Multiple Objects Tracking with a Surveillance Camera System

T.R.Bandaragoda,U.D.C.Dilhari,C.S.Kumarasinghe,D.T.R.Mallikarachchi, J.G. Samarawickrama, A.A. Pasqual

Abstract

Increasing use of CCTV for city and building surveillance has given rise to an environment where an object (a person) might traverse through the field of view of many cameras. In this paper we explore the problem of tracking multiple objects in a multi camera environment, which is a highly addressed area in computer vision. Our research involves real time tracking of objects while they are moving in a multi camera environment with non-overlapping field of views and detecting them when they re-appear in the same or another camera in the same system. Previous methods of using offline trained classifiers with huge databases are time consuming and have the drawback of incapability of detecting arbitrarily selected objects. We address this issue by online training with the initial sample given and is based on the TLD (Tracking, Learning, Detection) framework. We extend the idea to formulate our methodology to create a framework that can track multiple objects in multiple video streams in real time. We have developed upper layers as a thread based architecture in order to incorporate multiple video feeds and to handle multiple objects. We have integrated CUDA (Computer Unified Device Architecture) programming model to add parallelism to independent processes and execute compute intensive algorithms. GPU computing offers an ideal computing environment to improve our framework. Our optimization of the algorithms, careful usage of parallel computing and proper utilization of GPU resources have contributed in achieving a processing time of less than 60ms for multi objects in multi camera environment.

1. Introduction

Object tracking is a highly discussed area, especially in defense applications, surveillance, traffic control, medical imaging, robotics, autonomous navigation and also in business applications as well. Over the years numerous algorithms have been developed in the areas of object detection, tracking and video processing. But all these techniques inherit their own drawbacks in the areas of speed, accuracy and resource utilization when it comes to multiple objects tracking with multiple cameras. Moving object detection and tracking is a complicated task due to multiple parameter changes representing features and motions of objects. Human tracking is more challenging due to varying posture, different appearances and partial or full occlusions during movements, with the limited range and slow resolution of surveillance cameras and varying environmental conditions and lighting. The approaches like SIFT [1] and SURF have issues with real time performance where as classifier based approaches like [2] and [3] cannot be incorporated to arbitrary objects. Also when looking at the approaches like [4] we see that the problems arise when implementing it for a dense environment. Background subtraction fails when the environment is dynamic and full or partial occlusion is underway. But it is computationally less intensive and can be operated in real time. In [5] mean shift algorithm is utilized but it has some practical drawbacks. Using mean shift tracker alone shows issues when the currently tracking object goes out of the field of view of the camera. The tracker alone will produce inaccurate results when multiple objects with similar color histograms appear within the same camera. Techniques like mean shift tracking and Kalman Filter based tracking have shown inefficiencies when it comes to long discontinuities and occlusions. The human identity recognition [6] is the most promising work we came across related to object tracking with long occlusions and discontinuities. However it uses a huge dataset for the purpose of training and this required a blob extraction process and real time tracking capability is difficult to achieve. These approaches brought us to the conclusion that if we are to come up with a robust application we need to perform online training with the help of initial sample we provide and continue on tracking and learning simultaneously until the tracker is lost. The learning phase would generate the necessary classifier parameters to detect the object once it re-appears in the video stream. The closest work presented with respect to our project was done by Zdenek Kalal in his work under P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints. He has presented the TLD (Tracking, Learning, Detection) framework which is specialized for single object tracking in a single video stream. In our design we have exploited this framework to come up with a platform that can track multiple objects in a multi camera environment. We have incorporated parallel processing into our design by implementing most of the critical algorithms in CUDA [7]. We are using thread based architecture for the upper layers of the system to run multiple video streams. GPU Computing features gave us a platform to write our algorithms in most efficient ways and we have utilized the best suitable practices to manage GPU resources to achieve the optimum performance. With CUDA programming architecture we have achieved a high processing gain for complex algorithms which are now can be operated in real time for multiple objects in a multi camera environment.

