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. In the sequential algorithm, we consider each unprocessed point. Then, we check the neighborhood of points which are within an ϵ distance of the query point. If there are more than minPts neighborhoods, then the query point is labeled as a core point which can form its own cluster. We then check each of the neighbors and add them to the cluster if they are also core points. If they are not core points, then they are labeled as noise points. More formally, the algorithm can be given by 1.

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

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$(x, \epsilon)$
      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$(x', \epsilon)$
                  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
```

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. If there are more than minPts neighbors, then we iterate through the neighbors and union the query point's set with the neighbor points' sets, only if the neighbor point is also a core point. Otherwise, the query point and neighbor point pair is added to a list of merges that should be handled later. This is the cluster phase. In the merge phase, we parallelize over the pairs of the merge list and merge the clusters or disjoint sets of the points only if both of the points are core points. e use fine-grained locks on the parent mappings of each point in the disjoint-sets data structure to perform unions when there are multiple threads which may be handling a point. See the paper in the references for a more detailed description of the algorithm.

3.2 CUDA-DClust+

This approach also 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. Each of these unprocessed points can be thought of as the first point in an expanding chain, which will later be converted into a new cluster or merged into an existing one. We iterate until all points are processed. In each iteration, we launch four kernels, explained below:

- Fill Seed List Kernel: Here, we just check the cluster list to see if they're are still any unprocessed points. If there are, these are added to the seed lists, one for each CUDA block.
- Init Matrix Kernel: We maintain a collision matrix and a correction matrix. The collision matrix tracks which expanding chains have conflicting points. The correction matrix tracks which expanding chains have conflicting points with existing clusters from previous iterations. This kernel just clears both matrices for the start of the iteration.
- Expansion Kernel: This kernel performs the bulk of the DBSCAN algorithm work. It gets its chain index from the current CUDA block index. This is used to identify the chain that is being formed from this block. Then, the kernel pulls each an unprocessed point from its seed list and checks its neighborhood in parallel using the block's threads where each candidate neighbor point maps to a thread. If there are more than minPts neighbors, then we can mark the current seed point as a core point and then process the neighbor points similar to the normal DBSCAN algorithm. Except, we don't add points to the current chain unless they were either unprocessed or previously classified as a noise point. And we update the collision and correction matrices if the neighbor point was already classified as belonging to another currently growing chain or a previously constructed cluster.
- Cluster Merging Kernel: This kernel takes the cluster list, containing the chain and/or cluster classifications of each point, as well as the collision and correction matrices. Here, we assign the same chain index to each colliding chain according to the collision matrix. And if the collision matrix says any chains have conflicting points with existing clusters, those cluster labels are assigned to the chains. If there are multiple such conflicting clusters, each of their inhabiting points are assigned the same cluster label. In the case of no collisions with existing clusters, the chains are just assigned new cluster labels.

Finally, we use OpenMP to parallelize over the points on the CPU and relabel the cluster points to be readable for an output file (labels with value

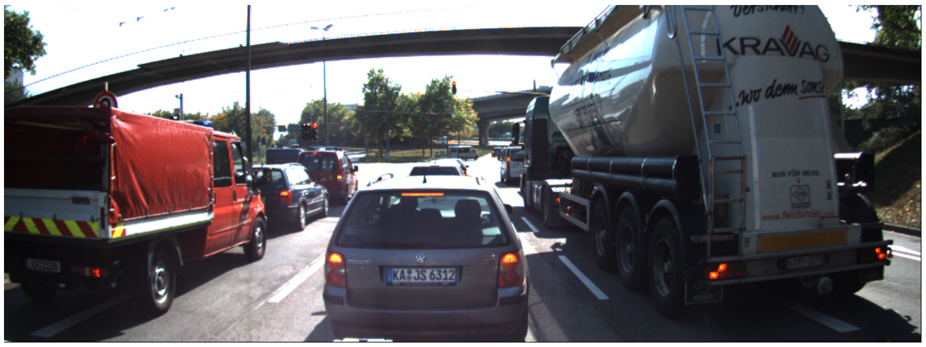

Figure 1: Road scene used for test inputs.

less than the number of points). For our implementation, we fix the maximum seed list size at 1024, the number of blocks or chains at 256, and the number of threads per block at 512. See the referenced papers on CUDA-DClust and CUDA-DClust+ for more details.

Originally, we would handle the merging of clusters, using the collision and correction matrices, on the CPU with OpenMP. However, this required significant data transfer costs because we needed to copy the cluster list, collision matrix, and correction matrix over from GPU device memory. Transitioning to performing cluster merging to a kernel instead significantly improved the performance.

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 sample clusters for each of the inputs, from our sequential DBSCAN implementation.

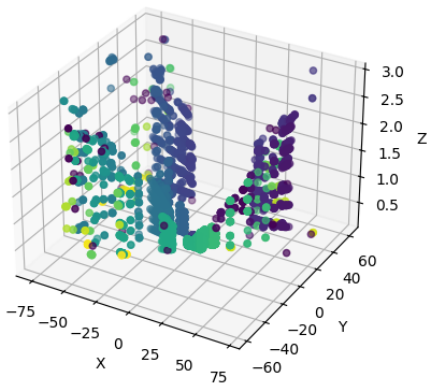

Figure 2: Cluster sample with 10000 points, $\epsilon = 1$, and minPts = 2.

