Exploring FPGA-GPU Heterogeneous Architecture for ADAS: Towards Performance and Energy

Xiebing Wang[1(✉)], Linlin Liu[2], Kai Huang[2], and Alois Knoll[1]

[1] Technische Universität München, 85748 Garching bei München, Germany
{wangxie,knoll}@in.tum.de
[2] Sun Yat-sen University, Guangzhou 510275, People's Republic of China
liull28@mail2.sysu.edu.cn, huangk36@mail.sysu.edu.cn

Abstract. This paper investigates the feasibility of using heterogeneous computing for future advanced driver assistance systems (ADAS) applications. In particular, we take lane detection algorithm (LDA) as a test case. The algorithm is customized into FPGA-GPU heterogeneous implementations which can be executed in either workload constant or balanced scheme. Then the heterogeneous executions are evaluated in view of performance and energy consumption, and further compared with the single-accelerator run. Experiments show that the heterogeneous execution alleviates both the performance and energy bottlenecks caused when only using a single accelerator. Moreover, compared with the single FPGA execution, the workload balance scheme increases the performance by 236.9% and 42.9% on our two tested platforms respectively, while ensuring the low energy cost.

Keywords: Advanced Driver Assistance Systems (ADAS) · OpenCL · FPGA · GPU

1 Introduction

For the automotive industry, advanced driver assistance systems (ADAS) are born to take full advantage of massive multi-sensor information so as to improve in-car and on-road safety. However, the input database space for ADAS applications is so large that it poses a big challenge for software developers to design both real-time and highly efficient algorithms. For these applications, time constraint and reliability guarantee are vital, due to the critical personal and property safety.

To flatten the real-time bound, commercial-off-the-shelf (COTS) hardware accelerators are used to precipitously shorten the execution time of the on-vehicle applications. For instance, since 2014 Nvidia has launched Jetson series [14] for GPU-accelerated parallel processing in the mobile embedded system market. Nevertheless, together with the high performance benefited from GPU also comes the inevitable significant energy consumption. Meanwhile, due to the low energy

S. Ibrahim et al. (Eds.): ICA3PP 2017, LNCS 10393, pp. 33–48, 2017.
DOI: 10.1007/978-3-319-65482-9_3

cost, FPGA as another mainstream accelerator, is widely used in integrated embedded systems.

Aiming at high performance computing (HPC) applications in embedded systems, heterogeneous computing emerges as it leverages different accelerators, such as FPGAs and GPUs, to strengthen the advantages of the individual counterpart. Moreover, this type of reconfigurable computing framework is very compatible with portable platforms because of its high flexibility and scalability. From 2008, open computing language (OpenCL) arises and turns out to be an ideal heterogeneous programming framework as it enables to scale computations among CPUs, GPUs and FPGAs without changing the source code. However, the performance portability on different COTS components cannot be guaranteed due to the diverse OpenCL implementations by respective board vendors. Moreover, to our best knowledge, it is still unknown to what extent the heterogeneous context could be used for the automotive applications.

This paper uses typical lane detection as case study to probe the feasibility of using FPGA-GPU heterogenous architecture for ADAS applications. Lane detection algorithm (LDA) is a well-tested technique and commonly used on conventional electronic control units (ECUs) to assist better driving. We adopted the algorithms developed by [10]. In [10], the authors proposed a particle-filter based algorithm that can detect and track on-road lane markings real-timely. However, the algorithm was only tested in view of performance, while using a single FPGA or GPU. We customized this algorithm into a data-level parallel program to enable its execution in heterogeneous context. Afterwards the program was deployed and executed on two heterogeneous platforms which were equipped with different COTS hardware accelerators. Furthermore, based on the workload constant scenario, we developed a lightweight workload balance scheme that could dynamically identify and adjust the workloads on FPGA and GPU. Experiments showed that the heterogeneous execution resolved both the performance and energy bottlenecks caused when only using a single FPGA or GPU. The workload balance scheme could further reduce the time cost to a large extent, while ensuring the low energy cost. Besides, the proposed scheme can robustly adjust and stabilize the workload according to the computation capacity of each computation device. The main contribution of this paper lies in:

- We use a real-life LDA as the test case and propose a time and energy efficient heterogeneous implementation of this widely-used automotive application.
- Based on the heterogeneous design, we give a lightweight workload balance scheme that can increase the performance by 236.9% and 42.9% on our two test platforms respectively, while ensuring the low energy cost. What's more, the scheme can robustly adjust the workload in diverse road scenarios, based on the computation capacity of each accelerator in use.
- Taking real-life road scenarios as input, we conduct a series of experiments on two heterogeneous platforms, on which different pairs of FPGA and GPU are equipped. Experimental results demonstrate the necessity of utilizing FPGA-GPU combined heterogeneous architecture for future ADAS.

The rest of this paper is organized as follows: Sect. 2 is related work and Sect. 3 overviews the procedure of the tested LDA. Section 4 presents the heterogeneous design and the workload balance scheme. Section 5 gives experimental analysis and Sect. 6 concludes the paper.

