ABSTRACT

Title of dissertation: ALGORITHMIC ISSUES IN VISUAL OBJECT RECOGNITION

Mohamed Hussein, Doctor of Philosophy, 2009

Dissertation directed by: Professor Larry Davis
Department of Computer Science

This thesis is divided into two parts covering two aspects of research in the area of visual object recognition.

Part I is about human detection in still images. Human detection is a challenging computer vision task due to the wide variability in human visual appearances and body poses. In this part, we present several enhancements to human detection algorithms. First, we present an extension to the integral images framework to allow for constant time computation of non-uniformly weighted summations over rectangular regions using a bundle of integral images. Such computational element is commonly used in constructing gradient-based feature descriptors, which are the most successful in shape-based human detection. Second, we introduce deformable features as an alternative to the conventional static features used in classifiers based on boosted ensembles. Deformable features can enhance the accuracy of human detection by adapting to pose changes that can be described as translations of body features. Third, we present a comprehensive evaluation framework for cascade-based human detectors. The presented framework facilitates comparison between cascade-

based detection algorithms, provides a confidence measure for result, and deploys a practical evaluation scenario.

Part II explores the possibilities of enhancing the speed of core algorithms used in visual object recognition using the computing capabilities of Graphics Processing Units (GPUs). First, we present an implementation of Graph Cut on GPUs, which achieves up to 4x speedup against compared to a CPU implementation. The Graph Cut algorithm has many applications related to visual object recognition such as segmentation and 3D point matching. Second, we present an efficient sparse approximation of kernel matrices for GPUs that can significantly speed up kernel based learning algorithms, which are widely used in object detection and recognition. We present an implementation of the Affinity Propagation clustering algorithm based on this representation, which is about 6 times faster than another GPU implementation based on a conventional sparse matrix representation.

ALGORITHMIC ISSUES IN VISUAL OBJECT RECOGNITION

by

Mohamed Hussein

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Advisory Committee:
Professor Larry Davis, Chair/Advisor
Professor Amitabh Varshney,
Professor Ramani Duraiswami
Professor Min Wu
Dr Wael Abd-Almageed
Dr Fatih Porikli

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First, I would like to thank my advisor, Professor Larry Davis for all what he has done for me. I thank him for giving me the opportunity to work in Computer Vision research despite my atypical background. I thank him for his patience and support throughout the whole process. I thank him for being always available whenever I needed his help, despite his busy schedule. I thank him for giving me the freedom to chose my research topics. I thank him for his thoughtful comments and insights on my work, which have significantly enriched my way of thinking. I thank him for all the other favors which are too many to enumerate here. It has been a great honor for me to work with Professor Larry Davis. I wish him all the best.

I am also in debt for Dr. Wael Abd-Almageed for his tremendous support and encouragement. He has been an honest mentor and a true friend who could meticulously hit the balance between the professional relationship and the sincerity of friendship. I also owe Dr. Fatih Porikli a lot for his extraordinary support, encouragement, and devotion during my internship at MERL. His continuous smile always gave me hope and tranquility. I would like also to thank Prof Amitabh Varshney for his care and support. His comments and enthusiasm left a lasting effect on my professional attitudes. Finally, I would like to thank Prof Ramani Duraiswami for his care and enthusiasm in following my status. I also thank him for

introducing me to the GPU Scientific Computing group meetings, in which I met with a number of brilliant people from whom I learned valuable lessons.

I would like also to thank Prof Min Wu for devoting a part of her valuable time to serve as the dean's representative in my committee.

I would like to thank my wife, parents, and parents in law for their love, support, and continuous prayers for me. I owe them a significant part of my success.

Last, but not least, I would like to thank my God for blessing me with all these wonderful people, and many others, and blessing me with the patience and perseverance needed to reach this point.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALMR	Average Log Miss Rate
AP	Affinity Propagation
API	Application Programming Interface
BAG	Band Approximation of Gram matrices
BFS	Breadth First Search
BK	Boykov-Kolmogrov's Graph Cut algorithm
COO	Coordinate sparse matrix representation
COV	Covariance descriptor-based human detector
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
CUDPP	CUDA Parallel Primitives
DET	Detection Error Tradeoff
GPU	Graphics Processing Unit
HOG	Histograms of Oriented Gradients
INRIA	Institut National De Recherche En Informatique Et En Automatique
KII	Kernel Integral Images
LRF	Local Receptive Fields
MERL	Mitsubishi Electric Research Labs
ML	Maximum Likelihood
MRF	Markov Random Field
NIR	Near Infrared
OS	Operating System
PCA	Principal Component Analysis
PR	Precision-Recall
RAM	Random Access Memory
RBF	Radial Basis Functions
RI	Repeated Integration
ROC	Receiver Operating Characteristics
RCM	Reverse Cuthill-McKee
SFC	Space Filling Curves
SIFT	Scale Invariant Feature Transform
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SMO	Sequential Minimal Optimization
SMP	Symmetric Multi-Processing
SP	Streaming Processor
SURF	Seeped Up Robust Features
SVM	Support Vector Machine

Part I

Human Detection

Chapter 1

Introduction to Part I: Human Detection

The problem of visual object detection is more specific than the general problem of visual object recognition. Object detection involves determining the location of the object in the image, not just whether it exits or not. Therefore, an object detector can work as a recognizer. But, the opposite is not generally true.

Despite being just an instance of the general object detection problem, human detection has received a special attention in the computer vision community. With no doubt, humans are more difficult to detect than many other objects due to their huge range of appearance variations. This makes the problem very challenging, and hence interesting. It is reasonable to believe that if a satisfactory solution found for humans, solutions for other objects would become much more tangible. Another reason to justify the effort devoted to this problem is its many useful applications, such as intelligent vehicles, video surveillance, and human-robot interaction.

In this part, we present techniques to enhance the speed and accuracy of human detection. We also present an evaluation framework for cascade-based human detectors that enhances over traditional evaluation methods in terms of reliability and clarity of comparisons. Each of these topics is briefly introduced in this chapter in Sections 1.1, 1.2, and 1.3, and then detailed in Chapters 2, 3, and 4, respectively.

1.1 Fast Construction of Feature Descriptors

A significant advancement in human detection was recently made through the discovery of the power of Histograms of Oriented Gradients (HOG) descriptors in human/non-human classification [20]. Nevertheless, real time human detection is still an unsolved problem. Construction of HOG descriptors involve computing weighted summations over rectangular regions of the image's gradient map, which is a computationally expensive process. The integral images framework [98] makes it possible to compute uniformly weighted summations in constant time. However, using non-uniformly weighted summations, *e.g.* bi-linear interpolation and Gaussian weighting, in constructing HOG descriptors was shown to improve detection results [20]. In Chapter 2, we introduce kernel integral images, which is an extension to the integral images framework that allows for constant time computation of non-uniformly weighted summations. We present two examples of using this framework: one is computing summations with bi-linear interpolation between neighboring cells, and the other is an approximation to computing summations with Gaussian weighting. The two forms of summations are commonly used in constructing region descriptors and appearance models in object recognition, detection, and tracking. The kernel integral images framework allows for robust construction of these representations without sacrificing fast computational time. A version of this work appeared in our paper [48].

1.2 Boosted Deformable Features

A promising class of classifiers that combines high accuracy with high classification speed is based on building cascades of boosted features [98, 103, 109, 94, 81, 102]. A cascade classifier gains its discrimination power from its ability to incorporate a very large number of negative examples in the training phase. It achieves high classification speed by effectively excluding easy negative samples from complex processing. Each layer of a cascade classifier is typically trained using boosting [36], where a boosting algorithm selects an ensemble of object features to form a powerful discriminative model. In this context, features are defined as subregions with *fixed* relative locations and extents with respect to the object's image window. In Chapter 3, we introduce using deformable features with boosted ensembles. A deformable feature adapts its location depending on the visual evidence in order to match the corresponding physical feature. Therefore, deformable features can better handle deformable objects. We empirically show that boosted ensembles of deformable features perform significantly better than boosted ensembles of fixed features for human detection. A version of this work appeared in our paper [50].

1.3 A Comprehensive Evaluation Framework

Despite recent advancement in the area of human detection in images, little effort has been devoted to evaluation methodologies. In Chapter 4, we introduce a framework for evaluating human detectors that considers the practical application of a detector on a full image using multi-size sliding window scanning. Plots for cascade

classifiers are generated based on confidence scores instead of varying the number of layers, which makes plots cover the same range of false alarm rates, and hence makes comparison between methods more meaningful. To assess a method's overall performance on a given test and compare different methods, we introduce an aggregate performance score that facilitates such analysis. To analyze the significance of the obtained results, we conduct 10-fold cross validation experiments. We applied our evaluation framework to two state of the art cascade-based detectors on the standard INRIA-Person dataset, as well as another dataset of near infrared images provided by Mitsubishi Electric Research Labs (MERL). Our experiments show the utility of the presented framework and leads to some interesting conclusions. A version of this work appeared in our paper [49].

Chapter 2

Kernel Integral Images

Integral images are commonly used in computer vision and computer graphics applications. Evaluation of box filters via integral images can be performed in constant time, regardless of the filter size. Although Heckbert [42] extended the integral images approach for more complex filters, its usage has been very limited, in practice. In this chapter, we present an extension to integral images that allows for application of a wide class of non-uniform filters. Our approach is superior to Heckbert's in terms of precision requirements and suitability for parallelization. We explain the theoretical basis of the approach and instantiate two concrete examples: filtering with bilinear interpolation, and filtering with approximated Gaussian weighting. Our experiments show the significant speedups we achieve, and the higher accuracy of our approach compared to Heckbert's.

2.1 Introduction

Filtering is a fundamental image processing operation. The computational complexity of image filtering depends on the complexity and size of the filter. For separable filters, for example, efficient computation is possible by applying two consecutive one-dimensional filters instead of the original two-dimensional filter. However, even when taking advantage of the filter's separability, the computational time increases

with the filter's size, which is unfavorable for large filters. In some cases, we do not even know the filter size in advance, *e.g.* when the filter size is determined dynamically based on feature values. In such cases, the separability of the filter does not help. For box filters, which are used to compute averages and summations over rectangular image regions, there is an elegant technique that can overcome these difficulties. Given an integral of image features (Figure 2.1), filtering with a box filter at any point can be performed in constant time regardless of the filter size. Unfortunately, using pre-computed integrals is limited, in practice, to box filters. In this chapter, we present a novel extension that makes pre-computed integrals usable for more complex filters.

The idea of using pre-computed integrals was first introduced, with the name *summed-area tables*, by Crow [18] to be used for texture mapping in computer graphics. Recently, it was popularized in the field of computer vision, with the name *integral images*, by Viola and Johns [98], who used it for fast computation of Haar wavelet features. Later on, integral images were generalized by Porikli [74] to *integral histograms*, which allow for fast construction of feature histograms. More recently, integral images and integral histograms were used to speed construction of Histograms of Oriented Gradient descriptors by Zhu *et al.* [109], Region Covariance descriptors by Tuzel *et al.* [93], and the SURF descriptors by Bay *et al.* [5].

To the best of our knowledge, usage of integral images in computer vision applications has been limited to the special case of box filtering although some of these applications can perform better when using non-uniform filters. For example, Dalal and Triggs [20] use bilinear interpolation between neighboring cells and Gaus-

sian weighting of pixels within a block of pixels in constructing their histograms of oriented gradients features for human detection. They show how these weighting schemes enhance the detector's accuracy. To develop a fast version of Dalal and Triggs' detector, Zhu *et al.* [109] sacrifice the benefits of these weighting schemes to enable usage of integral images. Another example is in the work of Bay *et al.* [5], where Gaussian derivative filters are approximated by box filters so that integral images can be used. Perhaps, a better approximation would be possible if integral images were able to handle non-uniform weighting filters. A third example is in building appearance models for tracking, where pixels closer to the center of the tracked region are given higher weights than pixels closer to the borders, *e.g.* Elgammal *et al.* [28]. Consider a particle filter tracker, *e.g.* Zhou *et al.* [108], where appearance models for hundreds of overlapping regions need to be constructed, possibly for many tracked targets, on every frame. Applying non-uniform weighting of pixels in such a situation without the aid of a fast technique similar to integral images can be impractical for real-time application.

Heckbert [42] introduced the theoretical foundation of the summed-area tables (integral images) technique and extended the theory to allow for more complex filters. However, his extension required a very high precession numerical representation even for moderate image sizes [44]. Similar to Heckbert, we present an approach to extend the integral images technique to allow for non-uniform filters. However, our approach has lower precision requirement than Heckbert's and is more suitable for parallel implementation. We call our approach *kernel integral images*. A kernel integral image is a group of integral images such that a linear combination of box fil-

ters applied to them is equivalent to applying a more complex filter. We instantiate two examples of applying our approach that are relevant to computer vision applications: feature filtering with bilinear interpolation, and approximation of filtering with Gaussian weighting. Our experimental analysis shows the significant speedups we achieve, and the superiority of our approach to Heckbert's in terms of accuracy.

The rest of the chapter is organized as follows: Section 2.2 introduces notation and explains integral images in an abstract form. Section 2.3 employs filtering with bilinear interpolation as an example to introduce our extension, which is afterwards formalized in Section 2.4. Then, the example of filtering with approximate Gaussian weighting is described in Section 2.5. In Section 2.6, we compare our approach to Heckbert's. Empirical analysis of speedups and numerical errors are presented in Section 3.6, followed by conclusions and future work in Section 2.8.

For clarity of presentation, we focus on one and two dimensional signals. The extension to higher dimensions is straight forward.

2.2 Fast Filtering via Integral Images

2.2.1 Preliminaries

Let $f : \mathbf{x} \rightarrow \mathcal{R}$ be a function that maps a point $\mathbf{x} = (x_1, x_2)$ to a real value, where $0 \leq x_i \leq N_i, N_i > 0, i = 1, 2$. Therefore, the domain of f, $\mathbf{D}_f$, is a rectangle bounded by the lines $x_i = 0$ and $x_i = N_i, i = 1, 2$. A rectangular region (referred to as a region from now on) $\mathbf{R} \subseteq \mathbf{D}_f$ is defined by a pair of points $\mathbf{x}^b$ and $\mathbf{x}^e$ such that $\mathbf{x}^b, \mathbf{x}^e \in \mathbf{D}_f$, and $x_i^b < x_i^e, i = 1, 2$. The two points $\mathbf{x}^b$ and $\mathbf{x}^e$ represent the two

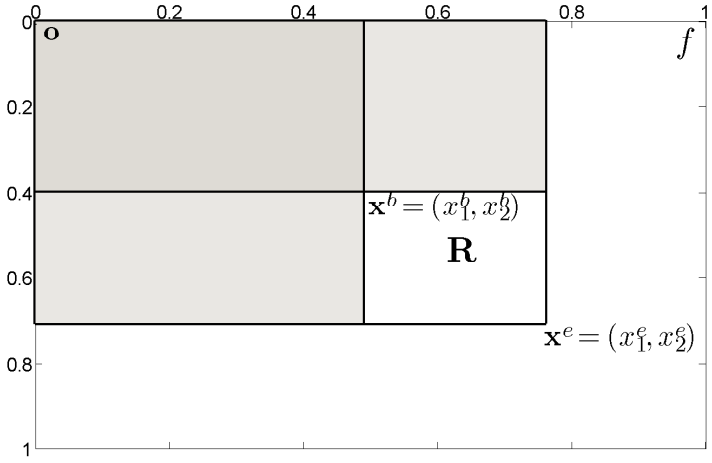


Figure 2.1: An integral of image features. The value of the integral at a point is the sum of the values of image features in the rectangular area from the origin to the point. The sum of feature values over any axis-aligned rectangular region (*e.g.* the small white rectangle) is determined by the value of the integral at the four corners of the region.

extreme points of the region $\mathbf{R}$. We refer to the ordered pair $\mathbf{r} = (\mathbf{x}^b, \mathbf{x}^e)$ as the region definition. Figure 2.1 illustrates some of these definitions. In practice, the function f represents the raw intensity value or some other feature at each point in an image. Its domain, $\mathbf{D}_f$, is the set of all pixel coordinates in the image. $N_1 \times N_2$ is the image size.

A filtering of the values of f over a region $\mathbf{R}$ can be defined as a function $\mathcal{A}_f : \mathbf{R} \rightarrow \mathcal{R}$ that maps the region to a real value. The form of the *filtering function* we consider can be expressed as

$$\mathcal{A}_f(\mathbf{R}) = \sum_{\mathbf{x} \in \mathbf{R}} a_f^{\mathbf{r}}(\mathbf{x}), \tag{2.1}$$

where the *contribution function* $a_f^{\mathbf{r}}(\mathbf{x})$ defines the contribution of the point $\mathbf{x}$ to the

filtering of the function f over the region $\mathbf{R}$. In general, as the superscript of $a_f^{\mathbf{r}}$ indicates, the contribution of a point $\mathbf{x}$ depends not only on the point coordinates and the function f, but also on the definition of the region, *i.e.* its two extreme points. In this section we first consider the simpler case, where the contribution of a point is *independent* of the region's definition . We handle the general case in sections 2.3 and 2.4. Thus, for now, we denote the contribution function by a_f instead of $a_f^{\mathbf{r}}$. Therefore, the filtering function is redefined as

$$\mathcal{A}_f(\mathbf{R}) = \sum_{\mathbf{x}\in\mathbf{R}} a_f(\mathbf{x}). \tag{2.2}$$

We call such a filtering function and its associated contribution function *region-independent* functions.

In its simplest form, the contribution function can be equal to the function f. That is

$$a_f(\mathbf{x}) = f(\mathbf{x}) \ . \tag{2.3}$$

But, in fact, we can use any function that can be evaluated independently from the filtering region's definition. For example, we can define the contribution function as

$$a_f(\mathbf{x}) = \|\mathbf{x}\| f^2(\mathbf{x}) \ . \tag{2.4}$$

Therefore, filtering with region-independent contributions is much more general than just summing feature values over a rectangular region.

2.2.2 Integral Images

When filtering is computed over many regions that overlap, using equation 2.2 is not efficient. This is because the computations performed in areas that are shared among more than one overlapping region will be repeated for each region. Luckily, the filtering equation has a sub structure that allows for a dynamic programming solution. This dynamic programming solution is what we refer to as integral images.

Define the *integral image* of a function f, I_f, as a function with the same domain and codomain as f, and of the form

$$I_f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbf{D}_f, y_i \leq x_i, i=1,2} a_f(\mathbf{y}). \tag{2.5}$$

The value of the integral image of a function f at a point $\mathbf{x}$ is the sum of the contributions of all points in the region defined by $(\mathbf{o}, \mathbf{x})$, where $\mathbf{o}$ is the origin or the coordinate system.

Given this formulation of integral images, it becomes much simpler to evaluate the filtering function over any region $\mathbf{R}$. A filtering function can be written in terms of an integral image as

$$\mathcal{A}_f(\mathbf{R}) = I_f(x_1^e, x_2^e) - I_f(x_1^b, x_2^e) - I_f(x_1^e, x_2^b) + I_f(x_1^b, x_2^b), \tag{2.6}$$

where $(\mathbf{x}^b, \mathbf{x}^e)$ defines the filtering region $\mathbf{R}$ (Figure 2.1).

In general, having the integral image, filtering over a region $\mathbf{R}$ is reduced to $O(1)$ computations compared to $O((x_1^e - x_1^b) \times (x_2^e - x_2^b))$ computations using the original filtering function formulation, equation 2.2. However, the cost of constructing the integral image itself is $O(N_1 \times N_2)$. Therefore, the utility of using integral

images is realized only when we filter over many overlapping regions. In the case of exhaustively filtering over the entire domain of regions, the speedups obtained when using integral images were reported in Porikli [74] to be several orders of magnitude for a broad range of parameter choices.

2.3 Extending Integral Images for Filtering with Region-Dependent Contributions

Before discussing the formal treatment of the general case, where the contribution functions are dependent on the filtering region's definition, we start with a concrete example. Consider filtering with bilinear interpolation. A practical example is constructing the SIFT descriptor [61], where filtering is performed over adjacent regions in a 4×4 grid of cells of pixels, such that each pixel contributes to more than one cell via bilinear interpolation.

We want to define the contribution function in this case. A region $\mathbf{R}$ is defined by $\mathbf{r} = (\mathbf{x}^b, \mathbf{x}^e)$, where $\mathbf{x}^b = (x_1^b, x_2^b)$ and $\mathbf{x}^e = (x_1^e, x_2^e)$. The center of the region is $\mathbf{x}^c = (x_1^c, x_2^c) = (\mathbf{x}^b + \mathbf{x}^e)/2$, half the width of the region is $hw = (x_1^e - x_1^b)/2$, and half the height of the region is $hh = (x_2^e - x_2^b)/2$. The contribution function at a point $\mathbf{x} = (x_1, x_2) \in \mathbf{R}$ is defined as

$$a_f^{\mathbf{r}}(\mathbf{x}) = \left(\frac{hw - |x_1 - x_1^c|}{hw}\right)\left(\frac{hh - |x_2 - x_2^c|}{hh}\right) f(\mathbf{x}). \tag{2.7}$$

Apparently, the contribution of a point is region-dependent. Hence, the simple integral image approach presented in Section 2.2 is not directly applicable here.

For simplicity of presentation, we consider only the case when $x_1 \geq x_1^c$ and

$x_2 \geq x_2^c$. The other cases can be handled similarly. By manipulating equation 2.7, we obtain

$$
\begin{aligned}
a_f^{\mathbf{r}}(\mathbf{x}) &= \left(\frac{hw - x_1 + x_1^c}{hw}\right) \times \\
&\quad \left(\frac{hh - x_2 + x_2^c}{hh}\right) f(\mathbf{x}) && (2.8)\\
&= \left(\frac{x_1^e - x_1}{hw}\right)\left(\frac{x_2^e - x_2}{hh}\right) f(\mathbf{x}) && (2.9)\\
&= \left(\frac{x_1^e x_2^e}{hw \times hh}\right) f(\mathbf{x}) - \\
&\quad \left(\frac{x_1^e}{hw \times hh}\right)(x_2 f(\mathbf{x})) - \\
&\quad \left(\frac{x_2^e}{hw \times hh}\right)(x_1 f(\mathbf{x})) + \\
&\quad \left(\frac{1}{hw \times hh}\right)(x_1 x_2 f(\mathbf{x})) && (2.10)\\
&= g_1^{\mathbf{r}} h_{1f}(\mathbf{x}) + g_2^{\mathbf{r}} h_{2f}(\mathbf{x}) + \\
&\quad g_3^{\mathbf{r}} h_{3f}(\mathbf{x}) + g_4^{\mathbf{r}} h_{4f}(\mathbf{x}), && (2.11)
\end{aligned}
$$

where $h_{1f} = f(\mathbf{x})$, $h_{2f} = x_2 f(\mathbf{x})$, $h_{3f} = x_1 f(\mathbf{x})$, $h_{4f} = x_1 x_2 f(\mathbf{x})$, and $g_1^{\mathbf{r}}$ through $g_4^{\mathbf{r}}$ are the corresponding coefficients from expression 2.10.

Now, we have expressed the original contribution function as a linear combination of simpler functions, h_{1f} through h_{4f}, with weighting coefficients $g_1^{\mathbf{r}}$ through $g_4^{\mathbf{r}}$. The interesting observation here is that all the h functions are region-independent, and none of the g coefficients depends on the point $\mathbf{x}$ or the function f, they only depend on the region's definition. We call functions such as the g coefficients *point-independent*. Substituting equation 2.11 into the filtering function, equation 2.1, yields

$$\begin{aligned}\mathcal{A}_f(\mathbf{r}) &= g_1^{\mathbf{r}} \sum_{\mathbf{x}\in\mathbf{R}} h_{1f}(\mathbf{x}) + g_2^{\mathbf{r}} \sum_{\mathbf{x}\in\mathbf{R}} h_{2f}(\mathbf{x}) + \\ & \quad g_3^{\mathbf{r}} \sum_{\mathbf{x}\in\mathbf{R}} h_{3f}(\mathbf{x}) + g_4^{\mathbf{r}} \sum_{\mathbf{x}\in\mathbf{R}} h_{4f}(\mathbf{x}) \ . \end{aligned} \tag{2.12}$$

Equation 2.12 expresses the original filtering function as a linear combination of other filtering functions. Moreover, all of the component filtering functions in this linear combination are region-independent. In fact, the linear combination obtained for the filtering function is exactly the same as the linear combination for the contribution function itself. Since each of the component filtering functions in equation 2.12 is region-independent, each can be computed efficiently using an integral image for its own contribution function. Then, by substituting the resulting values in equation 2.12, we obtain the desired filtering.

In summary, to use integral images in this example we express the desired region-dependent contribution function as a linear combination of several region-independent contribution functions. Then, the desired region-dependent filtering is easily computed as a linear combination of the corresponding region-independent filtering functions, which can be efficiently computed via integral images.

2.4 Kernel Integral Images

In this section, we treat the case of region-dependent filtering functions in a more formal way. Recall from the example of bilinear interpolation that the mechanism used to enable usage of integral images is expressing the filtering function as a linear combination of other region-independent filtering functions. To understand why this

works, we rewrite the final form of the contribution function, equation 2.11, in a more compact form as

$$a_f^{\mathbf{r}}(\mathbf{x}) \quad = \quad < \mathbf{g}^{\mathbf{r}}, \mathbf{h}_f(\mathbf{x}) >, \tag{2.13}$$

where

$$\mathbf{g}^{\mathbf{r}} \quad = \quad \begin{bmatrix} g_1^{\mathbf{r}} \\ g_2^{\mathbf{r}} \\ g_3^{\mathbf{r}} \\ g_4^{\mathbf{r}} \end{bmatrix}, \tag{2.14}$$

and

$$\mathbf{h}_f(\mathbf{x}) \quad = \quad \begin{bmatrix} h_{1f}(\mathbf{x}) \\ h_{2f}(\mathbf{x}) \\ h_{3f}(\mathbf{x}) \\ h_{4f}(\mathbf{x}) \end{bmatrix}, \tag{2.15}$$

In other words, we can express the contribution function as a dot product of two vector functions: one of them is region-independent and the other is point-independent. This is actually a necessary and sufficient condition to express the filtering function as a linear combination of region-independent filtering functions. We outline the proof of this fact rather informally here. The sufficiency direction is straight forward following the same argument as in the bilinear interpolation example. Basically, by distributing the summation of the filtering function over terms of the dot product, as we did to obtain equation 2.12, sufficiency immediately follows. The necessity direction is derived as follows. Starting from the linear

combination of filtering functions, as in equation 2.12, we can express the linear combination as a dot product. Then, by pulling the summation out, we obtain an expression of the contribution function that is a dot product of two parts, one of them is region-independent, and the other one is point-independent.

The dot product immediately reminds us of the kernel trick that is frequently used in machine learning, where feature vectors are implicitly transformed into a – typically – higher dimensional space by replacing a dot product by a kernel function that is equivalent to a dot product in the transformed space [16]. Since applying any transformation to the vectors $\mathbf{g^r}$ and $\mathbf{h}_f(\mathbf{x})$, in equation 2.13, will not change their region-independence or point-independence natures, the condition we stated above still holds on the transformed vectors. Therefore, we can generalize the form of the contribution functions we consider to

$$a_f^{\mathbf{r}}(\mathbf{x}) = \mathbf{H}(\mathbf{g^r}, \mathbf{h}_f(\mathbf{x})), \tag{2.16}$$

where H is a kernel function, *i.e.* a function that computes a dot product between its two arguments possibly after mapping them to another dimensional space. We call this generalization of integral images *kernel integral images*. In our case, even if the kernel performs a dot product implicitly, to compute our filtering function we have to perform it explicitly. Sometimes, the kernel computes the dot product in an infinite dimensional space. In these cases, approximation of the dot product with a small number of terms may be sufficient for the application in hand. This point will be clarified when we use it in an example in Section 2.5.

2.5 Filtering with Gaussian Weighting

In many applications of image feature filtering in computer vision, higher weights are given to pixels closer to the center of the filtering region and lower weights to pixels closer to the borders of the filtering region. That is applied, for example, in object tracking, *e.g.* Elgammal *et al.* [28], where higher weights are given to pixels that more likely belong to the object than the background. The same idea was shown to improve human detection performance in Dalal and Triggs [20]. In both cases, the weighting function used is a Gaussian weighting function.

