Parallel LiDAR Depth Image Rendering Tool

Project Report

Hengrui Zhang, Haowen Shi

1 Summary

We implemented a LiDAR depth image rendering tool, which takes in LiDAR point clouds in world frame and renders a depth image corresponding to camera's field of view. We implemented sequential (one process on single CPU core), OpenMP, and CUDA versions, and benchmarked the performance of each of the implementation.

2 Background

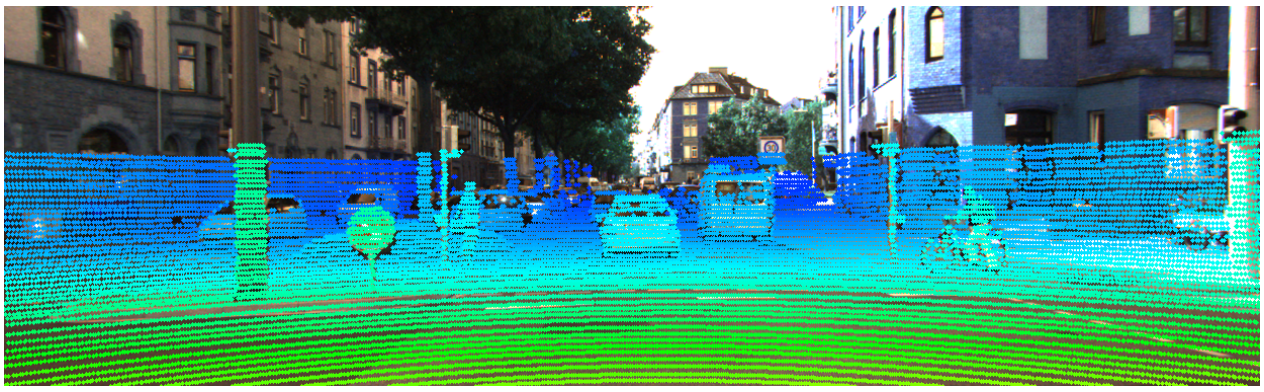


Figure 1: Example output of LiDAR depth rendering

Below are the key data structures we used in the implementation:

1. Point Cloud data structures
    - Point: `PCL::PointXYZ`, defines a single point with it's 3D location.
    - Point Cloud: `PCL::PointCloud<Point>`, defines a point cloud with type of `Point`. It's essentially is a `std::vector` of type `Point`.
    - Cloud Window: `std::deque<PointCloud>`, defines the fixed window of frames of point clouds that we want to render. Maintained with a double ended queue.
2. Render Parameters
    - `tf2::Transform`: defines the transformation (rotation and translation) between LiDAR frame and camera frame. This is used to transform points from LiDAR frame to the camera frame.
    - `sensor_msgs::CameraInfo`: defines the projection matrix of the camera. This is used to project the transformed 3D points in camera frame to 2D image plane, generating a depth image.
3. Result Image
    - `cv::Mat`: defines the result depth image. The pixel values of the image will be the projected depth (take minimum when multiple points projects onto same pixel).

