Leveraging Routing Information to Enable Efficient Memory System Management of ScanMatch in Autonomous Vehicles

Hengyu Zhao[‡], Haolan Liu[‡], Pingfan Meng[†], Yubo Zhang[†], Michael Wu[†], Tiancheng Lou[†], Jishen Zhao[‡]
[†]Pony.ai, [‡]University of California, San Diego

***Abstract*—In the autonomous vehicle (AV) system, localization is a safety critical module as it is responsible for identifying the current location of an AV, and ScanMatch is a common approach to obtain highly accurate location. To perform ScanMatch, an AV leverages vision sensors to continuously scan the surrounding environments and compares with a pre-stored high definition (HD) map, until the detected landmarks match with those on the HD map. We study ScanMatch by performing a field study with level-4 AV fleets over months, and we identify the ScanMatch is mostly bounded by inefficient memory access. To enable memory efficient ScanMatch, we propose a memory system optimization framework, and it leverages the routing and scanning information to guide memory management of AV computing systems.**

I. Introduction

The AV (autonomous vehicle) software stack includes three main driving tasks: localization, perception, and PlanControl. In the state-of-the-art high automation AV system, the ScanMatch is a mainstream approach to get the accurate position information [1], [5]. The ScanMatch module localizes AVs based on the static landmarks (e.g., buildings, traffic signs), for instance, human drivers often navigate themselves with static landmarks. In most high-level AV systems, the ScanMatch is composed of three components: high definition (HD) map, scanning and matching. The HD maps store the static information of the environments scanned by various sensors (e.g., LiDARs, cameras, radars, in our work, we only use LiDAR for ScanMatch). The scan stage collects real-time sensor information and the matching stage aligns the scanned sensor data with HD maps.

By performing a field study and running level-4 AV fleets, we observe that the conventional memory system is inefficient for ScanMatch tasks, and almost 80% latency is consumed by memory accesses. We identify three memory inefficiencies in ScanMatch. First, it is unnecessary to process the (1) moving objects (e.g., vehicles, pedestrians) and (2) distant static objects, because ScanMatch only needs static and nearby information; Second, the memory locality is highly sensitive to the AV driving direction, and point clouds are streaming randomly to processors that results in locality issues. Third, we observe that the HD map storage contains significant sparsity.

To address these three inefficiencies, we focus on optimizing the memory system in the ScanMatch and present following designs: First, we propose a point cloud filter that removes the misleading moving and distant point clouds. Second, we propose a route-aware HD map remapping scheme, that improves locality by remapping HD maps based on AV moving directions. Third, to further improve the locality, we propose an adaptive HD map loading scheme, which intelligently selects appropriate method to load HD maps to the on-chip memory. Moreover, we implement a multi-level block storage mechanism that efficiently store sparse HD maps.

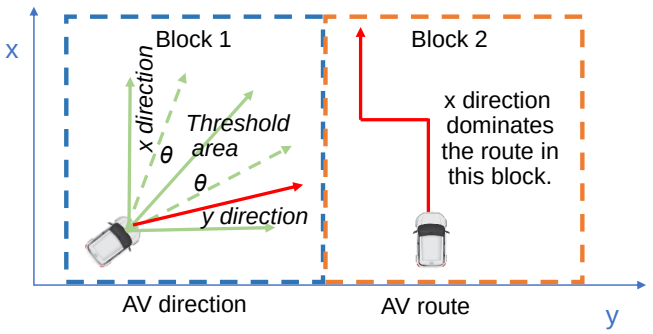


Fig. 1. Determine which storage type we should adopt.

II. Design

A. Point Cloud Filter

We propose a point cloud filter to remove misleading point clouds, which includes two functions: (1) removing point clouds that belong to moving objects and (2) removing point clouds that are out of the region of interests. First, our point cloud filter identifies if a point cloud can be found in HD maps, because any point cloud that can be found in the HD maps belong to static objects, and they should be considered during ScanMatch. Second, to remove the point clouds that fall out of the region of interests, the point cloud filter initially set a range for the region of interests. Any point cloud not in the region of interests should be dropped for ScanMatch purpose.

