ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

FAST FACE DETECTION AND RECOGNITION ON GRAPHICS PROCESSING UNITS

M.Sc. THESIS

Salih Cihan TEK

Department of Computer Engineering

Computer Engineering Programme

JUNE 2012

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

FAST FACE DETECTION AND RECOGNITION ON GRAPHICS PROCESSING UNITS

M.Sc. THESIS

Salih Cihan TEK
(504081510)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Prof. Dr. Muhittin GÖKMEN

JUNE 2012

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GRAFİK İŞLEMCİLER ÜZERİNDE HIZLI YÜZ SAPTAMA VE TANIMA

YÜKSEK LİSANS TEZİ

Salih Cihan TEK
(504081510)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Muhittin GÖKMEN

HAZİRAN 2012

Salih Cihan TEK, a **M.Sc.** student of ITU **Graduate School of Science Engineering and Technology** 504081510, successfully defended the **thesis** entitled "**FAST FACE DETECTION AND RECOGNITION ON GRAPHICS PROCESSING UNITS**", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. Muhittin GÖKMEN** İstanbul Technical University	....
Jury Members :	**Asst. Prof. Dr. Mustafa KAMAŞAK** İstanbul Technical University	....
	Asst. Prof. Dr. İlker BAYRAM İstanbul Technical University	....

Date of Submission : 2 May 2012
Date of Defense : 8 June 2012

FOREWORD

I would like to express my gratitude to my thesis advisor Prof. Dr. Muhittin Gökmen for his interest, guidance and encouragement during the development of this thesis. I would like to thank my family for always being there for me and supporting all of my endeavours throughout the years. I am also grateful to my friends at the CVIP Lab, especially to Birkan Tunç, Volkan Dağlı, Evangelos Sarıyanidi, Bora Akaydın and Mighty Itauma for their friendship, support and valuable comments during my research.

I would like to thank to The Scientific and Technological Research Council of Turkey (TÜBİTAK) for its financial support via the National Scholarship Programme for M.Sc. Students and via the research project 109E268. Finally, I would like to thank the Scientific Research Project Unit of Istanbul Technical University (İTÜ-BAP) for providing financial support for the acquisition of the hardware required for the development of this thesis under the grant number 34120.

June 2012 Salih Cihan TEK

TABLE OF CONTENTS

ABBREVIATIONS

API	: Application Programming Interface
CPU	: Cental Processing Unit
CUDA	: Compute Unified Device Architecture
FLOPS	: Floating-point operations per second
GPGPU	: General-Purpose Computing on Graphics Processing Units
GPU	: Graphics Processing Unit
HLSL	: High Level Shader Language
LBP	: Local Binary Patterns
LPP	: Locality Preserving Projections
MCT	: Modified Census Transform
NVCC	: NVIDIA Compiler Collection
OpenCL	: Open Computing Language
SIMD	: Single Instruction Multiple Data
SIMT	: Single Instruction Multiple Thread
SM	: Streaming Multiprocessor
SVM	: Support Vector Machine

LIST OF TABLES

LIST OF FIGURES

FAST FACE DETECTION AND RECOGNITION ON GRAPHICS PROCESSING UNITS

SUMMARY

Real-time face detection and recognition have been very active topics of research in the last decade. The main reason of this interest on these subjects is the number of their possible real-world applications both in commercial and non-commercial systems. Most of the complicated real-world applications like virtual reality, traffic and urban surveillance, video conferencing, robotics and entertainment systems make use of these algorithms at some point. Nearly all of these applications require the system to be able to run on real-time video streams. Therefore for these kind of applications, the speed of the algorithms used in the system is as important as the accuracy.

In the last decade, researchers have found faster and better face detection and recognition algorithms suitable for real-world applications. Even though these algorithms are fast, they are still not fast enough to run in real time in some cases. For example, even the fastest face detection algorithms developed to date are not fast enough to run in real-time on video streams having a high resolution as 1280×720 or above, which have become increasingly common, unless all processing power is dedicated for that task. For a system involving additional algorithms, like facial feature detection, face alignment and face recognition, it is not possible to dedicate all processing power to the face detection task. Therefore, in order to make these algorithms run on such high resolution video streams in real-time, one has to either sacrifice accuracy or use very application specific cues to limit the processing required, which will in turn limit the generalization ability of the system in question. While the same problem exists for the face recognition algorithms, the main problem for them is the number of people in the database and not the resolution. Even though some face recognition algorithms can run in real-time on relatively small database sizes, they are not able to do so on larger databases required by some real-world applications. Speed is a problem not only for applications that process video streams, but also for the ones that process still images. If the number of images that need to be processed is very high, the time needed to complete the processing can quickly become impractical, regardless of the resolution of images.

Considerable amount of effort has been made to speed up core face detection and recognition algorithms by eliminating or modifying some of their steps, but algorithmic modifications by themselves are proven to be insufficient to achieve the drastic speed improvements required.

Another approach to accelerate face detection and recognition algorithms is modifying their structure and developing parallelized versions of them. This can be done either on a CPU (Central Processing Unit) by taking advantage of its multiple cores and/or hyper-threading capability if exists, or on a GPU (Graphics Processing Unit) by

using one of the GPGPU (General-Purpose Computing on Graphics Processing Units) frameworks available. Even though it is much easier to develop the parallelized version of an algorithm on a CPU, putting more effort in development and implementing it on a GPU has very important advantages that make up for the additional effort required.

The most important property of a GPU is its ability to execute hundreds of threads concurrently and perform all the scheduling purely on hardware, in contrast to the CPUs that require software scheduling. GPU hardware is optimized for performing computer graphics computations and have excellent floating point performance. A modern GPU, for example a GTX 580, can reach very high computing rates up to 1581 GFLOPS. These properties make GPUs very suitable for demanding image processing, computer vision and pattern recognition algorithms. Also by offloading some compute intensive, parallelizable tasks to GPU, it becomes possible to use the CPU cores for other non-parallelizable tasks that involve less computation and more logic, leading to much more efficient usage of the hardware that already exist in a computer and hence to drastic speed improvements. This approach, which is called heterogeneous computing, is not only a very cost-effective way to implement computationally demanding high-performance algorithms, but also the best way to make these algorithms accessible to common users.

The purpose of this thesis is to present efficient, massively parallel GPU implementations of two different algorithms: A boosting based face detection algorithm that utilizes MCT (Modified Census Transform) based weak classifiers, and a feature based face recognition algorithm that uses weighted regional histograms of LBPs (Local Binary Patterns) as features. All steps of these algorithms are parallelized in a GPU firendly manner and efficient GPU implementations of them are given using the CUDA (Compute Unified Device Architecture) platform of NVIDIA. Some alternative methods for parallelization on the GPU and the problems with them are also discussed. The GPU implementations are further extended to utilize multiple GPUs.

For the sake of comparison, single and multi-threaded CPU implementations of the same algorithms are developed and compared with their corresponding GPU implementations both in terms of speed and accuracy. These comparisons showed that the GPU implementations, while generating the exact same results, run much faster than the CPU implementations, proving that a GPU is more suitable for executing these algorithms than a CPU.

For the face detection algorithm, comparisons are made both on video streams having 5 different input resolutions and on still images from the MIT+CMU frontal face detection test set. The implementations of the face recognition algorithm are compared for different feature vector lengths and database sizes using the images from the FERET database generated by the CSU Face Identification Evaluation System. It is observed that the difference between the speed of the GPU and CPU implementations increases as the resolution gets higher, feature vectors get longer or database size gets larger. In other words, the advantage of using GPUs became more apparent as the amount of data processed by the application got larger.

GRAFİK İŞLEMCİLER ÜZERİNDE HIZLI YÜZ SAPTAMA VE TANIMA

ÖZET

Gerçek zamanda yüz saptama ve tanıma, son yıllarda üzerinde en çok çalışılan konular arasındadır. Konulara karşı olan bu ilginin sebebi gerek ticari alanda, gerek ticari olmayan alandaki uygulamalarının fazlalığıdır. Sanal gerçeklik, trafik-yol gözlem ve güvenlik, görüntülü konferans, robotik ve eğlence sistemleri gibi bir çok uygulama da belli bir bölümlerinde yüz saptama ve/veya saptama algoritmalarından yararlanmaktadır. Bu örnek uygulamlardan büyük çoğunluğunun sabit bir resimden çok, hareketli bir video görüntüsü üzerinde gerçek zamanlı olarak çalışması gerekmektedir. Bu da söz konusu olan bu algoritmalar için hızın da en az başarım kadar önemli olduğu anlamına gelmektedir.

Son yıllarda araştırmacılar, uygulama geliştiricilerinin gerçek hayattaki bir problemi çözmek amacıyla kullanabilmeleri için daha hızlı ve daha yüksek başarımlı yüz saptama ve tanıma algoritmaları geliştirmişlerdir. Bu algoritmalar her ne kadar hızlılarsa da, bazı durumlarda gerçek zamanlı çalışamamaktadırlar. Örneğin yüz saptama algoritmaları günümüzde ucuzlaşan hızlı donanım fiyatlarının da etkisiyle kullanımı sıradan bilgisayar kullanıcıları arasında bile fazlasıyla yaygınlaşmış olan 1280×720 ve 1920×1080 gibi yüksek çözünürlüklerde ancak çok hızlı bir işlemcinin tüm işlem hesaplama gücü bu işlem için ayrılırsa gerçek zamanlı olarak çalışabilmektedir. Eğer birden fazla algoritmanın bir arada kullanılmasını gerektiren, örneğin yüz saptamaya ek olarak yüzdeki nitelikleri saptama, hizalama ve yüz tanıma gibi birden fazla algoritmayı içeren bir sistem söz konusu olduğunda, işlemcinin tüm hesaplama gücünün yüz saptama işlemi için ayrılması söz konusu değildir. Böyle durumlarda gerçek zamanlı çalışma elde edebilmek için ya başarımdan feragat edilmesi, ya da sistemin genelleştirilme kabiliyetini kısıtlayacak uygulamaya yönelik ip uçlarının kullanılması gerekmektedir. Yüz tanıma algoritmaları için en büyük problem ise veritabanındaki kişi sayısıdır. Günümüzde kullanılan bazı yüz tanıma algoritmaları her ne kadar az sayıda kişi içerden veri tabanlarında gerçek zamanda çalışabileseler de, gerçek hayattaki uygulamaların bazılarının gerektirdiği çok sayıda kişi içeren büyük veritabanları üzerinde gerçek zamanlı olarak çalışamamaktadırlar.

Yüz saptama ve tanıma algoritmaları için hız, sadece canlı görüntüler ya da videolar üzerinde gerçek-zamanlı çalışması gereken sistemler için değil, sabit görüntüler üzerinde çalışan uygulamalar için de sorun olabilmektedir. Örneğin çok sayıda yüksek çözünürlüklü görüntünün işlenmesini gerektiren bir uygulamada, işlemin saatler, hatta günler sürmesi bile söz konusudur. Böyle durumlarda algoritmaların hızlarının olabildiğince yüksek olması, uygulamanın pratikte kullanılabilirliği açısından çok büyük önem taşımaktadır. Bugüne kadar yüz saptama ve tanıma algoritmalarını hızlandırabilmek için önemli miktarda çaba sarfedilmiş olsa da,

algoritmik değişikliklerin tek başına önemli miktarda hızlanmayı sağlayabilmek için yeterli olmadığı görülmüştür.

Yüz saptama ve tanıma algoritmalarını hızlandırmak için başka bir yöntem de algoritmaların yapısını değiştirerek paralelleştirmektir. Bu, varsa çekirdekleri/çoklu iplik desteği kullanılarak bir MİB (Merkezi işlem birimi) üzerinde yapılabileceği gibi, mevcut olan GAGİB (Grafik işlem birimleri üzerinde genel amaçlı işlem) platformları yardımıyla bir GİB (Grafik işlem birimi) üzerinde yapılabilir.

Günümüzde işlem yükü fazla olan bir çok uygulama, çok çekirdekli MİB'ler üzerinde tüm çekirdekleri birden kullanacak şekilde yazılmaktadır. Ancak MİB ile birlikte GİB kullanan uygulamaların sayısı ise çok azdır. Bunun sebebi MİB üzerinde, çok iplikli uygulama geliştirmenin, GİB üzerinde uygulama geliştirmeye kıyasla çok daha kolay olmasıdır. Her ne kadar bir algoritmanın paralelleştirilmiş halini MİB üzerinde geliştirmek çok daha kolay olsa da, daha fazla çaba harcayıp algoritmayı GİB üzerinde çalışacak şekilde yazmanın harcanan fazladan vakti fazlasıyla karşılayacak kadar avantajı vardır.

Bir GİB'in en önemli özelliği aynı anda yüzlerce ipliği paralel olarak çalıştırabilmesi ve MİB'nin aksine tüm zamanlama işlemlerini donanımsal olarak yapmasıdır. Bu özelliği bir GİB içerisinde iplik yaratma, silme, başlatma ve durdurma gibi işlemlerin görmezden gelinebilecek kadar kısa sürelerde yapılabilmesini sağlamaktadır. GİB mimarisi bilgisayarla grafik işlemleri yapmak üzere tasarlanmıştır ve bu yüzden kayar noktalı sayılarla işlem performansı çok yüksektir. Modern bir GİB, örneğin bir GTX 580, 1581 GFLOPS gibi yüksek işlem hızlarına ulaşabilmektedir. Bu işlem kapasitesi, her yeni nesilde önemli miktarda yükselmektedir. Bunlara ek olarak, görüntüler üzerinde lineer interpolasyon ve adres sınırlama gibi bazı işlemleri yapabilmek için adanmış donanımlara sahip olmaları, görüntü yeniden boyutlandırma gibi işlemlerle bazı ikili işlemleri çok hızlı bir şekilde yapabilmelerini sağlamaktadır.

Bu özellikleri GİB'leri işlem yükü ağır olan görüntü işleme, bilgisayarla görü ve örüntü tanıma algoritmaları için çok uygun hale getirmektedir. Ayrıca matematiksel işlem ağırlıklı, paralelleştirilebilir olan işlemleri GİB'e yüklemek, MİB'i diğer, daha az matematiksel, daha çok mantıksal işlem ve bellek erişimi gerektiren, seri olarak yapılması gereken işlemler için kullanmaya olanak vermektedir. Bu şekilde hem MİB hem de GİB'e kendilerine en uygun olan tipte işlemlerin yaptırılmasıyla, bilgisayar içerisinde bulunmakta olan donanımlar daha verimli şekilde kullanılmakta ve çok büyük performans artışları elde edilmektedir. Modern GİB'lerin de günümüzde çoğu bilgisayarda bulunduğu göz önüne alınırsa, bu yöntemle MİB'lerde çalışması çok uzun süren bazı uygulamalar sıradan kullanıcılar için de bilgisayardaki tüm işlem gücünün kullanılması sayesinde kullanılabilir hale gelmektedir.

Bu tezin amacı iki farklı algoritmanın verimli ve yoğun bir şekilde paralelleştirerek bir GİB mimarisi üzerinde çalışacak şekilde nasıl gerçeklenebileceğini göstermektir. Ele alınan algoritmalardan ilki MCT (Değiştirilmiş Census Dönüşümü) temelli zayıf sınıflandırıcılar kullanan iteleme temelli bir yüz saptama algoritmasıdır. Diğeri ise bölgesel YİD (Yerel İkili Desenler) niteliklerinin ağırlıklandırılmış histogramlarını kullanan nitelik temelli bir yüz tanıma algoritmasıdır. Her iki algoritmanın da tüm adımları bir GİB'in mimarisine ve bu mimarinin dayattığı kurallara uygun şekilde paralleleştirilmiş ve NVIDIA tarafından geliştirilmiş olan CUDA (Compute Unified Device Architecture) platformu kullanılarak GİB üzerinde gerçeklenmiştir.

Parallelleştirme için alternatif yöntemler üzerinde de durulmuş ve bu yöntemlerdeki problemlerden bahsedilmiştir. Gerçeklenmiş olan uygulamalar daha da geliştirilmiş ve sistem üzerinde birden fazla GİB kullanacak hale getirilmiştir.

