An FPGA-based Neural Network Overlay for ADAS Supporting Multi-model and Multi-mode

Jiaxi Zhang[1,*], Tao Yang[2,*], Qingzheng Li[3,*], Bo Zhou[3], Yang Yang[3], Guojie Luo[1], and Jianping Shi[3]
[1]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing, China
[2]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
[3]SenseTime Group Limited, Beijing, China

2021 IEEE International Symposium on Circuits and Systems (ISCAS) | 978-1-7281-9201-7/20/$31.00 ©2021 IEEE | DOI: 10.1109/ISCAS51556.2021.9401751

***Abstract*—Advanced Driver-Assistance Systems (ADAS) are complex systems consisting of many computer vision tasks including image classification, object detection and semantic segmentation. FPGA is a feasible solution for deep learning based computer vision accelerator due to its high performance and energy efficiency. However, design a high performance FPGA accelerator requires good understanding of basic hardware concepts and consumes a long compilation time. Overlays can alleviate the above problems by accelerating applications in a software via a hardware architecture and a compiler. In this paper, we propose an FPGA-based neural network overlay processor for ADAS. The overlay architecture contains almost all common computation layers for learning based ADAS. In addition, we design a compiler that can automatically compile the high-level description of neural networks from deep learning framework like Caffe and Tensorflow into FPGA configurable codes, which can be executed by our overlay architecture without reprogramming. Experiments show that our overlay can process learning tasks in ADAS with low latency and low memory usage.**

I. Introduction

Deep neural networks (DNNs) are powering many modern artificial intelligence (AI) tasks from natural language processing, medical services, to advanced driver assistance systems (ADAS) in recent years. ADAS consist of many computer vision tasks [1] including vehicle detection, pedestrian detection, traffic sign recognition, lane detection, etc. Multi-task learning [2] can solve multiple learning tasks at the same time while exploiting commonalities and differences across tasks. It is a feasible solution for ADAS and requires hybrid networks of image classification, object detection and semantic segmentation.

FPGA has become promising solution for computation-intensive applications due to its high performance and energy efficiency. FPGA is reprogrammable and has the ability to accommodate late-stage design changes and produce prototype design fast for multi-task learning networks. However, FPGA design requires good understanding of basic hardware concepts such as pipelining and synchronization to get the best performance. In addition, compilation will consume a long time because the hardware design will take logic synthesis, placement, routing to obtain the bitstream.

Overlay [3] has been proposed to alleviate the above problems and helps designers to compile high performance applications to FPGA hardware in merely seconds rather than hours. An FPGA overlay is a virtual layer between applications and FPGA hardware. With this additional layer, user applications will be targeted toward the overlay architecture instead of implemented onto the physical FPGA directly. Software applications can be directly accelerated via a particular software programmable overlay architecture and an architecture-aware compiler.

FPGA-based DNN architectures have been extensively studied, from specific network accelerators to general DNN accelerations. Authors of works [4], [5], [6] proposed several specific network architecture designs of AlexNet, VGGNet and LSTM respectively. Later, several works [7], [8] focused on compilers of DNN accelerators to directly map the high-level description DNN onto hand-optimized design templates. Recently, DLA [9] and OPU [10] designed general DNN overlay architectures on FPGA, and delivered instructions sets and architecture-specific compilers to directly map DNN on overlay with high performance. However, all these works were not specially designed for multi-task acceleration in ADAS.

ADAS are complex systems and have higher requirements to FPGA accelerators. It is very common to use multiple learning models to process data from different sensors. And architectures of learning models required for each sensor are also completely different. We call this requirement as **multi-model**. In addition, a model can be used in different ways to finish computation tasks with high performance and energy efficiency. We call this requirement as **multi-mode** (See details in Section II). A complete ADAS task often has multiple models, and each model requires multiple modes. Previous works of DNN accelerator and overlays on FPGA described earlier did not explore more on multi-model and multi-mode supports.

