Journal of Machine Learning Research 1 (2022) 1-48 Submitted 05/9; Published 05/9

GPU Accelerated K-NN

Theodore Dyer TDYER4@JHU.EDU
Whiting School of Engineering
Johns Hopkins University
Introduction to GPU Programming

Editor: Theodore Dyer

Abstract

This paper details the implementation of my final project for Introduction to GPU Programming (EN605.617) at Johns Hopkins University.

Keywords: Machine Learning, GPU, K-Nearest Neighbors, Algorithms, Speedup, SIMD, Python, CUDA, NVIDIA, Performance, Euclidean Distance, Kernel Function, Data.

1. Project Statement

The objective of this project is to implement the machine learning algorithm k-nearest-neighbors (knn) using the GPU framework CUDA to generate performance gains over a standard CPU based execution of the algorithm. I will specifically be looking at implementing the 'classification' variant of the algorithm, in which case the goal will be to predict, based on medical data, the susceptibility of a patient to heart disease.

1.1 Hypothesis

With sufficiently large datasets, and sufficiently large values of k - utilizing CUDA to parallelize the calculations made by the standard k-nn algorithm will provide at minimum a 1.5x speedup over a CPU/in memory python execution on my local machine. In addition to this, I expect to see that calculations using a larger value for k will have even greater speedup.

2. Introduction - Machine Learning

Machine learning has been under a spotlight in recent times, given its flexibility to be applied to business problems in today's age of data availability. While definitely not an all encompassing solution to every company's needs, machine learning when used properly has been shown to provide tremendous value and utilize today's mass of data to solve unique problems - for example detecting fraud with logistic regression, or predicting susceptibility to certain diseases, etc. One drawback for some machine learning algorithms, however, is that because they can require and operate on such large amounts of data - performing these computations can become quite expensive. This is where I believe the power of a GPU could be (and historically has been) used to provide performance gains at lessened computational cost.

2.1 Why GPUs?

The reason why performing ML operations with GPUs should result in good synergy is due to the processing paradigm most machine learning algorithms follow: SIMD. Oftentimes in the context of machine learning you have a single instruction (SI) that you want to apply to your entire data set, or multiple data points (MD) - and this just so happens to directly reflect the primary advantage of using GPUs. One of the primary problems GPUs were built to solve was processing graphics, in which a goal is often to apply some transformation to a wide array of values in a graphical environment. Take for example a simulated room in which a light turns on - in this event you would likely have to apply some increase to every pixel in an environment to make it appear brighter (another SIMD problem). Therein is the overlap in these two technologies, and where I believe there should be power to leverage.

3. K-NN Algorithm

The logic of the k-nn algorithm is relatively straightforward - when considering a data point "A", the goal of the k-nn algorithm is to look at the "k" closest points to A, and determine the output value for A based on the output values of its 'k' closest points (often chosen to be an odd number for tie-breaking purposes).

Why I believe k-nn to be a great candidate to be sped up via the use of a GPU exists in the way that its computational complexity scales. K-nn is not a standard algorithm that produces a single model that can then be used to analyze future data, instead it must generate its weights (distances) for every single point in its dataset to make a prediction, and we can map out the complexity of this model according to the following:

- Step 1: For each point in our test set, generate distances to each point in the training set (utilizing a predetermined distance function).
- Step 2: Sort the list of neighbors and their corresponding distances produced from step 1 to isolate the top "k"
- Step 3 (Trivial): Determine the target class by looking at neighboring target classes.

The corresponding complexities would be:

- Step 1: O(length(test set) * length(training set))
- Step 2: O(Standard Sorting Algorithm)
- Step 3: O(k)

Breaking things down this way, the goal of my analysis is going to be to try to parallelize step 1 via the use of a GPU. This will be my focus because this calculation of distances makes up a large portion of the computational complexity of this algorithm, and additionally this step lends itself well to gpu kernel function setup.

3.1 K-NN Function Format

The K-nn algorithm is what is referred to as a nonparametric algorithm - as it doesn't have a predetermined way to describe our output classes - in the way that a linear model might

produce a prediction equation that has the target class in mind (target = ax + b), k-nn does not. K-nn instead does not have any idea what the target class even looks like until it has already produced your list of neighbors and is simply returning the majority class among those neighbors as your result.

A standard machine learning algorithm will produce and train some "model" for which you are then able to call "model.predict(x, y)", where instead k-nn treats the entire training set as its model and consults the entire set to calculate distances at the time of making a prediction.

This introduces some complexity in how we can actually implement k-nn in code, so for my implementation I chose to utilize function closer to provide a reference of our training set to an inner k-nn function where an example execution follows this structure:

K-nn = init-knn(training-data)
Result = k-nn(k, test[x])

This will likely make more sense when I talk through the structure in my code demonstration in the attached presentation.

