1-1-2015

A GPU Powered Mobile AR Navigation System

Mengshen Zhao
University of Mississippi

A GPU POWERED MOBILE AR NAVIGATION SYSTEM

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Mengshen Zhao

May 2015

ABSTRACT

This thesis presents a real-time Augmented Reality Navigation System(ARNavi) on Android smartphone that leverages the parallel computing power of mobile GPUs. Unlike conventional navigation systems, our proposed ARNavi augments navigation information onto real scene video streaming from device camera in real-time. To achieve this goal, we implement and accelerate compute intensive part of applications using OpenCL on GPU integrated on mobile Application Processor (AP). The contributions of this thesis are three-fold. First, we propose new lane detection algorithm and prediction mechanism based on geometric coordinates. The result shows that these two algorithms are fast and accurate. Second, we port and accelerate a complete application on mobile AP. By taking advantage of CPU-GPU heterogeneous computing techniques, we achieve more than 2.6 times performance boost compared to CPU only version. Lastly, we successfully integrate OpenCL and OpenCV on Android platform.

ACKNOWLEDGEMENTS

I would like to express my appreciation to my advisory Dr. Byunghyun Jang for his continuous support of my graduate study and research. I thank him for giving me an opportunity to be part of the HEROES research team and work under his supervision. I thank him for his great support, encouragement, suggestions and insightful comments on my research. On a personal note, Dr. Jang inspired me by his hardworking, passionate attitude and caring to all of his students. It was my honor to work with Dr. Jang.

My gratitude also goes to my thesis committee, Dr. Dawn E. Wilkins and Dr. Yixin Chen. I thank them for their great support and invaluable advice. I'm thankful to Dr. Wilkins for her great suggestions on mobile application development and related topics. I'm also thankful to Dr. Yixin Chen for giving me solid background of algorithms. Without the knowledge of algorithm, main algorithms in this thesis cannot be established. They are both excellent teachers.

Finally, many thanks go to the members of HEROES research team. I thank them for their continuous support and friendly suggestions. It was my great pleasure to work with them.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

GPS navigation is one of the most useful applications in our daily lives. It provides directions and other information while user traveling from one place to another. Most GPS navigators in the market run on mobile platforms such as smart phone. Given two locations, the traveling route is calculated and displayed on the traditional map on device's screen. Under most circumstances, this is sufficient to provide the user useful information. However, if the user does not have time to take a glance on the screen due to increasing traffics, or the user is not able to relate the animation graph with the real-life view, the user might get confused and miss making a turn or even get lost.

To improve user's experience, we propose a new navigation system called Real-Time Virtual Reality Navigation System(ARNavi). Different from conventional navigators' interface, ARNavi projects navigation information onto real-world camera inputs. The main tasks of this approach can be divided into two parts. The first part is accelerating lane detection and reconstruction using OpenCL and the other part is implementing our GPU based point-to-point coordinates mapping algorithm. Both methods are inspired by the GPU's massive thread execution model.

In our proposed ARNavi system, road lanes and driving directions as well as other information such as road name are highlighted and remapped to real-world image. All

necessary data are fetched from device's GPS module and Google Map. By fully exploiting the parallel computing power of GPU, our implementation delivers the real-time processing speed on mobile devices. On Samsung Galaxy Note 4, we achieve around 20 frame per second (FPS) with the help from its Adreno 805 GPU.

This thesis is organized as follows. Chapter 2 provides some necessary backgrounds including OpenCV, common image processing techniques in lane detection and GPGPU (General Purpose Computing on GPUs) programming techniques. Chapter 3 presents our approach to lane detection and reconstruction. In Section 4, we reveal the details of our GPU based mapping algorithm and their implementation. Chapter 5 presents the details of OpenCL implementation on AMD PC GPU. In Chapter 6, we go though the entire porting process of the application on Android platform. Experimental results are presented in Chapter 7. Finally, we conclude the thesis in Chapter 8.

CHAPTER 2

TECHNICAL BACKGROUND

2.1 OpenCV

OpenCV (Open Source Computer Vision) [8] is a open source library that focus on computer vision and machine learning. OpenCV libraries contains more then 2500 state-of-the-art computer vision algorithms such as general image processing, object detection, movement detection, 3D model extraction, etc.

OpenCV is originally written in C and C++. Now, its implementation has been extended to other languages such as C#, Python, Java, Matlab and Perl. OpenCV is a cross-platform interface and can run on various platforms such as Windows, OS X, Linux, embedded Linux, iOS, and Android.

Since 2010, a GPU based interface has been initiated with CUDA [3], which is a heterogeneous programming framework for NVIDIA GPU. Heterogeneous programming language allows programmers to program both GPU and CPU and distribute workloads between two devices and execute them in parallel. The parallelized OpenCV algorithms are highly optimized for GPU thus their performance is significantly increased. Another integration with OpenCL is released on 2012. OpenCL is not only the parallel programming language for AMD's GPU but also the standard for Apple, ARM, Qualcomm, Imagination Tech-

nologies and others hardware vendors. With the high portability of OpenCL, the OpenCL ported OpenCV library has became a truly high performance computer vision libraries across various platforms.

The OpenCV on Android platform is not promising as other platforms. As of the writing of this thesis, there is no attempt to utilize the OpenCL functionality on Android programming. However, we successfully enabled and applied the OpenCV's OpenCL modules to our Android application. More details will be presented in Chapter 5.

2.2 Lane Detection

Road lane detection is an important image processing technique in most vision based driving assistance system. Because of high demand of this technique, it has been well studied and implemented for many systems.

Entire lane detection consists of three stages: detection, tracking, and reconstruction. Most applications include only tracking and reconstructions and they take a similar routine in these two processes. Nowadays, most lane detection algorithms are derivations of edge detection and color segmentation. Some are even integrated with other technologies such as laser and ultrasonic systems.

Canny edge detection is the most widely used algorithm in detection stage because of its effectiveness and processing speed. However, if road has no painted lanes or paint is not very clear, this method performs poorly. Color segmentation [12] takes advantage of high color contrast between road and painted lane. By filtering each frame by a threshold in color space, lanes are separated from the image. A huge drawback of this method is processing speed.

A spline model is selected in most reconstruction stages due to its simplicity. With three or more sets of control points, the spline model is able to build any straight or curve lines. This is a huge advantage over other models. Works like [15], [22] and [23] are developed

based on this model. This model can be further derived into two subbranches, mono vision model, and stereo (or even multiple) vision model [16], [10]. Although this model has a huge advantage in running time, it has two main drawbacks. It has lower noise tolerance and inability of detect bashed lanes. Both of them are caused by its control points model. The first situation will cause an incorrect result if noise are selected. The other one will build a continuous lane instead of dashed lane.

Most processes we mentioned so far are related to image processing. As we know, this kind of algorithms usually requires better hardware and processor due to its high demand in computing power. Since image processing deals with data in pixel level, it could lead to an extreme slowness with the combination of high resolution image and complicated algorithms. Current solutions to this problem are either improving algorithm or updating hardwares.

2.3 Heterogeneous Computing

2.3.1 GPGPU

GPGPU stands for General-Purpose Computing on Graphic Processing Units. It is a framework that allows programmers to access many-cores GPU hardware and perform highly parallel computations on it. GPU consists of thousands of cores, each of which is capable of performing ALU operations independently. By moving compute intensive parts of applications to GPU, the result usually leads to a promising speed gain.

GPGPU is also known as Heterogeneous Computing. As the name indicates, GPU does not work as a standalone device, but it collaborate with CPU. To communicate with GPU, CPU has to evoke special APIs to get access and control the device. Such APIs are designed specifically for hardwares and varies from vendors to vendors. The most popular and widely used programming languages are OpenCL and CUDA. CUDA is a proprietary API that developed by Nvidia, thus it works only on NVIDIA GPUs. OpenCL is developed by Khronos group and gradually became the dominant programming language in industry

because of its cross platform capability. Many hardware vendors including some leading manufacturers such as AMD, Intel, Qualcomm, ARM and Imagination Technology have already adapted OpenCL.

2.3.2 OpenCL

There are challenges when programming with OpenCL. First, GPU architecture is quite different from CPU, thus programmers have to learn how to program in a parallel fashion. Second, all preparation works have to be done manually. From initializing hardware to data transferring between host and device, programmers have to follow certain routines and specify the configuration explicitly in each step. One small glitch could result in a failure in launching OpenCL program. Third, programming itself is difficult. Aside from the syntax itself, some implementations depend on the hardware architecture. The programmer is recommend to review the hardware programming guide beforehand in case the hardware vendor changes some functionality from their implementation. Fourth, the optimization level is highly dependent on the characteristics of application. Lastly, debugging is challenging. The only debug method in OpenCL kernel is *printf* function. This limitation makes the debugging process extremely time consuming especially when the data is huge.

Despite of these difficulties, more researchers and programmers use GPU computing because of the huge payback. In general, it is not uncommon that GPU ported program is around 40 to 80 times faster than CPU version. For highly optimized algorithms, acceleration can reach up to 100 to 200 times.

2.3.3 Basic Implementation of OpenCL

In general, OpenCL program is divided into four parts: initialization, data write to device memory, kernel execution, data read back from the device. Among these processes, the kernel is only part that executes on the GPU. In OpenCL's terminology, the code that executes on CPU is called host code, the code that executes on GPU is called a kernel. Kernel

code is written in OpenCL and compiled at runtime. A kernel can also be pre-compiled as an option.

Before and after launching kernel, most work must be done on host machine. It is a tedious work and has to be done manually by programmer. This process includes:

- **clGetPlatformIDs()**

  Specify the platform. Usually there is only one platform which is the host machine.

- **clGetDeviceIDs();**

  Specify GPU device that the program executes on.

- **clCreateContext()**

  Creates a context. The rest of the OpenCL tasks (creating devices and memory, compiling and running programs) is performed within this context.

- **clCreateBuffer()**

  Creates a buffer object that associates which the data will be passed to the device.

- **clCreateProgramWithSource()**

  Creates a program object for a context, and loads the source code specified by the text strings in a string array into the program object. To load a precompiled binary kernel, use $clCreateProgramWithBinary()$ instead.

- **clBuildProgram()**

  Builds (compiles and links) a program executable from the program source or binary.

- **clCreateKernel()**

  Creates a kernel object.

- **clSetKernelArg()** Passes the value of a specific argument of a kernel. One kernel usually has multiple arguments.

- **clCreateCommandQueue()**

  Create a common queue. All computations preformed on GPU are done using a command queue, which is a virtual interface for the device. It is created with the associated context. One program can has multiple command queues.

- **clEnqueueWriteBuffer()**

  Enqueue a command to copy data from host memory to device memory.

- **clEnqueueNDRangeKernel()**

  Enqueue a command to execute a kernel on a device. Programmer specifier the working dimension and workgroup size here.

- **clEnqueueReadBuffer()**

  Enqueue a command to copy data from device memory to host memory.

CHAPTER 3

LANE DETECTION AND RECONSTRUCTION

3.1 Camera calibration

One of the main purpose of camera calibration is to measure the distortion parameters inherited from the hardware itself. The distribution is caused by the misalignment during the assembling process. In computer vision area, a common technique to measure the intrinsics parameters is to take advantage of the chess board patter. As the name indicates, the chessboard pattern is a bunch of black and white squares that arranged as a chessboard. The number of the chessboard is flexible as long as the number of rows and columns are different. By counting the number of corners of the rectangle chessboard, the calibration function is aware of the position of the board. Since the pattern are predefined, by matching the corners while board placing at different position, the distortion coefficients can be calculated. The coefficient is used to eliminate distortion effect. For better result, the board need to be test at at least 10 positions.

The distortion is much obvious in cheap cameras because the quality of the lenses are very low. The misalignment during assemble process is another major reason of the

distortion. Even if with high quality lenses. The camera calibration is designed to minimize these effects and restore the images as they were in real-life. The detail of this method is well explained in Chapter 11 of [11], [19] and [24]. Under most circumstance, this is one-time task since the hardware does not change over time.

Our test platform is Samsung Galaxy Note 4, which has a high quality camera with various built-in optimizations. Including Back-Illuminated Sensor (BSI), CMOS image sensor, autofocus, and digital image stabilization, etc. According to our test result, the image obtained by this built-in camera does not have any distortion or deformation. Lines and angles in the original image are well preserved. Although the camera on this device and most smart-phones are well built, the camera calibration process is preserved in case the application runs on old generation hardware.

3.2 Perspective Transformation

In this process, three dimensional view image is transformed and remapped to a two dimensional view image. To be more specific, a bird-eye's view. In this view of the world, we can easily find the relationship between the transformed image and the conventional GPS view. These two views assume that the user is looking down on top of road. This transformation is appreciated since it makes subsequent processes much easier because the data from the GPS application will be mapped to the bird view image.

To do perspective removal, we need to apply a 3 by 3 transform matrix call Homography matrix to the 3D view image. This matrix is denoted as H. The value of this matrix varies since it depend on the position of the camera. However, once the position of the camera is fixed, there is need to do this process again. To obtain a correct matrix, we perform a similar procedure in previous step. First, we place a known size chessboard in front of our test vehicle, then take a picture from the camera that mounted on the vehicle. Since the size of the chessboard predefined, the number of corners and the real-world position of each chess

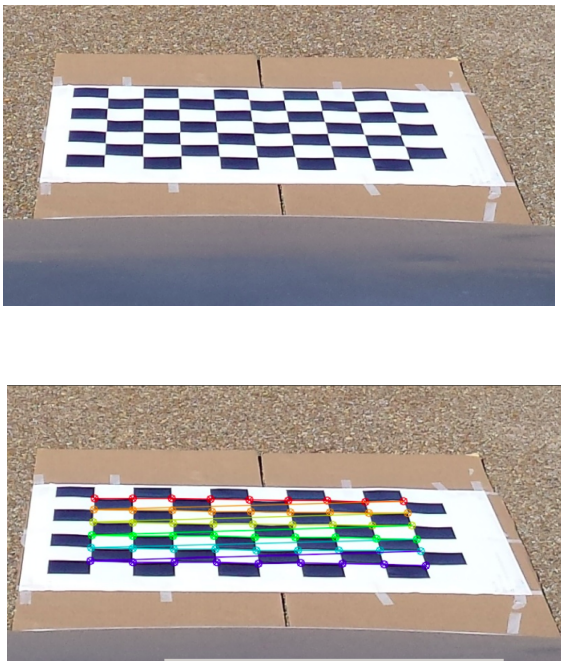

Figure 3.1. Chessboard, detected corners

box is known. By comparing the difference between real data and predefined data, all values in the H matrix can be calculated. [24] and Chapter 12 of [11] have a detailed description of how to obtain the homography matrix.

In our implementation, we only take the second half of the image as input because it's easier to calculate and close enough to the horizon vanish point. The input image and output image are shown in Figure 3.1.

The homography matrix is applied to the target image to remove the 3D perspective. In this process, each pixel in the 3D plane will be mapped into a 2D plane based on the value of the matrix. As a result, a 2D plane image is returned. See an example in Figure 3.2.

3.3 Pre-processing

The following list contains the OpenCV APIs in the pre-processing:

- *Rect*();

For selecting certain part of the input image. In our application, we use this function

to select the lower half of each frame as the input image.

- *warpPerspective*();

  For perspective transformation. This part takes the homography matrix and applies it to the input image to get a perspective removed image. Before out the final image, we this this function with the inversed homography matrix to restore the Perspective image.

- *cvtColor*();

  This function is used to convert three channel RGB image to a hue image.

- *inRange*();

  This function works as a color filter. Any pixel has the color outside the range will be removed.

- *threshold*();

  This function takes the result from *inRange*() and filter the results according to a threshold. By doing these, we can remove more noises from previous process. After this process, the result is a pure binary image with values 0 or 255.

- *GaussianBlur*();

  Used to blur the boundary of small object in the object in the input image. These small objects are usually consider as noises comparing to the target image. This process usually placed before the Canny edge detection to reduce some noise edges.

- *Canny*();

  This function is mainly used to detect edges in a image.

As a result of preprocessing, the RGB image is transformed into a binary image with detected edges. It's the input of our Lane Detection algorithm.