2. Methodology

The methodology is based on the TLD (Tracking, Learning, and Detection) framework. The tracking, learning and detection parts are implemented separately and algorithms are re-engineered to suite multiple objects tracking and parallel execution in GPU. The proper implementation for the algorithms are tested based on the amount of data to be processed and available GPU resources. The overall system has multi-threaded architecture and it embodies two kinds of parallelism. The video stream parallelism is achieved using C++ windows threads and object parallelism is achieved by OpenMP. Mutexes (Mutual exclusions) are used where necessary to manage common resource allocations.

Figure 1 shows the overall architecture of the system. Each video stream is treated as a separate master thread. Each image from a camera is subjected to gray scaling, integral image creation and square integral image creation. Each object that

T.R.Bandaragoda, U.D.C.Dilhari,C.S.Kumarasinghe,D.T.R.Mallikarachchi,J.G. Samarawickrama and A.A. Pasqual are with Department of Electronic and Telecommunication, University of Moratuwa, Sri Lanka.

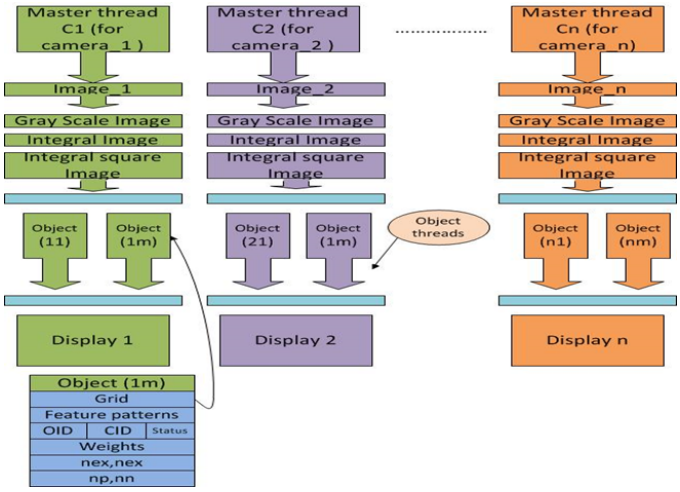

Figure 1: Overall Architectural Diagram

is selected to be tracked within a camera stream is treated as separate threads. Each of these objects possess an object ID, a search grid, feature patterns, Camera ID, weights, positive samples, negative samples, number of positives and number of negatives.

2.1 Preprocessing

When the images are queried from the cameras they are subjected to preprocessing operations. The images with a resolution of 640x480 are copied to the global memory of the GPU and they are gray scaled, smoothed and integral image is created and they are stored in the Global memory for further accesses. Gray scaling is done by R*0.1144 + G*0.5867 + B*0.2989. This is pixel wise parallelized allocating each pixel to a separate thread.

The image blurring is carried out by a 9*9 kernel. In order to parallelize the process we first perform a row wise multiplication and then a column wise multiplication.

In the row wise operating kernel, we divide each row into blocks of 248 pixels. Each thread block will calculate the blurred value of those 248 pixels as a thread block possesses 256 threads. These values are used as we need four pixels from both sides to do the blur function. (i.e. 4+248+4 = 256). An array of size 256 is used in shared memory to store those pixel values as the global memory access is slow. For the column wise operation we used a similar kernel at first, but due to its inefficiency we came up with another approach. The image is usually stored in row by row. So the above approach for the columns will not work as it will violate the memory coalescing. Memory coalescing is consecutive threads accessing consecutive memory address which would result is great performance increase. Hence here blocks of pixels from each row are accessed but shared memory is not used since we dont see an overlap between pixels accessed to get the column blur value.

