pySIML Documentation

Release 1.5

Imran S. Haque

March 07, 2010

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO PYSIML

pySIML gives you easy access to SIML, an extremely fast method for computing LINGO chemical similarities [Vidal05]. There are existing implementations of LINGO, such as that included in OpenEye's OEChem toolkit - so why bother with SIML?

Speed.

For several classes of important problems, SIML is significantly faster than existing implementations of LINGOs. In particular, for M x N similarity problems, in which one needs to compare every molecule in a set of size M against every other molecule in a set of size N (and M and N are reasonably large, on the order of hundreds), SIML on a single-core CPU is several times as fast as existing implementations of LINGO. SIML also supports computing LINGOs on a CUDA-capable GPU, which allows over eightyfold speedup relative to even fast CPUs.

SIML and pySIML live at https://simtk.org/home/siml.

CHAPTER

TWO

INSTALLING PYSIML

2.1 Prerequisites

- pySIML requires a working Python installation, since it is a collection of Python bindings. It has been successfully tested on Python 2.4 and 2.5 on both Linux and Mac OS X. It also requires that the development header files for the Python interpreter used be installed on the machine (e.g., on Ubuntu Linux, package `python-dev` must be installed, not just `python`).
- The NumPy package, and its headers, must also be installed. pySIML makes extensive use of NumPy to store input and output data for the SIML algorithm.
- An OpenMP-capable compiler is required to take advantage of multiple CPUs (parallel computations over multiple rows of a Tanimoto matrix in *CPULingo*).
- PyCUDA version 0.94 or greater is required for NVIDIA GPU support using *GPULingo*. Versions 0.93 and previous will not work properly! Note that at the time of this writing, 0.93 is the most recent release version. If this is still true, then you must retrieve a copy of the PyCUDA source code from the source repository (following the directions given on the PyCUDA homepage).
- PyOpenCL is required for NVIDIA/AMD GPU support using *OCLLingo*.

2.2 Setup Procedure

pySIML is distributed as a source tarball using a mostly-standard Python distutils-based setup procedure. After untarring the package, most people should be able to run:

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

In some cases, the setup script will not be able to detect one or more settings properly, in which case, the configure option can be used:

```
python setup.py configure <options>
```

The following options are available:

- `--enable-openmp`: Force pySIML to be built with OpenMP support.
- `--disable-openmp`: Force pySIML to be built without OpenMP support.

- `--numpy-include=<dir>`: Indicates that the C headers for numpy can be found in <dir>. Note that if, for example, `arrayobject.h` is in `/usr/include/python2.5/numpy/arrayobject.h`, <dir> should be specified as `/usr/include/python2.5`, NOT `/usr/include/python2.5/numpy`.
- `--python-include=<dir>`: Indicates that the C headers for Python (e.g., `Python.h`) can be found in <dir>.

CHAPTER

THREE

PYSIML CONCEPTS

3.1 Basics of LINGO

pySIML is designed to compute chemical similarities according to the LINGO method [Vidal05] [Grant06]. LINGO, strictly speaking, is a text-similarity algorithm (similar to the d-squared algorithm for sequence comparison); it is mapped to chemical similarites by representing molecules in some textual format. LINGO characterizes a molecule by taking its textual representation (typically a canonical or canonical-isomeric SMILES string), and breaking it into all its 4-character substrings. To compare two molecules, each is fragmented into its component substrings, and the similarity between the two is defined as:

$$\frac{\text{number of fragments in common}}{\text{total count of distinct fragments}}$$

Or, more technically, consider each molecule A and B to be a multiset, or bag, of these fragments (known as "Lingos"). The Tanimoto similarity between A and B is defined as

$$T_{AB} \equiv \frac{|A \cap B|}{|A \cup B|}$$

Several efficient algorithms to calculate LINGO similarities exist (e.g., [Grant06]), which work especially well when comparing a large number of molecules to only one single candidate molecule. SIML targets a slightly different application area, in which multiple molecules will be compared against the same set of "query" or "database" molecules, either at the same time or sometime in the future (by storing precalculated values). By precalculating part of the computation, SIML saves a large amount of computer time for these repeated searches. Additionally, it allows fast implementation on both standard processors (CPUs) as well as graphics cards (GPUs) and similar vector processors. For more details on the SIML algorithm, please see the publication (pending).