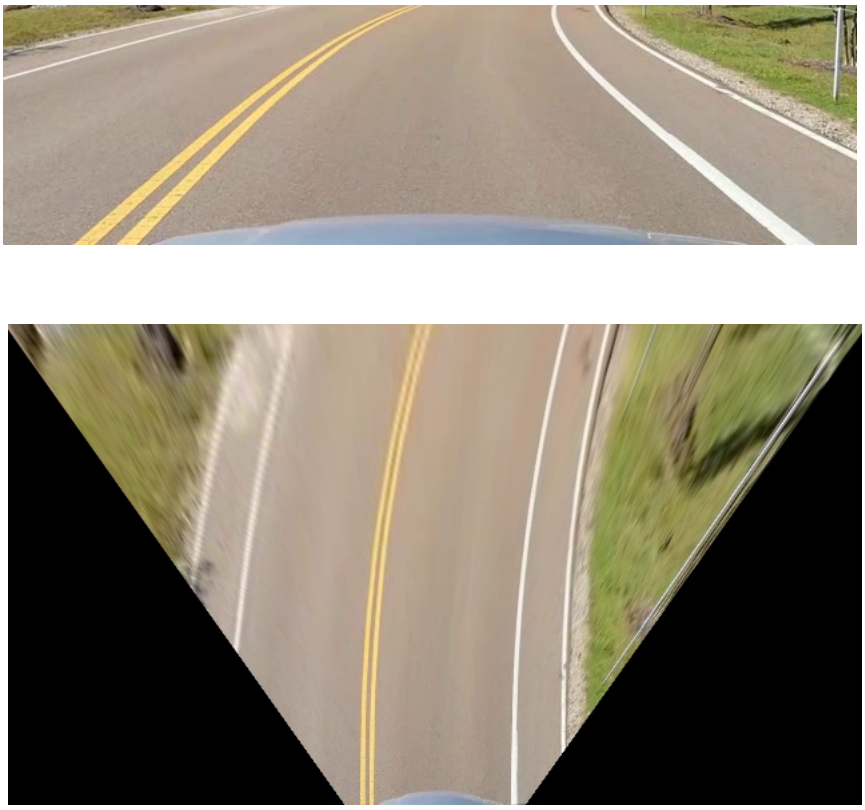

Figure 3.2. Perspective transformation

3.4 Lane Detection

In the context of traffic control, a lane is part of carriageway (roadway) that is designated for use by a single line of vehicles, to control and guide drivers and reduce traffic conflicts.

The goal of lane detection is to segment and highlight the lanes in the image obtained by the on-board camera. This process is critical to all Driver Assistance System (DAS) since lanes are the most important and useful signs on the road. With the lane detection, the DAS system is bale to notify the driver when getting closer to the edge of the road, or check the surrounding of the car when it detects an intention of changing lane. In our case, the detected lanes are used as guiding lines in the virtual reality navigation information mapping process.

Most lane detection algorithms are based on feature extraction [20], color segmenta-

tion [12][18] and Hough transformation[14] [13] [21]. These algorithm are usually followed by another modeling algorithm to reconstruct the lane, such as spline model. The correctness of these algorithms have been proved to be high enough under most conditions. However, their execution time are not as appealing as their correctness. On a high end CPU, the performance can only reach around 15 FPS on average ([21] [15] [10]), which is far less than what real-time processing requires, around 25-30 FPS. The slowness is caused by algorithms themselves. As well known, image processing algorithms are computation intensive algorithms. They perform operations at pixel level, which introduces a huge input data. What's more is that these algorithms usually use a scan window to scan though the image and bring the problem to a matrix level, which introduces more complexity. Some research groups realized this problem and implemented their algorithms based on GPGPU technique and reached the real-time speed by leveraging the parallel computing power of GPU.

To achieve the real-time processing speed, we designed a light-weight lane detection algorithm based on a basic property of the lanes, lanes are parallel and symmetric lines. By taking advantage of this characteristic, two lines are said to be a lane if it's the longest parallel lines through the entire image. This conclusion is based on the result of perspective transformation, see Figure-3.2. It might not seem correct from the driver's perspective, but it's the nature property of a road lane in real world. This algorithm greatly reduced the total operations by looping through each pixel only once.

Before we go though the algorithm, we introduce the terminologies that being used in this algorithm.

dist The distance between current non-zero value and previous non-zero value in each row of the image

order The order of current non-zero pixel in each row of the image. This value will help to eliminate some useless values in order to reduce the processing time.

node A structure that stores all information of a pixels. Each *node* has four values including

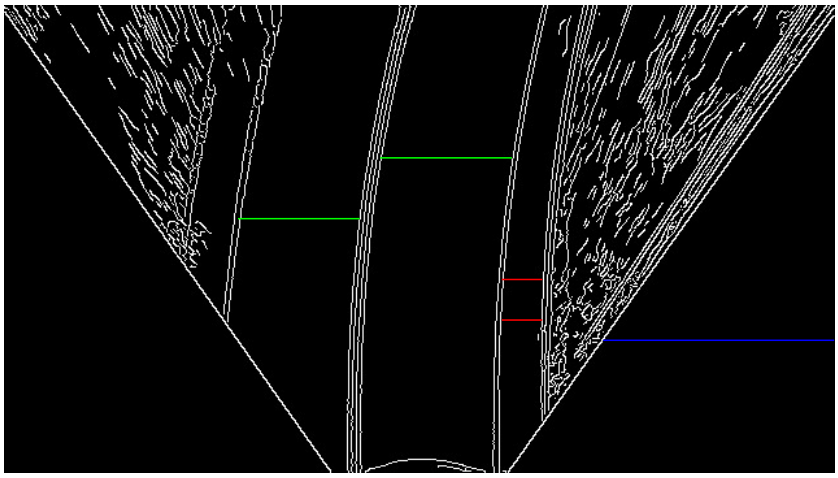

Figure 3.3. Distances between adjacent pixels

x,y coordinates, $dist$ and $order$.

$MFAS$ Most Frequently Appear Segment. MFAS is the accumulation of each $dist$ in the entire frame.

The lane detection algorithm consists of three steps. $Dist$ takes the result of pre-processing (Figure-6.5) as input and calculates the distance and order between two adjacent white pixels in each row. Examples of segments are shown as highlighted lines in Figure-6.5. Then, the $Histogram()$ function counts and returns the frequency of all segments that appeared in the image. At last, the $MaxSubarry()$ function analyzes the data from $Histogram()$ and returns the $MFAD$. The pseudo-code of each steps are listed in Algorithm 1 , Algorithm 2 and Algorithm 3 respectively.

3.4.1 Distance Between Two Points

The idea behind $Dist()$ function is trivial. While iterating though the entire binary image, the value of each pixel in each row will be inspected. Since the result of edge detection

is a binary image, the pixels has only two values, either 0(black) or 255(white). Whenever encounter a white pixel, the distance from previous white pixel to current white pixel will be calculated by subtraction. The position of previous pixel is denoted as $prePix$. The value of this distance is denoted as $dist$. $dist$ is stored at the corresponding position in the $nodeMat$, which is a 2D array which has the same size as the edge image. Then the value of $prePix$ is updated to the position of current white pixel. At last, the value of $order$ is incremented and stored in corresponding position of the $nodeMat$. The value of $dist, prePix, oder$ will be set to zero when reach the end of each row. This part is presented from line 8 to line 13 in Algorithm-1.

Algorithm 1 Dist

```
1: Input: Binary image I
2: Output: A node matrix nodeMat[row][cols]
3: for i to row do
4:     dist ← 0
5:     prePix ← 0
6:     order ← 0
7:     for J to col do
8:         if I(x, y) ≠ (0) then
9:             dist ← j − prePix
10:            nodeMat(i, j).dist ← dist
11:            prePix ← j
12:            order + +
13:            nodeMat(i, j).order ← order
14:        end if
15:    end for
16: end for
```

3.4.2 *Histogram*

The histogram is a simple yet efficient method to get the distribution of numerical data in a certain range. The histogram function iterates though the $nodeMat$ generated by the $Dist$ function. Each $dist$ is placed in corresponding bin. The bin spends from 0 to 640, representing the minimum and maximum length that can be found in a single row. The

details are shown in Algorithm-2. A visualized result of $Histogram$ is shown in Figure-6.2. In this particular example, the target distances is highlighted in the red box. In United States, the width of a road is between 2.7 meters to 4.9 meters. With the homography we obtained, 1 meter is equal to 30 pixels. Thus, the width of the road in United Stated varies from 81 to 147 pixels. In other words, if the length of a piece of a segment is 81 to 147 pixels, it's likely to be a part of the lane. Based on this fact, we can discard any values that outside this range.

Algorithm 2 Histogram

```
1: Input: A node matrix nodeMat[row][cols]
2: //The maximum length of each row is 640
3: Output: int dist_Hist[640]
4: for i to row do
5:     for j to col do
6:         dist_Hist[nodeMat[i][j]] ++
7:     end for
8: end for
```

3.4.2.1 *MaxSubarray*

The last part of lane detection is to select target range from the result of$Histogram()$. This algorithm is an optimized max sub array algorithm which runs in linear time $\mathcal{O}(n)$.

$MaxSunarray()$ takes the *dist_hist* array and its size as input. Values in the *dist_hist* is added to the $temp_sum$ one by one and compared with current *sum*. If the *temp_sum* is lager than the current sum, the value of current *sum* will be updated, which means the data is contiguous. Thus, the value of the temporary end point $temp_end$ will be increased by 1 (line 4 to 7).

On the other hand, if the *temp_sum* is equal (the minimum distance is 0) to the current sum, the value of current *sum* will be set to 0. Which means the data is being c̈utb̈y 0 and this piece of data loss its continuity. Since current value is zero, we update both temporary staring point *temp_start* and end point *temp_end* to next position and start to

Algorithm 3 MaxSubarray

```
1: Input: int dist_hist[640], size of dist_hist
2: Output: range.start, range.end
3: for m to size do
4:     temp_sum ← sum + dist_hist[m]
5:     if temp_sum > sum then
6:         sum ← sum + dist_hist[m]
7:         temp_end + +;
8:     else if temp_sum = sum then
9:         sum ← 0
10:        temp_start ← temp_end + 1
11:        temp_end ← temp_end + 1
12:    end if
13:    max_sum ← max(sum, max_sum)
14:    if sum ≥ max_sum then
15:        range.start ← temp_start
16:        range.end ← temp_end
17:    end if
18: end for
```

look for new contiguous data (line 8 to 11).

Finally, we compare and assign the lager value of *sum* and current *max_sum* to the *max_sum*. If the *sum* is larger, we will return current *temp_start* and *temp_end* as final *range.start* and *range.end* (line 13 to 16).

At this point, we have a $nodeMat$ and an int2 variable *range* indicates the range of the target distance.

3.5 Lane Reconstruction

3.5.1 Lane Reconstruction

In this process, the 2D perspective image will be restored to 3D perspective.

First we create a new OpenCV matrix $lane_frame$ that has same size as $nodeMat$. Since the lanes should be distinguishable from background image, they are highlighted in sharp color, in our case, we use pure green. In order to display green color, the $lane_frame$

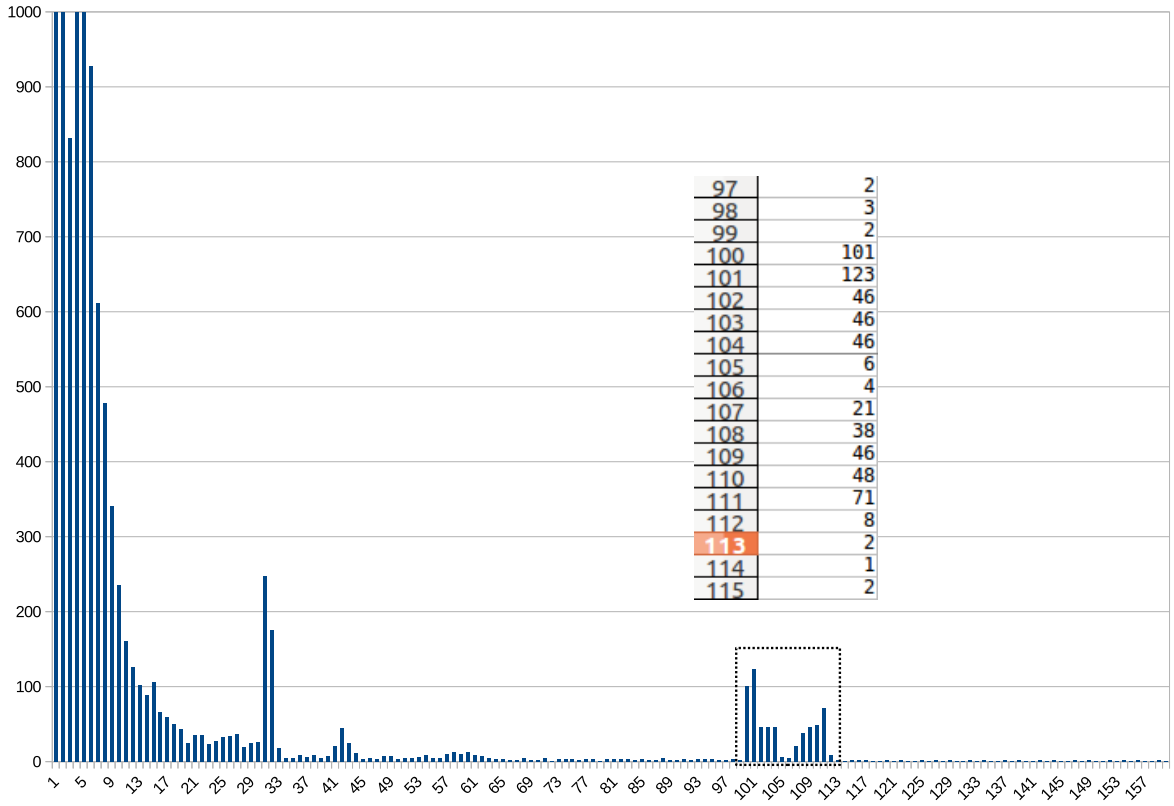

Figure 3.4. Histogram and range of MFAS

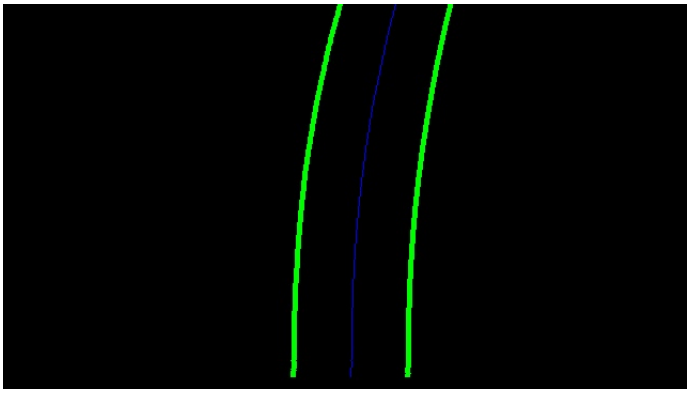

Figure 3.5. Reconstructed lane

is declared as 3 channels RGB matrix.

A simple approach is used to reconstruct the detected lanes in the RGB matrix. While iterating though $nodeMat$, whenever the value of the $dist$ falls into the $range$, the color of the pixel at that position is changed to green. When the entire image is processed, we will have a high lighted lane. Since the lanes are symmetric, the other half of the image is duplicated according to the value of $dist$. At this point, the Lane Detection process is finished. As a result, both edge of the lane is constructed. See Figure-3.5.

3.5.2 Remapping

This step is simply perform a inversed perspective transformation on the lane image (Figure-3.6). *OpenCV*'s API provides a very handy function to convert the homography matrix H to inversed matrix H^{-1}. The inverted perspective matrix is computed by an OpenCV function $inv()$ automatically. The inverted matrix is stored in $Math$. The process of perspective restoring is identical to the perspective removing process in section 3.2. The result of this process is overlap with the original image to generate the image with detected lanes. Result is shown in Figure-3.7.

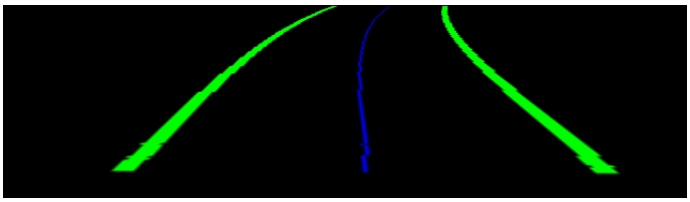

Figure 3.6. Reversed perspective transformation.

Figure 3.7. Output image.

CHAPTER 4

GPS COORDINATE MATCHING AND AR MAPPING

There are five steps in this process: data fetching, prediction points calculation, real-world coordinate calculation, coordinate matching and drawing.

The goal of the first step is to fetch the data from GPS module. The second and third steps are responsible for calculating two sets of data, predicted geometric coordinates and real world geometric coordinates. The fourth process is to compare and find the matched points from from previous step. Finally, the matched points are highlighted and overlapped on top of the input frame in order to indicate the directional information.

4.1 Fetching Data

4.1.1 Fetching Data From GPS Device

GPS receivers are character devices with embedded GPS chips. Once connected, it searches for satellites signals and returns a set of data that obtained from satellites.

Depending on application, the data we need are different. For a conventional navigation system, only few data are required including the current location, moving direction

and speed. To filter out the critical information, a parser is usually placed between the GPS device and the application.

There are bunch of open source GPS libraries available on the Linux system. All of them included the functionality for basic parsing task. However, these parser are complicated and returns more data then we need. Since the processing speed is critical in our case, we implemented a light weight parser to minimize the processing time.

4.1.2 GPS Data Format

There are two standards used in modern GPS devices, NMEA(National Marine Electronics Association) protocol [6] [4] [7] and SiRF protocol [17] [9]. The NMEA protocol generates human-readable character string sentences while the SiRF protocol returns binary strings.

4.1.3 Sentences

NEMA data are series of strings. A single line of string is a sentence. A sentence containing various information such as geometric coordinates, data and time, activated satellites and signal condition, etc. A sentence starts with an unique identifier which indicates the information that this sentence carried. There are more than 50 GPS identifiers. However, not all sentences are available on a single GPS device. The functionality of GPS device varies depending on the chips it used and purpose.

For most daily used devices, the following sentence are mandatory standard that required to be included in the GPS chip:

- *GPGGA*

  Global positioning system fix data.

