cuTWED

Release 2.0.2

Nov 30, 2021

Contents

CHAPTER 1

cuTWED v2.0.2.

A linear memory CUDA algorithm for solving Time Warp Edit Distance.

1.1 About

This is a novel parallel implementation of Time Warp Edit Distance. The original algorithm was described by Marteau in: `Time Warp Edit Distance with Stiffness Adjustment for Time Series Matching (2009).` A variant of the code from that paper is included in `reference_implementation`, and was used to compare results. There is also a wiki page.

The original TWED algorithm is `O(n^2)` in time and space. This algorithm is roughly `O(n * n/p)` in time for p (CUDA) cores. Most importantly, cuTWED is linear in memory, using storage for roughly `6*nA + 6*nB` elements.

In the process of understanding the dynamic program data dependencies in order to parallelize towards the CUDA architecture, I devised a method with improved memory access patterns and the ability to massively parallelize over large problems. This is accomplished by a procession of a three diagonal band moving across the dynamic program matrix in `nA+nB-1` steps. No `O(n^2)` matrix is required anywhere. Note, this is not an approximation method. The full and complete TWED dynamic program computation occurs with linear storage.

The code is provided in a library that has methods for `double` and `float` precision. It admits input time series in `R^N` as arrays of N-dimensional arrays in C-order (time is the slow moving axis).

For typical problems computable by the original TWED implementation, utilizing cuTWED and thousands of CUDA cores achieves great speedups. For common problems speedups are one to two orders of magnitude, capable of achieving 200x acceleration on a P100 GPUs in doubles. More so, the linear memory footprint allows for the computation of previously intractable problems. Large problems, large systems of inputs can be computed much more effectively now.

Some speed comparisons and a more formal paper will follow.

1.1.1 Reference Implementation

Marteau's original code with a some minor changes has been included in this package. It is built both as a C library and part of the Python package `cuTWED.ctwed`. The minor changes are an extra argument `dimension` to admit `R^N` inputs, and more common handling of norm (nth-root). These modications were made to facilitate refence testing cuTWED and also make the original code more general.

`ctwed` is also included for users without a CUDA card who find other implementations too slow.

1.2 Getting Started

For many users the prepackaged (linux) Python distribution is the simplest way to get the code. If you have CUDA10/11 and a Python 3.6+ manylinux compatible installation you can try the prepackged wheels with:

pip install cuTWED

For other situations, or users seeking maximum performance, instructions follow for building the core CUDA libray, installing, and building the Python bindings from source.

1.2.1 Requirements

For the CUDA code you will need NVCC, a CUDA capable card and CMake. Otherwise, the CUDA code has no dependencies.

If you do not have CMake or it is too old, a lot of people just pip install it `pip install cmake>=3.11`. Otherwise you'll need to refer to their (Kitware) docs for your situation.

For the Python binding `pip` manages the specific depends and installation of the Python interface after you have built the main CUDA C library. Generally requires `numpy`, `pycuda`, and `cffi`. I recommend you use virtualenv or conda to manage Python.

1.2.2 Building

Building has two stages. First the CUDA C library is built and installed. Second the Python bindings (if desired) are built on top of that.

The CUDA C library may be permanently installed to your system, in a standard fashion, with some customization via CMake. Alternatively a manual local install option is described below.

Building the core CUDA C library

Note you may customize the call to `cmake` below with flags like `-DCMAKE_INSTALL_PREFIX=/opt/`, or other flags you might require.

```
# git a copy of the source code
git clone https://github.com/garrettwrong/cuTWED
cd cuTWED

# setup a build area
mkdir build
cd build
cmake ../ # configures/generates makefiles
```

(continues on next page)

(continued from previous page)

```
# make
make -j  # builds the software
```

This should create several files in the `build` directory including `libcuTWED.so`, `*.h` headers, and some other stuff. To install to your system permanently (may require sudo):

```
make install
```

If you would rather just want to temporarily have the library available on your linux machine you can just use the LD path. This makes no changes to your system outside the repo and this shell process. Assuming you are still in your *build* directory, add the library to your path. Users may decide to add a similar line permanently to their *.bashrc* or equivalent.

```
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
```

Python

Once you have the CUDA C library readied, we can use `pip` for the Python bindings. From the root of the git repo:

```
pip install .
```

If you are planning to edit the code, you might prefer pip install in local editable mode instead.

```
pip install -e .
```

1.2.3 Checking

Assuming you have built both the CUDA library and the Python bindings, you can run the unit test suite:

```
pytest
```

Development Testing

For developers there is a whole mess of configured tests which you can run with:

```
tox --skip-missing-interpreters
```

I hope to improve this soon, but there are a *lot* of complication running hybrid codes with free CI tools, and also packaging properly with Python etc that need to be worked through. Some are most easily addressed by using a managed CI host, but this is non free.... I suspect this is largely why you do not see a plethera of free high performance hybrid codes... perhaps a future project...

