Apply Modern Image Recognition Techniques with CUDA Implementation on Autonomous Systems

Apply Modern Image Recognition Techniques with CUDA Implementation on Autonomous Systems

By YICONG LIU

B.Eng.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Master of Applied Science

McMaster University

McMaster University	MASTER OF APPLIED SCIENCE (2017)
Hamilton, Ontario	Mechanical Engineering
TITLE:	Apply Modern Image Recognition Techniques with CUDA Implementation on Autonomous Systems
AUTHOR:	YICONG LIU, B.Eng.
Supervisor	Professor Fengjun Yan Department of Mechanical Engineering
NUMBER OF PAGES	xiii, 73

ABSTRACT

Computer vision has been developed rapidly in the last few decades and it has been used in a variety of fields such as robotics, autonomous vehicles, traffic surveillance camera etc. nowadays. However, when we process these high-resolution raw materials from the cameras, it brings a heavy burden to the processors. Because of the physical architecture of the CPU, the pixels of the input image should be processed sequentially. So even if the computation capability of modern CPUs is increasing, it is still unable to make a decent performance in repeating one single work millions of times.

The objective of this thesis is to give an alternative solution to speed up the execution time of processing images through integrating popular image recognition algorithms (SURF and FREAK) on GPUs with the help of CUDA platform developed by NVIDIA, to speed up the recognition time.

The experiments were made to compare the performances between traditional CPU-only program and CUDA program, and the result show the algorithms running on CUDA platform have a significant speedup.

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my supervisor Dr. Fengjun Yan for his guidance in computer vision. This thesis would not have been possible without the knowledge and inspiration I acquired from him.

Many thanks to the Department of Mechanical Engineering of McMaster University, where I learn not only the advanced skills and knowledge but also the inspiration thoughts.

I would like to thank my friends for accepting nothing less than excellence from me. Last but not the least, I would like to thank my family: my parents and to my brothers and sister for supporting me spiritually throughout writing this thesis and my life in general.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
CUDA	Compute Unified Device Architecture
SM	Streaming Processor
SIMD	Single Instruction Multiple Data
LADAR	LAser Detection and Ranging
SURF	Speeded Up Robust Features
FREAK	Fast Retina Keypoint

CHAPTER 1: INTRODUCTION

1.1. Computer Vision

Computer vision is a field that combines biological science and engineering and studies how to reconstruct, interpret and understand a three-dimensional scene from its two-dimensional images. The goal of computer vision is to model and mimic the visual system of human beings through computer software and hardware and build autonomous systems [1][2][3].

Researchers in computer vision have been developing mathematical techniques for replicating the visual system of a human being on computers and rebuild the three-dimensional world from two-dimensional images. However, replicating such visual system has been proven to be difficult. Letting our computers to understand a picture at the same level as a human being, even a child, is still being considered as an untouchable goal. Part of the reasons are that computer vision is an inverse problem, which means computers need to rebuild the unknowns without enough information given, and lots of details of the real world would be lost during the formation of 2D images [4].

Beginning in the late 1960s, the concept of computer vision first appeared at universities pioneering artificial intelligence and robotics [4]. It originally aimed to

mimic the visual system of human beings, as a stepping stone to endowing robots with intelligent behavior. In 1966, Gerald Jay Sussman in MIT achieved it in his summer project, by equipping a computer with a camera and having the computer "describe what it saw" [5][6].

The main difference between computer vision and the pre-existing field of digital image processing at that period was a desire to obtain the three-dimensional structure from images to reach the level of full scene understanding [4]. In the 1970s, researches in computer vision formed the stepping stone for many of the algorithms existing today, for example detecting edges from images [7], non-polyhedral objects modeling [8][9], labeling of lines [10] and recovering 3D structure and camera motion [11][12].

In the 1980s, more rigorous mathematical analysis were implemented at the quantitative aspects of computer vision, including the concept of scale-space, the derivation of shape from various information such as shading and texture.

A significant development in computer vision was achieved during the 1990s, which was the increased interaction between the fields of computer graphics and computer vision [13]. With the help of image morphing techniques [14], we could create new images by manipulating the pictures from real world directly. Later applications like view interpolation and panoramic image stitching were all benefit from it.

Nowadays, the trend of computer vision is the connection between feature based methods and machine learning techniques. With the help of machine learning techniques, it's possible to learn objects more efficiently and even without human supervision. New approaches to improve the efficiency of the learning and recognition processes through both hardware and software are also proposed frequently [15][16].

As Table 1.1.1 shows, computer vision system can be divided into three levels from low to high by the information gained from it. Higher-level computer vision systems rely on low-level processes to perform accurately. For example, high-level operation, for instance object recognition, needs the information (features) extracted from low-level processes (feature detection and feature description).

Hierarchy	Description
Low-level	Process image for feature extraction (edges, corners or blobs).
Middle-level	Object recognition, motion analysis, and 3D reconstruction. Using features obtained from the low-level vision.
High-level	Interpretation of the evolving information provided by the middle-level vision as well as directing what middle and low-level vision tasks should be performed.

Table 1.1.1 Computer Vision Hierarchy

1.2. Computer Vision on Autonomous Vehicles

An autonomous car is a vehicle that can sense and recognize its surrounding environment and navigate without human input [17]. For human beings, we can easily obtain the information of our surrounding environment by the senses like vision and hearing. However, for autonomous vehicles, they gather the environment information from sensors. Thus, the quality and comprehensiveness of the data coming from sensors play an important role for autonomous vehicles [18].

1.3. History of Autonomous Vehicles

The history of autonomous vehicles can be dated back to the 1920s. A radio-controlled vehicle called “linrrican wonder” was demonstrated in 1925 on New York City streets. The linrrican wonder was equipped with a transmitting antennae and was operated by a second car that followed it. The following car was used to send out radio signals that could be caught by the transmitting antennae, then the antennae transferred the signals to circuit-breakers that directed the movements of the vehicle [19].

Promising experiments of autonomous vehicles took place in the 1950s. In 1953, RCA Labs produced a miniature car that was controlled by wires laid in a pattern on a laboratory floor. This system inspired some engineers and they decided to implement

such system in actual highway installations. In 1958, a full size system was developed successfully. In that system, the vehicle received the signals send by a series of detector circuits which were buried along the edge of the street. With the help of General Motors, the system were equipped with special radio receivers and visual devices to simulate automatic steering, accelerating and brake control [20].

The first self-sufficient and truly autonomous cars appeared in the 1980s. A vision-guided Mercedes-Benz robotic van successfully ran on streets without traffic at a speed of 63 kilometers per hour [21]. In the same decade, the Autonomous Land Vehicle (ALV) project in the United States achieved the first road-following system using lidar, computer vision and autonomous robotic control to direct a robotic vehicle at speeds up to 31 kilometers per hour [22][23]. HRL Laboratories developed the first off-road map and sensor based autonomous navigation on the ALV project. The car successfully passed 610 meters with various terrain. Since then, lots of companies and research organizations have developed working prototype autonomous vehicles.

In 1996, Alberto Broggi started a project which worked on enabling a modified traditional vehicle to follow the normal lane marks in an unmodified highway [24]. This vehicle only had two camera sensors and used stereoscopic vision algorithms to obtain the information of its surrounding environment.

In the 2010s, major automotive manufacturers, for example General Motors, Audi,

Mercedes Benz, and BMW are building their own driverless vehicle systems. Besides, some internet companies began to show the interests in autonomous vehicles. Google began developing its self-driving vehicles in 2009 [25].

In July 2013, some of the states in U.S. have allowed autonomous vehicles in traffic. Back then, fully autonomous vehicles are not yet available to the public, the car models only have limited autonomous functions, such as adaptive cruise control, lane assist and parking assistant [26]. In 2014, SAE (Society of Autonomous Engineers) published a standard for autonomous vehicles which identified six levels of driving automation, as shown in Figure 1.3.1.

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	No Automation	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	Conditional Automation	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	High Automation	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	Full Automation	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Figure 1.3.1 The Levels of Automation [27]

In October 2016, Tesla Motor announced that the tesla vehicles were built with the necessary hardware to allow the automation capability to reach SAE Level 5 (Full Automation). However, full automation is only likely after millions of miles of testing, and approval by authorities [28].

1.4. Censors on Autonomous Vehicles

Autonomous Vehicles use different sensors to obtain information of the surrounding environment. The most commonly used sensors are LADAR (LAser Detection and Ranging) [29], electromagnet wave radar, ultrasonic radar and passive vision sensor. Different kinds of sensor have different defects brought by their structure and mechanism. Figure 1.2.1 to Figure 1.2.4 illustrate the evaluations of these sensors in ten different aspects through a decagon representation.

LADAR, which stands for laser detection and ranging, is a measuring device that detects the distance to an object through a laser light. As Figure 1.4.1 shows, LADAR has a good resolution and is capable of working in both day and night circumstance, but it will be disrupted easily by rain drops and snow. LADAR is also good at detection objects in long distance, however it could not detect obstacles within a short range. The most important factor which limits the popularization of LADAR is the cost. The price

of one LADAR device ranges from tens to even thousands times more than a normal electromagnet wave radar.

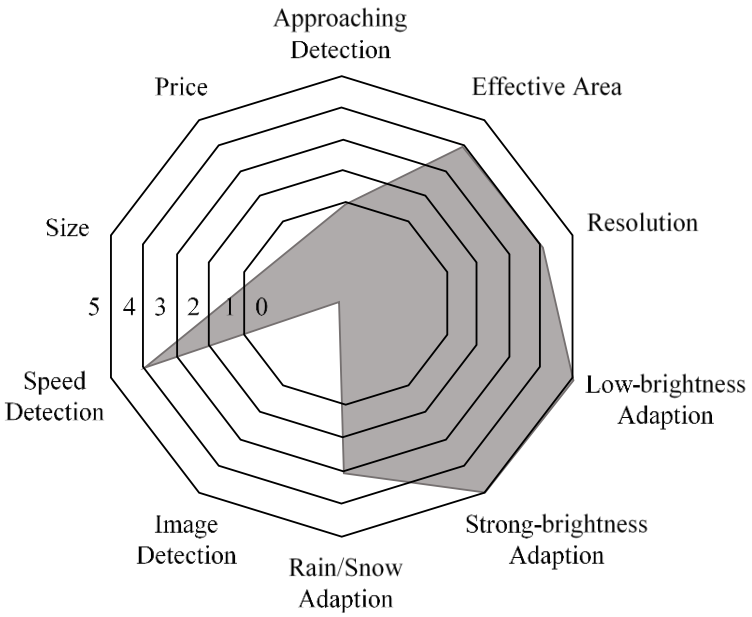


Figure 1.4.1 Evaluations of LADAR

Electromagnet wave radar uses the radio wave with higher wave length to determine the range of objects. Figure 1.4.2 shows the evaluations of the electromagnet wave radar. Compared to LADAR, electromagnet wave radar has a good adaption to different brightness, besides, it can work properly in extreme weather situation. Electromagnet wave radar is very effective in detecting speeds. Although the precision is slightly lower than LADAR, electromagnet wave radar has been widely used in autonomous vehicles due to its good performance and most importantly, its relatively lower price.

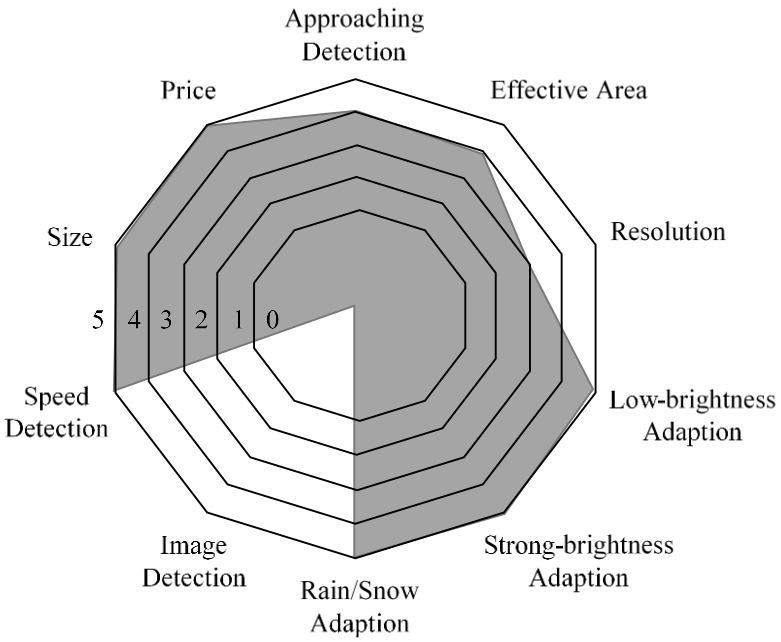


Figure 1.4.2 Evaluations of Electromagnet Wave Radar

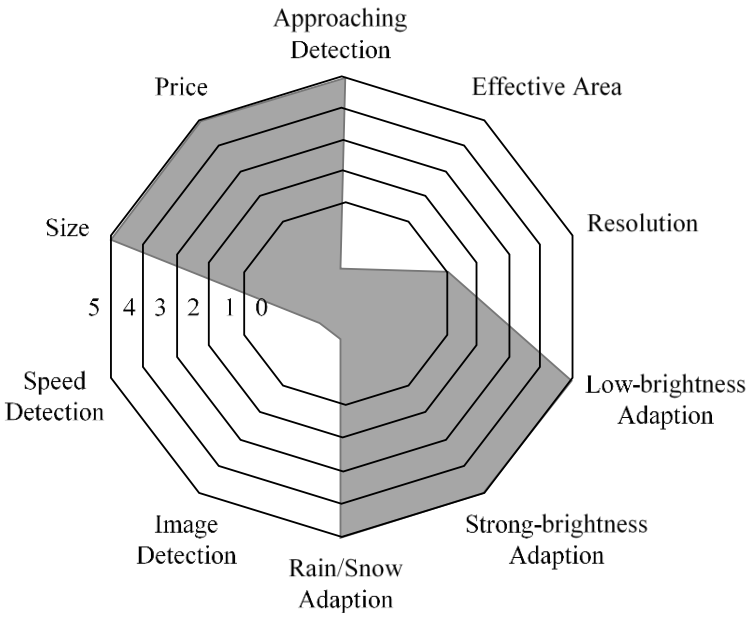


Figure 1.4.3 Evaluations of Ultrasonic Radar

Ultrasonic radar is one kind of radars that acquires the distance information from the obstacle to the radar by emitting a beam of ultrasonic wave [30]. As shown in Figure 1.4.3, ultrasonic radar is good at detecting the nearby objects, and it can be easily equipped onto a vehicle. However, it cannot detect the speed of the objects.

