International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 7 (2015) pp. 032-040
© MIR Labs, www.mirlabs.net/ijcisim/index.html

Optimization of countour based template matching using GPGPU based hexagonal framework

Mayank Bhagya, Sanjay Tripathi
Bachelor of Technology, Department of CSE
NITK Surathkal
Karnataka, India
mayankbhagya@gmail.com, sanjay.1506@gmail.com

P. Santhi Thilagam
Associate Professor, Department of COE
NITK Surathkal
Karnataka, India
santhisocrates@gmail.com

Abstract—**This paper presents a technique to optimize contour based template matching by using General Purpose computation on Graphics Processing Units (GPGPU). Contour based template matching requires edge detection and searching for presence of a template in an entire image, real time implementation of which is not trivial. Using the proposed solution, we could achieve an implementation fast enough to process a standard video (640 x 480) in real time with sufficient accuracy.**

Keywords-computer vision; image edge detection; image recognition; image sampling

I. Introduction

Template matching refers to identifying parts of an image that appear similar to a given template. This entails comparing all pixels of the template at all possible template locations of the image. This turns out to be very inefficient. Hence heuristics are used to optimize template matching.

These heuristics involve the use of image features like contours, blobs, corners, ridges, valleys et cetera to classify areas of the image as useful or not useful for full-fledged template match. Of all these heuristics, contours are most widely used because of the inherent nature of multiple objects to form edges when kept together in a scene. Other heuristics such as corners, blobs and ridges are characteristics of only a few kinds of images.

Contour based template matching hence is a process of detecting the edges in a template and looking for similar edge patterns in input images. Standard edge detection and template match routines are unsuitable for real time applications like automated navigation systems, content based video search et cetera.

This paper describes a technique to process input frames in real time using Graphics Processing Units (GPUs). Also, it suggests the use of hexagonal framework to improve the accuracy of edge detection and hence the template matches.

II. Proposed Solution

Most optimizations in template match have been by reduction in size of the input image. However, reducing the size also has severe effects on the quality of results. Hence, we propose the use of hexagonal framework, which reduces the number of pixels but with an increase in accuracy of edge detection. Further, the processing of input image and the template are offloaded to a GPU instead of a CPU for a real time implementation.

A. *Hexagonal framework*

Hexagonal framework samples the image on a hexagonal grid. Hence each pixel is hexagonal in shape. Changing the shape of the pixel affects all the stages of image processing: acquisition, addressing and display.

For producing such images, one needs special hardware with sensors, which are a grid of hexagons rather than squares or rectangles. Such hardware isn't easily available. Hence for processing regular images using hexagonal framework they should be resampled on to the hexagonal sampling grid by mathematical operations.

Fig. 1. A regular image when tiled on a hexagonal grid

Fig. 1 shows how a regular image can be resampled on a hexagonal grid. Resampling thus involves computation of intensities of each of the hexagon pixels.

Sampling to a hexagonal grid has various advantages. Hexagons have three characteristics that make them a better choice for sampling lattice than squares or rectangles. Hexagons are isoperimetric, which implies that the sampling density is highest. Unlike a square, all neighbors of a hexagon are equidistant and are of only one type (edge

connected). This ensures better detection of curves and hence better performance in morphological operations.

Another fundamental concern when using a hexagonal grid is the data structure that should be used for storing a hexagonal image in the memory.

