Improving the Performance of ADAS Application in Heterogeneous Context: a Case of Lane Detection

Xiebing Wang, Mingyue Cui, Kai Huang, Alois Knoll and Long Chen

Abstract—This paper investigates the optimization of OpenCL-based ADAS applications in heterogeneous context. In particular, we take the widely-used lane detection algorithm (LDA) as a case study. The application is profiled to identify the performance bottlenecks and then three optimization strategies are adopted. On the kernel side, the parallel granularity is regulated via compute unit replication and loop unrolling. On the host side, the kernel API function calls are scheduled in an interleaved manner to overlap the accelerator execution time. Moreover, the computation workload of the algorithm is tuned by dynamically adjusting the processed image ROI size. Experimental results reveal that the optimized implementation can achieve an average 2.27x speedup when compared with the naive parallel application.

I. INTRODUCTION

Modern vehicles are equipped with lots of electronic control units (ECUs) used for diverse functionalities such as driving, safety, navigation, and in-car entertainment, etc. This progressively complex system poses us a big challenge: how to utilize the limited computing power to deal with various real-time and safety-critical tasks? 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 hardware components. However, this proved to be both unrealistic and minimal due to the limitation of financial cost and Amdahl’s law [1].

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. Taking advanced driver assistance systems (ADAS) for instance, it is originally developed to adapt vehicle systems for safety and better driving. To be specific, the system collects environmental data by miscellaneous sensors and processes them as real-time as possible to make an evaluation of current vehicle runtime status. Then by means of different control strategies, the system either takes emergency measures by itself when necessary or gives response to drivers to assist their driving. Here the time constraints is extremely strict and therefore high computing power is required. With unified accelerators, all these operations are handled by, say, CPUs or GPUs, which could be time-consuming since each type of the processors is only favorable of specific data operations. Nevertheless the time cost can be greatly shortened when both CPUs, GPUs and FPGAs are used to perform their adept operations.

In heterogeneous context, real time constraint is well handled since different accelerators are used to perform their adept operations. However, how to schedule different accelerators to gain optimal performance is nontrivial. Taking commonly used lane detection algorithm (LDA) as case study, this paper gives a thorough profiling analysis of the heterogeneous OpenCL implementation and then optimizes the application using both compiler optimization options and built-in application program interface (API) function scheduling. The test LDA is based on [2]. In [2], the authors developed a particle-filter based algorithm that could detect and track on-road lane markings. However, they only proposed the algorithm design and demonstrated the performance on single accelerators. Heterogeneous architecture was not involved and little effort was given to improve the overall performance.

We customized this algorithm to enable its simultaneous execution under an FPGA-GPU combined heterogeneous architecture and then optimized it to a very large extent. First of all, the kernel and host code blocks are profiled to identify the performance bottlenecks. Afterwards on the kernel side, pragma primitives that adjust the parallel granularity are used to gain the performance enhancement. On the host side, the order of the API function calls is investigated and carefully scheduled to hidden part of the accelerator execution time. Finally with regard to the algorithm itself, computation task is dynamically changed to further increase the performance. Experimental results present an average 2.27x speedup when compared with the unoptimized parallel LDA application.

The rest of this paper is organized as follows: Section II is related work and Section III overviews the LDA and the heterogeneous design. Section IV presents our profiling and optimization strategies. Section V gives experimental results and Section VI concludes the paper.

II. RELATED WORK

LDA is mostly achieved via filtering techniques to capture lanes like the work in [3] [4] [5], however it is rarely adopted on the heterogeneous platform. In this paper we focus on the optimization of the OpenCL-based LDA application. As is known, the performance portability of OpenCL applications running among different accelerators remains an open problem. To solve this issue, some researchers proposed profiling and kernel optimization framework to assist better development of OpenCL applications. Authors in [6] presented...
A generic tool interface for performance measurement of OpenCL programs. They wrapped the OpenCL API functions and kernel events to exhibit time costs of kernel and function calls. Currently the tool is under construction and some features like memory and buffer routines are not implemented yet. In [7], the authors proposed a framework combing OpenCL application auto-tuning and runtime resource management. Authors in [8] presented a transparent OpenCL overlay called Helium, for inter- and intra-kernel optimization.

The work mentioned above are yet not mature and existing researches still stay in the phrase that optimizations are performed based on the specific algorithm, architecture and OpenCL specifications. In [9], the authors analyzed and profiled the components of the speedup 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 [10] and [11]. In their work, optimizations were mainly performed based on the CNN algorithm itself.