- *GPGSA*

  GPS dilution of precision and active satellites.

- *GPRMC*

  Recommended minimum specific GPS/Transit data.

- *GPGSV*

  GPS Satellites in view.

4.1.4 GPGGA

GPGGA stands for *Global Positioning System Fix Data*. In most circumstances, this information provided by this sentence is far more then enough. The most important information is the coordinates of current global position, which is our main purpose of using a GPS device. A GPGGA sentence is construct as following format:

$GPGGA,hhmmss.ss,llll.ll,a,yyyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx

hhmmss.ss = UTC of position

llll.ll = latitude of position

a = N or S

yyyyy.yy = Longitude of position

a = E or W

x = GPS Quality indicator (0=no fix, 1=GPS fix, 2=Dif. GPS fix)

xx = number of satellites in use

x.x = horizontal dilution of precision

x.x = Antenna altitude above mean-sea-level

M = units of antenna altitude, meters

x.x = Geoidal separation

M = units of geoidal separation, meters

x.x = Age of Differential GPS data (seconds)

xxxx = Differential reference station ID

Based to the format, the sentence return from the device can be dissemble and explained easily. However, we are interested only in some of the data such as *latitude of position*, *Longitude of position* and *which hemisphere (N-S, E-W) we are currently at*. The sentence listed below is the output of a real GPS device (i.e., GlobalSat BU-353).

$GPGGA,193040.660,3421.9688,N,08932.2571,W,0,00,,163.0,M,-29.3,M,,0000*45

By filtering out the unnecessary information, this sentence can be explained as follow: Current location is located at:
Northern Latitude (N): 3421.9688
Western Longitude (W): 8932.2571

4.1.5 Sentence Parsing

Parsing is an easy and efficient solution to separate keywords from context. The idea behind our sentence parser is quite simple, whenever a newline is returned by the GPS driver, the parser looks for the keyword GPGGA. If the keyword is found, the entire sentence is parsed to get target values.

In order to achieve the maximum parsing speed, this application has a very unique parsing algorithm. Because the GPGGA is the only sentence the application needs, the parsing processing are optimized to get this specific sentence only. Among the identifiers, the GPGGA is the only one that has a character 'G' at the fourth position. Thus, only the fourth character will be examined during the identifier parsing stage. This strategy guarantees the parsing time is minimum. See Algorithm 4 for details.

Algorithm 4 GPS Sentence Parser

```
1: Input: Character stream line form GPS device
2: Output: A gpsObject with latitude and longitude
3: while line is not empty do
4:     if line[4] =' G' then //Forth char 'G' in GPGGA
5:         for i to 7 do
6:             if i < 2 then
7:                 LAT[i] ← line[18 + i]  //Copy 18 to 19 char
8:             end if
9:             if i < 3 then
10:                LON[i] ← line[30 + i] //Copy 30 to 32 char
11:            end if
12:            lat[i] ← line[20 + i] //Copy 20 to 26 char
13:            lon[i] ← line[33 + i] //Copy 33 to 39 char
14:        end for
15:        NS ← line[28]
16:        WE ← line[41]
17:
18:        if NS ==' N' then
19:            gpsObject.Lat = LAT + lat/60
20:        else
21:            gpsObject.Lat = −(LAT + lat/60)
22:        end if
23:
24:        if WE ==' N' then
25:            gpsObject.Lon = LON + lat/60
26:        else
27:            gpsObject.Llon = −(LON + lat/60)
28:        end if
          return gpsObject
29:    end if
30: end while
```

4.1.6 Difficulties

After applying this parsing function to the application, it leads to a interesting result. Even though the parsing time is around 50 ms, the entire application freeze for 0.5 to 1 second repeatedly. The reason behind it is interesting. It's caused by the refreshing rate of the hardware. According to the official manual[1] of our GPS device, the refreshing rate of *GPGGA* is around 1Hz.

Since the GPGGA sentence refreshes around every 1 second, before the parser gets the content, the program will hold and the following code is halt until the process is finished.

By leveraging the multi-threading technique, this problem is solved with smooth and ease. The technique we used here is to assign another thread to the parser, then detach it from the main thread and execute the parser function only. Detach means separating a thread from the main context and let it execute freely. All detached threads run without effecting each other. The interrupt still exist but does not interfering the main thread.

4.2 Fetching Data From GooGle Data Base

There are two purpose of fetching data from a map data base. The first purpose is to get the starting point and destination points of our route. The other one is to get the waypoints between them. The starting points, destination point and waypoints are geometric coordinates used to represent a certain place on a map application.

4.2.1 Google Service

Google Map is the most widely used map application across all platforms. In order to access Google's data base, the developers are required to register for a google account for the **Google Directions API Service**. Depending on the account type, the service could be free or expensive. The free API allows 2,500 direction requests per 24 hours with 8 waypoints per request. The other license, API For Work Customer, however, allows 100,000

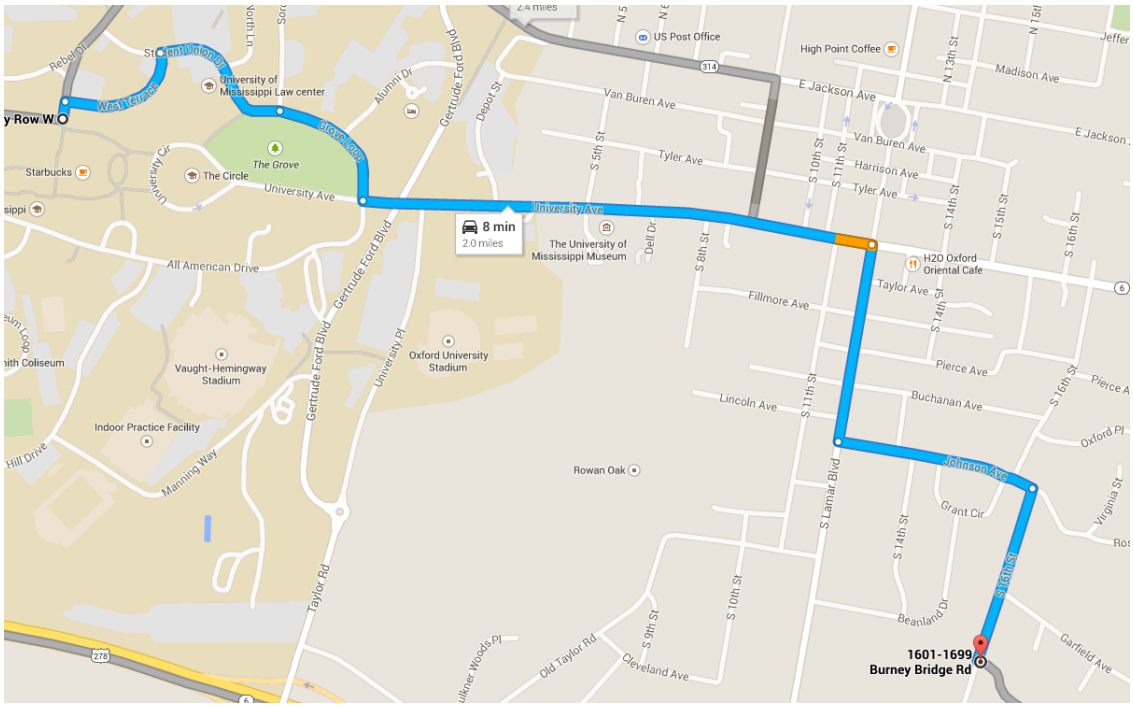

Figure 4.1. Waypoints in Google Map.

directions requests per 24 hours with 23 waypoints. This license costs more than a thousand dollar because it's for the commercial use. To avoid the high cost of using this service, a workaround is applied to get the required data. The process is discussed in section 4.2.2.

4.2.2 Waypoints

When navigating from place A to B on a map application, a highlight route will be shown. Users will follow this route as a guide when they traveling. If their current locations are away from the route, they adjust their moving direction according to the c̈orrectdirections. This is the basic idea of how navigation software works.

The route are constructed based on serious of points. These points are called *way-points*. Waypoints are distributed on the route unevenly. However, they are still distributed based on some patterns. For example, In figure 4.1 the main waypoints are highlighted at the corners and the joints of the road. This distribution pattern is used to ensure every segment between two adjacent way-points are approximately straight.

4.2.3 Fetching Waypoints

To obtain waypoints, we first specified staring point and destination point in the Google Map website. After a route is generated, we take the Url and put in a GPS analysis website (e.g., GPS Visualizer [2]). All waypoints will be extracted and stored in a plain text form. The number of the waypoints varies depending on the length of the route. It also depending on the shape of the route. The larger the curve, the more the waypoints. This mechanism is to make sure that the route between each pair of waypoints are approximately straight. By calculating the spacial relationship between two waypoints, some hidden information such as moving direction, angle to North are revealed. These data play a critical role in the following process, the prediction point calculation.

To verify the correctness of the extracted waypoints, we manually input the latitude and longitude of each points to the Google Map application. The results are identical.

With the fetched data, it ensures that the CPU and GPU version executes on the same set of data and eliminates the variance caused by different input. It is important to the comparing process.

The extracted waypoints from Figure4.1 are shown in Figure 4.2.

4.3 GeoPoints

Since the waypoints are recorded by satellites, they are consider as the "correct" position of any given location on the earth. For convenience, these points are referred to *GeoPoints*. Recall that after extracting the waypoints from the Google Map, waypoints are being chopped into smaller pieces and each pieces are approximately straight line. By leveraging this property, the distance between two waypoints can be further divided into even smaller pieces. In this application, the minimum distance between two subwaypoints is set to one meter for the reason of accuracy. In any navigation system, the accuracy is the first and foremost requirement.

```
 1 type    latitude        longitude
 2 W       34.366450000    -89.537840000
 3 W       34.356150000    -89.516740000
 4
 5 type    latitude        longitude
 6 T       34.366450000    -89.537840000
 7 T       34.366780000    -89.537780000
 8 T       34.366780000    -89.537780000
 9 T       34.366680000    -89.536740000
10 T       34.366710000    -89.536510000
11 T       34.366870000    -89.536030000
12 T       34.366990000    -89.535830000
13 T       34.367260000    -89.535630000
14 T       34.367490000    -89.535560000
15 T       34.367700000    -89.535590000
16 T       34.367700000    -89.535590000
17 T       34.367780000    -89.534890000
18 T       34.367710000    -89.534540000
19 T       34.367590000    -89.534300000
20 T       34.367420000    -89.534140000
21 T       34.366890000    -89.533860000
22 T       34.366640000    -89.533640000
23 T       34.366590000    -89.533480000
24 T       34.366590000    -89.532860000
25 T       34.366590000    -89.532860000
26 T       34.366450000    -89.532250000
27 T       34.366280000    -89.531750000
28 T       34.366090000    -89.531400000
29 T       34.365820000    -89.531090000
30 T       34.365640000    -89.530960000
31 T       34.364890000    -89.530940000
32 T       34.364890000    -89.530940000
33 T       34.364660000    -89.523470000
34 T       34.364560000    -89.522530000
35 T       34.364060000    -89.519240000
36 T       34.364060000    -89.519240000
37 T       34.360330000    -89.520010000
38 T       34.360330000    -89.520010000
39 T       34.359850000    -89.516760000
40 T       34.359620000    -89.516000000
41 T       34.359440000    -89.515550000
42 T       34.359440000    -89.515550000
43 T       34.356170000    -89.516820000
44 T       34.356170000    -89.516820000
45 T       34.356150000    -89.516740000
```

Figure 4.2. Waypoints From Waypoint Extraction.

In order to get the coordinates of the *GeoPoints*, we have to calculate the angle formed by two directions. The direction form waypoint one to waypoint two and the direction of True North direction. This problem is far more difficult than a basic geometric problem that calculating a coordinate in a 2D plane. The surface of the earth is curved, this problem is much similar to calculating the distance on a sphere. For simplicity, two main waypoints are denoted as WP_1, WP_2. The value of $lat1$, $lon1$, $lat2$, $lon2$ are corresponding latitude and longitude of WP_1, WP_2. See equation 4.1 for details.

$$\theta = \arctan\left(\frac{\cos\left[lat2 \times (\pi/180)\right] \times (lon2 - lon1)}{lat2 - lat1}\right) \times (180/\pi) \tag{4.1}$$

By calculating the distance between WP_1 and WP_2, the number of sub segments is revealed since the length of each segment is 1 meter. The total distance of segment is computed based on equation 4.2. Where $R = 6271(km)$ is the average radius of the Earth, d is the distance between two given waypoints WP_1 and WP_2.

Because the distance between two waypoints are varies, one of the main purpose of calculating d is to dynamically allocate an array that holds the coordinates of all sub-waypoints. The other purpose is notified the for loop outside the function *getGeoPoints*() that how many times it should be repeated.

$$\Delta lat = (lat2 - lat1) \times (\pi/180)$$

$$\Delta lon = (lon2 - lon1) \times (\pi/180)$$

$$Lat1\_Radian = lat1 \times (\pi/180)$$

$$Lat2\_Radian = lat2 \times (\pi/180)$$

$$d = 2 \times R \times \arcsin\sqrt{\left(\sin\frac{\Delta lat}{2}\right)^2 + \cos\left(Lat1\_Radian\right) \times \cos\left(Lat2\_Radian\right) \times \left(\sin\frac{\Delta lon}{2}\right)^2} \tag{4.2}$$

With WP_1, WP_2, θ and the length of segment seg, the final step is to calculate the

coordinates of all sub-waypoints between them. Notice that all adjacent waypoints are on an approximate straight line, it implies that each sub-waypoint has the exact same θ to the True North direction. As you can see in Figure **??**, the calculation is based on some basic geometry properties. Δlat, Δlon are the distances will be added to the WP_1 in order to get the sub-waypoints' coordinates. Notice that in this equation, we are manipulating the coordinates, no the length. On earth surface, 1° change of latitude or longitude results in 111 km change in length. Since the base of our segment is 1, the length of segment is converted in meter by multiplying 0.00001. See equation 4.3 for details.

The results are stored as *GeoCoords* type in an array named $geopoints\_inbetween[]$. Size of $geopoints\_inbetween[]$ is determined by d. Type *GeoCoords* is a user defined structure that contains two float type values, the latitude Lat and the longitude Lon. Please notice that the given example, only works for calculating the sun-waypoints while traveling from low to high latitude and low to high longitude and on Northern Hampshire and on Eastern Hampshire at the same time. The equation 4.3 will has some minor changes if any of these conditions is changed.

$$\begin{aligned} \Delta lon &= \sin(\theta \times (\pi/180)) \times (seg) \times 0.00001 \\ \Delta lat &= \cos(\theta \times (\pi/180)) \times (seg) \times 0.00001 \\ Lat &= lat1 + \Delta lat \\ Lon &= lon1 + \Delta lon \end{aligned} \tag{4.3}$$

Depending on the condition of the road, the distance between two main waypoints are vary. For example, a 5 kmroute on the highway might have only two waypoints since it's straight. In contrast, a 1 km route on urban area might have more than 10 waypoints if it has lost of turns and joints. The matching stage is compares two sets of points, the waypoints and prediction points, and try to find the matching points from them. It will greatly affect the computation time of the matching stage due to the length or the complexity.

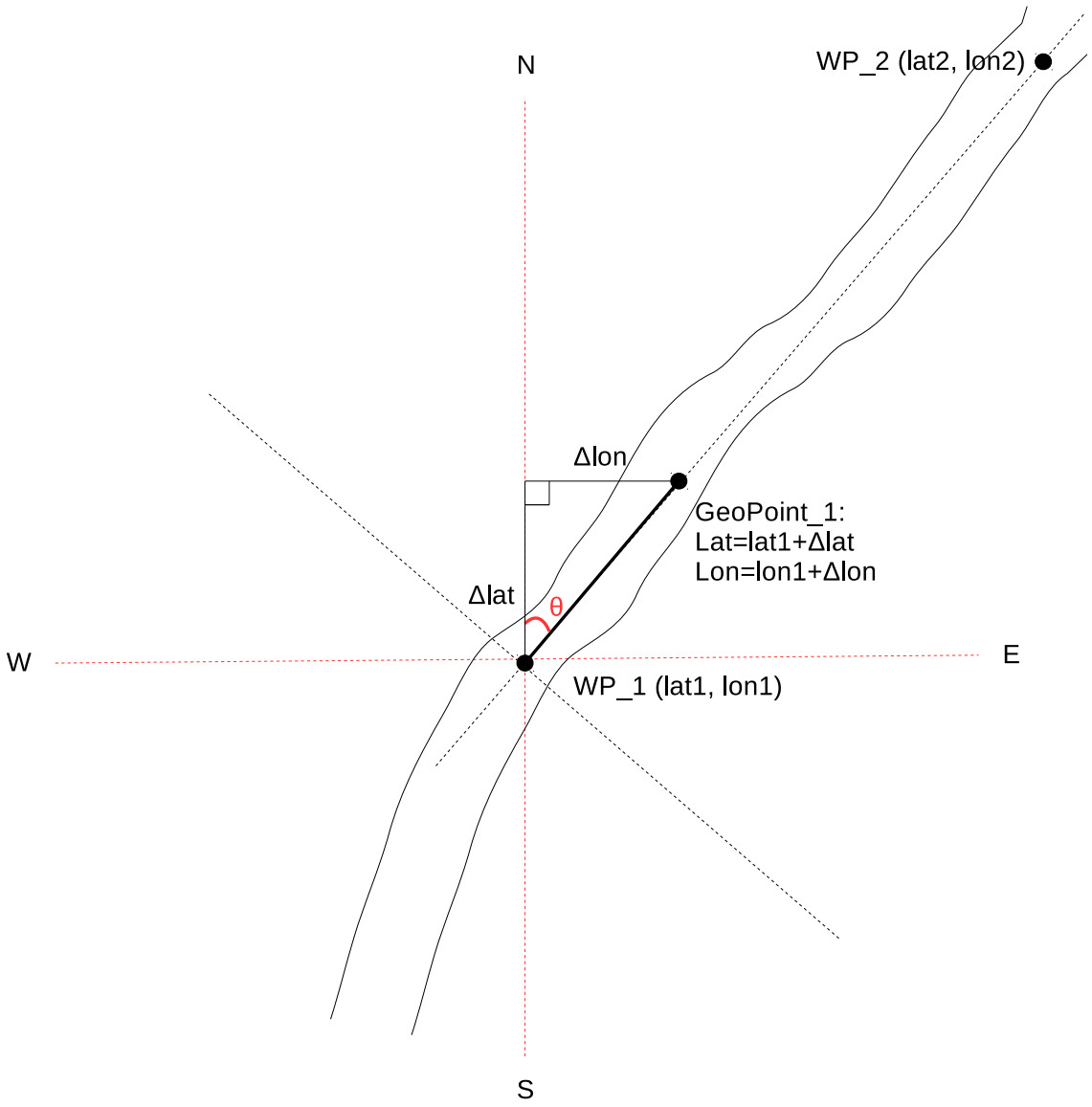

Figure 4.3. Relationship between theta and GeoPoints points.