To simplify the mathematical treatment, we consider the one dimensional case. Consider a region $\mathbf{R}$ defined by the two limiting points x^b and x^e. The center of $\mathbf{R}$ is defined as $x^c = (x^b + x^e)/2$. Denote the standard deviation of the Gaussian weighting function by $\sigma^{\mathbf{r}}$. The contribution function in this case can be defined as

$$a_f^{\mathbf{r}}(x) = e^{-\left(\frac{x-x^c}{\sigma^{\mathbf{r}}}\right)^2} f(x). \tag{2.17}$$

Clearly, the contribution function is region-dependent. Consider the Euler expansion of equation 2.17

$$a_f^{\mathbf{r}}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i (x - x^c)^{2i}}{\sigma^{\mathbf{r}\,2i} i!} f(x). \tag{2.18}$$

Equation 2.18 can be viewed as a dot product in an infinite dimensional space between two vector functions one of them is region-independent and the other one is point-independent. (To see this, consider expanding the expression $(x - x^c)^{2i}$ in each term of the power series.) Hence, the kernel integral image method applies. But, it requires computation of an infinite number of integrals. However, we can approximate the contribution function by taking a few of the initial terms of the

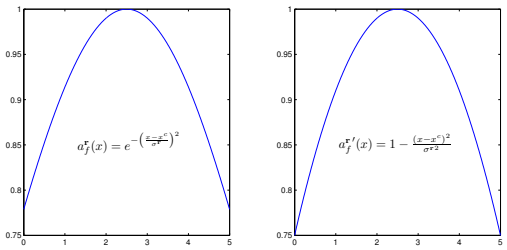


Figure 2.2: Comparison of the Gaussian weighting function and its approximation, equations 2.17 and 2.19, when the filtering region is between 0 and 5 and $\sigma^{\mathbf{r}}$ is 5.

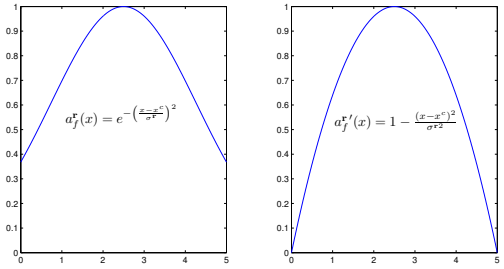


Figure 2.3: Comparison of the Gaussian weighting function and its approximation, equations 2.17 and 2.19, when the filtering region is between 0 and 5 and $\sigma^{\mathbf{r}}$ is 2.5.

expansion. For example, taking the first two terms only, we obtain the contribution function

$$a_f^{\mathbf{r}\prime}(x) = \left(1 - \frac{(x - x^c)^2}{\sigma^{\mathbf{r}2}}\right) f(x). \tag{2.19}$$

This approximation is valid, *i.e.* does not give negative weights, as long as $\sigma^{\mathbf{r}}$ is selected so that $\frac{(x-x^c)^2}{\sigma^{\mathbf{r}2}} \leq 1$. Figures 2.2 and 2.3 show plots of the original Gaussian weighting function, equation 2.17, and its approximation, equation 2.19, when $x^b =$

0, $x^e = 5$, and $\sigma^{\mathbf{r}} = 5$ and 2.5, respectively. In the case of $\sigma^{\mathbf{r}} = 5$ plots are very similar. However, for the case of $\sigma^{\mathbf{r}} = 2.5$, the difference is quite large. For applications that need weighting of pixels with respect to one another so that pixels closer to the center get more importance, the difference between the two functions – in case the selected value of $\sigma^{\mathbf{r}}$ makes a difference – is not expected to be important. In general, whether the approximation is accurate enough or not, and whether it is worth using more terms of the expansion to achieve higher accuracy or not, depends on the value of $\sigma^{\mathbf{r}}$ and on the application itself.

2.6 Kernel Integral Images vs. Repeated Integration

Heckbert [42] presented an elegant method, called *filtering by repeated integration*, to extend usage of pre-computed integrals to more complex filters. For completeness of presentation, we briefly compare our method to his method. For details, please refer to Heckbert [42].

Heckbert's approach is based on the fact that more complex filters can be constructed by convolving a box-filter with itself. For example, if we convolve a box filter with itself once, we obtain a triangular filter, which is very similar to filtering with bilinear interpolation in two dimensions. If we convolve a box filter with itself twice, we obtain a quadratic filter, which is similar to the approximation we use for Gaussian filters. In fact, convolution of a box filter with itself an infinite number of times produces the Gaussian filter. Suppose that we want to use a filter that is generated by convolving a box filter with itself n times. Heckbert's approach is

based on the fact that convolution with such a filter is equivalent to integrating the image n times and then convolving the n^{th} integral with the n^{th} derivative of the filter. The n^{th} derivative of such a filter turns out to be a simple sparse filter, which is very efficient to convolve with.

The main drawback of the repeated integration approach is integrating the image several times. The required precision to represent the integration values grow linearly with the number of integrations [44]. In our approach, we compute integrals of several functions. But, each is integrated only once. For example, in approximating a Gaussian filter by a quadratic filter, the repeated integration method requires integrating the image three times consecutively, while kernel integral images requires computing nine independent integrals. Experimentally, kernel integral images in this case produces smaller numerical errors using the standard double-precision floating point number representation, as we show in Section 2.7.3.

Another advantage of our approach is that the integrals computed are independent of one another. That allows for parallel computation of the integrals.

2.7 Experimental Results

2.7.1 Implementation Details

We evaluated our approach in terms of speedup by comparing to the conventional filtering approach (equation 2.1). We implemented filtering with bilinear interpolation, and filtering with approximate Gaussian weighting. Both are implemented in two dimensions.

For bilinear interpolation, equation 2.11 in Section 2.3 considers only the case where $x_1 \geq x_1^c$ and $x_2 \geq x_2^c$. If we consider the origin at the lower left corner of the filtering domain, then equation 2.11 considers only the case of the top right quadrant of the filtering region. Table 2.1 lists coefficients of different terms for the four quadrants.

Table 2.1: Coefficients of different contribution functions in the case of bilinear interpolation, equation 2.7, for the four region quadrants. All coefficients in the table have to be normalized by dividing by $hw \times hh$

	$f(\mathbf{x})$	$x_2 f(\mathbf{x})$	$x_1 f(\mathbf{x})$	$x_1 x_2 f(\mathbf{x})$
Top Right Quadrant	$x_1^e x_2^e$	$-x_1^e$	$-x_2^e$	1
Top Left Quadrant	$-x_1^b x_2^e$	x_1^b	x_2^e	-1
Lower Right Quadrant	$-x_1^e x_2^b$	x_1^e	x_2^b	-1
Lower Left Quadrant	$x_1^b x_2^b$	$-x_1^b$	$-x_2^b$	1

In order to perform fast filtering in this case, we compute four different integral images, one for each of the contribution functions. The integration itself is conducted in four steps, since each region's quadrant has a different coefficient for each of the integrals, as shown in Table 2.1.

For the case of approximating Gaussian weighting in two dimensions, by expanding equation 2.19 and extending the notation to two dimensions, we obtain

$$\begin{aligned} a_f^{\mathbf{r}\prime}(\mathbf{x}) &= \left[\left(1 - \frac{x_1^{c\,2}}{\sigma^{\mathbf{r}2}}\right) + \frac{2x_1^c}{\sigma^{\mathbf{r}2}} x_1 - \frac{1}{\sigma^{\mathbf{r}2}} x_1^2\right] \times \\ &\quad \left[\left(1 - \frac{x_2^{c\,2}}{\sigma^{\mathbf{r}2}}\right) + \frac{2x_2^c}{\sigma^{\mathbf{r}2}} x_2 - \frac{1}{\sigma^{\mathbf{r}2}} x_2^2\right] f(\mathbf{x}). \end{aligned} \tag{2.20}$$

Hence, to perform fast filtering, we compute nine integral images. These are

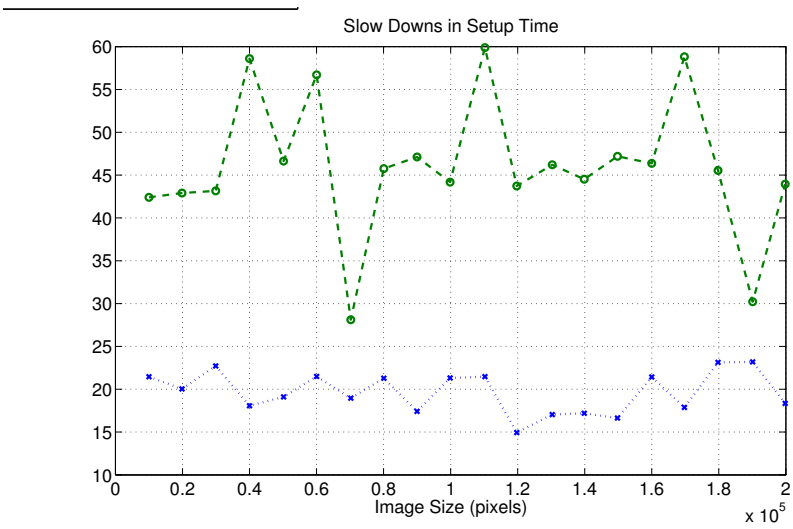


Figure 2.4: The slow down in setting up integrals vs the naive set up of conventional approaches.

integral images for the contribution functions: $f(\mathbf{x})$, $x_1 f(\mathbf{x})$, $x_2 f(\mathbf{x})$, $x_1 x_2 f(\mathbf{x})$, $x_1^2 f(\mathbf{x})$, $x_2^2 f(\mathbf{x})$, $x_1 x_2^2 f(\mathbf{x})$, $x_1^2 x_2 f(\mathbf{x})$, and $x_1^2 x_2^2 f(\mathbf{x})$. The coefficient of each region-independent filtering function can easily be obtained from equation 2.20. Unlike the case of bilinear interpolation, there is no need to handle each region quadrant separately since they all have the same coefficients.

2.7.2 Running Time Analysis

In the two filtering examples, the function filtered on, $f(\mathbf{x})$, is the intensity at point $\mathbf{x}$. Since intensity values do not affect the computation time, we generate images with a constant intensity value. Generated images are squares that differ in the number of pixels, *i.e.* area. Generated image areas range from 10000 to 200000 pixels, with an increment of 10000 pixels.

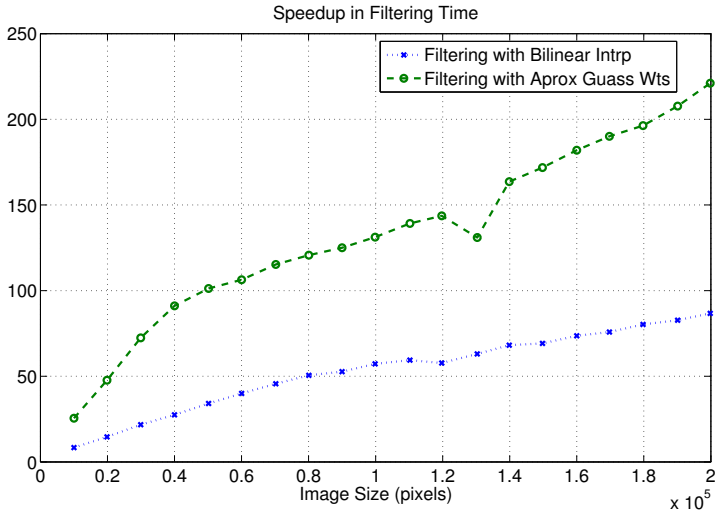


Figure 2.5: Speedups of using integral images compared to conventional method. These plots consider speedups in filtering time only.

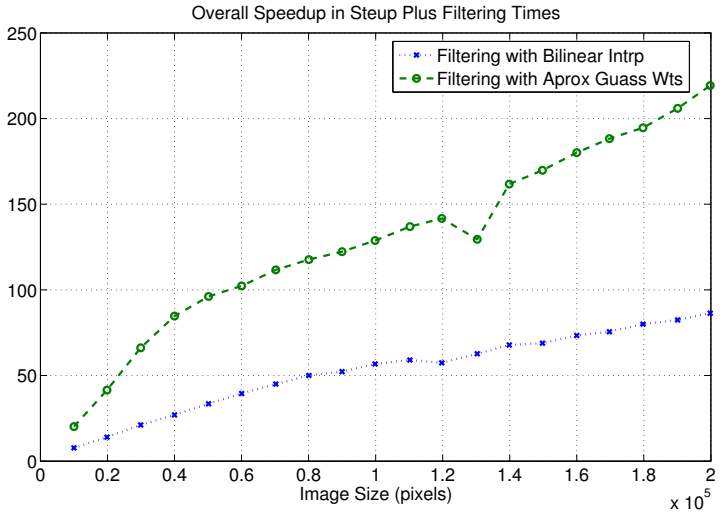


Figure 2.6: Speedups of using integral images compared to conventional method. These plots consider speedups when adding construction time to filtering time.

Each image is scanned with sampled region sizes and locations. The minimum region side length was set to 5 pixels, with side length increment of 5 pixels. Images are scanned with each region size in all possible locations with increments of 5 pixels in both directions. For each region, the two filtering types are computed using integral images and using conventional filtering. For each image, two time periods are measured: 1) the time to set up necessary structures, that is integral images or just type conversion when the conventional filtering is used, 2) and the time to scan the image and compute filtering over all scanned regions.

The plots in Figure 2.4 show the slow-downs in the setup time. In the case of bilinear interpolation, the slow down is around 20x, and in the case of approximate Gaussian weighting, it is around 45x. On the other hand, Figure 2.5 shows the speedups obtained when considering only the time to scan the image and evaluate the filtering function at all probed regions. The speedups are monotonically increasing with the image size. For an image size of 200000 pixels, we achieve a speedup of around 90x in the case of bilinear interpolation, and 220x in the case of approximate Gaussian weighting. This shows the significant benefit of using our approach, especially in the case of Gaussian weighting. Therefore, despite the complexity of computing more integral images during setup, filtering with Gaussian weighting benefits more from using integral images. Finally, Figure 2.6 shows speedups when adding the setup and filtering times together. The curves in this figure look very similar to the curves in Figure 2.5, which consider speedups on filtering time only. This shows that in the two weighting schemes evaluated, the setup time is almost negligible with respect to the filtering time.

2.7.3 Relative Error Analysis

In this set of experiments, we evaluate the two fast filtering methods, kernel integral images and repeated integration, in terms of their relative error. The error we measure here is the difference between the value computed by a fast filtering method and the value computed by conventional filtering (equation 2.1). The relative error is the ratio between this difference and the value computed by conventional filtering.

We generate 10 random images of size 1024×1024. We evaluate the filtering function on a region of size 31×31 at all possible locations in the image. For each location we compute the relative error and plot relative error values against the distance from the region's top-left corner to the image's top-left corner. The distance measure we use is the area of the rectangle bounded by these two corners. This distance measure is equivalent to the number of feature points that are added to produce the integral value(s) associated with the region's top left corner. The error is expected to increase with this distance measure.

In the case of bilinear interpolation, relative errors are always zeros, but not so for approximate Gaussian weighting. The problem with the approximate Gaussian weighting is the integration of higher order contribution functions, such as $x_1^2 x_2^2 f(x)$. These contribution functions require higher precision to represent. Their integrals require even higher precision that is outside the range the double-precision floating point representation. Figure 2.7 shows a third-degree polynomial fit of the relative errors in the case of approximate Gaussian weighting using kernel integral images. The figure compares two methods of computing integrals in terms of the error they

produce. The one-pass method scans the image once and computes the value of the integral at a pixel as a function of its three preceding pixels. The two-pass approach scans the image twice: once integrating horizontally and once vertically. The error generally increases with the distance from the origin. The two-pass method produces around an order of magnitude lower error than the one-pass method. That is expected since in the one-pass method, numbers grow more rapidly allowing for larger errors when adding two numbers that differ by many orders of magnitude.

Figure 2.7 also shows the relative errors, using two-pass integration, of the repeated integration method when used to approximate Gaussian filters with a quadratic filter. The error of our approach, even when using one-pass integration, is lower than the error of the repeated integration method. Similar to our approach, the repeated integration method produces no errors when applied to bilinear interpolation filtering.

In these experiments we use non-negative numbers to represent intensity and pixel coordinate values. These values can be linearly mapped to allow for both negative and positive numbers. In this way, the effective precision used can be increased by utilizing the sign bit in the binary representation, and therefore the accuracy can be enhanced, as shown in Hensley *et al.* [44].

2.8 Conclusion and Future Work

We presented an extension to the integral images framework that allows for fast filtering under non-unform region-dependent weighting of feature values. We refer

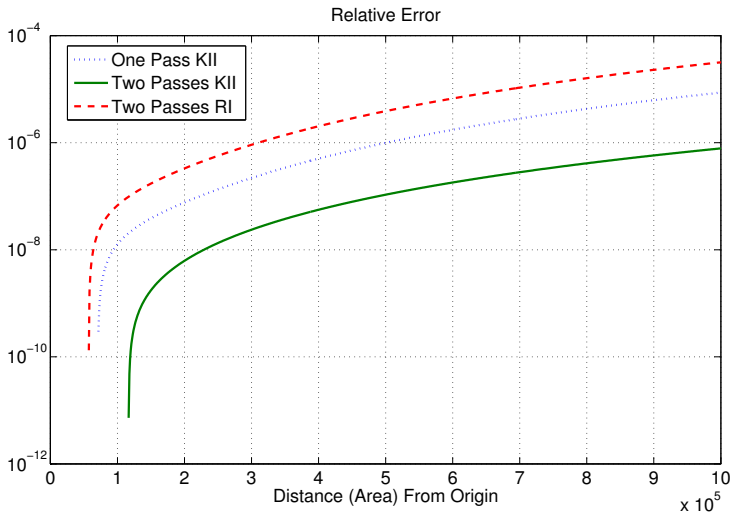


Figure 2.7: Relative errors of computing Gaussian weighted filtering as a function of distance (area) to the origin. KII stands for Kernel Integral Images. RI stands for Repeated Integration.

to the extended framework as kernel integral images. To show the utility of the extension, we provided two examples of widely used non-uniform filtering: one that can be implemented exactly via our framework, that is filtering with bilinear interpolation, and one that can be approximated, which is filtering with Gaussian weighting. Our experiments show that using our approach, significant speedups can be achieved. The presented technique provides a higher precision and more suitability for parallel implementation than the repeated integration approach [42], which also extended the integral images framework for complex filters.

The limitation of our approach, especially in a case such as Gaussian weighting, is the the reduction in precision may not be tolerable in high resolution images. In the future, we are planning to address this issue by developing appropriate image decomposition techniques. Another interesting future direction is to extend the

framework further to handle regions with different shapes than the upright rectangular one.

Chapter 3

Boosted Deformable Features

It is a common practice to model an object for detection tasks as a boosted ensemble of many models built on features of the object. In this context, features are defined as subregions with *fixed* relative locations and extents with respect to the object's image window. In this chapter, we introduce using deformable features with boosted ensembles. A deformable features adapts its location depending on the visual evidence in order to match the corresponding physical feature. Therefore, deformable features can better handle deformable objects. We empirically show that boosted ensembles of deformable features perform significantly better than boosted ensembles of fixed features for human detection.

3.1 Introduction

Human detection methods can be categorized into two groups based on the camera setup. For static camera setups, object motion is considered as the distinctive feature. A motion detector, either a background subtraction or an image segmentation method, is applied to the input video to extract the moving regions and their motion statistics [41, 75]. A real time moving human detection algorithm that uses Haar wavelet descriptors extracted from space-time image differences was introduced by Viola and Jones [97], where, using AdaBoost [83], the most discriminative frame dif-

ference features were selected, and a rejection cascade was constructed to efficiently reject negative examples. A shortcoming of the motion based algorithms is that they fail to detect stationary pedestrians, or when motion information is not available. In addition, such methods are highly sensitive to view-point and illumination changes.

The second category of methods is based on detecting human shape and silhouette. Approaches for shape-based human detection can be further categorized based on how the human body is modeled. In one subcategory, a holistic model is used, where the human body is modeled as a whole without being divided into smaller parts. Examples in this category include the hierarchical template matching used in Gavrila and Philomin [38], and Zhao and Davis [106], and the neural network based approach in Zhao and Thorpe [107]. In a second subcategory, a part-based model is used, where models for parts of the body are learnt, possibly along with global constraints, such as in Mohan *et al*. [65], Ioffe and Forsyth [52], Ronfard *et al*. [76], Mikolajczyk *et al*. [64, 63], Felzenszwalb and Huttenlocher [30], and Lin *et al*. [60]. Part-based models, in general, deliver better performance than holistic models because of their ability to model deformation and occlusion. However, the drawback of most part based models is that the number of parts and their locations have to be manually determined.

A third subcategory of approaches addresses this problem by modeling the body as an ensemble of local features. A feature in this context is a subregion of an object's image window. One way to model the human body as an ensemble of features is through detection of local features and combining them using global geometric information [57, 71]. Another way, which is more attractive and commonly

used, is though boosting techniques [36, 83], which select the most discriminative features among all possibilities dynamically. Boosting techniques are more attractive because they can be used to construct fast rejection cascades of classifiers. Following the pioneering work in this direction by Viola and Jones [97], a series of successful approaches were introduced using different feature representations, such as histograms of oriented gradient (HOG) [109], region covariance [94], edgelet features [103], and shapelet features [81]. Approaches such as Dalal and Triggs [20] and Papageorgiou and Poggio [72], where descriptors of many small blocks, overlayed on a grid layout, are concatenated to build a large holistic representation, are often viewed as holistic representations. We view them as static ensembles of features. Despite their static nature, these ensembles can handle many object types due to the large number of features and the overlap among them. Finally, it is worth noting that the aforementioned categorization is not a partitioning of the available models. Hybrid models also exist [103, 31].

The work presented in this chapter falls under the subcategory of feature-based models for shape-based detection. The common drawback of feature-based models that are built using boosting is the difficulty of handling deformation. This is because features are passive elements who do not adapt to an object's deformation or shape changes. For example, they are bound to a fixed location, relative to the object's window. However, in highly deformable objects, this is hardly the case. Consider for example the head part/feature in the human images in Figure 3.1, from the INRIA Person dataset [20]. The head is a discriminating physical feature between humans and background. A subregion that designates the head position

in Figure 3.1a is marked with a white rectangle. But, the same subregion, marked with dotted rectangles, in the rest of Figure 3.1 either has the head off-centered or even off border. This problem can result in a poor fit of the models built on such features. Felzenszwalb *et al.* [31] proposed deformable part models to handle this problem. However, as other part models, this work lacks the flexibility of automatically determining the number, locations, and sizes of parts. In this chapter, we introduce deformable features, instead of deformable parts, to be used in boosting ensembles.

It is also worth noting that there are approaches that combine motion and shape to deliver better performance. The approach of Dalal and Triggs [20], which uses HOG descriptors and SVM, was extended to optionally account for motion by extending the histograms to include flow information in Dalal *et al.* [21]. In our prior work on real time human detection in moving-camera videos [47], frame differencing was used to detect motion areas before applying a shape-base detector. Then, tracking and motion analysis were used to verify detections.

The rest of the chapter is organized as follows: Related work is discussed in Section 3.2. Section 3.3 introduces deformable features. Boosting of deformable features is explained in Section 3.4. Details of our implementation and experimental results are provided in Sections 3.5 and 3.6. Finally, the chapter is concluded and the future work is outlined in Section 3.7.

3.2 Related Work

Having uncertainty in feature locations have long been used in constellation models [32, 30]. These models are primarily used for object category recognition. They are too complex to use for object detection where multiple instances of the object may be existing in the image and they all have to be accurately and efficiently located. Deploying deformation in object detection have been recently introduced. In Tran and Forsythe [92], the body configuration is estimated first. Then, gradient based descriptors for different parts are concatenated to form a descriptor for the body, which is eventually classified using an SVM. In Felzenszwalb *et al.* [31], distance transform techniques are used to efficiently find the best location for each part. Then HOG descriptors for the parts are concatenated along with the HOG descriptor of the whole body to form a long feature vector for SVM classification. In both techniques, the number of parts are manually determined and each part's model is complex with many elements. We are interested in deforming simpler feature descriptors.

In Lin *et al.* [59], and Wu *et al.* [105], deformation is allowed at the feature's level. Similar to our approach, features on a grid layout are considered. However, different from our approach, features are not allowed to find their locations. Instead, a global process, pose estimation and MRF inference respectively, is used to select a subset of the features to be included in the final body descriptor based on the likelihood of lying on the human's silhouette. The global process in these cases does not allow for autonomous operation of individual features, which is one of our goals.

(a) (b) (c) (d) (e) (f)

Figure 3.1: An illustration of the desired behavior of a deformable feature for the head. The feature's initial location is marked by a dotted rectangle and the desired final location is marked with a solid rectangle. Notice how the initial location is often not aligned with the actual location of the physical feature (head).

3.3 Deformable Features

In the context of feature-based models for object detection, we define a deformable feature (d-feature) as a feature that is not bound to a fixed location in the object's model. Rather it can move (translate) in a small neighborhood around a central location. We would like a d-feature to be able to locate the physical feature it represents within this neighborhood. Figure 3.1 illustrates the desired behavior of a d-feature that represents the head of a human. Starting from an initial (typical) location for the physical feature, illustrated as a dotted rectangle, the feature moves to a better location to capture the physical feature. In this section, we explain how to train a model for a d-feature. In Section 3.4, we explain how to combine models for individual d-features to build an ensemble that represents the object as a whole.

3.3.1 Learning Deformable Features

The main advantage of feature-based models is the automatic selection of representative features from a very large pool. We do not even need to know what the underlying physical features are. Therefore, our framework has to be able to automatically learn d-features based solely on the image data.

Let $\mathbf{F} = (\mathbf{s}, \mathbf{z}_0, \mathbf{Z})$ be a d-feature identified by its size $\mathbf{s}$, its initial location $\mathbf{z}_0$ and a neighborhood $\mathbf{Z}$ relative to $\mathbf{z}_0$ in which the feature is allowed to move. Let $\Delta_{\mathbf{F}}(\mathbf{x}, \mathbf{z})$ be a descriptor of the feature's appearance in an example $\mathbf{x}$ at location $\mathbf{z} \in \mathbf{Z}$, *e.g.* a HOG descriptor [109]. For simplicity, we will omit the variable $\mathbf{x}$ when confusion is not expected. Let $\theta(\Delta_{\mathbf{F}}, \mathbf{z})$ be a scoring function that measures the likelihood of an example being positive given the appearance of the feature $\mathbf{F}$ at location $\mathbf{z}$, *i.e.* $p(\mathcal{O}|\Delta_{\mathbf{F}}(\mathbf{z}), \mathbf{z})$. Note that, θ depends on both $\Delta_{\mathbf{F}}(\mathbf{z})$ and $\mathbf{z}$. This allows us to model the case when the prior probability of $\mathbf{z}$ is not uniform.

On one hand, to learn the scoring function θ, we need to know the locations of the feature $\mathbf{F}$ in the training examples. On the other hand, to estimate the location of the feature in a given example, we need an objective function (scoring function) to optimize (maximize) over the feature's neighborhood $\mathbf{Z}$. To break this cycle, we can start with an approximation to the scoring function by assuming the feature's location in all training examples to be the initial location $\mathbf{z}_0$. Let θ_0 be the initial estimate for the scoring function obtained based on this assumption. Recall our prior assumption that features move within a small neighborhood around their initial (typical) locations. If we further assume also that typically the feature is

procedure DEFREFINE($\mathbf{F}, \mathcal{X}$)

▷ $\mathbf{F}$ is a feature, $\mathcal{X}$ is a set of N training examples

$\forall \mathbf{x}^i \in \mathcal{X}, \mathbf{z}_0^i \leftarrow \mathbf{z}_0$

for $j = 0$ to k **do**

Estimate θ_j based on $\mathbf{z}_j^i, i = 1..N$

$\mathbf{z}_{j+1}^i \leftarrow \arg\max_{\mathbf{z} \in \mathbf{Z}} \theta_j(\Delta_{\mathbf{F}}(\mathbf{x}^i, \mathbf{z}), \mathbf{z}), \forall i$

end for

end procedure

Figure 3.2: Pseudo-code for the d-feature model refinement procedure.

close to its initial location, then the initial model θ_0 is expected to capture the rough appearance of the feature. Therefore, we can use θ_0 to estimate the feature location in a given example by maximizing the function over the neighborhood $\mathbf{Z}$. Given these estimated locations, we can learn a better estimate for the scoring function θ. We can keep iterating over these two steps to reach a refined estimate for the scoring function θ. This procedure is illustrated in Figure 3.2.

