Speeding Up Camera Sabotage Detection on CUDA

Püren GÜLER, Deniz EMEKSİZ, Alptekin TEMİZEL
Graduate School of Informatics
Middle East Technical University, Ankara, Turkey
e170986@metu.edu.tr, e171000@metu.edu.tr, atemizel@ii.metu.edu.tr

Abstract

Camera Sabotage Detection (CSD) algorithms, namely Camera Moved Detection, Camera Out of Focus Detection and Camera Covered Detection, are used to detect tampering attempts on surveillance cameras in real-time. CSD algorithms are required to be run on a high number of cameras, bringing high computational load to the video analytics systems. Importance of speeding up of these algorithms are two fold:

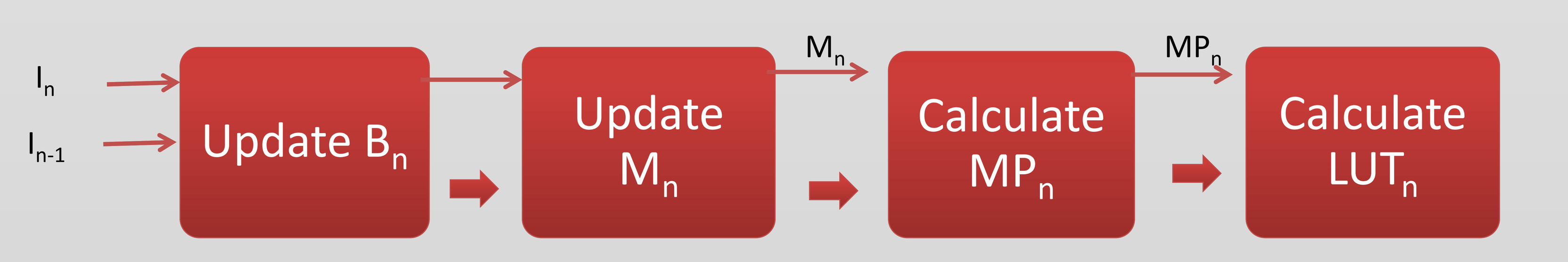
- **Enabling operation on all cameras and hence reducing security lapses ,**
- **Leaving valuable computational power to other video analytics algorithms such as object tracking and activity analysis.**

In this work, the CSD algorithms that were previously developed by our group [1] are accelerated by using parallelization methods in CUDA. While different algorithms have different speed-up rates, the overall system test results show that parallelization in GPU makes the system 18 times faster than its CPU counterpart and up to 400 cameras can be supported in real time on a GTX 470.