Karşılaştırma amacıyla algoritmaların tek ve çok iplikli halleri MİB üzerinde de gerçeklenmiş ve elde edilen sonuçlar ilgili GİB sonuçlarıyla hem başarım hem de hız açısından karşılaştırılmıştır. Yapılan bu karşılaştırmalar, algoritmaların GİB üzerinde çalışan sürümlerinin, MİB üzerinde çalışan sürümleriyle aynı sonuçları ürettiklerini ancak çok daha hızlı olduklarını göstermiştir. Bu da söz konusu olan algoritmalar için GİB kullanmanın MİB kullanmaktan daha mantıklı olduğunu kanıtlar niteliktedir.

Yüz saptama algoritması için karşılaştırmalar hem 5 farklı çözünürlükteki video görüntüleri üzerinde, hem de CMU+MIT önden yüz saptama test veritabanındaki sabit resimler üzerinde yapılmıştır. Yüz tanıma algoritmasının MİB ve GİB versiyonları arasındaki karşılaştırma da farklı nitelik vektörü uzunlukları ve veritabanı boyutları için "CSU Face Identification Evaluation System" tarafından FERET veritabanındaki resimlerden oluşturulmuş olan görüntüler üzerinde yapılmıştır. GİB ve MİB arasındaki hız farkının, çözünürlük yükseldikçe, nitelik vektörleri uzadıkça ve veritabanı büyüdükçe arttığı gözlemlenmiştir. Diğer bir deyişle, GİB kullanmanın avantajının işlenen veri miktarı büyüdükçe daha belirginleştiği görülmüştür. GİB’lerin aynı anda çok büyük miktarda veri üzerinde aynı işlemi yapmak üzere tasarlanmış donanımlar olduğu göz önüne alındığında, bu zaten beklenen bir sonuçtur.

1. INTRODUCTION

Real-time face detection and recognition have been very active topics of research in the last few years. The main reason of this interest on these subjects is the number of their possible real-world applications both in commercial and non-commercial systems. Most of the complicated real-world applications like virtual reality, traffic and urban surveillance, video conferencing, robotics and entertainment systems make use of these algorithms at some point. Nearly all of these applications require the system to be able to run on real-time video streams. Therefore for these kind of applications, the speed of the algorithms used in the system is as important as the accuracy.

In the last decade, researchers have found faster and better face detection and recognition algorithms that are suitable for practical real-world applications. Even though these algorithms are fast, they are not fast enough to run in real time in some cases. For example, even the fastest face detection algorithms developed to date are not fast enough to run in real-time on video streams having a high resolution as 1280×720 or above, which have become increasingly common, unless the whole processing power is dedicated for that task. To use these algorithms on such high resolution video streams in real-time, one either has to sacrifice accuracy or use very application specific cues to limit the processing required, which in turn limits the generalization ability of the system in question. For the face recognition algorithms, the main problem is the number of persons in the database. Even though some face recognition algorithms can run in real-time on relatively small database sizes, they are not able to do so on larger databases required by some real-world applications. Speed is a problem not only for applications that process video streams, but also for the ones that process still images. If the number of images that need to be processed is very high, the time needed to complete the processing can quickly become impractical, regardless of the resolution of images.

Considerable amount of effort has been made to speed up core face detection and recognition algorithms, but algorithmic modifications by themselves are proven to be insufficient to achieve the drastic speed improvements required.

Another approach to accelerate face detection and recognition algorithms is modifying their structure and developing parallelized versions of them. This can be done either on a CPU (Central Processing Unit) by taking advantage of its multiple cores and/or hyper-threading capability if exists, or on a GPU (Graphics Processing Unit) by using one of the GPGPU (General-Purpose Computing on Graphics Processing Units) frameworks available.

Even though it is much easier to develop the parallelized version of an algorithm on a CPU, putting more effort in development and implementing it on a GPU has very important advantages that make up for the additional time required for implementation. The most important property of a GPU is its ability to execute hundreds of threads concurrently and perform all the scheduling purely on hardware, in contrast to the CPUs that require software scheduling. GPU hardware is optimized for performing computer graphics computations and have excellent floating point performance. A modern GPU, for example a GTX 580, can reach very high computing rates up to 1581 GFLOPS. These properties make GPUs very suitable especially for demanding image processing, computer vision and pattern recognition algorithms. Also by offloading some compute intensive, parallelizable tasks to GPU, it becomes possible to use the CPU cores for other non-parallelizable tasks that involve less computation and more logic, leading to much more efficient usage of the hardware that already exist in a computer and hence to drastic speed improvements. This not only makes heterogeneous computing a very cost-effective way to implement computationally demanding high-performance algorithms, but also makes it the best way to make these algorithms accessible to common users.

In the scope of this thesis, efficient, massively parallel GPU implementations for two different algorithms are developed. First one of these algorithms is a boosting [2] based object detection algorithm that uses MCT (Modified Census Transform) [1] based weak classifiers. The reasons for using MCT based weak classifiers rather then Haar based ones are their superior distinguishing ability, structure suitable for GPU implementation and robustness to illumination variations. A cascade consisting of these weak classifiers is also significantly faster to compute than the cascade in [3] because it has very small number of stages and MCT based weak classifiers are much easier to evaluate than Haar based ones. The other algorithm for which a GPU implementation is developed is a feature based face recognition algorithm [4] that uses weighted regional histograms of LBPs (Local Binary Patterns) as features. All steps of these algorithms are parallelized in a GPU firendly manner and efficient GPU implementations of them are given using the CUDA (Compute Unified Device Architecture) platform of NVIDIA. Some alternative methods for parallelization on the GPU are also discussed and the problems with them are explained. The GPU implementations are further extended to utilize multiple GPUs.

1.1 Literature Review

In this section, most important developments about face detection and recognition algorithms in the last several years are covered separately, with a focus on attempts to accelerate the algorithms. GPU implementations proposed prior to this work are also mentioned.

1.1.1 Face detection

Earlier face detection methods that have good accuracy include [5], [6] and [7]. Even though these methods had good accuracy, they were too slow to be used in a real time application. Hence the researchers continued to look for faster face detection methods.

The first face detector that has both good detection speed and high accuracy at the same time has been proposed by Viola and Jones [3]. In this method, faces are detected using a cascade of classifiers. Each classifier in the cascade contains a set of Haar-like features that can be computed very efficiently using an intermediate image representation called integral image. The stages of the cascade are trained using a high number of positive and negative samples. Since the development of [3], various methods based on the same approach have been proposed that use either different features, different boosting algorithms or different detector structures.

The first example of the methods using different features is given in [8], in which the authors extended the original Haar-like feature set in [3] to include $45°$ rotated rectangular features and center-surround features. These features are computed efficiently using a rotated summed area table. An implementation of this algorithm can be found in the OpenCV [9] library. Li *et al.* [8] pointed out that the Haar-like feature set in [3] is limited for multi-view face detection and introduced another set of features that allows more flexible combination of rectangular regions. Viola and Jones [10] proposed the addition of diagonal features to the feature set. Another feature set for multi-view detection called sparse granular features is introduced by [11]. In their work, each feature contains various number of rectangles (granules) having different sizes and locations. They used heuristic search to select features from an overcomplete set. Maximum number of granules a feature can have is limited to 8. Mita *et al.* [12] proposed the usage of joint Haar-like features, which utilize the co-occurences of Haar-like features rather than the features themselves. In [13], a new type of feature called Local Rank Patterns is introduced. Another set of features called LAB (Locally Assembled Binary) features are proposed in [14].

The original Haar-like features required each window to be normalized for reducing the illumination sensitivity of the proposed method, but normalization alone is not sufficient for handling even moderate illumination variations. To make the method more robust to illumination variations, [1] proposed the usage of MCT (Modified Census Transform) based features. MCT involves comparing the pixels in a neighbourhood with their mean and obtaining a binary number, which is then used as an index number representing the structure of the neighbourhood. Learning is performed by examining te distributions of the index numbers on a high number of sample face and non-face images. A similar operator to MCT is the LBP (Local Binary Patterns), which is also used for face detection in [15] and [16] under a Bayesian and boosting framework, respectively.

Viola and Jones [3] used the Adaboost [2] algorithm to learn cascaded face detectors. Some of the follow-up works aimed to improve the detectors performance by utilizing different boosting methods and cascade decision structures. In [17], [18] and [12], the authors used the Realboost algorithm. In [19] and [20], a comparison between various boosting algorithms is performed. Both agreed on that Adaboost is inferior but they made different conclusions about Realboost and Gentleboost. [17] proposed the usage of Floatboost to overcome the monotonicity problem of the Adaboost algorithm.

Floatboost brings the ability to not only add features during learning, but also remove insignificant ones.

Viola and Jones [3] trained each stage of the cascade independently and hence did not incorparate the information learned by the previous cascades. In [21], the authors proposed to use a chain structure to incorparate the information learned before during training. At each node, the partial classifier learned is used as a prefix for further training. The resulting classifier has better performance than the original one. [18] also improved the performance of the original detector structure by using the confidence output of the partial classifier to build the first weak classifier of the next classifier. Sochman and and Matas [22] proposed using a monolithic classifier rather than a cascade and setting a rejection threshold after each weak classifier using a ratio test. They showed that the resulting classifier has less features than a cascade and faster to evaluate, yet still has better performance. Bourdev and Brandt [23] proposed using a parameterized exponential curve to set the rejection thresholds. Zhang and Viola used a data-driven scheme for setting the intermediate thresholds [16]. In their method, which is called multiple instance prunning, they took advantage of the fact that there are more than one rectangle around each face that can be considered a correct detection but only a single one of them is needed. Their scheme, while allowing to train classifiers with less features than cascades, guarantees the same detection rate.

There are also various efforts solely aiming to increase the speed of the detector. For instance, in [24], a feature centric cascade is designed based on the idea that many feature values computed are shared among multiple windows. By computing the feature values over the image beforehand, it is possible to prevent re-computing of the same values multiple times and achieving gains in speed. A similar approach is followed in [14].

For the scope of this thesis, of particular importance is the research done to accelerate face detection algorithms using GPU. The GPU implementation of Viola-Jones algorithm in [25] achieved detection rates of 19 and 46 Frames Per Second (FPS) on video streams of resolutions 1280×960 and 640×480, respectively, on a GTX 285 GPU. This implementation is 12-38x faster than the corresponding CPU implementation that runs on a Intel Xeon 3.33 GHz. Although there is an important increase in speed, having $\%81$ accuracy and 16 false positives on the CMU test set shows that a sacrifice has been made in the accuracy during the parallelization process. In [26], a multi-GPU implementation of the Viola-Jones algorithm is presented that runs at 15.2 FPS at 640×480 resolution on 4 Tesla GPUs. No information is given about the scanning parameters or accuracy. In [27], the author implemented the Viola-Jones algorithm using CUDA and modified the OpenCV implementation to be able to make a fair comparison. The resulting detector performs at 30, 14 and 8 FPS on a GTX 480 for resolutions 640×480, 1280×720 and 1920×1080, respectively. The modified OpenCV implementation runs at 15, 6 and 3 FPS at the same resolutions on a Intel Core i7-965. The scaling factors used to create the image pyramid before detection process is limited to be integers, which means that the detector will skip lower scales and will not be able to detect small faces accurately. In [28], the author achieved 2.8 FPS on a single GTX 280 GPU and 4.3 FPS on a dual GTX 295 GPU on VGA image sizes with another CUDA implementation of the Viola-Jones algorithm.

[29] is the only GPU accelerated object detection algorithm to date that uses a different feature and different classifier structure than [3]. It is based on the Waldboost [22] algorithm that uses a single monolithic classifier rather than a cascade with the ability to terminate the evaluation process of the classifier after any number of features. The features used by the weak classifiers are called Local Rank Patterns [13]. The reported detection speeds for 1280×720 and 720×540 resolutions are 58 and 97 FPS, respectively, on a GTX 280 GPU with no information about the scanning parameters or accuracy.

1.1.2 Face recognition

Numerous methods have been proposed to solve this difficult problem. Important ones among these methods can be categorized into two groups in general. These are holistic methods and feature based methods. Holistic methods have the advantage of using a global representation of the face that is not much sensitive to noise caused by partial occlusions, blurring and changes in the background. The most well-known holistic method that achieves relatively good performance is the Eigenfaces [30] method. This method involves projecting the face images into a lower dimensional subspace (the face space) using PCA (Principal Component Analysis) and obtaining a vector of weights that represent a point in the face space. The idea is, every face can be represented with such a vector of weights in the database. The classification is performed by comparing the distances of the points in the face space to the projection of the test image. A follow-up to the Eigenfaces method is the Fisherface [31] method, that projects the images using LDA (Linear Discriminant Analysis) instead of PCA. Since LDA is a supervised algorithm that takes into account the class labels of the images when determining the face space and tries to maximize the inter-person variance while minimizing the within-person variance, vectors belonging to different persons are easier to seperate after projection. Therefore the LDA results in better recognition rates than LDA. The main problem with LDA is, just as in PCA, the euclidean consideration of the data space. This property causes the method to fail in cases the data points lie in a non-linear subspace, which is most of the time the case in face data. Another holistic approach, which is called ICA (Independent Component Analysis) [32], minimizes higher order dependencies and finds a subspace along which the data becomes statistically independent after projection. In [33], authors used the neighbourhood structure of the original data space to determine the underlying non-linear subspaces. They generated Laplacianfaces using LPP (Locality Preserving Projections) and obtained better results under varying pose, expression and illumination.

In [34], a (SVM) Support Vector Machine is used with a binary tree recognition strategy. Heisele *et al.* [35] presented a component based method in which facial features are located, extracted and then combined into a single vector, which is then classified by a SVM. They also propose two global methods, both of which are outperformed by the component based method.

Another common approach for face recognition has been to use feature based methods. These methods involve extracting local descriptors from the images and using them for classification. An earlier example to these methods is Local Feature Analysis [36], in which local features are extracted using a dense set of local-topological fields. In another well-known feature based method called Elastic Bunch Graph Matching [37], a face is represented as a topological graph. Each node of this graph is a facial landmark (e.g. eye, nose, etc.) and the edges are labeled with 2D distance vectors. At each node, Gabor filter responses, which are called jets, are computed to obtain a local descriptor. T. Ahonen *et al.* [4] proposed another feature based method that uses weighted regional histograms of Local Binary Patterns (LBP) [38], which yielded impressive results on the FERET [39] database. They used a nearest neighbor classifier as the classification algorithm and Chi-square distance as the distance metric. In [40], face recognition is transformed into a two-class problem by classifying every two face images as intra-personal and extra-personeal ones. Adaboost algorithm is used to learn a similarity for every image pair. The Chi-square distance between corresponding LBP histograms of two face images is used as discriminative feature for intra/extra-personal classification. In [41], another set of features called Local Derivative Patterns (LDP) are proposed, that takes into account higher-order local patterns. Baochang et. al [42] introduced a novel descriptor called HGPP (Histogram of Gabor Phase Patterns) to represent faces. In HGPP, the quadrant-bit codes are first extracted from faces based on the Gabor transformation. Global Gabor phase pattern (GGPP) and local Gabor phase pattern (LGPP) are then proposed to encode the phase variations. They are both divided into the nonoverlapping rectangular regions, from which spatial histograms are extracted and concatenated into an extended histogram feature to represent the original image. Recognition is performed by using a nearest-neighbour classifier. In another recent work given in [43], the authors analyzed two moment based feature extraction methods, namely, Zernike moments and Complex Zernike moments for face recognition. They found that Complex Zernike moments perform not only better than Zernike moments, but also it is the descriptor that gives best recognition rate amongst the descriptors well known for face recognition.

There were no serious attempts to accelarate the feature extraction methods utilized by the face recognition algorithms using GPUs to date. On the other hand, there have been efforts to accelerate some of the classification algorithms, which are used during the classification process of face recognition algorithms. One example of these algorithms is the k-nearest neighbour (k-NN) algorithm. Different k-NN algorithms attempted to reduce the number of distance calculations to be performed. In [44] and [45], this is performed by constructing kd-tree where each node is a training point. Once the tree is constructed, a search can be performed very efficiently over only the closest neighbours of a point. The problem with this approach is that it reqires complex data structures and recursive traversal algorithms, none of which is suitable for a GPU implementation. In [46], the authors pointed out that if the number of dimensions exceeds 10 to 20, searching in kd-trees and related structures involves the inspection of a large fraction of the database, thereby doing no better than brute-force linear search. They proposed a scheme for approximate similarity search based on hashing. During training, a method called locality sensitive hashing (LSH) uses several hash functions to bin similar training points together. Classification of a query point can then be accomplished by hashing the point and retrieving elements stored in the bin that would contain the hashed query point.

