SODA: Similar 3D Object Detection Accelerator at Network Edge for Autonomous Driving

Conference Paper · May 2021

DOI: 10.1109/INFOCOM42981.2021.9488833

9 authors, including:

Wenquan Xu
Tsinghua University
13 PUBLICATIONS **39** CITATIONS
SEE PROFILE

Haoyu Song
Futurewei Technologies
61 PUBLICATIONS **2,326** CITATIONS
SEE PROFILE

Hui Zheng
Chinese Institute for Brain Research, Beijing
2 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Xinggong Zhang
Peking University
69 PUBLICATIONS **834** CITATIONS
SEE PROFILE

SODA: Similar 3D Object Detection Accelerator at Network Edge for Autonomous Driving

Wenquan Xu[1], Haoyu Song[2], Linyang Hou[1], Hui Zheng[3], Xinggong Zhang[3], Chuwen Zhang[1], Wei Hu[3], Yi Wang[4,5*], Bin Liu[1,5*]
[1]Tsinghua University, China [2]Futurewei Technologies, USA [3]Peking University, China
[4]Southern University of Science and Technology, China [5] Peng Cheng Laboratory, China

***Abstract*—Offloading the 3D object detection from autonomous vehicles to MEC is appealing because of the gains on quality, latency, and energy. However, detection requests lead to repetitive computations since the multitudinous requests share approximate detection results. It is crucial to reduce such fuzzy redundancy by reusing the previous results. A key challenge is that the requests mapping to the reusable result are only similar but not identical. An efficient method for similarity matching is needed to justify the use case. To this end, by taking advantage of TCAM's approximate matching capability and NMC's computing efficiency, we design SODA, a first-of-its-kind hardware accelerator which sits in the mobile base stations between autonomous vehicles and MEC servers. We design efficient feature encoding and partition algorithms for SODA to ensure the quality of the similarity matching and result reuse. Our evaluation shows that SODA significantly improves the system performance and the detection results exceed the accuracy requirements on the subject matter, qualifying SODA as a practical domain-specific solution.**

I. Introduction

Autonomous vehicles are gaining momentum and set to revolutionize the transportation industry in the near future. The autonomous driving system integrates several key subsystems including sensing, perception, and decision making. The massive volume of data produced by the sensing subsystem, amounting to multi-gigabytes per second [1], need to be timely processed by the perception subsystem to ensure safe decision making. The computation workload for object detection, localization, and tracking is intensive, which poses a huge challenge for autonomous vehicles. The limited on-board resource hampers the perception accuracy and breadth, and the high energy consumption incurred reduces the cruising time of the battery-powered electrical vehicles [2].

The emerging Mobile Edge Computing (MEC) equipped with advanced wireless communication technologies (*e.g.*, 5G) enables the Connected Autonomous Vehicles (CAV) to tap the edge computing resources [3]. While the critical control loops are still kept on board, certain computing tasks can be offloaded to MEC which helps produce higher quality results faster. Taking the 3D object detection as an example, the deep learning model inference for a frame of point cloud takes 60 to 200ms [4] on a general GPU (*e.g.*, GTX 2080 Super), which would take longer on other on-board platforms [5] (*e.g.*, NVIDIA Jetson TX2). If the task is instead offloaded to edge servers, more advanced and accurate models can be applied and the overall detection latency can be reduced [5].

This paper is supported by NSFC-RGC (62061160489), NSFC (62032013, 61872213), Tsinghua University (Department of Computer Science and Technology)-Siemens Ltd., China Joint Research Center for Industrial Intelligence and Internet of Things, National Key R&D Program of China (2019YFB1802600), "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019)". Corresponding Authors: Yi Wang (wy@ieee.org), Bin Liu (lmyujie@gmail.com).

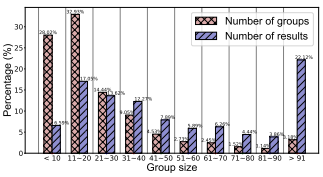


Fig. 1: Data similarity.

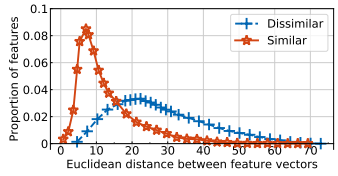


Fig. 2: Feature distance.

However, the MEC offloading for object detection raises some new challenges. Since both the networking and computing resources are shared, the huge amount of data and detection requests from a large number of vehicles can jam the communication channels and overload the edge servers, which in turn leads to unpredictable latency, rendering the detection results unreliable. The key to address such challenges lies in two often entangled aspects: reducing the data quantity and reducing the total number of detection requests.

LiDAR point clouds are the most representative formats of 3D sensory data, and the target of point cloud object detection is to estimate a 3D bounding box with a type for each object. The detection process comprises three steps: object partition, feature extraction, and result inference. Among these steps, the third step, relying on complex deep learning models, is the most time-consuming. As shown in Table I, the common point cloud detection models (*i.e.*, VoxelNet [6], Second [7], F-PointNet [8], F-ConvNet [9], and Point-RCNN [10]) take more than 90% of the overall processing time for just the result inference. The good news is that the feature vectors passed from the second step to the third step are orders of magnitude smaller in size than the original point clouds. Thus, it makes sense to only offload the third step to MEC and keep the first two steps on board.

The data reduction per request only relieves the bandwidth pressure on networks. We still need to reduce the total number of requests to relieve the computation pressure on edge servers. Fortunately, we have the opportunity to reuse the historically completed computation results for new requests by identifying and taking advantage of the request similarity rooted in the temporal and spatial locality. First, the sensing data per object does not necessarily change fast, which exposes significant temporal detection similarity from the same vehicle. Second, the vehicles in the vicinity of the same geographic location often issue similar detection requests for the same objects, and these requests tend to be relayed by the same base station or the

TABLE I: 3D detection model processing time.

Time (ms)	VoxelNet	Second	F-ConvNet	F-PointNet	PointRCNN
Step 1+2	4.13	3.56	7.22	6.19	11.87
Step 3	223.65	53.24	165.32	126.19	178.59

Road Side Unit (RSU) and then processed by the same edge servers. Therefore, reusing the previous computation results can be effective in reducing the edge server load and detection latency. Here resides our key contribution.

Fig. 1 exhibits the statistics of data similarity on the KITTI dataset [11], the most popular dataset in the field of autonomous driving. By dividing all data into different similar groups by their Intersection over Union (IoU)—IoU is a vital metric measuring the accuracy of a 3D object detection model [6, 8], where a detection result is judged to be correct if its IoU with ground truth is greater than 70% for vehicles, and 50% for pedestrian and cyclist, according to KITTI benchmark [4]—the results show a significant data similarity: the number of groups is only 5% of the number of data, among which 70% of groups covers more than 90% of data, and each group has similar results greater than 10.

Although repetitive requests are evident, the reuse of previous computation results is not so straightforward, mainly because the feature vectors for the same object generated by different vehicles or by the same vehicle at different time are different. The conventional exact-key hashing schemes are of no use in this case. Fortunately, as shown in Fig. 2, the feature vectors sharing the approximate reusable result have smaller Euclidean distance than features with unreusable results, implying that we can quantify result reusability by feature distance. However, direct feature distance computing and comparison are time-consuming, given the number of vectors to compare is large (*e.g.*, >70K) and each high-dimensional vector is composed of more than 1K floating-point values. We need an efficient approach for similar vector matching with high confidence (*e.g.*, high precision and recall ratios) to make the computation reuse practical. To this end, we develop a hardware-based similar object detection accelerator, SODA, using TCAM and an associated Near Memory Computing (NMC) module. Specifically, SODA uses TCAM's ternary matching specialty to realize the approximate matching and narrows down the candidate feature vectors to just a few similar ones, and then quickly compares the query vector with these candidates to retrieve a reusable result through NMC. To fit high-dimensional feature vectors in TCAM, we develop an algorithm to transform feature vectors into short binary codes while preserving the feature similarity. To improve TCAM matching precision and hit rate simultaneously, we design a greedy algorithm to aggregate the code words to ternary bit strings. Moreover, to improve the quality of the reusable results, we develop a novel culling strategy for NMC.

Architecturally, the best location for SODA is at the base station or the RSU. The previous object detection results can be cached here. Once a detection request is identified as similar enough with a previous one, the preserved result is directly returned; otherwise, the request is forwarded to an edge server for complete model inference. In this way, we achieve: (1) a large number of requests from vehicles are intercepted and served on the base station, leaving only a light detection load to MEC servers; (2) a vast number of requests can get results

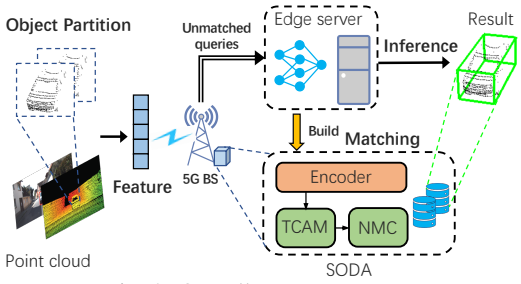


Fig. 3: Overall system structure.

with very low latency, greatly improving user experience; (3) the network bandwidth consumption is also reduced.

In summary, we make the following major contributions: (1) we are the first to apply the in-network similar object detection accelerating technique to the autonomous driving scenario and address its unique challenges; (2) we design a specialized dual-loss encoding algorithm to encode high-dimensional feature vectors to short binary codes while preserving the feature similarity, which is also applicable to other scenarios; (3) we propose a novel TCAM plus NMC architecture, in which TCAM performs coarse-grained approximate matching and NMC conducts fine-grained feature qualifying, ensuring fast and accurate reusable result search and retrieval.