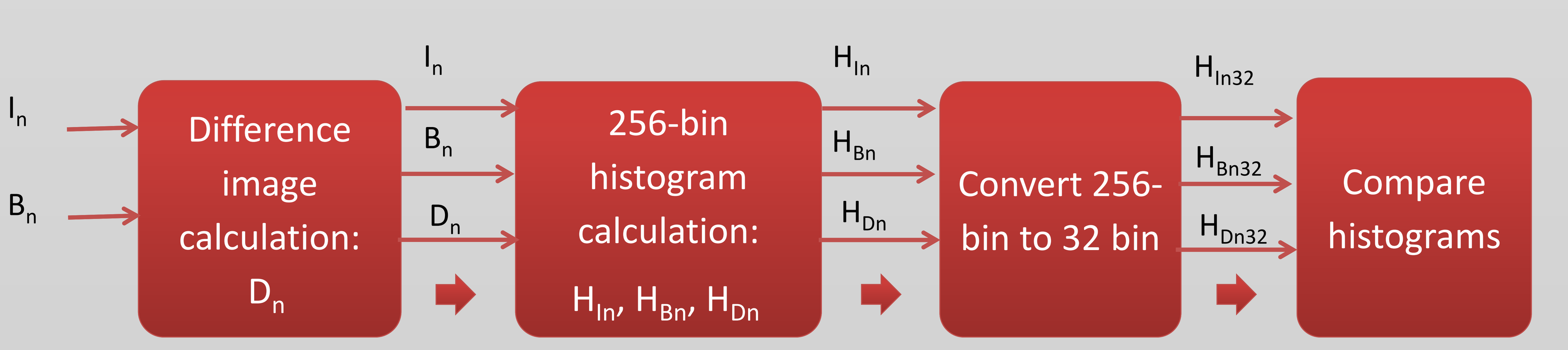
General Flow of The Algorithms

I_n : Current frame
I_{n-1}: Previous frame
B_n: Current background frame
B_{n-20} :20 frames prior to current background frame
M_n: Image that keeps track of changing pixels
MP: Moving pixel map
LUT_n: Lookup table matrix showing moving blocks
D_n : Difference image
H_{In} : 256 bin histogram of current image
H_{Bn} : 256 bin histogram of background
H_{Dn} : 256 bin histogram of difference image
H_{In32} : 32 bin histogram of current image
H_{Bn32} : 32 bin histogram of background
H_{Dn32} : 32 bin histogram of difference image

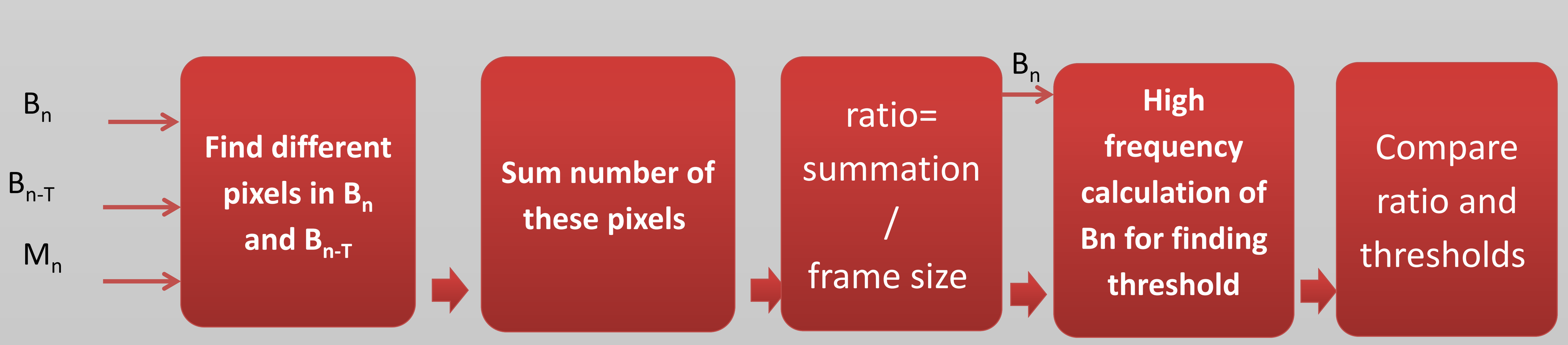
Background Subtraction



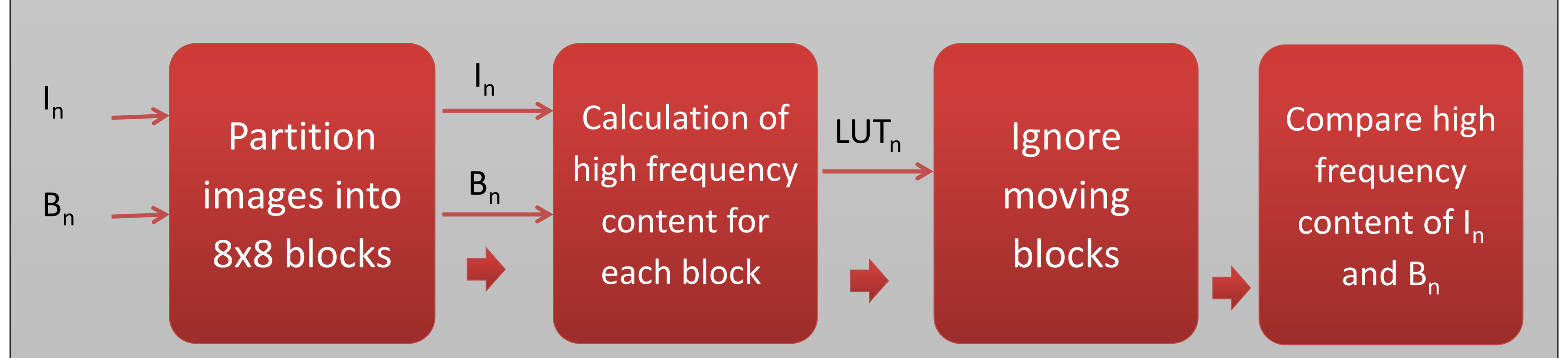
Camera Covered Detection



Camera Moved Detection



Camera Out of Focus Detection



Background Estimation

To integrate background subtraction to the system, CUDA code previously developed by our Virtual Reality and Computer Vision (VRCV) group is used [2].

