Efficient Computing Platform Design for Autonomous Driving Systems

Shuang Liang[*§], Xuefei Ning[*], Jincheng Yu[*], Kaiyuan Guo[*], Tianyi Lu[§], Changcheng Tang[§], Shulin Zeng[*], Yu Wang[*], Diange Yang[†], Huazhong Yang[*]

Department of Electronic Engineering, Tsinghua University[*] Novauto Co., Ltd.[§]
School of Vehicle and Mobility, Tsinghua University[†]

{s-liang,yu-wang}@tsinghua.edu.cn

ABSTRACT

Autonomous driving is becoming a hot topic in both academic and industrial communities. Traditional algorithms can hardly achieve the complex tasks and meet the high safety criteria. Recent research on deep learning shows significant performance improvement over traditional algorithms and is believed to be a strong candidate in autonomous driving system. Despite the attractive performance, deep learning does not solve the problem totally. The application scenario requires that an autonomous driving system must work in real-time to keep safety. But the high computation complexity of neural network model, together with complicated pre-process and post-process, brings great challenges. System designers need to do dedicated optimizations to make a practical computing platform for autonomous driving. In this paper, we introduce our work on efficient computing platform design for autonomous driving systems. In the software level, we introduce neural network compression and hardware-aware architecture search to reduce the workload. In the hardware level, we propose customized hardware accelerators for pre- and post-process of deep learning algorithms. Finally, we introduce the hardware platform design, NOVA-30, and our on-vehicle evaluation project.

CCS CONCEPTS

• **Computer systems organization → Embedded hardware**; • **Hardware → Hardware accelerators**; • **Computing methodologies → Search methodologies**.

KEYWORDS

autonomous driving, computing platform, neural networks, hardware accelerators

ACM Reference Format:
Shuang Liang[*§], Xuefei Ning[*], Jincheng Yu[*], Kaiyuan Guo[*], Tianyi Lu[§], Changcheng Tang[§], Shulin Zeng[*], Yu Wang[*], Diange Yang[†], Huazhong Yang[*]. 2021. Efficient Computing Platform Design for Autonomous Driving Systems. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3394885.3431620

1 INTRODUCTION

In recent years, both traditional vehicle manufacturers and their vendors are spending more and more efforts on autonomous driving. With the rapid development in algorithm and hardware platforms, the road towards the ultimate goal, fully autonomous driving, is becoming more and more clear.

But an autonomous driving system is still by no means simple to any of the companies in this region. An autonomous system usually consists of four parts: perception, localization, planning, and control. The input of the system includes cameras, LiDARs, radars, etc. To achieve these hard tasks with hybrid and high-dimensional inputs, autonomous driving developers are faced with great challenges.

The development of deep learning has certainly boosted our pace towards fully autonomous driving. It offers a universal framework to implement complex tasks like object detection and semantic segmentation and achieves much better performance compared with traditional algorithms with handcrafted features and rules. For vision based perception, deep learning helps to detect 2d-objects[10], 3d-objects[26], lane lines[16], monitor driver status. For LiDAR based perception, deep learning is applied to detect cars and pedestrians[8, 23]. Recent work also try to apply deep reinforcement learning for planning and control[22].

On the other hand, by introducing deep learning to autonomous driving, the workload increases together with the accuracy. Deep learning algorithms require 10^{10} to 10^{11} floating point operations on processing one frame of image or point cloud. Besides, to achieve high accuracy, deep learning algorithms also increase the size of input data, which brings high-complexity pre-processing and post-processing. Workload of this scale is usually deployed on servers on the cloud. For autonomous driving, the system cannot bear the latency for connecting a remote server. To put the system on car, we are also faced with the problem of power consumption and heat dissipation. Deep learning brings great challenge to deploy autonomous driving algorithms.

System level optimization is required to implement autonomous driving system. Researchers have taken great effort to reduce the size of neural network models while keeping the accuracy. Deep Compression[5] proposes the method of pruning and trained quantization, which reduces the AlexNet model by 35x and VGG-16 model by 49x. Recent work focuses more on designing a more efficient network structure. SqueezeNet[7] and MobileNet[6] try to reduce the network size by specially designed layer parameters and topology. Latest research takes the idea of Neural Architecture Search (NAS) and designs network topology automatically according to accuracy or speed requirements[31]. Substantial studies have shown that the automatically discovered architectures by NAS are able to achieve highly competitive performance.

Besides model optimization, researchers have also paid great attention on hardware optimization. With a customized accelerator design, one can achieve more than 10x the speed and energy efficiency of CPUs or GPUs. A brief summary of this region is available at our website[4]. Commercial chips for autonomous driving also takes neural network processing unit as a core component in their SoC design, like NVIDIA Xavier, TI TDA4, EyeQ5, etc. Besides, the pre-processing and post-processing part of deep learning algorithms are also heavy workload for CPUs. As we will show in this paper, customized hardware for those non-NN part in deep learning algorithms is also critical for an autonomous driving system.