As mentioned above, these radars neither cannot recognize complicated shapes nor detect colors, which means an autonomous vehicle can hardly read street signs or understand street signals only by radars. Therefore, we need passive vision sensor to be the ‘eyes’ of the autonomous vehicles [31]. Passive vision sensors, in short, are the cameras, which can record digital information of surroundings. After pictures taken from the cameras, computer vision will be applied as a complementary method through analyzing the images taken from censor cameras to further increase the reliability and intelligence of autonomous vehicles by providing the abilities of the street sign and street signal recognition and unexpected obstacles detection such as pedestrians on the street, under the conditions where active vision techniques say radar or LADAR can not perform well.

Figure 1.4.4 pictures the evaluations of the passive vision censor. Similar to human beings’ eyes, the performance of the passive vision censor will be affected by the brightness and weather. But it is able to detect and distinguish complicated shapes and

colors, which is impossible for the other three kinds of radars.

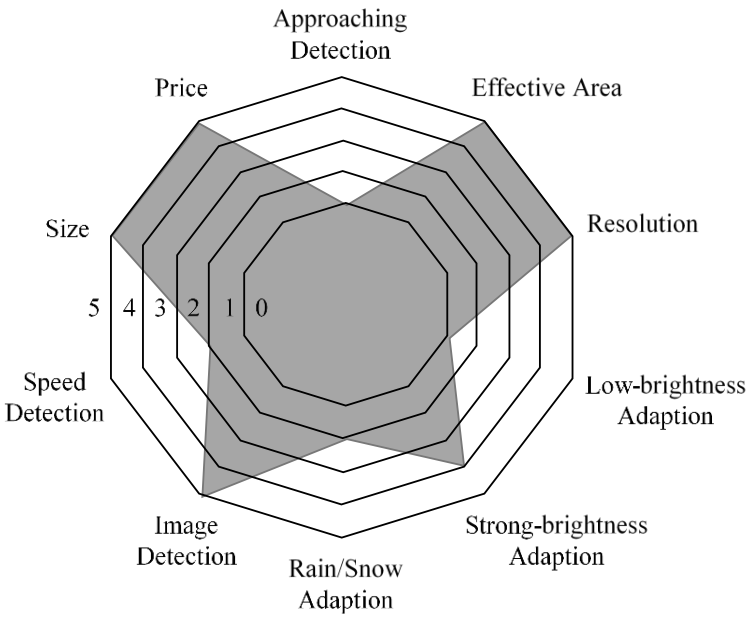


Figure 1.4.4 The Evaluations of Passive Vision Sensor

Figure 1.4.5 illustrates a combination of Figure 1.4.1 to Figure 1.4.4. To build a reliable autonomous vehicle system, it is necessary to combine the information acquired by these censors to eliminate blind spots of one specific censor. As an example, Figure 1.4.6 shows the how the various sensors been applied to a Google autonomous vehicle to establish a reliable autonomous driving system.

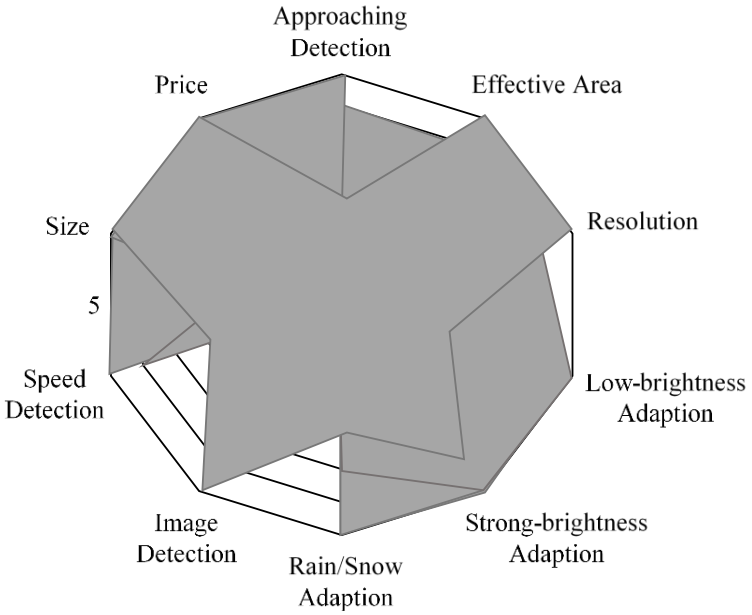


Figure 1.4.5 Combination of the of Four Sensors

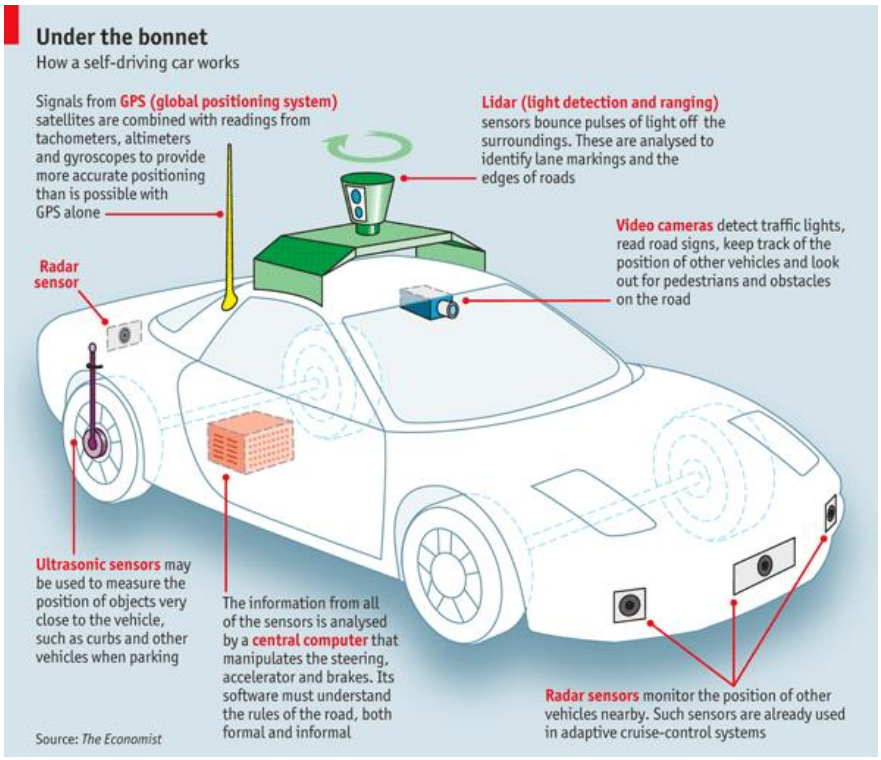


Figure 1.4.6 Google Autonomous Vehicle

1.5. Other Applications

Besides the autonomous car, computer vision is being applied nowadays in a wide variety of real-world applications, such as:

- **Surveillance:** monitoring for intruders, analyzing highway traffic, and monitoring pools for drowning victims [32];

- **Optical Character Recognition (OCR):** reading words like handwritten postal codes on letters and automatic number plate recognition [33];

- **Machine inspection:** rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts or looking for defects in steel castings using X-ray vision [34];

- **3D model building (photogrammetry):** fully automated construction of 3D models from aerial photographs used in systems such as Google Maps;

- **Match move:** merging computer-generated imagery (CGI) with live action footage by tracking feature points in the source video to estimate the 3D camera motion and shape of the environment. Such techniques are widely used in Hollywood; they also require the use of precise matting to insert new

elements between foreground and background elements [35].

- **Motion capture (mocap):** using retro-reflective markers viewed from multiple cameras or other vision-based techniques to capture actors for computer animation [36].

- **Fingerprint recognition and biometrics:** for automatic access authentication such as electronic device login, as well as forensic applications [37].

1.6. GPU Programing

Computer Vision plays a key role in autonomous vehicles. To increase the reliability and safety of the autonomous system, the system have to acquire and process the environment information as fast as possible. Thus, it requires a high computation ability of the processors.

Modern processors are made from transistors, and each year those transistors get smaller and smaller. Figure 1.6.1 shows the feature size of processors over time, where the feature size is the minimum size of a transistor or wire on a chip. We see that it's consistently going down over time. As the feature size decrease, transistors get smaller,

run faster, use less power and we can put more on a chip. The consequence is that we have more resources for computation every single year. However, as transistors are improved, processor designers would then increase the clock rates of processors, running them faster and faster every year. Figure 1.6.2 illustrates the clock speeds over the years. Over many years, clock speeds continue to go up, but over the last decade, we see that clock speeds have essentially remained constant. The reason why we're not increasing clock rate is not that transistors have stopped getting smaller and faster. Even though transistors are continuing to get smaller and faster and consume less energy per transistor, the problem is running a billion transistors generates an awful amount of heat and we cannot keep all these processors cool.

In computer vision, most of the times we need to perform the same kinds of operations on every single pixel, and the total pixel number is increasing tremendously as the camera technique developed rapidly over these years. This requires processors to be parallelism, which is the capability to execute the same kind of operations on thousands of pixels at the same time, to accelerate the computation. However, as Figure 1.6.3 shows, the structures between CPUs (Central Processing Unit) and GPU (Graphics Processing Unit) are different: CPUs have more powerful ALUs (Arithmetic Logic Unit) while GPUs own much more numbers of fewer efficiency ALUs.

The reasons for these discrepancies between the CPU and GPU exist mainly

because of the objective for the CPU and GPU. CPUs are designed to minimize latency, which makes CPUs able to complete a single task very efficiently. However, GPUs are designed to increase throughputs, which means GPUs will execute more tasks in a period. That is to say, GPU is specialized for compute-intensive, highly parallel computation.

