Trying to speed up homology computation algorithm with cupy

PhD-1 student Anton Dmitriev

Introduction

Homology is a tool from topology proposed for description of manifold invariants (holes).

Homology is a central definition in TDA (topological data analysis) which can be applied to cosmic web [1], image [2] and protein [3] analysis.

However, the main problem of homology is its computational complexity and memory consumption. Every combination of k+1 points is k-dimensional complex. Computing homologies of dimensions greater than 1 is still impossible in practical applications.

In this work I will try to speed up the algorithm using Cupy. Spoiler, I will have no success but I hope that an example when speedup is unreachable by only parallelization will be useful.

1) Van de Weygaert R, Vegter G, Edelsbrunner H, Jones BJ, Pranav P, et al. 2011. Alpha, betti and the megaparsec universe: on the topology of the cosmic web
2) Bonis T, Ovsjanikov M, Oudot S, Chazal F. 2016. Persistence-based pooling for shape pose recognition, In International Workshop on Computational Topology in Image Context.
3) Kovacev-Nikolic V, Bubenik P, Nikoli´c D, Heo G. 2016. Using persistent homology and dynamical distances to analyze protein binding. Statistical applications in genetics and molecular biology

Bottleneck

First of all, I managed to do a code profiling to see the time consumed by each function in the computation.

There are 3 stages

1) Building filtration
2) Reduction of boundary matrix
3) Building a persistence diagram

Now it’s clear what is a bottleneck

```
import time
t0 = time.time()
barc = VietorisRipsFiltration(cloud1)()
print(time.time() - t0)
```

```
0.008276224136352539
```

```
t0 = time.time()
barc.get_reduced_boundary_matrix()
print(time.time() - t0)
```

```
0.23734617233276367
```

```
t0 = time.time()
barc.get_persistence_diagram().as_numpy()
print(time.time() - t0)
```

```
0.012387275695800781
```

Looking at the code

The first cycle is a nested cycle which can be rewritten in cupy

```python
def get_reduced_boundary_matrix(self):

    def matrix_reduction(matrix: np.ndarray) -> np.ndarray:

        def low(column: np.ndarray) -> int:
            if np.any(column!=0):
                return np.flatnonzero(column)[-1]
            return -1

        def reduceable(matrix, j, lows, pivots):
            is_reduceable = False
            if lows[j]!=-1 and pivots[lows[j]]!=-1:
                is_reduceable = pivots[lows[j]]<j
            return is_reduceable

        # set lows and pivots
        lows = [low(column) for column in matrix.T]

        pivots = np.ones(matrix.shape[0]).astype(int) * -1
        for i in range(matrix.shape[0]):
            for j in range(i+1, matrix.shape[0]):
                if (matrix[i,j]!=0 and lows[j]==i):
                    pivots[i] = j
                    break

        pivots = list(pivots)
        for i in range(0, matrix.shape[1]):
            while reduceable(matrix, i, lows, pivots):
                j = pivots[lows[i]]
                matrix[:,i] = (matrix[:,j] + matrix[:,i]) % 2
                lows[i] = low(matrix[:,i]) # update lows

            if lows[i]!=-1:
                pivots[lows[i]] = i; # update pivots

        return matrix

    if (self.reduced_boundary_matrix is None): # cached
        self.reduced_boundary_matrix = matrix_reduction(self.boundary_matrix)
        # self.persistence_diagram = self.get_persistence_diagram()

    return self.reduced_boundary_matrix
```

```python
def get_reduced_boundary_matrix(self):

    def matrix_reduction(matrix: np.ndarray) -> np.ndarray:

        def low(column: np.ndarray) -> int:
            nz = cp.flatnonzero(column)
            if len(nz) > 0:
                return int(nz[-1])
            return -1

        def reduceable(matrix, j, lows, pivots):
            is_reduceable = False
            if lows[j] != -1 and pivots[int(lows[j])] != -1:
                is_reduceable = pivots[int(lows[j])] < j
            return is_reduceable

        # set lows and pivots
        lows = cp.array([low(column) for column in matrix.T])

        pivots = cp.ones(matrix.shape[0]) * -1
        mat_inds = (matrix != 0).astype(bool) & (lows.reshape((1, -1)) == cp.arange(len(lows)).reshape((-1, 1)))
        mat_inds = cp.flatnonzero(mat_inds)
        row = mat_inds // matrix.shape[0]
        col = mat_inds % matrix.shape[0]
        pivots[row] = col
        del mat_inds

        for i in range(0, matrix.shape[1]):
            while reduceable(matrix, i, lows, pivots):
                j = int(pivots[lows[i]])
                matrix[:,i] = (matrix[:,j] + matrix[:,i]) % 2
                lows[i] = low(matrix[:,i]) # update lows

            if lows[i] != -1:
                pivots[int(lows[i])] = i; # update pivots

        return matrix

    if (self.reduced_boundary_matrix is None): # cached
        self.reduced_boundary_matrix = matrix_reduction(self.boundary_matrix)
        # self.persistence_diagram = self.get_persistence_diagram()

    return self.reduced_boundary_matrix
```

Looking at the code

The second cycle actively changes the matrix inside itself and I didn't find a way to optimize it.

Moreover, using CuPy made the second cycle slower.

Before CuPy	0.216
After CuPy	3.150

Analysis of results

I see the following reasons of the problem

1) We operate with small datasets (because I don't have enough memory to run the algorithm even with 100 points of dimension 10) and it's much cheaper to operate with them on CPU than on GPU
2) Only a small amount of work in this algorithm can be done in parallel
3) A lot of binary and MOD operations which is not suitable for GPU

Conclusion

Unfortunately I couldn’t speedup my code with CuPy and, moreover, I made it worse.

However, there are faster algorithms (see Ripser) and even an algorithm appropriate for GPU (see Ripser++).

The conclusion I can make is that parallelism is not a magic wand and if you use it in bad conditions the code may become worse that in one thread.

Thank you for attention