1.2.4 Using cuTWED in other programs

C/C++

In C/C++ you should be able to `include "cuTWED.h"` and link with the shared library `libcuTWED.so`. This is what I do in `test.x`. The public methods are extern C mangled and should be usable from both C and C++ without issue.

Float (32bit) versions of all the public methods are included in the shared library. They simply have an `f` appended, for example, `twedf` is the float version of `twed`. You may choose which one is suitable for your application. I use floats in `testf.x`.

There are currently two main ways to invoke the cuTWED alogorithm, `twed` and `twed_dev`. First `twed` is the most common way, where you pass C arrays on the host, and the library manages device memory and transfers for you.

Alternatively, if you are already managing GPU memory, you may use twed_dev which expects pointers to memory that resides on the gpu. I have also provided malloc, copy, and free helpers in case it makes sense to reuse memory. See `cuTWED.h`. *All logic and size checks for such advanced cases are expected to be owned by the user.*

There is an additional batch method. Until I have a chance to write up better documentation, you may find example use in `test_batch`, `test_batch_dev`, and a small but respectable ML batch problem set in `test_synthetic_validation.py`.

I have included a Jupyter Notebook which demonstrates validation using the UCI Pseudo Periodic Synthetic Time Series Data Set. This is a much larger dataset.

Future plans include optimization and multi-gpu options for large batches..

Python

```
from cuTWED import twed
```

For Python I have included basic pip installable Python bindings. I use it in `tests/test_basic.py`. If you are curious, these are implemented by a `cffi` backend which parses the C header. which is built for you by `setuptools`. The main Python interface is in `cuTWED.py`. This requires that you have built the library, and have it installed in a location known to the system or available in your `LD_LIBRARY_PATH`.

I have also wrapped up the GPU only memory methods in Python, using PyCUDA gpuarrays. Examples in double and single precision are in `tests/test_basic_dev.py`.

```
from cuTWED import twed_dev
```

The batch interfaces are `twed_batch` and `twed_batch_dev` respectively. Currently it is doing a barbaric synchonization. I have a branch using streams with events, but I need to validate it is robust before I push it. That gives back about another 20% in batch mode afaict.

If you want to run Marteau's C code from Python you can try `ctwed`. For (very) small problems you may find his original C code is faster.

1.3 Troubleshooting and Known Issues

This software is early in its life cycle. The following are known issues:

- Portability, I expect you have linux at this time.
- I have not had time to profile or optimize it, there are things I know to have improvements.

If you find an issue or bug with the code, please submit an issue. More details about this can be found in the contributing document.

1.4 License

GPLv3

CHAPTER 2

Contributing

Contributions are welcome, and they are greatly appreciated!

2.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Your CUDA Toolkit version, driver, and card(s).
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

2.2 Documentation improvements

cuTWED could always use more documentation, whether as part of the official cuTWED docs, in docstrings, or even on the web in blog posts, articles, and such.

2.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/garrettwrong/cuTWED/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a voluntary project. You are welcome to improve things and push the improvements back to the author for review. Consider discussing such work in the issue.

2.4 Development

To set up *cuTWED* for local development:

1. Fork cuTWED (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:garrettwrong/cuTWED.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

2.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request as a draft.

For merging, you should:

1. Include passing tests (run `tox`)[1][2].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

2.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can try running against what you have with `tox --skip-missing-interpreters`.

[2] You may find the basic Docker containers helpful. They can be easily extended and still kept in isolation. See *.gitlab-ci.yml* for a basis.

CHAPTER 3

Authors

- Garrett Wright - https://www.gestaltgp.com

CHAPTER 4

Changelog

CHAPTER 5

Indices and tables

- genindex
- modindex
- search