Performance Optimisation of Parallelized ADAS Applications in FPGA-GPU Heterogeneous Systems: A Case Study With Lane Detection

Xiebing Wang, Kai Huang, and Alois Knoll

***Abstract*—The explosive growth of massive data captured by various sensors on modern vehicles has impelled the deployment of Commercial Off-The-Shelf (COTS) accelerators for the research and development of Advanced Driver Assistance Systems (ADAS). Although the advent of cross-platform programming framework such as Open Computing Language (OpenCL) facilitates the programmability of ADAS applications on heterogeneous devices, the performance portability is still vulnerable and subject to different hardware implementations by the heterogeneous manufacturers. With this issue in mind, in this article we propose a detailed procedure that helps guide the performance optimisation of parallelized ADAS applications in an FPGA-GPU combined heterogeneous system. Taking two different lane detection applications as case studies, we provide one intra-accelerator and two inter-accelerator optimisation methods, as well as both FPGA-specific and application-oriented optimisation strategies, to boost the program runtime performance. Experiment results on a heterogeneous platform with COTS FPGA and GPU components reveal that the optimal designs generated from the procedure can improve the runtime performance of the two applications by an average of 109.21% and 83.48% over the native parallel implementations, respectively.**

***Index Terms*—ADAS, FPGA, GPU, OpenCL, lane detection.**

I. Introduction

GUARANTEEING real-time performance is crucial for state-of-the-art Advanced Driver Assistance Systems (ADAS) applications as it can provide as much time as possible for drivers to make better decisions in a relatively short time frame. This bound not only comes from the inherent time-criticality of ADAS tasks, but also stems from the demand to efficiently process the massive amount of data captured by the various types of sensors equipped in modern vehicles. In this context, it is inevitable that high-performance accelerators are used to promote the development and deployment of advanced applications in ADAS. For instance at the Consumer Electronics Show (CES) 2017, Intel announced the GO automotive 5G platform [1], which would incorporate Xeon Phi processors and Cyclone V Soc FPGA, to accelerate automotive computing. Such heavy-computing functional components would become standard equipment for future self-driving vehicles.

With the continuously emerging effort of using Commercial Off-The-Shelf (COTS) components for ADAS development [2]–[4], both Graphic Processing Unit (GPU) and Field Programmable Gate Array (FPGA) are ready to be deployed in automated driving systems, due to their significantly higher computational capacity and lower research and development cost compared to dedicated ECU/ASIC-implemented counterparts. Meanwhile, during the last decade the popular use of GPUs, FPGAs, and other co-processors in academia and industry has spawned the advent of generic standardization for cross-platform parallel programming such as Open Computing Language (OpenCL) [5]. On the basis of this framework, different accelerators can leverage their respective advantages to complete the computational tasks in a collaborative way. However, despite the seamless code portability, performance portability still cannot be guaranteed due to the miscellaneous hardware implementations by the respective manufacturers. Program developers always need to elaborately define and assign the workload on these heterogeneous platforms so as to gain the best possible performance benefit.

This article investigates several key factors that influence the performance gain when deploying parallelized ADAS applications in heterogeneous systems. State-of-the-art studies mainly focus on the implementation and optimisation of applications accelerated with a single type of hardware accelerator, such as using GPU [6]–[8] or FPGA [9]–[11]. We propose a detailed procedure that helps guide the performance optimisation of parallelized ADAS applications on a heterogeneous platform consisting of GPU, FPGA, and multi-core CPU.

Our work differs from state-of-the-art with the following aspects: ① In this work, we focus on the heterogeneous systems with multiple types of hardware accelerator, i.e. with both FPGA and GPU. Therefore, the intra-accelerator workload consumption, which is a different scenario from previous work, is studied carefully and two optimisation methods called accelerator execution overlapping and dynamic workload tuning are presented.

Manuscript received July 15, 2018; revised December 24, 2018; accepted March 21, 2019. Date of publication August 28, 2019; date of current version November 21, 2019. This work was supported in part by China Scholarship Council under Grant 201506270152 and in part by National Natural Science Foundation of China under Grant 61872393. *(Corresponding author: Kai Huang.)*

X. Wang and A. Knoll are with the Department of Informatics, Technical University of Munich, Garching 85748, Germany (e-mail: wangxie@in.tum.de; knoll@in.tum.de).

K. Huang is with the Key Laboratory of Machine Intelligence and Advanced Computing, Ministry of Education, and School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China (e-mail: huangk36@mail.sysu.edu.cn).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIV.2019.2938092

② We customize the applications with data-level parallelism and the workloads assigned to different accelerators are identical. In this way, the comparison of the computational capacity of each accelerator is fair and intuitive. ③ We provide a detailed procedure that contains various approaches to optimise a native parallelized application in a fine-grained manner. Therefore, this procedure applies to any OpenCL application that is developed in the early-design stage and intended to be executed in such an FPGA-GPU heterogeneous system.

This article presents a substantial extended work of our previous study in [12]. We also provide a customized heterogeneous design of the work in [13] and thereupon apply the proposed procedure to obtain the optimal execution. We use two different Lane Detection Algorithms (LDA) as case studies and both of them are programmed with OpenCL for ease of use on a single GPU or FPGA device, but not fully optimised in a system combining both accelerators. The first application [14] customizes an FPGA-GPU combined implementation of particle-filter-based lane detection and tracking, while the second application [13] detects lane markings via the RANdom SAmple Consensus (RANSAC) approach. For brevity, we use p-*LDA* and r-*LDA* to respectively refer to the aforesaid Particle-filter-based and RANSAC-based Lane Detection Applications in latter sections.

After the identification of the performance bottlenecks in the target program, one intra-accelerator and two inter-accelerator sub-optimisation methods are taken into consideration so as to increase the task processing efficiency. Moreover, application-oriented optimisation of the workload is conducted to further improve the overall runtime performance. Experiment results reveal that for p-*LDA* and r-*LDA*, the optimal designs generated from the procedure can achieve performance gains with an average speedup of 2.09× and 1.83× over the native parallel implementations, respectively.

The remainder of this article is organized as follows: Section II is related work and Section III gives the background of the lane detection applications. Section IV illustrates in detail the proposed procedure and the sub-optimisation methods. Section V presents experimental results and Section VI concludes.

II. Related Work

A. Lane Detection Techniques

State-of-the-art methods for lane detection can be classified into two categories: camera-based methods with an image stream as input and multi-sensor-based methods combining camera and a minimum of one additional source for environmental perception such as LIght Detection And Ranging (LIDAR) [15], [16], Global Positioning System (GPS) [17], or other data receivers. The applications in this article adopt camera-based methods and therefore we mainly review the approaches which only use cameras as source of information.

In general, camera-based approaches are characterized by an execution procedure similar to that shown in Fig. 1. The Region Of Interest (ROI) in which the lane markings are located is either the whole raw image [7], [18], [19] or cropped from the captured image via manual boundary setting [2], [20] or dynamical lane area calculation [21], [22]. With the ROI defined,

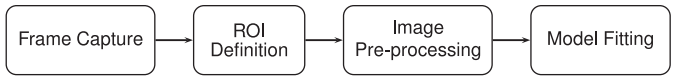


Fig. 1. General process flow of camera-based lane detection.

further image processing can be performed on either the raw image directly or a Bird's-Eye-View (BEV) image which is transformed via Inverse Perspective Mapping (IPM) [18], [20], [21] or Warp Perspective Mapping (WPM) [23], [24]. In contrast to IPM which uses intrinsic and extrinsic camera parameters to calculate the required transformation matrix, the WPM method is independent from the camera parameters. However, for WPM a minimum of four reference points in the original image as well as the transformed image are required to compute an affine matrix mapping. The models used to fit the candidate lanes generated from the image pre-processing step are miscellaneous. Some studies use the RANSAC model to conduct line fitting [7], [20], [22], while other research adopts filtering techniques to implement lane tracking [2], [21], [25], [26]. The biggest advantage of RANSAC-based lane fitting is the robust estimation of model parameters, even if the data set contains a significant amount of outliers [27]. However, the benefit of filter-based lane tracking is a considerable reduction of time consumption caused by the iterative lane detection.

B. ADAS Applications on Heterogeneous Platforms

Lane detection is mostly achieved via filtering techniques to capture lanes, however it is rarely deployed on heterogeneous platforms. In this article, we focus on the optimisation of OpenCL-based lane detection applications. The performance portability of OpenCL applications across different platforms remains an open problem. To solve this issue, some researchers have proposed profiling and optimisation framework to assist better development of OpenCL applications. The work in [28] provided a generic tool for performance measurement of OpenCL programs. In [29], the authors proposed a framework combining OpenCL application auto-tuning and runtime resource management. The study in [30] presented a transparent OpenCL overlay called Helium, for inter- and intra-kernel optimisation. The studies mentioned above are not yet mature and to the best of our knowledge, state-of-the-art research remains at the stage that optimisations are highly dependent on the specific algorithm, architecture, and programming features. In [31], the authors analyzed and profiled the components of the Speeded Up Robust Features (SURF) algorithm. Their work only involved the profiling of the program and this information can be referenced for performance improvement. Recently, FPGA devices are mainly used as the accelerator for Convolutional Neural Network (CNN) like the work in [32] and [33]. In their work, optimisations were mainly performed based on the CNN algorithm itself.