In the remainder of the paper, Sec. II briefs the related work, Sec. III details SODA's architecture and algorithms, Sec. V evaluates the performance, and Sec. VI concludes the work.

II. Related Work

Research on MEC-assisted autonomous driving is emerging but previous works have different focuses: survey the opportunities and challenges [12, 13], design models [5, 14, 15], support cross-vehicle cooperative perception [16–18], or construct a data analytic platform [19]. All these works directly rely on MEC servers and none of them considers applying in-network accelerators to improve the performance of critical tasks such as 3D object detection.

Edge-based detection result reuse has been proposed in other application scenarios. Cachier [20] first presents result reuse for recognition applications with a focus on cache entry optimization. Potluck [21] implements a cache service to share processing results between applications on individual devices for Augmented Reality (AR). Foggycache [22] proposes to perform cross-device approximate computation reuse on the edge server, where A-LSH is used to conduct content lookup and H-kNN for high-quality result retrieval. However, A-LSH has low efficiency in precision and recall, and H-kNN incurs large latency for high-dimensional point cloud data, making these methods inapplicable for autonomous driving.

For the first two steps of 3D object detection (*i.e.*, partition and feature extraction), some works [6, 7] divide a point cloud frame into many uniform cubes called voxel and extract features from them, which however easily break the structure of each object; some other works [8–10] utilize a learned network (*e.g.*, PointNet [23] and PointSIFT [24]) to segment point cloud into different objects with perfect structure. For the last step (*i.e.*, 3D bounding-box estimation), it is common to adopt various complex models based on some deep neural networks such as CNN.

TCAM is widely used in high-speed networking devices mainly for IP lookup and packet classification [25, 26]. In recent years, TCAM finds its application in other compute-intensive tasks. In [27], TCAM is used as an augmented memory to accelerate neural network training, and the work is extended with improved accuracy in [28]. In [29], TCAM is used to implement LSH, which improves searching accuracy for similar content. In [30], TCAM is also used for similarity search where the range encoding on binary reflected gray code is adopted to realize similar content matching.

NMC, as a Processing-In-Memory (PIM) technology, embedding dedicated computation logic alongside the memory, can greatly reduce the data I/O latency and boost computation speed. NMC has been widely used to accelerate data-intensive tasks (*e.g.*, machine learning) [31–33], which can achieve up to 320 GB/s throughput based on 3D-stacked memories (*e.g.*, HMC [34] and HBM [35]).

III. Design Of SODA Model and Algorithms

A. Overall System Structure

As illustrated in Fig. 3, different from the traditional MEC-based system in which all computing tasks are offloaded to the edge, we deploy a TCAM-NMC sub-system on the base station or RSU between vehicles and MEC to identify repetitive similar 3D object detection requests for approximate result reuse. We first build a knowledge database using the previously computed detection results to populate the TCAM and NMC tables. As a result, the TCAM-NMC sub-system can quickly detect if a query feature from a vehicle matches any existing result so unnecessary remote server computation can be avoided.

This process is expressed by two functions, namely `DB_build` and `DB_query`, as shown in Algorithm 1. `DB_build` has two stages: build a code matching table for TCAM, and build a reusable result database in NMC. In the first stage, we train a binary encoder (line 2) by leveraging Algorithm 2 in Sec. III-B, and obtain the binary codes of the dataset features by the encoder (line 3), which are aggregated to improve the storage and matching efficiency (line 4) using the aggregation strategy discussed in Sec. III-C. In the second stage, we leverage the *culling strategy* discussed in Sec. III-C to refine the dataset features to ensure reusability (line 5), and store the code table and the result database in TCAM and NMC respectively (line 6).

`DB_query` supports the query process. We first encode the query feature $\boldsymbol{x}$ coming from a vehicle into binary code using the trained encoder (line 9), and then perform code match in TCAM (line 10). The successful matches return indicators to guide the query feature $\boldsymbol{x}$ to search the reusable results in NMC (line 14). In case the TCAM miss-match occurs or the NMC search fails, $\boldsymbol{x}$ is delivered to MEC for object detection computing; otherwise, the locally stored similar result is directly returned to the requesting vehicle.

B. Dual-loss Supervised Encoding for TCAM

We claim two features are similar if they yield the reusable object detection result. Due to the limited depth and width of TCAM, a compact and accurate encoding scheme is needed to transform the high-dimensional features to short binary codes while preserving their similarity relationships. We use a supervised method to learn a binary encoder.

Algorithm 1: System functional process

Input: feature set $\boldsymbol{X}$, detection results $\boldsymbol{Y}$, query feature $\boldsymbol{q}$, and new insertion result $\boldsymbol{x}$.

1 **Function** `DB_build(`$\boldsymbol{X}$, $\boldsymbol{Y}$`)`**:**
2 $Encoder \leftarrow$ call Algorithm 2;
3 $\boldsymbol{B} \leftarrow Encoder(\boldsymbol{X})$;
4 $\boldsymbol{T} \leftarrow$ call Algorithm 3 to aggregate $\boldsymbol{B}$;
5 $< \boldsymbol{X}_{new}, \boldsymbol{Y}_{new} > \leftarrow Culling(\boldsymbol{X}, \boldsymbol{Y})$;
6 add $< \boldsymbol{X}_{new}, \boldsymbol{Y}_{new} >$ in NMC, add $\boldsymbol{T}$ in TCAM;
7 **return** $Encoder$, $\boldsymbol{Y}_{new}$;
8 **Function** `DB_query(`$\boldsymbol{q}$, $Encoder$`)`**:**
9 $\boldsymbol{b} \leftarrow Encoder(\boldsymbol{q})$;
10 $Addr \leftarrow TCAM - match(\boldsymbol{b})$;
11 **if** $Addr == Null$ **then**
12 Offload $\boldsymbol{q}$ to MEC;
13 **else**
14 $Result \leftarrow Calc\_NMC(\boldsymbol{q})$;
15 **if** $Result == Null$ **then**
16 Offload q to MEC;
17 **else**
18 **return** $Result$;

Problem Formulation: Let $\boldsymbol{\mathcal{X}} = \{\boldsymbol{x_i}\}_{i=1}^{N} \subset \mathbb{R}^d$ denote the set of training features, and their corresponding label is $\boldsymbol{\mathcal{Y}} = \{y_i\}_{i=1}^{N} \subset \mathbb{R}$. Based on $\boldsymbol{\mathcal{Y}}$, we have the similarity matrix $\boldsymbol{S} \subset \mathbb{R}^{N \times N}$, in which $s_{ij} = 1$ if $y_i = y_j$, or $s_{ij} = -1$ otherwise. The task is to learn a group of K encoding functions $H(\cdot) = \{h_k(\cdot)\}_{k=1}^{K}$ to map $\boldsymbol{\mathcal{X}}$ to K-bit binary codes $\boldsymbol{B}$=$\{\boldsymbol{b_i}\}_{i=1}^{N} \subset \{-1, 1\}^{N \times K}$ (*i.e.*, $h(x_i) \mapsto \{-1, 1\}$). Here we use -1 to denote 0 in B due to the need of subsequent inner product calculation.

Supervised Encoding: The learned binary code should preserve the similarity of $\boldsymbol{\mathcal{X}}$ (*i.e.*, the codes for the similar features should have small Hamming distance). Typically, the supervised encoding can be done in one or two steps. The one-step approach (*e.g.*, KSH, SDH [36, 37]) learns the encoding functions under the similarity information, which is hard to optimize as the code for each bit is interdependent and unable to share a common loss function during training. The two-step approach (*e.g.*, [38, 39]) enjoys a higher accuracy as it infers a binary code by the given label first, and then learns the encoding functions according to the binary code.

To ensure real-time encoding, we design a new two-step learning strategy. In step 1, the Block Graph Cut method [38, 40] is used to infer binary codes $\boldsymbol{B}$=$\{\boldsymbol{b_i}\}_{i=1}^{N}$ under the hinge loss function $\sum_{i=1}^{N}\sum_{j=1}^{N}\left(\frac{1}{2}(1 + s_{ij})D_h(\boldsymbol{b_i}, \boldsymbol{b_j}) + \frac{1}{2}(1 - s_{ij})max(\frac{1}{2}K - D_h(\boldsymbol{b_i}, \boldsymbol{b_j}), 0)\right)$, where $D_h(\cdot, \cdot)$ is the Hamming distance between two binary codes, $s_{ij} \in \{-1, 1\}$ indicates the similarity between $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$, and K is the width of the target binary code. For each bit of $\boldsymbol{B}$, in step 2, we construct a corresponding encoder (*i.e.*, a binary classifier) to label the features as '-1' and '1'.

For the high-dimensional binary classification problem, the existing solutions, such as kernel-based SVM [39, 41] and boosted decision trees [38, 40], cannot meet the demand of high-speed processing. The former incurs long time due to the complex kernel calculation; the latter is inefficient for the recursive feature segmentation and subtree establishment.

