SPEEDING UP BOOSTED CASCADE OF OBJECT DETECTION USING COMMODITY GRAPHICS HARDWARE

FENG JIMIN

(B.Sc., NATIONAL UNIVERSITY OF SINGAPORE, 2005)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor, Dr Terence Sim. His wide knowledge and enthusiasm has constantly guide me with correct research directions and altitudes. His kindness and patience has given great encouragement to me, without which this thesis wouldn't have been possible. I also thank Dr Samarjit Chakraborty, who leaded me to the research field, on which the experiences I would never regret. I would like to express my appreciation to my colleagues: Zhang Xiaopeng, Qi Yingyi, Hong Guangming, Guo Dong, Zhuo Shaojie, etc in Computer Vision Lab of School of Computing, NUS. I can never forget the pleasant stay with these brilliant people, from whom I got help and inspirations. Finally, I would like to thank my family, for their endless love and support year by year. This thesis is dedicated to my father, a great man and father. His generous love and his tireless support for his children has given me strength when facing difficulties. Without him, none of my achievements would have been possible.

Contents

Abstract

Visual object detection has conceivably prevailing applications in the Internet age for multimedia interactions. This thesis aims to explore the data-parallel architecture of commodity graphic processors that enables fast and promising object detection. Firstly, boosting is identified as the promising and widely usable approach for object detection. An efficient architecture in streaming paradigm is then designed to map the boosted cascade to GPU as data streaming coprocessor. We take into account the hardware characteristics to enable further speedup. Promising results were achieved, as up to 5 times speedup was obtained. By study the performance impacts from experiments on GPUs that are of different generations, we explore the suitable designs for future generation GPU models. Our experiences reveal the underlying principles when mapping the boosted detector to similar data-parallel architectures.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Visual Object Detection on natural images has been a long standing goal in computer vision research. Its conceivable applications in numerous fields like semantic image understanding, autonomous robot control, human-computer interactions, security video surveillance, etc have driven this trend. As Vint Cerf, the Chief Internet Evangelist in Google once envisioned[VC08], the future Internet will ”transform multimedia contents” (images, videos) to be more interactive, where medias containing user interested objects can be retrieved and labeled with relevant information. One can easily imagine that the tremendous amount of object detection work on vast amount of multimedia contents in the Internet is the basic step towards that goal.

Visual object detection studied in this thesis can be defined as detecting the presence of specific class of objects in the input image; and if presented, localizing them in the image. The problem is inherently difficult due to large variability in object appearance, poses, viewpoints, illumination conditions, and imaging conditions, etc. The vast differences in various types of objects add to the difficulty of coming out a general solution to the problem. For example, to detect structured objects(like face, car), or articulated objects (pedestrians, animals, etc) or textured objects, it is often rational to extract image features of different characteristics and use different object models for better performance.

A general principle for object detection is to devise object models that represent the object class by learning sufficient domain knowledge from the images. To this end the object models often employ image features and a representational framework that can minimize intra-class variation while emphasizing inter-class variations. Thus, extracting a suitable set of features, and use an efficient learning framework to acquire the appropriate object model is the key. Figure 1.1 shows a typical object detection framework.

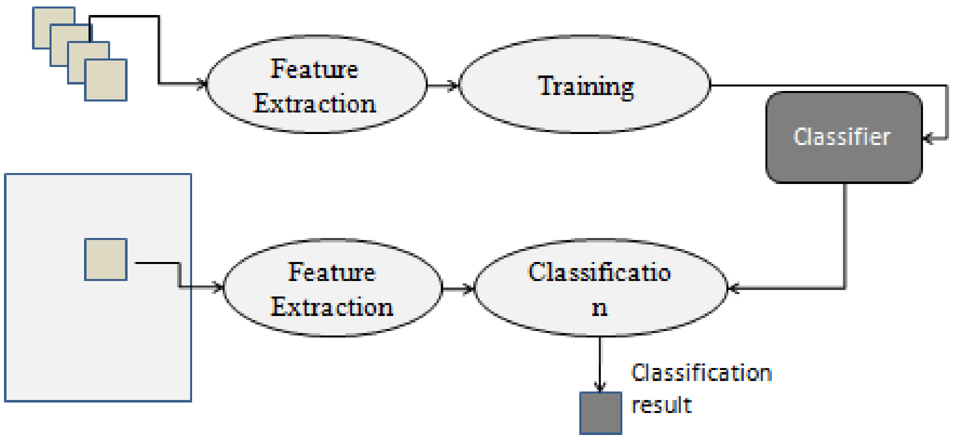

Figure 1.1: Typical object detection framework.

1.1.1 Boosting as Promising Approach

In the literature, an extensive body of works have proposed and evaluated different types of features and models suitable for object detection ranging from very specific class (e.g. objects with different viewpoint or poses treated as different classes), to generic class where the appearance of the objects belonging to the same category can vary a lot. Table

1.1 tabulates a summary of the popular approaches in the literature that solves problems of varying degree of complexity, using different type features and learning approaches.

As one of the earliest successful attempt in object detection, Lowe[Low99] proposed an object detection framework for detecting very specific object class. It defines the object class using a set of characteristic views. A view is represented by collections of keypoint descriptors (features invariant to affine transformations (SIFT)) extracted from training images. This framework can only cope with rigid structured objects with similar appearance, and performs poorly when object appearance or structures have large variations.

People detection is more difficult because of wide variability in appearance due to clothing, articulation and illumination conditions. Dalal & Triggs[DT05] presented an algorithm that detects people in a single image with high accuracy. The algorithm extracts histogram of oriented gradients (HOG) from a dense grid of image blocks. A linear support vector machine is learned from training images. A fixed size detection windows is then sliding over the images to decide the presence of human object. The success of this method greatly attribute to the use of HOG which represents spatially unordered measurements over local image region, and is invariant to a degree of variations in object appearance.

Zhu[ZYCA06] further extends the above approach by using AdaBoost, one of the variants of Boosting learning framework. Boosting is a learning meta-algorithm for performing supervised learning; it creates a single strong classifier from a set of weak-classifiers. A weak classifier is defined to be only slightly correlated with the true classification. In contrast, a strong classifier is arbitrarily well-correlated with the true classification.

Boosting is a powerful learning mechanism that combines the property of an efficient classifier and a feature selector. This framework can select most descriptive HOG features of human based on much larger set of variable-size image blocks. The detection

Approaches	**Solved problem**	**Feature-to-Use**	**In-class variability**
View matching based on SIFT[Low99]	Detect rigid object with variation in viewpoints	SIFT features	Handles only small variation in views for structured object
Linear SVM based on HoG[DT05]	Pedestrian detection	Histogram of oriented gradients	Large variability in appearance
AdaBoost based on HoG[ZYCA06]	Pedestrian detection	Histogram of Oriented Gradients	Large variability; faster and accurate
Boosting based on features in heterogenous/homogenous region[OPFA06]	Generic object detection	heteo/homo features, e.g. SIFT, Intensity distrib.	arbitrary object type, large variability, occlusions
Boosted cascade of Haar-like features[VJ01]	Structured object, e.g. frontal faces	Simple haar-like feature	Variability in structured object with similar pose

Table 1.1: A literature survey of some important object detection approaches; it shows the relative importance of the boosting approaches

performance improved substantially both on speed and accuracy. The Boosted histogram framework is also used in Laptev[Lap06] for detecting object classes like motorbikes, cars etc. The results presented in PASCAL VOC (visual object categories, a prominent object detection challenge) shows superiority over other approaches.

The powerfulness of the Boosting framework is further demonstrated by Opelt et al[OPFA06] in their work towards generic object detection. Their work can cope with object class with large variability in scales, viewpoints, poses, occlusions and is thus termed generic. They extract information using a combination of different feature descriptors like SIFT, Invariant Moments, Intensity distributions extracted from regions of discontinuity or homogeneity. Then they employ boosting to find the most representative set of features to form a strong classifier. Their approach detects generic object class in very complex images with promising results.

Obviously, Boosting methods are applied in a wide range of object classes detection tasks. Consequently, it is safe to say that Boosting is a very promising approach for object detection. Its effectiveness comes with two main reasons.

1. Boosting can learn from an over-complete set of features that are rich in describing domain knowledge. Other approaches like SVM or neural networks that provide a single strong classifier can’t make use of such a large feature set efficiently.

2. Boosting learns only a small representative set of features as the strong classifier. A cascade of the boosted classifiers can form a highly efficient final classifier, for discarding non-object region aggressively.

The first successful use of Boosting framework in object detection was proposed by Viola and Jones[VJ01] in the context of face detection. Based on haar-like simple features and employed classifiers cascade, their work achieves both high detection rate and high speed. The haar-like feature set encodes redundant representation of domain knowledge and yet is simple to compute. The boosted strong classifier ensures high acceptance rate for image sub-windows containing object, while rejecting majority of negative sub-windows. A cascade of strong classifiers can further improve the performance and speed. Most negative sub-windows are rejected at early stages, focusing processing on promising image regions.

1.1.2 GPU as Efficient Data Coprocessor

Considering the future Internet where the usage of multimedia contents will be equal or even richer than text, the current state-of-art object detection techniques are far from sufficient. In terms of detection speed, we need fast computational architecture that can deal with the ocean amount of multimedia contents growing each day. CPU alone is not particularly suitable for this task as CPU is designed for handling complex logic, which sacrifices data throughput.

From section 1.1.1, we understand that Boosting is an efficient and promising framework for object detection. Although the boosted cascaded detector is one of the fastest state-of-art detectors, its speed in CPU is not entirely up to expectations. For people detection in Zhu[ZYCA06], the system can only process about 5 images of resolution 320×280 per second (fps). For a simpler task of frontal face detection, Lienhart[LM02] reported a 5 fps for image resolution of 320×240 on a Pentium-4 2GHz. Given nowadays high resolution images, this speed is well below many applications' requirements.