While there exist substantial efforts to parallelize ADAS applications with GPU [6]–[8] or FPGA [9], [10], [11], [34], [35], few studies are reported to accelerate them with heterogeneous commodity hardware [36]. The study in [2] investigated the

feasibility of using COTS hardware for ADAS development, but the performance optimisation is not considered. The work in [6] presented a step-by-step optimisation of face detection algorithm in CPU-GPU heterogeneous systems. However, this study considered only the CPU-GPU heterogeneous architecture. Authors in [37] compared the performance of using GPU, FPGA, or both devices to accelerate pedestrian detection applications. Unlike the data-level parallel designs in our work, in [37] GPU and FPGA process different tasks to fulfil task-level parallelism. In [38], the authors exploited FPGA to accelerate a speed-limit-sign recognition application and showcased the performance and energy results compared with the GPU-implemented counterpart. Their work does not involve both FPGA and GPU, and the optimisation part is also limited to FPGA only. The most related work to this article is [6], where authors used optimisation methods including CPU execution time hidden, memory coalescing and variable parallel granularity. The difference of our work is that rather than using a single GPU, we tested the FPGA-GPU heterogeneous context so that (i) the execution time of both accelerators can also be hidden via changing build-in function order, (ii) parallelism on FPGA side could be further adjusted by using pragma primitives, and (iii) other optimisation methods like dynamical workload tuning are also presented.

III. Background

In this section we give a brief review of the two case study applications and then explain how they are parallelized on the FPGA-GPU heterogeneous platform.

A. p-LDA: Particle-Filter-Based Lane Detection Application

1) Algorithm Overview: This application mainly consists of three modules, namely *pre-processing*, *lane detection*, and *lane tracking*. For each frame, the *pre-processing* module extracts information about the lane markings and then passes it to the next step. Depending on whether or not the estimated state in previous frame can still be applied to current frame, the image is processed either using the *lane detection* module to redetect the positions of the lane markings or using the *lane tracking* module to track the previous position of the lane markings.

The *pre-processing* module crops the ROI from the raw image and then transforms it into a grayscale format where each pixel reflects the intensity of the pixel in the original image. To enhance the contrast of the pixel intensity, a Sobel filter [39] is applied to the grayscaled image to extract transitions and edges. To avoid the influence of noises, a threshold is used to tune the intensities of all pixels in the image.

During *lane detection*, first of all a set of *candidate lines* $\mathbb{X} = \{X_1, X_2, \ldots, X_n\}$ is randomly generated via assigning random values from a normal distribution, where n is the number of the candidate lines. For each candidate line X_i, a weight w_i is used to reveal how close the line is located to the real lane. The line with the highest weight is chosen as the *best line* and certain number of candidate lines are reserved as *good lines*, which is further used in the *lane tracking* module.

For *lane tracking*, a particle filter [40] is adopted to predict the lane markings, using both the ROI of the current frame and the *best line* and *good lines* of the previous frame. The particle filter consists of three steps: (i) the *prediction update* step modifies previous *good lines* as the prior probability distribution of lane markings in the current frame; (ii) the *importance weight update* step recalculates the weights of the particles; and (iii) the *resampling* step selects particles from the newly updated set so as to prevent particle set degeneration.

Finally the redetection checking step verifies whether the detected positions reasonably conform to the physical properties of the lane markings. If not, additional detection step is triggered to seek the lane markings again.

2) Parallel and Heterogeneous Design: In the native design, each of the above three modules is programmed as an OpenCL kernel. For the sake of brevity, we use `kernelPRE`, `kernelLD`, and `kernelPF` to refer to them respectively. Additionally, a random number generator is used to provide normally distributed random numbers for both lane detection and lane tracking tasks. This kernel, termed as `kernelRNG`, is executed only once, while the remaining kernels are executed frame by frame.

As for the heterogeneous design, the computation workloads of `kernelLD` and `kernelPF` are split and partially executed on FPGA and GPU, while the `kernelPRE` task is consumed on both devices, because of data dependency, in that lane detection and lane tracking relies on the calculated results of the pre-processing step. Fig. 2 depicts the parallel design of lane detection and lane tracking modules when these tasks are executed on a single FPGA/GPU (Fig. 2(a)) or both accelerators (Fig. 2(b)). In the heterogeneous design, the lines are sampled on each device (m lines generated on FPGA and n lines generated on GPU as shown in Fig. 2(b)) and then collected to calculate the *best line* from the merged outcome ($m + n$ lines).

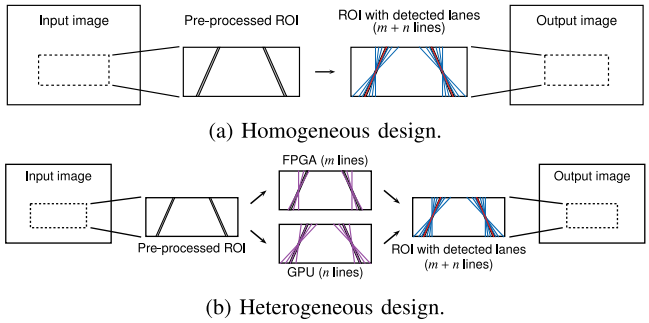


Fig. 2. Parallel design of lane detection and lane tracking modules in *p-LDA*. Here the lines refer to *candidate lines* and *good lines*, respectively for lane detection and lane tracking tasks. The red line represents the *best line*.

B. r-LDA: Ransac-Based Lane Detection Application

1) Algorithm Overview: This application also contains three steps, namely *homography matrix calculation*, *pre-processing*, and *model fitting*. The *homography matrix calculation* step is performed off-line only once, which uses a reference frame to generate the homography matrix. The *pre-processing* and *model fitting* steps are performed iteratively for each frame to extract the detected lane positions in the original input image.

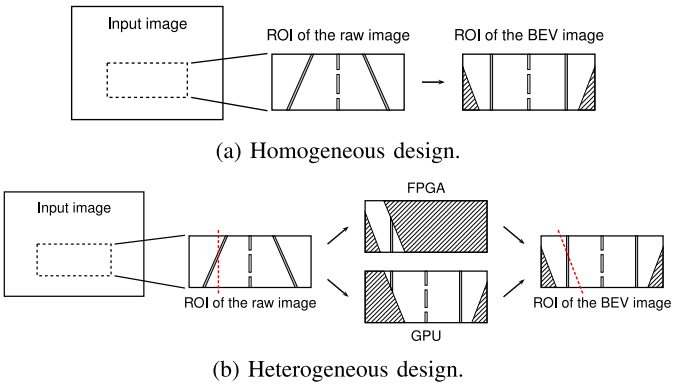


Fig. 3. Parallel design of the WPM transformation in r-*LDA*. The areas with slashes represent empty values of the pixels.

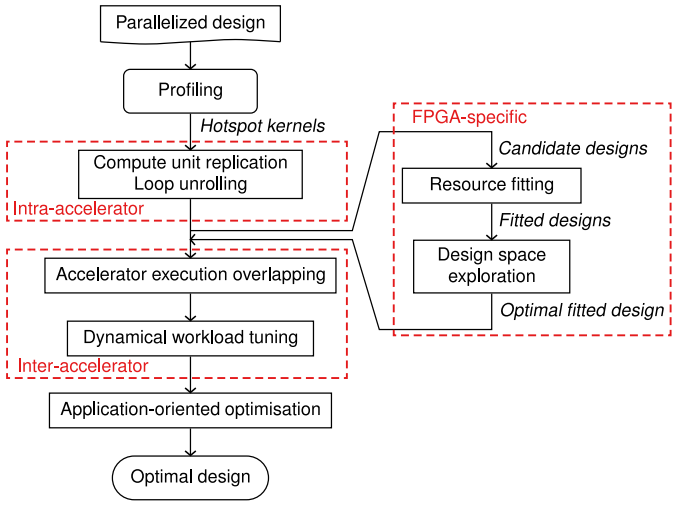


Fig. 4. Optimisation procedure for parallelized ADAS applications in FPGA-GPU heterogeneous platform. The three red dashed boxes indicate the intra-accelerator, inter-accelerator, and FPGA-specific optimisation module, respectively.

The *homography matrix calculation* is implemented via vanishing point estimation [41] and top-view mapping of a reference frame in the input video stream. The feature points are obtained with ROI bounding to estimate their corresponding points in the BEV image. Then the relations between the matching points are constrained by the underlying assumptions to compute the required homography mapping.