- Pixels are reached as integers rather than char to optimize memory access. So each four pixels are processed in one thread, since four pixels(char) create an integer [2].
- Background frame is created as a 2-D frame, since for DCT implementation of CUDA used in Camera Moved Detection, 2-D arrays are needed as inputs.

For 1-D:

```
i = blockDim.x * blockIdx.x + threadIdx.x;
```

For 2-D:

```
actualImg_row =  i/(frame_width/4);
actualImg_col = (i%(frame_width/4))*4;
index_bn = actualImg_row*bn_stride_int+(i%(frame_width/4));
```

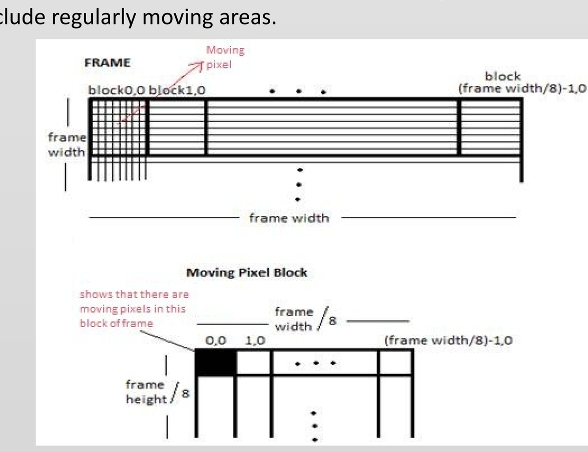
Detect Regularly Changing Pixels

- The difference from the background subtraction algorithm in [2] is detection of regularly changing pixels.

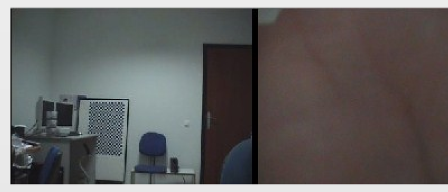
$$M_n(x,y) = \begin{cases} M_n(x,y) + \beta|I_n(x,y) - B_n(x,y)|, if\ (x,y) is\ moving \\ M_n(x,y) - \gamma|I_n(x,y) - B_n(x,y) + 1|, if\ (x,y) is\ non\ moving \end{cases}$$

Detect Moving Pixel Blocks

- Moving blocks are 8x8 blocks that contain at least one moving pixel. Detection of these blocks is necessary in to exclude regularly moving areas.

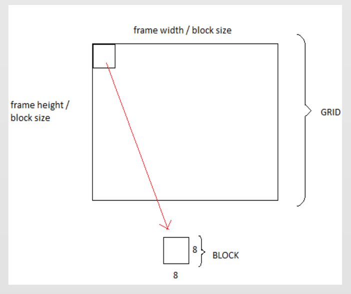


Camera Covered Detection



Difference Image Calculation

- Difference image is generated by subtracting background frame from the current frame and obtaining the absolute value of the result then the histograms of current, background and difference frames are checked to detect if the camera is covered.
- In CUDA version, each pixel is calculated in parallel.



Histogram Calculation

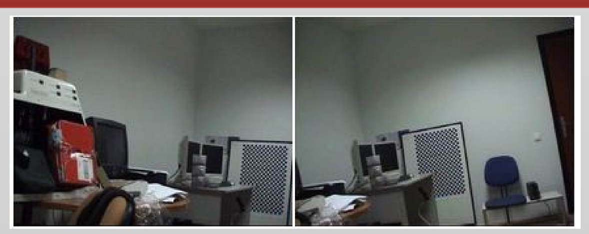
- For histogram calculation, the algorithm developed in our VRCV group is used [3].
- Histogram bins should be equal to the number of colors in the input image to get efficient results as atomicAdd() operation results in race condition.
- 256 bin histograms for current, background and difference frames are generated.

