TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitsysteme

Heterogeneous Computing for Advanced Driver Assistance Systems

Xiebing Wang

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Daniel Cremers
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. habil. Alois Knoll
	2.	Assistant Prof. Xuehai Qian, Ph.D.
	3.	Prof. Dr. Kai Huang

Die Dissertation wurde am 25.04.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.09.2019 angenommen.

Abstract

Advanced Driver Assistance Systems (ADAS) is an indispensable functionality in state-of-the-art intelligent cars and the deployment of ADAS in automated driving vehicles would become a standard in the near future. Current research and development of ADAS still faces several problems. First of all, the huge amount of perception data captured by massive vehicular sensors have posed severe computation challenge for the implementation of real-time ADAS applications. Secondly, conventional automotive Electronic Control Units (ECUs) have to cope with the knotty issues such as technology discontinuation and the consequent tedious hardware/software (HW/SW) maintenance. Lastly, ADAS should be seamlessly shifted towards a mixed and scalable system in which safety, security, and real-time critical components must coexist with the less critical counterparts, while next-generation computation resources can still be added flexibly so as to provide sufficient computing capacity.

This thesis gives a systematic study of applying the emerging heterogeneous computing techniques to the design of an automated driving module and the implementation of real-time ADAS applications. First of all, this thesis proposes a high-performance and heterogeneous ECU called h^{2}ECU, for automated driving. By incorporating multiple Multi-Processor System-on-Chips (MPSoCs), Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) into the module, h^{2}ECU can provide sufficient computing power while maintaining high scalability. With this platform, several typical ADAS applications are customized to investigate the performance and energy tradeoff in the heterogeneous system. Subsequently, this thesis gives a detailed procedure that helps guide the performance optimization of parallelized ADAS applications in the heterogeneous system. Last but not least, a hybrid framework that combines source-level static code analysis and trace-based dynamic simulation is proposed to fast and accurately predict the performance of running parallel kernels on the representative GPU accelerator. Based on this framework, this thesis further gives a solution to efficiently prune the vast program design space in order to locate the optimal program design configurations.

Zusammenfassung

Advanced Driver Assistance Systems (ADAS) ist eine unverzichtbare Funktion in intelligenten Automobilen auf dem neuesten Stand der Technik. Der Einsatz von ADAS in automatisierten Fahrzeugen würde in naher Zukunft zum Standard werden. Die derzeitige Forschung und Entwicklung von ADAS ist nach wie vor mit mehreren Problemen verbunden. Vor allem die enorme Menge an Wahrnehmungsdaten, die von massiven Fahrzeugsensoren erfasst werden, stellte die Implementierung von Echtzeit-ADAS-Anwendungen vor große Herausforderungen. Zweitens müssen herkömmliche Automobilindustrie Elektronische Steuergeräte (ECUs) der Automobilindustrie mit den kniffligen Problemen wie der Einstellung der Technologie und der damit verbundenen langwierigen Wartung von Hardware/Software (HW/SW) fertig werden. Schließlich sollte ADAS nahtlos auf ein gemischtes und skalierbares System umgestellt werden, in dem Sicherheit und kritische Echtzeitkomponenten mit den weniger kritischen Pendants zusammenleben müssen, während die Rechenressourcen der nächsten Generation noch flexibel hinzugefügt werden können, um ausreichend Rechenleistung bereitzustellen Kapazität.

In dieser Arbeit wird systematisch untersucht, wie die aufkommenden heterogenen Computertechniken auf den Entwurf eines automatisierten Fahrmoduls und die Implementierung von Echtzeit-ADAS-Anwendungen angewendet werden können. In dieser Arbeit wird zunächst ein hochleistungsfähiges und heterogenes Steuergerät mit dem Namen h^{2}ECU für automatisiertes Fahren vorgeschlagen. Durch die Integration mehrerer Multi-Prozessor-System-on-Chips (MPSoCs), Grafikprozessoreinheiten (GPUs) und Feldprogrammierbare Gate-Arrays (FPGAs) in das Modul kann das Steuergerät h^{2}ECU eine ausreichende Rechenleistung bei hoher Skalierbarkeit bereitstellen. Mit dieser Plattform werden mehrere typische ADAS-Anwendungen angepasst, um die Leistung und den Energiekompromiss im heterogenen System zu untersuchen. Anschließend wird in dieser Arbeit ein detailliertes Verfahren beschrieben, das die Leistungsoptimierung parallelisierter ADAS-Anwendungen im heterogenen System unterstützt. Zu guter Letzt wird ein hybrider Rahmen vorgeschlagen, das statische Quellcode-Analyse und dynamische Simulation auf Basis von Spuren kombiniert, um die Leistung von parallelen Kerneln auf dem repräsentativen GPU-Beschleuniger schnell und genau vorherzusagen. Basierend auf diesem Rahmen bietet diese Arbeit eine Lösung, um den großen Programmdesignbereich effizient zu beschneiden, um die optimalen Programmdesignkonfigurationen zu finden.

Acknowledgements

Time flies. Four years of the PhD study has thoroughly remoulded myself and endowed me with the dedication to scientific research and the passion to life. I would like to give my sincere thanks to those who help and accompany me during my PhD life.

The foremost thanks go to my supervisor, Prof. Dr. Alois Knoll, for offering me the opportunity of pursuing the PhD study and giving me continuous guidance and encouragement on my research topic. Without his support, this thesis would not have been possible.

I would also like to thank Prof. Dr. Kai Huang for his inspirations and constructive suggestions on the research direction.

I would like to thank: Dr. Biao Hu, Dr. Long Cheng, Dr. Gang Chen, Dr. Guang Chen, Dr. Feihu Zhang, and Dr. Caixia Cai for their valuable advices on my research; Zhenshan Bing, Mingchuan Zhou, Xiang Gao, and Zhuangyi Jiang for their companionship.

Special thanks go to Amy Bücherl, Ute Lomp, Gertrud Eberl, and all my former and current colleagues in the Chair of Robotics, Artificial Intelligence and Real-time Systems for their help and support.

Last but not the least, I would like to thank my beloved family for supporting me spiritually throughout my PhD study and my life in general.

The work presented in this thesis is supported in part by the China Scholarship Council (CSC) under the Grant Number 201506270152. This support is gratefully acknowledged.

Contents

List of Abbreviations

ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance Systems
BEV	Bird-Eye View
CAN	Controller Area Network
CES	Consumer Electronics Show
CFG	Control Flow Graph
CNN	Convolutional Neural Network
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DIP	Digital Image Processing
ECU	Electronic Control Unit
EMS	Engine Management System
EPS	Electric Power Steering
ESC	Electronic Speed Control
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCDC	Grand Cooperative Driving Challenge

LIST OF ABBREVIATIONS

GPS	Global Positioning System
GPU	Graphic Processing Unit
HD	High Definition
HPC	High Performance Computing
HW	hardware
ICD	Installable Client Driver
ICT	Information and Communication Technology
IPM	Inverse Perspective Mapping
IR	Intermediate Representation
LDA	Lane Detection Algorithm
LDW	Lane Departure Warning
LIDAR	LIght Detection And Ranging
LKA	Lane Keep Assistance
LOC	Lines Of Code
LRU	Least Recently Used
MPSoC	Multi-Processor System-on-Chip
OpenCL	Open Computing Language
PCA	Principle Component Analysis
PCIe	PCI Express interface
RANSAC	RANdom SAmple Consensus
RLD	Road Lane Detection
ROI	Region Of Interest
RTE	Run Time Environment

SE	Scalar Evolution
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SSA	Static Single Assignment
SURF	Speeded Up Robust Features
SW	software
TSR	Traffic Sign Recognition
WPM	Warp Perspective Mapping

List of Figures

List of Tables

Chapter 1

Introduction

Self-driving cars would be possible in the foreseeable future, with the maturity of multi-sensor fusion technology and the ever-increasing data processing capability. The development of electric vehicles has become a consensus for academia and industry, not merely because of the advantages, such as energy conservation and environment protection, over the conventional fuel cars, but also due to the reason that electric vehicles is able to serve as the infrastructure for intelligent transportation and autonomous driving. However, current research and development of autonomous driving still stays at the early stage as it faces several problems including technology, ethics, law, and management issues. Before deploying a real self-driving car, on modern vehicles a more common way is to use an embedded but powerful module to assist drivers in driving and decision-making. This module is called the Advanced Driver Assistance Systems (ADAS).

ADAS is originally developed to adapt vehicle systems for safety and better driving [4]. Specifically, the system collects environmental data by use of miscellaneous vehicular sensors (seen in Figure 1.1) and then processes the captured data as real-time as possible so as to make an evaluation of current vehicle runtime status. Afterwards, with the aid of different control strategies, the system either takes emergency measures by itself when necessary or gives response to drivers to assist their driving decisions. In this scenario, the time constraint is extremely strict and therefore hardware components with high computing power are required. As seen in Figure 1.1, typical ADAS essentially includes several functionalities, such as Adaptive Cruise Control (ACC), Lane Departure Warning (LDW), Lane Keep Assistance (LKA), Traffic Sign Recognition (TSR), etc.

The electronic component count and associated wiring content within modern vehicles have skyrocketed as the automobile continues its transformation into an electronic computing system. Nowadays a typical car contains dozens of Electronic Control Units (ECUs) while the premium ones could have more than 100 ECUs [5] [6]. These ECUs are used for diverse functionalities such as driving, park-

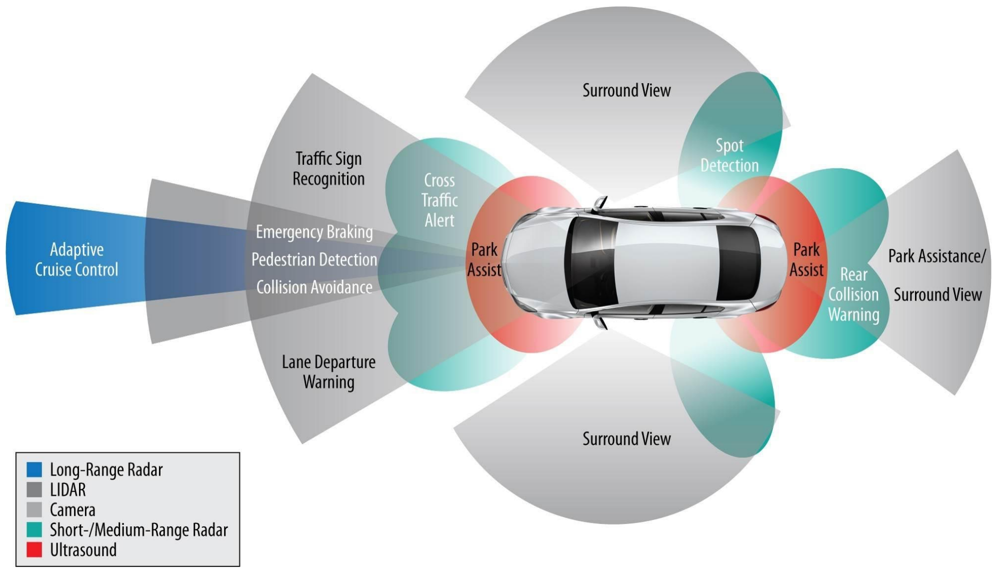


Figure 1.1: Overview of ADAS features and functionalities (figure from [1]).

ing, safety, air conditioning, navigation, and in-car entertainment, etc. This rapidly growing number of ECUs per vehicle has caused a paradigm shift in Information and Communication Technology (ICT) architectures by reversing the growth trend of dedicated ECUs for specific functions towards integrating more and more disparate functions into one or a few control units. This is the so-called ECU consolidation. The trend is not only adding more ECUs but also more computing power. Such consolidated ECUs must provide not only multiple functionalities but also significantly more performance in terms of computation resources. For instance, ADAS requires more and more computation resources as massive multi-sensor information needs to be aggregated and efficiently processed, within a short time period, to assist in-car and on-road safety control.

From an overall perspective, the automobile is a progressively complex system and it is posing a big threat to utilize limited computing power to deal with various real-time and safety-critical tasks on the vehicles. To improve on-board computation capacity, lots of early efforts have been paid by means of continuously increasing the quantity and improving the quality of the ECU components. However, this proved to be both unrealistic and minimal due to the limitation of financial cost and Amdahl's law [7]. Along with the widespread use of High Performance Computing (HPC) techniques, heterogeneous computing becomes a feasible method to solve this computation bottleneck since it is highly flexible and scalable. Heterogeneous computing system is defined as a unified platform on which different kinds of processors are integrated to leverage their unique capabilities. As accelerators or co-processors for Central Processing Units (CPUs), Field Programmable Gate Arrays (FPGAs) and

Graphic Processing Units (GPUs) are commonly used in heterogeneous systems due to their outstanding performance in high-speed computing and task parallelism.

1.1 Motivations

ADAS is an indispensable functionality in contemporary intelligent cars and the deployment of ADAS in automated driving vehicles would become a standard in the near future. State-of-the-art research and development of ADAS still faces several challenges:

- *Performance demand.* In 2015, each vehicle has an average of 60-100 sensors on board and this number is projected to reach as many as 200 sensors per car by 2020 [8]. These sensors would produce a huge amount of car- and road-related data and how to efficiently process these data in order to ensure real-time constraint becomes an urgent issue for the development of next-generation ADAS products.
- *Scalability.* Conventional vehicle contains lots of ECUs with dedicated functionalities to handle safety, security, and real-time critical tasks. However, the number of ECUs is gradually exceeding the finite space able to house all the electronics and associated cabling required to connect everything together [9]. Future ECU consolidation requires not only combing more ECUs but also adding scalable hardware components that can be easily adapted for ease of technology update and product upgrade.
- *Hardware/Software (HW/SW) maintenance.* Over the production time of a car model, not all the ECUs originally chosen are available in the market over the whole production period. Some of them will no longer be produced and have to be replaced by newer counterparts due to discontinuation of an ECU's specific technology [5]. Moreover, over the past 30 years, the software has evolved from nearly zero to tens of millions of Lines Of Code (LOC). For a consolidated ECU, how to cope with this trend still remains a pendent issue.

For ADAS applications, guaranteeing real-time performance is crucial 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 of efficiently processing 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 [10], which would incorporate Xeon Phi processors and Cyclone V Soc FPGA, to accelerate automotive computing applications. 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 [11] [12] [13], both GPUs and FPGAs are ready to be deployed in automated driving systems, due to their significantly higher computational capacity and lower research and development cost compared to the dedicated ECU/ASIC-implemented counterparts. Meanwhile, during the last decade, utilization 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 the popular use of Open Computing Language (OpenCL) [14]. On the basis of this programming model, different accelerators can leverage their respective advantages to complete the computational tasks in a collaborative way. Especially for such a heterogeneous computing system, how to consider the real-time performance and low energy tradeoff would be a very interesting topic [15].

The motivation of this thesis is to give a systematic study of applying heterogeneous computing techniques to the design of an automated driving module and the implementation of real-time ADAS applications. In this thesis, a rich set of COTS components are used for the development of ADAS applications, due to the following reasons: ① The time-to-market and development cost of COTS devices is much lower than the traditional dedicated ECU/ASIC implementations. ② COTS components can provide sufficient computing power and the performance per price is magnitude higher than automotive-specialized ones [16]. ③ The wide support of COTS devices for cross-platform programming has promoted program developers and board vendors to pay great attention to exploring the opportunity of applying heterogeneous computing to the development of future ADAS.

This thesis is intended to give partial answers to the following questions:

1. How could future ADAS applications benefit from heterogeneous computing?
2. How to apply heterogeneous computing to the development of future ADAS?
3. What needs to be taken into consideration, from a programmer's perspective, when the heterogeneous computing architecture is deployed to accelerate ADAS applications?

1.2 Thesis Contributions

With the challenges and questions in Section 1.1 in mind, this thesis first investigates the feasibility of using heterogeneous computing architecture to accelerate parallel ADAS applications. The overview

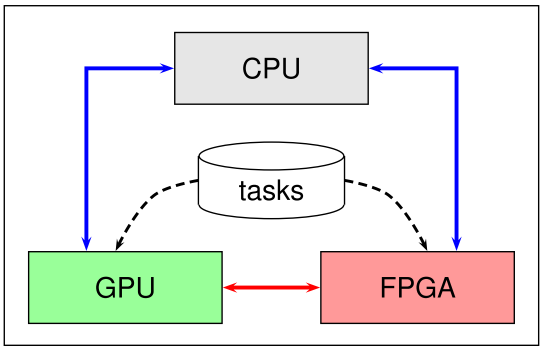


Figure 1.2: Overview of the heterogeneous system studied in this thesis. Dashed single-arrow lines are the task assignment flow, while the solid double-arrow lines refer to data communication between different types of processors or accelerators.

of the heterogeneous system studied in this thesis is shown in Figure 1.2. In this system, the CPU acts as the host scheduler and the hardware accelerators (GPU, FPGA) are used to accelerate the computational tasks. The tasks are typical ADAS applications, such as lane detection, pedestrian detection, traffic sign recognition, etc. During the processing of the tasks, the host CPU can assign the workloads to either of the accelerators, based on the scheduling policy and the runtime status of the system. Obviously in this system, how computational workloads are formed, scheduled, and executed on the hardware accelerators is critical to the overall system runtime performance.

First of all, this thesis proposes to use a high-performance and heterogeneous ECU called h^{2}ECU, for automated driving. By incorporating multiple Multi-Processor System-on-Chips (MPSoCs), GPUs, and FPGAs into the module, h^{2}ECU can provide sufficient computing power, while maintaining high scalability. With this platform, several typical ADAS applications are then customized to investigate the performance and energy tradeoff in the heterogeneous system. Evaluation results show that heterogeneous computing is a promising solution for the research and development of future ADAS. Subsequently, this thesis gives a detailed procedure that helps guide the performance optimization of parallelized ADAS applications in the FPGA-GPU combined heterogeneous system. Last but not least, a hybrid framework that combines source-level static code analysis and trace-based dynamic simulation is proposed to fast and accurately predict the performance of running parallel kernels on the representative GPU accelerator. Based on this framework, this thesis further gives a solution to efficiently prune the vast program design space in order to locate the optimal program design configurations. The contributions of this thesis can be summarized as follows:

1. This thesis proposes to use a novel ECU architecture called h^{2}ECU for future automated driving vehicles. The architecture is universal w.r.t. functionality, i.e., the different functionalities

scattered on different ECUs can be assembled into this h^{2}ECU. By integrating multiple hardware accelerators, the h^{2}ECU can provide sufficient computing power. In addition, adding new functionality is cost-free in terms of computing resources.

2. Typical lane detection algorithms are studied to probe the feasibility of using FPGA-GPU heterogeneous architecture for accelerating ADAS applications. Specifically, two different implementations of the lane detection algorithm are presented and afterwards customized into data-level parallel programs to enable the executions in heterogeneous context. Experimental results reveal that the heterogeneous architecture resolves both the performance and energy bottlenecks caused when using homogeneous accelerators.

3. A detailed procedure is proposed to help guide the performance optimization of parallelized ADAS applications in an FPGA-GPU combined heterogeneous system. The procedure consists of one intra-accelerator and two inter-accelerator sub-optimization methods, as well as both FPGA-specific and application-oriented optimization strategies, to boost the program runtime performance.

4. A hybrid framework that combines source-level static code analysis and trace-based dynamic simulation is given to fast and accurately predict the performance of running parallel kernels on the representative GPU accelerator. The high-level source code is analyzed to extract the kernel execution trace, which is used to dynamically mimic the kernel execution behavior to deduce the kernel execution time. The framework requires no prior knowledge about hardware performance counter metrics or pre-executed measurement results.

5. Based on the hybrid performance estimation framework, this thesis further gives a hybrid search framework to efficiently prune the vast program design space of OpenCL applications in order to locate the optimal program design configurations. By combing both static analysis and dynamical simulation methods, the framework can efficiently prune the program design space and notably it needs no program runs to find the optimal or near-optimal configurations.

1.3 Thesis Structure

The structure of this thesis is depicted in Figure 1.3. Chapter 1 summarizes the motivations and contributions of this thesis. Chapter 2 and 3 illustrate the design of the h^{2}ECU platform and the development of the ADAS applications, respectively. Afterwards, Chapter 4 performs the performance analysis of

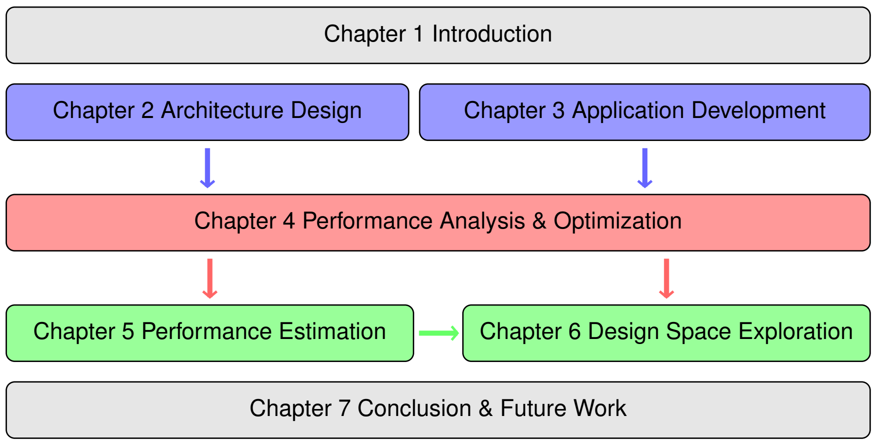

Chapter 1 Introduction
Chapter 2 Architecture Design
Chapter 3 Application Development
Chapter 4 Performance Analysis & Optimization
Chapter 5 Performance Estimation
Chapter 6 Design Space Exploration
Chapter 7 Conclusion & Future Work

Figure 1.3: The structure of this thesis.

the ADAS applications when they are deployed in the heterogeneous context, and further gives a detailed performance optimization procedure. Taking GPU as the representative accelerator, Chapter 5 models the execution of the parallel kernels and proposes a hybrid performance estimation framework. Based on this framework, Chapter 6 further gives an approach to efficiently prune the program design space of OpenCL applications in order to locate the optimal program design configurations. Finally, Chapter 7 concludes with a summary of the main results presented in this thesis. It also gives an outlook on future research directions of using heterogeneous computing for ADAS development.

To be specific, the details of the subsequent chapters are listed below and their contributions are summarized as well.

Chapter 2 presents the architectural design of the proposed h^{2}ECU and its hardware implementation. We implemented a prototype evaluation board to demonstrate the proposed architecture design and we also used a real-life electric vehicle as a testbed to harness the board. We modified a COTS sport utility vehicle DFM AX7 and deployed multiple sensors which are connected via Controller Area Network (CAN) bus with the board, so that output signals can be directly transmitted and converted to control the vehicle. At last, two frequently-used ADAS applications, namely road lane detection and traffic sign recognition, are presented to verify the efficiency and high flexibility of the h^{2}ECU.

Chapter 3 provides two different implementations of the widely-used lane detection application on heterogeneous commodity hardware. This chapter illustrates in details how the lane detection tasks are abstracted, implemented, and parallelized for ease of execution on the hardware accelerators. The performance and energy efficiency of the applications are demonstrated with experiments on different COTS heterogeneous platforms.

Chapter 4 proposes a systematic procedure that helps guide the performance optimization of parallelized ADAS applications in an FPGA-GPU combined heterogeneous system. The optimization takes into consideration both intra- and inter-accelerator workload processing, as well as FPGA-specific and application-oriented optimization strategies, to enhance the program runtime performance. The optimization results are demonstrated by applying the proposed procedure to the two case study applications presented in Chapter 3.

Chapter 5 gives a hybrid framework that combines source-level analysis and trace-based simulation to predict the performance of running GPU kernels. The framework contains a loop-based bidirectional branch search algorithm to extract the kernel execution trace, and a lightweight simulator to mimic the kernel execution so as to predict the runtime performance results. The accuracy and practicability of the framework are evaluated with benchmarks and a real-world application, on four Nvidia GPUs across two generations of recent architectures.

Based on the work in Chapter 5, Chapter 6 proposes a hybrid search framework to prune the design space of work-group sizes for OpenCL kernels running on GPUs. The framework first analyzes the source code statically to filter out redundant designs with duplicated execution traces and inferior pipelines, and then produces the estimated optimal design by searching the work-group sizes that yield the minimum predicted kernel execution time. The framework does not require any program runs to find the optimal or near-optimal configurations. Experiments show that the framework can significantly reduce the program design space and generate the estimated work-group size which can deliver runtime performance comparable to that with the truly optimal configuration.

At last, Chapter 7 summarizes the main results of this thesis and puts forward future research directions of using heterogeneous computing for the development of ADAS.

Chapter 2

h^2ECU: a High-performance and Heterogeneous ECU for Automated Driving

Nowadays, the massive sensor information poses huge computation challenge for the design of a real-time automated driving module. Meanwhile, conventional automotive ECUs cannot fulfill this objective due to technology discontinuation and the consequent tedious HW/SW maintenance. As an auxiliary but essential part for autonomous driving, ADAS is one of the computing-power demanding functionalities and this system requires more and more computation resources as massive multi-sensor information needs to be aggregated and efficiently processed to better assist in-car and on-road safety control. In 2012, Continental rolled out a schedule for autonomous driving, sketching a roadmap that fully automatic driving at higher speeds and in complex driving situations could be ready for mass deployment by 2020 and 2025, respectively [17]. Therefore, an ideal ECU module that demonstrates high performance but consumes low energy is imperative for automotive computing.

Traditional ECUs are not able to meet the requirements of autonomous driving not merely due to the huge performance demand, but also with the following reasons. First of all, over the production time of a car model, not all the ECUs originally chosen are available in the market over the whole production period. Some of them will no longer be produced and have to be replaced by newer counterparts due to discontinuation of an ECU's specific technology [5]. Additionally, over the last 30 years, the amount of software has evolved from nearly zero to tens of millions of LOC. A current premium car, for instance, implements about 270 functions deployed over about 70 embedded platforms. Altogether, the software amounts to about 100MB of binary code [5]. The next generation of upper class vehicles is expected to run up to 1GB of software. The problem here is how to design an ECU to seamlessly cope with such combination of mix-critical software as processor consolidation is closely aligned with the trend towards mixed criticality systems in which safety, security, and real-time critical components must coexist with the less critical counterparts.

2.1 Overview

This chapter illustrates the software design and hardware implementation of a novel architecture called h^{2}ECU—a high-performance and heterogeneous ECU for future automated driving. The h^{2}ECU is a modular and reconfigurable architecture in which accelerators can be flexibly embedded to achieve high computation power. The spatial redundancy of computing resources allows the coexistence of both safety and non-safety critical applications, providing appropriate partitioning mechanisms. Based on this architectural design, we implement a prototype evaluation board which incorporates state-of-the-art MPSoCs, GPUs and FPGAs into a heterogeneous system. The board is designed in modular manner, where multiple GPUs and FPGAs are connected via the PCI Express interface (PCIe), depending on the required computation power. In this way, next generation of GPUs and FPGAs can be used while the ECU architecture remains unchanged. To program the on-board software, OpenCL is adopted as the programming framework. Applications programmed with OpenCL can be seamlessly exploited on both GPU and FPGA devices. Finally, two frequently-used ADAS applications are given to demonstrate the practical use of the board in real world. The main contributions of this chapter can be summarized as follows:

- This chapter proposes a novel ECU architecture called h^{2}ECU for future automated driving vehicles. The architecture is universal w.r.t. functionality, i.e., the different functionalities scattered on different ECUs can be assembled into this h^{2}ECU. By integrating multiple hardware accelerators, the h^{2}ECU can provide sufficient computing power. In addition, adding new functionality is cost-free in terms of computing resources.
- A prototype evaluation board is implemented to demonstrate the proposed architecture. Additionally, a real-life electric vehicle is modified as a testbed to harness the board. On the vehicle, multiple sensors are connected via CAN bus with the board, so that output signals can be directly transmitted and converted to control the vehicle.
- Last but not least, two frequently-used ADAS applications, namely road lane detection and traffic sign recognition, are presented to verify the efficiency and high flexibility of the h^{2}ECU. Experimental results demonsrate the high performance and flexible reconfigurability of the h^{2}ECU-based evaluation board.

The remainder of this chapter is organized as follows: Section 2.2 reviews state-of-the-art work-in-progress of on-vehicle ECUs for autonomous driving. Section 2.3 overviews the architecture, hardware components, and on-vehicle connection of the h^{2}ECU. Section 2.4 discusses evaluation results and Section 2.5 summarizes the work in this chapter.

2.2 Related Work

Most of the traditional ECU products are for commercial use and consequently they scarcely release details about the technical implementations. At present the industry is pushing ahead to the development of ADAS ECUs for large-scale deployment. At CES 2016, Qualcomm released its prototype product based on snapdragon SoC 820A [18] for next-generation automotive applications. Meanwhile NXP unveiled MPC577xK series microcontroller [19] for ADAS and industrial radar applications. At CES 2017, Intel announced its GO automotive 5G platform [10] which would incorporate Xeon Phi processors and Cyclone V Soc FPGA, while Nvidia continues its promotion of the Drive PX 2 platform [20] and DriveWorks software.

The general use of autonomous driving vehicles is still not yet mature and the majority work lies in academic field. Earlier studies about on-vehicle autonomous driving modules mainly focus on the ADAS implementation, i.e., whether autonomous driving tasks can be fulfilled by virtue of COTS components. In this case, portability, thermal constraint and power consumption issues are entirely overlooked. For instance, the Caroline [21] developed by TU Braunschweig employs an array of automotive PCs to function as the hardware platform. Similar configuration can also be found in the KTH Scania tractor unit [22], which has participated the Grand Cooperative Driving Challenge (GCDC) in 2011. Meanwhile, the KIT AnnieWAY [23], which is the winner of GCDC 2011, even uses a customized Intel Xeon-based server which incorporates two six-core CPUs and a high-end GPU. The well-known Google driverless car does not publicly reveal any information about its computing system. However, the predecessor of the Google car, the Stanford Junior [24], who won the second place of the DARPA Urban Challenge 2007, is embedded with two Intel quad core workstations. The winner of the same race, the CMU Boss [25], is equipped with ten 2.16GHz Core2 Duo processors. As can be seen, such heavy-computing components will become standard equipment for future vehicles.

Autonomous driving component should, and must cooperatively work with off-the-shelf automotive ECUs and microcontrollers, which is presented in the previous work in [26], [27], and [28]. It has been a consensus that the functionalities of conventional on-vehicle ECUs have to remain unchanged in order to guarantee the hard real-time control of the vehicle, while new modules should be introduced to handle computing-power demanding automatic driving workload. From this point of view, due to

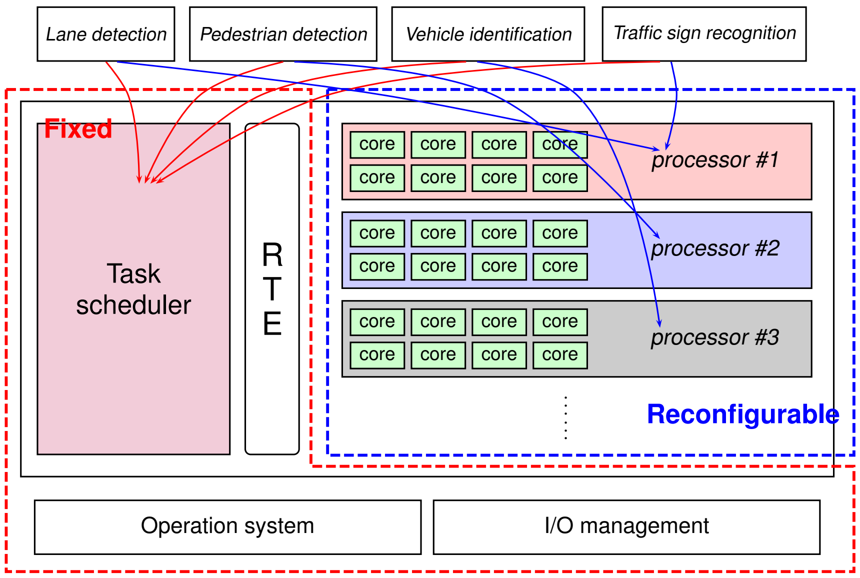


Figure 2.1: Sketch overview of h^{2}ECU. The components within red dashed line are fixed while the components within blue dashed line are reconfigurable.

the high performance and scalability, accelerators such as GPUs and FPGAs tend to be more active in future autonomous driving system since this solution can meet both the portability and real-time requirement of the autonomous driving module.

Following the above-mentioned trend, the work in this chapter proposes a new ECU architecture for future automated driving. By integrating multiple GPUs and FPGAs into an embedded platform, the execution of automated driving tasks is effectively accelerated while the whole system can be reconfigured, i.e., heterogeneous cores can be easily replaced without changing the ECU architecture. Additionally, a real-life COTS vehicle is modified and two typical ADAS applications are developed to demonstrate the proposed h^{2}ECU architecture.

2.3 System Architecture and Implementation

2.3.1 Architecture Design

In general, an autonomous driving module is used to execute ADAS applications and then translate the results to trigger the launch of lower-level hardware execution units. These applications are diverse functionalities that implicate different criticality levels, such as lane detection, pedestrian detection, vehicle identification, traffic sign recognition and so on.

Figure 2.1 reveals the sketch overview of the h^{2}ECU architecture. The system consists of a *fixed* module where fundamental runtime infrastructures (host scheduler, runtime environment, OS support, I/O management, etc.) are rooted and a *reconfigurable* module which enables the computation power tuning by utilizing flexible hardware accelerators. As an abstraction, each ADAS application is treated as a stand-alone task which would be executed on certain reconfigurable processing core, subjected to the task scheduling on the host. The host scheduler allocates computation tasks to different processing units according to both the scheduling policy and the working status of the processors. The reconfigurable processors are potentially high performance accelerators that contain hundreds of parallel cores. Normally, host CPU behaves as the scheduler while the processors are GPUs, FPGAs or CPUs as well.

To coordinate the computing of these heterogeneous processors, OpenCL is chosen as the Run Time Environment (RTE). OpenCL is an industrial standard for heterogeneous computing managed by Khronos Group [29]. By defining a high level abstraction layer for low level hardware instructions, it enables cross-platform execution from general purpose processors to massively parallel devices. In this way, autonomous driving tasks programmed with OpenCL can be seamlessly scaled and executed on a bundle of devices without any source code modification. Additionally, general-purpose operation system can be used to support the RTE and maintain the I/O communication of each component. With this design, adding new functionalities is cost-free, in terms of computing resources, as long as the RTE compatibility is met.

The benefit of this architectural design is multi-fold. First, the spatial redundancy of the computing resources allows the coexistence of both safety and non-safety critical applications, thus providing appropriate partitioning mechanisms. Secondly, this modular design shows high flexibility as multiple GPUs and FPGAs can be tuned depending on the required computing power. In this manner, next-generation GPUs and FPGAs can be used while the ECU architecture can remain unchanged. Lastly, by virtue of high performance accelerators, the execution of automated driving tasks are effectively accelerated to meet real-time constraint.

2.3.2 Hardware Implementation

Based on the proposed architecture, in the last few years several generations of the prototype product have been designed. The prototype evaluation board is a portable embedded platform where multiple MPSoCs, GPUs, and FPGAs can be assembled together. Figure 2.2 gives the top and side view of the third generation platform. The board is based on modular design so that as many components as possible can be reused in case of technology update and product upgrade. At run-time, the evaluation board is connected to another CAN communication board, which directly interacts with the auxiliary

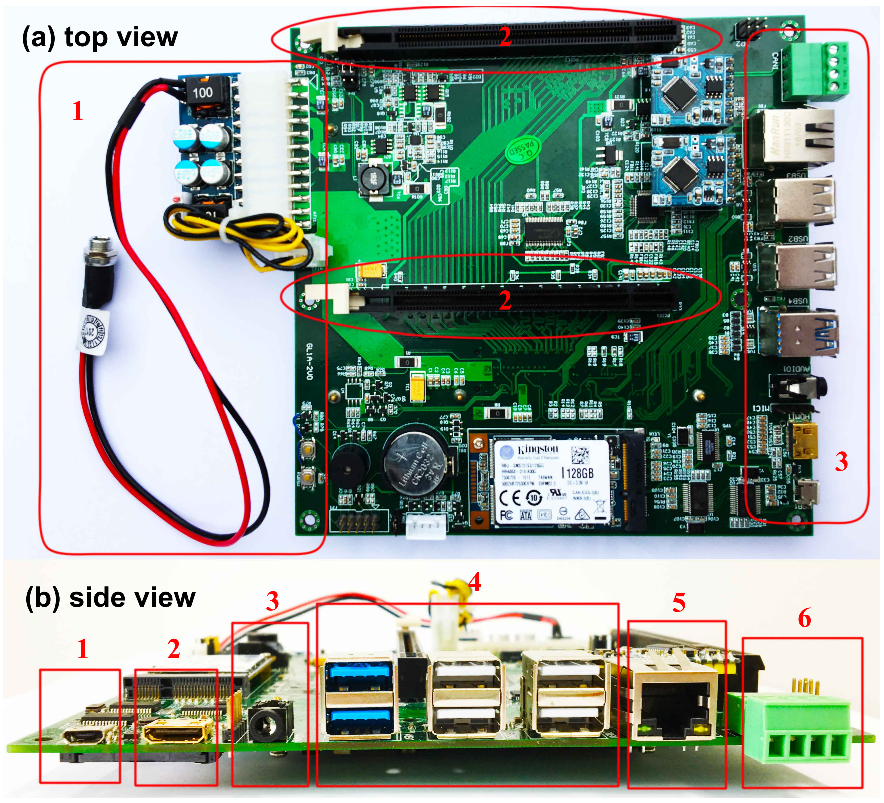


Figure 2.2: Hardware layout of the h^{2}ECU-based evaluation board. (a) Top view of the board, part 1 is power interface, part 2 are two PCIe slots and part 3 is peripheral interface. (b) Side view of the board, the listed peripheral interfaces are (1) mini-HDMI, (2) HDMI, (3) Audio port, (4) USB 3.0 and 2.0 ports, (5) Ethernet port, and (6) CAN.

controllers within a vehicle. The prototype product receives standard 12V DC as working voltage and currently the overall system is running under Linux OS.

