K Nearest Neighbors on Cudarc (Rust)

Ratnodeep Bandyopadhyay

Project Idea / Overview

K Nearest Neighbors (KNN) is a machine learning algorithm used for classification and regression tasks based on proximity to neighboring data points. In KNN, a data point is classified or predicted based on the majority class or average value of its k-nearest neighbors in the feature space. KNN can be benefitted by parallelization on GPUs as it allows for the efficient processing of large datasets and significantly accelerates the computation of distances between data points.

Rust is a programming language that offers several benefits for implementing KNN algorithms. Rust prioritizes memory safety without sacrificing performance, making it suitable for tasks where efficiency is crucial. It provides fine-grained control over system resources, facilitating optimization for parallel computing.

This implementation of KNN in Rust uses the Cudarc library, leveraging Rust's capabilities and harnesses the power of GPU parallelization for accelerated computations. The use of Cudarc implies integration with NVIDIA's CUDA platform, which is widely utilized for parallel programming on NVIDIA GPUs.

GPU Function

In my implementation of the K Nearest Neighbors (KNN) algorithm using Rust and the Cudarc library, I am using the parallel processing capabilities of the GPU to enhance the computation of Euclidean distances. Specifically, I am mapping the Euclidean distance metric onto a GPU kernel, a fundamental unit of parallel execution in CUDA. This kernel is designed to calculate the distance between a query point and multiple data points concurrently, taking advantage of the parallel nature of GPU architectures. By parallelizing this distance computation using CUDA, I aim to achieve significant acceleration in the KNN algorithm, as the independent nature of each distance calculation allows for efficient parallel execution on the GPU. This approach contributes to the overall optimization of the KNN algorithm, particularly when dealing with extensive datasets, showcasing the potential of GPU-accelerated computing in the realm of machine learning.

The library I’m using is one based in Rust and is called Cudarc. It’s relatively new, and functions well in building upon the existing CUDA API. However, kernel code must still be written in C.

Looking at the implementation, you can see that the portion of the euclidean distance metric that's been mapped onto the kernel is added as a CUDA C source, not rust.

Implementation details

Github Repository: https://github.com/UCR-CSEE217/finalproject-f23-ken-cudarc/tree/master
YouTube: https://youtu.be/fnoUHmhn4K4

Running the Code

Running the Code on the GPU

Clone the repo onto Bender
Modify the path to the input file in examples/knn-cudarc.rs

```
git checkout qsoe/gpu
cargo build --example knn-cudarc
time cargo run --example knn-cudarc
```

Running the Code on the CPU

Clone the repo onto Bender
Modify the path to the input file in src/main.rs

```
git checkout qsoe/cpu
cargo build
time cargo run
```

The inputs are predefined as constants at the top of knn-cudarc.rs or at the beginning of the main function. The parameters you should vary:

```
const FEATURE_SIZE: usize = 13;
const DATASET_SIZE: usize = 255;
const TESTSET_SIZE: usize = 48;

const K:            usize = 50;
const TOTAL_CLASS:  usize = 2;

    let path_to_dataset =
"/Users/rb/School/CS217/knn_cuda/assets/heart_data_norm.csv";
    let path_to_testset =
"/Users/rb/School/CS217/knn_cuda/assets/heart_data_norm_test.csv";
```

Evaluation/Results

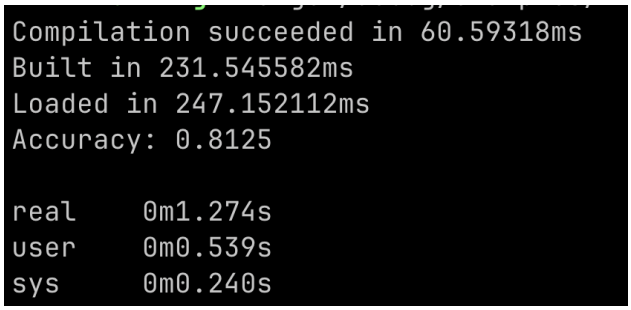

Fig 1. Successful GPU execution and timing of KNN with K=50, Dataset Size=255, feature size = 13, testset size = 48, and 2 classes

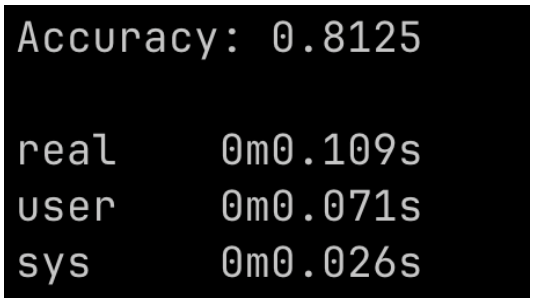

Fig 1. Successful CPU execution and timing of KNN with K=50, Dataset Size=255, feature size = 13, testset size = 48, and 2 classes

It seemed with smaller (around 500 points) dataset sizes, this model struggled to perform when compared to the CPU. Further testing may need to be performed with even larger datasets to see the benefit of the GPU parallelization. This alone indicates the overhead for transferring data to and from the GPU over these many data points is far too great to leverage any substantial gain from the parallelism. An even smaller data set with just 7 points (primarily for debugging) is added, but the results are discarded as the effect is the same.

Project Status

It can work with any number of neighbors. You can configure the K parameter to any value that's less than the training dataset size. The number of features can also be configured, as well as the test set, and data set sizes. The project is able to function as intended.

Limitations

One notable limitation of my project lies in the specific focus of the parallelized component, which optimizes only a portion of the Euclidean distance metric computation. The parallelized implementation efficiently handles the subtraction and squaring of two values, crucial steps in the Euclidean distance calculation. However, it does not extend to the square root operation, which, while not strictly necessary for K Nearest Neighbors (KNN) as the distances are compared without the need for actual distance values, does result in an underutilization of available compute resources. The omission of the square root operation in the parallelized section represents a trade-off made to prioritize computational efficiency, as avoiding unnecessary square root calculations saves significant processing time. Nonetheless, it's essential to acknowledge that this optimization is specific to the requirements of the KNN algorithm and may not be applicable in scenarios where exact distance values are needed.

Technical Challenges

In my implementation using Rust and the Cudarc library, I encountered a notable technical challenge related to the language's approach to global variables. Rust, by design, emphasizes ownership and borrowing mechanisms to ensure memory safety and prevent data races. However, this stringent control over memory management can pose challenges when dealing with global variables, as Rust tends to discourage their usage.

The specific difficulty arose when working with two variables required by Cudarc that needed to be accessed in a function called multiple times and nested deeply within the code. Rust's ownership system compelled me to carefully manage the lifetimes and ownership of these variables, which proved to be a complex task. Ensuring proper synchronization and avoiding potential data races while maintaining the required global state became a technical challenge.

Addressing this challenge involved a meticulous approach to ownership, borrowing, and lifetime annotations in Rust, adding an additional layer of complexity to the implementation. Despite these challenges, navigating Rust's ownership system ultimately contributes to a safer and more robust codebase, underscoring the language's commitment to preventing memory-related issues.

Contributions

I worked independently on this project.