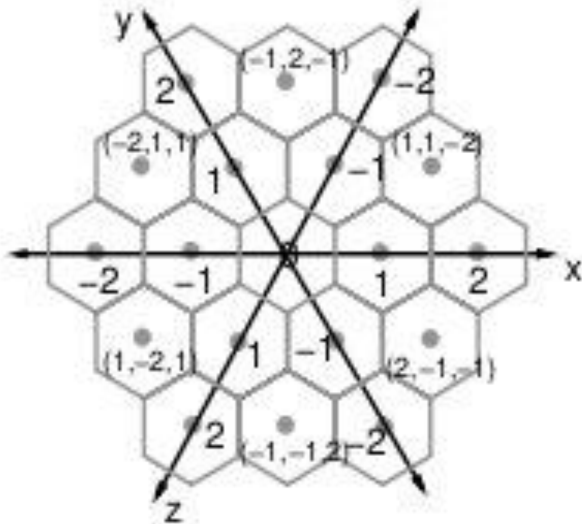

Fig. 2(a). Three dimensional addressing

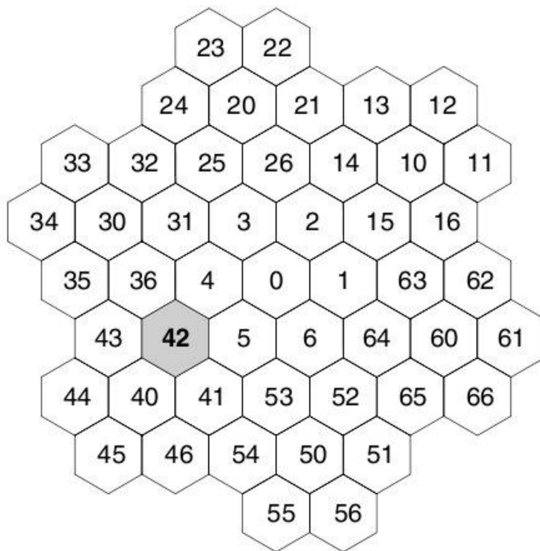

Fig. 2(b). Layered addressing

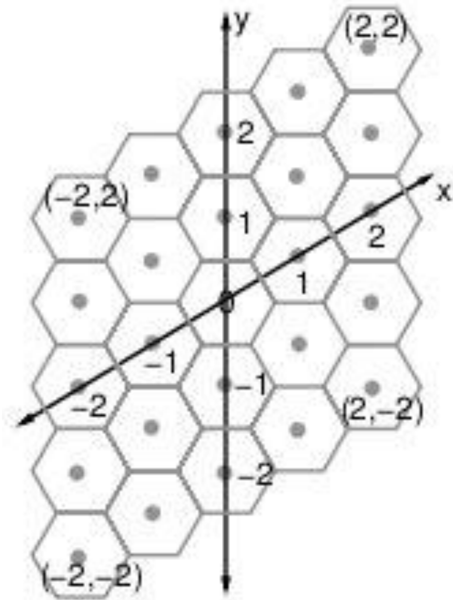

Fig. 2(c). Two dimensional (skewed) addressing

Fig. 2(a) shows three-dimensional addressing. Such an addressing scheme requires more space to store data than required in other addressing techniques. It requires that coordinates along each of the three axes be stored and hence there is a need of three dimensional data structures. Such data structures are sparsely filled and waste a lot of memory. Fig. 2(b) shows layered addressing but calculating neighbor pixels or accessing any pixel is complex in this scheme. Hence all data retrieval operations become time consuming. Fig. 2(c) however shows skewed addressing scheme which can be implemented using a two dimensional image with a little wastage of space. Also, accessing neighbor pixels has a worst case time complexity bounded by O(1).

Hence we chose skewed addressing as a choice of addressing scheme and the data structure thus required was a two dimensional array.

B. Graphics Processing Units

Let us take a look at the architecture of a GPU and what it offers for parallel programmers. A GPU is massively parallel because it is meant to perform graphics operations. Graphics operations involve similar processing on a lot of pixels, and hence inherently support single instruction multiple data parallelism.

Fig. 3 depicts the flow that programmers have to follow to exploit its parallelism.

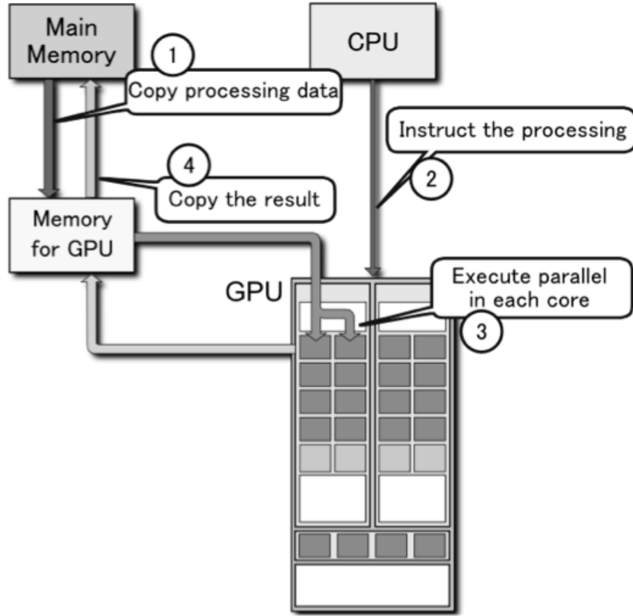

Fig. 3. Programming a Graphics Processing Unit

A programmer has to first offload all its data to the memory of GPU. Next the GPU has to be told to start executing desired instructions on all the cores. Once the processing is finished, the results can then be copied back to RAM.

There are both, vendor specific and vendor independent APIs available to access the GPU. However, we chose an open platform called Open Computing Language (OpenCL).