B. Driving Route-Aware HD Map Remapping

We propose a driving route-aware HD map remapping scheme, which modifies HD maps organization based on different AV routes and directions. Figure 1 and Figure 2 (a) shows how this scheme works. In Figure 1, we show an

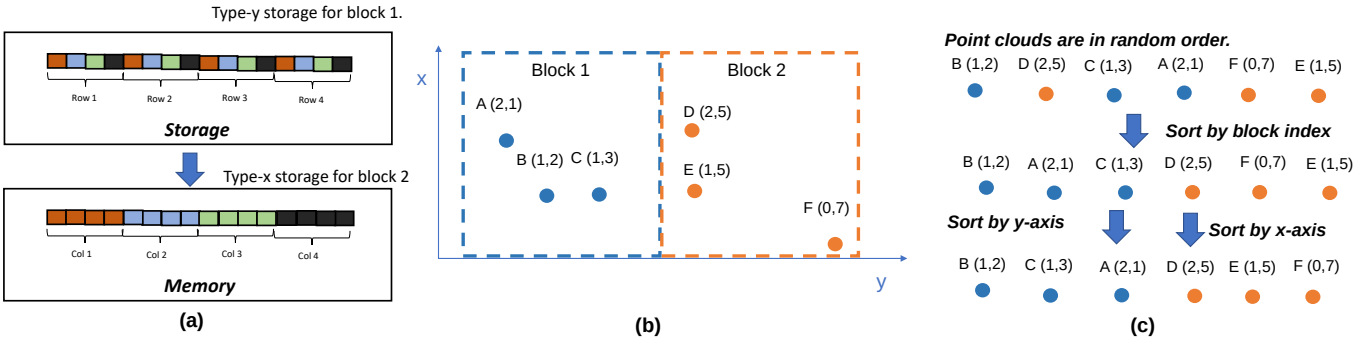


Fig. 2. (a) Transform type-y storage to type-x storage; (b) the LiDAR scans point clouds; (c) point clouds sorting.

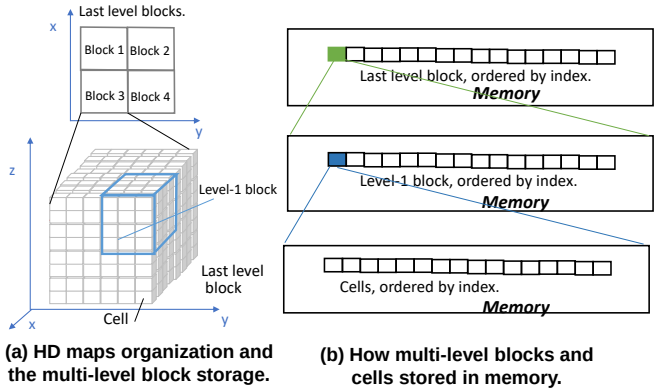


Fig. 3. Multi-level block storage mechanism.

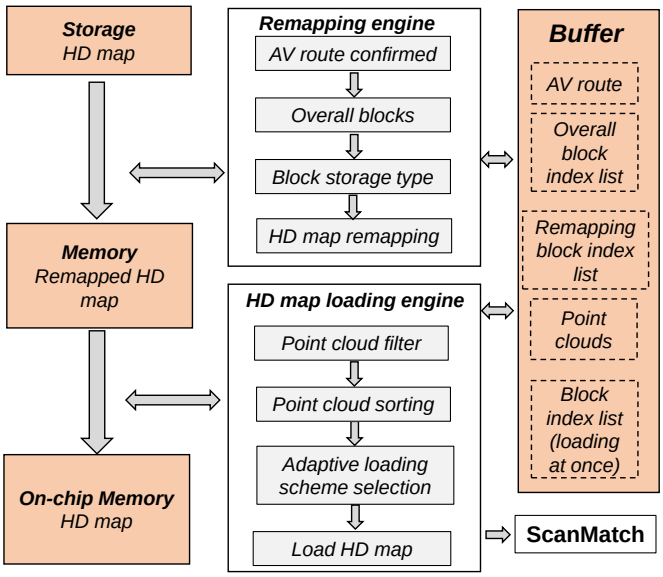


Fig. 4. The architecture overview.

