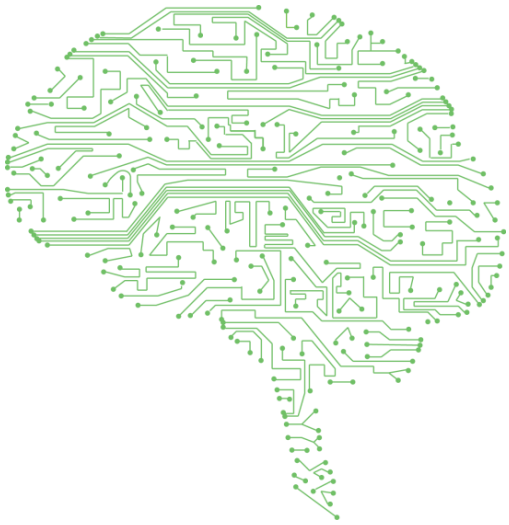
Technical University of Crete
Electronic and Computer Engineering Department
Microprocessor & Hardware Laboratory

Diploma Thesis



Implementation of Object Detecting and Tracking Algorithms on Parallel Graphical Processing Units

Author:
Theofilos Paganos

Supervisor:
Assistant Professor:
Yannis Papaefstathiou

June 2013

Abstract

Object Detection and Video Tracking are important subfields of Computer Vision science, which is a rapidly growing field of multimedia and autonomous robotic systems. In spite of the fact that a significant number of applications have been developed, generating very accurate results, they lack in processing efficiency, fast time execution and low power consumption. The reason is that these applications are highly complex schemes with many heavyweight sub-algorithms. For this thesis we studied two algorithms of Computer Science, the Receptive Field Cooccurence Histograms (RFCH) and the Tracking-Learning-Detecting (TLD). They belong to the subfields of Computer Science, Object Detection and Video Tracking respectively. Our focus was on the identification of the algorithms hotspots that are responsible for their execution bulk, and the improvement of their performance. For the acceleration of their performance we experimented with scalable hardware, especially with NVidia graphics processing units. These hardware cards can be programmed via a parallel computing platform called CUDA (Compute Unified Device Architecture), towards the needs of a developer. Our needs were to parallelize the hotspots of RFCH and TLD in an efficient way so as to improve their performance, a task in which we succeeded.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter we introduce the reader to computer vision, a field of computer science we have studied, and how it has been applied since robust algorithms have appeared. Also, we introduce the CUDA (Compute Unified Device Architecture) technology which uses NVidia GPUs for supercomputing, and the interaction between CUDA and computer vision.

1.1 Computer vision science

Computer vision is a field that includes methods for acquiring, processing and analyzing high-dimensional data from the real world in order to produce numerical or symbolic information. A theme in the development of this field has been to duplicate the abilities of human vision by electronically perceiving and understanding an image. This understanding of the image can be seen as the disentangling of symbolic information from image data using models constructed with the aid of geometry, physics, statistics, and learning theory. Computer vision has also been described as the enterprise of automating and integrating a wide range of processes and representations for vision perception [1].

Applications range from tasks such as industrial machine vision systems which, say, inspect bottles speeding by on a production line, to research into artificial intelligence and computers or robots that can comprehend the world around them. The computer vision and machine vision fields have significant overlap. Computer vision covers the core technology of automated image analysis, which is used in many fields. Machine vision usually refers to a process of combining automated image analysis with other methods and technologies to provide automated inspection and robot guidance in industrial applications.

As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. The image data can take

many forms, such as video sequences, views from multiple cameras, or multi-dimensional data from a medical scanner.

As a technological discipline, computer vision seeks to apply its theories and models to the construction of computer vision systems. Some examples of applications of computer vision may include systems for:

- Controlling processes, i.e., industrial robots
- Navigation, e.g., by an autonomous vehicle or mobile robot
- Visual surveillance or people counting
- Indexing databases of images and image sequences
- Modeling objects or environments, e.g., medical image analysis or topographical modeling
- Automatic inspection, as in manufacturing applications.

Sub-domains of computer vision include scene reconstruction, event detection, video tracking, object recognition, object detection, learning (training), indexing, motion estimation, and image restoration.

In most practical computer vision applications, the computers are pre-programmed to solve a particular task, but methods based on learning are now becoming increasingly common.

1.2 Compute Unified Device Architecture (CUDA)

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVidia and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA enabled GPUs. Using CUDA, the latest NVidia GPUs become accessible for computation like CPUs. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general-purpose problems on GPUs is known as GPGPU.

The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives and extensions to industry-standard programming languages, including C, C++ and Fortran. C/C++ programmers use CUDA C/C++ compiled with nvcc. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, MATLAB, IDL, and native support in Mathematica.

In the CUDA programming framework, the GPU is viewed as a compute device that is a co-processor to the CPU. The GPU has its own DRAM, referred to as device memory, and executes a very high number of threads in parallel. More precisely, data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads. In order to organize threads running in parallel on the GPU, CUDA organizes them into logical blocks. Each block is mapped onto a multiprocessor in the GPU. All the threads in one block can be synchronized together and communicate with each other. Because there are a limited number of threads that a block can contain, these blocks are further organized into grids allowing for a larger number of threads to run concurrently, as illustrated in Figure 1.1.
As we can see, there is a strict memory hierarchy for the kind of memory each resource can access. 32-bit registers and local memory are accessed only by

unique threads of the grid, a shared memory fragment can be accessed only by threads that reside on a unique block and any thread of any block can access Global, Constant and Texture Memory.
Threads in different blocks cannot be synchronized, nor can they communicate even if they are in the same grid. All the threads in the same grid run the same GPU code. The GPU contains many streaming multiprocessors (SMs) and each SM has a number of processor cores, depending on the compute capability and how technologically advanced a card is.
For this thesis we studied two cards of compute capabilities 1.3 and 2.0. These are the NVidia GTX 285 and the NVidia GTX 580 respectively. The GTX 285 has 30 SMs with 8 cores in each SM, while the GTX 580 has 16 SMs with 32 cores in each SM.

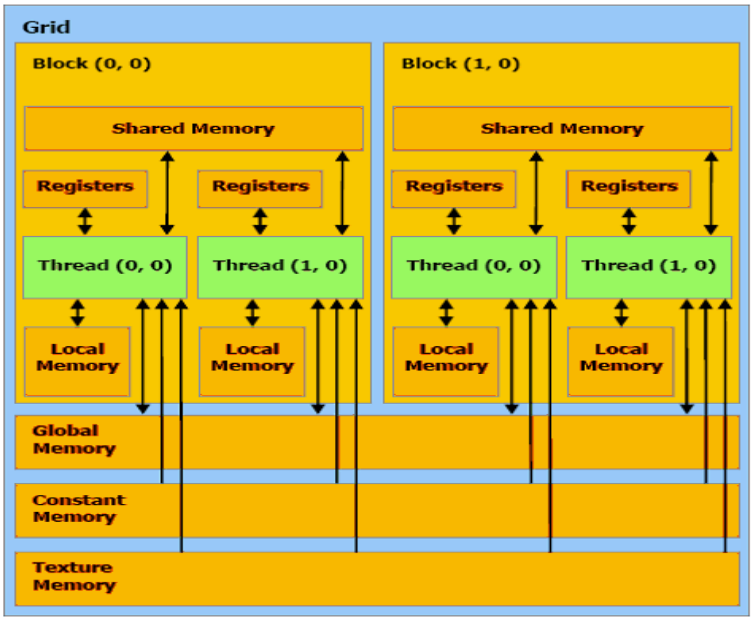


Figure 1.1: Architecture of a computational grid and the interaction between the GPU resources and different memory elements

As we mentioned, blocks are scheduled to a SM so they can be processed not only by the cores it contains, but also with the help of other resources, such are the maximum number of threads a SM can handle, shared memory, registers, constant and texture caches as shown in Figure 1.2. The number of blocks that are assigned to a SM each time depends on the maximum number of threads, shared memory and registers the kernel needs to run.

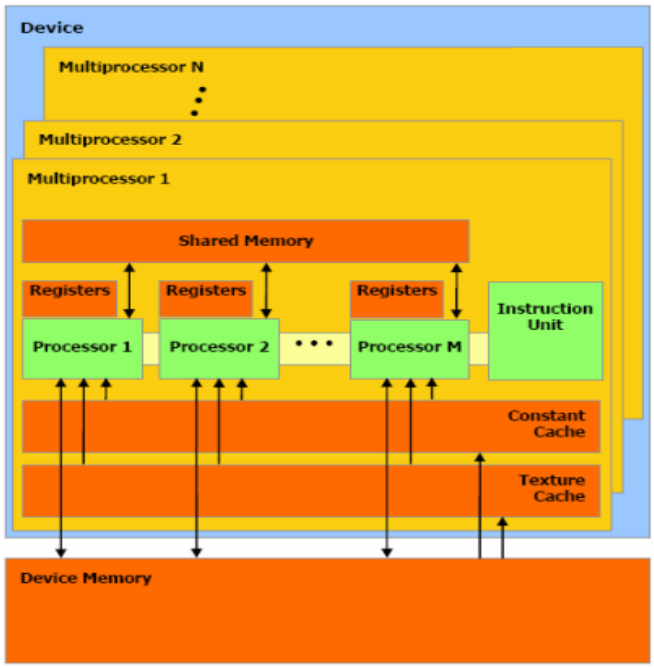


Figure 1.2: Multiprocessors with their components.

CUDA enabled GPUs have grown their capabilities in the past years with amazing results in computations over a period of time. The Figure 1.3 shows how many billions of floating-point operations are executed per second in comparison to the CPUs.

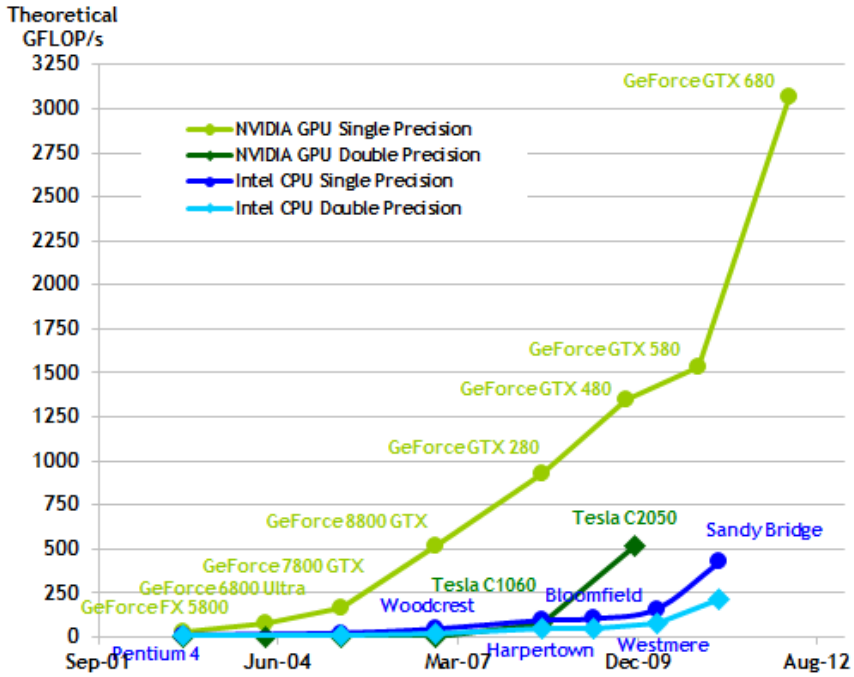


Figure 1.3: Floating operations per second

The Figure 1.4 shows the theoretical bandwidth of the Global Memory in different GPUs in comparison to CPU RAM.

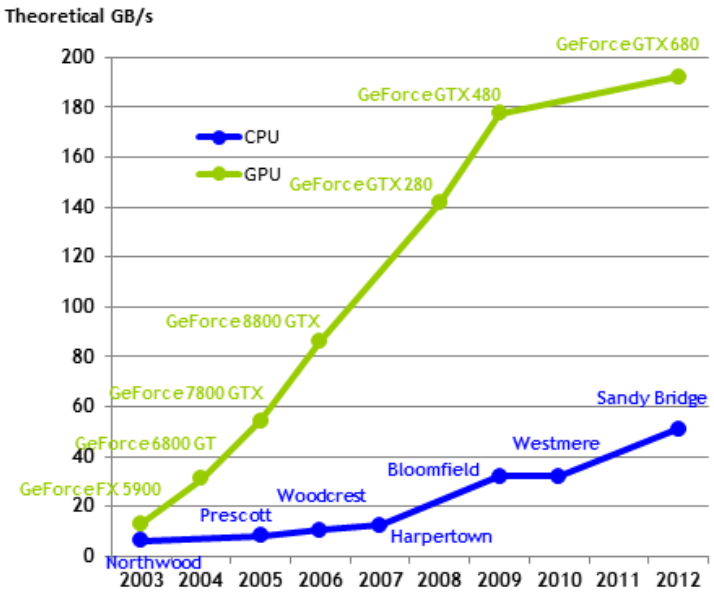


Figure 1.4: Comparison of the bandwidth of Global Memory towards RAM.

1.3 Computer Vision through Hardware

Although Computer Vision theory and its applications have been evolved and developed in the last years with great results and state of the art programs, they lack in processing efficiency, fast time execution and low power consumption. These facts are serious drawbacks for real world applications, such as real time robotic systems.

There are different ways in which computer vision algorithms analyze data but they almost all are heavy and complex, computation-wise. This increases the power consumption of the CPU and time execution. One way to overcome these problems is to wait for computer vision science to come up with new ideas for its data processing or try to implement parts of an algorithm that are heavyweight on hardware such are Field-programmable Gate Arrays (FPGAs) and Graphics Processor Units (GPUs).