The *pre-processing* step first generates a WPM-based BEV image of a pre-defined, fixed ROI inside the input frame and then filters the image with a second derivative Gaussian filter. Afterwards this top-view ROI is grayscaled and convoluted to highlight the vertical as well as quasi-vertical lane markings in the original ROI.

The *model fitting* step applies a simplified Hough transformation to give an initial guess about the positions of the lane markings and afterwards performs both linear and spline RANSAC fitting to locate the matched splines in the original input image. The first step matches linear lines using previous Hough transformation data and the second RANSAC iteration fits each lane to a third-degree Bezier spline after applying proper geometric checks.

2) Parallel and Heterogeneous Design: In the native design [13], the *homography matrix calculation* is executed off-line only once and therefore is not parallelized. The *pre-processing* step is characterized as two OpenCL kernels, i.e., `kernelWPM` and `kernelCONV`, which consume the WPM and image convolution tasks respectively. The *model fitting* step is performed on the host since its time cost is much smaller than that of the *pre-processing* step.

In this work we develop a customized heterogeneous execution with data-level parallelism, since both `kernelWPM` and `kernelCONV` process the pixels in the ROI independently, and therefore during calculation there is no intra-pixel data dependency. The heterogeneous design vertically divides the ROI into two parts, of which either one is taken as an input workload on one accelerator. Taking WPM as example, Fig. 3 presents its parallel design in the homogeneous (Fig. 3(a)) and heterogeneous (Fig. 3(b)) execution scenarios. In the heterogeneous implementation, FPGA and GPU individually transform part of the raw ROI and then piece together the results into the BEV image. It is the same case with the image convolution kernel.

IV. Optimisation Procedure

Figure 4 exhibits the proposed optimisation procedure. First of all, profiling of the application is needed to locate the hotspot kernels so that the bottleneck can be identified and further optimised. Given a native design of the hotspot kernels, generic optimisation is utilized on the intra-accelerator side. These optimisations include loop unrolling, memory access coalescing, global memory access elimination, etc. Particularly for the FPGA, since each OpenCL kernel is abstracted as a compute unit and further hardware synthesized as a dedicated circuit block on the board, this compute unit can be replicated multiple times to increase the processing efficiency. With a set of kernel configurations that indicate different combinations of compute unit replication and loop unrolling factors, a resource check is conducted at pre-compilation stage to examine whether the candidate designs can actually meet the on-board resource limitation. The designs that consume more ALMs, registers, RAM blocks, and DSP blocks than the maximal available number of the counterparts are discarded and the remaining designs are passed to the design space exploration module to obtain the optimal kernel design on FPGA.

The inter-accelerator sub-optimisation consists of two steps. On the host, the invocation order of the kernel functions for different accelerators can be interleaved to hide the kernel launch, host-device data transfer, and kernel execution overhead. Meanwhile, during the processing of each frame, the workloads for FPGA and GPU are dynamically tuned so as to balance the time consumption.

Finally, each application has its self-defined workload such as the ROI definition in p-*LDA* and the RANSAC iteration in r-*LDA*. This workload is flexible and can also be regulated to enhance the runtime performance.

A. Profiling

To figure out the execution time distribution of the program, the high-level source code is segmented into several blocks and the execution time of each block is subsequently measured. The executions of these code blocks express the skeleton of the whole program. For each code block, time stamps are inserted before and after the execution of the code and the proportion of time cost in the total time consumption is calculated after each run. In theory, the code block that consumes the most part of the total execution time is optimised with top priority.

B. Compute Unit Replication (CR)

In OpenCL, high-level source code of the kernel is instantiated as a work item running on a compute unit where a group of work items can execute simultaneously to accelerate the applications. For GPU, this compute unit is mapped to a stream multi-processing unit and therefore its implementation is hardware-dependent and its optimisation is beyond the scope of this article. While on the FPGA platform, the compute unit is hardware-implemented as a circuit block and by assigning more compute units, the performance can be enhanced to a large margin as long as the peak computation capacity and resource utilization are not reached. The kernel compute unit replication increases the data throughput at the expense of memory bandwidth contention among the compute units.

We perform the compute unit replication as follows: first we assume the most simplified configuration, i.e., setting only one compute unit for each kernel, to guarantee that the design meets the board resource limitation. Afterwards, the kernels is optimised in a step-by-step manner by assigning more and more compute units to the most time-consuming task load.

C. Loop Unrolling and Memory Access Coalescing (LU)

Loop unrolling is a code transformation technique used to reduce the program's execution time at the expense of its binary size, which is known as the space-time tradeoff. By unwinding the loop code several times, the control statements are reduced or avoided so that the number of branches are minimized. On the GPU side, loop unrolling is implemented by manually replacing the loop with repeated sequential statements which eliminates the branch penalty. Loop unrolling on the FPGA board increases the length of pipeline, thus overlapping the executions of more logic units. Similar to compute unit replication, loop unrolling on FPGA provides a trade-off between the potential higher performance, due to the hidden pipeline execution time, and more intense memory bandwidth contention, due to larger resource exploitation. On both platforms, the expansion of loop size coalesces memory access as long as they have adjacent memory addresses.

Additionally, the expensive global memory access in some kernels is eliminated by using pre-computed values to replace memory-reading operations with data manipulation of a fixed constant rather than a global variable.

D. Accelerator Execution Overlapping (EO)

As illustrated in Section III-A2 and Section III-B2, the heterogeneous implementation is data-level parallel and each hotspot kernel is executed on both FPGA and GPU platforms. Here, there is a trade-off of how and when the kernels are invoked from the host. In general, an overall execution of an OpenCL kernel can be abstracted as the following flow: ① The input data is stored in a host buffer and then written into device memory via the function `clEnqueueWriteBuffer`. ② The kernel is driven via the function `clEnqueueNDRangeKernel` to the command queue and is ready for execution. ③ The output data is generated after the kernel is completed and the results are read back from the device to the host via function call of `clEnqueueReadBuffer`. ④ The data on the device and the host is synchronized over and ready for future use, which is notified by the completion of corresponding kernel events via the function `clWaitForEvents`. In the heterogeneous design, both FPGA and GPU would call their own OpenCL runtime libraries to execute the API functions mentioned above. The call order of these functions should be carefully considered since not all of them are non-blocking invocations.

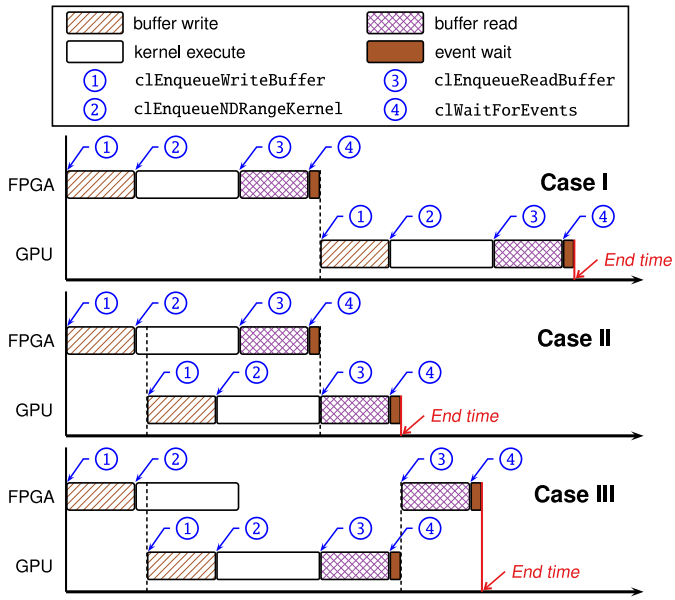


Fig. 5. A sample heterogeneous execution with different call order of OpenCL API functions for FPGA and GPU. The blue arrows indicate the exact time point when the API function in the blue circle would actually take effect.

Figure 5 gives an example to illustrate how the call order of the API functions influences the time cost of executing the kernel on FPGA and GPU. On each platform, the kernel is processed with the same function call order as the aforementioned flow, while the sequence of the functions that the host invokes for different platforms may vary. In Case I, all the invocations of FPGA-related API functions are before the GPU-related counterparts as if the kernel is sequentially processed one after another on the platforms. In this case, the first GPU-related API function is invoked after the last FPGA-related API function is served and the total time cost is the sum of the execution time on FPGA and GPU. In Case II, the first GPU-related API function is invoked immediately after the call of the FPGA-related function `clEnqueueNDRangeKernel`, with a subtle lag. The data read back function `clEnqueueReadBuffer`

on GPU is called later than the completion of the kernel on FPGA. Therefore, the termination of the GPU-related function `clWaitForEvents` indicates the end time point of the total execution. In this case, both the kernel execution and host-device data transfer time are well overlapped. As shown in the last case, the data read back part on FPGA is executed after the GPU kernel is processed and the total execution time is longer than that in Case II, since the host-device data transfer time is not hidden. As can be seen, calling FPGA- and GPU-related API functions in an interleaved way can overlap the inter-accelerator kernel execution and host-device data transfer time and consequently boost the runtime performance.

