# FLi-SLAM: Algorithm Hardware Codesign for Iterative Closest Point Registration in 3D LiDAR SLAM

Zhong Jia Xue*, Alec Lu*, and Zhenman Fang*†
*Simon Fraser University, Burnaby, BC, Canada
†University of Minnesota, Twin Cities, USA
{zjxue, alec_lu, zhenman}@sfu.ca

***Abstract*—LiDAR SLAM is one of the critical algorithms used in robots and autonomous vehicles. However, its core computation, multiple iterations of KDTree based k-nearest neighbors (KNN) correspondence search, is challenging for real-time processing on resource-limited edge devices.**

**In this paper, we propose FLi-SLAM, the first FPGA-based algorithm/hardware codesign for the entire point-to-feature based iterative closest point (ICP) registration process in 3D LiDAR SLAM. We build a fully-pipelined approximate KDTree on chip which theoretically maximizes data reuse for each bucket of off-chip points. To accelerate the subsequent exact KNN search, we propose a simple yet effective accuracy-preserving point cloud pruning algorithm and develop a high-throughput (fully pipelined) KNN hardware design. After accelerating the KDTree based KNN, we also optimize the new bottlenecks in point-to-feature calculations by proposing an algorithm/hardware codesign for their complex QR and Eigen decomposition modules. Evaluated using the widely used open-source LiDAR datasets, our FLi-SLAM running on the AMD/Xilinx ZCU102 embedded FPGA board achieves an average speedup of 56.4x and 22.9x over widely used LiDAR SLAM framework LIO-SAM running on a quad-core ARM A53 CPU, and over LIO-SAM-GPU running on a Nvidia Jetson Orin Nano GPU. Moreover, FLi-SLAM also achieves energy efficiency gains of 19.5x and 10.8x, respectively.**

## I. Introduction

In the past decade, there has been a growing deployment of millions of robots and autonomous vehicles worldwide [1], [2]. One of the key contributors to such success is the advancement of simultaneous localization and mapping (SLAM) algorithms. Compared to traditional vision-based SLAM (VSLAM) methods that are sensitive to poor lighting conditions and have subpar depth estimation, 3D LiDAR SLAM remains robust in variable conditions and provides highly accurate 3D maps [3]–[8]. On the other hand, 3D LiDAR SLAM is much more computation-intensive, as it must perform costly scan matching between adjacent LiDAR frames represented as point clouds. The primary bottleneck lies in the iterative closest point (ICP) registration, where its core computation requires multiple iterations of k-nearest neighbors (KNN) point correspondence search between the point cloud of the current LiDAR frame (about 5,000 query points after downsampling) and the point cloud of the current map built based on all prior frames (about 100,000s of map points after downsampling) [9]. Such high computational workload—tens of billions of distance calculations and comparisons for every LiDAR frame, i.e., in every 100ms for a 3D LiDAR scanner with 10Hz scanning frequency—creates a great challenge for real-time processing on resource-constrained edge devices.

To reduce latency, widely used LiDAR SLAM frameworks like LIO-SAM [3] employ two techniques, which involve partitioning map points via a $k$-dimensional ($k = 3$) KDTree binary search [10] to guide queries, and matching extracted surface (point-to-plane) and corner (point-to-edge) features [11] instead of full point clouds. However, profiling on a quad-core ARM A53 CPU reveals the overall registration pipeline remains 2x to 5x away from achieving real-time performance for a 10Hz 3D LiDAR scanner, not to mention the demands of future LiDAR sensors with higher scanning frequencies and higher density point cloud scans [12]. As illustrated in Section II-A2, this subpar performance can ultimately fail to produce meaningful trajectory results.

Unlike traditional VSLAM that has been well studied for hardware acceleration [9], [13], there is few work on accelerating the entire ICP registration process in 3D LiDAR SLAM. First, the KDTree traversal is inherently sequential and introduces irregular memory access patterns. While prior studies [14], [15] introduced approximate KDTrees to increase parallelism, they fail to fully exploit pipeline parallelism and data reuse across multiple queries. Second, searching these matched buckets for exact k-nearest neighbors (KNN) remains a major, repetitive bottleneck; however, we observe that roughly 86% of these points can be safely pruned—an optimization often overlooked [14]–[17]. Third, once KNN is accelerated, computing feature coefficients (QR and Eigen decompositions) on small matrices [11], [18] emerges as the new bottleneck. While standard iterative algorithms resist pipelining, the small matrix sizes provide more opportunities for better algorithm/hardware codesign.

To address the above challenges, in this paper, we propose FLi-SLAM, the first FPGA-based algorithm/hardware codesign for the entire point-to-plane and point-to-edge feature-based ICP registration process in 3D LiDAR SLAM. Our approach centers on the algorithm/hardware codesign for a delayed-processing architecture that maximizes the data reuse, coupled with a lightweight fully-pipelined tree traversal design. We also introduce a mathematically-proven lossless pruning method to reduce the massive computation of the exact search. Finally, we optimize linear algebra kernels to complete our solution.

Inspired by prior studies [14], [15], FLi-SLAM builds an approximate KDTree on chip to fully exploit pipeline

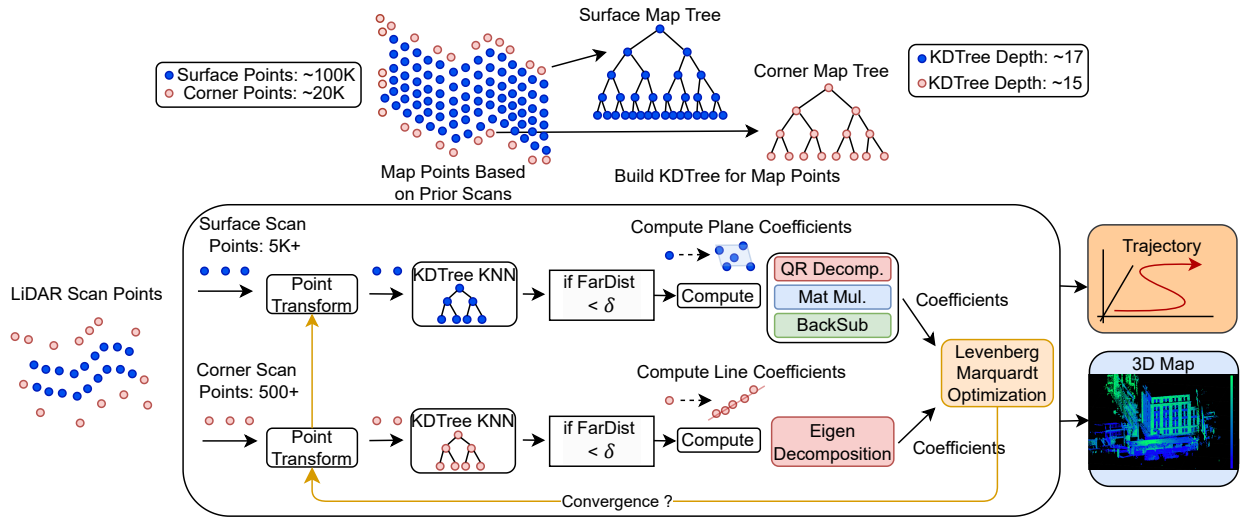



Fig. 1: Feature-based iterative closest point (ICP) registration pipeline in popular 3D LiDAR SLAM framework LIO-SAM [3]

parallelism and data reuse between multiple query points. To maximize this data reuse, we wait until all query points finish their KDTree traversals before performing the downstream exact KNN searches, ensuring each bucket of map points is read from off-chip memory only once. For the exact KNN search in the matched bucket, we propose an accuracy-preserving KNN pruning method that efficiently rejects points if their Euclidean distance to the query point is larger than a threshold $\delta$, alongside a fully pipelined KNN design that replaces complex floating-point comparisons with simpler integer comparisons. To optimize the new bottleneck of QR and Eigen decomposition in the feature coefficient calculation, we implement the modified Gram-Schmidt (MGS) algorithm for QR decomposition [19] and closed-form expressions for the Eigen decomposition of 3x3 Hermitian matrices [20]. Finally, we leverage ping-pong buffering to execute the bucket loading, KNN pruning, exact KNN computation, and feature coefficient calculation in a coarse-grained pipeline, separating the surface and corner points into two concurrent pipelines on the embedded FPGA.