CUDA accelerated implementations of the brute-force k-NN algorithm are given in [47] [48]. Both authors reported better results than a CPU using a GPU.

1.2 Organization of the Thesis

This thesis is organized in the following manner: Chapter 2 provides the background information required to understand the rest of the thesis. Details about both the face detection and face recognition algorithms are given along with an introduction to GPU architecture and the CUDA platform. Brief information about some other GPGPU frameworks are also given for the sake of completeness. Chapter 3 contains detailed information about the GPU implementations of the algorithms covered in this work. Alternative approaches for parallelization are also discussed, along with the extension of the GPU implementations to make use of multiple GPUs. Chapter 4 discusses the experimental results obtained from all implementations and compares the results obtained using GPU with those obtained using CPU to show how accurate and fast the proposed GPU implementations are. Chapter 5 concludes the thesis, provides closing remarks and shows future directions.

2. BACKGROUND

This chapter includes all the background information required to understand the rest of the thesis. Detailed information about the implemented face detection and recognition algorithm are given in relevant subsections. An introduction to the GPU architecture, CUDA and other GPGPU frameworks are also given so that the jargon used and explanations given in the following sections can be understood and the thesis becomes self-contained.

2.1 MCT Based Face Detection

This section includes the information required to understand the testing phase of the face detection algorithm for which a CUDA implementation is developed.

2.1.1 MCT based weak classifiers

MCT [1] is a transform that generates a binary number from the pixel values in a given neighbourhood N. The binary value corresponding to a pixel location is obtained by comparing all pixel values on a neighbourhood centered on that pixel with the mean of all pixel values in the neighbourhood in row-major order. Let $N(\mathbf{x})$ be a spatial neighbourhood centered at the pixel location $\mathbf{x}$ and $\bar{\mathbf{I}}(\mathbf{x})$ be the mean of the pixel intensity values on this neighbourhood. If $\bigotimes$ is the concatenation operator, then the MCT can be defined as follows:

$$\Gamma(\mathbf{x}) = \bigotimes_{\mathbf{y}\in\mathcal{N}} \zeta(\bar{\mathbf{I}}(\mathbf{x}), \mathbf{I}(\mathbf{y})) \tag{2.1}$$

where the comparison function $\zeta(\mathbf{I}(\mathbf{x}), \mathbf{I}(\mathbf{y}))$ is defined as

$$\zeta(\mathbf{I}(\mathbf{x}), \mathbf{I}(\mathbf{y})) = \begin{cases} 1, & \mathbf{I}(\mathbf{x}) < \mathbf{I}(\mathbf{y}) \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

The definition of MCT does not put any restrictions on the size of the neighbourhood, but for the purposes of object detection, the size of the neighbourhood is chosen as

3×3, because using larger sizes lead to memory inefficient weak classifiers as one can see later in this section. When a 3×3 neighbourhood is considered, the resulting MCT value becomes a 9 bit string that can take values in the range $[0, 510]$ when converted to a decimal value. These values correspond to local structure kernels, some of which are shown in Figure 2.1.

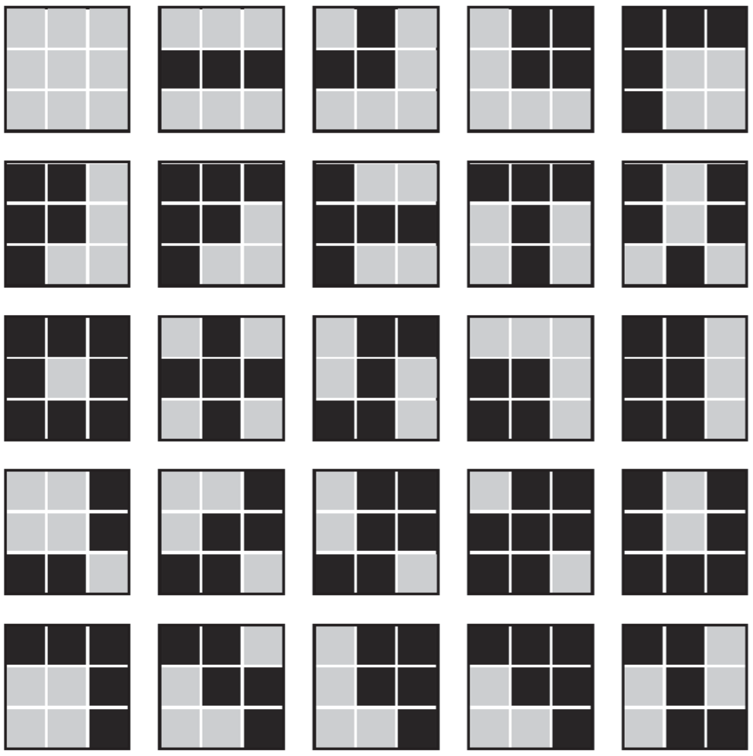

Figure 2.1: Some of the possible local structure kernels in a 3×3 neighbourhood [1].

These kernels can code information about the structures like edges, ridges, etc. in the image in binary form. An example of computing the MCT value from a 3×3 neighbourhood is shown in Figure 2.2.

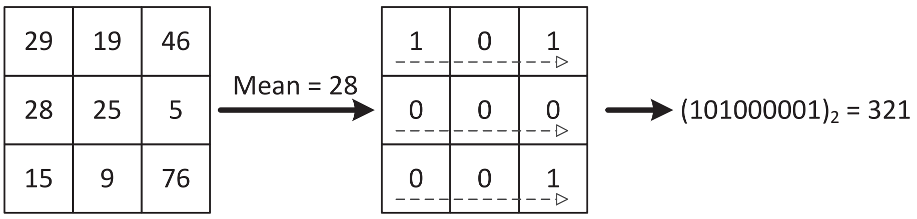

Figure 2.2: MCT Computation example in a 3×3 neighbourhood.

A MCT based weak classifier $h_{\mathbf{x}}$ consists of a coordinate pair $\mathbf{x} = (x, y)$ relative to the origin of the scanning window and a 511 element lookup table. The coordinates specify the center location of the neighbourhood that needs to be used when calculating the MCT value. The detector uses a base resolution of 24×24 and since MCT values cannot be calculated at the edges and corners, total number of different values $\mathbf{x}$ can take is limited to $22 \times 22 = 484$. The lookup table contains a weight for each kernel index γ such that $0 \leq \gamma \leq 510$. The output of a weak classifier on a window in the input image is determined by defining a 3×3 neighbourhood centered at the location $\mathbf{x}$, calculating the MCT value which is equal to the γ index, and getting the value from the lookup table that corresponds to γ.

An important advantage of MCT based weak classifiers is their robustness to illumination variations, which are commonly simulated by smooth, linear monotonic transformations [1]. Since the MCT is defined in a small neighbourhood, the order of its bits does not change after the application of such a transform and makes the detector more robust to illumination variations without bringing additional computational overhead. This property can be seen clearly from Figure 2.3 that shows the fact that results of applying MCT to two face images having very different illumination levels are nearly identical. Also, MCT based features are very easy to compute, because once MCT is applied to the grayscale input image, computing the value a weak classifier takes on a window becomes a matter of doing a single memory lookup.

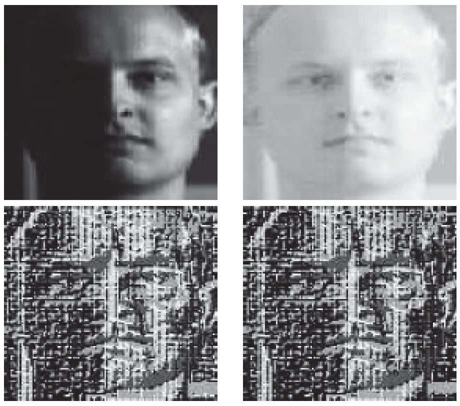

Figure 2.3: Illumination invariance of the MCT.

2.1.2 Classifier cascade

The detector has a cascaded structure as in [3], but has fewer stages because of the superior distinguishing ability of MCT based weak classifiers and a different learning algorithm. Each stage contains a strong classifier containing a number of weak classifiers, each of which having a different $\mathbf{x}$ position. Maximum number of allowed different positions in each stage is given as input to the training algorithm as explained in [1]. $H_j(\Gamma)$, the strong classifier of the j. stage, is defined as follows:

$$H_j(\Gamma) = \sum_{\mathbf{x} \in W'} h_{\mathbf{x}}(\Gamma(\mathbf{x})) \qquad \textbf{(2.3)}$$

where $W' \subseteq W$ is the set of unique positions used by the weak classifiers $h_{\mathbf{x}}$ in the strong classifier. A window W passes the j. stage if the sum of the responses of the weak classifiers on that window is less than or equal to the stage threshold T_j. Formally, stage j allows the window to pass if $H_j(\Gamma) \leq T_j$. Stage thresholds T_j are tuned after the strong classifiers are trained using a set of samples not used during training. A window is classified as the searched object by the algorithm when it passes all stages.

The cascade used in this work has 5 stages utilizing 10, 20, 40, 80, and 217 positions, respectively, as shown in Figure 2.4. The maximum number of possible positions allowed for the stages were 10, 20, 40, 80 and 484 for the stages in question. It should be noted that even though all 484 positions were allowed to be used in the last stage, only 217 of them are utilized by the training algorithm. During training, all stages were trained on the same positive set consisting of 8400 manually collected and scaled images of frontal faces with limited variance in pose, expression and illumination. 40000 negative samples were used for training each stage. These samples were obtained by using the partial cascade trained after each stage on 10000 large images collected from the internet that do not contain any faces and selecting the windows that are misclassified. This training method, which is adopted from [3], is the only difference between the implementation used in this work and the one explained in [1]. Since the focus on this thesis is on the testing algorithm and not the training stage, things explained this thesis can be used with a cascade trained using the method in [1] as well.

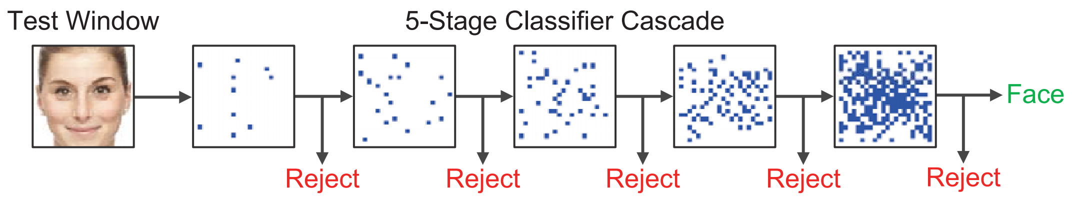

Figure 2.4: The 5-stage cascade used in this work.

2.1.3 Scanning process

The scanning process starts with the construction of an image pyramid from the input image. Then MCT is applied to every image in the pyramid to obtain the transformed image pyramid. The whole image pyramid is scanned with a sliding window and at each location the classifier cascade is used to find out whether the window contains the searched object or not. Then multiple detections whose area of intersection is close to the total area of the detections are grouped and merged into a single detection. As in any other sliding window approach, there are scanning parameters like horizontal and vertical step sizes (Δx and Δy), the ratio between consecutive scales, or the scale factor (s) and the starting scale. In this work, we choose $\Delta x = \Delta y = 1$, $s = 1.15$ and use a starting scale of 1, which results in a computationally demanding, fine-grained scanning process and makes it possible to detect faces as small as 24×24 accurately.

2.2 LBP Based Face Recognition

This section includes the necessary information required to understand the face recognition algorithm for which a CUDA implementation is developed.

2.2.1 The LBP operator and its extensions

The original LBP operator [49] generates a 8 bit binary number by thresholding the pixels in a 3×3 neighbourhood with the center pixel as shown in Figure 2.5. The operator is later extended to handle different neighbourhood sizes by using circular neighbourhoods and bilinear interpolation [38]. The recommended notation to describe such neighbourhoods is (P, R), where P is the number of sampling points on the circle and R is the radius of the circle. Figure 2.6 shows the location of the sampling points in a $(8, 2)$ neighbourhood. Values at non-integer coordinates are computed using bilinear interpolation.

Another extension to the LBP operator is the usage of uniform patterns [38]. A binary pattern is called uniform if it contains at most two bitwise transitions when the bit sequence is considered to be circular. For example, the patterns 00000000 (0 transitions), 01110000 (2 transitions) and 11001111 (2 transitions) are uniform whereas the patterns 11001001 (4 transitions) and 01010011 (6 transitions) are not.

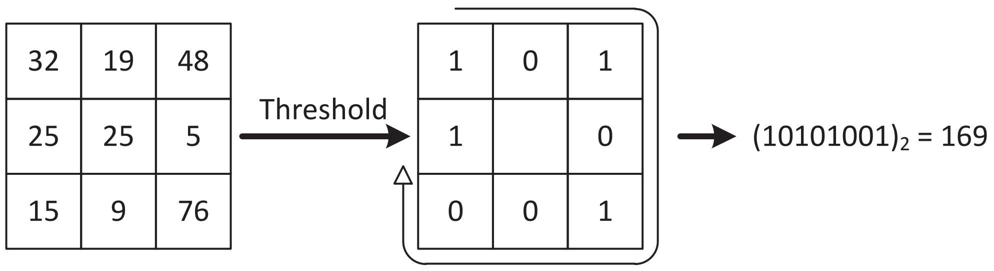

Figure 2.5: Example LBP computation.

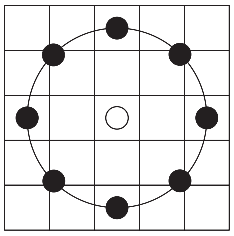

Figure 2.6: A circular $(8, 2)$ neighbourhood.

With 8 sampling points, the number of different uniform patterns is 58. This leads to a 59 bin histogram when all the non-uniform patterns are considered to be in the same bin. When 16 sampling points are used, the number of bins becomes 243. The usage of uniform patterns is motivated by the fact that most patterns in facial images are uniform. In [4], the authors have found that, 90.6% of the patterns in the (8,1) neighbourhood and 85.2% of the patterns in the (8,2) neighbourhood are uniform in case of preprocessed FERET [39] facial images. The usage of uniform patterns is indicated with the notation $(P, R)^{u2}$.

2.2.2 Construction of the feature vector and classification

In [4], to obtain a global description of the face that also contains information on a regional level, the input image is divided into m regions: $R_j, j = 1, ..., m$. The regions may overlap or have different sizes. A histogram is generated from each region and then all histograms are combined to obtain the final feature vector. Classification is performed with a nearest neighbour classifier and the distance metric used is the chi-square distance. Furthermore, a weight w_j is determined for each region R_j according to its importance in classification, which are then utilized during the distance computation. A straightforward way to determine these weights is measuring the classification performances in cases when only a single region is used when creating the feature vector. The weighted chi-square distance is defined as

$$\chi^2_w(\mathbf{S}, \mathbf{M}) = \sum_{i,j} w_j \frac{(S_{i,j} - M_{i,j})^2}{(S_{i,j} + M_{i,j})} \tag{2.4}$$

where $\mathbf{S}$ and $\mathbf{M}$ are the histograms to be compared and i and j denote the i-th bin in the histogram corresponding to the j-th region.

2.3 A Brief Overview of the GPU Architecture and the CUDA Platform

This section contains information about the GPU hardware, CUDA and GPGPU frameworks in general, that are required to understand the details of the CUDA implementationd developed in the following sections.

2.3.1 GPU architecture

With the increasing demand in real-time, high-definition 3D graphics, the GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational power as shown in Figure 2.7 that compares the floating performances of CPUs and GPUs released in the last few years. The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control [50].