To visualize the effect of refining the d-feature's model, consider the toy classification task illustrated in Figure 3.3. In this task, all images are 40×40. Positive samples contain circles with the same radius of 8 pixels. The circles can be at random locations in the 20×20 central square of the image. Negative images contain random points in the same central square. We trained a Linear Discriminant Analysis model on the raw binary pixel values of the internal 20×20 squares in all images.

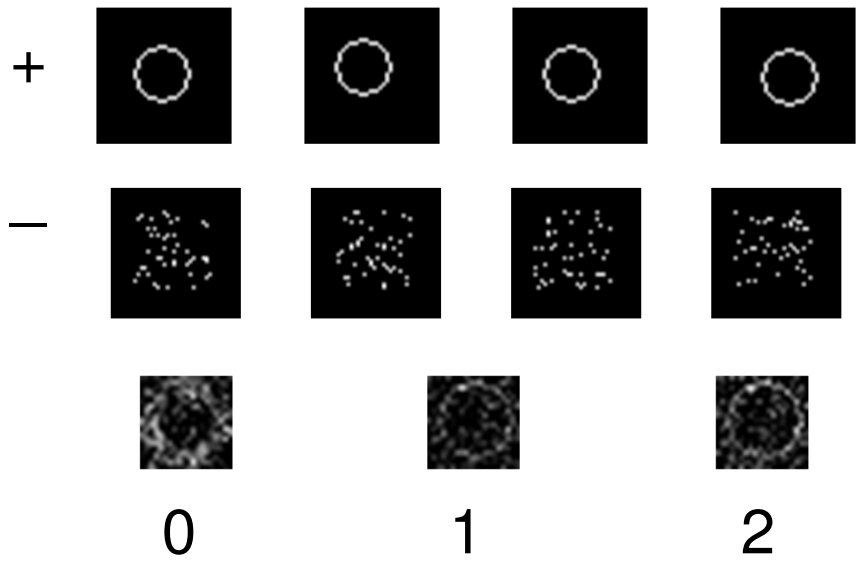


Figure 3.3: A toy classification task to illustrate the effect of refining d-feature's model. Sample positive and negative images are in the first the second rows. The learned weight vector after 0, 1, and 2 refinement iterations, Figure 3.2, are in the bottom row. Refinement enhances the match to the shape of the positive object.

In the bottom row of Figure 3.3, we show the obtained weight vectors after 0, 1, and 2 refinement iterations. We can observe that the more we refine the model, the better it matches the shape of the object we are training for, which is a circle in this case.

3.3.2 Classification With Deformable Features

We explained how a d-feature learns its best location on each training example and its object likelihood function through iteratively refining both in alternation. On a testing example, we select the feature location, $\mathbf{z}^*$, to be the location that maximizes the scoring function, and then consider the score at that location to be the object likelihood, equations 3.1 and 3.2.

$$\mathbf{z}^* = \arg\max_{\mathbf{z}\in\mathbf{Z}} \theta(\Delta_{\mathbf{F}}(\mathbf{z})) \tag{3.1}$$

$$\theta^* = \theta(\Delta_{\mathbf{F}}(\mathbf{z}^*)) \ . \tag{3.2}$$

This procedure is equivalent to finding the Maximum Likelihood (ML) estimate of the feature location and using the corresponding object likelihood value as the score of the feature for the given test sample. This is similar to the way parts are deformed by Felzenszwalb *et al*. [31].

3.4 Boosted Deformable Features

A boosting algorithm forms a strong classification ensemble out of weak classifiers. It adds ensemble members incrementally so that each newly added member performs the best in the training samples that are poorly learned by the current ensemble.

In feature-based detectors, each weak classifier is built on a single feature, and the boosting algorithm selects one feature to add to the ensemble in every iteration. There are several variants of boosting. We experimented with the LogitBoost algorithm [36]. For completeness of presentation, the algorithm is reproduced in Figure 3.4 with the necessary modifications to fit with our framework. The only change is in the fitting of z_i to x_i, where z_i is computed by the algorithm, and x_i in our case is the $\Delta_{\mathbf{F}}$ descriptors. In the case of d-features, we do not apply one step of least squares regression. Instead we use the iterative procedure in Figure 3.2 to allow the feature to find its best location. An important point to make here is that values of the θ function used to update $F(x)$ in the final step of the for loop of LogitBoost must be based on the estimates of the best locations $\mathbf{z}$ computed in the final iteration of DefRefine in Figure 3.2. It is tempting to skip computing the $\mathbf{z}$ values in the final iteration, since they are not used to update the model for θ again. However, they are used to compute the object likelihood scores, which, in turn, are used to update $F(x)$ of LogitBoost.

3.5 Implementation Details

We use the HOG descriptor [109] to represent the features. The HOG descriptor of a feature is a concatenation of four histograms, each built on one quadrant of the feature. Each histogram contains 9 bins representing 9 ranges of orientation directions. Each pixel contributes to two bins of the histogram by linear interpolation. Each pixel also contributes to the 4 quadrants with bilinear interpolation. Computing

procedure LogitBoost($\mathcal{F}, \mathcal{X}$)

▷ $\mathcal{F}$: set of M features, $\mathcal{X}$: set of N examples

$\forall \mathbf{x}^i \in \mathcal{X}, w_i = \frac{1}{N}, p(\mathbf{x}^i) = \frac{1}{2}, F(\mathbf{x}^i) = 0$

for $k = 1$ to K **do**

Compute the working response and weights

$$\begin{aligned} z_i &= \frac{y_i^* - p(\mathbf{x}^i)}{p(\mathbf{x}^i)(1 - p(\mathbf{x}^i))} \\ w_i &= p(\mathbf{x}^i)(1 - p(\mathbf{x}^i)) \end{aligned}$$

$\forall \mathbf{F} \in \mathcal{F}$ fit the function $\theta_{\mathbf{F}}$

by a weighted least-squares regression of z_i to $\mathbf{x}^i$

with weights w_i using the procedure in Figure 3.2.

Update $F(\mathbf{x}) \leftarrow F(\mathbf{x}) + \frac{1}{2} f_k(\mathbf{x})$, and

$p(\mathbf{x}) \leftarrow e^{F(\mathbf{x})} / (e^{F(\mathbf{x})} + e^{-F(\mathbf{x})})$,

where $f_k(\mathbf{x})$ is $\theta_{\mathbf{F}}$ that minimizes the residual.

end for

Output the classifier $sign[F(\mathbf{x})]$

end procedure

Figure 3.4: Pseudo-code for the LogitBoost algorithm on d-features. Note that y_i^* is set to 0 for a negative example and to 1 for a positive example.

these descriptors is very fast using kernel integral images [48] (Chapter 2).

The deformation neighborhood is made to be double the size of the feature in both dimensions, with a maximum of 16 pixels away from the feature's boundary. On searching for the best feature location, we use 5 steps in each dimension. We use two types of scoring functions. One is based only on the descriptor and the other is based on the descriptor and location together. In the latter version, similar to Felzenszwalb *et al.* [31], we concatenate $\delta = \mathbf{z}^* - \mathbf{z}_0$ and its element-wise square to the descriptor and estimate a function θ based on the concatenated descriptor. Since the function θ in our case is a linear function, the concatenation of δ values to the descriptor is equivalent to decomposing θ as $\theta_{descriptor} + \theta_{displacement}$. Therefore, this is equivalent to using an additive penalty term in the scoring function. This is also equivalent to learning a non-uniform prior for the feature location.

We use a rejection cascade [98] of 30 layers of LogitBoost classifiers. Each layer is adjusted to produce detection rate of 99.8% at false alarm rate of 65%.

3.6 Experimental Results

We trained and tested all our classifiers on the INRIA Person dataset [20]. In this dataset, training and testing positive images are resized so that the human body is around 96 pixels high. A margin of 16 pixels is added to the top and the bottom to make the height 128 pixels and the width 64 pixels. The negative testing images are scanned with this window size (64×128) with a step of 8 pixels in both dimensions, to create close to a million sample negative images.

In this section, we refer to the variant of d-features that uses an additive penalty term in the scoring function (Section 3.5) by Max-Def-Add, and the variant without penalty as Max-Def. We experimented with the two variants with number of refinement steps 1 or 2, along with the conventional Non-Def features (0 refinements). We use DET (Detection Error Tradeoff) curves to present the detection results, Figure 3.5, where the plots are generated by changing the number of cascade layers. In these plots the number of refinements appears at the end of the legend, when applicable. As the figure shows, only the Max-Def-2 and the Max-Def-Add-2 consistently outperform the Non-Def classifier. Max-Def-Add-1 compares favorably over most of the false alarm rate's range. Max-Def-1 is inferior to the Non-Def classifier beyond false alarm rates of 2×10^{-3}. Max-Def-Add-2 is the clear winner among all. At a false alarm rate 3×10^{-4}, Max-Def-Add-2 reduces the miss rate compared to Non-Def by 30%, from 10% to 7%. At the miss rate of 8%, it reduces the false alarm rate to about one third, from 8×10^{-4} to 2.5×10^{-4}. These results highlight the value of d-features and the importance of performing multiple refinement iterations during training.

In Figure 3.6, examples of detection errors obtained using Non-Def that are successfully corrected using Max-Def-Add-2 are shown. To produce these images, each classifier is applied to the image using a sliding window approach, where the search step is set to 5% of the size of the search window in each dimension. The search sizes are selected based on knowledge of ground truth annotations. The resulting detection windows are then grouped using the mean shift algorithm on the location and height of the windows. For each searching size, the image is resized so

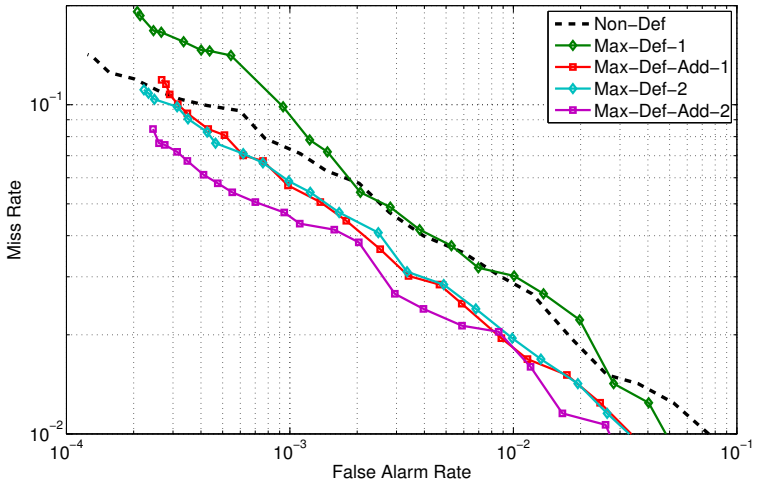


Figure 3.5: DET curves for cascaded boosted HOG features classifiers on INRIA Person dataset with and without d-features. Using d-features helps reduce the miss rate by up to 30% at false alarm rate of 3×10^{-4}, and reduce the false alarm rate by 66% at the miss rate of 8%.

that we always search using the size used in training.

Figure 3.6: Sample detections by the cascade boosted HOG features classifiers. The top image of every pair has the results from the Non-Def classifier (without d-features). The bottom image has the results with d-features with 2 refinement iterations and with a penalty term.

3.7 Conclusion and Future Work

We introduced deformable Features (d-features in short) and showed how they can be used to enhance the performance of boosted feature-based object detectors. The advantage of d-features over the regular ones is their ability to search for the locations of the corresponding physical features before computing their matching scores. This property makes them able to better handle complicated object structures and deformations than fixed location features. We experimented with d-features on human detection in a cascaded boosting framework. Our experiments showed a consistent enhancement in performance when using d-features.

We use brute force search in our current implementation, which makes training and testing classifiers using d-features slow. However, the distance transform techniques [30] can be used to make it more efficient. This approach can be extended in many other ways. We can apply the d-features using other common descriptors, such as the covariance descriptors [94], and other types of ensembles, such as the concatenation ones [20]. Other objects, rigid and non-rigid, can benefit from the approach.

Chapter 4

A Comprehensive Evaluation Framework

In this chapter, we introduce a framework for evaluating human detectors that considers the practical application of a detector on a full image using multi-size sliding window scanning. We produce DET (Detection Error Tradeoff) curves relating miss detection rate and false alarm rate computed by deploying the detector on cropped windows as well as whole images, using in the later either image resize or feature resize. Plots for cascade classifiers are generated based on confidence scores instead of varying the number of layers. To assess a method's overall performance on a given test, we use the ALMR (Average Log Miss Rate) as an aggregate performance score. To analyze the significance of the obtained results, we conduct 10-fold cross validation experiments. We applied our evaluation framework to two state of the art cascade-based detectors on the standard INRIA Person dataset, and another dataset of of near infrared images provided by MERL. We used our evaluation framework to study the differences between the two detectors on the two datasets with different evaluation methods. Our results show the utility of our framework. They also suggest that the descriptors used to represent features, and the training window size are more important in predicting the detection performance than the nature of the imaging process, and that the choice between resizing images or features has serious consequences.

4.1 Introduction

Despite the difficulty of the problem of human detection, there has been a significant advancement in this area of research recently. Nevertheless, little attention has been given to evaluation of detectors for practical applications. First, there is a notable mismatch between the way detectors are evaluated and the way they are applied in real world applications, such as smart vehicle systems. At one end, detectors are evaluated on "ideal" windows that are cropped to have the human subjects centered in them, and resized to match the window size used in training. However, at the other end, detectors are applied to whole images, typically using a multiple-size sliding-window approach, which results in probe windows that are far from being ideal. Second, most of the evaluations are performed on a single dataset, which leaves practitioners with uncertainty about the detection performance on other datasets, possibly with different modalities, or the significance of one detector's advantage over the other. Third, for detectors based on cascade classifiers, typically performance plots are created by changing the number of cascade layers. This technique sometimes leads to difficulty in comparing different methods when the resulting plots do not cover the same range of false alarm rates.

The main contribution presented in this chapter is an evaluation framework that handles the shortcomings of the existing evaluations. The main features of our evaluation are:

- Comparing between evaluation on cropped windows and evaluation on whole images to get a better prediction for a detector's performance in practice and

how it differs from ideal settings.

- Using 10-fold cross validation to be able to study the significance of the obtained results.

- Plotting DET curves based on confidence scores for detectors based on cascade classifier instead of plotting them based on varying the number of layers.

- Introducing an aggregate performance score and using it as the main metric to statistically compare methods.

- Comparing between building a multi-size image pyramid while fixing the scanning window size, and using a single image size and changing the scanning window size, when applying the detector on whole images. We refer to these two choices as *resizing images* and *resizing features*, respectively. This is an example of an implementation choice that can have a significant effect on the detection performance depending on the evaluated detector.

- Evaluation on near infrared images as well as visible images.

The goal of our study is not to provide a performance comparison for the state of the art human detection techniques. Instead, our goal is to introduce a comprehensive evaluation framework and to highlight the mismatch between the typical evaluation techniques and the practical deployment of the detectors. We utilized the two detectors in Zhu *et al.* [109] and Tuzel *et al.* [94] to demonstrate our evaluation framework. To the best of our knowledge, these are the best performing human detectors based on rejection cascades. We focus on rejection cascades because

they are appealing for practical applications, as explained in Section 4.2. Despite that our presentation focuses on human detection, our framework and observations apply to other objects as well.

Our experimental results show the utility of our framework in understanding the performance of a human detector in practice. They suggest that the descriptors used to represent features, Histograms of Oriented Gradients or Region Covariances in our study, and the size of the training window are more important in predicting the detection performance than the nature of the imaging process, such as the imaged electromagnetic band. They also show that the choice between resizing images or features can have a significant impact on the performance depending on the used descriptor.

The chapter is organized as follows. In Section 4.2, we briefly describe the two pedestrian detectors used in our evaluation. In Section 4.3, we explain the elements of our evaluation framework. In Section 4.4, we introduce the two datasets we use and how we prepared them for the experiments. In Section 4.5, we present the results and analysis of our evaluation. Finally, the conclusion and future directions are given in Section 4.6.

4.2 Evaluated Detectors

The two human detectors which we use in our evaluation are based on a rejection cascade of boosted feature regions. They differ in how they describe the feature regions and in how the weak classifiers are trained. One detector uses Region Co-

variance to describe feature regions and uses classification on Riemannian manifolds for the weak classifiers [94]. We refer to this detector as COV. The other detector uses Histograms of Oriented Gradients (HOG) to describe feature regions and uses conventional linear classification [109]. We refer to this detector as HOG. For the sake of completeness, we briefly describe here the notion of a rejection cascade of boosted feature regions, as well as the descriptors used by the two classifiers. The reader is referred to the original papers for more details.

4.2.1 Rejection Cascade of Boosted Feature Regions

Rejection cascades of boosted feature regions were popularized by their success in the area of face detection [98]. They are based on two main concepts: *boosted feature regions*, and *rejection cascades*.

In boosting [36], a *strong classifier* is built by combining a number of *weak classifiers*. Boosting *feature regions* can be understood as combining simple feature regions to build a strong representation of the object that can be used to distinguish the object from other stuff. Feature regions in our case are rectangular subregions from *feature maps* of input images, as shown in Figure 4.1. The concept of a feature map is explained in Section 4.2.2.

A *rejection cascade* is built of a number of classification layers. As shown in Figure 4.2, a test pattern is examined by layers of the cascade one after another until it is rejected by one of them, or until it is accepted by the final layer, in which case it is classified as a positive example. During training of the cascade, the first

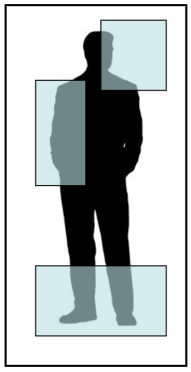


Figure 4.1: Shaded rectangular subregions of the detection window are possible features to be combined to build stronger boosted features.

layer is trained on all positive examples and a random sample of negatives examples. Each subsequent layer is trained on all positive examples and the false positives of the preceding layers. In this way, each layer handles harder negative examples than all the preceding layers. The benefit of this mechanism is two fold. One is the possibility of using a huge number of negative examples in training the classifier, which is not possible in training a traditional single layer classifier. The other is that, during testing, most negative examples are rejected quickly by the initial layers of the cascade and only hard ones are handled by the later layers. Since in our applications, it is likely that most of the examined patterns are negative, rejection cascades are computationally efficient since they quickly reject easy negative examples while spending more time on the hard negative or the positive examples. In our implementation, each cascade layer is trained using the LogitBoost algorithm [36].

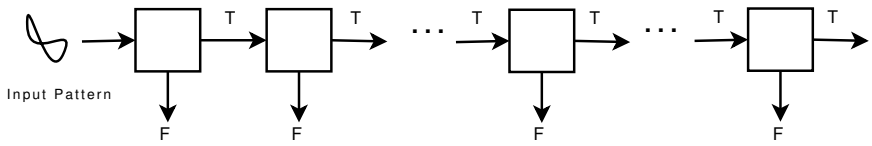


Figure 4.2: A rejection cascade consists of layers. A test pattern is examined by layers in the cascade from left to right until being rejected. A pattern is accepted if all layers accept it.

4.2.2 Region Covariances

Region covariances were first introduced as descriptors in Tuzel *et al.* [93] and then used for human detection [94], which outperformed other state of the art classifiers. Let I be a $W \times H$ one-dimensional intensity or a three-dimensional color image, and F be a $W \times H \times d$ dimensional feature map extracted from I

$$F(x, y) = \Phi(I, x, y) \tag{4.1}$$

where the function Φ can be any mapping such as intensity, color, gradients, filter responses, etc. For a given rectangular region $R \subset F$, let $\{\mathbf{z}_i\}_{i=1..S}$ be the d-dimensional feature points inside R. The region R is represented with the $d \times d$ covariance matrix of the feature points

$$\mathbf{C}_R = \frac{1}{S-1} \sum_{i=1}^{S} (\mathbf{z}_i - \mu)(\mathbf{z}_i - \mu)^T \tag{4.2}$$

where μ is the mean of the points.

For the human detection problem, the mapping $\Phi(I, x, y)$ is defined as

$$\left[x \;\; y \;\; |I_x| \;\; |I_y| \;\; \sqrt{I_x^2 + I_y^2} \;\; |I_{xx}| \;\; |I_{yy}| \;\; \arctan \frac{|I_x|}{|I_y|} \right]^T \tag{4.3}$$

where x and y represent pixel location, $I_x, I_{xx}, ..$ are intensity derivatives, and the last term is the edge orientation. With this definition, the input image is mapped to a $d = 8$ dimensional feature map. The covariance descriptor of a region is an 8×8 matrix and due to symmetry only the upper triangular part is stored, which has only 36 different values. To make the descriptor invariant to local illumination changes, the rows and the columns of a subregion's covariance matrix are divided by the corresponding diagonal elements in the entire detection window's covariance matrix.

Region covariances can be computed efficiently, in $O(d^2)$ computations, regardless of the region size, using integral histograms [74, 93]. Covariance matrices, and hence region covariance descriptors, do not form an Euclidean vector space. However, since covariance matrices are positive definite matrices, they lie on a connected Riemannian manifold. Therefore, classification on Riemannian manifolds is more appropriate to be used with these descriptors [94].

4.2.3 Histograms of Oriented Gradients

Histograms of Oriented Gradients were first applied to human detection in Dalal and Triggs [20], which achieved a significant improvement over other features used for human detection at that time. Histograms of Oriented Gradients were used in a rejection cascade of boosted feature regions framework in Zhu *et al.* [109] to deliver comparable performance to Dalal and Triggs [20] at a much higher speed.

To compute the Histogram of Oriented Gradients descriptor of a region, the

region is divided into 4 cells, in a 2×2 layout. A 9 bin histogram is built for each cell. Histogram bins correspond to different gradient orientation directions. Instead of just counting the number of pixels with a specific gradient orientation in each bin, gradient magnitudes at the designated pixels are accumulated. Bilinear interpolation is used between orientation bins of the histogram and spatially among the 4 cells. The four histograms are then concatenated to make a 36-dimensional feature vector, which is then normalized. In our implementation, we use L_2 normalization for HOG features.

Like Region Covariance descriptors, HOG descriptors can be computed fast using integral histograms. Bilinear interpolation among cells is computed fast using the kernel integral images approach [48] (Chapter 2).

4.3 Evaluation Framework

In most recent studies on human detection, evaluation results are presented in DET (Detection Error Tradeoff) curves, which relate the false alarm rate per window to the miss rate of the classifier in a log-log scale plot. Typically, positive examples used in the evaluation are adjusted to have the same subject alignment and size used in training the classifiers, and negative examples are human-free. In this section, we identify several shortcomings of this evaluation approach. We explain how we address these shortcomings in our evaluation framework.

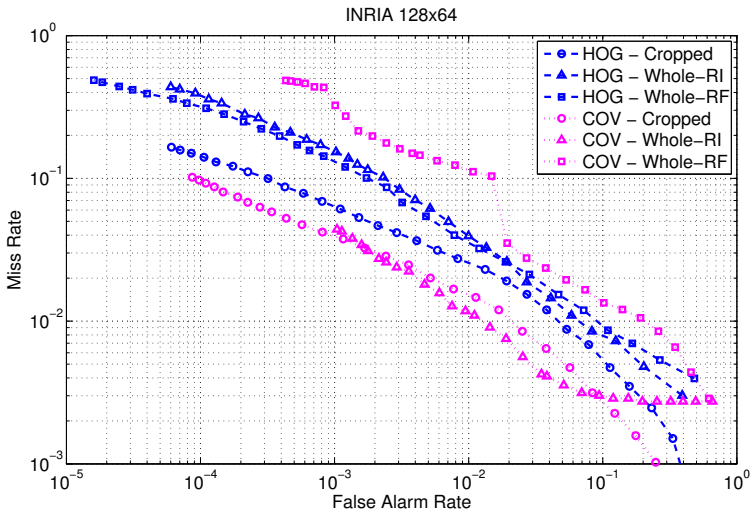


Figure 4.3: DET-Layer plots for the INRIA dataset with window size 128×64.

4.3.1 Score Plots for Cascade Classifiers

Typically, points on DET curves of cascade classifiers are generated by changing the number of cascade layers. The problem with this approach is that the generated plots are not guaranteed to cover a particular range for either the horizontal or the vertical axes, which makes it hard to compare different methods. Figure 4.3 shows examples of such plots. To overcome this problem, in our evaluation, we compute a confidence score for each sample and generate the plots based on these scores. We assume that each layer of the cascade can give a confidence score $\varphi(\mathbf{x}) \in (0, 1)$ to any given example $\mathbf{x}$. The overall confidence score over an n layer cascade can be expressed as

$$\Phi(\mathbf{x}) = \mathcal{N}(\mathbf{x}) + \varphi_l(\mathbf{x}) \ , \tag{4.4}$$

where $\mathcal{N}(\mathbf{x})$ is the number of layers that accepted $\mathbf{x}$, and $\varphi_l(\mathbf{x})$ is the confidence score of the last layer that examined it. The score in 4.4 reflects the way a cascade classifier works. It gives higher scores to examples that reach deeper in the cascade. If two examples leave the cascade at the same layer, their confidence scores will differ by the confidence scores assigned by the last layer. In this way, we get a real valued score. We can create DET curves from these scores by changing the threshold above which a test example is considered positive. At each point on the curve, we set the threshold appropriately to generate a specific level of false alarm rate. Then, we measure the miss rate at this threshold value. In this way, we have control over the range of false alarm rates to cover. Figure 4.7 shows the same results of Figure 4.3 using confidence scores.

In our implementation, each layer of the cascade is a boosted classifier. The real-valued outcome of such a classifier is proportional to the number of weak classifiers in it. Hence, we normalize this outcome by the number of weak classifiers to produce the layer's score in the range $(-6, 6)$. Then this value is mapped to the range $(0, 1)$ using the sigmoid function $\exp(x)/(\exp(x) + \exp(-x))$.

4.3.2 Evaluation on Whole Images

Evaluation on cropped windows is an optimistic estimate of the detector's performance in practice. Typically, detectors are applied to whole images using a multiple-size sliding window scanning. The windows fed to the classifier in this case can rarely have humans centered in them or have the proper size, which would yield a lower

performance than in the case of application on cropped windows. We evaluated the classifiers on both cropped windows and whole images to compare between them. In the case of evaluation on cropped windows, the positive and negative examples are well defined. However, in the case of evaluation on whole images, the situation is different. In this case, scanned windows are not all perfect positive or negative examples since they may contain parts of humans or full humans who are not in the proper location or relative size. In many applications, if the detection window is slightly shifted, or slightly smaller or larger than the subject, it is still useful. Therefore, we should not consider such windows as negative examples and penalize the classifier for classifying them as positives. However, if we consider all scanned windows that are close to a human subject as positive examples, we will be penalizing the classifier for missing any of them although detecting just one is good enough in practice.

Based on these considerations, in the case of evaluation on whole images, we consider any scanned window that is significantly far from all annotated human subjects in the image as a negative example. A missed detection is counted if an annotated human subject is significantly far from all scanned windows that are classified as positives by the classifier. In other words, a missed detection is counted if all scanned windows that are close enough to an annotated human subject are classified as negatives. The measure of closeness we use is the *overlap ratio*. Let $|R|$ be the area of a region R. Consider two regions R_1 and R_2. The overlap ratio between them is defined as

$$\mathcal{O}(R_1, R_2) = \frac{|R_1 \cup R_2|}{|R_1 \cap R_2|} \ . \tag{4.5}$$

This ratio is minimum (1) when the two regions are perfectly aligned and is maximum (∞) when they have no overlap. In our evaluation, we consider a scan window negative if its overlap ratio to the closest annotated human subject is above 16. We count a miss detection if all scanned windows within overlap ratio of 2 around an annotated human subject are all classified as negatives. The latter threshold is the same used in the Pascal challenge [29]. According to these thresholds, there are windows that are not counted as positives nor as negatives. The upper threshold is rather conservative so that we do not consider a window negative unless it is too far from all annotated human subjects. For assigning scores to windows, negative windows' scores are computed as in 4.4; and, each annotated human subject is assigned the maximum score over all positive windows associated with it.