By taking advantage of the parallel computing nature of hardware, processing time can be reduced dramatically while keeping low the wattage that is consumed for complex calculations.
Specifically for FPGAs, data can be divided into chunks and pass each chunk into custom processors that work in parallel, while for GPUs, data can be transferred to their vast device memory, and with the help of billions of available threads that work virtually in parallel, great results can be achieved by extracting the inherent parallelism of a computer vision algorithm.
All this comes down to better performance and to a realistic use of computer vision programs in everyday life.

Chapter 2

Object Recognition and Video Tracking

In this chapter we try to analyze two of the many subfields of computer vision science, object recognition and video tracking. We chose these because the algorithms we studied for this thesis belong to these two major categories.

2.1 Object Recognition and Object Detection

Object Recognition in computer vision is the task of finding a given object in an image or video sequence. Humans recognize a multitude of objects in images with little effort, despite the fact that the image of the objects may vary somewhat in different viewpoints, in many different sizes and scales or even when they are translated or rotated. Objects can even be recognized when they are partially obstructed from view. This task is still a challenge for computer vision systems in general [2].

An object recognition algorithm is typically designed to classify an object in one of several predefined classes, assuming that the segmentation of the object has already been performed. Commonly, the test images show a single object that is centered in the image and occupies most of the image area. The test image may also have a black background [3], making the task even more simple.

Most of the object recognition algorithms may be used for object detection by using a search window and scanning the image for the object. The task for an object detection algorithm is much harder. Its purpose is to search for a specific object in an image of a complex scene.
Regarding the computational complexity, some methods are more suitable for searching than others. Two fundamental goals of object detection algorithms are

to identify a known object in a realistic environment and also to determine its location.

There are two major methods that recognition algorithms base their calculations on: appearance based methods and feature based methods.

2.1.1 Appearance based methods

These methods use example images (called templates or exemplars) of the objects to perform recognition. Objects look different under varying conditions, such as changes in lighting or color, changes in viewing direction and changes in size and/or shape.

Some of these methods include:

- Edge matching. It uses edge detection techniques, such as the Canny edge detection [4] J.Canny, to find edges. Its main strategy is to detect edges in templates and images and then compare edges to find the template.

- Greyscale matching. Edges are robust to illumination changes but throw away a lot of information. A solution to this loss of information is to compute pixel distance as a function of both pixel position and pixel intensity.

- Gradient Matching. This is another way of saving information by comparing image gradients. Matching is performed like matching greyscale images.

- Histograms of receptive field responses. This method avoids explicit point correspondences, and the relations between different image points are implicitly coded in the receptive field responses. This is a very efficient method, while work based on it can be found in [5] by Ekvall and Kragic, [6] by Schiele and Crowley and in [7] by Linde and Lindeberg.

2.1.2 Feature based methods

A search is used to find feasible matches between object features and image features. The primary constraint is that a single position of the object must account for all of the feasible matches. The methods that extract features from the objects to be recognized and the images to be searched are based on surface patches, corners and linear edges.

Some of these methods are:

- Interpretation trees. A method for searching for feasible matches is to search through a tree. Each node in the tree represents a set of matches and the root node represents an empty set. Each other node is the union of the matches in the parent node and one additional match. Wildcard is used for features with no match, while nodes are “pruned” when the set of matches is infeasible. Also, a pruned node has no children.

- Pose consistency. It is also called Alignment, since the object is being aligned to the image. The correspondences between image features and model features are not independent; there are geometric constraints. A small number of correspondences yield the object position and the others must be consistent with this. The general idea of pose consistency is that we hypothesize a match between a sufficiently large group of image features and a sufficiently large group of object features, and then we can recover the missing camera parameters from this hypothesis (and so render the rest of the object).
  The strategy followed is that the hypothesis is generated using a small number of correspondences (e.g. triples of points for 3D recognition) and projects other model features into the image (backproject) and verifies additional correspondences.

- Pose clustering. The general idea is that each object leads to many correct sets of correspondences, each of which has (roughly) the same pose. Then an accumulator array that represents pose space for each object is used for pose voting. The main strategy is that for each object an accumulator array is set up that represents pose space – each element in the accumulator array corresponds to a "bucket" in pose space. Then each image frame group is taken, and hypothesizes a correspondence between it and every frame group on every object. For each of these correspondences, pose parameters are determined and make an entry in the accumulator array for the current object at the pose value.
  If there are large numbers of votes in any object's accumulator array, this can be interpreted as evidence for the presence of that object at that pose. The evidence can be checked using a verification method. It is noted that this method uses sets of correspondences, rather than individual correspondences.
  This implementation is easier, since each set yields a small number of possible object poses.

2.2 Video Tracking

Video tracking [8] is the process of locating a moving object (or multiple objects) over time using a camera or a video stream. It has a variety of uses, some of which are: human-computer interaction, security and surveillance, video communication and compression, augmented reality, traffic control, medical imaging and video editing. Video tracking can be a time-consuming process due to the amount of data that is contained in video. Adding further to the complexity is the possible need to use object recognition techniques for tracking.

The objective of video tracking is to associate target objects in consecutive video frames. The association can be especially difficult when the objects are moving fast relative to the frame rate. Another situation that increases the complexity of the problem is when the tracked object changes orientation over time. For these situations, video tracking systems usually employ a motion model which describes how the image of the target might change for different possible motions of the object.

Examples of simple motion models are:

- When tracking planar objects, the motion model is a 2D transformation (affine transformation or homography) of an image of the object (e.g. the initial frame).
- When the target is a rigid 3D object, the motion model defines its aspect depending on its 3D position and orientation.
- For video compression, key frames are divided into macroblocks. The motion model is a disruption of a key frame, where each macroblock is translated by a motion vector given by the motion parameters.
- The image of deformable objects can be covered with a mesh; the motion of the object is defined by the position of the nodes of the mesh.

To perform video tracking, an algorithm analyzes sequential video frames and outputs the movement of targets between the frames. There are a variety of algorithms, each having strengths and weaknesses. Considering the intended use is important when choosing which algorithm to use. There are two major components of a visual tracking system: target representation and localization, as well as filtering and data association.

Chapter 3

Related Work

In [9] Park et al. present their work as a real time image processing technique using modern programmable Graphic Processing Units (GPU). By utilizing NVIDIA's GPU programming framework as a computational resource, they realized significant acceleration in image processing algorithm computations. They showed that a range of computer vision algorithms map readily to CUDA with significant performance gains. Specifically, they demonstrated the efficiency of their approach by a parallelization and optimization of Canny's edge detection algorithm [4], and applying it to a computation and data-intensive video motion tracking algorithm known as "Vector Coherence Mapping" (VCM). Their results show the promise of using such common low-cost processors for intensive computer vision tasks. They managed to achieve 3.15 times speed enhancement with a 8600MGT card, and with a 8800GTS-512 showed 22.96 times performance enhancement.

The authors in [10] and [11] deal with the problem of the Canny edge detection, which is a basic step in image processing and one of the first steps performed by many computer vision algorithms, in order to identify sharp discontinuities in an image, such as changes in luminosity or in intensity due to changes in scene structure. The authors in [10] proposed a new self-adapt threshold Canny edge detector and also presented an FPGA implementation of their algorithm suitable for mobile robotic systems. Their hardware implementation uses the Altera Cyclone EP1C60240C8 and can perform the algorithm on a grey-scale image 360x280 in 2.5ms clocked at 27MHz. In [11] the authors present another implementation of the Canny edge detector that takes advantage of 4-pixel parallel computation, which increases the throughput of the design without increasing the need for on-chip cache memories. They showed increased throughputs for high resolution images and a computation time of 3.09ms for a 1.2Mpixel image on a Spartan-6 FPGA clocked at 200MHz.

Nikitakis, Papaioanou and Papaefstathiou [12] introduced a novel approach of an object recognition system implementing in Hardware FPGA the robust algorithm RFCH. Their main focus was to increase its performance so as to be able to handle the object recognition task of today's highly sophisticated embedded multimedia systems while keeping its energy consumption at low levels. Their low-power embedded system is at least 15 times faster than the software version on a low-voltage high-end CPU, while consuming at least 60 times less energy.

Changjian Gao et al. [13] presented a novel approach to use FPGAs to accelerate the Haar-classifier based face detection algorithm. With highly pipelined microarchitecture and utilizing abundant parallel arithmetic units in the FPGA, they have achieved real-time performance of face detection, having very high detection rate and low false positives. Their approach is flexible toward the resources available on the FPGA chip. This work also provides an understanding toward using FPGA for implementing non-systolic based vision algorithm acceleration. Our implementation is realized on a HiTech Global PCIe card that contains a Xilinx XC5VLX110T FPGA chip. They have achieved a x72 speedup for the Haar function and an overall application speedup of x20.

The authors in [14] present an FPGA-based system for detecting people from video. The system is designed to use JPEG-compressed frames from a network camera. Unlike previous approaches that use techniques such as background subtraction and motion detection, they use a machine-learning-based approach to train an accurate detector. They address the hardware design challenges involved in implementing such a detector, along with JPEG decompression, on an FPGA. They also present an algorithm that efficiently combines JPEG decompression with the detection process. The system is demonstrated on an automated video surveillance application and the performance of both hardware and software implementations are analyzed. The results show that the system can detect people accurately at a rate of about 2.5 frames per second on a Virtex-II 2V1000 using a MicroBlaze processor running at 75 MHz, communicating with dedicated hardware over FSL links.

Chapter 4

The Receptive Field Co-occurrence Histogram (RFCH) algorithm and the Tracking-Learning-Detecting (TLD) algorithm

In this chapter we will look into the basic principles behind the RFCH and TLD algorithms which make these particular applications so robust.

4.1 RFCH

4.1.1 Introduction

The authors of [5] came up with the present algorithm with one important difference from any other related work. The histograms that are generated are statistical representations of a handful of descriptor responses and the combinations between them, not just Laplacian responses or just color intensity as in [15].

A Receptive Field Cooccurrence Histogram (RFCH) is able to capture more than other methods of the geometric properties of an object. Instead of just counting the descriptor responses for each pixel, the histogram is built from pairs of descriptor responses.
The pixel pairs are constrained based on their relative distance. This way, only pixel pairs separated by less than a maximum distance, *dmax,* are considered. Hence the histogram represents not only how common a certain descriptor response is in the image, but also how common it is that certain combinations of descriptor responses occur close to each other.

This leads to the conclusion that this application can preserve more geometrical information since the representation of images through the receptive field cooccurence histograms are more consistent to filter and color responses of

pixels (array points) that lie relatively close to each other.

4.1.2 RFCH functionality

From now on, any reference to the word «object» is a type of image as shown in Figure 4.1 and by the word «scene» we mean an image as illustrated in Figure 4.2. The objects are the images that are trained and are detected in the scenes. The algorithm returns the coordinates of the object in the scene and a statistical representation of the match between the object and the object in the scene.

Figure 4.1: The Image of a training object. The black background helps the application on focusing only on the object itself.

Figure 4.2: The scene in which the application searches for the object that has been trained, in our case the yellow and blue truck.

i) Image Descriptors

One of the benefits of this algorithm is that it takes advantage of many image descriptors that an image can respond to. Some of these are the Laplacian, Gabor, Color and Gradient Magnitude. In [5] it is shown that using a mixture of the image descriptors is the optimal choice for object recognition and object detection tasks.

ii) Image Quantization

The histograms used in object recognition algorithms are eligible to become huge, thus increasing the computational complexity of their further analysis and the general performance of the application. This depends on the multidimensional nature of the receptive field histograms, and by using cooccurence histograms, their size increases exponentially. For example, using 15 bins in a 6-dimensional histogram 15^6 bins (~ 10^7) will be generated. And by using cooccurence histograms, the bins generated are close to 10^{14}.

To cope with this problem of exponential progression of the size of the histogram, the input data (image features) must be clustered so the dimensions of the histogram can be reduced. By choosing the number of clusters, we can control the histogram size to our choice. In [5] the authors have chosen 80 clusters and that

resulted in constructing dense histograms, while the different bins generated have high counts. The N cluster centers have a dimensionality equal to the number of image descriptors used. The algorithm which reduces the dimension uses the K-Means clustering algorithm [16] and after the quantization, each object ends up with its own cluster scheme and the RFCH is calculated based on the quantized training image.

In the detecting phase, when the program searches for an already trained object in a scene, the image of the scene is quantized with the same cluster centers as the cluster scheme of the object being searched for. Quantizing the search image also has a positive effect on object detection performance. Pixels lying too far from any cluster in the descriptor space are classified as the background and not incorporated in the histogram. This is because each cluster center has a radius that depends on the average distance to that cluster center. In this point a free parameter is used (α), which denotes the size of the cluster-centers.

Figure 4.3: Clustering an Image (left) and the pixels that have survived the procedure (right)

The above processing step in Figure 4.3 shows exactly what is mentioned above while searching for the Santa-cup in the whole scene. The image has been quantized with the cluster scheme of the Santa-cup and the pixels that lie too far away from their nearest cluster are ignored and have been set as a black background [3].

iii) Histogram Matching

After the histogram construction of both object and scene, the algorithm tries to find out in which level those two histograms match. This is done with a histogram intersection with the below formula,

$$\mu(h1,h2) = \sum_{n=1}^{N^2} min(h1[n], h2[n])$$

where hi[n] denotes the frequency of receptive field combinations in bin n for image i, quantized into N cluster centers. The higher the value of μ(h1,h2), the better the match between the histograms. Before matching, the histograms are normalized with the total number of pixel pairs.