Conversion from 256 bin to 32 bin

- A parallel sum reduction technique similar to [4] is applied.
- While converting 256-bin to 32-bin, each 8 consecutive bins put in the same block, thus block size is 8.
- Histogram size is divided into block size and then into 2 for optimization processed described in [4].
- In the Fermi architecture, volatile memory should be used in order to write the values into shared memory instead of registers [5].

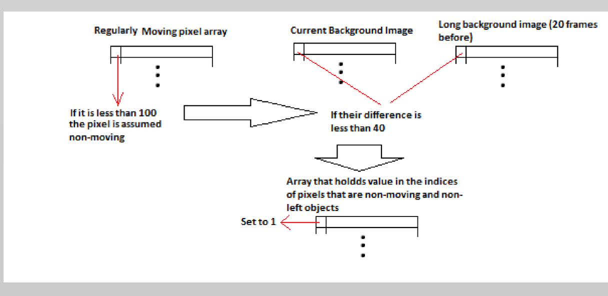
```
convert256To32Bin<<<(histogramSize/B/2),B>>>
(hist, hist_out, n)
{
  __shared__ sdata[B];
  //each thread loads one element from
  //global to shared mem
  tid = threadIdx.x;
  i = blockIdx.x*(blockDim.x*2) +
      threadIdx.x;
  gridSize = B*2*gridDim.x;

  sdata[tid] = 0;
  __syncthreads();

  while (i < n)
  {
    sdata[tid] += hist[i] + hist[i+B];
    i += gridSize;
  }
  __syncthreads();

  //since the block size for histogram does
  //not exceed 8 for a 256 bin histogram to
  //convert 32 bin
  if (tid < 8)
  {
    volatile int *smem = sdata;
    if (blocksize >= 8)
      smem[tid] += smem[tid + 4];
    if (blocksize >= 4)
      smem[tid] += smem[tid + 2];
    if (blocksize >= 2)
      smem[tid] += smem[tid + 1];
  }
  //write result for this block to global  mem
  if (tid == 0)
  {
     hist_out[blockIdx.x] = sdata[0];
  }
}
```

Camera Moved Detection



When a camera is moved to a different direction, the background image starts to be updated to reflect the changed view: A delayed background is also kept to detect such change.

- Each pixel is processed by one thread.
- Grid size is frame size/block size.
- Each pixel is checked and if it is regularly moving it is ignored. If non-moving , a proportion value is calculated by pixel on background to corresponding pixel on delayed background.



Sum Reduction

- Pixels that are different in current and delayed background should be counted in order to compute a proportion value.
- For sum reduction, the final optimized kernel in reduction implementation by CUDA [4] is used. The differences that are made in this work:
  - Adding the following condition to account for increased maximum number of threads/block :

```
if (blocksize >= 1024) {
    if (tid < 512)  sdata[tid] += sdata[tid + 512];
    __syncthreads();
}
```

  - Using volatile structure while doing operations in a single warp in order to write the values into shared memory instead of registers [5]. Because in Fermi architecture, without *__synchreads()*, volatile memory is required to synchronize threads in a warp.

Camera out of Focus Detection

DCT Calculation

- Discrete Cosine Transform (DCT) implementation of NVIDIA GPU Computing SDK 4.0 [6] is used instead of Fast Fourier Transform (FFT) used in the original algorithm.
- High pass filter kernel is modified accordingly.
- Frames are partitioned into 8x8 blocks to discard the regularly moving blocks to prevent false alarms.

A modified version of *WrapperCuda2* function of dct8x8 implementation of CUDA is used :

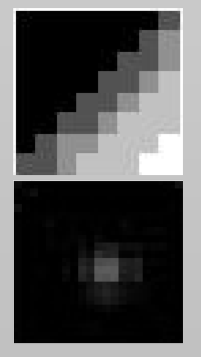
- Memory allocations are moved outside the function so that they are not performed for each frame.
- *CopyByte2Float* and *AddFloatPlane* C functions are moved into separate kernels. They are straightforward kernels that process each pixel in parallel.
- Instead of using *CUDAkernelQuantizationFloat* kernel, a new kernel is implemented that use Gaussian windowing function to have a similar effect as in [1] to filter out low frequencies:

CUDAkernelQuantizationFloat	Our kernel
int quantized = (int)((curCoef / curQuant)+0.5f); curCoef = (float)quantized * curQuant;	float quantized = curCoef * curQuant;