In this paper, we introduce our work on efficient computing platform design for autonomous driving systems. In section 2, we introduce our work on NN backbone optimization. In section 3, we propose low latency solutions to different algorithms based on Xilinx FPGA platform and Deep Learning Processing Unit (DPU) IP with our customized accelerators. In section 4, we introduce our hardware platform using the Xilinx ZU11EG MPSoC and NVIDIA Xavier SoC as computing cores, with abundant sensor interfaces. We plan to deploy the system on vehicle soon in Beijing 2022 Winter Olympic Park. Section 6 summarizes this paper.

2 NN MODEL BACKBONE OPTIMIZATION

CNN-based backbones are now widely used since their performances can surpass traditional methods significantly in a large range of perception tasks. And in recent years, many academic and industrial efforts have been devoted to optimizing the NN backbone for more efficient deployment. Generally speaking, there are three major directions to reduce the computational and storage burden of CNNs: 1) Network pruning [3, 5, 11, 28]: Remove redundant parameters or kernels; 2) Quantization [18, 19]: Quantize the weights and activations to low-precision values; 3) Compact block and architecture design [6, 7]: Design efficient blocks and architectures either manually or automatically.

2.1 Model Compression: Network Pruning

Deep Compression [5] proposes to remove small weights in the neural networks while maintaining its accuracy, thus significantly reducing the storage consumption. However, a specialized hardware accelerator design is needed to handle the resulting sparse computation pattern well enough and achieve improvements in terms of latency. For the ease of acceleration, structured pruning methods [3, 11, 28] introduce structured sparsity into the NN models, usually by regularizing the weights or batch normalization scales in a group-wise manner. In structured pruning, a major task

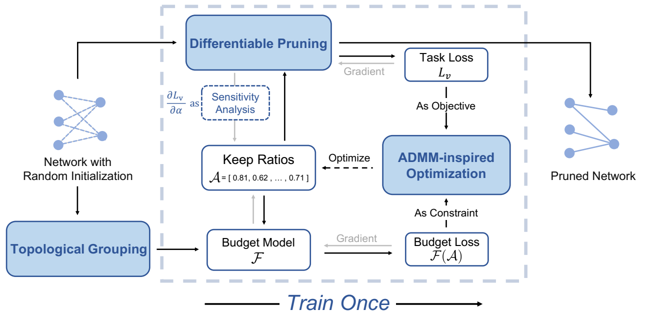

Differentiable Pruning
Task Loss L_v
Gradient
$\frac{\partial L_v}{\partial \alpha}$ as
Sensitivity Analysis
As Objective
Network with Random Initialization
Keep Ratios $\mathcal{A}$ = [0.81, 0.62 , … , 0.71]
Optimize
ADMM-inspired Optimization
Pruned Network
As Constraint
Topological Grouping
Budget Model $\mathcal{F}$
Gradient
Budget Loss $\mathcal{F}(\mathcal{A})$
Train Once

Figure 1: The workflow of Differentiable Sparsity Allocation.

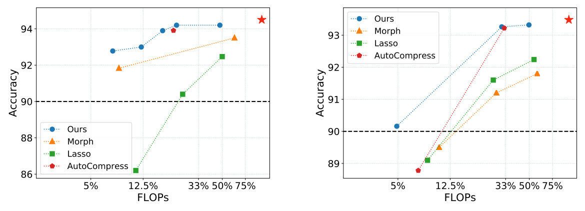
is to allocate sparsity across different network components, i.e., to decide how many channels to keep in each convolution. Intuitively, more channels need to be kept for sensitive operations, while fewer channels for unimportant operations.

In real applications, there are usually scenarios in which the resources of a NN model must be under certain budgets. Finding a suitable compressed model in these scenarios is formalized as the budgeted pruning problem. We propose differentiable sparsity allocation (DSA) to allocate sparsity across different network components, and assure that the resulting model is restricted under the resource budget [14]. Unlike the majority of budgeted pruning methods that follow an iterative pruning flow, we propose to find the proper sparsity allocation with a gradient-based method.

The workflow of DSA is illustrated in Figure 1. For the ease of hardware deployment, we need to make sure that the same set of channels are kept for the convolutions whose outputs are connected via elementwise operations, and we refer to this type of pruning constraints as topological constraints. Specifically, we first run a topological grouping procedure to analyze the data processing graph and find the topological constraints. And for each topological group $k = 1, \cdots, K$ with an original channel number of $C^{(k)}$, a keep ratio $\alpha^{(k)}$ and a mask variable $m^{(k)}$ is assigned. The keep ratio $\alpha^{(k)}$ indicates the expected proportion of kept channels for the convolutions in the k-th group, and $m^{(k)}$ is the corresponding 0-1 mask that indicates the channel choices. Then, in the differentiable sparsity allocation process, the differentiable pruning process takes $\alpha^{(k)}$ as the input, and output the corresponding $m^{(k)}$ through a randomized sampling process based on channel-wise batch normalization scales $\{b_i\}_{i=1,\cdots,C^{(k)}}$ as in Equation (1).

$$\begin{aligned} m_i &\sim \text{Bernoulli}(p_i); \quad p_i = f(b_i;\beta_1,\beta_2) = \frac{1}{1+(\frac{b_i}{\beta_1})^{-\beta_2}} \\ s.t. \quad & E[\sum_{i=1}^{C} m_i] = \sum_{i=1}^{C} f(b_i;\beta_1,\beta_2) = \alpha C, \end{aligned} \tag{1}$$

where the $^{(k)}$ superscript is omited for simplicity, β_2 is a hyperparameter that follows a increasing schedule, and β_1 is calculated by solving the expectation condition equation $E[\sum m_i] = \alpha C$.