example of the AV route. In block 2, the AV drives in two directions continuously: the x-axis direction and the y-axis direction. The HD map blocks are stored in the order of y-axis, so the better memory locality could be achieved when the AV is driving along y-axis. We define such a storage type as *type-y storage*. Accordingly, if we store HD map blocks in the order of x-axis, then we define it as *type-x storage*. We remap the HD map blocks to either type-x storage or type-y storage, when the AV is moving along the x-axis or y-axis.

C. Adaptive HD Map Loading

We propose this adaptive HD map loading scheme, which includes the point cloud sorting scheme and facilitates that the AV system loads HD maps from off-chip memory to on-chip memory, and alleviates the point cloud streaming randomness. Figure 2 (b) and (c) show our proposed point cloud sorting scheme, which sorts point clouds in specific orders. We sort the point clouds according to their (1) block index and (2) y-axis coordinates (type-y storage) or x-axis coordinates (type-x storage). Figure 2 (b) and (c) shows how we sort the point clouds. Assume the LiDAR scans six point clouds and streams them in random order: B, D, C, A, F, E. Next, we sort the six point clouds according to their block index: we gather B, A, C because they belong to block 1, and gather D, F, E because they belong to block 2, because the HD map can be loaded block by block. Finally, to facilitate the automatic update scheme, in block 1 (type-y), we sort the point clouds according to their y-axis coordinate; in block 2 (type-x), we sort the point clouds according to their x-axis coordinate.

III. Implementation

A. Multi-level Block HD Map Storage

Inspired by [3], we propose a multi-level block storage mechanism that leverages the significant sparsity in point cloud based HD maps. We first consider the entire HD maps as a huge 3-D block. Then, we split it into multiple equal-sized smaller blocks, until we reach to the smallest block size as defined. The HD maps are highly sparse, motivating us to not store these empty blocks but record their indexes that notify users which block is empty. We outline the high-level design of the proposed multi-level block storage mechanism in Figure 3. First, we show a bird view of how the HD maps are organized

TABLE I
THE ARCHITECTURE CONFIGURATIONS OF THE SIMULATOR AND REAL MACHINES.

Real machine	
CPU	Intel Xeon processor
GPU	NVIDIA Titan V (Volta architecture)
Main memory	12GB HBM2
Operating system	Ubuntu
Simulation	
Storage	1TB
On-chip memory	4 MB SRAM
Main memory	12GB DDR4

in the real world in Figure 3 (a). We split the entire real world HD maps into multiple last level blocks, and each last level block includes a bunch of 3-D cells. Furthermore, in each last level block, we generate multiple level-1 blocks. We show how we organize HD maps in the memory in Figure 3 (b). We store each level of blocks with sequential addresses in the memory, and we skip storing all empty blocks.

B. Architecture Overview

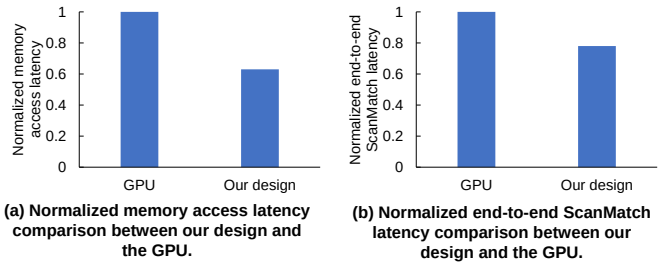
The overview of the proposed architecture is shown in Figure 4. This architecture implements three key functions. First, it pre-stores and pre-processes HD maps in the multi-level block storage, as it lets HD maps fit in limited storage capacity. Second, given any AV route, it identifies the level-1 blocks that will be covered, and remaps HD maps into required storage type, then loads HD maps to the memory from storage before the AV departs. Third, it loads HD maps to the on-chip memory with appropriate loading schemes and the point cloud filter removes misleading information. The architecture implements two key components: the remapping engine and the HD map loading engine. It also implements a buffer to store some temporary data.