Software Pipeline with Data Structures

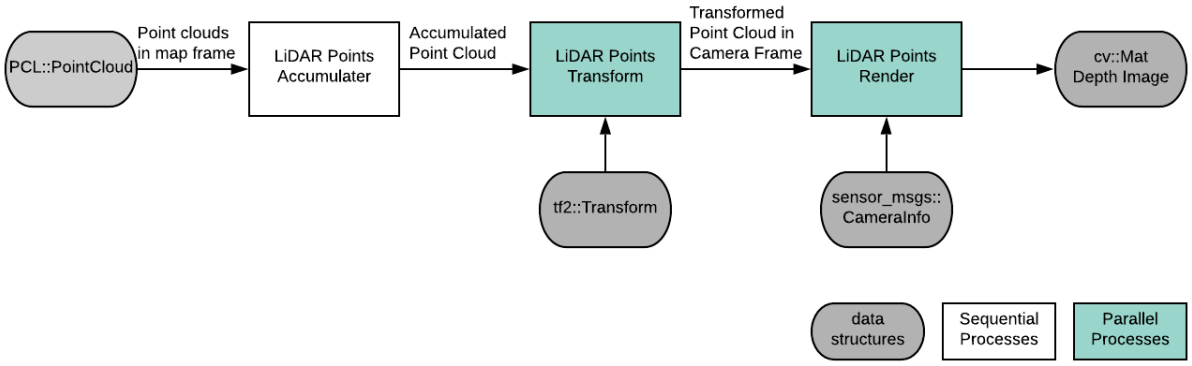


Figure 2: Software pipeline for LiDAR depth image renderer

There are three main procedures in the pipeline:

1. LiDAR Points Accumulate: This procedure maintains the `CloudWindow` data structure. It takes in each new LiDAR scan point cloud in `world` frame, adding it to the back of the double ended queue, and remove the oldest point cloud in the front of the queue. This is a quick operation and cannot be paralleled.

2. LiDAR Points Transform: This procedure takes in the accumulated point cloud in `world` frame and the current transformation from the world frame to the camera frame (robot's state estimate), then transform all the points in the accumulated point cloud from `world` frame to `camera` frame. This procedure can be done in parallel across all points, as the operation on each of the points is the same.

3. LiDAR Points Render: This procedure takes in the `CameraInfo` (camera projection matrix) and the 3D point cloud in `camera` frame, and generates a 2D depth image in camera's field of view. This procedure can be done in parallel across all the points as the operation on each of the point is the same.

Workload Break Down:

The main parallelism we can exploit in our program is parallel over all the 3D points, in both transformation and projection phase. This is a data-parallel scheme as all the data follows the same sequence of operations and does not have dependencies with each other. One dependency in the pipeline is at the end when we are propagating the final depth image, since we need to make sure only the smallest depth gets registered in the image in the case of multiple points project onto same pixel location.

3 Approach

Technologies Used

Below are the main technologies we used to implement our tool:

- Language: C++, (bash, python for benchmark and visualization)
- Parallel Interfaces: OpenMP, CUDA
- Machine: AMD Ryzen 7 3700X (X86-64 8-Core 2-Way SMT) , Nvidia GeForce 2070 Super (2304 CUDA Cores)
- Robot Operating System (ROS): A set of libraries and tools for robot software architecture, we used this to construct our software pipeline and interface with data.
- Point Cloud Library (PCL): A cross-platform, large scale, open project for point cloud representation and processing.
- OpenCV: Used for image representations and pixel manipulations.

Parallel Mapping Approach

We implemented our parallel versions in both OpenMP and CUDA. For the parallel region of the program, we aim to parallelize over all the points, since the same set of operations are applied on each individual point.

OpenMP

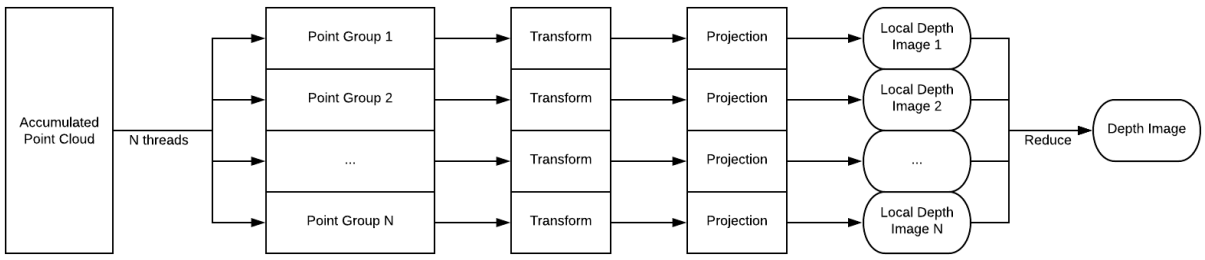


Figure 3: Parallelism with OpenMP

work division: To realize the data-parallel in OpenMP interface, we first identify the main for loop that performs the `transformation` and `projection` to all the points. We then apply `pragma` directives to spawn a gang of workers and statically divide the total points in the accumulated cloud to N threads number sets of points.