E. Dynamical Workload Tuning (DT)

For ADAS applications, the input data is normally from a captured road video stream and the program needs to process the video frame by frame to extract effective environmental information so as to assist drivers with decision making. These applications can be lane detection, pedestrian detection, traffic sign recognition, vehicle identification, etc. Consequently, the hardware accelerators need to process the workload of every image frame repeatedly, which offers the possibility of dynamically tuning the workloads among different platforms.

As FPGA and GPU show distinct computation capacities in consideration of different types of data manipulations, much like [14], we apply a dynamical tuning of the workload to ensure that the tasks can be finished within the shortest possible time. The basic idea of the dynamical workload tuning is that the workload to be assigned on a certain device should be proportional to its computation throughput. Therefore during each processing iteration of the image frame, the total workload is re-assigned to the involved accelerators based on their historical computation capacities.

Assume there are in total N accelerators in the system and in the previous iteration the i-th accelerator consumes an amount of workload W_i at the expense of time T_i, then the newly assigned workload W_i' for the current iteration can be calculated as

$$W_i' = \frac{c_i}{\sum_{i=1}^{N} c_i} \sum_{i=1}^{N} W_i$$
$$c_i = \frac{W_i}{T_i} \tag{1}$$

where c_i indicates the computation throughput of the i-th accelerator in the previous iteration. The newly assigned workload is a portion of the total workload, where the coefficient is calculated as the ratio of the computation throughput of the i-th accelerator to the computation throughput of all the accelerators in the system.

F. Application-Oriented Optimisation (AO)

Theoretically speaking, the performance can be improved as long as the amount of the total workload can be reduced while guaranteeing the accuracy of the final results. In the two case study applications, the tunable workload lies in the ROI definition and the RANSAC iteration, respectively.

Algorithm 1: ROI Tuning Scheme for *p-LDA*.

Input: Best line set $\mathbb{B}$, $imgWidth$, $initRoiStart$, $initRoiEnd$, $roiStart$, $roiWidth$
Output: $roiStartAdapted$, $roiWidthAdapted$

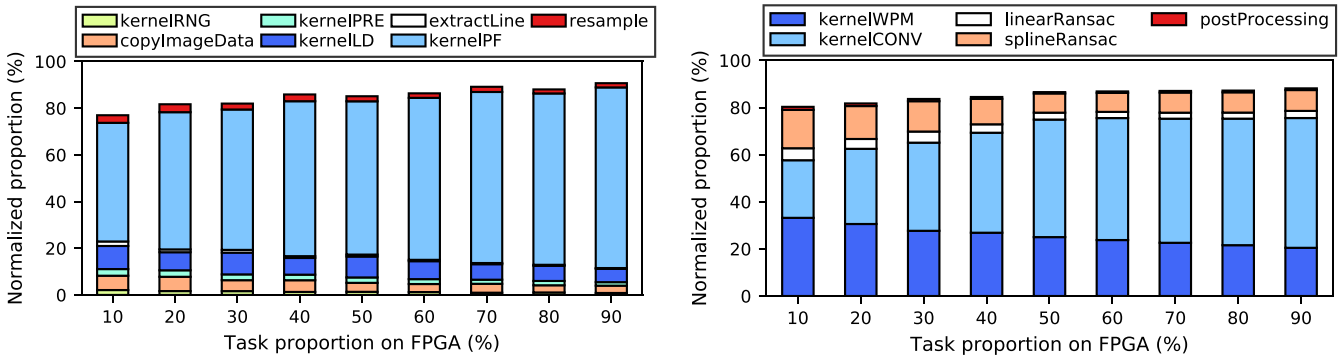
$T_{left} \leftarrow 1/4$, $T_{right} \leftarrow 3/4$
$roiStartAdapted \leftarrow roiStart$
$roiEndAdapted \leftarrow roiStart + roiWidth$
foreach *best line* $\mathtt{b} \in \mathbb{B}$ **do**
  $roiStartAdapted \leftarrow min\langle roiStartAdapted, \mathtt{b.start}\rangle$
  $roiEndAdapted \leftarrow max\langle roiEndAdapted, \mathtt{b.end}\rangle$
if $roiStartAdapted < initRoiStart$ **then**
  $roiStartAdapted \leftarrow initRoiStart$
else if $roiStartAdapted > imgWidth \times T_{left}$ **then**
  $roiStartAdapted \leftarrow imgWidth \times T_{left}$
if $roiEndAdapted > initRoiEnd$ **then**
  $roiEndAdapted \leftarrow initRoiEnd$
else if $roiEndAdapted < imgWidth \times T_{right}$ **then**
  $roiEndAdapted \leftarrow imgWidth \times T_{right}$
$roiWidthAdapted \leftarrow roiEndAdapted - roiStartAdapted$
if *redetection* **then**
  $roiStartAdapted \leftarrow initRoiStart$
  $roiWidthAdapted \leftarrow initRoiEnd - initRoiStart$

1) p-LDA: As described in Section III-A1, only the image ROI is processed and information of pixels falling in this area is further computed. Therefore decreasing the ROI size could distinctly shrink the calculation task load and improve the performance. For this application, our optimisation enables an adaptive ROI when processing the image frames iteratively.

Algorithm 1 gives the detail of the ROI tuning scheme for *p-LDA*. First the best line set $\mathbb{B}$, which contains the lane positions of the current frame, is traversed to get the minimal and maximal x-axis coordinates of the best lines. These two coordinates are seen as the candidate start and end x-axis positions of the updated ROI. Then the updated ROI is upper-bounded by the start and end x-axis positions of the initial ROI and lower-bounded by a certain proportion of the image width (here the coefficients of proportionality are set as 1/4 and 3/4). If the redetection step is triggered, the width of the ROI is reset as the initial ROI width. This scheme ensures that the computation workload of each frame is no more than that using the initial ROI and no less than that using a region of which the width equals only half of the image width.

Note that here we focus on the regulation of the ROI width, rather than the ROI height, since the ROI height is normally fixed within a visible area of the lane markings. In addition, the coefficients of proportionality of the lower-bounded ROI are empirically set. This is to ensure that the ROI size would not collapse from a plane to a line when the detected lanes are too close, which would prevent the ROI construction and further ruin the detected results. From Algorithm 1, it is seen that the ROI of the next frame is bounded by the positions of the lanes detected in the current frame. This indicates that the processed ROI is not subject to user interference and the detection accuracy does not suffer due to an incomplete ROI.

2) r-LDA: In this application the processed ROI is already adaptively bounded, based on the position of the estimated vanishing point. This drives us to turn to the optimisation of the model fitting part. As is known, the number of RANSAC



(a) *p-LDA* (b) *r-LDA*

Fig. 6. Normalized execution time distribution of the profiled code blocks in *p-LDA* and *r-LDA*.

iterations N is determined by

$$N = \frac{\log(1-\rho)}{\log(1-\omega^{\eta})} \tag{2}$$

where ρ is the probability that the best fitting model can be found, η is the minimal number of data points needed to define the model and ω is the probability that any selected data point is within the error tolerance of the model [27]. Since ω is preliminarily unknown in r-LDA, during the evaluation we first set N as a considerably large value and then gradually reduce this value until the accuracy hits a tolerable threshold. In this way, the number of RANSAC iterations is minimized while the accuracy is still guaranteed.

G. Discussion

The aforementioned optimisation methods constitute a systematic procedure for improving the performance of OpenCL-based ADAS applications in FPGA-GPU heterogeneous systems. Compared with state-of-the-art, this work targets a different scenario, i.e., the performance optimisation of lane detection applications when different hardware accelerators are involved. Apart from the conventional optimisation techniques, we also take into consideration the intra-accelerator kernel execution and give optimisation methods such as accelerator execution overlapping and dynamic workload tuning.

V. Evaluation and Analysis

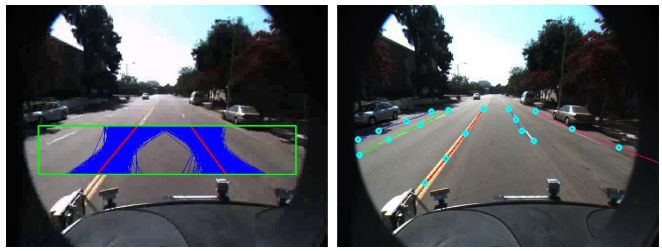
A. Evaluation Setup

We use a heterogeneous system consisting of multi-core CPU, GPU, and FPGA as the evaluation platform. The details about the hardware specification are shown in Table I. To evaluate the performance of the two case study applications, we utilize the benchmark videos from Caltech data set [42]. This data set consists of four clips on various urban street scenarios including straight and curved lanes, shadows, reflections, and street scenes to reflect real-world conditions.