The massively parallel structure of the GPU resembles a supercomputer. The GPU consists of a set of streaming multiprocessors (SMs) that contain a number of Single Instruction Multiple Data (SIMD) cores called "CUDA cores". The number of SMs vary between different GPUs. For example a GTX 580, which is the GPU used as a testbed for this research, contains 16 SMs, each of which contains 32 CUDA cores, resulting in a total of 512 CUDA cores [50]. These CUDA cores enable the GPU to execute hundreds of threads concurrently, making the GPU well-suited to solve problems that can be expressed as data-parallel computations. The GPU is targeted for data-parallel applications that involve massive parallelism, heavy computation and little logic. Therefore, as stated before, most of its die area is devoted to computational units rather than memory and hence it hides the memory access latencies with calculations instead of big data caches. Because of the very limited area devoted

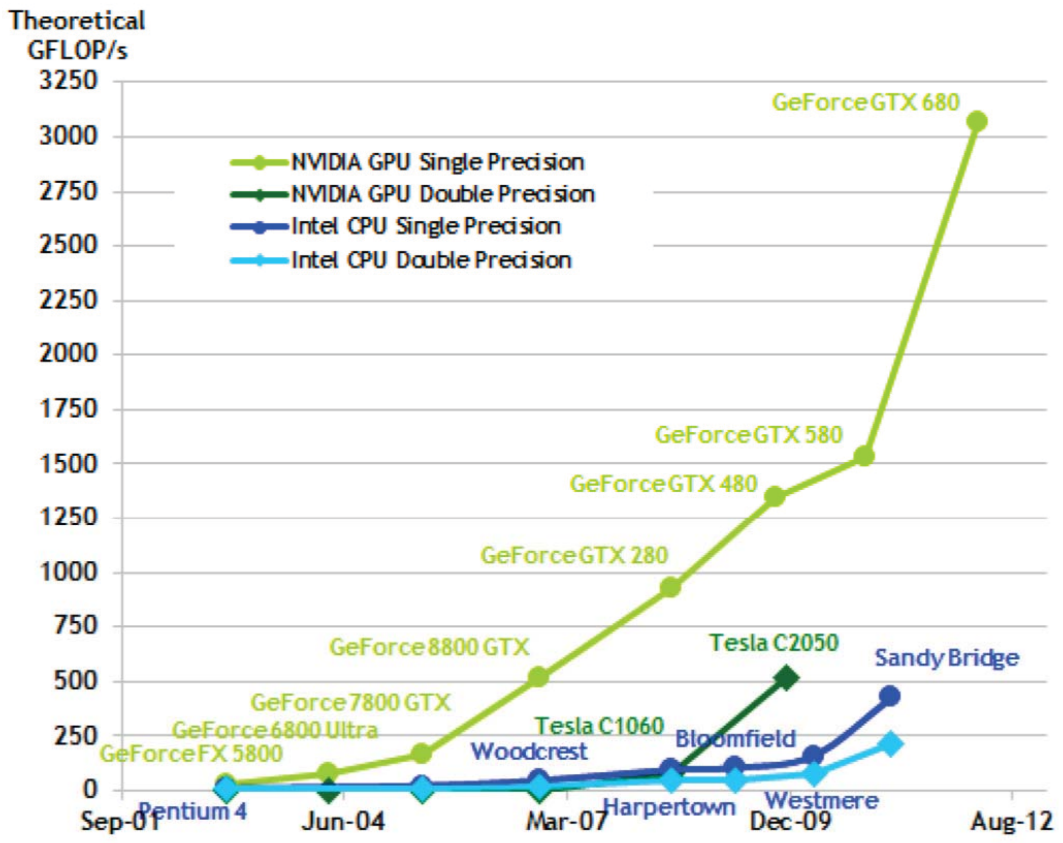

Figure 2.7: Floating point performances of various CPUs and GPUs (Courtesy of NVIDIA).

for memory, each SM in the GPU has very limited amount of registers and on-chip memory (shared memory) available to be shared among the threads that reside in it. Since these on-chip resources are very limited, they need to be used very efficiently. There are also off-chip, global resources available to all SMs. These include the texture, constant and the global memory. All three of them are implemented as off-chip high-latency DRAM, but both the texture memory and the read-only constant memory are cached and therefore they are faster to access if used properly. This memory arrangement is visualized in Figure 2.8.

2.3.2 CUDA platform

The CUDA platform exposes the processing power of a GPU to the developers and allow them to write massively parallel code for general purpose applications that can take advantage of the properties of a GPU like massively parallel structure, great floating point performance, transparent scalability and hardware scheduling with zero overhead. The CUDA platform refers the GPU and CPU as the "host" and "device",

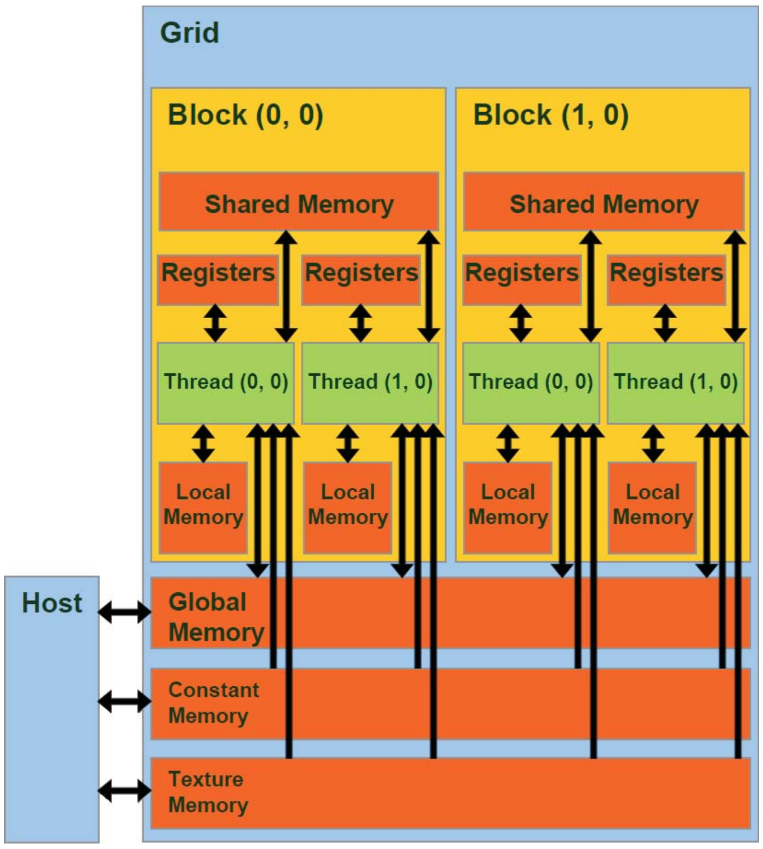

Figure 2.8: CUDA memory hierarchy (Courtesy of NVIDIA).

respectively. A CUDA program consist of the host code that runs on the CPU and the device code that runs on the GPU. In CUDA terminology, a kernel refers to a chunk of code that gets executed on the GPU. Parallel processing is started by launching a kernel from a CPU thread as a grid containing a 1D or 2D array of thread blocks, each of which contains a 1D, 2D or 3D array of threads. Figure 2.9 shows a sample 3×2 grid containing 4×3 sized blocks. Each thread in the grid executes the same chunk of code and is identified by its block index and thread index, which together form a unique thread id. If the total number of blocks in the grid is denoted by $grid\_size$ and the total number of threads in a block is denoted by $block\_size$, then the total number of threads in the grid can be calculated as $grid\_size \times block\_size$.

When a kernel is launched, the thread blocks are distributed to SMs with available resources. Threads of a block execute concurrently on one SM and a single SM can execute up to 8 blocks concurrently, depending on the resource requirements of each block. As thread blocks terminate, new blocks are launched on the vacated SMs. The hardware further divides consecutive threads with increasing indices in each block to groups of 32 threads called warps. A SM manages and schedules threads using warps

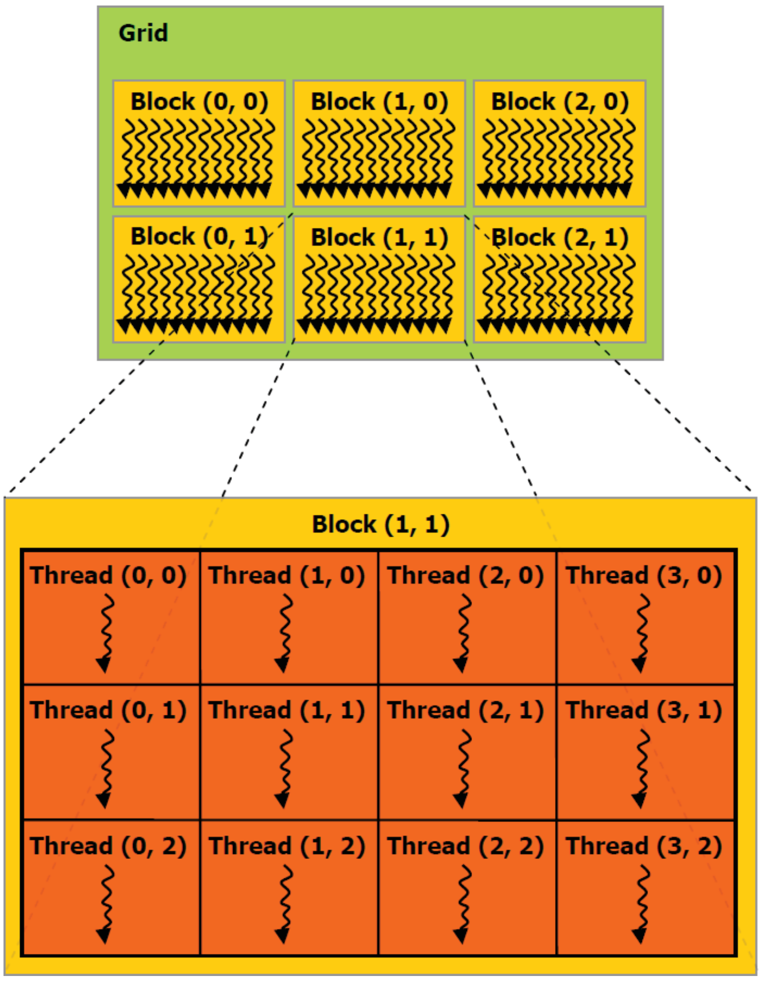

Figure 2.9: A sample launch grid (Courtesy of NVIDIA).

as the smallest unit. Threads in a warp execute the exact same instruction. In cases where the threads in a warp diverge because of a data dependent branch, the hardware groups the threads according to the path they will take and execute these groups serially until reaching a point where both groups can join and continue to execute the exact same instructions.

These abstractions provide fine-grained data and thread parallelism, nested within coarse-grained data and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Another result of this abstraction is transparent (or automatic) scalability to future GPUs. Since blocks are independent from each other and can be executed any time in the available SMs, concurrently or sequentially, a CUDA program can execute on any number of processor cores as illustrated by Figure 2.10. The CUDA runtime takes care of the distribution of the blocks to the available cores therefore and only the runtime system needs to know the physical processor count and not the developers themselves. This feature makes it possible for the applications written in CUDA to run faster automatically in faster devices, even in the ones that are not yet released.

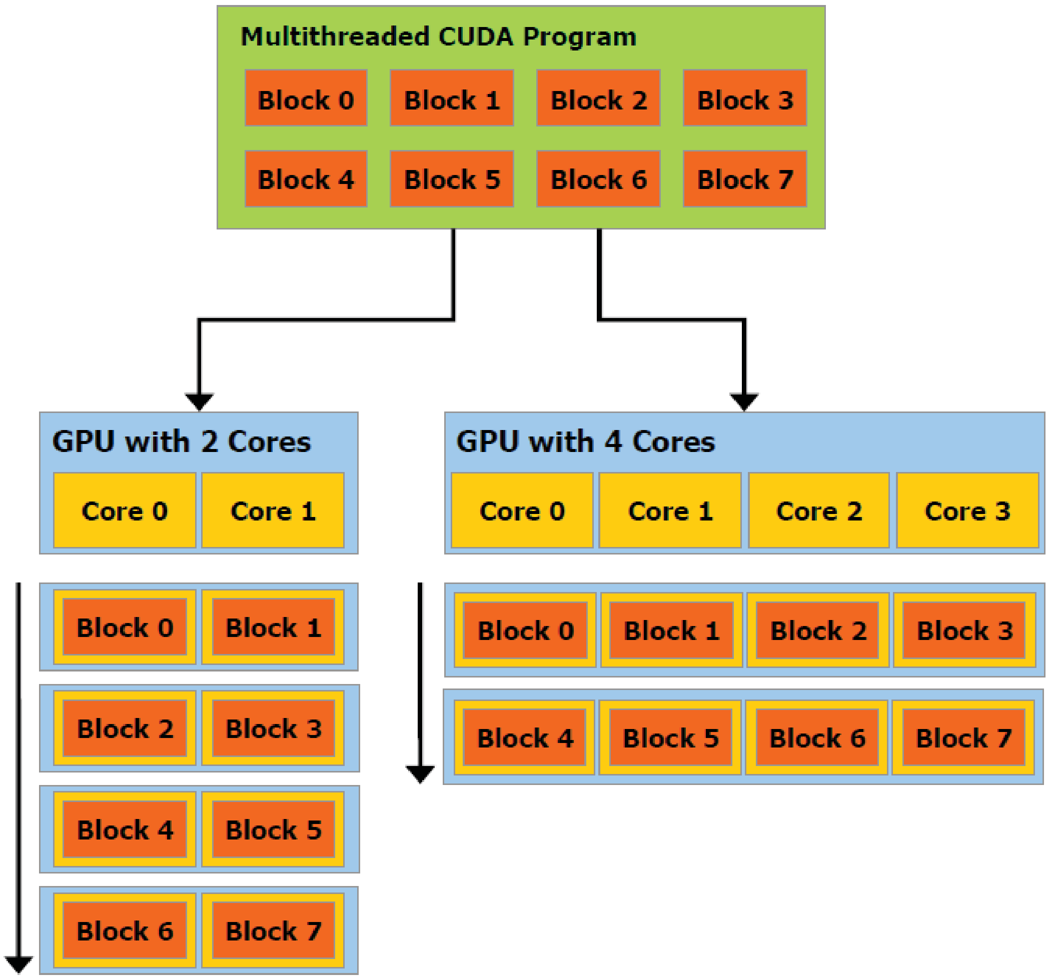

Figure 2.10: Transparent scalability demonstrated (Courtesy of NVIDIA).

2.3.3 Memory types in CUDA

The CUDA platform provides 5 types of memory to be used in GPGPU applications. The fastest one among them are the registers, which are on-chip. When a kernel is launched, the runtime distributes the registers available in each SM to the threads belonging to the blocks running on that SM. Each thread has its own set of registers and cannot write to or read from the registers belonging to other threads. Registers are the scarcest resource in the GPU and therefore are most of the time the limiting factor on the maximum number of threads that can reside in a SM.

Next comes the shared memory, which is also on-chip. Shared memory is banked and as long as there are no bank conflicts in the access patterns, it is as fast as the registers. Even though the size of the shared memory is larger than the register space, it is still very limited (16KB per SM in devices of compute capability 2.0). The most important property of the shared memory is that it is shared among the threads of a block. Therefore it is very useful for tasks like caching the data required by all threads in the block or for accumulating temporary results.

The other type of memory is global memory, which is implemented as off-chip high latency DRAM. The size of the global memory is very large (in the order of GBs), but its access latency is about 100-400 times longer than that of the shared memory. In order to make use of the global memory efficiently, accesses to it should be coalesced. Simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of memory segments. On devices with compute capability 1.3 and higher, coalescing is achieved for any pattern of addresses requested by the half-warp as long as the words accessed by all threads lie in the same segment. In older devices, not only all words have to be in the same segment, but also consecutive threads should access consecutive words. The segment size is 32,64 and 128 bytes for accessing 8-bit, 16-bit and 32/64-bit words, respectively. If the accesses are not coalesced, for instance, if the threads in a half-warp access n different segments, then n different memory transactions will be issued, leading to poor memory bandwidth utilization.

In cases when performing coalesced memory accesses is not possible, one can use one of the two alternatives to global memory. The first one is the texture memory. Texture memory is not exactly a new memory location but a special memory layout. The data resides still in the global memory, but it has a cache associated with it that is called the texture-cache. Texture cache has about 6-8kb size per SM and is optimized for 2D spatial locality, which prevents unnecesarry memory transactions when accesses are spatially local. Since the main purpose of the texture-memory is storing 2D textures, it has dedicated hardware to perform texture caching with the options of bilinear interpolation, clamping and range normalization, all of which can be very beneficial in some applications. A similar memory location is the constant memory, which has another cache, the constant-cache associated with it. This memory space is typically used for storing constant data that will not change during the lifetime of the application. The global, texture and constant memories have application-wide lifetime, meaning that data stored in any of them will persist during any number of grid launches until the application terminates.