By limiting the number of sub-waypoints to 1000 meters, the processing speed is relatively fast while guarantees that the hardware acquires enough data to work with. While the car reaches the end of the end of waypoint, it reads another 1000 points for next piece of segment.

4.4 Prediction Points

As its name indicates, the prediction points are geometric points being computed and estimated based on the coordinates of existing point. Different from sub-waypoints, all prediction points are stored as **GeoCoords_with_PixCoords** type. Type **GeoCoords_with_PixCoords**is similar to type **GeoCoords** but with two more integer values that describes its position in the input image. Which is another user defined type. Besides of two floating point coordinates, it has two more integer values, p_x and p_y. These two values are responsible for storing the pixel coordinates of each $predictPoints$. Prediction points change along with the car's moving direction and current position.

As mentioned in Chapter 3, the input frame is divided into two parts along the center line. The lower part is used as input in our application. The distance between the top to the bottom is exactly 12 meters. Thus, the row number of prediction matrix is set to 12. With this setting, The distance between each adjacent predication points on the vertical axis is 1 meter. In a similar way, the prediction points in the same row are placed every 1 meter. There are 15 points in each rows. The prediction points can be visualized as a 12 by 15 matrix that is placed in front of the car. In other word, there are 180 prediction points at any given time.

The entire matrix is divided into three smaller matrices. The first matrix is the center matrix. which has only one column, containing only 12 elements. The rest of the points are naturally grouped as two matrices sits aside of the center matrix. Both of them are 12 by 7 matrices. These matrices are denoted as Center Matrix Mat_C, Left Matrix Mat_L and Right

Matrix Mat_R. See Figure 4.4.

The prediction points are divided in this fashion on purpose, it is much faster to calculate the predictions points resides in the same side then calculate them in a row or column. In the matrix, only the coordinates of point P is known, which is the current location of the car. The coordinates of other points are calculated using prediction algorithm. The prediction algorithm will be explained shortly.

To calculate the coordinates of a prediction point, the first step is to get the angle between these two points and the True North direction. Different from calculating the sub-waypoints, only the staring point's coordinate is known. The destination point, remains unknown since the moving direction of the car is unpredictable. It could be any direction around the car. The next problem is to get the destination point and the angle θ.

The problem can only be solved after the car starts moving. Recall that the $GPGGA$ sentence is refreshed every 1 second and contains the information of current coordinates. Which means as long as the car moves more then 0.5 second or 1 second, there will be two coordinates. The first collected geometric coordinate is considered as the starting point. The second one is the destination point. Once there are two points, the theta will be calculated in the same way as in calculating sub-waypoints. Here, we assumed that the cat in the next second will still move along the same direction.

The $getPrediction\_center()$ function takes the theta as parameter and calculate the fist 12 prediction points in the center matrix. The result is denoted as PP_C. The PP_C will be stored in corresponding position of a 1-D array called $predictionPoints\_all$, which has 180 elements. The purpose of using 1D array is to reduce the complexity of the algorithm. The body of the $getPrediction\_center()$ function is very similar to equation 4.3, the only difference is that in $getPrediction\_center()$, Δlat and Δlon are added to ending waypoint End_P instead of the starting waypoint $Start_P$ since the ending point is the current location.

After the PP_C is calculated, another function $getPrediction\_right()$ takes the PP_C

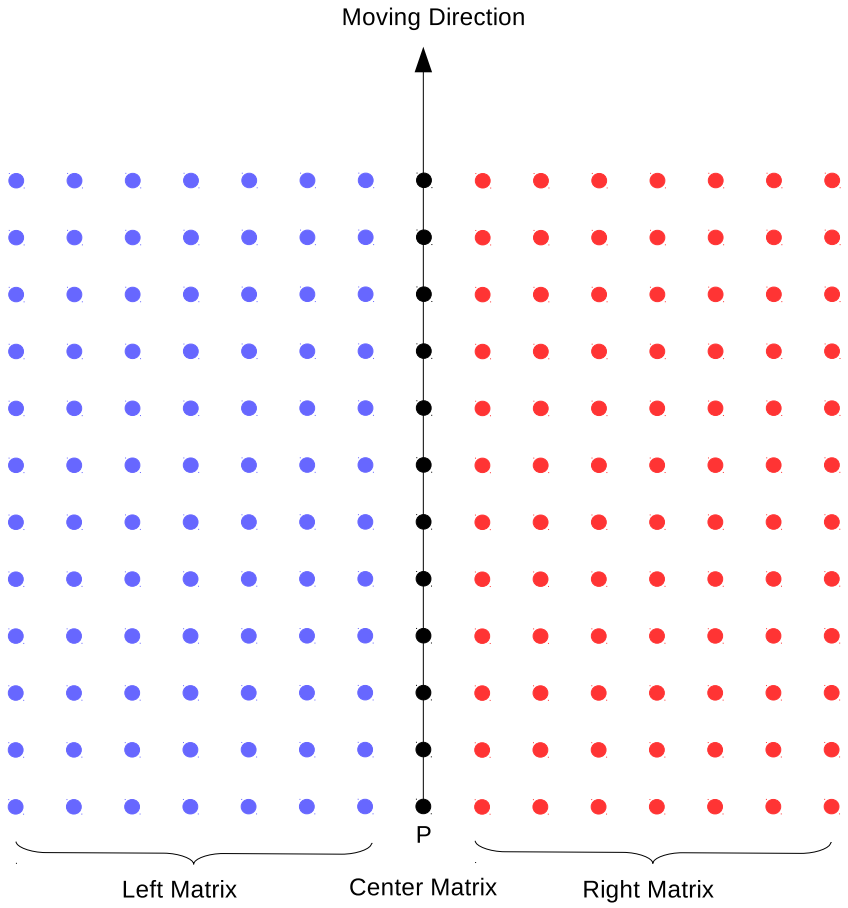

Figure 4.4. Prediction Points Matrix.

and θ as input and calculates the coordinates of points in the Right Matrix. Similarly, function $getPrediction\_left()$ calculates coordinates of the points resides in Left Matrix. Based on the basic trigonometric properties, the same θ can be used in all these functions. Some minor changes will be added according to the value ot *theta*. What's more, the prediction points in the Left Matrix subtracts the Δlat while the points in the Right Matrix add it. The Δlon will also be determined according to the value of θ. Again, all calculation we mentioned above are bound to the conditions mentioned in previous section. See figure 4.5.

Algorithm 5 Calculating Prediction Points

```
//For convenient, predict_Points_all is denoted as PP_all
for i to 12 do
    //Prediction Points in the center
    PP_all[i × 15 + 7] ← getPredictPoint(start_P, end_P, θ, i);
    PP_all[i × 15 + 7].y ← frame.cols/2;
    PP_all[i × 15 + 7].x ← (frame.rows/12) × (11 − i);
end for
for i to 12 do
    for j to 7 do
        //Prediction Points on the left
        PP_all[i × 15 + 6 − j] ← getPredictPoint_left(PP_all[i × 15 + 7], theta, (j + 1));
        PP_all[i × 15 + 6 − j].y ← PP_all[i × 15 + 7].y − (frame.rows/24) × (j + 1);
        PP_all[i × 15 + 6 − j].x ← PP_all[i × 15 + 7].x

        //Prediction Points on the right
        PP_all[i × 15 + 8 + j] ← getPredictPoint_right(PP_all[i × 15 + 7], theta, (j + 1));
        PP_all[i × 15 + 8 + j].y ← PP_all[i × 15 + 7].y + (frame.rows/24) ∗ (j + 1);
        PP_all[i × 15 + 8 + j].x ← PP_all[i × 15 + 7].x
    end for
end for
```

A tricky part here is to assign the geometric coordinates in a correct order. As shown in Figure , the indexing mechanism is little different when processing the left portion and right portion.

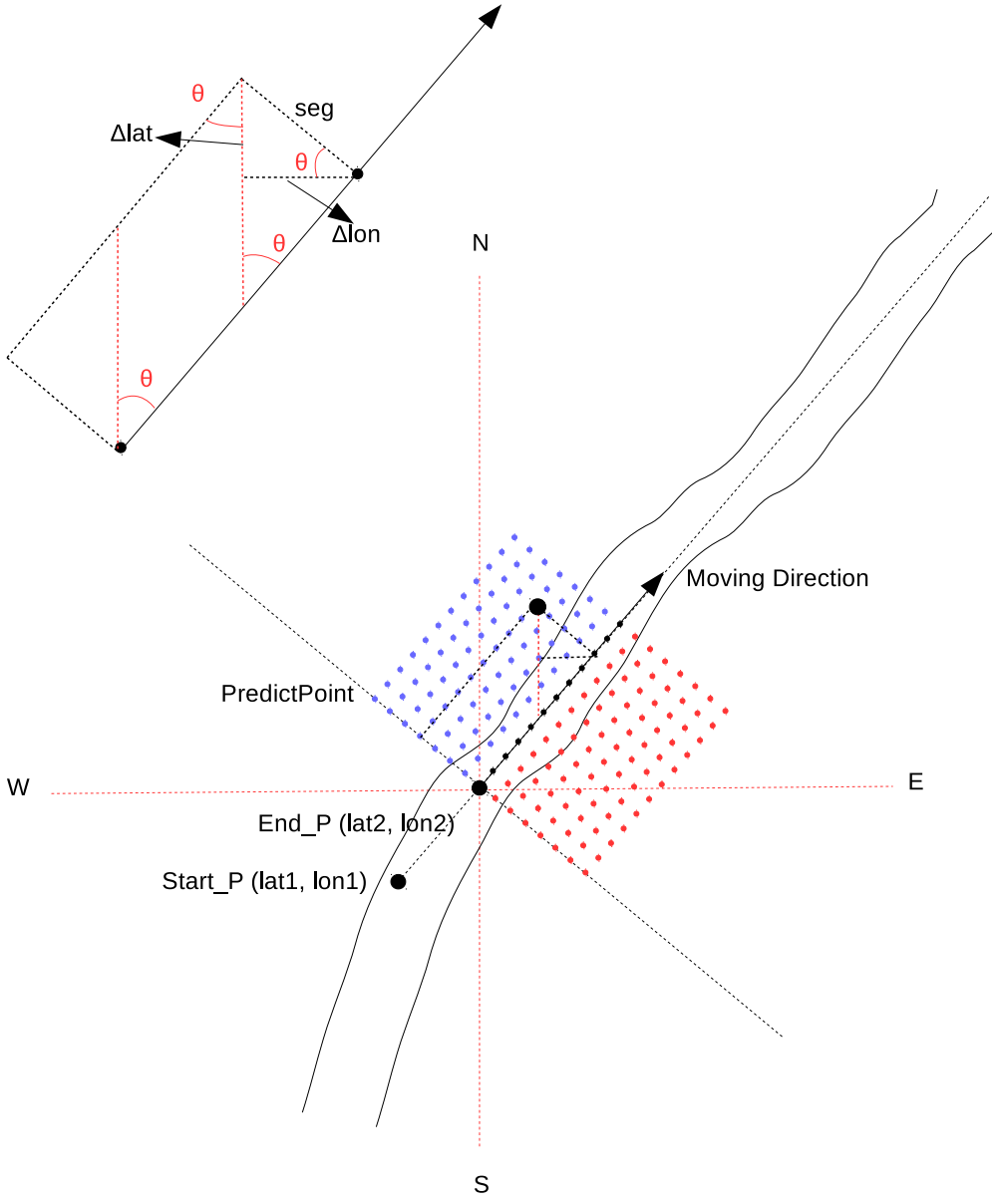

Figure 4.5. Relationship between theta and GeoPoints points..

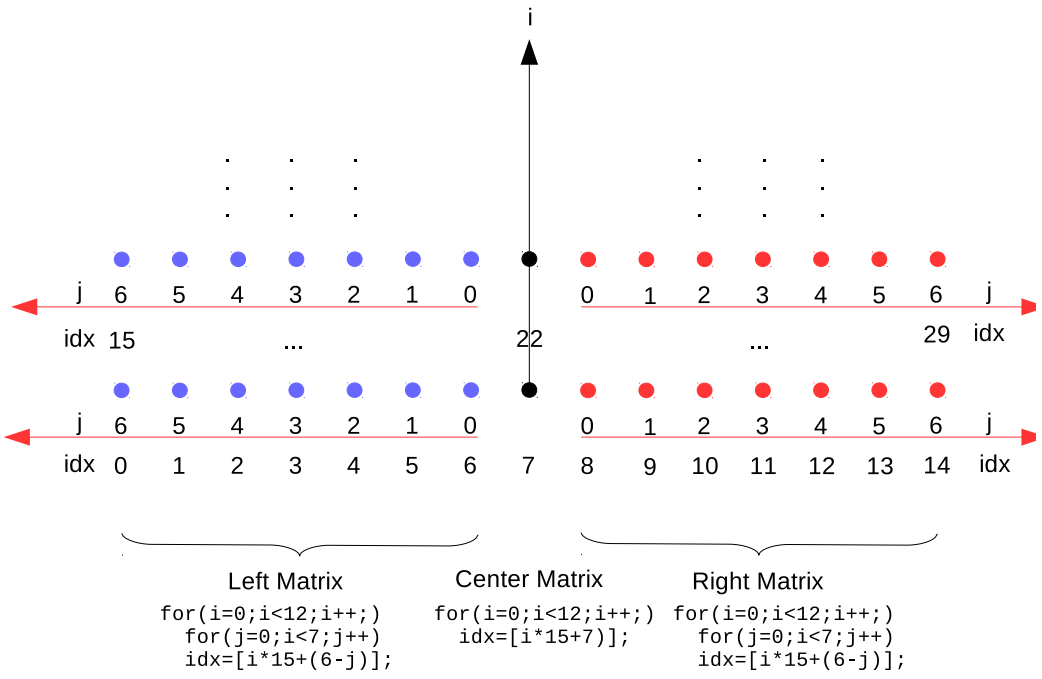

Figure 4.6. Indexing of Center/Left/Right Portions.

4.5 Coordinates Matching

This process has two stages. First, each sub-waypoint is subtracted by all prediction point. The prediction points that has the minimum difference in latitude and longitude of each row will be stored in array $min_idx[]$. This array has 12 elements, one for each row. The selected points are the **Matched Points**. These points are the closet points to the sub-waypoints. The complexity of this process is $O(n^2)$. This is another reason why we use 1-D array to store prediction points and waypoints are instead of 2-D arrays. In that case, the complexity decreases from $O(n^4)$ to $O(n^2)$. See algorithm 6

Algorithm 6 Matching

```
1: for i to 180 do
2:     for j to piece do
3:         min ← abs(PP_all[i] − geoPoints[j])
4:         if (min < Min[i]) then
5:             Min[i] ← PP_all[i]
6:         end if
7:     end for
8: end for
9:
10: for i to 12 do
11:     for j to 15 do
12:         if row_min == 0 then
13:             row_min ← Min[i × 15 + j]
14:         else
15:             temp_min ← Min[i × 15 + j]
16:             if temp_min < row_min then
17:                 min_idx[i] ← i × 15 + j
18:             end if
19:         end if
20:     end for
21: end for
```

Next step is to verify whether the selected points, or elements in $min\_idx[]$, are valid points or not. A prediction is valid if the absolute value of its value is less than a certain threshold. In our case, the threshold is 0.00002, or 2 meter in length. If a point is valid, constant 1 will be stored in corresponding slot of $valid\_dist[]$, which also has 12 elements.