4. Implementation Process

Because I have prior experience in machine learning, particularly in Python, I broke my development process into the following steps:

- 1: Clean, reformat, and analyze my input data using a Python Jupyter Notebook
- 2: Implement k-nn in Python
- 3: Implement k-nn in Cuda C (without GPU speedup)
- 4: Implement k-nn in Cuda C with GPU speedup

I did this for two reasons - I wanted to build out a model of the algorithm in code in a language that I was already familiar with before getting into the complexities of the GPU version to provide myself with a point of reference. Second, developing this algorithm in base Python and Cuda C provides me with a benchmark to use to measure the performance of the GPU sped up algorithm when performance testing.

During development, I made the decision in the interest of project scope to first focus purely on the execution time it takes each variant of the algorithm to perform its distance calculation step, as I ran into a number of issues handling data structures in CUDA.

5. Results

Looking back at my hypothesis stated above, I expected to see at least a 1.5x speedup in performing algorithm calculations with appropriate values of k, and an appropriately sized data set with the GPU accelerated algorithm over either of the other two.

As mentioned prior, in the interest of finishing within scope I'm going to purely look at execution time to produce distance calculations (not the following sorting step of the

Time to generate neighbor distances (1 point)

Python:	Time (seconds)
Test 1	0.11358
Test 2	0.11189
Test 3	0.11424
Test 4	0.13939
Test 5	0.11252
Avg	0.118324

CUDA (no GPU):	Time (seconds)
Test 1	0.000059
Test 2	0.000062
Test 3	0.000062
Test 4	0.000064
Test 5	0.000061
Avg	0.0000616

CUDA (GPU):	Time (seconds)
Test 1	0.000026
Test 2	0.000025
Test 3	0.000026
Test 4	0.000025
Test 5	0.000025
Avg	0.0000254

Time to generate neighbor distances (100 points)

Python:	Time (seconds)
Test 1	11.23476
Test 2	11.19123
Test 3	11.25811
Test 4	11.22819
Test 5	11.23178
Avg	11.228814

CUDA (no GPU):	Time (seconds)
Test 1	0.005587
Test 2	0.005712
Test 3	0.005547
Test 4	0.005595
Test 5	0.005562
Avg	0.0056006

CUDA (GPU):	Time (seconds)
Test 1	0.000025
Test 2	0.000024
Test 3	0.000026
Test 4	0.000025
Test 5	0.000024
Avg	0.0000248

Time to generate neighbor distances (Full test set = 183 Points)

Python:	Time (seconds)
Test 1	20.54717
Test 2	20.83191
Test 3	20.58919
Test 4	20.58582
Test 5	20.57125
Avg	20.625068

CUDA (no GPU):	Time (seconds)
Test 1	0.010195
Test 2	0.010126
Test 3	0.010173
Test 4	0.010151
Test 5	0.010181
Avg	0.0101652

CUDA (GPU):	Time (seconds)
Test 1	0.000025
Test 2	0.000025
Test 3	0.000025
Test 4	0.000025
Test 5	0.000037
Avg	0.0000274

GPU Speedup	Python	Non-gpu CUDA
1 Point	4658.425197	2.42519685
100 Point	452774.7581	225.8306452
183 Point	752739.708	370.9927007

algorithm), thus the following execution times are not inclusive of all steps of the algorithm, and the speedup produced by the GPU executions would likely be somewhat diluted over the remaining steps of the algorithm. The results seen are only reflective of (Step 1) speedup as defined above.

As we can see above, I was able to generate a considerable performance gain utilizing GPUs via CUDA. A few notes to keep in mind here, are that the Python version of these tests was executed on my local laptop, whereas both of the CUDA C versions were executed through a cloud environment in Vocareum with access to much more powerful compute hardware, so the Python results are essentially negligible.

Revisiting my hypothesis, these results far exceeded my initial prediction as I only thought I would observe around a 1.5x speedup, but when looking at the larger execution sizes the actual speedup was much greater.

These results again exist with the caveat previously mentioned that the remaining steps of k-nn do not leave much room for speedup via the use of a GPU, so these results

are essentially hyper-concentrated on the component of the algorithm that can be sped up, I will likely observe the corresponding full-algorithm GPU speedup to not maintain such a wide gap when further working on this project.

6. Conclusion

Despite the fact that I did not meet my initial goals of fully implementing the algorithm with GPU acceleration, I am very satisfied with the component that I was able to finish within the time constraints of this course project.

The performance of this algorithm's distance calculations far exceeded my expectations for what I thought I would be able to achieve. With this proof of concept I am going to continue to develop this project outside of EN605.617, as I am now very curious to see what the end result speedup of fully implementing the algorithm will be when considering steps 2 and 3 outlined above.