The entire ICP registration process used in 3D LiDAR SLAM is implemented on the embedded FPGA using Vitis HLS (high-level synthesis), except that the KDTree construction and loop closure (both non-bottleneck) are run on the ARM CPU.

We evaluate our FLi-SLAM on the AMD/Xilinx ZCU102 ARM-FPGA system-on-chip (SoC), using the widely used open-source LiDAR datasets from LIO-SAM [3], which are collected using a 3D LiDAR sensor with a scanning frequency of 10Hz. On average, FLi-SLAM achieves 56.4x, 3.4x, and 22.9x speedup over LIO-SAM [3] running on a quad-core ARM A53 CPU and a 24-core Intel Xeon Silver 4214 CPU, and over LIO-SAM-GPU [21] running on a Nvidia Jetson Orin Nano GPU, respectively. Moreover, FLi-SLAM achieves 19.5x, 51.0x, and 10.8x better energy efficiency over the ARM CPU, Intel Xeon CPU, and Nvidia Orin Nano GPU. For the trajectory outputs, FLi-SLAM well matches the reference trajectories and outperforms LIO-SAM running on the ARM CPU. FLi-SLAM is open sourced at https://github.com/SFU-HiAccel/FLi-SLAM.

In summary, this paper makes the following contributions:

1. The first FPGA acceleration framework for the entire feature-based ICP registration process in 3D LiDAR SLAM.
2. An approximate KDTree design that features fully pipelined tree traversal and maximizes data reuse between queries.
3. An accuracy-preserving KNN pruning algorithm and a high-throughput (fully pipelined) KNN hardware design.
4. An algorithm/hardware codesign for QR and Eigen decomposition in the new bottleneck of feature coefficient calculation.
5. Experimental results that demonstrate the promising performance and energy-efficiency gains of FLi-SLAM.

## II. Background and Motivation

### A. ICP Point Cloud Registration Algorithm

*1) Baseline Algorithm in LIO-SAM:* Fig. 1 shows the feature-based ICP registration in LIO-SAM [3] which is the current *de-facto* standard for (ICP) point cloud registration [22]. To find the optimal transformation between the current LiDAR scan and the global map, each query point is transformed with an initial pose and matched to surface or corner map points via KDTree-based KNN search. Valid matches (within a distance threshold $\delta$) trigger feature coefficient calculations using QR or Eigen decomposition [11], [18]. Finally, the LiDAR motion is optimized via the Levenberg-Marquardt (LM) method [23] until convergence.

*2) Performance and Accuracy Gaps on Embedded CPU:* We profile the execution time of the feature-based ICP registration in LIO-SAM (excluding KDTree construction time) for one LiDAR frame using the garden and campus small datasets, running on the quad-core ARM A53 CPU. The experimental setup is detailed in Section IV-A. As shown in Fig. 2 (baseline KDTree bar), its execution time is about 2x to 5x behind the real-time requirement (100ms) for the 10Hz LiDAR scanner. Since the ARM CPU speed cannot keep up with the input rate, batches of feature points are dropped thus reducing trajectory

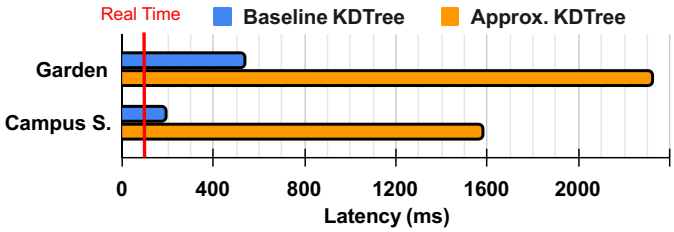



Fig. 2: Latency of ICP registration on ARM CPU

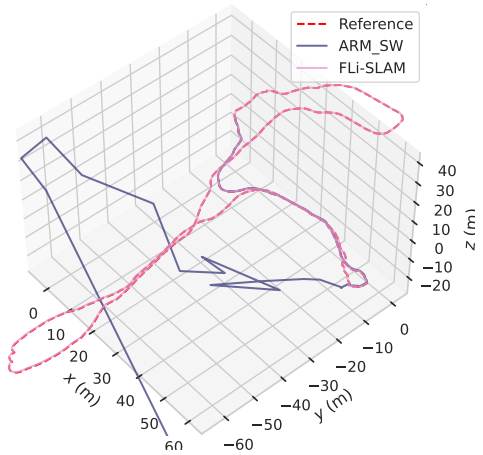



Fig. 3: Garden dataset trajectory

accuracy (Section IV-E1). As shown in Fig. 3, the ARM CPU fails to produce meaningful trajectory results for the garden dataset. A latency breakdown of the garden dataset reveals that the KDTree-based KNN search consumes 87.2% of the execution time, with the linear algebra kernels for coefficient calculations forming the secondary bottleneck at 12.2%.

### B. Prior Studies, Limitations, & Opportunities

*1) Existing Approximate KDTree-based KNNs:* To address the sequential tree traversal and random memory access issues, recent studies [14], [15] propose an approximate KDTree. The key idea is to create a smaller KDTree that can be buffered on chip, where each leaf node stores a bucket of neighbor data points off chip. As a result, the initial tree traversal can be performed with fast on-chip accesses, and the small KDTree can even be duplicated to enable parallel searches [15]. Current research [15], [24] also explores top-tree searches without backtracking, followed by an exact KNN search within the matched bucket. Moreover, prior studies [14], [15] try to cache the bucket of points for exact KNN searches by future queries.

Applying this backtracking-free optimization to LIO-SAM presents two issues. First, to maintain good accuracy, it requires a minimum bucket size; empirical selection shows a minimum bucket size of 4,096 is needed for the most challenging dataset, campus large. Second, the pure software implementation of the approximate KDTree-based KNN search is actually much slower than the original KDTree-based KNN search, as shown in Fig. 2. This is because it has to perform a lot more exact KNN computations. The approximate KDTree approach is only viable due to hardware's ability to customize the parallelization to search the tree buckets. Considering these factors, we choose a bucket size of 4,096 for all our experiments. This also ensures our approximate KDTree itself is small enough to fit on chip.

*2) Limitations of Existing Solutions and New Opportunities:* Existing solutions exhibit several limitations where critical optimization opportunities are often overlooked. During the traversal, pipeline parallelism between multiple queries is not fully exploited; while sequential for one query, on-chip KDTree traversal could be fully pipelined between multiple queries. Similarly, during exact KNN computation, the data reuse of each KDTree bucket is not fully exploited. Prior cache-based studies cannot guarantee theoretical maximum data reuse due to unpredictable queries and limited capacity, but by delaying exact KNN computations until all KDTree traversals finish, we can systematically maximize the data reuse for each KDTree bucket. Furthermore, the KNN pruning opportunity is often overlooked: we observe that the vast majority (86%) of points in the matched bucket are very far away from the query point and can be pruned away. In addition, KNN computation is also not fully optimized, as multi-cycle floating-point comparisons in top K distance sorting prevent fully pipelined KNN computation. Based on IEEE floating-point properties, this can be implemented using a single-cycle integer comparison, allowing the KNN computation to be fully pipelined. Finally, the new bottleneck of feature coefficient calculation is often overlooked. Accelerating the KNN search makes the time-consuming QR and Eigen decompositions the new bottleneck. While default iterative algorithms prevent effective pipeline processing between queries, the small matrix sizes—determined by the dimensionality (*3*) of a 3D point and its small number ($K$) of KNNs—provide more opportunities for algorithm/hardware codesign.

## III. FLi-SLAM Design and Implementation

### A. Architecture Overview and Novelties

FLi-SLAM is built based on top of the approximate KDTree-based KNN search idea, and the non-bottleneck approximate KDTree construction process is left to run on the ARM CPU. Below we summarize the key design innovations in FLi-SLAM.

1. We carefully pipeline the KDTree traversal between query points to achieve a pipeline II of 1. That is, every cycle, we can finish the KDTree traversal of a new query point.
2. To maximize the data reuse of each bucket, we delay the exact KNN computations by finishing the KDTree traversal for all queries first. Then we schedule all queries that fall into the same bucket to perform KNN computations together.
3. To reduce the exact KNN searches per bucket, we present an algorithm/hardware codesign for accuracy-preserving KNN pruning (by 86%), based on mathematic properties of Euclidean distance.
4. We also fully pipeline the exact KNN computation by converting multi-cycle positive floating-point comparisons into single-cycle integer comparisons, based on IEEE floating-point properties, which doubles its throughput over state-of-the-art (SOTA) KNN design [25].
5. We also propose an algorithm/hardware codesign to optimize the new bottlenecks in the feature coefficient calculation.