Integral image and square integral image calculation is another heavy process. Our first attempt was by assigning a thread for each row. Hence each thread will calculate sum and square sum for pixels in all columns using a for loop. But it is inefficient to run a for loop for $width$ number of iterations by each thread. Then the method used in [8] is implemented. The algorithm can be applied for images that have $2n$ number of rows and columns. So we zero pad to achieve an image of 1024x512. For the calculation of row sums, 512 threads and 1024 size shared memory is used while for row sum, 256 threads are used. In this way row sum is calculated in 20 iterations and column sum is calculated in 10 iterations.

2.2 Object Initiation

Object initiation is done with the marking of the bounding box around the desired object to be tracked. With that the search grid is created. Search grid is a grid consisting of 21 differently scaled bounding boxes of the originally drawn box. It is an array of (x,y) coordinates where the bounding box can be located. This is highly essential for the detection and learning phases. Random binary features are generated for each object for the Randomized Fern Classifier. Here 130 features are generated such that there are 10 trees and each tree has 13 features. The features points are found with respect to a 1x1 bounding box and then they are scaled by a scaling factor to suite to the 21 differently scaled boxes.

2.3 Finding Overlaps

This is an important task for the positive and negative patches identification. Overlap is the intersection ratio of the bounding box and grid boxes. It is found by [9]

$$Overlap = \frac{intersection}{(bboxArea + gridBoxArea - intersection)} \quad (1)$$

where $bboxArea$ is the bouding box area.

As the number of grid boxes are extremely high this takes a considerable amount of processing time. For the overlap calculation (number of grid boxes / 256) number of thread blocks are chosen with each having 256 threads. Hence each thread will calculate its own overlap independently.

2.4 Pattern Generation

Pattern generation is associated with the nearest neighbor classifier. Here a 45x45 size sample of the desired patch is taken and pattern value is generated by subtracting mean from each pixel within 45x45 patches. Here we use 2025 threads with shared memory of array size 45. Each thread will obtain its intensity value using bilinear interpolation. All pixel values in a row are added to the first pixel and they are added to get the sum.

2.5 Positive Data Generation

This is a collection of processes. Here first we find the grid boxes with high intersection ratio with drawn bounding box. Sorting all the intersection ratios in descending order and finding highest 100 values were inefficient. So we first selected all the grid boxes with intersection ratio greater than 0.6 and then we sorted those ones. This process is not done in GPU as we found that sorting a few amount of data provides less gain because memory transfers between Global memory and the host memory consumes a significant time. Finally a hull box is created considering the margins of the selected grid boxes. For these positive samples, patterns are generated as mentioned in above section. Binary feature pattern generation for Fern Classifier is done along with a warping operation. 20 warps are created for the first camera frame and 10 for each and every frame. Warping is a huge compute intensive process and it is an operation which we achieved a high gain.

Warping can be done easily on a GPU due to its hardware based interpolation capability in texture binded images. Image which is already in the GPU is texture bounded(only a logical binding) and a clone of the same image($imageWarped$) is taken in to a different location in GPU main memory(data transfer within the GPU main memory is fast). Pixel values for the pixels inside the hull box of the $imageWarped$ is calculated using the affined matrix values and the original image pixel values.

$$x = matrix[0] \times u + matrix[1] \times v + matrix[2]$$
$$y = matrix[3] \times u + matrix[4] \times v + matrix[5]$$
$$imageWarped(u, v) = ImageOriginal(x, y)$$

x and y are usually non integer values so we have to do interpolation. In CUDA this is a hardware based. In our methodology a single thread is allocated to each pixel in the Hull area. Since matrix values are common to each thread they can be stored in shared memory. But since new Fermi architecture contains a local cache which caches the matrix values automatically. For Fern classifier 10 integer values per bounding box is generated by evaluating 10×13 binary features. A single thread is responsible for one bounding box. There are data dependent branching which prevents from using shared memory. Thus L1 cache limit is increased. [10]

2.6 Negative Data Generation

Negative data is generated considering the grid boxes that have an intersection ratio of less than 0.2 with the drawn bounding box. Negative example patterns and binary feature patterns for Fern Classifier are found using a CUDA implementation. The variance of each grid box is calculated by one thread using integral and square integral images stored in global memory. If this variance is greater than a predefined threshold, those boxes are taken for intersection calculation. For each of the filtered boxes, binary feature patterns are calculated and they will serve as negative data for the object.