Another option to present the performance on whole images would be to use PR (Precision Recall) curves. It was shown [22] that PR and ROC curves are closely related in the sense that the dominant curve in one is the dominant curve in the other if they are generated using the same points. We preferred using DET curves, which are the loglog version of ROC curves, so that the the performance on whole images can be compared to that on cropped windows in our results and other published results. Also, to generate a PR plot, nearby detection windows have to be consolidated. First, we selected not to confound the detector's performance by a particular choice of this post processing step. Second, in our framework, consolidation will

have to be applied at each point of the plot, which is prohibitively expensive.

4.3.2.1 Resizing Images vs. Resizing Features

An implementation choice for evaluation on whole images turns out to have a strong effect on the detection performance. We train each classifier on single size images. In the case of applying them on whole images, which contain humans of different sizes, we have two options. One is to resize the images so that our scanning window size becomes the same as the training size. We refer to this option as *resizing images*. The other option is to resize the features selected by the classifier while maintaining their relative sizes to the scan window. We refer to this option as *resizing features*. Resizing features is faster since the preprocessing of the image, *e.g.* computing gradients and integral histograms, is performed only once. We evaluated on whole images using the two options to compare between them.

4.3.3 Statistical Analysis

Statistical analysis of detection performance is rarely conducted for human detection, possibly due to the long training time. To our knowledge, the only study that provided statistical analysis was in Munder and Gavrila [67], where a confidence interval for each point on the ROC curve was computed based on 6 observations (3 training sets $\times$ 2 testing sets). We found it confusing to plot confidence intervals with the plots since in our evaluation plots intersect and come close to one another. Instead, we compute confidence intervals for the aggregate performance

score ALMR, which is explained in Section 4.3.4. We conduct a 10-fold cross validation for all our experiments. Therefore, for each experiment, we obtain 10 different curves. Each curve yields an ALMR score. To compare different experiments, we plot the average curve for each experiment. We also present a box-plot for the mean, confidence interval, and range of the ALMR scores for all experiments in a separate plot. Confidence intervals are computed at the 0.95 confidence level.

4.3.4 Computing an Aggregated Performance Score

To analyze the significance of one method's advantage over another, we need an aggregated score that captures the difference between them over the entire curve. The log-log plots emphasize the relative difference instead of the absolute difference between two curves. We need a score that emphasizes the same difference in order to be consistent with the difference perceived from the plots. For two curves a and b, such a score can be expressed as

$$R_{ab} = \frac{1}{n} \sum_{i=1}^{n} \log \frac{mr_i^a + \epsilon}{mr_i^b + \epsilon} \ , \tag{4.6}$$

where mr is a miss rate value, ϵ is a small regularization constant, and the sum is over the points of the DET curve. We use 10 as the logarithmic base and $\epsilon = 10^{-4}$ in our experiments. We found the value of ϵ not significant in comparing curves. If this score is positive, it indicates that curve a misses more on average, and vice versa.

Instead of having a score for each pair of curves, it is better to have a score

for each curve and compare the curves by comparing the scores. The score R in 4.6 can be expressed as

$$R_{ab} = \frac{1}{n}\sum_{i=1}^{n}\log\left(mr_i^a + \epsilon\right) - \frac{1}{n}\sum_{i=1}^{n}\log\left(mr_i^b + \epsilon\right) \ . \tag{4.7}$$

This suggests that we can represent the performance of each curve as the average of the logarithm of the miss rate values over the curve. But, this score will be always negative. Therefore, we switch its sign to reach the following expression for the ALMR (Average Log Miss Rate) score

$$ALMR = \frac{-1}{n}\sum_{i=1}^{n}\log\left(mr_i + \epsilon\right) \ . \tag{4.8}$$

The higher the value of the ALMR score, the lower the miss rate over the curve on average, *i.e.* the better. The ALMR score is related to the R score in 4.6 and 4.7 by

$$R_{ab} = ALMR_b - ALMR_a \ . \tag{4.9}$$

The ALMR is related to the geometric mean of the miss rate values. It is also proportional to the area under the curve in the log-log domain when the curve is approximated using a staircase plot. Since our plots are on a log-log scale and the points are uniformly spaced, the ALMR score contains more samples from the low false alarm rate values. This is useful since in many applications we are more interested in the low false alarm rate range.

Finally, in our evaluation, we call the difference between the ALMR scores of

two experiments *significant* when the confidence intervals of the two experiments do not overlap. Otherwise, we call the difference insignificant.

4.4 Evaluation Datasets

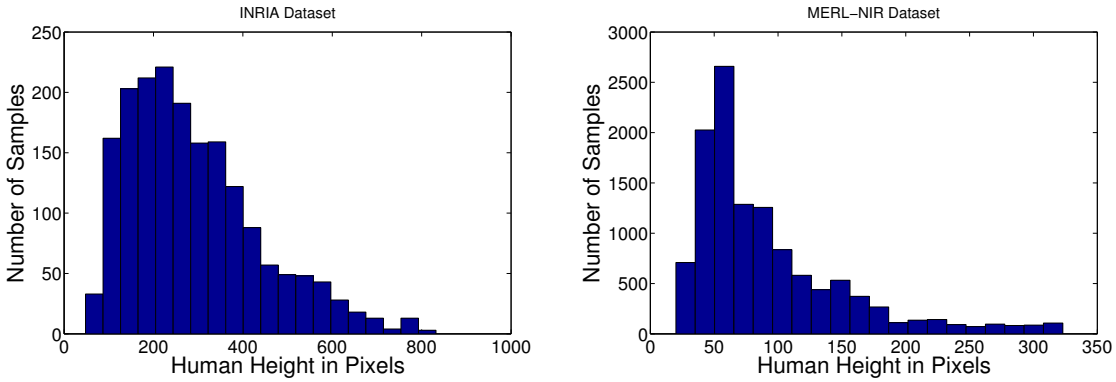
We evaluated the detectors on two different datasets, INRIA-Person and MERL-NIR. The INRIA dataset was introduced in Dalal and Triggs [20], and subsequently used to evaluate many human detectors. The MERL-NIR dataset consists of 46000 frames from a video sequence. The video was shot from a vehicle touring an Asian city, using a near infrared interlaced camera. From the frames that contained annotated human subjects, we uniformly sampled 1600 to be used as positive images. From the remaining frames, we randomly sampled 1100 to be used as negative images. The description of the two datasets along with statistics and histograms of human sizes are given in Table 4.1 and Figure 4.4. Sample whole images and cropped human windows used in training and testing are shown in Figure 4.5 and Figure 4.6. To conduct cross validation experiments, we divided the whole positive images in each dataset into 5 sets of a roughly equal number of annotated human subjects. We perform 10-fold cross validation by using 3 sets for training and 2 for testing in each fold. Negative images used in training and testing are common in all experiments. Table 4.2 describes the contents of each set and the number of negative images in the two dataset. The number of cropped windows in the table includes the left-right reflection of each window.

Table 4.1: A comparison between the two datasets used in our evaluation. Tracks are defined only in the case of MERL-NIR dataset. A track is a sequence of windows containing the same person in consecutive frames. More than one track can be associated with one person if she becomes partially or totally occluded and then fully visible again.

	INRIA	MERL-NIR
Electromagnetic Band	Visible	Near Infrared
Source of Images	Personal Photos	Interlaced Video Frames
Total Number of Images	2572	46000
Image Size	Variable	720×480
Number of Images Containing Humans	901	9823
Number of Human Samples	1825	11895
Number of Tracks	N/A	285
Min Person Height	48	20
Max Person Height	832	323
Mean of Person Height	290	92.66
Standard Deviation of Person Height	147.83	59.92
Median Person Height	260	72
Mode Person Height	208	50

Table 4.2: Division of each dataset into 5 positive subsets and two common negative sets for 10-fold cross validation experiments.

		INRIA Whole	INRIA Cropped	MERL-NIR Whole	MERL-NIR Cropped
Positive	Set # 1	179	730	320	766
	Set # 2	180	730	320	764
	Set # 3	180	730	320	764
	Set # 4	181	730	320	764
	Set # 5	181	730	320	764
Negative	Training	1218		800	
	Testing	453		300	



(a) INRIA Dataset (b) MERL-NIR Dataset

Figure 4.4: Distribution of human height in pixels in the two datasets used in our evaluation.

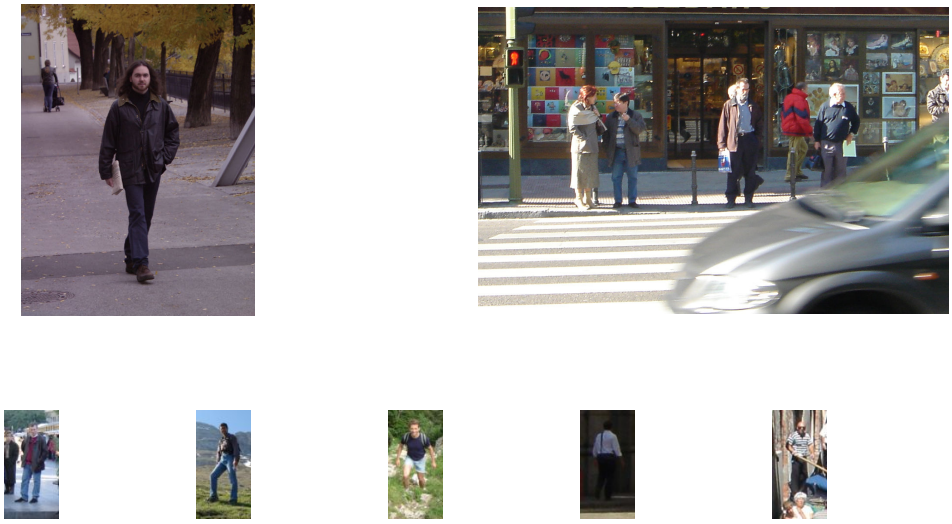


Figure 4.5: Sample whole and cropped human images from the INRIA-Person dataset.

Figure 4.6: Sample whole and cropped human images from the MERL-NIR dataset.

4.5 Evaluation Results

We train the cascade classifiers to have 30 cascade layers. Each layer is trained using the LogitBoost algorithm [36], and adjusted to produce 99.8% detection rate and 65% false alarm rate, using the algorithm in Viola and Jones [98]. The number of positive samples in each training session can be inferred from Table 4.2 by noting that we use three positive sets for training and the remaining two for testing in a 10-fold cross validation setup. The number of negative samples collected for each layer is set to 3.5 times the number of positive samples. Features are generated with the minimum side length set to 12.5% of the corresponding window side length, with a minimum of 8 pixels in order to have enough sample points to construct histograms and covariance matrices. The feature location stride and side length increment are

set to half the minimum feature side length. Every 5 boosting iterations, 5% of the features are randomly sampled, with a maximum of 200. The limit on the number of sampled features is for all descriptors to fit in memory instead of being re-computed on every boosting iteration.

For evaluation on whole images, each image is scanned with 9 window heights, starting from 75% of the training window height and using an increment of 30% of the last height used, while preserving the aspect ratio of the training window size. The scanning stride is set to 5% of the scanning window size in each dimension.

Our training and testing modules were run on a cluster of computers, with about 60 active nodes. Each node contained two Intel(R) Xeon(TM) CPU 3.06GHz processors with 512KB cache memory and 4GB RAM. The front end and compute OS was CentOS release 4.5.

In the remainder of this section, we first present the evaluation results on the INRIA dataset with the default training and testing window size of 128×64. Then, we present the results on the MERL-NIR dataset, in which we use a window size of 48×24. Alongside with this set of results, we present results for the INRIA dataset with window size 48×24 for the sake of comparison with the results on the MERL-NIR dataset. We present all the plots using the same limits in both axes for ease of comparison. In each plot, curves for the COV detector are drawn using dotted lines and curves for the HOG detector are drawn using dashed lines, with a different marker shape for each type of experiment. The legend of each experiment has two parts. The first is the descriptor, HOG or COV. The second is the evaluation method, which is either Cropped, Whole-RI, or Whole-RF, for

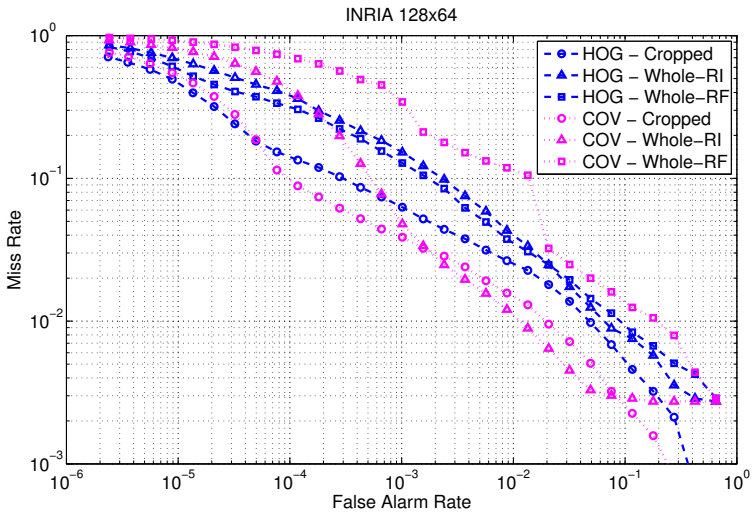


Figure 4.7: DET-Score plots for the INRIA dataset with window size 128×64.

cropped windows, whole images with resizing images, and whole images with resizing features, respectively.

4.5.1 Evaluation on INRIA 128×64

In this set of experiments, we evaluate our two detectors on the INRIA dataset using the original window size of 128×64, where each positive window is adjusted so that the height of the human body in it is 96 pixels.

Figure 4.7 shows the DET score plots for this set of experiments. Each curve is the average of the 10 curves produced by cross validation. However, the curves often intersect one another and there is no clear winner. Therefore, we will rely on the ALMR score statistics to compare experiments when it is hard to reach a conclusion by inspecting the curves.

Figure 4.8 shows the statistics of the ALMR score for each curve in Figure 4.7.

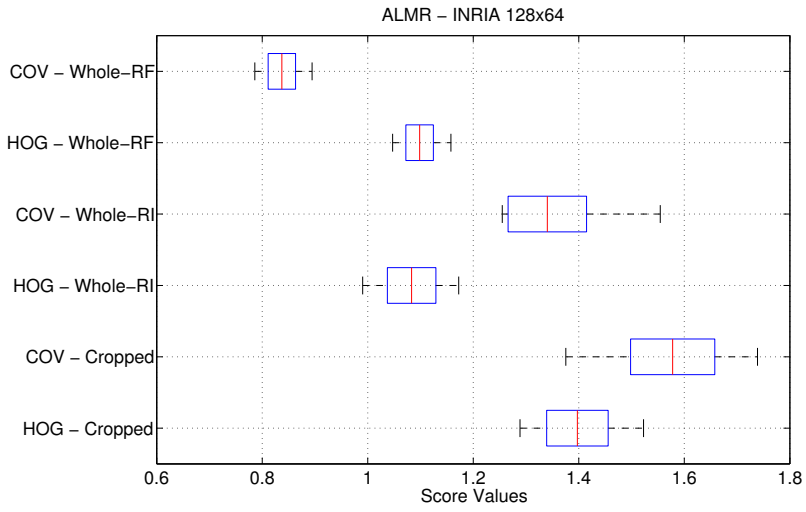


Figure 4.8: A box plot for the mean, confidence interval, and range of the ALMR score for the plots in Figure 4.7.

Note how comparing the mean values of the ALMR scores of two curves matches well with how the curves themselves compare to one another on average. The difference between the mean scores of two curves reflects the average relative advantage of one curve over the other in terms of miss rate. For example, the mean ALMR scores for the HOG-Cropped and COV-Cropped experiments are approximately 1.6 and 1.4, respectively. This means, on average, the miss rate of the HOG detector is $10^{0.2} \simeq 1.6$ times the miss rate of the COV detector, which is consistent with how the curves compare to one another.

For evaluation on cropped windows, the ALMR score shows the significant advantage of the COV detector on average. The confidence intervals of the two scores do not overlap. On average COV leads by around 0.2 points. Note how the ranges of the ALMR scores are large to the extent that they overlap. This signifies

the importance of using statistical analysis in order to have a reliable estimator for a detector's performance.

For evaluation on whole images, the COV detector maintains its lead over the HOG detector. The lead this time is even more evident since the ranges of the ALMR scores do not overlap. On average COV leads by around 0.2 points. However, the performance of the two detectors significantly deteriorates in this case by losing around 0.3 points on the ALMR scale on average. This deterioration signifies the importance of evaluation on whole images in order to predict the detector's performance in a typical practical setting.

Finally, for evaluation on whole images with resizing features, the picture is totally different. Without even inspecting the ALMR score statistics, we can notice that the HOG detector consistently outperforms the COV detector. By inspecting the ALMR scores, we notice that this difference is significant. On average HOG outperforms COV by around 2.5 points. The difference between the two detectors' behavior in this case may be due to the difference between the two descriptors, or due to the usage of learning on Riemannian manifolds in the case of COV. Further investigation is needed to understand this phenomenon. On the other hand, comparing evaluation on whole images for the HOG detector with resizing images and with resizing features, we find the difference between them insignificant. The mean score of each experiment lies in the confidence interval of the other. This gives the HOG detector a higher advantage over COV in terms of processing time. The COV detector is at least 10 times slower than the HOG detector. Resizing features saves about 40% of the processing time of the HOG detector without a significant loss

in detection performance. This makes the COV detector at least about 17 times slower than the HOG detector when resizing features is used for the latter.

Despite the advantage of the COV detector in most of the experiments on average, it is worth noting that the HOG detector often slightly outperforms the COV detector in the very low false alarm rate range, below around 10^{-4}. However, the points in this range of false alarm rates are often found only in the score-based plots and missing from the layer-based plots (compare Figure 4.7 to Figure 4.3). This may indicate the possibility of obtaining a more consistent advantage for the COV detector if we continue training more cascade layers to cover the entire range of false alarm rate. However, this is difficult in practice. It takes about 4 days to train a COV classifier for 30 layers. The bottleneck of the training process is finding enough miss classified negative samples for each new layer to be trained, and this time increases with the number of layers.

4.5.2 Evaluation on MERL-NIR

In this set of experiments, we evaluate our two detectors on the MERL-NIR dataset. Due to the smaller person heights in this dataset compared to the INRIA dataset, as shown in Figure 4.4, we have to use the reduced window size of 48×24 in this set of experiments. All positive windows are adjusted so that the height of the human body is 36 pixels. Because of this reduction in window size, we expect reduced detection performance.

Figures 4.9 and 4.10 show the DET plots and ALMR score statistics for this

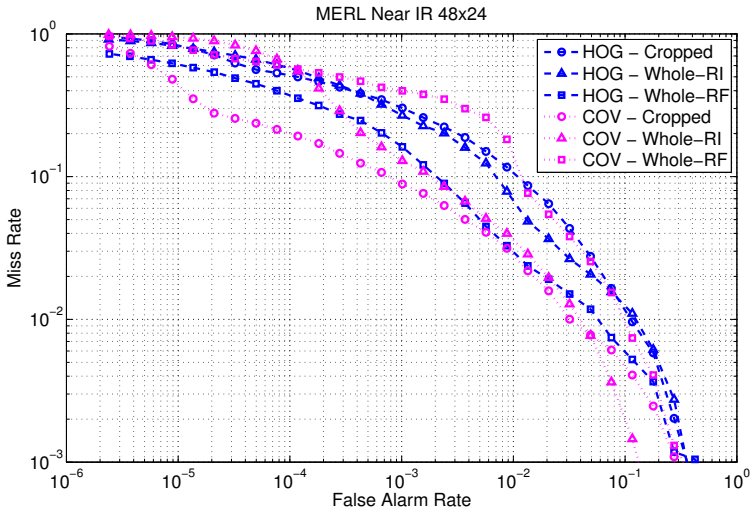


Figure 4.9: DET-Score plots for the MERL-NIR dataset.

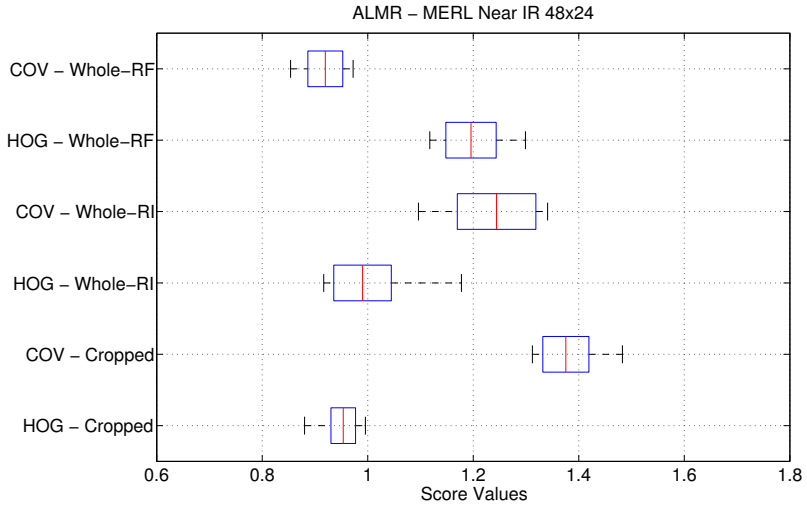


Figure 4.10: A box plot for the mean, confidence interval, min, and max of the ALMR score for the plots in Figure 4.9.

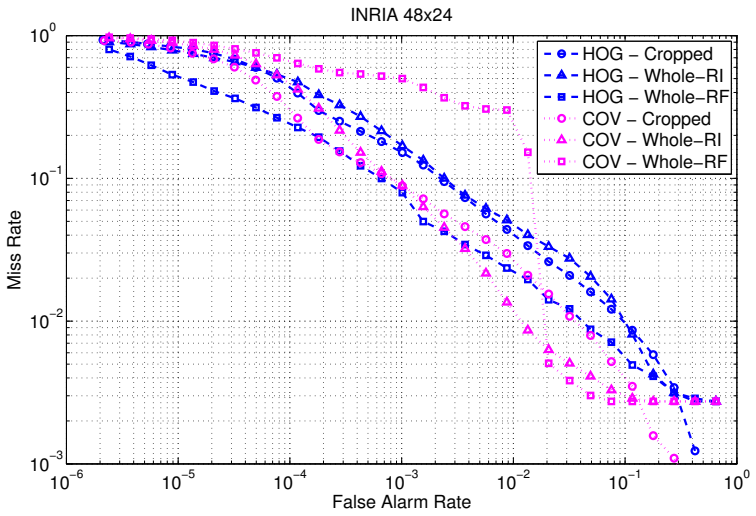


Figure 4.11: DET-Score plots for the INRIA dataset with window size 48×24.

set of experiments. Similar to the results on the INRIA 128×64 dataset, the COV detector's lead over the HOG detector in the case of cropped windows and whole images with resizing images, and the HOG detector's lead in the case of whole images with resizing features are significant. However, there are several differences between the two sets of results. The first notable difference is the improved performance for both detectors in the case of resizing features with respect to the other types of evaluation. In the case of HOG, using resizing features became even better than resizing images. The second notable difference is that the advantage of evaluation on cropped windows over evaluation on whole images with resizing images is no longer significant, with overlapping confidence intervals of the ALMR scores, and is reversed in the case of the HOG detector.

Before attempting to explain these differences, we present another set of results on the INRIA dataset, but, with the window size reduced to match the one used

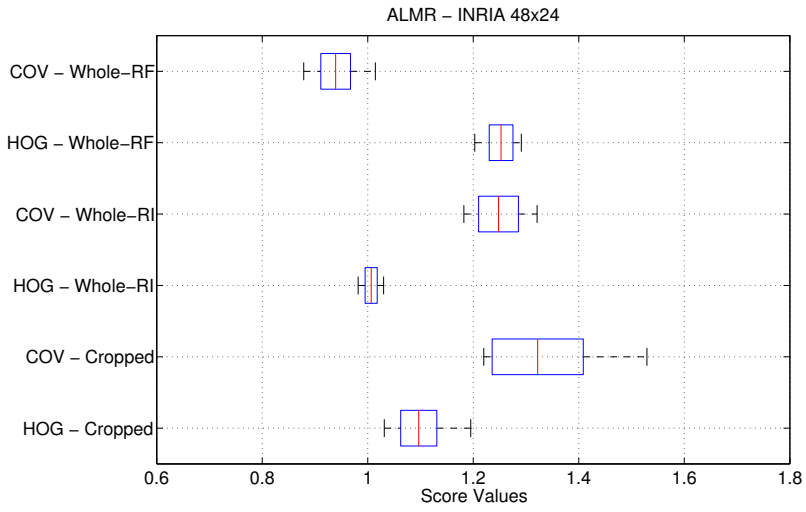


Figure 4.12: A box plot for the mean, confidence interval, min, and max of the ALMR score for the plots in Figure 4.11.

with MERL-NIR. In this set of experiments, all the INRIA dataset images used in training and testing are reduced in size with the same factor that reduces the window size of 128×64 to 48×24. Figures 4.11 and 4.12 show the results of this set of experiments. Comparing this set of results with those obtained on the MERL-NIR dataset, by comparing Figure 4.12 to Figure 4.10, we find that they are very similar. Most of the differences between them are either small or statistically insignificant. This observation gives us a clue about the differences between the results on the INRIA 128×64 dataset and those on the MERL-NIR dataset. It tells us that the difference is mostly due to the window size.

The reduced window size leads to a reduced stride when scanning whole images for evaluation since we set the stride to be 5% of the window side length. That makes the stride just 1 or 2 pixels in each dimension for a 48×24 window. Also, using a

reduced minimum scanning size results in a reduced scanning size range and hence a denser coverage of that range. These two factors could explain the reduction in the performance gap between the evaluation on cropped windows and evaluation on whole images. With reduced window sizes and window size range, there is a higher chance that the scanning window becomes close to annotated human subjects while having them centered. Also, with a smaller range of scanning window sizes, the effect of resizing features compared to resizing images should be less significant. Nevertheless, the enhanced performance of resizing features compared to resizing images in the case of HOG needs further investigation.

Finally, by comparing the ALMR scores in the case of evaluation on cropped images when using a large scan window size, Figure 4.8, versus using a small scan window size, Figures 4.10 and 4.12, we observe that the performance on small window sizes is significantly worse. Note that evaluation on cropped windows actually evaluates the classifier, not how it is used in the detection task. A classifier trained on a large window size has a richer set of features to select from. Therefore, it is expected to perform better, as the results show.

4.6 Conclusion and Future Work

We presented a comprehensive evaluation framework for object detectors that is geared towards a typical practical deployment paradigm. We demonstrated its utility on two state of the art human detection algorithms, that are based on cascade classifiers, on two different datasets, covering two bands of the electromagnetic

spectrum, visible and near infrared. In our evaluation we compare between the typically-used evaluation on cropped windows and the more practical evaluation on whole images. We introduced enhanced DET plot generation based on confidence scores instead of varying the number of layers in cascade classifiers. We introduced an aggregate performance score to summarize such plots for ease of comparison. We used 10-fold cross validation to statistically analyze our results.

Our experiments showed the effectiveness of our framework and led to the following findings:

- The COV detector maintains a significant lead over the HOG detector on average. However, sometimes it is very close or slightly inferior in the very low false alarm rate range, and it is at least 17 times slower.

- Application of detectors on whole images can yield a significant reduction in detection performance than what can be observed upon evaluation on cropped windows. However, when the application deploys a dense scanning in terms of strides and window sizes, the difference between them may not be significant.

- Detection performance may not be significantly affected by applying the same algorithm to images in the near infrared band instead of the visible band. However, it is significantly affected by the window size used in training the classifiers.

- Whether to use resizing images, or resizing features, when applying a detector to whole images, can have a significant effect on the detection performance

depending on the detector used. While the HOG detector can deliver the same or better performance when resizing features, the COV detector delivers significantly deteriorated performance.

Many directions can be taken for future extensions and enhancements of our framework. It is not clear how the extended plots we obtain for cascade classifiers using confidence scores are comparable to plots obtained by increasing the number of layers in the cascades. The ALMR aggregate confidence score gives an overall performance measure assuming that performance over the entire range of the false alarm rate is important. An investigation of using a weighted or limited-range version of the score for some applications can be useful. Comparison to PR curves and what we learn from both DET and PR curves on evaluation on whole images needs to be further studied. Finally, the framework in general needs to be applied to other state of the art detectors, especially ones that do not rely on cascade classifiers.

Part II

Visual Computing on GPUs

Chapter 5

Introduction to Part II: Visual Computing on GPUs

In recent years, Graphics Processing Units (GPUs) have been rapidly advancing from being specialized fixed-function processors to being highly programmable and parallel computing devices. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphics APIs. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Threads (SIMT) multi-core processors. The number of threads that can be executed in parallel on such devices is currently on the order of hundreds and is expected to grow further. Many applications that are not yet able to achieve satisfactory performance on CPUs can benefit from the massive parallelism provided by such devices.