The prototype evaluation board has five main components:

- Power supply. The power interface consists of two TPS54386 dual 3-A non-synchronous converters to ensure the stable and adequate dual power supply.
- Host scheduler. In current generation, the board employs an Intel Core^{TM} i5-3360M processor to schedule the ADAS applications. Although this power consumption is not suitable for practical use, it is sufficient for prototype development.
- Processing unit. The h^{2}ECU architecture supports a vast of PCIe-based hardware accelerators and at present several low-energy-cost GPUs and FPGAs are equipped to handle the parallel computation workload.

- RTE. In order to aggregate different devices, an Installable Client Driver (ICD) loader is constructed and this loader acts as a proxy between the user program and the actual implementations. It employs the C programming language library *dl* that is used on Linux to dynamically load libraries at runtime. In this way, OpenCL implementations of different vendors can be smoothly invoked without any conflicts.
- Data communication. The standard end-to-end PCIe data transfer protocol is used to facilitate the I/O interaction between the host CPU and computation devices.

Considering the component placement and space limit, the evaluation board consists of a 165mm$\times$ 165mm mainboard, a thermal module, and extra PCIe accelerators. The mainboard provides a power interface, which consumes external power supply, and several peripheral ports used to connect external sensors. In current generation, the mainboard contains two 16x PCIe slots which support major state-of-the-art COTS accelerators. The platform is equipped with 128GB SSD used for disk storage and 8GB DDR3 RAM used for internal storage. The host processor is located on the other side of the mainboard (not visible in Figure 2.2), due to the need of heat dissipation. The heat of the CPU is both actively and passively dissipated by the thermal module which is made up of an aluminum heat sink and a cooling fan.

2.3.3 On-vehicle Connection

We refitted COTS sport utility vehicle DFM AX7 to support the use of the evaluation board. Figure 2.3 gives the logic layout of this testbed vehicle. The testbed contains mainly four layers:

- The *upper layer* consists of a series of external sensors, such as ultrasound sensor, lidar sensor, stereo camera, etc. In this layer, the sensors capture the environmental information around the vehicle for further processing.
- The *h^{2}ECU-based platform* is used to handle ADAS tasks real-timely and gives response signal to the next layer.
- The *middle layer* includes an array of auxiliary controllers aiming at Engine Management System (EMS), Electric Power Steering (EPS), Electronic Speed Control (ESC), etc. These subcontrollers receive the command signals from the h^{2}ECU-based platform and transmit them to the microcontrollers and hardware components.
- The *lower layer* is made up of ① microcontrollers which correspond to the subcontrollers in middle layer and ② lower-level execution units like engine, steering motor, brake solenoid valve,

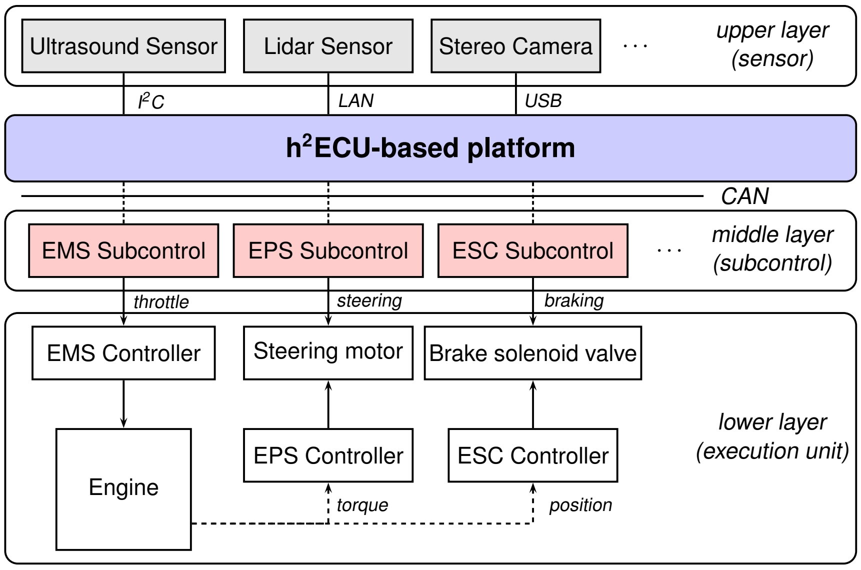


Figure 2.3: Abstraction of the layout of the testbed vehicle. The upper layer counts as the input for the h^{2}ECU-based platform. The ECU platform performs data processing and generates outputs to the middle layer, which drives the concrete execution units in the lower layer.

etc. This layer controls the actual driving of the vehicle.

In the upper layer, the sensors acquire the environmental information and then transmit the data to the h^{2}ECU-based platform. The ECU performs a series of ADAS algorithms and outputs the result, in form of command signals, to the middle layer. Afterwards each auxiliary controller invokes specific execution unit in the lower layer to control the driving of the vehicle. For instance, in Figure 2.3 the EMS controller receives throttle commands from the EMS subcontrol and then regulate the engine. Then the consequent torque and position signals generated by steering wheel and pedal are received by EPS and ECS controller, respectively. Subsequently EPS and ECS controller drive the steering motor and brake solenoid valve, together with the steering and braking commands from the EPS and ECS subcontrollers.

The connection protocols between the upper sensors and the h^{2}ECU-based platform are miscellaneous, while the communication between the ECU and the auxiliary controllers is via CAN bus.

2.4 Evaluation and Discussion

2.4.1 Evaluation Setup

Several low-end GPUs and FPGAs are integrated into the evaluation board to ensure sufficient computing performance with rather low energy cost. In particular, Nvidia Quadro K600, Quadro K620 and Nallatech PCIe-385N FPGA are used as the test PCIe accelerators. To demonstrate the proposed h^{2}ECU architecture and the use of the evaluation board, two commonly used ADAS applications are employed on the prototype product. The details of the applications are described as follows.

2.4.1.1 Road Lane Detection (RLD)

This application is used to detect the lane markings while the car is driving along the road. The algorithm processes video stream captured by a camera on a moving vehicle and highlights the positions of the lanes in the output stream. First of all, the image frames are pre-processed and transformed into gray-scale format so that the pixel intensity can be calculated. Then, by calculating the pixel weights of the lines from a randomly sampled line set, each of the lane markers is detected by selecting the line with the highest pixel weight. Afterwards the lanes are tracked by applying a particle filter over the candidate lines from the previous frame, to predict the positions of lanes in the current frame. The details of this algorithm can be referred in [30].

In this application, the lane markings in each frame are either detected by the pixel weight ranking of the candidate lines or tracked by virtue of the particle filter which predicts the positions of the lanes based on the position information of the previous detected frame. This functionality is implemented by an OpenCL kernel which can be computed across the different accelerators.

A series of videos recorded in different scenarios is used to reveal real-life road condition. Table 2.1 gives detailed information about the test videos. These videos represent various road situations

Table 2.1: Detailed information of the test videos for RLD.

Videos	Resolution	Total frames	Scenario
1	480 × 320	2287	broken lane
2	480 × 360	4601	crossing lane
3	640 × 360	899	dark light
4	640 × 360	1289	rural area
5	640 × 480	232	blur lane
6	640 × 480	250	bus view
7	640 × 480	337	street shade
8	640 × 480	406	blur lane
9	640 × 480	1718	high way
10	640 × 480	1897	broken lane
11	640 × 480	2654	heavy traffic
12	640 × 480	2799	night highway
13	640 × 480	3056	street road
14	640 × 480	4944	light disturbance
15	640 × 480	4992	night
16	1920 × 1080	1871	high way

including in day and night, with heavy traffic, with blurred and broken lines, in street and highway, in urban and rural areas, etc. To demonstrate the modular and heterogeneous features of h^{2}ECU, the experiment is conducted with different configurations where either GPU, FPGA or both are used to consume the kernel tasks. For the GPU-FPGA heterogeneous executions, a dynamic work load balance policy from [31] is used to automatically accelerate the computation. For each configuration, each video is run 10 times and the overall results are collected and averaged.

2.4.1.2 Traffic Sign Recognition (TSR)

This application is used to detect and recognize the traffic signs that appear in the images recorded via the in-car camera. For the detection, based on the Haar-like features extracted from the image, a stage classifier is established and used to judge whether the content of a scaled scanning window within the image is a traffic sign or not. By performing a series of stage classifiers at different hierarchies, an AdaBoost cascade classifier is subsequently generated to collect the classification results of each stage classifier. In this case, the target image window is deemed as a traffic sign only if all the stage classifiers give a positive value. As for the traffic sign recognition, firstly a Gabor filter is adopted to extract features from the detected image windows. Then these Gabor features are rarefied via Principle Component Analysis (PCA) of the feature dimensions. Finally the traffic signs are distinguished by means of linear discrimination analysis and template matching.

The Haar-like feature extraction is observed as the performance bottleneck which can be parallelized to decrease the execution latency. To demonstrate the high performance of the h^{2}ECU-based platform, an OpenCL kernel is designed for the Haar-like feature object detection. As comparison, a normal implementation of traffic sign recognition via Haar-like feature extraction is customized as the baseline and another implementation using OpenCV API function cvHaarDetectObjects is proposed to showcase the speedup.

For this algorithm, GPU is used as the accelerator, due to the reason that the OpenCL SDK for Altera FPGA v13.1 does not support images. During the test, the algorithm is evaluated with inputs as images under different resolutions, ranging from 160×120 to 1920×1080. For each implementation, the algorithm is run 30 times and the final results are averaged.

2.4.2 Results and Analysis

2.4.2.1 Road Lane Detection (RLD)

The performance of the RLD application is evaluated in terms of input video resolutions. Figure 2.4 summarizes the performance results. As shown, in all resolutions the executions on the evaluation

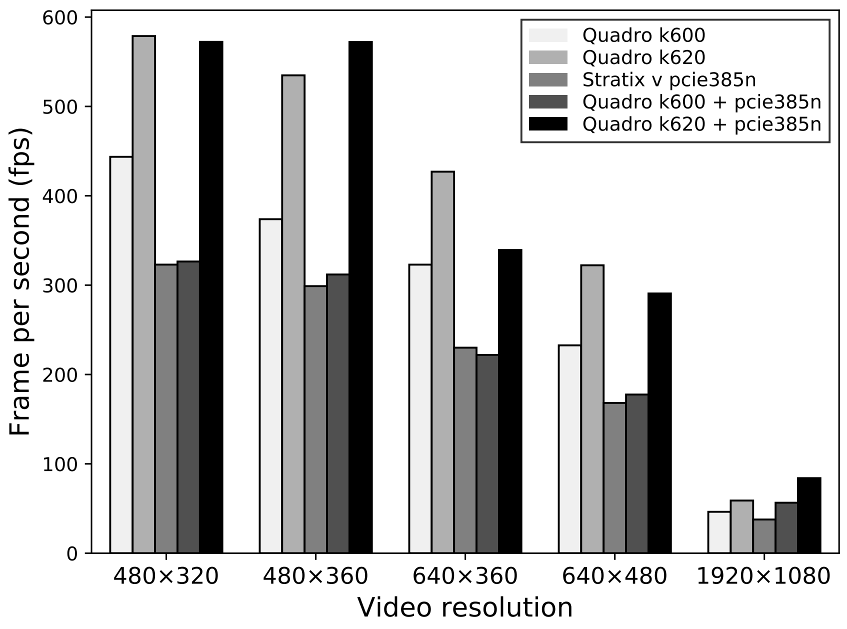


Figure 2.4: Performance of the RLD application in different video resolutions.

board can achieve real-time performance. Even for high definition 1920×1080 videos, the worst case performance is observed as 37.7275 fps, when the tasks are consumed by the single PCIe-385N FPGA card. The results reveal that the h^{2}ECU-based platform is able to real-timely handle the RLD application in a robust way.

Another interesting phenomenon shown in Figure 2.4 is the heterogeneity of h^{2}ECU. That is, the best performance does not always lie on the single-accelerator execution. The Quadro K620 GPU outperforms all other configurations in most cases when videos with 480×320, 640×360 and 640×480 resolutions are tested. However, the heterogeneous execution (Quadro K620 + PCIe-385N FPGA) turns out to be the best solution when processing 1920×1080 videos. This is extremely important as state-of-the-art ADAS requires 1920×1080 definition. This means that future ADAS applications would favor reconfigurable architecture, such as h^{2}ECU, when the single homogeneous processing core cannot meet the performance demand, not to mention the power and energy constraint.

2.4.2.2 Traffic Sign Recognition (TSR)

Table 2.2 gives the execution time of the OpenCL-version TSR application on the evaluation board. It's seen that all the executions can be completed in the second level, for different image resolutions. Although this application cannot fulfill the real-time requirement, the results are justified because the

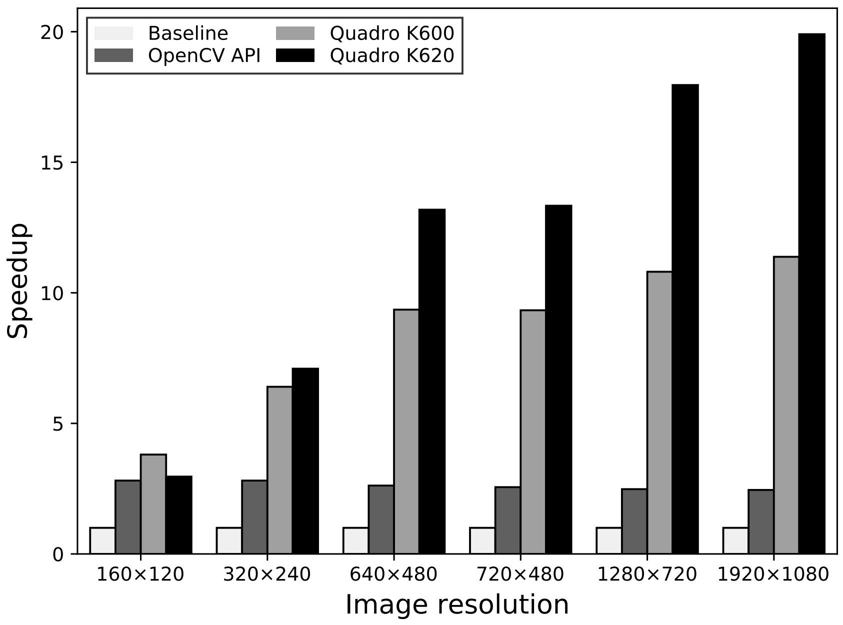


Figure 2.5: Speedup of the TSR application over customized baseline in different video resolutions.

procedure and data manipulations of TSR are far more complex than RLD. As the consequence, the computation task load of this algorithm is far larger than RLD since totally 14 stage classifiers are pipelined to do the calculation.

To better illustrate the high performance of the h^{2}ECU-based platform, Figure 2.5 gives the experimental results when comparing the performance of both the OpenCV- and OpenCL-based implementation of the TSR algorithm over the customized baseline. From the figure, it is seen that the speedup of the OpenCV implementation over the baseline is within 3×, which is rather stable across all image resolutions. However, the OpenCL implementation accelerates the application to a much larger extent and this speedup ratio becomes greater when the image resolution increases. Specifically for

Table 2.2: Execution time of the TSR application using OpenCL kernels.

Image resolution	**Execution time (ms)**	
	Quadro K600	**Quadro K620**
160 × 120	26.68965	34.29279
320 × 240	67.53467	60.93644
640 × 480	237.8679	168.7996
720 × 480	266.0748	186.1955
1280 × 720	699.5103	420.9746
1920 × 1080	1612.417	921.8685

1920×1080 image, the algorithm can gain a 11.38× and 19.90× speedup on Quadro K600 and K620, respectively. This reveals a huge potentiality of utilizing h^{2}ECU-based platform to accelerate future ADAS algorithms.

2.4.3 Discussion

Conventional ECUs for ADAS usually incorporates integrated SoCs to deal with real time workloads. This is, however, hard to scale across different platforms due to the technique upgrade and the consequent tedious software maintenance. By using reconfigurable architecture, the proposed h^{2}ECU in this chapter is able to leverage between the tradeoffs of performance demand and scalability design. Although it's hard to use state-of-the-art ADAS ECUs to test the pro and con of the platform, the experimental study depicted in this chapter throws light upon the future ADAS blueprint. The RLD application shows the performance benefit of heterogeneous architecture over state-of-the-art homogeneous counterpart, while the TSR application reveals the huge potentiality to accelerate automated driving tasks via COTS accelerators.

The current generation of the evaluation board is not yet mature and still has some drawbacks. The biggest issue lies on the power consumption of the platform. In the future, more energy-saving techniques and components are expected to be involved to address the power constraint.

2.5 Summary

This chapter proposes a new ECU architecture called h^{2}ECU and then a prototype platform is designed to facilitate the trend of its mass deployment for automated driving. The architecture consists of a fixed module in which the host can schedule the incoming tasks, and a reconfigurable module in which various COTS hardware can be flexibly inserted to accelerate the computation workload. By integrating multiple MPSoCs, GPUs, and FPGAs into the module, h^{2}ECU can provide sufficient computing power, while maintaining high scalability. The proposed architecture is universal w.r.t. functionality and cost-free w.r.t. functionality update. We implanted the evaluation board into a real-life sport utility vehicle to verify its feasibility. Finally the h^{2}ECU-based platform is tested with typical ADAS applications to reveal its high performance and flexible reconfigurability.

The h^{2}ECU presented in this chapter is intended to serve as the hardware platform for the studies in latter chapters. In the next chapter, the detailed implementations of typical ADAS applications and their deployment in heterogeneous context will be illustrated.

Chapter 3

Design of ADAS Applications on Heterogeneous Platforms

For the automotive industry, ADAS is 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 property safety.

The work in this chapter focuses on how to implement ADAS applications on heterogeneous platforms. Particularly the widely-used Lane Detection Algorithm (LDA) is chosen as a case study. As a fundamental functionality in ADAS, lane detection is a well-studied algorithm which has attracted much research attention since mid-1980's. Typically for lane detection, camera is the most frequently used sensor type not merely because of its fairly low cost, but also taking into account that roads and lanes are designed to be perceived by human drivers and the perception of visual cues is essentially the same as for human eyes. However, using camera as source of information requires large computational power to handle the amount of data, particularly to meet the state-of-the-art requirement of real-time High Definition (HD) image processing.

Apart from the performance requirement, how to ensure the robust utilization of lane detection application across various on-road scenarios is still nontrivial. For instance, to reduce noise influence and computational complexity, it is typically a pre-step to extract a Region Of Interest (ROI) within the whole image before it is actually analyzed. However, how to obtain an efficient, stable and reliable ROI is often scenario-specific and this size needs to be self-tuned if external on-road condition changes drastically. What's more, given a specific lane detection algorithm, it is often not easy-to-evaluate when taking an arbitrary road video stream as input, since camera intrinsic and extrinsic parameters are often unknown and need to be calibrated. On the one hand, this hinders the promotion of the algorithm since the third-party users or other researchers do not always assume the highly practicability of the sample videos provided by the developer. On the other hand, this prolongs the development period since it is time-consuming to construct a technically sound benchmark.

The last important issue lies in the scalability of the application, i.e., how convenient for the end-users to consider the tradeoff between the algorithmic accuracy and execution speed, and how expensive for the developers to transplant the application back and forth, for instance, from simulation environment to real world, and from one platform to another. The bottleneck here is not merely hardware configuration and code modification, but also the burdensome performance optimization across different architectures.

3.1 Overview

This chapter presents two different implementations of LDA across heterogeneous platforms. The first application [31] uses particle filter to track the detected lane positions, while the second application [32] 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 LDA in latter sections. For both applications, we customize the heterogeneous implementations from the naive parallel designs and then investigate the performance and energy tradeoff in heterogeneous context. The main contributions of this chapter can be summarized as follows:

- This chapter presents two different approaches to implement LDA and then customize their executions on heterogeneous platforms.
- For *p-LDA*, this chapter gives a lightweight workload balance scheme that can significantly increase the performance, while ensuring the low energy cost. The scheme can robustly adjust the workload in diverse road scenarios, based on the computation capacity of each accelerator in use.
- For *r-LDA*, this chapter shows how the parallel task loads are implemented and optimized so that they are mapped to the most suitable processor so as to achieve optimal performance. A detailed comparative study of using homogeneous and heterogeneous configurations to accelerate the application across different platforms is conducted.
- Taking real-life road scenarios as input, a series of experiments are performed to execute the LDA implementations on heterogeneous platforms. Experimental results demonstrate the necessity

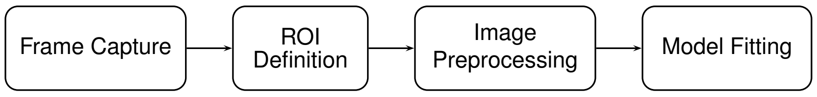


Figure 3.1: General processing flow of camera-based lane detection algorithms.

of utilizing FPGA-GPU combined heterogeneous architecture for future deployment of ADAS applications.

The remainder of this chapter is organized as follows: Section 3.2 reviews state-of-the-art research about lane detection and its acceleration on commodity hardware. Section 3.3 and 3.4 describe in detail the implementations of the two LDAs and their parallelization in the heterogeneous systems. Section 3.5 gives evaluation results and Section 3.6 summarizes the work in this chapter.

3.2 Related Work

3.2.1 Lane Detection Techniques

State-of-the-art methods for lane detection can be classified into two categories: camera-based methods with image stream as input and multiple sensor-based methods which combine a camera sensor and a minimum of one additional source of environmental information such as LIght Detection And Ranging (LIDAR) [33] [34], Global Positioning System (GPS) [35] or other vehicle data received via car-to-car communication. The algorithms in this chapter pertain to the camera-based methods and therefore the following paragraphs mainly review the approaches which only use camera as data source of information.

Basically camera-based approach are characterized by similar execution procedures shown in Figure 3.1. First the ROI within the camera-captured image is defined for further processing. This definition of the ROI size is often algorithm-dependent. Some approaches attempt to distinguish and extract the lanes from the whole image [36] [37] [38], while other researchers generate the ROI via either manually setting the ROI boundary [30] [39] [40] or dynamically calculating the lane area with inherent road properties and camera parameters [41] [42] [43].

After the ROI is generated, typical image processing methods are adopted to extract the features of the lane boundaries, such as color, gradients and edges, to distinguish between the markings and non-lane areas inside the image. In this step, the image on which the algorithm is operating can be raw or Bird-Eye View (BEV) images. By using raw image, the lane features can be directly extracted and mapped back to the image so as to highlight the candidate lanes. However, the perspective effect, i.e., the angle of view under which a scene is acquired and the distance of the objects from the camera, in fact must be taken into account in order to weigh each pixel according to its information content. Current researches often use two different approaches, namely Inverse Perspective Mapping (IPM) [36] [40] [41] [44] and Warp Perspective Mapping (WPM) [45] [46], to compute a BEV image from the input image. One advantage of this perspective effect elimination is the possibility to separate the relevant part of the image below the vanishing line from the irrelevant data above the vanishing points that may noise the data without any information gain.

In contrast to IPM which uses intrinsic (focal length, optical center) as well as the extrinsic camera parameters (pitch angle, yaw angle, roll angle, height above the ground) to calculated the required transformation matrix, the WPM method is independent from the camera parameters. However, 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 are miscellaneous. The studies in [38] [39] [40] [43] used RANSAC model to perform straight line and spline fitting to refine the detect lanes. In [30] and [47], the authors proposed to use particle filter to predict the lane markings after the previous lanes are detected, while the work in [41] adopted similar Kalman tracking. The biggest advantage of RANSAC is the robust estimation of model parameters, even if the dataset contains a significant amount of outliers [48]. However, the benefit of filter-based lane tracking is the reduction of the execution time cost caused by the iterative lane fitting and detection.

3.2.2 Acceleration of ADAS Applications on Heterogeneous Platforms

State-of-the-art research does not witness too much on the acceleration of lane detection using parallel platforms. The authors in [38] used CUDA and TBB to optimize the processing pipeline of a RANSAC-based road lane detection application. They reported a processing speed of 34.8 ms per frame for 640×480 images, with a speedup of around $3.55\times$. The work in [49] proposed a lane departure warning system implemented on an Xilinx FPGA. They used Single Instruction Multiple Data (SIMD) structure to implement vanishing point-based parallel Hough transform and reported a real-time performance of 40 Hz. Another similar study is presented in [50], which reports the performance result of 30 frames per second.

Meanwhile, there exist plenty of researches that propose to utilize COTS components to accelerate other ADAS algorithms, such as pedestrian detection [51], audio sensing [52], traffic sign detection [53], etc. While it is a potential trend to speedup ADAS applications with parallel processors, this chapter presents a comprehensive study of accelerating lane detection with OpenCL. The data-level parallelism is adopted for FPGA and GPU devices so that their performance characteristics can be

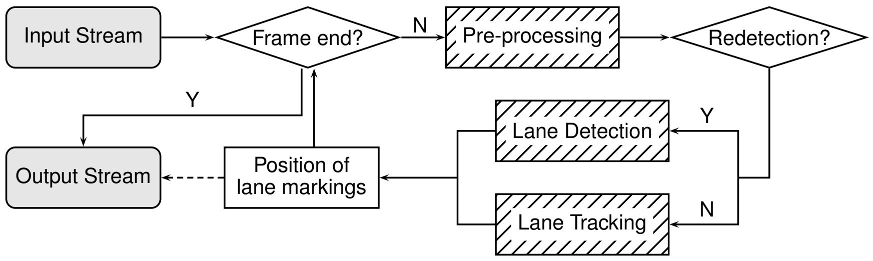


Figure 3.2: Flow chart of *p-LDA*.

directly and intuitively compared.

3.3 *p*-LDA: Particle-filter-based Lane Detection Algorithm

3.3.1 Algorithm Design

This section describes the naive design of *p-LDA* and the procedure is shown in Figure 3.2. As can be seen, the application analyzes the video stream captured by a moving vehicle frame by frame and attempts to extract the exact positions of the lane markings highlighted in output stream. For each frame, the *pre-processing* module extracts information about the lane markings via typical Digital Image Processing (DIP) techniques and then passes the processed image to the next step. Depending on whether or not the estimated state of the lane markings in previous frame can still be applied to the current frame, the image is processed either using *lane detection* module to redetect the positions of the lane markings or using *lane tracking* module to track the previous position of the lane markings.

3.3.1.1 Pre-processing

The *pre-processing* module contains four steps successively applied to the raw images. First a ROI is cropped from the raw image and only this ROI is further processed. This is reasonable since normally only a small part of the image, which shows the street and road condition, contains all the essential information for the subsequent lane detection and tracking steps. Then this ROI is transformed into a grayscale format where each pixel reflects the intensities 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 in ROI, a Sobel filter [54] is applied to the grayscaled image to extract transitions and edges in the image. The Sobel operator can produce optimal results when ① the lane markings in the image are extremely bright, ② the surrounding environment is extremely dark, and ③ there are no noises. However, in real application, images always contain noises whose sources are other road markings, signal posts at roadside, varying colors of street material and shadows, etc. These disturbances could produce additional and undesired edges in the image.

To avoid the influence of noises in real scenarios, a threshold is used to set the intensity of all pixels whose gradient falls below the threshold as zero and all pixels who has an intensity above the threshold as a maximum value. With thresholding the strong edges created by lane markings are further emphasized and also the computation overhead of the following lane detection and tracking task is alleviated since much of the noises are erased.

3.3.1.2 Lane Detection

Given a ROI with a band of pixels indicating the positions of the lane markings, the *lane detection* module has to extract this information and formulate it. It is assumed that the lane markings within the ROI are straight lines. This assumption can always hold since ① the ROI captures only a small part of the street ahead the vehicles and ② in the vast majority of cases the roads are straight or just moderately bent (i.e. with a small curvature). Moreover, when the road reveals a sharp bend, then the ROI can be horizontally split into several subblocks within which each has a straight lane marking.

Following this assumption, a lane marking can be represented as a 2-tuple $X : (x_{top}, x_{bottom})$, where x_{top} and x_{bottom} are the x-coordinates of the two points which intersect with the top and bottom border of the ROI. Then the slope of the lane s and any other point on the lane with y-coordinate value y_n can be determined by

$$\begin{aligned} s &= \frac{x_{bottom} - x_{top}}{ROI\_HEIGHT} \\ x_n &= x_{top} + s \cdot y_n. \end{aligned} \tag{3.1}$$

Note that Equation (3.1) holds if and only if the lane intersects with the top and bottom borders of the ROI. Therefore to detect lane markings crossing with the left and right borders of the ROI, the ROI is then split into proper *subregions* where each lane marking intersects only with the top and bottom borders of one subregion.

During the detection, a set of *candidate lines* are randomly generated via assigning random values from a normal distribution to the elements in the 2-tuple set $\mathbb{X} = \{X_1, X_2, \cdots, X_n\}$, 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. This weight is calculated by summing up the intensities of ① all the pixels which belong to part of the candidate line overlapping the ROI and ② pixels in an adjustable neighborhood around the line, accounting for the width of real lane markings. Hence a weight set $\mathbb{W} = \{w_1, w_2, \cdots, w_n\}$ is obtained. With this set, the line with the highest weight is chosen as the *best line* and certain number of candidate lines are also reserved as *good lines*, which would be further used in the *lane tracking* module.

3.3.1.3 Lane Tracking

In this step, to find the position of lane markings in current frame, both the ROI of the current frame generated from the *pre-processing* module and the *best line* and *good lines* of the previous frame calculated from the *lane detection* module are considered. The particle filter [55] is adopted to handle the lane tracking. Theoretically speaking, given a series of observable explicit states Y, the filter tries to determine the posterior density distribution $P(X|Y)$ of an implicit state X. With Bayesian recursion equation, the posterior density distribution is calculated as

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}, \tag{3.2}$$

where P(X) is the prior probability density of state X, $P(Y|X)$ is the likelihood of observation Y subject to condition state X, and P(Y) is the probability of the state Y.

The particle filter extends Equation (3.2) by sampling a range of status values X_i, i.e., so-called "particles", from the prior probability distribution of state X. Each of these particles is assigned an importance weight ω_i to express the likelihood that X_i is identical with the true state X, given the observation value of Y. Hence it is yielded from Equation (3.2) that

$$\begin{aligned} P(X_i|Y) =& \frac{P(Y|X_i)P(X_i)}{P(Y)} \\ =& \frac{P(Y|X_i)P(X_i)}{\sum_{i=1}^{N} P(Y|X_i)P(X_i)} \\ =& \frac{\omega_i}{\sum_{i=1}^{N} \omega_i}, \end{aligned} \tag{3.3}$$

where N is the number of the particles.

When applying particle filter to the *lane tracking* method, the 2-tuple set can be seen as the implicit state space, the *good lines* as the set of particles, and the *best line* is used as an approximation of the observation state Y. The module mainly consists of three steps:

Prediction update. The *good lines* represent the position of lane markings in the previous frame, but are used as the prior probability distribution of lane markings in the current frame. Therefore some modifications are required and here the *good lines* are shifted with sample value from 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.

Importance weight update. The importance weight of the particles is calculated by fitting the state of the particles to Gaussian function

$$\omega_i = \frac{1}{\sigma\sqrt{2\pi}} e^{-(X_i-\mu_f)^2/2\sigma_f^2}, \tag{3.4}$$

where $\mu_f = Y$ 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 each importance weight of the particles is normalized to obtain the updated weight

$$\omega_i^{updated} = \frac{\omega_i}{\sum_{i=1}^{N} \omega_i}. \tag{3.5}$$

Resampling. This step selects particles from the updated set, according to their importance weights, to generate a new particle set, with the particle number unchanged. The purpose of this is to increase the accuracy of the lane detection and prevent a degeneration of the particle set.

3.3.1.4 Redetection Checking

The redetection checking step verifies whether the detected positions are reasonable and conform to the physical properties of the lane markings themselves. If not, additional detection step is triggered to seek the lane markings again. The criteria of redetection are as follows: ① Lane markings do not cross. ② 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. ③ There should be a minimum percentage of the lane marking within the ROI. Again this parameter is flexible and can be user-defined.

3.3.2 Parallel and Heterogeneous Implementation

3.3.2.1 Parallel Implementation

The *pre-processing* module presents a high potential of parallelization since each pixel in the ROI can perform grayscaling and thresholding manipulations by itself. Moreover, the Sobel filter only requires knowledge about nine neighbors of the processing pixel. This again implies that all the pixels can be handled independently. Thus an OpenCL kernel kernelPRE is developed to perform the pre-processing operations entirely on hardware accelerators.

As for the *lane detection* module, notice that the *candidate lines* are randomly generated and hence they are mutually independent. However the selection of the *best line* is based on the aggregated result of all the *candidate lines* and consequently should be performed only once on the host. As a result another kernel named kernelLD is implemented to sample the lines and calculate their weights.

As can be seen, similar with lane detection, the prediction and importance weight update steps in the *lane tracking* module are executed on every single particle and therefore are unrelated with others.

Algorithm 1: *p*-LDA (basic version)

```
Input: raw camera-captured video stream
Output: video stream with lanes marked
1  initialization
2  random number generation                                  ▷ KERNEL_RNG
3  while not the end frame do
4      ROI image pre-processing                              ▷ KERNEL_PRE
5      if redetection then
6          lane detection                                    ▷ KERNEL_LD
7          candidate line generation
8      else
9          lane tracking                                     ▷ KERNEL_PF
10         good line resampling
11     best line extraction and mark lanes in current frame
```

The resampling step, in contrast, relies on knowledge from the whole particle set and thus is performed on the host. Again a kernel kernelPF is used to calculate the updated results of the particles.

Furthermore, it should be noted that both the lane detection and tracking module require normally distributed random numbers to process their following tasks. In this algorithm, these numbers are generated by MWC64X [56], which is a small and fast random number generator developed for use with GPUs via OpenCL. As this task is mandatorily executed on hardware accelerators, another kernel called kernelRNG is introduced to realize it. In current work this kernel initializes a stream of random numbers and splits them with a period of 2^{40}, which allows the processing of videos lasting far more than 24 hours and even in the worst case scenario where 10^6 random numbers per frame are used.

With above four kernels, the flow chart in Figure 3.2 is abstracted as the pseudo-code shown in Algorithm 1, where the red lines (Line 2, 4, 6, and 9 in Algorithm 1) represent the kernel tasks.

3.3.2.2 Heterogeneous Implementation

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 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: ① both the executions of KERNEL_LD and KERNEL_PF use the outputs of KERNEL_RNG and KERNEL_PRE, and ② 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.

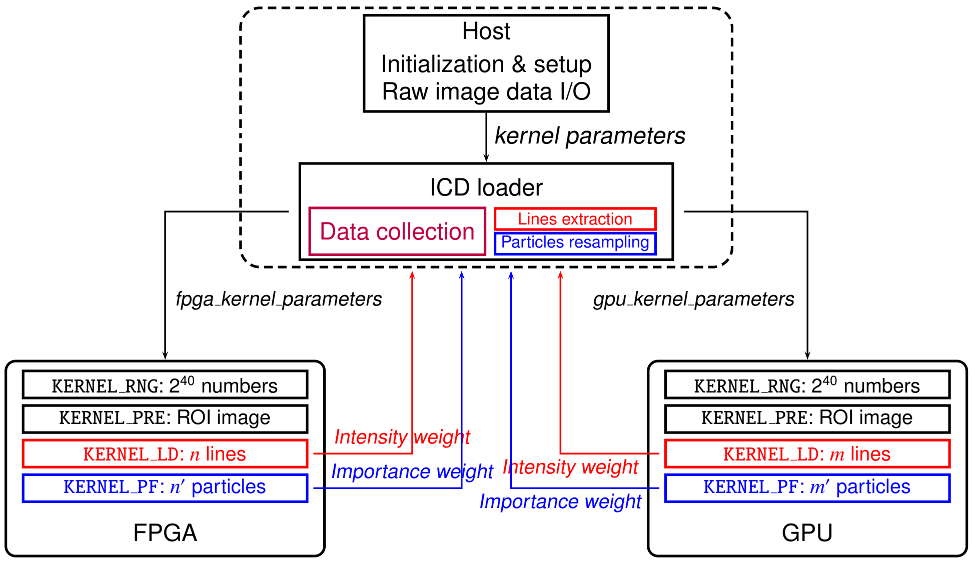


Figure 3.3: Overview of the execution of p-LDA in heterogeneous context. Red and blue items are distributed tasks on the FPGA and GPU. The italic items show the transfer of parameters.