From analyzing the boosting cascade in the classification stage, we know that the computation bottleneck resides on evaluating early stages of the classifier cascade. As all or most of the pixel locations are required to be evaluated at the stages, this task is particularly data intensive. When large resolution images or batch images are required to be processed, CPU is very limited in performance.

Graphics processing units (GPUs), on the other hand, provide attractive alternatives. Originally designed for speeding up graphics applications, GPU is best suited for data intensive computational task. As its computation capacity and programmability evolves rapidly over time, general purpose computations on GPU (GPGPU[OLG$^+$07]) has become feasible, and more and more popular. GPUs can potentially speed up applications that exhibit large data parallelism, as its built-in parallel data processing capacity over performs CPU substantially.

For generous purpose computations, a streaming computation paradigm is usually formalized on GPU, where input/output data are treated as streams, and computations on them are expressed as kernels operating on them. The programmability of GPU also gets mature in recent years. The OpenGL library[WNDS99] and Cg (stands for C for graphics)[MGA03] for programming GPU cores provide an expression of the computation in graphics terms. More recently, several high-level languages that encode the streaming concept are also implemented. Examples are Brook, and CUDA from NVIDIA.

In this thesis, we explore the use of GPU programming model to as an efficient architecture for object detection.

1.2 Thesis objective and organizations

From the above we understand that Boosting is a promising object detection approach, and GPU as a data streaming coprocessor provides an opportunity to speed up the detection significantly. Consequently, this thesis work aims to design and implement a computation framework that uses GPU as data coprocessor to speedup the Boosted Cascade detection framework. Note that our focus is on the classification stage, i.e. the detecting process on the images. The concrete objectives include:

1. To explore the possibility of using GPU to speedup detecting objects in large images, based on promising detection approach

2. To look for efficient design and implementation of GPU-based object detector, leveraging on GPU architecture and hardware characteristics

The contributions of our thesis are in three-fold:

1. The design principals and considerations were investigated to come out an efficient implementation for the detector.

2. Practical experiments on large images were performed on our GPU-based detectors on various cases to see the performance impact. Up to 5 times speedups were obtained over pure CPU-based version.

3. Hardware characteristics are also taken into account; as a result, we implemented both single-image based detectors and multi-image based version which required more memory operations.

4. We experimented the performance differences on two GPUs of different generations. This further explores the suitable designs for object detection on future GPUs.

To do this work, we firstly identify the computation part that requires large data-parallel processing and map it to GPU by expressing it in GPU computation paradigm. Experiments are performed to examine the effectiveness of the implementation. The principle and usability of GPU is investigated by discussion on the results. By doing this work, the limitation and future applicability of GPU on similar tasks can also be revealed.

In chapter 2, we have a closer look at the Boosting algorithm for object detection, in the context of frontal face detection. The core computation for classification that can be candidate for porting to GPU will be presented in this chapter. In chapter 3, GPU as a data streaming coprocessor will be shown in more detail, with a survey in the history of general purpose computation in GPU. After a literature review on application of GPU in computer vision, the design of GPU based object detection framework will be presented in chapter 4. Experimentations of the GPU-based boosted detector will be presented in chapter 5, where we will also discuss and analysis their results. Chapter 6 will conclude this work with some possible future outlook.

Chapter 2

Boosted Object Detection Framework

2.1 Overview of Object Detection Framework

Generally, a visual object detection framework consists a learning stage and a classification stage (figure 1.1). Both stages involve visual feature extraction, which extracts the local feature of the objects from the image. Depending on type of objects to detect, different types of features are extracted ranging from simple linear haar-like features to complex features that are in some sense invariant to scale or view-points. SIFT, histogram of gradient orientations are examples of complex features, which are more expensive to compute.

The learning stage learns a representational model and a discriminative classifier for the object class. The learning is supervised or semi-supervised, depending on how the object instances in the training images are labeled. Usually a rectangle enclosing the object region is drawn for supervised learning. The corresponding features are extracted from the labeled region. Clustering of the features and analysis of how representative of the feature clusters for the object class are performed. Statistical (probability density estimation or Gaussian modeling, etc) or discriminative methods (neural networks, AdaBoost learning, etc) can be applied to find the most distinctive and representative set of feature clusters that represent the object class.

The classification stage refers to the procedure to classify input data (image) according to the model or classifier learned. Most of the solutions slide a detection window over the input image, thus are termed as ”scanning window” approaches. As an image can contain object of variable scale and size, a fix-size window is usually scanned in different scales of the image (image pyramid). On the sub-window, the classifier is applied based on extracted feature value. Lastly, the classification results are combined to form the final detection results. At the core of most scanning-window algorithms is usually a discriminative classifier, AdaBoost classifier, neural networks or support vector machines, etc.

From the introduction we understand that Boosting algorithms are promising approaches for learning and detection, as they combine the procedure of learning an efficient classifier with selecting which features to extract. The boosted cascade classifier based on simple features proposed by Viola and Jones[VJ01] is the first robust and near real-time object detector in the literature. Lienhart et al[LKP02] gives a further extension and comprehensive analysis of the approach. It achieves state-of-art detection accuracy for highly structured object as frontal face. The following sections describe the features it uses and its learning and classification mechanism.

2.2 Boosted Cascade of Simple Features

2.2.1 Features Pool

In principle, the main purpose of using feature over raw data is to reduce/increase intra-/inter-class variability. The Boosted framework uses simple linear feature set, which are reminiscent of Haar Basis functions. A large and general pool of simple haar-like feature combined with feature selection encodes domain knowledge a lot better than raw input image data, thus is easier for classification.

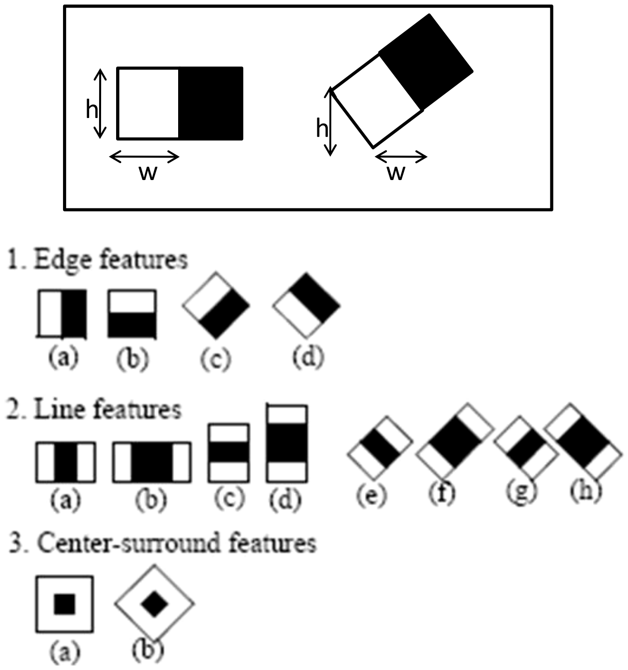

Figure 2.1: Haar-like feature in a subwindow, and the types of them.

In Lienhart's work[LM02], a feature is formally defined as the weighted combination of pixel sums of two upright or $45°$ rotated rectangles (figure 2.1). Assume the detection subwindow is of $W \times H$ pixels. A rectangle is specified by a tuple $r = (x, y, w, h, \alpha)$ with $0 \leq x, x{+}w \leq W$, $0 \leq y, y{+}h \leq H$, $x, y, w, h \geq 0$, and $\alpha \in \{0°, 45°\}$, with (x, y) being the top left corner of the rectangle; its pixel sum is denoted by $RecSum(r)$. The weights of two rectangles have opposite signs, and are used to compensate for the difference in area size between the two rectangles. In mathematics notation, we have $-w_0\dot{A}rea(r_0) = w_1\dot{A}rea(r_1)$. To evaluate the feature, we use $feature_I = \sum_{i\in I=\{1,2\}} \omega_i\dot{R}ecSum(r_i)$.

The feature pool consists of three basic types of feature prototypes as shown in the figure 2.1: *four-edge feature, Eight-line feature, and center-surround feature*. The aspect ratio of from rectangle r_0 to r_1 is fixed in each prototype. But the prototypes are scaled independently for its w and h, in order to generate an over-complete set of features.

The number of features within a detection window far exceed the number of pixels. For example, there are a total of $225, 897$ per detection window of size 24×24. This provides the basics for learning.

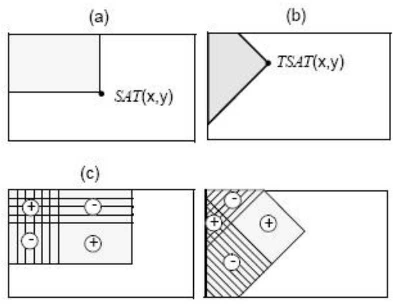

Figure 2.2: Computation of integral images(SAT and RSAT).

The haar-like feature can be computed in constant time for any size and location by means of an image representation called *Integral Image* (also named *Sum Area Table* for upright rectangles and *Rotated Sum Area Table* for 45° rotated rectangles), as shown in figure 2.2. The sum area table can be computed from input image I in one pass over all the pixels from left to right and top to bottom using

$$SAT(x, y) = SAT(x, y-1) + SAT(x-1, y) + I(x, y) - SAT(x-1, y-1),$$

The rotated sum area table can be computed with two passes. The first pass from left to right and top to bottom determines

$$RSAT(x, y) = RSAT(x-1, y-1) + RSAT(x-2, y) + I(x, y) - RSAT(x-2, y-1)$$

whereas the second pass from right to left and bottom to top calculates

$$RSAT(x, y) = RSAT(x, y) + RSAT(x-1, y+1) - RSAT(x-2, y)$$

From the Integral images, the sum of any rectangle can be computed by four table lookups. For upright rectangle $r = (x, y, w, h, 0)$, $RecSum(r) = SAT(x-1, y-1) +$

$SAT(x+w-1)+SAT(y+h-1)-SAT(x-1,y+h-1)-SAT(x+w-1,y-1)$. For $45°$ rotated rectangle, $RecSum(r) = RSAT(x+w,y+w)+RSAT(x-h,y+h)-RSAT(x,y)-RSAT(x+w-h,y+w+h)$.