The most related work to this paper is [12], where authors used step by step optimization of face detection algorithm including CPU execution time hidden, memory coalescing and variable parallel granularity. The difference of our work is that rather than using single GPU, we tested the heterogeneous context so that (i) the execution time of accelerators can also be hidden via changing build-in function order and (ii) parallelism on FPGA side could be further adjusted by using pragma primitives.

III. OVERVIEW AND IMPLEMENTATION OF THE LDA

A. Overview

As shown in Figure 1, the tested LDA mainly consists of three modules, namely pre-processing, lane detection and lane tracking. For each frame, the pre-processing module extracts information about the lane markings and then passes the processed image to the next step. Depending on whether or not the estimated state in previous frame can still be applied to current frame, the image is processed either using lane detection module to detect the positions of the lane markings or using lane tracking module to track the previous position of the lane markings.

The pre-processing module contains four steps successively applied to the original image. First a region of interest (ROI) is cropped from the raw image and only this ROI is further processed. Then the ROI is transformed into a greyscale format where each pixel reflects the intensities of the pixel in original image. To enhance the contrast of pixel intensity in the ROI, a Sobel filter [13] is applied to the grayscaled image to extract transitions and edges. To avoid the influence of noises, a threshold is used to tune the intensity of all pixels in the image.

During lane detection, a set of candidate lines are randomly generated via assigning random values from a normal distribution to the elements in the candidate line set $X = \{X_1, X_2, \ldots, X_n\}$, 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. With this set, the line with the highest weight is chosen as the best line and certain number of candidate lines are reserved as good lines, which would be further used in the lane tracking module.

For lane tracking, a particle filter [14] is adopted to predict the lane markings, using both the ROI of the current frame and the best line and good lines of the previous frame. The particle filter consists of three steps: (i) the prediction update step modifies previous good lines as prior probability distribution of lane markings in current frame; (ii) the importance weight update step recalculates the weights of the particles and (iii) the resampling step selects particles from the newly updated set so as to prevent a degeneration of the particle set.

B. Parallel implementation

The pre-processing module presents a high potential of parallelization since each pixel in the ROI can perform grayscale and thresholding manipulations by itself. Moreover, the Sobel filter requires only 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. The resampling step, in contrast, relies on knowledge from the whole particle set and thus is performed on the host. Again we use a kernel kernelPF 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 our tested LDA, these numbers are generated by MWC64X [15], 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, we also introduce a kernel called kernelRNG to realize it. In current work this kernel initializes a stream of random numbers and splits them with a period of $2^{40}$, which

![Flow chart of LDA](image-url)
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.

C. Heterogeneous context

In the heterogeneous context, the host utilizes an installable client driver (ICD) loader to coordinate the tasks handled on FPGA and GPU. When invoking OpenCL API functions, the program runtime passes kernel parameters to the ICD loader and then the loader enables FPGA- or GPU-specific functions with required fpga- or gpu-specific parameters. The host side is responsible for (i) kernel parameters initialization and raw image I/O at the program beginning, and (ii) data collection, weight updating and resampling during each iteration. On the hardware accelerators, the four kernels are deployed on both FPGA and GPU.

IV. PROFILING AND OPTIMIZATION

A. Profiling

Profiling of the application is needed to locate the hotspot of the source code so that the bottleneck can be identified and optimized. To figure out the execution time and flow distribution of the program, the high-level source code is segmented into several blocks and then the execution time of each block is measured. Table I gives a list of the main code blocks whose executions 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 time is optimized first.

As can be seen in Section V-B, the kernel execution time takes up the majority of overall time cost, consequently in the following we focus on the kernel optimization and give three strategies to improve the performance of LDA.

B. Optimization

1) Compute unit replication and loop unrolling: In OpenCL, the kernel code is instantiated as a work item running on a compute unit and a group of work items can execute simultaneously to accelerate the applications. 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. Especially on the FPGA platform, replicating kernel compute units ensures the increase of data throughput at the expense of memory bandwidth contention among compute units.

Loop unrolling is a code transformation technique used to reduce 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 branches are minimized. On the GPU side, loop unrolling is implemented by manually replacing the loop with repeated sequential statements which eliminates the branches penalty. Loop unrolling on the FPGA board increases the length of pipeline, thus overlapping the executions of more logic units. On both platforms, due to the expansion of loop size, memory read of data chunk can be coalesced as they have adjacent memory addresses.

For brevity, we use $\lambda_{RC}$ and $\lambda_{LU}$ as notes for the number of replicated compute units and loop unrolling factor, respectively. Due to device resource limitation, on our test platform the compute units can be replicated in maximum 3 times and the loop is unrolled at most 9 times.