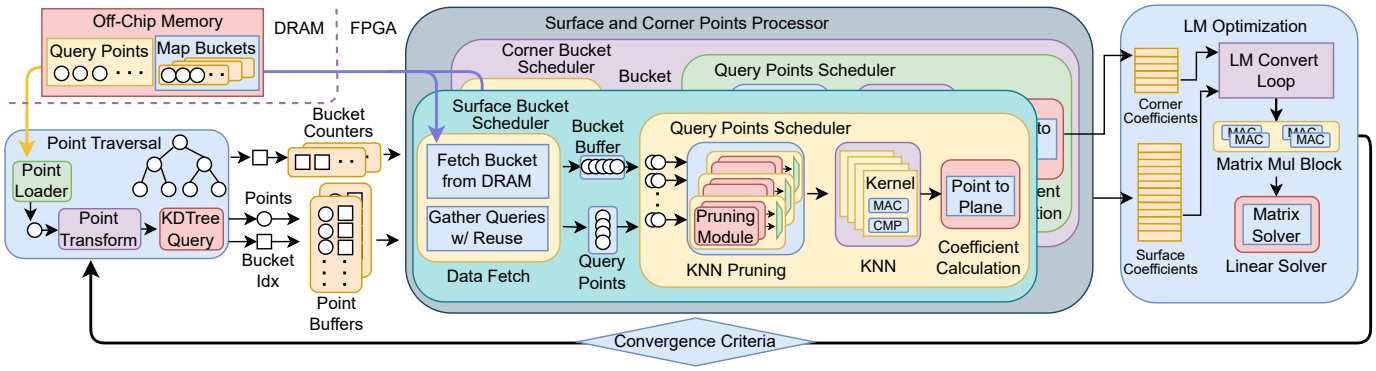



Fig. 4: An overview of our FLi-SLAM design on the FPGA

Specifically, we implement pipelined and parallelized modified Gram-Schmidt (MGS) algorithm for QR decomposition [19] and closed-form expressions for Eigen decomposition of 3x3 matrices [20], which achieve 105.6x and 132.0x speedup over default Vitis HLS library implementations [26].

The overall design of FLi-SLAM is shown in Fig. 4. It starts with the point traversal module to load the query points, apply a point transformation, and traverse the KDTree built for map points in a fully pipelined fashion between queries. This traversal generates the (query, bucket) pairs and their respective bucket reuse counts. Next, it goes to two parallel processors (surface and corner) to find correspondences and calculate their feature coefficients. Inside each processor, it first loads a bucket (with a nonzero counter) of map points into the bucket buffer and gathers all queries associated with that bucket to maximize the data reuse between queries. Batches of queries (considering hardware parallelism) are pipelined through our KNN pruning, exact KNN computation, and feature coefficient calculation modules, which are executed in a coarse-grained pipeline to overlap their execution latency. Moreover, the bucket loading and the query processing are double buffered to hide the bucket loading latency. At the end, the Levenberg Marquardt (LM) module analyzes all the feature coefficients and determines if the iterative process has converged. We leave the design space exploration to Section IV-B.

### B. Fully Pipelined KDTree Traversal

To enable our high-throughput architecture, we first introduce a fully pipelined KDTree traversal design that processes one query per cycle by utilizing a hardware-centric, pointer-free implementation. Our approach intentionally forgoes backtracking to reduce hardware complexity and resource utilization. First, recursively backtracking the tree inherently leads to intricate control logic and irregular memory access patterns—a major cause of the cache collisions common in previous work (Section V-B1). Second, the nature of backtracking requires a stack to store the traversal history. In a highly parallel design, this can consume a substantial amount of scarce on-chip memory. Furthermore, backtracking is not required in our design; instead, we enlarge the bucket size and reduce the additional search work via accuracy-preserving pruning (Section III-D).

Algorithm 1 illustrates how we implement the fully pipelined, non-backtracking, KDTree traversal between multiple queries (lines 1-2) on the FPGA. There are two key points. First, to process different tree sizes in different LiDAR frames, the maximum tree depth (constant *T_Max_Depth*) is chosen to enable the loop unrolling (parallelism, lines 4-5) for all tree levels in Vitis HLS. The traversal loop exits when it reaches the current KDTree depth (variable *TDepth*, line 6). Second, the tree nodes are flattened into a contiguous array and stored in a tree node buffer (*Tbuf*, line 7), which are fully partitioned into registers for parallel access to all tree nodes.

**Algorithm 1** Fully pipelined KDTree traversal on FPGA

```
1: for query in QueryPoints
2:    #pragma HLS pipeline II = 1
3:    index = 0;
4:    for depth = 0 to T_Max_Depth
5:       #pragma HLS unroll
6:       if depth < TDepth then
7:          offset = Tbuf[index] <
  query[depth % 3] ? 2 : 1;
8:          index = 2*index + offset;
```

The tree node pointers are implicitly encoded within the *Tbuf* indices ($index$). For each tree level, a query point is compared with the indexed KDTree node in a specific dimension ($depth$ % 3), and the $index$ in *Tbuf* is updated (lines 7-8). At the end of the traversal, the calculated $index$ allows the query point to find the correct bucket address to perform exact KNN searches. As a result, the traversal of the KDTree is fully pipelined, processing a new query every cycle, independent of the KDTree size. We used a tree depth of six with a bucket size of 4,096 in our experiments, while our FPGA implementation supports tree depths up to 10.

As shown in Fig. 4, the point traversal module also 1) loads the incoming query points via 128-bit AXI burst reads [27], and 2) unpacks and transforms them via matrix-vector multiplication. All of these steps—point loading, point transform, tree traversal, as well as output writing—are fully pipelined with an II of 1.

For the output buffers, since the number of queries that will fall into each bucket is unknown ahead of time, it is infeasible to create the (bucket, dynamic query list) pairs. Instead, we

output all the (query, bucket) pairs and leave it to the bucket scheduler to figure out which queries share which bucket. We also record the total number of queries that fall into each bucket, so that the bucket scheduler can easily skip buckets without any associated queries.

### C. Delayed Processing for Max Data Reuse

*1) Bucket Scheduling and Reuse:* After all queries finish their KDTree traversal, which is very fast thanks to our fully pipelined traversal design, our bucket scheduler can figure out all query points that reuse the same bucket of map points and schedule them to maximize the bucket data reuse. Bucket scheduling and reuse are confined to each ICP iteration, with queries transformed and reassigned after each iteration update. Our bucket scheduler first checks the bucket counters buffer. If a bucket's recorded counter is nonzero, it fetches this bucket of map points from off-chip memory to a bucket buffer using 128-bit wide AXI burst reads. At the same time, it gathers all the query points that reuse this bucket by checking all (query, bucket) pairs, and buffers those queries in a query points buffer. For all the later query processing, the bucket buffer will be reused among all associated queries. Moreover, a ping-pong buffer is used to hide the bucket loading time.

### D. Accuracy-Preserving KNN Pruning

*1) KNN Pruning Basis:* Within the ICP registration algorithm, as shown in Fig. 1, if the furthest point of the top k-nearest neighbors is more than a squared Euclidean distance—we use squared Euclidean distance to avoid the complex square root operation in our KNN distance computation—of $\delta$ away from the query point, then this set of feature correspondences is automatically rejected. Thus, any map point with a Euclidean distance greater than $\sqrt{\delta}$ away from the query point can be removed from the KNN search space. Moreover, if Equation 1 is true, where $P_q = (x_q, y_q, z_q)$ and $P_m = (x_m, y_m, z_m)$, then Equation 2 must also be true.

$$(|x_q - x_m| > \sqrt{\delta}) \vee (|y_q - y_m| > \sqrt{\delta}) \vee (|z_q - z_m| > \sqrt{\delta}) \quad (1)$$

$$d^2(P_q, P_m) = [(x_q - x_m)^2 + (y_q - y_m)^2 + (z_q - z_m)^2] > \delta \quad (2)$$

That is, if the difference in any dimension—x, y, or z—is greater than $\sqrt{\delta}$, for $\delta \in \mathbb{R}^+$, then the overall squared Euclidean distance must be greater than $\delta$. We can use Equation 1 as a substitute to avoid the full squared distance computation to prune those faraway map points.