The last type of memory is the local memory, which is essentially a location in global memory that is used to store local arrays and additional variables if there are not enough registers available. Table 2.1 summarizes the properties of the memory types in the GPU.

Table 2.1: Memory types in CUDA.

Memory	Location	Cached	Access	Lifetime
Registers	On-chip	No	Read/Write	Thread
Shared	On-chip	No	Read/Write	Block
Global	Off-chip	No	Read/Write	Application
Constant	Off-chip	Yes	Read-only	Application
Texture	Off-chip	Yes	Read-only	Application
Local	Off-chip	No	Read/Write	Thread

2.3.4 Compute capability

Each new generation of NVIDIA GPUs have additional capabilities resulting from architectural improvements and more resources compared to the older generation devices. The capabilities of a device can be determined from its compute capability, which is a major revision number, followed by a minor revision number. Currently, the newest devices have the compute capability 2.0 and 2.1 (Fermi architecture) which are very suitable for GPGPU applications. The CUDA implementations proposed in this thesis do not assume anything about the GPUs the algorithms will be executed on and do not use any property introduced after compute capability 1.0. GTX 580, the GPU used in this thesis is of compute capability 2.0.

2.3.5 Occupancy

Occupancy is a measure of utilization of the SMs in the GPU. It is affected by many factors, such as the selected block and grid dimensions, registers and shared memory used by a block and the compute capability of the device.

Occupancy is defined as the ratio of the number of active warps per SM to the maximum number of active warps. A higher occupancy results in the hardware being better utilized and the memory latency being better hidden. Even though increasing the occupancy improves the performance of the CUDA applications in general, there can be cases where higher occupancy may lead to lower performance. Therefore achieving %100 occupancy is not the primary concern during CUDA application development.

As stated in Section 2.3.1, each SM has a limited number of registers and shared memory that are shared among the blocks being executed in it. The maximum number of threads and blocks that can reside in a SM is also limited, latter of which is 8 in the current GPUs. All these properties, which change with the compute capability of the GPU, limit the maximum occupancy a kernel can have in a given GPU. It is up to the developer to select the block and grid dimensions with care and pay close attention to the register and shared memory usage of each kernel.

2.3.6 Multiple GPUs

CUDA supports the use of multiple GPUs in a single application. The GPUs are completely independent of each other, with their own memory space and instructions. Each GPU must be programmed and setup separately. Generally, a CPU thread is launched to manage each GPU. Starting from CUDA Toolkit 4.0, it is possible to use multiple GPUs with a unified address space.

2.3.7 Heterogeneous programming

A CUDA program consist of the combination of host and device code. Typically the host code performs sequential tasks that exhibit little to none parallelism and calls functions from the CUDA runtime or driver API to send commands to the GPU or perform memory transfers, while the device code performs other tasks that involve high amount of parallel computation. A CUDA application can contain arbitrary number of kernels and can launch them on the GPU at any time, in any order. Since kernels are launched asynchronously, the CPU can either wait for the GPU to finish its job or continue to do processing until reaching a point where results from the GPU are needed to continue execution. Table 2.11 shows the flow of an example CUDA application.

Host code is written in standard C/C++, while kernels and other device code are written in CUDA C, an extended version of the C language. Both the host and device code are sent to NVCC (NVIDIA Compiler Collection) for compilation. NVCC seperates the

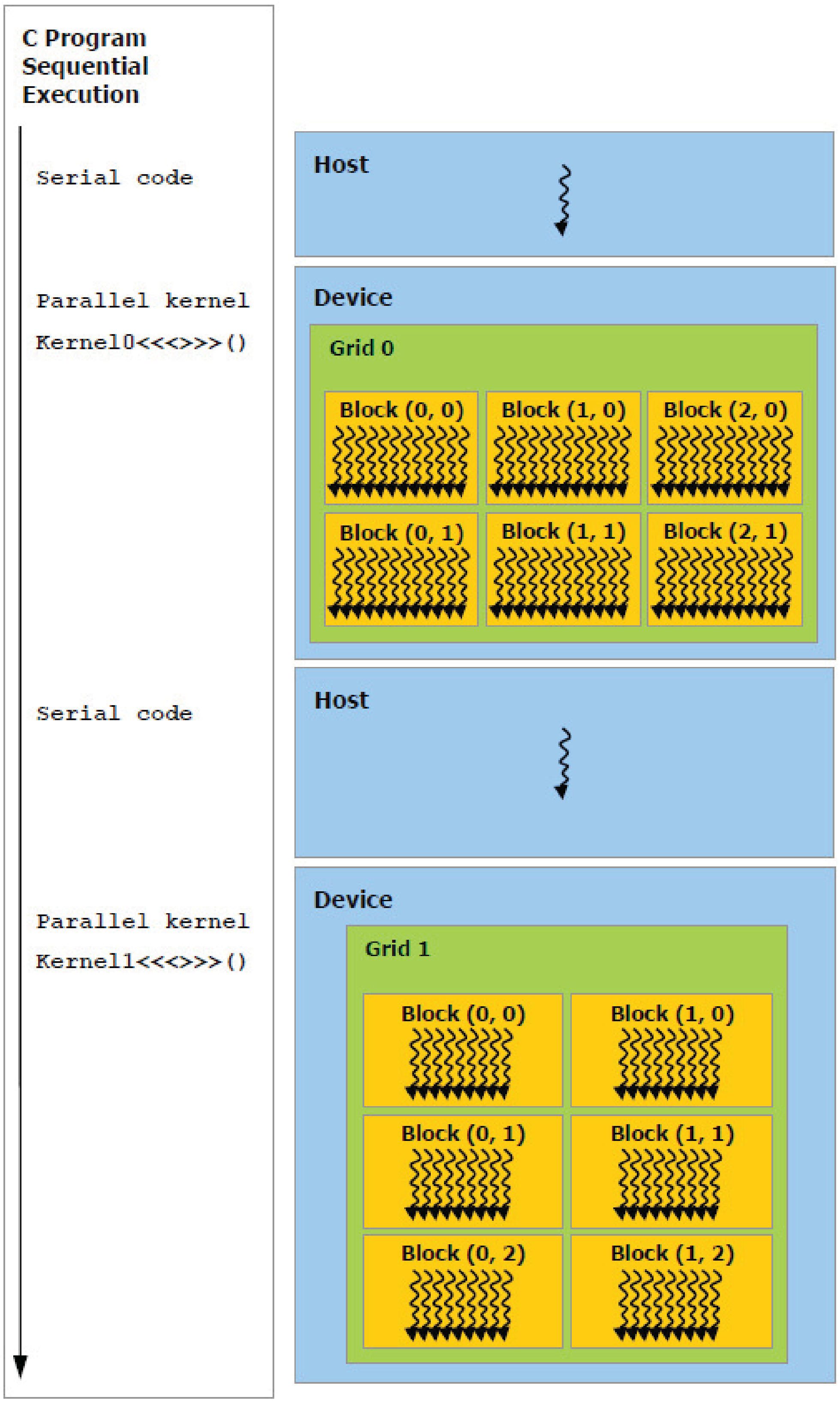

Figure 2.11: Heterogeneous programming with CUDA (Courtesy of NVIDIA).

host and device sections of the code, compiles the device code to be run on the GPU and sends the host code to the hosts compiler for further compilation. The host code runs as an ordinary CPU process.

2.4 Other GPGPU Frameworks

2.4.1 OpenCL

OpenCL (Open Computing Language) is an open specification developed by Apple for developing applications for heterogeneous platforms containing CPUs, GPUs DSPs and other processors. It is maintained by the Khronos Group. It provides a C99 based language with some extensions for writing kernels and a runtime API for managing the data transfers and execution in paralel processors. The most important property of OpenCL is that it is a truly cross-platform GPGPU framework. Each vendor (e.g. NVIDIA, AMD) develops its own implementation of the OpenCL specification and provides the developers with the necessary tools to develop GPGPU applications with them. OpenCL is similar to CUDA in many ways, but has lower level programming constructs and hence is harder to use. Also, even though it is an open platform that has been available since 2008, it has not developed as fast as CUDA. Therefore it is not used in the work done in this thesis.

2.4.2 DirectCompute

DirectCompute is a dedicated subset of the DirectX 11 API (Application programming interface) developed by Microsoft for GPGPU programming. Similar to OpenCL and CUDA, this API is used on the CPU side, to set up and execute the kernels on the GPU. The kernels are written in a high level language called HLSL (High Level Shader Language) very similar to C++, which is then compiled and sent to the GPU to utilize its compute shader stage. Since DirectCompute is a part of the DirectX API, it is currently only supported on Windows Vista and Windows 7. Linux or any other operating systems are not supported.

3. GPU IMPLEMENTATION

In this section, details about the GPU implementations proposed in this thesis are given. The steps followed during the parallelization and CUDA implementation process are given for both the detection and recognition algorithm in the relevant subsections. Detailed explanations are provided about some important decisions that need to be made during the design and development process along with the reasoning behind them and their final impact on the performance of the resulting system.

3.1 Face Detection

This section contains detailed information about the CUDA implementation developed for the MCT based face detection algorithm.

3.1.1 Preprocessing

In order for the detector to detect objects at various sizes, the input image needs to be scanned in multiple scales. In [3], this is performed by scanning the same input image with up-scaled classifiers. Even though this approach is very efficient for a CPU based implementation, it is not suitable for a GPU based one. The most important reason of this is that, as the classifiers get scaled up, the access pattern to the images in the GPU memory becomes very sparse. As described in Section 2.3.3, sparse memory accesses to global memory cannot be coalesced and therefore are very slow. Storing the input image in the texture memory does not help either, because of the very high number of cache misses resulting from the memory accesses that are not spatially local in the 2D space. Therefore the approach followed in this implementation is to scale the input image down multiple times and construct an image pyramid that will be scanned only once by the detector using a fixed size window.

The whole image pyramid is stored as a single large image as shown in Figure 3.1. Even though this layout has empty regions that increase the number of windows scanned, it greatly simplifies the scanning process described in the next section. Also the regions having a constant gray level are easily eliminated by the first stage of the cascade. Therefore the performance loss that comes with the increased number of windows is eliminated by the performance gain from the simple yet efficient scanning process that this sacrifice makes possible.

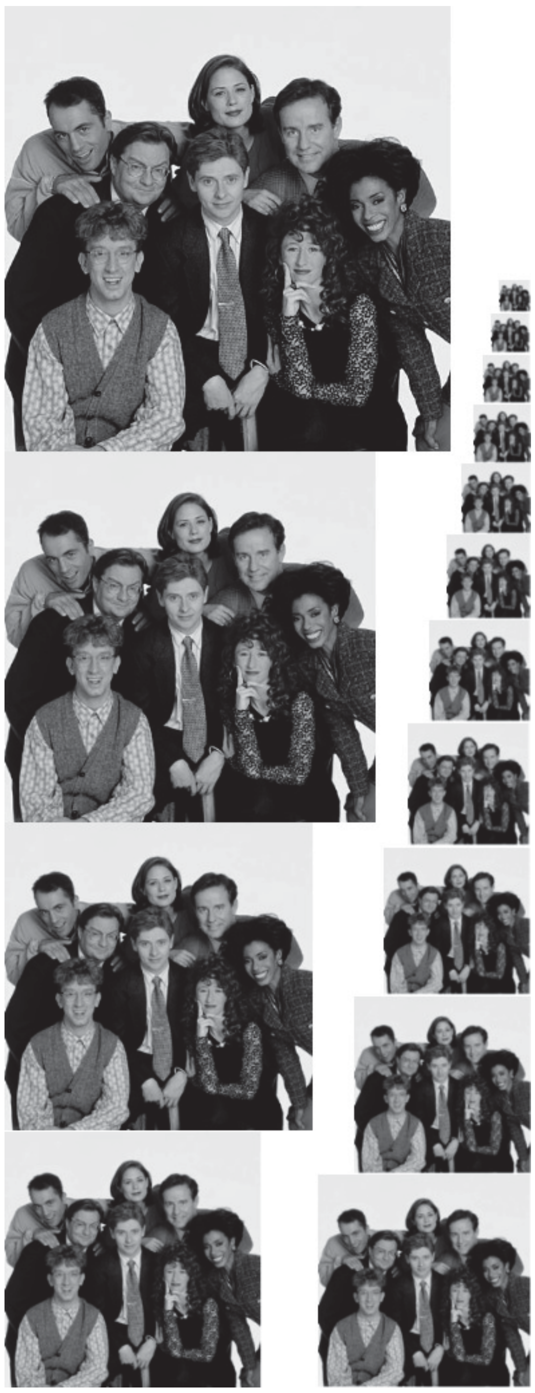

Figure 3.1: Image pyramid.

The texture memory of the GPU has the ability to make linear interpolation automatically when the memory is addressed using floating point coordinates. This proves to be very useful for constructing the image pyramid quickly. At each level of the pyramid, the downscaled image is constructed by binding the image from the previous level to a texture and then sampling it according to the scale factor.

As described in Section 2.1.1, evaluation of a MCT based weak classifier becomes a simple memory lookup when the MCT values are precomputed. This is achieved by applying MCT to the whole image pyramid. Application of MCT is performed by launching a grid of thread blocks with a block size of 16×16 that operate on the image pyramid. Rather than binding the original image pyramid that resides in global memory to a texture, shared memory, which is on-chip hence as fast as the registers, is utilized as a custom managed cache to further speed up the processing. Each thread block pulls in a patch from the global memory to its dedicated shared memory before further processing. This is performed by making each thread in a block pull in a single pixel from the global memory. Since the MCT computation at the edges and corners requires additional pixels, threads at the edges pull in one additional pixel, and threads at the corners pull in three additional pixels. After getting the data to shared memory, threads in each block are synchronized with each other using the $\_\_syncthreads()$ primitive of CUDA to make sure that all the required data is pulled in to the shared memory before the beginning of the MCT computation. Then all threads compute the value corresponding to their location and write the result to the device memory allocated for the transformed image pyramid. At this point, the original image pyramid is no longer needed and hence can be discarded. In the remaining section of this section, the term "image pyramid" refers to the one containing the result of the MCT, which is shown in Figure 3.2.

The other preparation that needs to be done before beginning the scanning process is loading the classifier cascade into the device memory. During the scanning process, the cascade data will be heavily accessed and therefore the speed of accessing them is crucial for the performance. Since every thread will access the classifier data sequentially in the same order until the classification ends, the best location for the cascade would be the constant memory because of its broadcasting ability that allows it to serve multiple threads when all of them issue a read request from the same

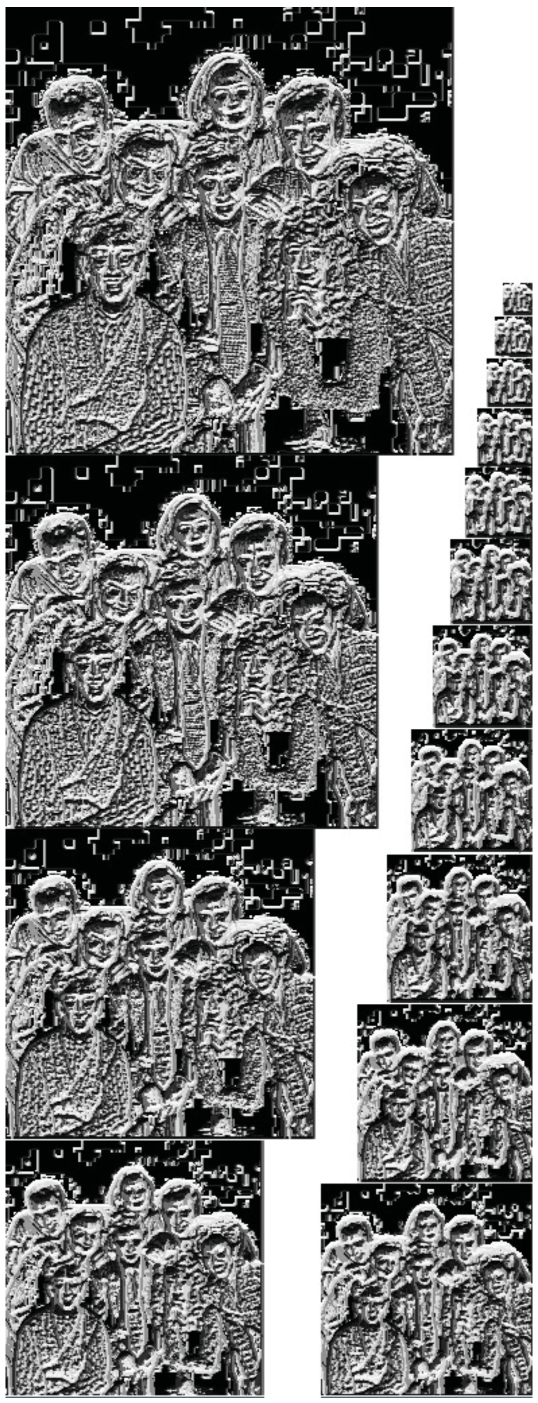

Figure 3.2: Result of applying MCT to the image pyramid.

memory address. Unfortunately, the size of the constant memory is too small store a cascade containing more than 32 weak classifiers. Therefore the cascade is stored in the texture memory as a 2D floating point texture. Each row starts with a single floating point value containing the x and y coordinate of the weak classifier in its upper and lower 2 bytes, respectively. Rest of the row contains the weights of the lookup table. Visualization of this layout can be seen in Figure 3.3. This approach helps to reduce texture cache misses by making the accesses to the feature weights as spatially local as possible in the 2D space.