The OpenCL API offers a programmer to write 'kernels'. These kernels are functions which run on each of the graphics processors in parallel with the only difference being an 'id' which a programmer retrieves using a get_global_id () routine.

The programmer has to specify the number of threads using the OpenCL programming model. He may use the logical hierarchy of work-groups and work-items to specify total number of threads. Fig. 4. shows the same.

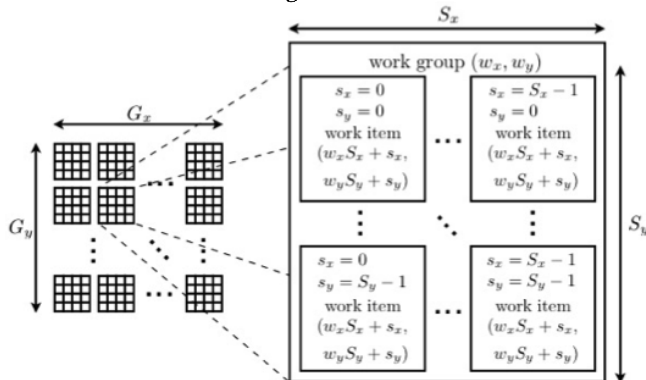

Fig. 4. OpenCL programming model

C. *Proposed Pipeline*

Thus we propose a pipeline as shown in Fig. 5 for template match operations.

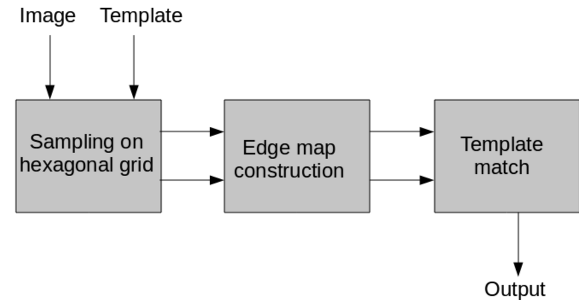

Fig. 5. Proposed pipeline for template matching

The pipeline has various stages, each of which should be implemented on the GPU. An input image and a template are fed into this pipeline. Both the images are then resampled on the hexagonal grid. Next, edge maps of both, the image and the template, are constructed. Once the edges are obtained, the two edge maps are compared against each other for a template match. The template match operation returns the coordinate of the input image where best match of the template is located.

III. DETAILS

The core pipeline thus uses three kinds of operations: Resampling the input image on hexagonal grid, edge detection and template match. Let us take a look at each of the operations in detail.

A. *Hexagonal resampling*

To perform such a resampling, we need to compute the intensity of each hexagonal pixel. For that, we need the relationship between a hexagonal pixel and a rectangular pixel. If we have a relationship between the hexagonal pixel and a rectangular pixel, we can compute the intensity of each hexagonal pixel using the rectangular pixels in the image.

If we observe carefully, the hexagonal grid has a three way symmetry which is analogous to the two way symmetry of a Euclidian plane. If x and y are the two axes of symmetry in a Euclidian plane, let i, j and k be the axes of symmetry of the hexagonal plane. A relationship between the two can be derived seen in Fig. 6.

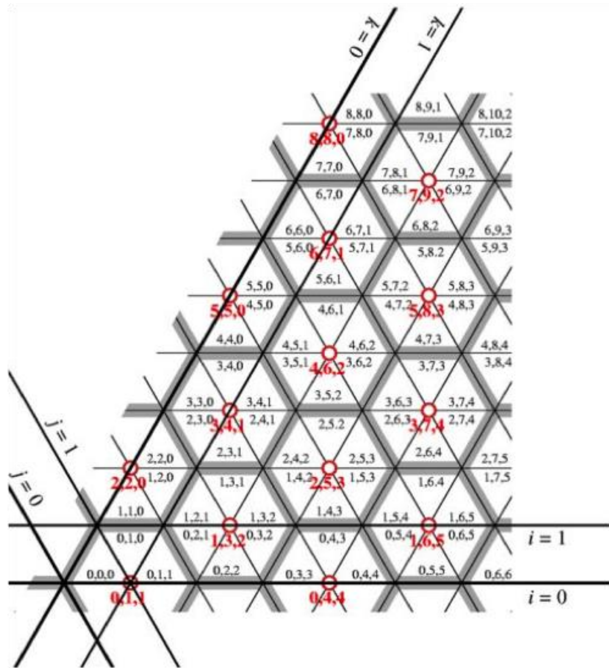

Fig. 6. Hexagonal grid on the Euclidian plane

The conversion of *i*, *j* and *k* to skewed coordinates *p* and *q* is straight forward (as shown in Fig. 7).