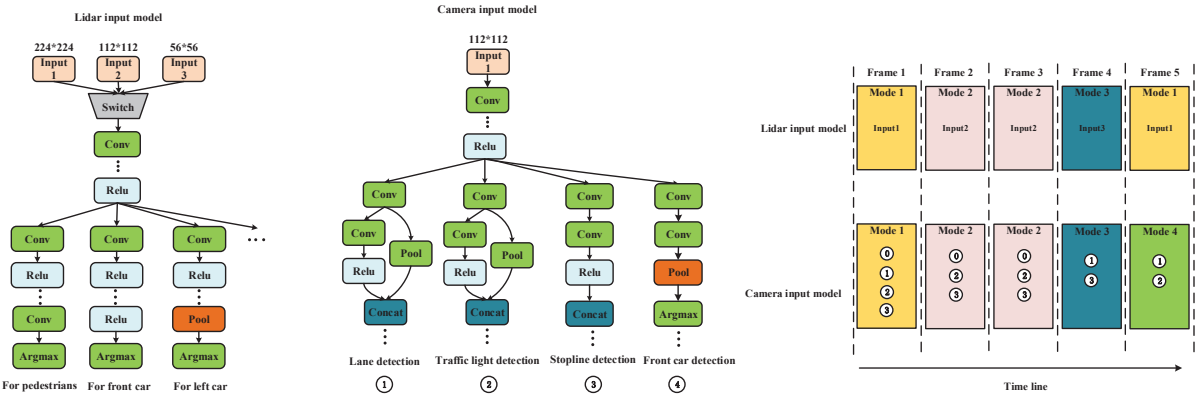
In this paper, we propose an RTL-based FPGA overlay targeting at ADAS DNN accelerations. The overlay leverages both hardware and software optimizations to further accelerate the processing speed. Our contributions can be summarized as the following:

- On the hardware side, we introduce a hardware architecture design consisting of several kinds of process elements (PEs) for executing different operations in ADAS models. The hardware architecture also contains buffers and datapaths that help compute neural networks with high performance.
- On the software side, we propose an associated compiler including operation rerank, operation fusion, and branch partition. The compiler will rerank the processing layers and perform operation fusion to improve the computation efficiency and reduce memory usage. It will also generate configuration instructions for data that are loaded into the FPGA.
- Due to hardware and software optimizations, our FPGA overlay design meets the requirements of multi-model and multi-mode in ADAS accelerations. We can map neural networks from deep learning frameworks to hardware architecture directly to enhance the programmability and flexibility of the overlay. Experiments show that our overlay can process learning tasks in ADAS with low latency.

*These authors contributed equally.

978-1-7281-9201-7/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: IEEE Xplore. Downloaded on May 16,2021 at 08:27:40 UTC from IEEE Xplore. Restrictions apply.



(a) Model 1: Lidar Input Model. (b) Model 2: Camera Input Model. (c) Multi-mode scenario for Model 2.

Fig. 1: Multi-model and Multi-mode Examples in ADAS.

II. Preliminaries

A. Details of Multi-model and Multi-mode

In ADAS tasks, it is common to use multiple models to process data from different sensors, and the model architectures required for each sensor are also completely different. Figure 1 gives typical tasks in ADAS. Figure 1a is a Lidar input model. The input of the Lidar model is lidar point cloud. There are three types of input point number, 224×224, 112×112, 56×56, corresponding to big detect scope, middle detect scope and small detect scope respectively. In addition, there are also several functions for pedestrians, front cars, left cars and so on. Figure 1b is a Camera input model. The overall model is responsible for the many detection sub-tasks including stopline, lane markings, traffic lights and other targets. These sub-tasks first share several layers and weights for computation, and then each has different sub-models to finish specific tasks. An FPGA accelerator should support all these learning models, which is **multi-model** property in ADAS.

On the other hand, a model can be used in different ways. Figure 1c shows a scenario of Model 2. The required detection sub-tasks are different in different frames. In frame 1, ADAS needs lane detection, traffic light detection, stopline detection and front car detection. In frame 2, ADAS does not require traffic light detection. So we do not have to execute the entire network in every frame, which can reduce latency and save power. Thus, each model requires multiple modes to finish computation tasks with high performance and energy efficiency. This is **multi-mode** property in ADAS.

B. Motivation

According to Section II-A, a complete ADAS task often has multiple models, and each model requires multiple modes. It is necessary to have multiple modes to adjust the quantizations or select partial network of different models to speed up the computation or save power. However, architecture design and memory allocation for ADAS to support multi-model and multi-mode is non-trivial due to the limited computing and memory resources on FPGA boards. On one hand, we should create a flexible architecture to compute all functions of different models with high performance and energy efficiency rather than a specific NN model. On the other hand, we have to provide an architecture-aware layer scheduling and memory allocation mechanism to efficiently run models in different modes.

