VTT Technical Research Centre of Finland

Optimizing 3D object detection for embedded systems in automated vehicles using sensor data fusion and CUDA computing

Miekkala, Topi; Kutila, Matti; Schneider, Mathias; Höß, Alfred

Published in:
2022 IEEE 18th International Conference on Intelligent Computer Communication and Processing (ICCP)

Accepted/In press: 22/09/2022

Document Version
Peer reviewed version

Optimizing 3D object detection for embedded systems in automated vehicles using sensor data fusion and CUDA computing

Topi Miekkala
VTT Techincal Research Centre of Finland Ltd.
Tampere, Finland
topi.miekkala@vtt.fi

Matti Kutila
VTT Techincal Research Centre of Finland Ltd.
Tampere, Finland
matti.kutila@vtt.fi

Mathias Schneider
Technical University of Applied Sciences
Amberg-Weiden, Germany
mat.schneider@oth-aw.de

Alfred Höß
Technical University of Applied Sciences
Amberg-Weiden, Germany
a.hoess@oth-aw.de

Abstract— **This article explores the utilization of the processing power of GPUs using CUDA computation for real-time aggregation of multi-sensor data and detection of 3D objects using parallel clustering algorithms. The purpose is to implement an algorithm that fuses raw lidar point cloud data and 2D camera image object detections to produce 3D object clusters in a lidar point cloud. Most of the computation has been implemented using CUDA parallelism to investigate the capability of GPU devices in this task, which is a common challenge in automated driving. The results indicate that processing times can be optimized within the algorithm, which is crucial when considering the large amounts of data provided by lidar and camera-based systems. The algorithm can perform inference on the Jetson Xavier AGX at rates of ~20 to ~220 ms depending on the number of objects and their corresponding point amounts in the KITTI dataset.**

Keywords—3D object detection, CUDA, sensor data fusion, automated driving

I. INTRODUCTION

A. Parallel Computation

With the constant increase in processing power of modern computation devices, engineers can design and implement increasingly complex and computationally demanding algorithms, which is an especially significant subject in the field of automated vehicles, where real-time processing of data is crucial. For this reason, the European AI4DI project is adopting a holistic view of developing generic artificial intelligence algorithms that are then optimized in a different industrial domain. In this case, we are focusing on developing AI for the benefits of the transport industry. In demanding computation tasks, GPU devices are often utilized due to them enabling efficient parallelization of repetitive calculations, which can significantly speed up data processing. Neural networks are an example of the effective utilization of parallel computing due to their repetitive calculations, which are exponentially sped up by distributing the operations across the GPU threads. Although neural networks are currently a very popular field of science, there are many other possibilities for the efficient utilization of the parallel computation power of GPUs. In automated driving, the real-time computation of vehicle sensor data is a critical safety requirement. With continuously improving sensor technology, the amount of data that needs to be processed by an automated vehicle is also increasing constantly. This demands more computation power and solutions that allow quick processing of high-resolution camera images and large lidar point clouds, for example.

Edge AI computing is an increasingly important part of automated driving due to the growing versatility of deep learning applications for vehicle sensor data. These embedded systems often combine compactness and power efficiency and maximize GPU capacity for optimized running of AI applications. These devices come in several variations of computation power and intended use cases. One popular example in the field of robotics is the Nvidia Jetson product line [1]. Jetson devices offer a range of developer tools to optimize and deploy AI applications in demanding industrial environments.

B. 3D Object Detection in Automated Driving

The task of 3D object detection is a challenging and critical subject in automated driving. The environmental perception of an autonomous vehicle must be reliable to enable safe automated vehicle operation, especially in crowded and narrow urban spaces where other road users can blend into the surroundings, and sometimes appear suddenly in front of the vehicle from behind obstacles. The vehicle should be able to detect other road users robustly, but also do it in real-time to react quickly enough. A 3D object detector neural network might be able to correctly detect and classify objects from the near vicinity of the vehicle, but if the neural network model is so computationally expensive that the vehicle computer takes several hundreds of milliseconds to process the data, the chance of collision and a serious accident increases greatly.