All LINGO algorithms do some preprocessing on the SMILES strings they are given. Typically, ring closure digits will be set to zero (to normalize ring assignment); in some variants (e.g. [Vidal05]), atom names will also be remapped. SIML implements two preprocessing methods, described in *Preprocessing SMILES in pySIML*.

3.2 pySIML Computation Model

The motivating task for pySIML is the calculation of a **Tanimoto matrix**, or matrix whose entries are chemical similarities. Each row corresponds to a molecule from a set called the **reference set**, and each column to a molecule from the **query set**. Given a Tanimoto matrix T, the entry at T_{ij} (row *i*, column *j*) is the similarity between reference molecule *i* and query molecule *j*.

This model is adaptable to a diverse set of problems in cheminformatics. Performing a database screen (in which one molecule is compared to many others) can be described as calculating a Tanimoto matrix of size 1 x N (one reference molecule, and a number of query molecules equal to the size of the database). A self-similarity matrix, as might be used for clustering a molecule set, corresponds to the creation of an N x N matrix, where the reference and query sets are identical. Finally, a multiple-screen or cross-similarity comparison, in which two distinct sets are compared to each other in an all-pairs manner, is an M x N matrix problem, with M and N being the size of either set.

In addition to the Tanimoto matrices SIML produces as its output, it takes matrices called **SMILES sets** as input. SIML's preprocessor/compiler code takes sets of SMILES strings, and converts them to a numerical matrix representation called a SMILES set. Each SMILES set contains 2 matrices and 2 vectors:

- One **Lingo matrix** contains numerical representations of the Lingos in each SMILES string. Each input SMILES string corresponds to one row in the Lingo matrix.
- One **count matrix** contains the multiplicity of each Lingo in the Lingo matrix. Each input SMILES string corresponds to one row in the count matrix.
- One **length vector** contains the number of distinct Lingos in each molecule.
- One **magnitude vector** contains the total number of Lingos (distinct or not) in each molecule.

In general, it is not important to worry about the internal details of these representations. However, what is useful to know is that it is possible to construct unions and subsets of these SMILES sets. The following code example demonstrates:

```
@# lingos, counts, lengths, and mags have been initialized as a SMILES set

lingoSub  = lingos[0:10,:]
countSub  = counts[0:10,:]
lengthSub = lengths[0:10]
magSub    = mags[0:10]

@# lingoSub, countSub, lengthSub, and magSub together now constitute a
@# SMILES set for the first ten molecules from the original set
```

These SMILES sets are constructed from SMILES strings by the pySIML compiler routines (*Preprocessing SMILES in pySIML*).

CHAPTER

FOUR

PREPROCESSING SMILES IN PYSIML

4.1 The Short Version

Just use `cSMILEStoMatrices()`, and don't look back.

4.2 The Long Version

As explained in the *pySIML Concepts*, two things must be done to SMILES strings before they can be used for LINGO similarity comparison:

- Certain transformations must be performed, such as changing ring closure digits and stripping names
- They must be converted to the SIML internal numerical representation.

The *pysiml.compiler* section provides details on how to do this conversion.

The `pysiml.compiler` module provides both a pure-Python converter `SMILEStoMatrices()` as well as one based around a C extension, `cSMILEStoMatrices()`. It is important to note that these two **DO NOT** produce the same output. The Python module transforms all digits in the SMILES string to zeroes; this will incorrectly affect charge numbers, isotope indicators, and hydrogen counts. The C module performs the following changes:

- Change all digits to zero, except for numbers following a '+' or '-' (charge counts), those following an 'H' (hydrogen counts), or a '[' (isotope indicators).
- Reduce multiple-digit ring-closure indicators (e.g., '%13') to one digit ('%0') to normalize ring formatting. Currently only works for molecules with under 100 rings, due to ambiguities in the SMILES specification.

Both the C and Python modules handle stripping of names and newlines from SMILES strings.