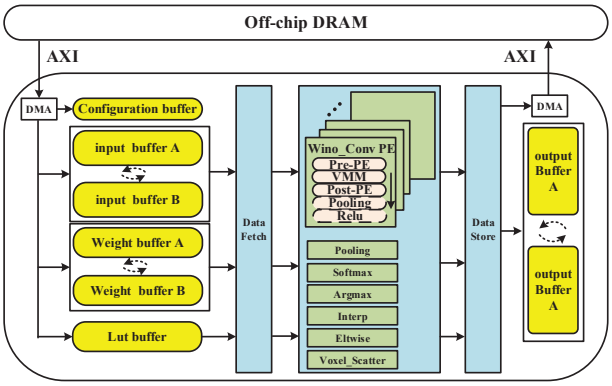


Fig. 2: The hardware architecture.

III. Overlay System

In this section, we will first show the hardware architecture of overlay design, and then introduce the associated compiler. Our FPGA overlay design meets the requirement of multi-model and multi-mode in ADAS accelerations due to hardware and software optimizations, and we can map neural networks from deep learning frameworks to hardware architecture directly to enhance the programmability and flexibility.

A. Hardware Architecture

The overlay architecture must be general to implement all neural network tasks in ADAS, and it also needs less control overhead when switching between different models and modes. Fig. 2 shows the overall architecture of our hardware accelerator. It consists of several kinds of computation kernels and buffers for memory optimization.

1) Computation Kernels: There are several processing elements in our design. We adopted Wino_Conv PE kernel for accelerating convolutional layers due to Winograd algorithm can speed up convolution by reducing the number of multiplication. The Wino_Conv kernel consists of pre-processing element (denote as pre-PE), VMM core and post-processing element(post-PE). When we execute the Winograd convolutions, the Pre-PEs execute Winograd input transformation. Then, the transformed Winograd input features are fed into the corresponding VMM-core to execute the VMM operations. After getting all the output vectors of VMM-cores, the Post-PEs execute post-Winograd transform to get the output tile and store

Authorized licensed use limited to: IEEE Xplore. Downloaded on May 16,2021 at 08:27:40 UTC from IEEE Xplore. Restrictions apply.

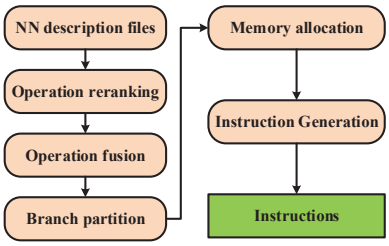


Fig. 3: Flow of compiler.

the results in output feature buffers. Besides the most important Wino_Conv PE, we also implement Pooling, Softmax, Argmax, Interp, Eltwise in our hardware to execute corresponding operations in CNN models. A Voxel_Scatter PE is allocated in our hardware to support the CNN models about Lidar.

2) Buffers: In order to make full use of computing kernels, we designed several buffers based on datapaths of the CNN computation, including input feature map buffer, weight buffer, output feature map buffer and LUT buffer. All these buffers are allocated to store the corresponding data and provide data to the processing elements (PEs) in a stream. Typically, it is impossible to load all the input feature maps at once due to the limited BRAM resources on FPGA and large feature maps. To solve this, we use ping-pong buffers to store tiles of input features, weights and output features. With this coarse-grained pipeline, we can overlap the transformation operations and computations.

After configuration buffer loading instructions from off-chip DRAM, input buffers and weight buffers will load input features and weights from off-chip DRAM according to address information in instructions. When input features and weights fill one of the ping-pong buffers, the processing elements is working and results are stored in output buffer. Finally, the output tile is sent back to off-chip DRAM via DMA.

B. Associated Compiler

To map a neural network from deep learning frameworks directly on our hardware architecture with high performance, we need a compiler. Figure 3 gives the running flow of our compiler. The compiler takes the neural network description files from learning frameworks. Actually, we use Open Neural Network Exchange (ONNX) as input format of compiler because models from different deep learning frameworks can be easily converted to ONNX. Overall compiling flow contains four phases, operation rerank, operation fusion, branch partition and memory allocation. Besides, memory allocation is determined by the first three phases. After these four steps, the generated instructions will guide FPGA configurations and processing.

1) Operation Rerank: Since a complex CNN model cannot be executed on FPGA at the same time due to the limited computing resource, we need to arrange the processing order of each layer, which is called model rerank. Computation graph of CNN models can be regarded as Directed Acyclic Graphs (DAGs). To ensure the correctness, and maintain high computation performance, rerank phase have to meets the following rules.

Rule 1. One layer can only be processed after all previous layers have been completed.