*2) Bounding Box Pruning Method:* For pruning against a group of query points on the same bucket, a bounding box approach can be used, as shown in Fig. 5. The bounding box is constructed by finding the min and max of x, y and z values for all query points, and then expanding the box in all dimensions by an additional distance of $\sqrt{\delta}$. For any search space points (i.e., map points) outside of this bounding box, it is mathematically impossible to be within a squared distance of $\delta$ to any query point within the box. Determining whether a search space point lies outside the bounding box only requires floating-point comparison operators. To maintain an effective pruning ratio, multiple tighter bounding boxes can be constructed on equally divided partitions of query points.

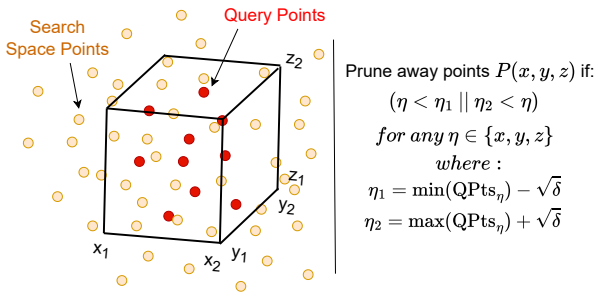



Fig. 5: KNN pruning visualization

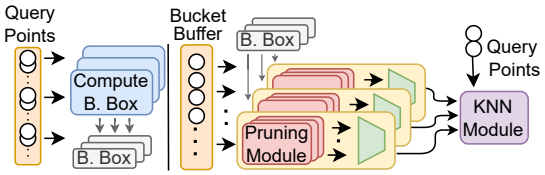



Fig. 6: KNN pruning module

*3) KNN Pruning Hardware Module:* As shown in Fig. 6, multiple sets of hardware efficient KNN pruning units are instantiated to prune the search space. First, a min-max module calculates the bounding boxes for each partition of query points. The bounding boxes are then fed to each set of pruning units. Within each unit, multiple pruning modules operate in parallel on an equally divided partition of the search space in a fully pipelined fashion. As pruning progresses, map points that are not pruned away are passed into a stream for further processing by the exact KNN modules.

### E. Exact KNN Computation Module

To accelerate exact KNN search, we build a fully pipelined KNN computation module with a similar streaming architecture to CHIP-KNNv2 [25] but with 2x better pipeline throughput by converting multi-cycle floating-point comparisons into single-cycle integer operations (see Section III-E1).

As shown in Fig. 7, each pipeline performs the KNN computation for one query point and we also implement the KNN parallelism along the query point dimension. The unpruned map points (search space points) are streamed in and shared among all parallel pipelines. Inside each pipeline, every cycle, the squared Euclidean distance between a new search space point and the query point is calculated. The search space point and its squared distance pair is then pushed into the zeroth position of a top $K + 1$ elements array (buffer). To sort the array, two sets of compare and swap (CAS) operations (a total of K CAS operations) are performed on the items ahead and the items behind in the array. Upon iterating through the entire search space, the top K neighbors are present in array indices 1 to K. More details about this KNN pipeline implementation and its correctness proof can be found in [25], [28].

For feature-based ICP registration, only the top 5 (K = 5) neighbors are required; however, our hardware can be adjusted

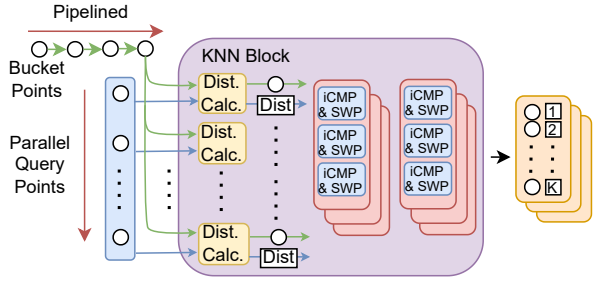



Fig. 7: Fully pipelined KNN computation module

for a variable K by simply adjusting the top $K+1$ elements array (buffer) and $K$ CAS operations, without introducing additional PEs. No extra distance computation modules are needed.

*1) KNN Pipeline Throughput Improvement:* We observe that prior KNN designs [25], [28] can only achieve a pipeline II of 2 for top K sorting due to the multi-cycle latency of the complex floating-point comparison (fcmp). Whereas, an integer comparison (icmp) only requires one cycle to complete. Based on the IEEE floating-point representation shown in Equation 3, we can convert a positive fcmp into an icmp; a squared Euclidean distance is positive.

$$sign(1) \quad | \quad exponent(8) \quad | \quad mantissa(23) \qquad (3)$$

To demonstrate the correctness of the method: if the fcmp is bitcast to an icmp, the sign bit will be compared first, then the exponent, and then the mantissa, as evident in Equation 3. For the case of both positive numbers, this comparison order of icmp is equivalent for fcmp, since a larger exponent/mantissa represents a larger number overall. With this conversion, we can achieve a pipeline II of 1.

### F. Surface and Corner Coefficient Modules

*1) Surface Coefficient Module:* Within the surface coefficients module, the main bottleneck is the 5x3 (i.e., $k*3$) linear least squares computation. We first attempt to use the QR decomposition from the Vitis FPGA libraries [26], which is based on Givens rotations to solve the equations. Since the Vitis solution is iterative, its processing latency is unknown at compile time. As a result, it cannot be pipelined between the processing of multiple incoming query points, leading to a major bottleneck. To address this issue, we choose to implement a solution based on the modified Gram-Schmidt (MGS) algorithm for QR decomposition [19], where the computation is fixed for each input and allows for pipelining between query points. We also implement a matrix multiplication and back-substitution kernel to complete the linear least squares kernel in a pipelined fashion (it can achieve an II of 1), shown in Fig. 8.

Shown in Table I, our MGS-based solution achieves a ∼366x improvement in pipeline initiation interval (II) and a ∼5.22x improvement in iteration latency (IL) over the Vitis library, resulting in an overall ∼**106x** speedup with similar resources. We utilize an II of 6 (instead of 1), considering the large IL and resource sharing within the pipeline. Note our module can handle other input sizes.

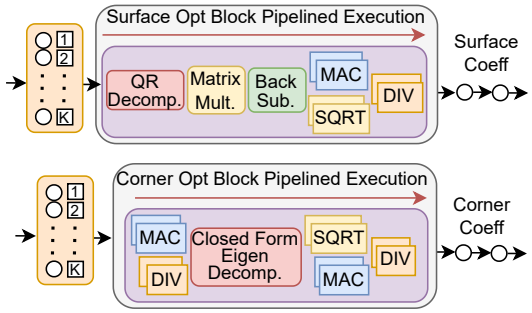



Fig. 8: Surface & corner coefficients processing pipelines

TABLE I: Surface coefficient block: Vitis library vs our MGS

| Imp. | BRAM | DSP | FF | LUT | II | IL | #pts |
|---|---|---|---|---|---|---|---|
| Lib | 0% | 1% | 4.8% | 5.3% | 2,199 | - | 28 |
| Our | 0% | 3.4% | 4.0% | 4.7% | 6 | 421 | |

TABLE II: Corner coefficient block: Vitis library vs ours

| Imp. | BRAM | DSP | FF | LUT | II | IL | #pts |
|---|---|---|---|---|---|---|---|
| Lib | 4.8% | 4.4% | 9.7% | 13.2% | 26,893 | - | 4 |
| Ours | 0% | 1.8% | 2.6% | 4.2% | 50 | 665 | |

*2) Corner Coefficient Module:* The main bottleneck is the Eigen decomposition of a 3x3 Hermitian matrix. First, we use the Jacobi Eigen solver from the Vitis libraries [26]. Similarly, due to its iterative computation, it cannot be pipelined between multiple query points; each kernel call takes ∼27K cycles, which is unacceptable. To fix this, we choose to implement closed-form expressions for Eigen decomposition of 3x3 Hermitian matrices [20]. Now the computation is the same for different input query points, allowing it to be pipelined across multiple inputs. We also pipeline it along with other computations within the corner coefficient calculation such as square root, division, and multiplication, shown in Fig. 8.