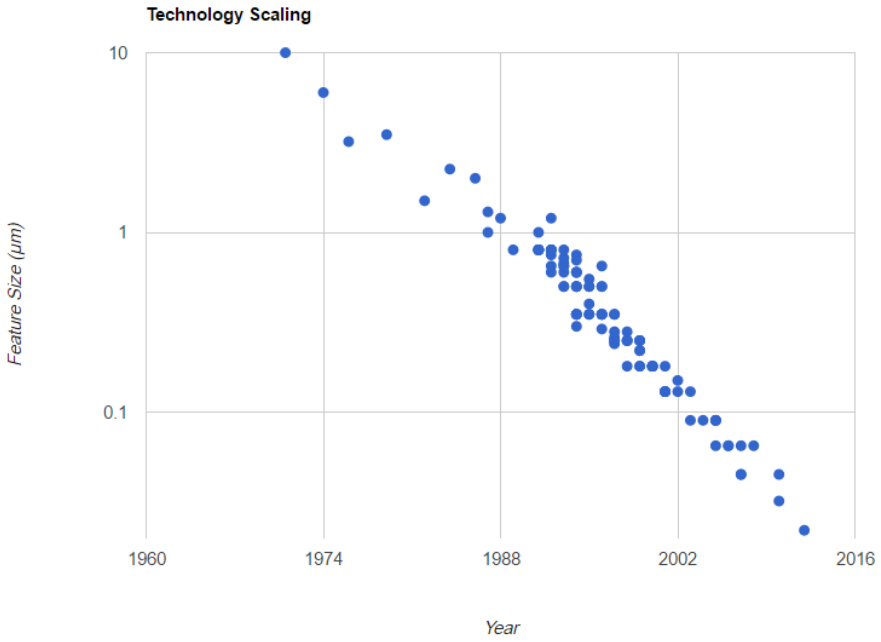


Figure 1.6.1 Feature Size of Processors Over Time [38]

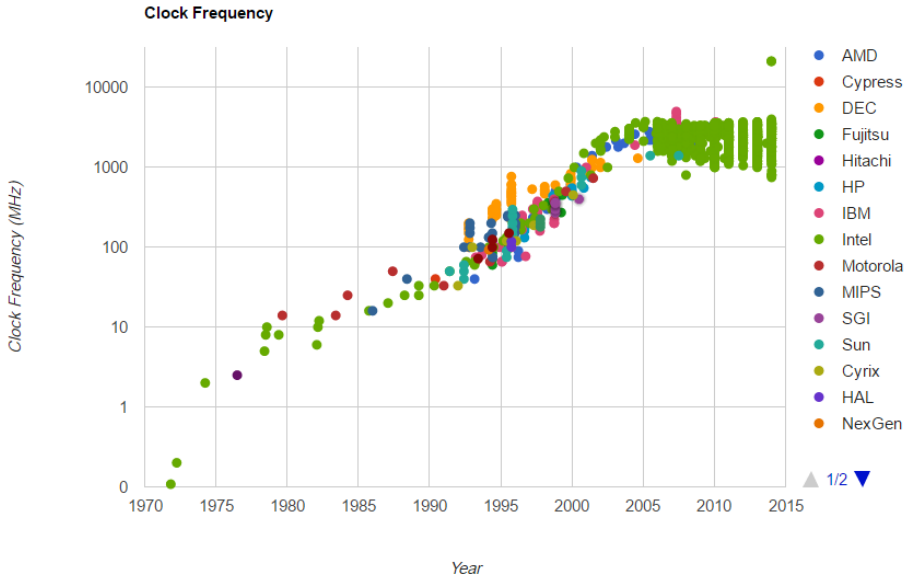


Figure 1.6.2 Clock Frequency of Processors Over Time [38]

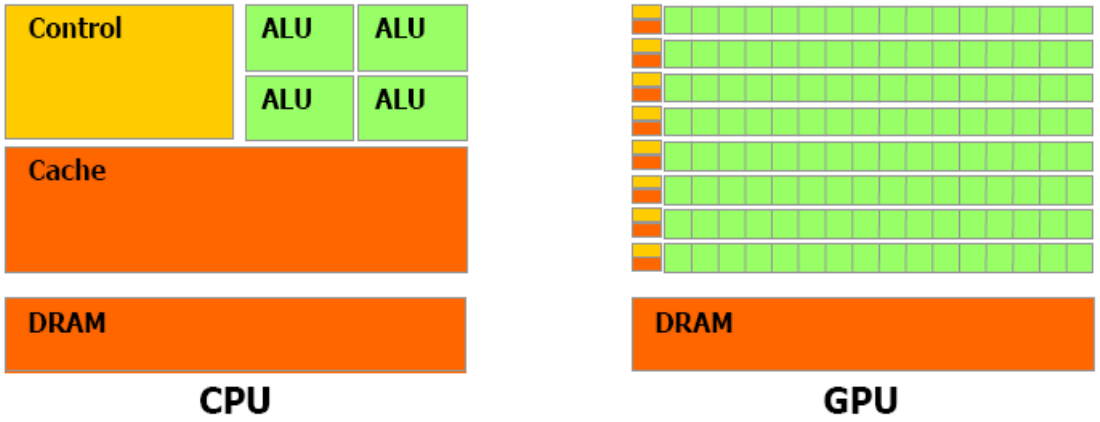


Figure 1.6.3 Structures of CPU and GPU [39]

GPUs have become increasingly programmable over the past few decades. NVIDIA has led the field in parallel computing with their intuitive software, CUDA (Compute Unified Device Architecture), and highly optimized GPGPU (General Purpose Graphics Processing Unit) hardware. The difference between GPU and

GPGPU is that data can only be transferred `from the host CPU to the GPU while GPGPUs allow for data transfers from the host CPU to the GPGPU and vice-versa to perform parallel computations. Note that in the rest of this thesis, when we mention GPU, it refers in particular to GPGPU.

In this thesis, we will discuss the implementation of modern image recognition algorithms (SURF and FREAK) on the NVIDIA GPU by utilizing the CUDA software platform. High-speed feature point detection and description are in high demand for computer vision systems in applications such as motion detection, video tracking, augmented reality, and object recognition.

1.7. Novelty and Contribution

This thesis presents an alternative solution to speed up the execution time of processing images through integrating popular image recognition algorithms on GPUs with the help of CUDA platform developed by NVIDIA.

Feature detection and description algorithms, which have been proved to be the high efficient image recognition approaches, are successfully been implemented on CUDA platform.

Details of the parallel implementation are given, and further performance improvement strategies are discussed, including the memory utilization and block distribution strategy.

1.8. Thesis Outline

This thesis consists of 6 chapters.

Chapter 1 gives a brief introduction to computer vision and GPU development, introduces the fundamental knowledge of computer vision history and differences between CPU and GPU. Novelties and contributions are clarified here.

Chapter 2 and Chapter 3 present the feature detection and description algorithms respectively. And the methods we implement them on CUDA platform.

In Chapter 4, further details and explanations focusing on the GPU programming. NVIDIA CUDA platform is introduced to implement GPU programming. Discussions are made for the advantages and challenges of CUDA, and the strategies to improve performances.

In Chapter 5, experiments were carried out with detailed discussion and analysis to demonstrate their performances.

The last Chapter 6 consists of the conclusion and future work recommendation, which illustrates a potential possibility for later research.

CHAPTER 2: FEATURE DETECTION

2.1. Feature Detection Overview

Feature detection is a low-level operation in the image processing which makes decisions at every pixel whether this pixel can be considered as a certain type of feature by calculating abstractions of image information, and it is usually performed on an image as the initial step [40].

The definition of "feature" depends on the problem or the type of application, so a feature is defined as a part of the given image that we interested in. In general, there are mainly three types of features: edges, corners and blobs.

The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges [41]. In general, an edge can be of almost arbitrary shape. A large amount of edge detection algorithms have been introduced such as Canny edge detector [42], Sobel operator [43] and Prewitt operator [44].

Corners, or interest points, refer to point-like features in an image. It's called "Corner" since in the early algorithms, in order to find interest points, people firstly applied edge detection, and then analyzed the edges to find rapid changes in direction, which are corners. However, as the algorithms developed, the corner features no longer have to be corners in the traditional sense, for instance a small dark spot on a bright

background may also be detected as corner point as well. Well-known corner detection algorithms include Harris operator [45], FAST [46], SIFT [47][48], SURF [49], etc.. In this thesis, we use SURF as the detection algorithm because of its accuracy and computational efficiency. A brief introduction to SURF algorithm will be given in the following sections.

Blobs can be considered as a bunch of points that have the similar properties. Blob detectors can find regions of an image in which properties are constant or similar, which is not obtained from edge or corner detectors, thus blob detectors can provide complementary information of image structures. Differential methods and local extrema are the two most used methods in blob detectors. Laplacian of Gaussian (LoG), Difference of Gaussians (DoG), Determinant of Hessian (DoH), Maximally Stable Extremal Regions (MSER) [50] are popular blob detection algorithms.

2.2. Scale Space Theory

Scale space is an important and widely used concept in the fields of computer vision and image processing. The meaning of one specific object in the real world depends on the scale of observation. For example, a piece of leaf can be a meaningful entity within the scales from a few centimeters to five to six meters at most. It hardly

makes any sense to describe the leaf object at the nanometer or the kilometer level. Similarly, when analyzing an image, our objective is to extract the structures not only in one scale but also the other information in different scales.

Thus, the scale space theory is developed to analyze image structures in different scales. The basic idea of scale space theory is representing an image with a family of smoothed images with multi-scales. Each image in this family contains the information from different scales of observation [51].

Lots of works have been done to find the way representing the image in different scales. One of the most widely used approaches is pyramid representation [52][53][54]. As Figure 2.2.1 shows, in the pyramid representation, an image is subjected to repeated subsampling and smoothing. The smoothing operation is utilized to reduce the affect brought by sub-sampling to the coarser scale images.

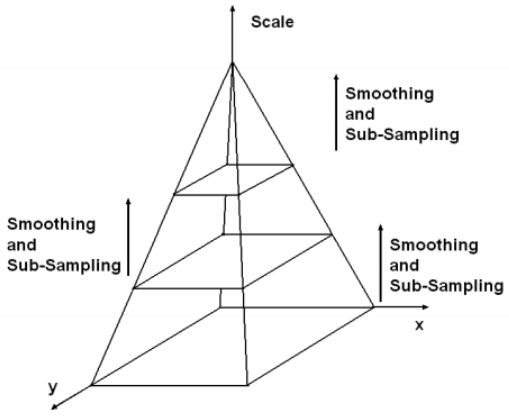


Figure 2.2.1 Scale Space Pyramid Representation

Pyramid representations have become popular and been applied in the fields of data compression, pattern matching, image analysis, etc. The biggest advantage of pyramid representations is that due to the sub-sampling operation, less data have to be computed.

2.3. Integral Image

Proposed by Viola and Jones in 2001 [55], an integral image is capable of calculating summations over image sub-regions rapidly. In the integral image, every pixel is the summation of the pixels before it (above and to the left), the location $\mathbf{x} = (\mathrm{x}, \mathrm{y})^T$ of an integral image $I_{\sum}(x)$ represents the sum of all pixels in the input image I within a rectangular area defined by

$$I_{\sum}(x) = \sum_{i=0}^{i \le x} \sum_{j=0}^{j \le y} I(i,j)\,.$$

As Figure 2.3.1 shows, the sum of intensities inside the rectangular area S can be calculated as $\mathrm{S} = \mathrm{C} - \mathrm{B} - \mathrm{D} + \mathrm{A}$. It only takes three additions and four memory

accesses, which will massively decrease the computational complexity.

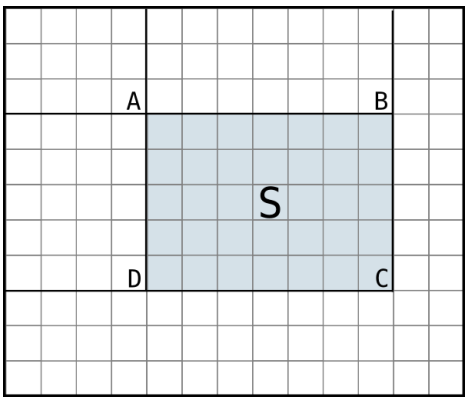


Figure 2.3.1 Integral Image Calculation Example

2.4. SURF Detector

As the methodology proposed by Lindeberg in 1998 [56], Hessian determinant is computed in SURF for automatic scale selection. Hessian matrix describes the 2nd order image intensity variations around the selected pixel and is widely used in to analyze the image structures. In a 2D image $\mathbf{I}$, the Hessian matrix of a given point $\boldsymbol{p}(\mathrm{x},\mathrm{y})$ at scale σ can be defined as

$$\mathrm{H}(\boldsymbol{p},\sigma)=\begin{bmatrix} L_{xx}(\boldsymbol{p},\ \sigma) & L_{xy}(\boldsymbol{p},\ \sigma) \\ L_{xy}(\boldsymbol{p},\ \sigma) & L_{yy}(\boldsymbol{p},\ \sigma) \end{bmatrix},$$

where $L_{xx},\ L_{xy},\ L_{yy}$ are the Gaussian second order derivatives calculated using Gaussian kernels of deviation σ. However, the Gaussians need to be discretized and

cropped due to the discontinuity of pixels.

As Figure 2.4.1 shows, the original Hessian matrix filter will be considered as the approximate Hessian matrix with box filter since the box filters can be computed using integral image with higher efficiency.

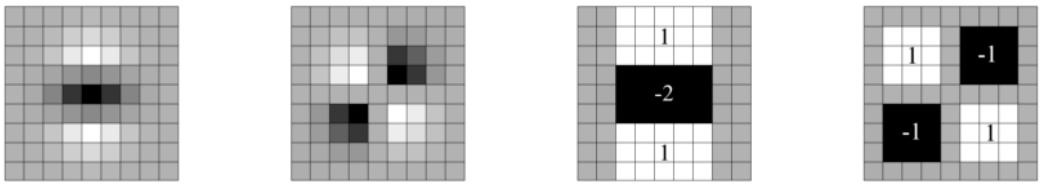


Figure 2.4.1 SURF Approximate Hessian Matrix [49]

To detect feature points in different scales, SURF applies an "up-side-down" scale pyramid (Figure 2.4.2) approach instead of iteratively reducing the image size. This approach analyses the scale space by up-scaling the filter size, thus it can make use of the integral image to gain more computational efficiency. The scale pyramid consists of several octaves and each octave contains four layers with different scales as Figure 2.4.3 illustrates. Note that the octaves are overlapping in order to cover all possible scales seamlessly.