Data fusion can be implemented to process the output of multiple sensor types to extract information that would not be available from simply using a single sensor. In automated driving, there are varying targets on the road that have multiple characteristics and features. Some of these features are more perceivable to specific sensor types such as cameras, while other features of the same object are perceived better by a different sensor such as a lidar. State-of-the-art 2D image-based object detectors such as the Yolov4 network [2] have reached remarkable performance levels, and they are useful tools in autonomous driving tasks that make use of camera data. 2D object detection often tends to be less computationally expensive than 3D object detection. Fusing the data of a lidar sensor and a 2D image object detector can provide the benefits of execution speed and accuracy of 2D detection and combine that directly with 3D spatial information provided by a lidar, without implementing deep learning point cloud detectors.

The following sections II and III present the basis and implementation for an example method that utilizes 2D image object detections and lidar point clouds, fusing their data together through CUDA (Compute Unified Device Architecture) accelerated algorithms [3]. Sections IV and V present the results that demonstrate the efficiency of applying CUDA operations on common point cloud and

image operations such as coordinate transformations, point projection and distance-based clustering in order to achieve 3D object clusters with fused camera and lidar data.

II. Related Work

A. Current Methods for Object Detection

Neural networks have gained great momentum in the past decade due to the emergence of big data and increases in computation power. The current era of deep learning advancements has provided several capable object detection models, with Yolov4 being a state-of-the-art example of a 2D image detector. It achieves 43.7% average precision (AP) and 65.7% AP50 accuracies for the COCO dataset [2].

For 3D object detection, several deep learning methods exist [4]. A detector that uses a single point cloud as an input usually either performs inference directly on the individual points, voxelization on the scene first, or uses a combination of the two. Popular 3D detectors such as PV-RCNN and Point-RCNN by S. Shi et al. [5][6] report their test results using high-end GPUs like the Titan RTX and the Tesla V100. In actual applications of autonomous driving, the computing is often performed using an embedded GPU, which most likely does not match the performance of high-end desktop GPUs. Another well-known 3D detector, PointPillars [7], addresses this fact in its test results. PointPillars is also one of the computationally lighter 3D detectors. Nvidia has released a TensorRT optimized model of PointPillars, which is claimed to perform at an inference rate of approximately 27.5 milliseconds, translating to approximately 36 FPS [8].

B. Object Detection using Embedded AI devices

Embedded AI devices such as the Nvidia Jetson product line do not necessarily match higher-end desktop GPUs in terms of computing capacity and CUDA cores, but their advantage regarding deep learning lies in the device-specific optimization of neural network models through libraries like TensorRT [9]. The Nvidia TensorRT library offers tools to modify existing and trained neural network models in ways that speed up calculations on the GPU device and depending on the network architecture can greatly accelerate neural network inference. While the Yolov4 network is a relatively computationally heavy 2D object detector, TensorRT can speed up its inference significantly. One such software library is the TkDNN library [10]. The TkDNN authors claim that the optimized Yolov4 runs at approximately 22 FPS on the Jetson Xavier AGX with a network resolution of 608x608.

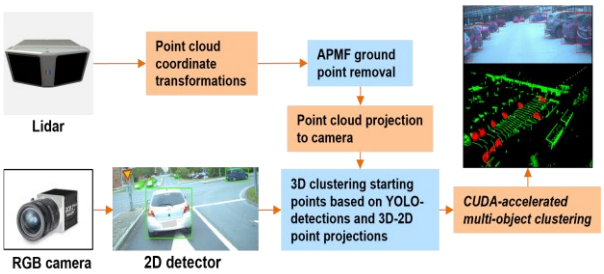

Fig. 1. Full detection system using a camera and a lidar sensor. the orange phases use CUDA accelerated calculation.

III. 3D Object Clustering Method

The algorithm used in this study uses a lidar point cloud and its corresponding 2D image object detection boxes as an input. The 2D detection boxes can be the result of any kind of detection algorithm, but in this study we used the Yolov4 detector accelerated by the TensorRT library. Another requirement for the data fusion between the camera and the lidar is the calibration matrix between the two sensors. This can be obtained in various ways, but for this study, we used the genetic algorithm of [11], which is implemented specifically for this purpose. The object clustering algorithm itself consists of two major phases: pre-processing and parallel multi-object 3D clustering. A diagram of the full system is shown in Fig. 1.