2 Related Work

Lots of previous research has compared the performance of using FPGA and GPU in different areas, like deep learning [13], information security [5] and image processing [3,6,7]. These studies present the distinct characteristics that FPGA and GPU show in their computing competence. Generally, FPGA is adept at floating-point arithmetic operations and GPU shows better performance on matrix manipulations. Due to these features, researchers attempt to explore heterogenous architecture to accelerate scientific computing applications. Authors in [1] proposed a heterogenous FPGA-GPU-CPU platform for a sport real-time locating system. The platform is task-level parallel as FPGA is used for data acquisition and GPU is mainly for object tracking. In their design, FPGA is used more as a data gathering processor than a computation accelerating device. Similarly, authors in [12] used the combined FPGA-GPU architecture to perform cardiac physiological optical mapping. In this system, the FPGA is responsible for camera data capture and the GPU mainly disposes fast fourier transform (FFT), inverse fast fourier transform (IFFT) and filtering operations.

Among the aforementioned research, GPU always handles the major computation workload and the performance of FPGA and GPU cannot be directly compared since the task granularity on each device is apparently different.

Several other research is also emerging to compare the performance of FPGA and GPU in the field of real-time processing. Authors in [7] presented a systematic approach to compare FPGA and GPU with five case study algorithms. Their work focused on the algorithmic, data and hardware characteristics of the applications and finally gave a throughput performance of the target devices. In [8], the authors used the roofline model [16] to identify the appropriate accelerator for candidate applications and then performed the comparison based on a pedestrian recognition application called fastHOG. Their work concentrated on the task distribution between different accelerators. Both of the studies in [7,8] do not involve the energy evaluation. Authors in [15] gave a thorough comparison of FPGA and GPU for computer vision algorithms, using a case study of threaded isle detection. Their evaluations are rather comprehensive, including performance, hardware cost, power efficiency and integratability. However, their work cannot reveal the impact of OpenCL on FPGA and GPU, since the algorithms are individually implemented using different programming languages. The most related work to this paper is [4]. In [4], pedestrian detection applications are implemented on a heterogeneous FPGA-GPU-CPU platform and then the authors compared the power, speed and accuracy of several different scenarios, where either FPGA, GPU or both are used for the computation. The difference to our work is that they also used the task-level parallelism like [1] and [12], among the accelerators.

Different to all the work mentioned above, we adopt the data-level parallelism for FPGA and GPU devices so that their performance characteristics can be directly and intuitively compared. Thereupon the heterogeneous designs are evaluated in consideration of time and energy consumption to demonstrate the advantage of using heterogeneous architecture for ADAS applications.

3 Particle-Filter Based LDA

3.1 Algorithm Overview

This section briefly describes the naive design of the LDA and the procedure is shown in Fig. 1. The algorithm mainly consists of three modules (the slash boxes in Fig. 1), namely *pre-processing*, *lane detection* and *lane tracking*. The algorithm analyzes the video stream captured by a moving vehicle and attempts to extract the exact positions of the lane markings highlighted in the output stream.

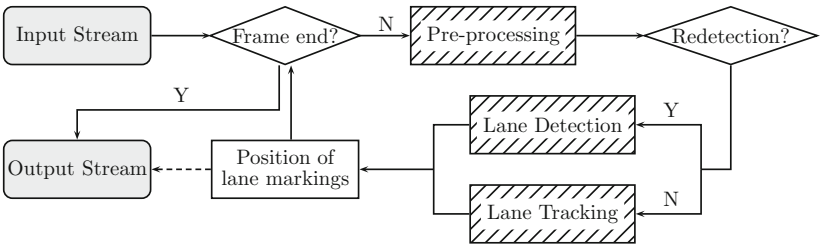

Fig. 1. Flow chart of LDA.

Pre-processing module includes four steps successively applied to the original image. First a region of interest (ROI) is cropped from the raw image and only this ROI is further processed. Then the ROI is transformed into grayscale space where each pixel reflects the intensity of the pixel in original image. After grayscaling, the edges of the lane markings are slightly obvious since they are substantially brighter than the streets and roads around. To enhance this contrast of pixel intensity, a Sobel filter is applied to the grayscaled image to detect pixel variations and extract edges. Finally, a threshold is used to tune the intensity of all pixels in the image to avoid noise influence.

Lane detection module generates a set of *candidate lines* via assigning random values from a normal distribution to form the candidate line set. For each candidate line, a weight is calculated to reveal how close the line is located to the real lane. Given this weight set, the line with the highest weight is chosen as the *best line* and certain number of candidate lines are reserved as *good lines*, which would be further used in the *lane tracking* module.