In this part, we explore the opportunity of developing very fast implementations of core algorithms used in visual object recognition on GPUs. First, we present an implementation of the Graph Cut algorithm on GPUs. Second, we present an efficient band approximation of Gram matrices that can be used to speed implementations of kernel-based methods on GPUs. As an application to the latter, we present an implementation of the Affinity Propagation algorithm.

In this chapter, we first give a brief introduction to CUDA. Then, we introduce our work on Graph Cut, and kernel-based methods in Sections 5.2 and 5.3, which

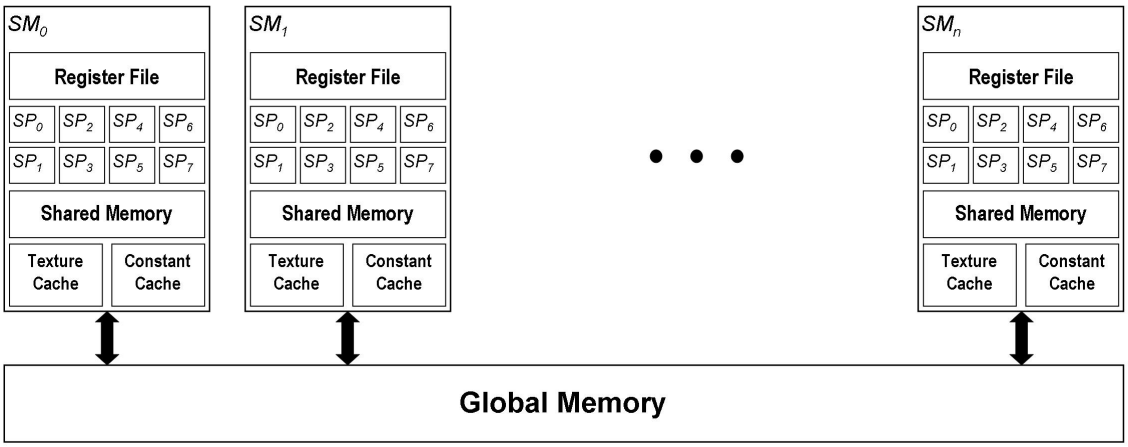


Figure 5.1: CUDA Architecture.

are later explained in detail in Chapters 6 and 7, respectively.

5.1 Compute Unified Device Architecture

We briefly present the main features of the Compute Unified Device Architecture (CUDA), which is the main stream architecture/model used for general purpose computing on GPUs nowadays. For a detailed description, the reader is referred to Nickolls *et al.* [68] and NVIDIA CUDA Programming Guide [69].

5.1.1 Architecture

In CUDA, a parallel compute device, such as the GPU, is referred to as a *device*. A CUDA device is responsible of running CUDA *kernels* in parallel. A CUDA kernel is a C function which specifies the operation performed by a single *thread* of execution. Launching CUDA kernels and controlling the path of execution from one kernel to

the next are performed by a separate serial processor, such as the CPU, referred to as a *host*.

Figure 5.1 is an illustration of CUDA's device architecture. A CUDA device consists of a number of *Streaming Multiprocessors* (SMs) that ranges from 2 to 30 in the high-end models. Each SM consists of 8 core *Streaming Processors* (SPs). Each SP has exclusive access to a designated number of registers in its SM's register file. All SP's in the same SM have access to a low latency *shared memory* space. The shared memory is organized in *banks* so that each bank can serve one memory access at a time. All SPs in all SMs have access to three common memory spaces, which are

1. *Global Memory:* A read/write non-cached memory space.

2. *Constant Memory:* A read-only cached memory space.

3. *Texture Memory:* A read-only cached memory space with hardware support for filtering operations and memory access modes needed for texture fetching.

Accessing constant and texture memory spaces is as fast as accessing local registers on cache hits. Accessing shared memory is as fast as accessing registers if there is no memory bank conflict, *i.e.* if no two SPs access two different locations within the same shared memory bank. On the other hand, accessing global memory is typically up to two orders of magnitude slower. In fact, accessing global memory is also two orders of magnitude slower than floating point multiply and add.

5.1.2 Execution Model

The execution of CUDA kernels follows a Single-Instruction Multiple-Thread (SIMT) model. Each thread executes a CUDA kernel on a single SP. Threads are virtually organized in a three dimensional discrete space referred to as a *grid*. This space is further divided into equally sized rectangular boxes called *blocks*. The number of threads and dimensionality of the thread blocks and grid are specified by the programmer depending on the operation to be performed and the size and dimensionality of the input data. All threads in the same block execute on SPs of the same SM. Therefore, threads within a block can communicate with one another through the shared memory space in the assigned SM. Threads in the same block can also use efficient barrier synchronization to coordinate their executions. The execution unit of the SM executes threads in parallel in groups of 32, called *warps*. Threads within the same warp need to follow the same execution path to obtain the maximum possible performance. Otherwise, divergent execution paths within a warp are serialized. Different warps are run in parallel by an SM in a time slicing fashion.

5.1.3 Performance Considerations

There are several important considerations that must be taken into account in order to maximally exploit the computational power of CUDA device. The most important of these considerations is optimizing global memory accesses. As noted earlier, accessing global memory is significantly slower than accessing other memory spaces

and than compute instructions. One way to alleviate this overhead is through sharing data loaded from global memory among threads by using the shared memory space. A significant gain can be achieved also by considering how accesses to global memory are realized by the hardware. If data is organized in a simple array structure so that each element is either 4, 8, or 16 bytes long, and threads within the same warp access consecutive data elements, these accesses are grouped (coalesced) in at most 2 memory access instructions in the latest Tesla devices. On the other hand, accessing random array elements by threads in the same warp can lead to launching a memory access instruction for each thread, which leads to a significant slow down. Other performance considerations include fine grain parallelism, and minimizing thread divergence and shared memory bank conflicts.

Taking these constraints into account is the key to achieving good performance on CUDA devices. As you will see, using a data structure that can easily be accessed in a way that respects the global memory access rules can achieve significant speedups.

5.2 Graph Cut on GPUs

The Graph Cut algorithm is a fundamental graph algorithm thereof some other graph algorithms can be modeled as special cases, such as shortest paths, and bipartite graph matching. It has many applications in computer vision, such as foreground/background segmentation [9, 77], image restoration [11], stereo matching [78], and multi-camera scene reconstruction [54].

In Chapter 6, we present several parallel implementations of the Graph Cut algorithm on the Compute Unified Device Architecture (CUDA). We evaluate our implementations on an image-segmentation task. Our general implementation achieves a consistent speedup compared to the best CPU implementation. We introduce two optimizations that utilize the special structure of grid graphs, which are prevalent in imaging applications of the algorithm. The first is *lockstep BFS*, which reduces the overhead of BFS traversal, a major component in the algorithm. The second is *cache emulation*, which regularizes the memory access pattern and thereby enhances the memory access throughput. Each of the two optimizations provides a substantial speedup over the general implementation. Using cache emulation is consistently the best. A version of this work appeared in our paper [51].

5.3 Band Approximation of Gram Matrices

Kernel-based learning methods require $O(N^2)$ space and computational complexities for computing the kernel (*i.e.* Gram) matrix, for N data points. These complexities significantly impact the application of kernel methods to large scale problems with millions of data points. In Chapter 7, we introduce a novel method to approximate a Gram matrix with a band matrix. Our method relies on the locality preserving properties of space filling curves, and the special structure of Gram matrices. Our approach has several important merits. First, it computes only those elements of the Gram matrix that lie within the projected band. Second, it is simple to parallelize. Third, using the special band matrix structure makes it space efficient and

GPU-friendly. We developed GPU implementations for the Affinity Propagation (AP) clustering algorithm using both our method and the COO sparse representation. Our band approximation is about 5 times more space efficient than COO. AP gains up to 6x speedup using our method with small degradation in its clustering performance. A version of this work appeared in our papers [46, 45].

Chapter 6

Graph Cut on GPUs

In this chapter, we present our results on implementing the Graph Cut algorithm on CUDA. Our primary focus is on implementing Graph Cut on grid graphs, which are extensively used in imaging applications. We explain our implementation of breadth first search (BFS) graph traversal on CUDA, which is extensively used in our Graph Cut implementation. We then present a basic implementation of Graph Cut that succeeds to achieve absolute and relative speedups when used for foreground-background segmentation on synthesized images. Finally, we introduce two optimizations that utilize the special structure of grid graphs. The first one is *lockstep BFS*, which is used to reduce the overhead of BFS traversals. The second is *cache emulation*, which is a general technique to regularize memory access patterns and hence enhance memory access throughput. We experimentally show how each of the two optimizations can enhance the performance of the basic implementation on the image segmentation application.

6.1 Introduction

The Graphc Cut problem is know in combinatoric optimization as Max-Flow/Min-Cut. The problem is defined as follows.

Let $\mathcal{G}$ be a graph $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of nodes, and $\mathcal{E}$ is a set of links. Let

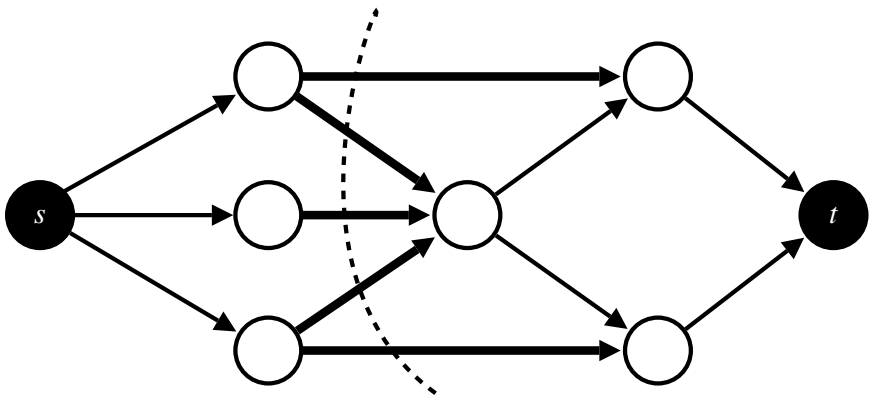


Figure 6.1: An example of an s/t cut in a graph. Terminal nodes s and t are marked in black. The cost of the cut is the sum of the weights of the thick links, which are the links that connect nodes in $\mathcal{S}$ to nodes in $\mathcal{T}$.

s and t be two designated terminal nodes in $\mathcal{V}$. An s/t cut in $\mathcal{G}$ is a partitioning of $\mathcal{V}$ into two disjoint subsets $\mathcal{S}$ and $\mathcal{T}$ such that $s \in \mathcal{S}$ and $t \in \mathcal{T}$. Figure 6.1 shows an example of an s/t cut of a sample graph. Let $w(u, v)$ be a cost function that assigns a real value to every link $(u, v) \in \mathcal{E}$. The cost of a cut $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ is defined as $c(\mathcal{S}, \mathcal{T}) = \sum_{u \in \mathcal{S}, v \in \mathcal{T}} w(u, v)$, which is basically the sum of the costs of all links linking a node in $\mathcal{S}$ to a node in $\mathcal{T}$. In Figure 6.1, the cost of the shown cut is the sum of the costs of the thick links. The Graph Cut algorithm finds the minimum cut in a graph, which is a cut with a minimum cost value. A cut in the graph defines a binary labeling over its nodes. If we are not interested in the cost of the minimum cut itself and interested instead in the best binary labeling of graph nodes according to some energy function, it can be shown that Graph Cut can be used to exactly minimize a wide class of functions of binary variables, and approximately minimize a wide class of functions of discrete variables in general [55].

The Graph Cut algorithm has many applications in different areas of research. It is a fundamental graph algorithm thereof some other graph algorithms can be modeled as special cases, such as shortest paths, and bipartite graph matching. However, our focus is on applications related to computer vision, computer graphics, and machine learning. For example, in computer vision, binary labeling via Graph Cut was applied in foreground/background segmentation [9, 77], where labels are either foreground or background. Discrete labeling was applied in many other applications such as image restoration [11], where labels are discrete intensity values, stereo matching [78], where labels are discrete disparity values, and multi-camera scene reconstruction [54], where labels correspond to different scene elements. In computer graphics there are many applications as well. In Agarwala *et al*. [2], Graph Cut was applied to interactive PhotoMontage, where different images for the same scene can be combined to form a better image based on interactively determined user's criteria. In Kwatra *et al*. [56], a method for texture synthesis using Graph Cut was proposed. Wu and Yang [104] proposed a method for labeling objects of interest in images, called SmartLabel, based on Graph Cut. An example application of Graph Cut in machine learning is in Blum and Chawla [8], where a method for labeling a large unlabeled dataset based on a small labeled one, via Graph Cut, was proposed. Among these many applications, we selected image segmentation [9] to evaluate our implementation on. That is primarily due to the simplicity of its implementation as well as non-reliance on field specific concepts. However, it is important to emphasis that our implementation of Graph Cut is general and is not targeted to any particular application.

While fast practical parallel implementations for Graph Cut have already been accomplished before [4, 3], to the best of our knowledge, our implementation is the first implementation on CUDA. It is also the first implementation on graphics processors that succeeds to achieve relative and absolute speedups for the type of graphs targeted in this work. An implementation on CUDA is particularly important due to the relatively low cost of CUDA-enabled devices. Moreover, many of the applications of Graph Cut, as mentioned above, are targeted to desktop applications, where assuming existence of many CPUs or clusters of computers is not practical, while existence of a CUDA-enabled device is almost guaranteed. Also, some of the applications of Graph Cut, such as in photo-editing [2, 77], produce visual outputs to be displayed to the user. Therefore, it is more efficient to perform the processing on the display card instead of transferring data back and forth between the GPU and the CPU.

The simplicity of CUDA's programming model, in fact, projects a number of challenges on implementing Graph Cut on it compared to the other architectures on which Graph Cut implementations were studied before. For example, mechanisms for memory locking, to prevent concurrent updates to the same memory location are not available on all devices. Also, multiprocessors in CUDA work in a SIMD fashion, where best performance is achieved when all core processors perform the same operation at the same time. Divergence among core processors in the same multiprocessor results in serialization of the different paths taken and hence can cause a large performance penalty.

Our implementation addresses these issues by taking a rather unusual way of

implementing Graph Cut in parallel. Our algorithm is a push-relabel style algorithm [40]. Instead of implementing the global relabeling heuristic in parallel with push and relabel operations [4, 3], we exclusively rely on BFS graph traversals to assign optimal labels to graph nodes, as explained in Section 6.4.

Although an implementation of Graph Cut on general graphs is our ultimate goal, in this work we focus only on grid graphs. In fact, implementing graph algorithms, such as Graph Cut, where the complexity of processing a node in the graph is a function of its degree, is not straight forward on SIMD architectures, such as CUDA, where divergence among different processors has to be avoided as much as possible. Grid graphs have the attractive property of having a constant out-degree for almost all nodes in the graph. Therefore, the number of operations performed when processing a node in the graph is generally the same as processing any other node, which almost eliminates divergence. Moreover, the special structure of grid graphs allows us to apply two optimization techniques. The first technique is the *lockstep BFS*, which is utilized to mitigate the overhead of our CUDA implementation of BFS traversal. The second is *cache emulation*, which is a general technique to regularize memory access patterns. Restricting our implementation to grid graphs by no means nullifies its utility. Almost all the applications of Graph Cut mentioned above, and many others, work on grid graphs.

The rest of the chapter is organized as follows. Section 6.2 summarizes related work. Section 6.3 explains our CUDA implementation of BFS graph traversal. Section 6.4 presents our approach for computing Graph Cut and its parallel implementation on CUDA. Then, Section 6.5 introduces performance improvements that

utilize the special structure of grid graphs. Section 6.6 contains performance results. Finally, the chapter is concluded and future directions are outlined in Section 6.7

6.2 Related Work

In Shiloach and Vishkin [88], the first parallel algorithm for Graph Cut was introduced. It was based on the augmenting paths approach. As many other researchers, our implementation is primarily based on the push relabel approach since it is more natural to implement in parallel. In Goldberg [39], the first parallel algorithm based on push-relabel techniques was introduced and was implemented on a connection machine. The first parallel implementation on a shared memory architecture was introduced in Anderson and Setubal [3], where they performed global relabeling concurrently with the main push and relabel operations. Recently, an extended and enhanced version of the same approach was implemented on a modern SMP architecture [4]. These implementations differ significantly from ours since they assume availability of a memory locking mechanism and assume asynchronous operation of different processors. In Dixit *et al.* [25], the Graph Cut algorithm was implemented on older GPUs. However, it was much slower than the CPU implementation.

After our work, Vineet and Narayanan [96] introduced another implementation on CUDA, which is reportedly much faster than our best implementation. The main difference between our work and theirs is that they do not use graph traversal and they perform both local and global relabeling. We think the simplicity of their implementation compared to ours is the key to the performance they obtain.

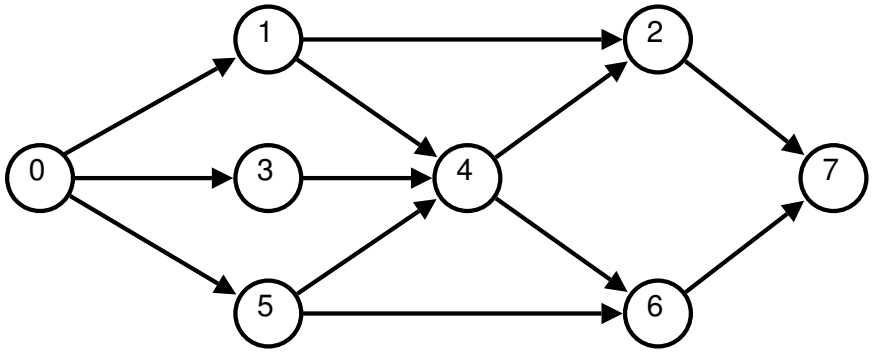


Figure 6.2: An example graph.

6.3 Parallel BFS Graph Traversal on CUDA

Let $\mathcal{G}$ be a graph $\langle \mathcal{V}, \mathcal{E} \rangle$. In breadth first search graph traversal, we start from a designated node in the graph $s \in \mathcal{V}$. We visit all nodes at a specific depth level from s before visiting any node in the next depth level. The output of the algorithm is a label for each node reachable from s, that specifies its depth level with respect to s. In our implementation, to visit nodes at depth level $k+1$, we start with a list of nodes at depth level k. All nodes at level k are processed in parallel. In processing each node, all its neighbors that have not been visited yet are marked to be added to level $k+1$. After finishing this process, we end up having an array of flags each element therein indicates whether the corresponding node in the graph is added to level $k+1$ or not. The size of this array is n, the number of nodes in the graph. To compact this list of flags and construct the list of nodes for level $k+1$, we use a parallel prefix sum operation [7]. Figure 6.3 illustrates this operation for traversing depth level 2 of the graph in Figure 6.2

This approach is not work efficient since the work complexity of visiting one

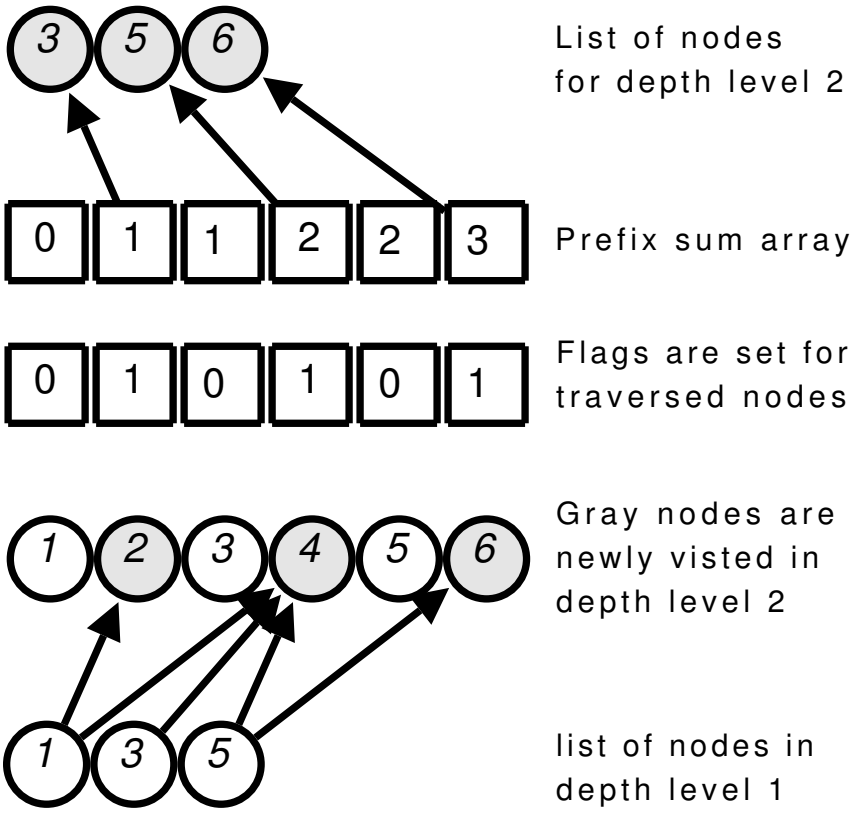


Figure 6.3: Traversal of depth level 2 for the graph in Figure 6.2 if traversal starts from node 0: creation of a list of nodes visited in depth level 2 given the list of nodes of depth level 1. Arrows at the bottom indicate traversals. Arrows at the top indicate moving nodes to their positions in the new list. Note that prefix sum values for traversed nodes correspond to their locations in the newly created list.

level of the graph is $\Theta(n)$, which is the complexity of performing the prefix sum operation on n elements. That makes the overall complexity of the BFS traversal $O(n^2)$. In fact, it may be much more efficient to implement the BFS traversal on the host (the CPU) instead of performing it inefficiently on the device. But, the overhead of transferring the results from the host to the device may be much more significant than the overhead of performing the traversal on the device. Therefore, we selected to implement the traversal on the device. In Section 6.5, we introduce an optimization for grid graphs that reduces the overall complexity of the traversal to $\Theta(n)$.

6.4 Parallel Graph Cut on CUDA

We first give an algorithmic background to Graph Cut and then explain our approach to implement the algorithm in parallel on CUDA.

6.4.1 Background on Graph Cut Algorithms

In a fundamental theorem in graph theory, Ford and Fulkerson [33] proved the duality between finding the maximum flow that can be pushed from a source node s to a target node t in a graph, and finding the minimum s/t cut in that graph. Based on this theorem, algorithms for solving the Min Cut problem typically do that by solving the dual problem, the Max Flow problem. In the Max Flow terminology, the cost of a link w is referred to as its *capacity*.

There are two main approaches to finding the maximum flow in a network,

the *augmenting paths* approach [33], and the *push-relabel* approach [40]. We briefly explain the basic idea of the two approaches. We elaborate more on the push relabel approach since we believe it is more appropriate to implement in parallel, and our algorithm is based on it. In both approaches, a residual graph is constructed and used throughout the algorithm. A residual graph $\mathcal{G}_f$ of a graph $\mathcal{G}$ is a graph that has the same layout as $\mathcal{G}$, but the capacities of its links are *residual capacities*. The residual capacity $w_f(u, v)$ of a link (u, v) after pushing flow $f(u, v)$ through it is $w(u, v) - f(u, v)$, where $w(u, v)$ is the capacity of the link (u, v).

6.4.1.1 Augmenting Paths

An augmenting-paths style algorithm tries to find a path from the source to the target in the residual graph and then pushes the maximum possible flow through that path. The algorithm continues until no path remains from the source to the target. The differences between augmenting paths algorithms lie mainly in the way they select the path through which to push.

6.4.1.2 Push Relabel

A push-relabel style algorithm assigns to each node in the graph an *excess* value and a *label*. The excess of a node is the total amount of flow it received from its neighbors minus the total amount of flow it sent to its neighbors. Initially all nodes have excess 0 except for those nodes that have links coming from the source. Each of the latter nodes initially has excess equal to the capacity of the link coming from

the source. The label of a node is a non-negative integer that underestimates the node's distance to the target (in terms of link count.) Initially all nodes have label 0, except for s that is given label n, where n is the number of nodes in the graph. The algorithm alternates between two operations, push and relabel:

- *Push:* The push operation applies to a node u in the graph if u has positive excess. If u has label k, the push operation finds a neighbor v of u such that v has label $k-1$ and the link (u, v) has positive residual capacity. If such a node exists, the maximum possible flow is pushed from u to v. That push results in updating the excesses of u and v as well as the residual capacities of the links (u, v) and (v, u). After a push operation, either u loses all its excess or the link (u, v) is saturated. The criterion of selecting v can be understood based on the interpretation of a node's label as an estimate of the distance to the target. The push operation basically tries to push flow towards the target through a node that is one step closer. It does that relying only on local information of a node and its neighbors.

- *Relabel:* The relabel operation applies to a node u if u has positive excess and has outgoing links but a push operation does not apply. That happens when all outgoing links from u are towards nodes with labels greater than or equal to that of u. The relabel operation tries to enable u to eliminate its excess by increasing its label to the minimum possible value that makes a push operation applicable.

The algorithm can apply push and relabel operations in any order until none of them applies, which happens when all nodes have 0 excess. It is guaranteed that either of the two operations must apply for a node with positive excess [17]. Upon termination, all extra excess that was initially pushed from the source to its neighbors and did not find its way to the target will have been pushed back to the source. The algorithm is guaranteed to terminate in $O(mn^2)$ time regardless of the order in which push and relabel operations are applied, where n is the number of vertices and m is the number of links in the graph. However, it turns out that the order of such operations has a great impact on the performance of the algorithm in practice. Actually, the differences among push relabel methods lie mostly in the way this order is determined and the way nodes are labeled.

6.4.2 Our Approach to Implementing Graph Cut on CUDA

One might think that a parallel implementation can process all nodes in parallel, and for each node with positive excess if a push operation applies it is performed, otherwise a relabel is performed. However, on CUDA, we would like to make all processors perform the same operation at the same time to avoid divergence. Also, a node cannot push flow and receive flow pushed to it at the same time since both operations update its excess value. Since, we do not assume any memory locking mechanism, we have to find a different way to prevent concurrent updates to the same value. In the following we explain how the push and relabel operations can be adapted to overcome these difficulties.

6.4.2.1 Parallel Labeling

As explained above, the label of a node is an underestimate of its distance to the target. But, in practice, relying only on the basic relabel operation results in very poor estimates and makes flow go back and forth between nodes many times before eventually reaching the target. That dramatically slows down the algorithm. A heuristic that was proposed to enhance the running time is *global relabeling* [40]. In this heuristic, the algorithm is frequently stopped and all nodes are labeled with their actual distances to the target. This is accomplished using a backward breadth first search traversal starting from the target. In our proposed algorithm, we use this heuristic as the only labeling scheme. In other words, this is the only way nodes get their labels. We employ the BFS traversal technique, explained in Section 6.3. The traversal in this case starts from the target, and goes backwards in the graph. In this way, all nodes are optimally labeled in parallel without introducing expensive divergence among processors. Note that applying global relabeling on every iteration eliminates the need for the *gap relabeling* heuristic that was suggested to be combined with global relabeling in Cherkassky and Goldberg [15].