The approximate hessian matrix will then be applied to each layers to get the responses. The SURF feature points candidates can be obtained by applying 3D non maximum suppression [57] on both spatially and over the neighboring scales within

one octave. Finally, those candidate points are interpolated in scale and image space with the method proposed by Brown et al. [58].

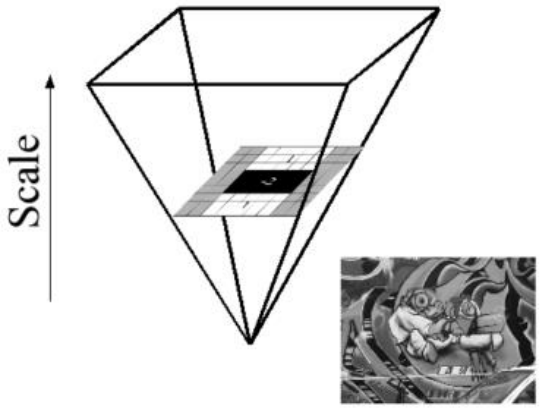


Figure 2.4.2 SURF scale space pyramid [49]

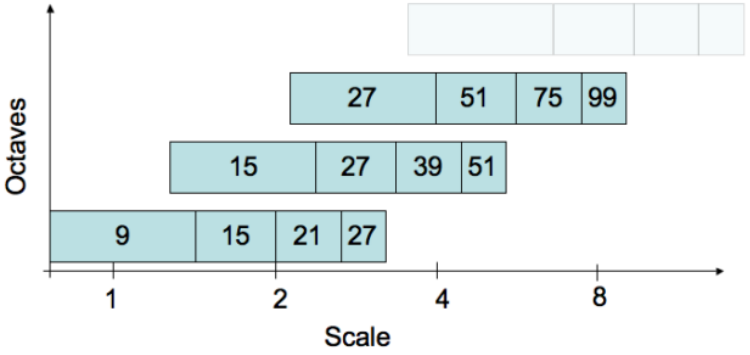


Figure 2.4.3 Graphical representation of the scales for three different octaves [49]

2.5. SURF Feature Detection Result

Figure 2.5.1 shows the SURF feature detection result. The green nodes represent the pixels which are detected as feature points. Mainly the points in the corner are selected as feature points. Note that there are around with the hessian response threshold of 0.0002, but we only draw 50 points with the highest response on Figure 2.5.1 to have a clear look.

Figure 2.5.1 SURF Feature Detection Result

CHAPTER 3: FEATURE DESCRIPTION

3.1. Feature Description Overview

A feature descriptor is an algorithm, which takes an image and outputs feature descriptors or feature vectors. It encodes features into a series of numbers and act as a sort of numerical ID that can be used to differentiate one feature from another. Such feature descriptors have been widely used in the fields such as object recognition [48], texture classification [59], generation of panoramas [60], and image indexing [61].

A descriptor is built by transferring the characteristics of the local region of pixels around the feature points into digital vector. The descriptor generated from one feature point will be different depending on the chosen descriptor. The objective of the descriptor is to compactly represent these features while being unique at the same time.

One of the frequently used methods to obtain descriptors is by representing the characteristics of the local area as a histogram, and transfer the histogram into a digital vector, that is feature descriptor.

Johnson and Herbet [62] presented a 3D shape-based object recognition system for multiple objects in cluttered scenes. Its recognition is based on matching surfaces by matching points using spin image. The spin image is a mapping of relative positions of 3D surface points to a 2D plane. 2D histogram of the local neighborhood around

each surface point was used to create spin images. Lazebnik et al. [59] used this concept and extended it to texture classification. In this algorithm, a histogram of pixel and intensity values was generated by encoding a normalized image patch.

In 2002, Belongie et al. [63] proposed shape context, which is able to match corresponding areas between two images by using their shape information. In this algorithm, a log polar histogram of edge point locations and orientations is computed where the locations are described relative to the reference point.

DG Lowe [47][48] introduced a novel descriptor called Scale-Invariant Feature Transform (SIFT), which used the gradient information around the feature point to generate orientation histograms. And it was proved to have good robustness and speed performance among other local descriptors. [64] Based on this, in 2008, Herbert et al. [49] proposed SURF. SURF utilized integral image to speed up the computation time.

Another novel method to build descriptors is binary descriptors. The core idea of binary descriptor is to encode most of the information of a patch as a binary string using only comparison of intensity values. This will further speed up the computation time of description as only intensity comparisons need to be made and the matching between two binary strings can be done fast using Hamming distance. Many feature description algorithms based on binary descriptor are proposed in the recent decade, such as BRIEF [65], ORB [66], BRISK [67] and FREAK [68].

In general, a binary descriptor is composed of three parts:

1) Sampling pattern which is a small patch centered around a feature point used to pick sampling points.

2) Orientation compensation. To make the descriptor invariant to rotation, we need to calculate the orientation of the feature point and rotate it by the degrees.

3) Sampling pairs used to make intensity comparison and build the descriptor.

In this thesis, FREAK (Fast Retina Keypoints) is chosen as the feature descriptor, and further discussion will be presented in the following section.

3.2. FREAK Descriptor

The FREAK (Fast Retina Keypoints) descriptor proposed by Alexandre Alahi and Raphael Ortiz [68] recently has become one of the most widely used binary descriptors. And it's proved to be much more efficient than normal HOG based descriptors.

As introduced previously, sampling patterns are used to build descriptors by making comparison between feature points and sampling points. Different algorithms have different sampling patterns, BRIEF applies random pairs, ORB uses learned pairs,

while BRISK uses a circular pattern where points are equally spaced on concentric circles.

Inspired by human retina areas (Figure 3.2.1), FREAK suggests to apply the retinal sampling grid. As shown in Figure 3.2.2, FREAK pattern is also circular, with a higher density of sampling points near the center. To get the descriptors, this pattern will be applied to all detected feature points. The small black dots are the sampling locations while the red circles are drawn at a radius corresponding to the standard deviation of the Gaussian kernel used to smooth the intensity values at the sampling points.

Figure 3.2.3 shows all the pairs generated in the FREAK sampling pattern. The selected pairs are segmented into four clusters corresponding to four areas in human retina: perifoveal, parafoveal, fovea and foveal. As Figure 3.2.3 illustrates, the first group of pairs mainly locate at the outer circular area of the pattern while the last pairs compare mainly points in the inner rings of the pattern. Each group has 128 sampling pairs. This approach mimics the way that our human retina works: higher resolution is captured in the fovea area while the perifoveal area forms an estimation of the object.

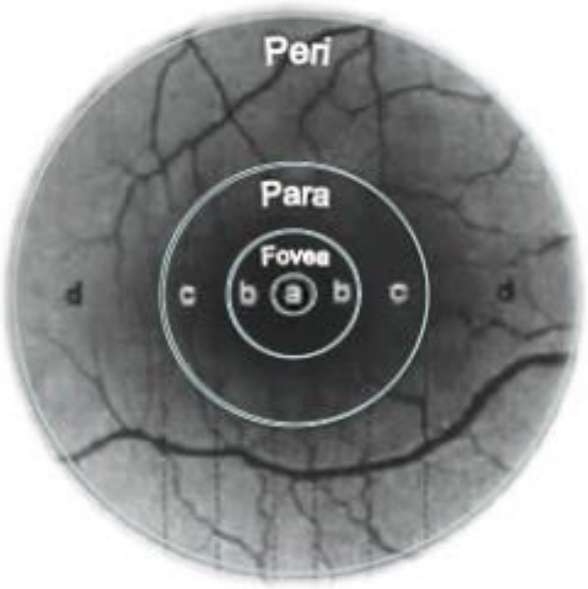


Figure 3.2.1 The Human Retina [68]

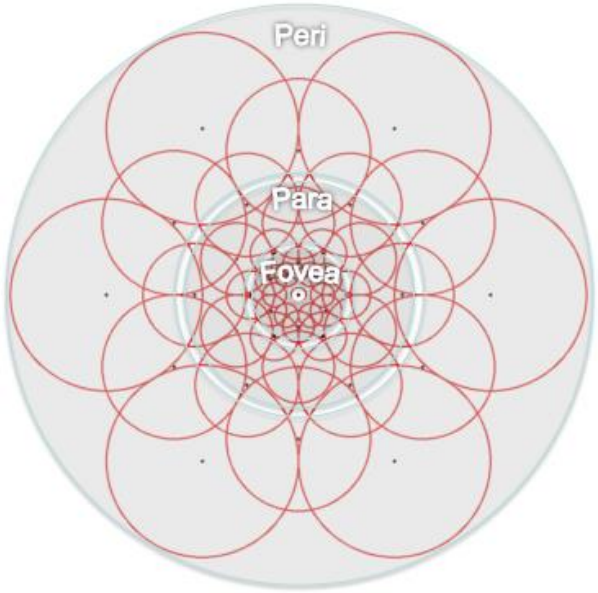


Figure 3.2.2 FREAK Sampling Pattern [68]

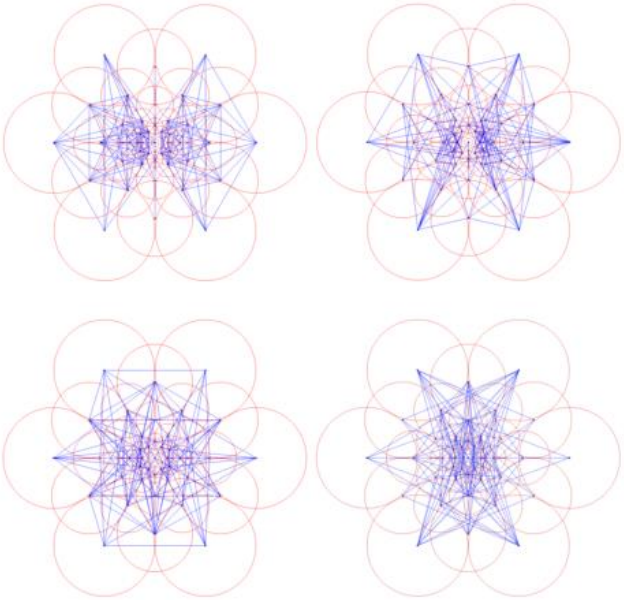
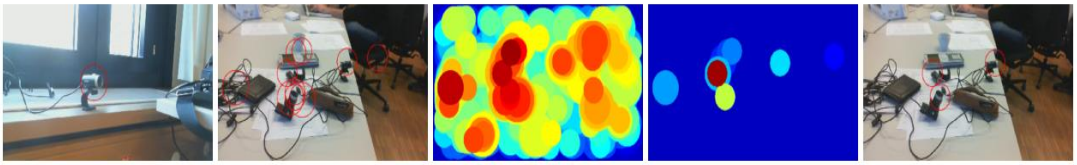


Figure 3.2.3 FREAK Sampling Pairs [68]

3.3. Saccadic Search

In the matching step, FREAK takes the advantages of the coarse-to-fine structure to speed up matching by applying a cascade approach. It first makes comparison between the pairs located in the perifoveal section, which is the first 128 bits of the descriptor. If the distance is smaller than a threshold, it continues to compare the next 128 bits in parafoveal area. As a result, more than 90% of the candidate pairs are discarded in the first step of comparison, thus these candidates would not cost additional

computation. Figure 3.3.1 Illustration of Cascade Search [68].



(a) An object of interest described by 1 FREAK (b) Best matches after 1^{st} cascade (c) Distance map with first cascade (in color jet) (d) Distance map with last cascade (in color jet) (e) Matched objects after last cascade

Figure 3.3.1 Illustration of Cascade Search [68]

3.4. FREAK Feature Description Result

The FREAK description and matching result is shown in Figure 3.4.1. The target image size is 600*800 pixels, and the processing time is 511ms on average. Note that not all the matching points are from the points shown in Figure 2.5.1. This is because Figure 2.5.1 didn't show all the feature points found through SURF algorithm as mentioned before. The matching result is decent even the object in the target image has some rotations and is taken from a different perspective.

Figure 3.4.1 FREAK Feature Description and Matching Result

CHAPTER 4: GPU PROGRAMMING

4.1. Nvidia GPU Introduction

As mentioned in Chapter 1, starting from the late 1990's, the NVIDIA GPU had become increasingly programmable. Since the overwhelming performance benefits over sequential computation, many developers began to run their graphical systems with GPU. Parallel computation has been more and more welcomed in the past few decades.

Driven by the insatiable market demand for real-time, high-resolution graphic, the Graphic Processor Unit (GPU) has evolved into a highly parallel, multithreaded processor with tremendous computational horsepower [39]. The high parallelism of GPU is achieved by the massive replication of simple SIMD (Single Instruction Multiple Data) processors, known as SM (Streaming Multiprocessors) [69].

In 2004, Ian Buck proposed Brook programming language, which was influential attempts to achieve general-purpose computing on GPUs [70]. NVIDIA then coupled the Brooke language extension into their specialized hardware and created the first solution to general purpose parallel computing --- CUDA (Compute Unified Device Architecture).