2.7 Confidence Calculation for Nearest Neighbour Classifier

NN Classifier is one of the two classifiers used and it is the stronger of the two. Here the distances between positive and negative patterns are compared with existing examples to find out which of the patterns have close resemblance to existing ones. Normalized Cross Correlation is measured as a distance and probability that a patch it not being similar to negative examples is calculated.

$$NCC = \frac{\sum_{i=0}^{n} a_i \times b_i}{\sqrt{\sum_{i=0}^{n} a_i^2 \times \sum_{i=0}^{n} b_i^2 + 1}} \times 0.5$$

where $A = \{a_1, a_2, a_3., a_n\}$ and $B = \{b_1, b_2, b_3, , b_n\}$ and NCC is the normalized cross correlation between A and B. For CUDA implementation this process is parallalized as follows. The size of the patch which is used for pattern generation is 45x45. (i.e. There are 2025 pixels). We have defined a thread block to have 1024 threads. This is due to the fact that warp size is 32 and multiple of 32 threads are executed when running the application. [11] [12] This setup would enable each thread to read two consecutive memory locations from each patch. Each thread would perform following operations and store the results in a shared memory.

$$f1f2[tid] = a1 \times b1 + a2 \times b2$$
$$f1[tid] = a1^2 + a2^2$$
$$f2[tid] = b1^2 + b2^2$$

Where $a1, a2$ are two consecutive elements from first patch, $b1$ and $b2$ are two consecutive elements from second patch, tid is the thread ID which varies from 1 to 1024 and $f1f2$, $f1$ and $f2$ are shared memory arrays. Then the threads 0 to $2 \times n$ of a block will carry out the following arithmetic operation on shared memory arrays.

$$f1f2[tid] = f1f2[tid] + f1f2[tid + 2 * n]$$
$$f1[tid] = f1[tid] + f1[tid + 2 * n]$$
$$f2[tid] = f2[tid] + f2[tid + 2 * n]$$

Here n changesfrom 512,256,128,64,32,16,8,4 in the order stated. The final value is calculated for tid equals to 0 as follows

$$f3[blockIdx.y] = \frac{f1f2[0] + f1f2[1]}{\sqrt{(f1[0] + f1[1]) \times (f2[0] + f2[1]) + 1}} \times 0.5$$

Since each calculation is local to a block, inputs that are required for a certain block calculation are loaded into shared memory. We have used 12288 bytes (1024*4*3) of shared memory per block. Based on the NCC values confidence values are calculated.

$$conf = \frac{(1 - \max(nDist))}{(1 - \max(nDist)) + (1 - \max(pDist))}$$

where $max(nDist)$ is the, maximum of distances between a certain patch and each negative example stored and $max(pDist)$ is the, maximum of distances between a certain patch and each positive example stored. Finding the maximum as in CUDA perspective is done by spawning of $n\times$number of +ve example elements and $n\times$ number of -ve example elements, threads for each case. Final confidence calculation is done using a separate kernel and number of threads spawned for this task is equal to the number of input patches.

2.8 Tracking

Tracking is another main operation in TLD framework which is used to track an object within the current video stream. At this point we need to make sure that tracker is run only for the objects that are not lost. For that we maintain a status variable for each object. Median flow tracker which is used is an extension of Lucas Kanade Optical Flow tracker [13] that tracks number of points between consecutive frames using median. A significant performance gain was achieved by implementing tracker as a separate module using CUDA. The algorithm is based on Pyramidal Implementation of the Lucas Kanade Feature Tracker [14]and it is modified accordingly to be executed in parallel. The 100 points selected within the bounding box are processed with 100 threads with no shared memory. This lead us to process optical flow for three step image pyramid in less than 5ms. Here we identified a bottleneck of having three step loop which is 10x13x13 per point. Thus we utilized shared memory such that one thread block per one point. 169 threads per block were used to handle two 13x13 steps in the previous loop. This improved the processing time to 2ms. Using good features as in [15] would result in losing number of points that can be tracked gradually

with the time. Hence 100 points are selected neglecting the good features. The images are texture bounded to performe bilinear interpolation for image pyramid creation.