Then a flag *valid_points* will calculated based on the values in *valid_dist*[]. Flag *valid_points* plays an important role in the drawing process, its value decides which drawing mechanism will be invoked.

Recall that in lane reconstruction process, the pixel coordinates of center line are stored in two arrays named *center_line_x*[] and *center_line_y*[]. Each of these matrices has 360 elements, the values of these matrix are the pixel coordinates of the blue pixels on the center line. The prediction points' x value in the Mat_C will be first compared with the values in *center_line_x*[]. If x can be found in the array *center_line_x*[], that means in the current row, there are both a valid prediction points and detected lane. Then, the value in *center_line_x*[] will be copy to the *center_line_y_selected*[]. The number of updated values will be accumulated and stored in *valid_y*. Result is shown in figure 4.7.

4.6 Drawing

At this point, the lane detection and GPS data matching is completed. The last step is to show the result on the screen. The following values play a crucial role in the drawing function, they are *valid_point*, *valid_y* and *center_line_y_selected*[].

- *valid_point*

  Keeps the number of valid prediction points of each row. One point in each row, twelve points in total.

- *valid_y*

  Keeps track of how many of these valid points has a detected lane at the same row. One pixel in each row, twelve pixels in total.

- *center_line_y_selected*[]

  Stores the y pixel coordinates of the valid prediction points.

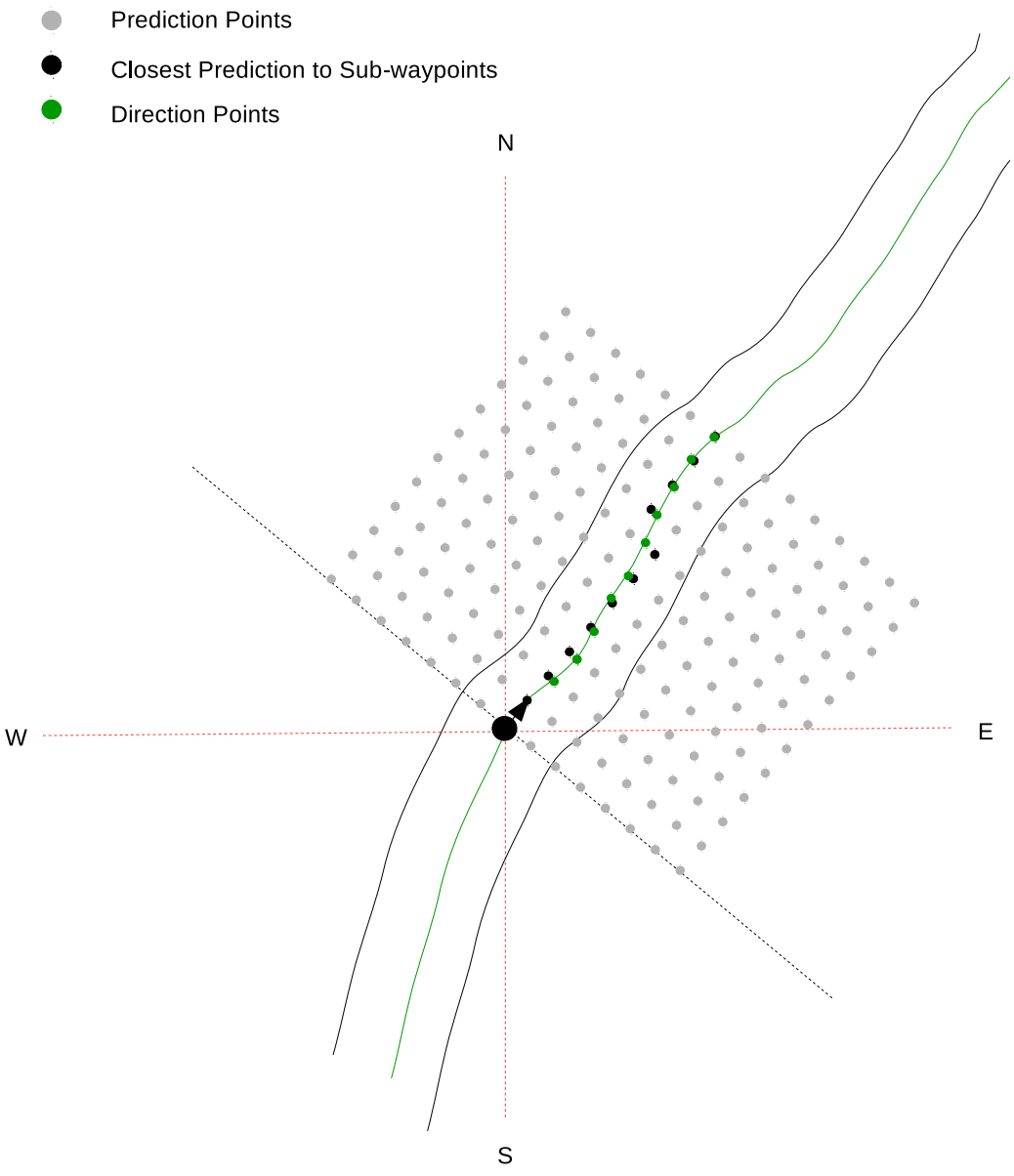

Figure 4.7. Result Of Matching Algorithm.

According to the value of two flags, different drawing functions will be invoked for different situations.

Currently there are two functions. The first function draws AR graph with detected lanes when $valid\_point$ and $valid\_y$ are more then 5. The other function directly draw the valid prediction points when the value of flags $valid\_y$ drops blow 5. However, both function depends on $flag1$, $flag2$ but only the first function requires the value in array $center\_line\_y\_selected[]$. However, Since the GPS device in our test is not very accurate.

The drawing functions will loop though the out put frame $lane\_frame$ and change the value of the pixel at according to the variables mentioned earlier. Both drawing functions draw on the $lane\_frame$. Figure 4.8 shows that there are both matched points 4.9and detected lane. Figure shows that there is matched points but no detected lane. Figure 4.10 shows that there is no matched points or detected lane.

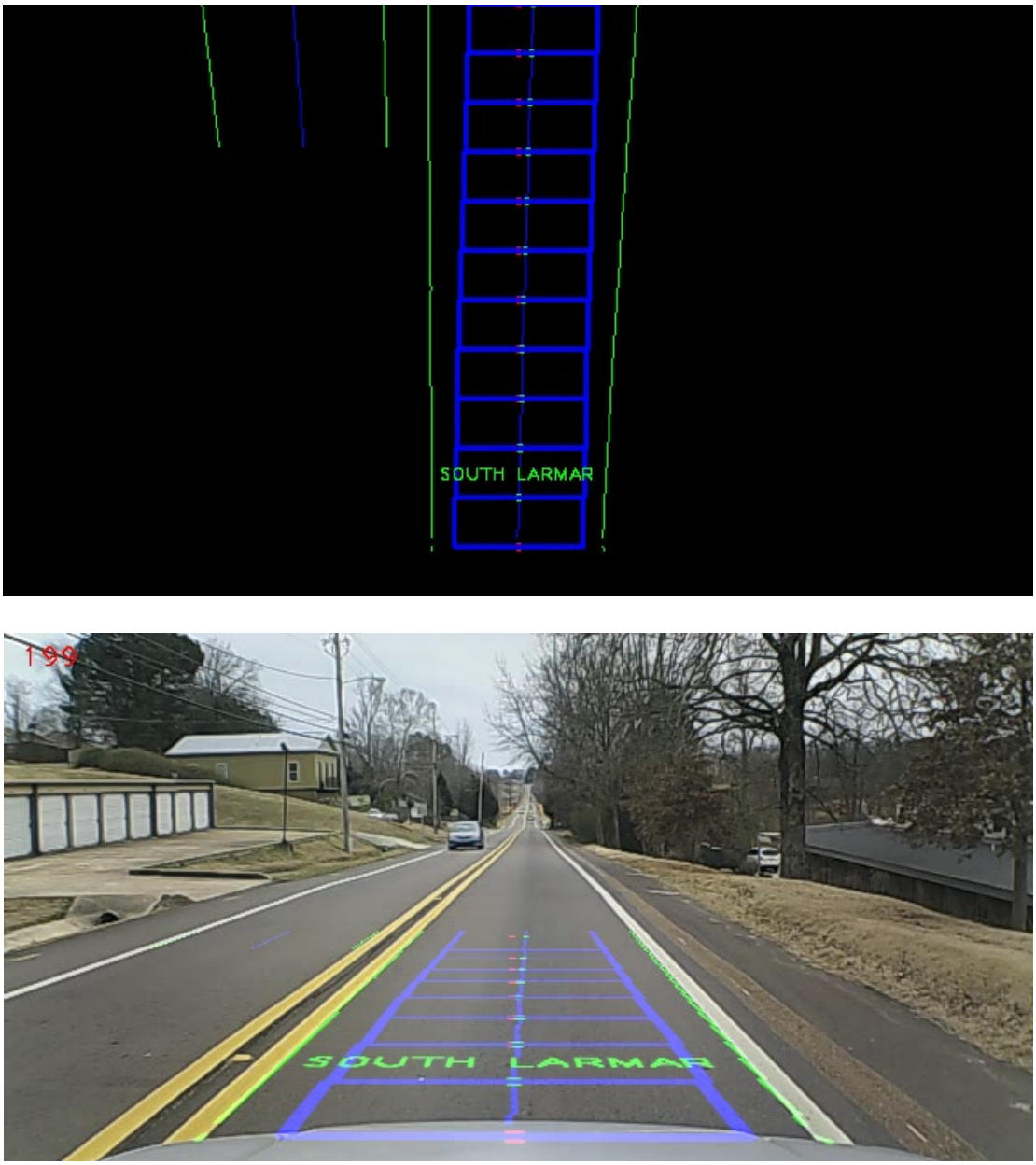

Figure 4.8. Matched Points-Detected Lane.

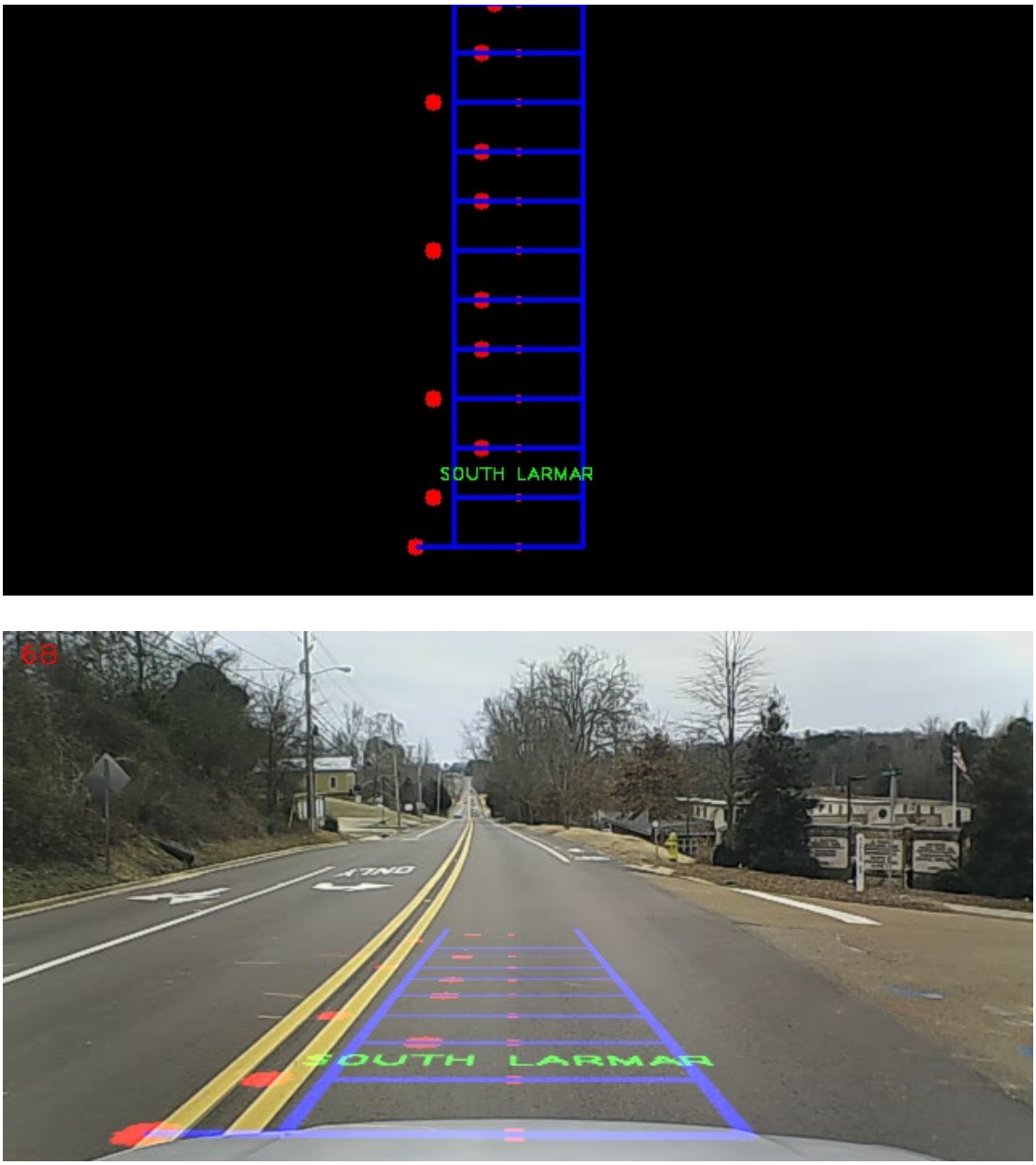

Figure 4.9. Matched Points-No Detected Lane.

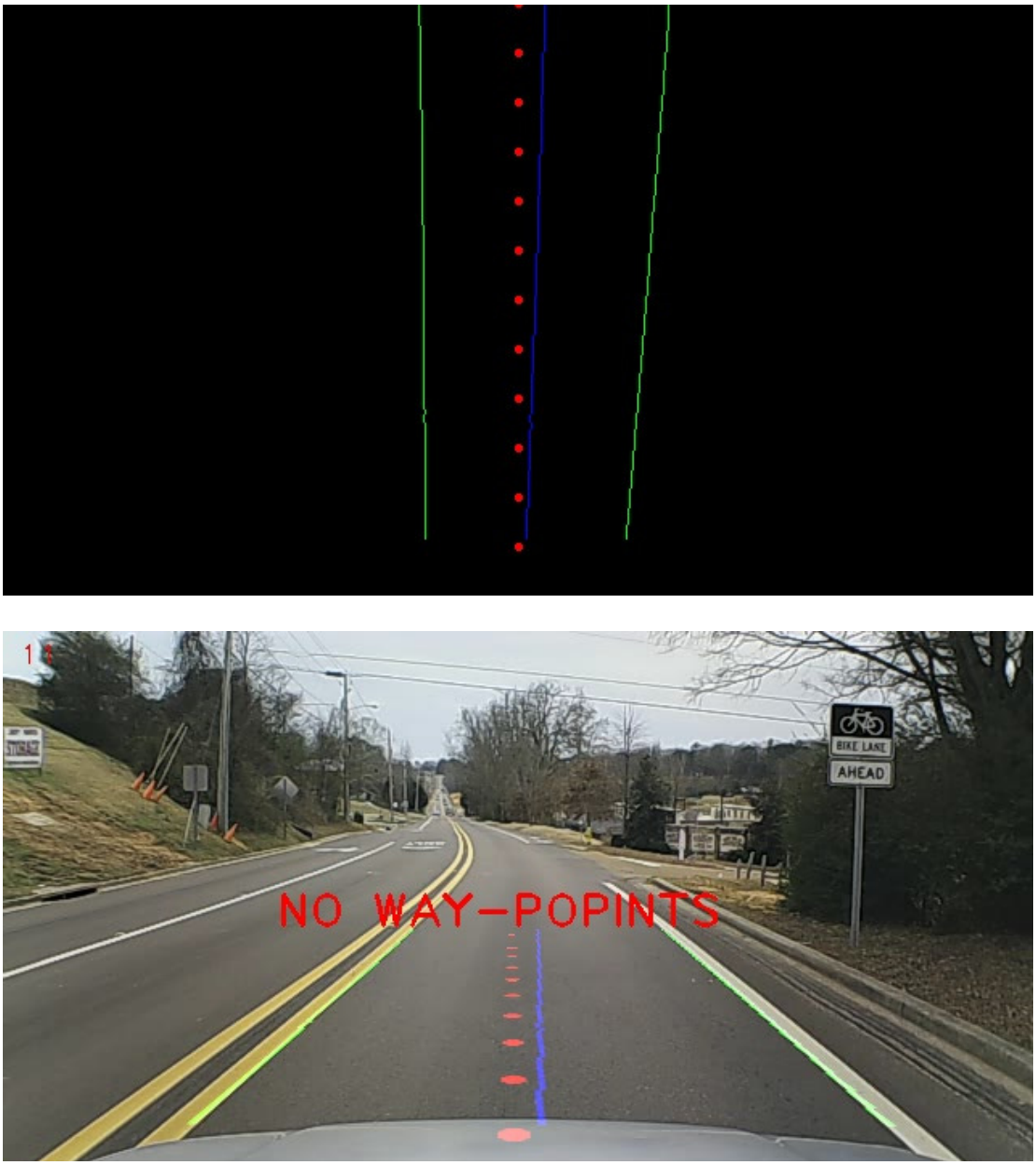

Figure 4.10. No Matched Points-No Detected Lane.