Instead, we adopt *Linear Discriminant Analysis* (LDA) [42] with a combined simple linear classifier, where LDA is used for projection and the classifier for quantization. Specifically, for a high-dimensional feature vector $\boldsymbol{x} \in \mathbb{R}^{1\times d}$, we establish a projection matrix $\boldsymbol{P} \subset \mathbb{R}^{d\times m}$ to extract its effective discriminative feature as a new compact m-dimensional vector, with the purpose of keeping the similarity structure of original data points in Euclidean space. That is, we expect the projected vectors of the similar features are close in the projection space, while keeping those dissimilar ones distant. To achieve this, we apply LDA to learn the effective projection matrix $\boldsymbol{P}$ under the label $\boldsymbol{Y}$: LDA computes the intra-class scatter matrix $\boldsymbol{S_w}$ and inter-class scatter matrix $\boldsymbol{S_b}$ according to label $\boldsymbol{Y}$, and learns the optimal projection matrix by maximizing the target:

$$\underset{\boldsymbol{P}}{max}\ J(\boldsymbol{P}) = tri(\frac{\boldsymbol{P}^T\boldsymbol{S_b}\boldsymbol{P}}{\boldsymbol{P}^T\boldsymbol{S_w}\boldsymbol{P}}) \quad s.t.\ \boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{I}, \tag{1}$$

where $tri(\cdot)$ indicates the trace operator. We obtain the optimal projection matrix $\boldsymbol{P^*} = \underset{\boldsymbol{P}}{argmax}\ J(\boldsymbol{P})$. This way, $\boldsymbol{x}$ is projected into a discriminative m-dimensional vector $\boldsymbol{xP^*}$.

Next, we adopt simple linear binary classifiers as encoding functions. Especially, to improve accuracy, we train a group of H linear classifiers, among which we select the high-performance classifiers by using a confidence coefficient vector $\boldsymbol{\alpha} \subset \mathcal{R}^H$. Thus, the encoding function for each bit of the target code is defined as:

$$h(\boldsymbol{x}) = sgn\big(\sum\nolimits_{h=1}^{H} \alpha_h C_h(\boldsymbol{xP^*})\big) \tag{2}$$

where $sgn(\cdot)$ is the sign function, $\boldsymbol{C} = [C_1, C_2, \cdots, C_H]$ are the simple linear classifiers which can be denoted as $C(\boldsymbol{x'}) = sgn(\boldsymbol{x'w}^T + b)$, and $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \cdots, \alpha_H]$ is the confidence coefficient vector for balancing the classifiers. The confidence coefficient $\boldsymbol{\alpha}$, weights $\boldsymbol{w} = [w_1, w_2, \cdots, w_m]$, and bias $\boldsymbol{b} = [b_1, b_2, \cdots, b_H]$ are the parameters we need to learn for each bit encoding function by minimizing the following loss Q_k:

$$Q_k = exp(-b_k \sum\nolimits_{h=1}^{H} \alpha_h sgn(\boldsymbol{xP^*w}_h^T + b_h)),\ k \in [1, K]. \tag{3}$$

The loss Q in Eq. (3) only considers the inferential binary code but ignores the similarity information contained in label Y which is informative for the classifier training. To amend this, we entail the pairwise similarity matrix S in encoding function learning, and introduce the loss $R = \|K\boldsymbol{S} - \boldsymbol{BB}^T\|_F^2$, where $\|\cdot\|_F$ represents the Frobenius norm, and $\boldsymbol{BB}^T = \sum_{k=1}^{K} h_k(\boldsymbol{X})h_k(\boldsymbol{X})^T$. We adopt a greedy method similar to that in [36, 38] to sequentially optimize R one bit a time, conditioning on the previously solved bits. Specifically, when dealing with the k-th bit, we have learned the optimal k-1 bits, and the cost for k-th bit is:

$$\begin{aligned} R_k &= \|kS - \sum\nolimits_{t=1}^{k-1} h_t^*(\boldsymbol{X})h_t^*(\boldsymbol{X})^T - h_k(\boldsymbol{X})h_k(\boldsymbol{X})^T\|_F^2 \\ &= -2h_k(\boldsymbol{X})^T(kS - \sum\nolimits_{t=1}^{k-1} h_t^*(\boldsymbol{X})h_t^*(\boldsymbol{X})^T)h_k(\boldsymbol{X}) + const \end{aligned} \tag{4}$$

where $h^*(\cdot)$ is the learned optimal encoding function for previous bits, and $h(\cdot)$ is the encoding function needs to be learned. Next, we get the final loss $L_k = Q_k + \lambda R_k$ with the balancing weight λ. During the optimization, as $sgn(\cdot)$ is discrete and hard to train, we relax the problem in Eq. (3) and (4) by replacing $sgn(x)$ with a sigmoid-shaped function $\psi(x) = \frac{2}{1+e^{-x}} - 1$. Then we conduct the gradient descent method to optimize $\boldsymbol{\alpha}, \boldsymbol{w}, and\ \boldsymbol{b}$ simultaneously by minimizing the loss L_k. Finally, we get the optimal K-bits encoding functions as $\boldsymbol{\Phi} = [h_1^*, h_2^*, \cdots, h_K^*]$. The complete algorithm is described in Algorithm 2.

Algorithm 2: Dual-loss Encoding

Input: feature set $\boldsymbol{X}$, pairwise similarity $\boldsymbol{S}$, projection dimension m, the width of binary codes K, iteration number T_{max}, and parameter λ.
Output: projection matrix $\boldsymbol{P}^*$, and K encoders $\boldsymbol{\Phi}$.

```
Step 1: apply Graph cut to get K-bits binary code B;
Step 2: get the optimal projection matrix P* by LDA;
for k ← 1 to K do
    randomly initialize α_k, w_k and b_k;
    compute loss L_k = Q_k + λR_k by Eq. (3) and (4);
    use gradient descent to minimize the loss L_k with T_max budget
      iterations, achieving α*_k, w*_k, b*_k;
    h*_k ← sgn(Σ_{h=1}^{H} α*_h sgn(xP* w*_h^T + b*_h));
```

C. High Speed TCAM-NMC Accelerator

As aforementioned, SODA uses TCAM to perform approximate matching and NMC to evaluate the quality of matched results to seek reuse opportunities. We partition the original data pairs, <feature, result>, so that the data pairs having similar detection results are grouped together and labeled with a partition ID. The partitions are stored in NMC. The feature lookup in TCAM returns one or more partition IDs, which are used to trigger searches in corresponding NMC partitions.

Weighted Patricia trie: Assume we have M sets binary codes $\boldsymbol{B} = \{\boldsymbol{b}_i\}_{i=1}^M$, and each code set $\boldsymbol{b}_i$ corresponds to a partition of similar results stored in NMC with partition ID denoted as $\{l_i\}_{i=1}^M$. As a preparation of code aggregation, we first construct a weight Patricia trie for binary codes $\boldsymbol{B}$, where each leaf node of the existing code will be set a weight to denote its proportion in code set $\boldsymbol{b_i}$, as shown on the left side of Fig. 4. Then we represent original codes $\boldsymbol{B}$ by more compact codes $\boldsymbol{B}^*$ which is composed of the discriminant bits (*e.g.*, '100110' can be denoted as '00' by the 3rd and 6th bits). Finally, Fig. 4 shows the complete binary trie of new code $\boldsymbol{B}^*$, where we express the non-existing codes by virtual leaf node called *empty code* which plays a vital role during aggregation.

Code aggregation for TCAM: The binary code aggregation has two purposes: (1) reduce the number of binary codes to fit in TCAM; (2) take advantage of the wildcard match '$*$' for approximate matching in TCAM to increase the correct query hit rate. The first purpose can be achieved by aggregating existing codes with similar detection results; the second purpose is based on the finding that, due to limited encoding accuracy, the difference between the binary codes of similar features makes some queries fail to find a match in TCAM even though the similar features exist in NMC. To address this issue, besides the aggregation between two existing codes, we especially allow the existing code to aggregate with *empty code* so as to construct a code coverage space within hamming distance h, where h is the number of aggregation bit '$*$' (*e.g.*, aggregating an existing code '1011' with an empty code '1000'

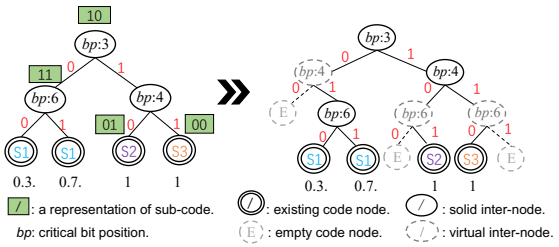


Fig. 4: *The weighted patrcia trie (left) and the extended trie (right), where 'S1', 'S2' and 'S3' denotes code node of dissimilar groups.*

to get the aggregated code '10∗∗', which covers a space with the hamming distance of 2).

Assume each $\boldsymbol{b}_i \in \boldsymbol{B^*}$ contains num_i different binary codes and each code is associated with a size denoted by c. We define the gain of the aggregation strategy f as:

$$G(f) = h_t(f) - \mu h_e(f) \quad s.t.\ C(f) \leq \Omega, \tag{5}$$

where h_f is the frequency of correctly matched queries under the aggregation strategy f, e_f is the frequency of wrongly matched queries, μ is the adjustable parameters to balance h_f and e_f, $C(f) = \sum_{i=1}^{M} c \cdot num_i$ is the total storage cost for all code entries, and Ω is the capacity of TCAM. The target problem can be stated as follows.

Definition 1. *Aggregation gain maximization problem:* From all the possible aggregation strategies F, find $f^* \in F$ such that for any aggregation strategy $f \in F$, the conditions $G(f^*) \geq G(f)$ and $C(f^*) \leq \Omega$ hold.

The problem is NP-hard, as the aggregation does not enlarge $C(f)$, and if the constraint in Eq. 5 were satisfied, the problem can be transformed into a non-prefix aggregation problem targeting for minimizing the storage cost, which has been proved NP-hard [43].