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 p-LDA is used for the heterogeneous context and Figure 3.3 gives the overall processing procedure. In general, the host utilizes an ICD loader to coordinate the tasks executed 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 ① kernel parameters initialization and raw image I/O when the program begins, and ② 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 Figure 3.3, 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, 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

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$(t_{pf_f}, t_{pf_g}, m', n')$
```

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.

3.3.2.3 Workload Balance Scheme

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 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 section gives a time optimization based workload balance scheme for the heterogeneous *p-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 Figure 3.3), and the output is the time-optimal executions 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 (Line 1, 4, 8, and 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 (Line 10 and 14 in Algorithm 2). This scheme assumes that for

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 $W_f \leftarrow \frac{c_f}{c_f+c_g}(W_f + W_g), W_g \leftarrow \frac{c_g}{c_f+c_g}(W_f + W_g)$
```

each frame, the execution times of KERNEL_LD and KERNEL_PF are proportional to the current task load size.

3.4 *r*-LDA: Ransac-based Lane Detection Algorithm

3.4.1 Algorithm Design

This algorithm is based on the previous work in [57], which generates an IPM-based BEV image of a pre-defined, fixed ROI inside the input frame. To extract lanes in the IPM transformed image, this approach filters the image vertically by a smoothing Gaussian filter and horizontally by a second derivative Gaussian filter. Based on the filtered IPM values, a simplified Hough transformation is applied, which gives an initial guess about the position of the lane markings. Two RANSAC iterations are afterwards performed to refine the lane detection. The first step matches linear lines using previous Hough transformation data. After applying geometric checks, the second RANSAC iteration fits the lane to a third degree Bezier spline. The subsequent post-processing step tries to re-localize the matched splines in the original input image and to extend the relocated results.

Although this approach is illumination invariant and provides the capability to detect straight as well as curved lane markings, it still suffers from several crucial drawbacks: ① The pre-defined ROI only provides a limited section of the available lane marking information inside each image frame and neglects other relevant information; ② The proposed IPM resolution of the ROI is downscaled to 160×120 pixels, which further decreases the lane information; ③ As the top-view IPM transformation is based on unknown camera parameters, these values have to be guessed, resulting in an inaccurate vanishing point estimation; ④ The vanishing point does not take road changes into account by assuming constant street gradients.

The *r*-LDA tries to overcome the limitations mentioned above. As the camera parameters of the lane detection data set are unknown and have to be guessed, which results in an inaccurate transformation matrix, in order to increase the accuracy of this matrix, in this design the top-view image is processed by a WPM instead, which is based on the reference frame of the video stream. The overall design of the *r*-LDA is shown in Figure 3.4. The homography matrix used in WPM is first generated off-line by virtue

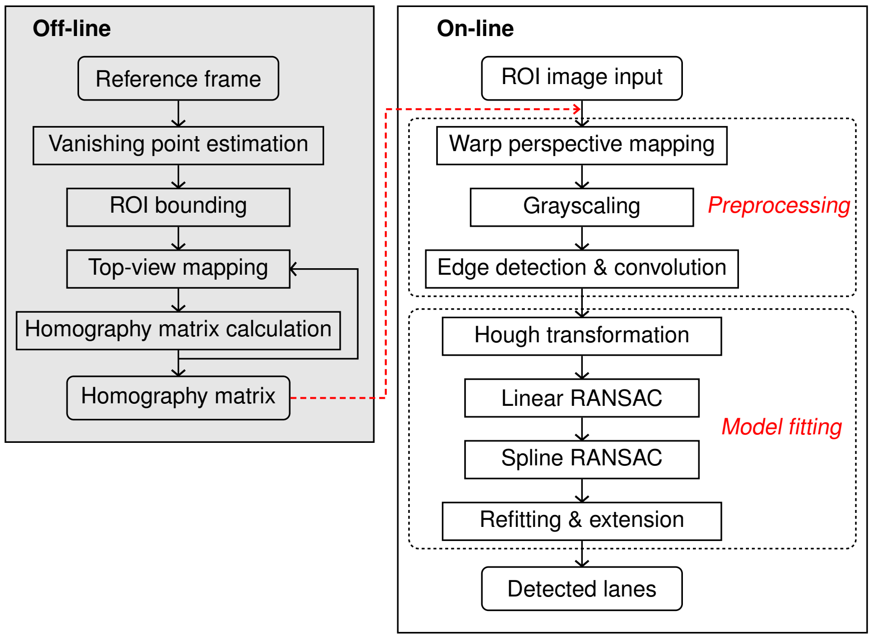


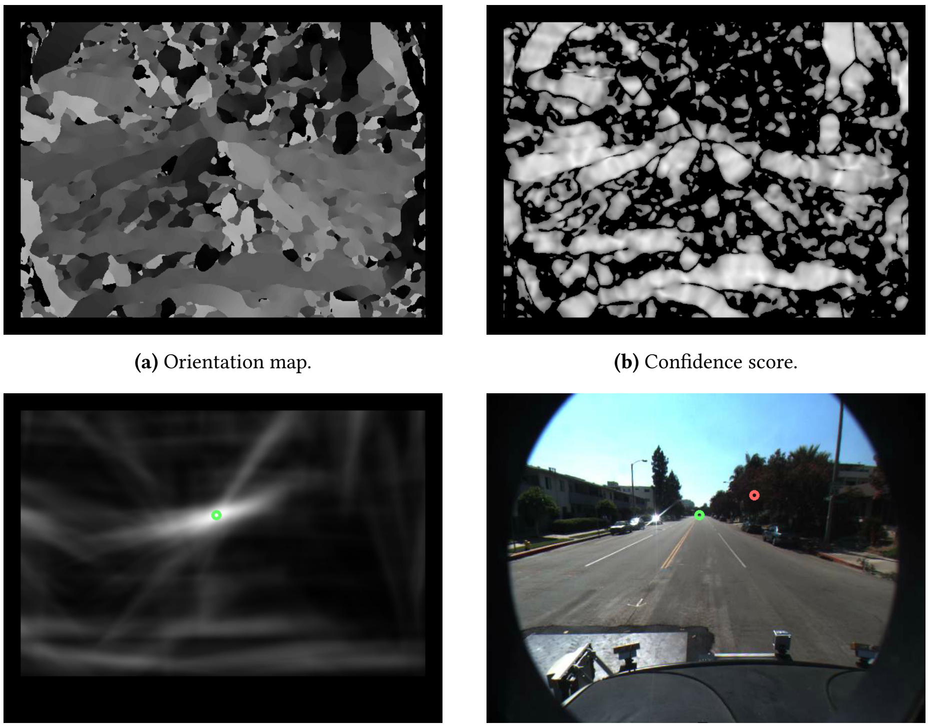
Figure 3.4: Processing flow of *r*-LDA overview. The off-line part on the left is executed only once to obtain the homography matrix. The actual detection is performed on-line iteratively over each image frame.

of the vanishing point estimation and top-view mapping of the reference frame. Subsequently, each image is processed on-line via WPM to generate the BEV image. Then the top-view ROI is grayscaled and convoluted to extract the vertical as well as quasi-vertical lane markings. Afterwards, the lane marking is refined with a simplified Hough transformation and two RANSAC fitting, which is similar to the work in [57].

The difference of the improved design with the previous work lies in the off-line homography matrix generation and the on-line WPM transformation, therefore the following sections focus on this part and illustrate in detail the implementations.

3.4.1.1 Vanishing Point Estimation

In contrast to IPM which uses intrinsic as well as the extrinsic camera parameters to calculated the required transformation matrix, the WPM method is independent from the camera parameters. However, a minimum of four corresponding point pairs in the original image as well as the transformed image are required to compute an affine matrix mapping. As the perspective transformation matrix is unknown, these correspondences cannot be calculated directly. To get the corresponding points in the raw image, the *r*-LDA uses a vanishing point based mapping from an arbitrary image to its corresponding top-view on the reference frame. To estimate the vanishing point the approach in [58] is



(a) Orientation map.

(b) Confidence score.

(c) Vanishing point voting.

(d) Estimated vanishing point.

Figure 3.5: Vanishing point estimation steps. Figure (a) shows the dominant orientation ranging from $[0, \pi)$ quantized in steps of 5°. Darker colors correspond to larger angles. Figure (b) illustrates the computed confidence score. Brighter values correspond to larger scores. The area constrained soft-voting values can be seen in Figure (c). The brightest value is estimated to be the vanishing point (green circle). Figure (d) compares the estimated vanishing point (green) to the one used by [57] based on guessed camera parameters (red).

implemented. The texture orientations are estimated by convoluting the reference image with multiple Garbor filters over 36 orientations from $[0, \pi)$ quantized with a step size of 5°. After a confidence score is calculated for the 36 values obtained per pixel coordinate for every orientation, a voting map is obtained based on the confidence scores inside a lower semicircle around each coordinate centered at the currently processed pixel. Then the final vanishing point is proposed to correspond to the largest value in the voting map. Figure 3.5 shows the result of the detected vanishing point in the reference frame. As can be seen, the position of the vanishing point (green circle in Figure 3.5d) estimated in this work is far more accurate than the one calculated by [57] (red circle in Figure 3.5d).

3.4.1.2 ROI Bounding

It is assumed there are some reasonable environment-related properties in the reference frame:

1. A minimum of two lanes are included in the reference frame, one to the left side and one to the

right side of the vanishing point.

2. The lane boundaries in the reference frame are parallel.
3. The intensity values of the lane markings are higher than that of the surrounding road surface.
4. The vanishing point lies within the reference frame.
5. The roll angle of the camera is close to zero and can be neglected.

Figure 3.6 gives the overall procedure of the ROI initialization. Due to Assumption (5), the vanishing line can be approximated as a parallel to the x axis that intersects the vanishing point. With Assumption (1) and (3), Canny edge detector is applied to extract lane boundary points. To eliminate outliers from the binarized edge data, the image below the vanishing line is sampled with straight lines centered at the vanishing point within an angle of $[0, \pi)$ between the x axis and the sampling line with an empirically defined step size of 4° (shown in Figure 3.6a). As a result, lines that overlap with a lane boundary edge score best. To compensate small vanishing point errors, this procedure iterates over an offset of three pixels to the left and to the right side of the vanishing point. The final scoring for each line is defined as the maximum response over all offsets. Based on Assumption (1), the set of sampling lines is separated into two equally sized subsets, ranging from $[0, \pi/2)$ and $[\pi/2, \pi)$, respectively, to detect one lane boundary in the left and the right part of the image according to the x coordinate of the vanishing point. Each lane marking is defined as the highest score in the corresponding subset. Additionally, an upper and lower y axis boundary (blue lines shown in Figure 3.6c) is further imposed to neglect image parts without any road information as well as unreliable areas with perspective effects.

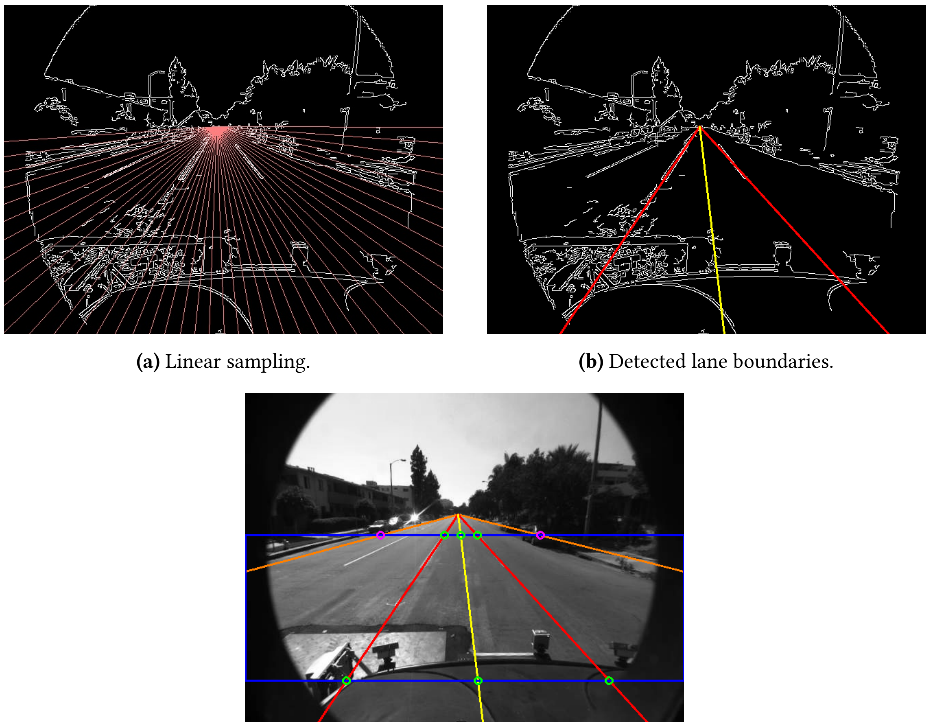
As observed in Figure 3.6c, the ROI is bounded by the upper and lower y axis limits (blue lines) and the image border crossing lines (orange lines). The y-coordinates of the upper and lower boundary are calculated by

$$\begin{aligned} U_y &= V_y + \alpha * (H - V_y) \\ L_y &= V_y + \beta * (H - V_y), \end{aligned} \tag{3.6}$$

where V_y is the y-coordinate of the vanishing point, H is the image height and α, β are the scale factors. The image border meeting points are defined by their equal y-coordinates, which is obtained from

$$B_y = U_y + \gamma * (L_y - U_y), \tag{3.7}$$

where U_y and L_y are the previously defined upper and lower y axis boundaries and γ is a parameter ranging from $[0, 1]$ to adjust the upper left and right triangles in the original image, which are neglected in the BEV as they are supposed to contain irrelevant information about the environment.



(a) Linear sampling.

(b) Detected lane boundaries.

(c) Initialized ROI.

Figure 3.6: ROI extraction from the reference frame to compute WPM. Figure (a) is the linear sampling from the previously estimated vanishing point with a step size of 4°. Figure (b) shows the detected lane boundaries (red) for the left and the right part of the image according to the vanishing point and the median line inbetween (yellow), which is required to increase the accuracy of the method. The upper and lower y axis limits (blue) as well as the extracted reference image feature points for the each lane boundary (green) and the upper BEV corners in original coordinates (violett) obtained by the defined left and right image border crossing lines (orange) can be seen in Figure (c).

With Equation (3.6) and Equation (3.7), the y-coordinate of the image border meeting points is calculated as

$$B_y = V_y + [\alpha + \gamma \times (\beta - \alpha)](H - V_y). \tag{3.8}$$

That is to say, given image height H, vanishing point y-coordinate V_y and empirically determined coefficients α, β and γ, the ROI size is adaptive to the video streams.

3.4.1.3 Top-view Mapping & Homography Matrix Adaption

The top-view mapping is separated into two steps. Initially, feature point coordinates are obtained from the reference frame to estimate 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. First the bottom corner distance in the BEV image is assumed to be a fixed value v and thereupon the homography matrix is computed. As the distance between the top-view transformed lane boundary feature points and the medium x axis (x_m) in the BEV has to be equal for the upper and lower image border according to Assumption (2), the bottom corner distance correction scaling s_c is calculated as

$$s_c = \frac{1}{n} \sum_{i=0}^{n} \frac{d_u({}^tP_i, x_m)}{d_l({}^tP_i, x_m)}, \tag{3.9}$$

where n is the number of feature points extracted from the input image and d_u, d_l are the upper and lower distances from the current feature point in BEV coordinates ${}^tP_i$ to x_m. The final bottom border distance d_f is computed as

$$d_f = 2 \times v \times s_c. \tag{3.10}$$

The corresponding points in the reference frame and the BEV image are used to compute the corrected homography matrix. To compensate vanishing point fluctuations due to changing road gradients, in each frame the homography matrix is adapted to the current street conditions. If the variance of the lane-to-vanishing point distance exceeds a certain threshold, then Equation (3.9) and Equation (3.10) are reused to adjust the homography matrix and restore parallel lane conditions.

In conclusion, the improved design solved the drawbacks mentioned above by the following: ① The ROI is adaptive to the captured video, based on the estimated vanishing point. ② The improved method extracts an isosceles trapezium with an approximated rectangular resolution of $W \times [1 - \alpha - \gamma(\beta - \alpha)](H - V_y)$ pixels transformed to a WPM with $W \times H$ pixels, where W, H are the image width and height, V_y is the y-coordinate of the vanishing point and α, β and γ are coefficients in Equation (3.6) and Equation (3.7). Consequently more potential lane marking coordinates can be included at a higher BEV resolution. ③ The estimated vanishing point is much more accurate (refer to Figure 3.5d). ④ The road gradient change is taken into consideration by dynamically regulating the homography matrix to restore parallel lane conditions.

3.4.1.4 On-line Iterative Processing

The input of the iterative detection is the cropped ROI based on the boundaries inferred in Section 3.4.1.2. Given homography matrix generated off-line, the main steps performed on-line are WPM, image GrayScaling (GS), Edge Detection (ED), image CONVolution (CONV), Hough Transformation (HT), Linear and Spline RANSAC (LR, SR) model fitting. The details of these steps are illustrated as below.

- **WPM** is implemented by mapping the pixel value at point P_c in camera coordinate system to its corresponding point P_g in BEV ground plane, with projection relationship as

$$P_g = \mathbf{H_a} P_c, \tag{3.11}$$

where $\mathbf{H_a}$ is the affine homography matrix.

- **Image GrayScaling (GS)** in this design only takes into account one color channel of the input image, for the sake of efficiency, when calculating the mean image value.
- **Edge Detection (ED)** of the grayscaled image is implemented by subtracting the intensity of each BEV coordinate by the mean image value to reduce the influence of non-lane marking points in further processing steps, whereby negative values are set as zero.
- **CONVolution (CONV)** of an original image $\mathbf{I}$ into the convoluted image $\mathbf{J}$ with a kernel matrix $\mathbf{K}$ can be represented as

  $$\mathbf{J}(x, y) = \mathbf{K} * \mathbf{I} = \sum_{i=-k_x}^{k_x} \sum_{j=-k_y}^{k_y} \mathbf{K}(i, j)\mathbf{I}(x - i, y - j), \tag{3.12}$$

  where k_x, k_y are half of the kernel size in x- and y-direction and x, y are the coordinates of the current processed pixel. In this design, the image is convoluted with a second order Gaussian derivative filter.
- **Hough Transformation (HT)** is implemented by adding up the pixel values in each column of the convoluted image, assuming that the lane boundaries highlighted by the previous steps achieve the highest values.
- **Linear and Spline RANSAC (LR, SR)** model fitting can be divided as two stages. First a random subset of the data points are used to compute the model parameters. Then the quality of the postulated model are tested and candidate data points that fit the model within a defined threshold according to a specific loss function are added into the consensus set. The iteration terminates if the cardinality of the consensus set exceeds a second threshold.

3.4.2 Parallel and Heterogeneous Implementation

3.4.2.1 Parallel Implementation

The main steps illustrated in Section 3.4.1.4 can be divided into two categories, namely pixel-wise dependent ($task_{p\_d}$) and independent tasks ($task_{p\_ind}$). The pixel-wise independent tasks can be highly parallelized since data manipulations over each image pixel do not interfere with each other and the

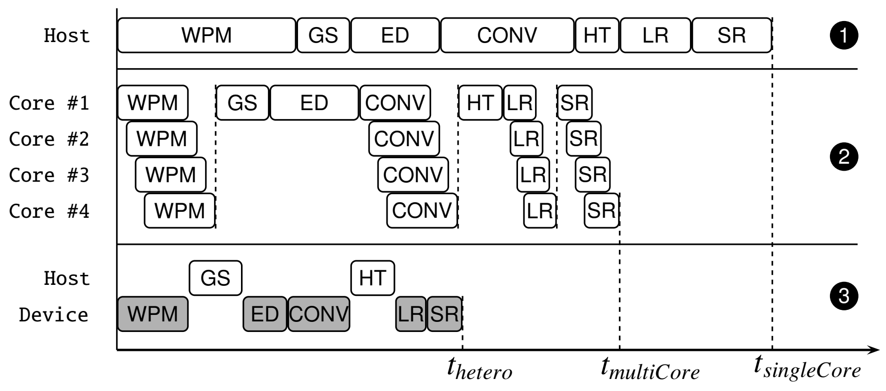


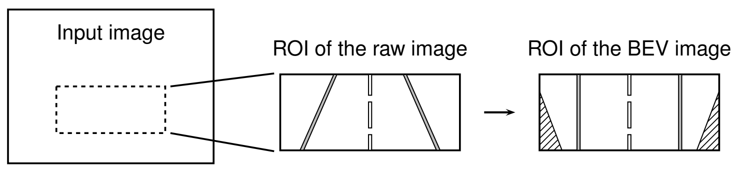
Figure 3.7: Overview of the designs of *r*-LDA on different platforms.

correctness of the final result can be guaranteed. However, the pixel-wise dependent tasks typically perform atomic data aggregation and therefore might block the parallel processing pipeline, which poses a potential trade-off. The WPM, edge detection, image convolution and RANSAC model fitting steps belong to the pixel-wise independent tasks, while the rest two are pixel-wise dependent.

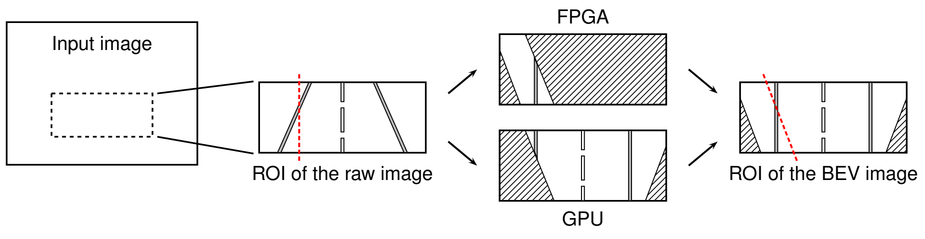
Figure 3.7 gives an overview of native and parallel implementations of the *r*-LDA. For the native CPU design, a single-core implementation is provided as the baseline and a multi-core design is afterwards proposed to showcase the speedup as well. The *single-core implementation* (Case ❶) handles the aforementioned steps sequentially, while the *multi-core implementation* (Case ❷) processes WPM and image convolution with multiple threads and executes all RANSAC iterations for one vertical line in a single thread. To ensure the scalability of the parallel implementation, OpenCL is used to program on GPU and FPGA. In the *OpenCL-based implementation* (Case ❸), WPM, edge detection, image convolution and two RANSAC model fitting workloads are formulated as 4 kernels. Edge detection and image convolution are combined into one kernel so that the number of writing operations is halved as both steps are performed simultaneously without caching the mean corrected top-view image. Moreover, each of the two RANSAC kernels processes a single iteration of one vertical line per invocation. Each pixel operation on the BEV coordinate is mapped to one work item and all BEV pixel coordinates of the ROI are divided evenly among the work groups. With regards to the pixel-wise dependent tasks, different implementations which either includes or excludes the atomic manipulations are presented in order to test the tradeoff of on-device versus out-of-device data aggregation.

3.4.2.2 Heterogeneous Implementation

In the native design, 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



(a) Homogeneous design.



(b) Heterogeneous design.

Figure 3.8: Parallel design of the WPM transformation in *r-LDA*. The areas with slashes represent empty values of the pixels.

kernelCONV, which consume the WPM and image convolution tasks respectively. During the experiments, we found the computation of the RANSAC model fitting were fast enough when processed on CPU side, therefore in the optimized design, the *model fitting* step is performed on the host since its time cost is much smaller than that of the *pre-processing* step.

The heterogeneous execution is developed 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, Figure 3.8 presents its parallel design in the homogeneous (Figure 3.8a) and heterogeneous (Figure 3.8b) 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.

3.4.2.3 Optimization for Heterogeneous Executions

The optimization of both GPU and FPGA kernels is two-fold. First of all, the convolution kernel (kernelCONV) uses pre-computed values to replace memory reading operations by a multiplication with a fixed constant, to avoid expensive global memory access. Secondly, each loop inside the kernels is manually unrolled to ensure the benefit of coalesced memory access.

The main optimization effort lies in the kernel configuration on the FPGA platform. As OpenCL kernel is hardware synthesized and then implemented as dedicated circuit blocks on FPGA, the board resource limits the actual performance and the resource utilization of each kernel needs to be coordinated. In the design the four OpenCL kernels are included into a single file before they are compiled as the executable binary file, taking into account the fact that on-board kernel switching overhead is significantly expensive and cannot be compensated by the full optimization of each kernel instead, as each frame is processed continuously in a loop manner.

In Intel FPGA SDK for OpenCL [59], several attributes can be used to guide the optimization of the kernel. To increase the data processing efficiency, num_simd_work_items can be used to specify the number of work-items within a work-group so that the kernel is executed in a SIMD manner. num_compute_units is another attribute that can instruct the off-line OpenCL compiler to generate multiple kernel compute units capable of executing multiple work groups simultaneously. In addition, max_unroll_loops can be indicated to decrease the number of executed iterations at the expense of increased hardware resource consumption. During the implementation of r-*LDA*, both num_simd_work_items and max_unroll_loops are set as 1 since each kernel is thread-ID dependent and each loop is manually unrolled.

The optimization of the aforementioned OpenCL kernels on the FPGA platform is as follows: first it is assumed that each kernel is using the most simplified configuration, i.e., with num_compute_units set as 1, to guarantee that the design meets the board resource limitation. Afterwards, each kernel is step by step optimized by assigning more compute units to the most time-consuming task load.

3.5 Evaluation Results

3.5.1 Evaluation Setup

The test environment is a heterogeneous system consisting of multi-core CPU, GPU and FPGA. The detail information about the hardware specification is shown in Table 3.1. To evaluate the performance of the implemented algorithm, the runtime as well as detection quality are measured for the manually labeled Caltech dataset [60]. This dataset consists of four clips on various urban street scenarios including straight and curved lane markings, shadows, other vehicles, reflections and street writings to reflect real-world conditions.

During the evaluation, the execution time of each run is measured to calculate the real-time performance. The power estimation method is the same as [61] and Altera PowerPlay power analyzer [62] is used to estimate the power consumption of running each OpenCL kernel on FPGA. As for the power estimation of GPU and CPU, the metric data come from official specifications of the COTS components.

During the runtime evaluation of p-*LDA*, 2^{12} *good lines* and 2^{13} *candidate lines* are used to detect 2 lane markings. As for the parameters of r-*LDA*, the number of RANSAC iterations is set to an optimal value of 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.

3.5.2 Results and Analysis

3.5.2.1 *p*-LDA: Particle-filter-based Lane Detection Algorithm

In p-*LDA*, 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 *heteroConstant* 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 *heteroBalanced* scenario, with given partitioned task, the workload balance scheme tunes the task proportions on FPGA and GPU during the processing of each frame.

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 heterogeneous executions.

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, ① the kernel execution times of the four designs are recorded and ② during the *heteroBalanced* run, the real-time task rates on both FPGA and GPU devices are monitored. Figure 3.9 gives the results of the recorded kernel execution time and Figure 3.10 shows the real-time task rates of the test videos.

Figure 3.9 indicates 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 7.171% of that of *singleFPGA* and 245.0% of that of *singleGPU*. It is seen that

Table 3.1: Detailed specification of the hardware platforms.

Platform	**Information**	
Host CPU	Intel Xeon E31225 @ 3.10GHz	
Thermal Design Power	95W	
PCIe generation	2.0	
Device	FPGA	GPU
Model	Terasic Arria 10	AMD W7100
Architecture	Arria 10 AX	FirePro
OpenCL SDK version	Intel FPGA SDK 16.0	AMD APP SDK 3.0
Peak GFLOPS	1366	3379.2
Peak board power (W)	95	150

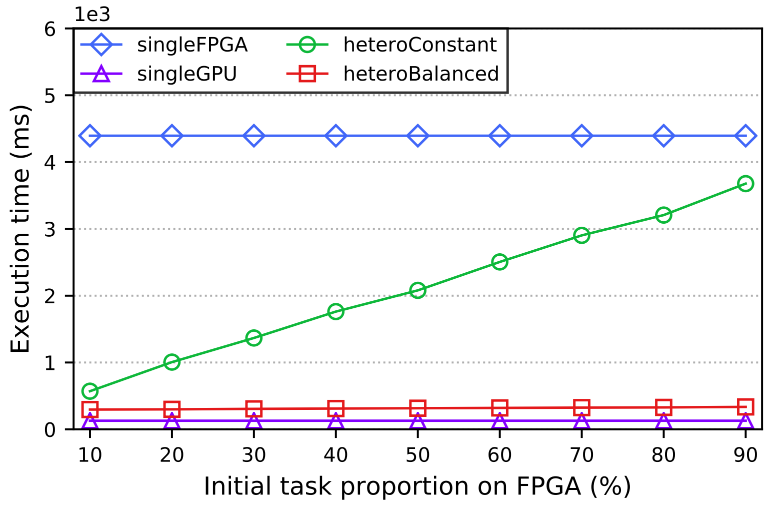


Figure 3.9: Kernel execution time of *p-LDA* in the different scenarios.

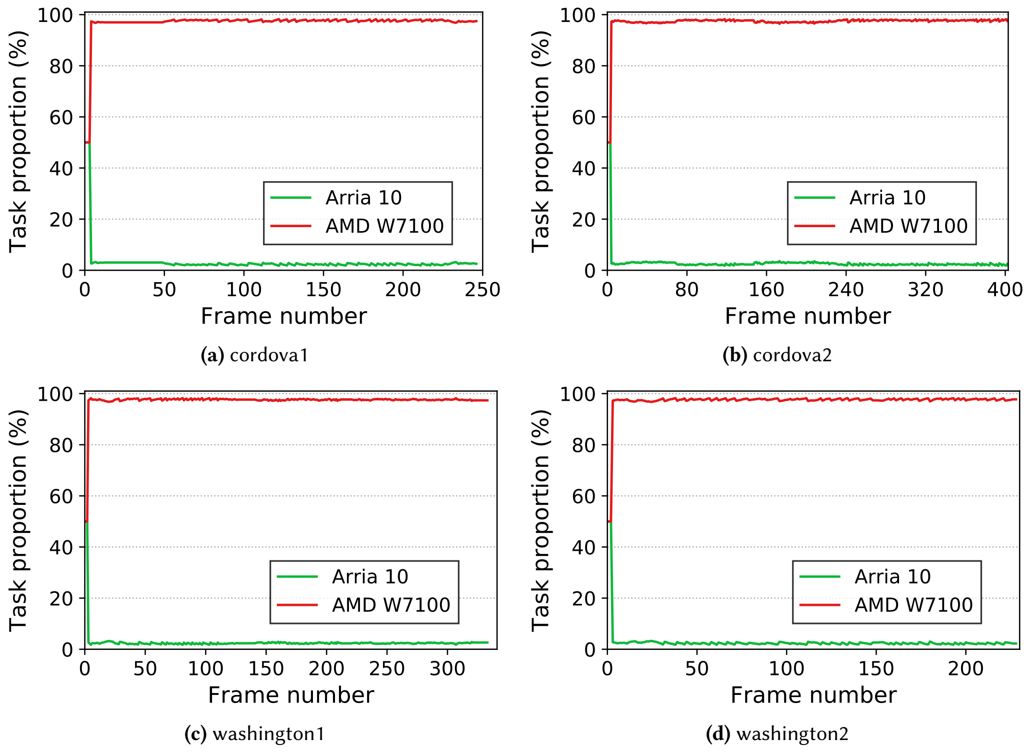


Figure 3.10: Real-time task rates of the test videos.

the time costs of the heterogeneous executions 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 observed, the

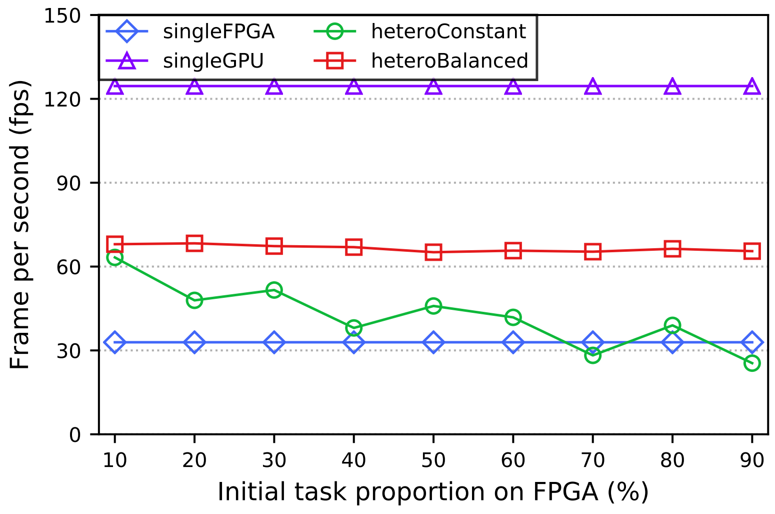


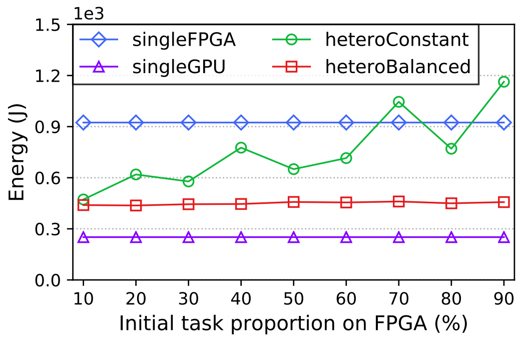
Figure 3.11: Performance of *p-LDA* in the different scenarios.

kernel execution time of *heteroConstant* always surpasses *heteroBalanced*, which verifies the validity of the workload balance scheme. In Figure 3.10, it is seen that the real-time task proportions of all the test 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. This demonstrates the robustness of the workload balance scheme.

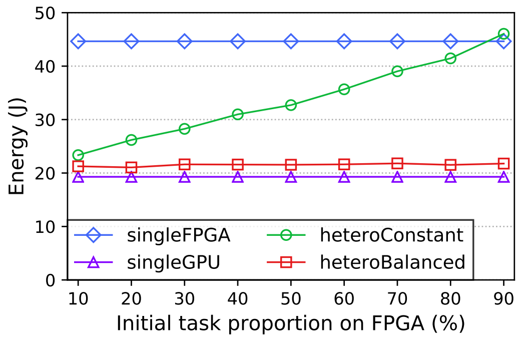
Runtime Performance. Figure 3.11 depicts the performance results of the four implementations running on the test platforms. From the figure, it is observed that the performance of *singleGPU* outperforms *singleFPGA* and this is reasonable due to the lower computation capacity of FPGA (refer to the peak GFLOPS in Table 3.1). Both of the heterogeneous runs gain a performance increase than *singleFPGA*, which without doubt benefits from the high performance GPU. 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 the above-mentioned results in Figure 3.10.

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, the *heteroBalanced* case increases the performance by an average of 102.1% when compared with the *singleFPGA* case.

Energy Efficiency. Figure 3.12 shows the overall energy cost for the four different designs. Figure 3.12a presents the energy cost of the overall system, while Figure 3.12b gives the results of the



(a) Overall system energy cost.



(b) Accelerator energy cost.

Figure 3.12: Energy cost of *p-LDA* in the different scenarios.

accelerator energy consumption.

As indicated by Figure 3.12a, 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, the heterogeneous designs are able to consume much less energy than *singleFPGA*. With regards to the on-device energy cost (Figure 3.12b), 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.

Based on the performance and energy results, the energy efficiency results of the four scenarios are calculated and presented in Figure 3.13. As observed, FPGA turns out a huge advantage over GPU in terms of the energy efficiency. The heterogeneous executions, on the other hand, show values in the middle since they leverage the performance and energy tradeoff of both platforms. The energy efficiency of the *heteroBalanced* design is lower than that of *heteroConstant*, due to the reason that more task load is assigned to the energy-consuming GPU card.

In summary, the heterogeneous executions consume less energy, when compared with the most-energy-cost single accelerator (i.e., the *singleFPGA* case). Using the workload balanced scheme not only “smoothes” the heterogeneous execution, but also shortens the energy cost regardless of the initial task rates. Moreover, using the heterogeneous architecture improves the energy efficiency when certain performance bound is guaranteed. The heterogeneous implementations could solve both the

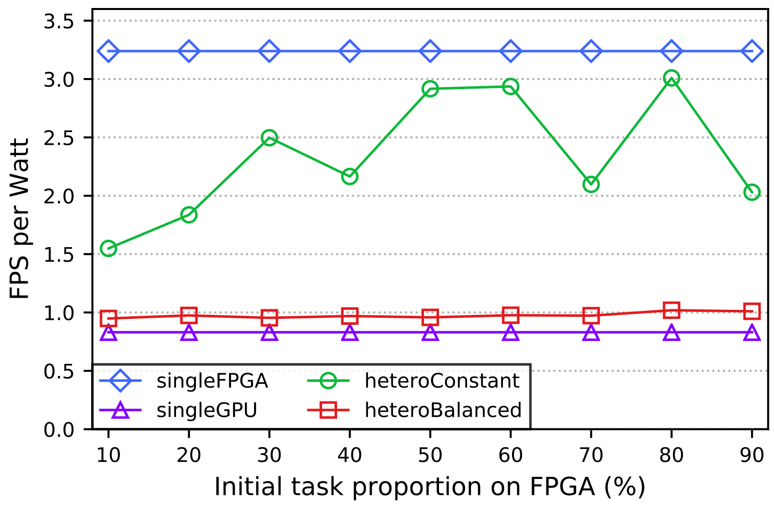


Figure 3.13: Energy efficiency comparison of *p-LDA* in the different scenarios.

performance and energy bottlenecks caused when only using a single accelerator.

3.5.2.2 *r*-LDA: Ransac-based Lane Detection Algorithm

The whole algorithm is partitioned into four parts and each part is recorded with time stamps to obtain the final execution time. Table 3.2 details the information of each partition. **Partition I** includes the WPM transformation, image grayscaling and the computation of the image mean gray value. **Partition II** performs the edge detection, image convolution and the simplified Hough transformation. Subsequently, **Partition III and IV** perform the linear and spline RANSAC model fitting, respectively. The reason for this partitioning is straightforward, since each partition natively corresponds to one OpenCL kernel in the later design.