CHAPTER 5

OPENCL PORTING

5.1 Motivation

GPUs are originally developed for image related task such as display and texture rendering. Because of their highly parallel structure, GPUs are very efficient at SIMD (Single Instruction Multiple Data) operations. Some applications such as matrix multiplication or image processing algorithm, have been ported to the GPU and been proved that the execution time are significantly reduced by utilizing GPU cores in a highly parallel fashion. This computing paradigm is known as GPGPU (General Purpose Computing on Graphic Processing Unit). This chapter is focusing on parallelization our algorithms using OpenCL.

OpenCL is the the currently dominant programming language for GPGPU. It's not only because it's an standard for GPGPU programming but also because its cross-platform portability. Comparing to CUDA, which is a standard exclusively for Nvidia graphic cards, OpenCL works on any mainstream GPUs. The code on PC GPU can be executed on other platforms with minor changes. To ensure the potability of the our application, OpenCL is preferred.

The following section reveals the details of the OpenCL implementation of the ARNavi program.

Before we start the porting process, we want to know the capability of the hardware we use. We implemented a GPU based matrix multiplication and test it on Samsung Galaxy Note4 as a benchmark. By comparing the performance between CPU and GPU, we will have a rough idea of the computation capability of the hardware.

Samsung Galaxy Note4 has a SnapDragon 805 chipset including a Qualcomm Quad-core Krait 450 CPU and a Adreno 420 GPU. The input data are 128×128,256×256 , 512×512 and 1024×1024 matrices. The workgroup size is 16×16 for all input data. The profiling data is listed in table 5.1. According to the result, the acceleration rate on Andreno GPU is quite impressive, with large input 1024, the GPU implementation is almost 25 times faster on the CPU. We did not further increase the input size since the Matrix Multiplication took too long to finished on the CPU. See Figure 5.1 for details.

Table 5.1. Profiling Data On Adreno 420 GPU

Input Size	Workgroup Size	CPU Time (sec)	GPU Time (sec)	Ratio
128×128	16×16	0.0143	0.0023	6.2174
256×256	16×16	0.1195	0.0187	6.3903
512×512	16×16	2.6246	0.1807	14.5247
1024×1024	16×16	38.2791	1.5770	24.2733

5.2 Porting OpenCV Functions

In general, image processing algorithms are usually performed on a pixel level. The image processing algorithm loops though all pixels in a single image to perform some operations. For example, to convert a RBG image into gray scale image, each pixel's 24 bit RGB value will be converted to a 8 bit gray scale value and assigned to that pixel. However, some algorithms might access each pixel multiple times, such as Gaussian Blur and Affine Transformation. Depending on the complexity of the algorithm and image size, the execution time of image processing could take a long time. At this point, the CPU version of ARNavi has been completed and tested on PC. For 640x480 input, the average frame rate is 32 fps.

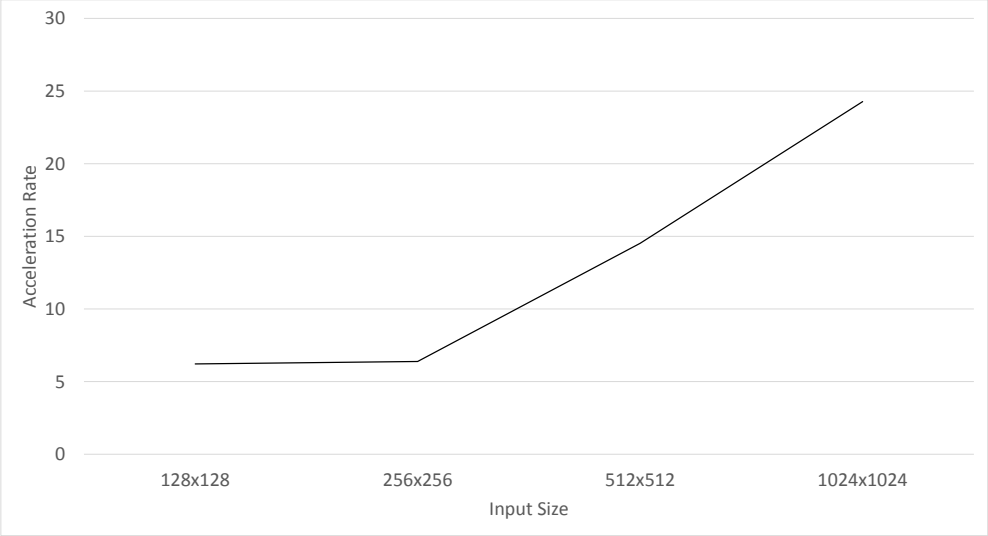

Figure 5.1. Acceleration Rate Of Matrix Multiplication On Adreno 420 GPU.

In the ARNavi, image processing algorithms are the most time consuming algorithms. It takes about 80% of total execution time for the image processing. If these function are executed on GPU, their performance will be increased significantly. OpenCV research team already realized the power of GPU and developed two GPU implementations of their OpenCV functions. The only difference of these two versions is the target hardware. *gpu* module is for Nvidia GPU while *ocl* module is for AMD and other OpenCL supported devices.

In OpenCV version 2.4.9 and beyond, ocl functions has became standard and does not required any manual initialize work as previous releases. As long as the *libOpneCL.so* library presents on host machine and detected by OpenCV, the ocl functions are called instead of serial version. There is some trade offs when invoking the ocl function. The first one is the overhead of converting data type. The data type must converted form Mat to $ocl :: Mat$ before the GPU can process it. The other trade off is the overhead of data copy time. Since the data are copied from host memory to device memory, the overhead could

be large if the image is large. However, if the data transferring time is relative small to the overall processing time, it can be omitted.

To convert data between Mat and $ocl :: Mat$, OpenCV provides two convenient functions:

- To copy data from Mat to ocl::Mat
  $oclmat\_name.uploads(mat\_name)$
- To copy data from ocl::Mat to Mat
  $oclmat\_name.downloads(mat\_name)$

Inside these function, the values of each pixels are read and then stored in a opencl vector type cl_float4. cl_float4 holds up to 4 floating point values which is big enough to fits in the value of pixel. No matter the pixel has 3 or 4 channels. Most image type such as RGB, RGBA, BGRA can be fit in this vector without problem. Another reason to use vector is that GPUs are vector processors, it's much faster to process vector type than scalar type on the GPU.

To use a *ocl* functions, just simply put *ocl* :: class name in front of a CPU functions that will be executed on GPU.

5.3 Porting User Functions

According to the profiling data, the following functions are considered as time consuming functions. These functions are ported to GPU using OpenCL.

- **Lane Detection**
- **Prediction Points And Waypoints Matching**

Since the details of the functions are explained in previous chapter, in this chapter, we will mainly focus on analysis the structure of the key algorithms and the implementation of the OpenCL code.

5.3.1 Lane Detection

5.3.1.1 Parallel Implementation

In previous section, we presented the details of the Lane Detection algorithm (LD). Here, we will only go through the implementation details of the parallel version of Lane Detection.

A quick recap of how Lane Detection algorithm works. In the Canny Edge Detection process, the three channel RGB image transformed into a single channel gray image. To be more specific, a binary image. For each pixel, it can has only two values, either 255 for white or 0 for black. The task for Lane Detection is to find out the white pixels that belongs to a lane. Thus, the LD goes through each row and calculates the Most Frequently Appear Segment of the image. Starting from the first element, the segment counter increases by one as it goes through the row. When encountering a white pixel, the distance between two pixels is calculated and stored at the corresponding position in another matrix. After that, the counter reset itself to zero and ready for next row.

In this process, the value of length is calculated between two adjacent pixels, the current white pixel and the previous white pixels. Thus, there is a data dependence between pixels. A thread must continuously process the entire row until it reaches the last element of the row. This process cannot be subdivided into more sub processed because of the data dependence. As a result, each row is treated as a chunk, or workgroup in OpenCL terminology.

The key of GPU parallelization is to chunk the data into pieces and process them parallelly. In our case, the minimum size of chunks are on row. That limits the number of threads that can being process at the same time. In our case, there are only 360 threads executing at the same time for a 360 by 640 image. The implementation is shown in Figure 5.2.

Algorithm 7 is the pseudo-code of the parallelized Lane Detection. On CPU, the LD

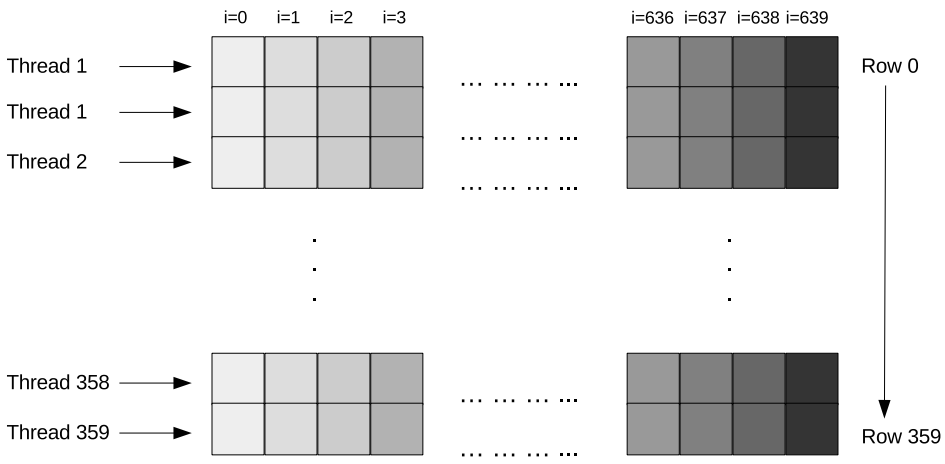

Figure 5.2. Scheme Of Parallel Lane Detection.

use a double for loop the go through the pixels in the input image. the complexity of this operation is $O(mn)$, where m,n is the number of the row and column of the image. On GPU, the complexity is reduced to $O(n)$ since there are 360 threads executing at the same time.

Algorithm 7 Parallel Implementation of Lane Detection

```
gid ← get_global_id(0); //1-dimension
dist ← 0;
predist ← 0;
order ← 0;
for i to cols do //cols = 640
    if mat[cols × gid + j] ≠ 0; then
        dist ← j − predist;
        nl_dist[cols × gid + j] ← dist;
        predist = j
        order + +;
        nl_order[cols × gid + j] ← order;
    end if
end for
```

The parallelization level is highly depends on the algorithm. If the algorithm does not have any data dependency, the performance can be further improved because more threads are involved. For example, to convert a color image to a gray image, every element is treated as a stand alone item. Thus, on the GPU, a thread is assigned to every single of the pixel. In this case, there are 230400 threads executing at the same time (although the real number of executing threads are limited by the number of CUs and size of wavefront).

5.3.2 Prediction Points And Waypoints Matching

5.3.2.1 Parallel Implementation

Before diving into the OpenCL implementation of the Prediction Points and Waypoints Matching algorithm (Matching for short), let's briefly go though the matching algorithm works on CPU.

The main purpose of this function is to find the matched (or closest) points to the waypoints in the prediction points matrix. These mathed points are considered as valid if

they fulfill some conditions. The process is simple, for every elements in the prediction point matrix (180 in total), a subtraction is performed with all elements in the waypoints array ($WP[]$). The minimum value is first selected then compared with a threshold. If the value is less that the threshold, the index of that prediction point is stored in a corresponding position of another matrix, the matched points array. Then, we go through the matched point array and change the color of the matched points. The highlighted route indicates the direction the user should follow.

The CPU implementation is listed in 6. The complexity of this function is $O(mn)$. Where m,n are the size of PP_all array and WP array. The total number of operations of this process is $180 \times 1271 = 228,926$.

The OpenCL implementation of this function is listed in Algorithm 8. Here, we use one dimension threads only. Although this process can be done using two dimension threads, the overhead is too high comparing to the execution time of the algorithm itself. To use two dimension threading technique, these two 1D arrays have to be converted into two 2D arrays. In our case, we need to duplicated the 180 elements array and 1000 elements array into two 180×1000 arrays. This is the major overhead with this approach. After the GPU finished the computation, we will use another double for loop to find the minimum values of each row from the 2D array. These processes adds another overhead to the application.

Algorithm 8 Parallel Implementation of Lane Detection

```
gid ← get_global_id(0); //1-dimension
for i to pieces do //pieces=1000
    temp ← fabs(PP_all[gidy] − WP[gidx]);
    if (temp < min and < 0.00002) then
        min ← temp
        min_idx[gid] ← min
    end if
end for
```

CHAPTER 6

ANDROID PORTING

6.1 Android

6.1.1 Backgrounds

Android is a mobile operating system based on Linux kernel and developed by Google. It's free of use to users and developers. Programming language of Android system is Java, a easy-to-use object oriented programming language. With it's highly developed UI and high hardware compatibility, Android has become one of the most popular operating system on mobile platforms. To extends the profitability of our application, we choose Android as our develop platform.

Because of the success and popularity of Android OS, the OpenCV team and Khronos Groups finally extend their interests to this platform. Recently, OpenCV team released a library and Java package are specially designed for Android platform. The package is named OpenCV4Android. This release includes most frequently use functions and modules. Some function, due to the lack of the backend support on Android, are missing from the original library. On the other hand, the OpenCL is already well supported by the mobile GPUs. However, due to its complex environment configuration and hardware requirement, this technology is still remain silent on Android platform. One of our goal is to enable and

combine these two APIs on Android platform.

In our previous work, we successfully ported the LBP facedection on a Linux embedded system with OpenCV-CL support. This experiment not only achieved a performance boots but also proved that by porting parts of the code to the GPU, the power consumption can be reduce significantly. Similar results are expected after ported the ARNvi to Android.

Unfortunately, the ocl module, which is the glue between OpenCV and OpenCL, was no included in OpenCV4Android 2.4.10 release. As this thesis being written, there is no attempts to utilize the ocl modules of the OpenCV on Android platform.

However, with numerous test and experiments, the ocl modules are eventually set up and functioning correctly via the help of JNI (Jave Native Interface) and other tweaks. In remaining chapters, more details of the environment configuration and porting process will be revealed.

6.2 Java Native Interface And Java Native Development Kit

6.2.1 Introduction

In Java, the Java Native Interface is a programming frameworks that allows the Java program to call and be called by the native functions. By interacting with native code, the Java program expends it's functionality by embedding user defined classes that does not provided by Java standard APIs.

Native functions on Android platform is usually written in C/C++. At compile time, the native code are compiled as a shared libraries and stored on the device. This library can be loaded and accessed by Java code later. To compile a native library, the Android SDK has to to access to the headers and libraries in the Java Native Development Kit which are specifically implemented for the ARM structure. In other word, Java NDK just a toolset that

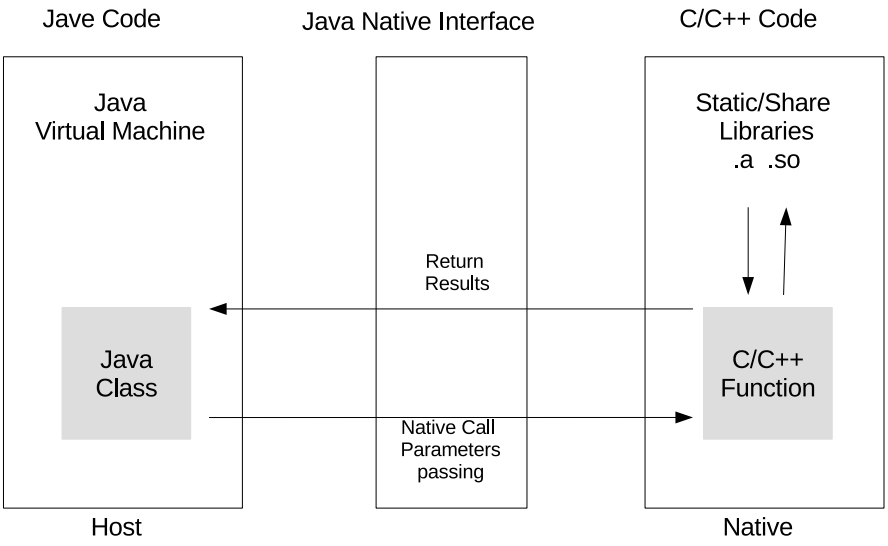

Figure 6.1. JNI Workflow.

providing C/C++ support on Android platform. It contains most frequently used headers such as *stdlib.h*, *stdio.h*, *iostream* and $fsstream$. As at run time, the Java program must loads the compiled binary before calling the native functions. Data from Java side will be passed to the native function as a Java Objects (jobject). In side the native function, the jobject will be dereference and treated as standard C/C++ variables. After the computing is done, all data will be rebuild as jObject and passed back to the Java side. Figure-6.1 shows the basic work flow of a Android JNI application.

6.2.2 Environment Configuration And Building Process

To get more information about programming environment set up for Java SDK and NDK,or compiling process for JNI programs, please refer to [5] for more details.