Another method for testing histogram similarity is χ^2;

$$\mu(h1,h2) = \sum_{n=1}^{N^2} \frac{(h1[n]-h2[n])^2}{h1[n]+h2[n]}$$

in this case the lower the value of μ(h1,h2), the better the match between object and scene.
In [5] it is mentioned that using the latter way of histogram intersection drops the performance in object detection tasks, while it boosts performance in object recognition image databases.

iv) Detecting an object

After quantizing the image scene with the cluster scheme of the object that is to be detected, the image is scanned using a small search window. The window is shifted such that consecutive windows overlap to 50 % and the RFCH of the window is compared with the object's RFCH. This is the first step of the detecting phase, while the second step uses only one search window with a size of the whole scene and then returns the best match. The first step shows the coordinates of the object in the image, while the second step returns the best match.

The matching vote $\mu(h_{object},h_{window})$ indicates the likelihood that the window contains the object. Once the entire image has been searched through, a vote matrix provides a hypothesis of the object's location. In Figure 4.4 it is shown how the vote matrix reveals a strong response of the yellow soda can's real position. It also has a slight response of the far left yellow raisin box because of the same color responses of the two objects.

Figure 4.4: Example of searching for the yellow soda can and its relative response

v) Free parameters

The free parameters are in the discretion of the programmer to select the right values for their use of the application and by experimenting on them getting the best performance. These are:

- Number of cluster-centers, N. Too few cluster-centers reduce the detection rate. We experimented with 80 cluster centers and 7 image descriptors.

- Maximum pixel distance, d_{max}. Best performance is noticed when using a range of 1 to 10. We experimented with different pixel distances but settled to $d_{max}=10$.

- Size of cluster-centers, α. Pixels that lie outside of the cluster centers are classified as background and not taken into account. We have selected $\alpha=2.0$ and ruled that it is optimal.

- Search window size. This is the serious drawback of the object detection task because the optimal search window is unknown. We have used 165 search windows, which are boxes of 6400 pixels each.

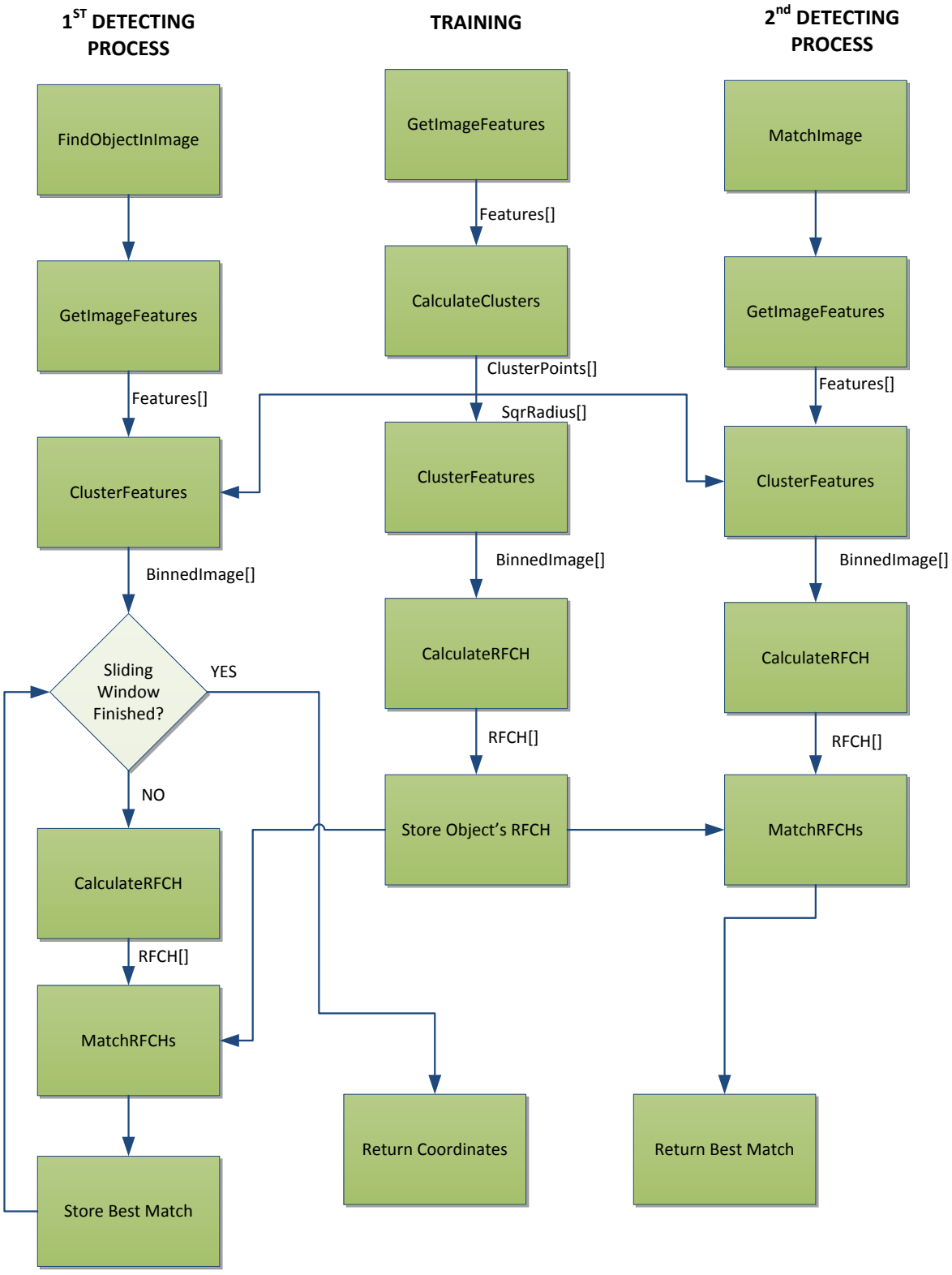


Figure 4.5: Dataflow steps of the Software version of RFCH until the completion of the detection

4.2 The TLD (Tracking-Learning-Detection) Algorithm

4.2.1 Introduction

The TLD algorithm was introduced by Zdenek Kalal et al. [17] which performs accurate tracking in real time. As a software implementation reference, we used the BPTLD [18] which has been the first published implementation of this scheme in C++.
The present application is a video tracking algorithm and utilizes a video stream, either from an existing video or a web camera. On the video stream, the user draws a bounding box around the object that is to be detected and tracked, with the help of the mouse interface. The projection of the video stream and the interface events are assisted by the OpenCV library[1] [19].

In [17] the authors describe the present application as a long-term online tracking method with minimum prior information. Long-term refers to sequences of possibly infinite length that contain frame-cuts, fast camera movements and the possibility the object may temporarily disappear from the scene or cause limited shape changes. Online means that the tracking does not exploit information from the future and processes the footage in one pass. Minimum prior information indicates that the object is not known in advance and the only information about it comes from the first frame, where it was selected by the user.

4.2.2 TLD functionality

i) Features

The object detector of the TLD is based on features called 2bit Binary Patterns (2bitBP) and they measure gradient orientation within a certain area, quantize it and output 2^2 different processing codes. The certain area that is quantized is a pixel area of the frame and is called a patch. Each Image patch is described by a number of local 2bitBP which position, scale and aspect ratio are generated at

[1] OpenCV (*Open Source Computer Vision Library*) is a library of programming functions mainly aimed at real-time computer vision, developed by Intel, and now supported by Willow Garage and Itseez.

random and these features are arbitrarily partitioned into several groups of the same size.
Each group represents a different view of the patch appearance and the response of each group is represented by a discrete vector that is called a branch.
The branch belongs to a randomized forest which is utilized as a classifier for the detection task.

ii) Randomized Forests

The randomized forest approach, which yields high classification accuracy, is based on an ensemble of trees which can vote for the most popular object. In order to create these ensembles, the above-mentioned random vectors govern the growth of each tree in the ensemble. The random forests consist of a combination of tree predictors such that each tree depends on the values of a random vector sampled independently while the same distribution is applied for all the trees in the forest.

A random forest classifier utilizes several decision trees and each decision tree is passed through on a number of features. The leaf-nodes of each tree specify the probability of an object match and the probabilities from all trees are averaged to yield a final probability. All decision trees sample the same Integral image, which is a digital representation of an actual image. The Integral Image is often seen in computer vision applications as Viola and Jones in [20] showed that by applying a large number of simple rectangular filters in an image, a robust description of the Image (Integral Image) is built based on the features that describe it. These rectangular features provide a very good representation of the Image and cope very well with training tasks.

Chapter 5

Assessing the Applications

5.1 Introduction

In [21] an optimal procedure is introduced for porting programs to an NVidia GPU via CUDA, which we tried to follow as much as possible. This is done by the steps of assessing, parallelizing and optimizing the CUDA code written.
For an existing project, in our case RFCH and BPTLD, the first step is to assess the application and to locate the parts of the code that are responsible for the bulk of the execution time. Armed with this knowledge, we could evaluate these bottlenecks for parallelization and start to investigate GPU acceleration.
Many codes accomplish a significant portion of the work with a relatively small amount of code. Using a profiler, the developer can identify such hotspots and start to compile a list of candidates for parallelization. We used the Intel VTune Amplifier XE and a Intel® Core™2 Duo Processor @1.83GHz for a profiling analysis on both RFCH and BPTLD.

5.2 RFCH

We are going to show the most important functions of the RFCH application and then the hotspots identified.

5.2.1 RFCH main functions

GetImageFeatures(). After a real image is processed so the machine can interact with it, the image is passed to the present function and returns the array Features[] which represents the desired combination of feature descriptors.

CalculateClusters(). This function takes as input the Features[] array and calculates the cluster scheme of an object to be detected. Two arrays are produced that carry the information of the scheme, the ClusterPoint[] and

SqrRadius[].

ClusterFeatures(). Using as input the Features[] array and the two arrays calculated at CalculateClusters, a quantized version of the image that the Features[] represents is generated. This could be an object or a whole scene and the output of this function is the binnedImage[] array.

CalculateRFCH(). This function creates a normalized receptive field cooccurence histogram of either an object or scene that is quantized with a cluster scheme of an object. Its input is the binnedImage[] array and it returns the histogram RFCH[], which is an array of 6400 elements.

MatchRFCHs(). The histogram intersection is performed here and takes as input two histograms, one of the object that is to be detected and another of the histogram of the scene. A floating variable is returned that presents the best match.

FindObjectInImage().This function scans a scene with a sliding window, calculating the RFCH of the window and then compares it with the object's RFCH. When the whole image is scanned, it returns the best match along with the coordinates of the object that had the best match.

5.2.2 Identifying Hotspots

With the help of the Intel VTune Amplifier XE, three functions were identified as the major hotspots of the algorithm, taking over the execution time of the application as much as 97.8%. These are the CalculateClusters(), ClusterFeatures() and CalculateRFCH(). They are selected to be implemented on hardware NVidia GPUs via CUDA. All other functions will be running on software, such are the GetImageFeatures() and MatchRFCHs().

The next step is to figure out the computational complexity of each function and the reasons the algorithms behind each function are drawbacks for better performance in the software version. These reasons will be the base of our parallelization strategy.

- CalculateClusters:
  The computational complexity of this function is O(nfNT), where n is the number of samples, f the number of features, N the number of clusters and T the number of iterations until all clusters have been fully updated.
  CalculateClusters works in an iterative manner of the K-Means algorithm in a while-loop. Inside the while-loop there is a for-loop that generates 307,200 iterations (the total number of pixels), while other smaller nested loops inside the big for-loop exist and execute code. In addition to the above, the algorithm contains many if-conditions making it even more heavyweight. After the 307,200 iterations an update occurs to the ClusterPoint[] array and the while-loop performs one more iteration. Based on our experiments the average iterations of the while-loop are 70.
  All of the above makes this training function the most time consuming in the RFCH application.

- ClusterFeatures:
  The computational complexity of this function is O(nfN), where n is the number of samples, f the number of features and N the number of clusters.
  This algorithm quantizes the image and starts with a for-loop of 640 iterations that is nested inside a for-loop of 480 iterations. These two loops generate 307,200 iterations while other nested for-loops of size 80 and 7 can produce 172,032,000 iterations of specific calculations, if-conditions and memory accessing.

- CalculateRFCH:
  The complexity of this function is $O(nd^2)$, where n is the image input size and d is the maximum distance set between pixels.
  There are two different inputs for this function depending on the phase of the algorithm. One uses all of the pixels of an image or object, meaning 307.200 iterations for starters, and the second one takes as input the sliding window which is set to 6400 pixels, leaving us with 6400 iterations.

In addition to the iterations, CalculateRFCH performs a while-loop that can reach up to 221 iterations with many if-conditions.

Taking the above into account, our goal for the CUDA ports to a GPU is mainly to parallelize the 307,200 iterations generated by the algorithms, with each thread performing the work of each iteration. Except for the CalculateRFCH in the first phase of the detection process, where we have to parallelize 6400 iterations.

5.3 TLD

We will show the results of the profiler applied on the TLD and the reasons of the execution bulk.

5.3.1 Identifying Hotspots

Running the BPTLD algorithm on the Intel VTune Amplifier XE we could evaluate the hotspots and be directed on how the TLD could be boosted performance-wise.
It showed that a particular function, the sumRect(), occupies more than 50% of the execution time. SumRect() is part of the detection process of the application, while the whole detecting task occupies 90% of the overall execution.
SumRect() itself is a very simple function; it is basically an Integral Image computation. Four different integers access four different values of the Integral Image array and then the integers are added up. The result of this operation is returned to the detection task.
While sumRect() has no large complexity, the fact that is called by the detector many times makes this function the application's major hotspot. For each frame it is called about 1,863,680 times by the Detect() method of the C++ Detector class of the BPTLD. SumRect() computes values and is called four times in the heart of five nested for-loops, the first one generating 5 loops, the second and the third could generate 30 to 32 iterations each depending on the width and height of the Integral Image. The fourth loop creates 13 iterations

and the last one generates 7 iterations. 13 and 7 are the number of total ferns and total nodes of the classification tree respectively.
Having identified the problem and understanding the above, we have selected to parallelize the Detect() method with an immediate minimization of the utilization of sumRect() by the CPU. Our strategy is based on parallelizing the two larger loops, which together can create a maximum of 32x32 = 1024 iterations. Following the RFCH assessing methodology our goal is to create 1024 threads and modify any needed changes in the original C++ code so each thread replaces an iteration, and to parallelize the detecting task to an efficient level of performance acceleration.

