Parallel DBSCAN Clustering on LiDAR Point Cloud with OpenMP and CUDA

Ethan Ky (etky), Nicholas Beach (nbeach)

April 4, 2024

1 Summary

We wrote a parallel implementation of DBSCAN clustering in CUDA on the GPU and in OpenMP on the CPU. Experiments are run on GHC computing clusters with NVIDIA GeForce RTX 2080 B GPUs.

2 Background

In our project, we parallelized the DBSCAN clustering algoritm. DBSCAN distinguishes itself from other methods such as k-means because it forms an arbitrary number of clusters based on dense collections of points. The algorithm takes in the dataset of interest as well as two hyperparameters, ϵ and minPts. One advantage is that it is much less sensitive to outliers, and performs much better on data distributions with nested patterns. It does so by differentiating between core, border, and noise points to only form clusters whose points are density connected according to the given parameters. As such, it is often used to pre-process LiDAR point clouds, which vary significantly in data density, for 3D object detectors and other computer vision tasks. The sequential algorithm is given below. More formally, the algorithm can be given by 1.

- While there are unlabelled points:
  - Get a random unlabeled point p.
  - Get neighbor set $N(p)$ of points with distance of at most ϵ from p.
  - If $|N(p)| \geq$ minPts, p is labelled as a core point.
  - If $N(p)$ contains a core point, p is labelled as a border point.
  - Otherwise, point p is labelled as a noise point.
  - If p is a core point, for each neighbor $q \in N(p)$:
    - If q has no label, repeat the previous steps.
    - If q is a core point or border point, assign it to the same cluster as p.

Algorithm 1 Sequential DBSCAN Algorithm

Require: $\epsilon >= 0$, minPts > 0
  for each unprocessed point $x \in X$ **do**
    mark x as visited
    $N \leftarrow$ GetNeighbors(x, ϵ)
    if $|N| <$ minPts **then**
      mark x as a noise point
    else
      $C \leftarrow \{x\}$
      for each neighbor point $x' \in N$ **do**
        $N \leftarrow N \setminus x'$
        if x' is not visited **then**
          mark x' as visited
          $N' \leftarrow$ GetNeighbors(x', ϵ)
          if $|N'| \geq$ minPts **then**
            $N \leftarrow N \cup N'$
          end if
        end if
        if x' is not in a cluster yet **then**
          $C \leftarrow C \cup \{x'\}$
        end if
      end for
    end if
  end for

The key data structures in this algorithm would be how we store the individual points and their cluster assignments. A key operation on these data structures is how we can efficiently query a point in the point cloud and retrieve all of its neighbors within an ϵ radius. This is one particularly expensive component, which could be parallelized in a way which avoids serially iterating through each point to determine its distance from a query point. There could also be significant parallelism in the outer loop for considering each unprocessed point. However, there are dependencies in the neighbor search for each unprocessed point. Whenever we consider an unvisited neighbor, we must also check if its been assigned to a cluster yet, which requires some careful synchronization between these threads. Parallelizing the actual neighbor search is an example of a parallelism approach which aligns with data-parallel because each neighbor candidate can share the same program instruction of determining whether it is within the ϵ radius of the query point. This also indicates that the neighbor search could be amenable to SIMD execution because several points can be processed in parallel using the same instruction.

In our implementation, we specifically apply DBSCAN to point cloud input data pulled from the KITTI vision benchmark suite. The inputs consist of 4-dimensional points, where the first three values comprise the 3-dimensional coordinate and the last value indicates the reflectance.

3 Approach

In our approach, we implemented two different algorithms from the papers we've reviewed, with one on the CPU and the other on the GPU. The former is written in C++ and using OpenMP, while the other is written in CUDA. We targeted the machines in GHC which contain the NVIDIA GeForce RTX 2080 B GPUs. We also wrote validation and visualization scripts in Python for debugging purposes.

3.1 Disjoint-Set DBSCAN

This approach parallelizes DBSCAN across the points being visited or processed. It utilizes the disjoint-set and kd-tree data structures. The former is implemented from scratch using Rem's algorithm. The latter is adapted from an implementation in the nanoflann library. The kd-tree is used for fast neighborhood queries when given the ϵ radius.

At the start, each point is initialized as a disjoint set. Each of the points and their corresponding disjoint sets are assigned to threads. For each point in each thread, we query the kd-tree to get the neighbors. Then we iterate through the neighbors and union the query point's set with the neighbor point's set, only if the neighbor point hasn't been visited yet. See the paper in the references for a more detailed description of the algorithm.