Rule 2. One branch needs to finish the last layer in it before the next branch can be processed.

Algorithm 1 Reversing topo-order based rerank algorithm

Input: Computation Graph of a Model$G$
Output: List Q for the computation order
Initialize $Stack$
Initialize List R for reversing order
Initialize $node\_visited[]$ to the out degree of each node in G
Initialize $edge\_visited[]$ to false
for $node$ in G **do**
  if $node\_visited[node]$ = 0 **then**
    $Stack$.push(node)
  end if
end for
while !$Stack$.empty() **do**
  $node$ = Stack.top()
  Stack.pop()
  if $node\_visited[node]$ = 0 **then**
    R.push(node)
    for $edge$ in $node$.edges_in **do**
      if !$edge\_visited[edge]$ **then**
        $node$ = $edge$.in_node
        $node\_visited[node]$ -= 1
        $edge\_visited[edge]$ = true
        if $node\_visited[node]$ = 0 **then**
          $Stack$.push(node)
        end if
      end if
    end for
  end if
end while
Q = R.reverse()

The first rule is to keep the feature map order of the model while the second rule is to increase data reuse to enhance the performance. A reversing topo-order based algorithm is used to solve this problem, as shown in Algorithm 1. We can get the reversing order R by a modified reversing topo-order algorithm and the correct execution order Q can be obtained by reversing R. We use $node\_visited$ to determine whether all branches of a layer have been visited. Only after all branches of a node has been traversed, this node will be pushed to R, which will ensure Rule 2 in Q.

2) Operation Fusion: Operation fusion fuses the adjacent operations into an operation according to the hardware implementation. We just need to load data from and to the off-chip DRAM for a time for all these fused operations by this step, which reduces the time by avoid loading and pushing back data multiple times. On the other hand, operation fusion can also reduce data movement between on-chip and off-chip memory. According to our basic hardware implementation shown in Figure 4c, we can choose whether to execute the Pooling layer and Relu layer after Conv layer.

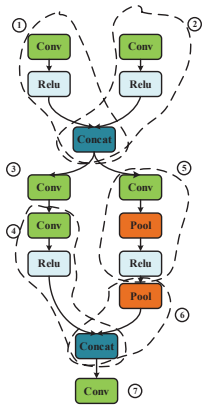
Rule 1. Adjacent Conv-Pooling-Relu layers or a subset of them will be fused.

Rule 2. A layer with concat layer will be fused.

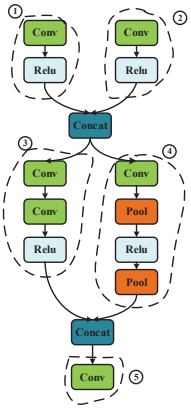
The first rule is designed based on the hardware architecture, and the second one is designed for branch optimization.

3) Branch Partition: Branch Partition Phase divides the CNN graph into several branches. For each branch, we just need to allocate a ping-pong memory according to the layer with the biggest feature maps. The rule we dividing a series of layers into a branch

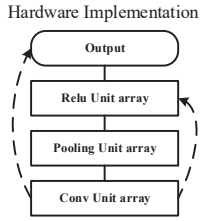
Authorized licensed use limited to: IEEE Xplore. Downloaded on May 16,2021 at 08:27:40 UTC from IEEE Xplore. Restrictions apply.



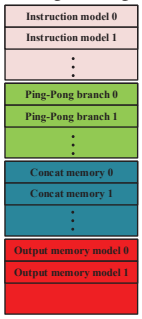
(a) An example for fusion.



(b) An example for partition.



(c) Wino_Conv PE.



(d) Memory distribution.

Fig. 4: Details of compiler.

is as follows:

Rule 1. A branch should start from feature map, and no concat layers and no forks are allowed during the process.

For example, in Figure 4b, the CNN computation graph is divided into five branches using the rule before. Then we allocate a ping-pong buffer for each branch. Layers in a branch fetch input feature maps from one of the ping-pong memory and store the results of this layer into another one iteratively. In this way, the computation of a branch can be executed using a ping-pong memory.

Besides, for hardware efficiency, IPs in our design need to fetch data from a continuous storage space. As shown in Figure 4d, we allocate a separate space for concat layers. The output features of layers connected to concat layers are stored in this *concat* memory continuously, then the first layer of the next branch could fetch the input features from the concat.

Using these branch partition and memory allocation strategy, we will save a lot of memory, which make it possible to map big models on an FPGA platform with only 1GB or 2GB DRAM.