Homogeneous Execution. The performance of the single- and multi-core executions on CPU are evaluated at first. Figure 3.14 exhibits the execution time of each partition with single or multiple

Table 3.2: Task partitions of r-*LDA*.

Module	Step	Partition	Category
Preprocessing	WPM	I	$task_{p\_ind}$
	Image grayscaling	I	$task_{p\_d}$
	Edge detection	II	$task_{p\_ind}$
	Image convolution	II	$task_{p\_ind}$
Model fitting	Hough transformation	II	$task_{p\_d}$
	Linear RANSAC	III	$task_{p\_ind}$
	Spline RANSAC	IV	$task_{p\_ind}$

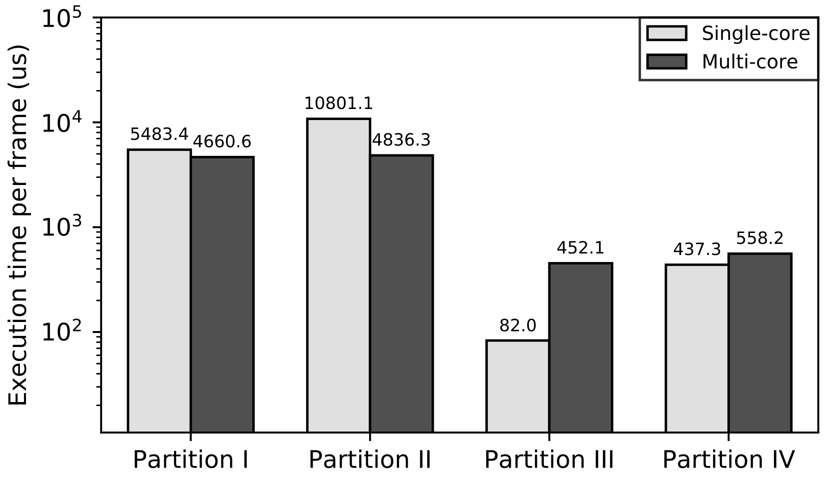


Figure 3.14: Execution time of the homogeneous execution on CPU.

threads. Note the logarithmic scale of the y-axis. Here the multi-core implementation is tested with 8 threads. From the figure, it is seen that the time costs of Partition III and IV are far less than that of Partition I and II. This can be explained by the reason that the computation load of the linear and spline RANSAC model fitting is rather small, as in practice the number of detected potential lane markings is always below 10. In this case, the benefit of concurrently performing RANSAC model fitting cannot compensate for the overhead of initializing new threads and additional data synchronization, resulting in the longer execution time of the multi-core implementation. However, the preprocessing of the ROI image is computation-intensive and therefore can be accelerated by the parallel implementation. In this design, the multi-core execution outperforms the single-core version and decreases the execution time of Partition I and II by 15.01% and 55.22%, respectively.

Single-accelerator Execution. The OpenCL-based implementation follows the top-down design principle. First of all, the naive version wraps each of the aforementioned partition into one kernel. Afterwards, by gradually pruning lightweight and data aggregation task loads, the runtime performance of each kernel is re-evaluated to demonstrate the tradeoff of the parallelization of these lightweight workloads and atomic operations. The step-by-step optimization of the single-accelerator execution (CPU-GPU and CPU-FPGA executions) is shown in Table 3.3. The steps which are marked with ticks are executed on the GPU (or FPGA), while the rest are run on the host CPU. First, all the steps in Table 3.3 are assumed to be parallelized (*heteroGPU*$_1$, *heteroFPGA*$_1$), then the model computation part in the RANSAC step is excluded to exhibit the pro and con of parallelizing this lightweight task load (*heteroGPU*$_2$, *heteroFPGA*$_2$). Subsequently, atomic data aggregation tasks are excluded to weigh the benefit of implementing them with OpenCL (*heteroGPU*$_3$, *heteroFPGA*$_3$). At last, the optimal execution

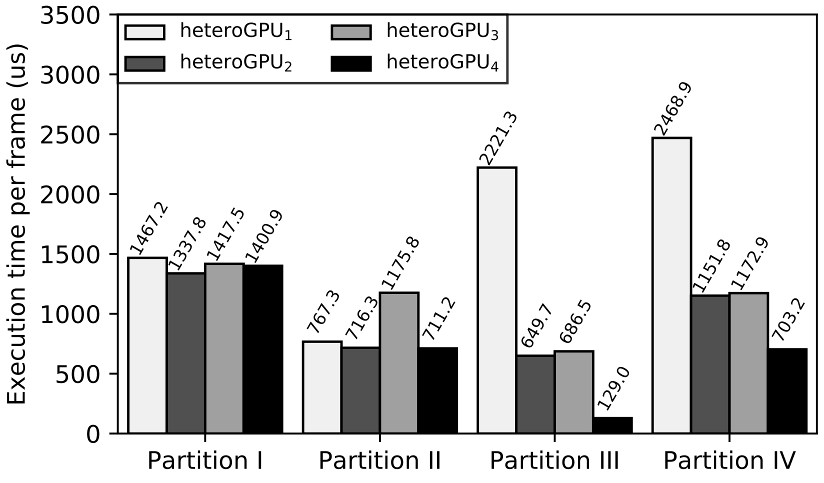


Figure 3.15: Time cost of CPU-GPU executions.

(*heteroGPU*$_4$, *heteroFPGA*$_4$) is inferred from the previous contrast tests.

Figure 3.15 gives the runtime results of the CPU-GPU executions. It is clearly seen that in *heteroGPU*$_1$ the time cost of RANSAC model fitting (Partition III and IV) is nearly two times large as the image pre-processing (Partition I and II), although the input task size of the RANSAC part is rather small. As observed from *heteroGPU*$_1$ to *heteroGPU*$_2$, the execution time is decreased rapidly when shifting the RANSAC model computing part to the CPU side. However, this time cost is still large than that of CPU homogeneous execution (shown in Figure 3.14). Comparing the results of *heteroGPU*$_2$ and *heteroGPU*$_3$, it is seen that the execution of the atomic operations consumes longer time when they are processed on the CPU side, which means the parallelization of these data aggregation tasks are worthwhile on GPU. As a consequence, the optimal *heteroGPU*$_4$ execution excludes the RANSAC part and processes

Table 3.3: Configurations of the single-accelerator execution.

Configuration	WPM	GS	ED	CONV	HT	RANSAC	
						Model Computation	Model Evaluation
$heteroGPU_1$	✓	✓	✓	✓	✓	✓	✓
$heteroGPU_2$	✓	✓	✓	✓	✓		✓
$heteroGPU_3$	✓		✓	✓			✓
$heteroGPU_4$	✓	✓	✓	✓	✓		
$heteroFPGA_1$	✓	✓	✓	✓	✓	✓	✓
$heteroFPGA_2$	✓	✓	✓	✓	✓		✓
$heteroFPGA_3$	✓		✓	✓			✓
$heteroFPGA_4$	✓		✓	✓			

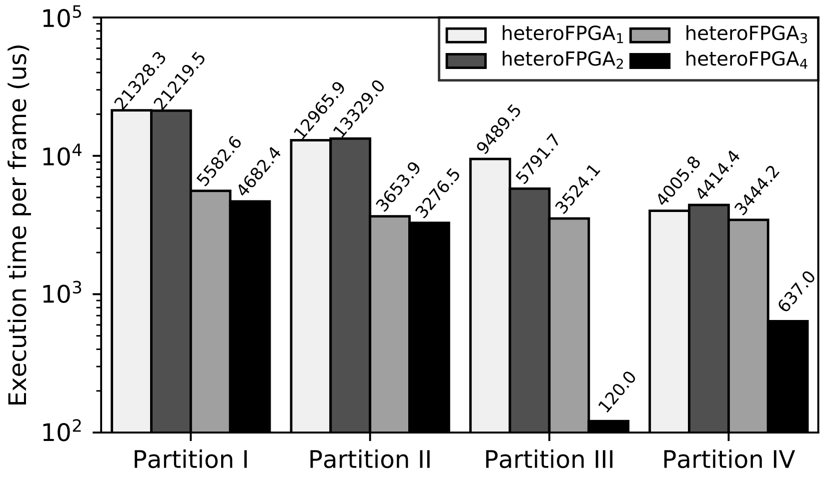


Figure 3.16: Time cost of CPU-FPGA executions.

all the other steps on GPU side, which takes an overall time consumption of less than 3 ms.

The runtime evaluation of CPU-FPGA executions is shown in Figure 3.16. Note the logarithmic scale of the y-axis. The execution time of Partition III turns out an apparent decline when the model computation of the RANSAC fitting is processed on the host CPU (from *heteroFPGA*$_1$ to *heteroFPGA*$_2$), while there exists a minor increase in the time cost of Partition IV. From the executions of *heteroFPGA*$_2$ and *heteroFPGA*$_3$, it is clearly seen that FPGA suffers much more than GPU when performing the atomic operations on device side. Removing the data aggregation tasks can decrease the runtime cost of Partition I and II by 73.69% and 72.59%, respectively. Different to GPU, the optimal CPU-FPGA execution (*heteroFPGA*$_4$) excludes both the aggregation workloads (image grayscaling and simplified Hough transformation) and RANSAC model fitting from the OpenCL kernels.

Optimal & Heterogeneous Execution. As observed in Table 3.3 and Figure 3.16, the optimal CPU-FPGA heterogeneous execution contains two kernels and the WPM kernel consumes more time than that of the image convolution kernel. Consequently during the optimization more resources are allocated to the WPM kernel. Specifically in the optimal designs, on the FPGA board, the num_compute_units attribute of the WPM and convolution kernel is respectively set as 12 and 4. Figure 3.17 presents the performance of the single- and multi-core implementation, the single-accelerator execution, as well as the heterogeneous GPU-FPGA design.

As depicted in Figure 3.17, all the 5 different implementations are capable of processing 640×480 videos real-timely. The CPU-GPU execution achieves the best performance of 169.8 fps, while the CPU-FPGA execution can process the video stream with time cost of 7.07 ms per frame. Nevertheless, the CPU-only implementations cannot handle 1920×1080 images with a performance demand of 30

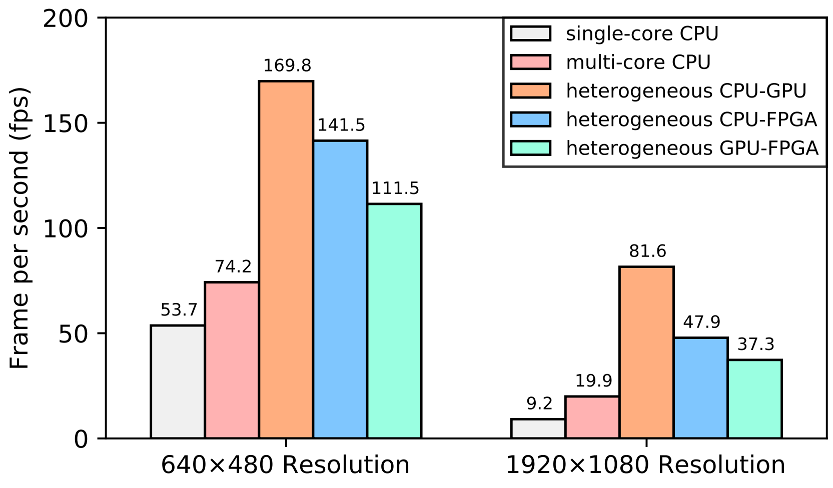


Figure 3.17: Performance with different parallel configurations.

fps. The single-accelerator and the heterogeneous executions can still fulfill this requirement, resulting in speedups of 8.87×, 5.21×, and 4.05× over the single-core implementation, respectively in the CPU-GPU, CPU-FPGA, and GPU-FPGA-combined scenarios.

The performance of the heterogeneous GPU-FPGA execution is unable to outperform the single-accelerator executions, due to the reasons that: ① As demonstrated in Figure 3.15 and 3.16 the configurations of the optimal CPU-GPU and CPU-FPGA scenarios are not the same, therefore we applied the same configuration of the CPU-FPGA design to the GPU-FPGA execution in order to ensure the data consistency. This inevitably sacrifices the performance of the GPU side. ② The data communication between the host and devices causes an increasing latency when more accelerators are involved in the system. Moreover, the consumed data in the singe-accelerator case are integrated data blocks and they are split in the GPU-FPGA execution. This results in data transfer of small data fragments, which is considerably time-consuming.

Despite the performance loss, the heterogeneous GPU-FPGA execution still provides a solution of better energy utilization. The results are shown in Figure 3.18. Similar to *p-LDA*, using heterogeneous architecture for *r-LDA* also enhances the energy efficiency, which is rather important when performance is not the primary consideration.

3.6 Summary

In this chapter, two different approaches to implement the typical LDA in ADAS are presented and their deployment in the heterogeneous context is further illustrated in detail. The evaluation results

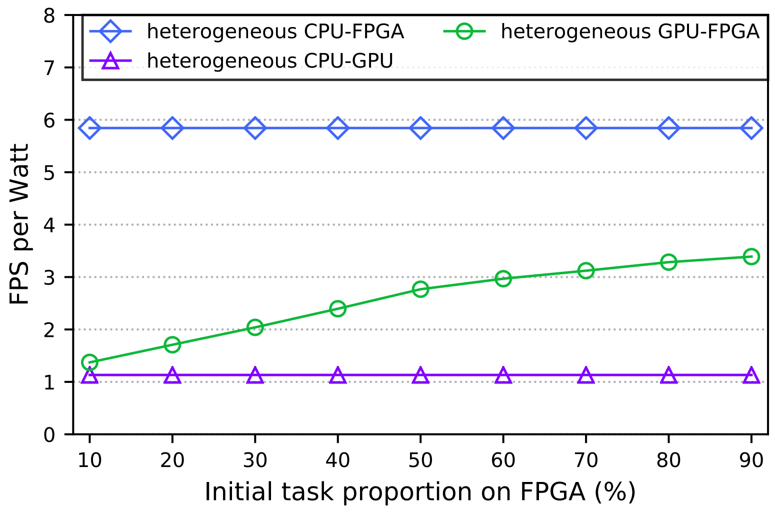


Figure 3.18: Energy efficiency comparison of *r-LDA* with different parallel configurations.

reveal that heterogeneous computing is a promising solution for future ADAS since it is able to regulate the performance and energy tradeoff in the system. In the first application, this chapter provides a lightweight workload balance scheme which can robustly tune the tasks between the two accelerators. The optimization results of the second application reveal that GPU and FPGA exhibit different characteristics when performing the same task load. FPGA is more sensitive to atomic data aggregation operations, while both GPU and FPGA are unsuitable to parallelize lightweight workloads. For lightweight workload, both platforms show a larger overhead when transferring data between the host and device, compared to the processing of the task itself. However, the atomic data aggregation operations turn out a severe performance bottleneck for the FPGA platform, which in the other way still deserves to being accelerated by GPU.

Inspired by all these observations, the next chapter focuses on the performance aspect and propose a detailed procedure to optimize the two applications in GPU-FPGA heterogeneous system.

Chapter 4

Performance Optimization of ADAS Applications in Heterogeneous Systems

Chapter 3 investigated how the naive designs of the two LDAs are parallelized to enable their executions on the COTS hardware accelerators. Inspired by the observations, the work in this chapter proposes a detailed procedure that helps guide the performance optimization of parallelized ADAS applications on a heterogeneous platform consisting of GPU, FPGA, and multi-core CPU.

The explosive growth of massive data captured by various sensors on modern vehicles has impelled the deployment of COTS accelerators for the research and development of ADAS. Aiming at HPC applications in ADAS, heterogeneous computing emerges as it leverages different accelerators 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, OpenCL arises and turns out to be an ideal heterogeneous programming framework as it specifies a high-level abstraction for low-level hardware instructions, which enables to scale computations among different brands of platforms without changing the source code.

In heterogeneous context, real time constraint is well handled since different accelerators can be arranged to consume the computational tasks that fit their inherent processing characteristics. However, from the host side, how to schedule different accelerators to gain the optimal system performance is nontrivial. Although the advent of cross-platform programming framework such as OpenCL facilitates the programmability of ADAS applications on heterogeneous devices, the performance portability is still vulnerable and subject to the miscellaneous hardware implementations by the respective manufacturers. Program developers always need to elaborately and repeatedly define and assign the workload on different platforms so as to gain the best possible performance benefit. This error-prone procedure requires a substantial amount of program deploying, debugging, and tuning efforts.

4.1 Overview

State-of-the-art studies mainly focus on the implementation and optimization of applications accelerated with a single type of hardware accelerator, such as using GPU [38] [52] [63] or FPGA [51] [53] [64]. This chapter investigates several key factors that influence the performance gain when deploying parallelized ADAS applications in heterogeneous systems. The work in this chapter differs from state-of-the-art with the following aspects:

1. The study focuses 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 optimization methods called accelerator execution overlapping and dynamic workload tuning are presented.

2. The applications are customized 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. The proposed procedure contains various approaches to optimize 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.

After the identification of the performance bottlenecks in the target program, one intra-accelerator and two inter-accelerator sub-optimization methods are taken into consideration so as to increase the task processing efficiency. Moreover, application-oriented optimization of the workload is conducted to further improve the overall runtime performance. The two different implementations of LDA illustrated in Chapter 3 are taken as case studies and 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 chapter is organized as follows: Section 4.2 is related work and Section 4.3 illustrates in detail the proposed procedure and the sub-optimization methods. Section 4.4 presents experimental results and Section 4.5 summarizes the work in this chapter.

4.2 Related Work

Lane detection is mostly achieved via filtering techniques to capture lanes, however it is rarely deployed on heterogeneous platforms. The work in this chapter focuses on the optimization 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 optimization framework to assist better development of OpenCL applications. The work in [65] provided a generic tool for performance measurement of OpenCL programs. In [66], the authors proposed a framework combining OpenCL application auto-tuning and runtime resource management. The study in [67] presented a transparent OpenCL overlay called Helium, for inter- and intra-kernel optimization. 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 optimizations are highly dependent on the specific algorithm, architecture, and programming features. In [68], 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 [69] and [70]. In their work, optimizations were mainly performed based on the CNN algorithm itself.

While there exist substantial efforts to parallelize ADAS applications with GPU [38] [52] [63] or FPGA [49] [50] [51] [53] [64], few studies are reported to accelerate them with heterogeneous commodity hardware [71]. The study in [11] investigated the feasibility of using COTS hardware for ADAS development, but the performance optimization is not considered. The work in [63] presented a step-by-step optimization of face detection algorithm in CPU-GPU heterogeneous systems. However, this study considered only the CPU-GPU heterogeneous architecture. Authors in [72] compared the performance of using GPU, FPGA, or both devices to accelerate pedestrian detection applications. Unlike the data-level parallel designs in this work, in [72] GPU and FPGA process different tasks to fulfill task-level parallelism. In [73], 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. The most related study to the work presented in this chapter is [63], where the authors used optimization methods including CPU execution time hidden, memory coalescing, and variable parallel granularity. The difference of the work in this chapter is that rather than using a single GPU, the FPGA-GPU heterogeneous context is considered so that ① the execution time of both accelerators can also be hidden via changing build-in function order, ② parallelism on FPGA side could be further adjusted by using pragma primitives, and ③ other optimization methods like dynamical workload

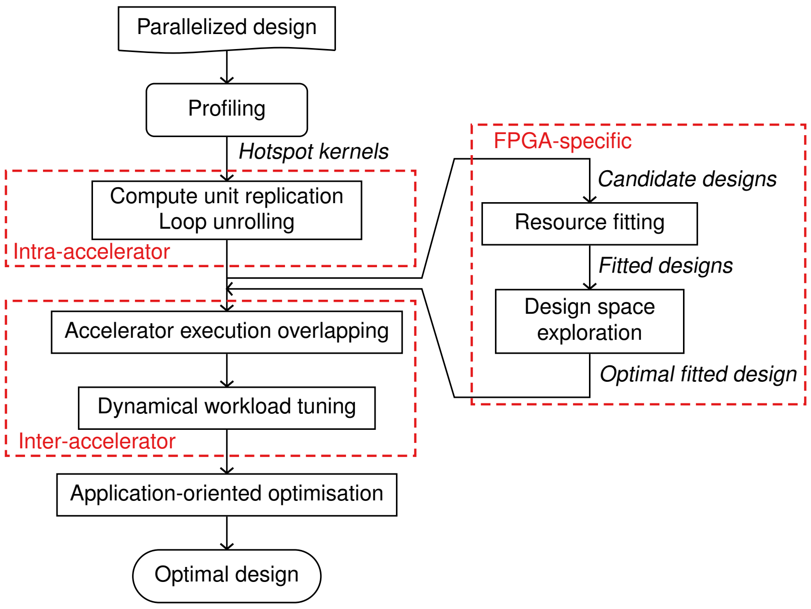


Figure 4.1: Optimization procedure for parallelized ADAS applications in FPGA-GPU heterogeneous platform. The three red dashed boxes indicate the intra-accelerator, inter-accelerator, and FPGA-specific optimization module, respectively.

tuning are also presented.

4.3 Optimization Procedure

Figure 4.1 exhibits the proposed optimization procedure. First of all, profiling of the application is needed to locate the hotspot kernels so that the bottleneck can be identified and further optimized. Given a native design of the hotspot kernels, generic optimization is utilized on the intra-accelerator side. These optimizations 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-optimization 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 type of workload is flexible and can also be regulated to enhance the runtime performance.

4.3.1 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 optimized with top priority.

4.3.2 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 beyond of optimization. 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.

The compute unit replication in this chapter is performed as follows: first it is assumed that by default, each kernel is using 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 is optimized in a step-by-step manner by assigning more and more compute units to the most time-consuming task load.

4.3.3 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.

4.3.4 Accelerator Execution Overlapping (EO)

As illustrated in Chapter 3, the heterogeneous implementations of the two applications are data-level parallel. Therefore 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. It is noted that the call order of these functions should be carefully considered since not all of them are non-blocking invocations.

Figure 4.2 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,

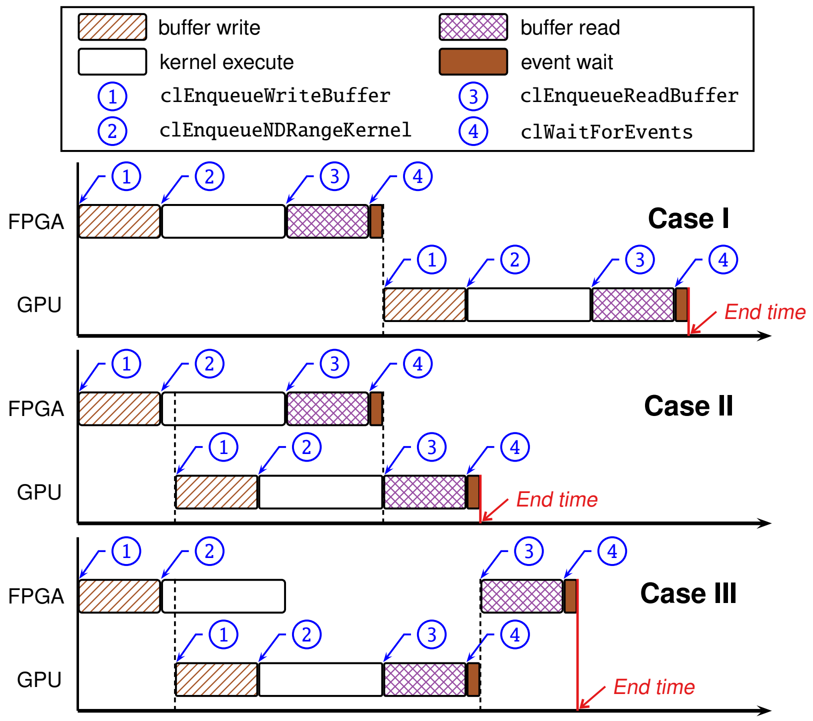


Figure 4.2: A sample heterogeneous execution with different call order of OpenCL API functions for FPGA and GPU. The blue arrows indicate the exact time point when the API function in the blue circle would actually take effect.

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. In this way, the runtime performance can be improved.

4.3.5 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, inspired by the work in [31], this chapter applies 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

$$\begin{aligned} W_i' &= \frac{c_i}{\sum_{i=1}^{N} c_i} \sum_{i=1}^{N} W_i \\ c_i &= \frac{W_i}{T_i}, \end{aligned} \tag{4.1}$$

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.

4.3.6 Application-oriented Optimization (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.

4.3.6.1 *p*-LDA: Particle-filter-based Lane Detection Algorithm

As described in Section 3.3.1.1, 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, the optimization enables an adaptive ROI when processing the image frames iteratively. The size of the ROI is adjusted each time after the frame is processed, so that the proper ROI size for the next frame is obtained.

Algorithm 4: ROI tuning scheme for *p-LDA*

```
Input: Best line set $\mathbb{B}$, $imgWidth$, $initRoiStart$, $initRoiEnd$, $roiStart$, $roiWidth$
Output: $roiStartAdapted$, $roiWidthAdapted$
1  $T_{left} \leftarrow 1/4$, $T_{right} \leftarrow 3/4$
2  $roiStartAdapted \leftarrow roiStart$
3  $roiEndAdapted \leftarrow roiStart + roiWidth$
4  foreach best line $b \in \mathbb{B}$ do
5      $roiStartAdapted \leftarrow min\langle roiStartAdapted, \text{b.start}\rangle$
6      $roiEndAdapted \leftarrow max\langle roiEndAdapted, \text{b.end}\rangle$
7  if $roiStartAdapted < initRoiStart$ then
8      $roiStartAdapted \leftarrow initRoiStart$
9  else if $roiStartAdapted > imgWidth \times T_{left}$ then
10     $roiStartAdapted \leftarrow imgWidth \times T_{left}$
11 if $roiEndAdapted > initRoiEnd$ then
12     $roiEndAdapted \leftarrow initRoiEnd$
13 else if $roiEndAdapted < imgWidth \times T_{right}$ then
14     $roiEndAdapted \leftarrow imgWidth \times T_{right}$
15 $roiWidthAdapted \leftarrow roiEndAdapted - roiStartAdapted$
16 if redetection then
17     $roiStartAdapted \leftarrow initRoiStart$
18     $roiWidthAdapted \leftarrow initRoiEnd - initRoiStart$
```

Algorithm 4 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 the optimization focuses 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 4, 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.

4.3.6.2 *r*-LDA: Ransac-based Lane Detection Algorithm

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 optimization of the model fitting part. As is known, the number of RANSAC iterations N is determined by

$$N = \frac{\log(1-\rho)}{\log(1-\omega^{\eta})}, \tag{4.2}$$

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 [48]. Since ω is preliminarily unknown in r-LDA, during the evaluation N is first set as a considerably large value and then gradually reduced until the accuracy hits a tolerable threshold. In this way, the number of RANSAC iterations is minimized while the accuracy can be still guaranteed.

4.3.7 Discussion

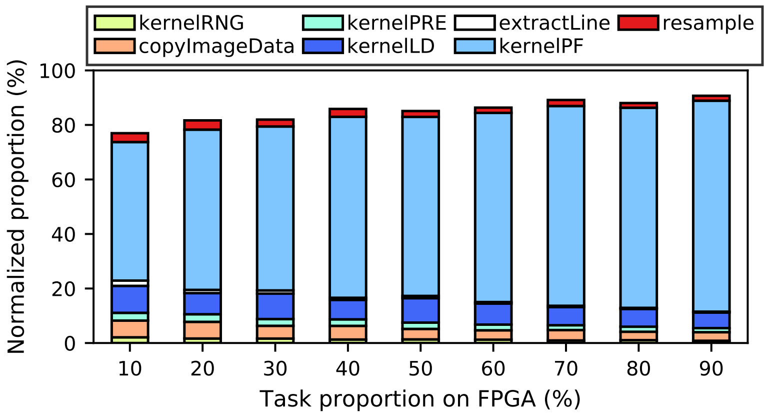
The aforementioned optimization methods constitute a systematic procedure for improving the performance of OpenCL-based ADAS applications in FPGA-GPU heterogeneous systems. Compared with state-of-the-art, the work in this chapter targets a different scenario, i.e., the performance optimization of lane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

4.4 Evaluation Results

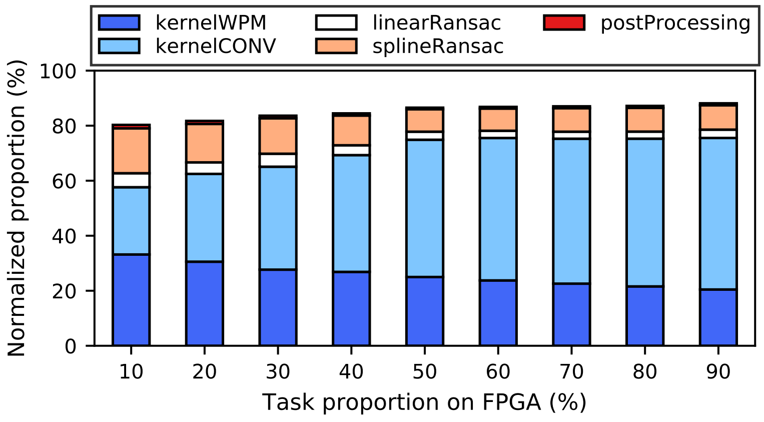
4.4.1 Evaluation Setup

The h^{2}ECU presented in Chapter 2 is used as the evaluation platform and the details about the hardware specification are shown in Table 4.1. To evaluate the performance of the two case study applications, the benchmark videos from Caltech data set [60] are utilized. 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*, 2^{12} *good lines* and 2^{13} *candidate lines* are used to detect 2 lane markings. As for the parameters of r-*LDA*, the initial value of the number of RANSAC iterations is set to 300 and the observed optimal value is 40. Each time the FPGA device is assigned the workload



(a) *p-LDA*



(b) *r-LDA*

Figure 4.3: Normalized execution time distribution of the profiled code blocks in *p-LDA* and *r-LDA*.

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 4.1: 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

4.4.2 Profiling Results

4.4.2.1 *p*-LDA: Particle-filter-based Lane Detection Algorithm

This application is segmented into 7 main code blocks and the detailed description of them is shown in Table 4.2. Figure 4.3a reveals the normalized execution time of these code blocks when the workloads are distributed to FPGA and GPU with different proportions. As observed in Figure 4.3a, 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 optimization. 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 optimization of this code block is to reduce the transmitted data size, which is done by the ROI tuning scheme.

4.4.2.2 *r*-LDA: Ransac-based Lane Detection Algorithm

This application contains 5 main functional modules of which the detailed information is listed in Table 4.3 and their respective execution time distribution is shown in Figure 4.3b. kernelWPM and kernelCONV are deemed as the 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. Optimization of this part is done by reducing the number of RANSAC iterations.

4.4.3 Optimization Results

4.4.3.1 Compute Unit Replication (CR)

In this 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

Table 4.2: 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

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 4.4 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.

For p-*LDA*, the maximum CR factor is 3 and Figure 4.4 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$\times$ and 1.52$\times$, when the compute unit is replicated 2 and 3 times, respectively.

From Table 4.4, 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, the performance evaluation of CR optimization for r-*LDA* is conducted via the control variable method, i.e. varying either λ_{CR}^{wpm}

Table 4.3: 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

Table 4.4: 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

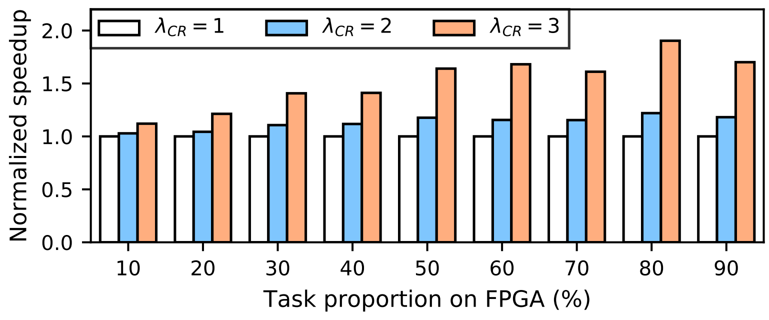
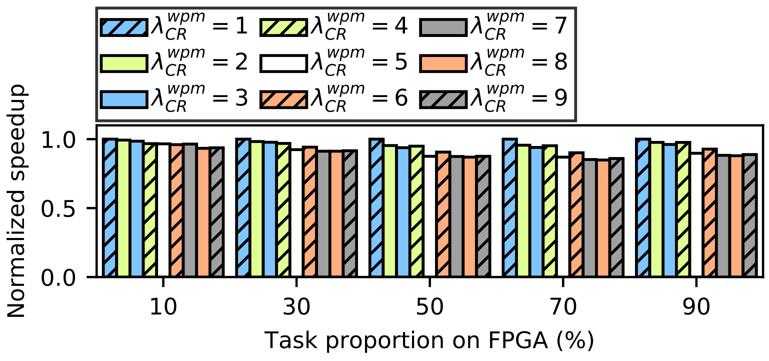
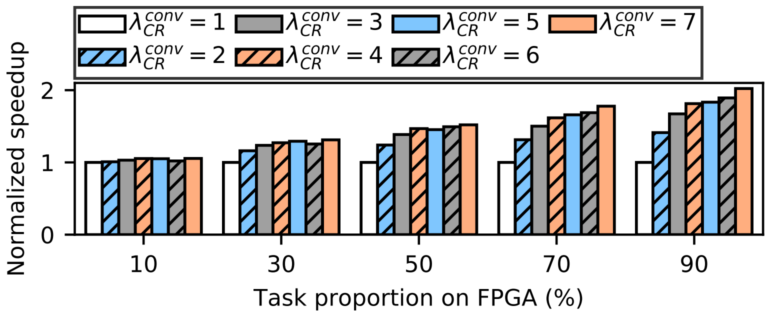


Figure 4.4: Performance comparison of *p-LDA* when replicating different number of compute units.



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



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

Figure 4.5: Performance comparison of *r-LDA* when replicating different number of compute units.

or λ_{CR}^{conv} while setting the other one as a constant. Figure 4.5 gives the detailed comparison results when the FPGA task proportion is 10%, 30%, 50%, 70%, and 90%. An interesting point shown in Figure 4.5a is that merely replicating kernelWPM actually degrades the runtime performance. This slow-down 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

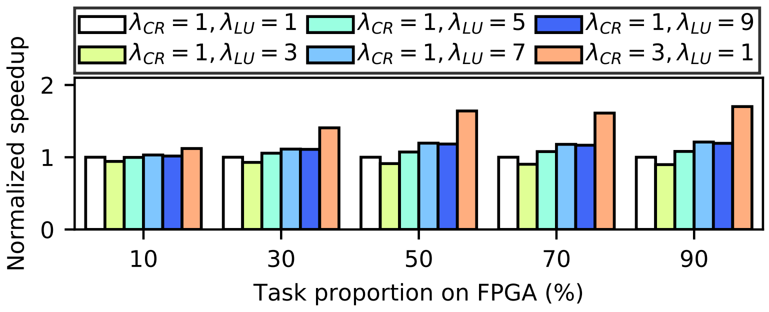


Figure 4.6: Performance comparison of *p-LDA* using different loop unrolling factors.

on-board 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 Figure 4.5b. 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 optimizations of kernelWPM and kernelCONV is a trade-off between the computation- and memory-intensiveness of these two kernels. During the 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.

4.4.3.2 Loop Unrolling and Memory Access Coalescing (LU)

To showcase the influence of loop unrolling on the runtime performance, the fitted design space is exhaustively searched 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 4.4 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 optimization 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 optimization can be performed ($\widehat{\lambda}_{LU} = 1$) when replicating the compute unit three times. Figure 4.6 reveals the results of LU optimization when $\lambda_{CR} = 1$ and a similar trend is also observed in other cases. By way of contrast, the performance result when $\lambda_{CR} =$

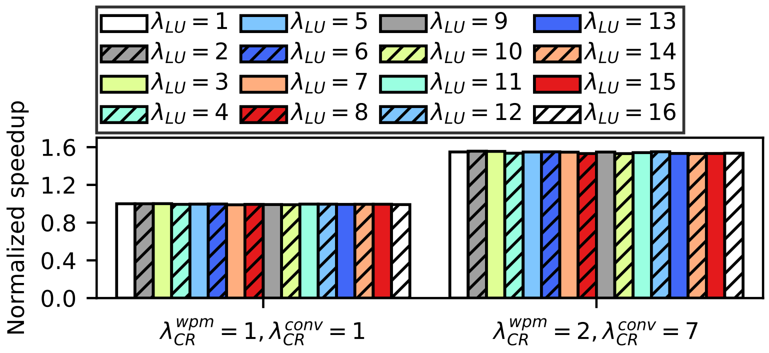


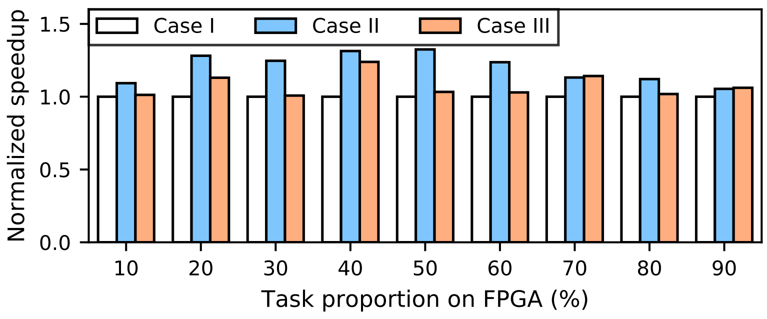
Figure 4.7: Performance comparison of *r-LDA* using different loop unrolling factors, the FPGA task proportion is 50%.