NVIDIA GPUs are parallel processing units which have the capability of running

thousands of concurrent threads in parallel. The GPU streaming multiprocessors (SMs) have shared resources and on-chip memory which can run parallel tasks with higher performance [39].

4.2. CUDA Introduction

As mentioned above, CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface model developed by NVIDIA. CUDA is not a programming language itself, rather it is a C/C++ extension which enables parallel constructs. In CUDA, no prior knowledge of the graphics pipeline is required, and general algorithms can be transferred into thousands of concurrent threads, executed in parallel, to achieve a significant performance improvement.

The CUDA platform provides three key abstractions: thread group hierarchy, shared memories, and barrier synchronization [39]. CUDA revolves around the idea of a kernel, or GPU function, which is executed for every thread, in every block, within the configured grid. Invoking a CUDA kernel involves firstly creating a thread hierarchy composed of the thread blocks, and threads per a block. Threads are grouped into what are called thread blocks. First building an N-dimensional array of blocks, then

defining how many threads exist in each block forms a thread grid. Figure 4.2.1 shows the typical grid configuration used for image processing (2 dimensional grid of blocks, 2 dimensional blocks of threads). In the case of image processing, the grid size would be dependent on the image's dimensions. For example, if the input image size is 512*512, and the number of threads per block was set to 16*32 (512 threads per block), then the CUDA grid would consist 16*32 thread blocks to process each pixel individually.

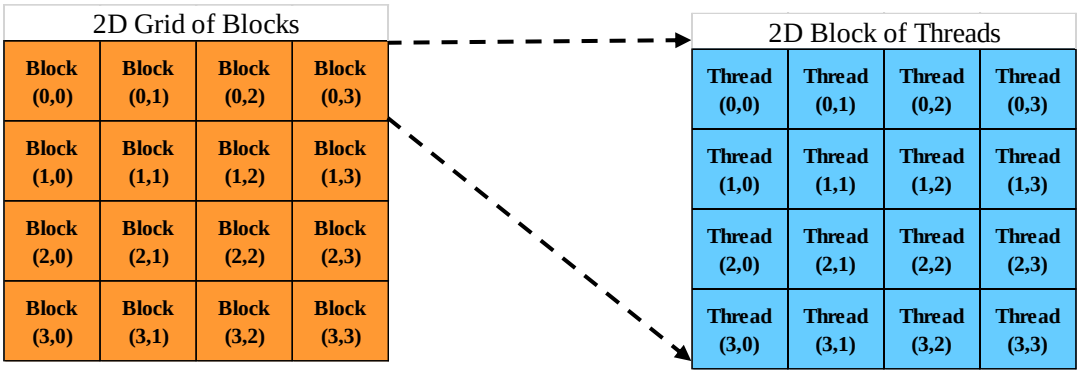


Figure 4.2.1 Typical CUDA Thread Configuration for Image Processing

4.3. CUDA Scalability Programming Model

As Moore's law predicts, the number of transistors in a dense integrated circuit doubles approximately every two years. Thus, the parallelism of GPUs will increase continuously over time. The biggest challenge for a parallel program is to transparently scale its parallelism to leverage the increasing number of processor cores [39].

The CUDA parallel programming model is designed to solve this problem. As illustrated in Figure 4.3.1, within CUDA, each block of threads can be scheduled on any of the available SMs within a GPU in any order, so that a compiled CUDA program can be executed on any number of SMs. With the Scalability feature, programmers can create general parallel codes on CUDA and expect a better performance by updating GPU only.

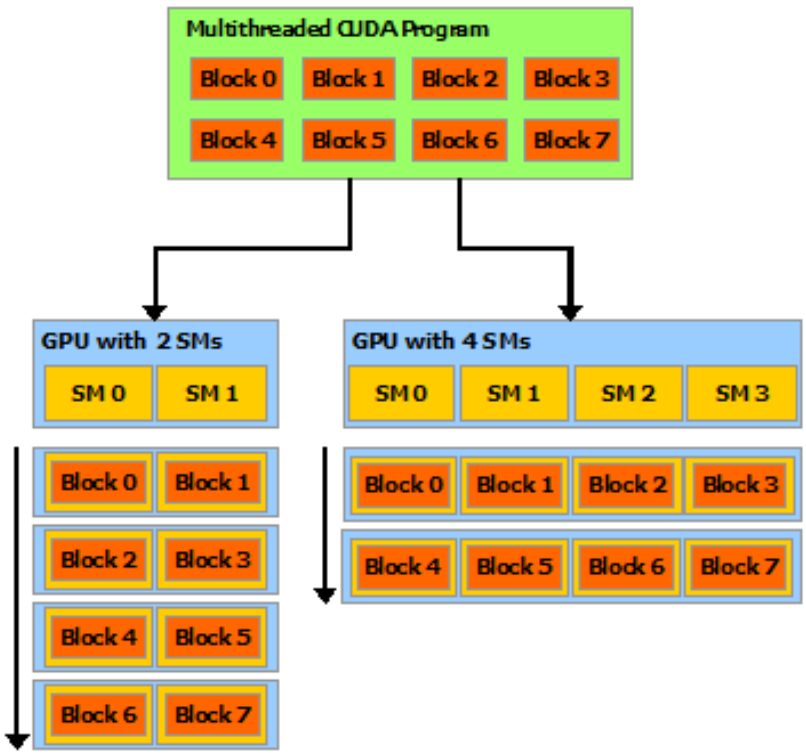


Figure 4.3.1 CUDA Automatic Scalability [39]

4.4. Streaming Multiprocessor

The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs) and the number of SMs on a GPU determines the degree of possible physical parallelism [39]. As mentioned previously, the massive sets of CUDA threads are divided into thread blocks with fixed size in the execution configuration. CUDA threads are grouped into blocks, and CUDA blocks are configured into grids.

When a CUDA program invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available SMs. The threads of a block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor as well. When thread blocks terminate, new blocks will be launched on the vacated SM.

The SM will further partition the blocks into warps, where each warp will be scheduled independently to run all its threads with lock-step level parallelism. Threads within a block will run on the same SM, therefore threads within the same block can utilize local on-chip memory types, say shared memory. The NVIDIA global scheduler will schedule the thread blocks to particular SMs basing on the number of thread blocks, and the number of threads per block in the execution configuration. Figure 4.4.1 shows an example of how CUDA thread blocks are mapped to streaming multiprocessors on the NVIDIA GPU.

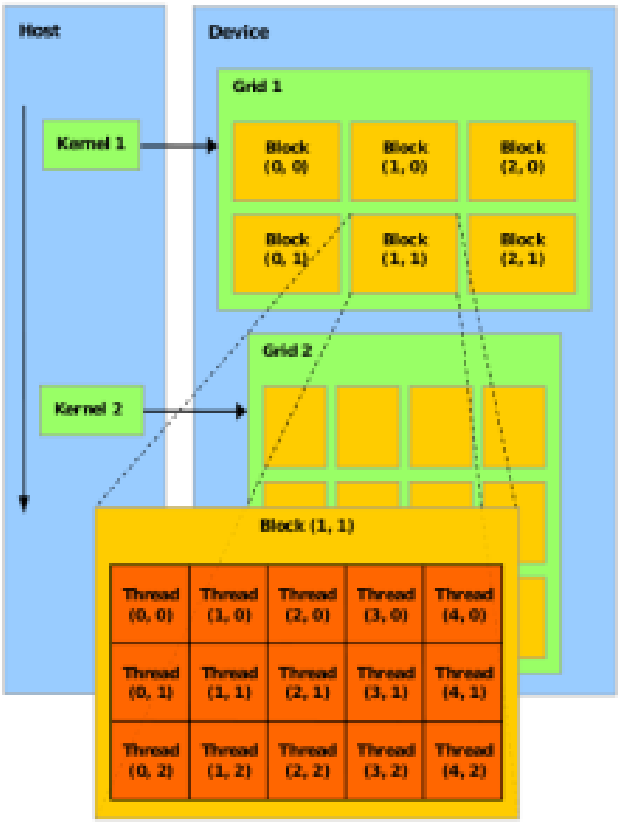


Figure 4.4.1 CUDA Thread Block Configuration [39]

A SM consists of a large array of SIMD (single instruction multiple data) processing cores. SIMD implies that the processing units within an SM will run the same instruction in lock-step level parallelism on different data.

As discussed above, the SM will further partition the scheduled block of threads into units called warps. Warp is the fundamental unit of parallelism defined on NVIDIA GPU hardware. Since the CUDA cores have an SIMD architecture, each thread within

a warp must run the same instruction, or be idle.

4.5. GPU Performance Optimization

Tuning CUDA codes for specific hardware configurations can greatly improve the performance of the system.

Several specifications can improve the performance of NVIDIA GPUs, as shown in Table 4.5.1. By knowing the GPU specifications for the hardware being programmed, an algorithm's implementation can be optimized to exploit all resources on the specific GPU.

GPU Specification	Method to Improve Performance
SM Number	Increase the number of SMs to increase the number of concurrent threads executing in parallel
Warp Size	Increase the warp size to increase the number of threads running in parallel within a single SM
Shared Memory Size	Increase the shared memory size per block to allow for higher SM thread occupancy
Warps per SM	Increase the number of warps in an SM to increase the number of threads executing in parallel

Table 4.5.1 Methods to improve GPU performance

The performance can also be improved by using different types of memory. Figure 4.5.1 illustrates the memory architecture of GPU:

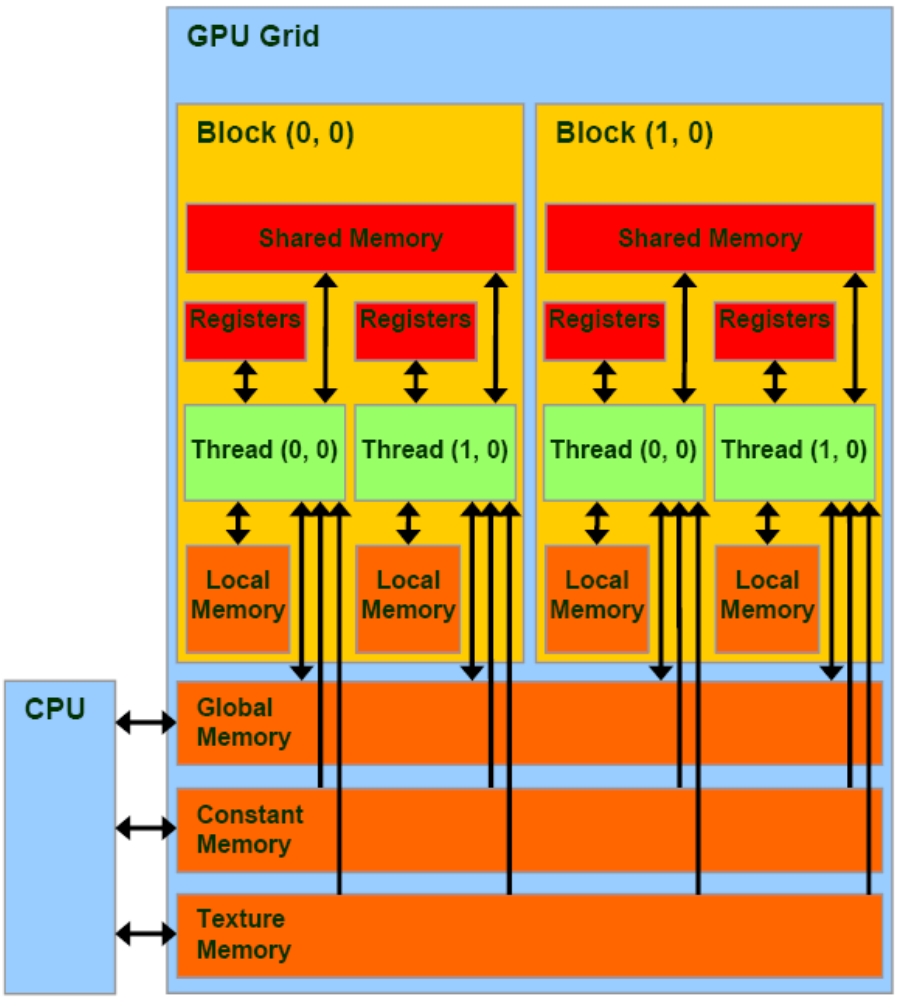


Figure 4.5.1 GPU Memory Architecture [39]

1) Texture Memory

Texture memory on the GPU is the memory normally used for the graphics pipeline, however it is also available for general purpose computing. Texture memory

is cached on-chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. And it has great performance benefits when memory accesses exhibit spatial locality (Figure 4.5.2). In the SURF algorithm, convolution operations are of high reference of locality in terms of memory access since neighboring pixels are required to work on one pixel. So that GPUs are able to make use of the texture memory to achieve better performance when executing convolution operations.