Shown in Table II, our resource utilization is reduced by ∼3x. Moreover, there is a ∼538x improvement in pipeline II, a ∼40x improvement in IL, and an overall throughput gain of **132x**. Similarly, we utilize a pipeline II of 50 (instead of 1, which can be achieved), considering the large IL and resource sharing within the pipeline.

### G. LM Optimization Module

To avoid frequent CPU-FPGA communication in each iteration, we implement the non-bottleneck Levenberg Marquardt (LM) module on the FPGA as well, which includes three main parts: the LM convert loop, a matrix multiplication block and a 6x6 matrix solver. First, the LM convert loop performs coordinate transformations on the input coefficients, which are then used to populate the matrix $A$. We pipeline this loop with an II of 1. Second, the matrix multiplication within the LM optimization module consists of the multiplication between the matrix $A$ and $A^T$. Since the resulting matrix product is symmetric we only need to compute the upper triangle, which

is overlooked in prior works [29]–[32]. We design a tiled and pipelined kernel to perform this operation. Lastly, to compute the 6x6 matrix solver, we use the same linear least squares algorithm described in Section III-F1.

## IV. Experimental Results

### A. Experimental Setup

Our FPGA design achieves 250MHz on the AMD-Xilinx ZCU102 board using Vitis HLS and Vivado 2023.1. We evaluate FLi-SLAM with the 10Hz LiDAR datasets from LIO-SAM [3]. Since FLi-SLAM is the first FPGA solution for point-to-edge and point-to-plane feature-based ICP registration, we compare it against leading software implementations. The CPU baselines run LIO-SAM on a quad-core ARM Cortex-A53 and a dual-socket server with two 12-core Intel Xeon Silver 4214 CPUs (24 physical cores, 48 logical threads). The ARM baseline uses Linaro GCC 7.3 with four OpenMP threads, while the Intel baseline uses GCC 5.4 with 24 OpenMP threads at 2.20GHz. Both are compiled with `-O3`.

For the GPU baseline, we run LIO-SAM-GPU [21] on an Nvidia Orin Nano with JetPack 5.1.1/L4T 35.3.1 in 15W mode with dynamic clocks. Observed CPU and GPU frequencies are 1.5GHz and 624MHz. Desktop GPUs should be faster but consume substantially more power than our embedded target.

All platforms replay identical recorded sensor data, but runtime state changes processed frames and ICP iterations. Reported speedups compare average registration latency, including all ICP iterations for a frame. FLi-SLAM registration timing includes memory transfers with ROS running concurrently and excludes ARM–FPGA control overhead. KDTree construction is timed separately (Section IV-C).

We measure power using ZCU102 INA226 monitors, Intel RAPL, and Orin Nano `jetson_stats` [33]. FPGA resource utilization is reported after place-and-route, and trajectory RMSE is evaluated with evo [34].

### B. FLi-SLAM Design Space Exploration

To allocate parallelism according to the workload, we first examine the relative sizes of the surface and corner point sets. The average number of map and query points in the corner and surface, and the number of feature coefficients, are summarized in Table III. The larger surface query workload motivates separate parallelism factors for the two processors.

TABLE III: Average statistics across LiDAR datasets (maximum number of points in one LiDAR frame is larger)

| Dataset | Corner Map Points | Corner Queries | Surface Map Points | Surface Queries | Num. of Coefficients |
|---|---|---|---|---|---|
| Campus Large | 12,658 | 515 | 70,844 | 5,776 | 4,673 |
| Campus Small | 11,479 | 400 | 85,810 | 5,522 | 4,498 |
| Garden | 20,172 | 463 | 67,107 | 4,358 | 3,942 |
| Kitti | 21,620 | 987 | 52,733 | 6,548 | 5,703 |
| Park | 12,533 | 403 | 67,699 | 4,985 | 3,773 |
| Rotation | 7,346 | 499 | 21,659 | 3,345 | 3,272 |
| Walking | 15,021 | 375 | 59,417 | 3,740 | 3,204 |

We perform design space exploration on FLi-SLAM by systematically sweeping key performance impacting parameters to achieve the lowest latency to process one LiDAR frame.

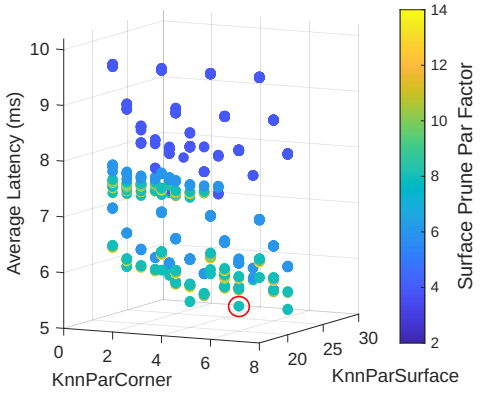



Fig. 9: FLi-SLAM design space exploration

We mainly focus on sweeping the parallelism factor of exact KNN computation (denoted as $\kappa_{sur}$ and $\kappa_{cor}$) and KNN pruning (denoted as $\rho_{sur}$ and $\rho_{cor}$) modules, as well as number of parallel bounding boxes ($\beta_{sur}$ and $\beta_{cor}$), in both the surface and corner point processors. For the surface and corner coefficient modules, they are bottlenecked by their large pipeline iteration latencies as presented in Section III-F, and further parallelization does not help improve their performance. As will be presented in Section IV-F, other modules are non-bottlenecks and do not need further parallelization.

Fig. 9 visualizes the design space exploration, plotting the latency per LiDAR frame for a range of design parameters. For visualization purposes, we display only three dimensions, omitting $\rho_{cor}$, $\beta_{sur}$ and $\beta_{cor}$, as they are less significant and we consistently select the optimal design point. Our optimal FPGA design chooses $\kappa_{sur}$=28, $\kappa_{cor}$=4, $\rho_{sur}$=8, $\rho_{cor}$=6, $\beta_{sur}$=2 and $\beta_{cor}$=1. After place-and-route, it achieves 250MHz frequency with resource utilization reported in Section IV-G. We use this configuration for all results.

Query points generally follow LiDAR arrival order, helping form tight bounding boxes. With this query ordering and bounding box configuration, pruning ratios are 85.2%, 83.5%, 83.3%, 85.1%, 86.1%, 85.9%, and 90.2% for Campus Large, Campus Small, Park, Garden, Walking, KITTI, and Rotation datasets, respectively, averaging 85.6% across datasets.

### C. Overall Performance Comparison

Shown in Fig. 10, on average, our FLi-SLAM achieves a 56.4x, 3.4x, and 22.9x speedup over LIO-SAM on the ARM CPU and Intel Xeon CPU, and LIO-SAM-GPU on the Orin Nano GPU, respectively. Both the ARM CPU and Orin Nano GPU fail to complete the feature-based ICP registration within real-time LiDAR sensor (10Hz) requirements. Only FLi-SLAM is able to consistently achieve the real-time registration requirements even for 100Hz LiDAR sensors, which is future proof for new sensor development.

*1) KDTree Construction Latency:* To ensure that the approximate KDTree construction latency does not negatively impact the overall LiDAR SLAM performance, we measure its latency with a bucket size of 4,096 on the ARM CPU. We also compare it against the original KDTree construction time in LIO-SAM, which is implemented using the FLANN library