In general, **there is almost no reason to use the Python compiler**; `cSMILEStoMatrices()` is nearly 100 times faster and is more correct to the LINGO flavor outlined in [Grant06]. The Python compiler `SMILEStoMatrices()` is included only as a substitute in case a pure-Python replacement is needed, or it is necessary to compute SMILES strings that have been transformed in an identical way to the SIML compiler (e.g., to pass into a different LINGO package for comparison).

4.3 pysiml.compiler - Transforming SMILES strings into SIML internal representations

This module provides "compilers" to convert SMILES strings into the sparse-vector representation required for SIML. C and pure-Python implementations are provided.

`SMILEStoMatrices`(*smileslist*)

Convert the sequence of SMILES strings *smileslist* into a SIML SMILES set and list of molecule names (if present in the SMILES strings). Uses a pure Python implementation. See pySIML preprocessing documentation for details on transformations performed on the SMILES strings. Note that this does NOT perform the same transformations as the C version `cSMILEStoMatrices()`.

Returns a tuple of 5 values: a Lingo matrix, a count matrix, a length vector, a magnitude vector, and a list of molecule names (all but the molecule names make up the "SMILES set").

`SMILEStoMultiset`(*smiles*)

Returns Lingo and count vectors for a single SMILES string *smiles*, as would correspond to a row in the Lingo or count matrices from `cSMILEStoMatrices()` or `SMILEStoMatrices()`. Performs no transformations on *smiles* prior to conversion.

Note that in general, the results of this function will not be the same as those obtained from `SMILEStoMatrices()` or `cSMILEStoMatrices()` because this function does not preprocess the input strings.

`cSMILEStoMatrices`(*smileslist*)

Convert the sequence of SMILES strings *smileslist* into a SIML SMILES set and list of molecule names (if present in the SMILES strings). Uses the SIML compiler C extension. See pySIML preprocessing documentation for details on transformations performed on the SMILES strings. Note that this does NOT perform the same transformations as the pure-Python version `SMILEStoMatrices()`.

Returns a tuple of 5 values: a Lingo matrix, a count matrix, a length vector, a magnitude vector, and a list of molecule names (all but the molecule names make up the "SMILES set").

`preprocessNumbers`(*smi, xtable=None*)

Given a SMILES string, return a copy of the string with the same translations performed on it as would be done by the pure-Python preprocessor `SMILEStoMatrices()`.

This method is primarily useful to compare the results of SIML Tanimoto calculation functions with those from other LINGO calculation packages, to ensure that identical SMILES strings are given to each.

xtable is an internal parameter and should always be set to None when called from user code.

CHAPTER

FIVE

SIMILARITY CALCULATIONS WITH PYSIML

5.1 Generic API for computing LINGOs with pySIML

Both of the currently supported methods for LINGO computations with pySIML share the same general data flow and methods, to make it easy to switch between CPUs and GPUs as needed. The overall structure of a Tanimoto computation with pySIML is as follows:

- Read SMILES (from file, database, generator, etc)
- Preprocess SMILES for reference and query sets (see section *Preprocessing SMILES in pySIML*) into a pair of 'SMILES sets', each consisting of a Lingo matrix, count matrix, length vector, and magnitude vector.
- Create a LINGO comparator object (a *CPULingo*, *GPULingo*, or *OCLLingo* object)
- Initialize the comparator with the reference and query SMILES sets using the `set_refsmiles` and `set_qsmiles` functions.
- Request a single row from the Tanimoto matrix using the `getTanimotoRow` or `getTanimotoRow_async` methods, or a contiguous range of rows using `getMultipleRows` or `getMultipleRows_async`.