Lane tracking module adopts a particle filter to predict the positions of 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:

1. The *prediction update* step amends previous *good lines* with a normal distribution $N(\mu, \sigma^2)$, with mean $\mu = 0$ and standard deviation $\sigma > 0$. $\mu = 0$ means no shift is expected in optimal case, while $\sigma > 0$ reveals a deviation in real scenarios. The updated lines are seen as prior probability distribution of the lane markings in current frame.
2. The *importance weight update* step recalculates the weights of the particles via Gaussian function

$$\omega_i = \frac{1}{\sigma_f \sqrt{2\pi}} e^{-(X_i - \mu_f)^2 / 2\sigma_f^2}, i = 1, 2, \cdots, N \tag{1}$$

   where N is the particle number, μ_f indicates the *best line* in previous frame and σ_f expresses the noise that accounts for a possible error in case the position of the lane marking does not change within two frames. Then the importance weight of each particle is normalized to obtain the updated weight

$$\omega_i^{updated} = \frac{\omega_i}{\sum_{i=1}^{N} \omega_i}. \tag{2}$$

3. Based on the importance weights, the *resampling* step selects particles from the newly updated set to prevent a degeneration of the particle set.

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. The criteria of redetection is as follows: (i) Lane markings do not cross. (ii) There exists a minimum distance between each two detected lane markings. This value is adjustable and can be small when lots of lanes have to be detected. (iii) There should be a minimum percentage of the lane marking within the ROI. This parameter is flexible and can be user-defined.

3.2 Initial Design

In the basic version, each of the modules depicted in Sect. 3.1 is programmed as an OpenCL kernel which will be executed on the hardware accelerator. For simplicity, `KERNEL_PRE`, `KERNEL_LD` and `KERNEL_PF` are used as their individual kernel names. Note that both lane detection and tracking require normally distributed random numbers to process their following tasks. Hence these numbers should be generated by a random number generator. Therefore another kernel called `KERNEL_RNG` is required. With above four kernels, the flow chart in Fig. 1 is abstracted as the pseudo-code shown in Algorithm 1, where the red lines (lines 2, 4, 6, 9 in Algorithm 1) represent the kernel tasks.

4 Heterogeneous Design

4.1 Data-Level Parallelism

The heterogeneous version of the application tries to distribute the kernel tasks among different accelerators. From Algorithm 1 it is seen that for each input

Algorithm 1. LDA (basic version)

Input: raw camera-captured video stream
Output: video stream with lanes marked

```
initialization
random number generation //KERNEL_RNG
while not the end frame do
  ROI image pre-processing //KERNEL_PRE
  if redetection then
    lane detection //KERNEL_LD
    candidate line generation
  else
    lane tracking //KERNEL_PF
    good line resampling
  end if
  best line extraction and mark lanes in current frame
end while
```

video stream, random number generation (KERNEL_RNG) is run only once and the other three kernels are executed repeatedly inner the frame loop. For this reason, KERNEL_RNG can be performed on every accelerator since its time cost is rather small, while the other kernels should be scattered across the accelerators as they are the main tasks.

Meanwhile, it is worth noting that two layers of data dependencies exist here: (i) both the executions of KERNEL_LD and KERNEL_PF use the output of KERNEL_RNG and KERNEL_PRE, and (ii) if the current frame is the first tracking frame, then it will need the detected positions of lane markings in the previous frame, in this case the execution of KERNEL_PF relies on the output of KERNEL_LD. Consequently, task-level parallelism for these three kernels is not desirable as it requires the indirect Device→Host→Device data transfer, which is considerably time-consuming due to the lack of state-of-the-art commercial direct FPGA-GPU data communication mechanism.

From the above, data-level parallelism of the basic LDA is used for the heterogeneous context and Fig. 2 gives the overall processing procedure. In general, the host utilizes an installable client driver (ICD) loader to coordinate the tasks executing on FPGA and GPU. When invoking OpenCL API functions, the program runtime passes kernel parameters to the ICD loader and then the ICD loader calls FPGA- and GPU-specific functions with *fpga*- and *gpu*-specific parameters respectively.

The host side is responsible for (i) kernel parameters initialization and raw image I/O when the program begins, and (ii) result collection, weight updating and line resampling during the frame loop. On each hardware accelerator, the ROI of the image is preprocessed and then the detection kernel (KERNEL_LD) samples a set of *candidate lines* and calculates their intensity weights individually. As shown in Fig. 2, KERNEL_LD processes n lines on the FPGA and m lines on the GPU, and subsequently returns the intensity weights to the host. On the host,