6.4.2.2 Parallel Pushing

The order of applying the push operations also impacts the performance of the algorithm. A heuristic that is used to enhance the performance is to apply push to nodes with higher labels before nodes with lower labels [14]. We apply this heuristic in our implementation. During the labeling phase, we store the nodes that are

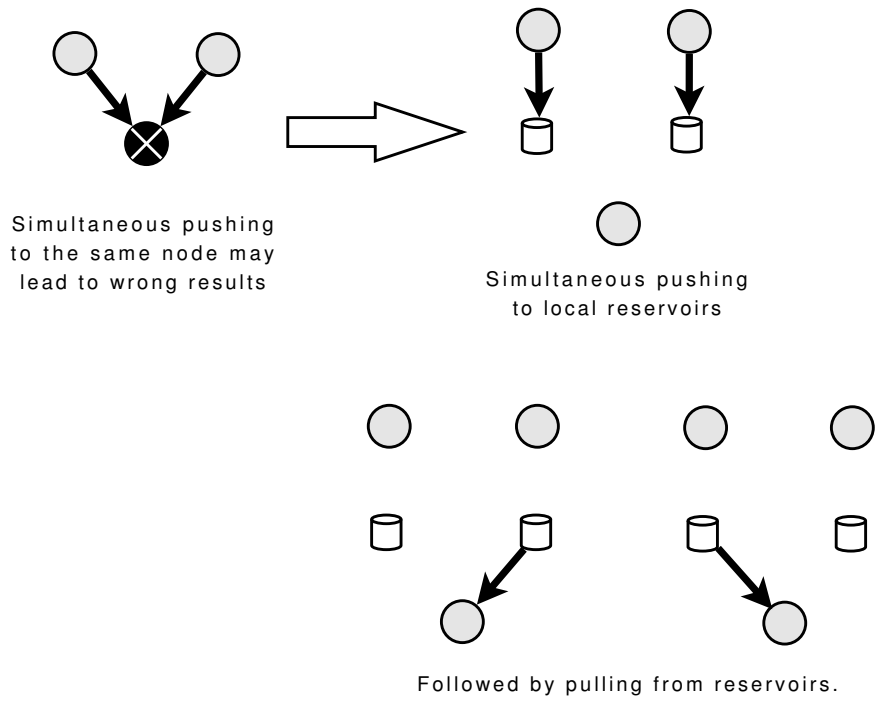


Figure 6.4: Dividing parallel pushing into two steps.

visited at each depth level in a separate list. This comes almost for free because of the way the breadth first search technique works (Section 6.3). We call the resulting structure the traversal *lattice*. During pushing, we start from the top level of the lattice going downwards to the target. At each level pushing is done in parallel. All nodes in the current level in parallel push flow to their neighbors in the lower level.

Unfortunately, pushing in parallel in this way may not produce correct results. A node cannot receive flow from more than one neighboring node simultaneously. To resolve this problem, we divide the push operation into two phases, *push* and *pull*. In the push phase no node updates the excess of its neighbor. Instead, each node keeps a reservoir for each outgoing link in which it stores the amount of flow it

```
procedure ParallelPush(LabelsLattice)
    EmptyReservoirs
                                        ▷ initializes all node reservoirs to 0
    Level ← PopTopLevel(LabelsLattice)
    ParallelPushToLowerLevel(Level)
    while NumLevels(LabelsLattice) > 1 do
        Level ← PopTopLevel(LabelsLattice)
        ParallelPullFromUpperLevel(Level)
        ParallelPushToLowerLevel(Level)
    end while
    Level ← PopTopLevel(LabelsLattice)
    ParallelPullFromUpperLevel(Level)
    ParallelPushToTarget(Level)
end procedure
```

Figure 6.5: Pseudo-code for the parallel push operation.

pushes on that link in the current pushing phase. In the pull phase, all nodes at a given depth level in parallel collect flow pushed to them from their neighbors in the upper level by reading the values stored in the appropriate reservoirs. Each node then updates its excess value and residual capacities accordingly. Figure 6.4 clarifies this operation. The final parallel push algorithm is depicted in the pseudo-code in Figure 6.5.

6.4.2.3 Termination Criteria

After pushing flow all the way from the top level of the traversal lattice down to the target, the layout of the graph changes due to saturation of some links. So, the lattice has to be rebuilt before the next pushing phase. Therefore, the whole algorithm works by alternating between parallel labeling and parallel pushing phases. But, when should the algorithm be terminated? There are two conditions; reaching either of them causes the algorithm to terminate.

- *Failure to Construct the Lattice:* That happens when all links to the target are saturated. In this case, the resulting cut is $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ such that $\mathcal{T} = t$ and $\mathcal{S} = \mathcal{V} - \{t\}$.

- *No Excess in Lattice:* In this condition, all nodes in the lattice, *i.e.* all nodes having at least a path to the target, have no excess. In this case, there is no flow to push down to the target. The cut in such a case is defined as $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ such that $\mathcal{T} = \{u \in \mathcal{V}, u \text{ has a path to } t\}$ and $\mathcal{S} = \mathcal{V} - \mathcal{T}$.

The entire algorithm is depicted in the pseudo-code in Figure 6.6. The pseudo-function ParallelBFS performs the BFS traversal and all proper initializations needed for it. The variable $ExcessFlag$ represents whether there is positive excess at any node in the traversal lattice or not. It is set during the parallel labeling phase.

```
procedure ParallelGraphCut
    InitNodeLabels                          ▷ initializes all node labels to 0
    ExcessFlag ← 0
    LabelsLattice ← ParallelBFS
    while ExcessFlag = 1 do
        ParallelPush(LabelsLattice)
        InitNodeLabels
        ExcessFlag ← 0
        LabelsLattice ← ParallelBFS
    end while
end procedure
```

Figure 6.6: Pseudo-code for the parallel Graph Cut algorithm.

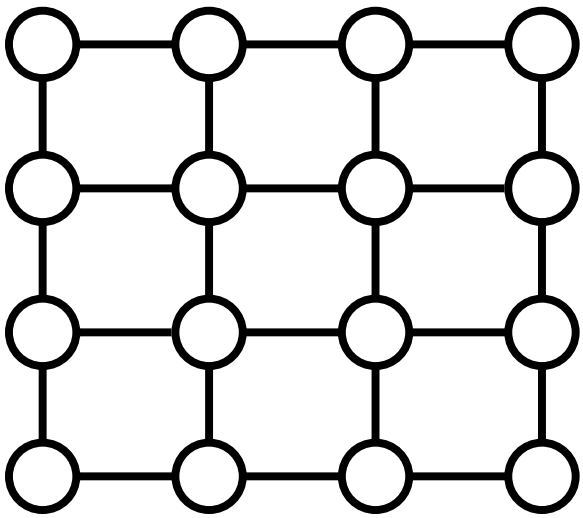


Figure 6.7: An example of a grid graph.

6.5 Optimizations for Grid Graphs

There are two main issues with the aforementioned algorithm:

- *Memory Access Pattern:* Either in labeling (building the lattice) or pushing, threads do not share data read from global memory. That is because two adjacent threads in a block could generally be processing two nodes in the graph that have no neighbors in common.

- *Prefix Sum:* During constructing the lattice, in the labeling stage, prefix sum operations are performed over the entire set of nodes regardless of the number of nodes actually visited.

We propose two approaches to alleviate the effect of these problems by utilizing the special structure of grid graphs. For simplicity of presentation, we focus on two dimensional grids. The concept can easily be extended to higher dimensional grids. A two dimensional grid graph can be viewed as a matrix of nodes, where each node can be uniquely identified by a two dimensional index specifying its row and column in the graph. A general node in the graph has links with nodes only within a fixed neighborhood surrounding it in the grid. Figure 6.7 shows a sample 4×4 grid graph. In the following two sections, we explain the two proposed optimizations, *lockstep BFS traversal*, and *cache emulation*.

6.5.1 Lockstep BFS Traversal

The technique for BFS graph traversal explained in Section 6.3 utilizes an array of flags of length n, and performs prefix sum operation on it on traversing each depth level of the graph. That is important for general graphs, where the number of nodes traversed at each depth level is arbitrary and does not depend on the number of nodes in the preceding level. Also, each node can be traversed from several neighbors at the same time. Therefore, it is important to keep a unique flag for each and every node in the graph for the operation to produce correct results.

In grid graphs, on the other hand, each node has a fixed number of neighbors. Therefore, the number of nodes traversed at each depth level is at most a constant multiple of the number of nodes traversed at the preceding level. Moreover, if we represent the graph in a way where links to neighboring nodes have a fixed order based on their directions (*e.g.* left, right, top, and then bottom for a 4-connected neighborhood graph,) and during graph traversal only a single direction is traversed at a time, then the number of nodes traversed at a certain direction at a certain depth level is at most the same as the number of nodes traversed at the preceding depth level. Also, performing traversal in this way guarantees that a node is traversed-to from exactly one neighbor. This paradigm of traversal in which only one direction is traversed at a time is what we call the *lockstep BFS* traversal technique. For example, in a 4-connected neighborhood graph, constructing a given depth level is divided into four steps instead of being done all in one step. In each step neighbors along one direction are traversed. Figure 6.8 gives an illustration of this operation.

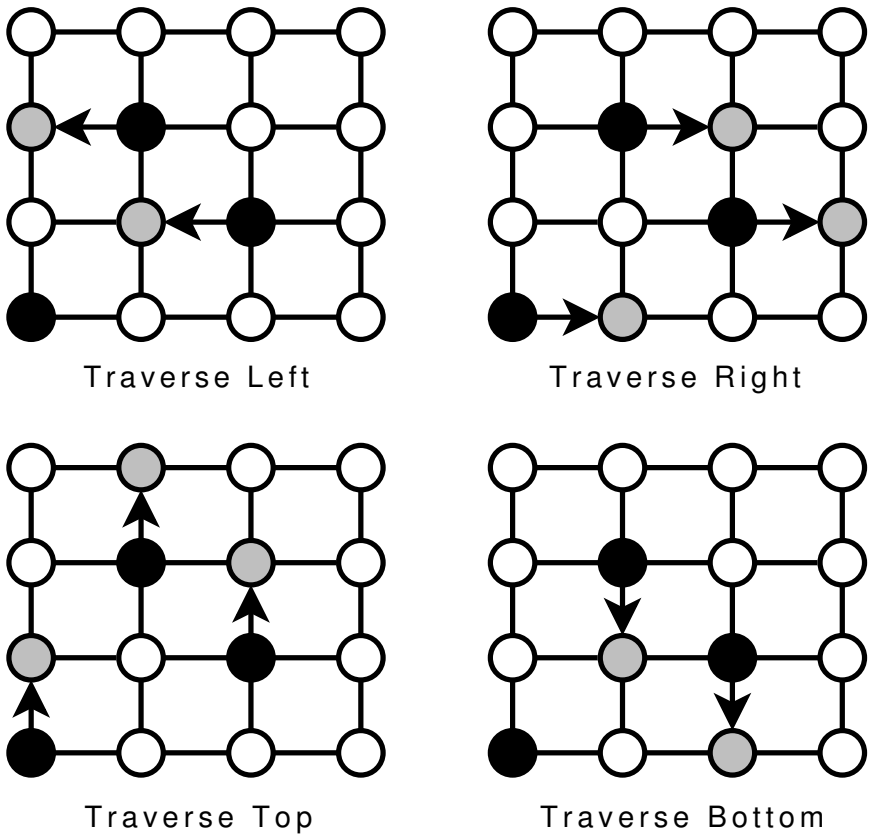


Figure 6.8: Black nodes are on the same depth level. The illustration gives an example of how lockstep BFS traversal works to create the next depth level from these nodes. Gray nodes are the nodes traversed in each direction.

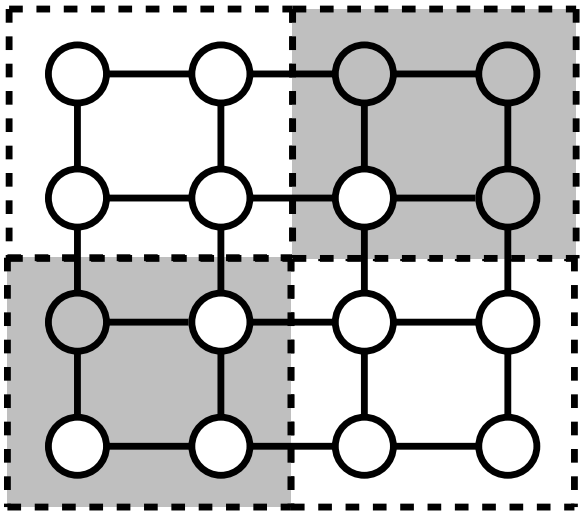


Figure 6.9: A 4×4 grid graph divided into 2×2 tiles. Dotted lines indicate tile boundaries. Active nodes and active tiles are shown in gray. A tile is active if and only if it contains at least one active node.

When traversing nodes at depth level k from nodes at depth level $k-1$, applying the lockstep BFS traversal technique allows us to use an array of flags whose size is equal to the number of nodes in level $k-1$, n_{k-1}. That reduces the order of work complexity of building level k of the traversal lattice to $\Theta(n_{k-1})$ instead of $\Theta(n)$. The overall complexity of the traversal becomes $\Theta(\sum_k n_k) = \Theta(n)$.

6.5.2 Cache Emulation

Because nodes' data – excesses, labels, and outgoing link capacities – are updated and then read during the pushing and labeling phases, we selected to store all nodes' data in the global memory space, which is a read/write space [1]. The problem

[1]In fact, since data updated during a kernel invocation are not read later on during the same kernel invocation, texture memory space can be used as well. However, in the current CUDA implementation this trick is restricted only to 1D arrays, which limits its utility. Also, according

with the global memory space is that it is not cached. Moreover, threads have to respect a specific memory access pattern in order for reads from global memory to be coalesced (Section 5.1). Graph algorithms generally exhibit an irregular memory access pattern, which makes requirements for coalescing not guaranteed. The *cache emulation* technique aims at regularizing accesses to global memory by enforcing memory coalescing requirements on global memory accesses. It basically works by emulating the operation of a multi-dimensional cache memory unit. For the technique to work, data accessed have to be structured as a one dimensional or multi-dimensional array. For the technique to be useful, processing a data element has to rely only on its local neighborhood in the array; hence comes the restriction to grid graphs in this case. Unlike a hardware cache memory, whose operation is independent of any algorithm, the cache emulation technique in fact requires modifying the way the algorithm works in order to be used. We will explain the technique in the context of our Graph Cut implementation on two dimensional grid graphs. Nevertheless, we believe the technique is fairly general and deploying it in other problems is straightforward.

Nodes' data of a two dimensional grid graph are assumed to be stored in two dimensional arrays. To process a node in the graph, we actually load from global memory its data and the data of the 2D *tile* of nodes in which it resides, and process all the nodes in the tile in parallel. In other words, the graph is divided into equally sized *tiles* of nodes. Figure 6.9 shows a simple 4×4 grid graph divided into 2×2 tiles. The algorithm proceeds exactly as explained earlier. The only difference is

to our tentative experiments, the technique presented in this section performs much better.

in building the breadth first search lattice. For each level of the lattice, instead of constructing a list of nodes, a list of tiles is constructed. Each tile added to a level of the lattice must have at least one node that belongs to that level. Figure 6.9 depicts the relationship between an active tile and an active node. Activity here means eligibility for the currently performed operation. For example, when pushing flow from level k of the lattice to level $k-1$, nodes that belong to level k are the active nodes, and the tiles containing them are the active tiles. Note that a node that belongs to an active tile is not necessarily active. But, at least one node in tile must be active for the tile to be active.

In the CUDA implementation, each block of threads corresponds to a tile of nodes. Each thread of a block loads data of one node in the tile from global memory to shared memory. Then, if a node is active for the current operation, its thread proceeds and processes the node, otherwise, the thread terminates. For example, if the operation is parallel labeling with label k, a thread processes a node only if it has label $k-1$, otherwise the thread terminates after loading the node's data. If the operation is parallel pushing from level k to level $k-1$ in the lattice, a thread processes a node only if it has label k.

When a thread terminates without processing a node, it actually does a useful job before termination. Indeed it helps in the most time consuming operation. It helps in reducing the overhead of memory access for other threads in the block that are processing active nodes. Specifically, the benefit from this technique is two fold:

- *Efficient Memory Access:* Two factors enhance memory access performance

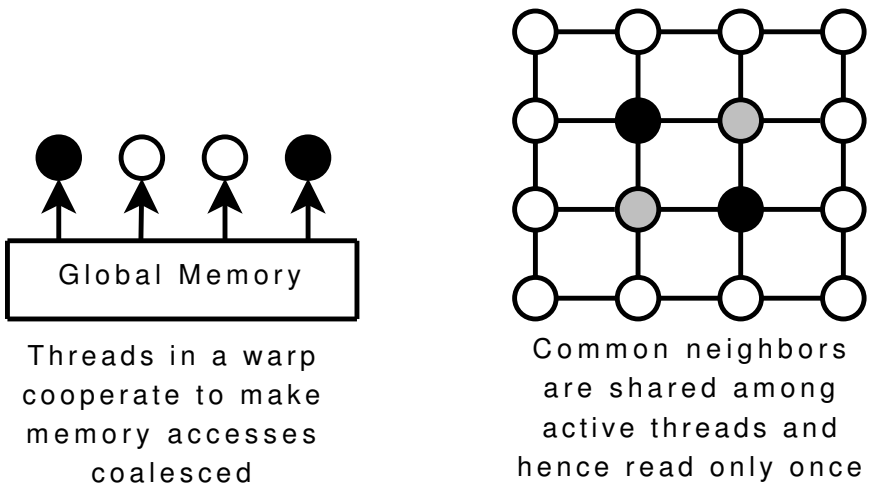


Figure 6.10: Black circles represent active nodes. Gray circles are common neighbors to more than one active node. The illustration shows how active and non-active nodes can cooperate and how active nodes can share data to save memory access time.

when using cache emulation. One factor is *cooperation* among threads by helping one another to achieve memory read coalescing. All threads in a warp of a thread block read adjacent global memory addresses with proper alignment. As explained in Section 5.1, that makes these reads coalesced together. The other factor is *sharing* among threads. Data read by one thread are stored in shared memory and become available for other threads in the block. Figure 6.10 depicts these notions.

- *Less Prefix Sum Overhead:* When using cache emulation, prefix sum operations are performed over tiles not individual nodes. If each tile on average contains b active nodes in the same level of the lattice, the number of elements processed by prefix sums is reduced by a factor of b on average.

Since the computations performed per node during the pushing and labeling phases are very simple computations, the time for global memory access is much more significant. That is proved by the superiority of the implementation that utilizes the cache emulation technique over others, as shown below.

6.6 Experimental Results

We experimented our implementations on an image segmentation task. The segmentation algorithm we use is the one in Boykov and Jolly [9]. It basically divide pixels in the image into two partitions, foreground and background, depending on user's input that marks some pixels as foreground and others as background. In the results shown here, we did not use hard constraints, *i.e.* enforcing some pixels to belong to the foreground or background. We set the weights of singleton and pairwise energy terms to 1 and 2.5, respectively, and the noise value to 50. Only 4-connected neighborhoods are used in the generated graphs.

We compare the running times of five different implementations: a CPU implementation of the proposed technique, a basic CUDA implementation without any optimization, a CUDA implementation using lockstep BFS, a CUDA implementation using cache emulation, and the CPU implementation introduced in Boykov and Jolly[10], which is reported to be the fastest in practice for grid graphs, and whose implementation is available online. For the CPU implementation of our technique, BFS traversal is implemented in the regular sequential way without using prefix sum operations.

For all our CUDA implementations, the graph nodes' data are stored in three two-dimensional arrays of type `float4` (a structure of four floating point elements.) One array contains reservoirs, one array contains outgoing residual link capacities, and the last array contains for each node its excess, label, and link capacities to s and t. The results presented here for the proposed technique when implemented with cache emulation are for tile size of 8×4. We experimented with tile sizes 16×8, 16×12, and 16×16 as well. The best results we got were for the tile size 8×4. This is actually at odds with the recommended block sizes for CUDA devices [69]. This issue needs further investigation.

On running the algorithm on real images, it turned out not to be easy to adjust the user input, and weights for the singleton and pairwise terms to get desirable segmentation results. Alternatively, we report results here for experiments on synthetic images only. Each synthesized image is produced by drawing a foreground with intensity values generated from a Mixture of Gaussian density function, on a background with intensity values generated from a different mixture. Each mixture has three components. The foreground shape of each image is a collection of ellipses in random locations, orientations, and sizes. In each generated image, we enforce the condition that the 32×32 patch at the center of the image belongs to the foreground, and the 32×32 patch at the top left corner of the image belongs to the background. These are the patches based on which foreground and background intensity histograms are constructed. The enforcement here is in the distribution from which the random intensity values for these patches are generated. But, we do not enforce the resulting segmentation to assign these patches specific labels.

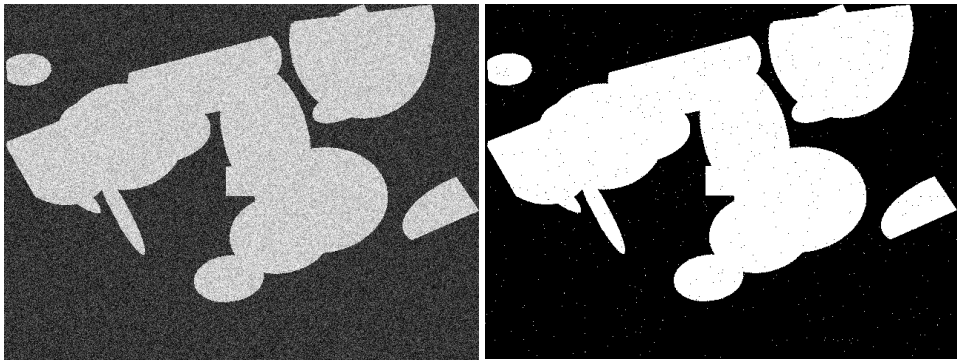


Figure 6.11: Left is a sample randomly generated image. Right is its segmentation result.

Each generated image is resized to make a set of six different sizes. We compare the segmentation results from the five implementations and make sure they are exactly the same for each image. Each implementation is run for 10 times on each image. An example image with the resulting segmentation is shown in Figure 6.11. The CPU implementations are run on an Intel Xeon 3.2 GHz processor with 1GB RAM. The GPU implementations are run on a GeForce 8800 GTX graphics card.

The plot in Figure 6.12 compares the running times of the five implementations with different image sizes. This is the time of running the Graph Cut on the generated graph 10 times. The time to generate the graph itself is excluded. Also, for the CUDA implementations, the time to transfer the graph data from the host to the device and the time to transfer the result from the device to the host are excluded. That is because the input graph can actually be constructed on the device as well, which should be much faster than constructing it on the host. But, that is not done in our implementation. Also, the resulting cut might be postprocessed on

the device, or directly rendered to the screen buffer in some applications.

The implementation of the proposed algorithm on CPU is not always faster than the implementation of Boykov-Kolmogrov's (BK) method [10]. The plot in Figure 6.12 shows also that the basic CUDA implementation, without optimizations, consistently outperforms the two CPU implementations. The two optimizations proposed introduce another considerable speedup over the non-optimized CUDA implementation. The cache emulation technique in particular is consistently the fastest. That emphasis the importance of memory access optimization on such devices. That is particularly important in graph algorithms in general since they are typically memory intensive algorithms.

The plot in Figure 6.13 shows the speed up of the CUDA implementation of the proposed algorithm with cache emulation when compared to the faster of the two CPU implementations. The speedups gained are in the range 1.7-4.5, depending on the image size.

6.7 Conclusion and Future Work

In this chapter, we presented our results and findings on implementing Graph Cut on CUDA. To address the unique architectural features of CUDA, we resorted to an unusual way of implementing Graph Cut in parallel. Our approach relies on BFS traversals solely to assign node labels. This computationally inefficient approach facilitates turning around limitations of CUDA, such as the simple SIMD execution model and the unavailability of memory locking constructs. Nevertheless, the

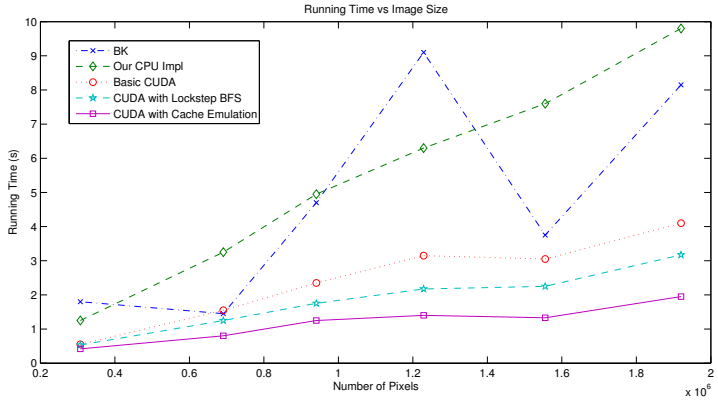


Figure 6.12: Comparison of running times vs image size for different implementations of Graph Cut. *BK* is Boykov-Kolmogriv's method and *Our CPU* is the implementation of the proposed approach on CPU

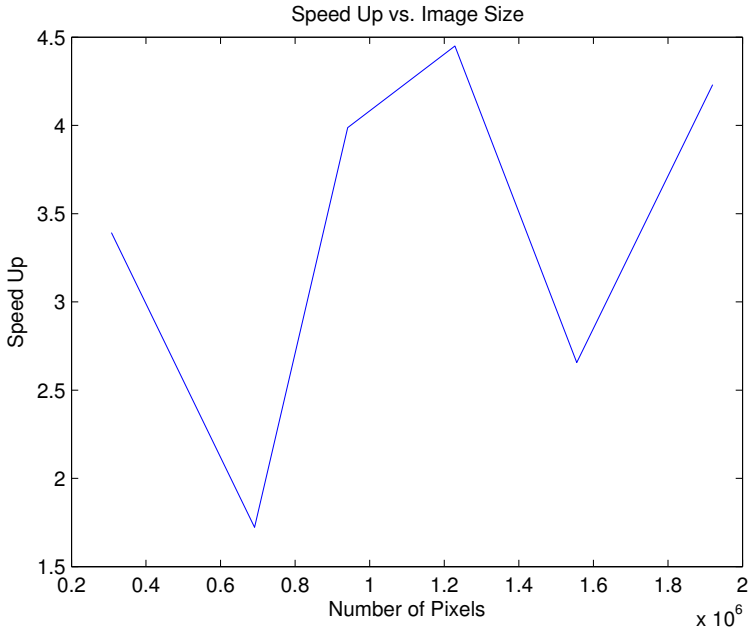


Figure 6.13: The speed up of the the proposed method with tiling on CUDA over the faster CPU implementation for each image size.

performance of this approach provides both relative and absolute speedups when experimented on image segmentation of synthesized images. We proposed two improvements to this technique that make use of the special structure of grid graphs to deliver better performance, lockstep BFS and cache emulation. The lockstep BFS utilizes the special structure of grid graphs to make BFS traversal implementation on CUDA work efficient. The cache emulation technique is fairly general. It aims at regularizing memory access patterns to enforce memory read coalescing as much as possible through emulating the operation of a cache memory unit. The experimental results showed that the proposed techniques indeed enhance the performance, especially the cache emulation technique. That was expected since graph algorithms in general are memory bound and enhancing memory throughput is crucial to enhance their overall performance.

We are investigating how to enhance the speed further by applying the global relabeling heuristic concurrently with the push and relabel operations. We would like also to experiment our implementation on a wider range of images and imaging applications. In particular, having a fast implementation for Graph Cut is much more important when the algorithm is applied to general discrete labeling, instead of binary labeling, since in the former the algorithm is actually run for many times until it converges. Therefore, we would like to experiment our techniques on such applications. Finally, we are interested in extending our approach to work with higher connectivity graphs, such as in 3D grids. In such case, finer parallelism at the link level could deliver higher performance desirable.

Chapter 7

Band Approximation of Gram Matrices for Kernel Methods on GPUs

Kernel-based methods require $O(N^2)$ time and space complexities to compute and store non-sparse Gram matrices, which is prohibitively expensive for large scale problems. We introduce a novel method to approximate a Gram matrix with a band matrix. Our method relies on the locality preserving properties of space filling curves, and the special structure of Gram matrices. Our approach has several important merits. First, it computes only those elements of the Gram matrix that lie within the projected band. Second, it is simple to parallelize. Third, using the special band matrix structure makes it space efficient and GPU-friendly. We developed GPU implementations for the Affinity Propagation (AP) clustering algorithm using both our method and the COO sparse representation. Our band approximation is about 5 times more space efficient and faster to construct than COO. AP gains up to 6x speedup using our method without any degradation in its clustering performance.

7.1 Introduction

Kernel-based machine learning methods [62, 84, 85, 95] have gained significant attention within the machine learning community and other applied fields for more than a decade. They are commonly used for many purposes, such as classification

[43], regression [89], clustering [35], and dimensionality reduction [84]. The main advantage of kernel methods is their ability to learn non-linear functions using simple linear methods. They achieve this by implicitly mapping the input points from the *input* space to a typically higher, and possibly infinite, dimensional *feature* space. The mapping is realized via a kernel function, which computes a dot product between a pair of data points in the feature space without explicitly performing the mapping to that space. This is popularly known as the *kernel trick*. The pairwise dot product values are stored in the so-called *Gram matrix* or *kernel matrix*.