For the aggregation process, we have the following observations. Each aggregation operation can only lead to three results: (1) aggregation expands covering range, which increases both h_f and e_f; (2) aggregation overlaps the other dissimilar codes, which increases e_f; (3) the number of codes is reduced, which reduces $C(f)$. We have the gain $\Delta G(a)$ and the cost $\Delta C(f)$ for each aggregation operation $a \in f$ as:

$$\Delta G(a) = \begin{cases} \sum_{i=1}^{n_1} \sum_{k=1}^{K} \sum_{j=1}^{S_{ik}} (-\mu)^{\epsilon} P(l_j) \omega_j p_b^k (1-p_b)^{K-k}, & if ① \\ -\sum_{i=1}^{n_2} \mu P(l_i) \omega_i, & if ② \end{cases} \tag{6}$$

$$\Delta C(a) = -n_3 c. \qquad if ③$$

For the case 1, the positive gain $\Delta h_t(a)$ is derived from the possibility that similar queries with non-existing codes covered by aggregation hit the TCAM, which cannot occur before the aggregation a. Meanwhile, the aggregation a also brings in negative gain $\Delta h_e(a)$ when the dissimilar queries hit aggregation code. Hence we compute the probability for the above two cases to evaluate $\Delta h_t(a)$ and $\Delta h_e(a)$. The computation requests with non-existing codes are two kinds (*i.e.*, either similar with preserved results or not), and we approximately estimate the probability of the two cases as: probability $P_q(b_e|l)$ denotes a query with non-existing *empty code* b_e has the similar request with group $\boldsymbol{b}_l$. Specifically, we assume each binary code can be transformed to b_e in a probability $P_t(k)$,

Algorithm 3: Greedy Aggregation

Input: the M sets of binary codes $\boldsymbol{B} = \{\boldsymbol{b}_i\}_{i=1}^{M}$, balancing parameter μ, and TCAM capacity Ω.
Output: the desired aggregation codes.

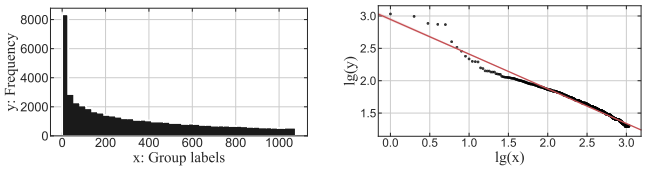
```
B*, E, ω ← construct_weighted_patricia_trie(B);
G ← ∅, C ← cost(B*);
for i ← 1 to M do
    t ← B*[i];
    for s1 ← 1 to length(t) do
        for s2 ← s1 + 1 to length(t) do
            add pair < t[s1], t[s2] > to G[i];
        for s3 ← 1 to length(E) do
            add pair < t[s1], t[s3] > to G[i];
while max(calc_gain(G)) > 0 or C > Ω do
    if C ≤ Ω then
        i, < x, y > ← arg max_{<x,y>} calc_gain(G);
    else
        for pair in G do
            if ΔC(pair) < 0 then
                add pair to Pr;
        i, < x, y > ← arg max_{<x,y>} calc_gain(Pr);
    t_code ← aggregate(x, y);
    M_code ← find_match(t_code, B*[i]);
    for code in M_code do
        remove pair < code, ∗ > from G[i];
    remove M_code from B*[i];
    for code in B*[i] do
        add pair < code, t_code > to G[i];
    add t_code to B*[i];
return B*;
```

which is inversely related to the number of different bits k. We define the probability of a one-bit change ('1' to '0' or the reverse) as p_b, and thus $P_t(k) = p_b^k(1-p_b)^{K-k}$, where K is the bit width of b_e. Then we can approximate $P_q(b_e|l) = \sum_{b \in \boldsymbol{b}_l} P(l)\omega_b P_t(k_b)$, where $P(l)$ (detailed in the next part) is the probability of a query having the similar request with the group $\boldsymbol{b}_l$, and ω_b is the weight of b in Patricia trie. Thus, we have $\Delta h_t(a) = \sum_{i=1}^{n_1} P_q(b_{ei}|l_i)$, $\Delta h_e(a) = \sum_{i=1}^{n_1} P_q(b_{ei}|l \neq l_i)$, and $\Delta G(a) = \sum_{i=1}^{n_1}\sum_{k=1}^{K}\sum_{j=1}^{S_{ik}}(-\mu)^{\epsilon}P(l_j)\omega_j p_b^k(1-p_b)^{K-k}$, where n_1 is the number of *empty codes* covered by the aggregation, S_{ik} is the number of binary codes having k different bits compared with the i-th *empty code*, and $\epsilon \in \{0,1\}$ is 0 when l_j equals to the label of group $\boldsymbol{b}_i$ and 1 otherwise.

The case 2 only involves negative gain. Each covered code b that has the dissimilar computation request with the aggregation code will wrongly hit the aggregation code with a probability of $P(l_b)w_b$. Thus similarly, $\Delta G(a) = -\sum_{i=1}^{n_2} P(l_b)w_b$, where n_2 is the number of dissimilar codes covered by the aggregation. The case 3 reduces the needed storage, and thus $\Delta C(a) = -n_3 c$, where n_3 is the number of reduced entries. Now we can rewrite $\Delta G(a)$ in Eq. 6 as:

$$\sum_{i=1}^{n_1}\sum_{k=1}^{K}\sum_{j=1}^{S_{ik}}(-\mu)^{\epsilon}P(l_j)\omega_j p_b^k(1-p_b)^{K-k} - \sum_{i=1}^{n_2}\mu P(l_i)\omega_i \tag{7}$$

Based on Eq. 7, we develop a greedy aggregation algorithm as described in Algorithm 3, where in each step we greedily select two codes with maximum gain ΔG to aggregate among all codes if the constraint $C(f) \leq \Omega$ is satisfied, or from those



(a) Frequency histogram (b) Logarithmic relationship

Fig. 5: *Distributions of group labels.*

code pairs with $\Delta C < 0$ otherwise. To be specific, line 1 in Algorithm 3 constructs a Patricia trie to generate the new compressed binary codes set $\boldsymbol{B}^*$, the *empty codes* set $\boldsymbol{E}$, and the weights $\boldsymbol{\omega}$; line 2 obtains the initial storage cost C of all binary codes; line 3-9 list all possible aggregation code pairs $\boldsymbol{G}$ on $\boldsymbol{\omega}$ and $\boldsymbol{E}$; all the aggregation decisions are made in the loop from line 10 to line 25, where line 11-17 select the appropriate code pair $<x, y>$ based on whether $C \leq \Omega$ is satisfied, and line 14-16 extract code pairs from $\boldsymbol{G}$ with $\Delta C < 0$. Next, line 18 aggregates the selected code pair into a new ternary code, and line 19-22 remove the codes and pairs covered by the aggregation code from $\boldsymbol{B}_{new}$ and $\boldsymbol{G}$. At last, line 26 returns $\boldsymbol{B}_{new}$ as the final codes to be stored in TCAM.

Query distribution. For the probability $P(l)$ of a query having the similar computation request with $\boldsymbol{b}_l$, we derive an estimation model based on the real data distribution. Fig. 5(a) shows the statistics on the frequency distribution of KITTI dataset by similarity of group labels that measured in Section I, and Fig. 5(b) shows their logarithmic relationship. The observation confirms our assumption that the $P(l)$ obeys the power-law distribution with the probability density function $P(l) = \lambda l^{-\alpha}$, on which we take logarithm on both sides and get: $lg(P(l)) = -\alpha lgl + lg\lambda$. Assume we have observation data of logarithmic group labels $\{x_i = lg(l_i)\}_{i=1}^{M}$ and logarithmic frequency $\{y_i\}_{i=1}^{M}$. We apply the least squares method to estimate $k = -\alpha$ and $b = lg\lambda$ by minimizing the sum of squared errors $\sum_{i=1}^{M}(y_i - \hat{y}_i)$, where $\hat{y}_i = kx_i + b$ is the predicted value of y_i. We get:

$$k = \frac{\sum_{i=1}^{M}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{M}(x_i - \overline{x})^2}, \; b = \overline{y} - k\overline{x},$$

where $\overline{x}$ and $\overline{y}$ are average observation values. Finally we get the estimation values $\hat{\alpha} = -k$ and $\hat{\lambda} = 10^b$.

Homogeneity-based Culling for NMC: The match in TCAM may be false positive for the following reasons: (1) some impure data with dissimilar features may result in the false match; (2) the similarity in the feature cannot guarantee the reusability of the detection result due to detection error, feature error, etc. Therefore, we need to determine a qualified result for reuse. Naively, one can traverse all the matched data to find a result with similar feature to the query, which solves only the first issue, and also incurs large search latency. The work in [22] applies kNN to search k results having the closest feature with the query returns a result with high proportion, which addresses both issues but still suffers high latency when performing kNN on all high-dimensional features.

To tackle the issues with low cost, we propose an offline homogeneity-based culling strategy for NMC. On the one hand, we qualify each stored result and eliminate redundant,

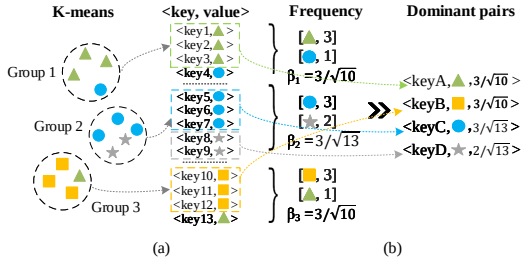


Fig. 6: *Homogeneity-based culling: (a) all similar key-value pairs in a partition are first categorized into different groups via K-means on keys; (b) each group is then qualified based on composition frequency and the sufficient groups vote for high-quality dominant pairs.*

insufficient results during NMC construction; on the other hand, we derive a quality-first search scheme to further reduce the search time. As illustrated in Fig. 6, we denote the contents to be stored in NMC as *<key,value>* pairs, where *key* is the high-dimensional feature and *value* is the corresponding detection result. For each partition of similar pairs, we first utilize the K-means to cluster them into $\mathcal{K}$ groups by *keys*. Then for the contents in each group, we define *homogeneity level* β to quantify the result reusability.