Chapter 6

Parallel Implementation of the two Algorithms

6.1 Introduction to System Architecture

For the better comprehension of this part of the thesis, we will try to explain some technical terms and to introduce the reader to basic CUDA characteristics. When we refer to the word «module», we mean a function of our application which contains both CPU (Host) and GPU utilities.
Initializing arrays and a general use of the CPU thread was needed and we tried to occupy it as little as possible.

As for the GPU, utilities could be CUDA APIs that perform instructions on GPU and Host memory; such are memory allocations and deallocations, memory copies and initializing arrays that reside on the GPU.
Our modules are responsible for the preparation and the call of a kernel. A kernel is considered a part of the module and it is a function executed by the GPU. The module guides the kernel by setting its grid dimension and the right of accessing specific variables stored in the Global Memory of the GPU.
The grid dimension are the boundaries of what a kernel can use in terms of resources of the GPU, as are the number of blocks and the number of threads containing each block.
It is called by the Host as:
kernel<<<threads_per_block, blocks_per_grid>>>(arguments).

The threads_per_block and blocks_per_grid are of type dim3. This type is an integer vector type that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1.
Each distinct thread and block has an ID number and they can range up to three dimensions (X, Y, Z). We experimented with both one-dimension and two-dimension blocks and threads.

The CUDA API provides built-in variables only to be accessed by the kernel that store the index of the blocks and threads and also the block and grid dimensions. For the threads, these variables are the threadIdx.x, the threadIdx.y and the threadIdx.z, depending on the dimensionality of the threads. If the threads are one-dimensional, threadIdx.y and threadIdx.z are set to 1. If the threads are two-dimensional, only threadIdx.z is set to 1.
These variables show the position of the threads in a thread block.
The threadIdx.z cannot exceed 64, while the first two dimensions have a maximum of 512 threads per thread block or 1024 per thread block depending on the compute capability of the GPU. 512 threads is the maximum for cards with compute capability of 1.3 and 1024 threads is the maximum for compute capability 2.x GPUs.
The blocks have indexing IDs that range up to 65535 for the first two dimensions of compute capability 1.x, while all dimensions of cards with compute capability 2.x enjoy IDs that range up to 65535. The variables that characterize the IDs are the blockIdx.x, the blockIdx.y and the blockIdx.z. Each block dimension contains the threads that have IDs referenced to the same dimension.
The blockdim.x variable has a constant value, which is the number of threads that have been set to that dimension by the CPU. The same is enforced to the rest of the two dimensions.
The grid is defined by its own variables, the gridDim.x and the gridDim.y. For cards of compute capability 2.x one more dimension is added with the variable gridDim.z.

As we have mentioned, threadIdx is the index number of a thread in a certain block. This is very useful for the kernel code as the developer can handle threads and guide them. But most of the times the kernel needs the global ID of a thread, meaning the ID number that is associated with the thread in the whole grid.
The global index of a one-dimensional thread can be computed as
tidx = blockIdx.x * blockDim.x + threadIdx.x, marking it with its own ID in the block and in which block of the grid it exists.
In the same way the thread index of a two-dimensional thread will be assigned as:
tidx = blockIdx.x * blockDim.x + threadIdx.x
tidy = blockIdx.y * blockDim.y + threadIdx.y

A very important fact of CUDA computing is the organization of threads in groups of 32 called warps. This division is done by the multiprocessors. Calculations and accessing memory are the main utilities of the threads. These calculations and memory accesses are all done by the warps in clock cycles. The warp scheduler unleashes a warp for an execution and when a warp has finished its work, the warp scheduler will let the next batch of 32 threads perform its work. Needed clock cycles differ for each instruction; two clock cycles are needed to issue Global Memory instruction.

This is the true nature of the GPU, that not all threads run in parallel. What is really running in parallel are the multiprocessors with their cores and a few threads every time in clock cycles.

It is always a better practice to set the total number of threads in the grid to a multiple of 32. In this way a multiprocessor will use as much of its resources it can and keeping it busy by doing maximum work in each clock cycle. Most importantly, warp divergence will be avoided, a phenomenon we will analyze in the next paragraph. We have managed to keep all of our threads in the RFCH GPU implementation to a multiple of 32 and noticed some performance increase. For the BPTLD GPU implementation a multiple of 32 could not always be selected, since the grid dimension changes due to other parameters.

Although scheduling threads in groups has its advantages, when if-else statements or similar conditions are needed for the kernel execution, it is very likely that warps will be divided and branched. Then, threads in a warp will take different directions of execution, which is called warp divergence. This is an inevitable loss in performance.
For example:

```
if(tidx<10)
  do A
else
  do B
```

The above code divides the warp into two different paths, and the code below

```
if(tidx<32)
  do A
else
  do B
```

will allow all the threads in the first warp of the kernel to follow its own path, while the next warps will take different paths but they won't be diverged. The above will definitely perform better.

6.2 RFCH Parallel Implementation

6.2.1 CalculateClusters Module

This module is applied only in the training process of the RFCH algorithm and calculates the needed cluster points and square radius characteristics of the object that is about to be detected.
Both cluster points and square radius characteristics derive from the Feature[] array, stored in the Global Memory via a function made just for that reason, the cuda_storeFeature.
It transforms this two-dimensional array into one-dimension and then by using the CUDA API *cudaMemcpyToSymbol* it's copied to the GPU as shown in Figure 6.1.
This function will copy a number of elements of an array that resides in RAM to an existing array that resides in Global Memory or Constant Memory. These GPU arrays are defined as:
__device__ (__constant__) data_type name_of_array[size_of_array].
In our case the array is defined as
__device__ unsigned char dev_feature[2150400].
With this syntax, the nvcc compiler will identify the variable as a device variable

and allocate memory for it with the size depending on the data_type and size_of_array.
The function cuda_storeFeature and the fact that Features[] is stored in the Global Memory so it can be accessed any time by any module, is tremendously useful for our application and for later calculations.

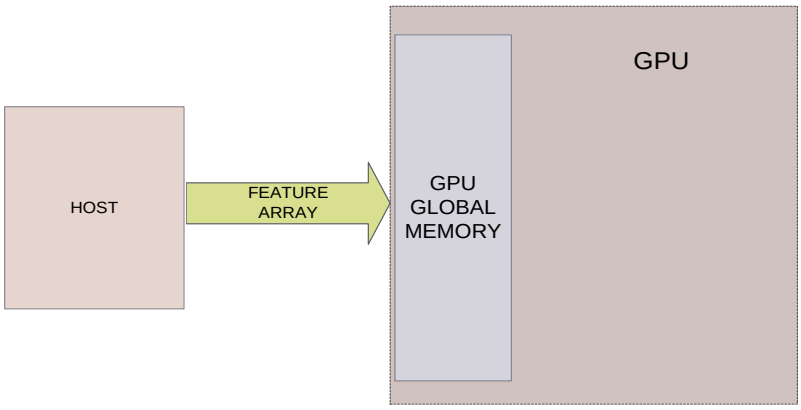


Figure 6.1: the cuda_storeFeature function

Our kernel takes as input the clusterPoint array, which is initialized by the Host, and by accessing the Feature[] array via threads, it calculates two intermediate arrays that will update the cluster points on the Host.

This is done in an iterative manner. Once the clusters have been updated on Host, the kernel will be called again and again with input the updated clusters. The two intermediate arrays will be updated in the kernel and copied back to the Host, which will update the clusters one more time. This process will be done until all clusters have been fully updated.

When the cluster points are fully updated, the squareRadius array is generated using one of the intermediate arrays.
Then both clusterPoint and squareRadius are stored in the Constant Memory of the GPU by calling the appropriate CUDA API *cudaMemcpyToSymbol*.
This time our arrays are defined as Constant Memory variables as:

__constant__ unsigned char dev_clusterPointx[560] and *__constant__ float dev_sqrRadiusx[80],* where *x* the ID of the object to be detected.

The architecture of our grid in the NVidia GTX 285 is one-dimensional with the number of blocks being 600 and the maximum threads the card can offer for each block, 512. This creates a grid of 600*512 = 307,200 active threads.
For the NVidia GTX 580 we used 300 blocks and the maximum threads per block the card offers which is 1024. The grid and the architecture of our module are shown in Figure 6.2.

The arguments passed to the kernel from the Host is the clusterPoint[] array, the two intermediate arrays, and xsize and ysize. The xsize is an integer variable with a value of 640, while ysize is an integer variable with a value of 480. They represent the X-axis and Y-axis of the image and their combination is a pixel pair.

Both grids will generate 307,200 active threads, exactly the amount of iterations we wanted to parallelize. This number represents the 640x480 dimension of our images.

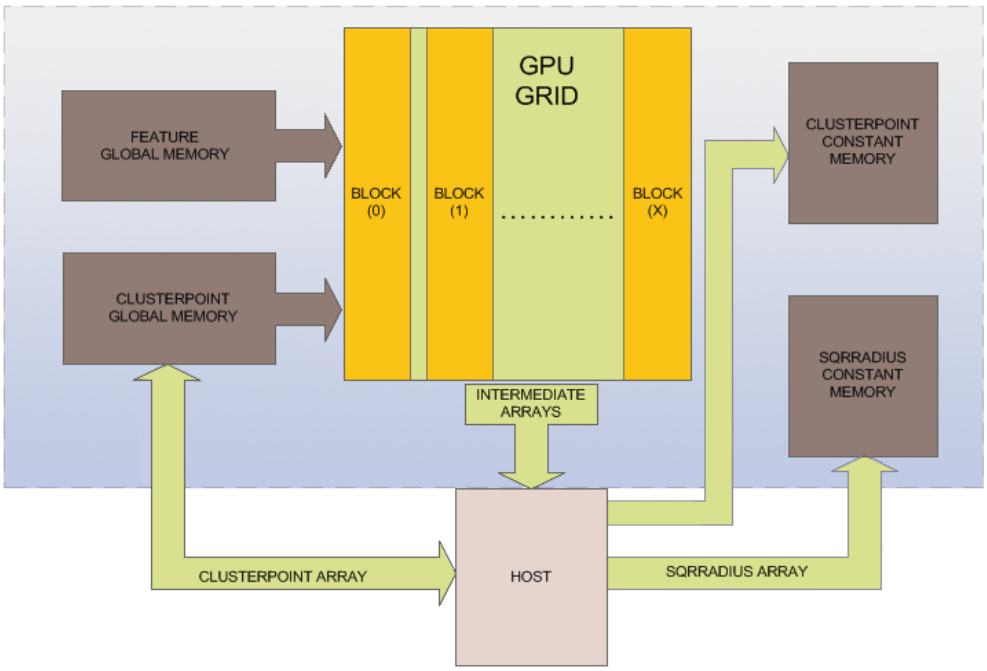


Figure 6.2: High level architecture of CalculateClusters module. X depends on the GPU we are running the module, 599 for the GTX 285 and 299 for GTX 580.

Loop unrolling was very important in this module, since the present algorithm utilizes important unparalleled loops.

An example from our code is shown below.

```
#pragma unroll 80
for (i = 0; i < 80; i++)
        {
          dist = 0;
#pragma unroll 7
           for (f = 0; f < 7; f++)
         {

            dist += (float) ((dev_feature1[f * 307200 + tidx] -
dev_clusterPoint[f * 80 + i]) * (dev_feature1[f * 307200 +
tidx] - dev_clusterPoint[f * 80 + i]));
           }
         }
```

By using the nvcc compiler optimization directive #pragma unroll N, where N is the number of iterations the for-loop will iterate, the compiler tries to parallelize the loops as much as possible, and therefore gain performance.

Below we can see the thread divergence that occurs in our kernel.

```
for (f = 0; f < 7; f++)
 {
  if (dev_feature[f * 307200 + tidx] != 0)
    {
      ok = true;

    }
 }
```

The threads (tidx) will diverge in this if-statement, thus dividing the warps that they belong in, and threads in the same warp will execute different instructions.

For this example, the values of the Feature[] array that are not equal with zero are very few compared to the whole array. Keeping in mind that this is a training procedure for an object with a black background, the 0 value represents a black colored pixel. Actually, most threads won't do any work in this kernel from this step and beyond since the object takes over a small part of the whole image. But we had to launch the kernel anyway with all the necessary threads to access the Features[] array, since we don't know in advance which indices of the array are greater than zero.

6.2.2 ClusterFeatures Module

This part of the RFCH algorithm is used both in the training and the detecting part of the application. For the training procedure, it quantizes the object we want to

match with the whole scene and in the detecting process it quantizes that scene with a respective cluster scheme.

Our goal in this CUDA module was to parallelize two nested for-loops, the first one generating 480 iterations with the second one executing 640 iterations of each iteration of the first loop.
To achieve this, we decided to take advantage of the two-dimensional blocks and threads that CUDA and the GPU offer with each dimension replacing a for-loop.

In the NVidia GTX 285 graphics card, we set the first dimension to 40 blocks and 16 threads in each block, while the second dimension has 30 blocks of 16 threads each. The indexing IDs of the blocks range from (0, 0) to (39, 29) and the 256 threads containing a block have indexing IDs ranging from (0, 0) to (15, 15).