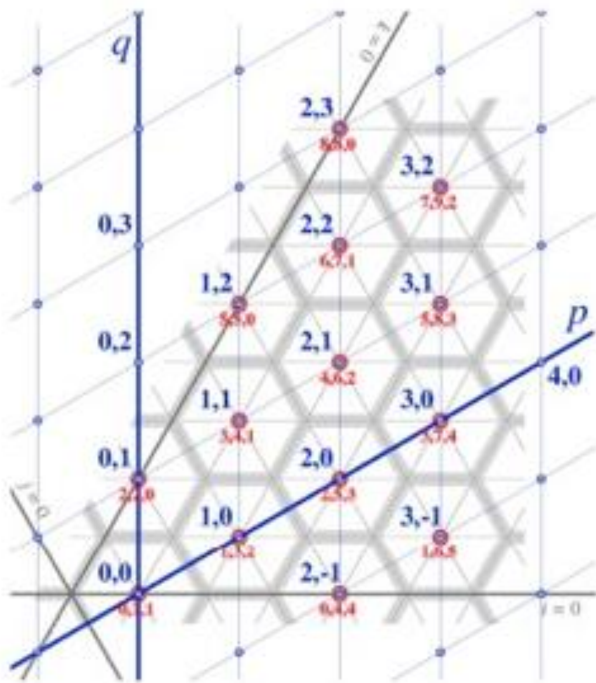

Fig. 7. Relationship between skewed coordinates and i, j and k

It is governed by the following equations:

$$i = p + 2q$$
$$j = 2p + q + 1$$
$$k = p - q + 1$$

Thus, the regular coordinates of Euclidian plane are related to *i*, *j* and *k* as:

$$y = i$$
$$x = \frac{(j + k)}{\sqrt{3}}$$

The above relationship leads to one hexagon having color components from six rectangular pixels. However, only four major contributing pixels are considered and linearly interpolated. Fig. 8 depicts two cases of how one hexagon can be mapped to rectangular pixels.

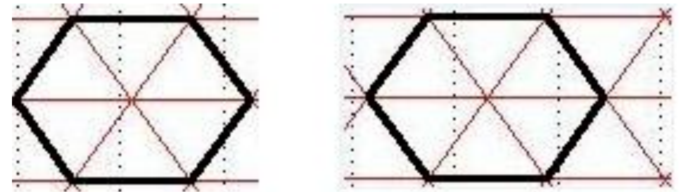

Fig. 8. Two possible mapping positions of hexagon with respect to rectangular grid

In the image displayed on the left in Fig. 8, the four major contributing pixels are clearly visible. However, in the image displayed on the right in Fig. 8, the four pixels are not so clear but can be computed by position of the center of hexagon with respect to the rectangular pixels.

Thus the conversion from rectangular to hexagonal sampling grid can be described as a step by step procedure:

1. Compute dimensions of the output hexagonal plane
2. For each pixel on the output plane:
   2.1 Determine corresponding four input pixels
   2.2 Linearly interpolate to obtain the color of the input pixel

B. Edge detection

An edge, in an image, is defined as a point where the intensity changes sharply. They can be computed by computing the magnitude of gradient at each point. This further is achieved by convolving the image with a suitable operator. One of the trivial operators used in detection of edges is Prewitt:

$$P_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} \quad P_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ -1 & +1 & +1 \end{bmatrix}$$

Convolution at a point is achieved by cross correlation. Hence at a point I_{11} in the following image matrix:

$$\begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}$$

the result R of cross correlation with Prewitt operator would be determined by:

$$R_x = (-I_{00}) + (I_{02}) + (-I_{10}) + (I_{12}) + (-I_{20}) + (I_{22})$$
$$R_y = (-I_{00}) + (I_{02}) + (-I_{10}) + (I_{12}) + (-I_{20}) + (I_{22})$$
$$R = \sqrt{R_x^2 + R_y^2}$$

However, in the hexagonal scenario, the Prewitt operator is given by:

$$H_1 = \begin{vmatrix} & -1 & & -1 & \\ 0 & & 0 & & 0 \\ & +1 & & +1 & \end{vmatrix} \quad H_2 = \begin{vmatrix} & 0 & & -1 & \\ -1 & & 0 & & +1 \\ & -1 & & 0 & \end{vmatrix} \quad H_3 = \begin{vmatrix} & -1 & & 0 & \\ -1 & & 0 & & +1 \\ & 0 & & -1 & \end{vmatrix}$$