parallel region logic: Each thread will perform sequential operations on the assigned group of points, same as the sequential version, and writes the rendering result to a thread local image buffer so that we don't need to synchronize across threads in the parallel computation.

reduce final result: Note that the final depth image needs to have the minimum depth value in case of there are multiple points projected onto the same pixel location. The correctness is guaranteed sequentially within a thread, but the local image buffers from different threads could have different values in the same pixel location.

We utilized OpenCV's builtin functions to reduce multiple local image buffers into the final depth image, which have SIMD optimizations in them and are very fast. We did not dig into the specific details of the OpenCV implementations as it's not the main part of our project.

CUDA

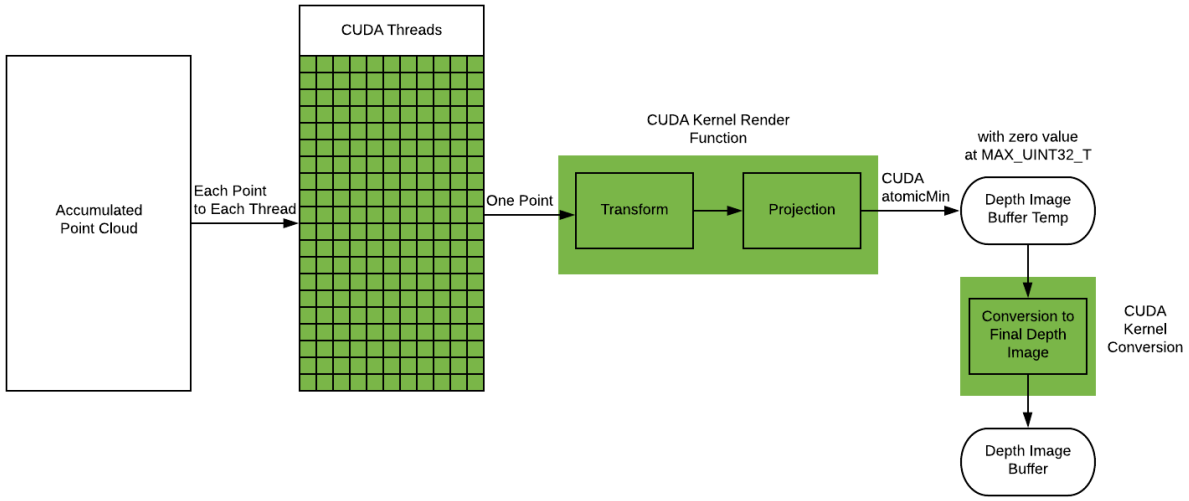


Figure 4: Parallelism with CUDA

work division: We mapped each point in the accumulated point cloud to a CUDA thread in the GPU implementation.

parallel region logic: The CUDA kernel function will apply `Transforma` and `Projection` in each thread (to each of the point). To ensure correctness, `Depth Image Temp Buffer` is set to `MAX_UINT32_T` initially, and each thread will use `cudaAtomicMin` to compare and swap the smaller depth into the temporary buffer atomically.

convert to final result: Finally we need to convert the `Depth Image Temp Buffer` to `Depth Image Buffer`, where we need to check each pixel in the temp buffer and change it to

value `0` when we encounter `MAX_UINT32_T` (a max value indicates there's no 3D point being projected on to that pixel location and thus should have 0 depth).

We mapped each pixel in the `Depth Image Buffer` to a CUDA thread, and used GPU parallel to accomplish this conversion procedure. Since each thread reads and writes independent memory location (different pixels in image buffers), we don't need to worry about race conditions in this phase.