Since 100 points we selected are not good features, the reliability of the predicted points remains a question. So the unreliable points are discarded based on median values of two metrics, NCC and Euclidean Distance. Euclidean Distance is found between original points and points found from backward optical flow. [16]. These operations are CUDA implemented and calculating Euclidean distance is the last operation in tracking module. At this point we copy all the results from GPU memory to CPU memory to ensure that memory transfers are minimum.

2.9 Detection

Detection is a cascade of Fern classifier and NN classifier. Fern being a lightweight classifier is used to find 100 possible boxes where the object may reside out of all the grid boxes. Then the NN which operate based on NCC takes in these 100 boxes and find out the most probable one. Fern functionalities based on [17] are completely implemented using CUDA.

We identified that the kernels executed in tracker and detector has no dependencies and can be executed in parallel. The GTX 480 specification states that its compute capability is 2.0 and it can have one concurrent kernel. So by avoiding the function call cudaThreadSynchronize() we execute tracker kernels and detector kernels in two CUDA Streams. We avoided using cudaHostAlloc() as it required an additional memory copy, so instead we used page locking mechanism available in Visual C++, VirtualLock().

The final decision on the next location of the desired object is taken on the basis of tracker and detector outputs.[9] (original TLD paper) This involves a lot of decision makings and clustering. As the best practices of CUDA [7] suggests and also with practical implementation issues we avoided these decision making and clustering processes from CUDA implementation process.

2.10 Learning

During learning phase the Randomized Fern and Nearest Neighbor Classifiers are updated. They are updated based on the confidence values generated for both classifiers. Fern learning process is started by copying the positive and negative patterns generated by random binary features into GPU global memory. In parallel architecture one thread is assigned to train Fern using one pattern. Here due to too much branching, code exhibit a lot of data dependencies. Hence we do not utilize shared memory which will result in increasing the L1 cache memory size from 16KB to 48 KB. From binary feature point generation we have 81920 patterns and each pattern has a weight. During training process it is updated using

$$Weight = \frac{nP}{nP + nN}$$

where nP is the number of times this pattern marked as positive and nN is the number of times this pattern marked as negative. [18]

2.11 Lost Objects

An object in the system has its own object ID, capture ID which keeps the current camera ID and status variable. When an object is lost, the status is changed to false and capture ID is set to NULL. When the object is lost its capture ID is changed to the next camera in the system so that when master thread belonging to that camera operates it will run the detector for lost objects. When they are found, the status and capture ID are set back to respective values. Our initial method of changing the capture ID randomly was abandoned as it tends to generate same number over and over again.

3. Results and Discussion

Keeping the TLD framework as a basis we have come up with a methodology that can track multiple objects in a surveillance environment. We have also introduced a platform where you can use GPGPU (General Purpose Graphic Processing Unit) computing for compute intensive surveillance applocations. In this implementation we have introduced optimum ways for parallel computing for certain algorithms. We have demonstrated proper usage of CUDA architecture and how to utilize GPU features such as registers, shared memory, texture memory and streams. We have highlighted the concept of minimizing memory transfers and using branchless operations for parallel processing to gain better performance.

The following sequence depicts the tracker results for two cameras tracking two human beings. It clearly shows how the objects are tracked when they are full or partially occluded and unique identification is displayed by using two colors.