All these values are combined to construct 2-D blocks and 2-D threads.
The number of 2-D blocks is generated by the multiplication of the number of blocks in each dimension. The above grid for example contains 40*30=1200 blocks.
The total amount of 2-D threads in a 2-D block is the product of the multiplication of the amount of threads containing each 1-D block. In our example this number is 16*16 = 256.
The total amount of threads that we can take advantage of for the whole grid is the total amount of threads in a 2-D block, multiplied by the number of the 2-D blocks, 256*1200 = 307,200, which is exactly the amount of iterations we want to parallelize.

For the NVidia GTX 580 we also used two-dimension threading and the maximum threads per block the card offers.
The first dimension has 20 blocks with 32 threads in each block producing 640 threads and the second dimension has 15 blocks with 32 threads each producing 480 threads, thus creating a computational grid of 300 active blocks with 32x32 = 1024 threads per block.
The indexing IDs of the blocks range from (0, 0) to (19, 14) and the 1024 threads containing a block have indexing IDs ranging from (0, 0) to (31, 31).

The arguments passed from the Host to the kernel are the fid, xsize and ysize. The fid is an integer, which represents the ID of the object we are processing at the time and the kernel will understand which cluster scheme it will use for its computations.

The ClusterFeatures module produces the array binnedImage[], which has a size of 307,200 integers and represents a quantized version of the 640 pixels x 480 pixels of each image or object.
The production of the array is done by using the Feature[] array of the object or scene given- depending on which phase of the application ClusterFeatures is in - and the ClusterPoint and SquareRadius arrays of an object that have been created by the CalculateClusters module.

The Feature[] array resides in the non-cached off-chip Global Memory of the GPU after we transferred it to the Device as shown in Figure 6.1. If the application is in the training phase, cuda_storeFeature has already been called and the Feature[] array of an object is stored.
In the detecting phase it is called right before ClusterFeatures and stores the Feature[] array of the image.
The ClusterPoint and SquareRadius arrays reside in the faster cached off-chip Constant Memory.
The reason we decided to store the ClusterPoint and SquareRadius arrays in the Constant Memory is that both meet the restrictions and requirements of that memory.
The crucial factor of the Constant Memory is its limited size, only 64 KB.
Also, variables and arrays residing in Constant Memory have read-only permissions.
The size of ClusterPoint is 560 bytes and SquareRadius has a size of 80 float variables, which means that in memory are stored 80 x 8 = 640 bytes.
Concluding, a total of 1200 bytes are needed for each object and we can store in the Constant Memory 64000 bytes/1200 bytes = 53 objects.
This is a theoretical result because in practice Constant Memory is used by CUDA to store arguments for the kernel.
The latter arrays are needed only to extract the information they carry and not to store new ones, meeting the restriction of read-only permission.
The same restriction is also met by the Feature[] array but its size is 2.15 MB,

which exceeds the 64 KB boundary limit of the Constant Memory.

The binnedImage[] array is later needed by the function CalculateRFCH, and since we have ported to CUDA only the versions of CalculateRFCH that execute the detecting phases, the binnedImage[] must be copied to the Host memory when the application is in training mode.
Figure 6.3 shows exactly that and how the needed arrays interact with the computational grid.
In Figure 6.4 is shown how the binnedImage[] just resides in the Global Memory of the Graphics card after calculations, waiting for the CUDA version of CalculateRFCH to access it when the application is in detecting mode.
This means that the module in the latter case is basically only a kernel call, since we have eliminated any other GPU API and CPU execution.

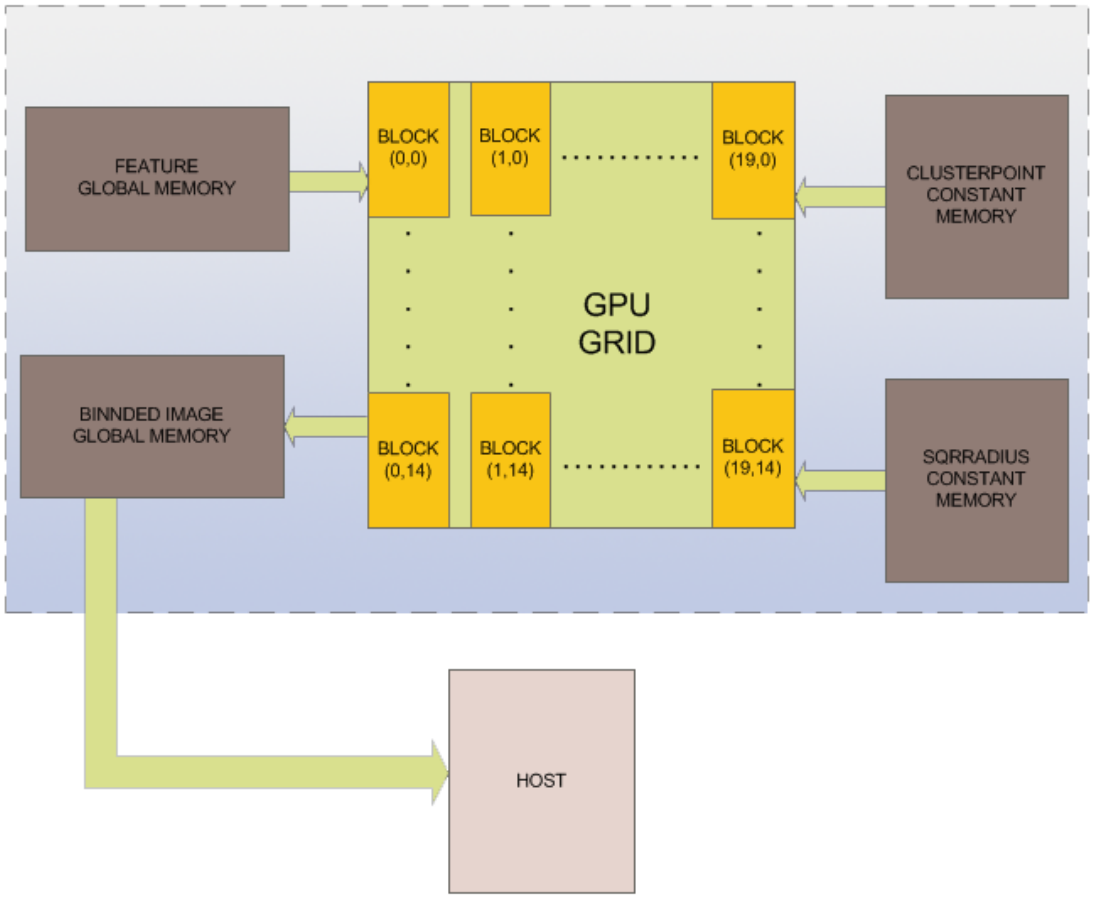


Figure 6.3: ClusterFeatures training phase GPU implementation

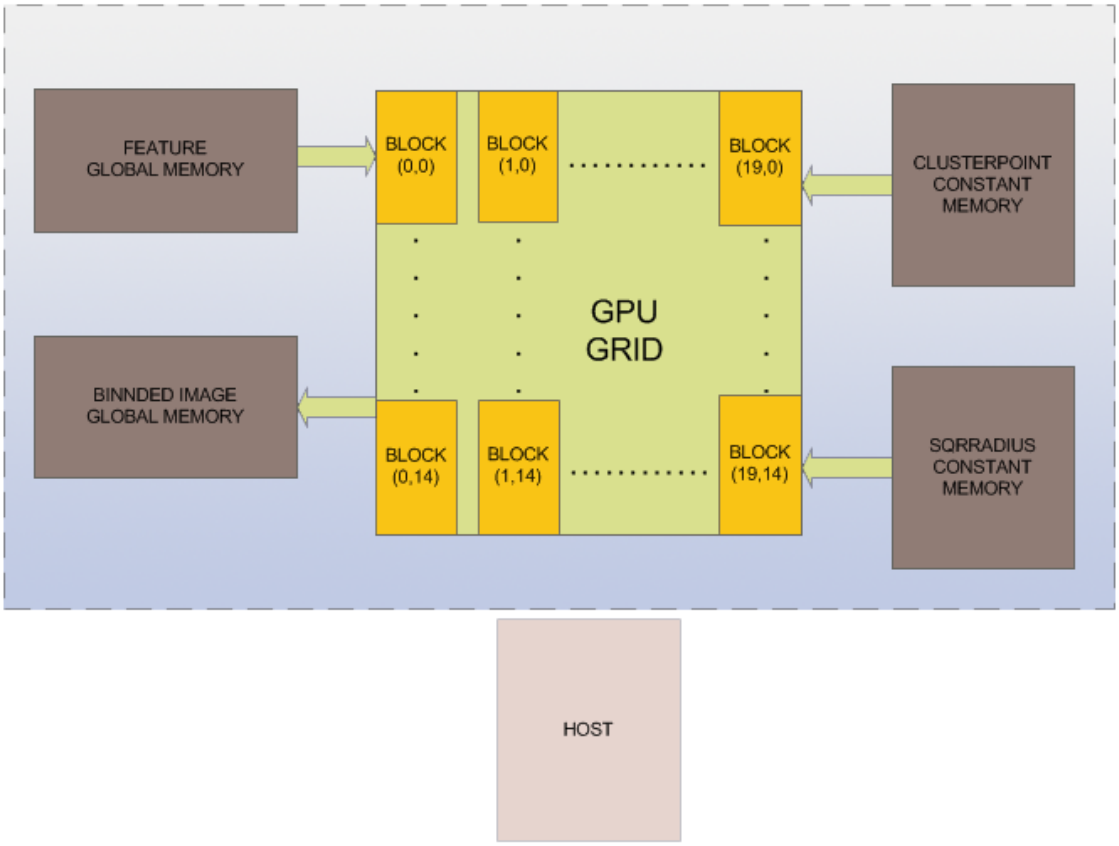


Figure 6.4: ClusterFeatures detecting phase GPU implementation

In this kernel we also used the directive #pragma unroll for loops that had to be optimized.
For example in this kernel code segment :

```
#pragma unroll 80
for (i = 0; i < 80; i++)
        {
          dist = 0;
#pragma unroll 7
          for (f = 0; f < 7; f++)
         {

            dist += (float) ((dev_feature1[f * 307200 + (xsize *
tidy + tidx)]  - dev_clusterPoint[f * 80 + i]) * (dev_feature1[f
* 307200 +  (xsize * tidy + tidx)] - dev_clusterPoint[f * 80 +
i]));
```

```
        }
      }
```

This piece of code is the same with the CalculateClusters mentioned in the previous model, with the difference that the Feature[] array is accessed by two-dimensional threads, characterized by tidx and tidy for each thread.

Also, warp divergence is noticed in the kernel of ClusterFeatures via the below code:

```
for (f = 0; f < 7; f++)
 {
  if (dev_feature[f * 307200 + (xsize * tidy + tidx)] != 0)
    {
      ok = true;

    }
 }
```

As we can see, the two-dimensional threads that access the Features[] array are branched, which leads to warp divergence.

For the training phase of the ClusterFeatures most values of Features[] are equal to zero in the same way as in CalculateClusters. But all threads that access the Features[] array are needed since we don't know its values in advance.
For the detecting phase, the values of the Features[] array are mostly different than zero, because it is an interpretation of a scene image. The values that are different from zero exceed 83% of the total values in the examples that we used. In this case, most of the threads perform maximum work keeping the multiprocessors fully occupied.

6.2.3 CalculateRFCH Module

CalculateRFCH calculates the Receptive Field's Co-occurrence Histogram and uses as input the binnedImage[] array which resides in the Global Memory of the GPU.

This part of the application was the least time-consuming compared to the previous functions, but the nature of its algorithm is challenging for a CUDA port, due to its many divergent branches and the calculation of a histogram.

The CalculateRFCH is used in both detecting phases, slightly altered, with the main algorithm being the same.
In the first phase it analyzes small windows of the quantized image (binnedImage), different every time, repeatedly until the RFCH application can find the object in the image.
In the second phase it analyzes the whole binnedImage[], creates the co-occurrence histogram of the image and compares it with the histogram of an object via histogram intersection to get the best match.

In the first phase of the detection our objective was to parallelize two nested for-loops, each one generating 80 iterations, a total of 6400 iterations, and in the second phase the two nested for-loops generate 640x480 = 307,200 iterations which had to be parallelized.
For that reason we used two-dimension threading but the selection of the computational grid for each card differs this time because of the crucial factors of the compute capability and processing schedule of each card, in addition to the nature of our kernel.

In the NVidia GTX 580, each dimension has been selected to have 20 blocks with 4 threads per block. Overall, our grid consists of 400 active blocks with 16 threads per block, producing 6400 active threads, each thread representing an iteration of the combination of the two for-loops.
Likewise, for the Matching Image algorithm, we used two-dimension blocks and threads but with a different grid so as to parallelize 307,200 iterations. We used 16 threads per block, 4 threads in each block of a dimension, the first dimension consisting of 160 blocks and the second dimension 120 blocks, a total of 19200 blocks.

For the NVidia GTX 285 and for the first phase of the detection, we set our grid to 25 blocks with 256 threads each, each dimension containing 5 blocks of 16 threads.
For the second phase of the detection and for the same graphics card, we also

used 256 threads per block but since the two nested loops that ought to be parallelized generated 307,200 iterations, we used 307200/256 = 1200 blocks.

The kernel's arguments are the xsize, ysize and the limits of the sliding window. These limits change every time the CalculateRFCH is called in the first detection stage, while in the second phase these limits will produce one sliding window with a size of 307,200.

