DAT470/DIT065
Computational techniques for large-scale data

Assignment 7

Deadline: 2024-05-27 23:59

In this assignment, we are going to implement ourselves linear scan nearest neighbors for finding the (hopefully unique) nearest neighbor to a query point. We will consider three datasets of different sizes:

- English pubs dataset [1], located in `/data/2024-DAT470-DIT065/pubs.csv`. The dataset records the names and coordinates of English pubs, sourced from the UK food hygiene rating data. In particular, we are going to make use of the Euclidean coordinates stored in the data, called *easting* and *northing* that record how many meters we must traverse east or north from a fixed origin to arrive at the pub. Although points on Earth's surface generally form a non-Euclidean geometry, the surface has a nice manifold property: it is locally (almost) Euclidean, so we can approximate distances in limited areas (such as England) as taking place on an Euclidean plane (this is why the Earth looks flat locally). Duplicate coordinates have been removed; the dataset contains 46,988 rows and the data is by its nature 2-dimensional.
- The file `/data/2024-DAT470-DIT065/glove.6B.50d.txt` contains 50-dimensional Euclidean embeddings of 400,000 words, trained with data from Wikipedia [2]. This is the same dataset we used in the previous assignment.
- The file `/data/2024-DAT470-DIT065/glove.840B.300d.txt` contains 300-dimensional Euclidean embeddings of 2,196,017 words, trained with data from Common Crawl [2]. This is the same dataset we used in the previous assignment.

The files `/data/2024-DAT470-DIT065/pub_queries_*.txt` contain premade queries of various sizes (tiny, small, medium, big correspond to 10, 100, 1000, 10,000 query vectors, respectively). Similar sets of query vectors can be found for `/data/2024-DAT470-DIT065/glove.6B.50d_queries_*.txt` and `/data/2024-DAT470-DIT065/glove.840B.300d_queries_*.txt`. These are complemented with files that end with `*_names.txt` that contain the correct answers to the queries: the identity of the pub or the word that is closest to the query, in terms of Euclidean distance. These *labels* can be used to verify the correctness of your implementation.

Problem 1: Batching (6 pts)

The file `nnquery.py` contains a very simple implementation of the linear scan. The input consists of two matrices X and Q, of shapes $n \times d$ and $m \times d$, respectively, where the matrix X contains the dataset by the rows, and the matrix Q contains the query vectors by the rows. We want to return the vector I of length m where I_i satisfies $I_i = \arg\min_{j \in \{1,2,\ldots,n\}} ||q_i - x_j||$ where q_i and x_j are the ith and jth row of Q and X, respectively.

The implementation is very inefficient: it is implemented as two nested Python loops, has conditionals, and performs a large number of accesses into the arrays. As such, it is a poor candidate for GPU processing.

We will seek to improve this implementation. We are using 32-bit floating-point arithmetic as that is the native (and as such, efficient) representation of data on GPUs. This has some implications which we will consider later.

(a) For improved performance, we will adjust the algorithm to make use of *batch processing*. Instead of computing each query individually, we will use array functions to process process b queries at a time. This means that we will need to process $\lceil m/b \rceil$ batches, and the last batch is potentially smaller (of size $m \mod b$).

    Consider the following algorithm: we compute an $m \times n \times d$ array D of *differences*, such that D_{ijk} is equal to $Q_{ik} - X_{jk}$. That is, the array contains all possible vector differences between the rows of X and Q. Then, computing the Euclidean distance amounts to computing the Euclidean norm along the last axis of D, and we can finally simply select the indices of the largest elements by the second axis of the result.

    As the amount of memory used by D is rather large, for larger number of queries of high-dimensional data, it is infeasible to compute D for all queries at once; hence, you will need to split Q into b-sized chunks.

    Implement the algorithm using exclusively NumPy array operations. Your implementation should not contain any Python loops or conditionals for processing individual batches (it is permissible to loop over the batches themselves, though). (4 pts)

(b) Evaluate your implementation against the original implementation. Report the query times of both implementations on all three datasets. Choose an appropriate batch size. Try all different query sizes that you can, that is, that finish in a reasonable time (you probably do not want to run big queries on the GloVe datasets with the initial implementation because they take several hours; *reasonable* could be defined in this term to be, say, 30 minutes). Report the batch sizes you used, running times, and the throughput (e.g., if you perform 1000 queries in total, the average query time would be 1/1000 of the total time). (2 pts)

Problem 2: CuPy version (6 pts)

(a) Convert the code from Problem 1 into a GPU version using CuPy (copy the data from host to device, and replace all NumPy functions and arrays with their respective CuPy versions; finally copy the results back to host). Take care to synchronize your code correctly so that the measurements are meaningful (use blocking transfers, and synchronize the default stream after computations to make sure that they finish before you record the timestamp). (4 pts)

(b) Evaluate the GPU implementation. Report the query times on all three datasets. Try all query set sizes you can, and report the throughput. Also report the times it takes to transfer the dataset and queries from host to device, and the time it takes to transfer the results from device to host. (2 pts)

Problem 3: Matrix multiplication (4 pts)

The algorithm from Problem 1 has a serious flaw: it uses an immense amount of memory which makes it inapplicable to large batch sizes, especially if the data is high-dimensional. We will try to address this and consider using the efficient matrix multiplication primitive to boost our computations. Consider the following fact: given two d-vectors $\mathbf{x}$ and $\mathbf{q}$, we observe that $||\mathbf{x} - \mathbf{q}||^2 = ||\mathbf{x}||^2 + ||\mathbf{q}||^2 - 2\langle \mathbf{x}, \mathbf{q} \rangle$.