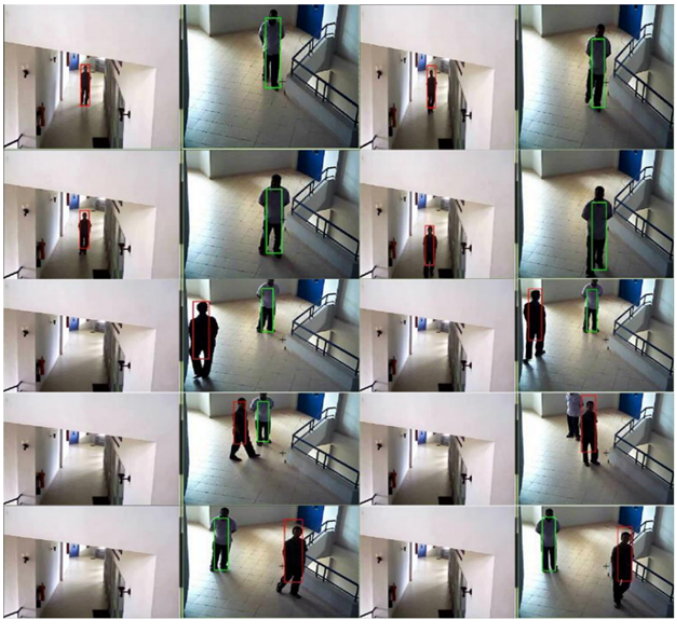

Figure 2: Sequence of images depicting tracking multiple people in two cameras

In CPU/GPU implementation processing times for individual kernels are monitored with the help of NVIDIA visual profiler and occupancy calculator is used to measure the resource utilization of the GPU. The following table gives the kernel processing times and resource utilizations for optimum performance level we achieved. These results are for a single execution for an image with a resolution of 640x480.

The kernel processing times given are the processing times without memory copies. Figure 3 compares the overall results for multiple objects tracking with two cameras. The times indicated here include memory copies between GPU and

Table 1: Kernel Information and execution times of major components.

Kernel Name	Registers	Shared Memory(bytes/block)	Threads/ block	Processing Time(Kernel)
Image blurring row kernel	9	256	256	37.8us
Image blurring col kernel	13	0	256	69.12 us
Row Integral Image	13	1024	512	235.91us
Col Integral Image	13	512	256	236.8us
Row Square Integral Image	13	1024	512	236.2us
Col Square Integral Image	13	512	256	234.28us
Finding Overlap	12	0	256	46.83us
Pattern generation	20	49152	[15 15]	29.78us
Warping	11	0	[16 16]	81.6us
Positive Data generation	11	0	256	591.7us
NN Distance	18	1536	128	103.335us
Optical flow	24	8220	196	288.9664us
Gray scaling	6	0	256	25.25us
Normalized Cross Corelation	20	12288	1024	103.335us
Euclidean distance	7	0	128	3.12us
Fern learning	13	0	256	18.3us
Fern thresholding	6	0	256	483.456us
Pattern calculation	14	0	256	1142.66us
Calculate confidence	18	0	256	547.26us
Fern positive data	14	0	256	591.74us
Fern negative data	14	0	256	1035us

RAM. The results were obtained using a machine with Core i5 2.8 GHz processor and a 4GB RAM. The GPU used was a GTX 480 with the compute capability of 2.0. The CPU results were also obtained using the same machine without GPU support. The important fact that need to be highlighted is that CPU only code is unable to optimize to the extent in which CPU/GPU code is optimized. The reason is the CPU/GPU code is based on the CUDA programing architecture.