As it can be observed, that $H_1 = H_2 – H_3$, the computation is reduced and only two convolutions need to be performed. H_2 and H_3 can be used to compute the horizontal gradient. We then need to find the neighbors of each point (p, q) so that convolution can be performed. If we observe, the choice of our data structure facilitates this and the coordinates of neighbors will be given by:

$$I_0 = (p + 1, q)$$
$$I_1 = (p - 1, q)$$
$$I_2 = (p - 1, q + 1)$$
$$I_3 = (p, q + 1)$$
$$I_4 = (p, q - 1)$$
$$I_5 = (p + 1, q - 1)$$

The value of convolution at (p, q) with H_2 and H_3 is thus given by:

$$R_2 = (+1 * I_1) + (-1 * I_2) + (+1 * I_4) + (-1 * I_5)$$
$$R_3 = (-1 * I_0) + (-1 * I_2) + (+1 * I_4) + (+1 * I_6)$$
$$R_1 = R_2 – R_3$$

The resultant R is the vector sum of R_1, R_2 and R_3. The value R is then subjected to a threshold to obtain desired result.

To summarize, here is the pseudo code for edge detection:

1. For an image of (m+2) x (n+2), allocate an output image of m x n
2. For each point on the output image:
   2.1 Calculate convolutions R_2 and R_3
   2.2 Compute R_1 using R_2 and R_3
   2.3 Find resultant vector sum of R_1, R_2 and R_3
   2.4 Threshold to a binary value

C. *Template matching*

Template matching is performed on the outputs of the second stage in the pipeline.

Once the contours of the template and the image are ready, the template is cross-correlated at all possible locations of the image. The position of best match correlates to the maximum extent. Thus, when normalized over a range of [0, 255], we get a grayscale map with the brightest point indicating the point of best match. Fig. 9 shows an illustration of template match.

Fig. 9. An image, a template and the grayscale output of template match

The cross correlation between the image and the template is measured as a score at each point obtained by taking ratio of matched pixels at that point to total template pixels.

So, the algorithm for template match can be described as:

1. For an image of size (m x n) and template of size (p x q):
2. Allocate an output image of (m – p + 1) x (n – q + 1)
3. For each point on the output image:
   3.1 Compute template correlation with image at that point
   3.2 Normalize the correlation value to [0, 255]
4. Return brightest point's location and value

IV. PARALLELING THE PIPELINE ON GPU

Now that we've seen the serial implementation of the pipeline, we'll see how to implement this on a GPU. While implementing functions on GPU we need to keep in mind that data has to be transferred to the GPU memory before any kind of processing can be done. Hence, while developing modules, we must not copy data back to RAM at the end of each pipeline stage (unless we're debugging and want to see the sample results).

Since all the operations of our pipeline are image transformations, these are embarrassingly parallel and can be paralleled on the GPU according to one thread per output pixel basis.

A. *Hexagonal resampling*

In resampling of image on the hexagonal grid, the CPU calculates the dimensions of the output image and invokes one thread per output pixel on the GPU. The thread in turn computes location of four corresponding hexagonal pixels and linearly interpolates their intensities. Once done, the output is stored in the GPU memory. The output of this stage of the pipeline is a hexagonal image.

B. *Edge detection*

Since edge detection is performed by convolution of Hexagonal Prewitt operator at each of the pixel locations of the image, one output thread is made to perform one convolution and threshold it to produce a binary image.

Thus the output of this stage is a binary image containing the edge map of the hexagonal image.

C. Template matching

The final stage of the pipeline also has a fixed size output. The output represents the output of cross-correlation of the template with each of the input image pixels. Hence each thread of the GPU is responsible for cross-correlated with the template and thus generating one pixel of the output image.

V. RESULTS

A. Example

Fig. 10 displays the sample image and a template fed into the pipeline. Note that the template is a general image of a car and does not belong to the input image.

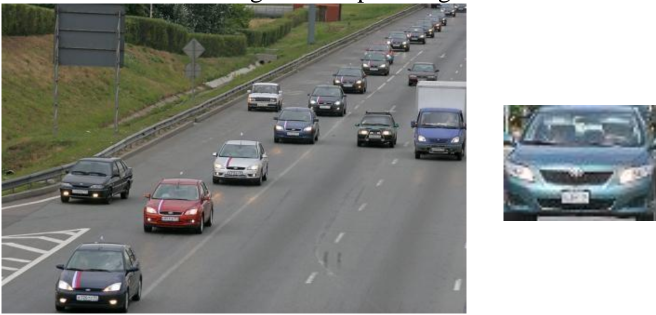

Fig. 10. Sample image and sample template

Fig. 11 displays hexagonally sampled image and template projected on a rectangular grid. These images appear skewed because of the mismatch in coordinate systems. These images are meant to be hexagonal images and requires displays / paper with hexagonal pixels. However, since we are trying to view this on a screen / paper with square pixels, the image appears skewed.