3, $\lambda_{LU} = 1$ is also presented, so as to compare the performance boost of CR and LU optimizations. As seen in Figure 4.6, 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 optimization, as seen from the last bar in Figure 4.6. For this application, CR optimization gains a larger performance improvement than LU optimization.

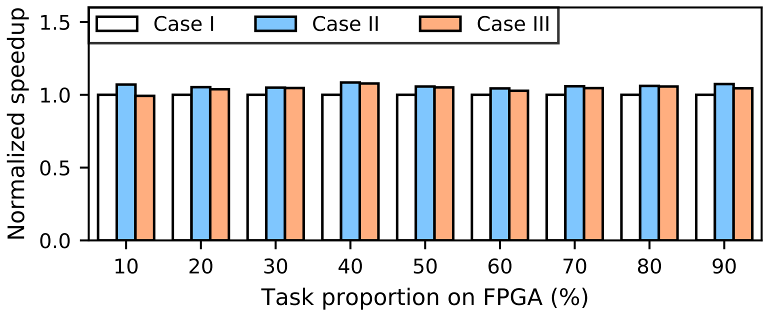
Figure 4.7 depicts the LU optimization results for *r-LDA*. The figure only shows the results when the FPGA task proportion is 50% and other cases turn out similar results. For comparison of CR and LU optimization, the native ($\lambda_{CR}^{wpm} = 1, \lambda_{CR}^{conv} = 1$) and the best ($\lambda_{CR}^{wpm} = 2, \lambda_{CR}^{conv} = 7$) CR configurations are chosen to exhibit their corresponding LU optimization results. As observed from Figure 4.7, 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.

4.4.3.3 Accelerator Execution Overlapping (EO)

Following the configurations in Figure 4.2, the three execution scenarios with different call orders of OpenCL API functions are constructed and the resulted performance outcomes are depicted in Figure 4.8. EO optimization of the *p-LDA* application achieves a considerable performance boost (shown in Figure 4.8a), while the speedup for *r-LDA* is slightly lower (shown in Figure 4.8b). 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



(a) *p-LDA*



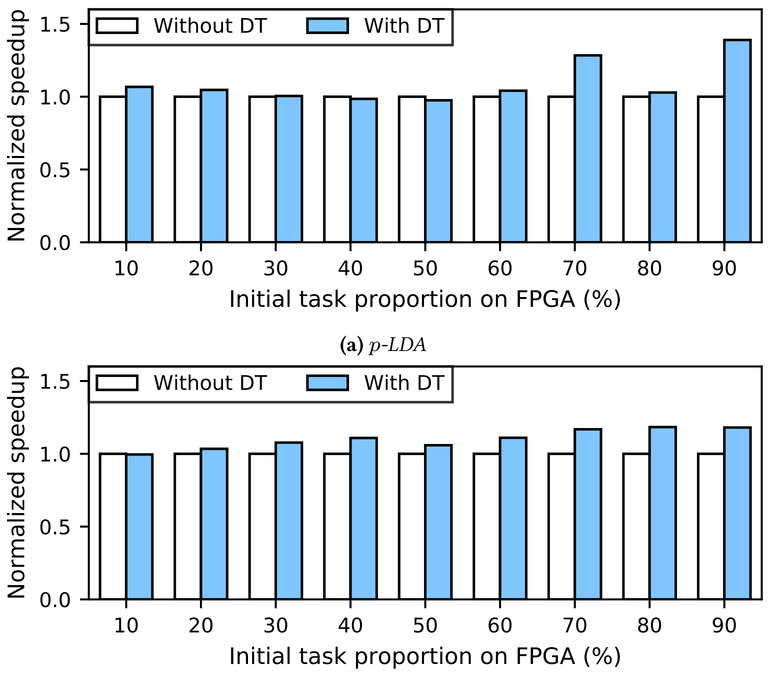
(b) *r-LDA*

Figure 4.8: Performance comparison with different call order of OpenCL API functions.

the previous analysis in Section 4.3.4.

4.4.3.4 Dynamical Workload Tuning (DT)

The dynamical workload tuning mechanism illustrated in Section 4.3.5 is applied to the two case study applications and the comparison of the performance results is presented in Figure 4.9. As can be observed, DT optimization achieves many more performance gains when the task proportion on FPGA is larger. This is especially obvious for *p-LDA*, where the DT-optimized 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 during the evaluation the Quadro K600 GPU has a higher computation power than the Nallatech PCIe-385N 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.



(a) *p-LDA*

(b) *r-LDA*

Figure 4.9: Performance comparison with and without dynamical workload tuning.

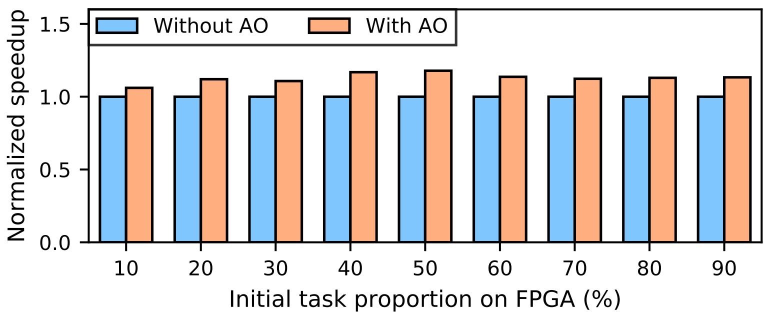
4.4.3.5 Application-oriented Optimization (AO)

Figure 4.10 gives the performance results of using aforementioned AO optimization 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* 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.

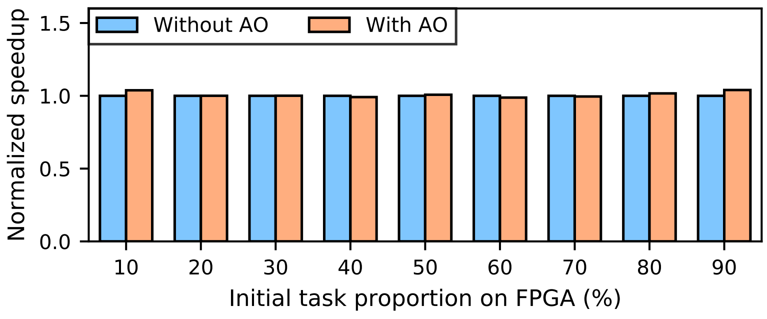
4.4.4 Discussion

4.4.4.1 Performance Benefit

Figure 4.11 shows the performance speedup of the two case study applications with the step-by-step optimizations mentioned above. Overall, the proposed optimization procedure improves the runtime performance of both applications with a large extent. During the evaluation, the observed optimal executions of *p-LDA* and *r-LDA* can improve the performance by an average of 109.21% and 83.48%



(a) *p-LDA*



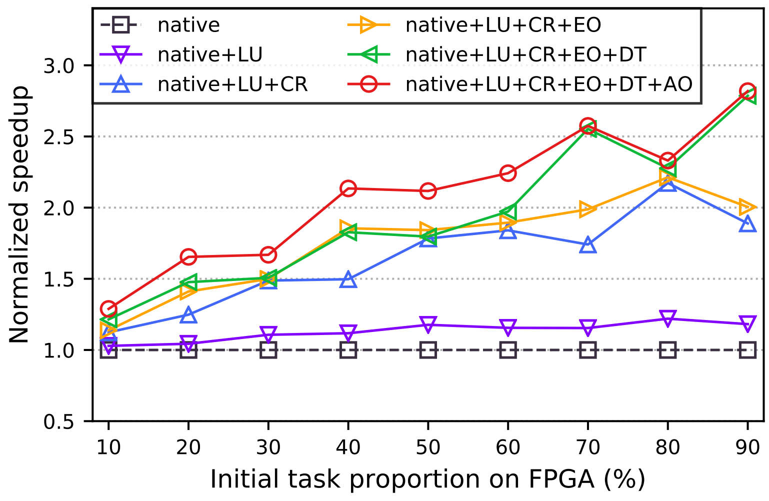
(b) *r-LDA*

Figure 4.10: Performance comparison with and without AO optimization.

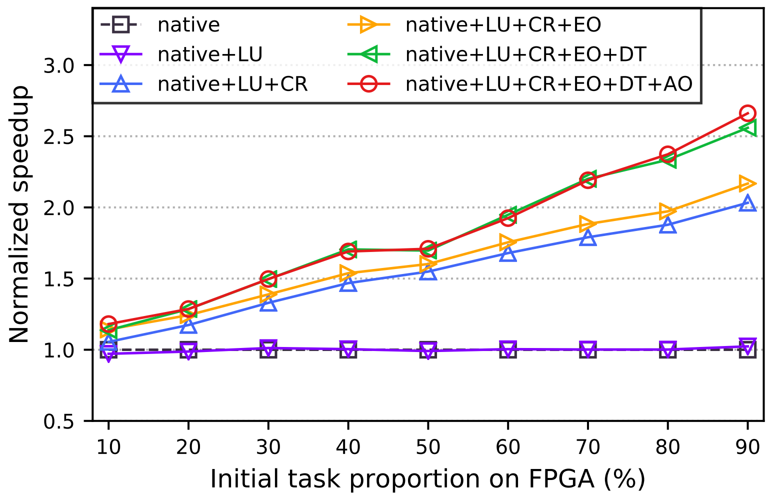
over the native parallel implementations, respectively.

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 optimizations in Figure 4.11a and 4.11b. This reveals that the optimizations favor the FPGA platform and are more efficient when processing time-consuming workloads.

The LU optimization 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 4.4.3.2. A further study on other ADAS or generic scientific computing applications may better demonstrate the effectiveness of the LU optimization. The CR optimization 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 optimizations, on the other hand, further improve the runtime performance. This is extremely important since these three optimization methods are platform-independent and therefore can be seamlessly applied to other heterogeneous systems.



(a) *p-LDA*



(b) *r-LDA*

Figure 4.11: Performance comparison overview with step-by-step optimizations.

4.4.4.2 Scalability Analysis

The optimization procedure proposed in this chapter can be applied to other applications and other heterogeneous parallel systems as well. The reasons are multi-fold. First, the CR optimization is FPGA-related and the LU optimization is valid for both GPU and FPGA. As nowadays GPU and FPGA are mainstream hardware accelerators used for high performance scientific computing, these two optimization methods are applicable for any parallel applications running on GPU and FPGA platforms. Secondly, the EO and DT optimizations 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 optimization methods are therefore suitable for general heterogeneous and reconfigurable computing. Lastly, the AO optimization 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.

4.5 Summary

This chapter proposes a detailed procedure to help guide the performance optimization of parallelized ADAS applications in FPGA-GPU heterogeneous systems. The optimization procedure contains one intra-accelerator and two inter-accelerator optimization methods, as well as both FPGA-specific and application-oriented optimization strategies, to boost the program runtime performance. The optimization results are demonstrated by the evaluation with the two different lane detection applications presented in Chapter 3. Experimental results show that the procedure can effectively reduce the time consumption, and the optimal designs of the two case study applications improve the runtime performance by an average of 109.21% and 83.48% respectively, over the native parallel implementations. Moreover, the procedure can be applied to other applications and other heterogeneous parallel systems as well.

To further investigate the performance of consuming the computation workload on the hardware accelerators, the work in the next chapter uses GPU as the representative platform and proposes a novel performance estimation framework to predict the execution time of running OpenCL kernels on GPUs.

Chapter 5

Performance Estimation for OpenCL Kernels on GPUs

The studies in the previous chapters present how heterogeneous computing can be applied to the implementation of typical ADAS applications, such as LDA. It is observed that performance portability poses a big challenge to programmers for the development of parallel applications in the early design stage. Therefore, motivated by this, this chapter devises a performance modeling approach for the hardware accelerators. We choose GPU as the representative platform since GPU dominates state-of-the-art high performance computing realm. Moreover, the performance modeling work on GPU could also provide insights on similar study with other hardware accelerators.

As a mainstream accelerator, GPU plays a crucial role in scientific computing and therefore lots of research works focused on its performance analysis and prediction. To fully exploit the computing power of GPU, program developers need a deep understanding of its parallel working mechanism, in order to efficiently process the workload at runtime. This poses a challenge for non-expert users because they have no prior knowledge about elaborate parallel programming. To solve this, two approaches, namely auto-tuning and performance estimation, are used to help seek the optimal execution from the vast program design space. Traditional auto-tuning searches through either the whole [74] [75] or a sliced [76] [77] design space, which causes a considerable amount of time. Although this time cost can be reduced by optimization [78] or machine learning based algorithms [79], the relevance between the program input configuration and the resulted performance gain still remains obscure. Therefore, performance estimation is essential to crack the internal program runtime behavior so as to improve the external program execution efficiency.

State-of-the-art GPU performance estimation still suffers from several constraints. First, performance model always needs to be subtly tuned for the appropriate configurations of the target program to obtain convincing estimations. This makes it rather difficult to derive a general-purpose instead of application-oriented method. The reason is that general parallel applications intrinsically include large amounts of adjustable parameters which could individually or jointly affect the runtime performance. Secondly, performance estimation approaches can hardly keep up with the rapid architectural change of contemporary GPUs, due to the continuously promotion and upgrade of COTS products. Although machine learning based methods [80] [81] [82] are applicable to general platforms, the off-line feature sampling of the hardware counter metrics over the huge design space incurs a significant amount of time and the trained model is sensitive to unknown applications. Last but not least, there still exists possibility to improve the accuracy and usability of state-of-the-art GPU performance models [83]. Although fine-grained GPU simulators could give rather accurate estimations, the extremely large time consumption makes it unsuitable for practical use [84] [85].

5.1 Overview

To address the aforementioned issues, this chapter gives a hybrid framework to estimate the performance of parallel applications on the GPU. The framework targets the cross-platform OpenCL [14] workload so that it can still be applied to other accelerators. The high-level kernel source code is first transformed into LLVM [86] Intermediate Representation (IR) instructions, from which the program execution trace is generated based on GPU's philosophy of parallelism. Afterwards a lightweight simulator is developed to dynamically consume the arithmetic and memory access operations in the execution trace in granularity of 32 work items or so-called warps. The hardware specification and micro-benchmarking metrics are also fed to this simulator to obtain the estimated execution time.

In contrast to conventional analytical or machine learning based methods, the proposed framework does not require extra hardware performance counter metrics captured by a third-party profiler, or measurement results which are obtained after executing the whole or a portion of the target kernel before the estimation. Meanwhile, unlike fine-grained GPU simulators that spend simulation time in the scale of hours [87] [88], this framework can give estimation results in a few seconds. For the evaluation, the framework is validated with 20 different kernels from the Rodinia [2] benchmark. The contributions of the work in this chapter are as follows:

- This chapter proposes a hybrid framework that combines source-level analysis and trace-based simulation to predict the performance of GPU kernels. The execution trace of the target kernel is statically generated and then simulated to estimate the runtime performance.
- A loop-based bidirectional branch search algorithm is given to extract the kernel execution trace

that models the warp execution flow of the GPU kernel.

- A lightweight simulator is developed to mimic the kernel execution and then predict the runtime performance results, taking into consideration both the IR instruction pipeline and cache modeling. The simulator can accurately predict the performance of kernels running across different GPU platforms in a few seconds.

- The accuracy and practicability of the framework is demonstrated with the Rodinia [2] benchmark and a real-world application, on four Nvidia GPUs across two generations of recent architectures.

The remainder of this chapter is organized as follows: Section 5.2 is related work and Section 5.3 gives overview of the proposed framework. Section 5.4 and Section 5.5 presents the source-level analysis and trace-based simulation, respectively. Section 5.6 gives evaluation results and Section 5.7 presents a lane detection case study. Section 5.8 concludes the work in this chapter.

5.2 Related Work

There exist lots of studies targeting performance estimation of applications or benchmarks [89] [90] on CPUs [91] [92]. These approaches can provide reference to the performance analysis of GPUs.

The last decade has witnessed an overwhelming amount of research work targeting the performance modeling of GPU platform. Generally speaking, GPU performance modeling techniques can be divided into four categories: analytical, machine learning based, measurement based and simulation based methods. The following paragraphs briefly review the literatures and introduce these approaches in an inductive way.

Analytical methods give an abstraction of the workload and hardware and then use equations to deduce the elapsed time of executing the workload on the target platform. Hong et al. [93] proposed a static model using memory and computation warp parallelism as metrics to estimate the kernel execution time. Kothapalli et al. [94] presented a high level prediction method based on BSP [95], PRAM [96] and QRQW [97] models. A similar work in [98] is also based on BSP [95]. Baghsorkhi et al. [99] used work flow graph and program dependency graph to abstract GPU kernels and then extracted thread- and warp-level parameters to calculate kernel execution time. Zhang et al. [100] presented a quantitative model using micro-benchmarks to obtain hardware metrics consisting of instruction pipeline and memory access time. Song et al. [101] proposed a method using similar mechanism. Wang et al. [102] presented a model involving core and memory frequency scaling. Zhou et al. [103]

proposed a performance analysis framework at assembly instruction level. Most analytical methods require hardware performance counter parameters which indicates the pre-execution of the kernel before prediction. In addition, some models are either outdated for new architectures or difficult to use due to the substantial calibration effort.

Machine learning based methods construct the training data set by sampling program- and platform-related metric features and then use trained model to predict the runtime performance. Baldini et al. [104] used K-nearest neighbor algorithm to estimate GPU performance from multi-core CPU runs. A similar work in [80] used forward feature selection stepwise regression and ensemble prediction to predict GPU performance with CPU implementation. Wu et al. [81] estimated the performance and power of GPU using K-means clustering and neural network. O'neal et al. [82] used random forest to analyze DirectX workload in the pre-silicon design stage. Zhang et al. [105] adopted random forest to investigate performance and power consumption of ATI GPUs. Amarís et al. [106] compared the accuracy of different machine learning methods to predict the performance of Nvidia GPUs. Machine learning based methods can estimate the performance of GPU applications with fast response, since the burdensome training stage is performed off-line. However, it lacks a clear explanation of the relationship between the trained features and the predicted outcome. In addition, the indispensable feature sampling of the hardware counter values over the huge design space is tedious and violates the principle of predicting the performance before the actual execution.

Measurement based methods grasp the program behavior by sample running a fraction of the target workload, or the so-called mini-kernel, to seek the correlation and interference between individual work groups and then estimate the consumed time when the entire kernel is to be executed. Dao et al. [107] developed hand-crafted kernels to crack GPU's warp scheduling policies and then proposed a saturation point based linear model to estimate the kernel execution time. In general, measurement based approaches are universally applicable to different architectures, however the effort to calibrate model parameters for various applications and platforms is onerous.

Simulation based methods simulate in details how GPU processes the workloads in cycle level and reserve the intermediate status of the hardware and software functional modules during runtime. In this way, program behavior and performance can be effectively and accurately sketched. Gerum et al. [108] proposed a method for source level performance simulation of GPU. There exists some widely used simulators such as GPGPU-Sim [3], Barra [109] and Ocelot [110]. However, these simulators are either not actively maintained or subjected to out-of-date architectures. Recently a RTL-level simulator [111] is announced but still few studies are reported.

With regards to GPU simulation acceleration, there exist some research that either choose a portion

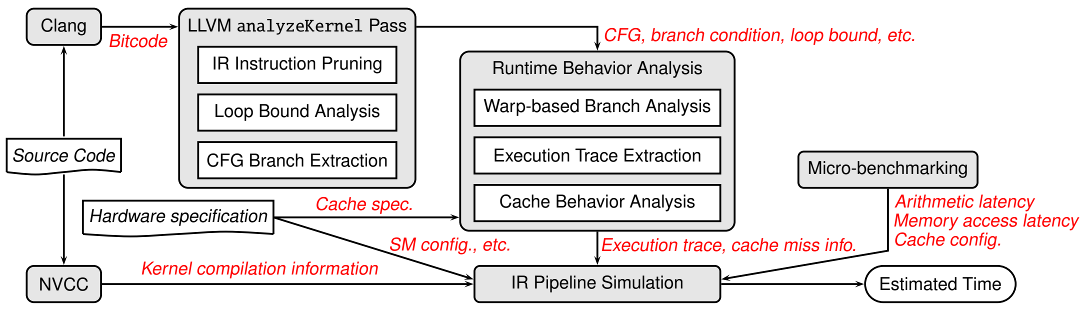


Figure 5.1: Overview of the performance estimation framework.

[112] or perform a pre-characterization [113] of target workloads and then derive the execution time from the simulation results. There are also studies that focus on the generation of GPU benchmarks [114] to reveal GPU's performance spectrum, and modeling of GPU memory systems [115]. These studies are supplementary for GPU performance estimation techniques.

Different to the methods mentioned above, the work in this chapter proposes an analytical and simulation combined framework to predict the performance of parallel workloads on GPU. The framework contains a lightweight IR-level simulator to perform a dummy execution of target kernels on GPU, which does not need any hardware performance counter values as model inputs. With this mechanism, an accurate estimation of the kernel execution time is obtained with rather little simulation time cost.

5.3 Framework Overview

Figure 5.1 gives the overview of the proposed performance estimation framework. The kernel source code is first processed by Clang compiler to generate the LLVM bitcode file that contains IR instructions of the target kernel. Meanwhile, the source file is passed to NVCC compiler to obtain kernel compilation information that includes the detailed runtime resource usage of the kernel, such as the number of used on-chip registers and used shared memory size. The framework mainly contains two modules, i.e. the source-level analysis and the subsequent trace-based simulation. In the source-level analysis module, the kernel bitcode file is processed by an LLVM analyzeKernel pass and the execution trace is subsequently extracted from the kernel runtime behavior analysis. The analyzeKernel pass prunes IR instructions in the basic blocks so that only the arithmetic and memory access operations, which contribute to the kernel execution time, are retained. The execution flow information, such as the loop statements and the branches, is extracted and analyzed for the following execution trace generation.

Given the Control Flow Graph (CFG) and the execution flow information, the kernel runtime behavior is then analyzed and the execution trace is generated in granularity of warps. The cache mis-s/hit information is subsequently obtained according to the cache specification and the execution trace. The simulation module mimics the kernel runtime behavior by virtue of constructing an IR instruction pipeline and consuming the execution trace iteratively. A set of micro-benchmarks are used to calibrate the target GPU to obtain the hardware metrics such as latencies of the arithmetic operations, latencies of the memory access operations, and the cache configurations. These hardware metrics, together with the hardware specification, the kernel compilation information, the kernel execution trace, and the cache miss information, are fed to the simulator to estimate the final execution time.

5.4 Source-level Analysis

5.4.1 LLVM analyzeKernel Pass

The analyzeKernel pass collects the basic blocks and builds the CFG of the target kernel. For each basic block, the IR instructions are documented to construct the intra-block execution trace. The execution flow information used to generate the execution trace is obtained via the three steps illustrated as follows.

5.4.1.1 IR Instruction Pruning

This work assumes that the execution time is mainly consumed by the arithmetic and memory access operations. Therefore for each basic block, the time-cost-irrelevant instructions, such as the LLVM-specific intrinsic annotations llvm.lifetime.start, llvm.lifetime.end, memory address calculation instruction getelementptr, the data type conversion instructions trunc, ext, and so on, are filtered out. Note that here these instructions are only removed from the execution trace, but are still used for the later control flow analysis.

As for function calls, it is observed that the call instruction appears only when invoking ① the OpenCL work-item built-in functions, such as get_global_id, get_local_id, etc., ② the synchronization function barrier, or ③ the LLVM intrinsic functions such as llvm.fmuladd.f32, etc. The subfunctions in the source code are replaced by detailed instructions and therefore non-existent in the bitcode file. Consequently, all the related information about these function calls is recorded to assist the execution trace generation whenever necessary. The OpenCL work-item built-in functions are highlighted because their return values typically serve as memory address indices that directly determine the memory access pattern. The synchronization function is labelled as a flag that notifies the wait signal of the warp execution in the pipeline. The LLVM intrinsic functions are also converted to the corresponding arithmetic operations in the kernel execution trace.

5.4.1.2 Loop Bound Analysis

Instead of deducing a precise value of the loop trip count, this work attempts to estimate the loop bound of each basic block in the loop. The reasons are multifold. First, state-of-the-art static loop analysis is still an open problem [116] and therefore it is impossible to adopt a generic method to obtain the loop trip count of arbitrary code blocks. Secondly, in general, the input of parallel applications is a regular rectangle- or cuboid-like grid that can be ideally decomposed and mapped to the threads on the GPU. The formation of the high-level loop code is regular in the majority of the cases. Lastly, the loop bound manifests an extreme case of the execution of the loop and this scenario should also be considered when analyzing the performance of the kernel executions.

The framework first uses Loopus [117] to analyze the loop bound. It is observed that Loopus can handle simple loops, i.e., when the loop induction variable is a fixed constant. For more complicated loops, the loop bound is first determined by performing an LLVM Scalar Evolution (SE) analysis [118] of the basic blocks in the loop. The SE analysis gives an explicit bound if the target basic block either is within a single-exit loop or has a predictable backedge taken count.

When both Loopus and LLVM SE analysis fail to give outputs, an extra static analysis of the loops is performed to further extract the loop bound. The main idea of this static analysis is to identify the loop induction variable and track its value at the scope of the entire kernel function. First, the exit basic blocks of the loop are collected, from which the true exit basic block is set as the loop latch block. The terminator of the true exit basic block is the loop induction instruction and it is observed that for all the test kernels this instruction is a conditional branch form of a br instruction. The conditional branch has two arguments, of which the first is either the loop induction variable or the loop induction variable updated with an increment of the loop step size, and the second is the end value, which is loop invariant, of the loop induction variable. In LLVM, the loop induction variable is represented as a Static Single Assignment (SSA) and this SSA could be: ① binary operation such as `add`, `mul`, etc. ② `load` instruction. ③ `phi` instruction. For case ①, all the `phi` nodes in the loop header block are traversed and the loop induction variable is set as the `phi` node of which the return value equals the updated loop induction variable, when taking the loop latch block as the incoming value. With regards to case ②, all the store instructions that write data to the pointer argument of this `load` instruction are tracked, by virtue of the memory dependency analysis. The memory write value of the store instruction that lies outside of and closest to the loop is deemed the start value, which is also loop invariant, of the loop induction variable. For case ③, all the phi nodes in the loop header block are also traversed and the `phi` node which equals the loop induction instruction is extracted. Then the updated value of the loop induction variable equals the return value of this `phi` node when taking the loop latch block as the incoming value. With the start value, the end value, and the step size of the loop induction variable obtained, the loop bound is calculated as the induction time of the loop induction variable within the loop: $loopBound = \frac{endValue - startValue}{stepSize}$.

With regards to the nest loops, the analyzed result only indicates the loop bound of the basic block at its current loop level and each of the outer loop bound values equals the loop bound value of one of the preceding basic blocks, which lies exactly at its corresponding loop level. For each basic block in the nest loop, at each upper loop level, the closest preceding basic block is recorded so that the nest loop chain is maintained, for ease of the later execution trace generation. If the deduced loop bound relies on the induction variable of the outer loop, then the different loop bound values are also recorded when the outer loop iterates. During the experiments, the aforementioned static analysis manages to give the loop bound of all the loop basic blocks in the test kernels.

5.4.1.3 CFG Branch Extraction

The triggering condition of each branch is extracted by analyzing the phi and br instructions within the head and tail basic blocks of that branch path. The br instruction is associated with a cmp instruction from which the branch condition can be deduced. The branch condition is an expression that contains the logical operation combination of several variables of which some are conditional variables and the other are constants. The conditional variable is represented as an SSA and it can be further refined with one or more SSAs associated with it. This is done by an iterative search, which terminates when the termination SSA is: ① a kernel argument. ② a temporary variable. ③ a memory load of the data pointed by a kernel argument, which is a pointer parameter.

5.4.2 Runtime Behavior Analysis

5.4.2.1 Warp-based Branch Analysis

To determine whether a branch condition is hit or miss, the execution of the branch paths is evaluated in granularity of warps. As shown in Section 5.4.1.3, the values of the branch conditional variables can be classified into three cases. For case ①, this branch path is easily determined to be hit or miss since the input kernel arguments are known. In case ②, if the temporary variable is thread-ID-dependent, i.e., the variable is the return value of the aforementioned OpenCL work-item built-in functions, then this branch path can also be determined to be hit or miss, given the warp ID and the global and local work size configuration of the target kernel. If the temporary variable is the loop induction variable, this branch path can also be masked or unmasked, depending on the logical result of the branch condition at different loop iterations. For the remaining cases this branch path is assumed to be always hit.

For case ③, because the value of this memory load can only be determined at runtime, for the sake of conservation this branch path is also assumed to be always hit.

5.4.2.2 Execution Trace Generation

Let's first consider how GPU walks along the CFG to execute the kernel. For Nvidia GPUs, each OpenCL work item instance is mapped to a thread and a group of 32 threads are bound together to execute the instance in lock-step manner. This group of threads is called a warp for Nvidia GPUs and the counterpart for AMD GPUs is termed wavefront. When there exists branch divergence within a warp, the threads would consume the instructions in both branch paths and each thread only reserves the processed result of the path where the branch condition is hit. Turning back to the CFG, the basic blocks within different branches are consecutively visited as if they are sequentially processed.

The execution trace is generated in granularity of warps. Therefore for the case when the branch condition is thread-ID-dependent, the branch miss information is transformed and associated with the warp ID, given the global and local work size configurations. The basic block is represented as the data structure shown in Listing 5.1.

```
struct BasicBlockInfo {
        string BBName; // name of the current BB
        list<int> branchMissWarpID; // IDs of the warps that trigger the branch miss
        // branch miss information at different loop iteration
        // string: name of the basic block that triggers the branch miss
        // int: the exact iteration number for basic block #string when branch miss
        map<string, int> branchMissLoopConfig;
        int loopDepth; // greater than 1 when current BB is in a loop
        string loopBoundExpr; // the loop bound expression
        // BBs of which the loop bounds determine current BB's loop bound
        vector<string> associatedBBs;
        string precedBB; // preceding BB closest to current BB at upper loop level
        vector<int> bounds; // integer values of the loop bounds at each loop level
        vector<int> unvisitedCount; // store the visited counters at each loop level
        bool isVisited; // true if current BB is visited over at each loop level
};
list<BasicBlockInfo> BBInfoList; // list of data description for BBs in the CFG
```

Listing 5.1: Sample code of the basic block data description.

The information about the branch miss due to warp divergence and loop iterations is respectively stored in the branchMissWarpID and branchMissLoopConfig fields. The loopDepth field indicates the loop depth of the basic block. Particularly, this value is set to 1 if the basic block is not in a loop. The analyzed loop bound result is stored in the loopBoundExpr field. As this expression only indicates the loop bound of the basic block at its current loop level, the actual loop bound values at each loop

Algorithm 5: Execution Trace Generation

```
Input: CFG Entry Node B, CFG Exit Node E, Backedge Set $\mathbb{BE}$, Non-backedge Set $\mathbb{NE}$, Basic Block Data
       Description List BBInfoList, Warp ID wid, Mask Array $M$
Output: Kernel Execution Trace T
1  $\mathbf{T} \leftarrow \varnothing, \mathbb{T} \leftarrow \varnothing, \tau \leftarrow$ B        ▷ Initialize the execution trace with entry node B
2  updateExecTrace($\tau$, $\mathbb{T}$, BBInfoList, wid, $M$)
3  ST $\leftarrow \varnothing$        ▷ Initialize a stack to store the header nodes of multiple branch paths
4  while $\tau \neq E \wedge \neg$ E.isVisited do        ▷ Terminate when exit node is visited
5      $\tau \leftarrow$ getTraceSuccNode($\tau$, B, BBInfoList, ST, $\mathbb{T}$, $\mathbb{BE}$, $\mathbb{NE}$)
6      updateExecTrace($\tau$, $\mathbb{T}$, BBInfoList, wid, $M$)
7  for $i \leftarrow 0$ to $\mathbb{T}$.size() do
8      if $M[i]! = 0$ then        ▷ Remove branch miss nodes from the generated trace
9          $\mathbf{T} \leftarrow \mathbf{T} + \{\mathbb{T}.at(i)\}$
10 return T
```

level are calculated each time this basic block is visited and these values are stored in the bounds field. If the loop bound of the basic block is dependent on other basic blocks, these associated basic blocks are stored as well (the associatedBBs field). The preceding basic block that is closest to the current basic block but lies at the upper loop level is stored in the precedBB field so as to maintain the nest loop chain. During the execution trace generation, the visited counters of the basic block (the unvisitedCount field) are recorded to indicate the visited status of the basic block, i.e., at which loop level and with how many times the current basic block is already visited. The isVisited boolean is set to TRUE only if the basic block is visited over at each loop level with the number of times equal to the actual loop bound. Finally, the data descriptions of all the basic blocks in the CFG are stored in a list BBInfoList.

Given the kernel CFG $G = (\mathbb{V}, \mathbb{E}, \mathsf{B}, \mathsf{E})$, where $\mathbb{V}$ is the set of basic block nodes, $\mathbb{E}$ is the set of basic block connections, B is the entry node and E is the exit node, the kernel execution trace is generated via a loop-based bidirectional branch search shown in Algorithm 5. The CFG is first passed to a circular check to split the edge set $\mathbb{E}$ into the backedge set $\mathbb{BE}$ and the non-backedge set $\mathbb{NE}$. In this way, the CFG is transformed into a Directed Acyclic Graph (DAG) and the paths between any two nodes can be represented as finite sequences of which all the nodes belong to the non-backedge set $\mathbb{NE}$. By default we have the following assumption:

*Denote $\mathbb{V}_{\boldsymbol{c}}$ as a set of nodes that construct a circle **c** in the CFG, if there exists another circle node set $\mathbb{V}_{\boldsymbol{c}'}$, then formula $(\mathbb{V}_{\boldsymbol{c}} \subset \mathbb{V}_{\boldsymbol{c}'}) \vee (\mathbb{V}_{\boldsymbol{c}} \supset \mathbb{V}_{\boldsymbol{c}'}) \vee (\mathbb{V}_{\boldsymbol{c}} \cap \mathbb{V}_{\boldsymbol{c}'} = \varnothing)$ always holds.*

This assumption is reasonable for real-world program because a node in a loop can only be reached from the nodes in its surrounding loops but can never reach the nodes in another loop that is beyond all of the outer loop layers of the original loop. The above assumption ensures that no backedge would

Algorithm 6: updateExecTrace(τ, $\mathbb{T}$, **BBInfoList**, wid, M)

```
Input: Candidate Node $\tau$, Candidate Trace $\mathbb{T}$, Basic Block Data Description List BBInfoList, Warp ID
       wid, Mask Array $M$
1  $\tau$.bounds $\leftarrow$ calcLoopBound($\tau$, BBInfoList)                    ▷ update loop bounds
2  loopLevelVisitedCount $\leftarrow$ 0, unvisitedLoopLevel $\leftarrow$ 0
3  isBranchMissWarp $\leftarrow$ FALSE, isBranchMissLoop $\leftarrow$ FALSE
4  for $i \leftarrow 0$ to $\tau$.loopDepth do
5      if $\tau$.unvisitedCount$\langle i \rangle$ = 0 then                    ▷ i-th loop level is visited
6          loopLevelVisitedCount $\leftarrow$ loopLevelVisitedCount+1
7      else              ▷ currently the trace iterates exactly at the i-th level of the loop
8          unvisitedLoopLevel $\leftarrow i$
9          break
10 if loopLevelVisitedCount $\neq \tau$.loopDepth then
11     for $j \leftarrow 0$ to unvisitedLoopLevel do                             ▷ reset loop bounds
12         $\tau$.unvisitedCount$\langle j \rangle \leftarrow \tau$.bounds$\langle j \rangle$
13     $\tau$.unvisitedCount$\langle 0 \rangle \leftarrow \tau$.unvisitedCount$\langle 0 \rangle - 1$
14 else          ▷ $\tau$ is visited over when the visited-loop-level count equals the loop depth
15     $\tau$.isVisited $\leftarrow$ TRUE
16 isBranchMissWarp $\leftarrow$ checkBranchMissWarp($\tau$, wid)
17 isBranchMissLoop $\leftarrow$ checkBranchMissLoop($\tau$, BBInfoList)
   // set the mask value to 0 when $\tau$ is a branch miss node, otherwise set it to 1
18 $M$.add($\neg$ isBranchMissWarp $\wedge$ $\neg$ isBranchMissLoop)
19 $\mathbb{T} \leftarrow \mathbb{T} + \{\tau\}$
```

wander among different circles in the CFG.

To generate the kernel execution trace, a loop-based bidirectional branch search of the CFG is performed. As shown in Algorithm 5, the execution trace starts from the entry node B and terminates when the exit node E is visited. A node stack ST is used to store the header nodes of multiple branch paths. An array M is used to store the mask values for each node in the candidate trace $\mathbb{T}$. The mask value is set to 0 when the node to be appended to $\mathbb{T}$ is a branch miss node. For each candidate node τ to be appended to $\mathbb{T}$, a function updateExecTrace() is invoked to update the visited counters of τ and another function getTraceSuccNode() is used to obtain the successor node of τ to be appended to $\mathbb{T}$. Finally, the branch miss nodes are removed from $\mathbb{T}$, based on the mask array M, to generate the kernel execution trace **T**.