IV. EVALUATION

A. Experimental Setup

Configurations of the simulator and real machines. Table I shows the configuration of the simulator. We model the DDR4 memory as the main memory, SRAM as the on-chip memory. We set LRU (Least Recently Used) as the on-chip memory update policy. Our simulator is built on CACTI [6], and it also adopts memory statistics in [2], [4]. We use this simulator to evaluate the performance and energy consumption of our design. Moreover, We also exploit TITAN V GPU to perform the ScanMatch and compare the result with our design. The GPU configuration is shown in Table I.

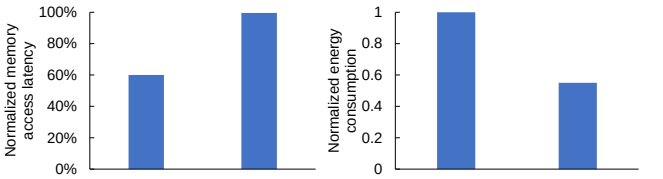
Data. We perform a field study, by running real industrial level-4 autonomous vehicle fleets to collect point cloud data and HD maps. Our field study lasts for three months, and we collected more than 2000 miles driving data. The data includes: the coordinate of each point cloud, cell value and cell coordinate in the HD maps. The HD maps are collected and built into multiple blocks during the daily road tests. The default block size is 512 cells $\times$ 512 cells $\times$ 512 cells. The default cell size is 0.125m $\times$ 0.125m $\times$ 0.125m.



(a) Normalized memory access latency comparison between our design and the GPU.

(b) Normalized end-to-end ScanMatch latency comparison between our design and the GPU.

Fig. 5. The normalized memory access latency and the normalized ScanMatch end-to-end latency between our design and the GPU.



(a) On-chip memory hit rate comparison between our design and the GPU.

(b) Normalized energy consumption comparison between our design and the GPU.

Fig. 6. The on-chip memory hit rate and normalized energy consumption between our design and the GPU.

B. Evaluation Results

Comparison between our design and GPUs. Figure 6 shows that the comparison of on-chip memory hit rate and the energy consumption between our design and the GPU. Figure 5 shows that the comparison of memory access latency and ScanMatch end-to-end computation latency between our design and the GPU. Compared with GPUs, we focus on optimizing the memory access operations in our design. Therefore, we achieve significant hit rate improvement, as the hit rate is as high as 99.6% after our optimization. Moreover, we reduce the memory access latency by 37%, and we reduce the ScanMatch end-to-end latency by 22%. Finally, we achieve 45% energy consumption reduction.

V. CONCLUSION

We observe three main inefficiencies in ScanMatch: (1) misleading information, (2) memory locality issues and (3) sparse HD maps. We propose three designs: (1) point cloud filter, (2) route-aware HD map remapping, (3) adaptive HD map loading, and implement an efficient architecture.

REFERENCES

[1] N. e. a. Akai, "Autonomous driving based on accurate localization using multilayer lidar and dead reckoning," in *ITSC*. IEEE, 2017, pp. 1–6.

[2] R. DD, "Micron. 2014. sdram, 4gb: x4, x8, x16 ddr4 sdram features, white paper," *Micron Technology, Inc*, 2014.

[3] J. e. a. Elseberg, "Efficient processing of large 3d point clouds," in *ICAT*. IEEE, 2011, pp. 1–7.

[4] T. M. O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX security*, 2007.

[5] E. B. Olson, "Real-time correlative scan matching," in *ICRA*. IEEE, 2009, pp. 4387–4393.

[6] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.