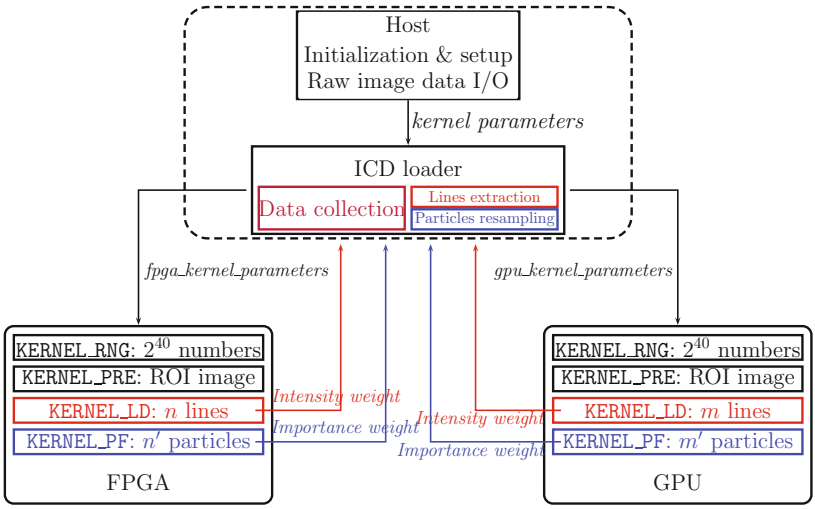

Fig. 2. Execution of LDA in heterogeneous context overview. Red and blue items are distributed tasks on the FPGA and GPU. The italic items show the transfer of parameters. (Color figure online)

after extracting a series of *good lines* and one *best line*, the lane detection operation outputs the position of the lane markings as the form of *best line*. Similarly for lane tracking kernel (`KERNEL_PF`), a group of particles are extracted from the output data of `KERNEL_LD`. Again these particles are scattered and processed on the two accelerators. Here n' and m' particles are respectively disposed on the FPGA and GPU. When the importance weights of the particles are finished calculating, they are returned back to the host side and new particles are resampled based on the aggregated results to step into the new iteration.

4.2 Workload Balance

To get the optimal execution, the workload of `KERNEL_LD` and `KERNEL_PF` on GPU and FPGA needs to be dynamically assigned since GPU and FPGA show distinct computation capacities in consideration of different types of data manipulations. This is especially important when the application is intended to be scaled across different platforms, where different FPGA and GPU boards are used. Since time and energy costs are two of the most important indicators when monitoring ADAS applications, this paper gives a time optimization based workload balance scheme for the heterogeneous LDA and the energy cost is afterwards investigated.

Algorithm 2 briefs the workload balance scheme. Here *funcRNG*, *funcPRE*, *funcLD* and *funcPF* are corresponding kernel functions, from which the timing information can be profiled. The details of function *funcAdjustWL* are shown in Algorithm 3. Assume that the input is the initial task load for FPGA and GPU devices (i.e., m, n, m', n' in Fig. 2), and the output is the time-optimal executions

Algorithm 2. Workload balance scheme

Input: m, n, m', n'
Output: t_{kernel}
1: $t_{rng_f} \leftarrow funcRNG(m+n)$, $t_{rng_g} \leftarrow funcRNG(m+n)$
2: $t_{kernel} \leftarrow max(t_{rng_f}, t_{rng_g})$
3: **while** not the end frame **do**
4: $t_{pre_g} \leftarrow funcPRE(m)$, $t_{pre_f} \leftarrow funcPRE(n)$
5: $t_{pre} \leftarrow t_{pre_f} + t_{pre_g}$
6: $t_{kernel} \leftarrow t_{kernel} + t_{pre}$
7: **if** redetection **then**
8: $t_{ld_g} \leftarrow funcLD(m)$, $t_{ld_f} \leftarrow funcLD(n)$
9: $t_{kernel} \leftarrow t_{kernel} + max(t_{ld_f}, t_{ld_g})$
10: m, $n \leftarrow funcAdjustWL(t_{ld_f}, t_{ld_g}, m, n)$
11: **else**
12: $t_{pf_g} \leftarrow funcPF(m')$, $t_{pf_f} \leftarrow funcPF(n')$
13: $t_{kernel} \leftarrow t_{kernel} + max(t_{pf_f}, t_{pf_g})$
14: m', $n' \leftarrow funcAdjustWL(t_{pf_f}, t_{pf_g}, m', n')$
15: **end if**
16: **end while**

Algorithm 3. Function *funcAdjustWL* in Algorithm 2

Input: t_f, t_g, W_f, W_g
Output: W_f, W_g
1: $c_f \leftarrow \frac{W_f}{t_f}$, $c_g \leftarrow \frac{W_g}{t_g}$
2: $W_f \leftarrow \frac{c_f}{c_f+c_g}(W_f + W_g)$, $W_g \leftarrow \frac{c_g}{c_f+c_g}(W_f + W_g)$

of the program (indicated as kernel execution time t_{kernel}). The idea is that the workload for a device should be proportional to its computation capacity, i.e., its throughput. Hence, after each frame is processed complete, the kernel execution time on each device is recorded (lines 1, 4, 8, 12 in Algorithm 2) and the throughput is calculated. Then the total work load is re-assigned based on the current throughputs of the computing devices (lines 10, 14 in Algorithm 2). This scheme assumes that for each frame, the execution times of `KERNEL_LD` and `KERNEL_PF` are proportional to their current task load.

4.3 Performance and Energy Evaluation

In our context, totally four scenarios are involved, namely, single FPGA execution (***singleFPGA***), single GPU execution (***singleGPU***), work-load-constant (***heteroConstant***) and work-load-balanced (***heteroBalanced***) heterogeneous execution. In work-load-constant scenario, the whole task is partitioned in advance and then fed to FPGA and GPU devices. Thus the task proportions on FPGA and GPU are always constant. While in work-load-balanced scenario, with given partitioned task, the workload balance scheme tunes the task proportions on FPGA and GPU during the processing of each frame.