Specifically, for the p-th group contents, we assume there are z distinct kinds of *values*, and correspondingly we can obtain a *value* frequency vector $\boldsymbol{F}_p = [f_{p1}, f_{p2}, \cdots, f_{pz}]$, where each element f_{pq} ($q \in [1, z]$) records the frequency of $q-th$ distinct *value*. Then, we define $\beta_p = max(f_{pq})/\|\boldsymbol{F}_p\|_2$. As a toy example illustrated in Fig. 6, the group 1 has 4 *<key,value>* pairs with 2 distinct *values*, and thus $\boldsymbol{F}_1 = [3, 1]$ and $\beta_1 = 3/\sqrt{3^2+1}$. Then we determine whether the content is reusable according to the homogeneity level β. The group with β exceeding a predefined threshold β_{th} indicates a high-quality result, which should be preserved; otherwise, the group is discarded. Then for each reusable group, the voting is performed to acquire dominant pairs with high confidence (*i.e.*, take the voted results as *value*, and take the central feature as *key*), as illustrated in Fig. 6. We use these dominant pairs to rebuild the partitions with higher quality and smaller size, and store them in NMC in the order of decreasing β.

During the query process, each query feature compares with the feature of dominant data pairs by computing their Euclidean distance to find the desired result in the order of β (*i.e.*, reusability). If the distance is lower than the expected threshold, the corresponding *value* is returned.

IV. System Implementation

First, we introduce the complete architecture of our line-speed accelerator engine in general. As demonstrated in Fig 7, the whole system has two functional part, i.e., multi-match logic and parallel search logic. For an incoming high-dimensional feature to be calculated, it will first be encoded into a binary code by encoder, and then be delivered to multi-match logic for a matching operation. The multi-match logic will return a set of partition numbers indicating which subsequent memories and which block inside each memory may contain the result if the query feature is matched by TCAM. Otherwise, the multi-match logic will return an enable signal to forward the feature to remote MEC for conventional object detection calculation. Then according to those returned

message by TCAM, the parallel search logic will deliver the original query feature into different NMC units for further parallel result retrieval. Considering that TCAM has a much faster speed than NMC in search logic, we adopt multiple independent NMCs to pursue a parallel search for different queries that matched by TCAM, and especially we apply a load-balanced strategy to allocate the data partitions among different NMCs with a uniform traffic load. Correspondingly, we store the aggregated binary codes of data features in TCAM and the data location messages in SRAM for fast match.

1) The Multi-match logic:

Conventional TCAM only return a single matched result with the highest priority, which is not appropriate for our approximate matching where the purpose is to match similar content as much as possible. To enable TCAM with multi-match function, there are two approaches: (1) design a new TCAM by replacing the inside priority encoder with a multi-match encoder; (2) equip the existing TCAM with a peripheral function circuit to achieve multi-match. The former approach is not practical for no available commercial product support and not flexible due to fixed matching number. Thus, we adopt the latter approach by designing an extra multi-match logic.

We divide the code entries in TCAM into different clusters where any two clusters have no overlap, and then we can draw an obvious conclusion that for an any query code, all codes that can be matched only exist in the one of above clusters. Therefore, we assign each code in above non-overlap cluster with a discriminator index (the code in different cluster can have same index), as shown in Fig 7 where each ternary code is associated with a cluster-level discriminator filed (with length of $\lceil \log_2 C \rceil$ where C is the number of code entries of the largest cluster).

To well balance the matching accuracy and search overhead, we design a two-stage multi-match logic. In stage-1, we iteratively match m times (if the number of code that can be matched by a query is less than m, we can stop iteration directly) for a query where we set m a relatively small value by statistic data to attain an acceptable accuracy. The detailed process is similar in [44], for the first round match, the query code q in "FIFO1" will be concatenated with a mask postfix composed of wild characters '$*$' of length $\lceil \log_2 C \rceil$, and so that the first query can match any codes in TCAM without limitation of discriminator field. For cluster matched by q, we suppose all discriminator index are from range $[s, e]$ (we let the index sequence same with priority sequence in TCAM, i.e., e has the highest priority.), and the fist matched code is $a \in [s, e]$, then for the subsequent match of q, we can concatenate it with a mask with range $[s, a)$ [1] by feedback logic ($(a, e]$ should not be considered). In stage-2, the query code with a mask from stage-1 will be delivered to the "buffer1", where it will wait until the output signal of parallel search logic to determine whether to be matched by TCAM again or to be dropped. During the waiting period, subsequent other queries still can be delivered to "FIFO1" and perform first stage matching of TCAM, and the switching between the two stage matching is controlled by a selector. According to [**?**], the time for matching continuous m results is $(1 + \lceil \log_2 C \rceil + (m-2) * (\lceil \log_2 C \rceil - 1))\ cycle$.

2) The parallel search logic: In the parallel search logic, the FIFOs denoted as "FIFO3" in Fig 7 for each NMC consist of two parts, i.e., a long queue to be processed with low priority and a short queue with high one. For the results returned in the first stage match of TCAM, we deliver them with query feature to the long queue, and the short queue is specially designed for results from the second stage match which should be processed with high priority to compensate the time cost for failing to retrieve result in the first stage. During search process, the query feature will be delivered to corresponding NMC unit to compute the euclidean distance with each feature of matched similar results. The above calculation for high-dimensional features will incur very high latency in traditional von Neumann architecture for there existing high volumes data transfer from the memory to processor. To break the "memory wall" [45], processing-in-memory (PIM) [46, 47] architecture has been proposed to reduce the above latency by embedding the calculation logic inside the memory, which cater the growing demand for large-scale data analytic. PIM has two category: (1) near-memory computing (NMC) places calculation logic near large-volume memory array, which has been widely used in [31–33] based on products of HMC [34] and HBM [35] [2]; (2) in-memory computing (IMC) redesigns the memory array with inside calculation logic, which has low capacity and is unavailable at present.

NMC greatly reduces the latency from $n * (t_{IO} + t_{calc})$ of traditional memory to $t_{IO} + n * (t_{intra} + t_{calc})$ where n is the number of entries to be calculated, t_{IO} is the I/O latency between CPU and memory, t_{intra} is the intra latency involved by data transfer between stacked memory and logic layer, and t_{calc} is the calculation time. Although NMC has much lower latency than traditional CPU-memory structure, it still has speed gap with TCAM [3]. Therefore, we adopt multiple independent NMCs which are allocated with different partition of results to achieve parallel search for different features. According to the query distribution derived from statistics results in section III-C, we apply a load-balanced strategy proposed in [48] to evenly distribute the partition-based result groups into different NMC chips with at most 25% duplication of results to pursue high throughput of search logic with balanced traffic load among all NMCs. Then for each result matched by TCAM, the scheduler will parse it and concatenate it with the corresponding original feature insured by ordering unit, and then send the concatenated result to the proper NMC according to the returned information of TCAM and the FIFO status of NMC.

3) The match and search process:

In order to explain the system logic clearly, we present the data lookup process in the form of state transition diagram, as illustrated in Fig 9(a) and Fig 9(b). There are both 4 states for

[1]E.g., suppose $[s, e] = [0, 7]$ and $a = 5$, and then the range $[0, 4]$ can be divided as $[0, 3] = 0 * *$ and $[4] = 100$, and the mask for the next round match is 100 that has a higher priority than the other one.

[2]Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM) are products from different companies, which consist of a logic die stacked with several DRAM devices that has at most 320 GB/s bandwidth between stacked DRAM and logic die connected by through-silicon vias (TSV).

[3]For TCAM, the average time for a match is only $(1 + \lceil \log_2 C \rceil + (m - 2) * (\lceil \log_2 C \rceil - 1))/m = 7.25\ cycle$ if $m = 4$ and $C = 1024$.

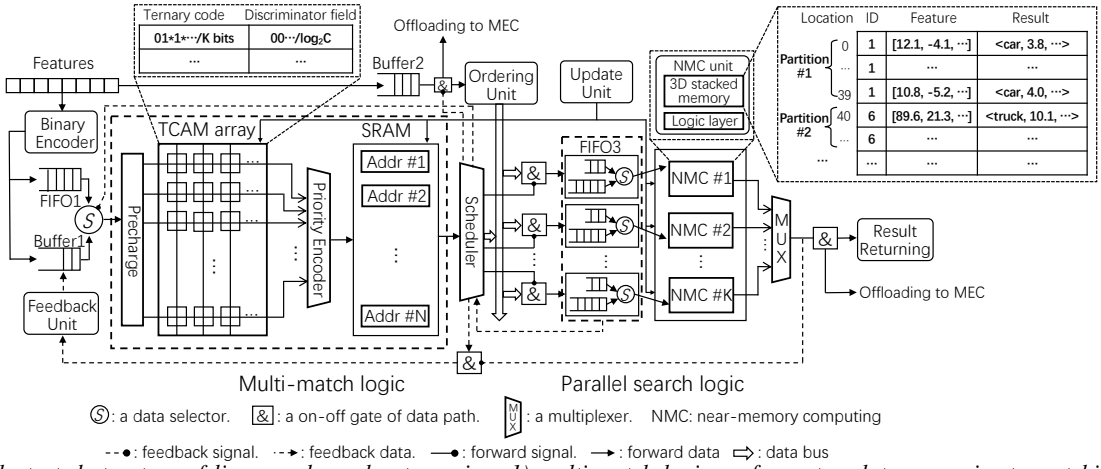
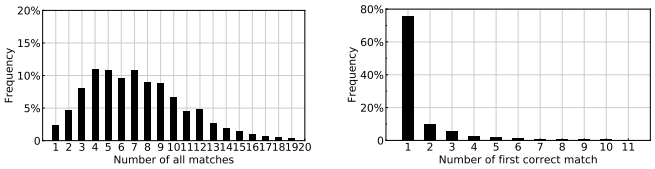
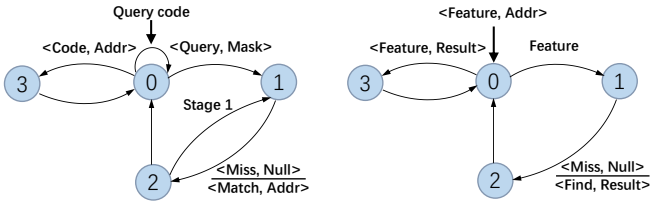