(a) ResNet-18 (b) VGG-16

Figure 2: Pruning results on CIFAR-10.

With this differentiable pruning process, the gradient of the validation loss L_v w.r.t. the keep ratios $\mathcal{A} = \{\alpha^{(k)}\}_{k=1,\cdots,K}$ can be derived as

$$\frac{\partial L}{\partial \alpha^{(k)}} = \frac{\partial \beta_1}{\partial \alpha} \sum_{i=1}^{C} \frac{\partial L}{\partial p_i} \frac{\partial p_i}{\partial \beta_1} = C \sum_{i=1}^{C} \frac{\partial L}{\partial p_i} \hat{f}_i'; \quad \hat{f}_i' = \frac{f_i'}{\sum f_i'}, \tag{2}$$

where $\frac{\partial L}{\partial p_i}$ could be approximated using Monte-Carlo samples of the reparametrization gradients, and the interpretation of this equation is that the update of $\alpha^{(k)}$ is instructed using a weighted aggregation of the gradients of the task loss L w.r.t. the keep probabilities of channels $\frac{\partial L}{\partial p_i^{(k)}}$, and the aggregation weights are $f_i', i = 1, \cdots, C^{(k)}$.

Then, the gradient of task validation loss $\frac{\partial L_v}{\partial \mathcal{A}}$ and the gradient of the budget model $\frac{\partial \mathcal{F}(\mathcal{A})}{\partial \mathcal{A}}$ are orchestrated in an gradient-based algorithm framework. This orchestration is inspired by the ADMM algorithm to solve the constrained optimization problem of $\mathcal{A}$.

$$\begin{aligned} \mathcal{A}^* &= \arg\min_{\mathcal{A}} L_v(W^*(\mathcal{A}), \mathcal{A}) \\ \text{s.t.} \quad \mathcal{F}(\mathcal{A}) &\leq B_{\mathcal{F}}, \quad 0 \leq \mathcal{A} \leq 1 \\ W^*(\mathcal{A}) &= \arg\min_{W} L_t(W, \mathcal{A}), \end{aligned} \tag{3}$$

where L_t and L_v are the training and validation losses, respectively. And $\mathcal{F}(\mathcal{A})$ is the consumed resource corresponding to the keep ratios $\mathcal{A}$, and $B_{\mathcal{F}}$ is the resource consumption budget.

As an example, Figure 2 shows the pruning results of ResNet-18 and VGG-16 on CIFAR-10. We can see that DSA can maintain good performances while respecting the budget strictly. More details and experimental results can be found in our published paper [14].

2.2 Hardware-aware Neural Architecture Search

Human experts have been designed efficient building blocks for hardware platforms, such as the MobileNet series [6]. However, there is a wide range of applications and hardware platforms, and an automatic way of discovering suitable architecture with high hardware efficiency and promising application performance would help building up more and more AI-powered systems. Hardware-aware neural architecture search (NAS) [1, 25] has recently drawn much attention, in which the hardware efficiency is considered as a part of the reward besides the application performance.

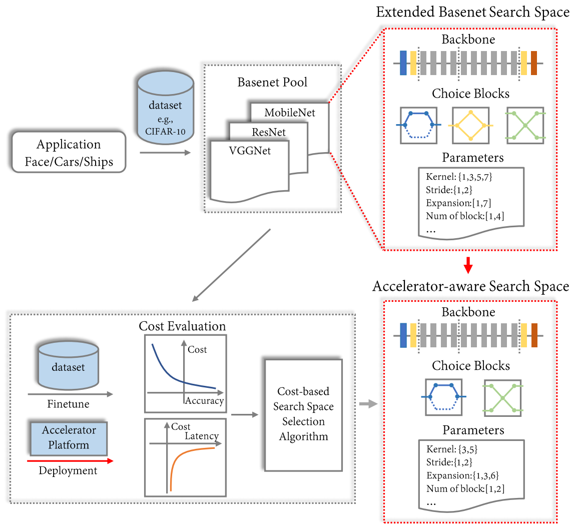


Figure 3: The workflow of the black-box search space profiling and selection method.

Pre-search: Black-box search space profiling and selection There are three major building components in a NAS algorithm: 1) Search space (SS) describes which architectural decisions need to be made; 2) Search strategy describes how we explore the search space based on the past exploration rewards; 3) Evaluation strategy gives out the evaluation rewards of candidate architectures to instruct the search process. In the past few years, lots of studies have been focusing on improving the efficiency of the NAS process by improving the sample efficiency of the search process [15, 20, 31], or by accelerating each architecture evaluation process [1, 17]. On the other hand, we facilitate an efficient search process by constructing a compact and effective search space that is suitable for the targeted hardware accelerator. To achieve that, we propose a black-box search space profiling and selection strategy as a preposition stage before the normal NAS process [30].