To reveal the tradeoffs between these situations, we record the execution time of all the four implementations to evaluate their real-time performance over the energy cost. In order to calculate the energy cost, we construct the runtime environment where both FPGA and GPU cards are working in full load mode, so that the peak power consumption can be reached. To fulfill this, we use extremely large computation task load since the particle filter is highly scalable and consequently the larger the particle number is, the more the computation task load would be. During the evaluation, the execution time of each run is measured to calculate the real-time performance. We use the same power estimation method as [9] and Altera PowerPlay power analyzer [11] is used to estimate the power consumption of running each OpenCL kernel on FPGA. As for the power estimation of GPU and CPU, we use data from official specifications of the COTS components.

5 Experiment and Analysis

5.1 Experimental Setup

The applications are run in two different heterogeneous contexts listed in Table 1. Both platforms contain one FPGA and one GPU board. For contrast, they are equipped with two groups of boards which show rather different computation capacities. Platform #1 is deployed with a Terasic Arria 10 FPGA and an AMD W7100 GPU, while a Nallatech pcie385n FPGA and an Nvidia Quadro K600 are used on platform #2. Note that the computation capacities of the FPGA and GPU boards on each platform are rather different. AMD W7100 presents an obviously superior performance than Arria 10, while Nallatech pcie385n and Quadro K600 have comparable computing capacities. The purpose of this is to demonstrate the robustness of our applications in heterogeneous contexts where accelerators have unbalanced computation competence.

Table 1. Detailed specification of the hardware platforms

Platform	#1		#2	
Host CPU	Intel Xeon E31225 @ 3.10 GHz		Intel Core 2 Quad Q9300 @ 2.50 GHz	
Thermal Design Power	95 W		95 W	
Device	FPGA	GPU	FPGA	GPU
Model	Terasic Arria 10	AMD W7100	Nallatech 385	Quadro K600
Architecture	Arria 10 AX	FirePro	Stratix V GS	Kepler GK
OpenCL SDK version	Intel FPGA SDK 16.0	AMD APP SDK 3.0	Intel FPGA SDK 13.1	CUDA 8.0
Peak GFLOPS	1366	3379.2	294.7	336.4
Peak board power (W)	95	150	25	40

To demonstrate the high availability of using the tested LDA for real-life driving conditions, we use video streams from different data sets with different scenarios. The detailed information of these videos is listed in Table 2, of which `cordova1`, `cordova2`, `washington1` and `washington2` are from Caltech lanes dataset [2], while the others are self-recorded. These videos are captured in different resolutions and the frame numbers have a great range from 232 (`washington2`) to 4992 (`night_land_car`). Moreover, these videos represent various road situations including in day and night, with heavy traffic, with blurred and broken lines, in street and highway, in urban and rural areas, etc. The purpose of this is to obtain as actual results as possible.

Table 2. Detailed information of the test videos

Videos	Name	Total frames	Resolution	Scenario
1	cordova1	250	640×480	bus view
2	cordova2	406	640×480	blur lane
3	washington1	337	640×480	street shade
4	washington2	**232**	640×480	blur lane
5	street	3056	640×480	street road
6	day_highway	1718	640×480	high way
7	Frontfacingobstacle	4601	480×360	crossing lane
8	HighSpeedDrivingShort	1871	1920×1080	high way
9	clip2	1289	640×360	rural
10	clip4	899	640×360	dark
11	night_land_car	**4992**	640×480	night
12	night_traffic	2654	640×480	heavy traffic
13	oli_4	2287	480×320	broken lane
14	night_4	2799	640×480	night highway
15	night_brokenlanes	1897	640×480	broken lane
16	Weilerhemmen	4944	640×480	light disturbance

During the experiments each video is run 10 times per platform and the overall results are collected and averaged. To construct the large task load, we use rather large numbers of particles to iterate over each generation of the line sets. In details, during each run we use 2^{12} *good lines* and 2^{13} *candidate lines* to detect 2 *best lines*. As for the heterogeneous executions, the initial task proportion on FPGA is set as the range from 1% to 99% and the rest part is executed on GPU. When using the workload balance scheme, an initial task proportion is given and afterwards both the task proportions of FPGA and GPU are recorded frame by frame to present the real-time work load distribution.

Note that for *singleFPGA* and *singleGPU* scenarios, the task proportions on FPGA are constant 100% and 0%, respectively. Hence the results of *singleFPGA* and *singleGPU* are used as reference for evaluating the heterogenous executions.

5.2 Results and Analysis

Workload Balance Scheme. The objective of the workload balance scheme is to minimize the kernel execution time (t_{kernel} in Algorithm 2). To validate the correctness and robustness of this scheme, (i) the kernel execution times of the four designs are recorded and (ii) during the *heteroBalanced* run, the real-time task rates on both FPGA and GPU devices are monitored. Figure 3 summarizes the experimental results. Due to the page limit, Fig. 3(c) and (d) only show the real-time task rates of `washington2` and `night_land_car` as they are the two videos with the smallest and largest frame numbers.