The implementation of function updateExecTrace() is shown in Algorithm 6. First, the loop bounds of the candidate node τ can be determined because these values are related to the loop bound expression (τ.loopBoundExpr) and the current loop iterations and loop bounds of the associated basic blocks (τ.associatedBBs), and all these information can be calculated before visiting τ at its current loop level (Line 1 in Algorithm 6). Subsequently, the visited counters of τ are checked to determine at which loop level the node τ is visited (Line 4—9 in Algorithm 6). Each time the unvisited count value at the innermost loop level is decreased by 1 (Line 13 in Algorithm 6). The update of the visited counters is implemented via a decrement operation with borrowing, i.e., each time the unvisited count value at loop level λ is reduced to zero, this value is reset to the loop bound at loop level λ and the unvisited count at loop level $(\lambda + 1)$ is decreased by 1 (Line 11—12 in Algorithm 6). If the unvisited count values of τ at all loop levels are zero, then this node is labeled as visited (Line 15 in Algorithm 6). At last, the branch miss information is used to determine whether τ is a branch miss mode. The corresponding mask value is written to the mask array M and node τ is appended to the candidate trace $\mathbb{T}$ (Line 16—19 in Algorithm 6).

Algorithm 7 gives the detailed implementation of the function getTraceSuccNode(). To find the successor node of τ to be appended to $\mathbb{T}$, the backedge set $\mathbb{BE}$ is first searched to get the destination node (element in $\mathbb{D}_{\mathbb{BE}}$) of the backedge whose source node is τ (Line 2—15 in Algorithm 7). The candidate backedge nodes (elements in $\mathbb{D}'_{\mathbb{BE}}$) are chosen from the nodes in $\mathbb{D}_{\mathbb{BE}}$ of which the unvisited count value at the innermost loop level equals neither zero nor the loop bound value (Line 4—6 in Algorithm 7). The successor node of τ to be appended to $\mathbb{T}$ is either itself if τ is in $\mathbb{D}'_{\mathbb{BE}}$ or the closest-to-τ node in the intersection set of $\mathbb{D}'_{\mathbb{BE}}$ and the path node set $\mathbb{P}_{\mathbb{B}}$ in which each node denotes a reachable path to τ (Line 8—15 in Algorithm 7).

If there exists no backedge that starts from τ, or all the backedges starting from τ are visited N times where N is the loop bound in the innermost level, the non-backedge set $\mathbb{NE}$ is searched to obtain the closest-to-τ non-backedge destination node set $\mathbb{D}'_{\mathbb{NE}}$ (Line 16—38 in Algorithm 7). The first node in $\mathbb{D}'_{\mathbb{NE}}$ is chosen as a candidate successor node s_{ne} if none of the nodes in $\mathbb{D}'_{\mathbb{NE}}$ is a source node of a backedge, otherwise this source node becomes s_{ne} (Line 26 in Algorithm 7). If node stack ST is not empty and the stack top node ST$\langle$topElement$\rangle$ lies between a reachable path from the entry node B to s_{ne}, then the successor node of τ to be appended to $\mathbb{T}$ is ST$\langle$topElement$\rangle$, otherwise the successor node to be appended to $\mathbb{T}$ is s_{ne} (Line 27—33 in Algorithm 7). If ST is empty, then s_{ne} is the successor node of τ to be appended to $\mathbb{T}$ and the remaining nodes in $\mathbb{D}'_{\mathbb{NE}}$ are pushed into ST (Line 35—38 in Algorithm 7).

If all the edges starting from τ are visited, then the stack top node ST$\langle$topElement$\rangle$ is popped as successor node of τ to be appended to $\mathbb{T}$ (Line 40—41 in Algorithm 7).

5.4.2.3 Cache Behavior Analysis

As modern GPUs have rather complex memory hierarchy that comprises caches, we first use micro-benchmarks to obtain the cache hit and miss latencies of the local, constant, and global memory ac-

Algorithm 7: getTraceSuccNode(τ, B, **BBInfoList**, ST, $\mathbb{T}$, $\mathbb{BE}$, $\mathbb{NE}$)

Input: Current Trace Tail Node τ, CFG Entry Node B, Basic Block Data Description List **BBInfoList**, Node Stack ST, Candidate Trace $\mathbb{T}$, Backedge Set $\mathbb{BE}$, Non-backedge Set $\mathbb{NE}$

Output: Candidate Trace Successor Node τ (overwritten)

```
$\mathbb{D}_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{BE}} \leftarrow \varnothing$, $\mathbb{D}_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{D}'_{\mathbb{NE}} \leftarrow \varnothing$, $\mathbb{S}_{\mathbb{NE}} \leftarrow \varnothing$
// first try to find a candidate successor node from the backedges
if $\exists$ $be \in \mathbb{BE}$, $\tau = be.srcNode$ then
    $\mathbb{D}_{\mathbb{BE}} = \{$be.destNode | be $\in \mathbb{BE}$, $\tau =$ be.srcNode$\}$
    foreach $d_{be} \in \mathbb{D}_{\mathbb{BE}}$ do                  ▷ get candidate nodes that are not visited over
        if $d_{be}.unvisitedCount\langle 0\rangle \,\%\, d_{be}.bounds\langle 0\rangle \neq 0$ then
            $\mathbb{D}'_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} + \{d_{be}\}$
    if $\mathbb{D}'_{\mathbb{BE}} \neq \varnothing$ then
        if $\tau \in \mathbb{D}'_{\mathbb{BE}}$ then                  ▷ there is a backedge from $\tau$ to itself
            return $\tau$                                            ▷ $\tau$ is not visited over at its current loop level
        else
            foreach $d'_{be} \in \mathbb{D}'_{\mathbb{BE}}$ do
                $\mathbb{P}_{\mathbb{B}} \leftarrow$ getNodesInPath($d'_{be}$, $\tau$)
            $\mathbb{I}_{\mathbb{BE}} \leftarrow \mathbb{D}'_{\mathbb{BE}} \cap \mathbb{P}_{\mathbb{B}}$
            if $\mathbb{I}_{\mathbb{BE}} \neq \varnothing$ then
                return $\mathbb{I}_{\mathbb{BE}}\langle 0\rangle$                  ▷ return the closest-to-$\tau$ node
// backedge search fails, try to find the successor node from the non-backedges
else if $\exists$ $ne \in \mathbb{NE}$, $\tau = ne.srcNode$ then
    $\mathbb{D}_{\mathbb{NE}} = \{$ne.destNode | ne $\in \mathbb{NE}$, $\tau =$ ne.srcNode$\}$
    foreach $d_{ne} \in \mathbb{D}_{\mathbb{NE}}$ do                  ▷ get the closest-to-$\tau$ non-backedge nodes
        $\mathbb{P}_{\mathbb{N}} \leftarrow$ getNodesInPath($\tau$, $d_{ne}$)
        if $\mathbb{D}_{\mathbb{NE}} \cap \mathbb{P}_{\mathbb{N}} = \varnothing$ then
            $\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} + \{d_{ne}\}$
    if $\mathbb{D}'_{\mathbb{NE}} \neq \varnothing$ then
        foreach $d'_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ do              ▷ get nodes in other backedges
            if $\exists$ $be^n \in \mathbb{BE}$, $d'_{ne} = be^n.srcNode$ then
                $\mathbb{S}_{\mathbb{NE}} \leftarrow \mathbb{S}_{\mathbb{NE}} + \{d'_{ne}\}$
        $s_{ne} = (\mathbb{S}_{\mathbb{NE}} \neq \varnothing)$ ? $\mathbb{S}_{\mathbb{NE}}\langle 0\rangle$ : $\mathbb{D}'_{\mathbb{NE}}\langle 0\rangle$          ▷ candidate successor
        if $ST \neq \varnothing$ then
            $\mathbb{P}_{\mathbb{S}} \leftarrow$ getNodesInPath(B, $s_{ne}$)
            if $ST\langle topElement\rangle \in \mathbb{P}_{\mathbb{S}}$ then
                $\tau \leftarrow$ ST$\langle$topElement$\rangle$, ST.$pop()$
            else                                                     ▷ the stack top node denotes another branch path
                $\tau \leftarrow s_{ne}$                                    ▷ but the current path is not visited over
            return $\tau$
        else                                                         ▷ the current path is the last path of the current branch
            $\mathbb{D}'_{\mathbb{NE}} \leftarrow \mathbb{D}'_{\mathbb{NE}} - \{s_{ne}\}$
            foreach $d''_{ne} \in \mathbb{D}'_{\mathbb{NE}}$ do          ▷ store the remaining header nodes
                ST.$push(d''_{ne})$
            return $s_{ne}$
// all edges starting from $\tau$ are visited, get the successor from the node stack
else
    $\tau \leftarrow$ ST$\langle$topElement$\rangle$, ST.$pop()$
    return $\tau$
```

cesses. As the local memory in OpenCL is mapped to the GPU shared memory, notice that the local memory access has no caching issue, therefore it does not differentiate the cache hit/miss access, which is also observed and demonstrated by the micro-benchmarking results. When handling the constant and the global memory accesses, the SMs first try to fetch the data in the constant or L2 data cache and if cache miss occurs, the data are fetched again from the off-chip DRAM. To model this caching behavior, the constant data cache and the L2 cache are dissected with micro-benchmarks [119] [120] to obtain the detailed cache configurations, such as the cache size, the cache line size, and the cache associativity. In OpenCL, the observed constant memory size is 64KB and the DRAM size is obtained from the official documents. The L2 cache size is obtained from the CUDA built-in querying commands. This work assumes that all the caches use the Least Recently Used (LRU) replacement policy.

For each memory access, i.e. the load or store IR instruction in the execution trace, the memory referencing address is obtained to analyze the number of memory transactions that a warp would perform for this memory instruction, since the threads in a warp often coalesce the data fetch if the memory addresses for the threads are contiguous. As the kernel is not executed on the real platform, a virtual addressing space of the constant data cache and the L2 cache is constructed, and then the specific addresses are assigned to the constant and global variables according to their data size. In this way, the cache behavior is analyzed using the reuse distance theory and the cache hit/miss for each memory transaction is estimated given the cache configuration [121].

5.4.2.4 Discussion

Limitation. As this framework does not use profiling or measurement results of the target kernel, the execution behavior of irregular kernels cannot be exactly determined by the static analysis. Consequently the loop bound analysis and the warp-based branch analysis produce slightly pessimistic results when the values of the loop trip count and the branch condition rely on the values of the program runtime parameters. However, the major part of the applications that can potentially benefit from GPU acceleration exhibit relatively regular shapes, i.e., the loop trip count is rather stable and the number of branches is minimized by the program developer as well. With regards to the kernels with data-dependent divergence, because the static analysis module can still extract the branch condition and loop iteration variables of the control statements, the dynamic execution flow can also be determined if all the input data are known in advance. However this needs the step-by-step simulation of the program execution, which may incur much more time consumption.

Scalability analysis. The proposed performance analysis framework in this chapter targets OpenCL kernels and therefore it can be extended to any platform that supports OpenCL applications. For other parallel languages such as CUDA, since the framework takes LLVM bitcode files as input, CUDA kernels can also be analyzed if either the LLVM bitcode file of the kernel can be obtained or the CUDA kernels can be transformed into the OpenCL counterparts.

5.5 Trace-based Simulation

The execution trace **T** generated from the source-level analysis is warp-ID-dependent and during the simulation each warp consumes its corresponding trace. To estimate the kernel execution time with given program input and the global and local work size configurations, the framework constructs an IR instruction pipeline and then simulate the trace on the pipeline in granularity of a round of active work groups.

5.5.1 IR Instruction Pipeline

5.5.1.1 Determining the Number of Active Work Groups

The number of active warps can be obtained by the official CUDA occupancy calculator. For descriptive integrality, the derivation is given in details as follows. Given a kernel with NDRange configuration as global work size S_{global} and local work size S_{local}. Each work item consumes N_{reg} on-chip registers (private memory) and N_{sm} bytes shared memory (local memory). The number of active work groups N_{awg} per Streaming Multiprocessor (SM) is subject to three constraints: the architectural limit, the register limit, and the shared memory limit. The architectural limit of the allocatable work groups is

$$N_{lim\_wg\_arch} = min(B_{wg\_SM}, \lfloor \frac{B_{warp\_SM}}{N_{warp\_per\_wg}} \rfloor) \tag{5.1}$$

$$N_{warp\_per\_wg} = \lceil \frac{S_{local}}{T_{warp}} \rceil, \tag{5.2}$$

where $N_{warp\_per\_wg}$ is the number of warps per work group, T_{warp} is the number of threads per warp, B_{wg_SM} and B_{warp_SM} is the maximum allocatable work groups and warps per SM, respectively. The number of total on-chip registers limits the maximum concurrent work group as

$$N_{lim\_wg\_reg} = \begin{cases} 0, \quad N_{reg} > B_{reg\_wi} \\ \lfloor \frac{N_{lim\_warp\_reg}}{N_{warp\_per\_wg}} \rfloor \times \lfloor \frac{B_{reg\_SM}}{B_{reg\_wg}} \rfloor, otherwise \end{cases} \tag{5.3}$$

$$N_{lim\_warp\_reg} = floor(\frac{B_{reg\_wg}}{ceil(N_{reg} \times T_{warp}, G_{reg})}, G_{warp}), \tag{5.4}$$

where B_{reg_wi}, B_{reg_SM}, and B_{reg_wg} are the maximum allocatable registers per work item, SM, and work group, respectively. G_{reg} and G_{warp} are the minimum allocation unit of register and warp, respectively. $N_{lim\_warp\_reg}$ is the maximum number of potentially allocatable active warps subject to limited on-chip registers. $ceil(x, y)$ and $floor(x, y)$ are functions used to round the value x up and down to the nearest multiple of y, respectively. The number of active work groups due to shared memory limit is calculated as

$$N_{lim\_wg\_sm} = \begin{cases} 0, & N_{sm\_alloc} > B_{sm\_wg} \\ \lfloor \frac{B_{sm\_SM}}{N_{sm\_alloc}} \rfloor, & otherwise \end{cases} \tag{5.5}$$

$$N_{sm\_alloc} = ceil(N_{sm}, G_{sm}), \tag{5.6}$$

where B_{sm_wg} and B_{sm_SM} are the maximum allocatable shared memory size per work group and SM, respectively. N_{sm_alloc} is the actual allocated shared memory size per work group and G_{sm} is the minimal shared memory allocation size.

With Equation (5.1), (5.3) and (5.5), the number of active work groups for a kernel is therefore determined as

$$N_{awg} = min(N_{lim\_wg\_arch}, N_{lim\_wg\_reg}, N_{lim\_wg\_sm}). \tag{5.7}$$

5.5.1.2 Determining the Latencies of the Arithmetic and Memory Access Operations

The execution trace consists of the arithmetic and memory access operations to be executed on the target GPU. To obtain the latencies of these operations, a set of OpenCL micro-benchmarks is used to measure the arithmetical and memory throughput of the target GPU [122]. This work considers the basic arithmetic operations listed in Table 5.1 and the latencies of memory access from the OpenCL local, constant, and global memory. The private memory access is essentially on-chip register read/write and this memory access is deemed arithmetic operation since the pre-allocated registers are excluded by individual work item and therefore accessing them incurs no contention latency. The profiling of basic arithmetic operations is conducted over a set of computation-intensive kernels which repeatedly execute the desired operations for millions of times. To prevent the compiler optimization, the source and destination operands are exchanged after each time the operation is completed. By fine tuning the local and global work size of each kernel, the number of active warps per SM is thereupon dynami-

Table 5.1: List of profiled arithmetic operation types.

Data type	Operations
int/uint	add, sub, mul, div, rem, mad, shl, shr
float/double	add, sub, mul, div, mad
float/double	sin, cos, tan, exp, log, sqr, sqrt

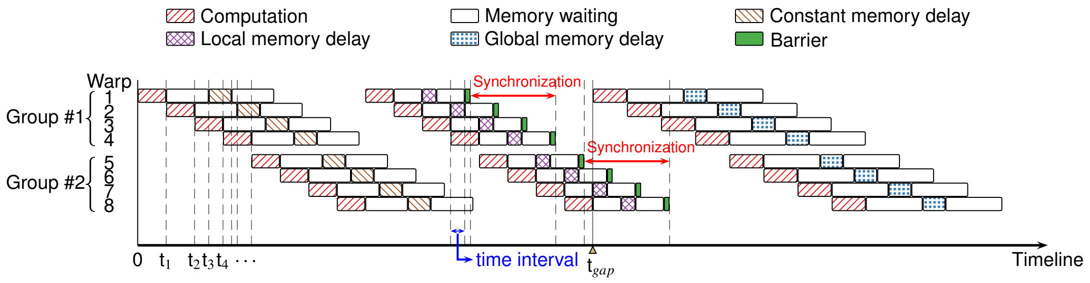


Figure 5.2: Simulation of a sample execution trace on the warp pipeline.

cally regulated so as to obtain the corresponding execution latencies ranging from the minimal to the maximal attainable number of active warps.

With regards to the memory access, the framework uses pointer chasing to generate continuous data access to a large array filled in the respective memory space. To measure the cache hit and miss latencies, the pointer chasing stride offset is set to 1 and the cache line size, respectively. During the simulation, the memory latency is scaled with a factor equaling the ratio of the maximal to the actual number of active warps, since all the active warps share the memory bandwidth equally. The profiled results of the memory access characterize the average time period that starts from the memory instruction issue stage to the final data acquisition stage. We term this whole time period as the memory access "latency" and this time cost is differentiated from the time period when the data is actually read/written by the hardware control circuit, which is called memory access "delay". This work assumes that memory access delay is fixed while memory access latency varies depending on whether the access is a cache hit or miss.

5.5.2 Calculating the Trace Simulation Time

Given the kernel execution trace **T**, the latencies of the arithmetic and memory access operations LAT, and the cache miss information cacheMissInfo in the trace, a lightweight simulator is developed to maneuver a dummy execution of the kernel with a round of active work groups N_{awg}. A sample simulation of this active work groups is conducted on the IR instruction pipeline and the time consumption can be denoted as $T^{spec(\mathsf{LAT},\mathsf{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})}$. The estimated execution time of the kernel run is calculated as

$$T_{kernel} = T^{spec(\mathsf{LAT},\mathsf{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})} \times \lceil \frac{S_{global}}{S_{local} \times N_{SM}} \rceil \times \frac{1}{N_{awg}}, \tag{5.8}$$

where N_{SM} is the total number of SMs on the target GPU.

The trace simulation is implemented with a group of active warps continually consuming the arithmetic and memory access operations in presence of the shared resource and cache contention. For each memory access, it is assumed that memory read/write delay is constant while the waiting period of servicing memory read/write varies depending on whether the memory access is cache hit or miss. The latency of memory read/write is modeled as three parts: the pre-waiting latency, the read/write delay and the post-waiting latency, of which the sum is the profiled cache hit or miss latency.

For better illustration, Figure 5.2 gives an example to illustrate how an execution trace is fed into the warp pipeline. The sample trace is defined as (*comp, constMemAccess, comp, localMemAccess, barrier, comp, globalMemAccess*). The number of active work groups is 2 and each work group consists of 4 warps. Each time before a warp consumes a new operation in the trace, it will first check whether the required contention resource is idle. If so it would lock the resource and notify a value denoting the latency of consuming the current operation, otherwise it would notify a value denoting the time needed to wait until the resource is released. If the warp hits a barrier for synchronization, it will notify value 0 and wait for the other warps in the same work group to arrive at this barrier. A global timer starts at time point 0 and increases by a unit of time interval (indicated by the time point of $t_1, t_2, t_3, \dots$ on the Timeline-axis in Figure 5.2) when all the active warps have notified a time value. During every time interval, the timer checks the notification time of each warp and chooses the minimum positive time value as the incremental time interval. Once all the active warps finish their own traces, the global timer gives the total time of consuming the execution trace.

5.5.3 Discussion and summary

As observed in Figure 5.2, the execution time of the sample trace is computation-bound and the synchronization latency is hidden by the computation pipeline. However, if there exist more memory access operations before the barrier, there would be a gap between the 2-nd and 3-rd computation component (indicated by time point t_{gap} in Figure 5.2) and in this case the synchronization latency would contribute to the final execution time. Consequently, analytical performance estimation methods are normally subject to kernel variances because the order that the computation and memory components appear in the execution trace is inconstant and unpredictable, which has a tremendous impact on calculating the consuming latency of the instruction pipeline.

Table 5.2 summarizes the parameters used in the proposed framework. As shown, this method requires neither the pre-execution of the whole or a portion of the target kernel nor the profiled results of the hardware performance counter metrics. The used information are the program configuration parameters, kernel compilation report, and the hardware specifications. The micro-benchmarking metrics are obtained by calibrating the target GPU once and these data can be reused for the performance prediction of all the kernels running on this platform. During the simulation, each kernel takes the same kernel compilation results and the same group of execution traces as inputs. For each specific run, only the corresponding global and local work size configurations are fed to the simulator to obtain the estimated results. Moreover, only a round of active work groups is actually fed to the pipeline and therefore the simulation time cost is small.

Overall speaking, compared with traditional architectural simulation methods [3], the proposed framework requires less input information and can give faster estimation outcomes. In addition, the framework does not require the instruction trace representatives generated from the kernel runs, which is subject to specific workloads and may incur substantial effort when the input parameters vary a lot.

5.6 Evaluation Results

5.6.1 Evaluation Setup

Four COTS GPUs are used to evaluate the performance estimation framework and the detailed information is shown in Table 5.3. These GPUs are from recent Kepler and Maxwell architectures with different compute capacities so as to demonstrate the robustness of the framework. The framework is tested with 20 OpenCL kernels from the Rodinia [2] benchmark. The evaluation uses the default

Table 5.2: Summary of the parameters used in the performance estimation framework.

No.	Parameter	Definition	Obtained
1	S_{global}	Number of global work size	Program configuration
2	S_{local}	Number of local work size	Program configuration
3	N_{reg}	Number of registers used per work item	Kernel compilation
4	N_{sm}	Bytes of shared memory used per work item	Kernel compilation
5	B_{wg_SM}	Maximum allocatable work groups per SM	Hardware specification
6	B_{warp_SM}	Maximum allocatable warps per SM	Hardware specification
7	B_{reg_wi}	Number of maximum allocatable registers per work item	Hardware specification
8	B_{reg_SM}	Number of maximum allocatable registers per SM	Hardware specification
9	B_{reg_wg}	Number of maximum allocatable registers per work group	Hardware specification
10	B_{sm_wg}	Bytes of maximum allocatable shared memory per work group	Hardware specification
11	B_{sm_SM}	Bytes of maximum allocatable shared memory per SM	Hardware specification
12	G_{sm}	Number of minimum allocation bytes of shared memory	Hardware specification
13	G_{reg}	Number of minimum allocation unit of registers	Hardware specification
14	G_{warp}	Number of minimum allocation unit of warps	Hardware specification
15	$FREQ_{core}$	Clock frequency of the thread core on target GPU	Hardware specification
16	N_{SM}	Number of SMs on target GPU	Hardware specification
17	T_{warp}	Number of thread cores per warp	Hardware specification
18	N_{awg}	Number of active work groups	Equation 5.7
19	LAT	Latencies of arithmetic and memory access operations	Micro-benchmarking
20	cacheMissInfo	Cache hit/miss information about the memory access	Cache behavior analysis
21	**T**	Kernel execution trace	Algorithm 5
22	$T^{spec(\text{LAT},\text{cacheMissInfo})}_{pipeline(N_{awg},\mathbf{T})}$	Estimated execution time in a round of active work groups	Simulation
23	T_{kernel}	Estimated total kernel execution time	Equation 5.8

input from the benchmarks and conduct a design space exploration that results in a total of 306,558 estimation runs. The simulation is performed on a desktop computer with an Intel® Core™ i7-3770 CPU.

5.6.2 Prediction Results

5.6.2.1 Accuracy

Table 5.4 presents the experimental results. The third column in Table 5.4 lists the number of total design configurations of each kernel and the fourth column indicates the average number of IR instructions in the execution trace during the simulation. The average MAPE on the four GPUs is 17.04% and on each GPU, the optimal kernel prediction can achieve an average MAPE of less than 7%. Overall, the performance estimation framework is robust and accurate.

To observe how close the predicted outcome can get to the actual measured results, the result comparison of Quadro K620 is shown in Figure 5.3 and the remaining GPUs show similar trends. To clearly

Table 5.3: Hardware specification of the test GPUs.

Name	Architecture	SMs/Cores	Clock freq.(MHz)
Quadro K600	Kepler GK107	1/192	876
GeForce GTX645	Kepler GK106	3/384	824
Quadro K620	Maxwell GM107	3/384	1058
GeForce 940M	Maxwell GM108	3/384	1072

Table 5.4: Accuracy and simulation time consumption of testing the performance estimation framework on the Rodinia [2] benchmark.

Bench. name	Kernel name	Number of total design configurations	Average trace length	MAPE (%)				Time per run (ms)
				Quadro K600	GeForce GTX645	Quadro K620	GeForce 940M	
backprop	bpnn_adjust_weights	11450	41	24.24	24.34	22.16	22.12	23.03
	bpnn_layerforward	11450	74	19.38	27.05	21.14	26.55	40.08
bfs	BFS_1	14028	79	11.55	7.969	10.16	20.47	60.09
	BFS_2	14028	7	14.43	20.24	9.879	11.73	10.40
b+tree	findK	42000	100	35.68	31.12	8.941	12.86	72.93
	findRangeK	42000	163	40.63	39.80	13.42	13.42	119.66
cfd	compute_flux	3072	616	15.32	19.81	9.077	14.41	77.55
	compute_step_factor	3072	33	12.83	27.76	43.04	**3.315**	14.01
	initialize_variables	3072	18	11.89	9.149	29.65	7.902	15.38
	memset	12	2	8.085	25.80	6.803	18.83	7.081
	time_step	3072	31	5.191	18.38	18.04	16.20	23.78
hotspot	hotspot	1024	22093	15.36	14.21	**4.325**	9.389	4130.09
kmeans	kmeans_c	40000	2338	12.95	22.99	19.06	20.37	824.60
	kmeans_swap	40000	533	10.23	19.80	15.76	17.74	219.67
lud	lud_internal	8267	108	17.41	23.18	8.278	34.18	45.10
nn	nearestNeighbor	66	9	10.89	26.84	7.090	12.38	6.030
nw	nw_kernel1	19408	1431	9.228	21.88	9.090	25.86	65.27
	nw_kernel2	19408	1431	9.239	24.69	8.551	24.87	63.99
particlefilter	particle_naive	104	52387	19.59	16.99	13.82	11.93	11751.93
pathfinder	dynproc	31025	1469	**3.716**	**6.560**	14.33	9.737	1055.97
Average		15327.9	4148.15	15.39	21.43	14.63	16.71	**931.33**
				17.04				

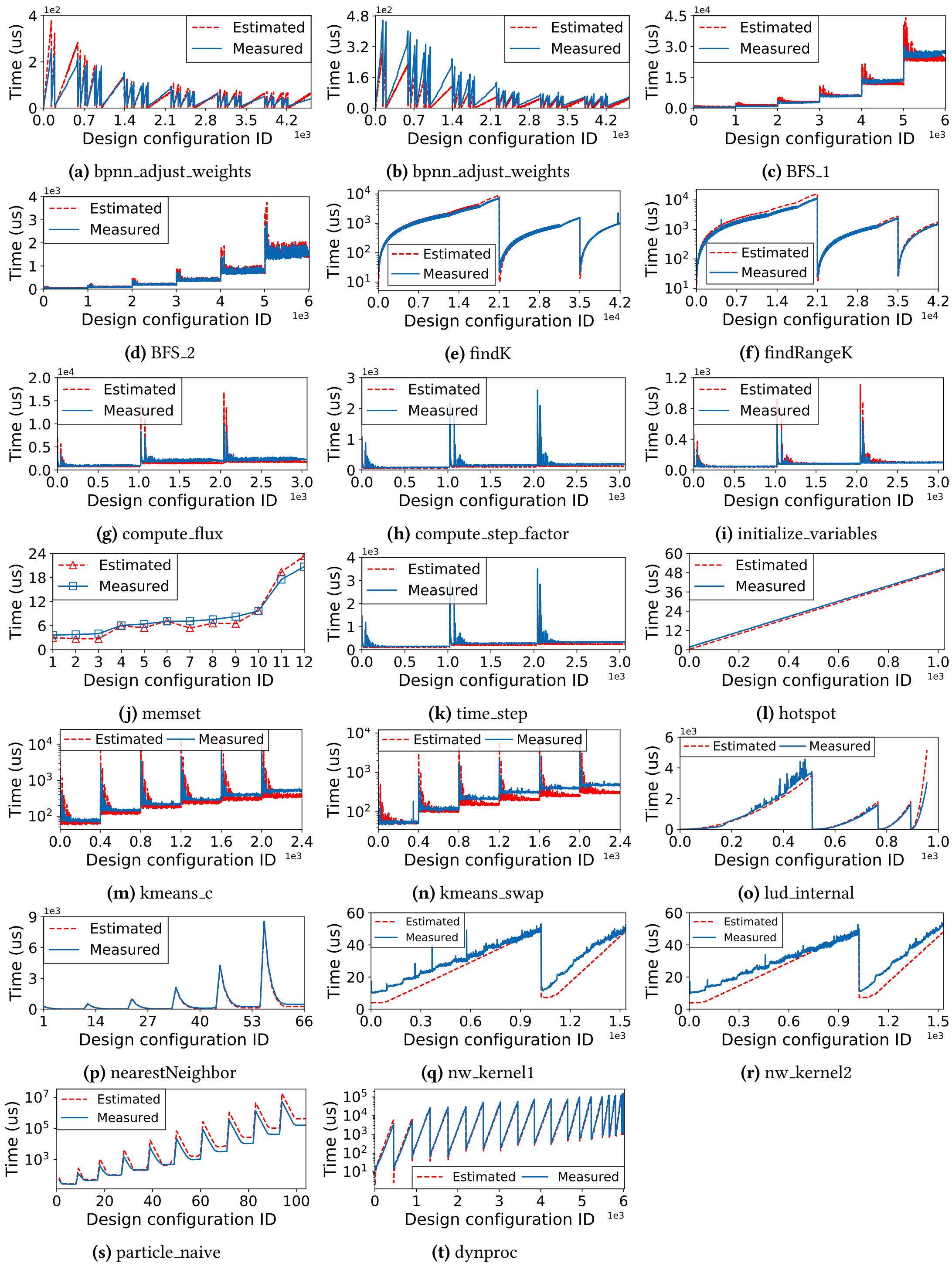


Figure 5.3: Comparison of the estimated and measured execution time of the test kernels (Quadro K620).

show the variation trend of the execution time, for some kernels only partial results in the whole design space are depicted because the curves become too dense if the total number of design configurations is too large. The design configuration ID on the x-axis represents the number of different program input and local work size settings. The execution time results are sorted in an ascending order with the global and local work size as primary and secondary key, respectively. For some kernels, the program input is also taken as the sorting key. Note that the number of total design configurations is very large and therefore is represented in the scientific notation format, except for kernel memset, nearestNeighbor, and particle_naive. The y-axes of kernel findK, findRangeK, kmeans_c, kmeans_swap, particle_naive, and dynproc are represented in logarithmic scale because the execution time shows several orders of magnitude difference in the absolute value. On the whole view, the predicted results accurately follow the variation trend of the actual execution time across the design space. This reveals that the execution trace and the simulation remarkably reflect the runtime behavior of the kernels, which means that the framework can also help users find the optimal execution even for a vast design space.

As observed in Figure 5.3, the MAPE turns out higher when the actual execution time is a few microseconds, particularly for kernel nw_kernel1 and nw_kernel2 (shown in Figure 5.3q and 5.3r). This is because in these cases the kernel overhead dominates the execution time and the predicted time is only a small portion that contributes to the final runtime performance. The kernel overhead includes prerequisite resource allocation, warp scheduling, and kernel launching, etc. The measurement of kernel overhead is infeasible as it is strongly associated with the specific kernel. A possible way is to attach a fixed threshold to the predicted outcome, but again how to set this threshold is pendent.

backprop The MAPEs of this application across four GPUs are quite stable (around 25% in Table 5.4). The main error source of kernel bpnn_adjust_weights is that there are multiple thread-ID-dependent branches and nest branches in the execution flow. The generated execution trace covers as more branches as possible if the estimated run might step into that branch, thus incurring slight overestimation in some cases (shown in Figure 5.3a). For kernel bpnn_layerforward, the underestimation in Figure 5.3b comes from barrier synchronization and kernel overhead.

bfs The prediction of this application is better than *backprop*, due to the much less branches. As seen in Figure 5.3c and 5.3d, kernel BFS_1 suffers from larger overestimation than BFS_2 when the work group size is very small, this is caused by the assumed more cache misses than expected.

b+tree The MAPEs of the kernels in this application are higher on Kepler than Maxwell GPUs. One possible explanation is that the kernels contain structure data and how these data are organized in memory varies across architectures. Moreover, the multiple runtime-dependent nest branches in the main loop body of both kernels cause workload imbalance and also deteriorate the prediction accuracy.

cfd Estimation of kernel initialize_variables shows slightly better accuracy in the variation amplitude (Figure 5.3i), which is the same case as kernel memset (Figure 5.3j). For the remaining three kernels, the error stems from the variant memory access behavior.

hotspot This application contains rather regular workload distribution across work items and the framework performs the prediction very well, as shown in Figure 5.3l. The minor underestimation is caused by the kernel overhead, because the execution time of this kernel is less than 60 us.

kmeans Figure 5.3m and 5.3n show that the predicted outcome of kmeans_swap reveals larger fluctuations than kmeans_c. We attribute this to the continuous global memory data exchange which incurs irregular memory access.

lud & nn These two applications exhibit rather accurate predictions since both kernels have no branch divergence and lud_internal only has a loop with fixed bound.

nw Both nw_kernel1 and nw_kernel2 have several runtime-dependent branches, which makes the estimation more pessimistic. However, Figure 5.3q and 5.3r reveal counter-expectation results. The reason is that kernel overhead also contributes to the MAPE and it is nonnegligible because the total execution time is only a few microseconds. Consequently, kernel overhead compensates for the overestimation and even increases the time consumption for most cases.

particlefilter The predicted execution time shows overestimation for kernel particle_naive in Figure 5.3s, because there exists runtime-dependent branches in the loop, which constructs the unevenly distributed workload across work items. The estimation always assumes the longer execution trace for all the warps and therefore is conservative.

pathfinder Similar to lud and nn, prediction results on this application is rather accurate, as loops are iterated with fixed times and the branches are equally visited by the warps.

To summary, the hybrid framework performs well on the test benchmarks in terms of MAPE. The variation trend of the kernel execution time in the design space is accurately captured by the estimated results. However, the influence of the kernel overhead is significant when the overall execution time is very small, i.e., a few microseconds in the test. In these cases, the dominant factor that contributes to the kernel execution time is not the computation and memory access latency but the interference from the overhead. The proposed framework may incur overestimation for irregular workloads, due to the conservative branch divergence analysis. However, note that *bfs* is also an irregular application and the framework can still give rather good estimation results.

5.6.2.2 Simulation Time Cost

The last column in Table 5.4 presents the average simulation time of predicting the execution time of each kernel run. As shown, on average the framework can give prediction results within 0.931 second, which is much faster than using a fine-grained simulator [87] [88]. The consumed times of estimating kernel hotspot and particle_naive are longer than the remaining kernels due to their extremely long execution traces.

The simulation time cost of the framework is also compared with the widely-used GPGPU-Sim [3] and Table 5.5 gives the results. As shown, the simulation cost of the proposed method is only a few seconds, while GPGPU-Sim takes time in magnitude of minutes. The framework achieves an average speedup of 164.39× over GPGPU-Sim, in terms of the simulation time cost, on the test benchmarks.

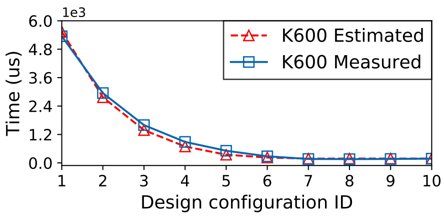
5.7 Case Study with Lane Detection

To demonstrate the effectiveness of the proposed framework, the *p*-LDA presented in Chapter 3 is used as the test case. The algorithm consists of three steps, namely pre-processing, lane detection, and lane tracking. For each image frame, the pre-processing step extracts the information about the lane markings and then passes the processed image to the next step. Depending on whether the estimated positions of the lane markings in previous frame can still be applied to the current frame, the image is processed either reusing the lane detection step to detect the positions or using particle filter to track the previous positions of the lane markings. The aforementioned three steps are mapped to three kernels and Table 5.6 gives the program configuration of the application during the experiment. For 640×480 input videos, the ROI size of KERNEL_PRE is 512×96 and the other two kernels are configured with global work size ranging from 2^{10} to 2^{13}.

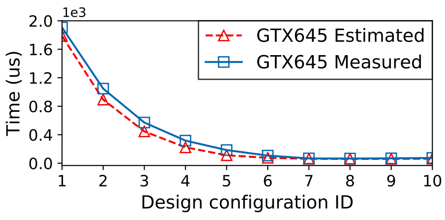
The timing information of these kernels is collected for the whole video and then the averaged

Table 5.5: Comparison of the simulation time costs of GPGPU-Sim [3] and the proposed framework in this chapter.