Given a data set of N points, computing the pairwise kernel values requires $O(N^2)$ computations and the values are stored in an $N \times N$ matrix. For large values of N, the time and space complexities to compute and store the Gram matrix can be prohibitively expensive. A common solution to the space complexity problem is to compute the elements of the kernel matrix on-demand, which trades the memory requirements for a much longer computational time. These complexities limit the use of kernel methods to relatively small problems. However, our era is marked with the availability of tremendous amounts of digital data that needs to be analyzed. Moreover, in many learning applications, increasing the number of training data points significantly improves the model's performance. For example, Munder and Gavrila [67] showed that the classification error of their Support Vector Machine (SVM) classifier, for human detection in images, was reduced by approximately a factor of two by only doubling the size of the training data set. They also noted that the reduction in the classification error obtained by increasing the number of training points exceeded any reduction obtained by using better features or learning

algorithms. A similar observation was made by Torralaba *et al.* [91]. Inspired by such observations, our research focuses on enabling large scale learning with kernel based methods.

Fortunately, the availability of massive datasets nowadays and the increased demand and motivation for large scale learning is accompanied with the emergence of several new computing architectures, such as Graphics Processing Units (GPUs). GPUs have been rapidly advancing towards higher levels of parallelism, and have recently become readily programmable with simple thread-based Application Programming Interfaces (API) instead of using graphics primitives. However, the tremendous computing power provided by such devices is both a bless and a challenge at the same time. The virtue of parallelism offered by GPUs comes at the expense of several restrictions on the algorithm design in order to achieve the promised performance. One of the most critical of these restrictions is on the memory access pattern exhibited by the algorithm.

In this chapter, we present a novel approach to address the limitations of kernel based methods for large-scale machine learning applications. Specifically, we introduce a new method to construct a band sparse matrix approximation to the Gram matrix. The idea is to order the input points so that the significant elements of the Gram matrix become confined to a limited band. Having a sparse structure for the Gram matrix is generally one of the ways to address the space complexity problem of kernel methods. However, having a band sparse matrix structure in particular offers more advantages. It allows for a very simple representation that has significantly lower memory overhead than general sparse matrix representations.

Moreover, this simple representation naturally adheres to the restrictions on memory access patterns on GPUs for many common matrix operations. Therefore, it allows for significantly more efficient implementations for most algorithms that operate on the matrix.

To construct a band approximation to the Gram matrix, we order the input points so that those that are close to one another in the resulting ordering are more likely to be close in the Euclidean space in which they reside. To efficiently obtain the desired ordering, we rely on the locality preserving properties of space filling curves [82]. To construct the matrix, we evaluate the kernel function only within a fixed neighborhood around each point in the obtained ordering. This results in a band Gram matrix by construction. The assumptions here are that the value of the kernel function is monotonically decreasing with the Euclidean distance between the input points, and the significant values of the function occur between points within the selected neighborhood size.

To illustrate the validity of the proposed approach, we use Affinity Propagation [34, 35], an unsupervised clustering algorithm, as an example of kernel methods. Affinity Propagation (AP) operates on a *similarity matrix*. Similar to a kernel matrix, a similarity matrix has an element for every pair of points, whose value represents a measure of similarity between the two points. Typical choices of similarity functions, such as the negative sum of squared differences, and its exponential, can be shown to be dot products in higher dimensional mappings of the input points. We developed two GPU implementations for AP: one is based on our method, and the other is based on the COO (Coordinate) general sparse matrix representation [80].

As a baseline for comparison, we also developed a CPU implementation for AP based on COO. Our results show that the band matrix representation used in our method is 5 times more space and time efficient to construct than the COO representation on GPUs. Moreover, the GPU implementation of AP using our approach is 6 times faster than the GPU implementation using COO and 114 times faster than the CPU implementation. This significant speedup for AP comes with no loss in its clustering performance despite the approximation in our approach.

The main contributions presented in this chapter are:

- An efficient method to construct a band approximation of a Gram matrix, without having to compute all elements of the matrix first. The simplicity of representing a band matrix allows for space efficiency and time efficiency on processing the matrix on GPUs. Hence, our method can effectively address the space and time complexities associated with kernel based learning methods for large scale problems.

- An efficient GPU implementation of the Affinity Propagation algorithm using our method. This implementation achieves 114x speedup over the CPU implementation and 6x speedup over another GPU implementation based on the COO sparse matrix representation. These speedups are achieved without compromising the quality of the output clustering.

The rest of the chapter is organized as follows: We briefly present the related work in Section 7.2. We explain our method to construct band approximations to Gram matrices on GPUs in Section 7.3. We introduce AP and its implementation

on GPUs in Section 7.4. In Section 7.5, we present the experimental results. Finally, we outline our conclusions and plans for future work in Section 7.6.

7.2 Related Work

The work addressing the limitations of large scale kernel methods can be broadly classified into two main categories – (1) methods that depend on constructing low-rank approximations of the kernel matrix and (2) efficient implementations for computing the kernel matrix. Low-rank methods depend on the observation that the eigen-spectrum of the kernel matrix rapidly decays, especially when the kernel function is a Radial Basis Function (RBF) [84, 86, 100]. Hence, for a kernel matrix K with eigenvalues $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_N \geq 0$ and corresponding eigenvectors $\mathbf{v}_i$, $K = \sum_i^N \lambda_i \mathbf{v}_i \mathbf{v}_i^T$. However, since the eigen-spectrum decays rapidly (*i.e.* most of the information is stored in the first few eigen vectors), the kernel matrix can be approximated by $\tilde{K} = \sum_i^M \lambda_i \mathbf{v}_i \mathbf{v}_i^T$, and $M << N$. Williams and Seeger [101] use the Nystrom method [23] to compute the most significant M eigenvalues and eigenvectors. The number of computed eigenvectors is inversely proportional to the approximation error. Nystrom-based methods are $O(M^2N)$ where M is the number of computed eignvectors. Also, Drineas and Mahoney [27], and Smola and Schökopf [90], for example, compute a rank-k approximation of the kernel matrix using a subset of the column (or basis functions) of the kernel matrix. These methods generally are $O(N)$ in both space and time.

Due to the importance of kernel-based methods, they have been the target of

prior GPU implementations. Ohmer *et al.* [70] use GPUs to implement the classification step of SVM classifier, in which the kernel values are computed between the input *test* vector and the set of support vectors. While focusing on the classification phase rather than the training phase of the computation can be justified by the higher frequency of using a trained model for classification in practical applications, for large scale learning the training phase becomes the main obstacle. In SVM classifiers, for example, the number of support vectors in a trained model can be much smaller than the training vectors used to train the model. Catanzaro *et al.* [13] presented an implementation of Platt's Sequential Minimal Optimization (SMO) [73] on GPUs for training SVMs. In this implementation, the kernel size issue was handled by caching recently used values and computing other values on demand upon cache misses. For large scale problems, cache misses are more likely to happen. Hence, computing the kernel values on cache misses is expected to be the computational bottleneck in large scale problems.

The idea of using space filling curves to order points for efficient access on GPUs was recently used by Leiberman *et al.* [58] with the similarity joint operation and was suggested also for use to approximate k-nearest neighbor search. Our sparse matrix representation actually uses approximate k-nearest neighbor search to obtain the band matrix structure.

In contrast to band reduction techniques, such as the RCM algorithm [19], our method does not start from an already constructed general sparse matrix and *reduce* it to a band matrix. Instead, our method directly *constructs* a band matrix from the input points. This is a fundamental difference since constructing a sparse matrix

typically requires the computation of the full dense matrix first to determine which elements to keep. Our method computes only the elements within the projected band. Moreover, band reduction techniques typically use graph algorithms, which are hard to implement in parallel. Our method can be easily implemented in parallel in an efficient way.

7.3 Representation of Gram Matrices on GPUs

Since the Gram matrix often has a rapidly decaying eigen-spectrum, as explained in Section 7.2, especially when using kernel functions with compact support, it is customary to assume that the matrix is approximately sparse and use sparse matrix structures to store (and operate on) its significant values. We are particularly seeking a sparse matrix representation that is efficient to construct, has low space overhead, and efficient to perform common matrix operations on when implemented on GPUs.

The Compressed Sparse Row (CSR) is a common sparse matrix representation on GPUs [37]. It supports efficient sparse matrix-vector multiplication, and other operations, through efficient segmented scans [26]. A closely related representation is the Coordinate (COO) representation [80, 6]. Despite the larger spatial complexity of COO compared to CSR, COO exhibits a better memory access pattern on GPUs for some operations. We use COO as a baseline representation in our experiments. Nevertheless, in the following discussions, all our arguments about COO, except for the space complexity issue, apply equally to CSR.

COO is a general sparse matrix representation that does not assume any special

structure for the matrix. As we will show below, due to its generality, the COO representation has several shortcomings in terms of its space overhead and the memory access pattern exhibited with it in common matrix operations on GPUs. To overcome these limitations, we aim to find an approximation to the Gram matrix with a special structure that can be represented efficiently on GPUs in terms of space and computations. In our approach, this special structure is the band matrix structure.

In the rest of this section, we first explain the COO representation and its limitations in Section 7.3.1. Then, we explain the band matrix structure and its merits in Section 7.3.2. Finally, we explain how we obtain the band approximation of the Gram matrix in our approach in Section 7.3.3.

7.3.1 General Sparse Matrix Representation Using COO

Consider the COO representation of a general sparse matrix, as shown in Figure 7.1. In this representation, three arrays are used to store the row index, column index, and the value of each significant element in the matrix. For m significant elements to store, we use the space of $3m$ elements, which is a significant overhead factor. Beside the space overhead of this representation, when processing each element of the matrix depends on its row and column indices, three arrays need to be accessed in order to process all elements, which results in a significant time overhead if the processing is not compute intensive.

To perform a scan operation on the rows of the matrix efficiently in parallel, the three arrays need to be sorted according to the row index values. Having ar-

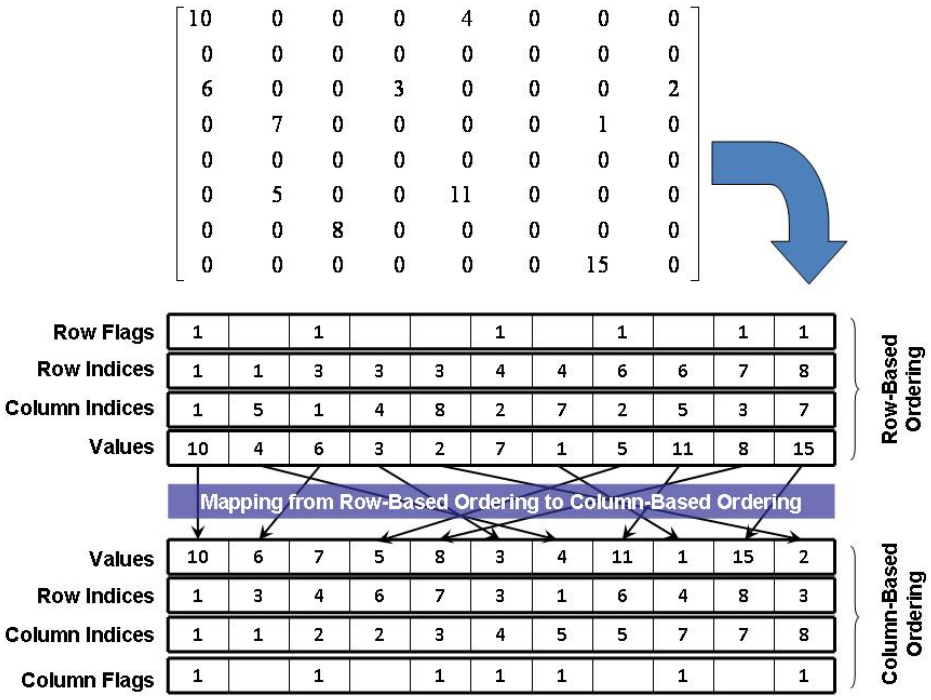


Figure 7.1: Representation of an example general sparse matrix using the Coordinate (COO) representation. The representation consists of three arrays to store row indices, column indices, and values of non-zero elements. The arrays can be sorted according to either the row indices or the column indices depending on whether we need to perform scan over the rows or the columns of the matrix. If scanning is performed on rows and columns interchangeably, a mapping from one ordering to the other is retained with the structure. Finally flags arrays indicating the beginning of each row and column is needed for the segmented scan operation.

rays sorted in such a way, a segmented scan operation can be used to perform the scan operation in parallel. Fortunately, there are efficient algorithms for segmented scans [87, 26]. However, the segmented scan operation requires as input an array of flags, whose elements designate the beginning of each row of the matrix. This is on top of the internal arrays used by the operation itself. Therefore, the operation can be performed efficiently on a GPU with the usage of extra space. Similarly, if we are to perform a scan operation on the columns of the matrix, we need to have the arrays sorted according to column indices, and an array of flags to mark the beginning of each column. Figure 7.1 shows the COO representations of a sample sparse matrix with using row-based ordering and column based ordering.

Another shortcoming of this representation arises when we need to perform the scan operation on both rows and columns interchangeably. In this case, the COO representation must be extended. One solution is to keep the arrays sorted according to the row indices, for example, and two extra arrays: one to store the mapping from the row-index-based order to the column-index-based order of the arrays, Figure 7.1, and the other is the flags array of the column-based ordering. When we need to perform a scan operation over the columns, we use the mapping array to reorder the values and perform a segmented scan on the reordered array. Note that we need an extra array to temporarily store the reordered values. Finally, the mapping from one order to the other requires random device memory access during write, which does not respect the conditions for coalescing, as discussed in Section 5.1.

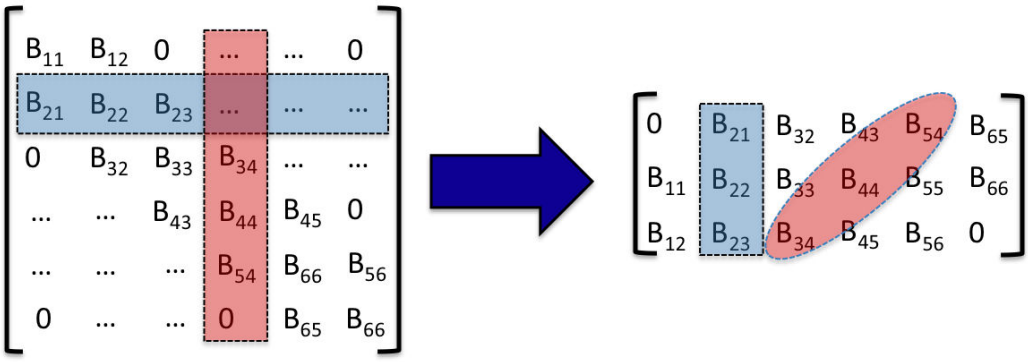


Figure 7.2: Representation of a band matrix using a 2D array. Each diagonal of the matrix is stored in one row of the array. Each row is stored in one column of the array. And each column of the matrix is stored as a diagonal in the array.

7.3.2 Band Matrix Representation

Consider an $N \times N$ band matrix with a bandwidth k, *i.e.* the non-zero (or significant) elements are confined to at most k diagonals. A simple representation of such a matrix on the GPU is a 2D array, where each diagonal of the matrix is stored as a row of the array, and each row of the matrix is stored as a column of the array [6], as shown in Figure 7.2. To perform a parallel scan operation on the rows of the matrix, we can assign each thread to a column of the representation array. Each thread loops over the elements of its assigned column and performs the operation. Hence, consecutive threads in a block of threads read consecutive elements in memory. Furthermore, if the array is allocated so that each row starts at a properly aligned memory address, and the block width is selected appropriately, all conditions for memory access coalescing are satisfied, and hence the read operation is performed efficiently.

Note that in our 2D array representation, columns of the matrix are stored in diagonals of the array, as shown in Figure 7.2. If we need to perform a scan operation on the columns of the matrix rather than the rows, we can still assign each thread to a column in the matrix. Each thread loops over the elements of its assigned column. Consecutive threads still read consecutive memory addresses. However, since columns of the matrix are stored as diagonals in the representation arrays, looping over elements in a column require non-aligned memory access. Fortunately, as we mentioned in Section 5.1.3, consecutive accesses, even if the addresses are not properly aligned, can always be made to satisfy coalescing requirements either directly through the hardware in latest models, or through software by making good use of the shared memory space.

As we have shown, we can efficiently perform simple scan operations on the rows or columns of a band matrix represented as a 2D array. Another advantage of this representation is that the row and column indices of each element can be calculated instead of being read from separate arrays, which saves a lot of time in memory bound processing. The space overhead of the band representation depends on the location of the significant diagonals with respect to the main diagonal. Suppose that the bandwidth $k = 2h + 1$, so that the bandwidth is divided as the main diagonal, and h diagonals below it and h diagonals above it. In this case, we use a space sufficient for kN elements to actually store $kN - h^2 - h$ elements. Therefore the space overhead is $\frac{h^2+h}{kN}$. Note that the space overhead is always smaller than 1. The scan operations do not require any extra space in the device memory.

7.3.3 Band Approximation of Gram Matrices

We have shown the advantages of the simple representation for band matrices over the COO representation for general sparse matrices in terms of the space overhead and the efficiency of the memory access pattern for performing simple scan operations over the rows and columns of the matrix. To exploit these advantages for sparse kernel matrices, we need to find an ordering of the rows and columns that puts most of the significant elements within a fixed number of diagonals and use a band matrix representation for the resulting matrix. If the scan operations to be performed are both commutative and associative, changing the order of the rows and columns will not affect the results. We refer to this approach by BAG, for Band Approximation of Gram matrices.

Each element of a Gram matrix is the value of a kernel function on two points. We assume that the data points lie in a Cartesian space and the kernel value between pairs of points is inversely proportional to the distance between the points in the Cartesian space. These assumptions are satisfied by a variety of kernel functions, including the most popular RBF kernel. A typical kernel function choice in Affinity Propagation is the negative sum of squared differences, which satisfies these assumption too.

In order to obtain a band kernel matrix of bandwidth k, the problem is to find an ordering of the points such that nearby points in the Cartesian space are at most k elements apart in that ordering. This ordering may not optimally exist. Therefore, we need an ordering that satisfies this property for most of the elements. We can

formulate the problem of finding such ordering as an optimization problem. This is basically the idea of band reduction techniques for sparse matrices [19]. However, band reduction techniques require the construction of the sparse matrix first, and are complex to parallelize as we explained in Section 7.2.

We use Space Filling Curves (SFC) [82] to obtain the desired reordering. A space filling curve is a path through the points of a discrete Cartesian space that passes through each point exactly once. There are many types of SFCs. Typically, an SFC is more likely to connect points that are close to one another in the space than to connect points that are far away in the space. However, this *locality preserving* property varies from one type of curves to another. The family of Hilbert curves [12] is known to have good locality preserving properties. However, they are complex to construct. A much simpler curve to construct is the Z-curve [66]. Despite being inferior to the Hilbert curves in locality preserving, it is good enough in many applications. As we will show, it works remarkably well for kernel matrix reordering with the affinity propagation algorithm.

The construction of the space filling curve is performed implicitly (*i.e.* we need not know how exactly the curve looks like.) Given a set of points in a Cartesian space, all what we need to know is their ordering along the curve, *i.e.* in which order the points are encountered upon traversing the curve from its starting to its ending points. For the Z-curve, this ordering is known as the z-order, or the Morton code. The z-order can be computed very efficiently using bit interleaving of the point coordinates in the Cartesian space [82]. For real valued data, we first map the points to the unit hypercube. Then, we discretize the coordinates by mapping each

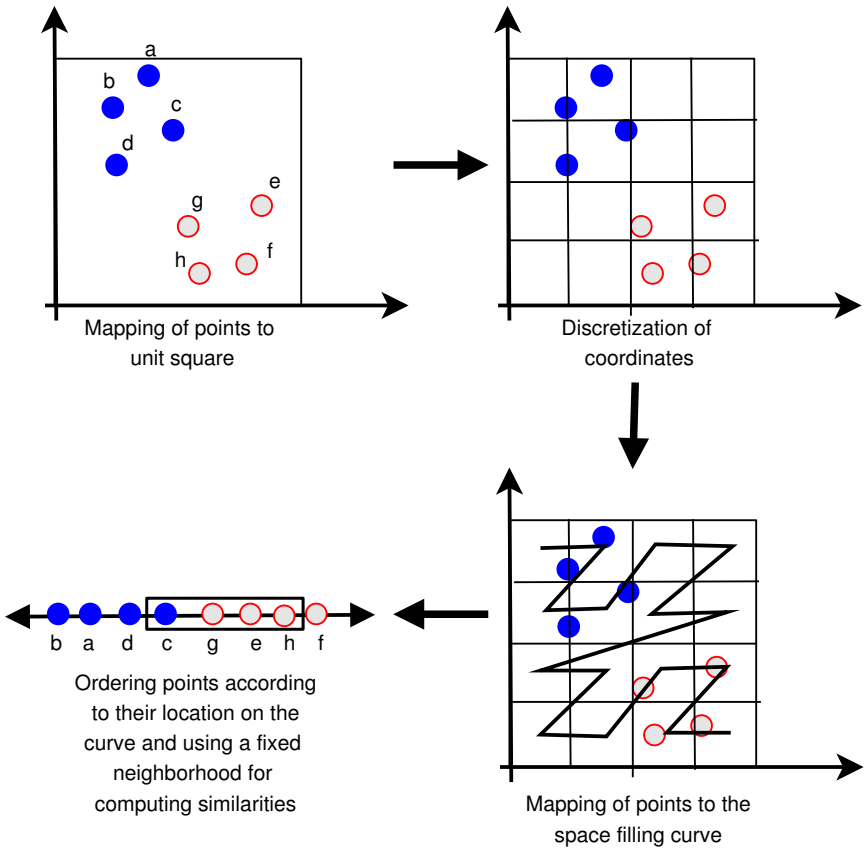


Figure 7.3: An illustration of the construction of the similarity matrix using space filling curves. The Z-curve is used in this illustration, and in our implementation.

point to the closest cell of a discrete grid over the unit hypercube . Morton codes are computed using the discrete coordinates of the assigned grid cells, and the points are sorted by their ascending or descending Morton code values. Figure 7.3 shows an illustration of these steps for a simple 2D example. In our implementation, we use the bitonic sort algorithm [53] to sort the codes. Although the complexity of bitonic sort is $\mathcal{O}(N \log^2 N)$, the time to compute the codes and sort them is negligible with respect to the rest of the computations.

7.4 Affinity Propagation on GPUs

To illustrate the effectiveness of our band approximation of the Gram matrix, we use Affinity Propagation (AP) as an example of kernel methods. In this section, we briefly explain the AP algorithm, and describe our GPU implementation.

7.4.1 Affinity Propagation

AP is an unsupervised data clustering algorithm introduced by Frey and Dueck [34, 35]. There are two main advantage of data clustering using AP: First, the number of clusters K need *not* be a priori specified. Second, AP operates on pair-wise similarity values which can be computed on non-Euclidean manifolds. For completeness, we briefly describe Affinity Propagation clustering.

Let $\mathbf{X} = \{\mathbf{x}_i; i = 1, 2, \ldots, N\}$ be a set of data points (*i.e.* observations vectors) with unknown cluster structure and $\mathbf{X} \subset \mathrm{R}^d$. The objective is to find a subset $\mathbf{X}_e = \{\mathbf{x}_k; k = 1, 2, \ldots, K\} \subset \mathbf{X}$ of cluster exemplars where $K \ll N$. This problem is classically handled using the K-center algorithm in which K points are selected at random from $\mathbf{X}$ and the subset is iteratively refined by minimizing the distance between the data points and the exemplars. The procedure is usually repeated more than once in order to converge to the best solution. AP, on the other hand, considers all points to be possible exemplars. Based on similarity (as opposed to distance) $s(i, j)$ between $\mathbf{x}_i$ and $\mathbf{x}_j$. Self similarity values $s(k, k)$ are referred to as the preference values. The higher the preference given to a sample point, the more likely it can be selected as an exemplar by the algorithm.

AP operates by iteratively exchanging two types of messages between data points – availabilities and responsibilities. Responsibility $r(i,k)$ indicates the desire of point i to belong to a cluster for which point k is the exemplar. Availability $a(i,k)$ indicates the willingness of point k to serve as the exemplar of the cluster to which point i belongs. All availabilities are initialized to zeros. Responsibilities are updated as soft assignments using Equation 7.1.

$$r(i,k) \leftarrow \left\{ s(i,k) - \max_{k' \text{ s.t. } k' \neq k} \left(a(i,k') + s(i,k') \right) \right\} \tag{7.1}$$

This responsibility update ensures that all potential exemplars compete for data points. Availabilities are updated using Equation 7.2.

$$a(i,k) \leftarrow \min \left\{ 0, r(k,k) + \sum_{i' \text{ s.t. } i' \notin \{i,k\}} \max\left(0, r(i',k)\right) \right\} \tag{7.2}$$

$$\text{and} \quad i \neq k$$

The self-availability $a(k,k)$ is updated differently in order to reflect the evidence that point k can be an exemplar, as shown in Equation 7.3.

$$a(k,k) \leftarrow \sum_{i' \text{ s.t. } i' \neq k} \max\left(0, r(i',k)\right) \tag{7.3}$$

The algorithm proceeds by iterating over the responsibility and availability update steps in Equations 7.1, 7.2, and 7.3 until convergence or the maximum number of iterations is reached [35].

7.4.2 GPU Implementation

The dense matrix implementation of AP is $O(N^2)$ in both computational and memory requirements. In practice, the similarity values s_{ij} can be thresholded so that

	1	$\cdots$	i	$\cdots$	N
$-h$			$s_{(h-1)i}$		
$\vdots$			$\vdots$		
-1					
0		$\cdots$	s_{ii}	$\cdots$	
1					
$\vdots$			$\vdots$		
h			$s_{(h+1)i}$		

Figure 7.4: The layout of the similarity matrix, S, used in AP's implementation with the BAG method. Each column of the matrix contains similarities to neighbors ranging from $-k$ to k apart from the column's index. The responsibilities and availabilities matrices, R and A, use the same structure. elements in row index 0 represent preference, self-responsibility, and self-availability values, in the S, R, and A matrices, respectively.

small values are ignored and the pairwise similarity values can be stored in a sparse matrix. Using the massive parallelism available in modern GPUs, we can effectively address the computational complexity problem.

For simplicity of presentation let's assume that full matrices are used to implement AP. To store the similarity values $s(i, j)$, and the preference values $s(k, k)$, we need an $N \times N$ array S. To store the availability and responsibility messages sent from one point to another, we need another two arrays of the same size, A and R, respectively. From equation 7.1, to update the responsibility values, we need to scan rows of the A and S arrays. Specifically, we need two passes over each row. In the first pass, we compute the maximum two $a(i, k) + s(i, k)$ values in each row. In the second pass, we compute the updated responsibility value for each element in the row, using the two maximums computed in the first pass. From equation 7.2,

to update the availability values, we need to scan the columns of the R array twice as well. In the first scan pass, we compute the sum of all positive elements in the column excluding the self responsibility values. In the second pass, we update the availability value of each element using the sums computed in the first pass. Implementing row and column scans on full matrices on the GPU is straight forward and efficient. However, we cannot store full matrices in memory even for moderately large problems. Therefore, we must use a sparse structure.

Both the COO and BAG representations support row and column scan operations which we need to perform interchangeably in AP. We will show that the COO representation will be highly inefficient for this purpose compared to the BAG representation. The COO structure is constructed by first sampling random pairs of points and computing similarity values between them. Then, we select a threshold below which similarity values are discarded. The threshold is selected based on the random sample and based on a pre-specified limit on the final storage size. Note that to construct the COO structure, we need to compute the similarity values between all pairs of points in order to threshold them and keep the significant ones only. Also, recall from Section 7.3.1 in order to support both row and column scans in this structure, we need to keep a mapping from an ordering based on row indices to an ordering based on column indices. In our implementation, we construct the structure first ordered by row indices, then use bitonic sort to obtain the ordering based on column indices, and retain the mapping between the two orderings. After constructing the S matrix using this representation, the A and R arrays are represented only as values arrays. They share the row and column indices arrays with

the structure for S. In our implementation, we compute a row of the full similarity matrix at a time. Then, we threshold the values and use a compact operation to move the significant elements to the the COO structure.