Figure 1: Road scene used for test inputs.

3.2 CUDA-DClust+

This approach parallelizes DBSCAN across the points being visited or processed. But, instead of using a kd-tree to query a point's neighbors, we can utilize each of the threads in a block to process the neighbors in parallel as SIMD execution. Over the duration of the program, we track whether each point has been processed or not. We take in a seed list size parameter, and for each iteration we sample an unprocessed point to be expanded by each CUDA block. See the referenced papers on CUDA-DClust and CUDA-DClust+ for more details.

4 Results

We tested our programs on 3 different test inputs, which are cropped from point cloud data taken from KITTI. We had difficulties running our program on the full point cloud test files of > 100000 points due to memory overflow where we could not allocate any more contiguous memory. For future work, we could consider lowering the space complexity of our implementations. Our 3 test inputs were taken from a single road scene shown in 1.

We label our 3 test inputs toy1.txt, toy2.txt, and toy3.txt. These contain 1000, 5000, and 10000 points respectively, from the original point cloud data file. We tested our code with varying parameters, one configuration with $\epsilon = 1$ and minPts $= 2$ and one configuration with $\epsilon = 5$ and minPts $= 10$. Below in 2 and 3 are some samples clusters for each of the inputs, from our sequential DBSCAN implementation.

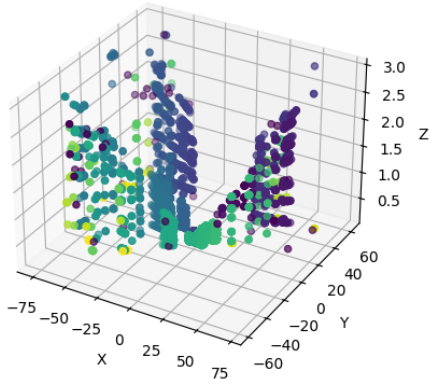

Figure 2: Cluster sample with 10000 points, $\epsilon = 1$, and minPts $= 2$.

We measured performance by speedup and total execution time. This is because speedup is important in any parallelized algorithm but for this application, total execution time is also important in aiming to be able to run the algorithm in real time. We tested and tweaked several implementations of the sequential DBSCAN and OpenMP parallelization of the Disjoint-Set DBSCAN algorithm which resulted in different speedups and total execution time. Below in 4 are the results of the speedups for our tests for the implementation with the best balance of total computation time and speedup. The baseline is OpenMP single thread optimized code. Different workloads did in fact exhibit different execution behavior. The toy3 test input with the largest number of points had significantly higher speedup compared to the smaller toy test inputs. The speedup was actually a bit shocking suggesting that single-thread OpenMP code was significantly slower than normal sequential code. The nature of the DBSCAN hyperparameters also played an interesting role in speedup and total execution time. Epsilon being 1 and minimum points being 2 resulted in less speedup but also less total execution time compared to epsilon being 5 and minimum points 10.

For the smaller input tests, toy1 and toy2, speedup was limited for the test setup with epsilon 1 and min points 2. This is due to the ratio of computation required to the communication overhead from more threads. This was examined through the command perf record which showed that a significant amount of time was spent in the OpenMP library rather than the actual execution of the DBSCAN algorithm. For the test setup with epsilon 5 and min points 10, the speedup did not seem to be limited. An important note though is that at lower thread counts speedup was limited due to workload imbalance.

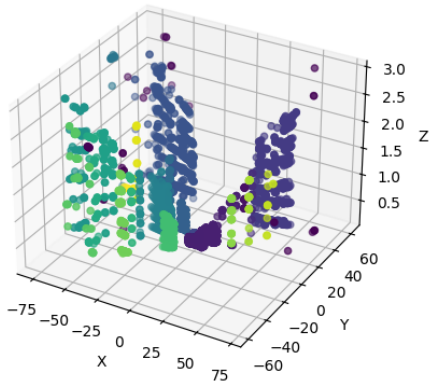

Figure 3: Cluster sample with 10000 points, $\epsilon = 5$, and minPts $= 10$.