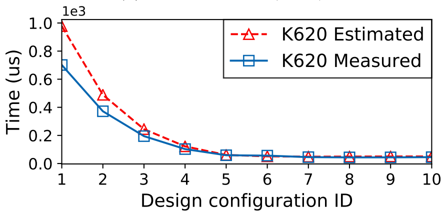
Benchmark	**Simulation time (ms)**		**Speedup**
	GPGPU-Sim	**The proposed framework**	
bfs	4517000	70.49	64080.01
hotspot	200000	4130.09	48.43
lud	168000	45.10	3725.06
nn	3000	6.030	497.51
nw	1673000	129.26	12942.91
pathfinder	280000	1055.97	265.16
Geo. mean	244433.52	148.69	**164.39**



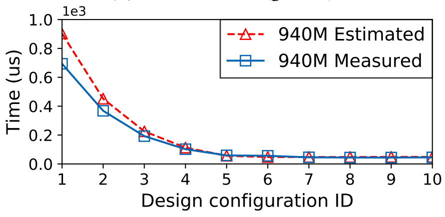
(a) KERNEL_PRE (k600)



(b) KERNEL_PRE (gtx645)

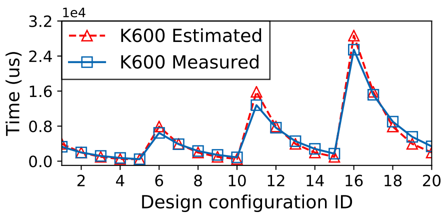


(c) KERNEL_PRE (k620)

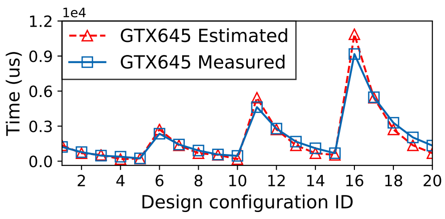


(d) KERNEL_PRE (940m)

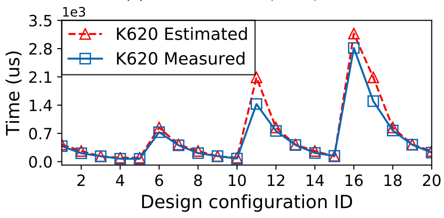
Figure 5.4: Comparison of the estimated and measured results of KERNEL_PRE on different GPUs.



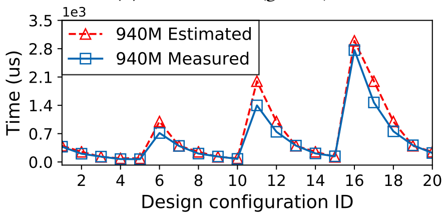
(a) KERNEL_LD (k600)



(b) KERNEL_LD (gtx645)



(c) KERNEL_LD (k620)



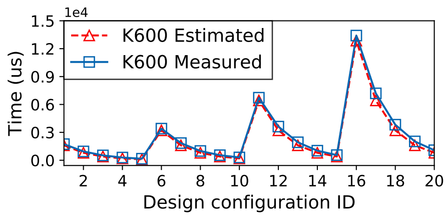
(d) KERNEL_LD (940m)

Figure 5.5: Comparison of the estimated and measured results of KERNEL_LD on different GPUs.

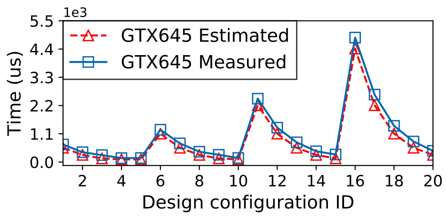
results per frame are calculated. Figure 5.4, 5.5, and 5.6 respectively presents the results of the predicted and the measured time of KERNEL_PRE, KERNEL_LD, and KERNEL_PF. As can be observed, for all the kernels across the different GPUs, the estimations keep the same variation trend with the measured

Table 5.6: Configuration of the lane detection kernels.

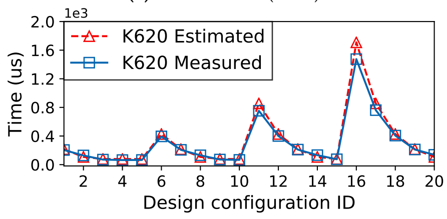
Kernel name	Global size	Local size	No. of designs
KERNEL_PRE	49152	$\{2^1, 2^2, 2^3, \ldots, 2^{10}\}$	10
KERNEL_LD	$2^{10}, 2^{11}, 2^{12}, 2^{13}$	$\{2^1, 2^2, 2^3, 2^4, 2^5\}$	20
KERNEL_PF	$2^{10}, 2^{11}, 2^{12}, 2^{13}$	$\{2^1, 2^2, 2^3, 2^4, 2^5\}$	20



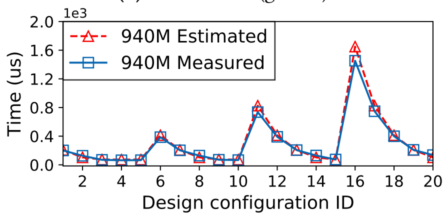
(a) KERNEL_PF (k600)



(b) KERNEL_PF (gtx645)



(c) KERNEL_PF (k620)



(d) KERNEL_PF (940m)

Figure 5.6: Comparison of the estimated and measured results of KERNEL_PF on different GPUs.

results. The average MAPEs of the three kernels are 15.45%, 19.60%, and 17.10%, respectively. The average prediction error for this application is 17.38%.

5.8 Summary

Performance modeling of GPUs is critical for parallel program developers and end-users as the urgent need of hardware accelerators in high performance scientific computing fields. Extant work on GPU performance prediction is either out-of-date for the emerging new generations of architectures, or cumbersome to use due to the substantial effort of tedious calibration and parameter tuning on target platform and target applications. This chapter proposes an analytical and simulation combined framework to predict the performance of parallel workloads on GPUs. The hybrid framework contains a static module that analyzes high-level source code to extract kernel execution trace and a simulation module that dynamically mimics the kernel execution behavior to deduce the final kernel execution time. A software tool chain is further provided to enable the fast performance prediction of target kernels. The framework requires no prior knowledge about the hardware performance counter metrics or pre-executed measurement results. Experimental results reveal that the framework can accurately grasp the variation trend and predict the execution time in the design space with high accuracy and little time cost. Furthermore, it can help users to find the optimal execution in the vast design space.

Following the work in this chapter, the next chapter presents an approach to efficiently prune the program design space of OpenCL kernels running on GPUs.

Chapter 6

Design Space Pruning for OpenCL Kernels on GPUs

Chapter 5 proposed a hybrid framework to predict the performance of running OpenCL kernels on GPUs. Although this method cannot provide absolutely precise estimation results, it is capable of grasping the variation trend of the kernel execution time, which means it can help users to find the optimal execution in the vast design space. Motivated by this, the work in this chapter investigates how to efficiently prune the program design space of an OpenCL kernel so that the optimal design configuration can be quickly located.

6.1 Overview

Over the last decade, GPUs have attracted an overwhelming amount of attention, and utilization of GPUs has dominated state-of-the-art research about image processing, machine learning, high performance computing, and even embedded system design.

One of the critical problems for GPU programmers is how to locate the optimal configuration of a specific parallel application so that it can deliver the best runtime performance. In general, GPU applications take very large workloads as input and these workloads are evenly split and further mapped onto the numerous thread cores that handle substantial data manipulations. Given the huge workload size, the search for a proper sub-workload size that yields optimal performance is nontrivial. For instance, on Nvidia GPUs, the work-group size of an OpenCL [14] kernel in one dimension can vary from 1 to 1024, and the number of dimensions can be up to 3. Such a huge design space poses a challenge for programmers to pick a suitable configuration in a relatively short time period, especially in the early design stage.

To address the aforementioned issue, two main methodologies are adopted in state-of-the-art research. The first technique is called measurement-based performance auto-tuning [78] [123], which samples a portion of the kernel executions selected from the entire design space and then tries to identify the optimal configuration with the aid of miscellaneous search strategies [78] or machine learning algorithms [79]. This approach assumes that neighboring executions are sufficiently related that the search strategy can detect a trend in the variations of the execution time to identify the optimal design. The second approach is program abstraction, which first extracts key features from the static source code [124] or dynamic kernel execution results [125], then defines metrics that are directly or indirectly correlated with the runtime performance. By optimizing these performance metrics, the optimal designs are finally derived. In this method, it is critical to deduce performance metrics that have a significant impact on the kernel runtime performance.

While the aforementioned methods are effective for finding the optimal or near-optimal configurations, there are still some drawbacks. First, performance auto-tuning depends heavily on the execution results of profiling runs on the target GPUs. This can incur substantial efforts, including program deployment, debugging, and profiling time cost, when applying the method to different platforms. Secondly, although performance metrics can reasonably reflect the kernel runtime behavior, providing a clear explanation of how these metrics can influence the kernel execution time is not straightforward. Finally, static performance metrics are often subject to specific programming language or hardware platform, so extra effort is needed when applying them to different architectures.

This chapter proposes a hybrid search framework to find the optimal work-group size for OpenCL kernels running on GPUs, given the constant and known input workload. The framework includes a static analysis module that filters out redundant designs with duplicated execution traces and inferior pipelines, and a dynamical simulation module that produces the estimated optimal design by searching the work-group sizes that yield the minimum predicted kernel execution time. Notably, the proposed framework does not require any program runs to find the optimal or near-optimal designs. The effectiveness of the framework is evaluated with a set of OpenCL kernels from the Rodinia [2] benchmark, on two Nvidia GPUs across two recent architectures (Maxwell and Pascal).

The remainder of this chapter is organized as follows: Section 6.2 is related work and Section 6.3 gives problem formulation. Section 6.4 presents the details of the hybrid search framework. Section 6.5 gives evaluation results and Section 6.6 concludes the work in this chapter.

6.2 Related Work

The literature on performance auto-tuning and design space exploration for parallel applications has grown over the last decade. At first, performance auto-tuning research mainly focused on commonly

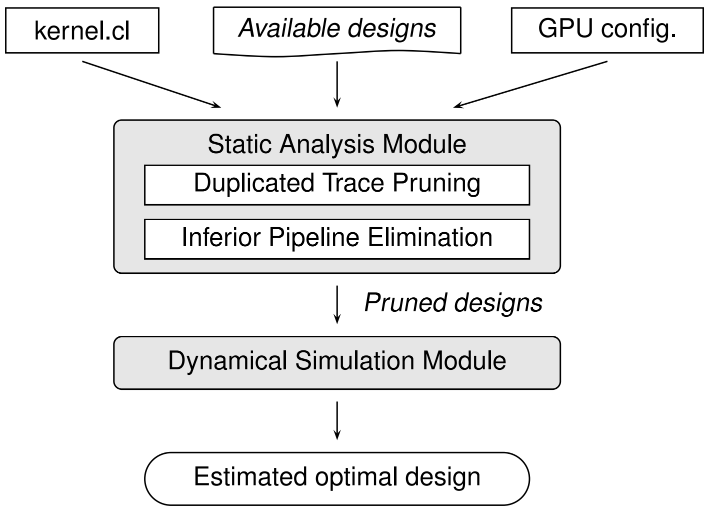


Figure 6.1: Overview of the hybrid search framework.

used but computationally demanding algorithms, such as matrix multiplication [126] [127], stencil computation [128], fast Fourier transform [74], etc. The popular use of GPUs for general-purpose scientific computing has driven the emergence of auto-tuners used for generic algorithms. Measurement-based auto-tuners prune the program design space with sampling executions and then identify the optimal configuration using various search strategies [78] [79] [123]. Other studies have proposed performance metrics that are used as object function during optimization to derive the parameters related to the optimal executions [124] [125]. There are also studies that combine measurement outcomes and extracted program metrics to identify the optimal parameter settings [129].

Apart from the aforementioned studies, other researchers have proposed methods to optimize the kernel in the pre-execution stage, via code generation [130] and compiler optimization techniques [131]. These methods are orthogonal to the work in this chapter since this work assumes that the kernel source code is given. Therefore, these techniques can be applied to pre-optimize the target program before applying the framework presented in this chapter.

In this work, the kernel is analyzed using the IR instructions generated by the LLVM compiler [86]. The method applies static execution trace and pipeline analysis results to prune the program design space and then identifies the optimal or near-optimal work-group size via dynamical execution trace simulations. Therefore, the proposed framework does not require any program runs. Unlike comparable state-of-the-art methods, the framework presented in this chapter neither requires tedious profiling to obtain representative execution results, nor does it adopt performance metrics extracted from high-level kernel source code.

6.3 Problem Statement

In OpenCL, the input workload is represented as a number of work items and this total number of work items (or the global size, GS) can be divided into several work groups. Each group of work items with a work-group size WG, is a set of work instances mapped to a single SM on GPUs. The present work solves the following problem: Given a kernel with NDRange configuration of global size GS and a set of possible work-group sizes $WG = \{wg_0, wg_1, \cdots, wg_K\}$, where K is the number of possible designs, find the appropriate work-group size wg_{opt} so that the kernel execution time is minimal.

6.4 Framework

6.4.1 Framework Overview

For Nvidia GPUs, each OpenCL work item instance is mapped to the thread cores on the SM and a group of 32 threads, which is also called one warp, are bound together to execute the instance in lock-step manner. All threads in one warp consume the same instructions, while the consumed instructions in different warps may vary. Due to resource and architectural limit, given a specific work-group size (wg), the number of active warps (N_{aw}^{wg}) for a kernel can be determined at compile time. In general, a larger value of N_{aw}^{wg} delivers better performance, though the best case execution does not always reveal the largest value of N_{aw}^{wg}.

Figure 6.1 gives an overview of the proposed hybrid search framework. Given the kernel source code (kernel.cl) and target GPU specifications (GPU config.), for a fixed and known input workload size, the possible design configurations can be obtained via an exhaustive traversal of the valid work-group sizes. The static analysis module contains two main steps that analyze program runtime behavior based on the execution traces that result from choosing different work-group sizes. The work-group sizes that yield the same execution traces and the same in-memory layouts for the input kernel arguments, are grouped in one batch, and only the configuration with the minimum number of rounds of active warps in each batch is chosen as a representative design. The representative designs extracted from different batches are then pruned further by filtering out inferior cases in which the lower bound of the best-case execution time is larger than the upper bound of the worst-case execution time yielded by another representative design. The pruned designs are then fed to the dynamical simulation module, in which an exhaustive search is performed to estimate the kernel execution time. Finally, the configuration with the shortest predicted execution time is chosen as the estimated optimal design.

6.4.2 Static Analysis Module

An execution trace is defined as the sequence of IR instructions that one warp would consume. The static analysis module uses the loop-based bidirectional branch search algorithm from the work in Chapter 5 to obtain the execution trace of each warp. Suppose the work-group size is set as wg and the set of execution traces for a round of active warps is defined as $\mathbf{T}$, the kernel execution time is estimated as follows:

$$t_{kernel} = \underbrace{t^{spec(\text{LAT},\text{cacheMissInfo})}_{pipeline(N^{wg}_{aw},\mathbf{T})}}_{\text{Sub-item ❶}} \times \underbrace{\frac{\lceil \frac{GS}{wg \times N_{SM}} \rceil \times \lceil \frac{wg}{N^{wi}_{warp}} \rceil}{N^{wg}_{aw}}}_{\text{Sub-item ❷}}, \tag{6.1}$$

where LAT is the latency of the arithmetic and memory access operations on the target GPU, cacheMissInfo is the cache miss information in the execution traces $\mathbf{T}$, N_{SM} is the total number of SMs on the target GPU, N^{wi}_{warp} is the number of work items per warp (fixed to 32 for current Nvidia GPUs), and Sub-item ❶ in Equation (6.1) is the time needed for a round of active warps to complete the execution traces $\mathbf{T}$.

6.4.2.1 Duplicated Trace Pruning

In Equation (6.1), the latency of the arithmetic and memory access operations (LAT), the number of work items per warp (N^{wi}_{warp}), and the total number of SMs (N_{SM}) are hardware-dependent and therefore remain constant when varying the work-group size. The cache miss information in the execution traces (cacheMissInfo) is determined by the in-memory layout of the input kernel arguments (denoted as argMemLayout) and the hardware specifications of the memory hierarchy (denoted as gpuMemConfig). Consequently, when varying the work-group size wg, the kernel execution time is only determined by argMemLayout, N^{wg}_{aw}, and $\mathbf{T}$, since the remaining parameters are hardware-dependent and constant.

First of all, the details of argMemLayout and $\mathbf{T}$ for each work-group size are compared and the work-group sizes with the same contents of argMemLayout and $\mathbf{T}$ are combined into one batch. Here the same execution traces $\mathbf{T}$ means that both the execution trace for each warp and the number of active warps are identical. Then information of argMemLayout is extracted from the invoked argument list of the kernel function, and the memory size and data type of each argument is obtained from the host source code. It is observed that for most of the kernels, argMemLayout remains constant when varying the work-group size, as it is only affected by the input workload. For the cases in which the memory buffer sizes of kernel arguments are determined by the work-group size, these work-group sizes are assigned to different batches.

Algorithm 8: Inferior pipeline elimination

```
Input: Representative design set $\mathbb{WG}$, Global size $GS$, Total number of SMs $N_{SM}$
Output: Pruned design set WG
1  WG ← ∅
2  foreach $wg \in \mathbb{WG}$ do
3      isPruned ← FALSE
4      $L_B^{wg}$ ← calcPipelineLowerBoundLatency($wg$)
5      $lat_B^{wg} \leftarrow \lceil \frac{GS}{wg \times N_{SM}} \rceil \times \lceil \frac{wg}{N_{warp}^{wi}} \rceil \times \frac{L_B^{wg}}{N_{aw}^{wg}}$
6      foreach $wg' \in \mathbb{WG}$ do
7          $L_W^{wg'}$ ← calcPipelineUpperBoundLatency($wg'$)
8          $lat_W^{wg'} \leftarrow \lceil \frac{GS}{wg' \times N_{SM}} \rceil \times \lceil \frac{wg'}{N_{warp}^{wi}} \rceil \times \frac{L_W^{wg'}}{N_{aw}^{wg'}}$
9          if $lat_B^{wg} < lat_W^{wg'}$ then
10             isPruned ← TRUE
11             break
12     if isPruned == FALSE then
13         $wg$ ← WG
14 return WG
```

The work-group sizes in the same batch show the same execution behavior for a round of active warps, so the values of Sub-item ❶ in Equation (6.1) are equivalent. Sub-item ❷ in Equation (6.1) indicates the number of rounds of active warps needed for the entire kernel run. Therefore, the work-group size with the lowest value of Sub-item ❷ in a batch reveals the minimal kernel execution time, and this design is chosen as the representative design from this batch. After all representative designs from all batches are collected, they are fed to the following pruning module.

6.4.2.2 Inferior Pipeline Elimination

The *duplicated trace pruning* step filters out the work-group sizes that showcase the same execution behavior within a round of active warps. Afterwards, the execution times among different execution traces are analyzed, and design configurations with inferior pipelines are eliminated. Algorithm 8 gives the detail of this procedure. For each work-group size wg, the lower bound of the best-case pipeline execution latency (L_B^{wg}) is deduced, and thereby the lower bound of the best-case execution time (lat_B^{wg}) is calculated. If this lower bound is still larger than the upper bound of the worst-case execution time yielded by another design ($lat_W^{wg'}$) within the representative design set, the selected design results in an inferior pipeline and is removed from the search space. As shown in Line 5 and 8 in Algorithm 8, lat_B^{wg} and $lat_W^{wg'}$ are calculated in a similar way, i.e., by replacing Sub-item ❶ in Equation (6.1) with L_B^{wg} and $L_W^{wg'}$, respectively. The details of how L_B^{wg} and L_W^{wg} are deduced (Line 4 and 7 in Algorithm 8) are illustrated as follows.

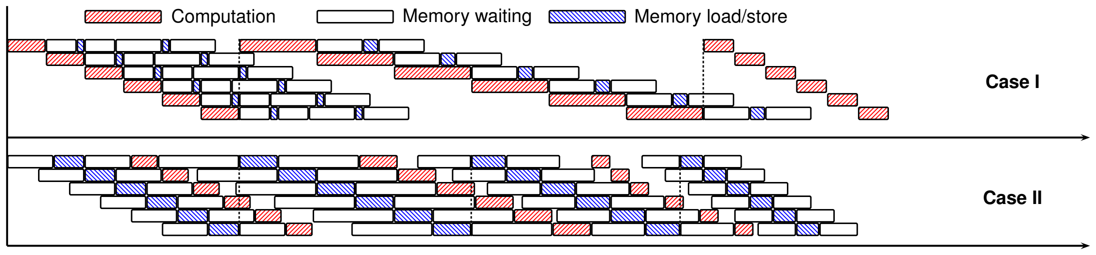


Figure 6.2: Best case of the pipeline execution.

Preliminaries. For Nvidia GPUs, the resources shared by a warp pipeline are the Floating Point Unit (FPU), the Special Function Unit (SFU), and various memory load/store units. Since the OpenCL kernel execution is modeled at the IR level, the types of memory access are differentiated as memory access from Local Memory (LM), Constant Memory (CM), and Global Memory (GM). Private memory access is performed on-chip and is not mutually exclusive for the work items in one work group. Moreover, the cache accesses of CM and GM load/store are considered as memory accesses from the Constant Cache (CC) and the Global Cache (GC). For each memory access, i.e., the load/store IR instruction in the execution trace, the memory referencing addresses are extracted and thereupon the number of memory transactions and cache hit/miss accesses that a warp needs to perform when consuming the execution trace [132] are analyzed. The memory access latencies are obtained via micro-benchmarking and the latency results characterize the average time from the memory instruction issue stage to the final data acquisition stage. The memory delay is denoted as the time during which the data is read/written by the hardware control circuit. The memory latency is modeled as three parts: pre-waiting latency, memory delay (memDelay) and post-waiting latency, of which the sum is the micro-benchmarked memory latency (memLatency).

Given an arbitrary execution trace running on a warp pipeline, the mutually exclusive resources are the computation components (FPU, SFU) and the memory components (LM, CM, GM, CC, GC). Suppose the execution trace is defined as $\mathbf{T} = \{\tau_0, \tau_1, \cdots, \tau_m\}$, where τ_i is the execution trace for N_i warps and $\sum_{i=0}^{m} N_i = N_{aw}^{wg}$. Let's consider the best and worst cases of the kernel execution.

Best-case analysis. In the best-case execution, computation and memory access instructions are consumed in an interleaved manner, so that the execution times are overlapped as much as possible. For instance, Figure 6.2 presents two extreme cases in which the memory access latency in Case I and the computation latency in Case II overlap perfectly. In these two cases, the overall kernel execution time is equal to either the computation latency or the memory access latency. Therefore, L_B^{wg} is

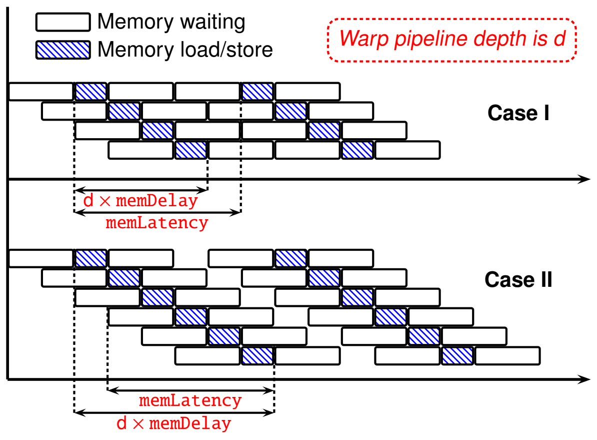


Figure 6.3: Sample cases of the memory component pipeline.

calculated as

$$L_B^{wg} = max(L_{fpu}^{N_{aw}^{wg}}, L_{sfu}^{N_{aw}^{wg}}, Lb_{lm}^{N_{aw}^{wg}}, Lb_{cm}^{N_{aw}^{wg}}, Lb_{gm}^{N_{aw}^{wg}}, Lb_{cc}^{N_{aw}^{wg}}, Lb_{gc}^{N_{aw}^{wg}}), \tag{6.2}$$

where the latencies of the computation components are calculated as follows:

$$L_{comp}^{N_{aw}^{wg}} = \sum_{i=0}^{m} L_{comp}^{N_i} \tag{6.3}$$

$$L_{comp}^{N_i} = \sum^{opType(\tau_i)} num(OP, \tau_i) \times lat(OP), \tag{6.4}$$

where $comp = \{fpu, sfu\}$, $opType(\tau_i)$ is the number of instruction types in the execution trace τ_i, $num(OP, \tau_i)$ is the number of OP instructions in the execution trace τ_i, and $lat(OP)$ is the micro-benchmarked latency of OP instruction. Here OP refers to the basic arithmetic instructions for the FPU, and the transcendental instructions for the SFU, respectively.

With regard to the latencies of the memory components, because memDelay, instead of mem-Latency, is the main factor that stalls the memory component pipeline, the overall latency is directly related to the pipeline depth, i.e., the number of active warps running on the pipeline. Consider the two cases in Figure 6.3, where the pipeline depth d influences the latency of the memory components. In Case I, the pipeline is not deep enough, so the memory load/store delay time cannot hide the memory waiting time ($d \times memDelay \leq memLatency$).

Suppose each warp has κ memory components, the pipeline latency in this case is calculated as

$$L_I = \kappa \times memLatency + (d-1) \times memDelay. \tag{6.5}$$

However, as the pipeline becomes deeper, as shown in Case II, the memory delay becomes the dominant factor in the pipeline latency and the memory waiting time is totally overlapped with it ($d \times memDelay > memLatency$). In this case, the pipeline latency is

$$L_{II} = \kappa \times d \times memDelay + memLatency - memDelay. \tag{6.6}$$

From Equation (6.5) and (6.6), the latency of each type of the memory components is

$$L_{comp}^{N_i} = \kappa^{N_i} \times l_{max}^{N_i} + l_{min}^{N_i} - memDelay_{comp} \tag{6.7}$$

$$l_{max}^{N_i} = max(N_i \times memDelay_{comp}, memLatency_{comp}) \tag{6.8}$$

$$l_{min}^{N_i} = min(N_i \times memDelay_{comp}, memLatency_{comp}), \tag{6.9}$$

where $comp = \{lm, cm, gm, cc, gc\}$, $i = 0, 1, \cdots, m$, and κ^{N_i} is the number of memory components in the execution trace τ_i. The best-case memory component latency is calculated as

$$Lb_{comp}^{N_{aw}^{wg}} = max(L_{comp}^{N_0}, L_{comp}^{N_1}, \cdots, L_{comp}^{N_m}), \tag{6.10}$$

where $comp = \{lm, cm, gm, cc, gc\}$.

Worst-case analysis. In the worst-case execution, the IR instructions in the execution traces construct a permutation that the latencies of the computation and memory components barely overlap. In this work, the upper bound of the pipeline latency is derived as an extreme case in which at any time only one IR instruction in the execution trace is consumed as if there is always synchronization between each IR instruction. Consequently, L_W^{wg} is deduced as

$$L_W^{wg} = L_{fpu}^{N_{aw}^{wg}} + L_{sfu}^{N_{aw}^{wg}} + Lw_{lm}^{N_{aw}^{wg}} + Lw_{cm}^{N_{aw}^{wg}} + Lw_{gm}^{N_{aw}^{wg}} + Lw_{cc}^{N_{aw}^{wg}} + Lw_{gc}^{N_{aw}^{wg}} \tag{6.11}$$

$$Lw_{comp}^{N_{aw}^{wg}} = \sum_{i=0}^{m} L_{comp}^{N_i}, comp = \{lm, cm, gm, cc, gc\}, \tag{6.12}$$

where $L_{fpu}^{N_{aw}^{wg}}$ and $L_{sfu}^{N_{aw}^{wg}}$ are calculated from Equation (6.3), and $L_{comp}^{N_i}$ is calculated from Equation (6.7).

6.4.3 Dynamical Simulation Module

The pruned designs output by the static analysis module are a part of the possible work-group sizes in the entire program design space. Subsequently, the dynamical simulation method from Chapter 5 is used to predict the kernel execution time needed for each design. The simulation performs a dummy execution of each IR instruction in the execution trace set **T** for a round of active warps and outputs the total time of consuming the execution traces, given LAT and cacheMissInfo. The kernel execution time is estimated from Equation (6.1). After the exhaustive simulation search, the work-group size that produces the minimum predicted execution time is chosen as the optimal work-group size wg_{opt}.

6.4.4 Discussion

The proposed hybrid search framework does not require any program runs of the target kernel on the target GPU. Although the dynamical simulation module performs an exhaustive simulation search of the pruned designs from the static analysis module, this time cost is rather small compared with the exhaustive simulation search of all the possible designs from the entire program design space, because the two-stage static pruning step filters out most of the redundant design configurations. The bound analysis of the best- and worst-case executions of the IR instruction pipeline manifests the extreme cases of the pipeline latency overlapping, given arbitrary execution traces and arbitrary pipeline depth. Although the actual execution of the pipeline can always be determined once the execution traces are fixed, this information can only be obtained through dynamical simulations. Hence, the bound estimation of the pipeline latency is intended to provide an effective filter that reduces the number of simulated designs.

6.5 Evaluation Results

6.5.1 Evaluation Setup

Two Nvidia GPUs are used to evaluate the hybrid search framework and Table 6.1 gives the detailed hardware specifications. The proposed framework is tested with a set of OpenCL kernels from the Rodinia [2] benchmark. The detailed configuration is given in Table 6.2. The evaluation uses the default input workloads from the benchmarks and excludes the kernels whose work-group size cannot be varied when the global size is fixed. For backprop, the number of possible work-group sizes is very small because this application allocates shared memory according to the selected work-group size.

Table 6.1: Hardware specifications of the test GPUs.

Device name	Quadro K620	Titan XP
Architecture	Maxwell GM107	Pascal GP102
Compute capacity	5.0	6.1
Number of SMs	3	30
Number of cores per SM	192	128
Clock frequency (MHz)	1058	1405
Number of cores per warp	32	32
Maximum active work groups per SM	32	32
Maximum active warps per SM	64	64

Therefore, the shared memory quickly reaches the capacity limit when the work-group size is set to a large value. For kmeans, the number of features may vary even if the global size is fixed to (10000, 1, 1), therefore different cases are tested, in which the number of features was set to 16, 32, and 64, respectively. The simulations are performed on a desktop with an Intel® Core^{TM} i7-3770 CPU.

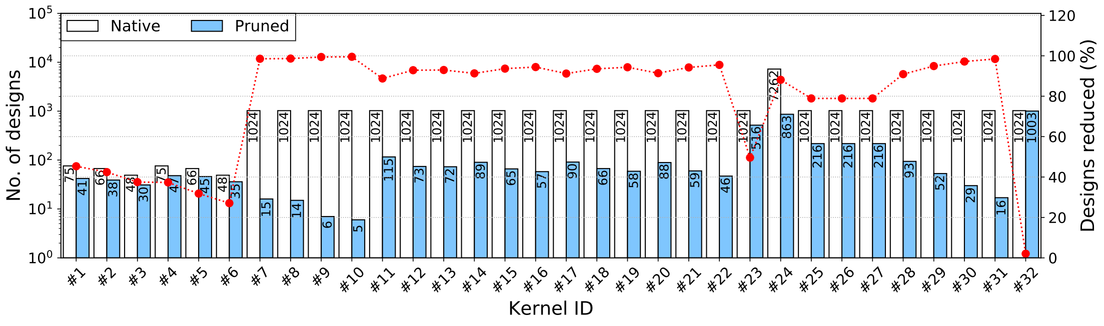
6.5.2 Results

6.5.2.1 Design Space Reduction

Figure 6.4 shows the design space reduction results on the test GPUs. The x-axis represents the kernel IDs (Column 2) in Table 6.2. The bar results show the number of designs for each kernel. Specifi-

Table 6.2: Design configurations of the test OpenCL kernels.

Benchmark	Kernel ID	Kernel name	Global work size	No. of total designs	
				Quadro K620	Titan XP
backprop	#1	adjustWeight_1	(100, 1600, 1)	75	84
	#2	adjustWeight_2	(100, 3200, 1)	66	78
	#3	adjustWeight_3	(100, 6400, 1)	48	75
	#4	layerForward_1	(100, 1600, 1)	75	84
	#5	layerForward_2	(100, 3200, 1)	66	78
	#6	layerForward_3	(100, 6400, 1)	48	75
bfs	#7	bfs1_1	(1048576, 1, 1)	1024	1024
	#8	bfs1_2	(2097152, 1, 1)	1024	1024
	#9	bfs2_1	(1048576, 1, 1)	1024	1024
	#10	bfs2_2	(2097152, 1, 1)	1024	1024
cfd	#11	computeFlux_1	(97152, 1, 1)	1024	1024
	#12	computeFlux_2	(193536, 1, 1)	1024	1024
	#13	computeFlux_3	(232704, 1, 1)	1024	1024
	#14	computeStepFactor_1	(97152, 1, 1)	1024	1024
	#15	computeStepFactor_2	(193536, 1, 1)	1024	1024
	#16	computeStepFactor_3	(232704, 1, 1)	1024	1024
	#17	initVariable_1	(97152, 1, 1)	1024	1024
	#18	initVariable_2	(193536, 1, 1)	1024	1024
	#19	initVariable_3	(232704, 1, 1)	1024	1024
	#20	timeStep_1	(97152, 1, 1)	1024	1024
	#21	timeStep_2	(193536, 1, 1)	1024	1024
	#22	timeStep_3	(232704, 1, 1)	1024	1024
gaussian	#23	fan1	(4096, 1, 1)	1024	1024
	#24	fan2	(4096, 4096, 1)	7262	7262
kmeans	#25	kmeansSwap_feat16	(10000, 1, 1)	1024	1024
	#26	kmeansSwap_feat32	(10000, 1, 1)	1024	1024
	#27	kmeansSwap_feat64	(10000, 1, 1)	1024	1024
nn	#28	nearestNeighbor_1	(131072, 1, 1)	1024	1024
	#29	nearestNeighbor_2	(262144, 1, 1)	1024	1024
	#30	nearestNeighbor_3	(524288, 1, 1)	1024	1024
	#31	nearestNeighbor_4	(1048576, 1, 1)	1024	1024
particleFilter	#32	particleNaive	(2048, 1, 1)	1024	1024


(a) Quadro K620

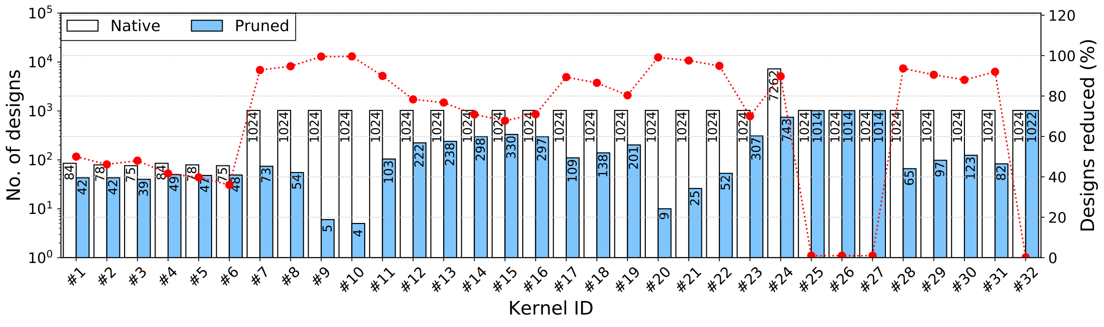

(b) Titan XP

Figure 6.4: Design space reduction results of the hybrid search framework on the test GPUs. The bar results indicate the number of design configurations of the native and the pruned search. The red point values indicate the percentage of the pruned designs from the native search space.

cally, the *Native* bar represents the number of total designs and the *Pruned* bar represents the number of pruned designs yielded by the static analysis module. The red points give the design reduction percentage for each test kernel. Note that the number of total designs varies from a few to several thousand and therefore is represented in logarithmic scale.

As can be seen, the proposed approach is quite effective for most of the test kernels. Specifically, the proposed framework can prune the number of designs by more than 90%, for bfs, nn, and the kernel timeStep. On average, the static analysis module reduces the program design space by 77.81% and 68.06%, respectively, on Quadro K620 and Titan XP GPUs. For both GPUs, the limiting factor on the percentage of the program design space reduction is backprop and particleFilter. As for backprop, the number of total designs is rather small, so the number of redundant configurations are greatly outnumbered by the remaining kernels. With regard to particleFilter, the kernel execution times across all designs are quite stable, so the static analysis module is not able to significantly prune the design configurations, resulting in a still very large pruned design space. The case is the same with kmeans running on the Titan XP GPU.

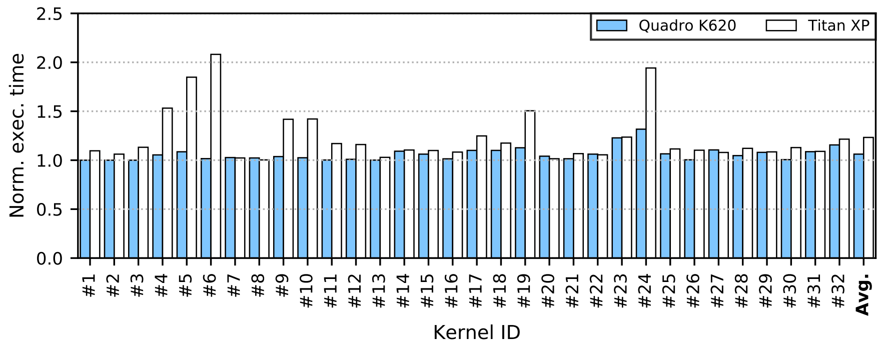


Figure 6.5: Normalized execution time of the selected design on the test GPUs.