Changes to Serial Program

We initially used the PCL builtin point cloud transformation function and found out that the PCL transformation took a long time in our parallel modification and was the bottleneck for our performance.

Since we can't modify the PCL builtin transformation, we change it with our own transformation function which manually applies transform to each of the point. Then, we have more control over the pipeline and can apply parallel optimizations in the transformation phase as well.

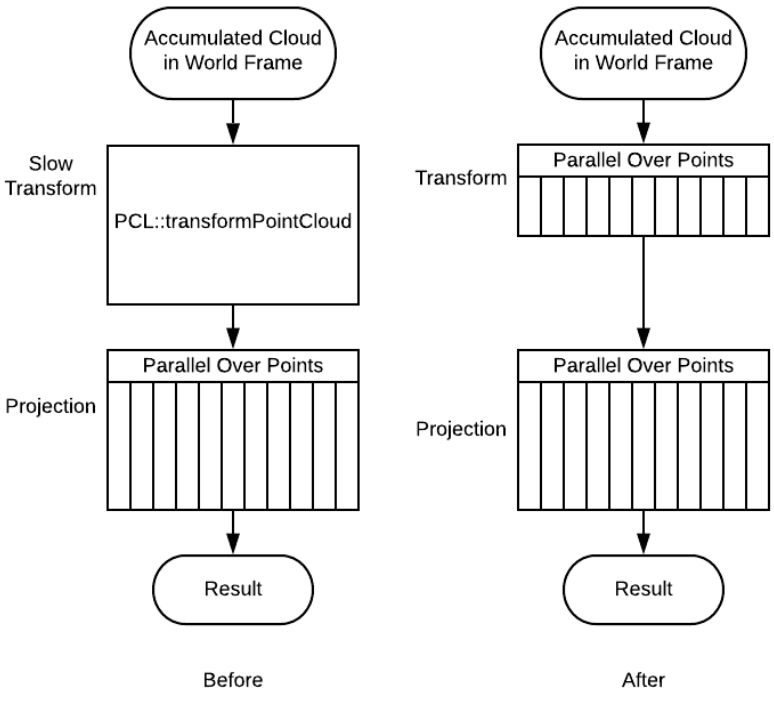


Figure 5: Change of transformation function

4 Results

Here is a video showcasing our LiDAR depth renderer on a pre-recorded dataset:
https://youtu.be/pxZWXLIlUps

The metric we will be using to evaluate the performance of our implementations is the time taken for the renderer to finish rendering one frame given similar workload which is the number of points in the accumulated point cloud. We will refer to this metric as `Frame Time`. We measure the frame time as the wall-clock time instead of CPU time because our goal is to make the rendering real-time. The `Frame Time` is measured using C++ built-in wall-clock wrapped around the render function call and it includes the computation time as well as the overhead cost incurred by the chosen implementation.

4.1 Dataset

The dataset we will be using to benchmark our implementations is collected from the ground robot that the CMU & OSU Team Explorer built for the DARPA Subterranean (SubT) Challenge.

4.2 Experimental Setup

In a realistic scenario our renderer will not be the only task running on the robot. However, some of the other computation intensive tasks on the robot have unpredictable, fluctuating performance characteristics. To get an accurate measure on our implementation speed-up, we decided to *isolate-and-test* the renderer by pre-recording information generated by other tasks and playing them back for the renderer's consumption.

To make sure our measurements and comparisons are meaningful, We play back the same part of dataset each time for different implementations or parameters. We test the serial implementation once, OpenMP implementation for all core count configurations, and CUDA implementation once and gather the frame time for each render call. The raw measurements gets written into a CSV file which will later be parsed by a visualization script we wrote to make comparisons. To make the benchmark process easier for us we wrote a test automation script to automatically run all the tests.

