1*, *plat_dev_ids={0: [0]}*, *allowed_train_mem_percent_chunk=0.15*, *allowed_test_mem_percent=0.55*, *n_jobs=1*, *verbose=0*, ***kwargs*)

The 'NearestNeighbors' provides access to all nearest neighbor implementations. It has simmilar parameters as the corresponding implementation of the scikit-learn package.

The main method is the "buffer_kd_tree", which can be seen as mix between the "brute" and the "kd_tree" implementations.

Parameters **n_neighbors** : int (default 5)

Number of neighbors used

algorithm : {"brute", "kd_tree", "buffer_kd_tree"}, optional (default="buffer_kd_tree")

The algorithm that shall be used to compute the nearest neighbors. One of - 'brute': brute-force search - 'kd_tree': k-d tree based search - 'buffer_kd_tree': buffer k-d tree based search (with GPUs)

tree_depth : int or None, optional (default=None)

Passed to the 'kd_tree' and 'buffer_kd_tree' implementation. In case 'tree_depth' is specified, a tree of such a depth is built ('tree_depth' has priority over 'leaf_size').

leaf_size : int, optional (default=30)

Passed to the 'kd_tree' and 'buffer_kd_tree' implementation. In case 'leaf_size' is set, the corresponding tree depth is computed (is ignored in case tree_depth is not None).

splitting_type : {'cyclic'}, optional (default='cyclic')

Passed to the 'kd_tree' and 'buffer_kd_tree' implementation. The splitting rule that shall be used to construct the kd tree. Currently, only "cyclic" is supported.

n_train_chunks : int, optional (default=1)

Passed to the 'buffer_kd_tree' implementation. The number of chunks the training patterns shall be processed in; only needed in case the training patterns do not fit on the GPU (in case n_train_chunks is too small, it is increased automatically).

plat_dev_ids : dict, optional (default={0:[0]})

Passed to the 'brute' and the 'buffer_kd_tree' implementation. The platforms and devices that shall be used. E.g., plat_dev_ids={0:[0,1]} makes use of platform 0 and the first two devices.

allowed_train_mem_percent_chunk : float, optional (default=0.15)

Passed to the 'buffer_kd_tree' implementation. The amount of memory (OpenCL) used for the training patterns (in percent).

allowed_test_mem_percent : float, optional (default=0.55)

Passed to the 'buffer_kd_tree' implementation. The amount of memory (OpenCL) used for the test/query patterns (in percent).

n_jobs : int, optional (default=1)

Passed to the 'kd_tree' implementation. The number of threads used for the querying phase.

verbose : int, optional (default=0)

The verbosity level (0=no output, 1=output)

Notes

The brute-force implementatation is only used for comparison in relatively low-dimensional spaces; the performance is suboptimal for higher dimensional feature spaces (but even superior over other matrix based implementations making use e.g., CUBLAS).

The performance of the GPU implementations depends on the corresponding architecture. An important ingredient is the support of automatic hardware caches.

Examples

```
>>> import numpy
>>> from bufferkdtree.neighbors import NearestNeighbors
>>> X = numpy.random.uniform(low=-1,high=1,size=(10000,10))
>>> nbrs = NearestNeighbors(n_neighbors=10, algorithm="buffer_kd_tree", tree_depth=9, plat_dev_i
>>> nbrs.fit(X)
>>> dists, inds = nbrs.kneighbors(X)
```

fit (*X*)

Fit the model to the given data.

Parameters **X** : array-like, shape (n_samples, n_features)

The set of training/reference points, where 'n_samples' is the number points and 'n_features' the number of features.

Returns **self** : instance of NearestNeighbors

The object itself

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
Finds the nearest neighbors for a given set of points.

Parameters **X** : array-like, shape (n_samples, n_features)

The set of query points. If not provided, the neighbors of each point in the training data are returned (in this case, the query point itself is not considered its own neighbor.

n_neighbors : int or None, optional (default=None)

The number of nearest neighbors that shall be returned for each query points. If 'None', then the default values provided to the constructor is used.

return_distance : bool, optional (default=True)

If False, then the distances associated with each query will not be returned. Otherwise, they will be returned.

Returns **dist** : array

The array containing the distances

idx : array

The array containing the indices

1.4.2 Adapting Buffer K-D Trees

If you wish to adapt the buffer k-d tree implementation, then you might want have a look at the C and OpenCL code that is available in the *bufferkdtree/src* directory.

Developer C API

A documentation of the underlying C code can be found here.

A good starting point for diving into the details of the underlying implementation is the base.c file in bufferkdtree/src/neighbors/buffer_kdtree directory.

1.5 License

The source code is published under the GNU General Public License Version 2 (GPLv2).

1.6 Changes

1.6.1 Release 1.3 (November 2016)

- Updated documentation (which now also includes a documentation of the C/OpenCL source code)
- Adapted/Added examples
- Adapted installation process (e.g., CPU based k-d trees can now be built without OpenCL support)
- Small code modifications and bugfixes

1.6.2 Release 1.2 (July 2016)

- Added support for double precision
- Added Python 3 support
- Small bug fixes
- Fixed memory leak (GH1)

1.6.3 Release 1.1.1 (December 2015)

- Updated documentation

1.6.4 Release 1.1 (December 2015)

- Fixed wrong parameter assignment in 'kneighbors' method of both neighbors/kd_tree/base.py and neighbors/buffer_kdtree/base.py
- Added Multi-GPU support to brute-force approach (for benchmark purposes)
- Adapted parameter settings for buffer k-d tree implementation
- Added benchmark example

1.6.5 Release 1.0.2 (September 2015)

- Adapted building process

1.6.6 Release 1.0.1 (September 2015)

- Adapted building process
- Fixed small bugs

1.6.7 Release 1.0 (September 2015)

- First major release
- Python wrappers for three implementations ('brute', 'kd_tree', 'buffer_kd_tree')
- Performance improvements for both kd-tree based implementations
- Multi-OpenCL-Device support for 'buffer_kd_tree' implementation
- Large-scale construction for 'buffer_kd_tree' implementation
- Multi-OpenCL-Device support for query phase (queries are processed in chunks)
- Added Sphinx documentation

1.7 Citations

If you wish to cite a paper that describes the techniques and the implementation for buffer k-d trees, please make use of the following work:

Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. *Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs*. In: Proceedings of the 31st International Conference on Machine Learning (ICML) 32(1), 2014, 172-180. [pdf] [bibtex]

Index

- genindex

F

K

N