6.5.2.2 Search Quality

Figure 6.5 presents the performance results on the test GPUs. The x-axis represents the kernel IDs (Column 2) in Table 6.2, and the y-axis represents the ratio of the runtime performance of the test kernels with the selected work-group size to that with the truly optimal design. On average, the performance with the selected designs is 1.062 times and 1.233 times slower than that with the truly optimal configurations, respectively for the Quadro K620 and Titan XP GPUs. For the kernel adjustWeight, the hybrid search framework manages to find the truly optimal work-group size configuration for the Quadro K620 GPU.

The worst results come from the kernel layerForward and fan2 running on the Titan XP GPU. This is due to the reason that although the dynamical simulation can give a fairly good estimation of the kernel execution time on the whole, it fails to accurately predict some extreme cases that might showcase the minimal estimated execution time. To demonstrate the effectiveness of the static analysis module, for the kernel layerForward and fan2 running on the Titan XP GPU, the test kernels with pruned designs are also exhaustively executed and the work-group size with the minimal execution time is chosen as a candidate work-group size configuration. Results show that the candidate configurations always match the truly optimal work-group size.

6.5.2.3 Search Cost

The search cost is measured by recording the time needed to obtain the optimal configuration for the test kernels when using the exhaustive simulation search and the hybrid search, respectively. The results are shown in Figure 6.6. On average, the proposed hybrid search framework takes only 24.70% and 34.57% of the time required by the exhaustive simulation search to find the optimal work-group size, for the Quadro K620 and Titan XP GPUs, respectively. Similar to the percentage of the program

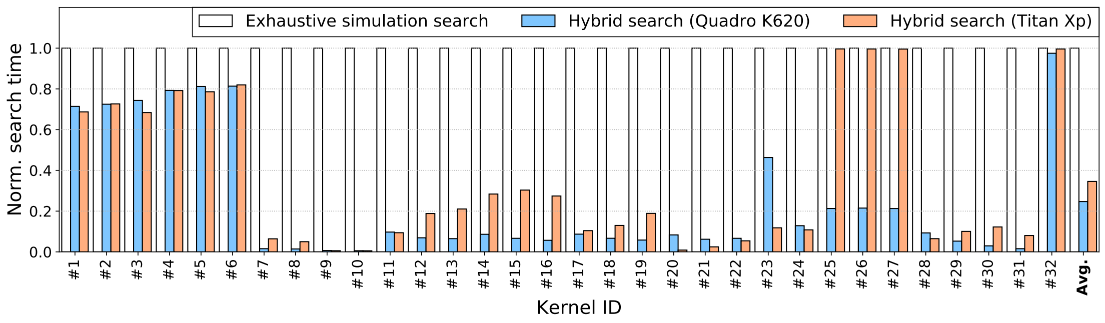


Figure 6.6: Normalized time costs of the exhaustive simulation search and the hybrid search on the test GPUs.

design space reduction, the limiting factor is the large search time needed for backprop, particleFilter, and kmeans on the Titan XP GPU.

6.6 Summary

Designing an efficient design-space exploration of parallel workloads meant for GPUs is non-trivial due to the large numbers of possible design configurations. The work in this chapter addresses the selection of work-group size for OpenCL kernels and proposes a hybrid-search framework for pruning the design space of possible work-group sizes for OpenCL kernels running on GPUs. The proposed space-search framework includes two modules. First, a static analysis module applies two stages of design-space pruning by filtering out design configurations that have duplicated execution traces and inferior pipelines. Second, a dynamical simulation module estimates the optimal design configuration by exhaustively searching the work-group sizes that yield the minimum predicted kernel execution time. The proposed framework does not require any program runs to find optimal or near-optimal work-group size configurations. Experiments show that the framework can significantly reduce the program design space and generate the estimated work-group size which can deliver runtime performance comparable to that with the truly optimal configuration.

Chapter 7

Conclusion and Future Work

This chapter makes a brief summary of the main contributions of the work presented in this thesis. Meanwhile, it will also give discussions about the main results of the thesis and then come up with the prospect of further research direction when using heterogeneous computing systems for the development of ADAS applications.

7.1 Main Contributions

The primary contributions of the work in this thesis lie in the investigation of applying the heterogeneous computing techniques for the development of parallel ADAS applications. The work is done in a systematic way, i.e., both the hardware architecture design and the software application implementations are carefully considered, before the detailed studies about performance analysis, optimization, estimation, and program design space exploration are conducted. Specifically, the main points illustrated in this thesis are listed as follows:

1. **Hardware architecture design.** To cater to the requirements of high-performance-guarantee and low-energy-consumption for future ADAS applications and to tackle the problems of tedious HW/SW maintenance and low scalability existed in conventional automotive ECUs, Chapter 2 proposes to adopt the modular design philosophy for the implementation of a novel ECU architecture called h^{2}ECU. The platform integrates multiple hardware accelerators so as to provide sufficient computing power, while traditional dedicated functionalities can be seamlessly executed on the fixed host module. The high performance, flexibility, and scalability of this architecture is demonstrated by the deployment on a modified real-life vehicle and evaluations with two customized ADAS applications.

2. **Software application implementations.** Based on the heterogeneous context, two different

approaches to realize the lane detection application are presented in Chapter 3. The comprehensive study of how to abstract, parallelize, and optimize the computation tasks using the commodity hardware accelerators provide a paradigm for deploying generic ADAS applications in the heterogeneous systems. The performance and energy efficiency of executing the applications with heterogeneous configurations are also demonstrated by a series of experiments on real-world road scenarios.

3. **Performance optimization procedure.** With the heterogeneous platform and the case applications in hand, Chapter 4 gives further study towards the aspect of the performance optimization. In this chapter, a detailed optimization procedure is provided for those ADAS applications which are parallelized for single-accelerator scenario, but not fully optimized for multi-accelerator heterogeneous configurations. The optimization procedure takes into consideration the perspectives from the host, intra-accelerator, inter-accelerator, accelerator-specific and algorithm-oriented executions and thus the performance enhancement is presented in a step-by-step manner. Moreover, it is applicable to generic parallel applications programmed with OpenCL, and other heterogeneous and reconfigurable computing systems.

4. **Hybrid performance estimation and design space pruning framework.** In Chapter 5 and 6, detailed studies about the performance estimation and design space pruning of OpenCL kernels on the representative GPU platforms are given. The work in these two chapters concentrates on a hybrid framework that combines static source code analysis and dynamical execution trace simulation approaches to model and monitor the runtime behavior of the parallel OpenCL kernels. The robustness, accuracy, and time cost of the hybrid framework are afterwards evaluated and analyzed.

7.2 Discussion

Heterogeneous computing is an emerging technique for the development and deployment of scientific computing applications in the scenarios where high performance needs to be guaranteed, while other expenses such as the energy, power, or thermal consumptions are expected to be minimized. The work in this thesis applies the heterogeneous computing architecture to the automotive domain and depicts relatively span-new and preliminary studies of using heterogeneous computing for the development of ADAS applications.

The work in Chapter 2 and 3 shows the possibility and feasibility of running ADAS applications on a customized heterogeneous platform. It is seen that within the heterogeneous context, the real-time performance of the ADAS applications can be effortlessly achieved, meanwhile the energy cost can also be regulated. This type of reconfigurable computing exhibits considerably high flexibility and scalability for the development of next-generation ADAS applications and the design of future automated driving module.

The work in Chapter 4 looks into the various aspects of the heterogeneous systems and the proposed optimization procedure gives insights and guidance for potential program developers who wish to maximize the performance of their projects by virtue of leveraging various hardware computation resources. The FPGA-GPU heterogeneous system studied in this chapter serves as a typical template for other reconfigurable computing systems, and the associated performance optimization methods can be expanded to other scenarios as well.

The work in Chapter 5 and 6 demonstrated that, despite the diversity of applications and architectures, the performance results are somehow predictable within tolerable error range, and the inherent vast program design space of the parallel applications can be efficiently pruned. The GPU platform studied in this chapter throws light upon the adoption of hybrid approach for the performance modeling and analysis of parallel applications on commodity hardware.

7.3 Future Work

The contributions in this thesis present initial research and provide partial solutions for the development and deployment of ADAS applications in heterogeneous computing systems. Possible future work regarding to this topic are listed in the following:

- *Hardware design.* The performance demand is considered as the primary goal for the embedded systems, especially for ADAS. The proposed h^{2}ECU is a prototype platform for current research use. However, as a portable device, power consumption and the associated thermal management are also important topics that needs to be considered. For the real automotive products, how to address these issues simultaneously is a critical topic which deserves further study.
- *Application implementation.* The work in this thesis chooses LDA as the case application as it represents a category of applications which require high computing power and thus deserve well-performed parallelization. For ADAS, there exists a wide range of other applications using miscellaneous sensors and how to involve them into the heterogeneous context is also a very promising future work.
- *Analysis and optimization of other indices.* Performance analysis is the main focus of the work in this thesis and the energy efficiency is also investigated for the parallelized ADAS applications.

A potential research direction in the next step would be how to optimize other indices, such as energy, power, or both, for the workload processed in the heterogeneous systems.

- *Involvement of other accelerators.* The aim of heterogeneous computing is to foster strengths and circumvent weaknesses of various hardware accelerators to maximize the task processing efficiency. Currently, GPUs and FPGAs are two types of mainstream devices that dominate the scientific computing realm. How to model and analyze the performance of the overall system which consists of both GPUs and FPGAs, or even with other commodity hardware components, is still a rather new topic for future study.

List of Publications

[1] Xiebing Wang, Linlin Liu, Kai Huang, and Alois Knoll. **Exploring FPGA-GPU Heterogeneous Architecture for ADAS: Towards Performance and Energy**, In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 33–48. Springer, 2017.

[2] Xiebing Wang, Mingyue Cui, Kai Huang, Alois Knoll, and Long Chen. **Improving the performance of ADAS application in heterogeneous context: A case of lane detection**, In *IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2017.

[3] Xiebing Wang, Christopher Kiwus, Canhao Wu, Biao Hu, Kai Huang, and Alois Knoll. **Implementing and Parallelizing Real-time Lane Detection on Heterogeneous Platforms**, In *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018.

[4] Xiebing Wang, Kai Huang, Alois Knoll, and Xuehai Qian. **A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation**, In *IEEE 25th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 506–518. IEEE, 2019.

[5] Xiebing Wang, Kai Huang, Long Chen, and Alois Knoll. **h^{2}ECU: A High-Performance and Heterogeneous Electronic Control Unit for Automated Driving**, *IEEE Micro*, **38**(5):53–62, 2018.

[6] Mingchuan Zhou, Long Cheng, Manuel Dell'Antonio, Xiebing Wang, Zhenshan Bing, M Ali Nasseri, Kai Huang, and Alois Knoll. **Peak Temperature Minimization for Hard Real-Time Systems Using DVS and DPM**, *Journal of Circuits, Systems and Computers*, 1950102, 2018.

[7] Xiebing Wang, Kai Huang, and Alois Knoll. **Performance Optimisation of Parallelized ADAS Applications in FPGA-GPU Heterogeneous Systems: A Case Study With Lane Detection**, *IEEE Transactions on Intelligent Vehicles*, **4**(4):519–531, 2019.

[8] Xiebing Wang, Xuehai Qian, Alois Knoll, and Kai Huang. **Efficient Performance Estimation and Work-group Size Pruning for OpenCL Kernels on GPUs**, *IEEE Transactions on Parallel and Distributed Systems*, **31**(5):1089–1106, 2020.

References

[1] Texas Instruments. **Advanced Driver Assistance (ADAS) Solutions Guide**. https://uk.farnell.com/wcsstore/ExtendedSitesCatalogAssetStore/cms/asset/images/europe/common/applications/automotive/pdf/ti-adas-solution-guide.pdf, 2015. xiii, 2

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. **Rodinia: A benchmark suite for heterogeneous computing**. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009. xv, 78, 79, 95, 96, 104, 112

[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. **Analyzing CUDA workloads using a detailed GPU simulator**. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174. IEEE, 2009. xv, 80, 95, 100

[4] Wikipedia. **Advanced driver assistance systems**. https://en.wikipedia.org/wiki/Advanced_driver_assistance_systems, 2017. 1

[5] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. **Engineering automotive software**. *Proceedings of the IEEE*, **95**(2):356–373, 2007. 1, 3, 9

[6] Jeffrey A Cook, Ilya V Kolmanovsky, David McNamara, Edward C Nelson, and K Venkatesh Prasad. **Control, computing and communications: technologies for the twenty-first century model T**. *Proceedings of the IEEE*, **95**(2):334–355, 2007. 1

[7] David P Rodgers. **Improvements in multiprocessor system design**. In *ACM SIGARCH Computer Architecture News*, **13**, pages 225–231. IEEE Computer Society Press, 1985. 2

[8] MEMS Journal. **Automotive Sensors and Electronics Expo 2017**. http://www.automotivesensors2017.com/, 2017. 3

[9] Tuxera. **What does ECU consolidation mean for automotive storage?** `https://www.tuxera.com/blog/ecu-consolidation-mean-automotive-storage/`, 2018. 3

[10] Intel. **Intel® GO™ Automotive Solutions**. `https://www.intel.com/content/www/us/en/automotive/go-automated-driving.html`, 2017. 3, 11

[11] Kai Huang, Biao Hu, Long Chen, Alois Knoll, and Zhihua Wang. **ADAS on COTS with OpenCL: a case study with lane detection**. *IEEE Transactions on Computers (TC)*, 2017. 4, 57

[12] Jörg Fickenscher, Sebastian Reinhart, Frank Hannig, Jürgen Teich, and Mohamed Essayed Bouzouraa. **Convoy tracking for ADAS on embedded GPUs**. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 959–965. IEEE, 2017. 4

[13] Matina Maria Trompouki, Leonidas Kosmidis, and Nacho Navarro. **An open benchmark implementation for multi-CPU multi-GPU pedestrian detection in automotive systems**. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 305–312. IEEE, 2017. 4

[14] Aaftab Munshi. **The OpenCL specification**. In *IEEE Hot Chips Symposium (HCS)*, pages 1–314. IEEE, 2009. 4, 78, 103

[15] Mohamed Zahran. **Heterogeneous computing: here to stay**. *Communications of the ACM (CACM)*, **60**(3):42–45, 2017. 4

[16] Majdi Ghadhab, Jörg Kaienburg, Martin Süsskraut, and Christof Fetzer. **Is Software Coded Processing an Answer to the Execution Integrity Challenge of Current and Future Automotive Software-Intensive Applications?** In *Advanced Microsystems for Automotive Applications*, pages 263–275. Springer, 2016. 4

[17] Continental. **Continental Strategy Focuses on Automated Driving**. `https://www.continental-corporation.com/en/press/press-releases/automated-driving-128072`, 2012. 9

[18] Qualcomm. **Snapdragon 820 Automotive Platform**. `https://www.qualcomm.com/products/snapdragon/processors/820-automotive`, 2016. 11

[19] NXP. **MPC577xK: Ultra-Reliable MPC577xK MCU for Automotive ADAS & Industrial Radar Applications**. `https://www.nxp.com/products/processors-and-`

microcontrollers/power-architecture-processors/mpc5xxx-55xx-32-bit-mcus/ultra-reliable-mpc57xx-32-bit-automotive-industrial-microcontrollers-mcus/ultra-reliable-mpc577xk-mcu-for-automotive-adas-industrial-radar-applications:MPC577xK, 2016. 11

[20] Nvidia. **NVIDIA DRIVE PX: Scalable AI Supercomputer For Autonomous Driving**. https://www.nvidia.com/en-us/self-driving-cars/drive-px/, 2017. 11

[21] Fred W Rauskolb, Kai Berger, Christian Lipski, Marcus Magnor, Karsten Cornelsen, Jan Effertz, Thomas Form, Fabian Graefe, Sebastian Ohl, Walter Schumacher, et al. **Caroline: An autonomously driving vehicle for urban environments**. *Journal of Field Robotics (JFR)*, **25**(9):674–724, 2008. 11

[22] Jonas Mårtensson, Assad Alam, Sagar Behere, Muhammad Altamash Ahmed Khan, Joakim Kjellberg, Kuo-Yun Liang, Henrik Pettersson, and Dennis Sundman. **The development of a cooperative heavy-duty vehicle for the GCDC 2011: Team Scoop**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, **13**(3):1033–1049, 2012. 11

[23] Andreas Geiger, Martin Lauer, Frank Moosmann, Benjamin Ranft, Holger Rapp, Christoph Stiller, and Julius Ziegler. **Team AnnieWAY's entry to the 2011 grand cooperative driving challenge**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, **13**(3):1008–1017, 2012. 11

[24] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. **Junior: The stanford entry in the urban challenge**. *Journal of Field Robotics (JFR)*, **25**(9):569–597, 2008. 11

[25] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. **Autonomous driving in urban environments: Boss and the urban challenge**. *Journal of Field Robotics (JFR)*, **25**(8):425–466, 2008. 11

[26] Inwook Shim, Jongwon Choi, Seunghak Shin, Tae-Hyun Oh, Unghui Lee, Byungtae Ahn, Dong-Geol Choi, David Hyunchul Shim, and In-So Kweon. **An autonomous driving system for unknown environments using a unified map**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, **16**(4):1999–2013, 2015. 11

[27] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. **Development of autonomous car–Part II: A case study on the implementation of an autonomous driving system based on distributed architecture**. *IEEE Transactions on Industrial Electronics (TIE)*, **62**(8):5119–5132, 2015. 11

[28] Stefanos Kokogias, Lars Svensson, Gonçalo Collares Pereira, Rui Oliveira, Xinhai Zhang, Xinwu Song, and Jonas Mårtensson. **Development of Platform-Independent System for Cooperative Automated Driving Evaluated in GCDC 2016**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, 2017. 11

[29] Khronos Group. **OpenCL Overview**. `https://www.khronos.org/opencl/`, 2017. 13

[30] Kai Huang, Biao Hu, Jan Botsch, Nikhil Madduri, and Alois Knoll. **A scalable lane detection algorithm on cotss with OpenCL**. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 229–232. IEEE, 2016. 17, 25, 26

[31] Xiebing Wang, Linlin Liu, Kai Huang, and Alois Knoll. **Exploring FPGA-GPU Heterogeneous Architecture for ADAS: Towards Performance and Energy**. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 33–48. Springer, 2017. 18, 24, 62

[32] Xiebing Wang, Christopher Kiwus, Canhao Wu, Biao Hu, Kai Huang, and Alois Knoll. **Implementing and Parallelizing Real-time Lane Detection on Heterogeneous Platforms**. In *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018. 24

[33] Albert S Huang, David Moore, Matthew Antone, Edwin Olson, and Seth Teller. **Finding multiple lanes in urban road networks with vision and lidar**. *Autonomous Robots*, **26**(2):103–122, 2009. 25

[34] Qingquan Li, Long Chen, Ming Li, Shih-Lung Shaw, and Andreas Nuchter. **A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios**. *IEEE Transactions on Vehicular Technology (TVT)*, **63**(2):540–555, 2014. 25

[35] Junaed Sattar and Jiawei Mo. **SafeDrive: A Robust Lane Tracking System for Autonomous and Assisted Driving Under Limited Visibility**. *arXiv preprint arXiv:1701.08449*, 2017. 25

[36] Byambaa Dorj and Deok Jin Lee. **A precise lane detection algorithm based on top view image transformation and least-square approaches**. *Journal of Sensors*, **2016**, 2016. 25, 26

[37] Jianwei Niu, Jie Lu, Mingliang Xu, Pei Lv, and Xiaoke Zhao. **Robust Lane Detection using Two-stage Feature Extraction with Curve Fitting**. *Pattern Recognition*, **59**:225–233, 2016. 25

[38] Somchok Sakjiraphong, Andre Pinho, Matthew N Dailey, Mongkol Ekpanyapong, and Adriano Tavares. **Real-time road lane detection with commodity hardware**. In *Proceedings of the International Electrical Engineering Congress*, pages 1–4. IEEE, 2014. 25, 26, 56, 57

[39] Shuliang Zhu, Jianqiang Wang, Tao Yu, and Jiao Wang. **A method of lane detection and tracking for expressway based on RANSAC**. In *IEEE 2nd International Conference on Image, Vision and Computing*, pages 62–66. IEEE, 2017. 25, 26

[40] Huachun Tan, Yang Zhou, Yong Zhu, Danya Yao, and Keqiang Li. **A novel curve lane detection based on improved river flow and ransa**. In *IEEE 17th International Conference on Intelligent Transportation Systems (ITSC)*, pages 133–138. IEEE, 2014. 25, 26

[41] Hanyu Xuan, Hongzhe Liu, Jiazheng Yuan, and Qing Li. **Robust Lane-mark Extraction for Autonomous Driving under Complex Real Conditions**. *IEEE Access*, 2017. 25, 26

[42] Jongin Son, Hunjae Yoo, Sanghoon Kim, and Kwanghoon Sohn. **Real-time illumination invariant lane detection for lane departure warning system**. *Expert Systems with Applications*, **42**(4):1816–1824, 2015. 25

[43] Hunjae Yoo, Ukil Yang, and Kwanghoon Sohn. **Gradient-enhancing conversion for illumination-robust lane detection**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, **14**(3):1083–1094, 2013. 25, 26

[44] Wenjie Lu, Emmanuel Seignez, Roger Reynaud, et al. **Monocular multi-kernel based lane marking detection**. In *IEEE 4th Annual International Conference on Cyber Technology in Automation, Control, and Intelligent Systems*, pages 123–128. IEEE, 2014. 26

[45] Seung-Nam Kang, Soomok Lee, Junhwa Hur, and Seung-Woo Seo. **Multi-lane detection based on accurate geometric lane estimation in highway scenarios**. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 221–226. IEEE, 2014. 26

[46] Tin Trung Duong, Cuong Cao Pham, Tai Huu-Phuong Tran, Tien Phuoc Nguyen, and Jae Wook Jeon. **Near real-time ego-lane detection in highway and urban streets**. In *IEEE International Conference on Consumer Electronics-Asia*, pages 1–4. IEEE, 2016. 26

[47] Raghuraman Gopalan, Tsai Hong, Michael Shneier, and Rama Chellappa. **A learning approach towards detection and tracking of lane markings**. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, **13**(3):1088–1098, 2012. 26

[48] Martin A Fischler and Robert C Bolles. **Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography**. *Communications of the ACM (CACM)*, **24**(6):381–395, 1981. 26, 64

[49] Xiangjing An, Erke Shang, Jinze Song, Jian Li, and Hangen He. **Real-time lane departure warning system based on a single FPGA**. *EURASIP Journal on Image and Video Processing*, **2013**(1):38, 2013. 26, 57

[50] Roberto Marzotto, Paul Zoratti, Daniele Bagni, Andrea Colombari, and Vittorio Murino. **A real-time versatile roadway path extraction and tracking on an FPGA platform**. *Computer Vision and Image Understanding*, **114**(11):1164–1179, 2010. 26, 57

[51] Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, and Konrad Doll. **FPGA-based real-time pedestrian detection on high-resolution images**. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 629–635, 2013. 26, 56, 57

[52] Petko Georgiev, Nicholas D Lane, Cecilia Mascolo, and David Chu. **Accelerating Mobile Audio Sensing Algorithms through On-Chip GPU Offloading**. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 306–318. ACM, 2017. 26, 56, 57

[53] Weijing Shi, Xin Li, Zhiyi Yu, and Gary Overett. **An FPGA-Based Hardware Accelerator for Traffic Sign Detection**. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, **25**(4):1362–1372, 2017. 26, 56, 57

[54] Irvin Sobel. **An isotropic 3× 3 image gradient operator**. *Machine Vision for three-demensional Sciences*, 1990. 27

[55] Neil J Gordon, David J Salmond, and Adrian FM Smith. **Novel approach to nonlinear/non-Gaussian Bayesian state estimation**. In *IEE Proceedings F-Radar and Signal Processing*, **140**, pages 107–113. IET, 1993. 29

[56] David B. Thomas. **The MWC64X Random Number Generator**. `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`, 2011. 31

[57] Mohamed Aly. **Real time detection of lane markers in urban streets**. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 7–12. IEEE, 2008. 34, 35, 36

[58] Hui Kong, Jean-Yves Audibert, and Jean Ponce. **General road detection from a single image**. *IEEE Transactions on Image Processing (TIP)*, **19**(8):2211–2220, 2010. 35

[59] Intel. **Intel® FPGA SDK for OpenCL™**. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf`, 2017. 43

[60] Mohamed Aly. **Caltech Lanes**. `http://www.vision.caltech.edu/malaa/datasets/caltech-lanes`, 2014. 43, 64

[61] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. **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 (ISFPGA)*, pages 47–56. ACM, 2012. 43

[62] Intel. **PowerPlay Early Power Estimators and Power Analyzer**. `https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/pow-powerplay.html`, 2017. 43

[63] Weiyan Wang, Yunquan Zhang, Shengen Yan, Ying Zhang, and Haipeng Jia. **Parallelization and performance optimization on face detection algorithm with OpenCL: A case study**. *Tsinghua Science and Technology*, **17**(3):287–295, 2012. 56, 57

[64] Xuechao Wei, Yun Liang, Tao Wang, Songwu Lu, and Jason Cong. **Throughput optimization for streaming applications on CPU-FPGA heterogeneous systems**. In *22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 488–493. IEEE, 2017. 56, 57

[65] Robert Dietrich and Ronny Tschüter. **A generic infrastructure for OpenCL performance analysis**. In *IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, **1**, pages 334–341. IEEE, 2015. 57

[66] Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, and Cristina Silvano. **Opencl application auto-tuning and run-time resource management for multi-core platforms**. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 127–133. IEEE, 2014. 57

[67] Thibaut Lutz, Christian Fensch, and Murray Cole. **Helium: a transparent inter-kernel optimizer for OpenCL**. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs (GPGPU)*, pages 70–80. ACM, 2015. 57

[68] Perhaad Mistry, Chris Gregg, Norman Rubin, David Kaeli, and Kim Hazelwood. **Analyzing program flow within a many-kernel OpenCL application**. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, page 10. ACM, 2011. 57

[69] Jialiang Zhang and Jing Li. **Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network**. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pages 25–34. ACM, 2017. 57

[70] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. **Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks**. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pages 16–25. ACM, 2016. 57

[71] Yang Xing, Chen Lv, Long Chen, Huaji Wang, Hong Wang, Dongpu Cao, Efstathios Velenis, and Fei-Yue Wang. **Advances in Vision-Based Lane Detection: Algorithms, Integration, Assessment, and Perspectives on ACP-Based Parallel Vision**. *IEEE/CAA Journal of Automatica Sinica*, **5**(3):645–661, 2018. 57

[72] Calum Blair, Neil M Robertson, and Danny Hume. **Characterizing a heterogeneous system for person detection in video using histograms of oriented gradients: Power versus speed versus accuracy**. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, **3**(2):236–247, 2013. 57

[73] Matthew Yih, Jeffrey M Ota, John D Owens, and Pinar Muyan-Özçelik. **FPGA versus GPU for Speed-Limit-Sign Recognition**. In *IEEE 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 843–850. IEEE, 2018. 57

[74] Akira Nukada and Satoshi Matsuoka. **Auto-tuning 3-D FFT library for CUDA GPUs**. In *Proceedings of International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, page 30. ACM, 2009. 77, 105

[75] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. **Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures.** In *5th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, pages 111–125. Springer, 2010. 77

[76] Jee W Choi, Amik Singh, and Richard W Vuduc. **Model-driven autotuning of sparse matrix-vector multiply on GPUs**. In *ACM SIGPLAN notices*, **45**, pages 115–126. ACM, 2010. 77

[77] Andrew Davidson, Yao Zhang, and John D Owens. **An auto-tuned method for solving large tridiagonal systems on the GPU**. In *IEEE 25th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 956–965. IEEE, 2011. 77

[78] Cedric Nugteren and Valeriu Codreanu. **CLTune: A generic auto-tuner for OpenCL kernels**. In *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 195–202. IEEE, 2015. 77, 104, 105

[79] Thomas L Falch and Anne C Elster. **Machine learning based auto-tuning for enhanced OpenCL performance portability**. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 1231–1240. IEEE, 2015. 77, 104, 105

[80] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. **Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance**. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 725–737. ACM, 2015. 78, 80

[81] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. **GPGPU performance and power estimation using machine learning**. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576. IEEE, 2015. 78, 80

[82] Kenneth O'Neal, Philip Brisk, Ahmed Abousamra, Zack Waters, and Emily Shriver. **GPU Performance Estimation using Software Rasterization and Machine Learning**. *ACM Transactions on Embedded Computing Systems (TECS)*, **16**(5s):148, 2017. 78, 80

[83] Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort. **The landscape of GPGPU performance modeling tools**. *Parallel Computing*, **56**:18–33, 2016. 78

[84] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy John, Hai Jin, and Chengzhong Xu. **Accelerating GPGPU architecture simulation**. In *ACM SIGMETRICS Performance Evaluation Review*, **41**, pages 331–332. ACM, 2013. 78

[85] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S Lee. **TBPoint: Reducing simulation time for large-scale GPGPU kernels**. In *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 437–446. IEEE, 2014. 78

[86] Chris Lattner and Vikram Adve. **LLVM: A compilation framework for lifelong program analysis & transformation**. In *Proceedings of 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, page 75. IEEE Computer Society, 2004. 78, 105

[87] Sangpil Lee and Won Woo Ro. **Parallel GPU architecture simulation framework exploiting work allocation unit parallelism**. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–117. IEEE, 2013. 78, 100

[88] Geetika Malhotra, Seep Goel, and Smruti R Sarangi. **Gputejas: A parallel simulator for gpu architectures**. In *IEEE 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014. 78, 100

[89] Karthik Ganesan, Jungho Jo, and Lizy K John. **Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads**. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 33–44. IEEE, 2010. 79

[90] Reena Panda, Xinnian Zheng, Shuang Song, Jee Ho Ryoo, Michael LeBeane, Andreas Gerstlauer, and Lizy K John. **Genesys: Automatically generating representative training sets for predictive benchmarking**. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 116–123. IEEE, 2016. 79

[91] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. **Modeling program resource demand using inherent program characteristics**. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and modeling of computer systems*, pages 1–12. ACM, 2011. 79

[92] Xinnian Zheng, Haris Vikalo, Shuang Song, Lizy K John, and Andreas Gerstlauer. **Sampling-based binary-level cross-platform performance estimation**. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 1713–1718. European Design and Automation Association, 2017. 79

[93] Sunpyo Hong and Hyesoon Kim. **An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness**. In *ACM SIGARCH Computer Architecture News*, **37**, pages 152–163. ACM, 2009. 79

[94] Kishore Kothapalli, Rishabh Mukherjee, M Suhail Rehman, Suryakant Patidar, PJ Narayanan, and Kannan Srinathan. **A performance prediction model for the CUDA GPGPU platform**. In *IEEE 16th International Conference on High Performance Computing (HiPC)*, pages 463–472. IEEE, 2009. 79

[95] Leslie G Valiant. **A bridging model for parallel computation**. *Communications of the ACM (CACM)*, **33**(8):103–111, 1990. 79

[96] Steven Fortune and James Wyllie. **Parallelism in random access machines**. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 114–118. ACM, 1978. 79

[97] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. **The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms**. *SIAM Journal on Computing*, pages 638–648, 1997. 79

[98] Marcos Amarís, Daniel Cordeiro, Alfredo Goldman, and Raphael Y de Camargo. **A simple bsp-based model to predict execution time in gpu applications**. In *IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 285–294. IEEE, 2015. 79

[99] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. **An adaptive performance modeling tool for GPU architectures**. In *ACM SIGPLAN Notices*, **45**, pages 105–114. ACM, 2010. 79

[100] Yao Zhang and John D Owens. **A quantitative performance analysis model for GPU architectures**. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 382–393. IEEE, 2011. 79

[101] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. **A simplified and accurate model of power-performance efficiency on emergent GPU architectures**. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 673–686. IEEE, 2013. 79

[102] Qiang Wang and Xiaowen Chu. **GPGPU performance estimation with core and memory frequency scaling**. In *IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 417–424. IEEE, 2018. 79

[103] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. **A performance analysis framework for exploiting GPU microarchitectural capability**. In *Proceedings of the International Conference on Supercomputing (ICS)*, page 15. ACM, 2017. 79

[104] Ioana Baldini, Stephen J Fink, and Erik Altman. **Predicting gpu performance from cpu runs using machine learning**. In *IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 254–261. IEEE, 2014. 80

[105] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. **Performance and power analysis of ATI GPU: A statistical approach**. In *IEEE 6th International Conference on Networking, Architecture and Storage (NAS)*, pages 149–158. IEEE, 2011. 80

[106] Marcos Amarís, Raphael Y de Camargo, Mohamed Dyab, Alfredo Goldman, and Denis Trystram. **A comparison of GPU execution time prediction using machine learning and analytical modeling**. In *IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333. IEEE, 2016. 80

[107] Thanh Tuan Dao, Jungwon Kim, Sangmin Seo, Bernhard Egger, and Jaejin Lee. **A performance model for gpus with caches**. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **26**(7):1800–1813, 2015. 80

[108] Christoph Gerum, Oliver Bringmann, and Wolfgang Rosenstiel. **Source level performance simulation of gpu cores**. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 217–222. EDA Consortium, 2015. 80

[109] Sylvain Collange, Marc Daumas, David Defour, and David Parello. **Barra: A parallel functional simulator for gpgpu**. In *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360. IEEE, 2010. 80

[110] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. **Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems**. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. ACM, 2010. 80

[111] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. **Enabling GPGPU low-level hardware explorations with MIAOW: an open-source RTL implementation of a GPGPU**. *ACM Transactions on Architecture and Code Optimization (TACO)*, **12**(2):21, 2015. 80

[112] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy K John, Hai Jin, Chengzhong Xu, and Junmin Wu. **GPGPU-MiniBench: Accelerating GPGPU micro-architecture simulation**. *IEEE Transactions on Computers (TC)*, **64**(11):3153–3166, 2015. 81

[113] Kishore Punniyamurthy, Behzad Boroujerdian, and Andreas Gerstlauer. **GATSim: abstract timing simulation of GPUs**. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 43–48. European Design and Automation Association, 2017. 81

[114] Jee Ho Ryoo, Saddam J Quirem, Michael Lebeane, Reena Panda, Shuang Song, and Lizy K John. **Gpgpu benchmark suites: How well do they sample the performance spectrum?** In *44th International Conference on Parallel Processing (ICPP)*, pages 320–329. IEEE, 2015. 81

[115] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy K John. **Statistical pattern based modeling of GPU memory access streams**. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, page 81. ACM, 2017. 81

[116] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. **Automatic Loop Summarization via Path Dependency Analysis**. *IEEE Transactions on Software Engineering (TSE)*, (1):1–1, 2017. 83

[117] Moritz Sinn and Florian Zuleger. **Loopus: A Tool for Computing Loop Bounds for C Programs.** In *Proceedings of the Workshop on Invariant Generation (WING)*, pages 185–186, 2010. 83

[118] Robert A Van Engelen. **Efficient symbolic analysis for optimizing compilers**. In *International Conference on Compiler Construction (CC)*, pages 118–132. Springer, 2001. 83

[119] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. **Demystifying GPU microarchitecture through microbenchmarking**. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010. 90

[120] Xinxin Mei and Xiaowen Chu. **Dissecting GPU memory hierarchy through microbenchmarking**. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **28**(1):72–86, 2017. 90

[121] Siqi Wang, Guanwen Zhong, and Tulika Mitra. **CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications**. *ACM Transactions on Embedded Computing Systems (TECS)*, **16**(5s):146, 2017. 90

[122] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. **Automatic OpenCL device characterization: guiding optimized kernel design**. In *17th International European Conference on Parallel Processing (Euro-Par)*, pages 438–452. Springer, 2011. 92

[123] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. **Opentuner: an extensible framework for program autotuning**. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 303–316. ACM, 2014. 104, 105

[124] Shane Ryoo, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, John Stratton, and Wen-mei Hwu. **Program optimization space pruning for a multithreaded GPU**. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 195–204. ACM, 2008. 104, 105

[125] Thanh Tuan Dao and Jaejin Lee. **An Auto-Tuner for OpenCL Work-Group Size on GPUs**. *IEEE Transactions on Parallel & Distributed Systems (TPDS)*, pages 283–296, 2018. 104, 105

[126] Changhao Jiang and Marc Snir. **Automatic tuning matrix multiplication performance on graphics hardware**. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 185–194. IEEE, 2005. 105

[127] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. **Optimization of sparse matrix-vector multiplication on emerging multicore platforms**. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*, pages 1–12. IEEE, 2007. 105

[128] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. **Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures**. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*. IEEE, 2008. 105

[129] Robert Lim, Boyana Norris, and Allen Malony. **Autotuning GPU Kernels via Static and Predictive Analysis**. In *46th International Conference on Parallel Processing (ICPP)*, pages 523–532. IEEE, 2017. 105

[130] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. **Automatic OpenCL work-group size selection for multicore CPUs**. In *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 387–397. IEEE, 2013. 105

[131] Simon Moll, Johannes Doerfert, and Sebastian Hack. **Input space splitting for OpenCL**. In *25th International Conference on Compiler Construction (CC)*, pages 251–260. ACM, 2016. 105

[132] Xiebing Wang, Kai Huang, Alois Knoll, and Xuehai Qian. **A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation**. In *IEEE 25th International Symposium on High Performance Computer Architecture (HPCA)*, pages 506–518. IEEE, 2019. 109