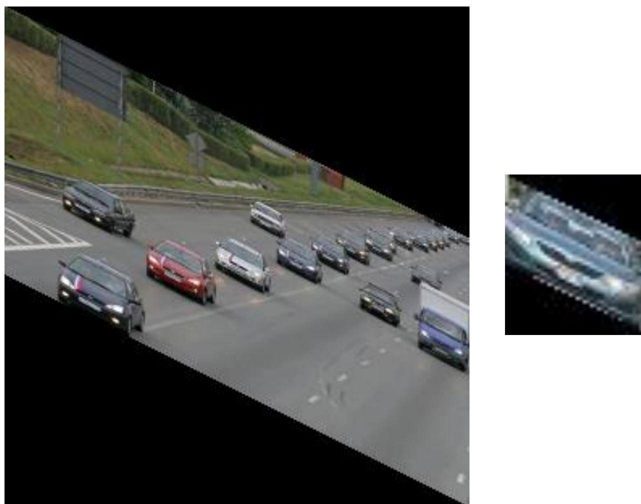

Fig. 11. Hexagonally sampled images

Fig. 12 shows convolution of images shown in Fig. 8 and corresponding edge maps. Notice how each of the edges in the original image turn white in this image.

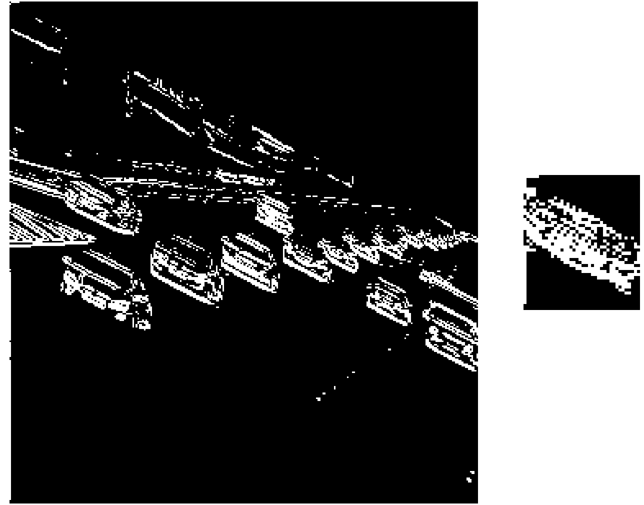

Fig. 12. Edge detection of the image and template shown in Fig. 9.

Fig. 13 shows the final result of the normalized template match. The areas where correlation scores are good turn white and the areas where the score is bad, remain black. The higher the cross-correlation score, the brighter (close to white) the pixel in the output image. As it can be seen, there has been a good correlation of the template in the areas where there were cars in the original image.

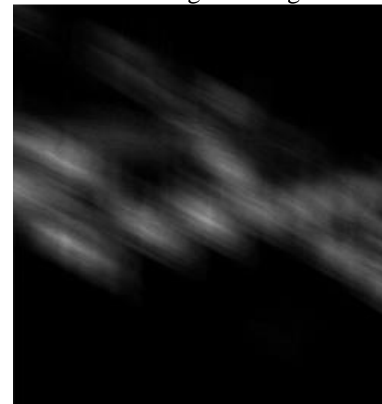

Fig. 13. Template match result. Brightness indicates good correlation.

B. Implementation details

Table I gives the details of the system that was used for benchmarking. We used a system with Intel Pentium 4 processor clocked at 3.0 GHz and having 2 GB DDR2 memory. The system had a GPU by NVIDIA GeForce GTX465 running at 1.2GHz with 1GB DDR5 memory.

TABLE I

SPECIFICATIONS OF SYSTEM USED FOR BENCHMARKING

CPU	
Vendor	Intel
Model	Pentium 4
Memory	2 GB DDR2
GPU	
Vendor	NVIDIA
Model	GeForce GTX 465
Memory	1 GB GDDR5

We used the C programming language to implement the pipeline. The video input / output was taken care by OpenCV 2.2 library. We used NVIDIA's implementation of the OpenCL library to write kernels that ran on the GPU.

C. *Performance analysis*

We used the pipeline to build a content based video search application. The application, as the name suggests, searches for a template in a given video and displays the results in real time. It highlights the areas inside the video wherever a match is found.