The final result of CalculateRFCH is a normalized histogram and it derives from an original pre-histogram and the variable pixels, which represents the total number of bins calculated. Pixels is calculated by using an array that resides in the shared memory of each block in a way that each thread of a block (whose ID is an index of the array) counts how many times it has generated a bin for the pre-histogram. Using a parallel reduction technique for all the arrays in each shared memory of a block, we can count how many bins have been generated for the pre-histogram, and this is the total number of pixel pairs.

Parallel reduction is a method for summing values in an array. Starting with the whole array and using half of the threads that can access it, each thread sums two values, one value that the thread can access respectively to its ID and array index, and one value which has the array index of the thread ID plus half of the size of the array as shown in Figure 6.5.
By dividing by 2 the quantity of threads used and the size of the array repetitively until only one thread is used (with ID 0) and two elements of the array, we have stored the summation result in the index 0 of the array.
Every step above is utilized almost simultaneously in each shared memory of a block and since shared memory accesses are very fast because they are on-chip memories, we get very good results performance-wise.

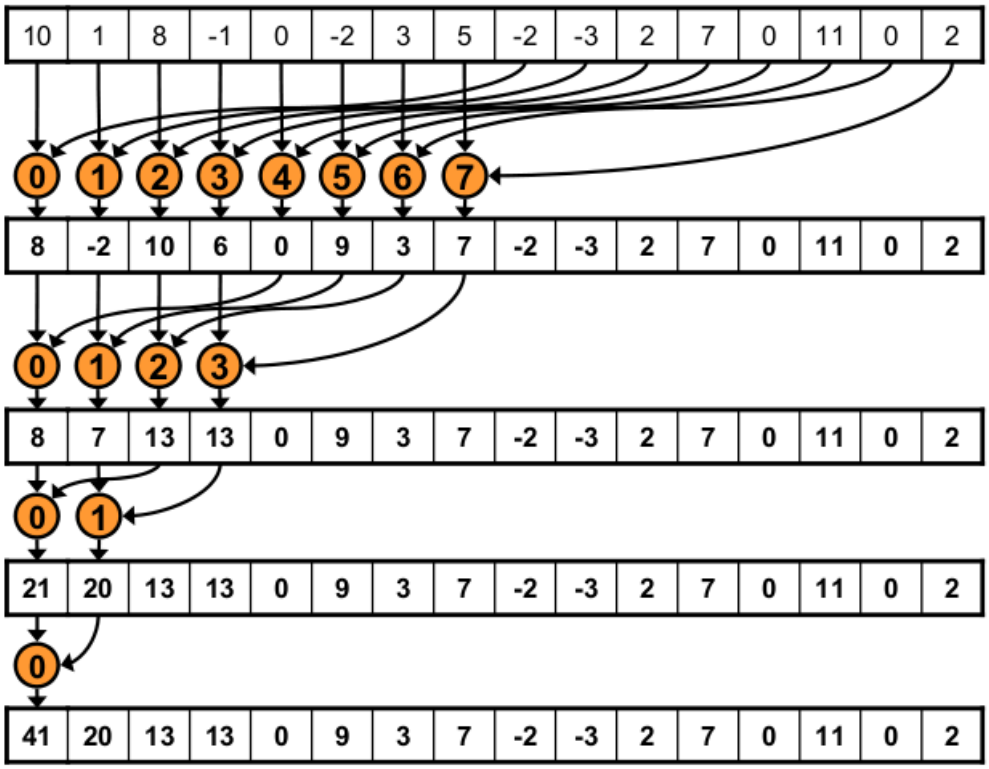


Figure 6.5: A parallel reduction example

Except for the variable pixels, our kernel stores in a large array all the bins that have been generated, and this will be the main source for the calculation of the pre-histogram.

Residing the array in the Global Memory of the GPU, a sample CUDA code is called, which is found in the NVidia forum [22] and it is a fully optimized Histogram Generator. By running this CUDA code on the GPU, we could extract the pre-histogram.

Histogramming procedure in computer science is the method of constructing an array depending on an input array. This is done by counting how many times a certain value has appeared in the latter array. This value will be the index of the new array and its respective value will be the times it has appeared.
Because this is a difficult procedure to parallelize and also since it is used in many applications, we used the above sample code which can vary to the size of the histogram (number of bins) and the size of the original array.
In our case and in the second phase of the detection, the input array calculated

from our kernel had a size of approximately 68,000,000 integers with values ranging from 0 to 6399.
Concluding our results, dividing each value of the histogram with the variable pixels, we finally compute the normalized histogram RFCH.

We could not avoid warp divergence in this module, too. The code below proves it.

```
if (dev_binIm[tidx + tidy * imxsize] >= 0)
  {
       ...
  }
```

This code line is executed in the beginning of the kernel and allows the threads that access the quantized image and extract a non-negative value, to continue with the instructions that lead to the kernel's results.
This is an obvious warp divergence.

For the first detection phase that utilizes the sliding window, 6400 values of the array are checked every time, while in the second phase 307,200 values are checked. In the software version both training phase and the second detecting phase access the whole binnedImage[] of 307,200 values. In the training phase the if-statement allows a small number of iterations to proceed, which makes it relatively fast. But the number of iterations that will proceed are unknown, so a CUDA port of this phase would have to be implemented with 307,200 threads. In this case CPU time is faster than the GPU implementation.

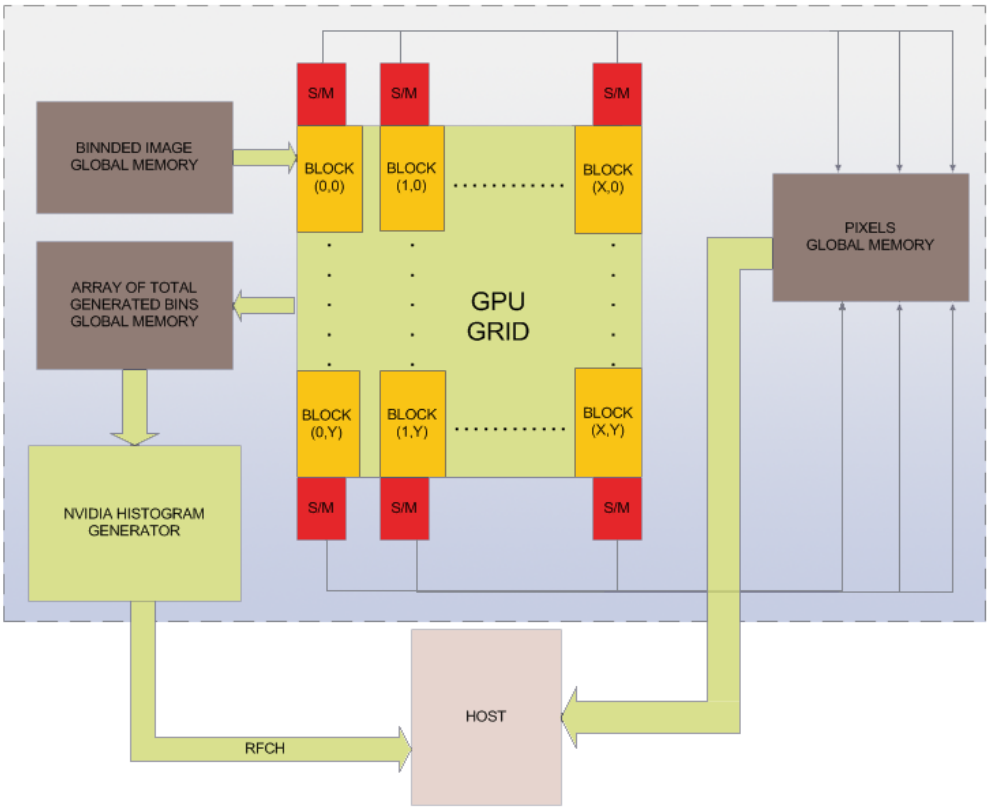


Figure 6.6: High level architecture of CalculateRFCH.
X and Y are the max indexes of a block dimension and vary in each situation depending on the GPU and the phase of the algorithm.

6.3 BPTLD Parallel Implementation

As mentioned in the previous chapter, the selected part of the BPTLD that we decided to port to an NVidia GPU is the Detect() method of the application. The rest of the algorithm will be executed by the Host CPU.
This leads to a new function, which is a version of the original Detect() method, the Detector module.

6.3.1 Detector Module

This module is basically a function containing CUDA APIs, which allows the Global memory of the GPU and the memory of the CPU to interact with each other. Also, it contains the preparation and the call of the kernel.
As input for our kernel we used the arrays:

- ferns_nodes_feature_xp
- ferns_nodes_feature_yp
- ferns_nodes_feature_wp
- ferns_nodes_feature_hp
- ferns_posteriors

They are all two-dimensional and have a size of 91 float variables, except for the latter array with 212,992 float variables. One of the actions of the CPU thread is to initialize these arrays only the first time the module is called, hence the first detection, except for ferns_posteriors which is updated every time the module is called. To calculate the values of these arrays, the application calls the Classifier Class.
Since the CPU would have a considerable amount of work of memory copying the above arrays to the GPU, we decided to use asynchronous memory copy. These CUDA APIs won't block the CPU thread as other default APIs would, the Host stream will keep processing the work we have assigned it to do without stalling, and the result is an obvious performance gain.

One more array that was essential for our kernel was the planar_data which we had to retrieve from the IntegralImage Class, which is the image frame's digital representation. Without it we could not have used the function sumRect, which is the cornerstone of the detecting process because of its most important computational results and the busiest function of the TLD. The array has a size of

1,237,776 integers, and due to its considerable size, we decided to try the zero copy memory optimization in which we succeeded with noticeable performance. Zero copy enables GPU threads to directly access Host memory.

Asynchronous memory copies and the zero copy method require mapped pinned memory from the arrays that reside at the Host side of the module. A page-locked or pinned memory transfer attains the highest bandwidth between the host and the device, and is allocated using the *cudaHostAlloc()* function in the Runtime API. It basically allocates Host variables in a page-locked fragment of the RAM.

Pinned memory is physical RAM that is set aside and not allowed to be paged out by the OS. So once pinned, that amount of memory becomes unavailable to other Host processes, and the Device can fetch it without help from the CPU using DMA (direct memory access). Without DMA, when the CPU performs I/O, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.
Not-locked memory can generate a page fault on access, and it is stored not only in memory, so the Driver needs to access every page of non-locked memory, copy it into a pinned buffer and pass it to DMA. This is a typical slow synchronous page-by-page copy.

The Detector module receives as input from the application the tbb, which is an array of 5 double precision values. We store it in the cached Constant Memory of the GPU and it is used to check if and how the bounding boxes overlap by using a device function accessed only by active threads.

After setting the dimensions of the computational grid using the NVidia GTX 285 compute capability 1.3, the kernel is ready to launch with two-dimensional 64 blocks and two-dimensional 16 threads in each block activating a total of 1024 threads. 1024 is the maximum number of iterations we might need to parallelize depending on the width and height of the integral Image.

The module must return back to the application a C++ two-dimensional type of vector, the bbs, which CUDA cannot support. So we had to calculate the elements of the vector through our kernel and push them into the bbs using standard vector functions at Host.
Each value of the vector contains an array of six values, the bb, and these arrays are calculated in the kernel and copied back to the Host. Every time a kernel execution has finished its work, the module updates the bbs vector.
The kernel outputs have as a base of their calculation the results of the sumRect function, which we implemented as an exclusive device function only to be run by the threads of the GPU. In figure 6.7 we can see the illustration of the Detect module GPU implementation on the GTX 285.

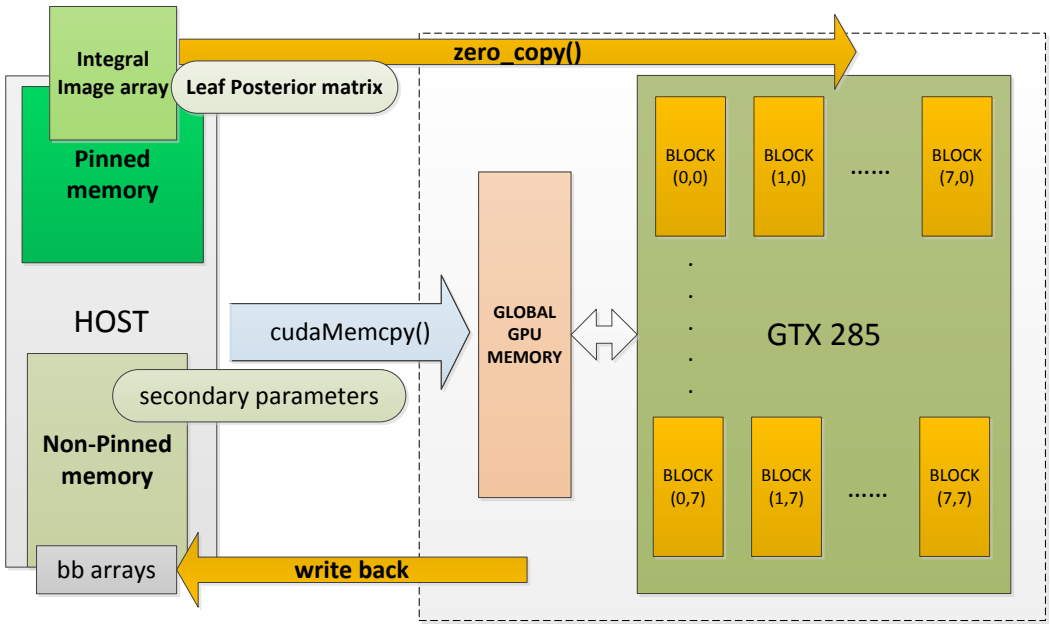


Figure 6.7: High level architecture of the Detector Module

6.4 Difficulties Experienced

During the experimental part of this thesis we dealt with many drawbacks, both technical and theoretical. The CUDA platform was something new for us with its own computing philosophy, different from anything we had met before. We had to get used to not only parallel computing, which has its own mentality, but also to all the new APIs that CUDA offers, and to the code that the GPU can understand and compile. This process took us a respectable amount of time and effort, starting with simple examples and gradually gaining experience to achieve an efficient implementation of our modules.