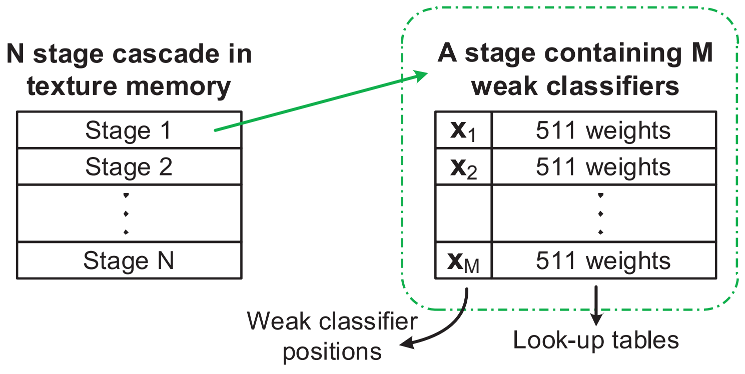
N stage cascade in texture memory
Stage 1
Stage 2
Stage N
A stage containing M weak classifiers
x_1 511 weights
x_2 511 weights
x_M 511 weights
Weak classifier positions
Look-up tables

Figure 3.3: Memory layout of the the cascade data in texture memory.

3.1.2 Detection

The detection process involves scanning the image pyramid with a sliding window of fixed dimension, classifying the window at each location and writing results back to the device memory. There are various ways to parallelize this process and map it to the resources of the GPU. We experimented with various methods and thread/block arrangements as detailed in section 3.1.4 and found that the best performance overall is obtained when making each GPU thread classify a single window on the image pyramid. In this arrangement, a kernel is launched with a grid having as many threads as the number of windows that needs to be classified. The block size is selected as 16×16 because it results in the highest occupancy, hence in the best utilization of the GPU. Figure 3.4 shows the arrangement of the thread blocks on a sample image pyramid.

One should immediately notice that this arrangement implicitly introduces a limitation. If d_w and d_h is the width and height of the image containing the pyramid and d_s denotes the width or height of the square window size at the base resolution, the following conditions should be satisfied:

$$\frac{d_w - d_s + 1}{16\Delta x} = \text{an integer} \tag{3.1}$$

$$\frac{d_h - d_s + 1}{16\Delta y} = \text{an integer} \tag{3.2}$$

If this is not the case, then the image containing the pyramid needs to be padded accordingly. Otherwise, thread blocks at the right and bottom edges would have more

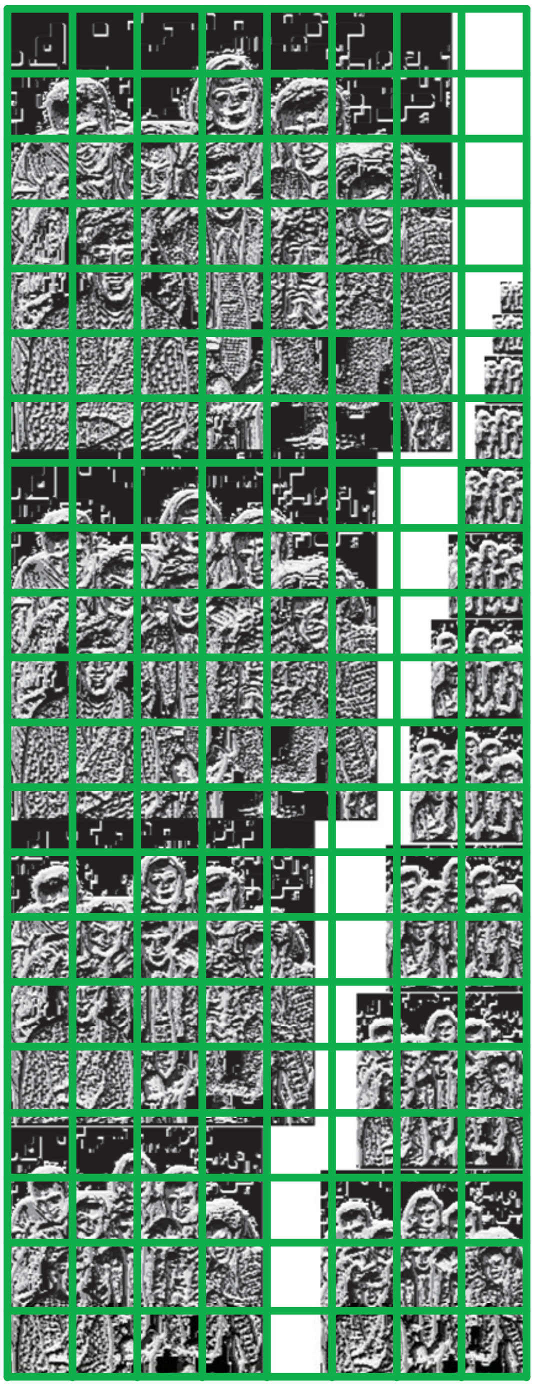

Figure 3.4: Arrangements of the thread blocks during the scanning process.

threads than the number of windows they need to classify and therefore the kernel would have to check whether the window corresponding to a thread exists or not. This conditional check would result in warp divergence and since the execution of the threads in such warps gets separated to 2 groups by the runtime and serialized, the performance would slightly decrease.

When a window is classified as the searched object, its coordinates are written to the corresponding location in a preallocated 1D array of floats that resides in global memory and has the same number of elements as the number of windows classified. Each element of this array can store the x and y coordinates of a window in its upper and lower two bytes, respectively. It is not known beforehand to which elements in this array the results will be written to. Therefore it is not possible to coalesce these global memory accesses, leading to long memory access latencies. Since the windows classified as positive is so rare, the GPU hardware easily finds another warps to execute until the memory access is finished, hiding the memory latency. Therefore the effect these access times have on the performance of the system is negligible.

3.1.3 Splitting the cascade into groups

As a natural consequence of using a cascaded classifier structure, the number of stages that will be evaluated in a window cannot be predetermined. Threads that classify their windows as negative in early stages have to wait idle until all other threads in the same block finish their tasks, after which the processing of a new block can be started. This results in under-utilization of GPU resources, especially in cases where vast majority of the threads in a block terminate early and wait for a small number of threads to finish. It also leads to divergence within warps because threads are grouped by the hardware according to the stage after which they terminate the classification process and the execution of these groups are serialized.

This problem can be dealt with by means of splitting the cascade into several smaller groups containing one or more stages and performing a separate kernel launch for each one of them. Deciding on how many times and at which locations the split should made is an optimization problem, which is easier to solve in the case of MCT based cascades because of their low stage numbers. Experiments showed that it is not beneficial to split the 5-stage cascade used in this work to more than 2 groups. Figure 3.5 shows the detection times according to the index of the stage after which the split is done. These timings do not include preprocessing times, because they are irrelevant for the comparison in question.

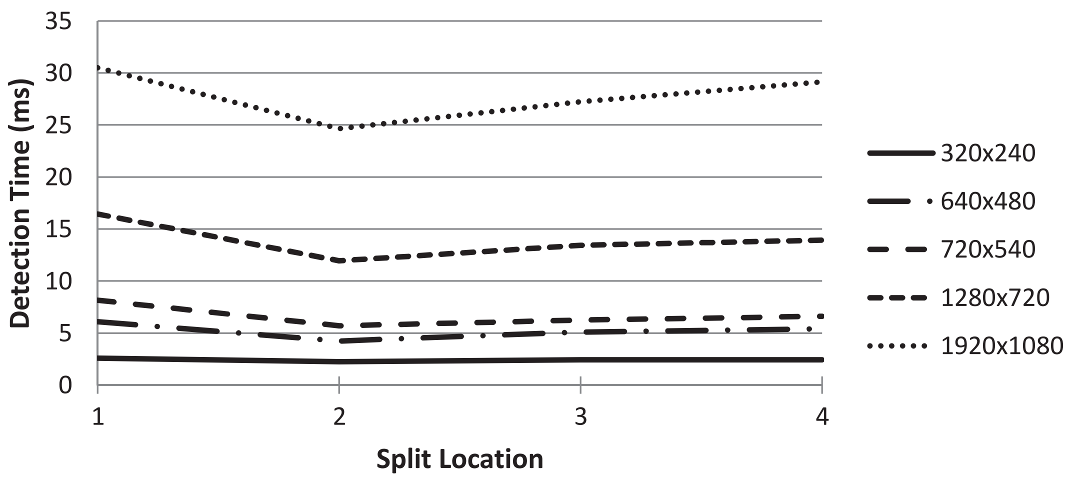

Figure 3.5: The effect of split location on the detection times.

Values in Figure 3.5 shows that the best location for the split is after the 2nd stage. Figure 3.6 shows the 2 partial cascades obtained after splitting the cascade into 2 parts. Each kernel launch utilizes only one of the cascade parts.

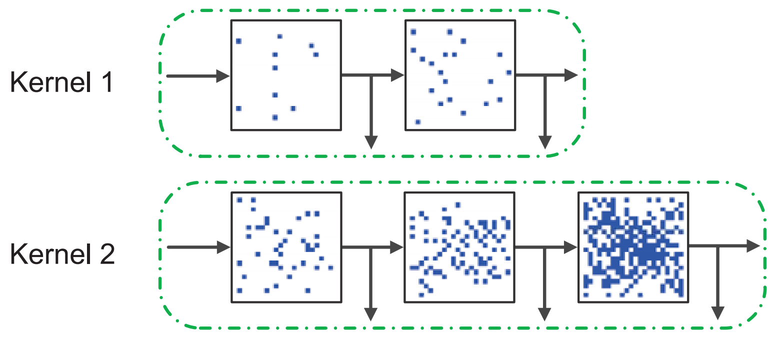

Figure 3.6: 2 partial cascades obtained after doing the split. Each kernel uses a separate part for classification.

Implementing the scanning process using two smaller cascades requires two different kernel launches. The first kernel, which is the same as the one before, classifies all windows using the first part of the cascade and writes the coordinates of the detections to a preallocated array containing dummy values in the global memory. Then another kernel is launched with a 1D grid of thread blocks, each one containing a 1D array of 256 threads. Each thread in the grid classifies a single window whose coordinates are fetched from the array filled by the first kernel. The final classification results are written back to another array containing dummy values in global memory.

This new scheme introduces another problem. The array generated by the first kernel contains sparse data. Most of its elements still contain dummy values that were set when it is first allocated. A good solution to this problem is using a stream compaction algorithm that copies all elements having meaningful values to the beginning of the array. Stream compaction can be performed on the GPU in 3 steps as shown in Figure 3.7. In the first step, each element in the input array is tested using a predicate and a mask array is obtained. In the case of object detection, this predicate is a function that checks whether the value of the cell is different than the dummy value. In the next step, the mask array is prefix summed. The elements of the resulting array show to which location each element should be copied. The final step involves doing a copy operation according to the indices found in the previous step and obtaining an array containing the meaningful values at its beginning.

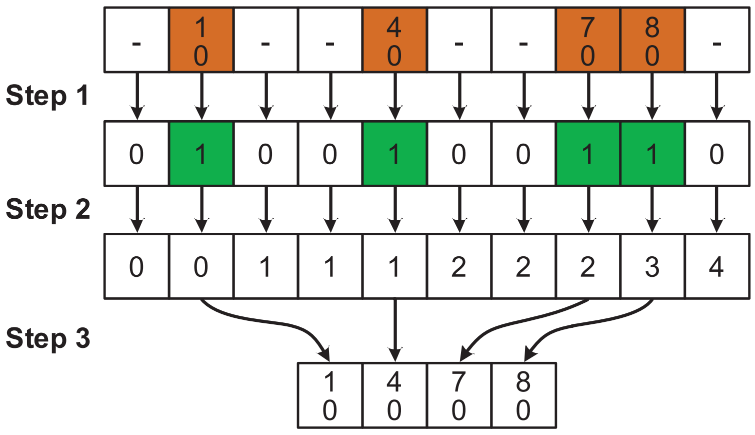

Figure 3.7: Steps of the stream compaction algorithm. Each element in the input array contains either a dummy value (shown as a dash), or a (x, y) coordinate pair.

This new approach does not completely eliminate the problem, but reduces it significantly. Vast majority of the windows get eliminated after the first 2 stages and therefore by launching a new kernel after the 2nd stage, the performance of the system is significantly improved, especially for larger resolutions. Figure 3.8 shows the steps of the whole scanning process for the first 10 windows in the input image.

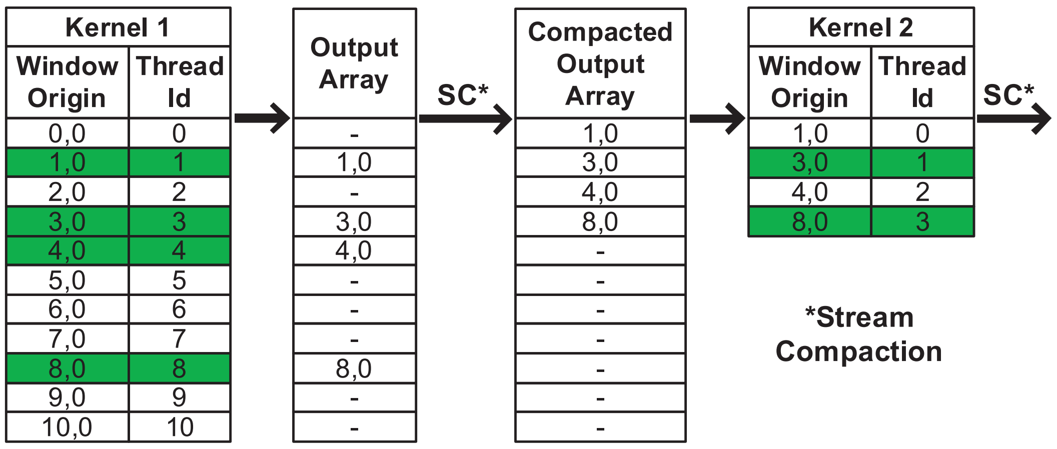

Figure 3.8: Steps of scanning with 2 kernels. Window locations that are detected as positive by the corresponding threads are highlighted.

3.1.4 Alternative approaches

One possible alternative to parallelizing by assigning each window to a single thread is, to make each thread classify multiple windows. The most important issue with this approach is the distribution of the windows to the threads, because it requires either inter-thread communication, or large amount of shared memory to be done properly, both of which is expensive in CUDA. The performance is somewhat improved when using smaller block sizes, but the results are not as fast as the other approach. Therefore it is preferred to use the simpler solution.

Another alternative is increasing the granularity of the parallelism by assigning each block to a single window location and making the threads in a block evaluate different weak classifiers. This approach completely eliminates the under-utilization problem as long as the number of weak classifiers in the stage is more than the number of CUDA threads. This immediately shows that the MCT based cascades are not suitable for this approach because unlike Haar based classifier cascades, even the last stage of the cascade has at most 484 weak classifiers when the base resolution of the classifiers is 24. Hence no matter how the grid and block sizes are chosen, the resources available in the GPU becomes heavily under-utilized.