Performance was analyzed in terms of two parameters: speed and quality.

Fig. 14 shows a comparison between GPU time and CPU time, taken to process a frame by the pipeline of varying size. The template size is kept constant at 100 x 100 pixels and the image dimensions are varied.

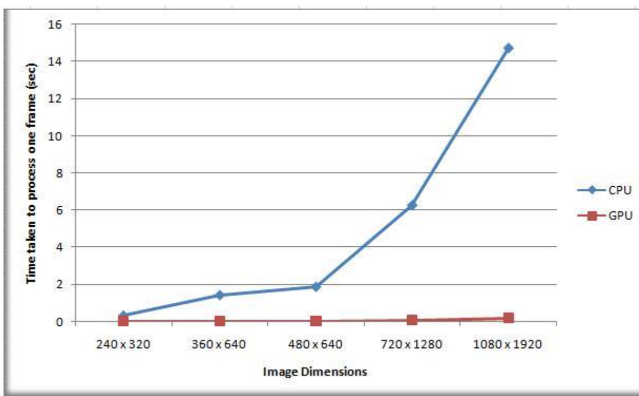

Fig. 14. Time taken by CPU and GPU vs Image dimensions

As it can be seen, the time taken by the CPU rises exponentially whereas the time taken by the GPU remains almost constant as the image size scales.

Fig. 12 shows a comparison between GPU and CPU time, taken to process a frame of 480x640 pixels with varying template sizes.

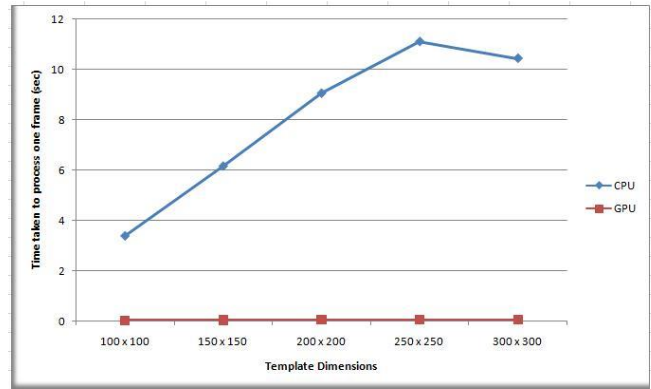

Fig. 15. Time taken by CPU and GPU vs Template dimensions

The results are similar to the ones shown in Fig. 14. When the template size is increased, the time taken by the CPU increases exponentially whereas the time taken by the GPU remain almost constant. It is independent of the template size.

Fig. 13 shows a ratio of time taken by the CPU and by the GPU for a fixed template of size 100 x 100 pixels and frames of varying size.

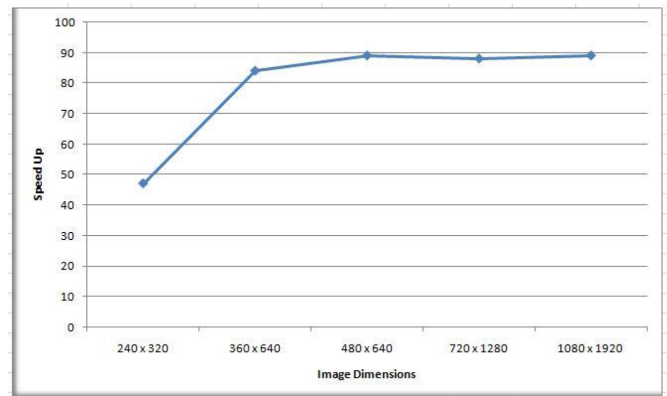

Fig. 16. Ratio of time taken by CPU and GPU vs Image dimensions

For smaller images, the gain by using a GPU based pipeline is not as significant as with larger images. After crossing a certain image size, the speedup remains almost constant. As it can be seen, full HD videos can also gain up to 90 times with a GPU based pipeline.

Fig. 14 shows a quality metric. It shows how well the algorithm works for different video qualities. It is done by blurring the input video at multiple levels and then performing template match.

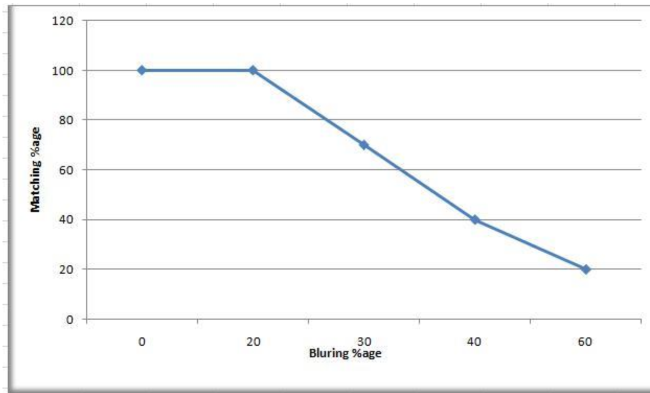

Fig. 17. Image blurring percentage vs matching percentage

The blur applied to each image is a Gaussian blur. As long as the images do not have a high blur (say > 25%), the template match pipeline was able to find the template correctly.