6.2.3 Implementation

The code snips of *HelloJNI* demonstrate the basic implementation of a JNI application on Android. The *HelloJni.java* and *HelloJni.cpp* are the implementations on the Java and native code. This application simply return a message using C++ native function *stringFromJNI*(). HelloJni.java

```
public class HelloJni extends Activity{
    ......
    public native String  stringFromJNI();
    static {
        System.loadLibrary("HelloJni");
    }
}
```

HelloJni.cpp

```
#include <jni.h>
extern "C"{
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env, jobject thiz )
    char *mes = "HelloWorld_From_Native_Code!"
    char msg[200];
    sprintf(msg,"%s",mes);
    return env->NewStringUTF(msg);
}
}
```

There are three important steps when programming with Java JNI. The first step is to declare a public native method in Java code. As shown in the HelloJni.java file, the native must be added and placed in front of the return type. The second step is to create a native

function includes the method name and the package name. Since one Java program might contains several package, each package might have their own native functions. This naming convention prevents the situation that the host code calling each other's native functions when the native functions have same name. The last and the most important step is to load the share library compiled from the source code in the host side. Whitout this step, the native function cannot be invoked.

In theory, by employing JNI supported to Android applications, it is possible to embed any C/C++ functions or libraries to a Android. As long as the C/C++ are compile correctly using the toolchain for the processor. Thus, OpenCV and OpenCL should run on Android platform with correct configuration.

6.3 OpenCV on Android

OpenCV4Android is the official release from OpenCV research team that targeting Android platform. The latest version was 2.4.10 which released on 2014-10-02 along with the PC and iOS version. The OpenCV4Android SDK contains several pre-built Android Application Packages files(apk) called **OpenCV Manager** and bunch Java class files.

The apk files are pre-built for most commonly seen hardwares structures on the market. For examples, $armeabi$, $armV7a$,$armv7a - neon$ and $mips$. The user has to make sure the OpenCV library matches the device's structure. Otherwise, OpenCV application won't correctly invoke the functions in the libraries. If the target device is not in the supporting list of the pre-builrt package, the user can also build their own libraries with the source code. For Smasung Galaxy Note 4, we use $armv7a$.

The responsibility of the OpenCV Manger is to maintain and manage the OpneCV libraries on the user device. It provides some kind of managing mechanism to allow the dynamic OpenCV libraries shared between applications on the same device.

The OpenCV Java class are Java wrappers with built-in native OpenCV function

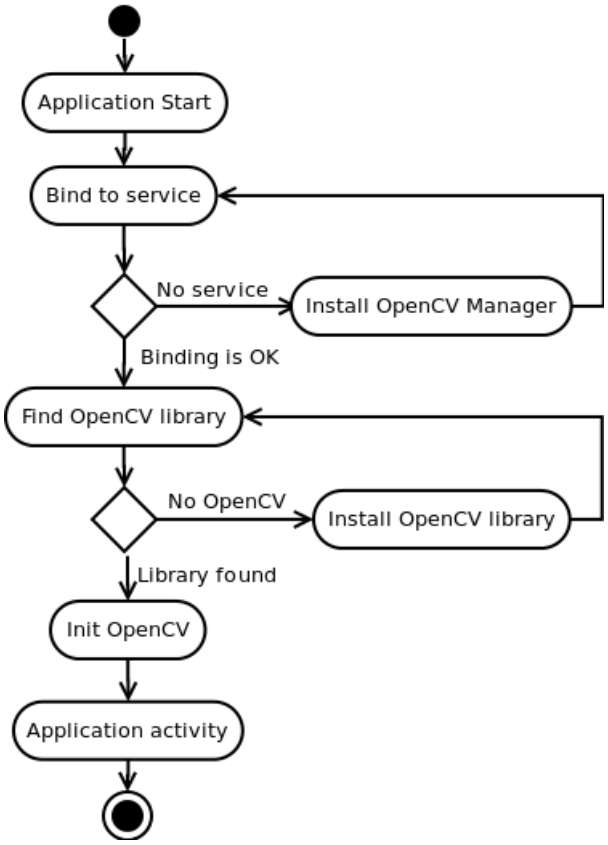

Figure 6.2. OpenCV4Android Model For End User

calling routines. These wrappers hides the Java Native Interface from the user and greatly simplified the initialization work.

To embedded OpenCV in Android Applications, the corresponding **OpenCV Manager** must be install on the target device. More importantly, the NDK environment must be set up correctly before hand. Otherwise, the native code will not be compiled. As we mentioned before, the NDK is the key to compile the native functions since it contains the basic support of C/C++. Here, the OpenCV libraries are considered as part of the native libraries because they are written with C/C++. Since the OpenCV libraries are already compiled as dynamic shared libraries (libopencv_ java.so), the android application simply loads it to the System instead of recompiling it. For all dynamic (.so) and static(.a) libraries, programmer can put them in the Android.mk file and compile them with C or C++ source files. Thus, the native files will be compiled as one big library instead of many small libraries. In such fashion, it's more easy to manage and keep track of the libraries.

There are two approaches of invoking OpenCV functions on Android. Via Java OpenCV APIs (see Figure 6.3) or directly invoked from the the native code (see Figure 6.4). The code snips produce same results but with two approaches. They both read in a RGB image, convert it to gray scale image, and then display on a $ImageView$ widget. Is easy to tell that using Java API is not efficient as native interface. For each function call in Java code, there is a corresponding native call. But in approach 2, there is only one native call for the entire process, which has less overhead.

The first example shows the routine of calling OpenCV functions from Java. Since the Java APIs inherits the structure of original design from standard OpenCV, the calling routines are very similar to the routines on PC. As you can see, the function calls are almost identical exception some minors changes. The OpenCV APIs on Java makes writing OpenCV functions extremely easy. If programmers has programing experience with OpenCV, they can easily adopt this programming style and writing OpenCV programs immediately. However, there is a main drawbacks of this approach. The OpenCV are not fully ported to Java.

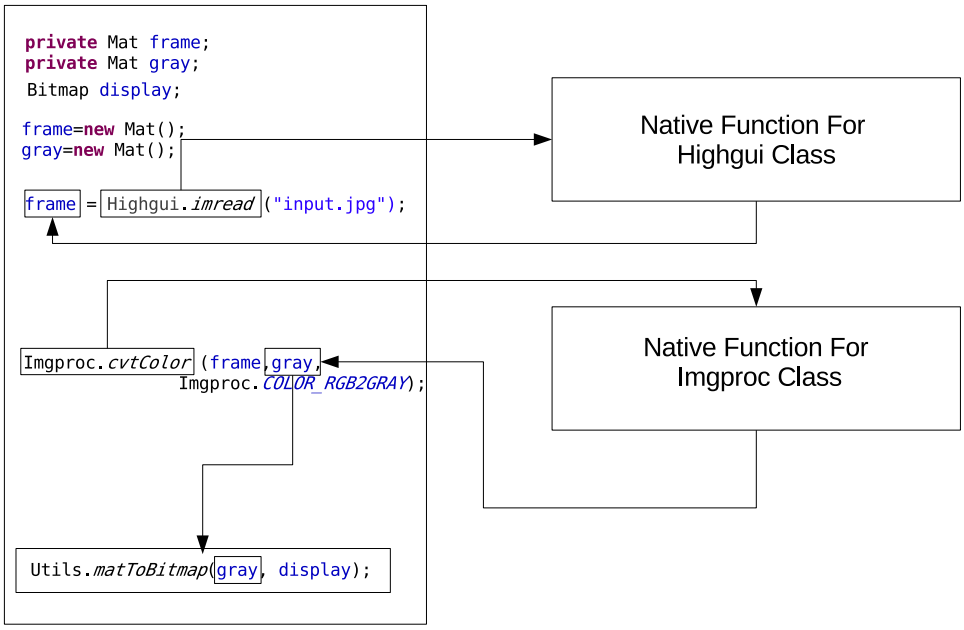

Figure 6.3. Calling OpenCV Functions In Java Code.

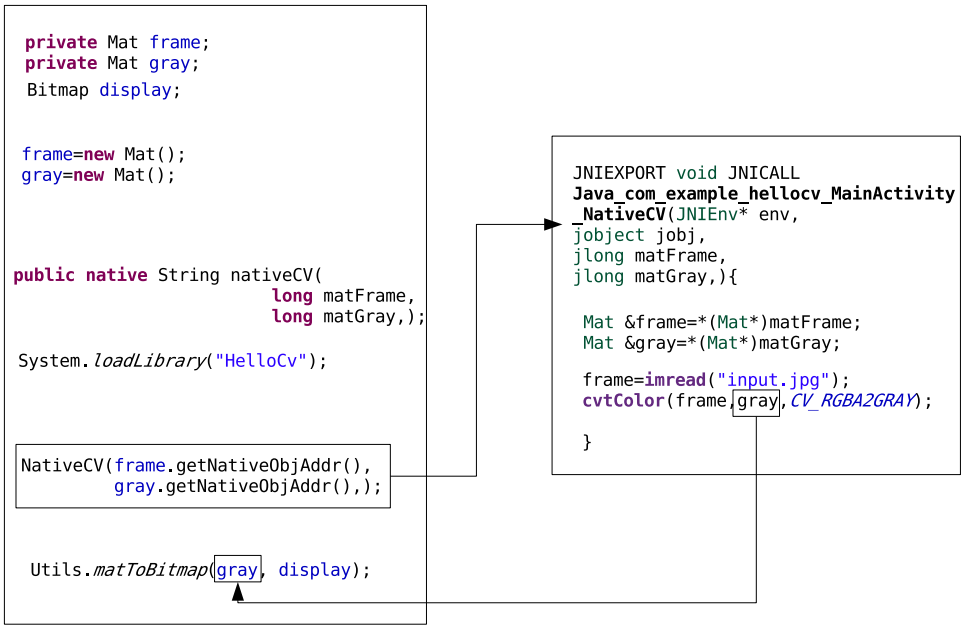

Figure 6.4. Calling OpenCV Function In Native Code.

Some functions are missing from the original implementation due to the limitation of Java language and lack of the back end support.

The other approach, comparing to the first approach, the initializing part are the same. However, the following process, is slightly different. The addresses of the matrices will be passed stored as *long* variables and passed as parameters in user defined native function. In the native scope, all addressed passed from Java side will be resolved as type *jlong* and dereferenced as a pointer. Each pointer points to the address of a OpenCV matrices declared in the memory and assigned to C++ OpenCV matrices. From here, the rest of the code is identical to the original code. In fact, except for the dereference part, the programs source code can be use mutually on the PC and Java. More importantly, the OpenCV function call does not rely on the Java APIs, that means, even if the there is no Java packages for the features we want to use, we call still call the functions as long as the corresponding libraries are presented on the system. Even the OCL modules. Still, if there is no hardware support or back end support for that function, an error occur. This approach, in my opinion, is the best way to integrated native functions in Android application.

6.4 OpenCL on Android

To enable the OpenCL functionality on Android device, the development device must be OpenCL compatible. The OpenCL library is usually placed in the */system/vendor/lib* folder. If the *libOpenCL.so* is missing on the device but the GPU did support OpenCL, the user can always download the source code of the driver from hardware vendor's website and compile it manually. The latest mobile GPUs made by Qualcomm (Adreno 4xx serious) and ARM (Mali T7xx serious), are officially supporting OpenCL 1.2.

There are two prerequisites before programming OpenCL programs on Anrdoid. First prerequisite is to set up the NDK and enable building native functions on Java. Second, the OpenCL headers and library on the device must be copied and placed in the local project. To

ensure the compatibility, the programmer can always pull the existing lirary from the device. The device manufacturer might modified part of the original source code while leaving the user unaware. In this case, the compiler might complain about the undeclared function names while during the compile and linking processes.

In previous section, we discussed the implementation routine of an native OpenCV programs on Android. Actually, the same routine can be applied to building OpenCL programs. A quick recap, there are five stages of a native program. First, create entry point on Java side. Then, pass data to native code as Java objects. On third stage, Java objects will be dereference and stores as native variables. After calling the native functions, which is stage four, the processed data will be passed back to the Java side in the final stage. As we mentioned in Chapter 5, for all OpenCL programs, the kernel configuration is important and highly depends on the hardware. Even a small mismatch on the configuration could leads to a unexpected errors. In worse cases, the GPU device might stuck in a dead loop and freeze the device. To make sure the OpenCL code works properly on the mobile GPU, its the programmer's responsibility to find out the hardware's specification and change the kernel configuration correspondingly.

In this thesis, we choose the Adreno 420 GPU as our main test device. It's not only because it supports the full OpenCL 1.2 specifications but more importantly, the workgroup size and local workgroup size hava the exactly the same as AMD graphic cards. In fact, we can reuse the code for AMD graphic cards without only minor changes.

In Section 5.1 , we tested Matrix multiplication on the Samsung Galaxy Note 4 with Adreno 420 GPU. The result is quite impressive. The performance of GPU implementation increased almost 3 times. By porting our application to the GPU, we believe there should be a promising speed gain in the parallel version.

6.5 Implementation on Android

With the help of the Java JNI, we are able to port entire C/C++ application to Java without struggling too much. Since the most implementation can be reuse as native code without any changes, the only thing we need to do is to create an entry point that triggers the native processes. That is, except declaring native function and data on Java, the entire application will be execute natively. This is the only way to invoke OpenCV and OpenCL function calls on Android platform. The workflow of the application is shown in Chart X.

As demonstrated, all OpenCV Mat matrices are first declared on the host side. Then, after the address are resolved and associated with native variables on the native side, native code takes care of the rest of the job. As mentioned before, to utilize the OCL module, data type Mat must be converted to $ocl :: oclMat$. Then, the workload will be transfered to the GPU and being processed parallelly. After two other OpenCL ported function are executed, the final result, which is a Mat, will be passed back to the host side and store as Bitmap. The last step is simply displaying the result using Android APIs.

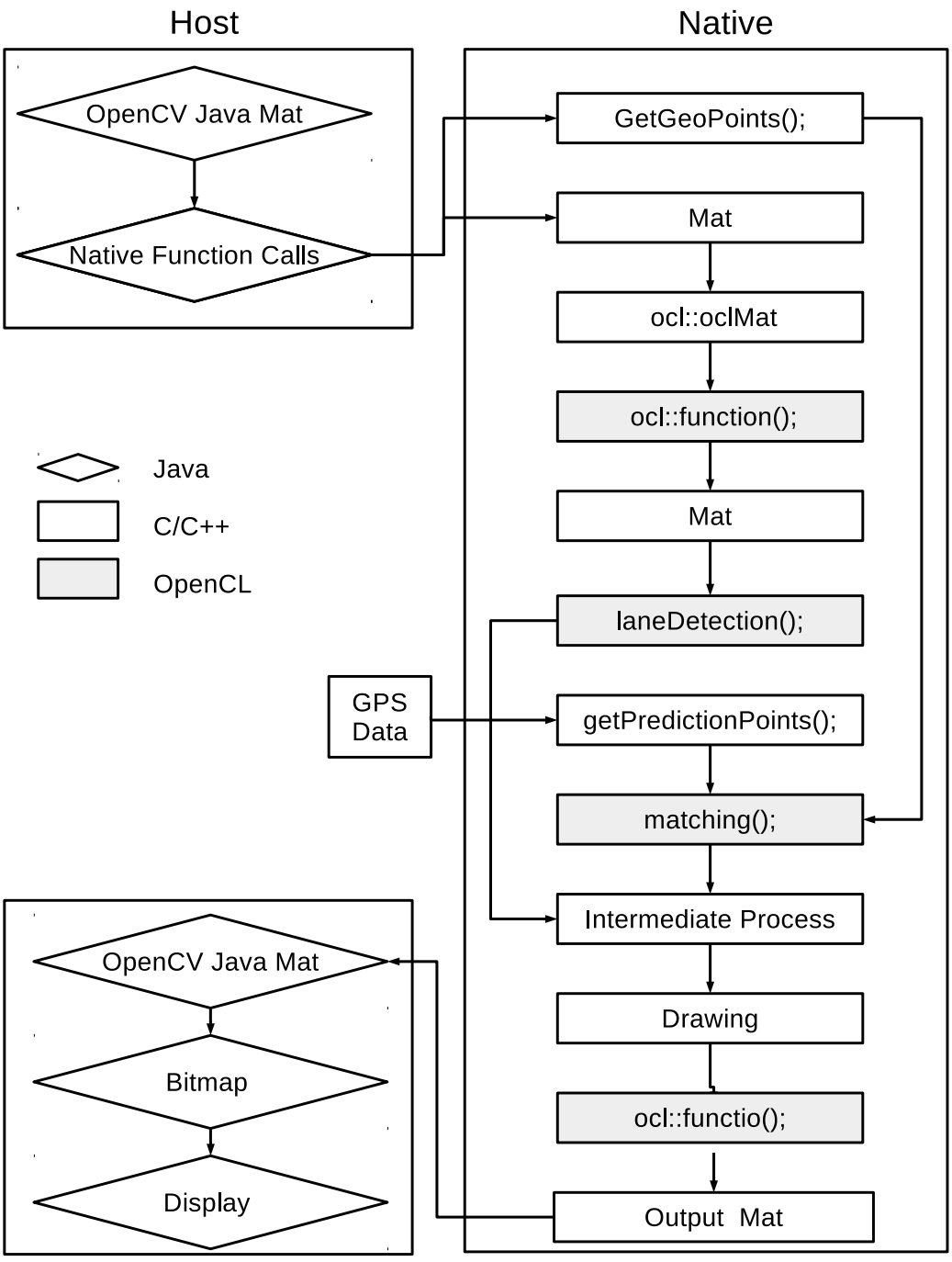

Figure 6.5. Workflow of ARNavi Application

CHAPTER 7

EXPERIMENTAL RESULTS

7.1 Test Data

Instead of streaming live data from camera and GPS device, we made several changes to the application to be able to work on prerecorded data. This way we can minimize the affect caused by uncontrollable factors such as light condition or strength of signal.