During the runtime evaluation of *p-LDA*, we use 2^{12} *good lines* and 2^{13} *candidate lines* to detect 2 lane markings. As for the parameters of *r-LDA*, we set the initial value of the number of RANSAC iterations to 300 and the observed optimal value is 40. Each time the FPGA device is assigned the workload with different proportions, i.e. from 10% to 90% and vice versa, the task proportion on the GPU is from 90% to 10%, with a step of 10%. Each video is run multiple times and the overall results are finally collected and averaged.

TABLE I
Detailed Specification of the Evaluation Platform

Platform	Information	
Host CPU	Intel Core i5-3360M @ 2.80GHz 2 Cores	
Device	FPGA	GPU
Model	Nallatech 385	Quadro K600
Architecture	Stratix V GS	Kepler GK
OpenCL SDK version	Intel FPGA SDK 13.1	Nvidia CUDA 8.0
Peak GFLOPS	294.7	336.4



(a) *p-LDA* (b) *r-LDA*

Fig. 7. Detected lane results of the two applications.

B. Detection Results

Figure 7 presents the detection results of the two applications. As can be seen, *r-LDA* detects more lanes than *p-LDA*, since the expected number of detected lanes of *p-LDA* is preset to 2. In this work, we mainly focus on the performance optimisation and the accuracy results are demonstrated in the previous studies. Note that our performance optimisation does not significantly influence the accuracy of these two applications. Details about the accuracy results of *p-LDA* and *r-LDA* are reported in [2] and [13], respectively.

TABLE II
LIST OF MAIN CODE BLOCKS IN *p-LDA*

No.	Name	Function Description
1	kernelRNG	random number generation
2	copyImageData	copy image matrix data into array
3	kernelPRE	pre-processing of raw image ROI
4	kernelLD	lane detection
5	extractLine	extract good and best lines
6	kernelPF	lane tracking
7	resample	particles resampling

TABLE III
LIST OF MAIN CODE BLOCKS IN *r-LDA*

No.	Name	Function Description
1	kernelWPM	warp perspective mapping
2	kernelCONV	image convolution
3	linearRansac	linear RANSAC fitting
4	splineRansac	spline RANSAC fitting
5	postProcessing	post-processing

C. Profiling Results

1) p-LDA: This application is segmented into 7 main code blocks and the detailed description of them is shown in Table II. Fig. 6(a) reveals the normalized execution time of these code blocks when the workloads are distributed to FPGA and GPU with different proportions. As observed in Fig. 6(a), the time consumption of kernelLD and kernelPF accounts for a minimum of 60.71% (when the FPGA task proportion is 10%) and a maximum of 83.08% (when the FPGA task proportion is 90%) of the total execution time. These two kernels are therefore the hotspot kernels and need further optimisation. Note that the execution of copyImageData also consumes a considerable amount of time. This is inevitable since the raw image data have to be read into the host memory before the pre-processing. One possible optimisation of this code block is to reduce the transmitted data size, which is done by the ROI tuning scheme.

2) r-LDA: This application contains 5 main functional modules of which the detailed information is listed in Table III and their respective execution time distribution is shown in Fig. 6(b). kernelWPM and kernelCONV are deemed as hotspot kernels as they occupy the majority part of the total time consumption (from 57.63% when the FPGA task proportion is 10% to 75.54% when the FPGA task proportion is 90%). Aside from them, the splineRansac task dominates the remainder of the time cost. Optimisation of this part is done by reducing the number of RANSAC iterations.

D. Optimisation Results

1) Compute unit Replication (CR): In our design, the code snippets of kernelLD and kernelPF in *p-LDA* are within the same OpenCL kernel function, due to their similar functionalities with minor differences, and consequently they are always replicated with the same number of compute units. As for *r-LDA*, kernelWPM and kernelCONV belong to two separate kernel functions and hence they can be replicated with different configurations. For brevity, we use λ_{CR} to denote the factor that a compute unite is replicated in the FPGA design, and $\widehat{\lambda}_{CR}$ to denote the maximum number that a compute unit can be replicated subject to a given specific constraint. The first, third, and fourth columns in Table IV exhibit the detailed CR configurations of the two applications. Here the fourth column ($\widehat{\lambda}_{CR}^{conv}$) gives the maximal compute unit replication factor for kernelCONV when kernelWPM is replicated with the factor given in the third column (λ_{CR}^{wpm}). It is seen that fewer resources can be assigned to kernelCONV when kernelWPM is replicated an increasing number of times.

TABLE IV
CR AND LU CONFIGURATIONS OF THE FPGA DESIGN

p-LDA		*r-LDA*		
λ_{CR}	$\widehat{\lambda}_{LU}$	λ_{CR}^{wpm}	$\widehat{\lambda}_{CR}^{conv}$	$\widehat{\lambda}_{LU}$
1	9	1	7	16
2	3	2	7	16
3	1	3	7	16
-	-	4	7	16
-	-	5	6	16
-	-	6	4	16
-	-	7	3	16
-	-	8	2	16
-	-	9	1	16

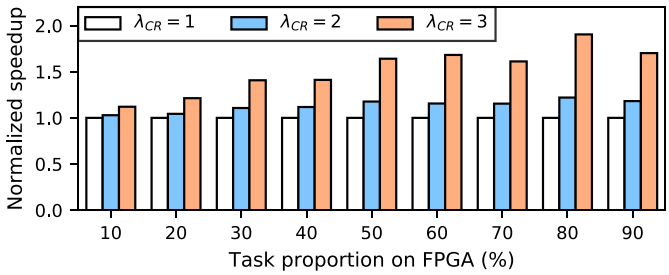
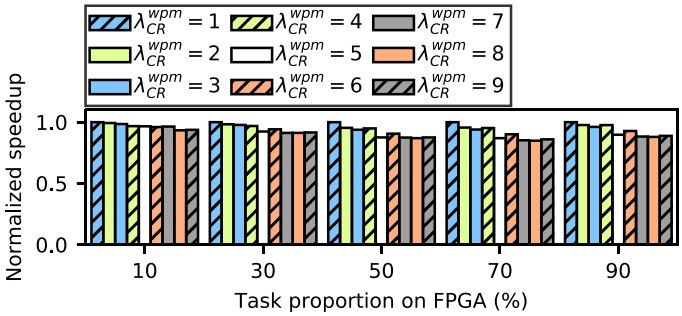


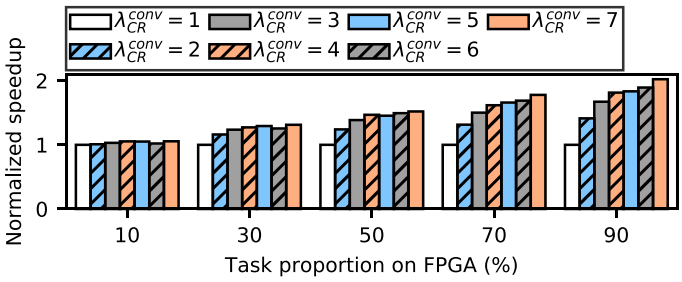
Fig. 8. Performance comparison of *p-LDA* when replicating different number of compute units.

For *p-LDA*, the maximum CR factor is 3 and Fig. 8 shows the performance results. As can be seen, for all task proportion scenarios, replicating the compute unit can boost the runtime performance. This speedup becomes larger when λ_{CR} increases. The average speedup is 1.13× and 1.52×, when the compute unit is replicated 2 and 3 times, respectively.

From Table IV, the maximum CR factor for kernelWPM is 9 and kernelCONV can be replicated up to 7 times when λ_{CR}^{wpm} is no greater than 4. For clarity of description, we compared the performance results of CR optimisation for *r-LDA* via the control variable method, i.e. varying either λ_{CR}^{wpm} or λ_{CR}^{conv} while setting the other one as a constant. Fig. 9 gives the detailed comparison and due to space limitations, we have only shown the results when the FPGA task proportion is 10%, 30%, 50%, 70%, and 90%. An interesting point shown in Fig. 9(a) is that merely replicating kernelWPM actually degrades the runtime performance. This slowdown becomes larger when λ_{CR}^{wpm} gradually increases. The observed worst performance loss is 15.15% when the FPGA task proportion is 70% and $\lambda_{CR}^{wpm} = 8$. A possible explanation is that kernelWPM is memory-operation dominant and therefore creating multiple instances of this kernel aggravates the on-board



(a) Replication of `kernelWPM` when $\lambda_{CR}^{conv}=1$.



(b) Replication of `kernelCONV` when $\lambda_{CR}^{wpm}=1$.

Fig. 9. Performance comparison of *r-LDA* when replicating different number of compute units.

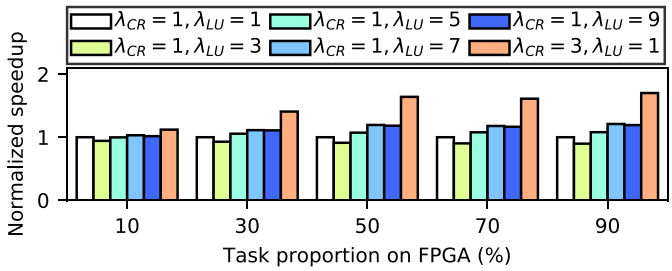


Fig. 10. Performance comparison of *p-LDA* using different loop unrolling factors.