Table II shows how the template match algorithm behaves when subjected to images of different edge density. Different edge intensities have been simulated by varying the edge detection threshold of the images. Thus, images with a low threshold behave similar to images with very high edge density. And images with a high threshold have a very few edges.

TABLE II
TEMPLATE MATCH RESULTS FOR IMAGES WITH DIFFERENT EDGE CHARACTERISTICS

Image Threshold	Result
500	Not Matched
1000	Not Matched
1500	Matched
2000	Matched
2500	Matched
3000	Matched
3500	Matched
4000	Matched
4500	Matched
5000	Matched
5500	Matched
6000	Matched
6500	Not Matched
7000	Not Matched
7500	Not Matched

Thus images with very high edges and very few edges are not matched accurately. A similar behavior is observed with a constant image and varying template qualities.

TABLE III
TEMPLATE MATCH RESULTS FOR TEMPLATES WITH DIFFERENT EDGE CHARACTERISTICS

Template Threshold	Result
500	Not Matched
1000	Not Matched
1500	Not Matched
2000	Matched
2500	Matched
3000	Matched
3500	Matched
4000	Matched
4500	Matched
5000	Matched
5500	Matched
6000	Matched
6500	Matched
7000	Not matched
7500	Not matched

The template behavior is similar to the image behavior. Thus if the templates have too many edges or too few edges, the pipeline will fail to identify the template in a given set of images or video frames.

VI. Conclusion and Future Scope

The project is aimed towards optimizing contour based template matching. By the proposed pipeline, we have been able to implement a hexagonal framework on a GPU. Hexagonal framework reduces amount of data to be processed and thus offers performance gain without loss of accuracy. Using the GPU's parallelism, we have been able to search for template in frames in less than 10 ms (100 frames per second). The quality of template match suffices if both the template and the image have moderate edge characteristics.

Since processing time for a frame can go up to 33 ms for a frame (keeping in mind the real time limit of 30 frames per second), there is a scope for utilization of the remaining time and GPU computing power. It can be used for considering other attributes such as scaling and rotation of templates on the runtime. For instance, given a template, the GPU should be able to compute the edge map of the template, rotated at various angles. And templates at all the rotations can then be searched for in the input images or video frames.

Acknowledgment

We'd like to thank the Department of Computer Science and Engineering, NITK Surathkal for providing us an opportunity and the infrastructure that was required to accomplish this project.

REFERENCES

[1] R Brunelli, "Template matching and testing" in *Template Matching Techniques in Computer Vision: Theory and Practice*, West Sussex, U.K.: Wiley, 2009.

[2] L. Middleton et al., *Hexagonal Image Processing: A Practical Approach*., U.S.A.: Springer, 2005.

[3] Gonzalez et al., *Digital Image Processing*, vol. 2, New Jersey: Prentice Hall, 2002.

[4] Vidya et al., "*Performance Analysis of Edge Detection Methods on Hexagonal Sampling Grid,*" Dept. ECE, Amrita Vishwa Vidya Peetham, Coimbatore, 2009.

[5] Lee Middleton et al, "*Edge Detection in a Hexagonal-image Processing Framework*", Dept. Elect. Eng., Univ. of Auckland, Auckland, 2004.

[6] P.J.H.M. Boots, "*Object Recognition by Contour Matching*", 2002.

[7] Chia-Yen Chen, "*Image Stitching – Comparison and New Techniques*", University of Auckland, 1998.

[8] J. P. Lewis, "*Fast Normalized Cross Correlation*", 1995.

[9] Timothy Poston, "*Hexagonal Cell Management*", unpublished.

[10] Longin Jan Latecki, "*Template Matching*". Temple University.

[11] Dmitrij Csetverikov, "*Basic Algorithms for Digital Image Analysis: A course*", Eotvos Lorand University.

[12] NVIDIA, "*OpenCL programming guide*", v3.1.

[13] Khronos OpenCL working group, "*The OpenCL specification*", v1.1.

[14] Victor Podlozhnyuk, "*Image Convolution on CUDA*", NVIDIA.