To record input video with GPS data included, we implemented a small application for this specific task. The idea is simple that while recording a video, each frame is associated with three variables, the number of current frame, the latitude and longitude acquired from the GPS parser. These data are stored in three separate arrays of the same size. After the recording process is finished, the arrays are written to a text file in the format shown in Table 7.1.

Numbers in first column are frame numbers. The second and third columns are geometric coordinates recorded at the moment when the frame is written. When the application starts to query frames from the video file, we use a variable to keep track of how many frames are being read. Then, we can use that number to get the latitude and longitude for a specific frame. Of course, the text file must be parsed and stored as two separate arrays, one for latitude and the other for longitude.

Table 7.1. Prerecorded GPS Data.

Frame#	Latitude	Longitude
1	34.34580833	-89.5212
2	34.34580833	-89.5212
3	34.34566833	-89.52117667
4	34.34552167	-89.52114833
.	.	.
.	.	.
.	.	.
100	34.34521167	-89.52109167
.	.	.
.	.	.
.	.	.

7.2 Test Method

Both serial and parallel implementations are tested on Samsung Galaxy Note 4. The only difference is that the parallel version runs part of its code on GPU. The average processing time of one frame will be the main comparison criterion.

Wall clock timer is the main tool in our profiling work. When profiling on GPU, we use OpenCL profiler due to two reasons. The accuracy and the resolution. Since the OpenCL profiler is designed for the hardware, it works better on the GPU hardware thus return more precise data. For resolution, using OpenCL profiler is able to break down the application and measure the performance of each OpenLC function. In our case, we are able to get much detailed profiling data. The total execution of our OpenCL function can be divided into three parts, the read/write buffer time and kernel execution time. Read/write buffer times are the data copying overhead between the device memory and host memory. The kernel execution time is the actual execution time of the parallelized algorithm. How to evaluate these data is critical to the final result. To give a more fare comparison, the read/write buffer overhead are considered as part of the GPU execution time since they are the trade off of using OpenCL. The performance of PC will also be presented.

7.3 Test Platform

Mobile Platform

Device: Samsung Galaxy Note 4

OS: Android Chipset: Snapdragon 805

CPU:Quad-core Qualcomm Krait 450 , $2.7GHz \times 4$

GPU:Qualcomm Adreno 420 GPU @ 600MHz

OpenCL:1.2 full

OpenCV:Open4Android 2.4.10

7.4 Test Results

In this section, we evaluated the performance of two implementations of our application. First, we will take a look at the total execution time to get a rough idea of the overall performance. Then, we tear down the application and exam the performance of each ported parts. At last, we give a detailed analysis to the Lane Detection and the Matching algorithm by contrasting the detail profiling data. For convenient, two implementations will be refer as ARNavi_C and ARNavi_G, the image processing part one and part two will be referred as Imgproc1 and Imgproc2. The Lane Detection and Matching algorithm will be referred as LD and Match respectively.

Figure 7.1 shows the total execution time of the two implementations. The result is quite impressive. Whit 640×360 input image, the CPU version takes 0.1381 second to process one frame while the GPU version takes only 0.05291 second. ARNavi_G is 2.63 times faster the the ARNavi_C.

Figure 7.2 shows the execution time of 4 ported sections, Imgproc1, Imgproc2, LD and Match. The detail profiling data is listed in Table 7.2. As you can see, the major accelerations come from Imgproc1 and Imgporc2. The execution time of Imgproc1 reduced from 0.051274

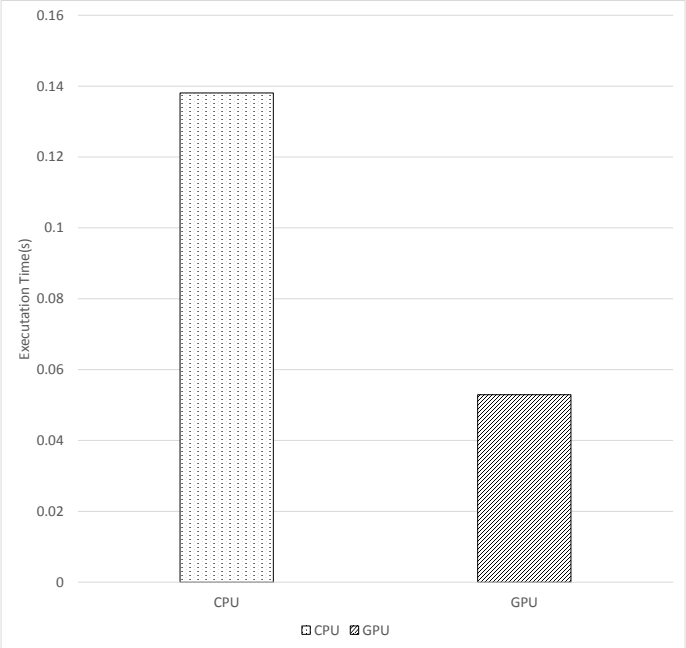

Figure 7.1. CPU vs GPU: Total Execution Time.

second to 0.033178 second (1.5 times faster). The processing speed of Imgproc2 reduced from 0.09358 second to 0.016878 second (5 times faster). The Match algorithm is 6.7 times faster on GPU (0.00) comparing to the CPU version. However, the LD algorithm, its performance drops from 0.00207 to 0.002595 second.

Table 7.2. Profiling Data Of Ported Portion.

	Imgproc1	LD	Match	Imgproc2
CPU(s)	0.051274	0.00207	0.001792	0.069358
GPU(s)	0.033178	0.002595	0.000264	0.016878

Figure 7.3 compares the performance of the two implementation of LD algorithm. The GPU version is 0.5 ms slower than the CPU version. This conclusion is made on the behalf of the total execution time. But if we compare the execution time of the algorithm only (the kernel execution time on the GPU), the GPU version is actually faster then the CPU version. However, to make a fair comparison, we have to include the data transfer time,

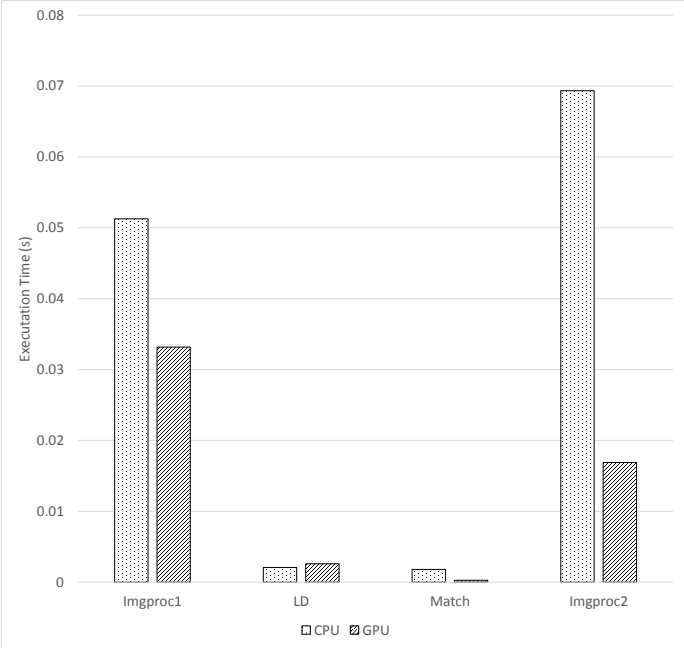

Figure 7.2. CPU vs GPU: Performance Comparison Of Ported Portion.

or the overhead. In this function, the overhead is the read1, read2 and write operation. Overhead cannot be avoided because it's part of the process. Before the LD starts, the data being processed must be copy from the host memory to the device memory. Which means the data of a 360 single channel image is being transfered. Recall that in order to reduce the complexity of the algorithm, we convert the image into a 1 dimension array which has 230400 elements. After the data copy is done, the computation starts. Recall that in the LD algorithm, besides the computation, the threads have to go though a if statement and series statements. Although there are 180 threads running at the same time, if one thread has more work to do, other threads have to wait until it finished its job. That's why the kernel execution time does not improve as much as the image processing part. As a result, the algorithm generates two arrays to store the result. They have the same size as the input array, 230400 elements each. To transfer these amout of data to the host memory, the GPU version is suffering from the long data read back time.

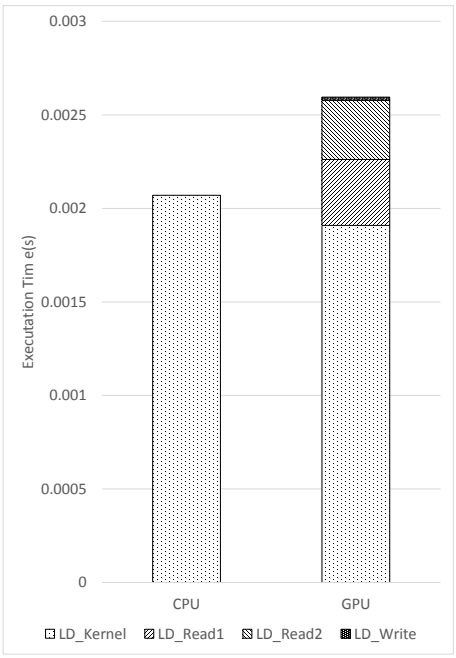

Figure 7.3. CPU vs GPU: Lane Detection.

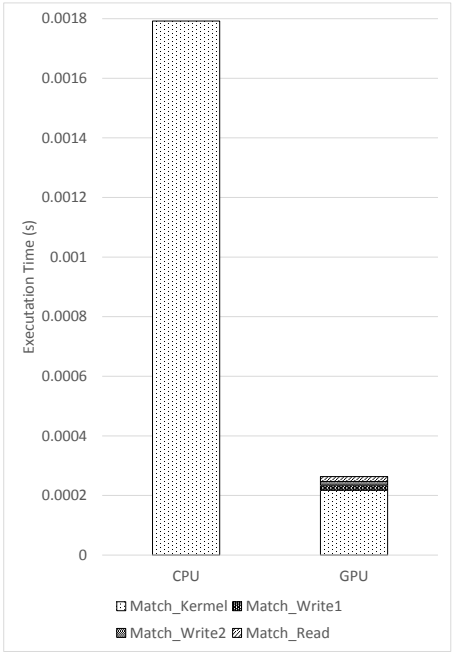

Figure 7.4. CPU vs GPU: Matching Algorithm.

Figure7.4 is the detailed comparison of the Matching function on CPU and GPU. Different from the Lane Detection, this algorithm runs much faster on the GPU then the CPU, even includes the data transfer time. This can be explained based on the data size and algorithm itself. If you compare the the amount of data that being transfer in Lane Detection algorithm and Matching algorithm, the answer is quite obvious. The data being transfered is much smaller. Only two 1D arrays are being passed to the OpenCL Kernel, one has 180 elements and the other has 1000 elements. As result, the data copy time read1 and read2 are very small. When the kernel runs, the 180 threads simultaneously starts the process. Different from LD, the computation in Match algorithm is much simpler. After 1000 loop, a new array is returned, which requires low data copy time.

If you take a close look to Figure 7.3 and Figure 7.4, you may notice something interesting about the data copy time in these two functions. For data writing time(copy data to device memory), the transferring time of LD and Match is almost identical even though the data size of LD is much larger then Match. What's more, with same amount of data, the data read back time in LD is much larger than the data write time. One of the possibility is that the OpenCL library released by QualComm is being modified. The $clEnqueWriteBuffer()$ function might automatically converted to $clEnqueMapBuffer()$, which is known as the "Zero Cpoy" function in OpenCL. What this function does is that when copying data from host to device, a pointer points to the data is being passed to the kernel instead of copying the data. In this case, the data copy time will be shorten significantly. However, this is only valid on APU because CPU and GPU shares the same memory. On desecrated GPU, the data will be copy to device anyway. To verify the assumption, we measured the data read/write time of the Matrix Multiplication test program. With varies input size, the average buffer write fluctuates around 0.000009 to 0.000023, but the buffer read back time increased along with the buffer size, see Table 7.3. This result proved our assumption.

Table 7.3. Buffer Read/Write Time Of Matrix Multiplication.

Size	WorkgroupSize	Write_1	Write_2	Kernel	Read
128×128	16×16	0.000007	0.000007	0.001629	0.000059
256×256	16×16	0.000012	0.000012	0.018122	0.000082
512×512	16×16	0.000018	0.000018	0.167533	0.000104
1024×1024	16×16	0.000007	0.000004	1.572792	0.000223
2048×2048	16×16	0.000006	0.000004	14.9273	0.004804

CHAPTER 8

CONCLUSION

This thesis presents the implementation of a navigation system based on our proposed lane detection algorithm and prediction algorithm. The result shows that these two algorithms are fast and accurate.

We also demonstrate that by porting applications to an embedded GPU on mobile processor, the performance of the application can be boosted significantly. Our test program, ARNavi, is 2.63 times faster then its CPU implementation. We also successfully integrate the OpenCL and OpenCV on Android platform. As of the writing of this thesis, there is no attempt on this. We believe that more and more computer vision applications will benefit from this combination of OpenCV and OpenCL.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Bu-353s4 gps receiver datasheet. http://usglobalsat.com/store/download/688/bu353s4_ds.pdf. Accessed: 2015-01-03.

[2] Convert a gps file to plain text or gpx. http://www.gpsvisualizer.com/convert_input. Accessed: 2015-01-10.

[3] Cuda parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2015-01-03.

[4] Gps - nmea sentence information. http://http://aprs.gids.nl/nmea/. Accessed: 2015-01-03.

[5] Java developer. https://developer.android.com/index.html. Accessed: 2015-01-03.

[6] Nmea 0183 standard. http://www.nmea.org. Accessed: 2015-01-03.

[7] Nmea reference manual. https://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual1.pdf. Accessed: 2015-01-03.

[8] Opencv official website. http://opencv.org/. Accessed: 2015-01-03.

[9] Sirf binary protocol reference manual. http://usglobalsat.com/downloads/SiRF_Binary_Protocol.pdf. Accessed: 2015-01-03.

[10] M. Bertozzi and A Broggi. Gold: a parallel real-time stereo vision system for generic obstacle and lane detection. *Image Processing, IEEE Transactions on*, 7(1):62–81, Jan 1998.

[11] Bradski, Gary R., and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. 2008.

[12] Kuo-Yu Chiu and Sheng-Fuu Lin. Lane detection using color-based segmentation. In *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, pages 706–711, June 2005.

[13] CHEN Huaizhong and JIN Zheliang. A method of lane detection and two dimensional reconstruction based on machine vision. *Sensors and Transducers (1726-5479)*, 160(12):645 – 652, 2013.

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

[15] Zhao Kun, M. Meuter, C. Nunn, D. Muller, S. Muller-Schneiders, and J. Pauli. A novel multi-lane detection and tracking system. University of Duisburg-Essen, Intelligent Systems Group, Duisburg, D-47057, Germany, 2012.

[16] S. Nedevschi, R. Schmidt, T. Graf, R. Danescu, D. Frentiu, T. Marita, F. Oniga, and C. Pocol. 3d lane detection system based on stereovision. In *Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on*, pages 161–166, Oct 2004.

[17] Bartlomiej Oszczak and Krzysztof Serzysko. Decoding of sirf binary protocol. *Artificial Satellites*, 46(4):127, 2011.

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

[19] R.Y. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *Robotics and Automation, IEEE Journal of*, 3(4):323–344, August 1987.

[20] M.A. Turk, D.G. Morgenthaler, K.D. Gremban, and M. Marra. Vits-a vision system for autonomous land vehicle navigation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):342 – 361, 1988.

[21] Bin Yu and A.K. Jain. Lane boundary detection using a multiresolution hough transform. In *Image Processing, 1997. Proceedings., International Conference on*, volume 2, pages 748–751 vol.2, Oct 1997.

[22] Wang Yue, Shen Dinggang, and Teoh Eam Khwang. Lane detection using spline model. *Pattern Recognition Letters*, 21(8):677 – 689, 2000.

[23] Wang Yue, E.K. Teoh, and D. Shen. Lane detection and tracking using b-snake. *Image and Vision Computing*, 22(4):269 – 280, 2004.

[24] Zhengyou Zhang. Flexible camera calibration by viewing a plane from unknown orientations. 1:666–673 vol.1, 1999.

VITA

Mengshen Zhao

EDUCATION

University of Mississippi, Oxford, MS USA **Jan, 2012 - May, 2015**

M.S., Computer and Information Science

Shenzhen University, ShenZhen, GD CHINA, **Sep, 2006 - Jun, 2010**

B.A., Biological Science

PUBLICATION

M. Hassan, M. Zhao, S. Son , H. Lee, H.Kim, B. Jang, **A Low Power and High Performance Face Detection on Mobile GPU**

HONORS AND AWARDS

Computer and Information Science SAP Awards 2014, University of Mississippi

Member of UPE Society (Upsilon Pi Epsilon)