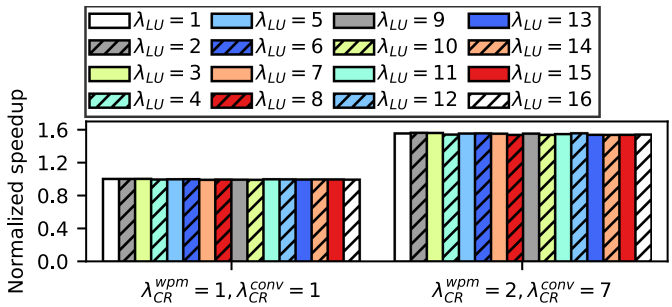


Fig. 11. Performance comparison of *r-LDA* using different loop unrolling factors, the FPGA task proportion is 50%.

memory bandwidth contention, incurring a larger performance penalty over the benefit of computation scalability.

For `kernelCONV`, replicating this kernel can be expected to result in much more performance benefit, and this gain becomes even larger when more workloads are allocated on FPGA, as is clearly shown in Fig. 9(b). The observed maximum speedup is 2.02× when 90% of the tasks are processed on FPGA, with `kernelCONV` replicated 7 times.

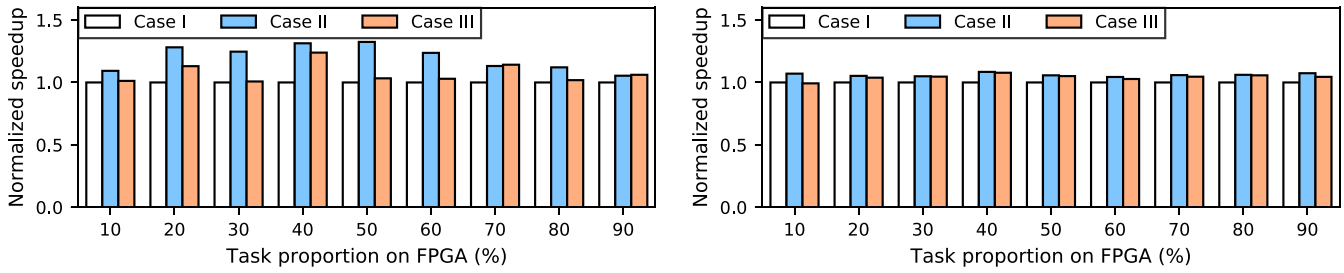
The combination of the CR optimisations of `kernelWPM` and `kernelCONV` is a trade-off between the computation- and memory-intensiveness of these two kernels. In our evaluation, the exhaustive exploration of all the fitted CR designs shows that the best case exists when $\lambda_{CR}^{wpm}=2$ and $\lambda_{CR}^{conv}=7$, which exhibits an average 1.55× speedup.

2) Loop Unrolling and Memory Access Coalescing (LU): To showcase the influence of loop unrolling on the runtime performance, we have searched through the fitted design space to obtain the available LU configurations. For brevity, we use λ_{LU} to denote the factor that a loop is unrolled, and $\widehat{\lambda}_{LU}$ to denote the maximum number that a loop can be unrolled subject to compiler setting and resource constraint. The second and fifth columns in Table IV give the LU configurations for the two applications, given the pre-determined CR settings of the kernels. For *p-LDA*, loop unrolling works on both `kernelLD` and `kernelPF` since they share the same code snippet. With regards to *r-LDA*, LU optimisation is only valid for `kernelCONV` as only this kernel contains a loop.

For *p-LDA*, $\widehat{\lambda}_{LU}$ decreases when the kernel is replicated more times. The loop can be unrolled 9 times when $\lambda_{CR}=1$, while no LU optimisation can be performed ($\widehat{\lambda}_{LU}=1$) when replicating the compute unit three times. Fig. 10 reveals the results of LU optimisation when $\lambda_{CR}=1$ and a similar trend is also observed in other cases. By way of contrast, we also show the performance result when $\lambda_{CR}=3,\lambda_{LU}=1$, so as to compare the performance boost of CR and LU optimisations. As seen in Fig. 10, the *p-LDA* application can only gain a subtle performance benefit when the loop is unrolled 5 or more times. However, this performance gain cannot rival the counterpart from the CR optimisation, as seen from the last bar in Fig. 10. For this application, CR optimisation gains a larger performance improvement than LU optimisation.

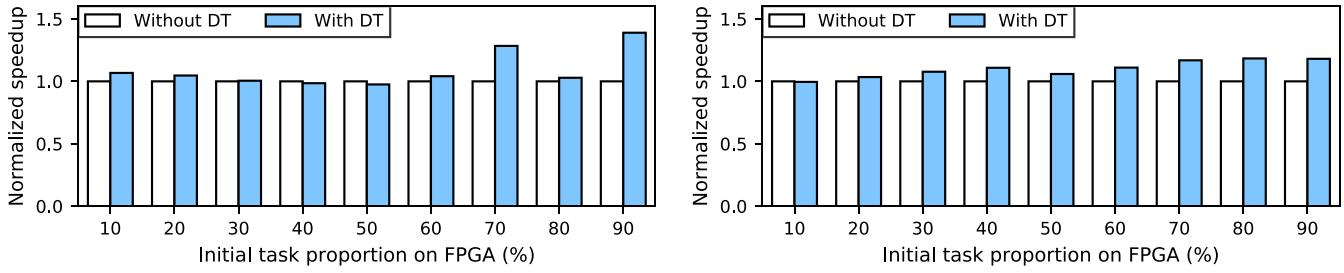
Figure 11 depicts the LU optimisation results for *r-LDA*. Due to space limitations, we have only shown the results when the FPGA task proportion is 50% and other cases turn out similar results. For comparison of CR and LU optimisation, we chose the native ($\lambda_{CR}^{wpm}=1,\lambda_{CR}^{conv}=1$) and the best ($\lambda_{CR}^{wpm}=2,\lambda_{CR}^{conv}=7$) CR configurations and exhibit their corresponding LU optimisation results. As observed from Fig. 11, loop unrolling results in nearly no performance gain when the CR factors are determined. This phenomenon is also demonstrated in all of the remaining CR configurations. The reason for this is that the maximum LU factor for the loop is 16 (the compiler throws out errors when setting the LU factor at any value larger than 16), while the loop itself processes an additive and multiplicative operation of an array containing 17 elements. As a result, the design space excludes the ideal LU setting and hence all the executions use non-ideal configurations and show nearly the same performance results.

3) Accelerator Execution Overlapping (EO): Following the configurations in Fig. 5, we conduct the three execution scenarios with different call order of OpenCL API functions and the results are depicted in Fig. 12. EO optimisation of the *p-LDA*



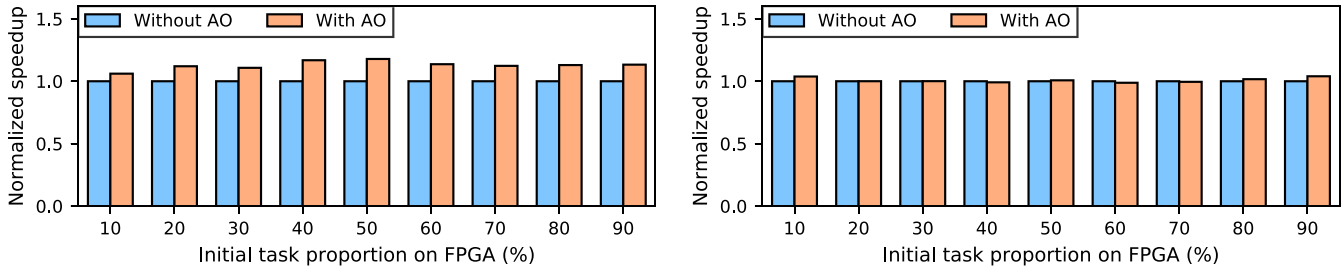
(a) *p-LDA* (b) *r-LDA*

Fig. 12. Performance comparison with different call order of OpenCL API functions.



(a) *p-LDA* (b) *r-LDA*

Fig. 13. Performance comparison with and without dynamical workload tuning.