Before training or detection, the detection windows are normalized against illumination effects. The special properties of the haar-like features also enable fast contrast stretching of the form $\overline{I}(x,y) = \frac{I(x,y)-\mu}{c\sigma}, c \in R^{+}$. μ can be easily looked up from $SAT(I)$, whereas σ can be computed by looking up from $SAT(I^2)$ which is the integral image of squared image of I.

2.2.2 Learning Cascade of Boosted Classifiers

Boosting is a learning technique combining a number of weak-classifiers to form a strong classifier. In particular, AdaBoost provides an effective learning algorithm and strong bounds on generalization performance. Viola. et al[VJ01] stated that its training error approaches zero exponentially in the number of rounds, and it achieves large margin of training samples rapidly.

Learning is based N weighted training samples $(x_1,y_1),\ldots(x_N,y_N)$ where $y_i \in \{1,-1\}$, x_i is a vector of feature values evaluated from the sample image. A weak classifier is only required to better than chance. In a simple case it can be represented by $f_m(x|t_m) \in \{1,-1\}$ where t_m is the threshold for the selected feature value. In each training round, a weak-classifier is produced by selecting one or several features that minimize the weighted training error $err_m = E_w[1_{(y \neq f_m(x))}]$; weights of the samples are updated so that more weights are put to misclassified samples. The final strong classifier learned is of the form $C(x) = sign(\Sigma_{m=1}^{M_c} \alpha_m \dot{f}_m(x|t_m) + b)$, where $\alpha = \log((1-err_m)/err_m)$.

Lienhart et al[LKP02] extended the simple weak-classifier that only depends on one feature (therefore called ”stump-based”) to two or more features. A weak-classifier employs $NSPLIT$ features is in the form of CART tree. This permits the expression of dependency between different features, therefore is often superior than stump-based in detection tasks.

From the observation it is clear that vast majority of image regions contains no object of interests. Thus it is essential to be able to reject those regions in early stages for fast processing. Based on this principle, a cascade of boosted classifiers are built. The detection process is then essentially of a degenerated decision tree. If any classifier in one stage rejects the sub-window, no further processing is performed.

To preserve detection accuracy, however, in training any stage, the classifier must be able to detect almost all faces while rejecting certain fraction of non-faces. That is, to train any successive stage, one should set a target of minimum detection rate and maximum allowed false positives. For example, in each stage we require a hit rate of 99.8% while eliminating 50% of the non-faces. After 13 stages we can expect false positive rate of $0.5^{13} \approx 1.2e-04$ and a hit rate of $0.998^{13} \approx 0.97$. This is illustrated in figure **??**. The negative training patterns for stage n of the cascade are collected by scanning the partial cascade (stage 0 to $n-1$) across non-face images and collect false positives.

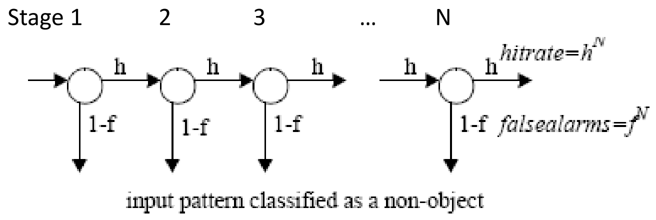

Figure 2.3: Cascade of the Boosted Classifiers.

The cascade can be viewed as an object-specific focus-of-attention mechanism. However it provides no statistical guarantees that discarded regions are unlikely to contain the object of interest.

2.2.3 Analysis of the Detection Cascade

The detection cascade employed in the classification stage has increasingly more weak-classifiers in successive stages. For example, in a frontal face detector cascade (total of 20 stages and features) provided by Lienhart[LKP02], the number of weak-classifiers for the first five stages is $3, 9, 14, 19, 20$ and 27 respectively. This is obvious as in training each successive stage, meeting the target hit rate and false positive rate becomes increasingly difficult.

During detection, a sliding window was moved pixel by pixel over the picture at each scale. Instead of scaling the picture, it is more efficient to scale the detection window as evaluating the scaled feature takes constant time using integral image.

Lienhart[LKP02] gives an empirical analysis of different configurations and variants of the detection cascade of Boosted classifiers. Experimented on the CMU frontal face test set, they reported 20×20 as optimal input pattern size; Gentle AdaBoost as the better variant of the AdaBoost learning algorithms; and CART tree as better weak-classifier than stump-based.

The detection time they reported is about $5fps$ on 320×240 images with a scaling factor of 1.2 on detection windows. The detection speed is closely related to the average number of features evaluated per detection sub-window. Viola & Jones[VJ01] reported an average of 10 features out of 6061 was evaluated per sub-window in their version of detector. This is possible because a large portion of sub-windows are rejected at first few stages.

While the speed is promising for small image size or small set of images for processing, it is surely not satisfiable for large amount of processing. This is particularly true for today's internet with huge amount of multimedia data.

From the above analysis, we observed that the largest portion of the computation focus on the first few stages of the cascade. From the initial stage, the detection sub-window must be put regularly on image pixels; in each sub-window, table lookups must be performed to evaluates corresponding features in the classifiers, making the computation extremely data-intensive. As memory is the main bottleneck in CPU computations, the first few stages contribute most in the detection time.

GPUs, on the other hand, provide an best alternative for data intensive applications. In the next chapter, we will look at GPU as efficient data streaming coprocessors, and give an implementation of the detection cascade in terms of GPU computation model.

Chapter 3

GPU as Data Streaming Coprocessor

In the past few years, graphics processing units (GPU) have evolved to be one of the most powerful computational hardware for a given price. Specifically designed for speeding up graphics applications, GPUs provide tremendous memory bandwidth and computational horsepower. Currently, the peak performance of state-of-the-art consumer graphics cards is more than ten times faster than that of comparable CPUs. For example, high-end GPUs such as NVIDIA GeForce 9800 GTX+ boasts 70.4GB/sec memory bandwidth, and can sustained a measured 705 GFLOPS floating-point arithmetics, compared to 8.5 GB/sec and 25.6GFLOPS theoretical peak for a dual-core 3.7GHz Intel Pentium Extreme as high-end CPU. GPUs are faster and getting faster quickly. The GeForce 7800GTX (165GFLOPS) more than triples that of its predecessor, GeForce 6800GTX. In general, the computational power of GPUs compound at a yearly rate of about 1.7, significantly out-paced the Moore's Law as applied to microprocessors, which is 1.4 for CPU performance.

This disparity arises from their fundamental architecture differences. CPUs are optimized for sequential code, with transistors dedicated to extracting instruction-level parallelism. On the other hand, the data-parallel nature of graphics applications enable GPUs dedicating transistors uses to computation.

GPUs are commodity components nowadays. Their programmability is increasingly enhanced. As a result, many researchers have advocated the use of GPUs as a co-processor[BFH$^+$04, Pur04] for general purpose computations, commonly termed GPGPU. GPGPU are deployed in a wide range of applications, scientific computing, database processing, bio-informatics, and image processing, to name a few. Owens et al.[OLG$^+$07] provides a detailed survey of the GPGPU framework and a variety of applications. An active online community can also be found at gpgpu.org.

Architecturally, GPGPU implements what is referred to as data streaming processor [BFH$^+$04]. Streaming computation paradigm provides an efficient abstract of computing tasks that are data-centric, i.e. involving intensive data processing that are parallel in nature.

The following section examines the GPGPU programming model in details.

3.1 GPGPU Programming Model

3.1.1 Graphics Pipeline

Computation on a GPU follows a fixed order of processing stages, called the graphics pipeline (see Figure 3.1). The pipeline consists of three stages: vertex processing, rasterization and fragment processing. The vertex processing stage transforms three-dimensional vertex world coordinates into two-dimensional vertex screen coordinates. The rasterizer then converts the geometric vertex representation into an image fragment representation. Finally, the fragment processor forms a color for each pixel by reading texels (texture values of pixels) from the texture memory. Modern GPUs support programmability of the vertex and fragment processor. Fragment programs for instance can be used to implement any mathematical operation on one or more input vectors (textures or fragments) to compute the color of a pixel.

In order to meet the ever increasing performance requirements set by the gaming industry, modern GPUs use two types of parallelism. Firstly, multiple processors work on the vertex and fragment processing stage, i.e. they operate on different vertices and fragments in parallel. For example, a typical mid-range graphics card such as the nVidia GeForce 7800GTX has 10 vertex processors and 24 fragment processors. Secondly, operations on 4-dimensional vectors (the four channels Red/Green/Blue/Alpha (RGBA)) are natively supported without performance loss.