Fig. 7: *The illustrated structure of line-speed accelerate engine: 1) multi-match logic performs template approximate matching for query by conventional commercial TCAM with special design; 2) parallel search logic carries the further accurate search by near-memory computing.*



(a) All matches (b) First correct match

Fig. 8: The frequency of average umber of matches



(a) Multi-match logic (b) Parallel search logic

Fig. 9: The state transition diagram

multi-match logic and parallel search logic. For multi-match logic, the functions of 4 states are exhibited below:

- State-0 concatenates the query code either in FIFO (stage-1) or buffer (stage-2) with a postfix mask according to signal from feedback unit. If the update signal is active, state-0 transits to state-3 for update. Then unless the scheduler sends back a "blocking" signal indicating congestion in parallel logic, the state will transits to state-1. Otherwise, the state will remain state-0.
- State-1 performs the match in TCAM for concatenated query code, and the state will transits to state-2 directly.
- State-2 parses the output of TCAM by scheduler, and if TCAM matches a result, the scheduler will send the result to the corresponding NMC for searching. Then if the number of iterations is less than m, the scheduler will send the matching index back to feedback unit, and state-2 transits state-1 for next stage-1 matching on the same query, and otherwise, state-2 transits to state-1 to directly start new matching on new query.
- State-3 performs update on TCAM and transits to state-0.

And the states of parallel search logic are shown as:

- State-0 iteratively check the status of FIFO and NMC. If update signal is active, state-0 transits to state-3 for update. Then if the NMC is idle and FIFO is not empty, selector will deliver feature from short queue and long queue to NMC for searching, and state-0 transits state-1.
- State-1 does searching on NMC and transits to state-2.
- State-2 parses the output of NMC, and if a satisfied result is found, the result will be returned. Otherwise, a failure signal is fed back to start stage-2 matching in multi-match logic. State-2 transits to state-0.
- State-3 carries update on NMC and transits to state-0.

V. Evaluation

We conduct the function simulation to compare SODA with the state-of-the-art approaches, and then prototype the system on an FPGA platform to demonstrate its resource consumption and timing performance.

A. Function simulation

We first evaluate the encoding algorithm, TCAM matching module, and NMC searching module respectively, and then verify the performance of the integrated system.

1) General setup:

Dataset. We use KITTI 3D object detection benchmark [11] as the primary dataset. It contains 7,481 labeled training point clouds and 7,518 unlabeled test point clouds, covering three categories: Car, Pedestrian, and Cyclist. For each frame of point cloud data, we leverage pointnet [23] to further partition it into different independent segments (a common operation as in [8–10]), and extract 1,024-dimensional feature vectors, to obtain 73,408 object feature-result pairs in total. We divide these objects into different groups based on their similarity measured by IoU, and get 4,093 different similar groups (named *similar-group types*).

Parameter setting. We split the 73,408 labeled object dataset in the ratio of 7:3 into a train set and a validation set for encoder training. We set the LDA projection dimension m and the number of linear classifiers H to 300 and 12 respectively. For TCAM and NMC, we use the train set for database construction and the validation set for computation querying. Furthermore, we assume the query type of computation request

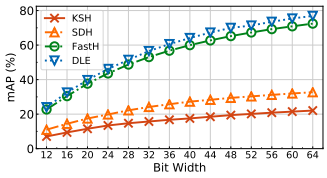


Fig. 10: mAP (%) on bit width.

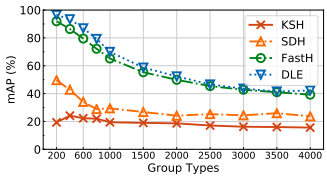


Fig. 11: mAP (%) on query types.

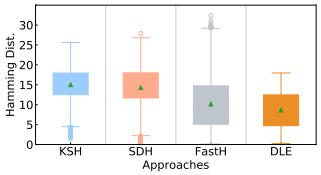


Fig. 12: Intra-group distance.

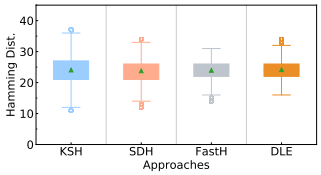


Fig. 13: Inter-group distance.

has a power-law distribution $P(x) = cx^{-\alpha}$, in which c and α are set to the empirical value 0.536 and 0.018 respectively.

2) Result analysis:

Encoder performance. We compare the performance of our Dual-Loss Encoding (DLE) algorithm with the state-of-the-art KSH [36], SDH [37], and FastH [38] in terms of mean average precision on hamming ranking (mAP), hamming distance, and encoding time. The average precision is defined as $AP = \frac{1}{n}\sum_{i=1}^{n}\frac{i}{rank_i}$, where n is the number of similar items in the results of top 100 shortest Hamming distance with a query feature, and $rank_i$ is the rank number of i-th similar item. The mAP is the mean AP among all queries.

Fig. 10 and Fig. 11 exhibit the comparison of mAP on varying code width and *similar-group types* respectively. In the bit width range in Fig. 10, while all approaches have an increasing mAP, DLE remains the best. DLE and FastH lead the other two by almost twice, and the gap keeps widening. The prominent performance of DLE corroborates the superiority of dual-loss supervision. In Fig. 11, all approaches however encounter an mAP decline as group types increase, showing that a fixed code width cannot sustain a growing number of types. Nevertheless, DLE still shows the best scalability.

Fig. 12 and Fig. 13 illustrate the distribution of intra-group and inter-group hamming distances, where DLE and FastH achieve lower intra-group hamming distance and higher inter-group distance than KSH and SDH, which is due to the two-step learning. However, the two-step learning adopting the independent classifiers for each bit binary code may incur a serious problem: it focuses on bit-level code fitting while ignoring the feature-level similarity, which leads to high variance on intra-group hamming distance, as shown in Fig. 12. DLE addresses this issue by taking feature similarity loss in each bit-classifier, making for a much lower variance.

TABLE II: Execute time for per-query feature encoding

Approaches	KSH	SDH	FastH	DLE
Time (us)	16	3.8	230	7.43

In addition to accuracy, Table II compares the per-query encoding time on an Intel Core i7-10710U CPU. DLE is close to SDH and much better than the decision tree-based FastH. In SODA, DLE will be implemented in hardware and we expect at least 10x speedup [49].

TCAM matching performance. Using the validation set as queries, we evaluate the matching performance of our Aggregated TCAM (A-TCAM) in terms of precision and recall, against Rene [30], Hierarchical Clustering tree (HC-tree) [50], and LSH [51].

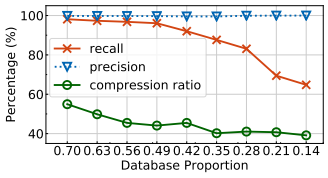


Fig. 14: Database proportion.

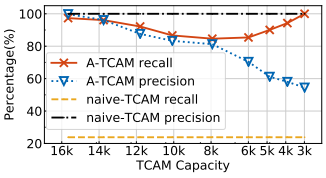
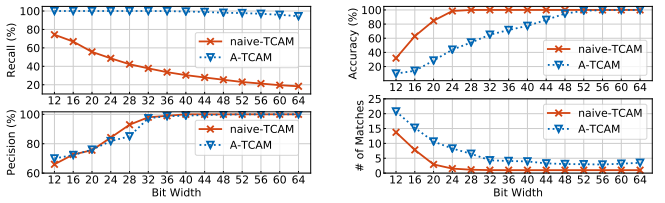


Fig. 15: Performance trade-off.

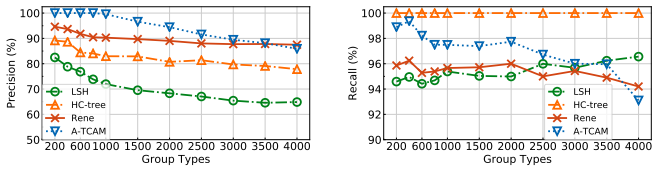
In Fig. 14, we reduce the proportion of the train set (for constructing the TCAM matching database) from 0.7 to 0.14 and verify the robustness of A-TCAM. As shown in the figure, both the recall and the compression ratio decrease with the reduction of database proportion while the precision keeps almost 100%. Especially, the recall has a slow decrease on a wide database range. This is mainly attributed to our gain-oriented greedy aggregation strategy which maintains the balance between TCAM matching range and storage occupation while minimizing the loss of precision. The precision degradation in Fig. 15 is because the decreasing TCAM capacity breaks the recall-precision balance. In Fig. 15, A-TCAM has a high precision comparable to naive-TCAM (with no aggregation) when TCAM depth is 9,000. However, as TCAM capacity reduces further, most similar codes are aggregated to one code, leading to a recall and precision loss. Fig. 16 exhibits the performance of A-TCAM on varying bit width. As shown in Fig. 16(a), the precision of both A-TCAM and naive-TCAM increases with bit width due to the improving encoding accuracy illustrated in Fig. 10. When the bit width is low, both naive-TCAM and A-TCAM have a high recall because most of the codes are the same. As bit width grows, the recall of naive-TCAM decreases while that of A-TCAM still holds as the improved encoding accuracy make aggregation work fine. This matches the line trend in Fig. 16(b), where A-TCAM and naive-TCAM both reveal a high number of matches and low top1-priority match accuracy when the bit width is low, but high matching accuracy when the bit width is high.