A. Pre-processing

Firstly, there is pre-processing, which includes CUDA accelerated combination of the 3D lidar point cloud data and the 2D image object detection data. The pre-processing phase first applies an approximate progressive morphological filter (APMF) to remove ground points from the lidar point cloud. The APMF is also the only phase that does not make use of CUDA kernels, but instead uses the implementation provided by the Point Cloud Library (PCL) [12][13]. After removing ground points, CUDA calculations are used to apply the appropriate coordinate transformations to transform the lidar point cloud to the coordinate system of the camera, and then to project the points on the 2D image. After this, projected points that fall inside proportionally shrunken 2D image detections boxes (magenta boxes in Fig. 5 and Fig. 6) are used to determine the starting points for the 3D object clustering in the lidar point cloud.

B. Parallel Multi-Object 3D Clustering

From the pre-processing phase the algorithm has acquired the 3D starting points for the clustering in the original lidar point cloud. The multi-object clustering iteration is presented in Fig. 2. The clustering works by checking each point in the original lidar point cloud. If a point has been marked to be part of an object cluster, a

Algorithm 1 CUDA multi-object clustering

Input: α, P, δ
Output: C

```
foundNew = false
iter = 0
while iter < δ do
  for p in P do
    if p in C then
      FindNeighborsCUDA(α, p, P, C, foundNew)
    end if
  end for
  if (not foundNew) then
    return C
  end if
  iter ← iter + 1
end while
return C
```

Fig. 2: Pseudocode of the multi-object clustering iterations. α defines the neighbour search box size, *P* is the lidar point cloud with ground points filtered out, δ is the limit of iterations, and *C* is a point cloud of object clusters.

CUDA kernel-based distance search is performed for all other lidar points. If any of the other lidar points are closer to the cluster point than a specific threshold α, then those points are tagged as part of the same object cluster. More precisely, a single iteration checks all lidar points in the observed cloud and launches CUDA distance searches at every point which belongs to an object cluster. The operations inside a single CUDA thread are shown in Fig. 3. Instead of clustering each object one by one, the objects are clustered simultaneously, with more lidar points added to each cluster every cycle. This cycle is repeated until either a pre-determined iteration limit δ is reached, or there are no more points added to any of the object clusters during an iteration. To make calculations faster the z-axis is ignored, and the distance calculation inside the CUDA kernel only considers the x and the y coordinates of points, as the ground points have previously been filtered out. The distance check is done using conditional checks of points residing inside a square search area. This was more computationally efficient than Euclidean or Manhattan distance calculations in large quantities. The algorithm outputs 3D point clusters representing the objects of interest, which were originally given by the 2D image detector, which in this case was Yolov4. The neighbour search inside the CUDA kernels is visualized in Fig. 4.

Algorithm 2 FindNeighborsCUDA (single CUDA thread)

Input: α, p, P, C, $foundNew$
Output: P, C, $foundNew$

$cudaThreadIdx$ = assigned in kernel launch
$c = P[cudaThreadIdx]$
if c already in a cluster **then**
  return
end if
if $c.x < p.x + \alpha$ and $c.x > p.x - \alpha$ **then**
  if $c.y < p.y + \alpha$ and $c.y > p.y - \alpha$ **then**
    $C[cudaThreadIdx] = c$
    $markAsClusterPoint(P[cudaThreadIdx])$
    $foundNew = true$
  end if
end if

Fig. 3: Pseudocode of distance search in a single CUDA thread. A single thread handles a single point from cloud P by checking if it is already a part of any cluster. If not, a distance check based on α is performed, and point c is handled accordingly.

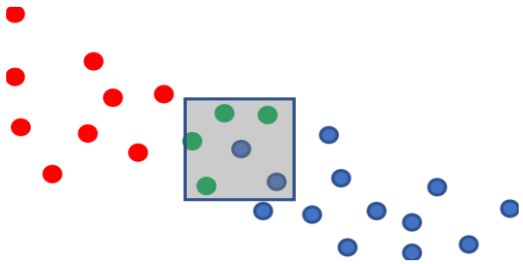

Fig. 4: Example of a box neighbour search in the CUDA multi-object clustering algorithm. The size of the grey search box depends on α in Fig. 1, blue points are already included in a cluster, red points are not part of any cluster, and green points are about to be included in the blue cluster.