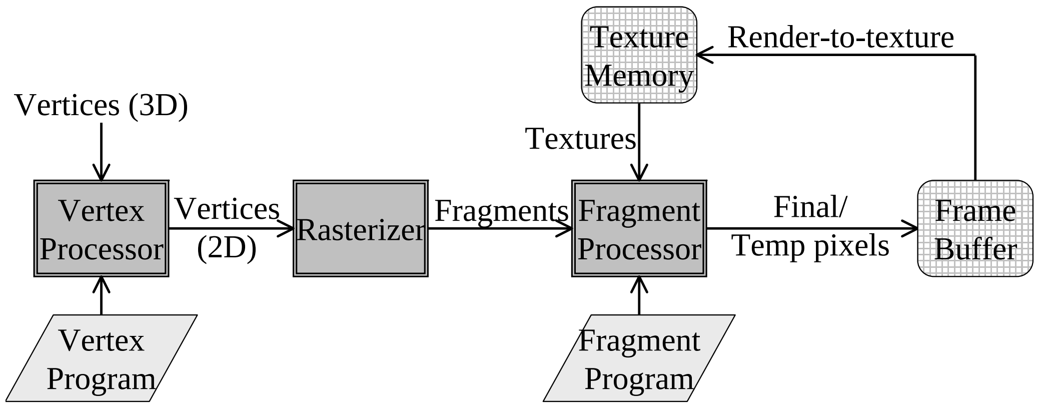

Figure 3.1: Graphics Pipeline

3.1.2 Streaming Paradigm for General-purpose Computation

Several researchers have described GPU as *streaming processor* (Buck et al [BFH$^+$04], Lefohn et al [LKO05], Owens et al [OLG$^+$07] and Purcell [Pur04]). Streaming processors read an input stream, apply the kernel to the stream and write the results into an output stream. In case of several kernels, the output stream of the leading kernel is the input stream for the following kernel (see Figure 3.2). Pixel computations must be completely independent. Scatter operations that enable different fragment processors writing to the same pixel are not supported.

The vast majority of general-purpose GPU applications use only fragment programs

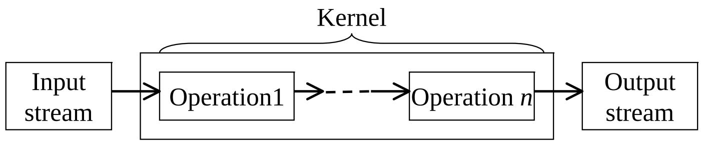

Figure 3.2: Streaming model that applies kernels to an input stream and writes to an output stream.

for their computation. In this case textures are considered as input streams and the textures as render targets are output streams. Because currently fragment processors are SIMD architectures, only one program can be loaded at a time. Applying several kernels thus means to do several passes (see figure 3.3).

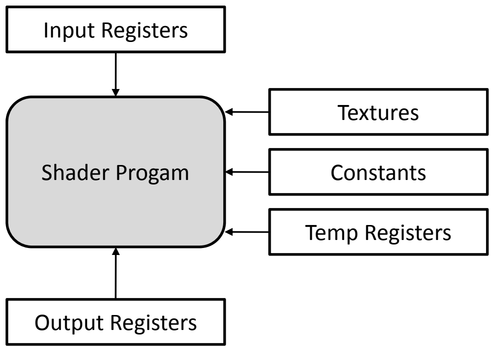

Figure 3.3: Fragment program in a GPU pass as kernel computation

A typical GPGPU program is structured as follows.

(a) Data-parallel sections of the application are identified by the programmer. Each such section can be considered a kernel and is implemented as a fragment program. The input and output of each kernel are one or more data arrays, which are often stored in 2D textures in GPU memory.

(b) To invoke a kernel, the range of the computation (or the size of the output stream) must be specified. The programmer does this by passing vertices to the GPU. A typical GPGPU invocation is a quadrilateral (*quad*) oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array.

(c) The rasterizer generates a fragment for every pixel location in the quad, producing thousands to millions of fragments.

(d) Each of the generated fragments is then processed by the active kernel fragment program. The fragment program can read from arbitrary texture memory locations but can only write to memory locations corresponding to the location of the fragment in the frame buffer.

(e) The output of the fragment program is a value (or vector of values) per fragment. This output may be the final result of the application, or it may be stored as a texture and then used in subsequent passes. This feedback loop is realized by using the output buffer of a completed pass as input texture for the following one (known as *render-to-texture* (RTT) (figure 3.1)).

(f) If the output of the fragment program will be further processed on the CPU, data read-back from GPU to CPU is required. Because of the relatively low bus bandwidth between the CPU and GPU, the read-back operation is a known bottleneck for GPGPU applications and should be minimized.

3.1.3 Programming Languages

The high-level GPU programming languages are often referred to as *shading languages*, as they are designed for rendering pictures from graphics primitives. Cg[MGA03] or HLSL enable programming in a C-like programming syntax. The programs then are compiled into vertex or fragment shader which are bound as computation kernel when rendering graphics. OpenGL[WNDS99] enables communication between CPU and GPU by setting the rendering context in GPU. The input and output of the program are set up as textures. Data is communicated between CPU and GPU memories using OpenGL library calls.

Many efforts are in place to make the abstraction of GPU to streaming paradigm seamless, which provide programming languages for expressing streaming computation without graphics knowledge. These includes Brook[BFH$^+$04] from Stanford, Accelerator[TPO06] from Microsoft, and most notably CUDA[NBGS08] from NVIDIA. However, we choose to implement our object detector to base on OpenGL and Cg, mainly because by using them we can have more controls over hardware details. Consequently we can study more detailed effects of object detector in GPU with hardware constraints.

3.2 Hardware Characteristics and Constraints

As GPU architecture is designed for graphics rendering tasks, there are a number of hardware characteristics and constraints. Firstly, input and output are bound as separated textures; inputs as textures permit random access, however, outputs for a pixel are written to a fix position. This is to ensure high data throughput. As a result, GPU data structure is not as flexible as CPU's. We need to re-design the data structure for efficient processing on GPU.

Secondly, current GPU architectures have limited support for flow controlling[HB05]. Fragment processors process pixels in SIMD groups, within which both sides of branch must be evaluated if one or more processors evaluate the condition differently, thereby leading to reduced performance. This has special performance implications to our implementations of GPU-based object detectors. If the algorithm on GPU is to detect one image at a time, branching support can be employed to efficiently evaluate detector. On the other hand, if multiple images are to be processed in GPU at a time (i.e. each pixel is meant for computing multiple images), all computation paths have to be taken by each pixel. However, there will still be gain in performance. More details of the algorithms and their effects are presented in chapter 4 and 5.

Thirdly, there is also a limited number of constant registers in current GPUs. For example, the NVIDIA GeForce 7900GTX has 32 constant registers. This limits the complexity of the program we can use.

The positive sides of the architectures include the native support for vector arithmetics, e.g. RGBA color computation; and support for multiple outputs per any graphics primitives, termed as *multiple render targets (MRT)*. Up to 4 render textures are supported, which help GPGPU programs a lot as it leads to larger program inputs, hence harnessing greater GPU computation power.

In the next chapter, our design of cascaded boosted object detector on GPU will be presented. The design principle is generally applicable while catering hardware characteristics.

Chapter 4

GPU-based Cascaded Boosted Detector

4.1 Related Work

The use of graphics processor for general-purpose computation grows rapidly in recent years. A wide range of algorithms from graphics-related global illuminations to signal and image processing, from database operations to system designs[FCS$^+$07] etc have been ported to GPU. Owens et al.[OLG$^+$07] provides an excellent survey on GPGPU.

In computer vision, GPGPU has been explored in topics such as image filtering, tracking and geometrical transformations. NVIDIA's *"GPU Gems 2"*[PF05] book has a chapter dedicated to computer vision, which presents Canny edge detector, tracking hands etc. Yang et al.[YPL04] implements real-time stereo on GPU. Ohmer[OMB05] implemented a face recognition system using Kernel PCA and SVM. Labatut et al.[LKP06] implemented level-set based multi-view stereo. The well-known SIFT algorithms was implemented by Heymann et al.[HMS$^+$07] in 2007. There are also libraries providing re-usable implementations of basic vision algorithms such as OpenVIDIA[FM05] and GPUCV[pFHGA06].

However, the use of GPU in high level vision task such as object detection is rare. Li Zhang et al.[ZN08] built on GPU a pedestrian detector that uses histogram of oriented gradient (HOG) features and SVM classifier. Their detector is scan-window based, as most of the prevailing approaches are. But the classifier is a single strong classifier, unlike our cascaded boosted classifiers. Consequently, our implementation of cascaded boosted classifiers provides more general knowledge on applying complex and promising object detectors on GPU.

4.2 Design and Implementations

The success of GPU-based algorithms depend on the degree of data parallelism of the computation, and that the computation intensity significantly prevails the load to transfer data from CPU to GPU. Therefore, we first analyzed the cascaded boosted classifiers to identify the portion exhibiting that characteristics.

In the approach, feature extraction and classification steps are repeated for every scan window. The number of input scan windows reduced dramatically in successive stages, as a nature of the attentional cascades. Therefore we can predict that the first few stages are good candidates for porting to GPU. Windows scanning is also performed at every scales. Detections on different scales can be implemented on GPU as multi-pass by rendering a target with the re-scaled sizes. In our approach, the scan-windows is resized instead of input image because of the easy in computing feature in scan window at different scale. The classification results from GPU can then be downloaded and used in CPU to form the final detection results.