5.2 Example of computing similarities with pySIML

The following is a simple demonstration of how to calculate a full N x N similarity matrix on a set of compounds read in from a file. Note that it lacks niceties such as error-checking; a more detailed example code is present in the examples directory:

```
import sys
import numpy
from pysiml.compiler import cSMILEStoMatrices
from pysiml.CPULingo import CPULingo

f = open(sys.argv[1],"r")
smiles = f.readlines()
f.close()

numMols = len(smiles)

@# We use cSMILEStoMatrices because it is almost 100x as fast as
@# SMILEStoMatrices, and more correct to boot.
```

```
@#
@# The SMILES compiler also returns the molecule name associated
@# with each row of the output matrices
(lingos,counts,lengths,mags,names) = cSMILEStoMatrices(smiles)

@# Construct a similarity object. This could also be a GPULingo
comparator = CPULingo()

@# Initialize the comparator with our SMILES sets. Since this
@# computation is a self-similarity matrix, the reference and
@# query sets are the same
comparator.set_refsmiles(lingos,counts,lengths,mags)
comparator.set_qsmiles(lingos,counts,lengths,mags)

@# Create an empty storage place to put the result
similarityMatrix = numpy.empty((numMols,numMols))

@# CPULingo-specific: see if we can run the computation in parallel
numProcs = 1
if comparator.supportsParallel():
    @# If we can do a row-parallel computation (OpenMP supported), choose the
    @# number of processors here
    numProcs = 4
similarityMatrix[:,:] = comparator.getMultipleRows(0,numMols,nprocs=numProcs)

print similarityMatrix
```

The following sections explain details of the CPULingo and GPULingo APIs and differences in their respective behavior.

5.3 pysiml.CPULingo - Computing LINGO similarities on a CPU

This module exposes the API for computing LINGO similarities on a CPU. Calculations of multiple rows can be parallelized across multiple CPUs, if the library has been built with OpenMP support. There is currently no support for parallelizing the computation of a single row across multiple CPUs.

The CPULingo object is the interface to compute LINGOs on a CPU. Creating multiple CPULingo objects will not parallelize computations on each across multiple CPUs (as with GPULingo); the only parallelism currently exposed is across rows, using OpenMP.

For interface consistency, CPULingo exposes asynchronous operation methods (`getTanimotoRow_async()` and `getMultipleRows_async()`); however, these methods as currently implemented are not actually asynchronous operations.

5.3.1 CPULingo object documentation

class **`CPULingo`**()

Object to handle computation of LINGO similarities on the CPU

`asyncOperationsDone`()

Return True if all asynchronous operations on this object have completed.

In current implementation, always returns True.

`getMultipleRows`(*rowbase, rowlimit, nprocs=1*)

Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set, and returns this block of rows.

If pySIML has been built with OpenMP enabled, *nprocs* may be set higher than 1 to parallelize computations over multiple CPUs (each CPU will handle a disjoint set of rows). If called with *nprocs* larger than 1 on a non-OpenMP build of pySIML, print a warning to stderr and compute with one CPU.

getMultipleRows_async(*rowbase, rowlimit, nprocs=1*)
Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set, and stores this block of rows internally as the last asynchronous result value.

If pySIML has been built with OpenMP enabled, *nprocs* may be set higher than 1 to parallelize computations over multiple CPUs (each CPU will handle a disjoint set of rows). If called with *nprocs* larger than 1 on a non-OpenMP build of pySIML, print a warning to stderr and compute with one CPU.

To retrieve the result block, call `retrieveAsyncResult()`.

Note that this function is actually synchronous, due to the limitations of running on the CPU; it will not return until the block has been completely calculated.

getTanimotoRow(*row*)
Returns the single Tanimoto row *row* corresponding to comparing every SMILES string in the query set with the single reference SMILES string having index *row* in the reference set.

getTanimotoRow_async(*row*)
Computes the single Tanimoto row *row* corresponding to comparing every SMILES string in the query set with the single reference SMILES string having index *row* in the reference set, and stores this row internally as the last asynchronous result value.

To retrieve the result row, call `retrieveAsyncResult()`.

Note that this function is actually synchronous, due to the limitations of running on the CPU; it will not return until the row has been completely calculated.

retrieveAsyncResult()
Returns result from last asynchronous computation (`getTanimotoRow_async()` or `getMultipleRows_async()`).

Note that this result is only guaranteed to be valid if no operations have been run on this CPULingo object since the asynchronous call, except for `asyncOperationsDone()` and `retrieveAsyncResult()`.

If no asynchronous operations have been invoked on this object, result is undefined.

set_qsmiles(*qsmilesmat, qcountsmat, querylengths, querymags=None*)
Sets the query SMILES set to use Lingo matrix *qsmilesmat*, count matrix *qcountsmat*, and length vector *querylengths*. If *querymags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed from the count matrix.

set_refsmiles(*refsmilesmat, refcountsmat, reflengths, refmags=None*)
Sets the reference SMILES set to use Lingo matrix *refsmilesmat*, count matrix *refcountsmat*, and length vector *reflengths*. If *refmags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed from the count matrix.

supportsParallel()
Return True if this installation of pySIML was built with OpenMP support for parallel calculations.