4.3 Workload Behavior

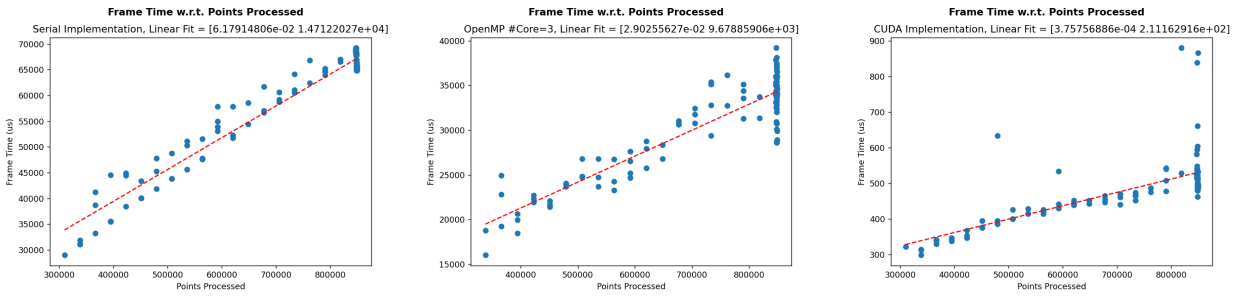
The work required to transform each individual point from 3D to 2D is constant, and the only factor affecting the renderer's workload is the number of points in the accumulated point cloud. We accumulate point cloud readings from the LiDAR across a sliding window, and in the beginning there would be less points to process than later when the accumulated point

cloud is larger. In all implementations we observe the behavior of frame time increasing when the robot increases its speed, especially from stationary to a moving state. When the robot slows down the accumulated cloud size will drop again because we exclude spatially redundant points from the sliding window. A stationary robot means the points captured from the LiDAR would be similar across frames and the total accumulated point cloud size would decrease.

To summarize, the renderer's workload is proportional to number of points in the accumulated point cloud, which is affected by the following:

- Time since last stationary
- Velocity of the robot

In our dataset, there is a period of time where the robot is moving at a uniform speed, giving us a uniform workload. We use the `Frame Time` recorded in that period of time to evaluate the performance of different implementations.



Serial OpenMP (#Core=3) CUDA

Figure 6: Workload is proportional to number of points processed

Figure 6 visualizes the frame times plotted against number of points processed by the renderer. We can see they are linearly correlated across all three implementations. This is within our expectation.

4.4 Performance Increase

Again, our performance metric is the time taken for all 3D point cloud points to be transformed and rendered onto each 2D image. Figure 7 illustrates the speed up we were able to get out of the OpenMP and CUDA implementations compared to the baseline serial version in green.

