bufferkdtree Documentation

Release 1.0

Fabian Gieseke

October 01, 2015

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) using OpenCL. The source code is published under the GNU General Public License (GPLv2).

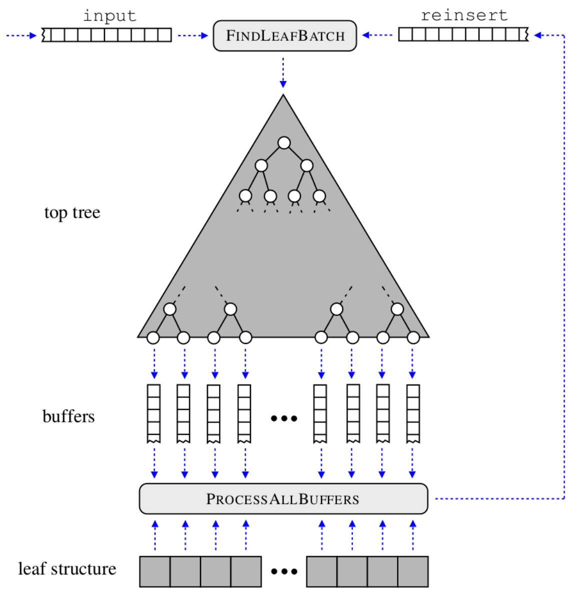
CHAPTER 1

Contents

1.1 Quick Overview

The main framework provided by the bufferkdtree package is an efficient many-core (e.g., GPU) implementation 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 make use of the massive parallelism provided by today's many-core devices (such as GPUs) to process the leaves of the tree.

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 or more points) with an input space of moderate dimensionality (e.g., from d=5 to 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 by a standard k-d tree traversal (given the same 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 for 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

1.2.1 Dependencies

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

To install the package, a working C/C++ compiler, OpenCL, Swig, and the Python development package (header files) need to be available. Further, the NumPy package (>=1.6.1) is needed.

On Ubuntu 12.04, for instance, the following command installs all dependencies (except for OpenCL):

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

On an OpenSUSE system, the corresponding commands are:

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

Compatibility

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

1.2.2 OpenCL

OpenCL needs to be installed correctly. Make sure that the OpenCL header files are available, for example by setting the C_INCLUDE_PATH environment variable in the .bashrc file on Linux systems. For instance, in case CUDA is installed with header files being located in `/usr/local/cuda/include`, then the following command should update the environment variable:

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

1.2.3 Quick Installation

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

The package is available on PyPI, but can also be installed from the sources. For instance, to install the package via PyPI on Linux machines, type:

```
$ sudo pip install bufferkdtree
```

To install the package from the sources, first get the current version 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
```

1.2.4 Virtualenv & Pip

We recommend to install the package via virtualenv and pip. On Ubuntu 12.04, for instance, the following commands can be used to install virtualenv and pip:

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

Afterwards, create a new virtual environment and install the Numpy package:

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

Given the activated virtual environment, follow the instructions above to install the bufferkdtree package.

1.3 Examples

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

1.3.1 Toy Example

```
import numpy
from bufferkdtree.neighbors import NearestNeighbors

n_neighbors = 10
plat_dev_ids = {0:[0]}
```

```
n_jobs = 1
verbose = 0
```

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 this example, the first platform and its first device are used.

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" + unicode(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" + unicode(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" + unicode(dists[0]))
print("")
```

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

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

```
import time
import generate
from bufferkdtree.neighbors import NearestNeighbors

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

Note that four devices of the first platform are used now (0,1,2,3). 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, \
                            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()
    dists, inds = 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:

```
Xtrain, Ytrain, Xtest = generate.get_data_set(NUM_TRAIN=2000000, NUM_TEST=10000000)
print "-------------------------------- DATA --------------------------------"
print "Number of training patterns:\t", Xtrain.shape[0]
print "Number of test patterns:\t", Xtest.shape[0]
print "Dimensionality of patterns:\t", 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 Nvidia driver. For instance, we observed similar speed-ups (per device) with a weeker Gefore GTX 770 given CUDA 5.5 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

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

class `bufferkdtree.neighbors.`**`NearestNeighbors`**(*n_neighbors=5*, *algorithm='buffer_kd_tree'*, *tree_depth=None*, *leaf_size=30*, *splitting_type='cyclic'*, *n_train_chunks=1*, *plat_dev_ids={0: [0]}*, *allowed_train_mem_percent_chunk=0.2*, *allowed_test_mem_percent=0.8*, *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=0.2)

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=0.8)

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.

Only single-precision is supported until now.

Examples

```
>>> import numpy
>>> from bufferkdtree.neighbors.base 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

`compute_optimal_tree_depth` (*Xtrain*, *Xtest*, *target='test'*, *tree_depths=None*)

Computes the optimal tree depth for the tree-based implementations. The method tests various assignments of the parameters and simply measures the time needed for the approach tp finish.

Parameters **Xtrain** : 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.

Xtest : array-like, shape (n_samples, n_features)

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

target : {'train', 'test', 'both'}, optional (default='test')

The runtime target, i.e., which phase shall be optimized. Three choices: - 'train' : The training phase - 'test' : The testing phase - 'both' : Both phases

tree_depths : list or None, optional

The range of different tree depths that shall be tested. If None, then the default ranges are used by the different implementations:

- buffer_kd_tree : range(2, max_depth - 1)
- kd_tree : range(4, max_depth - 1)

where max_depth = int(math.floor(math.log(len(Xtrain), 2)))

Returns **opt_height** : int

The optimal tree depth

1.5 License

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

1.6 Changes

1.6.1 Release 1.0

- 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]

CHAPTER 2

Index

- genindex

Index