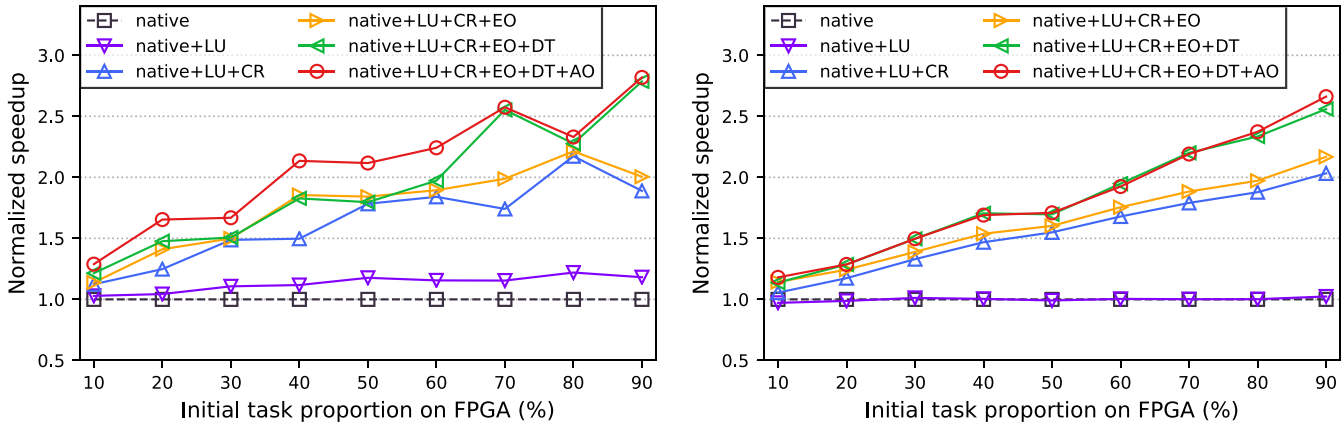
(a) *p-LDA* (b) *r-LDA*

Fig. 14. Performance comparison with and without AO optimisation.

application achieves a considerable performance boost (shown in Fig. 12(a)), while the speedup for *r-LDA* is slightly lower (shown in Fig. 12(b)). The average speedups of Case II over Case I are 1.20× and 1.06×, respectively for *p-LDA* and *r-LDA*. For both applications, executions of Case II always spend less time than that of Case I and III, which demonstrates the previous analysis in Section IV-D.

4) Dynamical Workload Tuning (DT): We apply the dynamical workload tuning mechanism illustrated in IV-E to the two case study applications and obtain the performance comparison results as presented in Fig. 13. As can be observed, DT optimisation achieves many more performance gains when the task proportion on FPGA is larger. This is especially obvious for *p-LDA*, where the DT-optimised execution can improve the performance by up to 39% (when 90% of the task is initially distributed on FPGA). The reason is that the dynamical workload tuning mechanism always regulates and assigns the appropriate amount of workloads for each accelerator. In this way, the tasks are gradually migrated to GPU when the initial task proportion on FPGA becomes larger, since in our evaluation the Quadro K600 GPU has a higher computation power than the Nallatech 385 FPGA. Therefore, the performance speedup is larger when a higher quantity of workloads are initially assigned to FPGA but subsequently consumed by GPU, compared with the fixed-task-proportion executions.

5) Application-Oriented Optimisation (AO): Figure 14 gives the performance results of using aforementioned AO optimisation methods for both applications, i.e. the ROI tuning scheme for *p-LDA* and the RANSAC iteration reduction for *r-LDA*. Tuning the ROI size improves the runtime performance of *p-LDA*



(a) *p-LDA* (b) *r-LDA*

Fig. 15. Performance comparison overview with step-by-step optimisations.

by an average of 12.86%, which reveals that shrinking the ROI size can effectively reduce the computational workload. The performance gain for *r-LDA*, however, is negligible, which is due to the minor proportion of RANSAC computation in the whole execution time. Hence even a huge reduction of RANSAC computation time cannot make a significant contribution to improving the overall runtime performance.

E. Summary and Discussion

1) Performance Benefit: Figure 15 shows the performance speedup of the two case study applications with the step-by-step optimisations mentioned above. Overall, the proposed optimisation procedure improves the runtime performance of both applications with a large extent. In our evaluation, the observed optimal executions of *p-LDA* and *r-LDA* can improve the performance by an average of 109.21% and 83.48% over the native parallel implementations, respectively.

As the task proportion on FPGA gradually increases, the performance speedup turns out a climbing trend as well, which is especially evident as seen from the curves of CR, EO, DT, and AO optimisations in Fig. 15(a) and 15(b). This reveals that the optimisations favour the FPGA platform and are more efficient when processing time-consuming workloads.

The LU optimisation for the test applications is not very significant. We attribute this to the resource constraint (for *p-LDA*) and non-ideal compiling issue (for *r-LDA*), which is illustrated in Section V-D2. A further study on other ADAS or generic scientific computing applications may better demonstrate the effectiveness of the LU optimisation. The CR optimisation contributes the most part to the final performance gain, since it enables the scaling of the kernel computation in a linear manner. The EO, DT, and AO optimisations, on the other hand, further improve the runtime performance. This is extremely important since these three optimisation methods are platform-independent and therefore can be seamlessly applied to other heterogeneous systems.

2) Scalability Analysis: The optimisation procedure proposed in this article can be applied to other applications and other heterogeneous parallel systems as well. The reasons are multi-fold. First, the CR optimisation is FPGA-related and the LU optimisation is valid for both GPU and FPGA. As nowadays GPU and FPGA are mainstream hardware accelerators used for high performance scientific computing, these two optimisation methods are applicable for any parallel applications running on GPU and FPGA platforms. Secondly, the EO and DT optimisations take effect when more than one accelerators, even multiple of the same type of processors, like either GPUs or FPGAs, are deployed for task processing. These optimisation methods are therefore suitable for general heterogeneous and reconfigurable computing. Lastly, the AO optimisation is algorithm-specific and can be flexibly adapted to other applications as long as the inherent parallel workloads in the target program are tunable, which is the normal case for state-of-the-art parallel applications.

VI. Conclusion

In this work, we propose a detailed procedure to help guide the performance optimisation of parallelized ADAS applications in FPGA-GPU heterogeneous systems. We provide one intra-accelerator and two inter-accelerator sub-optimisation methods, as well as both FPGA-specific and application-oriented customizations, to boost the runtime performance. Evaluation results show that our optimisation procedure can effectively reduce the time consumption, and the optimal designs of the case study applications can significantly improve the runtime performance over the native parallel implementations.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments that greatly contributed to improving the final version of the article.

REFERENCES

[1] Intel, "Intel GO Automotive Solutions," https://www.intel.com/content/www/us/en/automotive/go-automated-driving%.html, 2017.

[2] K. Huang, B. Hu, L. Chen, A. Knoll, and Z. Wang, "Adas on cots with opencl: a case study with lane detection," *IEEE Trans. Comput. (TC)*, 2017.

[3] J. Fickenscher, S. Reinhart, F. Hannig, J. Teich, and M. E. Bouzouraa, "Convoy tracking for adas on embedded gpus," in *Proc. IEEE Intell. Vehicles Symp. (IV)*. IEEE, 2017, pp. 959–965.

[4] M. M. Trompouki, L. Kosmidis, and N. Navarro, "An open benchmark implementation for multi-cpu multi-gpu pedestrian detection in automotive systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*. IEEE, 2017, pp. 305–312.

[5] A. Munshi, "The opencl specification," in *Proc. IEEE Hot Chips Symp. (HCS)*. IEEE, 2009, pp. 1–314.

[6] W. Wang, Y. Zhang, S. Yan, Y. Zhang, and H. Jia, "Parallelization and performance optimization on face detection algorithm with opencl: A case study," *Tsinghua Sci. Technol.*, vol. 17, no. 3, pp. 287–295, 2012.

[7] S. Sakjiraphong, A. Pinho, M. N. Dailey, M. Ekpanyapong, and A. Tavares, "Real-time road lane detection with commodity hardware," in *Proc. Int. Elect. Eng. Congr.*. IEEE, 2014, pp. 1–4.

[8] P. Georgiev, N. D. Lane, C. Mascolo, and D. Chu, "Accelerating mobile audio sensing algorithms through on-chip gpu offloading," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Serv. (MobiSys)*. ACM, 2017, pp. 306–318.

[9] X. Wei, Y. Liang, T. Wang, S. Lu, and J. Cong, "Throughput optimization for streaming applications on cpu-fpga heterogeneous systems," in *Proc. 22nd Asia South Pacific Des. Autom. Conf. (ASP-DAC)*. IEEE, 2017, pp. 488–493.

[10] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "Fpga-based real-time pedestrian detection on high-resolution images," in *Proc. IEEE Conf. Comput. Vision Pattern Recognit. Workshops*, 2013, pp. 629–635.

[11] W. Shi, X. Li, Z. Yu, and G. Overett, "An fpga-based hardware accelerator for traffic sign detection," *IEEE Trans. Very Large Scale Integr. Syst. (VLSI)*, vol. 25, no. 4, pp. 1362–1372, 2017.

[12] X. Wang, M. Cui, K. Huang, A. Knoll, and L. Chen, "Improving the performance of adas application in heterogeneous context: A case of lane detection," in *Proc. IEEE 20th Int. Conf. Intell. Transp. Syst. (ITSC)*. IEEE, 2017.