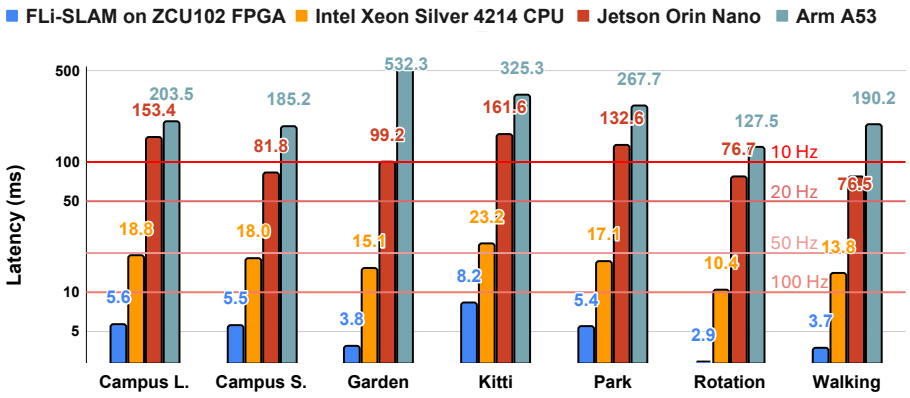



Fig. 10: Latency per frame comparison between CPU, GPU, and FPGA

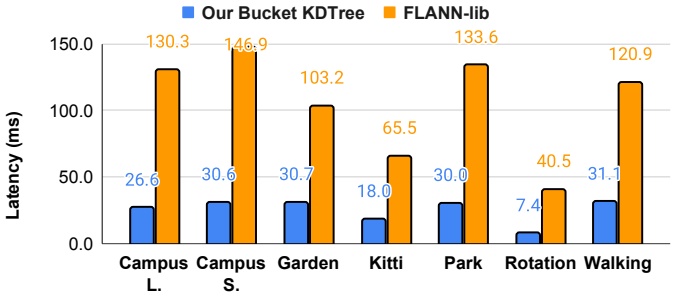



Fig. 11: KDTree construction latency

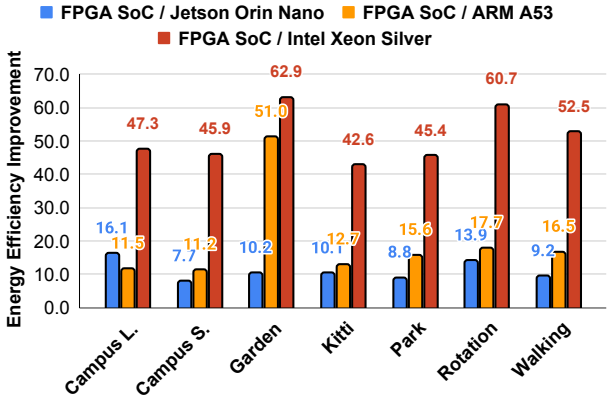



Fig. 12: Energy efficiency improvement

[10]. Shown in Fig. 11, our implementation achieves an average speedup of 4.36x over the FLANN KDTree construction, since our approximate KDTree (tree depth of six) is much smaller than the original KDTree. Even combining with the KDTree construction latency, FLi-SLAM is able to meet the real-time registration requirements for 20Hz LiDAR sensors.

### *D. Overall Energy Efficiency Comparison*

Fig. 12 presents the energy efficiency improvements of FLi-SLAM on the FPGA over LIO-SAM on the ARM CPU and Intel Xeon CPU, and LIO-SAM-GPU on the Orin Nano GPU. The energy consumption is calculated in joules (i.e., $time \times power$). On average, FLi-SLAM achieves a 19.5x, 51.0x, and 10.8x energy gains compared to the ARM CPU, Intel Xeon CPU, and Orin Nano GPU, respectively. Across datasets, recorded chip power averages 85.57W for Intel, 1.94W for ARM, 5.77W for the ZCU102 CPU and FPGA, and 2.74W for Orin Nano's CPU/GPU/CV. Measurements exclude external DRAM and other board components.

### *E. Overall Accuracy Comparison*

We measure the trajectory outputs of FLi-SLAM on the FPGA and LIO-SAM on the ARM CPU, and calculate their RMSE against the reference trajectories. For the LIO-SAM dataset, which lacks a public ground truth, we established a baseline based on the Intel Xeon CPU. Shown in Table IV, FLi-SLAM always outperforms LIO-SAM on the ARM CPU in accuracy. While their trajectory differences are hard to observe visually in most datasets, as presented in Fig. 3 in Section II-A2, LIO-SAM on the ARM CPU fails to track for the garden and campus small datasets (trajectory figure for campus dataset omitted due to space constraints) due to dropping batches of feature points, as discussed below in Section IV-E1. Meanwhile, FLi-SLAM well matches those reference trajectories.

TABLE IV: RMSE for trajectories on ARM and FLi-SLAM

| RMSE | Camp. L. | Camp. S. | Garden | Kitti | Park | Rot. | Walk. |
|---|---|---|---|---|---|---|---|
| ARM | 0.280 | **2.097** | **640.862** | 3.102 | 0.054 | 0.009 | 0.150 |
| Ours | 0.266 | **0.057** | **0.020** | 1.339 | 0.039 | 0.008 | 0.062 |

To further validate these findings against a certified ground truth, we benchmarked our system on the KITTI dataset [35] (sequences 05-10). As shown in Table V, these results confirm FLi-SLAM's accuracy is on par with the Intel Xeon CPU, while the ARM CPU often fails registration.

TABLE V: Extended RMSE Evaluation on KITTI Sequences

| Seq. | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|
| ARM | 87.848 | 418.451 | 0.480 | 200.790 | 421.9456 | 143.659 |
| Ours | 1.226 | 14.425 | **0.432** | 5.776 | **9.766** | 2.066 |
| Intel | **0.921** | **14.347** | 0.450 | **5.223** | 10.494 | **1.988** |

*1) Runtime Effects on Trajectory Failures:* To investigate ARM trajectory failures, we reduce the input rate to one-tenth of real time, allowing all batches of feature points to reach mapping; in this case, it can produce the correct trajectory. However, at real time input rate, the ARM CPU cannot process data fast enough, so batches are dropped. Only 33.3% of

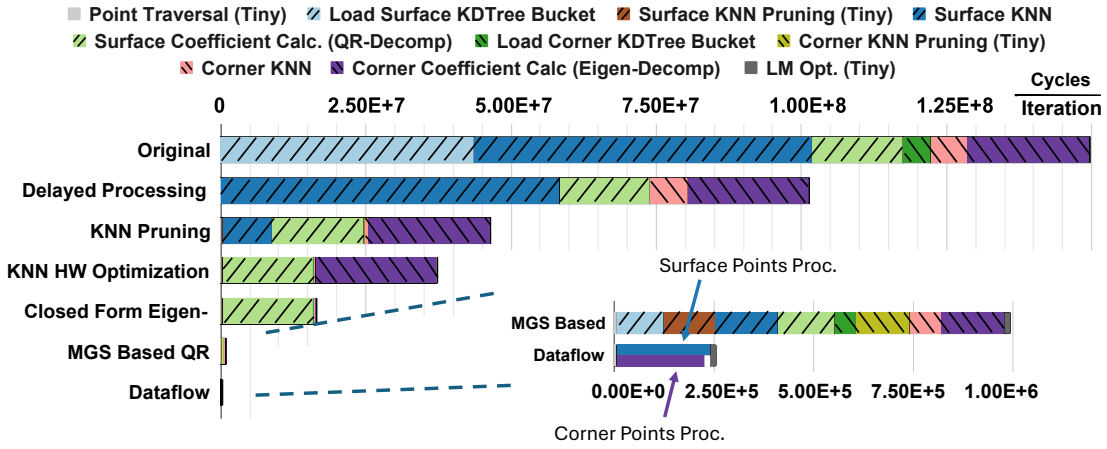



Fig. 13: Step-by-step performance breakdown for major optimizations

Garden and 25.1% of Campus Small batches reach mapping. This associates ARM trajectory failures with data loss.

### F. Step-by-Step Performance Breakdown

Fig. 13 presents the latency breakdown of the major modules in FLi-SLAM after each major optimization. The key takeaway is that our holistic algorithm/hardware codesign systematically removes bottlenecks in sequence, demonstrating the need for a full-system approach. For visualization simplicity, we plot the figure as if different modules run one after another.

The unoptimized FPGA baseline is heavily bottlenecked by bucket loading latencies. Introducing our delayed processing architecture maximizes data reuse, reducing memory access latency by ∼282x for a 1.48x overall speedup. With memory optimized, exact KNN computations become the primary bottleneck; our accuracy-preserving pruning eliminates 86% of this workload (yielding an overall 2.18x speedup), while parallel queries and pipeline II optimizations further reduce KNN latency by ∼40x (a 1.25x speedup). To resolve the subsequent coefficient processing bottleneck, we apply pipelined closed-form Eigen and MGS-based QR decompositions, achieving consecutive overall speedups of 2.26x and 16.72x. Finally, mapping these modules to a coarse-grained dataflow execution across parallel surface and corner processors adds an extra 3.98x speedup, culminating in a 604x total performance improvement over the initial baseline.

### G. Resource Utilization

Table VI shows the detailed resource utilization breakdown for each major component of FLi-SLAM on the ZCU102 FPGA. The most constrained resource is the LUT and BRAM. The largest contributor to the LUT utilization is the surface KNN module, which utilizes about 19.66% of LUTs. The two KNN pruning modules are resource light, utilizing about 2.92% and 1.24% LUTs. Our fully pipelined KDTree traversal module is also lightweight.