Remembering that $XQ^\top$ can be thought of as computing all inner products between the rows of X and the columns of $Q^\top$, that is, $(XQ^\top)_{ij} = \langle \mathbf{x}_i, \mathbf{q}_j \rangle$, this suggests that we can compute the distance matrix as follows: compute the sums of squares of all rows of X and Q. Then, broadcast them into appropriate shape to get all possible combinations (e.g., row i contains $||x_i||^2$ and column j contains $||q_j||^2$). Finally, subtract $2XQ^\top$. The nearest neighbor can then be found by taking a columnwise arg-minimum.

A downside with this algorithm is that it is not numerically very stable. While we would probably be fine if we use 64-bit floating-points, using 32-bit floating-points probably yields much more errors, as the precision may be insufficient to tell nearby points apart.

(a) Implement this algorithm on the CPU using NumPy array operations. (2 pts)

(b) Evaluate the implementation on all three datasets as before (all query sizes you can, throughput, batch size). In addition, report the number of incorrectly reported points for each dataset and query set combination. (2 pts)

Problem 4: Matrix Multiplication on the GPU (4 pts)

(a) Create a GPU version of the code from Problem 3. (2 pts)

(b) Evaluate the GPU implementation on all three datasets as before (all query sizes you can, throughput, batch size, number of errors). (2 pts)

Problem 5: RAPIDS (4 pts)

We are going to use the RAFT library[1] as a baseline. Specifically, we will make use of the brute force nearest neighbors routine `knn`[2].

(a) Create an implementation that uses the `knn` function from RAFT. (2 pts)

[1] `https://docs.rapids.ai/api/raft/stable/`

[2] `https://docs.rapids.ai/api/raft/stable/pylibraft_api/neighbors/#pylibraft.neighbors.brute_force.knn`

(b) Evaluate your implementation like before (all reasonable datasets and query sets, throughput, number of failed queries). (2 pts)

Hints

- The L_2 norm can be computed using `np.linalg.norm`.
- Use the `axis` parameter to only apply an operation along a certain axis (e.g., `axis=1` applies the operation by the rows).
- You can *broadcast*[3] the results of an operation; this allows, for example, one to take all pairs of values from arrays of lower dimensionality. This makes it efficient to, e.g., compute all pairwise differences. No looping occurs on Python side which is the important bit!
- `np.argmin` allows one to find indices of minimal values.
- When you are running code using CuPy, you *must* use the Anaconda environment `cupy` (i.e., set the flag `-e cupy` when running using the `run_job.sh`).
- When measuring the time it takes to copy the data onto the device (and back), you *must* set the `blocking` argument[4]; otherwise (in an attempt to hide latency), the copy is performed in the background, and you cannot measure it.
- Also, the first transfer to the GPU will take considerably longer than transfers after that due to some delay of having to initialize the device; perform a dummy transfer (e.g., construct a small array) before doing the actual data transfer.
- You should synchronize the default CUDA stream after you've completed computation, but *before* measuring the time it takes to transfer the data back to host; otherwise, the computation might still be running and the transfer function (`asnumpy`) blocks until results are available for transfer; you can do this by issuing the command `cp.cuda.Stream.null.synchronize()`.
- You will need to use the Anaconda environment `rapids` to use RAFT.
- The `knn` function only accepts arrays that are C-contiguous (row major) and use 32-bit floating-points as the dtype. This has already been taken care of in `nnquery.py`.
- The arrays returned by `knn` are not CuPy arrays but NUMBA arrays that conform to the CUDA array interface[5]. They lack some important properties, such as slicing and reshaping. However, they can be converted to CuPy arrays without incurring a copy using `cp.asarray`.

[3]See `https://numpy.org/doc/stable/user/basics.broadcasting.html`
[4]See `https://docs.cupy.dev/en/stable/reference/generated/cupy.asarray.html`.
[5]`https://numba.pydata.org/numba-doc/latest/cuda/cuda_array_interface.html`

Returning your assignment

Return your assignment on Canvas. Your submission should consist of a report that answers all questions as PDF file (preferably typeset in LaTeX) called `assignment7.pdf`. In addition, you should provide the code you used in Problems 1a, 2a, 3a, 4a, and 5a as `assignment7_problem1.py`, `assignment7_problem2.py`, `assignment7_problem3.py`, `assignment7_problem4.py`, and `assignment7_problem5.py`, respectively. That is, you should produce the code for your implementation; you needn't produce your experiments. The code must match the interface of `nnquery.py`. Do *not* deviate from the requested filenames and do *not* produce the plots in these files; these files will be used for evaluating the quality of your implementations automatically.

References

[1] GetTheData. *Open Pubs.* 2020. URL: `https://www.getthedata.com/open-pubs`.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014).* 2014, pp. 1532–1543.