The workflow is illustrated in Figure 3. Firstly, we generate search space base networks (SSBNs) for each candidate SS, which has a different architecture of candidate network block and multiple parameters to be searched. Secondly, once we have the base network pool consists of different SSBNs, we need to evaluate the accuracy and latency of each candidate network in the pool. The latter is easier by running each base network on the accelerator platform. As for the accuracy, we directly train the specified network of each candidate SS on the target data set with 20 epochs. Then we use the weights of the specified network to finetune each base network with a few epochs to obtain accuracy. Next, we can compute the cost of each base network as

$$\text{Cost} = \exp\left(-\frac{\text{acc} - \text{thres}}{\text{scale}}\right) - \frac{\lambda}{\text{latency}}. \tag{4}$$

Finally, in order to find the search space that minimizes Cost(SS) in Equation (4), we design a simple cost-based search space selection algorithm to derive an optimized SS. The optimized SS has a much smaller controllable size and can provide a tradeoff between

Table 1: Results of ResNet-based network blocks on CIFAR-10.

	Fix Search Space			dynamic search space			
Method	Acc. (%)	Lat. (ms)	Cost	λ	Acc. (%)	Lat. (ms)	Cost
DNAS	86	1.287	0.9872	0.01	86.7	1.299	**0.9838**
			0.9173	0.1	86	1.28	**0.9169**
			0.2180	1	85.8	1.255	**0.1992**
RL-NAS	83.5	1.385	1.0003	0.01	86.5	1.292	**0.9848**
			0.9353	0.1	85.4	1.27	**0.9193**
			0.2855	1	84.7	1.25	**0.2015**

accuracy and latency with the user-defined parameter λ. And the SS generated by our method is *dynamic* in that it can have different sets of candidate blocks at each layer.

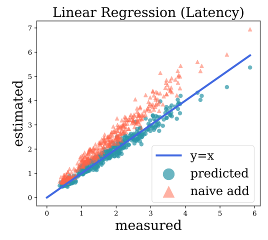
We compare the performance between the generated dynamic SS and fix SS using ResNet-based network blocks as an example on CIFAR-10, as shown in Table 1. We can see that the dynamic SS under three different λ configurations has the lowest cost compared with the fixed SS. This means that our proposed search space profiling method can preserve the useful information of the original SS, while pruning away redundant information in SS, thus making the search process fastly converge to a better result faster. More details and experimental results can be found in our published paper [30].

In-search: Estimation model of hardware latency and energy During the search phase, hardware latency or energy prediction models are necessary because deploying all the candidate networks to a target platform is often cost-prohibitive. A commonly-used strategy is to add up the latency or energy of the building blocks to estimate the candidate network's latency or energy. However, this linear assumption might not hold, e.g., on platforms with complex cache mechanism and massive parallelism such as CPUs and GPUs, the summation of block latencies can significantly deviate from the actual network latency. Thus, we construct latency and energy correction models to amend hardware-aware NAS.

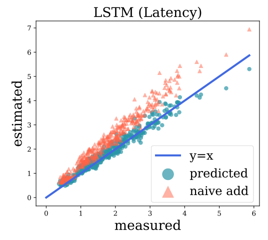
We compare two types of prediction models, including linear regression (LR) and LSTM-based models. Figure 4 shows their performances on predicting GPU (RTX-2080Ti) latencies and FPGA (Xilinx ZCU102) energies of candidate architectures from a MobileNet-v2-like search space. Each model is trained with 2k random samples and tested on another 1k samples. And we can see that the estimated costs have a much smaller rMSE and better linear correlation than the naive summation estimation.

3 NON-NN CUSTOMIZED OPTIMIZATION

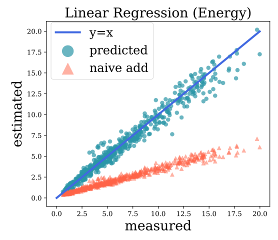
A series of hardware accelerators, deployment toolchains, and NN architecture design methods, have enabled NN backbone to be deployed efficiently in automatic driving systems. However, besides the NN backbone, the pre- and post-processing of NN-based algorithms are also computation intensive, and become the bottleneck of autonomous driving systems. In this section, we will introduce our recent systems including the customized non-NN optimization together with the NN backbone on general NN accelerators.



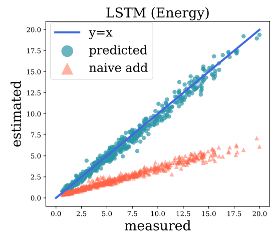
(a) GPU latency, rMSE=0.174



(b) GPU latency, rMSE=0.187



(c) FPGA energy, rMSE=0.709



(d) FPGA energy, rMSE=0.51

Figure 4: GPU latency and FPGA energy estimation using two types of prediction models (Linear regression, LSTM). For GPU latency, estimation by naively adding up block latency results in rMSE=0.63. For FPGA energy, naive addition results in rMSE=4.94. Using the prediction models, we can achieve 3.6× and 9.7× better rMSE for GPU latency and FPGA energy respectively.

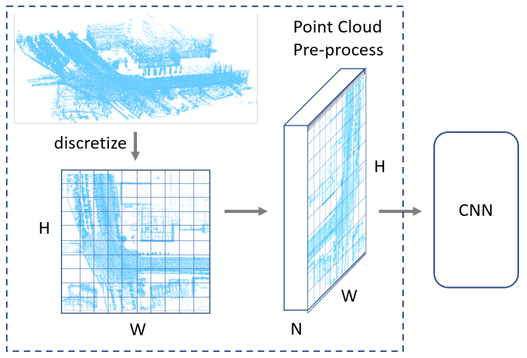


Figure 5: Data pre-process flow of point-cloud-based object detection.