Note that even if this function returns False, the `getMultipleRows()` and `getMultipleRows_async()` methods can still be called with nprocs > 1, but only one processor will actually be used

5.4 pysiml.GPULingo - Computing LINGO similarities on a CUDA-capable GPU

This module exposes the API for computing LINGO similarities on a CUDA-capable GPU. It uses the pycuda library to interface with the GPU; in particular, due to bugs related to context management in pycuda 0.93 and before, pycuda 0.94 or greater is required.

The GPULingo object is the interface to compute LINGOs on a single GPU. To do similarity calculations on multiple GPUs, create multiple GPULingo objects, passing a different CUDA device ID to each one's constructor:

```
gpu0 = pysiml.GPULingo(0)
gpu1 = pysiml.GPULingo(1)
```

By using the asynchronous operations on each object (`getTanimotoRow_async()` and `getMultipleRows_async()`), similarity calculations can be carried out simultaneously on multiple GPUs using only one host thread:

```
@# gpu0 and gpu1 have been initialized with reference and query SMILES sets

@# Carry out simultaneous computation of rows 0 to 10 of each set on both GPUs
gpu0.getMultipleRows_async(0,10)
gpu1.getMultipleRows_async(0,10)

@# The busy waits could be replaced by a sleep, or any other work
while not gpu0.asyncOperationsDone():
    pass
gpu0result = gpu0.retrieveAsyncResult()

while not gpu1.asyncOperationsDone():
    pass
gpu1result = gpu1.retrieveAsyncResult()
```

After an asynchronous computation has been requested on a GPULingo object, check `asyncOperationsDone()` to see when the job is complete. Once the job is done, `retrieveAsyncResult()` can be called to retrieve the result. Note that the retrieved result is guaranteed to be valid only if no methods were called on the GPULingo object after the asynchronous request, except for `asyncOperationsDone()` and `retrieveAsyncResult()`.

5.4.1 GPULingo object documentation

class **`GPULingo`**(*deviceID=0*)

Object to handle computation of LINGO similarities on GPU with CUDA device ID *deviceid*

`asyncOperationsDone`()

Return True if all asynchronous operations on this object have completed.

`getMultipleRows`(*rowbase, rowlimit*)

Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set, and returns this block of rows.

This method is synchronous (it will not return until the block has been completely computed).

`getMultipleRows_async`(*rowbase, rowlimit*)

Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set, and stores this block as the most recent asynchronous result.

This method is asynchronous (it will return before the block has been completely computed). After calling this method, check `asyncOperationsDone()`; once that method returns True, the result may be retrieved by calling `retrieveAsyncResult()`.

`getTanimotoRow`(*row*)
Returns the single Tanimoto row *row* corresponding to comparing every SMILES string in the query set with the single reference SMILES string having index *row* in the reference set.

This method is synchronous (it will not return until the entire row has been computed and brought back from the GPU).

`getTanimotoRow_async`(*row*)
Compute the single Tanimoto row *row* corresponding to comparing every SMILES string in the query set with the single reference SMILES string having index *row* in the reference set, and store it as the most recent asynchronous result.

This method is asynchronous (it will return before the row has been completely computed). After calling this method, check `asyncOperationsDone()`; once that method returns True, the result may be retrieved by calling `retrieveAsyncResult()`.

`getMultipleHistogrammedRows`(*rowbase, rowlimit*)
Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set. Histograms each row into its own histogram of 101 bins with boundaries 0, 0.01, 0.02, ... , 0.99, 1.0, 1.01. Returns this block of row-wise histograms.

This method is synchronous (it will not return until the histograms have been completely computed).

`getMultipleHistogrammedRows_async`(*rowbase, rowlimit*)
Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set. Histograms each row into its own histogram of 101 bins with boundaries 0, 0.01, 0.02, ... , 0.99, 1.0, 1.01. Returns this block of row-wise histograms.