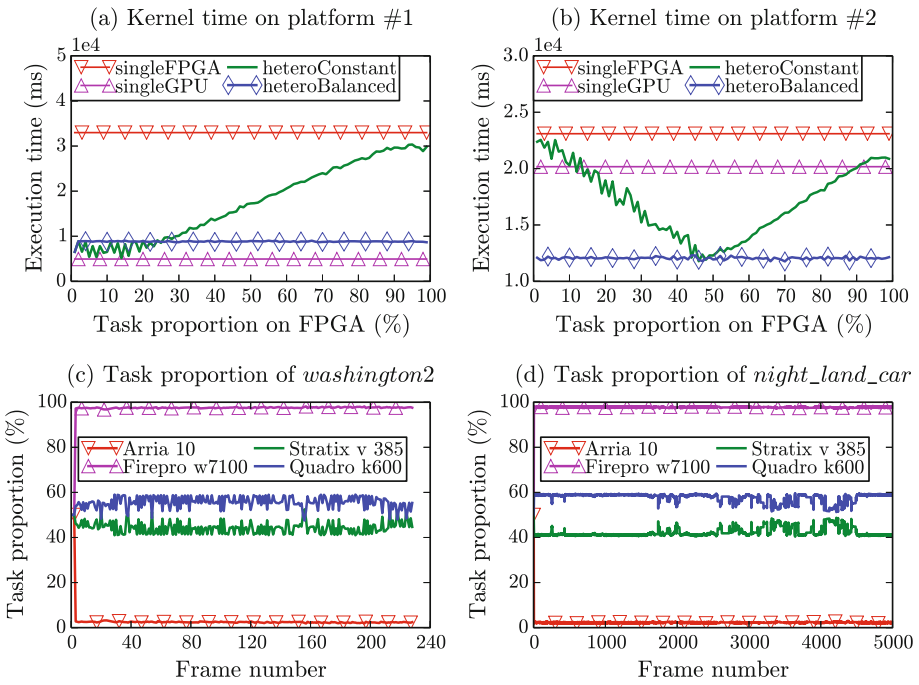

Fig. 3. Validity and robust test results of the workload balance scheme.

Figure 3(a) and (b) indicate that when compared with *singleFPGA*, both of the *heteroConstant* and *heteroBalanced* implementations can shorten the kernel execution time to a large degree. The kernel time cost of *heteroBalanced* is 26.94% and 51.96% of *singleFPGA*, 177.76% and 59.49% of *singleGPU* on

platform #1 and #2, respectively. It's seen that the time costs of the heterogeneous executions on platform #1 are larger than the *singleGPU* case. This is because the time cost of *singleFPGA* is an order of magnitude larger than that of *singleGPU*. Therefore simply shifting the task a little from GPU to FPGA would incur considerable latency. As can be oberserved, on both platforms the kernel execution time of *heteroConstant* always surpasses *heteroBalanced*, which verifies the validity of the workload balance scheme. In Fig. 3(c) and (d), it is seen that the real-time task proportions of both videos converge within 5 frames and then keep relatively constant with minor fluctuations. What's more, the workload balance scheme can identify the optimal task distributions on FPGA and GPU, regardless of the input video. To be specific, the optimal task rates on the FPGA of platform #1 and #2 are around 2% and 41%, respectively. This demonstrates the robustness of the workload balance scheme.

Performance. Figure 4 depicts the performance of the four implementations running on the two test platforms. From the figure we observe that on both platforms the performance of *singleGPU* outperforms *singleFPGA* and this is reasonable due to the lower computation capacity of FPGA (refer to the peak GFLOPS in Table 1). Both of the heterogeneous runs gain a performance increase than *singleFPGA*, which without doubt benefits from the high performance GPU.