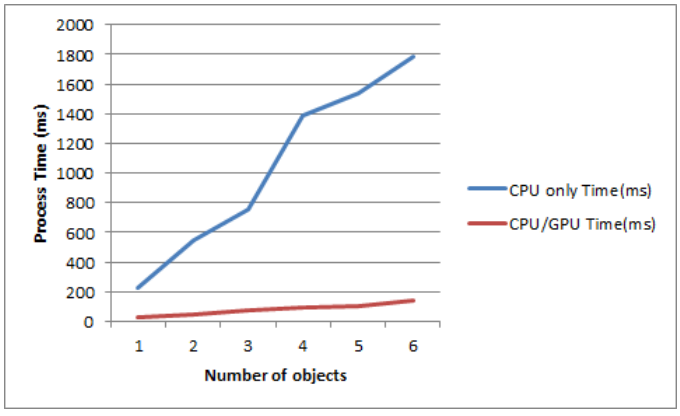

Figure 3: The graph depicts the comparison of processing times

We have identified the main challenges that would deteriorate the results of the system. One such major issue is the lighting angle effect which would require us to place the cameras in such a way that lighting conditions are homogenous. Marking of the initial object also plays a major role in achieving better results. When the background information is large in the marked object the bounding box tends to stick to the location as it would contain more samples with background information.

In terms of GPU computing we came across occasions where we needed to correctly measure the number of shared memory we need use and number of threads need to be used to get the desired speed. In kernel execution the thread synchronization is extremely important to achieve correct results. When the kernel is called the control is given back to CPU. If the data are dependent it is extremely important to place kernel calls at proper places and introduce thread synchronizing methods. The atomic operations allow the sequential access of threads to a common shared resource. Even though we used these operations at our earlier stages, later we avoided using them as they increased the processing time.

GPU computing has given us a proper environment to introduce real time surveillance with great amount of compute intensive processing. The best practices and optimization results we have introduced would be of great advantage for the future of high performance computing as well as surveillance.

References

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," in *International Journal of Computer Vision, 2004*, Vancouver, Canada, January 2004.

[2] P. Viola and M. Jones, "Robust real-time object detection," in *Second International Workshop on Statistical and Computational of Vision Modeling, Learning, Computing, and Sampling*, Vancouver, Canada, July 2001.

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *CVPR Vol 2*, June 2005, pp. 886–893.

[4] J. Zhou and J. Hoang, "Real time robust human detection and tracking system," in *CVPR*, 2005 June, pp. 149 – 149.

[5] E. Ben-Israel and Y. Moses, "Tracking of humans using masked histograms and mean shift," march 2007.

[6] R. M. Omar Oreifej and M. Shah, "Human identity recognition in aerial images," in *CVPR*, 2005.

[7] J. Sanders and E. Kandrot, *CUDA by Example: An IntroductIon to General-Purpose GPU Programming*, 1st ed. Addison-Wesley, 2010.

[8] B. K. H. Berkin Bilgic and I. Masaki, "Efficient integral image computation on the gpu," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*, June 2010.

[9] J. M. a. M. Zdenek Kalal, "P-n learning: Bootstrapping binary classiers by structural constraints," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, San Francisco, June 2010.

[10]NVIDIA, "Nvidias next generation cuda compute architecture," no. 22.

[11]nvidia corporation, "Cuda c best practices guide," 2011.

[12]nvidia, "Cuda c programming guide," 2011.

[13]C. Tomasi and T. Kanade, "Detection and tracking of point features," in *Carnegie Mellon University Technical Report*, April 1991.

[14]J.-Y. Bouguet, "Pyramidal implementation of the lucas kanade feature tracker description of the algorithm," in *Intel Corporation, Microprocessor Research Labs*.

[15]J. Shi and C. Tomasi, "Good features to track," in *IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.

[16]K. M. Z. Kalal and J. Matas, "Forward-backward error: Automatic detection of tracking failures," in *ICPR*, Istanbul, August 2010.

[17]A. Z. Anna Bosch and X. Munoz, "Image classification using random forests and ferns," in *ICCV*, October 2007.

[18]V. L. Mustafa zuysal, Michael Calonder and P. Fua, "Fast keypoint recognition using random ferns," in *(EPFL) Computer Vision Laboratory, I and C Faculty*, Switzerland.