This method is asynchronous (it will return before the block has been completely computed). After calling this method, check `asyncOperationsDone()`; once that method returns True, the result may be retrieved by calling `retrieveAsyncResult()`.

`getNeighbors`(*rowbase, rowlimit, lowerbound, upperbound=1.1, maxneighbors=None*)
For each reference SMILES string with index in *rowbase:rowlimit* (i.e., strings with index *row*, *row+1*, ... ,*rowlimit-1*, finds all SMILES in the query set that have LINGO similarity >= *lowerbound* and < *upperbound* ("neighbors"), up to a maximum of *maxneighbors* (by default, size of query set).

Result is a tuple of (matrix,vector). The vector contains, for each reference string in *rowbase:rowlimit*, the number of neighbors found. The matrix is of size (rowlimit-rowbase, maxNeighborsFound), where maxNeighborsFound is the maximum value in the returned vector. Each row of the matrix (corresponding to one reference SMILES string) has as its elements the query indices of neighbors. In row i, only the first vector[i] elements are valid (that is, the values elements of the matrix beyond the number of neighbors found for a given row are undefined).

This method is synchronous (it will not return until the neighbors have been completely computed. Returns a tuple of (neighborMatrix,neighborCountVector).

`getNeighbors_async`(*rowbase, rowlimit, lowerbound, upperbound=1.1, maxneighbors=None*)
For each reference SMILES string with index in *rowbase:rowlimit* (i.e., strings with index *row*, *row+1*, ... ,*rowlimit-1*, finds all SMILES in the query set that have LINGO similarity >= *lowerbound* and < *upperbound* ("neighbors"), up to a maximum of *maxneighbors* (by default, size of query set).

Result is a tuple of (matrix,vector). The vector contains, for each reference string in *rowbase:rowlimit*, the number of neighbors found. The matrix is of size (rowlimit-rowbase, maxNeighborsFound), where maxNeighborsFound is the maximum value in the returned vector. Each row of the matrix (corresponding to one reference SMILES string) has as its elements the query indices of neighbors. In row i, only the first vector[i] elements are valid (that is, the values elements of the matrix beyond the number of neighbors found for a given row are undefined).

This method is asynchronous (it will return before the block has been completely computed). After calling this method, check `asyncOperationsDone()`; once that method returns True, the result pair may be retrieved by calling `retrieveAsyncResult()`.

`retrieveAsyncResult`()

Returns result from last asynchronous computation (`getTanimotoRow_async()`, `getMultipleRows_async()`, `getMultipleHistogrammedRows_async()`, or `getNeighbors_async()`).

Note that this result is only guaranteed to be valid if no operations have been run on this object since the asynchronous call, except for `asyncOperationsDone()` and `retrieveAsyncResult()`.

If no asynchronous operations have been invoked on this object, result is undefined. If an asynchronous operation is still pending, this method will block until completion.

`set_qsmiles`(*qsmilesmat, qcountsmat, qlengths, [qmags]*)

Sets the reference SMILES set to use Lingo matrix *qsmilesmat*, count matrix *qcountsmat*, and length vector *querylengths*. If *querymags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed (on the GPU) from the count matrix.

Because of hardware limitations, the query matrices (*qsmilesmat* and *qcountsmat*) must have no more than 65,536 rows (molecules) and 32,768 columns (Lingos). Larger computations must be performed in tiles.

`set_refsmiles`(*refsmilesmat, refcountsmat, reflengths, [refmags]*)

Sets the reference SMILES set to use Lingo matrix *refsmilesmat*, count matrix *refcountsmat*, and length vector *reflengths*. If *refmags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed (on the GPU) from the count matrix.

Because of hardware limitations, the reference matrices (*refsmilesmat* and *refcountsmat*) must have no more than 32,768 rows (molecules) and 65,536 columns (Lingos). Larger computations must be performed in tiles.

5.5 pysiml.OCLLingo - Computing LINGO similarities on an OpenCL-capable GPU or CPU

Very Beta - only getMultipleRows currently supported

This module exposes the API for computing LINGO similarities on an OpenCL-capable GPU, CPU, or other accelerator device. It uses the pyopencl library to interface with OpenCL.