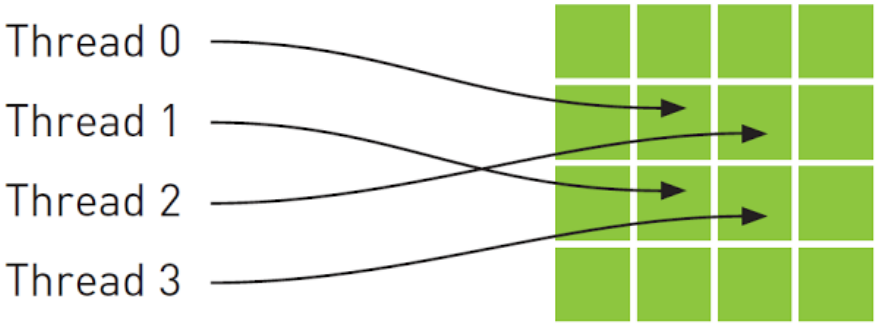


Figure 4.5.2 Spatial Locality Memory Access Pattern

In general, for implementations that have read-only memory access patterns with high spatial locality, texture memory can be exploited to improve performance by avoiding the costs of accessing global memory.

Kernel code is not able to write data into texture memory though it can read from texture memory, which means execution should be switched between CPU code and GPU kernel code repeatedly whenever it generates a texture with the calculation results

at each step. Therefore there will be four times data transferring between host and device in the implementation using texture memory. This is costly since the bandwidth between the device memory and the host memory is much lower compared to the bandwidth between the device and the device memory.

2) Global Memory

Device memory reads texture fetching present several benefits over reads from global or constant memory:

i. Texture memory are cached, potentially exhibiting higher bandwidth if there is high locality in the texture fetches.

ii. The method of addressing is much easier because they are not subject to the constraints on memory access pattern that global or constant memory reads have to obey to get good performance.

iii. The latency of addressing calculations is better hidden.

However, performance of switching between device and host in order to write data into texture memory versus no switching between device and host but using global memory for manipulating data still have to be considered.

3) Shared Memory

Shared memory is the streaming multiprocessor on-chip memory normally with sizes ranged from 16 KB to 64 KB, thus its access is faster than global memory access. In fact, shared memory latency is approximately 100 times lower than uncached global memory latency. Each thread block has its own dedicated segment of shared memory since each block is scheduled to run exclusively on a particular SM. Threads can access data in shared memory loaded from global memory by other threads within the same thread block, however shared memory cannot be accessed between SMs, and therefore shared memory cannot be shared between thread blocks. The amount of shared memory is orders of magnitude smaller than global memory, thus the use of shared memory may increase the complexity of CUDA implementation.

In summary, optimizing the GPU specification naively may sometimes produce worse performance over the CPU implementation. Correct optimization strategies must be known in order to maximize the parallel performance. The purpose of running algorithms in parallel is to maximize algorithmic performance, so a basic knowledge of the hardware specifications is imperative.

4.6. Parallel Implementation for the Integral Image

Before discussing the parallel implementation for the integral image, we first

introduce a popular approach to calculate the sum of an arbitrary array.

The exclusive scan operation is defined as the one-dimensional accumulation of an arbitrary array, for each value computed is equal to the sum of all previous values, excluding the current value. The exclusive scan of an array of *n* elements can be represented as the following equation:

$$[a_0, a_1, \dots, a_{n-1}] \rightarrow [0, a_0, (a_0 + a_1), \dots, (a_0 + a_1 + \cdots + a_{n-2})].$$

In 1990, Blelloch proposed an approach to calculate exclusive scan efficiently using parallel algorithms [71]. The parallel implementation of the exclusive scan operation involves two separate steps: the up-sweep step and the down-sweep step. As shown in Figure 4.6.1, the up-sweep operation can be considered as a binary tree, which reduces the number of nodes at each level by half and makes one summation per node [72]. Each thread will execute log2(N) iterations where N is the length of the input array. Only half of the data from the previous iteration are used to generate the next layer.

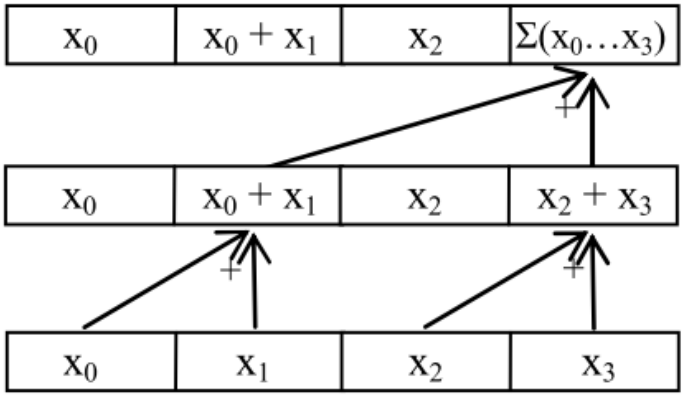


Figure 4.6.1 Parallel Up-Sweep Scan Implementation [72]

Figure 4.6.2 shows the parallel implementation of the down-sweep phase. In the down-sweep step, the last element from the up-sweep result is set to zero first, then from the root to the leaves, partial sums computed in the up-sweep step are used to get the final exclusive scan result. At each iteration, twice the number of data will be executed than the previous iteration. The parallel exclusive scan operation is work efficient and has a O(N) work complexity, same as the sequential implementation.

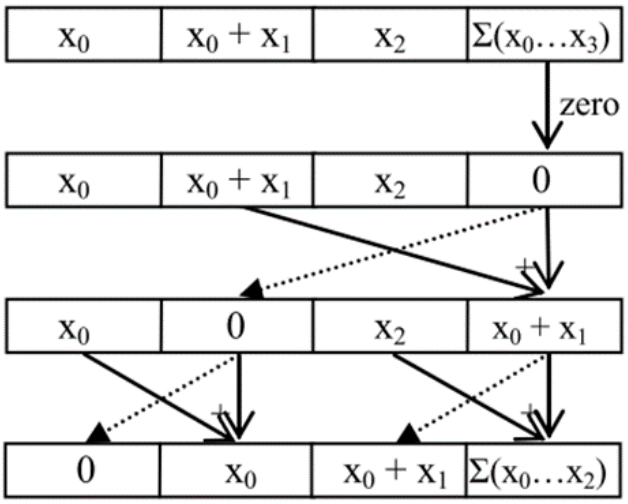


Figure 4.6.2 Parallel Down-Sweep Scan Implementation [72]

Even though this usual exclusive scan approach is work efficient, it suffers from the limitation of one block, as indicated in [73]. It loads the entire input data into shared memory and executes the whole exclusive scan operation within one block, which means it is unable to scan arrays with larger size since the maximum number of threads

per block is 512. And using one single thread block will also leave the other SMs inactivated, bringing a low SM utilization ratio.

To solve this problem and achieve more parallelism, we divide the input array into several parts, and distribute them into different thread blocks. For example, we have an input array of size *n* and *b* thread blocks to launch. To make full use of all the SM, the number of thread blocks should be no less than the number of SMs. Thus, the number of thread *t* we need to launch per block is equal to *n / s*, where *s* is the number of SM on the GPU. And the number of thread blocks we need to launch is *n / t*. Table 4.6.1 shows an example of our approach.

Variable	Expression	Value
Input Data Size	N	2048
Number of SMs	S	8
Threads Per Block	t = (n/s)	256
Number of Blocks	b = (n/t)	64

Table 4.6.1 Thread Configuration Example

The CUDA exclusive scan implementation spanning over multiple thread blocks is visually represented in Figure 4.6.3. Each block will execute the exclusive scan operation on individually. The last element of each block contains the summation result

of all the elements in that segment, we extract it to build an auxiliary array [72]. Then the exclusive scan operation will be performed on the generated auxiliary array. And the results of the auxiliary array will be added back into the segmented scan arrays to complete the exclusive scan operation of the whole input array.

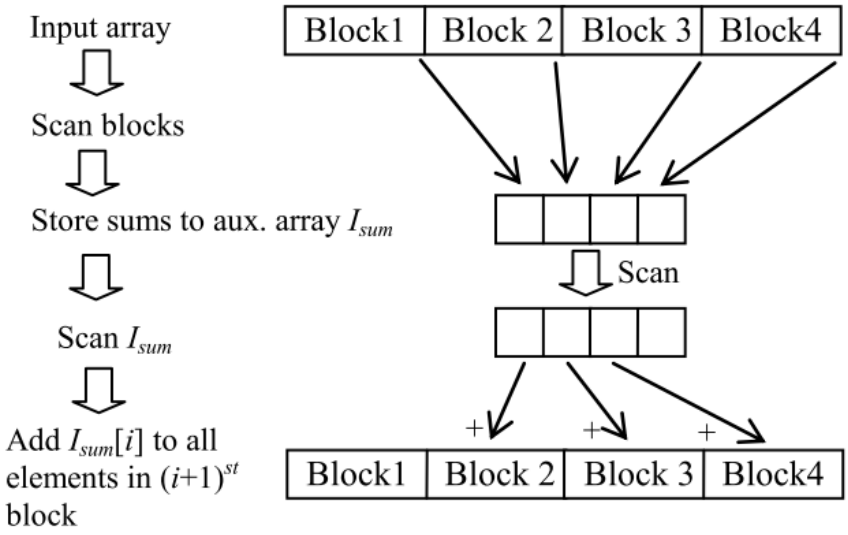


Figure 4.6.3 Parallel scan involving multiple thread blocks [72]

The integral image is computed by taking the rows of the input image as arbitrary arrays, then performing exclusive scan operation on all the rows. After that, in order to obtain the scan result of the columns, we take the transpose of the resultant array and use the same scanning kernel again. Once the second exclusive scan operation is completed, the resultant array will be transposed again in order to obtain the final integral image result.

Figure 4.6.4 shows the comparison of integral image processing times on CPU and GPU. Further discussion of the result will be presented in chapter 5.

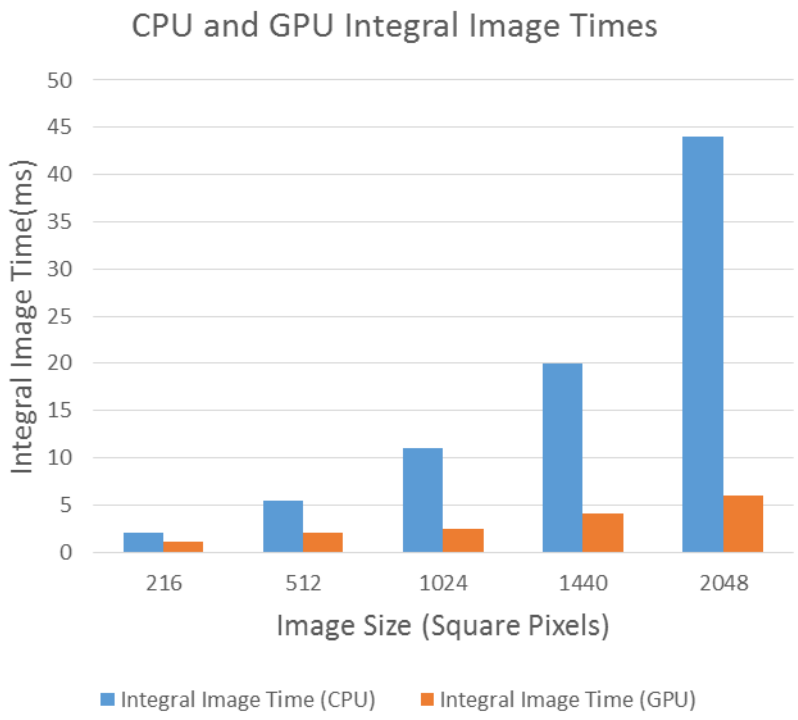


Figure 4.6.4 CPU vs. GPU Integral Image Processing Time

4.7. Parallel Implementation

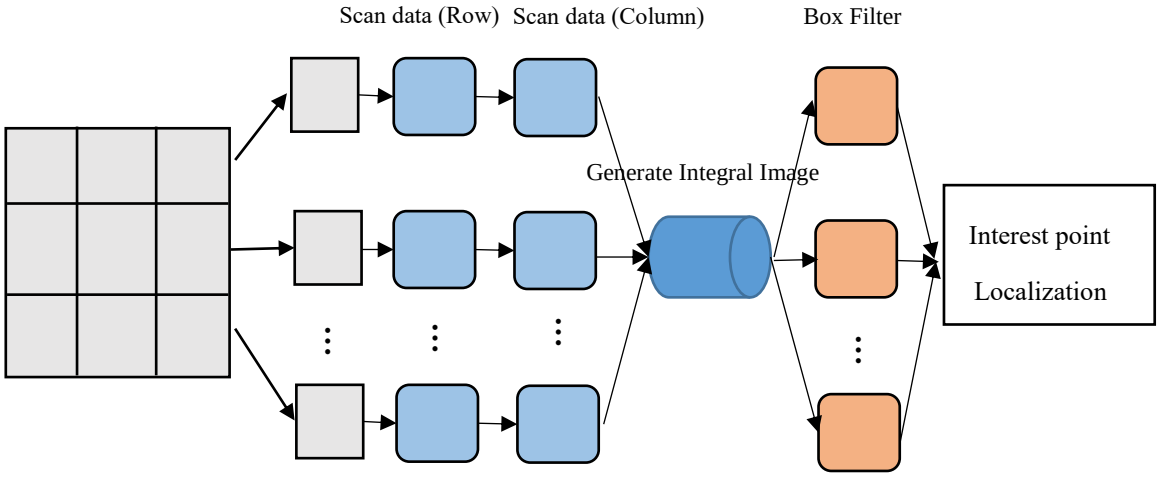


Figure 4.7.1 Architecture of the Parallelization of SURF Detection

The whole procedure of parallel implementation of SURF detector is illustrated in Figure 4.7.1. As mentioned in the previous section, the input image is divided into several segments first and assigned to different thread blocks to be processed. The system first performed the exclusive scan operation for all the rows of the input segment, and transposed the column-scanned image back to the original orientation. Then it repeated the same operation on the transposed image to obtain the integral image.