5 References

- A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure
- CUDA-DClust+: Revisiting Early GPU-Accelerated DBSCAN Clustering Designs
- Design and optimization of DBSCAN Algorithm based on CUDA
- HY-DBSCAN: A hybrid parallel DBSCAN clustering algorithm scalable on distributed-memory computers
- KITTI Vision Benchmark Suite
- nanoflann
- STRP-DBSCAN: A Parallel DBSCAN Algorithm Based on Spatial-Temporal Random Partitioning for Clustering Trajectory Data

6 Work Distribution

- Ethan Ky - 50%
  - Wrote Python validation script.
  - Wrote Python visualization script.
  - Set up header file for point cloud kd-tree with nanoflann.

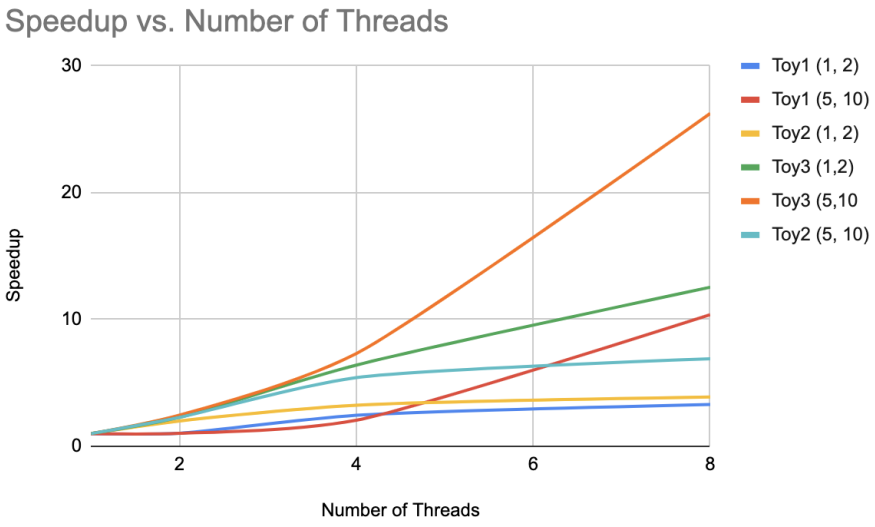

Figure 4: OpenMP implementation speedup for all 3 toy examples for $\epsilon = 1$ minPts $= 2$ and $\epsilon = 5$ minPts $= 10$.

  - Pulled point cloud input files from KITTI vision benchmark suite.
  - Wrote parallel Disjoint-Set DBSCAN implementations, on OpenMP.
  - Wrote CUDA-DClust+ implementation of DBSCAN.
  - Wrote Makefile for code compilation.

- Nicholas Beach - 50%
  - Wrote sequential DBSCAN implementation.
  - Optimized parallel Disjoint-Set DBSCAN implementation.
  - Collected performance results for Disjoint-Set DBSCAN and CUDA-DClust+ implementation.

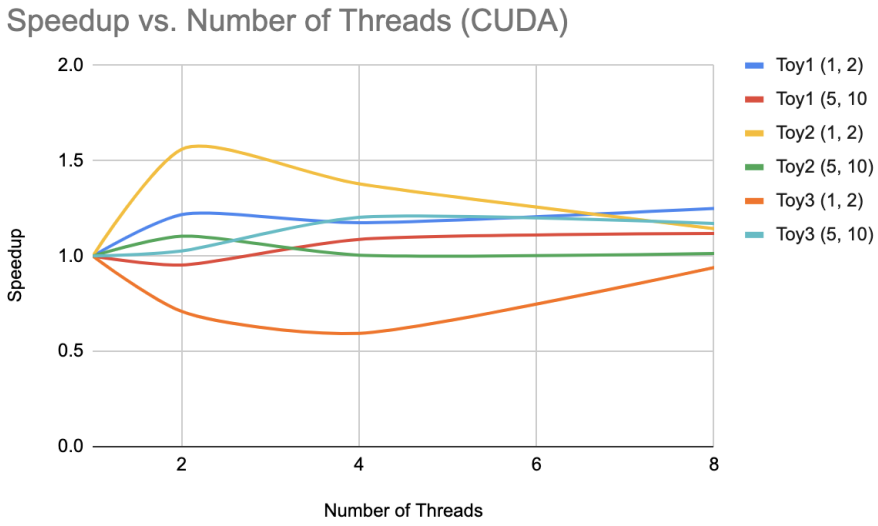

Figure 5: Cuda implementation speedup for all 3 toy examples for $\epsilon = 1$ minPts $= 2$ and $\epsilon = 5$ minPts $= 10$.