3.1 Pre-processing for Point Cloud

Even though vision based methods have shown outstanding performance on object detection tasks, it is still not enough for autonomous driving of which the environment can be dark sometimes. LiDAR serves as a good assistant to cameras in many cases because it gives depth information and works better in dark environments. Different algorithms try to detect objects from point cloud by extracting geometry features.

Convolutional Neural Networks (CNNs) are widely applied in vision-based and LiDAR-based detection tasks. Different from images, point clouds need pre-processing to be transformed into a

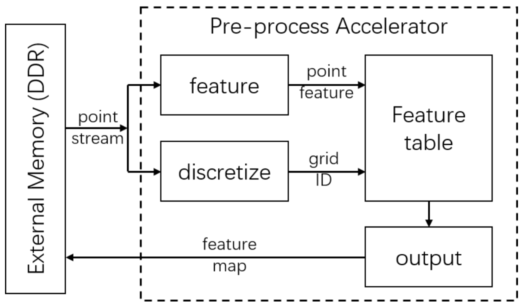


Figure 6: Basic architecture of a point cloud pre-process accelerator.

feature map as neural network input. For autonomous driving, a commonly used method is to project the point cloud onto X-Y plane and discretize the points with a $W \times H$ 2D grid. To save computation, the algorithm usually limits the number of points in each grid to about 64 ∼ 200, and drops the extra points. Then the algorithm compute a feature vector for each grid using the corresponding points. We summarize the point-cloud pre-processing flow in Figure 5.

Complex YOLO[23] chooses 3 features: the maximum z, the maximum intensity, and the logarithm-normalized point number in this grid. So the pre-process result is a $3 \times W \times H$ tensor like a normal CNN input. PointPillars[8] first represents each point with a M-D feature vector using the relative position to the gird, intensity, absolute position, etc. Then the algorithm transforms each M-D feature to an N-D feature with a trained fully connected network. Finally, for each grid, all the corresponding feature vectors are merged by max pooling. Thus the pre-processing result is an $N \times W \times H$ tensor.

As we will show later in the experiment part, after applying model compression, pre-processing becomes a speed bottleneck of the algorithm. To address this problem, we propose FPGA-based pre-processing accelerators to work with the NN accelerator. For different algorithms, the accelerators differ in function modules but share a same basic architecture, as shown in Figure 6. The accelerator first takes the input in a point stream format from the external memory. The feature module computes the feature of each point. The discretize module computes the corresponding grid ID of each point. The hardware then update the grid feature table with the feature and the gird ID. After all the points are processed, the output module traverse all the grids, write the result back to the external memory. In this paper, we introduce our pre-process accelerator designs for Complex YOLO and PointPillars.

For Complex YOLO, the feature module simply extracts z and intensity value of each point and applies quantization to z and i to save memory. The feature table stores the maximum z, intensity, and the number of points in each. A typical feature map contains 1216×608 grids for Complex YOLO, which means more than 2MB storage consumption. This will greatly limit the resource for the NN accelerator in system. So we partition the feature map into different blocks and process one block each time. The merge module computes the logarithm of the point count of each grid. As the algorithm limits the point count, we implement the logarithm operation with a lookup table.

For PointPillars, the feature module computes the NN part. PointPillars uses much larger feature vectors, with 32 or more elements, compared with Complex YOLO. Thus we choose to store the feature table in external memory. Directly do max-pooling when updating the feature table means the latency of accessing external memory will greatly limit the speed of processing each point. So we directly store all the point feature vectors in the external memory. To help traverse all the grids, we maintain a point count map on-chip. For each grid, the output module reads the point feature vectors from external memory, do max-pooling and write the results back.

We evaluate our design on Xilinx ZCU102 development board using a Xilinx B4096 DPU as the deep learning processing unit. We configure the pre-processing accelerators to write the result directly to the DPU input location, which reduce the cost of I/O APIs. The results are shown in Table 2. For the default and C configuration, all the algorithm except the NN part runs on the embedded ARM A53 in the SoC. For the C+P configuration, the pre-process part runs with customized accelerator on FPGA. With our model compression methods, the total latency can be reduced by 27% ∼ 56% and pre-process becomes the bottleneck. With our customized pre-processing accelerators, we can further reduce the latency by 54% ∼ 72%. The resource consumption of the proposed pre-processing accelerators are shown in Table 3. With the proposed pre-processing accelerators, we achieve 10fps point-cloud-based object detection, that matches the speed of LiDAR scan.

Table 2: Latency breakdown of point-cloud based detection on FPGA (ms). C: with model compression. P: with pre-process acceleration.

Algorithm	Config	Pre-proc	NN	Post-proc	Total
Complex YOLO	default	87.05	246	9.43	342.48
	C	84.31	60.4	5.59	150.3
	C+P	9.6	60.4	12	82
PointPillars car	default	88.4	59.4	4	151.8
	C	88.4	18.5	4	110.9
	C+P	8	18.5	4	30.5
PointPillars ped & cyc	default	87.9	236	16	339.9
	C	87.9	75.1	16	179
	C+P	8	75.1	16	99.1

Table 3: Resource consumption of the point cloud pre-process accelerators on ZU9EG FPGA.