To characterize the effects of different parallel optimization strategies for our image recognition algorithm, we implement SURF algorithm using different memories and access patterns.

First, we use global memory only in the implementation. In this approach, an

image is loaded into a global memory, a region of the image block is multiplied with the box filters, add up the results, and then output pixel is written back to global memory for next step. Since whole image is loaded into the global memory, we only need to consider boundary when a pixel is at the edge of the image. Then the input image is segmented into 16×16 sub-windows and a set of CUDA blocks process these sub-windows for convolution operations until all the sub-windows are completed.

For the implementation using texture memory only, since texture memory is read-only for kernel code on GPU, which means operations should be transferred between CPU code and GPU kernel code repeatedly whenever it generates a texture with the calculation results at each step. Therefore four times data transferring between host and device in the implementation using texture memory.

When we implement SURF algorithm with shared memory, a block of pixels from the image is loaded into an array in shared memory, convolution or thresholding operation is applied to the pixel block, and then the output image is written into shared memory array for the use on the next step.

In all cases of implementation, the kernel configuration is of 16×16 threads of each block and 32 of blocks on a 512x512 image. The convolution is parallelized across the available computational threads where each thread computes the convolution result of its assigned pixels sequentially. Pixels are distributed evenly across the threads. The

result shows that the shared memory and texture memory implementations have a big advantages in speed, however texture memory processing time is longer than shared memory. The main reason for that is due to the repeatedly data transfer operation between host and device. Thus in our GPU SURF program, we only use texture memory to store the integral image and original image to avoid the use of global memory. The convolution operations of box filter will be implemented in shared memory to increase the speed.

Figure 4.7.2 shows the schematic of the whole system. The parallel computation allowed the system to finish the operations on pixels efficiently, which are a major part of SURF and FREAK algorithm.

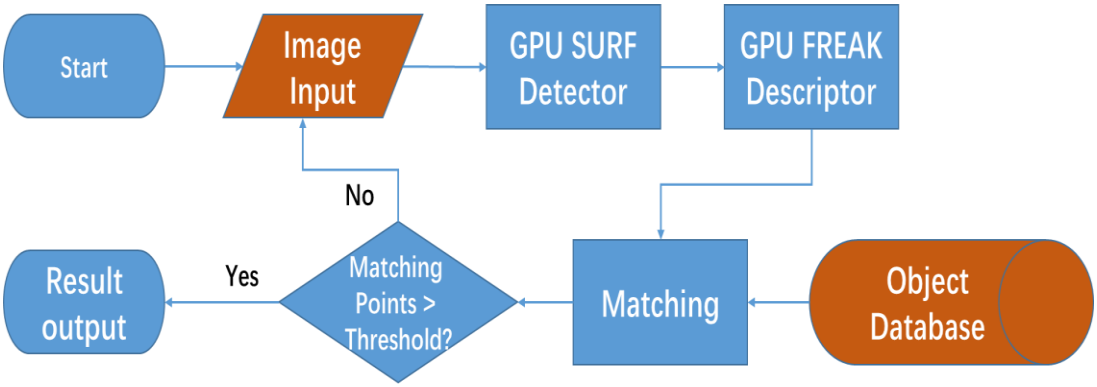


Figure 4.7.2 Schematic of the System

CHAPTER 5: EXPERIMENT SETUP AND RESULTS

5.1. Experiment Setup

Table 5.1.1 shows the configurations of our experiment setup. Note that the results of our experiments are all based on the environment specification below. Table 5.1.2 GPU Specifications shows the specification of NVIDIA GeForece GT 755M. The CUDA code has been optimized based on the specification of this GPU.

Environment Specification	
CPU	Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz
GPU	NVIDIA GeForce GT 755M
RAM	8.00 GB
OS	Microsoft Windows 10 (64bit)
OPEN CV	Version 2.4.13
Visual Studio	Visual Studio 2013
CUDA	CUDA Toolkit 8.0

Table 5.1.1 Experiment environment specification

NVIDIA GeForce GT 755M Specification	
Architecture Type	Kepler
CUDA Capable Version	3.0
Threads per Warp	32
Max Warps per SM	64
Max Thread Blocks per SM	16
Max Active Threads per SM	2048
Max Threads per Block	1024
Shared Memory per SM (bytes)	49152
Max Shared Memory per Block (bytes)	49152

Table 5.1.2 GPU Specifications

5.2. Experiment Results

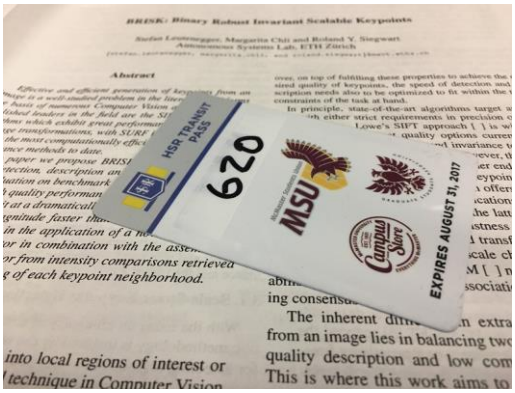
The schematic of the object recognition system is shown in Figure 4.7.2. The detection and description stages are implemented in CUDA platform to take full advantage of the parallel computing capability of NVIDIA GPU. The feature matching system is executed by first providing an original training image to build a feature database, then providing target images where the features from the training image will be matched to. Figure 5.2.1 (a) shows the training image chosen for this thesis, and Figure 5.2.1 (b, c, d) illustrates the target images for measuring the performance of the feature matching computer vision system.

The performance results were analyzed by feature matching a training image to multiple images, as shown in Figure 5.2.1. Then the target images were sub-sampled from image dimensions 2048*2048 to 216*216 to measure performance processing time over various image resolutions. Figure 5.2.2 illustrated the results of our feature matching system.

(a) Training Image

(b) Target Image 1

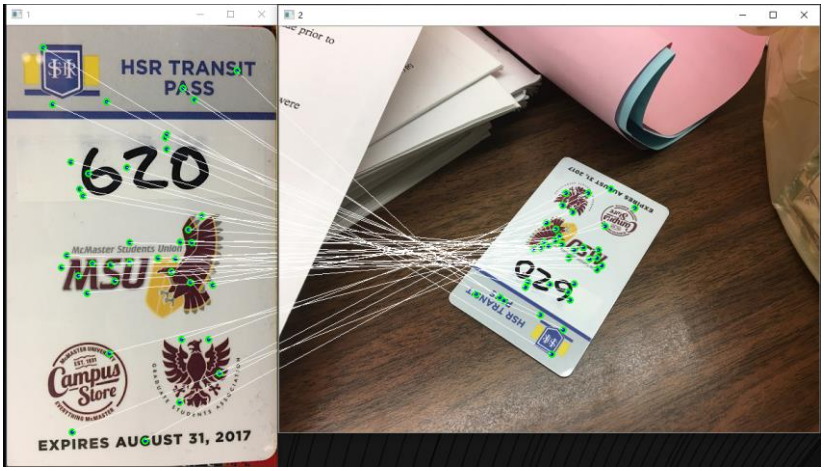
(c) Target Image 2



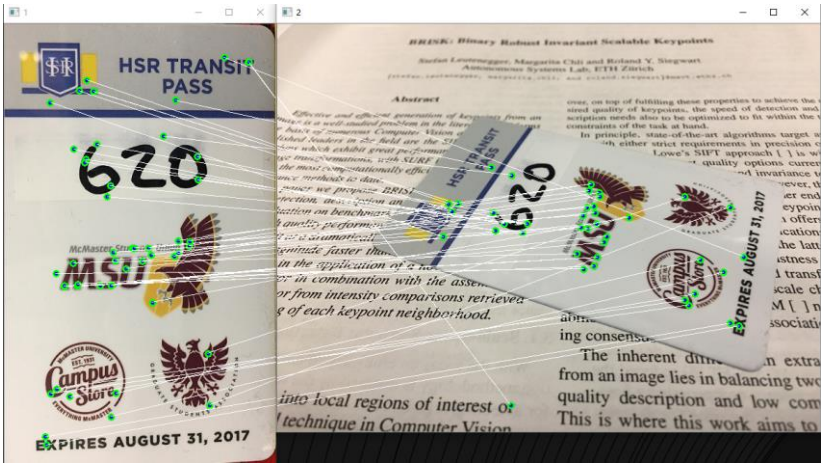
(d) Target Image 3

Figure 5.2.1 Training Image and Target Images

(a) Matching with Target Image 1



(b) Matching with Target Image 2



(c) Matching with Target Image 3

Figure 5.2.2 Feature Matching Results

Figure 5.2.3 to Figure 5.2.5 list the execution time comparisons of the integral image, feature detection and recognition between CPU and CUDA, where the input images are the training image 1(Figure 5.2.1 (a)) and scaled target image 2 and 3(Figure 5.2.1 (b,c)).

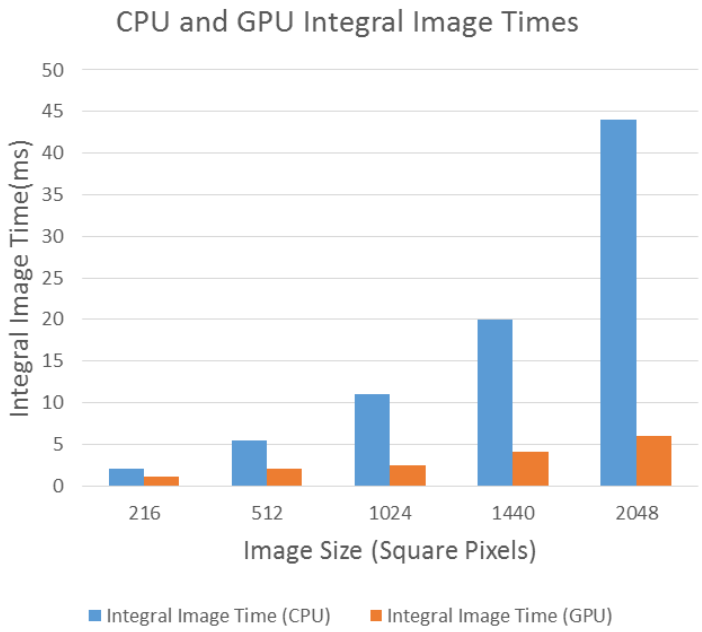


Figure 5.2.3 Integral Image Generation Time Comparison between CPU and GPU

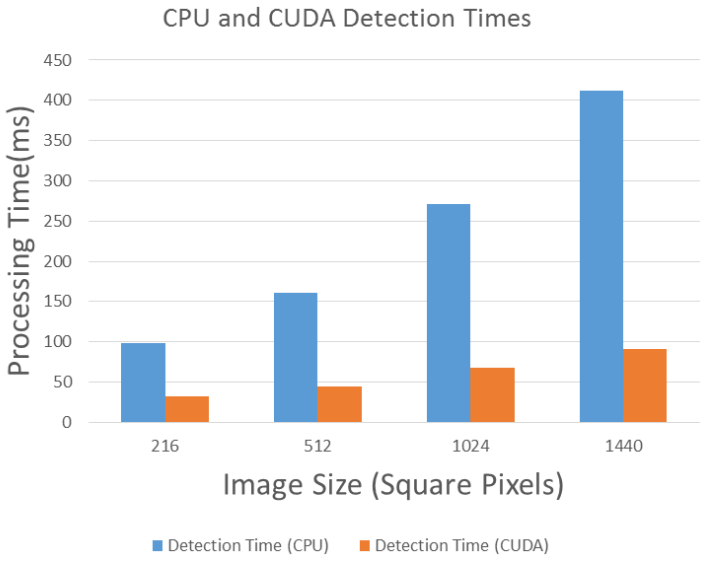


Figure 5.2.4 SURF Feature Point Detection Time Comparison between CPU and CUDA

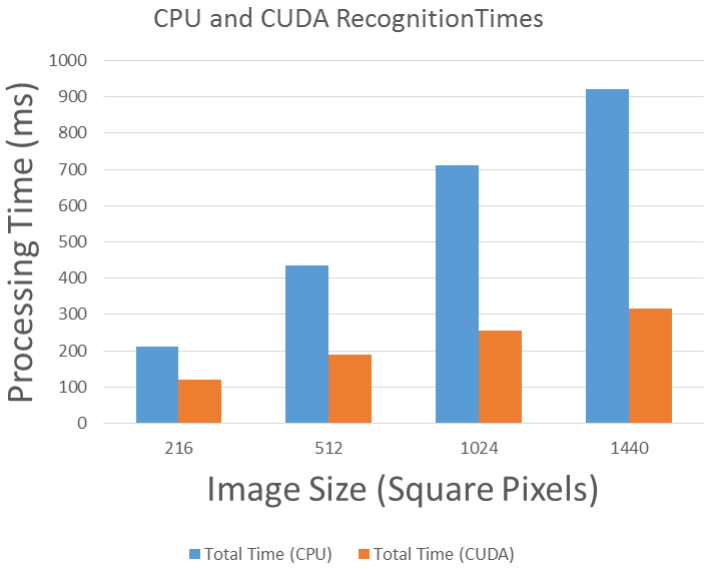


Figure 5.2.5 SURF and FREAK Execution Times Comparison between CPU and CUDA

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1. Conclusion

This thesis presents modern CUDA optimizations strategies to decrease the processing time for real-time performance. As shown in Figure 5.2.3, the processing times of the optimized CUDA implementation did not exceed 350 ms for all image dimensions ranging up to 1440x1440. The optimized CUDA implementation had an average 3 times faster over CPU-only implementation. The optimized CUDA implementation did not compromise did not compromise precision for performance, since the implementation has the same precision as the other implementations.