The architectural design of our GPU-based detector is shown as in figure 4.1. This includes design considerations for all function modules for feature extractions and classifications. Among them, *integral images* computations and classifications of last stages are mapped into CPU. Integral image computation is highly sequential, and take only one to two pass of the image to compute. As last stages are more complex, and there are only a small number of sub-windows to compute, mapping them to GPU will hinder the performance. This is because the SIMD-based fragment processors will be stalled until all the expensive computations finished. Each pass of the GPU corresponds to a detection

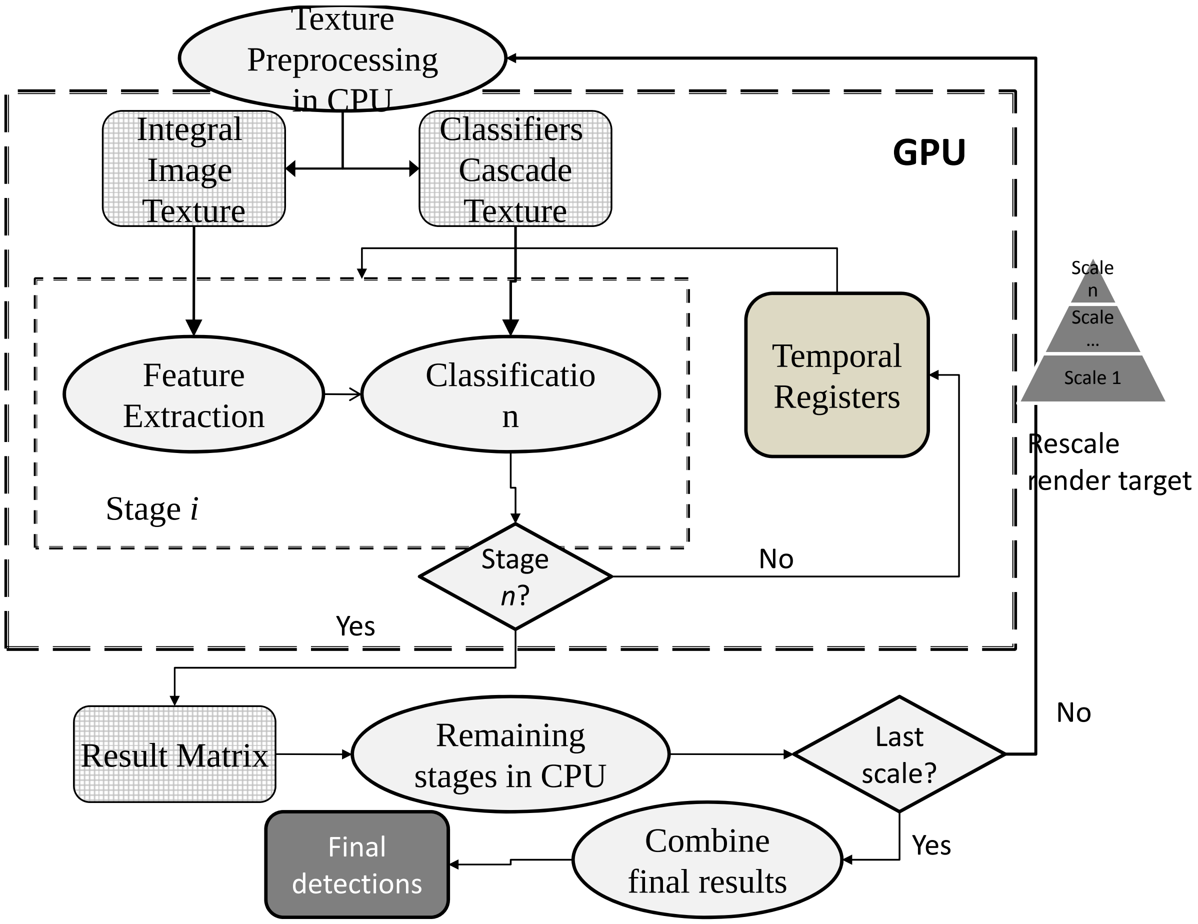

Figure 4.1: GPU-based architecture for boosted cascade object detection.

in different scale. Resizing the feature is also done in CPU between successive passes. As the reference rectangle coordinates of the features are used identically across all scan windows, it is beneficial to map the computation to CPU to avoid redundant computations in GPU in every pixel.

The data-intensive parts of feature extraction and classifications in the number of stages in the cascade are mapped onto GPU.

To further harness GPU's capacity in vector arithmetics, we also implement versions of detecting on multiple images in a single pass. For experiments, we implements 4 images in a pass.

4.2.1 Data Structures

Careful design considerations have been given to GPU data structure. The main objectives are to enable efficient memory operations in GPU for feature evaluation and classifications, as well as to minimize CPU-GPU data transfer.

For feature computations, we represent integral images as texture identical in the size of input image I. Basically 4 types of textures are used to store the integral images, corresponding to *(rotated) area sum tables*, denoted $SAT(I), SAT(I^2)$ and $RSAT(I), RSAT(I^2)$ of the squared image as in chapter 2. For detection on 4 images in a single pass, one *RGBA* texture is used to store one type of integral images for 4 images, e.g $4SAT(I)$ or $SAT(I^2)$. The integral-image textures have to be download to GPU only once for the multi-pass algorithm as they are constant during the computation. For classification, we

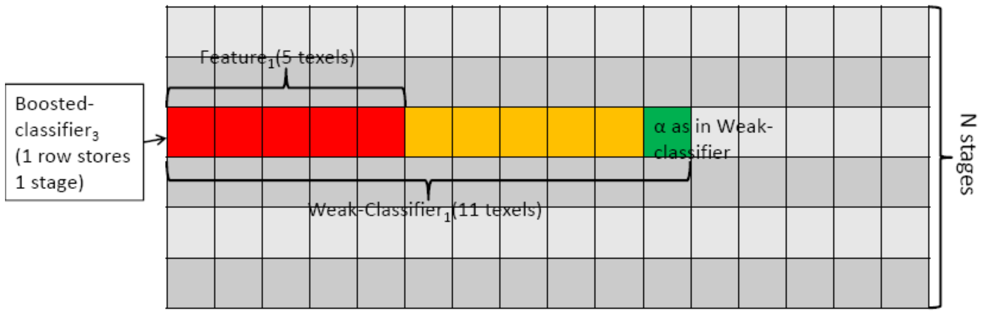
Feature₁(5 texels)
Boosted-classifier₃ (1 row stores 1 stage)
α as in Weak-classifier
Weak-Classifier₁(11 texels)
N stages

Figure 4.2: GPU-based classifier cascade texture: updated every pass to adapt fractional re-scaling.

need to store the classifiers cascade in GPU as texture. Unlike CPU which uses structure to store classifiers cascade and use pointers pointing to classifiers and features, we need to re-formulate it to texture data layout. Figure 4.2 shows our implementation of the cascade data structure in GPU. To store n stages in GPU, we define a *RGBA* texture with height n.

For a cascade using CART-based weak-classifier, each row of the texture align a regular interval of grids to store each simple classifier; inside the interval, 5 grids of pixels are allocated to store a feature, where 3 pixels are used to store the rectangles information (RGBA corresponding to the reference top-left (x, y), *width* and *height*), and another 2 pixels store weights of the rectangles, classifier threshold, left and right indexes (used to reference to α); the last pixel in the interval stores the α values of the weak-classifier.

In each re-scaling step, fractional re-scaling of the feature leads to fractional position of rectangles. Unfortunately, a simple rounding to integral position leads to severe degrade of performance, therefore a weight-correction is necessary to preserve the feature properties, which is relatively expensive computation. Keeping this computation in fragment processing would generate highly redundant computation load as it is repeated at each pixel. As a result, we use CPU to update the classifier data structure at each re-scaling of the multi-pass algorithm.

The results matrix of sub-window classification results (1 if positive, ≤ 0 if negative) is of the size of the scaled image. In each pass, a texture of that size bound as render target is defined. At the end of each pass, the results matrix is downloaded to CPU to use as mask for computation in the remaining stages. *Render-to-texture* support is provided by the framebuffer object (FBO) extension of the OpenGL library.

4.2.2 Algorithms and Program Flow

To start the computation in GPU, OpenGL and Cg context are firstly initialized, followed by creation and upload of integral-image textures. In a loop over different detection scales, the classifier data texture is computed and uploaded to GPU; a quadrilateral of the scaled image size is rendered to trigger the actual computation in fragment processors; the results matrix is downloaded for computation in remaining classification stages. Finally, the results from all scales are combined to produce the final detections.

Algorithm 1 depicts the whole algorithm that implements GPU as data streaming coprocessor.

The algorithm 1 can either detect a single image at a time, or multiple (we choose 4 for experiments) images. The algorithm computes the results matrices R corresponding to the scanning subwindow in each pixel location in each scale during executions. The final output is a sequence of detected sub-windows in Seq. The $StreamRead$, and $StreamWrite$ operations do the uploading and downloading of the GPU data. They are termed according to the streaming computation paradigm. Inside the while loop, the $ClassifyOnGPU$ function evaluates the a number of beginning stages on GPU, which is also the computational intensive part. CPU evaluates the remaining stages according to results matrix returned by GPU. After the loop, the overlapping list of positive sub-windows are combined to produce the final detection results.

The following algorithms depict the fragment processing part in GPU for feature extraction and classification. Two types of implementations are included: one is for detecting one image at a time, as shown in algorithms 2 and 3; the other (algorithms 4 and 5) is for detecting multiple images at a time, for which the results matrix are stored in *RGBA* channels of a single or multiple render targets. The algorithms are designed to be able to handle CART tree as weak-classifier.

Note that algorithm 2, 3 and algorithm 4, 5 differs primarily in the branch handling parts. All ”if” and ”else” in algorithm 2 and 3 have been replaced with other constructs in algorithm 4 and 5. As each pixel of GPU stores intensities of 4 images, in order to use the *RGBA* vector arithmetics, both sides of the branches have to be taken. The correct results are only written to destinations at the final step. This is essentially ”predication”, one of the branch handling approach.