	LUT	FF	BRAM	DSP
Complex YOLO	8.8k (3.2%)	11k (2.0%)	58 (6.4%)	15 (0.6%)
PointPillars	23k (8.4%)	18k (3.4%)	83 (4.6%)	138 (5.5%)

3.2 Post-processing for feature-point extraction

In vision-based automatic driving systems, feature-point extraction is a basic component for visual odometry or map building [13]. The feature-point extraction method usually has two steps: 1) feature-point detection and 2) feature descriptors generation. SIFT[12] and ORB[21] are two popular handcrafted methods for

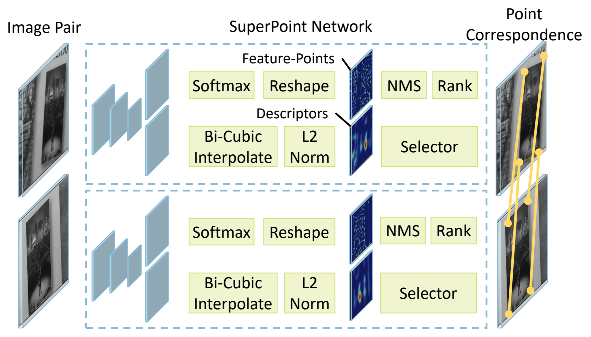


Figure 7: SuperPoint with CNN backbone and post-processing operations such as NMS, Rank, Pixel-wise normalization.

feature-point extraction. Compared with the handcrafted methods, the CNN-based feature-point extraction methods, such as SuperPoint [2], have made significant progress in both feature-point detection and descriptor generation. Figure 7 shows the structure of SuperPoint. It includes both feature-point detection step and descriptor generation step in a single forward pass, and surpasses the traditional handcrafted methods in accuracy. Our work [29] designs specific hardware architecture along with the previous CNN accelerator (DPU) to implement real-time feature-point extraction on the embedded system.

In addition to the CNN backbone, there are many *post-processing operations* in CNN-based feature-point extraction networks, such as Non-Maximum Suppression (NMS) and confidence ranking in the feature-point detection, as well as pixel-wise normalization in the descriptors generation. To deploy the entire process of feature-point extraction on real-time embedded systems, we propose a hardware-software co-design CNN-based feature-point extraction structure and accelerate the process on the Xilinx ZCU102 platform.

There are two components which consume most of the computation. The *Softmax* operation to find out the most possible position of a feature-point inside a small local area, and the *normalization* operation to calculate the normalized descriptor for each detected feature-point.

As Softmax's is to locate the point with the highest confidence, we only need to calculate the final Softmax result of the pixel with the highest confidence. The results of other pixels with low confidence can be skipped and discarded. Thus, we design a hardware for Softmax shown in Figure 8(a). It consists of three parts, the add tree, the compare tree and the divider. Softmax reads 65 numbers from a grid region at once. Adder tree computes input to the power of 2 using shift operation and calculates their sum. Compare tree reads the values of the first 64 channels and returns the maximum value and its channel number, which contains position information. The divider uses the shift operation to calculate the reciprocal of confidence.

Besides Softmax, the pixel-wise normalization also consumes huge computation. Thus, we also design a specific hardware for this operation to parallel the operation, as shown in Figure 8(b). The normalization module can read 8 numbers per clock cycle. The normalization process is divided into three stages and requires each

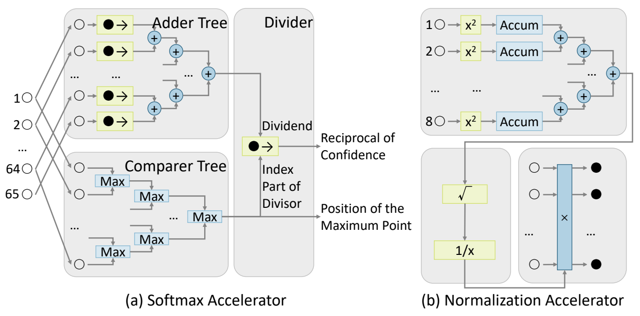


Figure 8: Hardware Architecture for SuperPoint Post-processing

Table 4: Time Comparison of Each Operation of SuperPoint

	CNN backbone*	Post-processing				
		Softmax	NMS	Rank	Norm	Total
CPU	24ms	31ms	27ms	0.97ms	42ms	100.97ms
Ours		1.97ms	0.7ms	0.12ms	1.44ms	4.23ms

* The CNN backbone runs on DPU.

Table 5: Accuracy and Time Results on TUM SLAM Dataset[24] of Different Feature-point Extraction Methods

	RPE(m/s)	ATE(m)	Run time(ms)
SIFT	0.0319	0.4219	2397
ORB	0.0577	0.6105	229
Origin Superpoint	0.0280	0.3671	259
Ours	0.0283	0.3976	59

* RPE is the mean Relative Pose Error to indicate the translational drift per second. ATE is the root mean square Absolute Trajectory Error to indicate the translational drift of the entire trajectory. The less, the better.

descriptor to be read twice. In the first stage, we compute the sum of the squares of the descriptors, which takes 32 clock cycles when the descriptor dimension is 256. Then the reciprocal of square root of sum is computed as the normalization coefficient. In the final stage, the descriptor is read a second time and multiplied by the normalization coefficient.