2) Accelerator execution time overlapping: The heterogeneous implementation of the LDA is data-level parallel and each kernel is executed on both FPGA and GPU boards. Here exists a trade-off of how and when the kernels are invoked from the host side. Generally speaking, an overall execution of an OpenCL kernel contains at least two API functions: clEnqueueNDRangeKernel function drives the kernel code to run and clWaitForEvents function waits on the host for kernel commands to complete. The kernel function clEnqueueNDRangeKernel is non-blocking, while function clWaitForEvents is blocking.

In our implementation, the ICD loader is used and each cl-function call passes the handler to the corresponding FPGA or GPU libraries to execute the fpga- or gpu-specific functions. With regard to the aforementioned four kernels, each time functions fpgaEnqueueNDRangeKernel, fpgaWaitForEvents, gpuEnqueueNDRangeKernel and gpuWaitForEvents are respectively called once. These function calls can be interleaved and consequently their execution order should be carefully considered. For simplicity, we use short symbols to represent these API functions and the details are shown in Table II. To investigate how the call order of these functions influences the final performance, the functions are permuted to obtain the full sample space and each case is tested (results shown in Section V-C2).

3) Adjustable ROI size: As described in Section III-A, only the ROI of the image frame is processed and information of pixels falling in this area is further computed. Therefore decreasing the ROI size could distinctly shrink the calculation

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>kernelRNG</td>
<td>random number generation</td>
</tr>
<tr>
<td>2</td>
<td>cpMatToArray</td>
<td>copy image matrix data into array</td>
</tr>
<tr>
<td>3</td>
<td>kernelPRE</td>
<td>pre-processing of raw image</td>
</tr>
<tr>
<td>4</td>
<td>kernelLD</td>
<td>lane detection of ROI</td>
</tr>
<tr>
<td>5</td>
<td>extractLine</td>
<td>extract good and best lines</td>
</tr>
<tr>
<td>6</td>
<td>kernelPF</td>
<td>lane tracking of ROI</td>
</tr>
<tr>
<td>7</td>
<td>resample</td>
<td>particles resampling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbols</th>
<th>API functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>fpgaEnqueueNDRangeKernel</td>
</tr>
<tr>
<td>$P_2$</td>
<td>fpgaWaitForEvents</td>
</tr>
<tr>
<td>$G_1$</td>
<td>gpuEnqueueNDRangeKernel</td>
</tr>
<tr>
<td>$G_2$</td>
<td>gpuWaitForEvents</td>
</tr>
</tbody>
</table>
Algorithm 1 ROI adjusting scheme

\begin{verbatim}
Input: B, roiStart, roiWidth, initRoiStart, initRoiEnd, imgWidth
Output: roiStartAdapted, roiWidthAdapted
1: roiStartAdapted ← roiStart
2: roiEndAdapted ← roiStart + roiWidth
3: for all bestLine ∈ B do
4:    roiStartAdapted ← min{roiStartAdapted, bestLineStart}
5:    roiEndAdapted ← max{roiEndAdapted, bestLineEnd}
6: end for
7: if roiStartAdapted < initRoiStart then
8:   roiStartAdapted ← initRoiStart
9: else if roiStartAdapted > imgWidth * 0.25 then
10:  roiStartAdapted ← imgWidth * 0.25
11: end if
12: if roiEndAdapted > initRoiEnd then
13:  roiEndAdapted ← initRoiEnd
14: else if roiEndAdapted < imgWidth * 0.75 then
15:  roiEndAdapted ← imgWidth * 0.75
16: end if
17: roiWidthAdapted ← roiEndAdapted − roiStartAdapted
18: if redetection then
19:   roiStartAdapted ← initRoiStart
20:   roiWidthAdapted ← initRoiEnd − initRoiStart
21: end if
\end{verbatim}

Figure 2 reveals the normalized execution time distribution of the profiled code blocks.

B. Profiling results

Figure 2 reveals the normalized execution time distribution of the code blocks in Table I. As can be observed, kernelLD and kernelPF account for minimum 61.28% (when FPGA task proportion is 10%) and maximum 85.61% (when FPGA task proportion is 90%) of the total execution time. These two kernels are therefore the hotspot of the program and need to be optimized with top priority. The optimization methods illustrated in Section IV-B1 and IV-B2 is aimed at this and the ROI adjusting scheme is used to accelerate the whole program. Besides, note that cpMatToArray also consumes considerable time, this is inevitable since the raw image data has to be read into memory. Optimization of this code block is viable by either using faster transmission medium, which is beyond the scope of this paper, or reducing the transmitted data, which is done by the ROI adjusting scheme.