The application of the optimized CUDA SURF detection implementation towards the feature matching computer vision system showed an average speed-up of 3 times faster over the traditional CPU implementation. The CUDA implementation of the object recognition algorithms provides an efficient and flexible system which can be utilized by passive vision censors on the autonomous vehicle or other higher level computer vision systems: motion detection, image registration, video tracking, panorama stitching, 3D modeling, and object recognition.

6.2. Future Work

The significant speed advantages of CUDA implementation over traditional CPU implementation due to NVIDIA CUDA scalability, as discussed in Section 2.2, the optimized CUDA implementation will scale to future NVIDIA GPUs with higher performance specifications. This implies that the same optimized CUDA implementation is contemporary. Future improvements to NVIDIA GPU hardware will effectively improve the performance of this implementation.

The NVIDIA GPU and CUDA platform is continually upgrading and the GPU performance is always increasing. For instance, the GeForce GTX 480 released in 2010 contains 480 processing cores, while the GPU used to conduct this thesis research, GeForce GT 755M, released in 2013 contains 1344 processing cores, nearly tripling the parallel processing capability over the span of 3 years. Areas of future work in the area of GPU image recognition include optimizing the algorithm on the most recent GPU hardware architecture (Maxwell) and scaling the algorithm's implementation to a multi-GPU environment.

Maxwell is one of the NVIDIA's latest GPU architecture, which released in 2014. The Maxwell architecture provides dramatic improvements to the streaming multiprocessor design in areas of energy efficiency, control logic partitioning (avoids warp divergence), workload balancing, instructions executed per clock cycle, and many

more. The Maxwell architecture supports dynamic parallelism which allows for CUDA kernels to invoke kernels themselves. The same implementation discussed throughout this thesis will receive a performance benefit when run on Maxwell architecture; however, further performance improvement can be achieved by re-implementing the algorithm specifically to utilize all resources on the NVIDIA Maxwell architecture.

CUDA supports the invocation of multiple GPU execution asynchronously away from the host. Future work for the research discussed in this thesis includes scaling the single GPU CUDA SURF and FREAK image recognition implementation to a multi-GPU environment. The existence of multiple GPUs in the environment allow for optimized load balancing of threads per SM between all GPUs, thus increasing GPU efficiency and performance.

REFERENCES

[1] D. Ballard and C. Brown, “Computer vision,” 1982.

[2] T. S. Huang, “Computer Vision: Evolution and Promise.”

[3] M. Sonka, V. Hlavac, and R. Boyle, “Image Processing, Analysis, and Machine Vision Second Edition.”

[4] R. Szeliski, “Computer vision: algorithms and applications,” 2010.

[5] S. A. Papert, “The Summer Vision Project,” 1966.

[6] M. A. Boden, *Mind as machine : a history of cognitive science*. Clarendon Press, 2008.

[7] L. Davis, “A survey of edge detection techniques,” *Comput. Graph. image Process.*, 1975.

[8] B. Baumgart, “Geometric modeling for computer vision,” 1974.

[9] H. Baker, “Three-Dimensional Modeling.,” *IJCAI*, 1977.

[10] V. Nalwa, “A guided tour of computer vision,” 1994.

[11] S. Ullman, “The interpretation of structure from motion,” *Proc. R. Soc.*, 1979.

[12] H. Longuet-Higgins, “A computer algorithm for reconstructing a scene from two projections,” *Comput. Vis. Issues, Probl. ...*, 1987.

[13] S. Seitz and R. Szeliski, “Applications of computer vision to computer graphics,”

Comput. Graph. (ACM)., 1999.

[14]T. Beier and S. Neely, "Feature-based image metamorphosis," *ACM SIGGRAPH Comput. Graph.*, 1992.

[15]N. Sebe, *Machine learning in computer vision*. Springer, 2005.

[16]W. Freeman, P. Perona, and B. Schölkopf, "Guest Editorial," *Int. J. Comput. Vis.*, vol. 77, no. 1–3, pp. 1–1, May 2008.

[17]S. Gehrig and F. Stein, "Dead reckoning and cartography using stereo vision for an autonomous car," *Robot. Syst. 1999. IROS'99. ...*, 1999.

[18]E. Dickmanns and A. Zapp, "Autonomous high speed road vehicle guidance by computer vision," *Autom. Control. World Congr. (10th). ...*, 1988.

[19]"'Phantom Auto' to Be Operated Here," *Google News Arch.*, 1932.

[20]"Autonomous Cars Will Make Us Safer," *WIRED*, 2009.

[21]J. Schmidhuber, *Prof. Schmidhuber's highlights of robot car history*. 2009.

[22]T. Kanade, C. Thorpe, and W. Whittaker, "Autonomous land vehicle project at CMU," in *Proceedings of the 1986 ACM fourteenth annual conference on Computer science - CSC '86*, 1986, pp. 71–80.

[23]R. Wallace, "First results in robot road-following," *JCAI'85 Proc. 9th Int. Jt. Conf. Artif. Intell.*, 1985.

[24]C. Albanesius, "Google Car: Not the First Self-Driving Vehicle," *PC Mag.*, 2010.

[25]M. Harris, “How Google’s autonomous car passed the first US state self-driving test,” *IEEE Spectr.*, 2014.

[26]D. P. Howley, “The Race to Build Self-Driving Cars,” *Laptop*, 2012.

[27]O. A. V. S. Committee, “SAE J3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems,” *SAE Int.*

[28]*Autopilot: Full Self-Driving Hardware on All Cars*. Tesla Motors.

[29]W. Stone, M. Juberts, N. Dagalakis, J. Stone, and J. Gorman, “Performance analysis of next-generation LADAR for manufacturing, construction, and mobility,” 2004.

[30]F. Alonge, M. Branciforte, and F. Motta, “A novel method of distance measurement based on pulse position modulation and synchronization of chaotic signals using ultrasonic radar systems,” *IEEE Trans.*, 2009.

[31]S. Seitz, “An overview of passive vision techniques,” *cs.cmu.edu*.

[32]B. Coifman, D. Beymer, P. McLauchlan, and J. Malik, “A real-time computer vision system for vehicle tracking and traffic surveillance,” *Res. Part C Emerg. ...*, 1998.

[33]H. Herbert, “The history of OCR, optical character recognition,” *Manchester Center, VT Recognit. Technol.*, 1982.

[34]D. Vernon, “Machine vision-Automated visual inspection and robot vision,” *NASA STI/Recon Tech. Rep. A*, 1991.

[35] Y. Chuang, A. Agarwala, and B. Curless, "Video matting of complex scenes," *ACM Trans.*, 2002.

[36] T. Moeslund and E. Granum, "A Survey of Computer Vision-Based Human Motion Capture," *Comput. Vis. Image*, 2001.

[37] N. Ratha and R. Bolle, "Automatic fingerprint recognition systems," 2007.

[38] A. Danowitz, K. Kelley, J. Mao, and J. Stevenson, "CPU DB: recording microprocessor history," *Commun.*, 2012.

[39] NVIDIA, "Cuda C Programming Guide," *Program. Guid.*, no. September, pp. 1–261, 2015.

[40] T. Lindeberg, "Scale-Space," in *Wiley Encyclopedia of Computer Science and Engineering*, vol. IV, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2008, pp. 2495–2504.

[41] S. Umbaugh, "Digital image processing and analysis: human and computer vision applications with CVIPtools," 2016.

[42] J. Canny, "A computational approach to edge detection," *IEEE Trans. pattern Anal. Mach.*, 1986.

[43] H. Works, "Sobel Edge Detector," *cse.secs.oakland.edu*.

[44] J. Prewitt, "Object enhancement and extraction," *Pict. Process. Psychopictorics*, 1970.

[45] C. Harris and M. Stephens, "A combined corner and edge detector.," *Alvey Vis.*

Conf., 1988.

[46]E. Rosten and T. Drummond, “Machine learning for high-speed corner detection,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3951 LNCS, pp. 430–443, 2006.

[47]D. G. Lowe, “Object recognition from local scale-invariant features,” *Proc. Seventh IEEE Int. Conf. Comput. Vis.*, vol. 2, no. [8, pp. 1150–1157, 1999.

[48]D. Lowe, “Distinctive image features from scale-invariant keypoints,” *Int. J. Comput. Vis.*, 2004.

[49]H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-Up Robust Features (SURF),” no. September, 2008.

[50]J. Matas, O. Chum, M. Urban, and T. Pajdla, “Robust wide-baseline stereo from maximally stable extremal regions,” *Image Vis. Comput.*, 2004.

[51]T. Lindeberg, “Scale-space theory: A basic tool for analyzing structures at different scales,” *J. Appl. Stat.*, 1994.

[52]E. Adelson, C. Anderson, and J. Bergen, “Pyramid methods in image processing,” *RCA*, 1984.

[53]P. Burt and E. Adelson, “The Laplacian pyramid as a compact image code,” *IEEE Trans. Commun.*, 1983.

[54]J. Crowley and A. Parker, “A representation for shape based on peaks and ridges in the difference of low-pass transform,” *IEEE Trans. Pattern Anal.*, 1984.

[55]P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” *Comput. Vis. Pattern Recognit.*, vol. 1, p. I--511--I--518, 2001.

[56]T. Lindeberg, “Feature Detection with Automatic Scale Selection,” *Int. J. Comput. Vis.*, vol. 30, no. 2, pp. 79–116, 1998.

[57]A. Neubeck and L. Van Gool, “Efficient Non-Maximum Suppression,” *18th Int. Conf. Pattern Recognit.*, vol. 3, no. 1, pp. 850–855, 2006.

[58]M. Brown and D. Lowe, “Invariant Features from Interest Point Groups,” pp. 253–262, 2002.

[59]S. Lazebnik, C. Schmid, and J. Ponce, “A sparse texture representation using affine-invariant regions,” *Comput. Vis. Pattern*, 2003.

[60]M. Brown and D. Lowe, “Recognising panoramas.,” *ICCV*, 2003.

[61]C. Schmid and R. Mohr, “Local grayvalue invariants for image retrieval,” *IEEE Trans. Pattern Anal. Mach.*, 1997.

[62]A. Johnson and M. Hebert, “Using spin images for efficient object recognition in cluttered 3D scenes,” *IEEE Trans. pattern Anal.*, 1999.

[63]S. Belongie, J. Malik, and J. Puzicha, “Shape matching and object recognition using shape contexts,” *IEEE Trans. pattern*, 2002.

[64]K. Mikolajczyk, K. Mikolajczyk, C. Schmid, and C. Schmid, “A performance evaluation of local descriptors,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp. 1615–1630, 2005.

[65] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary robust independent elementary features," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6314 LNCS, no. PART 4, pp. 778–792, 2010.

[66] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2564–2571, 2011.

[67] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust invariant scalable keypoints," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2548–2555, 2011.

[68] A. Alahi, R. Ortiz, and P. Vandergheynst, "FREAK: Fast retina keypoint," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 510–517, 2012.

[69] R. Farber, "CUDA application design and development," 2011.

[70] I. Buck, T. Foley, D. Horn, and J. Sugerman, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans.*, 2004.

[71] G. Blelloch, "Prefix sums and their applications," 1990.

[72] B. Bilgic, B. Horn, and I. Masaki, "Efficient integral image computation on the GPU," *Intell. Veh. Symp. (*, 2010.

[73] M. Harris, S. Sengupta, and J. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems*, 2007.