The proposed CNN-based feature-point extractor is implemented and evaluated on the ZCU102 board. The CNN backbone is calculated by DPU. We compare the running time of each operation in SuperPoint before and after the optimization. The results are shown in Table 4. The total running time of post-processing operations is reduced by more than 20×.

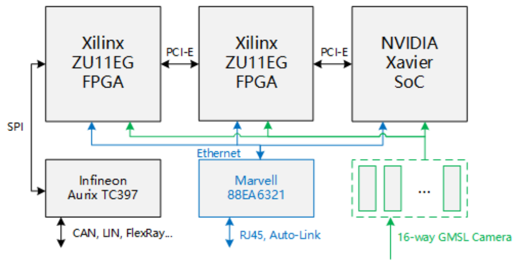
We use Superpoint as the feature-point part of VO, and evaluate the VO performane on the *TUM* [24] dataset. We evaluate SuperPoint against two well-known detector and descriptor systems: SIFT[12] and ORB[21]. We perform nearest neighbor matching from descriptors in adjacent frames. We use an OpenCV implementation (solvePnP()) [9] with all the matches to compute the transform matrix, and use Bundle Adjustment [27] to optimize results. All the computation of ORB and SIFT is done on the CPU. And all the computation of the original SuperPoint is done on the CPU except the CNN backbone.

The results are shown in Table 5. In terms of accuracy, SuperPoint outperforms ORB and SIFT. Our optimizations, including fixed-point quantization, and post-processing acceleration, do not introduce a significant loss of accuracy. In terms of calculation speed, SuperPoint takes less time than SIFT and is equivalent to ORB. After optimization, the running speed is increased by 4×, making real-time processing possible.

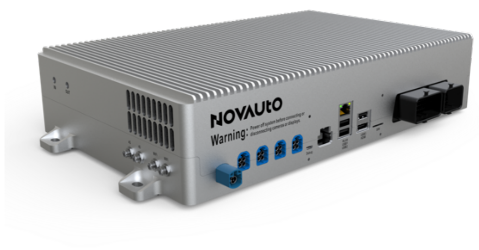
4 COMPUTING HARDWARE

To support the real-time processing of data from multiple sensors, we need a computing hardware platform with powerful performance cores. A main CPU is also required for the processing of other tasks, such as planning, and the basic software environment including the operation system.

Mostly for the demonstration test vehicles, an industrial server will be installed for computing. The configuration can usually be a X86 CPU with multiple GPU graphic cards for acceleration. The overall power cost can reach up to thousands of watts and a fan or water cooling system is required. To control the power down, more efficient architecture needs to be considered. For the consideration of safety, an ASIL-D level MCU is also necessary to connect the performance cores and the CAN-bus of vehicles. Hence, we have taken a heterogeneous design for our level-4 autonomous driving computing platform, which we call as NOVA-30. As shown in Fig.9a, we can see there are two Xilinx ZU11EG FPGAs and one NVIDIA Jetson Xavier SoC module integrated on our platform. Gigabit ethernet and PCI-E are implemented for cross-chip communication. The GMSL and ethernet ports are provided for HD cameras, LiDAR and radar inputs. An Infineon Aurix TC397 bridges the performance cores and vehicle CAN-bus. The overall power of NOVA-30 can be controlled in 90 watts.



(a) the block diagram of hardware architecture of NOVA-30



(b) the actual picture of NOVA-30

Figure 9: The computing hardware of Novauto - NOVA-30

5 VEHICLE PLATFORM

5.1 Sensor Configuration

The vehicle platform incorporates various refined sensors, such as cameras, LiDAR, and millimeter wave radar. Detailed sensor configuration is shown in Figure 10. Through the multi-sensor fusion of LiDAR, radar and cameras, 360-degree environment perception and multi-range perception are realized. The localization subsystem is fused with vision-based sensors for high-precision localization in different scenarios.

Efficient computing platform is critical to the deployment of autonomous vehicles. The environment perception functions, including multi-object detection and tracking, lane detection, traffic sign detection, real-time high-precision localization, are built by leveraging the computing platform. All the sensors are powered by the computing platform that efficiently process massive data in real-time.

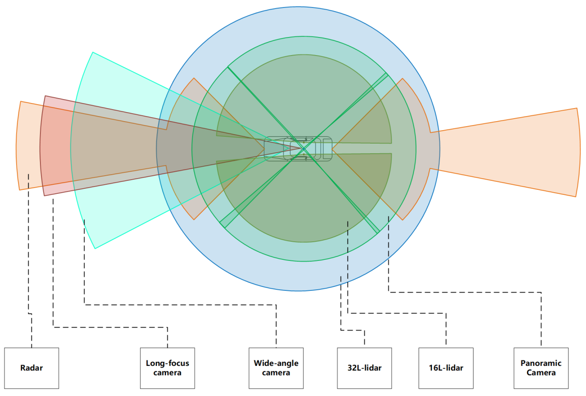


Figure 10: Top view of the vehicle platform with suites of sensors.