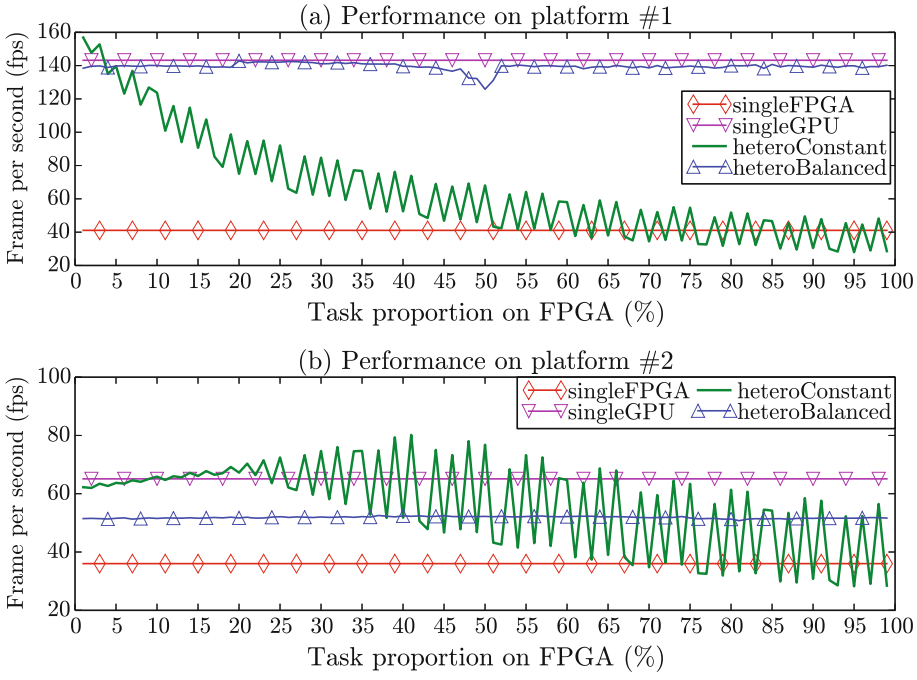

Fig. 4. Performance results overview.

The *heteroConstant* execution displays a considerable fluctuation. This is because when gradually increasing the task rate, due to the OpenCL specification, the task load on FPGA shows a discrete step change, which greatly influences the CPU↔FPGA data transfer latency since direct memory access (DMA) requires data alignment of the transmitted data. Intuitively, the performance declines when more and more tasks are shifted to FPGA. As for *heteroBalanced* scenario, the performance turns out very stable since the task load is dynamically allocated and the heterogeneous execution would rapidly converges to equilibrium after several frames, which is verified in Sect. 5.2. Moreover, on platform #1 the balanced run could achieve a comparative performance over the *singleGPU* run.

On the whole, using heterogeneous architecture improves the performance when compared with the *singleFPGA* lower bound. The workload balance scheme reconciles the heterogeneous system and during all task rates, *heteroBalanced* increases the performance by 236.9% and 42.9% on platform #1 and #2 respectively, when compared with *singleFPGA*.

Energy. Figure 5 shows the overall energy cost for the four different designs. Figure 5(a) and (b) present the energy cost of the overall system, while Fig. 5(c) and (d) give the results of the accelerator energy consumption.

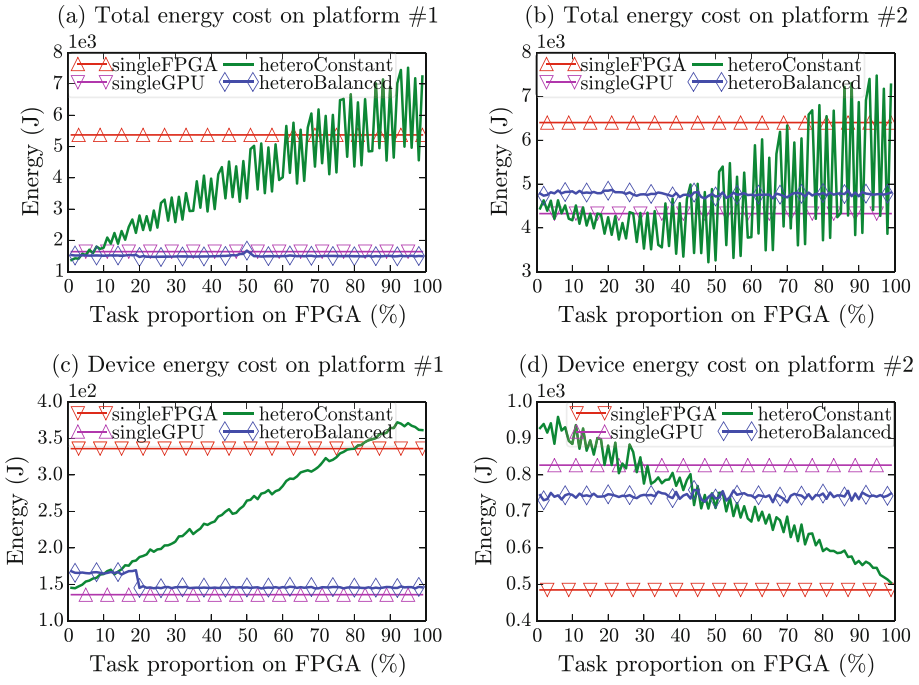

Fig. 5. Energy consumption overview.

As indicated by Fig. 5(a) and (b), the system energy is much larger when using a single FPGA, compared with the energy cost of *singleGPU*. This is mainly because the overall execution time of *singleFPGA* is much longer than *singleGPU*, which poses a huge increment of the CPU energy cost. However, on both platforms our heterogeneous designs are able to consume almost as less energy as *singleGPU*. The *heteroBalanced* implementation utilizes the least energy on platform #1 and increases the energy by only 10.98% when compared with *singleGPU* on platform #2.