3.1.5 Utilizing multiple GPUs

The implementation is extended to reduce the detection times even further on the devices that have multiple GPUs. Starting from CUDA Toolkit 4.0, it is possible to use multiple GPUs with a unified address space [50]. In order to support older platforms, the traditional approach is followed and spawn and $M + 1$ CPU threads are spawned, where M is the number of GPUs in the system. The task of the main CPU thread is acquiring frames from the video stream and doing preliminary computations. Each one of the other CPU threads performs CUDA runtime calls and memory transfers between the GPU assigned to it and the host. Each GPU generates several levels of the image pyramid and scans only those levels. This prevents unnecessary memory transfers between the host and the device. Since all GPUs need the original image to generate the levels assigned to them, and the whole cascade data to process these levels, both the original image and the cascade is copied to the memories of each GPU separately. Distributing different levels of the pyramid to different GPUs makes it possible to achieve nearly linear speed-up when the number of levels each GPU will process is carefully determined. This is achieved by pre-computing the total number of levels the pyramid will have and size of each of its levels and performing the distribution of the work according to these values.

3.2 Face Recognition

This section contains detailed information about the CUDA implementation developed for the LBP based face recognition algorithm.

3.2.1 Feature extraction

Feature extraction involves the computation of LBP values from the input image and the construction of regional histograms. These two operations can be done in a single kernel, without requiring to temporarily store the LBP values in the GPUs global memory, which has very high access latency.

In the proposed arrangement, each thread block computes the histogram of a single region R_j. This is achieved by making each warp in a thread block compute a per-warp histogram in shared memory, which are then combined by the threads in the block to obtain the regional, per-block histogram. By constructing the histograms in shared memory, this method eliminates the need of doing expensive atomic operations in global memory. The arrangement of the thread blocks for a 130×150 input image generated by the CSU face identification evaluation system (FIES) [51] divided to 7×7 regions is shown on the left side of Figure 3.9.

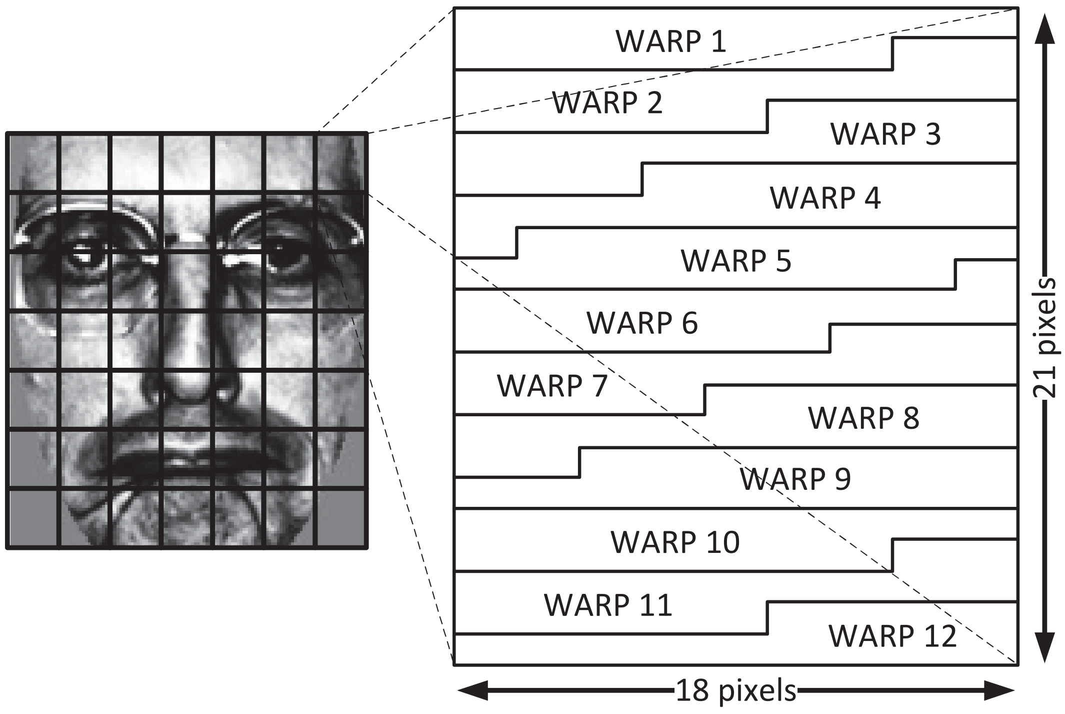

Figure 3.9: Arrangement of the thread blocks for a sample image divided to 7×7 regions. Each warp in a thread block constructs a per-warp histogram for a 32 pixel area in the corresponding region as shown on the right side of the figure.

If the number of warps in a block is N_w, and the number of bins in the histogram is N_b, the size of the shared memory required for a single block is $N_w * N_b * 2$ bytes, provided that the histogram values are chosen to be 16 bit integers. Figure 3.10 shows this memory layout for a single block.

The construction of a per-warp histogram is performed as follows: Each thread in the block applies the LBP operator to the neighbourhood centered at its location in the input image. In order to take advantage of the hardware linear interpolation capability of the GPU during the computation of LBP values, the input image is stored in the texture memory. The computed LBP value is converted to a uniform LBP value with the help of a look-up table stored in constant memory. This uniform LBP value is

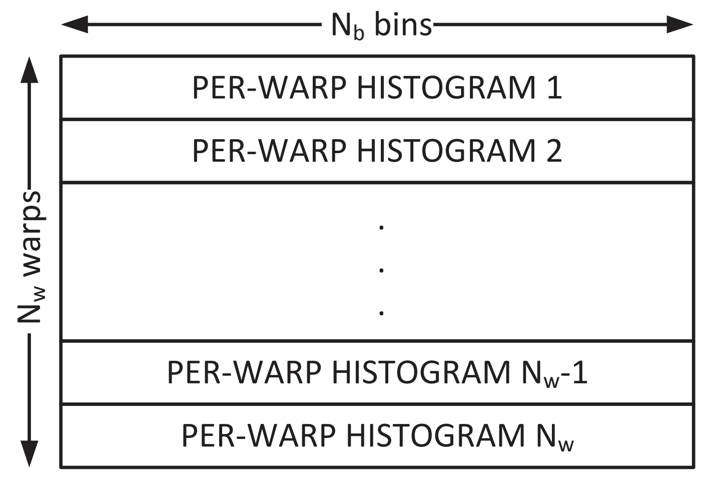

Figure 3.10: Shared memory layout for histogram computation

immediately used to update the corresponding per-warp histogram residing in shared memory. Since more than one thread in the same warp may try to increment the same bin, the increment operation needs to be atomic. In the case of using a GPU that does not have the capability to do atomic operations in shared memory, one can fall back to an alternative method called "tagging" as explained in [52]. Arrangement of the warps in a sample region is shown on the right side of Figure 3.9.

Before starting to construct the per-block histogram, to make sure that the construction of all per-warp histograms are finished, the threads in the block are synchronized with each other using the *__syncthreads()* primitive of CUDA. Then each block combines all of its per-warp histograms to obtain the regional histogram and writes the result to global memory.

3.2.2 Classification using the k-NN algorithm

The simplest approach to implement the k-NN algorithm in GPU is to make each GPU thread compute distances between two or more feature vectors as in [47]. This approach leads to good performance when the dimensionality is low. However, in LBP based face recognition, the dimensionality of the feature vector is high. For example, using a 7×7 grid and $(8, 2)^{u2}$ neighbourhoods (which results in a 59 bin histogram for each region) for LBP leads to a feature vector length of 2891. Because of this fact, a different approach is followed that is optimized for working with high dimensional feature vectors.

In order to utilize the GPU resources as much as possible even in cases where the number of vectors in the database is small, the feature vectors are split to p sub-vectors. The block sizes are set to be $32 \times p$ and a 1D grid of thread blocks that contains $\lceil N/32 \rceil$ blocks is launched, where N is the number of reference points in the database. The reason of selecting the block width as 32 is to make sure that a block contains exactly p warps and each row of threads in the block belongs to a single warp. i-th thread of the j-th warp computes the distance between the j-th sub-vector of a query point and j-th sub-vector of the i-th reference point. The computed partial distances are stored in shared memory. To make sure that all partial distances are computed before proceeding, the threads are synchronized with each other. Then, the threads in the first warp of each block sum the p partial distances computed by the p warps in the block and write the results to global memory. After all distances are computed, another kernel is launched that finds the index of the minimum (nearest) value. For $k > 1$, one can easily construct a loop in which the value at the index found by the previous iteration is replaced with a very high value and then the index of the new minimum value is found using the same kernel. However, as far as face recognition is concerned, the value of k will in most cases be set to 1.

The query point and region weights are stored in texture memory and constant memory, respectively, because they are heavily accessed by all threads and their small sizes lead to nearly $\%100$ cache hit ratio. Accesses to the reference points are coalesced, which eliminates the need for them to be stored in a cached memory by utilizing the memory bandwidth very effectively. Therefore the reference points are stored in global memory.

It is possible to extend the implementation to handle multiple query points at once by making each thread compute distances between a reference point and all query points, in which case the amount of shared memory requirement will increase. One can also simply launch the distance calculation and reduction kernels one time for each query point without doing any modifications in the algorithm, if the number of query points is small.

4. RESULTS

In this section, the performances of the proposed GPU implementations are evaluated by comparing their speed and accuracy with those of the corresponding CPU implementations. All comparions are performed on a desktop PC containing a Intel Core i5-2500k processor, 3 GTX 580 GPUs and 8GB RAM. The operating system on the test PC is Windows 7 and the version of the CUDA Toolkit is 4.0.

4.1 Face Detection

The performance of the CPU and GPU implementations are tested both on video streams of 5 different resolutions and on still images in the CMU+MIT frontal face test set f [5]. All measurements include memory transfers between the host and the device.

4.1.1 GPU vs CPU comparison on video streams

Figure 4.1 shows the average number of frames processed per second by the GPU and CPU implementations on video streams of various sizes. These measurements include the time required to perform preprocessing and memory copies between the host and the device, but not the time required for video decoding or displaying. The multi-threaded CPU implementation uses OpenMP to distribute the processing to different cores.

As it can be seen from Figure 4.1, even the single-GPU implementation outperforms the single-threaded and multi-threaded CPU implementations by a factor of 12-18x and 4-6x, respectively. As the resolution increases, so does the difference between the speed of the GPU and CPU implementations, clearly showing that a GPU is better suited to process high resolution videos than a CPU. These results also show that the performance of the GPU based implementation scales nearly linearly with the number

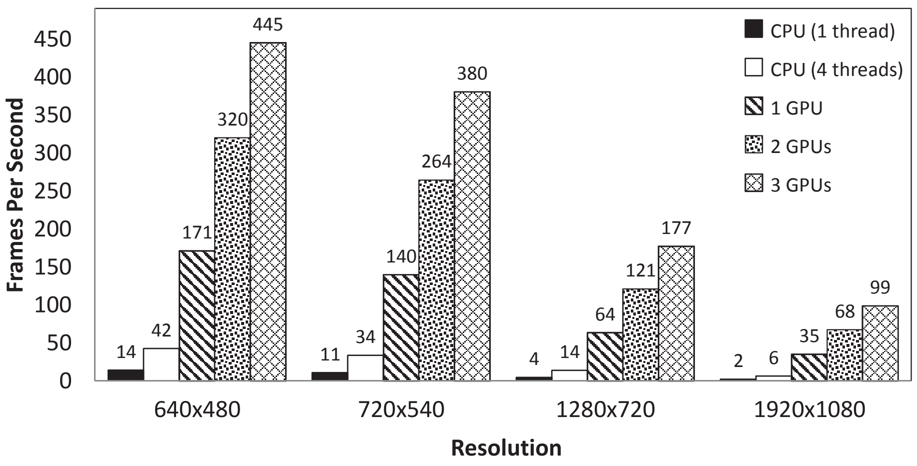

Figure 4.1: Frame rates of GPU and CPU implementations on various input resolutions.

of GPUs, in contrast to the CPU which wastes considerable amount of time because of the overhead involved with software scheduling.

In Table 4.1, the preprocessing and scanning times are listed separately for the single-GPU case. Values in the table show that for smaller resolutions, the preprocessing time is comparable to that of the scanning time. As the resolution gets higher, the difference between the time required to scan the pyramid and perform the preprocessing increases.

Table 4.1: The preprocessing and scanning times for the single-GPU case.

Resolution	Preprocessing [ms]	Scan [ms]	Total [ms]
320×240	0.54	2.25	2.79
640×480	1.32	4.52	5.84
720×540	1.54	5.61	7.15
1280×720	3.74	11.94	15.68
1920×1080	3.88	24.64	28.52

It should be noted that these measurements have been done when using the scanning parameters given in Section 2.1.3. It is possible to increase the detection speed drastically just by increasing starting scale to 2, in which case the system will not be able to detect faces smaller than 48×48. This might not be important for some applications running on high resolution video streams.

4.1.2 GPU vs CPU comparison on still images

The performance of the single-GPU and single-threaded CPU implementations are tested on the still images in the CMU+MIT frontal face test set. The total time the GPU needed to process all 132 images in the dataset, excluding the time required to read the images from disk, is measured as 1.82 seconds, while it took 22.1 seconds for the CPU to do the same processing. According to the measurements performed, because of the empty regions in the image pyramid, the GPU implementation had to evaluate %20 more windows, but still managed to finish its job 12x quicker than the CPU implementation. The reason that the GPU did perform only 12x faster is the fact that most of the images in the dataset have low resolutions. The detection rate for both implementations are measured as %90.8, while the total number of false positives is 32. This proves that the GPU implementation has the exact same detection accuracy with the CPU implementation.

4.2 Face Recognition

The performance of the single-threaded CPU and GPU implementations are tested on the FERET [39] face database using the 130×150 sized images generated by CSU FIES [51]. Fa and Fb subsets are used as the gallery and test sets, respectively, when measuring the average processing times. All measurements include memory transfers between the host and the device.

4.2.1 Feature extraction

Table 4.2 shows the feature extraction times of the GPU and CPU implementations for various cases including the ones that give the best accuracy according to [4]. Length of the resulting feature vector (d) is also listed for each case. Values in the table indicate that the GPU performs 23-44x faster than CPU. When using a GPU that supports concurrent kernel execution, one can use multiple streams to compute feature vectors of multiple input images concurrently as long as the number of SMs in the GPU is sufficient. Results listed in the table are obtained using a single stream. It should be noted that, because of its hardware interpolation capability, the performance of the

GPU decreases only by $\%5$ when the number of sampling points is doubled, in contrast to the performance of the CPU that decreases by $\%75$.

Table 4.2: Feature extraction times of the GPU and CPU implementations for various cases using a single stream.

LBP Type	Region Size	Feature Length	GPU [ms]	CPU [ms]
$(8,2)^{u2}$	11×13	8496	0.17	4.25
$(8,2)^{u2}$	18×21	2891	0.17	3.85
$(8,2)^{u2}$	26×30	1475	0.17	4.01
$(16,2)^{u2}$	18×21	11907	0.18	7.45
$(16,2)^{u2}$	26×30	6075	0.18	7.86

4.2.2 Classification

Table 4.3 compares the performance of the proposed k-NN implementation (denoted as GPU) with those of the CUDA implementation of k-NN in [47] (denoted as GPUX) and the ANN C++ library [53] for some of the feature lengths (d) listed in Table 4.2 and various numbers of reference points (N).

Table 4.3: 1-NN search times (in ms) of GPU and CPU implementations for various values of d and N when p=8.

Size	Method	d=2891	d=8496	d=11907
N=1000	GPU	0.88	1.12	1.52
	GPUX	2.06	2.78	3.65
	ANN	2.55	7.47	10.46
N=5000	GPU	1.09	1.61	2.15
	GPUX	7.48	10.13	13.32
	ANN	12.72	37.32	52.32
N=10000	GPU	1.52	2.77	3.72
	GPUX	13.68	19.44	25.44
	ANN	25.46	74.71	104.51
N=20000	GPU	2.21	5.18	7.38
	GPUX	18.55	37.40	51.72
	ANN	52.47	151.13	210.46

In order to increase the number of reference points in the database when needed, some of the images in the Fa subset are duplicated. Parameters of the ANN library are set to give the exact nearest neighbours and best performance. For all implementations, k is selected as 1 because it is most of the time the case due to the limited number of reference images in real-world applications. Finally, the value of p in the GPU implementation is set to 8 because it results in the best performance overall. The results indicate that the proposed GPU implementation performs 3-29x faster than ANN and 2-9x faster than GPUX. The speed difference between the CPU and GPU implementations increases with both the number of reference points and the length of the feature vector.