IV. Test Arrangements

A. Datasets used in the evaluation

To gain an understanding of the computational performance of the 3D clustering, the algorithm was applied to two separate datasets, and various information about the calculation times was collected. The KITTI dataset was used to place the measurements into the context of a popular public dataset. The KITTI 3D object detection dataset contains lidar measurements from a Velodyne HDL-64E sensor [14]. Due to the design of the algorithm, the amount of lidar points yielded by the sensor has the most significant impact on the performance times data-wise, and a custom dataset was also collected using a higher resolution lidar that provided more 3D points to process. Examples of the 3D object clustering are shown in Fig. 5 and Fig. 6.

The custom dataset was collected using a Luminar Hydra sensor and a Basler Ace colour camera that were installed on top of an autonomous research vehicle

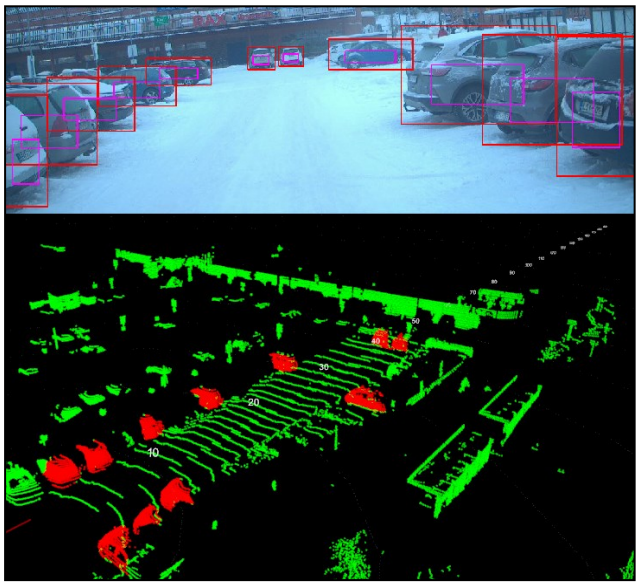

Fig. 5: Example of the 3D clustering on the custom dataset.

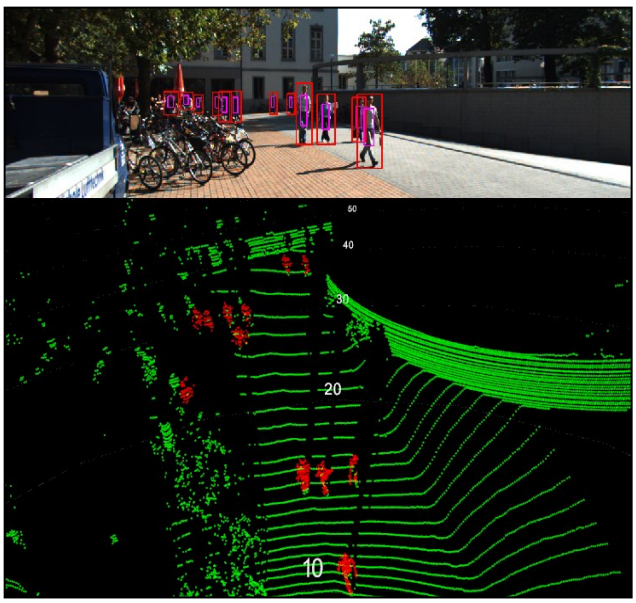

Fig. 6: Example of the 3D clustering on the KITTI dataset.

[15][16]. The custom dataset was collected in winter conditions and was used to obtain computation performance data from a real-world implementation for reference.

B. Methods of performance evaluation

The 3D clustering performance was measured by collecting various details from every measurement frame. In this context, a measurement frame refers to a camera image and its corresponding point cloud. In KITTI this included the 7841 training images and their point clouds, and in the custom dataset there were 7804 similarly corresponding images that were processed.