5.2 Application Demonstration

Based on the vehicle platform, a Level 4 driving automation demonstration is carried out in real and open environments to prove the capability of the efficient computing platform. Our reference autonomous driving system is shown in Figure 11. The autonomous vehicle runs in Shougang Park, which is rebuilt for the 2022 Beijing Winter Olympics. A variety of autonomous driving functions are implemented to ensure the safety of pedestrians, vehicles and other traffic participants. The functions comprise of standing start, vehicle stop, leading vehicle following, lane changing, passing through intersection, turning at intersection, turning over at intersection, vehicle meeting, overtaking, parking, emergency braking and response to traffic light. The total test mileages are greater than 900km, which includes a variety of different weather conditions and a full coverage of traffic scenarios. The typical cruising speed of the vehicle is 30km/h, .

6 SUMMARY

There is still a long way to go to achieve high-level autonomous driving. Deep learning is an effective solution but we need more

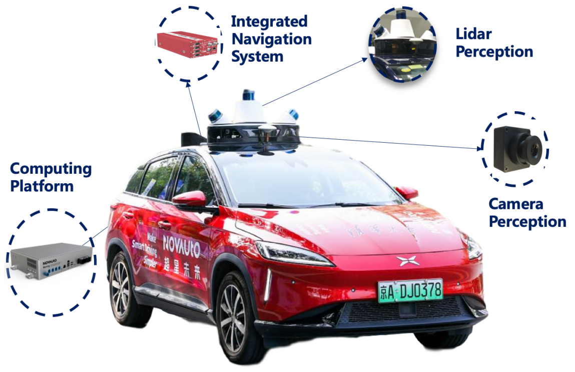


Figure 11: Novauto's autonomous driving vehicle.

work to fit it into the on-vehicle computing platform. In this paper, we introduce our work on efficient computing platform design for autonomous driving system. We introduce both the software-level model optimization and hardware-level accelerator designs which proves the importance of system-level optimizations. We will do more experiments on-vehicle with our NOVA-30 platform in the future.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (2018YFB0105000), Meituan Group award, National Natural Science Foundation of China (No. U19B2019, 61832007, 61621091), Tsinghua EE Xilinx AI Research Fund award, Beijing National Research Center For Information Science And Technology award and Beijing Innovation Center for Future Chip award.

REFERENCES

[1] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations*. https://arxiv.org/pdf/1908.09791.pdf

[2] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 2018. Superpoint: Self-supervised interest point detection and description. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 224–236.

[3] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1586–1595.

[4] Kaiyuan Guo, Wenshuo Li, Kai Zhong, and et al. 2020. Neural Network Accelerator Comparison. http://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/.

[5] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[6] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[7] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[8] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. 2019. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 12697–12705.

[9] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. 2009. EPnP: An Accurate O(n) Solution to the PnP Problem. *International Journal of Computer Vision* 81, 2 (2009), 155–166.

[10] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[11] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*. 2736–2744.

[12] David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60, 2 (2004), 91–110.

[13] Ral Mur-Artal and Juan D. Tards. 2016. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics* 33 (2016), 1255–1262. https://doi.org/10.1109/tro.2017.2705103

[14] Xuefei Ning, Tianchen Zhao, Wenshuo Li, Peng Lei, Yu Wang, and Huazhong Yang. 2020. DSA: More Efficient Budgeted Pruning via Differentiable Sparsity Allocation. In *ECCV*.

[15] Xuefei Ning, Yin Zheng, Tianchen Zhao, Yu Wang, and Huazhong Yang. 2020. A Generic Graph-based Neural Architecture Encoding Scheme for Predictor-based NAS. In *ECCV*.

[16] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. 2016. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv preprint arXiv:1606.02147* (2016).

[17] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[18] Jiantao Qiu, J. Wang, Song Yao, K. Guo, Boxun Li, Erjin Zhou, J. Yu, T. Tang, N. Xu, S. Song, Yu Wang, and H. Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *FPGA '16*.

[19] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*.

[20] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

[21] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2012. ORB: An efficient alternative to SIFT or SURF.

[22] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. 2017. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging* 2017, 19 (2017), 70–76.

[23] Martin Simony, Stefan Milzy, Karl Amendey, and Horst-Michael Gross. 2018. Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 0–0.

[24] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. 2012. A Benchmark for the Evaluation of RGB-D SLAM Systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*.

[25] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2815–2823.

[26] Bugra Tekin, Sudipta N Sinha, and Pascal Fua. 2018. Real-time seamless single shot 6d object pose prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 292–301.

[27] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. 2000. Bundle Adjustment — A Modern Synthesis. In *Vision Algorithms: Theory and Practice*, Bill Triggs, Andrew Zisserman, and Richard Szeliski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–372.

[28] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*. 2074–2082.

[29] Zhilin Xu, Jincheng Yu, Chao Yu, Hao Shen, Yu Wang, and Huazhong Yang. 2020. CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 33–37.

[30] Shulin Zeng, Hanbo Sun, Y. Xing, Xuefei Ning, Y. Shan, X. Chen, Yu Wang, and Hua zhong Yang. 2020. Black Box Search Space Profiling for Accelerator-Aware Neural Architecture Search. *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2020), 518–523.

[31] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).