IV. Experimental Evaluations

A. Experimental Setup

We adopted a common winograd convolutional layer implementation from several academic papers [11]. Data type in all PEs can be set to fixed 4/8/16 bit based on network accuracy requirements. We implemented the overlay architecture on Xilinx embedded FPGA board ZCU102, and compiler run in Arm A53. The overlay architecture consumed 540 BRAM18K, 532 DSPs, 91874 FFs and 89628 LUTs, which are 50%, 59%, 21% and 41% of the ZCU102 FPGA board respectively. And the operating frequency is 330MHz.

model	mode	latency (ms)
Vision: Resnet50-Backbone	LD + TLD + SLD + FCD	52.44
	LD + TLD	33.70
	LD + SLD	30.45
	LD + TLD + SLD	45.10
	TLD + SLD	35.50
	SLD + FCD	27.30
Vision: Vgg16-Backbone	LD + TLD + SLD + FCD	39.48
	LD + TLD	24.54
	LD + SLD	22.62
	LD + TLD + SLD	31.00
	TLD + SLD	35.93
	SLD + FCD	26.80
Lidar: Resnet50-Backbone	input point number: 256*256	47.2
	input point number: 128*128	18.4
	input point number: 64*64	8.97

TABLE I: Latency of multi-model and multi-mode.

	Baseline 1	Baseline 2	Our
Vision: Resnet50-Backbone	307MB*6	102MB*6	106MB
Vision: Vgg16-Backbone	247MB*6	59MB*6	62MB
Lidar: Resnet50-Backbone	266MB*3	87MB*3	90MB
Total	4.12GB	1.27GB	258MB

TABLE II: Memory usage of multi-model and multi-mode.

B. Evaluation

We did experiments on three models with two typical ADAS applications, Resnet50-Backbone vision model [12], Vgg16-Backbone [13] vision model and PointPilar-Backbone [14] lidar model. We implement three models on the same FPGA. The vision model owns different vision tasks like Lane Detection, Traffic Light Detection, Stop Line Detection and Front Detection. For simplicity, we abbreviate these tasks to LD, TLD, SLD, FCD.

From Table I, we can see that latency is the highest at full function mode (LD+TLD+SLD+FCD) while other modes have lower ones. The introduction of multi-mode allows us to call different modes in different frames, which can help reduce the overall latency in ADAS.

In addition, we also give the memory usage in Table II. Baseline 1 represents accelerating each network without branch partition ping-pong buffer and without multi-mode memory optimization mentioned in Section III-B. Baseline 2 represents accelerating each network with branch partition ping-pong buffer but without multi-mode memory optimization. Our means accelerating networks with both branch partition and multi-mode memory optimization. The results show that our overlay design can reduce memory usage from 4.12GB to 258MB.

V. Conclusion

Advanced Driver-Assistance Systems (ADAS) are complex systems consisting of many computer vision tasks, which requires multi-model and multi-mode support of accelerators. In this paper, we propose an FPGA-based neural network overlay processor for ADAS. The overlay architecture contains almost all common computation layers for learning based ADAS. In addition, we design a compiler that can automatically compile the high-level description of neural networks from deep learning framework like Caffe and Tensorflow into FPGA configurable codes, which can be executed by our overlay architecture without reprogramming. Experiments show that our overlay design can process learning tasks in ADAS with low latency and low memory usage.

Authorized licensed use limited to: IEEE Xplore. Downloaded on May 16,2021 at 08:27:40 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] S. S. BV and A. Karthikeyan, "Computer vision based advanced driver assistance system algorithms with optimization techniques-a review," in *International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018.

[2] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[3] H. K.-H. So and C. Liu, "FPGA overlays," in *FPGAs for Software Programmers*. Springer, 2016, pp. 285–305.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[5] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[6] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in *International Symposium on Circuits and Systems (ISCAS)*, 2017.

[7] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[8] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang, "DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 10, pp. 2668–2681, oct 2020.

[9] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling *et al.*, "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[10] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2019.

[11] T. Yang, Y. Liao, J. Shi, Y. Liang, N. Jing, and L. Jiang, "A winograd-based cnn accelerator with a fine-grained regular sparsity pattern," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 254–261.

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[13] A. Kirillov, R. Girshick, K. He, and P. Dollár, "Panoptic feature pyramid networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[14] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Authorized licensed use limited to: IEEE Xplore. Downloaded on May 16,2021 at 08:27:40 UTC from IEEE Xplore. Restrictions apply.