The performance was evaluated with regard to the number of objects that the algorithm attempted to cluster in the image, and the number of iterations that the multi-object clustering in Fig 2. executed. The processing time was collected for all numbers of iterations and object amounts, and the mean values were calculated to represent the average processing time for different situations. In addition to this, the processing times were evaluated with regard to the total number of points clustered in the image over all the objects. The processing times were further divided to separate the pre-processing durations, and the multi-object clustering durations. The algorithm was tested on both datasets on a Jetson Xavier AGX and an RTX 2070 Super GPU. A CPU version of the algorithm was also implemented for reference and evaluated using an Intel i7 10750H processor. The algorithm is not suitable for a 3D bounding box formulation in a similar way to how a neural network might be, since the algorithm in this study relies on distance-based searches and does not consider the possible shapes of the objects. As a result, KITTI accuracies are not included in this study. However, to gain some reference, a test was conducted where each obtained cluster was matched to a KITTI ground truth 3D box by

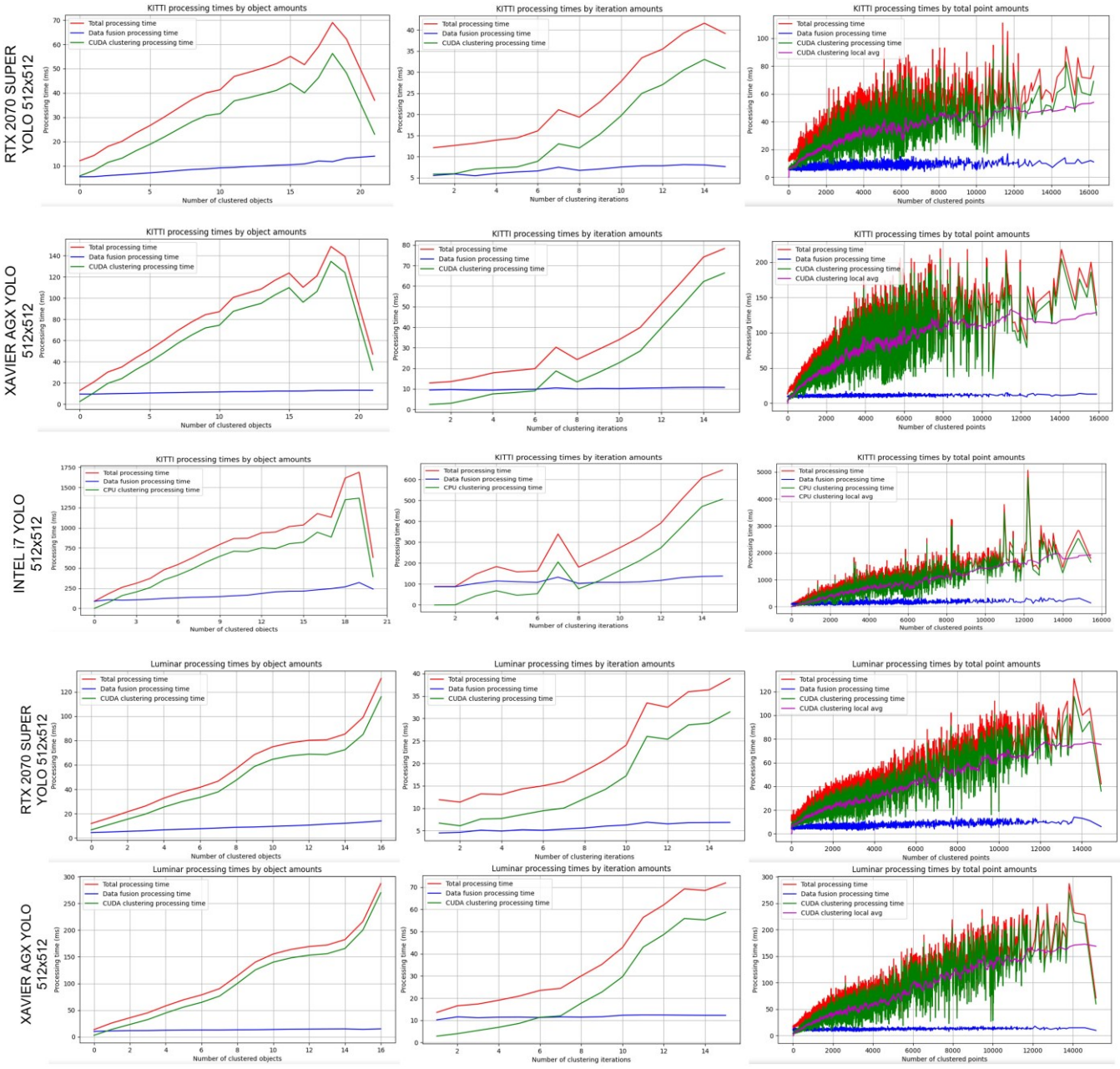

Fig. 7: Processing time performance results on both datasets. The first graph column is average processing time by detected object amount. The second is average processing time by algorithm iteration amount. The third is processing times by clustered point amount.

calculating the mean of the 3D cluster and checking whether the mean resides inside a KITTI ground truth box.