We believe that every software application contains an inherent parallelism that is waiting for a CUDA developer to extract. But even an experienced developer might not be able to overcome the software time of an application because of its deep serial nature, or achieve a very small speedup that is not worth trying for. And even if a well-designed parallelization has been drawn, the hardware has limits on its resources. This is the job of a hardware engineer, to smoothly tie together the limits and the power of the hardware as well as the designed parallelism of the application. Of course, this job demands many experiments and a lot of time, especially if it is a new studied technology.

For the completion of this thesis, we had to learn a great deal about how Linux libraries work and how they are linked to each other for compiling and running programs. Using Linux is the easiest way to experiment with CUDA, CUDA-enabled GPUs, as well as using OpenCV for the TLD implementation. Since we do not own a personal computer with a CUDA- enabled NVidia GPU, we had to use servers that belong to the Technical University of Crete and contain NVidia GPUs with a CUDA framework. These servers are Linux installed machines and give a user the opportunity to connect to them via the ssh protocol. These servers were the “iraklis” with an NVidia GTX 285 and the “pc16” with an NVidia GTX 580.

The “pc16” has a very good GPU, whereas the rest of the machine, especially the motherboard, cannot cope with its capabilities. We had many difficulties with it

because of its sporadically crashing while running the RFCH modules at their most difficult computations. With the help of the Laboratory staff that administrates the server, we could finally extract the results we wanted.

The TLD could only be run on the “iraklis” server, which is a more stable machine than the «pc16». We experienced problems with the OpenCV library. Although OpenCV was installed to “iraklis” it was under root privilege, meaning that only an administrator could use it. As a simple user, we had to compile and install OpenCV at the hard disk fragment of “iraklis”, which was given to us for our uses. After that we had to link the TLD application to the OpenCV library that was installed in our space of the server. Only then could the TLD be run successfully by “iraklis”. For the installation of OpenCV and the projection of the video sequence from the TLD, we had to connect to the server with ssh –X. This command option enables the user to utilize the graphics interface of the server.

We feel that we have learned many useful utilities around Linux and Linux servers, but until we could finally run our applications it took us a lot of time and effort to deeply understand Linux Operating Systems. Finally, we overcame the difficulties and we are contented with the result.

Chapter 7

Performance and Evaluation of the Parallel Implemented Algorithms

7.1 Introduction

In this chapter we will show our results through the implemented RFCH and TLD modules and try to analyze them.

We used very accurate timers written in C that NVidia uses for its CUDA samples. CUDA programs can be very fast, so each millisecond and even microsecond counts for comparing performance; no standard C or C++ timers are that accurate. CUDA offers timers itself, but we noticed a slight aggravation in our performance as they are not lightweight APIs and we preferred not to use them. We also used the NVidia Visual Profiler, an excellent tool for evaluating the performance of a CUDA application.

7.2 RFCH

As we expected, the GTX 580 could accelerate our modules faster than the GTX 285 because of its superior capabilities and resources. This acceleration could be done by understanding the hardware differences and scheduling processes of the multiprocessors.

We achieved a very good speedup for the ClusterFeatures Module especially for the detecting phase because the CPU thread does minimal work; the kernel does

everything. We also used the directive #pragma unroll for the rest of the for-loops inside the kernel, which extracts very good results.

CalculateRFCH as noted in the previous chapter was very challenging and interesting for a CUDA port implementation. Its branch divergence, a while-loop inside the body of the algorithm, the unexpected number of bins each thread will produce and the calculation of a histogram are serious drawbacks for a decent speedup and parallelization.

A while-loop will almost never perform well, as most loops will not on the GPU. Most of the times we know the amount of iterations in a for-loop and we can use #pragma unroll *N*, where *N* the number of iterations. In a while loop the number of iterations is unexpected in most applications.

7.2.1 Speedup on the GTX 285

Modules	Software time (s)	CUDA time (s)	Speedup
CalculateClusters – 1 object	23.4	1.79	13
ClusterFeatures – 1 object	0.14	0.00147	95
ClusterFeatures – 1 scene in the 1st detecting phase	1.97	0.0049	402
ClusterFeatures – 1 scene in the 2nd detecting phase	1.95	0.0048	406
CalculateRFCH – Part of the sliding window Detection Algorithm	0.013	0.0055	2.4
CalculateRFCH – Part of the MatchImage Detection Algorithm	0.7	0.218	3.2

Table 7.1: Speedup of the RFCH modules with the GTX 285

The above measurements have been recorded with the help of the “iraklis” server of the Technical University of Crete. The CPU of this server is the

Intel® Xeon® Processor E5430, 12M Cache, 2.66 GHz, 1333 MHz FSB.

7.2.2 Speedup on the GTX 580

Modules	Software time (s)	CUDA time (s)	Speedup
CalculateClusters – 1 object	23.4	1.69	13.8
ClusterFeatures – 1 object	0.14	0.00122	114.8
ClusterFeatures – 1 scene in the 1st detecting phase	1.96	0.00210	933
ClusterFeatures – 1 scene in the 2nd detecting phase	1.95	0.00170	1147
CalculateRFCH – Part of the sliding window Detection Algorithm	0.013	0.0018	7.2
CalculateRFCH – Part of the MatchImage Detection Algorithm	0.7	0.06	11.7

Table 7.2: Speedup of the RFCH modules with the GTX 580

The above measurements have been recorded with the help of the “pc16” server of the Technical University of Crete. The CPU of this server is the Intel® Core™2 Duo Processor E8400 6M Cache, 3.00 GHz, 1333 MHz FSB.

7.2.3 Comparing Results

First of all, we noticed that all the modules perform better when running on the GTX 580 rather than the GTX 285.

For the CalculateClusters module, we noticed slight performance acceleration in the GTX 580 using 1024 threads in a block which this card can offer. The reasons for the small difference in performance between the two cards is that although the kernel runs faster with the compute capability 2.0 card, its execution time is sidelined by the many memory copies to and from the device, the inevitable

occupation and calculations of the CPU thread and the fact that the kernel is called iteratively with a while-loop until all the cluster features are fully updated. Inserting the mentioned while-loop inside the kernel could be a logical thought but not realistic because of the terrible performance it will produce.

For the ClusterFeatures module by using the maximum threads per block, 1024, we achieved a great acceleration for our module, even though very good performance is also noticed by the GTX 285.
The vast differences between the training module and the detecting module of the ClusterFeatures are due to the fact that in the training phase we have to copy back to the Host the binnedImage[] array. Also we have a much greater branch divergence in the training phase, which practically means that most threads will perform less work in comparison to the detecting phase.

Using many threads per block is generally a good practice because it hides the latency in Global Memory accesses. If a group of threads (warp) stalls on an access then the next warp will try to access it, and the scheduler will return back to the stalled warp when it has finally finished its work. More warps per multiprocessor means more latency hidden.
Sometimes adding more threads will not perform better and this depends mostly on the nature of the kernel and the scheduling patterns of each card. If the kernel doesn't need many memory accesses and the threads are occupied mostly on calculations using their 32-bit registers, then using fewer threads per block could accelerate the speedup. The selection of threads is not an exact science; the developer must experiment with block sizes and observe performance.
A multiprocessor has a floor on how many threads can be scheduled on it at a time, so if the number of threads per block is relatively small, then more blocks can be scheduled on each multiprocessor; thus more blocks are processed in a clock cycle.
Also, if fewer threads occupy a multiprocessor then fewer 32-bit registers are occupied and more registers can be assigned to each thread. Registers have a throughput of TBs/s.

An example of the above phenomenon is the CalculateRFCH module parallel implementation on the 2.0 compute capability card. By using only 16 threads per block, we achieved the maximum performance for this module. We experimented with the same grid architecture on the GTX 285 but not with the same results, for the kernel performed worse than with the GTX 580. We believe that this is a combination of parameters. Since we reduced the threads in the block, the number of blocks increases. The GTX 285 does not process blocks as fast as the GTX 580, due to a worse clock rate. Also, the much needed shared memory in this module is accessed by the threads slower than the shared memory of the GTX 580, about 2x slower.

The difference in performance of this module run by the two cards is not relative only to the kernel; the Histogram Generator is faster on a compute capability 2.0 card, about 5 times.

7.2.4 Overall Performance

Overall speedup with the help of GTX 285

Configuration	Software time (s)	Hardware time (s)	Speedup
1 object - 1 scene	28.28	4.19	6.7
4 objects – 1 scene	114.30	14.85	7.7
7 objects – 1 scene	202.35	25.24	8.0

Table 7.3: Experiments on our system by increasing the objects on the GTX 285.

The measurements of the software time and the functions of our CUDA version that were not implemented in the GPU were done by the “iraklis” CPU.

Overall speedup with the help of GTX 580

Configuration	Software time (s)	Hardware time (s)	Speedup
1 object - 1 scene	30.54	3.07	7.89
4 objects – 1 scene	123.3	9.6	12.85
7 objects – 1 scene	218.06	16.67	13.1

Table 7.4: Experiments on our system by increasing the objects on the GTX 580.

The measurements of the software time and the functions of our CUDA version that were not implemented in the GPU were done by the “pc16” CPU.

By keeping the number of scenes stable and increasing the number of objects to be detected, we noticed a slight increase in performance. When adding an object, then both training and detecting execution time increases. Since we have managed to achieve a considerable speedup, the fact that performance increases by adding more objects is a positive confirmation of our overall system.

Configuration	Software time (s)	Hardware time (s)	Speedup
7 objects - 1 scene	202.35	25.24	8
7 objects - 2 scenes	234.07	38.34	6.1
7 objects - 3 scenes	265.84	50.87	5.22
7 objects - 4 scenes	299.14	62.23	4.8

Table 7.5: Experiments on our system by increasing the scenes on the GTX 285

The measurements of the software time and the functions of our CUDA version that were not implemented in the GPU were done by the “iraklis” CPU.

By keeping the number of objects to be detected stable and adding more scenes, we noticed a decline in performance. In this case the training execution time does not change depending on the configuration; just the detecting procedure time grows. Since our speedup for the main module of detection, the CalculateRFCH, on the NVidia GTX 285 is relatively low, the above results do not surprise us.

7.2.5 Applied Optimization on the RFCH modules

A considerable difference in performance was noticed when we identified that certain methods of GPU computations could be altered in the search of higher acceleration.

- Using Constant Memory. The cluster scheme, the ClusterPoint[] array and SqrRadius[] array, was first stored in the Global Memory which generates low throughput. Since the arrays are relatively small and have been used only for extracting their elements, we decided to store them in the faster Constant Memory. Constant Memory is faster but variables stored there have read-only permissions. Also, one has to be careful with Constant Memory due to its limited size.

- Eliminating data transfers. We tried as much as possible not to transfer unnecessary data from and to the GPUs for a smaller execution time. For example, the CalculateRFCH detecting module needs the binnedImage[] array which is generated in the ClusterFeatures detecting module. We could have copied the result back to the Host and pass it to the CalculateRFCH module by reference and then the latter module could copy it back to the Device again. Instead, when the binnedImage[] is created, it just resides in the Global Memory waiting for the CalculateRFCH to utilize it. We have eliminated two memory transactions.

- Constrain the Global Memory stores. We tried not to store unnecessary variables in the Global Memory. This does not help for an acceleration of the application, but mostly to avoid memory segments overlapping, thus losing important data. For example, we have used only one device array of Features, replacing the elements of it each time an object or scene is about to be processed. We could have used a Features[] array for each object or scene, but since its size is 2.15 MB and in the case the algorithm is about to process over 500 images, the Global Memory will overflow since the Device Memory is limited to about 1 GB.

- Shared memory and its attributes. While our first design of the CalculateRFCH module was with the use of shared memory, we will mention it as a very important CUDA optimization. That's because it is a very fast on-chip memory fragment and able to use techniques such as parallel reduction or parallel scan.

- Selecting block sizes. Depending on the kernel, different block sizes might boost the kernel execution as mentioned above. A characteristic example is the CalculateRFCH module.

7.3 BPTLD

For our experiments on this hardware algorithm, we used the NVidia GTX 285. We selected two videos for the verification and the metrics of our system, while we were able to compare the C++ implementation with ours.

7.3.1 Evaluation of our modules through experimental videos

The first video shows the meteorite that hit Russia in December of 2012.

Figure 7.1: The meteorite seems small to the human eye

Figure 7.1 shows one of the first frames of the video sequence and it shows the bounding box around the meteorite while it is still far from earth.

Figure 7.2: While the meteorite heads towards earth it seems larger

Figure 7.2 shows the bounding box around the meteorite after a few frames. This time the meteorite has come much closer to earth. We selected this video to show that the BPTLD actually does identify an object while it changes shape as is stated in [17].
In this case the meteorite seems much larger to the human eye but also to the machine. Its tail has grown and its illumination characteristics have altered.

Below is the table of the software and parallel implementation execution times and comparisons.

Studied Entity	Software time (ms)	Hardware time(ms)	Speedup
Detect module with I/O	110	7.5	14.5
Detect module without I/O (kernel)	110	3	36.7

Table 7.6: Results of the TLD GPU implementation in software and hardware on the meteor video.

We experimented with one more video which shows a moving white car. We made this video to test if the Detector will identify the object by keeping the camera unsteady as it is mentioned in the original paper [17].