C. Optimization results

1) Compute unit replication and loop unrolling: As mentioned in Section IV-B1, the maximum values of $\lambda_{RC}$ and $\lambda_{LU}$ are 3 and 9, respectively. We iterated the overall conditions and found that $\lambda_{RC}$ and $\lambda_{LU}$ cannot reach the maximum value at the same time, because of the resource limitation on the FPGA. In details, when $\lambda_{RC} = 1$, $\lambda_{LU}$ is valid with data range of $[1, 2, \cdots, 9]$. When $\lambda_{RC} = 2$, maximum of $\lambda_{LU}$ is

Table III is the detailed information about the platform used in our evaluation and Table IV lists the video streams used in our experiment, of which the videos cordova1, cordova2, washington1 and washington2 are from caltech lanes dataset [16], while others are self-recorded. From the table it is seen that the frame numbers of the videos have a great range from 232 to 4992. Moreover, these videos represent various road situations including day and night, heavy traffic, blurred and broken lines, street and highway, etc. This aims to demonstrate a high availability of using the tested LDA for real scenarios and hence obtain as actual results as possible.

As for the parameters of the LDA, we use $2^{12}$ good lines and $2^{12}$ candidate lines to detect 2 lane markings. Each time the FPGA side is allocated with different task proportions, i.e., from 10% to 90% (vice versa the task proportion on the GPU is from 90% to 10%). Each video is run 10 times per device and at last the overall results are collected and averaged.

Table III: Detailed specification of the hardware platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host CPU</td>
<td>Intel Core 2 Quad Q9300 @ 2.50GHz 4 Cores</td>
</tr>
<tr>
<td>Device</td>
<td>FPGA</td>
</tr>
<tr>
<td>Model</td>
<td>Nallatech 385</td>
</tr>
<tr>
<td>Architecture</td>
<td>Stratix V GS</td>
</tr>
<tr>
<td>OpenCL SDK version</td>
<td>Intel FPGA SDK 13.1</td>
</tr>
<tr>
<td>Peak GFLOPS</td>
<td>294.7</td>
</tr>
</tbody>
</table>

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 27,2022 at 14:27:44 UTC from IEEE Xplore. Restrictions apply.
3. As $\lambda_{RC}$ achieves 3, the valid value of $\lambda_{LU}$ is only 1. Table V summarizes all the possible conditions.

Figure 3 gives the result of the program performance when $\lambda_{RC}$ and $\lambda_{LU}$ adopt different values. Due to space limit, the figure only shows results when task proportion on the FPGA is assigned as 10%, 30%, 50%, 70% and 90%, respectively. From the figure it is seen that using compute unit replication gains a larger performance improvement than performing loop unrolling. When comparing Condition #1 and Condition #2(#3), or comparing Condition #10 and Condition #11(#12), we can see that loop unrolling could even degrade the program performance. The lesson from here is that compute unit replication is always preferred and loop unrolling speeds up the performance only when $\lambda_{LU}$ is very large (greater than 2 in our case).

2) Accelerator execution time overlapping: Section IV-B2 indicates that the order of function call influences the total execution time. Consequently we explored the permutations of the functions in Table II and tested the possible cases.

Since functions $F_2$ and $G_2$ must always be called before $F_1$ and $G_1$ respectively, in total six conditions are deduced and the performance results of each case is presented in Figure 4. It is interesting to observe that according to their respective performance, the six cases can be divided into three groups: ① $\{F_1,F_2,G_1,G_2,F_2,F_1\}$, ② $\{F_1,G_1,F_2,G_2,G_1,F_1,F_2,G_2\}$, and ③ $\{F_1,G_1,F_2,G_2,G_1,F_1,F_2,G_2\}$. Each case in the same group gains the equivalent performance. Result of Group ① is easily understood as the kernel executions on FPGA and GPU are sequential, thus there is no hidden execution time. Cases in Group ② and ③ exhibit the accelerate execution time overlapping and the total time cost is shortened. As can be seen, Group ② always consume less time than Group ③ and this is due to the reason that function call of $F_2$ is before $G_2$. Since $F_2$ and $G_2$ are blocking the process after called, they have to wait after kernel operations are completed and then return the handler to the host. Libraries of GPU and FPGA drivers use different function handling mechanisms so their behaviours are implementation- and vendor-specific. The lesson learned from here is that calling API functions in an interleaved way can overlap the kernel executions on different devices and hence boost the performance.