V. RESULTS

Using the KITTI medium difficulty and only considering the car and pedestrian classes, the above ground truth test provided a 90.1 % accuracy for cars and 82.6 % accuracy for pedestrian, meaning that most of the clusters acquired in the KITTI dataset were relevant for the processing performance evaluation. The results are presented in Fig. 7. From Fig. 2, the xy-coordinate search range parameter α was set to 0.2 meters for the pedestrian class, and 0.3 meters for the car class. Pedestrians often huddle together more than vehicles, which is why α was set to a lower value for pedestrians. A smaller α reduces the chance falsely clustering lidar points which do not actually belong to the object, but to the object next to it. The iteration limit δ was set to 15 for vehicles, and to 5 for pedestrians. The iteration limit is set to enable full clustering of an object based on α. If α is set to 0.3 meters, in 15 iterations the clustering could select points from as far as 4.5 meters from the starting point. This is a sufficient range to cluster most passenger cars. Similarly, the α and δ of pedestrians result in a maximum clustering range of 1 meter. The processing times of Fig. 7 were obtained using Yolov4 with an input size of 512x512 and half-precision floating point calculations.

The inference times of Yolo are not included in the graphs as they remained constant. Average inference times for various Yolo input sizes in the KITTI training data are shown in Tab. I. The CPU tests were conducted using multithreading to maximize processor usage.

TABLE I. YOLOV4 AVERAGE KITTI INFERENCE TIMES IN TKDNN

	512x512	**608x608**	**800x800**
Xavier AGX	38.9 ms	52.3 ms	85.0 ms
RTX 2070 Super	11.8 ms	16.0 ms	23.6 ms

Fig. 8 is shown as a reference for the difference in lidar point amounts between the KITTI dataset and the custom dataset. The most important limiting factor in the processing is the GPU capacity, including the number of CUDA cores available for parallel computing. The RTX 2070 Super has 2560 of these, while the Xavier AGX only has 512. This means that as the number of objects being clustered and the amount of point comparisons increases, there is more bottlenecking as the CUDA functions have to wait for device resources to be freed. In the KITTI dataset there is a large drop in processing times when the object amounts are the highest. This goes against intuition, and upon closer inspection it is explained by the KITTI dataset containing only a handful of frames where there are ~20 objects detected in these measurements, and in these specific frames the objects were mostly pedestrians, which also meant that the algorithm was able to cluster the objects quicker, which on the other hand distorts the graph as is seen in the top-left graph and the second top-left graph in Fig. 7. The other graphs of processing times with regard to the number of objects and the number of iterations show a rather consistent correlation. **The processing times of the RTX 2070 Super are on average approximately twice as fast as those of the Xavier AGX on both datasets**. The performance of the CPU is many times slower. The third column shows a very uneven development of the processing times with the total amount of points in a single frame. This is expected behaviour, because the numbers of objects and iterations have the biggest impact on how many CUDA kernels and point comparisons are executed during the pass of the algorithm. Since the point distance comparisons are done in parallel, the number of points obtained during a single algorithm iteration does not affect the computing times in the same way as in the first two columns of graphs. There is still a general increase in the times with increasing total point amounts, which is better observed by plotting the local average with the ten previous values. The single slowest inference performance of the Xavier AGX was **219 ms on the KITTI data, and 287 ms on the denser Luminar data**. On the other hand, we see from the graphs that the majority of the inferences perform at a clearly faster rate.

VI. CONCLUSIONS AND FUTURE WORK

Optimizing artificial intelligence algorithms is crucial, since automated vehicles are dealing with high resolution lidars and cameras onboard. The amount of data has been increasing more than available computation power, which is partly due to the limited amount of power available in electric vehicles. Therefore, the performance and optimization of the algorithms remains an unsolved problem, even though high-power computation units are available today.