To implement the BAG structure, the data points are mapped to the unit hypercube, discretized, converted to Morton codes, and sorted based on such codes. Then, the similarity matrix is constructed to include only similarity values between points that are at most h elements apart on the final SFC order, where h is 128 in our implementation. We refer to the value $2h$ as the neighborhood size. The similarity matrix S is represented as a 2D array with $2h + 1$ rows and N columns, as shown in Figure 7.4. Column i of the matrix contains similarity values between element i and elements from $i - h$ to $i + h$ in order. The h^{th} row of the matrix contains the preference values. The responsibility and availability matrices, R and A, are constructed to have the same size and structure of the similarity matrix S.

7.5 Experimental Results

We implemented the Affinity Propagation on CUDA using both the COO representation and our BAG representation, for the similarity matrix. We also implemented a version for the CPU based on the COO representation. We conducted our experiments on randomly generated point sets. The number of points in these sets ranged from $1K$ to $512K$. We used an NVIDIA Tesla C1060 compute card, which has 240 core processors and 4GB RAM, installed on an Intel Xeon 3.2 GHz workstation with 3GB RAM running 32-bit Windows XP with SP3. We used CUDA version 2.2 for

our experiments. We used the CUDA Parallel Primitives Library (CUDPP) [1] in all scan and segmented scan operations.

Due to the limit on grid dimensions, we were not able to experiment with more than $128K$ points with the COO-GPU implementation. This problem arises only with the COO representation since we use scan operations in its implementation. The scan operation in CUDPP creates one thread for every 4 elements of the input array (the similarity matrix values in this case), which results in too many threads required to process the $256K$ points case and beyond. While this issue can be fixed by modifying the kernel functions for segmented scans in CUDPP, we cannot run this implementation with more than $256K$ points anyways because of the memory requirement of the COO representation exceeds the size of the device memory in this case. For the CPU representation, we were not able to run the experiment beyond $128K$ points either due to the extremely long time it requires. We used the negative sum of squared differences as the kernel (here similarity) function. The preference value was set to the mean similarity over the elements kept in the matrix representation in use. We set the maximum number of iteration for the AP to 2000. The neighborhood size was fixed at 256 for the BAG representation, which means the bandwidth of the resulting band matrix is 257. For the COO representation, we retain values above some threshold. To have a fair comparison, we select the threshold value to obtain approximately the same number of elements in the BAG representation. We compute this threshold based on a random selection of one million pairs of points.

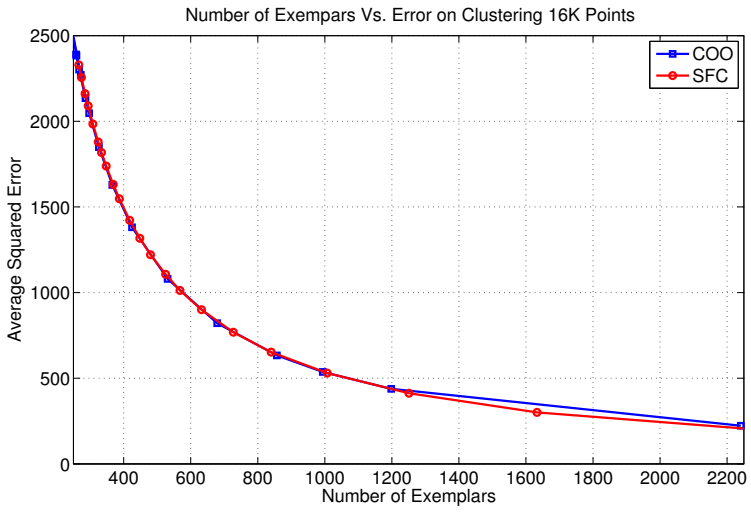


Figure 7.5: This plot compares the average squared error (distance between each point and its assigned exemplar), of the clustering obtained by affinity propagation on the COO and BAG representations, as a function of the number of exemplars. In both cases, the number of points is fixed at 16K. The plot shows how the usage of the approximate BAG sparse representation does not affect the clustering performance of affinity propagation.

7.5.1 Error Versus Number of Exemplars

The BAG representation is an approximation to the sparse kernel matrix, which is in turn an approximation to the full matrix. In this experiment, we want to assess how much the performance of the affinity propagation is affected by using the BAG representation, rather than the COO representation, in terms of the clustering error. We do not compare to the performance using the full matrix representation since the size of such a matrix is prohibitively huge and computing its elements upon need is prohibitively computationally expensive.

The clustering error is measured as the average square distance between each

point and its assigned exemplar, $\frac{1}{N}\sum_i s(i, e_i)$, where e_i is the index of the exemplar assigned to point i. The closer a point on average to its exemplar the better the clustering. However, we cannot use this measure without referring to the number of exemplars since increasing the number of exemplars reduces this measure. In Figure 7.5, we show the clustering error with changing the number of exemplars. In this experiment, we fix the number of points to $16K$ and change the preference value to obtain different points on the curve. We compare between the two sparse matrix representations. The plot clearly shows that the difference between the two representations is negligible in terms of clustering error. Therefore, the approximation introduced by the BAG representation does not have any negative effect on the AP algorithm.

7.5.2 Time Versus Number of Points

In this set of experiments, we measure the computational time versus the number of points. We vary the number of points from $1K$ to $512K$, except with the COO representation on both CPU and the GPU where the maximum is $128K$. Figure 7.6 compares between the three implementations based on the convergence time of AP clustering. The time complexity of the three implementations grow almost linearly with the number of points. Since we fix the neighborhood size, this is consistent with the theoretical complexity of the algorithm, which is linear in the number of similarity values used (quadratic in the number of points for a full matrix representation). Most of the time all the implementations run until the maximum number of

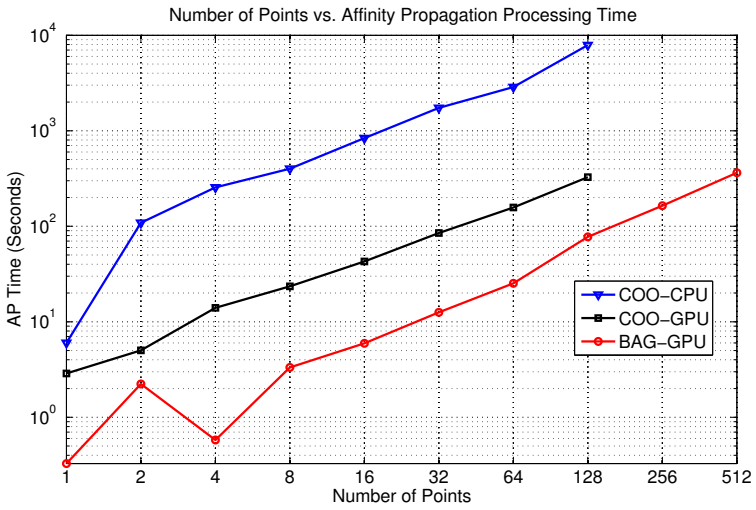


Figure 7.6: This plot compares the running times of affinity propagation, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the number of input points. The number of points shown is in units of K (1024). The times shown do not include the time to construct the similarity matrix from the input points. Neither the COO-GPU nor COO-CPU implementations handles more than 128K points. The COO-GPU version achieves up to 18x speedup, while the BAG-GPU version achieves up to 114x speedup over the CPU implementation.

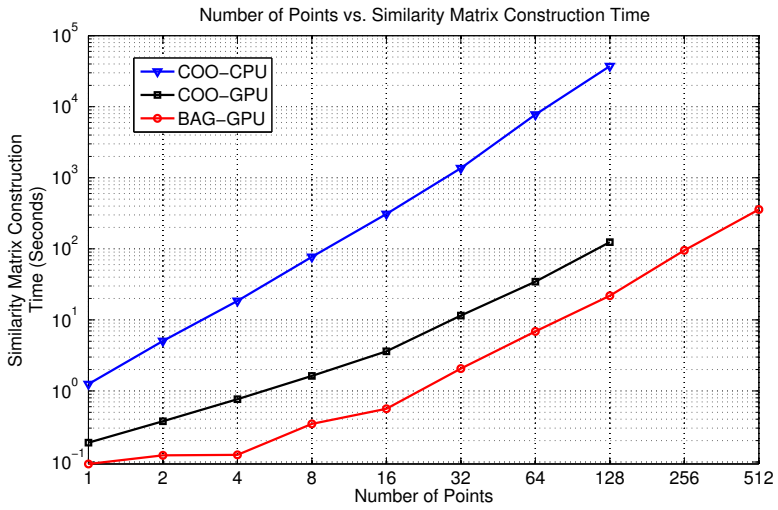


Figure 7.7: This plot compares the times of constructing the similarity matrix, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the number of input points. The number of points shown is in units of K (1024). Neither the COO-GPU nor COO-CPU implementations handles more than 128K points. The COO-GPU implementation achieves up to 300x speedup, while the BAG-GPU implementation achieves up to 1700x speedup.

iterations, 2000. The few exceptions for this are the points that significantly deviate from the linear trend in the plots, which are the $1K$ and $4K$ points on the BAG-GPU curve and the $1K$ point on the COO-CPU curve. Excluding these points, the two GPU implementations consistently outperform the CPU implementation, with up to 18x speedup for the COO representation and up to 114x speedup for the BAG representation. Figure 7.7 shows the times to construct the similarity matrix representations for the same set of experiments. Since in the COO representation we need to compute all elements of the similarity matrix to compare them to the threshold, the construction of the similarity matrix on the CPU becomes the computational bottleneck as the number of points increase, while the GPU implementations are less affected due to parallelism. The GPU implementations score larger speedups in this part of the computation than the AP part, with the COO achieving up to 300x speedup, and the BAG achieving up to 1700x speedup. The simplicity of the BAG representation is the key to this tremendous speedup.

7.5.3 Time Versus Dimensionality

In this experiment, we study the effect of point dimensionality on the time to construct the kernel matrix representation. We fix the number of points at $16K$ points. We change the point dimensionality from 32 to 512. Figure 7.8 shows the results of this experiment. The advantage of using the GPU becomes more evident when the dimensionality increases. At 512 dimensions, both GPU implementations are about 1000 times faster than the CPU. The BAG representation is at least two times faster

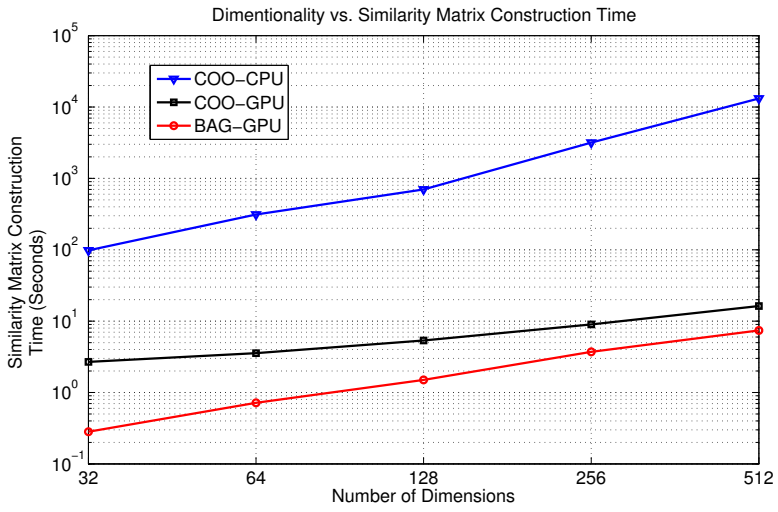


Figure 7.8: This plot compares the times of constructing the similarity matrix, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the points dimensionality. As the dimensionality grows, the two GPU implementations achieve around 1000x speedup compared to the CPU implementation.

than the COO representation on the GPU. This again emphasizes the advantage of having a simple representation, such as the BAG over a complex representation such as the COO.

7.6 Conclusion and Future Work

We presented a novel method to construct a band approximation to Gram matrices, based on space filling curves. The proposed method is very simple to construct and efficient to work with on modern graphics processing units than the conventional Coordinate (COO) representation. We applied the new representation to Affinity Propagation, a recently introduced unsupervised clustering algorithm. Our results show a significant speedup, of up to 114x, when using our algorithm on the GPU

compared to the CPU implementation, compare to 18x speedup when using the COO representation. If we include the time to construct the sparse matrix structure, the speedup jumps up to 330x. This speedup does not come at any expense in terms of the clustering performance of the AP algorithm.

There are many interesting experiments to be conducted on our work, such as studying the effect of neighborhood size on the time and clustering performance of the algorithm, and studying the approximation error to the kernel matrix incurred by our representation compared to the COO representation. Nevertheless, enabling large scale clustering via an effective algorithm such as Affinity Propagation is by itself an important achievement. We are planning on apply this method to real world large scale machine learning applications. Given the success on AP, we are encouraged to investigate the applicability of our approach to other kernel methods, such as SVMs. We are also investigating other types of codes that can be used to order input points other than space filling curves.

Chapter 8

Future Work

At the end of each preceding chapter, we provided our vision for future research directions in the presented topic. In this chapter, we provide our vision for long term research based on the findings of this dissertation.

In summary, the thesis presented research in two categories. The first is human detection for video surveillance and smart vehicle systems. The second is vision computing on Graphics Processing Units (GPUs). The presented research in human detection covered several aspects of the problem. Particulary, it covered feature extraction, classifier training, and evaluation of detection algorithms. The presented research in vision computing on GPUs included development of efficient implementations for Graph Cut and an approach for kernel based learning on GPUs.

The presented research in this thesis shapes our vision for future research. We observed from the research in human detection the power of training classifiers with large datasets. This observation is supported by findings of other researchers [67, 91]. From our research in vision computing on GPUs, we realized the power of parallel computing on GPUs and how it can be harnessed to solve large scale computer vision problems. These are the two ingredients of our future research, which is about leveraging the power of parallel computing to enable large scale learning for computer vision applications.

With the popularity of image and video sharing over the Internet, and the widespread availability of high resolution digital cameras, an important resource for training computers how to see has become available. Much of this data is tagged with keywords that are useful for training. On top of that, researchers have collected large sets of images and developed online annotation tools to obtain rich information about these images either by volunteer annotators [79], annotators for enjoyment [99], or paid annotators [24]. Having this data available, it remains to use them in training for computer vision applications.

Fortunately, this explosion of data is accompanied by an explosion in parallel computing devices and systems. Modern graphics processing units have up to 240 cores on chip [69]. The number of cores are expected to grow according to Moore's law. Cluster and cloud computing have also become common in computing. Nevertheless, there is a wide gap between the available data and computing power on one side, and the algorithms that make the best use of them on the other side. Our future research aims at closing this gap for learning problems with a focus on applications in computer vision. Our goal is to push the state of the art in computer vision research through harnessing the availability of large scale datasets and the power of parallel computing. We believe that this research direction is not only important and interesting, but, it is also necessary for the advancement of the field.

Bibliography

[1] CUDPP: CUDA Data-Parallel Primitives Library.

[2] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive digital photomontage. In *ACM SIGGRAPH*, pages 646–653, 2004.

[3] Richard J. Anderson and J.C. Setubal. On the parallel implementation of goldbergs maximum flow algorithm. In *4th Ann. Symp. Parallel Algorithms and Architectures (SPAA-92)*, pages 168 – 177, 1992.

[4] David A. Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, 2005.

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European Conference on Computer Vision (ECCV)*, 2006.

[6] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA, December 2008.

[7] Blelloch. Prefix sums and their applications. In *John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann*. 1993.

[8] Avrim Blum and Shuchi Chawla. Learning from labeled and unlabeled data using graph mincuts. In *18th International Conference on Machine Learning*, pages 19–26. Morgan Kaufmann, San Francisco, CA, 2001.

[9] Yuri Boykov and Marie-Pierre Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images. In *International Conference on Computer Vision (ICCV)*, 2001.

[10] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1124–1137, 2004.

[11] Yuri Boykov, Ogla Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1222–1239, 2001.

[12] A. R. Butz. Alternative algorithm for hilbert space filling curve. *IEEE Trans. on Computers*, 20:424–42, April 1971.

[13] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML*, 2008.

[14] J. Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18, 1989.

[15] Boris Cherkassky and Andrew Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[16] Vladimir Cherkassky and Filip Mulier. *Learning from Data: Concepts, Theory, and Methods*. Wiley, 1998.

[17] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition edition, 2001.

[18] Franklin Crow. Summed-area tables for texture mapping. *Computer Graphics (ACM SIGGRAPH)*, 18(3):207–212, 1984.

[19] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th national conference*, 1969.

[20] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* San Diego, CA, pages 886–893, 2005.

[21] Navneet Dalal, Bill Triggs, and Cordelia Schmid. Human detection using oriented histograms of flow and appearance. In *Proc. European Conf. on Computer Vision,* Graz, Austria, page 428441, 2006.

[22] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *International Conference on Machine Learning (ICML)*, pages 233–240, 2006.

[23] Leonard Michael Delves and Joan Walsh, editors. *Numerical Solution of Integral Equations*. Oxford University Press, 1974.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 1–8, Miami Beach, FL, June 2009.

[25] Nandan Dixit, Renaud Keriven, and Nikos Paragios. GPU-Cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction. Technical report, CERTIS, 2005.

[26] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[27] Petros Drineas and Michael W. Mahoney. Approximating a gram matrix for improved kernel-based learning. In *in Proceedings of the 18th Annual Conference on Learning Theory, 2005*, pages 323–337, 2005.

[28] Ahmed Elgammal, Ramani Duraiswami, and Larry Davis. Efficient computation of kernel density estimation using fast gauss transform with applications for segmentation and tracking. In *Second International Workshop on Statistical and Computational Theories of Vision*, 2001.

[29] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[30] Pedro Felzenszwalb and Daniel Huttenlocher. Pictorial structures for object recognition. In *Int'l J. of Computer Vision*, volume 61, pages 55–79, 2005.

[31] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* , Anchorage, AK, pages 1–8, 2008.

[32] Robert Fergus, P. Perona, and Andrew Zisserman. Object class recognition by unsupervised scale-invariant learning. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* Madison, WI, 2003.

[33] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[34] Brendan Frey and Delbert Dueck. Mixture modeling by affinity propagation. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 379–386. MIT Press, Cambridge, MA, 2006.

[35] Brendan J. Frey and Delbert Dueck. Clustering by Passing Messages Between Data Points. *Science*, 315:972–976, 2007.

[36] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, 28:337407, 2000.

[37] Michael Garland. Sparse matrix computations on manycore GPU's. In *Annual ACM IEEE Design Automation Conference*, 2008.

[38] D.M. Gavrila and V. Philomin. Real-time object detection for smart vehicles. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* Fort Collins, CO, pages 87–93, 1999.

[39] Andrew Goldberg. *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, MIT, 1987.

[40] Andrew Goldberg and Robert Tarjan. A new approach to maximum-flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.

[41] Ismail Haritaoglu, David Harwood, and Larry Davis. w^4: Real-time surveillance of people and their activities. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):809–830, 2000.

[42] Paul Heckbert. Filtering by repeated integration. *ACM SIGGRAPH*, 20(4):315–321, 1986.

[43] Bernd Heisele, Purdy Ho, and Tomaso Poggio. Face recognition with support vector machines: global versus component-based approach. In *In Proc. 8th International Conference on Computer Vision*, pages 688–694, 2001.

[44] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *EUROGRAPHICS*, 24(3):547–555, 2005.

[45] Mohamed Hussein and Wael Abd-Almageed. Approximate kernel matrix computation on GPUs for large scale learning applications. In *ACM International Conference on Supercomputing*, pages 511–512, Yorktown Heights, NY, June 2009.

[46] Mohamed Hussein and Wael Abd-Almageed. Efficient band approximation of gram matrices for large scale kernel methods on GPUs. In *Supercomputing*, Portland, OR, November 2009.

[47] Mohamed Hussein, Wael Abd-Almageed, Yang Ran, and Larry Davis. A real-time system for human detection, tracking and verification in uncontrolled camera motion environments. In *IEEE International Conference on Computer Vision Systems*, 2006.

[48] Mohamed Hussein, Fatih Porikli, and Larry Davis. Kernel integral images: A framework for fast non-uniform filtering. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* , Anchorage, AK, pages 1–8, June 2008.

[49] Mohamed Hussein, Fatih Porikli, and Larry Davis. A comprehensive evaluation framework and a comparative study for human detectors. *IEEE Transactions on Intelligent Transportation Systems*, 10(3):417–427, 2009. Invited Paper.

[50] Mohamed Hussein, Fatih Porikli, and Larry Davis. Object detection via boosted deformable features. In *IEEE International Conference on Image Processing*, Cairo, Egypt, November 2009.

[51] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on CUDA. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[52] S. Ioffe and D. A. Forsyth. Probabilistic methods for finding people. *Int'l J. of Computer Vision*, 43(1):45–68, 2001.

[53] Peter Kipfer and Rudiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPU Gems2*, pages 733–746. Addison Wesley, 2005.

[54] Vladimir Kolmogorov and Ramin Zabih. Multi-camera scene reconstruction via graph cuts. In *ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part III*, pages 82–96, London, UK, 2002. Springer-Verlag.

[55] Vladimir Kolmogrov and Ramin Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:147–159, 2004.

[56] Vivek Kwatra, Arno Schodl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. In *SIGGRAPH*, pages 277–286, 2003.

[57] B. Leibe, E. Seemann, and B. Schiele. Pedestrian detection in crowded scenes. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* San Diego, CA, volume 1, pages 878–885, 2005.

[58] Michael Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A fast similarity join algorithm using graphics processing units. In *IEEE International Conference on Data Engineering*, 2008.

[59] Zhe Lin and Larry Davis. A pose-invariant descriptor for human detection and segmentation. 2008.

[60] Zhe Lin, Larry Davis, David Doermann, and Daniel DeMenthon. Hierarchical part-template matching for human detection and segmentation. In *Proc. 10th Intl. Conf. on Computer Vision,* Rio de Janeiro, Brazil, 2007.

[61] David Lowe. Distinctive image features from scale-invariant keypoints. *nternational Journal of Computer Vision*, 60:91–110, 2004.

[62] Juwei Lu, K.N. Plataniotis, and A.N. Venetsanopoulos. Face recognition using kernel direct discriminant analysis algorithms. *Neural Networks, IEEE Transactions on*, 14(1):117–126, Jan 2003.

[63] K. Mikolajczyk, B. Leibe, and B. Schiele. Multiple object class detection with a generative model. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* New York, NY, volume 1, pages 26–36, 2006.

[64] K. Mikolajczyk, C. Schmid, and A. Zisserman. Human detection based on a probabilistic assembly of robust part detectors. In *Proc. European Conf. on Computer Vision,* Prague, Czech Republic, volume 1, pages 69–81, 2004.

[65] A. Mohan, C. Papageorgiou, and T. Poggio. Example-based object detection in images by components. *IEEE Trans. Pattern Anal. Machine Intell.*, 23(4):349–360, 2001.

[66] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[67] S. Munder and D. M. Gavrila. An experimental study on pedestrian classification. *IEEE Trans. Pattern Anal. Machine Intell.*, 28(11):1863–1868, November 2006.

[68] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[69] NVIDIA. *NVIDIA CUDA Programming Guide Version 2.2*, April 2009.

[70] 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.

[71] Andreas Opelt, Axel Pinz, and Andrew Zisserman. A boundary-fragment-model for object detection. In *European Conf. on Computer Vision*, pages 575–588, Graz, Austria, 2006.

[72] P. Papageorgiou and T. Poggio. A trainable system for object detection. *Int'l J. of Computer Vision*, 38(1):15–33, 2000.

[73] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[74] Fatih Porikli. Integral histogram: A fast way to extract histograms in cartesian spaces. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* San Diego, CA, pages 829–836. IEEE Computer Society, 2005.

[75] Yang Ran, Isac Weiss, Qinfen Zheng, and Larry Davis. Pedstrian detection via periodic motion analysis. *Int'l J. of Computer Vision*, 71(2):143–160, February 2007.

[76] R. Ronfard, C. Schmid, and B. Triggs. Learning to parse pictures of people. In *Proc. European Conf. on Computer Vision,* Copehagen, Denmark, volume 4, pages 700–714, 2002.

[77] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "grabcut": interactive foreground extraction using iterated graph cuts. volume 23, pages 309–314, New York, NY, USA, 2004. ACM.

[78] Sebastien Roy and Ingemar Cox. A maximum-flow formulation of the n-camera stereo correspondence problem. In *Proceedings of the Sixth International Conference on Computer Vision (ICCV)*, page 492, Washington, DC, USA, 1998. IEEE Computer Society.

[79] Bryan Russell and Antonio Torralba. Labelme: a database and web-based tool for image annotation. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 77:157–173, 2008.

[80] Yousef Saad. *SPARSKIT: A basic tool kit for sparse computations*, June 1994.

[81] P. Sabzmeydani and G. Mori. Detecting pedestrians by learning shapelet features. In *CVPR07*, pages 1–8, 2007.

[82] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics, 2006.

[83] R.E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37:297–336, 1999.

[84] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998.

[85] M. Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(2), 2004.

[86] Matthias Seeger. Bayesian model selection for support vector machines, Gaussian processes and other kernel classifiers. In S. A. Solla, , T. K. Leen, and K. R. Müller, editors, *NIPS*, pages 603–609. MIT Press, 2000.

[87] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John Owens. Scan primitives for GPU computing. In *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2007.

[88] Y. Shiloach and U. Vishkin. An $o(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[89] A. Smola. Regression estimation with support vector learning machines. Master's thesis, Technische Universitat Munchen, 1996.

[90] Alex J. Smola and Bernhard Schökopf. Sparse greedy matrix approximation for machine learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 911–918, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[91] Antonio Torralba, Rob Fergus, and Yair Weiss. Small codes and large image databases for recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 1–8, Miami Beach, FL, June 2008.

[92] Duan Tran and David Forsyth. Configuration estimates improve pedestrian finding. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *NIPS*, pages 1529–1536, Cambridge, MA, 2008.

[93] Oncel Tuzel, Fatih Porikli, and Peter Meer. Region covariance: A fast descriptor for detection and classification. In *Proc. European Conf. on Computer Vision,* Graz, Austria, volume 2, pages 589–600, 2006.

[94] Oncel Tuzel, Fatih Porikli, and Peter Meer. Human detection via classification on riemannian manifolds. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[95] V. Vapnik. *The nature of Statistical Learning Theory.* Springer-Verlag, 1995.

[96] Vibhav Vineet and P. J. Narayanan. CUDA Cuts: Fast graph cuts on the GPU. In *Computer Vision and Pattern Recognition Workshops*, 2008.

[97] P. Viola, M. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* New York, NY, volume 1, pages 734–741, 2003.

[98] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* Kauai, HI, page 511518, 2001.

[99] Luis von Ahn. Games with a purpose. *IEEE Computer Magazine*, pages 96–98, 2006.

[100] C. K. I. Williams and M. Seeger. The effect of the input density distribution on kernel-based classifiers. In *International Conference on Machine Learning*, 2000.

[101] C. K. I. Williams and M. Seeger. Using nystrom method to speed up kernel machines. In T. K. Leen, T. G. Diettrich, and V. Tresp, editors, *NIPS*, volume 13. MIT Press, 2001.

[102] Bo Wu and Ram Nevaita. Optimizing discrimination-efficiency tradeoff in integrating heterogeneous local features for object detection. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* , Anchorage, AK, pages 1–8, 2008.

[103] Bo Wu and Ram Nevatia. Detection of multiple, partially occluded humans in a single image by bayesian combination of edgelet part detectors. In *Proc. 10th Intl. Conf. on Computer Vision,* Beijing, China, page 9097, 2005.

[104] Wen Wu and Jie Yang. SmartLabel: An object labeling tool using iterated harmonic energy minimization. In *ACM Multimedia*, 2006.

[105] Ying Wu, Ting Yu, and Gang Hua. A statistical field model for pedestrian detection. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* San Diego, CA, 2005.

[106] Liang Zhao and Larry S. Davis. Closely coupled object detection and segmentation. In *ICCV*, pages 454–461, 2005.

[107] Liang Zhao and Chuck Thorpe. Stereo and neural network-based pedestrian detection. *IEEE Transactions on Intelligent Transportation Systems*, 1(3):148 –154, September 2000.

[108] Shaohua Kevin Zhou, Rama Chellappa, and Baback Moghaddam. Visual tracking and recognition using appearance-adaptive model in particle filters. *IEEE Transactions on Image Processing*, 13(11):1491–1506, November 2004.

[109] Qiang Zhu, Shai Avidan, Mei-Chen Yeh, and Kwang-Ting Cheng. Fast human detection using a cascade of histograms of oriented gradients. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* New York, NY, volume 2, pages 1491 – 1498, New York, June 2006.