Combining the computation times given Table 4.2 and Table 4.3 shows that the proposed GPU implementation performs the whole recognition process 6-29x faster than the CPU implementation that uses ANN and 2-8x faster than the GPU implementation that uses GPUX.

5. CONCLUSIONS AND FUTURE WORK

In this thesis, efficient GPU implementations for a boosting based, real-time face detection algorithm and a feature based face recognition algorithm is presented. For the sake of comparison, efficient CPU implementations of the same algorithms are also developed.

The performances of the CPU and GPU implementations of the face detection algorithm are evaluated on video streams with resolutions ranging from 640×480 to 1920×1080 and on a widely used face database containing still images. Comparisons between the performances of the single-GPU, multi-GPU, single-threaded CPU and multi-threaded-CPU implementations are performed. The results showed that even the single-GPU implementation is able to detect faces up to 6x and 18x faster than the single-threaded and multi-threaded CPU implementations running on a modern CPU, respectively. It is pointed out that, because of the GPUs massively parallel architecture, the speed difference between the GPU and CPU implementations increases with the resolution of the input image and therefore GPUs are more suitable for working with high resolution videos or images than CPUs. The proposed implementation, with its ability to detect objects in a video stream having resolutions as high as 1920×1080 in real-time, can easily be used in modern multimedia, entertainment and surveillance systems.

The frontal face detection system implemented in this thesis can be extended to detect multi-view faces. Each GPU or CPU core can be used for detecting faces from a specific angle or better arrangements could be found as a subject of future research. In any similar object detection application requiring the evaluation of multiple cascades, implementing a heterogeneous system utilizing all CPU cores and all GPUs at the same time can lead to even more improvements in the speed, leaving more processing time for the other algorithms that will follow.

In addition to the face detection, efficient GPU implementations for LBP computation, regional histogram construction and k-NN classification, are also presented, which are the 3 steps of a LBP based face recognition algorithm. By utilizing the GPU in the face recognition process, recognizing faces in real time ceases being an issue even on large databases.

The increase in the speed and more efficient use of the resources of the computer will prove much more useful when the face processing is done in a multi-view and rotation invariant manner, which involves much higher amount of computations. Development of such heterogeneous systems that take advantage of not only the GPU cores, but also all cores the CPU are the subject of future research.

With the price of fast computing hardware going down, the number of cores in processors are going up and high definition videos becoming increasingly common, the need for fast, heterogeneous algorithms that can utilize all available processors in the system and run real-time on these high resolution streams will certainly increase in the future and the development of such systems will become an important research topic.

REFERENCES

[1] **Fröba, B. and Ernst, A.**, (2004). Face detection with the modified census transform, *Proceedings of the Sixth IEEE international conference on Automatic face and gesture recognition*, FGR' 04, IEEE Computer Society, Washington, DC, USA, pp.91–96.

[2] **Freund, Y. and Schapire, R.E.**, (1995). A Decision-theoretic Generalization of On-line Learning and an Application to Boosting, *Proceedings of the Second European Conference on Computational Learning Theory*, Springer-Verlag, London, UK, pp.23–37.

[3] **Viola, P. and Jones, M.**, (2001). Rapid Object Detection using a Boosted Cascade of Simple Features, volume 1, IEEE Computer Society, Los Alamitos, CA, USA, pp.511–518.

[4] **Ahonen, T.**, **Hadid, A. and Pietikainen, M.**, (2006). Face Description with Local Binary Patterns: Application to Face Recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **28**, 2037–2041.

[5] **Rowley, H.A.**, **Baluja, S. and Kanade, T.**, (1998). Neural Network-Based Face Detection, *IEEE Trans. Pattern Anal. Mach. Intell.*, **20**, 23–38.

[6] **Schneiderman, H. and Kanade, T.**, (2000). A statistical method for 3D object detection applied to faces and cars, *Proceedings IEEE Conference on Computer Vision and Pattern Recognition CVPR 2000*, **1**, 746–751.

[7] **Sung, K.K. and Poggio, T.**, (1998). Example-Based Learning for View-Based Human Face Detection, *IEEE Trans. Pattern Anal. Mach. Intell.*, **20**, 39–51.

[8] **Lienhart, R. and Maydt, J.**, (2002). An Extended Set of Haar-Like Features for Rapid Object Detection, *Proceedings of the IEEE 2002 International Conference on Image Processing*, pp.900–903.

[9] **Bradski, G.**, (2000). The OpenCV Library, *Dr. Dobb's Journal of Software Tools*.

[10] **Viola, M.**, **Jones, M.J. and Viola, P.**, (2003). Fast Multi-view Face Detection, *Proc. of Computer Vision and Pattern Recognition*.

[11] **Huang, C.**, **Ai, H.**, **Li, Y. and Lao, S.**, (2006). Learning Sparse Features in Granular Space for Multi-View Face Detection, *Proceedings of the 7th International Conference on Automatic Face and Gesture Recognition*, FGR '06, IEEE Computer Society, Washington, DC, USA, pp.401–407.

[12] **Mita, T.**, **Kaneko, T. and Hori, O.**, (2005). Joint Haar-like Features for Face Detection, *Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*, ICCV '05, IEEE Computer Society, Washington, DC, USA, pp.1619–1626.

[13] **Hradis, M.**, **Herout, A. and Zemcik, P.** Local Rank Patterns — Novel Features for Rapid Object Detection, *Proceedings of the International Conference on Computer Vision and Graphics: Revised Papers*, ICCVG '08.

[14] **Yan, S.**, **Shan, S.**, **Chen, X. and Gao, W.** *Computer Vision and Pattern Recognition, IEEE Conference on*.

[15] **Jin, H.**, **Liu, Q.**, **Lu, H. and Tong, X.**, (2004). Face Detection Using Improved LBP under Bayesian Framework, *Proceedings of the Third International Conference on Image and Graphics*, ICIG '04, IEEE Computer Society, Washington, DC, USA, pp.306–309.

[16] **Zhang, L.**, **Chu, R.**, **Xiang, S.**, **Liao, S. and Li, S.Z.** Face Detection Based on Multi-Block LBP Representation, *Advances in Biometrics: International Conference, Proceedings of*.

[17] **Li, S.Z.**, **Zhu, L.**, **Zhang, Z.**, **Blake, A.**, **Zhang, H. and Shum, H.**, (2002). Statistical Learning of Multi-view Face Detection, *Proceedings of the 7th European Conference on Computer Vision-Part IV*, ECCV '02, Springer-Verlag, London, UK, pp.67–81.

[18] **Wu, B.**, **Ai, H.**, **Huang, C. and Lao, S.**, (2004). Fast rotation invariant multi-view face detection based on real adaboost, *Proceedings of the Sixth IEEE International Conference on Automatic Face and Gesture Recognition*, FGR '04, IEEE Computer Society, Washington, DC, USA, pp.79–84.

[19] **Lienhart, R.**, **Kuranov, A. and Pisarevsky, V.**, (2003). Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection, pp.297–304.

[20] **Brubaker, S.C.**, **Wu, J.**, **Sun, J.**, **Mullin, M.D. and Rehg, J.M.**, (2008). On the Design of Cascades of Boosted Ensembles for Face Detection, *Int. J. Comput. Vision*, **77**, 65–86.

[21] **Xiao, R.**, **Zhu, L. and Zhang, H.J.**, (2003). Boosting Chain Learning for Object Detection, *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, IEEE Computer Society, Washington, DC, USA, pp.709–715.

[22] **Sochman, J. and Matas, J.**, (2005). WaldBoost - Learning for Time Constrained Sequential Detection, *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '05, IEEE Computer Society, Washington, DC, USA, pp.150–156.

[23] **Bourdev, L. and Brandt, J.**, (2005). Robust Object Detection via Soft Cascade, *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '05, IEEE Computer Society, Washington, DC, USA, pp.236–243.

[24] **Schneiderman, H.**, (2004). Feature-centric evaluation for efficient cascaded object detection, *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, IEEE Computer Society, Washington, DC, USA, pp.29–36.

[25] **Sharma, B.**, **Thota, R.**, **Vydyanathan, N. and Kale, A.**, (2009). Towards a robust, real-time face processing system using CUDA-enabled GPUs., *HiPC '09*, pp.368–377.

[26] **Hefenbrock, D.**, **Oberg, J.**, **Thanh, N.T.N.**, **Kastner, R. and Baden, S.B.**, (2010). Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs, *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, IEEE Computer Society, Washington, DC, USA, pp.11–18.

[27] **Obukhov, A.**, (2004). Haar Classifiers for Object Detection with CUDA, **R. Fernando**, editor, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, chapter 33, Addison Wesley, pp.517–544.

[28] **Harvey, J.P.**, (2009), GPU Acceleration of Object Classification Algorithms Using NVIDIA CUDA.

[29] **Herout, A.**, **Jošth, R.**, **Juránek, R.**, **Havel, J.**, **Hradiš, M. and Zemčík, P.**, (2010). Real-time Object Detection on CUDA, *Journal of Real-Time Image Processing*, **2010**(1), 1–12.

[30] **Turk, M. and Pentland, A.**, (1991). Eigenfaces for recognition, *J. Cognitive Neuroscience*, **3**, 71–86.

[31] **Belhumeur, P.N.**, **Hespanha, J.a.P. and Kriegman, D.J.**, (1997). Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **19**, 711–720.

[32] **Bartlett, M.**, **Movellan, J. and Sejnowski, T.**, (2002). Face Recognition by Independent Component Analysis, *Neural Networks, IEEE Transactions on*, **13**(6), 1450–1464.

[33] **He, X.**, **Yan, S.**, **Hu, Y.**, **Niyogi, P. and Zhang, H.J.**, (2005). Face Recognition using Laplacianfaces, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **27**, 328–340.

[34] **Guo, G.**, **Li, S. and Chan, K.**, (2000). Face Recognition by Support Vector Machines, *Proceedings of the Fourth IEEE International Conference on Automatic Face and Gesture Recognition*, FGR '00, pp.196–201.

[35] **Heisele, B.**, **Ho, P. and Poggio, T.**, (2001). Face Eecognition with Support Vector Machines: Global versus Component-based Approach, *Proceedings of the Eighth IEEE International Conference on Computer Vision*, volume 2 of *ICCV '01*, pp.688–694.

[36] **Penev, P. and Atick, J.**, (1996). Local Feature Analysis: A General Statistical Theory for Object Representation, *Network Computation in Neural Systems*, **7**, 477–500.

[37] **Wiskott, L.**, **Fellous, J.M.**, **Krüger, N. and von der Malsburg, C.**, (1997). Face Recognition by Elastic Bunch Graph Matching, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **19**, 775–779.

[38] **Ojala, T.**, **Pietikäinen, M. and Mäenpää, T.**, (2002). Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **24**, 971–987.

[39] **Phillips, P.J.**, **Moon, H.**, **Rizvi, S.A. and Rauss, P.J.**, (2000). The FERET Evaluation Methodology for Face-Recognition Algorithms, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **22**, 1090–1104.

[40] **Zhang, G.**, **Huang, X.**, **Li, S.**, **Wang, Y. and Wu, X.**, (2005). Boosting Local Binary Pattern (LBP)-Based Face Recognition, **S. Li**, **J. Lai**, **T. Tan**, **G. Feng and Y. Wang**, editors, Advances in Biometric Person Authentication, volume3338 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp.179–186.

[41] **Zhang, B.**, **Gao, Y.**, **Zhao, S. and Liu, J.**, (2010). Local Derivative Pattern versus Local Binary Pattern: Face Recognition with High-order Local Pattern Descriptor, *IEEE Transactions on Image Processing*, **19**, 533–544.

[42] **Zhang, B.**, **Shan, S.**, **Chen, X. and Gao, W.**, (2007). Histogram of Gabor Phase Patterns (HGPP): A Novel Object Representation Approach for Face Recognition, *IEEE Transactions on Image Processing*, **16**(1), 57–68.

[43] **Singh, C.**, **Mittal, N. and Walia, E.**, (2011). Face Recognition using Zernike and Complex Zernike Moment Features, *Pattern Recognition and Image Analysis*, **21**, 71–81.

[44] **Bentley, J.L.**, (1975). Multidimensional Binary Search Trees used for Associative Searching, *Commun. ACM*, **18**, 509–517.

[45] **Muja, M. and Lowe, D.G.**, (2009). Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration, *Proceedings of the International Conference on Computer Vision Theory and Applications*, VISAPP '09, INSTICC Press, pp.331–340.

[46] **Gionis, A.**, **Indyk, P. and Motwani, R.**, (1999). Similarity Search in High Dimensions via Hashing, *Proceedings of the 25th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp.518–529.

[47] **Garcia, V.**, **Debreuve, E. and Barlaud, M.**, (2008). Fast k Nearest Neighbor Search using GPU, *CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA, pp.1–6.

[48] **Liang, S.**, **Wang, C.**, **Liu, Y. and Jian, L.**, (2009). CUKNN: A Parallel Implementation of k-Nearest Neighbor on CUDA-enabled GPU, *IEEE Youth Conference on Information, Computing and Telecommunication*, pp.415 –418.

[49] **Ojala, T.**, **Pietikäinen, M. and Harwood, D.**, (1996). A comparative study of texture measures with classification based on featured distributions, *Pattern Recognition*, 51–59.

[50] **NVIDIA**, (2011), NVIDIA CUDA C Programming Guide Version 4.0, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA

[51] **Beveridge, J.R.**, **Bolme, D.**, **Draper, B.A. and Teixeira, M.**, (2005). The CSU Face Identification Evaluation System, *Machine Vision and Applications*, **16**, 128–138.

[52] **Shams, R. and Kennedy, R.A.**, (2007). Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices, *Proc. Int. Conf. on Signal Processing and Communications Systems*, pp.418–422.

[53] **Arya, S.**, **Mount, D.M.**, **Netanyahu, N.S.**, **Silverman, R. and Wu, A.Y.**, (1998). An optimal algorithm for approximate nearest neighbor searching fixed dimensions, *J. ACM*, **45**, 891–923.

CURRICULUM VITAE

Name Surname: Salih Cihan TEK

Place and Date of Birth: İstanbul, 1986

Adress: İTÜ, Fen Bilimleri Enstitüsü, Ayazağa Kampüsü, 34469, Maslak, İstanbul

E-Mail: tek@itu.edu.tr

B.Sc.: İstanbul Technical University, Faculty of Electrical and Electronics Engineering, Department of Electrical Engineering, 2008

Professional Experience and Rewards:
National Scholarship for M.Sc. Students, The Scientific and Technological Research Council of Turkey (TÜBİTAK)

Best Student Paper Award in IEEE 20th Conference on Signal Processing and Communications Applications (SIU 2012)

List of Publications and Patents:

- Sarıyanidi E., Dağlı V., **Tek S. C.**, Tunç B., and Gökmen M. (2012). Local Zernike Moments: A New Representation for Face Recognition. *IEEE 19th International Conference on Image Processing (ICIP)*, Accepted.
- Sarıyanidi E., Dağlı V., **Tek S. C.**, Tunç B., and Gökmen M. (2012). A Novel Face Representation using Local Zernik Moments. *IEEE 20th Conference on Signal Processing and Communications Applications (SIU 2012)*, 18-20 April, Muğla, Turkey.
- Sarıyanidi E., **Tek S. C.**, and Gökmen M. (2011). Efficient Face Detection using Coarse Sampling. *IEEE 19th Conference on Signal Processing and Communications Applications (SIU 2011)*, 20-22 April, Antalya, Turkey.

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- **Tek S. C.**, and Gökmen M. (2012). GPU Accelerated Real-Time Object Detection on High Resolution Videos using Modified Census Transform, *Proceedings of the International Conference on Computer Vision Theory and Applications - Volume 1*, VISAPP '12, SciTePress, pp.685-688.