TABLE VI: Resource utilization of FLi-SLAM on ZCU102

| Modules | | BRAM | DSP | FF | LUT |
|---|---|---|---|---|---|
| Tree Traversal | | 0.38% | 5.20% | 2.97% | 5.07% |
| Surface Points Processor | Prune | 0% | 0.95% | 1.35% | 2.92% |
| | KNN | 0% | 21.11% | 16.55% | 19.66% |
| | Coeff. | 0% | 3.41% | 3.21% | 4.59% |
| | Buf. | 16.45% | 0% | 0.30% | 1.21% |
| Corner Points Processor | Prune | 0% | 0.48% | 0.93% | 1.24% |
| | KNN | 0% | 3.02% | 3.50% | 4.00% |
| | Coeff. | 0% | 1.79% | 2.33% | 5.07% |
| | Buf. | 7.24% | 0% | 0.30% | 0.95% |
| LM Opt. | LM | 17.00% | 6.27% | 6.24% | 9.62% |
| | Buf. | 17.82% | 0% | 0% | 0.06% |
| Misc. | | 4.22% | 2.26% | 6.90% | 7.68% |
| Total | | **63.10%** | **44.48%** | **44.58%** | **62.07%** |

## V. Related Work

To the best of our knowledge, FLi-SLAM is the first work to accelerate the entire point-to-plane and point-to-edge feature-based ICP registration process on FPGAs, which is the major bottleneck in 3D LiDAR SLAM. Table VII summarizes the key differences between FLi-SLAM and prior point registration acceleration studies for depth-aware SLAM and KNN acceleration studies.

FLi-SLAM is the only work that exploits the novel fully pipelined KDTree traversal, delayed processing for maximum data reuse, accuracy-preserving KNN pruning, fully pipelined KNN computation, and algorithm/hardware codesign to accelerate QR and Eigen decomposition. Compared to other FPGA work for depth-aware SLAM, we achieve the fastest average registration latency of 5.0ms.

### A. Registration for Depth-Aware SLAM

*1) 3D LiDAR SLAM:* Eisoldt et al. [36] implement HATSDF SLAM where they accelerate point-to-TSDF ICP on FPGA. They use a truncated signed distance field (TSDF) to represent the global map and achieve 38ms for registration with 16 LiDAR scan lines. However, their design is bandwidth limited during ICP registration due to off-chip map data access, an issue that our work solves with maximum data reuse.

In terms of other registration methods, Deng et al. [37] implement a normal distribution transform (NDT) based registration for 3D LiDAR SLAM on FPGA. They implement a specialized KNN acceleration data structure to speedup the KNN bottleneck and their registration achieves an average

TABLE VII: Comparison of related work on KNN and registration

| Group | Type | Works | Platform | KNN | | | | Registration | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Fully Pipelined KDTree Traversal | Maximize Data Reuse Between Query Pts. | KNN Accel. | KNN Pruning | LiDAR Based | Full Registration on HW | QR Dcmp. Accel. | Eigen Dcmp. Accel. | Latency (ms) |
| Registration in SLAM | 3D LiDAR | [36] | FPGA | ✘ | ✘ | ✘ | ✘ | ✔ | ✔ | ✘ | ✘ | 38 |
| | | [37] | FPGA | ✘ | ✘ | ✔ | ✘ | ✔ | ✔ | ✘ | ✘ | 30.4 |
| | RGB-D | [38] | FPGA | ✘ | ✘ | ✔ | ✘ | ✘ | ✔ | ✘ | ✘ | 720 |
| | | [39] | FPGA | ✘ | ✘ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ | 500 |
| | | [40] | FPGA | ✘ | ✘ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ | 36.4 |
| | 2D LiDAR | [41] | FPGA | ✘ | ✘ | ✘ | ✘ | ✔ | ✔ | ✘ | ✘ | 98.2 |
| KNN | KDTree | [14] | ASIC Sim. | ✘ | ✘ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | | [15] | FPGA | ✘ | ✘ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | | [24] | ASIC Sim. | ✘ | ✔ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | Other Search Structs. | [16] | FPGA | ✘ | ✘ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | | [17] | FPGA | ✘ | ✘ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | | [42] | FPGA | ✘ | ✔ | ✔ | ✘ | ✔ | ✘ | ✘ | ✘ | - |
| | Pruning | [43] | ASIC Sim. | ✘ | ✘ | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ | - |
| | Exact KNN | [25] | FPGA | ✘ | ✘ | ✔ | ✘ | ✘ | ✘ | ✘ | ✘ | - |
| - | - | Ours | FPGA | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | **5.0** |

latency of 30.4ms. However, they do not apply any KNN pruning or any strategies for maximum data reuse.

*2) RGB-D SLAM:* Gautier et al. and Gkeka et al. [39], [40] implement a RGB-D based point-to-TSDF SLAM as well, achieving a registration latency of 500ms and 36.4ms, respectively.

### B. KNN Acceleration

*1) Approximate KDTree-based KNN:* Tigris [14], QuickNN [15] and Crescent [24] implement approximate KDTrees and perform each tree traversal sequentially with load/compute/store stages. However, they do not exploit pipeline parallelism between multiple query points, and incur cache collisions when accessing tree node data.

Our work fully pipelines the KDTree traversal between query points by allowing each query uncontested access to one level of the tree, and thus completely removing cache collisions. Furthermore, our work stores the tree nodes in a contiguous array which allows for predictable calculation of the next tree node address, in contrast to the method previous works use, which requires explicit pointers for the next node. Since FLi-SLAM can perform fast tree traversal with a pipeline II of 1, there is no need for instantiating multiple tree traversal units, which Tigris, QuickNN and Crescent do.

Compared to Crescent's implementation, our work achieves greater efficiency through a delayed processing architecture; we delay the query-to-bucket bin sorting and substitute it with a streaming scan to avoid large bins for this sorting. Therefore, between KDTree traversal and KNN computation, our point buffer retains all query points on-chip, whereas Crescent offloads these large query point bins to DRAM and then fetches them again. Furthermore, Crescent reduces the exhaustive KNN search nodes by 41% with an extra KDTree, whereas our pruning method can reduce about 86% on average of nodes using mathematical properties in LiDAR SLAM.

*2) Linear Algebra Acceleration:* A few efforts accelerate the QR and Eigen decomposition for large matrices.

For example, Langhammer et al. [44] accelerate QR decomposition with MGS on an Intel Arria 10 FPGA, and achieve 80us for a 256x256 floating-point matrix. Muñoz et al. [45] implement QR decomposition using the Givens rotation method with a 2D systolic array architecture. However, prior methods have not focused on implementing a completely unrolled and pipelined QR decomposition for high throughput with small input matrices.

Zhang et al. [46] accelerate Eigen decomposition with a systolic array implementation of parallel Jacobi algorithm for 16-bit data type. Burger et al. [47] use a triangular systolic array to solve Eigen decomposition with the QR algorithm. However, previous hardware methods have not explored the possibility of using closed-form expressions of Eigen decomposition for small input matrices.

## VI. Conclusion

In this paper we have presented FLi-SLAM, the first FPGA-based 3D LiDAR SLAM accelerator for point-to-plane and point-to-edge ICP registration. FLi-SLAM proposes 1) a fully pipelined approximate KDTree traversal, 2) a delayed processing architecture to maximize the data reuse between multiple queries, 3) an algorithm/hardware codesign to prune and accelerate exact KNN searches, and 4) an algorithm/hardware codesign to accelerate often overlooked feature coefficient calculation. Experimental results demonstrate competitive performance and energy gains of FLi-SLAM on the embedded ZCU102 FPGA. Compared to widely used LIO-SAM on ARM A53 CPU and LIO-SAM-GPU on Orin Nano GPU, on average, it achieves 56.4x and 22.9x speedups, and 19.5x and 10.8x energy savings. FLi-SLAM is currently integrated with LIO-SAM and is open sourced at https://github.com/SFU-HiAccel/FLi-SLAM.


## Acknowledgment

We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

## REFERENCES


[1] "Worldwide - robotic car LiDAR shipments by segment 2032 — Statista — statista.com," https://www.statista.com/statistics/880140/global-robotic-car-lidar-shipments-by-segment/, [n. d.], [Accessed 13-10-2024].