This approach appears not efficient. However, provided GPU’s inherent support for vector arithmetics, it may still be satisfiable. In addition, predication is also common approach in branch handling for data-parallel architectures. The SIMD architecture of

Algorithm 1 The whole algorithm for GPU-based Boosted Cascade of object detection

```
Require: Intensity image array $I[n], n \in \{1,4\}$ where $n$ is the number of images in
    processed in GPU at a time; a cascade of boosted classifier $C$ as a CPU structure; a
    scale factor $r$
Ensure: Results matrix $R[n], n \in \{1,4\}$; detected sub-windows sequence $Seq$
 1: scale $\leftarrow 1$
 2: Build Integral images $SAT(I), SAT(I^2)$ and $RSAT(I), RSAT(I^2)$
 3: $StreamRead(SAT(I), SAT(I^2)$ and $RSAT(I), RSAT(I^2))$
 4: while scale $*$ scan window size $<$ image size do
 5:     Build classifier texture $C_t$, $StreamRead(C_t)$
 6:     Build $R$ with size (image size / scale)
 7:     $ClassifyOnGPU(SAT(I), SAT(I^2), RSAT(I), RSAT(I^2), R)$
 8:     $SreamWriteR$
 9:     for all image coordinates $(x, y) \in R$ do
10:         if $R(x, y) = 1$ then
11:             $ClassifyOnCPU(SAT(I), SAT(I^2), RSAT(I), RSAT(I^2)$, and $Seq)$
12:         end if
13:     end for
14:     scale = scale $*r$
15: end while
16: $CombineResults(Seq)$
17: Output the final sequence of detected sub-windows in $Seq$
```

Algorithm 2 Fragment program in GPU for detecting objects on single image

```
Require: $SAT(I), SAT(I^2), RSAT(I), RSAT(I^2)$ and classifier cascade texture $C_t$
Ensure: Results matrix $R$ as single channel render texture
 1: Variance normalized factor $\sigma \leftarrow eval\_norm(SAT(I^2), RSAT(I^2))$
 2: for $i \leftarrow 1$ to number of stages to compute in GPU do
 3:     initialize $stage\_sum$ to $0$
 4:     for $j \leftarrow 1$ to number of weak-classifiers in stage $i$ do
 5:         $stage\_sum \leftarrow stage\_sum + eval\_CART\_tree(SAT(I^2), RSAT(I^2), \sigma C_t[j])$
 6:     end for
 7:     if $stage\_sum < stage\_threshold$ then
 8:         return $R \leftarrow 0$
 9:     end if
10: end for
11: return $R \leftarrow 1$
```

Algorithm 3 compute $eval\_CART\_tree$

```
Ensure: $\alpha$ by the evaluated tree node
 1: Root node haar-like feature value $v_{feat} \leftarrow eval\_feat(SAT(I), RSAT(I))$
 2: if root node threshold $*\sigma > v_{feat}$ then
 3:     continue to evaluate child node
 4: else
 5:     return $\alpha$ by this node
 6: end if
```

Algorithm 4 Fragment program in GPU for detecting objects on multiple image

```
Require: $SAT(I), SAT(I^2), RSAT(I), RSAT(I^2)$ and classifier cascade texture $C_t$
Ensure: Results matrix $R$ as *RGBA* render texture
1: Variance normalized factor $\sigma \leftarrow eval\_norm(SAT(I^2), RSAT(I^2))$
2: $R \leftarrow 1$
3: for $i \leftarrow 1$ to number of stages to compute in GPU do
4:     $stage\_sum \leftarrow 0$
5:     for $j \leftarrow 1$ to number of weak-classifiers in stage $i$ do
6:         $stage\_sum \leftarrow stage\_sum + eval\_CART\_tree(SAT(I^2), RSAT(I^2), \sigma C_t[j])$
7:     end for
8:     $R\_temp \leftarrow (stage\_sum < stage\_threshold)?0 : 1$
9:     $R \leftarrow (r = 1)?R\_temp : 0$
10: end for
```

Algorithm 5 compute $eval\_CART\_tree$

```
Ensure: $\alpha$ by the evaluated tree node
1: Root node haar-like feature value $v_{feat} \leftarrow eval\_feat(SAT(I), RSAT(I))$
2: $index$ =(root node threshold $*\sigma > v_{feat}$)?$left$:$right$
3: continue to evaluate child node, return corresponding $index1$
4: $index = (index > 0)?index1 : index$
5: return $\alpha[index]$
```

current fragment processors often need to evaluate both sides of branch if group of them evaluate the branch differently, which essentially reduced to ”predication”. Thus we predict that processing multiple images in a GPU-pass will still provide benefits. Experiments are needed to uncover the myth.

In the next chapter, we will present the experiments setup and discuss on the results.

Chapter 5

Experiments and Results Discussions

The primary goal of our experiments is to prove the superiority of the GPU-based streaming architecture for Boosted cascade of object detection, in terms of speed. In addition, the experiments also aim to examine the architecture's scalability and robustness in terms of handling different set of inputs and running programs of different complexity, while providing benefits.

To this aim, we have implemented different versions of the GPU-based detector, and tested on different inputs. Experiments that address different goals are briefly described as following.

- As each GPU-pass is applied in a level in the input image pyramid, the top levels perhaps provide too small input for GPU computation. We tested variants where top levels of the pyramid smaller than a threshold size are not mapped to GPU.

- Different numbers of stages of the cascade are included in the kernel (computed on fragment processors), ranging from 1 stage to a maximum of 8. This proves the scalability of GPU computation as well as showing the performance characteristics of GPU running the detection cascade.

- We varied the inputs by using images in different scales to see the performance impacts of GPU-based detector on input sizes.

- Besides the versions that compute a single image, there are versions that process multiple images (4) in a single GPU-pass, where results are stored in the *RGBA* channels of render targets. This shows benefits in making use of GPU hardware characteristics; its performance implications further reveal guidelines in mapping the computations.

In the following sections, we first describe our experimental setup and then present the results and discussions.

5.1 Experiment Setup

Our experiments were carried out on a Windows PC with a 3.0 GHz Pentium 4 CPU with 1G of RAM. On the same PCI-express board, we experimented both a NVIDIA GeForce 9800GTX card with 1GB RAM, and a GeForce 8800GTS card with 640MB RAM. The PCI-Express bus provides high data bandwidth between CPU and GPU. OpenGL library with Shader Model 3.0 support and Cg is used as the programming platform. The frame-buffer object (FBO) extension supporting *render-to-texture* is used to store the rendered results.

While the GeForce 8800GTS measures a 64 GB/sec memory bandwidth, GeForce 9800 GTX+ features 70.4 GB/sec memory bandwidth. In terms of computational power, GeForce 8800GTS can sustain a measured 345.6 Gigaflops, while the GeForce 9800GTX+ can sustain up to 705 Gigaflops. Depending on application types, we expect that GeForce 9800GTX+ should be 1 to 2 times faster than GeForce 8800GTS. On applications with high data bandwidth requirements, the speedups should not be as obvious as on computation intensive applications.

5.1.1 Software Preassumptions

For the experiments we used a frontal face detector cascade trained by Lienhart[LKP02], which is included in the open source OpenCV library[Bra08]. This detector provides state-of-art performance both on accuracy and speed. As reported in[LKP02], evaluated on the complete CMU face test set, at only tens of false positives, the detector achieves near 90% hit rate. The detector was trained using an input pattern size of 20×20. This indicates that for detection, we will slide a scanning window starting from that size. During detection, we fixed a re-scale factor of 1.1, as this is reported to provide good tradeoff between hit rate and false positives.

The detector cascade consists of 20 stages, and a total of 1047 weak-classifiers. Each weak-classifier is a CART tree. Our input images test set consists of 40 digital photos taken under a variety of scenes, both outdoor and indoor. This collection contains variety of faces in different scale and poses, and is typical in everyday life. The images are of the same resolution, 2912×2184. Note that this resolution is common in nowadays digital collections. Experiments on larger resolution also reveal the speed advantage of our GPU-based face detector better.

In the following sections, unless otherwise specified, all running time reported is based on the average time measured from processing the images from the test set. Different input image sizes were also tested. These were obtained by dividing the resolution by a factor $f^2, f \in \{1.0, 1.1, 1.2, 1.3, 1.4\}$.

5.2 Results and Discussion

During detection, a scan window at every pixel location is processed in the image scale pyramid in our GPU-based detector. To facilitate performance comparison, we also run the CPU detection in the same manner.

A detection sub-window that goes through the feature evaluations of all stages is

Stage of Cascade	1	2	3	4	17	18	19	20
number of features	6	18	28	38	182	194	180	218
rejected subwindows (use CPU, %)	30.7	30.6	20.3	9.4	0.01	0.002	0.001	0.0008
difference in GPU classifications (%)	0.32	0.41	0.15	0.07	-	-	-	-

Table 5.1: The classifiers cascade and rejected subwindows in each stage.

classified as positive. The percentage of such positive sub-windows over an input image is very small. A majority of them are rejected in the early stages. Table 5.1 presents the number of features for the first and last 4 stages of the cascade, and the average percentages of rejected subwindows in each stage. From the table it is obvious that the most computational intensive part are at the first few stages.

5.2.1 Preliminary Results

In terms of accuracy, our GPU-based detector must perform more or less identical to the CPU version. Because of GPU's compliance to different floating point arithmetics standard, the results might differ to a certain extend. But we have to ensure that difference is negligible, especially in the final detection results.

We measured the accuracy of our GPU-based detector in terms of the differences from the CPU version, in the classification results, stage-by-stage. The third row of table 5.1 shows the average percentage of sub-windows in an image that are classified differently in the CPU version and GPU-based version. The percentage difference in stage i is calculated as $p_i = (\mid n_i^{GPU} - n_i^{CPU} \mid)/t \times 100\%$, where n is the number of negative subwindows in each stage, and t is the total number of subwindows evaluated. From the results, it is obvious that a negligible percentages of such difference are found in each stage. More importantly, measured over our test images set, there is no difference in the final detection results, i.e. all the detections on CPU version and our GPU-based version are identical. This sufficiently proves the correctness of our GPU-based detector.

Next, as each GPU-pass involves overheads in setting up the appropriate classifier data as well as the rendering pipeline, GPU is perhaps inefficient for classifications on the top-levels of the image pyramids. Rather it might be helpful to define a threshold in image size so that the levels with smaller scales will map entirely to CPU instead. Thus we experimented such thresholds to measure the total running time. The measurements are based on running 4 stages on GPU and detecting single-image at a time. Figure 5.1 shows the graph of detection time on both GPU cards with respect to different thresholds. The graph shows that with large thresholds, the detection speeds are slower; however, as threshold becomes smaller than 100×100, the performance difference is negligible. This indicates that the GPU object detector is superior with larger image sizes, and even with very small image size, GPU processing does not cause noticeable overheads. Thus, for all following experiments, we applied full GPU-based detector, i.e. all image scales in the pyramid are processed in GPU.