Fig. 17 presents the precision and recall of A-TCAM and the other state-of-the-art approaches with *similar-group types* varying from 200 to 4,000. In Fig. 17(a), the increasing *similar-group type* makes codes hard to differentiate and thus decreases all approaches' precision, while Rene shows the smallest precision loss. Its precision even exceeds A-TCAM when the number of *similar-group type* is greater than 3,500, because Rene adopts the accurate binary reflected Gray code (BRGC) for each floating-point value which makes codes easily differentiable, albeit suffering the curse of dimensionality (*e.g.*, a 1K-dimensional vector corresponds to a more than 12,000-bit code in Rene). As for recall, HC-tree searches the near point for a query along the hierarchical clustering tree and thus has a 100% recall but pays extra computation cost as well. A-TCAM has higher recall than the other two at the beginning but the recall decreases as the group types increases and eventually becomes lower than Rene and LSH. Since the limited 48 bit-code is hard to cover increasing content, A-TCAM needs to balance the trade-off between precision and recall. For matching speed, A-TCAM and Rene both can complete a query in a few clock cycles, while HC-tree



(a) Precision, recall (b) Top-1 accuracy, number of match

Fig. 16: The performance on varying bit width



(a) Precision. (b) Recall

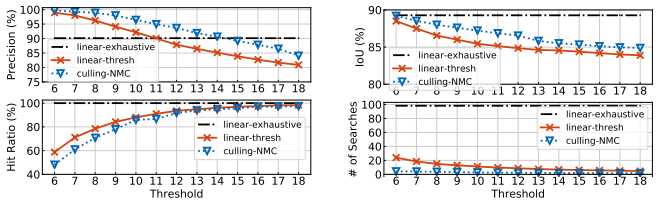
Fig. 17: The performance on varying *similar-group types*.

needs 10x more calculation for Euclidean distance on 1K-dimensional vectors, requiring thousands of clock cycles on a 64-bit CPU, let alone the huge data I/O time. As for LSH, the average bucket size mapped by a query is 84 which needs even longer calculation time. For storage overhead, A-TCAM only occupies 91KB TCAM storage while Rene needs 264MB, or 2,900x more storage than A-TCAM.

NMC searching performance. We evaluate the NMC searching performance and compare it with linear-exhaustive search (*linear-ex*), linear-thresholded search (*linear-th*), and the state-of-the-art method H-kNN [22]—*linear-ex* exhaustively searches the nearest result for a query among all results in a partition matched by TCAM, and *linear-th* sequentially searches a result until a result's distance with the query is less than a threshold—in the metrics of reuse precision, hit ratio, IoU, and search time (number of searches). Especially, to verify the effect of culling strategy, we add 10% fake feature-result pairs in the database that have close features with existing data but combining with completely wrong results.

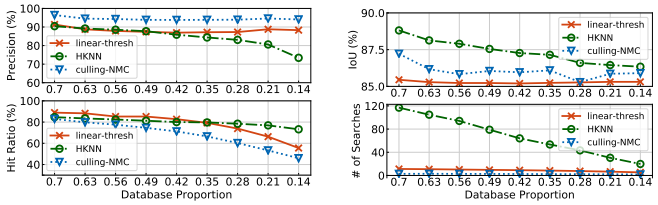
Fig. 18 exhibits the trade-off for precision, hit ratio, IoU, and search time on different distance thresholds. As illustrated in Fig. 18(a), for small thresholds, only the results with very close features to the query will be selected, so the precision is high while the hit ratio is low. As threshold increases, more results meet the distance condition and the hit ratio quickly increases, but meanwhile, some queries are wrongly matched by unrelated features which degrades the precision. *linear-th* preserves all stored results without culling and thus finds results easier while some results are wrong, making for a larger hit ratio than culling-NMC but a reduced precision. In contrast, culling-NMC keeps a high precision because the culling strategy prevents incorrect queries. This also conforms to the trend in Fig. 18(b) where culling-NMC has a much higher IoU and lower number of searches for a query. For *linear-ex*, the exhaustive searches cause plenty of errors especially due to the added fake data, As a result, *linear-ex* has a 100% hit ratio but much lower precision and IoU, and a long search time.

Fig. 19 compares culling-NMC with H-kNN and *linear-th* when the train set proportion is reduced from 0.7 to 0.14. As shown in Fig. 19(a), H-kNN has an almost constant hit



(a) Precision, recall (b) IoU, number of searches

Fig. 18: *NMC performance on distance threshold, where precision is the ratio of correct search results over all queries. A result is judged correct if its 3D bounding box IoU over ground-truth is greater than the threshold (70% for vehicles and 50% for pedestrians and cyclists).*



(a) Precision, recall (b) IoU, number of searches

Fig. 19: The performance comparison on varying database

ratio but a large decline in precision, while the other two are opposite and especially culling-NMC keeps the precision stable at over 90%. This is because H-kNN always chooses the major result from K nearest neighbors of the query without distance limitation, causing mismatches and precision loss. *linear-th* and culling-NMC select a result under distance threshold, so they can keep a relatively stable precision, but the decreasing train set reduces the satisfied results and the hit ratio. The problem of *linear-th* is that it has a large possibility to match a query to a wrong result with a close feature (*e.g.*, the added fake results). Culling-NMC selects a result with the constraint of both distance threshold and homogeneity level, resulting in a more stable and higher precision than the others. This explains why culling-NMC has higher IoU and shorter search time than *linear-th* in Fig. 19(b), where H-kNN achieves the highest IoU but with much longer search time.

TABLE III: System performance comparisons.

Approaches	precision	recall	IoU-0	IoU-1	Latency
SODA	94.60%	86.74%	88.68%	61.89%	$2.76{*}t_{calc}$
FoggyCache	75.68%	92.95%	85.96%	59.95%	$168.14{*}t_{calc}$
Detection Model	85.88%	83.23%	93.27%	65.09%	>50 ms

Overall system. First, we evaluate the overall system performance as the function of the number of matches m per query from TCAM. As exhibited in Fig. 20, the system has an increasing recall and keeps a high and stable precision with the growing number of matches. This is because the increasing number of matches improves the matching precision of TCAM and enlarges the probability of successful searches in NMC. Linking to the conclusion drawn from Fig. 18, this indicates that the levels of system precision, hit rate, and search period are tunable through the parameters m and the threshold of NMC. Correspondingly, we describe the trade-off between precision, hit rate, and search rounds (proportional to search latency) in Fig. 21, where improving any two metrics will degrade the performance of the third.

Next, we compare the system performance with foggycache

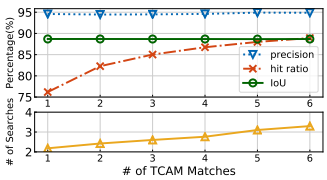


Fig. 20: Performance on number of TCAM matches.

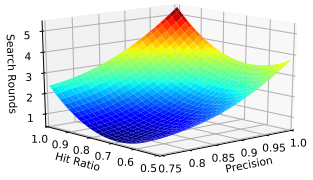


Fig. 21: Trade-off on overall system.

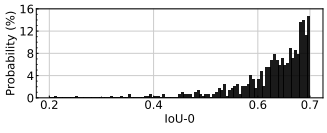


Fig. 22: Distribution of IoU-0.

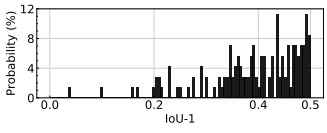


Fig. 23: Distribution of IoU-1.

(LSH+H-kNN) [22] and the ground-truth level of state-of-the-art 3D-object detection models [8–10] in Table III. While the IoU of SODA is lower than the state-of-the-art 3D object detection model (*e.g.*, PointRCNN [10]), SODA keeps the sub-microsecond level search latency (the time for calculating Euclidean distance on about 2.76 1024-d floating-point vectors per query plus one I/O latency) and higher precision. Moreover, culling-NMC only consumes 29.7 MB memory while FoggyCache needs five times more. Although the recall of SODA is lower than FoggyCache, FoggyCache suffers precision loss and long searching time. Furthermore, for the about 5% precision loss of SODA, the stats on all false-positive (*fP*) results show that no one mistakes a wrong object type (*e.g.*, recognize a pedestrian as a car), and all errors are derived from low IoU. The distribution of IoU-0 (for vehicles) and IoU-1 (for pedestrian and cyclist) of all *fP* results is shown in Fig. 22 and Fig. 23 respectively. IoU-0 of the most results is between 0.5 and 0.7, and IoU-1 between 0.2 and 0.5, meaning that even if *fP* occurs, the error is still in the acceptable safe range.

TABLE IV: FPGA resource consumption.

Module	LUTs	Registers	Block Ram	DSPs
TCAM	360, 071	441, 275	8	0
NMC	413, 065	370, 628	1112.5	5118
Control logic	3833	1373	14	0
Percentage	56.38%	28.9%	56.73%	44.47%

B. Prototype synthesis and timing analysis

To verify the system implementation feasibility and evaluate the throughput and latency performance, we implement SODA on a Xilinx Alveo U250 FPGA platform. The size of the on-chip TCAM is 8K×48b, and each entry's associated SRAM is 32b. The NMC memory contains 10K 1033×32b entries (each entry includes a 32b partition ID, a 256b result, and a 1024×32b feature vector). Two FIFOs with a depth of 10 are added between TCAM and NMC for asynchronous adaption. The data communication is carried by *AXI bus* [52].