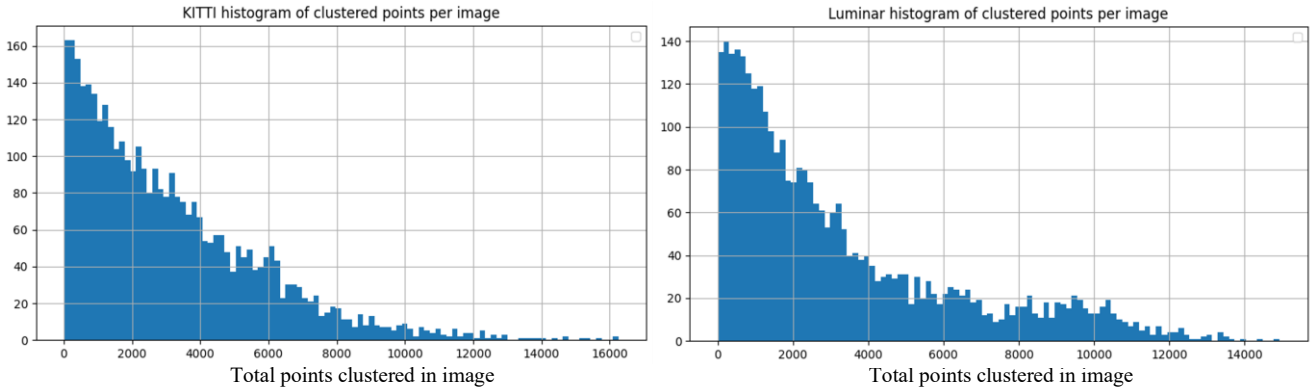

Fig. 8: Histograms of the clustered lidar point amounts for both datasets using the 512x512 Yolov4 network as a basis.

We presented a novel approach for a GPU-accelerated 3D object detection pipeline. In this section, we would like to raise some prospective considerations to deploy this pipeline efficiently in a wide range of vehicles. These comprise thoughts on the deployment at the deep edge, as well as the benefit of heterogeneous accelerator architectures, potentially enabling task offloading. However, when introducing these architectural changes, it is crucial to implement metrics quantifying the impact on functional, such as the accuracy, as well as non-functional indicators, such as latency and power consumption. Based on these metrics, possible trade-offs can be evaluated for different setups. Some concluding directions are provided in the following paragraphs.

A. Edge GPU device selection

For power-efficient on-board computation, the selection of a suitable device is decisive. Accordingly, we compared the performance of an RTX 2070 Super GPU with the edge GPU device Jetson Xavier AGX, which is a more realistic option for automated vehicle deployment. Although the edge platform is capable of running the proposed algorithms sufficiently, the edge platform performs increasingly worse during peak loads with a higher number of objects. To alleviate this disadvantage, Nvidia's next generation edge device Jetson AGX Orin is a promising candidate for high-end edge computing. Based on the latest Ampere GPU architecture, it is equipped with 2048 CUDA cores and 64 Tensor Cores [17], resembling the specification of the RTX 2070 Super GPU (2560 CUDA cores, 320 Tensor Cores).

B. Co-processors for 2D object detection inference

As shown in section *V,* the amount of available GPU resources, such as the number of available CUDA cores, significantly impacts the performance of the presented 3D object clustering algorithm. Accordingly, offloading the mandatory 2D object detection models from the GPU to another processor could be beneficial. Heading for a more heterogeneous partitioning architecture, low-end co-processors, such as Coral Edge TPU or Intel's Visual Processing Unit (VPU), could be employed to reduce the load on the GPU. However, these accelerators are more limited due to less memory and computation units, which consequently requires the use of more lightweight object detection architectures, e.g. the TinyYolov4 or Yolov5 Nano [18] models, which must be deployed to the updated target platform. Due to our inherent component-based architecture of the overall pipeline, this partitioning does not imply direct changes for the 3D clustering algorithm, but still requires the evaluation of end-to-end performance metrics.