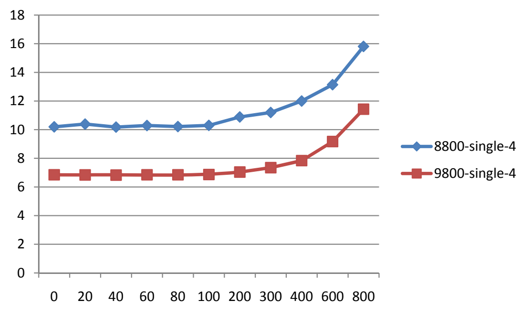

Figure 5.1: Detection time (sec) VS. image size threshold: smaller image not mapped to GPU; Series 9800-single-4 is the running time on GF9800GTX+.

5.2.2 Single-image GPU based detector

In Chapter 4 algorithms 2 and 3 depict the implementation of GPU-based detector that detects single image at a time. For this detector, we uses the *if...else* construct in the

Cg language, which provides the most efficient branch handling in GPU's SIMD based fragment processors.

Firstly, experiments are performed to find out the number of stages running in GPU that gives the best overall detection time. As we know GPU is more computationally powerful than CPU, thus it should run much faster to evaluate more stages on GPU. However, as a great number of subwindows are rejected in the early stages, the later stages which are more complex are only needed to be evaluated in far less positions. Mapping those stages to GPU may cause lots of unnecessary processing on GPU, thus hinders the performance.

Series 9800-single and 8800-single in figure 5.2 gives the running time on GF 9800GTX+ and GF 8800GTS respectively as a curves against the number of stages running in GPU. The curves essentially depict the performance tradeoff and show the best number of stages to map to GPU. The experiments were run on input images size of ($(2912 \times 2184)/f^2, f = 1.1$).

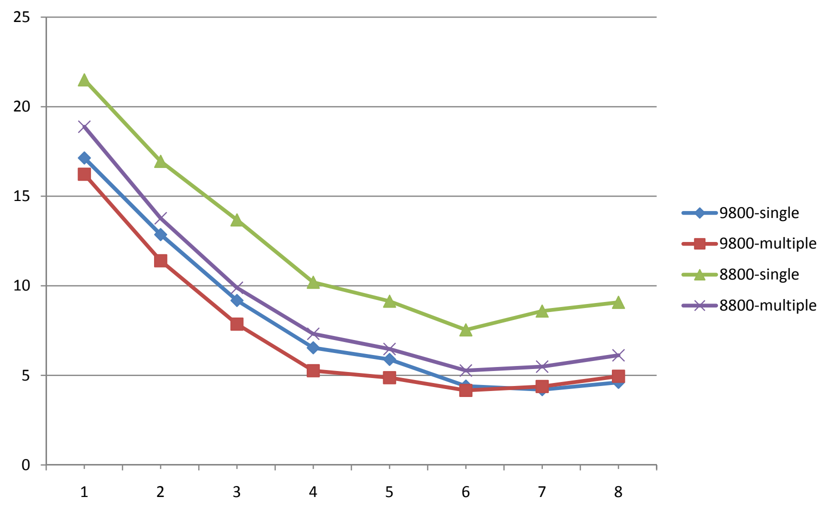

Figure 5.2: Detection time with different number of stages running on GPU.

From the series, we can see that for single-image based GPU detection, 5 to 7 stages usually provide the best overall running times. On GF 8800GTS, stage 6 gives the best time 7.54 seconds. On GF 9800GTX+, stage 7 gives the best running time 4.2 seconds. After that stage, the detection time rises.

This is as expected if we consider the number of rejected sub-windows in different stages in table 5.1. After 7 or 8 stages processing, most of the sub-windows has been rejected so that each remaining stage will reject only very small portion of them. And adding each additional stage to GPU means adding more complex computation on GPU, which is unnecessary on most fragments to be processed. Thus it is safe to predict that, for single-image GPU based detector, mapping 6 or 7 stages to GPU provides the best speedup.

For different input image sizes, GPU may give different speedup compared to the CPU version. Experiments are run to find out its performance in different input image size. Series 9800-single-7 and 880-single-6 in figure 5.3 give the running time on GF 9800GTX+ and GF 8800GTS respectively for images with resolution ($(2912 \times 2184)/f^2, f \in \{1, 1.1, 1.2, 1.3, 1.4\}$).

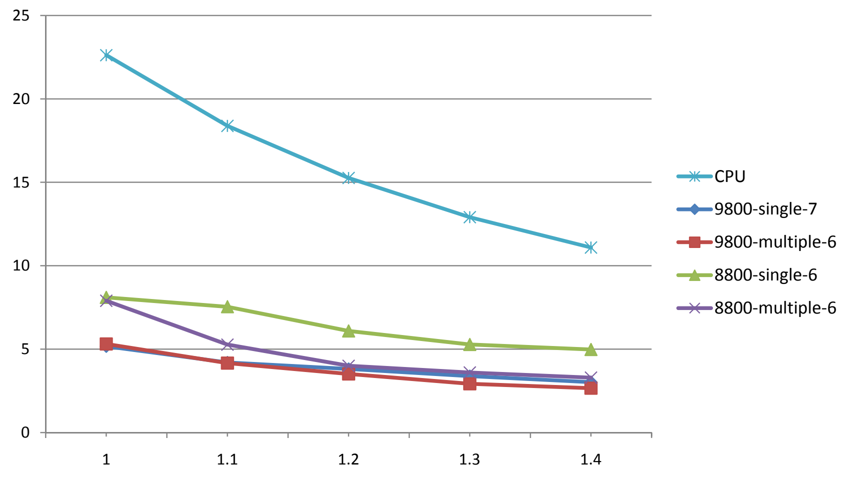

Figure 5.3: GPU Detection time for different input image sizes: time(seconds) as a function of f.

On resolution of ((2912×2184)), GF 9800GTS (5.17 seconds) provides about 4.38 times speedup over CPU(22.62 seconds). As from the speedup ratios presented in figure

5.4 and in table 5.2, larger input image size normally leads to greater speedup. This proves the advantage of GPU architecture in handling data intensive computations.

5.2.3 Multiple-image GPU based detector

Algorithms 4 and 5 in chapter 4 describe the GPU-based detector processing multiple images at a time, by using the *RGBA* channels of the render targets. As each image are processed in a single channel of the fragments, multiple images will possibly taken different branches. Thus the only way is to evaluate both branches and only write the correct result at the final step. This is essentially *"predication"* in branch handling, and is not very efficient. However due to reasons we already presented in chapter 4, we expect there will still be benefits.

First of all, experiments are designed to evaluate the detection time of running different number of stages on GPU, taking as input 4 random images (of the same resolution) from the test images set. The average running time is obtained by divide the total time by 4, and is compared to the single-image GPU based detector. The series 9800-multiple and 8800-multiple in figure 5.2 show the average running time against different number of stages mapping to GPU. Similar to single-image versions, running 5 to 7 stages in GPU gives the best speedup. For GF 9800GTX+, the best running time is 4.16 seconds when running 6 stages; while for GF 8800GTS, the best running time is 5.27 seconds also running 6 stages.

Notice that when mapping more than 6 stages to GPU, multiple-image based versions has a steeper increase in detection time than single-image versions. Particularly for GF 9800GTX+, when mapping 7 or 8 stages, it needs even more time than its single-image counter part.

For different input image sizes, the speedups of the multi-image GPU based detector may also be different. Series 9800-multiple-6 and 8800-multiple-6 in figure 5.3 give the running time on GF 9800GTX+ and GF 8800GTS for different image sizes, by running 6 stages of the cascade in GPU. Series CPU gives the corresponding running time of CPU-based version. Figure 5.4 shows the speedup ratios of different GPU based detectors.

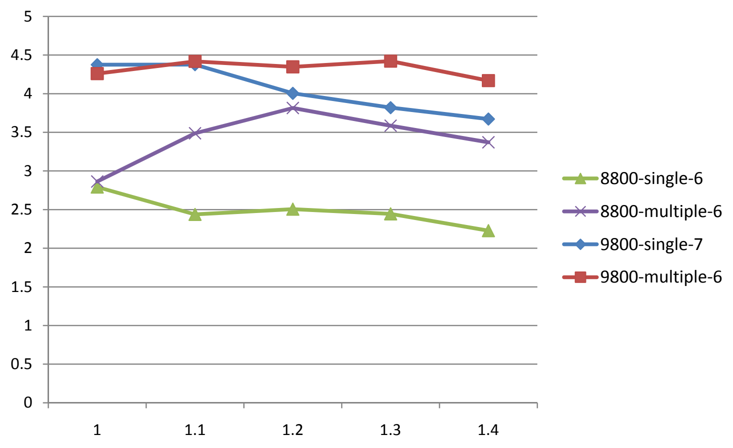

Figure 5.4: Speedup ratios for different GPU based detectors on different input image sizes.

From the figure, there are some interesting observations. First of all, multiple-image based detectors normally provide more speedup. However, not all larger resolution leads to better speedup, particularly for GF 8800GTS, at $f = 1$, the speedup is negligible. This is perhaps due to its poor memory bandwidth support. As multiple-image detection introduced a lot more than 4 times texture lookups, memory bottleneck hinders most of the performance.

The best overall speedup over CPU is 4.42 times, when running 6 stages in GF 9800GTX+ for the multiple-image based detector for the resolution of ($(2912\times2184)/f^2, f = 1.1$).