4.1 Disjoint-Set DBSCAN

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. The

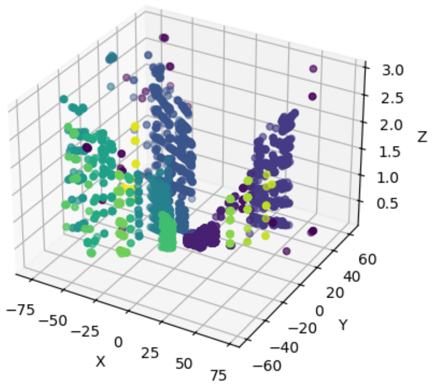

Figure 3: Cluster sample with 10000 points, $\epsilon = 5$, and minPts = 10.

toy3 speedup never seemed to be limited and was much higher than expected. We speculate that this is due to poor optimizations from our single-threaded OpenMP baseline.

Figure 5 is a plot of the workload distribution for the parallel clustering and merging steps of our OpenMP implementation. These results were taken from the toy3 input. By thread utilization, we refer to the execution time of the thread divided by the program execution time. These results reinforced our conclusions of speedup being possibly limited by workload imbalance.

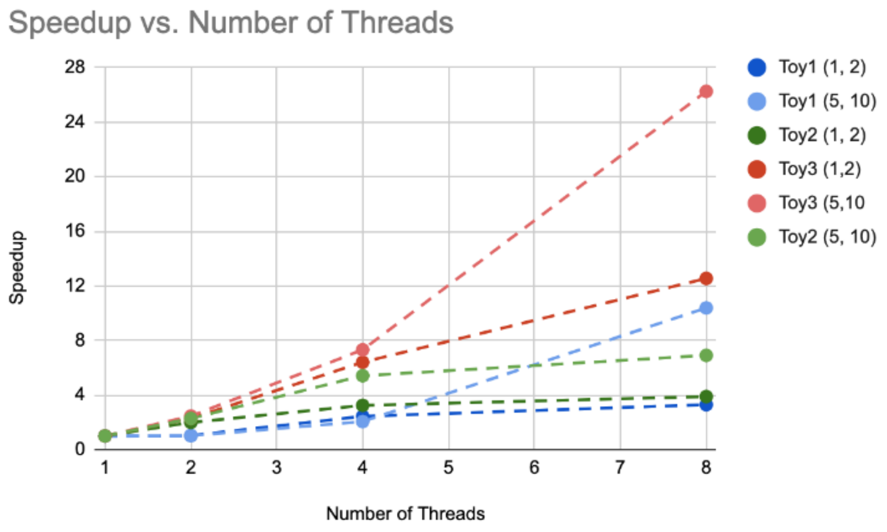

Figure 4: OpenMP implementation speedup for all 3 toy examples for $\epsilon = 1$ minPts = 2 and $\epsilon = 5$ minPts = 10.

4.2 CUDA-DClust+

Below in 6 are the results of the speedups with our implementation of CUDA-DClust+. These speedups are with respect to the execution time of our single-thread OpenMP implementation. Note that the final cluster labels are computed using OpenMP to parallelize across the points using the threads. So, the number of threads in the figure indicates the number of CPU threads. In most of the test inputs, the speedup stays the same across the number of threads, which makes sense as the bulk of execution is on the GPU, which is not impacted by the number of allocated CPU threads. However, we did also notice that the speedup increases rapidly for the toy3 example with parameters of $\epsilon = 5$ and minPts = 10. This is confusing behavior, but we speculate it could be cause. Based on our observations from the CPU implementation's results, we were not surprised to see better performance for the larger inputs. Additionally, we are not surprised to see that the speedup is better for larger ϵ and minPts. Our justification is that larger minPts suggests that fewer seed points can be labeled as core points and be expanded, so the amount of work required for each seed point would generally be lower.

Figure **??** shows the workload distribution across the first 16 CUDA blocks for the first iteration of cluster expansion. We define utilization similarly to our results from the OpenMP implementation. It should be the execution time of each block divided by the execution time of the kernel after synchronization.

Overall, throughout the tested cases, the speedup is lower than expected. We speculate that the speedup is perhaps limited by communication overhead.