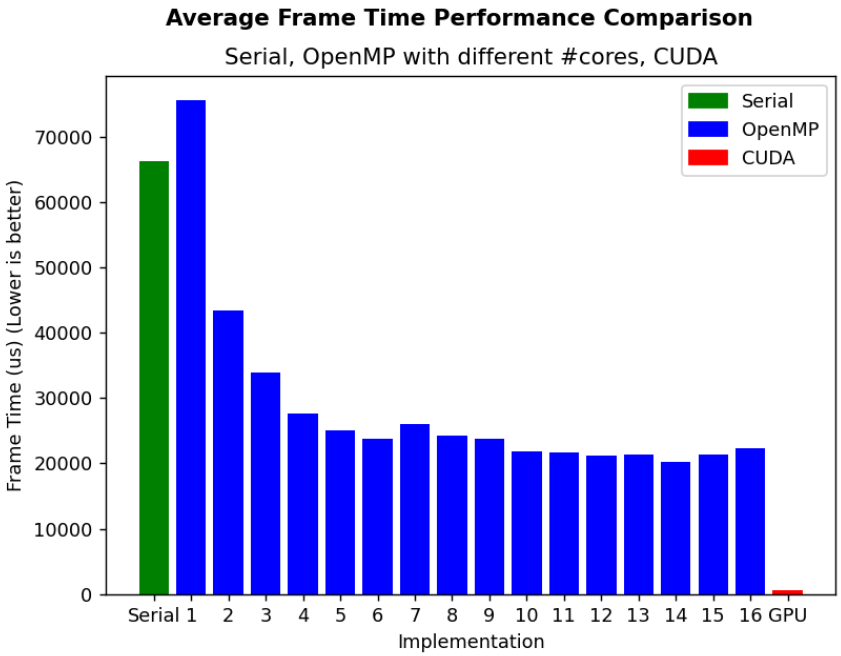


Figure 7: Performance comparison between different implementations and parameters

Implementation	Serial	OMP1	OMP2	OMP4	OMP6	OMP8	OMP16	CUDA
Frame Time	66356	75631	43447	27658	23856	24189	22373	553
Frame Rate	15.1	13.2	23.0	36.2	41.9	41.3	44.7	1808
Speed Up	1	0.877	1.527	2.399	2.782	2.743	2.966	**120.0**

Table 1: Average frame time (microseconds)

OpenMP Acceleration

As we can see in figure 7, the performance increase we get is sub-linear with respect to number of cores employed to split the workload. We do not gain much more performance increase after using more than 6 cores. We checked the CPU utilization during these computation and the CPU's were not fully utilized. Utilization for each core hovered around 30% There was no synchronization involved so we suspect the arithmetic intensity is not high enough for the cores to be fully utilized but we did not have the time to explore the actual reason. Another interesting phenomena we found is that if we designate one core to the OpenMP implementation, it runs slower than the serial version. After inserting timing functions we confirmed this was due to the unnecessary extra work spent reducing the result produced by the only core to the global buffer. There was no motivation for us to fix this edge case.

The best performance increase we got out of our OpenMP implementation is 3.28x at 14

cores.

CUDA Acceleration

We achieved unexpected performance gain from GPU acceleration. Using CUDA we managed to get a whopping 120.0x speedup compared to the sequential baseline version. In figure 7, the CUDA's frame time bar plot is almost invisible. Our CUDA implementation was able to render the typical uniform workload of 800,000 points per frame in around 500 microseconds, which converts to 2000 FPS. This is significantly higher than the baseline's 15 FPS processing speed and enables our depth renderer pipeline to run at much higher refresh rate in real time.

4.5 Correctness

Figure 8: Output snapshot. From top to down: Serial, OpenMP, CUDA

We verify the correctness of the parallel implementations by comparing the rendered depth image output against the serial baseline. Figure 8 shows the depth image produced (2nd and 3rd from top) on the left match serial output correctly.

4.6 Platform of Choice

We initially wanted to test our implementation on ARM64 based Nvidia Xavier development board with 384 CUDA cores. However, due to some unexpected events we did not have access to the Xavier board mounted on the robot. We later decided to test our implementations on a X86-64 based PC with 8-core 2-way SMT, and 2304 CUDA cores.

Our consideration is to have a portable implementation and test them on a PC first before migrating to the Xavier board with CUDA capability. We expect decent performance increase on Xavier but not as much as we got on a very powerful desktop graphics card.

Conclusion

According to our benchmark the conclusion is clear that GPU acceleration works much better than OpenMP CPU acceleration for our workload. The sheer number of points that need to be processed and the simplicity of the computation make heterogeneous acceleration very effective. With our CUDA implementation we were able to achieve real-time depth image rendering that we aimed for.

5 References

1. J. Zhang, M. Kaess, and S. Singh, *Real-time Depth Enhanced Monocular Odometry*, Paper Link
2. Camera-Lidar Projection: Navigating between 2D and 3D, blog post

6 Project Code Link

Please refer to this GitHub repository for our implementation.
Repository name: lidar-depth-image-renderer

7 Division of Work

Equal work was performed by both project members.