5.2.4 GF 9800GTX+ VS. GF 8800GTS

From previous descriptions we know that GF 9800GTX+ provides only slightly better memory bandwidth than GF 8800GTS, while sustain about 2 times more computational

Input resolution (/f)	1	1.1	1.2	1.3	1.4
9800-single-7	4.37	4.38	4.0	3.82	3.67
9800-multiple-6	4.3	4.42	4.41	4.35	4.16
8800-single-6	2.8	2.43	2.50	2.44	2.22
8800-multiple-6	2.86	3.48	3.81	3.58	3.37

Table 5.2: The speedup ratio of GPU-based detectors with different image sizes.

capacity (measured in GFLOP). In terms of their performance on our single- or multiple-image based detectors, there are also interesting observations revealing underlying principles. In figure 5.5, series 88s/98s and 88m/98m depict speedup of GF 9800GTX+ over

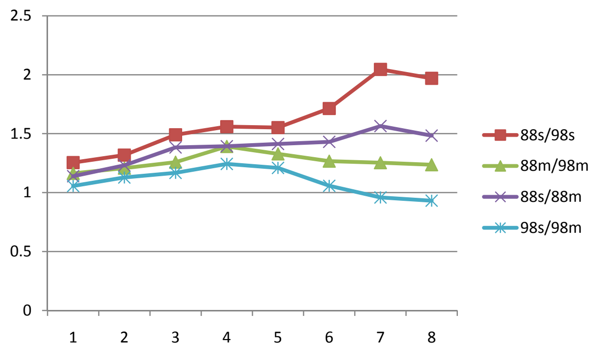

Figure 5.5: Comparison on GF 9800GTX+ VS. GF 8800GTS: mapping different number of stages

GF 8800GTS on single-image and multiple-image based detectors respectively; series 98s/98m and 88s/88m, on the other hand, depict speedup of the different GPUs' own running of multiple-image and single-image based detectors. Through analysis of the series, we can see the following.

1. There are a lot more speedup of 9800GTX+ over 8800GTS on single-image based detector than on multiple-image based detectors.

2. On 9800GTX+, the speedup of multiple-image over single-image based detector is

	Data preprocess	GPU compute	download	CPU compute	total time
8800-single-6	1.02	4.55	0.6	1.37	7.54
8800-multiple-6	1	2.34	0.63	1.3	5.27
9800-single-7	0.98	1.36	0.59	1.27	4.2
9800-multiple-6	0.96	1.31	4.41	0.6	4.16

Table 5.3: A detailed breakdown of the running time.

much less than the corresponding speedup on the 8800GTS.

These two facts are correlated. As we know that GPUs are evolving in the direction of becoming more computationally powerful, yet its memory speeds are improved in a much slower fashion. So as there are more memory operations on the multiple-image based detector, 9800GTX+ inherently improved more on single-image based detector. This also shows that 9800GTX+ is more efficient in branch handling than its predecessor.

5.2.5 A detailed breakdown

The overall detection time of our GPU-based detector consists of four components, data structure preprocessing time (including upload time to GPU), GPU computation time, GPU results downloading time, and CPU processing time. A detailed breakdown of the time spent on each components gives more insights on our GPU-based detector architecture better.

For this purpose, experiments were performed to find out the average processing time of each components, both for the single-image and multi-image based version. The input image size is of ($(2912 \times 2184)/f^2, f = 1.1)$). Table 5.3 presents the results. The multi-image based result is the total time divided by four.

Figure 5.6 presents the results in more intuitive way. From the figure, we know that the data preprocessing time take quite a big portion, as it is needed in every round. The portion of CPU computation in multi-image based detector is larger, simply because GPU takes less time in processing each image. For GF 9800GTX+, the GPU compute time for multi-image and single-image based detector is almost the same. This again illustrate the points above.

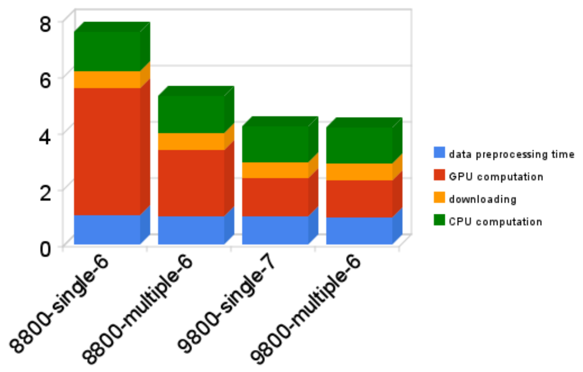

Figure 5.6: GPU detection time detailed breakdown

Chapter 6

Conclusions and Future Work

The importance of object detection as fundamental step towards semantic multimedia content understanding can not be overly emphasized. It is conceivable application in internet today and tomorrow also strive for the need to have a specialized and efficient object detection architectures. While today's applications are mostly based on CPU, this thesis explores the possibility to use commodity graphics processor's architecture to do the task.

Firstly, the boosting approach is identified as an promising and widely applicable object detection techniques. Detailed analysis are then given to identify the computational bottleneck in CPU, i.e. the data intensive portion of the detection cascade. The architecture design that use GPU as data streaming coprocessor is properly formulated. Experiments were performed to identify the benefits and tradeoffs. Up to about 5 times speedup was obtained on the GPU-based detectors over the pure CPU-based versions.

As GPU is specially designed for graphics rendering task, there are a number of hardware characteristics that may help or hinder our performance. To this end, we also implemented the GPU-based detector that processes multiple images at a time, using GPU's native vector arithmetics and predication branching support. Our implementation shows that by using these, even greater speedup was obtained.

We did experiments on two GPU models that are of two generations: NVIDIA GF 9800GTX and GF 8800GTS. The continuing evolution of GPUs definitely has an impact on the performance of our single-image and multi-image based detectors. To summarize, the more recent model improves more substantially on the single-image than multi-image based detectors. This is because the later has more memory operations, for which GPUs evolves much slower.

Consequently, we predict that as GPUs evolves, single-image based detector which make use of more complex and flexible logic will be more suitable to implement.

While our work is specifically for mapping boosted cascade of simple features, the design principle is generally applicable. For those approaches making use of complex features like SIFT [HMS+07] and histogram of oriented gradient, there would be possibly be more benefits. This could be our possible future work.

GPU architectures evolve rapidly, both their computational capacity and programmability. GPU can be seen as early generation of commodity data parallel co-processors. As envisioned by Owens et al.[OLG+07], a ”right” high level programming model for aggressively multi-threaded parallel computation would emerge. One can thus predict, the implementation of object detection’s work would be more intuitive and promising in the future.

Bibliography

[BFH$^+$04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware, 2004. submitted to ACM Transactions on Graphics, 2004.

[Bra08] Gary R Bradski. The open source opencv library, 2008. Electronic site, The sourceforge OpenCV project.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1*, pages 886–893, Washington, DC, USA, 2005. IEEE Computer Society.

[FCS$^+$07] Jimin Feng, Samarjit Chakraborty, Bertil Schmidt, Weiguo Liu, and Unmesh D. Bordoloi. Fast schedulability analysis using commodity graphics hardware. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 400–408, Washington, DC, USA, 2007. IEEE Computer Society.

[FM05] James Fung and Steve Mann. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international* *conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM.

[HB05] Mark Harris and Ian Buck. Gpu flow-control idioms. In Matt Pharr, editor, *GPU Gems 2*, pages 547–555. Addison-Wesley, March 2005.

[HMS$^+$07] S. Heymann, K. Muller, A. Smolic, B. Frohlich, and T. Wiegand. Sift implementation and optimization for general-purpose gpu. 2007.

[Lap06] I. Laptev. Improvements of object detection using boosted histograms. page III:949, 2006.

[LKO05] Aaron Lefohn, Joe M. Kniss, and John D. Owens. Implementing efficient parallel data structures on gpus. In Matt Pharr, editor, *GPU Gems 2*, pages 521–545. Addison-Wesley, March 2005.

[LKP02] Rainer Lienhart, Alexander Kuranov, and Vadim Pisarevsky. Empirical analysis of detection cascades of boosted classifiers for rapid object detection. Technical report, Microprocessor Research Lab, Intel Labs, December 2002.

[LKP06] Patrick Labatut, Renaud Keriven, and Jean-Philippe Pons. Fast level set multi-view stereo on graphics hardware. In *3DPVT '06: Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, pages 774–781, Washington, DC, USA, 2006. IEEE Computer Society.

[LM02] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *IEEE ICIP 2002*, pages 900–903, 2002.

[Low99] David G. Lowe. Object recognition from local scale-invariant features. *Computer Vision, IEEE International Conference on*, 2:1150, 1999.

[MGA03] W. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a c-like language, 2003.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.

[OLG$^+$07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[OMB05] Julius Ohmer, Frederic Maire, and Ross Brown. Implementation of kernel methods on the gpu. In *DICTA '05: Proceedings of the Digital Image Computing on Techniques and Applications*, page 78, Washington, DC, USA, 2005. IEEE Computer Society.

[OPFA06] Andreas Opelt, Axel Pinz, Michael Fussenegger, and Peter Auer. Generic object recognition with boosting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(3):416–431, 2006.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[pFHGA06] Jean philippe Farrugia, Patrick Horain, Erwan Guehenneux, and Yannick Alusse. Gpucv: A framework for image processing acceleration with graphics processors. *Multimedia and Expo, IEEE International Conference on*, 0:585–588, 2006.

[Pur04] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, 2004. Adviser-Patrick M. Hanrahan.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[VC08] Chief Internet Evangelist Vint Cerf. The next internet, 2008. Electronic document, the official Google blog. Date of publication: September 25, 2008.

[VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 1:511, 2001.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[YPL04] Ruigang Yang, Marc Pollefeys, and Sifang Li. Improved real-time stereo on commodity graphics hardware. *Computer Vision and Pattern Recognition Workshop*, 3:36, 2004.

[ZN08] L. Zhang and R. Nevatia. Efficient scan-window based object detection using gpgpu. pages 1–7, 2008.

[ZYCA06] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. *Com-*

puter Vision and Pattern Recognition, IEEE Computer Society Conference on, 2:1491–1498, 2006.