3) Adjustable ROI size: It is evident that the ROI adjusting scheme can increase the program performance as long as

\begin{table}[h]
\centering
\caption{Detailed information of the test videos}
\begin{tabular}{|l|c|c|l|}
\hline
Video name & Total frames & Resolution & Scenario \\
\hline
cordova1 & 250 & 640×480 & bus view \\
cordova2 & 406 & 640×480 & blur lane \\
ashington1 & 337 & 640×480 & street shade \\
washington2 & 232 & 640×480 & blur lane \\
street & 3056 & 640×480 & street road \\
day_highway & 1718 & 480×360 & high way \\
Frontfacingobstacle & 4601 & 480×360 & crossing lane \\
HighSpeedDrivingShort & 1871 & 1920×1080 & high way \\
clip2 & 1289 & 640×360 & rural \\
clip4 & 899 & 640×360 & dark \\
night_land_car & 4992 & 640×480 & night \\
night_traffic & 2654 & 640×480 & heavy traffic \\
oil_4 & 2287 & 480×320 & broken lane \\
night_4 & 2799 & 640×480 & night highway \\
night_brokenlanes & 1897 & 640×480 & broken lane \\
Weilerhemmen & 4944 & 640×480 & light disturbance \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\caption{Possible conditions of compute unit replication and loop unrolling on the test platform}
\begin{tabular}{|l|c|c|c|c|c|l|c|c|}
\hline
Condition & $\lambda_{RC}$ & $\lambda_{LU}$ & Condition & $\lambda_{RC}$ & $\lambda_{LU}$ \\
\hline
#1 & 1 & 1 & #8 & 1 & 8 \\
#2 & 1 & 2 & #9 & 1 & 9 \\
#3 & 1 & 3 & #10 & 2 & 1 \\
#4 & 1 & 4 & #11 & 2 & 2 \\
#5 & 1 & 5 & #12 & 2 & 3 \\
#6 & 1 & 6 & #13 & 3 & 1 \\
#7 & 1 & 7 & #14 & 3 & - \\
\hline
\end{tabular}
\end{table}
the best lines do not always locate near the border of the initial ROI. The red and black polylines in Figure 5 show the performance gains when running LDA with and without the ROI adjusting scheme. The results indicate that the adjustable ROI size can increase the performance by 12.88% (average) and 21.66% (maximum when FPGA task proportion is 30%).

For brevity, Figure 5 also presents the step-by-step optimization results of the strategies explained in Section IV-B1, IV-B2 and IV-B3. Note that each optimization method is gradually used. From the figure it is observed that compute unit replication and loop unrolling boosts the performance to the largest margin, with an increase by 66.41% (average) and 130.20% (maximum when FPGA task proportion is 80%). Accelerator execution time overlapping further improves the performance by 21.60% (average) and 40.48% (maximum when FPGA task proportion is 20%). In summary, our optimization gains a 2.27x (average) and 2.90x (maximum when FPGA task proportion is 20%). In summary, our optimizations of the strategies explained in Section IV-B1, IV-B2 and IV-B3. Note that each optimization method is gradually used. From the figure it is observed that compute unit replication and loop unrolling boosts the performance to the largest margin, with an increase by 66.41% (average) and 130.20% (maximum when FPGA task proportion is 80%). Accelerator execution time overlapping further improves the performance by 21.60% (average) and 40.48% (maximum when FPGA task proportion is 20%).

In summary, our optimizations of the strategies explained in Section IV-B1, IV-B2 and IV-B3. Note that each optimization method is gradually used. From the figure it is observed that compute unit replication and loop unrolling boosts the performance to the largest margin, with an increase by 66.41% (average) and 130.20% (maximum when FPGA task proportion is 80%). Accelerator execution time overlapping further improves the performance by 21.60% (average) and 40.48% (maximum when FPGA task proportion is 20%).

VI. CONCLUSION AND FUTURE WORK

This paper investigates the optimization of the heterogeneous executions of an OpenCL-based LDA. The program is first profiled to locate the performance bottlenecks of the implementation. Then three optimization strategies are used to accelerate the application, from the perspective of kernel, host and the algorithm itself. Compute unit replication and loop unrolling is performed on the kernel code and the kernel function calls are scheduled in an interleaved way to hidden the on-device execution time. Finally the ROI size is tuned during every iteration of the frame processing to speed up the overall program performance. Experimental results indicate that our three methods can effectively reduce the time consumption and on average the optimization implementation increases the performance by 127% when compared with the naive parallel LDA application.

Our future work is to use the optimization methods to speed up more ADAS applications and give a general optimization paradigm. Further research is also necessary to investigate the workload distribution among the accelerators so that load balance can be achieved to obtain an optimal heterogeneous execution.

ACKNOWLEDGMENT

This work is supported in part by the scholarship from China Scholarship Council (CSC) under the Grant Number 201506270152.

REFERENCES