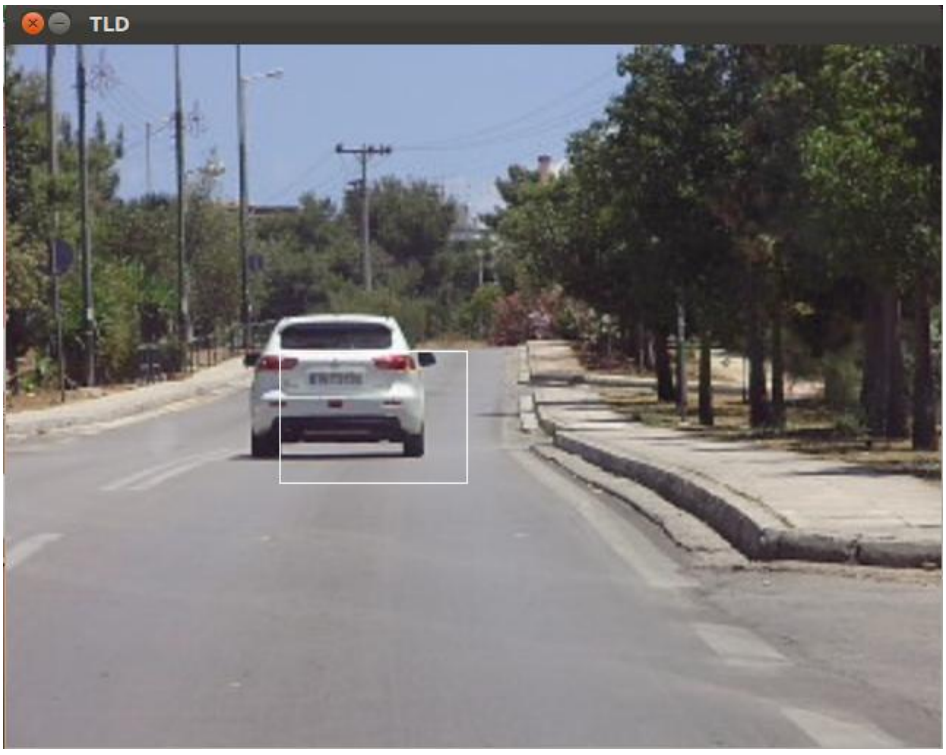


Figure 7.3: Tracking bounding box of a moving car.

The next frame capture is two frames following for Figure 7.3 while the camera has changed its position to the left. We can understand that because the frame captures less of the pavement on the right.

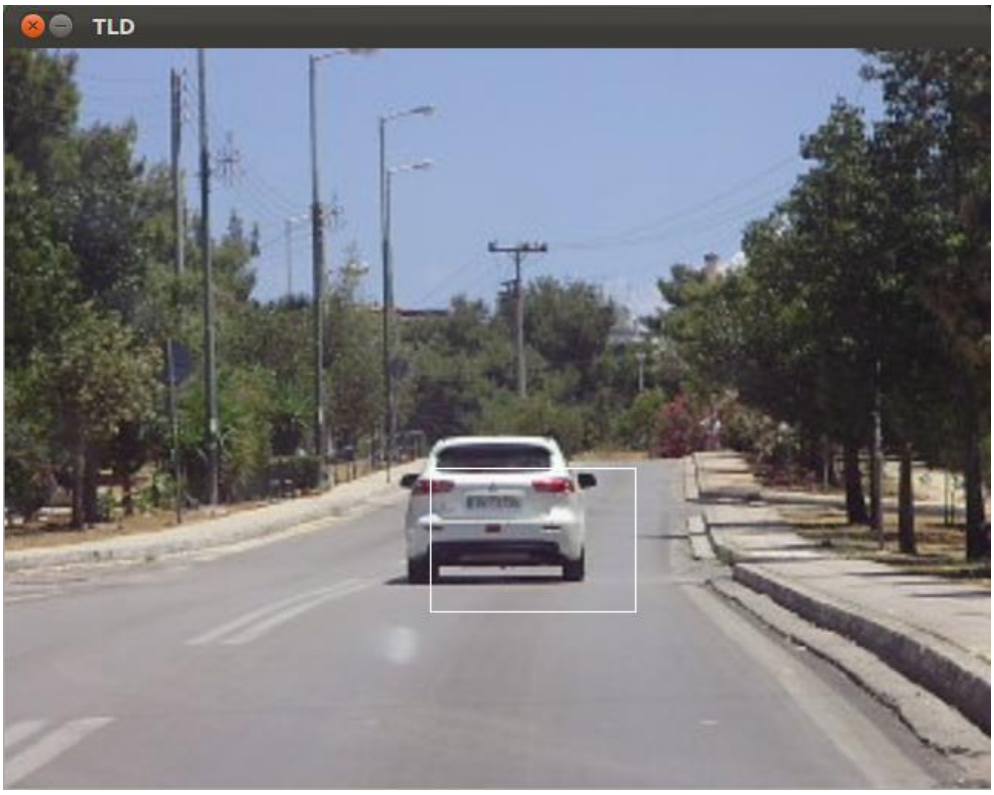


Figure 7.4: Two frames after Figure 7.3.

Although the camera was unsteady on purpose, the Detector manages to track the object.

The software and hardware comparison is shown in the table below.

Studied Entity	Software time (ms)	Hardware time(ms)	Speedup
Detect module with I/O	123	8.4	14.5
Detect module without I/O (kernel)	123	0.65	37.8

Table 7.7: Results of the TLD GPU implementation in software and hardware on the car video.

We believe that the small differences in time between the two experimental videos are due to the fact that the bounding box of the second video is larger and also due to the fast camera movement.

7.3.2 Applied Optimization on the Detector module

We applied the following optimization techniques in the Detector module:

- Asynchronous memory copy between the CPU host and the Device Memory. By not blocking the CPU thread, execution time is reduced.

- Using Constant Memory. The input of the Detector module, the tbb is stored in Constant Memory because of its very small size and we only need threads to extract information from it and not for threads to store new ones.

- Selecting block sizes. As mentioned, the optimal block size, which was found experimentally, is 16 threads per block. This way our kernel runs faster.

7.3.3 An FPGA Implementation of the BPTLD Algorithm

We will show the results of another implementation [25] of the BPTLD algorithm based on FPGA hardware, which utilizes a distributed memory subsystem divided in blocks. The CPU used for timing the software version is a 2.4 GHZ dual core state of the art CPU.

When the above implementation executes the complete algorithm with a conventional memory subsystem accessing a single block at a time, the performance generated is virtually the same with the one the CPU generates as we can see in Table 7.7. This is because the classification problem, when solved with the Random Forest approach, is memory bound, and the prototype is used with an on-chip memory which has roughly the same bandwidth as the one utilized in the Intel CPU.

When the number of blocks is increased, thus creating a distributed memory, the implementation ends up with a considerable speedup over the software. The speedup which is achieved does not include any I/O overhead between the FPGA and the Host in software-hardware co-design scenario.

Memory partitioning in blocks (dual port)	**Average collisions (with scrambling)**	**Speedup @ 200 Mhz**	**Effective Memory BW (TB/sec)**
1	32	0.96	0.29
8	2.65	14.64	3.54
16	3.15	23.27	5.95
32	**3.6**	**41.98**	**10.42**

Table 7.7: Performance evaluation on a Virtex-6 VLX130T device at 200Mhz

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we described an efficient way to improve the performance of the Receptive Field Cooccurence Histogram and the Tracking-Learning-Detecting Algorithms using NVidia GPUs via the CUDA architecture.
We have designed CUDA modules based on the function hotspots of the above algorithms, so as to accelerate the overall performance of these applications. These modules have a dual nature of both software and hardware. The GPU works as co-processor to the CPU which runs C++ code.
Our hardware modules are based on parallelizing the loops that are responsible for the bulk of execution, by utilizing the threads of the GPUs which virtually run in parallel. Our experiments showed that we achieved an overall speedup of over 13X in both algorithms, while the maximum speedup we achieved for a single module (ClusterFeatures) was over 1000X.
For the RFCH, the hardware is not fixed to a specific input image size because it is not designed to transform a real image into a form that a machine can understand. This is the work of the software. The hardware modules must know at least the size of the input image for them to launch the appropriate number of threads that can access every pixel. We experimented with input images of size 640x480.
For the BPTLD, we concluded that its respective module can process 640x480 size of frames of high resolution (as the meteor example) and low resolution (as the white car example), in addition to handling object and camera alterations to a certain limit.

8.2 Future Work

Although we are content with our results, we believe a greater acceleration can be achieved. CUDA and the NVidia GPUs have evolved tremendously in the last years, due to the successful architecture and the relatively developer-friendly framework NVidia has constructed. The very good documentation and the many sample codes NVidia has published is a proof of the successful developing product that they have created.
The features and capabilities of the CUDA-enabled GPUs are truly vast and impressive. For a better performance of the algorithms we have studied, additional research could be done on the inherent parallelism of our modules, while experimenting on the different GPUs that are in the market.

Each card, having its own features, could be selected depending on what the developer wants to get out of the card. Even cards that are not as fast computation-wise may win in energy efficiency.
The CalculateClusters module was hard to parallelize efficiently because of the complexity of the K-Means algorithm. Additional research should be done, perhaps experimenting with the very fast shared memory using array decomposition techniques. These techniques should be studied for every module and for every CUDA application, since the results of using shared memory are astonishing.

For the TLD implementation, we believe that if it is to be used in real world applications, the video stream must not be generated and displayed by the NVidia card itself. The card will be busy in other tasks, which will not cope well with the kernel, the CUDA APIs and the execution of the CUDA code. These problems could be memory collisions in the Device Memory with a result of data losses, segmentation faults on pointers residing in the memory as a result of unexpected termination while the application is running. As a result we may face an overall unstable execution of the TLD implementation. Also, it would be interesting to experiment with different machines and cards on this module with the restriction

that the machine will support OpenCV compiled with the CUDA compiler, which is an important factor of the BPTLD (as in the NVidia GTX 285).

Bibliography

[1]: http://en.wikipedia.org/wiki/Computer_vision

[2]: http://en.wikipedia.org/wiki/Outline_of_object_recognition

[3]: Sameer A. Nene and Shree K. Nayar and Hiroshi Murase, \Columbia Object Image Library (COIL-100)" in Technical Report CUCS-006-96, Department of Computer Science, Columbia University, 1996.

[4]: JOHN CANNY. A Computational Approach to Edge Detection. IEEE TRANSACTIONS ON PATTERN ANALYSIS
AND MACHINE INTELLIGENCE, VOL. PAMI-8, NO. 6, NOVEMBER 1986

[5]: S.Ekvall, D.Kragic, \Receptive Field Coocuurrence Histograms for Object Detection", in
IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 84 - 89, 2005

[6]: B. Schiele and J. L. Crowley, \Recognition without correspondence using multidimensional receptive fields histograms", International Journal of Computer Vision, vol. 36, no. 1, pp. 31{50, 2000.

[7]: O. Linde and T. Lindeberg, \Object recognition using composed receptive field histograms of higher dimensionality", 17th International Conference on Pattern Recognition, ICPR'04, 2004.

[8]: http://en.wikipedia.org/wiki/Video_tracking

[9]: Seung In Park, Sean P. Ponce, Jing Huang, Yong Cao and Francis Quek. Low-Cost, High-Speed Computer Vision Using NVIDIA’s CUDA Architecture. Center of Human Computer Interaction,Virginia Polytechnic Institute and University Blacksburg, VA 24060, USA

[10]: Wenhao He and Kui Yuan, An Improved Canny Edge Detector and its Realization on FPGA", in Proceedings of the 7th World Congress on Intelligent Control and Automation. June 25 - 27, 2008, Chongqing, China.

[11]: Christos Gentsos, Calliope-Louisa Sotiropoulou, Spiridon Nikolaidis and Nikolaos Vassiliadis, Real-Time Canny Edge Detection Parallel Implementation for FPGAs", International Conference on Electronics, Circuits, and Systems (ICECS), pp.499, 2010.

[12]: Antonis Nikitakis, Savvas Papaioanou, Ioannis Papaeftstathiou, A novel low-power embedded object recognition system working at multi-frames per second. October 11-October 12, Tampere, Finland

[13]: Changjian Gao and Shih-Lien Lu, \Novel FPGA based HAAR classifier face detection algorithm acceleration", International Conference on Field Programmable Logic and Applications, pp.373 - 378, 2008.

[14]: Vinod Nair and Pierre-Olivier Laprise and James J. Clark, \An FPGA-Based People Detection System", in EURASIP Journal on Applied Signal Processing 2005:7, 1{15.

[15]: P. Chang and J. Krumm, “Object recognition with color cooccurrence histograms,” in CVPR’99, pp. 498–504, 1999..

[16]: J. B. MacQueen, \Some Methods for classification and Analysis of Multivariate Observations", Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, pp. 1:281{297, University of California Press, 1967.

[17]: KALAL ZDENEK, MATAS JIRI, MIKOLAJCZYK KRYSTIAN. 2009. Online learning of robust object detectors during unstable tracking. In 3rd On-line Learning for Computer Vision Workshop, Kyoto, Japan, IEEE CS.

[18]: BPTLD.2011. https://github.com/Ninjakannon/BPTLD.git

[19]: http://opencv.org/.

[20]: VIOLA PAUL, JONES MICHAEL. 2001. Rapid Object Detection using a Boosted Cascade of Simple Features.
International Conference on Computer Vision and Pattern Recognition.

[21]: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[22]: https://devtalk.nvidia.com/default/topic/511531/code-general-purpose-histogram/

[23]: http://www.nvidia.com/object/cuda_home_new.html

[24]: NVIDIA, CUDA Programming Guide Version 4.0. 2011, NVIDIA Corporation: Santa Clara, California.

[25]: ANTONIS NIKITAKIS. 2013. A novel Embedded system for vision tracking. (Unpublished).