We synthesize the design on Vivado [53]. The FPGA resource consumption is summarized in Table IV. Both TCAM and NMC need a large amount of LUTs and registers, which are used by TCAM for building the parallel matching memory, and by NMC for the calculation units. NMC also consumes Block Ram for data storage and DSPs for calculation.

Fig. 24 exhibits the core throughput and latency of SODA with different number of TCAM matches for each query, where the maximum core throughput is approaching 1.4M

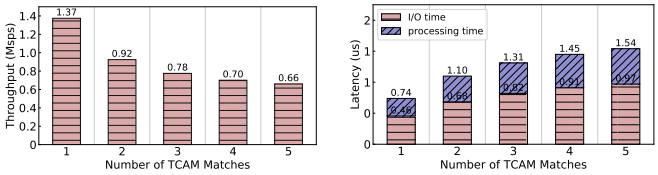


Fig. 24: Core throughput and latency on number of TCAM matches.

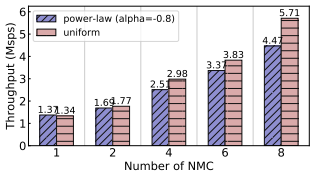


Fig. 25: Throughput on the parallel number of NMC.

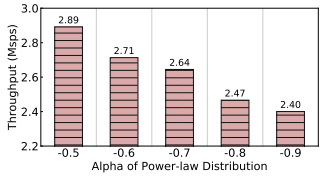


Fig. 26: Throughput on alpha of power-law distribution.

search per second (Msps) for a single TCAM match with the minimum latency of 0.74μs (I/O accounts for almost twice processing time). As the number of TCAM matches increases, the performance degrades but is still far beyond the system demand for assisting autonomous driving in high speed (*i.e.*, a vehicle needs a tail latency lower than 100ms at a frame rate of higher than 10 frames per second [2, 54]). Besides, to address the system bottleneck on NMC, we can deploy multiple modules to support parallel search where all results are uniformly distributed on different NMC modules. The result is illustrated in Fig. 25. When the query has a uniform distribution, the throughput improves linearly with the number of NMC modules; when the query follows a power-law distribution, the throughput improvement is sluggish. Specifically, the throughput is decreasing with α of the power-law distribution ranging from -0.5 to -0.9, as shown in Fig. 26, which indicates the throughput can be improved by parallel NMC if the results are well distributed (*e.g.*, adopt a well-designed load-balance algorithm [48]).

VI. Conclusion

SODA accelerates the MEC-assisted similar 3D object detection for autonomous driving. We designed efficient algorithms by leveraging TCAM's approximate matching capability and NMC's computing efficiency. The extensive evaluations confirmed the architecture feasibility and performance superiority on the subject matter.

REFERENCES

[1] Intel, "3d object detection evaluation," [EB/OL], https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving/#gs.dnjtcq. 2016.

[2] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 751–766.

[3] I. Mavromatis, A. Tassi, G. Rigazzi, R. J. Piechocki, and A. Nix, "Multi-radio 5g architecture for connected and autonomous vehicles: application and design insights," *arXiv preprint arXiv:1801.09510*, 2018.

[4] KITTI, "The coming flood of data in autonomous vehicles," [EB/OL], http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d/. 2017.

[5] Y. Matsubara and M. Levorato, "Neural compression and filtering for edge-assisted real-time object detection in challenged networks," *arXiv preprint arXiv:2007.15818*, 2020.

[6] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4490–4499.

[7] Y. Yan, Y. Mao, and B. Li, "Second: Sparsely embedded convolutional detection," *Sensors*, vol. 18, no. 10, p. 3337, 2018.

[8] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas, "Frustum pointnets for 3d object detection from rgb-d data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 918–927.

[9] Z. Wang and K. Jia, "Frustum convnet: Sliding frustums to aggregate local point-wise features for amodal 3d object detection," *arXiv preprint arXiv:1903.01864*, 2019.

[10] S. Shi, X. Wang, and H. Li, "Pointrcnn: 3d object proposal generation and detection from point cloud," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 770–779.

[11] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[12] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[13] S. Raza, S. Wang, M. Ahmed, and M. R. Anwar, "A survey on vehicular edge computing: architecture, applications, technical issues, and future directions," *Wireless Communications and Mobile Computing*, vol. 2019, 2019.

[14] F. Hu, D. Yang, and Y. Li, "Combined edge-and stixel-based object detection in 3d point cloud," *Sensors*, vol. 19, no. 20, p. 4423, 2019.

[15] J. Feng, Z. Liu, C. Wu, and Y. Ji, "Ave: Autonomous vehicular edge computing framework with aco-based scheduling," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 12, pp. 10 660–10 675, 2017.

[16] S. Lu, Y. Yao, and W. Shi, "Collaborative learning on the edges: A case study on connected vehicles," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[17] Q. Chen, S. Tang, Q. Yang, and S. Fu, "Cooper: Cooperative perception for connected autonomous vehicles based on 3d point clouds," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 514–524.

[18] Q. Chen, X. Ma, S. Tang, J. Guo, Q. Yang, and S. Fu, "F-cooper: feature based cooperative perception for autonomous vehicle edge computing system using 3d point clouds," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 88–100.

[19] Q. Zhang, Y. Wang, X. Zhang, L. Liu, X. Wu, W. Shi, and H. Zhong, "Openvdap: An open vehicular data analytics platform for cavs," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1310–1320.

[20] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*. IEEE, 2017, pp. 276–286.

[21] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 271–284.

[22] P. Guo, B. Hu, R. Li, and W. Hu, "Foggycache: Cross-device approximate computation reuse," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 19–34.

[23] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.

[24] M. Jiang, Y. Wu, T. Zhao, Z. Zhao, and C. Lu, "Pointsift: A sift-like network module for 3d point cloud semantic segmentation," *arXiv preprint arXiv:1807.00652*, 2018.

[25] V. Ravikumar and R. N. Mahapatra, "TCAM Architecture for IP Lookup Using Prefix Properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, 2004.

[26] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based Parallel Multi-Pipeline Architecture for Terabit IP Lookup," in *Proc. of IEEE INFOCOM*, 2008.

[27] P. Huang, R. Han, and J. Kang, "AI Learns How to Learn with TCAMs," *Nature Electronics*, vol. 2, no. 11, pp. 493–494, 2019.

[28] A. F. Laguna, X. Yin, D. Reis, M. Niemier, and X. S. Hu, "Ferroelectric FET Based In-Memory Computing for Few-Shot Learning," in *Proc. of Great Lakes Symposium on VLSI*, 2019.

[29] R. Shinde, A. Goel, P. Gupta, and D. Dutta, "Similarity Search and Locality Sensitive Hashing Using Ternary Content Addressable Memories," in *Proc. of ACM SIGMOD*, 2010.

[30] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Hel-Or, "Ultra-fast similarity search using ternary content addressable memory," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, 2015, pp. 1–10.

[31] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 113–124.

[32] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal, "Napel: Near-memory computing application performance prediction via ensemble learning," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[33] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[34] H. M. C. Consortium *et al.*, "Hybrid memory cube specification 2.1," *hybridmemorycube. org*, 2014.

[35] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[36] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang, "Supervised hashing with kernels," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012, pp. 2074–2081.

[37] F. Shen, C. Shen, W. Liu, and H. Tao Shen, "Supervised discrete hashing," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 37–45.

[38] G. Lin, C. Shen, Q. Shi, A. Van den Hengel, and D. Suter, "Fast supervised hashing with decision trees for high-dimensional data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1963–1970.

[39] G. Lin, C. Shen, D. Suter, and A. Van Den Hengel, "A general two-step approach to learning-based hashing," in *Proceedings of the IEEE international conference on computer vision*, 2013, pp. 2552–2559.

[40] G. Lin, C. Shen, and A. Van den Hengel, "Supervised hashing using graph cuts and boosted decision trees," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 11, pp. 2317–2331, 2015.

[41] M. Rastegari, A. Farhadi, and D. Forsyth, "Attribute discovery via predictable discriminative binary codes," in *European Conference on Computer Vision*. Springer, 2012, pp. 876–889.

[42] S. Balakrishnama and A. Ganapathiraju, "Linear discriminant analysis-a brief tutorial," *Institute for Signal and information Processing*, vol. 18, pp. 1–8, 1998.

[43] R. McGeer and P. Yalagandula, "Minimizing rulesets for tcam implementation," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 1314–1322.

[44] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 193–204, 2005.

[45] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[46] S. Khoram, Y. Zha, J. Zhang, and J. Li, "Challenges and opportunities: From near-memory computing to in-memory computing," in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, 2017, pp. 43–46.

[47] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[48] K. Zheng, C. Hu, H. Lu, and B. Liu, "A tcam-based distributed parallel ip lookup scheme and performance analysis," *IEEE/ACM Transactions on networking*, vol. 14, no. 4, pp. 863–875, 2006.

[49] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[50] M. Muja and D. G. Lowe, "Fast matching of binary features," in *2012 Ninth conference on computer and robot vision*. IEEE, 2012, pp. 404–410.

[51] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*. IEEE, 2006, pp. 459–468.

[52] F.-m. Xiao, D.-s. Li, G.-m. Du, Y.-k. Song, D.-l. Zhang, and M.-l. Gao, "Design of axi bus based mpsoc on fpga," in *2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*. IEEE, 2009, pp. 560–564.

[53] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[54] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.