With regards to the on-device energy cost (Fig. 5(c) and (d)), the two platforms exhibit different features. On platform #1, using a single GPU costs the least device energy and we owe this to the huge speedup of the AMD W7100 card. The energy cost of FPGA is not able to outperform the GPU because the low-power advantage of FPGA over GPU simply cannot compensate for the far-behind performance gap. As the consequence, the device energy increases linearly when tasks are migrated on FPGA, which is clearly observed via the *heteroConstant* curve. Nevertheless, the *heteroBalanced* design commendably suppresses the energy cost, as it manages to identify the power-performance tradeoff of FPGA and GPU and subsequently always distributes more task load on GPU. As for platform #2, the performance gap between Stratix V 385 and Quadro K600 is much narrower and in this case FPGA fully displays its low-power characteristic, when comparing the result of the *singleFPGA* and *singleGPU* curves. Compared with the *singleGPU* upper bound, *heteroConstant* reduces the energy cost to 89.49%.

In summary, the heterogeneous executions consume less energy, when compared with the most-energy-cost single accelerator (i.e., *singleFPGA* in Fig. 5(a), (b) and (c), *singleGPU* in Fig. 5(d)). Using the workload balanced scheme not only "smoothes" the heterogeneous execution, but also shortens the energy cost regardless of the initial task rates. On both platforms, when using the heterogeneous architecture, the performance can be boosted while ensuring the low energy cost.

6 Conclusion and Future Work

Heterogeneous computing is a promising solution for future ADAS since it is able to regulate the performance and energy tradeoff in the system. This paper used typical lane detection as case study to probe the feasibility of using FPGA-GPU combined heterogenous architecture for ADAS applications. The performance and energy costs were carefully evaluated among the heterogeneous and single-accelerator executions. We demonstrated that the heterogeneous implementations could solve both the performance and energy bottlenecks caused when only using a single accelerator. Moreover, the proposed workload balance scheme can further boost the performance, while ensuring the low energy cost.

Our future work is to use more ADAS applications to verify the pros and cons of the heterogeneous computing.

Acknowledgments. This work is supported in part by the scholarship from China Scholarship Council (CSC) under the Grant No. 201506270152.

References

1. Alawieh, M., Kasparek, M., Franke, N., Hupfer, J.: A high performance FPGA-GPU-CPU platform for a real-time locating system. In: 23rd European Signal Processing Conference (EUSIPCO), pp. 1576–1580. IEEE (2015)
2. Aly, M.: Caltech lanes. http://www.vision.caltech.edu/malaa/datasets/caltech-lanes. Accessed 10 Mar 2017
3. Asano, S., Maruyama, T., Yamaguchi, Y.: Performance comparison of FPGa, GPU and CPU in image processing. In: 19th International Conference on Field Programmable Logic and Applications (FPL), pp. 126–131. IEEE (2009)
4. Blair, C., Robertson, N.M., Hume, D.: Characterizing a heterogeneous system for person detection in video using histograms of oriented gradients: power versus speed versus accuracy. IEEE J. Emerg. Sel. Top. Circ. Syst. **3**(2), 236–247 (2013)
5. Che, S., Li, J., Sheaffer, J.W., Skadron, K., Lach, J.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Proceedings of the 6th IEEE Symposium on Application Specific Processors (SASP), pp. 101–107. IEEE (2008)
6. Chen, D., Singh, D.: Fractal video compression in OpenCL: an evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In: 18th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 297–304. IEEE (2013)
7. Cope, B., Cheung, P.Y., Luk, W., Howes, L.: Performance comparison of graphics processors to reconfigurable logic: a case study. IEEE Trans. Comput. **59**(4), 433–448 (2010)
8. Da Silva, B., Braeken, A., D'Hollander, E.H., Touhafi, A., Cornelis, J.G., Lemeire, J.: Comparing and combining GPU and FPGA accelerators in an image processing context. In: 23rd International Conference on Field Programmable Logic and Applications (FPL), pp. 1–4. IEEE (2013)
9. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 47–56. ACM (2012)
10. Huang, K., Hu, B., Botsch, J., Madduri, N., Knoll, A.: A scalable lane detection algorithm on COTSs with OpenCL. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 229–232. IEEE (2016)
11. Intel: Powerplay early power estimators and power analyzer. https://www.altera.com/support/support-resources/operation-and-testing/power/pow-powerplay.html. Accessed 10 Mar 2017
12. Meng, P., Jacobsen, M., Kastner, R.: FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping. In: International Conference on Field-Programmable Technology (ICFPT), pp. 37–42. IEEE (2012)
13. Nurvitadhi, E., Sheffield, D., Sim, J., Mishra, A., Venkatesh, G., Marr, D.: Accelerating binarized neural networks: comparison of FPGA, CPU, GPU, and ASIC. In: International Conference on Field-Programmable Technology (ICFPT), pp. 37–42. IEEE (2016)
14. Nvidia: Nvidia® jetson™: the embedded platform for autonomous everything. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html. Accessed 10 Mar 2017

15. Struyf, L., De Beugher, S., Van Uytsel, D.H., Kanters, F., Goedemé, T.: The battle of the giants: a case study of GPU vs FPGA optimisation for real-time image processing. In: Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS), vol. 1, pp. 112–119. VISIGRAPP (2014)
16. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)