The OCLLingo object is the interface to compute LINGOs on a single OpenCL device. Multiple OCLLingo objects can be used (on the same device or multiple devices); in particular, similarity calculations may be parallelized across multiple GPUs by creating multiple OCLLingo objects, one per device. To build an OCLLingo object, an OpenCL device (obtained from an OpenCL Platform using pyopencl) must be passed to the constructor:

```
import pyopencl as cl
platform = cl.get_platforms()[0] @# Use first platform
dev0 = platform.get_devices()[0]
dev1 = platform.get_devices()[1]
gpu0 = pysiml.OCLLingo(dev0)
gpu1 = pysiml.OCLLingo(dev1)
```

By using the asynchronous operations on each object (`getTanimotoRow_async()` and `getMultipleRows_async()`), similarity calculations can be carried out simultaneously on multiple GPUs using only one host thread:

```
@# gpu0 and gpu1 have been initialized with reference and query SMILES sets

@# Carry out simultaneous computation of rows 0 to 10 of each set on both GPUs
gpu0.getMultipleRows_async(0,10)
gpu1.getMultipleRows_async(0,10)

@# The busy waits could be replaced by a sleep, or any other work
while not gpu0.asyncOperationsDone():
    pass
gpu0result = gpu0.retrieveAsyncResult()

while not gpu1.asyncOperationsDone():
    pass
gpu1result = gpu1.retrieveAsyncResult()
```

After an asynchronous computation has been requested on a OCLLingo object, check `asyncOperationsDone()` to see when the job is complete. Once the job is done, `retrieveAsyncResult()` can be called to retrieve the result. Note that the retrieved result is guaranteed to be valid only if no methods were called on the OCLLingo object after the asynchronous request, except for `asyncOperationsDone()` and `retrieveAsyncResult()`.

5.5.1 OCLLingo object documentation

class `OCLLingo`(*device*)

Object to handle computation of LINGO similarities on GPU with CUDA device ID *deviceid*

`asyncOperationsDone`()

Return True if all asynchronous operations on this object have completed.

`getMultipleRows`(*rowbase, rowlimit*)

Computes multiple Tanimoto rows *rowbase:rowlimit* corresponding to comparing every SMILES string in the query set with the reference SMILES strings having index *row*, *row+1*, ..., *rowlimit-1* in the reference set, and stores this block as the most recent asynchronous result.

This method is synchronous (it will not return until the block has been completely computed).

`set_qsmiles`(*qsmilesmat, qcountsmat, qlengths, [qmags]*)

Sets the reference SMILES set to use Lingo matrix *qsmilesmat*, count matrix *qcountsmat*, and length vector *querylengths*. If *querymags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed (on the GPU) from the count matrix.

Because of hardware limitations, the query matrices (*qsmilesmat* and *qcountsmat*) must have no more than 65,536 rows (molecules) and 32,768 columns (Lingos). Larger computations must be performed in tiles.

`set_refsmiles`(*refsmilesmat, refcountsmat, reflengths, [refmags]*)

Sets the reference SMILES set to use Lingo matrix *refsmilesmat*, count matrix *refcountsmat*, and length vector *reflengths*. If *refmags* is provided, it will be used as the magnitude vector; else, the magnitude vector will be computed (on the GPU) from the count matrix.

Because of hardware limitations, the reference matrices (*refsmilesmat* and *refcountsmat*) must have no more than 32,768 rows (molecules) and 65,536 columns (Lingos). Larger computations must be performed in tiles.

CHAPTER

SIX

REFERENCES

CHAPTER

SEVEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[Grant06] Grant JA, Haigh JA, Pickup BT, Nicholls A, and Sayle RA. Lingos, Finite State Machines, and Fast Similarity Searching. Journal of Chemical Information and Modeling, 2006, 46(5), 1912-1918. DOI:10.1021/ci6002152

[Vidal05] Vidal D, Thormann M, Pons M. LINGO, an Efficient Holographic Text Based Method to Calculate Biophysical Properties and Intermolecular Similarities. Journal of Chemical Information and Modeling, 2005, 45(2), 386-393. DOI:10.1021/ci0496797

MODULE INDEX

INDEX