[13] X. Wang, C. Kiwus, W. Canhao, H. Bian, K. Huang, and A. Knoll, "Implementing and parallelizing real-time lane detection on heterogeneous platforms," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Architectures Processors*. IEEE, 2018.

[14] X. Wang, L. Liu, K. Huang, and A. Knoll, "Exploring fpga-gpu heterogeneous architecture for adas: Towards performance and energy," in *Proc. 17th Int. Conf. Algorithms Architectures Parallel Process. (ICA3PP)*. Springer, 2017, pp. 33–48.

[15] A. S. Huang, D. Moore, M. Antone, E. Olson, and S. Teller, "Finding multiple lanes in urban road networks with vision and lidar," *Auton. Robots*, vol. 26, no. 2, pp. 103–122, 2009.

[16] Q. Li, L. Chen, M. Li, S.-L. Shaw, and A. Nuchter, "A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios," *IEEE Trans. Veh. Technol. (TVT)*, vol. 63, no. 2, pp. 540–555, 2014.

[17] J. Sattar and J. Mo, "Safedrive: A robust lane tracking system for autonomous and assisted driving under limited visibility," *arXiv preprint arXiv:1701.08449*, 2017.

[18] B. Dorj and D. J. Lee, "A precise lane detection algorithm based on top view image transformation and least-square approaches," *J. Sensors*, vol. 2016, 2016.

[19] J. Niu, J. Lu, M. Xu, P. Lv, and X. Zhao, "Robust lane detection using two-stage feature extraction with curve fitting," *Pattern Recognit.*, vol. 59, pp. 225–233, 2016.

[20] H. Tan, Y. Zhou, Y. Zhu, D. Yao, and K. Li, "A novel curve lane detection based on improved river flow and ransa," in *Proc. IEEE 17th Int. Conf. Intell. Transp. Syst. (ITSC)*. IEEE, 2014, pp. 133–138.

[21] H. Xuan, H. Liu, J. Yuan, and Q. Li, "Robust lane-mark extraction for autonomous driving under complex real conditions," *IEEE Access*, 2017.

[22] H. Yoo, U. Yang, and K. Sohn, "Gradient-enhancing conversion for illumination-robust lane detection," *IEEE Trans. Intell. Transp. Syst.*, vol. 14, no. 3, pp. 1083–1094, 2013.

[23] S.-N. Kang, S. Lee, J. Hur, and S.-W. Seo, "Multi-lane detection based on accurate geometric lane estimation in highway scenarios," in *Proc. IEEE Intell. Vehicles Symp. (IV)*. IEEE, 2014, pp. 221–226.

[24] T. T. Duong, C. C. Pham, T. H.-P. Tran, T. P. Nguyen, and J. W. Jeon, "Near real-time ego-lane detection in highway and urban streets," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia*. IEEE, 2016, pp. 1–4.

[25] R. Gopalan, T. Hong, M. Shneier, and R. Chellappa, "A learning approach towards detection and tracking of lane markings," *IEEE Trans. Intell. Transp. Syst. (TITS)*, vol. 13, no. 3, pp. 1088–1098, 2012.

[26] M. Nieto, A. Cortés, O. Otaegui, J. Arróspide, and L. Salgado, "Real-time lane tracking using rao-blackwellized particle filter," *J. Real-Time Image Process.*, vol. 11, no. 1, pp. 179–191, 2016.

[27] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[28] R. Dietrich and R. Tschüter, "A generic infrastructure for opencl performance analysis," in *Proc. IEEE 8th Int. Conf. Intell. Data Acquisition Adv. Comput. Syst.: Technol. Appl. (IDAACS)*, vol. 1. IEEE, 2015, pp. 334–341.

[29] D. Gadioli *et al.*, "Opencl application auto-tuning and run-time resource management for multi-core platforms," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. (ISPA)*. IEEE, 2014, pp. 127–133.

[30] T. Lutz, C. Fensch, and M. Cole, "Helium: a transparent inter-kernel optimizer for opencl," in *Proc. 8th Workshop Gen. Purpose Process. Using GPUs*. ACM, 2015, pp. 70–80.

[31] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazelwood, "Analyzing program flow within a many-kernel opencl application," in *Proc. 4th Workshop Gen. Purpose Process. Graph. Process. Units (GPGPU)*. ACM, 2011, p. 10.

[32] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (ISFPGA)*. ACM, 2017, pp. 25–34.

[33] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (ISFPGA)*. ACM, 2016, pp. 16–25.

[34] X. An, E. Shang, J. Song, J. Li, and H. He, "Real-time lane departure warning system based on a single fpga," *EURASIP J. Image Video Process.*, vol. 2013, no. 1, p. 38, 2013.

[35] R. Marzotto, P. Zoratti, D. Bagni, A. Colombari, and V. Murino, "A real-time versatile roadway path extraction and tracking on an fpga platform," *Comput. Vision Image Understanding*, vol. 114, no. 11, pp. 1164–1179, 2010.

[36] Y. Xing *et al.*, "Advances in vision-based lane detection: Algorithms, integration, assessment, and perspectives on acp-based parallel vision," *IEEE/CAA J. Automatica Sinica*, vol. 5, no. 3, pp. 645–661, 2018.

[37] C. Blair, N. M. Robertson, and D. Hume, "Characterizing a heterogeneous system for person detection in video using histograms of oriented gradients: Power versus speed versus accuracy," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 3, no. 2, pp. 236–247, 2013.

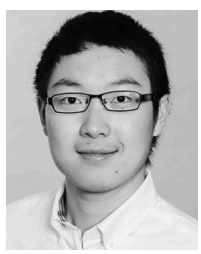
[38] M. Yih, J. Ota, J. Owens, and P. Muyan-Ozcelik, "Fpga versus gpu for speed-limit-sign recognition," *UC Davis Previously Published Works*, 2018. [Online]. Available: https://escholarship.org/uc/item/8ww3d2gg

[39] I. Sobel, "An isotropic 3 × 3 image gradient operator," *Mach. Vision Three-Demensional Sci.*, 1990.

[40] N. J. Gordon, D. J. Salmond, and A. F. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," in *Proc. IEE Proc. F-Radar Signal Process.*, vol. 140, no. 2. IET, 1993, pp. 107–113.

[41] H. Kong, J.-Y. Audibert, and J. Ponce, "General road detection from a single image," *IEEE Trans. Image Process. (TIP)*, vol. 19, no. 8, pp. 2211–2220, 2010.

[42] M. Aly, Caltech lanes. Accessed: 2018-04-06. [Online]. Available: http://www.vision.caltech.edu/malaa/datasets/caltech-lanes



Xiebing Wang received the B.Eng. and M.Eng. degrees from the Computer School, Wuhan University, Wuhan, China, in 2012 and 2015, respectively. He is currently working toward the Ph.D. degree with the Department of Informatics, Technical University of Munich, Garching, Germany. His research interests include parallel computing, heterogeneous computing, and autonomous driving.

Kai Huang received the B.Sc. degree from Fudan University, Shanghai, China, in 1999, the M.Sc. degree from the University of Leiden, Leiden, The Netherlands, in 2005, and the Ph.D. degree in ETH Zurich, Zurich, Switzerland, in 2010. He joined Sun Yat-Sen University as a Professor in 2015. He was the Director of the Institute of Unmanned Systems of School of Data and Computer Science in 2016. He was a Senior Researcher with the Computer Science Department, Technical University of Munich, Garching, Germany, from 2012 to 2015, and a Research Group Leader with fortiss GmbH, Munich, Germany, in 2011. His research interests include techniques for the analysis, design, and optimization of embedded systems, particularly in the automotive and robotic domains. He was awarded the Program of Chinese Global Youth Experts 2014 and was granted the Chinese Government Award for Outstanding Self-Financed Students Abroad 2010. He has served as a member of the technical committee on Cybernetics for Cyber-Physical Systems of IEEE SMC Society since 2015.

Alois Knoll received the M.Sc. degree in electrical/communications engineering from the University of Stuttgart, Stuttgart, Germany, in 1985, and the Ph.D. degree in computer science from the Technical University of Berlin, Berlin, Germany, in 1988. He served on the faculty of the Computer Science Department, TU Berlin until 1993. He was a Full Professor with the University of Bielefeld until 2001. Since 2001, he has been a Professor of computer science with the Department of Informatics, Technical University of Munich, Garching, Germany. Between April 2004 and March 2006, he was the Executive Director of the Institute of Computer Science, TUM. Between 2007 and 2009, he was a member of the EUs highest advisory board for information technology, ISTAG, the Information Society Technology Advisory Group, and a member of its subgroup for Future and Emerging Technologies (FET). In this capacity, he was actively involved in developing the concept of the EU's FET Flagship projects, and he was one of the authors of the original FET-Flagship report. His research interests include cognitive, medical and sensor-based robotics, multi-agent systems, data fusion, adaptive systems, multimedia information retrieval, model-driven development of embedded systems with applications to automotive software and electric transportation, as well as simulation systems for robotics and traffic.