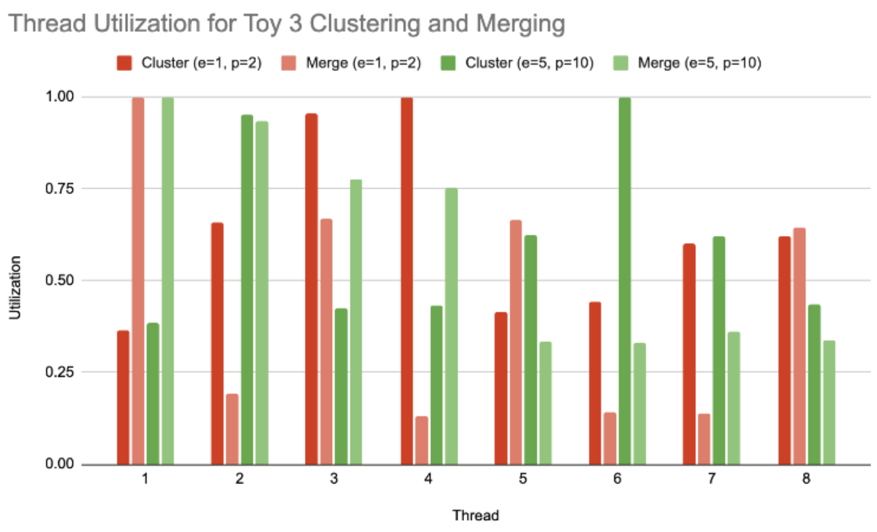

Figure 5: Workload distribution for cluster and merging steps of Disjoint-Set DBSCAN implementation.

When running the expansion kernel, each of the blocks may be performing many reads and writes from the maintained collision and correction matrices, which are stored in global device memory. There may also be some divergent execution in the intra-block parallelized processing of the neighbor points. The reason is that some of the points may only need to be reassigned a cluster labels, while others may require an update to the correction matrix. The times for these cases vary. From perf profiling, a majority of the GPU execution is in the expansion kernel so that should be the primary target of optimization.

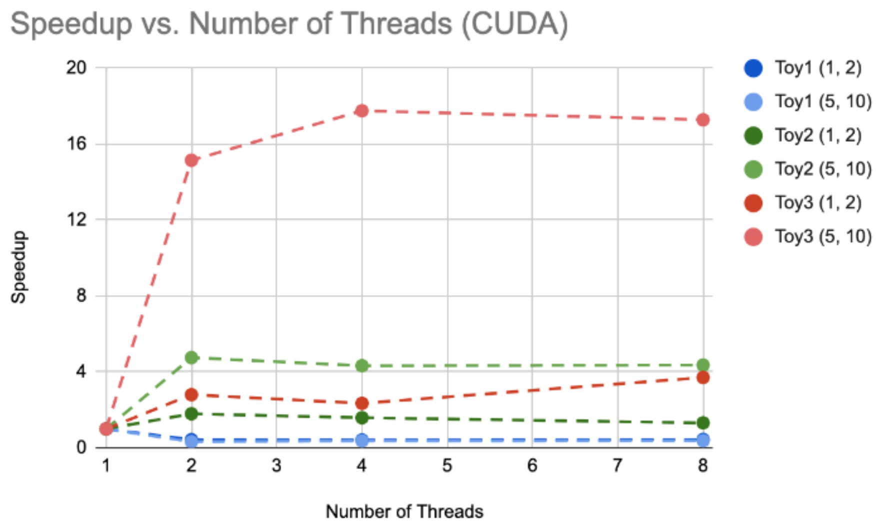

Figure 6: CUDA implementation speedup for all 3 toy examples for $\epsilon = 1$ minPts = 2 and $\epsilon = 5$ minPts = 10.

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.

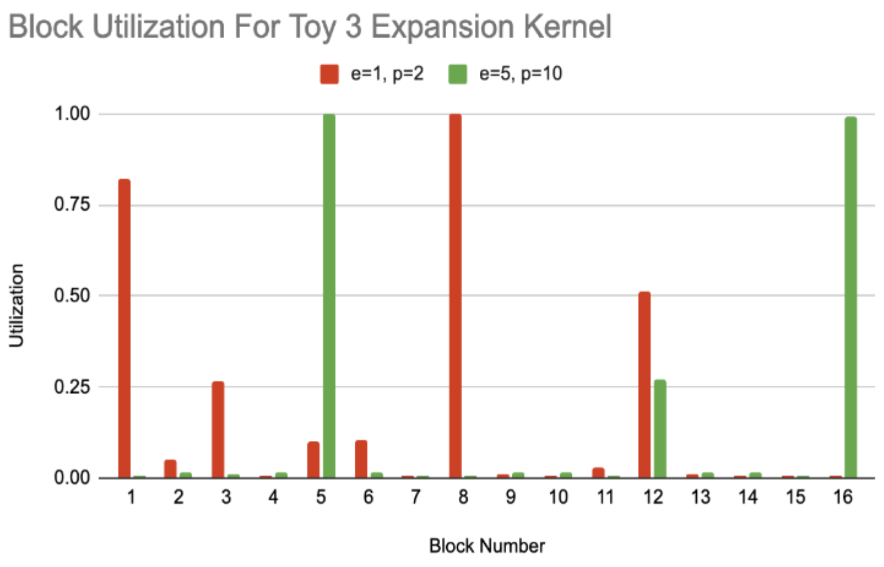

Figure 7: Workload distribution for cluster expansion kernel of CUDA-DClust+ implementation.

  - Wrote Python visualization script.
  - Set up header file for point cloud kd-tree with nanoflann.
  - 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.