C. Hardware invariant 3D object detection clustering

While the current version of the algorithm benefits from the parallelism of the implementation CUDA, other frameworks for accelerated parallel computation could be leveraged. High-level APIs, capable of abstracting underlying hardware architecture, could be used to increase the portability of the component. Depending on the level of abstraction, the algorithm could be implemented with the Vulkan API [19], adding low overheads for deployment on supported CPUs and GPUs. Alternatively, it could be investigated whether the algorithm can be written in common machine learning frameworks such as TensorFlow using operators for tensor computation. Afterwards, the graph is exported to a standardized representation such as ONNX and is finally inferred on the target device using the ONNX runtime with an arbitrary supported engine provider (e.g. OpenVINO or TensorRT), which manages the accelerated computation.

REFERENCES

[1] Nvidia Corporation, "Embedded Systems with Jetson", [online] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/, accessed: 2022-01-31.

[2] A. Bochkovskiy, C.-Y. Wang, H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy for Object Detection", Apr. 2020.

[3] Nvidia Corporation, "CUDA Toolkit", [online] https://developer.nvidia.com/cuda-toolkit, accessed: 2022-01-31.

[4] R. Qian, X. Lai, X. Li, "3D Object Detection for Autonomous Driving: A Survey", Jun. 2021.

[5] S. Shi, C. Guo, L. Jiang, Z. Wang, J. Shi, X. Wang, H. Li, "PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection", Dec. 2019.

[6] S. Shi, X. Wang, H. Li, "PointRCNN: 3D Object Proposal Generation and Detection from Point Cloud", Dec. 2018.

[7] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, O. Beijbom, "PointPillars: Fast Encoders for Object Detection from Point Clouds", De. 2018.

[8] Nvidia Corporation. "PointPillars inference with TensorRT", [online] https://github.com/NVIDIA-AI-IOT/CUDA-PointPillars, accessed: 2022-02-01

[9] Nvidia Corporation, "Nvidia TensorRT", [online] https://developer.nvidia.com/tensorrt, accessed: 2022-01-31.

[10] M. Verucchi, G. Brilli, D. Sapienza, M. Verasani, M. Arena, F. Gatti, A. Capotondi, R. Cavicchioli, M. Bertogna, M. Solieri, "A Systematic Assessment of Embedded Neural Networks for Object Detection", *25th IEEE International Conference on Emerging Technologies and Factory Automation,* vol 1, pp. 937-944, 2020.

[11] Y. Liu, "WPI LiDAR – Camera calibration toolbox", [online] https://github.com/YechengLyu/WPI-LiDAR-Camera-Calibration-Toolbox, accessed: 2022-01-31.

[12] K. Zhang, S.-C. Chen, D. Whitman, M.-L. Shyu, J. Yan, C. Zhang, "A Progressive Morphological Filter for Removing Nonground Measurements From Airborne LIDAR Data", *IEEE Trans. Geoscience and Remote Sensing*, vol. 41, no. 4, pp. 872–882, Apr. 2003.

[13] R. B. Rusu, S. Cousins, "3D is Here: Point Cloud Library (PCL)", *IEEE International Conference on Robotics Automation,* May 2011.

[14] A. Geiger, P. Lenz, C. Stiller, R. Urtasun, "Vision meets Robotics: The KITTI Dataset", *The International Journal of Robotics Research,* Vol. 32, Issue 11, pp. 1231-1237.

[15] Level Five Supplies, Luminar Hydra Specifications, [online] https://levelfivesupplies.com/wp-content/uploads/2020/08/Luminar-Hydra-Datasheet.pdf, accessed: 2022-01-31.

[16] Basler a2A2590-60ucBAS Specifications, [online] https://www.baslerweb.com/en/products/cameras/area-scan-cameras/ace2/a2a2590-60ucbas/, accessed: 2022-01-31.

[17] NVIDIA Corporation, "NVIDIA Jetson AGX Orin – Data Sheet", Version DS-10662-001_v0.2, 2021, [online] https://developer.nvidia.com/embedded/secure/jetson/agx_orin/jetson_agx_orin_ds-10662-001_v0.2.pdf

[18] Glenn Jocher et al. (2021). ultralytics/yolov5: v6.0 - YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support (v6.0). Zenodo. https://doi.org/10.5281/zenodo.5563715

[19] The Khronos Vulkan Working Group, "Vulkan 1.3.204 - A Specification", Version 1.3.204, 2022, [online] https://www.khronos.org/registry/vulkan/specs/1.3/pdf/vkspec.pdf