[2] "Autonomous mobile robotics market size worldwide 2016-2028 — Statista — statista.com," https://www.statista.com/statistics/1285835/worldwide-autonomous-robots-market-size/, [Accessed 13-10-2024].

[3] T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti, and R. Daniela, "Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 5135–5142.

[4] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[5] S. Zheng, J. Wang, C. Rizos, W. Ding, and A. El-Mowafy, "Simultaneous localization and mapping (slam) for autonomous driving: Concept and analysis," *Remote Sensing*, vol. 15, no. 4, 2023. [Online]. Available: https://www.mdpi.com/2072-4292/15/4/1156

[6] C. Debeunne and D. Vivet, "A review of visual-lidar fusion based simultaneous localization and mapping," *Sensors*, vol. 20, no. 7, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/7/2068

[7] M. Chghaf, S. Rodriguez, and A. E. Ouardi, "Camera, lidar and multi-modal slam systems for autonomous ground vehicles: a survey," *Journal of Intelligent & Robotic Systems*, vol. 105, no. 1, p. 2, 2022.

[8] H. Yin, X. Xu, S. Lu, X. Chen, R. Xiong, S. Shen, C. Stachniss, and Y. Wang, "A survey on global lidar localization: Challenges, advances and open problems," *International Journal of Computer Vision*, pp. 1–33, 2024.

[9] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A survey of fpga-based robotic computing," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 48–74, 2021.

[10] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[11] J. Zhang and S. Singh, "Loam: Lidar odometry and mapping in real-time," *Robotics: Science and systems*, vol. 2, no. 9, pp. 1–9, 2014.

[12] "Long-range lidar sensor for autonomous vehicles & trucking — Ouster — ouster.com," https://ouster.com/products/hardware/os2-lidar-sensor, [Accessed 13-10-2024].

[13] R. Eyvazpour, M. Shoaran, and G. Karimian, "Hardware implementation of slam algorithms: a survey on implementation approaches and platforms," *Artificial Intelligence Review*, vol. 56, no. 7, pp. 6187–6239, 2023.

[14] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 629–642. [Online]. Available: https://doi.org/10.1145/3352460.3358259

[15] R. Pinkham, S. Zeng, and Z. Zhang, "Quicknn: Memory and performance optimization of k-d tree based nearest neighbor search for 3d point clouds," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 180–192.

[16] J. Xiao, H. Sun, Q. Deng, X. Liu, H. Zhang, C. He, Y. Shu, and Y. Ha, "Rps-knn: An ultra-fast fpga accelerator of range-projection-structure k-nearest-neighbor search for lidar odometry in smart vehicles," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2023, pp. 1–5.

[17] H. Sun, X. Liu, Q. Deng, W. Jiang, S. Luo, and Y. Ha, "Efficient fpga implementation of k-nearest-neighbor search algorithm for 3d lidar localization and mapping in smart vehicles," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 9, pp. 1644–1648, 2020.

[18] T. Shan and B. Englot, "Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 4758–4765.

[19] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2013.

[20] C.-A. Deledalle, L. Denis, S. Tabti, and F. Tupin, "Closed-form expressions of the eigen decomposition of 2 x 2 and 3 x 3 hermitian matrices," Ph.D. dissertation, Université de Lyon, 2017.

[21] qdLMF, "Lio-sam-gpu-scantomapopt," https://github.com/qdLMF/LIO-SAM-GPU-ScanToMapOpt, 2023, accessed: 2024-10-03.

[22] S. Fontana, D. Cattaneo, A. L. Ballardini, M. Vaghi, and D. G. Sorrenti, "A benchmark for point clouds registration algorithms," *Robotics and Autonomous Systems*, vol. 140, p. 103734, 2021.

[23] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[24] Y. Feng, G. Hammonds, Y. Gan, and Y. Zhu, "Crescent: taming memory irregularities for accelerating deep point cloud analytics," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 962–977.

[25] K. Liu, A. Lu, K. Samtani, Z. Fang, and L. Guo, "Chip-knnv2: A configurable and high-performance k-nearest neighbors accelerator on hbm-based fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3616873

[26] Xilinx, "Vitis libraries," https://github.com/Xilinx/Vitis_Libraries, accessed: 2024-10-03.

[27] A. Lu, Z. Fang, and L. Shannon, "Demystifying the soft and hardened memory systems of modern fpgas for software programmers through microbenchmarking," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, Jun. 2022. [Online]. Available: https://doi.org/10.1145/3517131

[28] A. Lu, Z. Fang, N. Farahpour, and L. Shannon, "Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 139–147.

[29] A. Abba, F. Caponio, A. Geraci, and G. Ripamonti, "Implementation of high efficiency non-linear least-squares in fpga devices for digital spectroscopy," in *IEEE Nuclear Science Symposuim & Medical Imaging Conference*. IEEE, 2010, pp. 1371–1376.

[30] W. Liu, B. Yu, Y. Gan, Q. Liu, J. Tang, S. Liu, and Y. Zhu, "Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 479–493.

[31] M. Cavuslu and S. Sahin, "Fpga implementation of ann training using levenberg and marquardt algorithms," *Neural Network World*, vol. 28, no. 2, pp. 161–178, 2018.

[32] J. M. Blasco, E. Sanchis, V. González, J. D. Martín, F. J. Egea, D. Barrientos, and D. Granero, "Maximum likelihood estimation and non-linear least squares fitting with levenberg-marquardt algorithm implementation in fpga devices for high resolution hodoscopy," in *2012 18th IEEE-NPSS Real Time Conference*. IEEE, 2012, pp. 1–5.

[33] R. Bonghi, "jetson_stats: A python package for monitoring and controlling nvidia jetson devices," https://github.com/rbonghi/jetson_stats, 2018, accessed: 2024-10-03.

[34] M. Grupp, "evo: Python package for the evaluation of odometry and slam." https://github.com/MichaelGrupp/evo, 2017.

[35] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361.

[36] M. Eisoldt, J. Gaal, T. Wiemann, M. Flottmann, M. Rothmann, M. Tassemeier, and M. Porrmann, "A fully integrated system for hardware-accelerated tsdf slam with lidar sensors (hatsdf slam)," *Robotics and Autonomous Systems*, vol. 156, p. 104205, 2022.

[37] Q. Deng, H. Sun, F. Chen, Y. Shu, H. Wang, and Y. Ha, "An optimized fpga-based real-time ndt for 3d-lidar localization in smart vehicles," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 9, pp. 3167–3171, 2021.

[38] A. Kosuge, K. Yamamoto, Y. Akamine, and T. Oshima, "An soc-fpga-based iterative-closest-point accelerator enabling faster picking robots," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 4, pp. 3567–3576, 2020.

[39] Q. Gautier, A. Althoff, and R. Kastner, "Fpga architectures for real-time dense slam," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 83–90.

[40] M. R. Gkeka, A. Patras, C. D. Antonopoulos, S. Lalis, and N. Bellas, "Fpga architectures for approximate dense slam computing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 828–833.

[41] K. Sugiura and H. Matsutani, "A unified accelerator design for lidar slam algorithms for low-end fpgas," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–9.

[42] F. Chen, R. Ying, J. Xue, F. Wen, and P. Liu, "Parallelnn: A parallel octree-based nearest neighbor search accelerator for 3d point clouds," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 403–414.

[43] M. Han, L. Wang, L. Xiao, H. Zhang, T. Cai, J. Xu, Y. Wu, C. Zhang, and X. Xu, "Bitnn: A bit-serial accelerator for k-nearest neighbor search in point clouds," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 1278–1292.

[44] M. Langhammer and B. Pasca, "High-performance qr decomposition for fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 183–188. [Online]. Available: https://doi.org/10.1145/3174243.3174273

[45] S. D. Muñoz and J. Hormigo, "High-throughput fpga implementation of qr decomposition," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 9, pp. 861–865, 2015.

[46] S. Zhang, X. Tian, C. Xiong, J. Tian, and D. Ming, "Fast implementation for the singular value and eigenvalue decomposition based on fpga," *Chinese Journal of Electronics*, vol. 26, no. 1, pp. 132–136, 2017.

[47] A. Burger, P. Urban, J. Boubin, and G. Schiele, "An architecture for solving the eigenvalue problem on embedded fpgas," in *International Conference on Architecture of Computing Systems*. Springer, 2020, pp. 32–43.