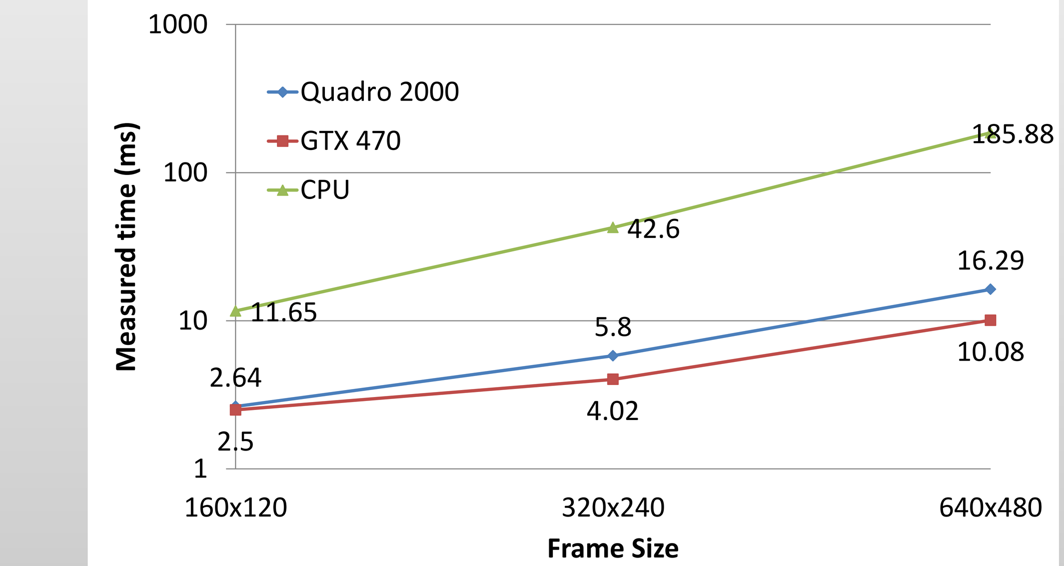
High Frequency Calculation

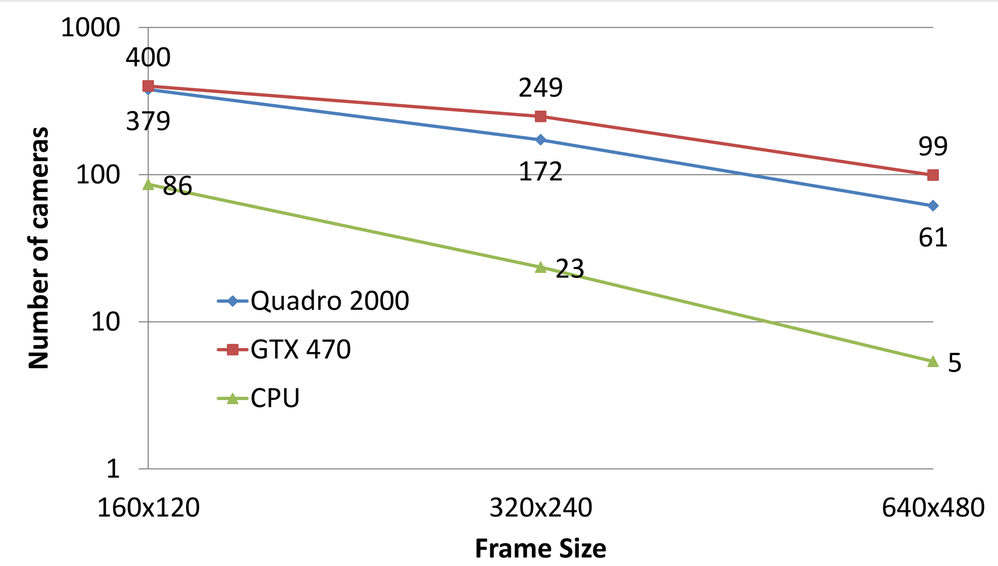
- High frequency data is summed inside the kernel.

```
 // Block index
 bx = blockIndex.x;by = blockIndex.y;
 row = by * B; col = bx * B;
 localSum=0;
// lut_mpb holds the moving block information so
that high frequency data in a block that has
moving pixel is excluded from the calculation
if(lut_mpb[by*strideLut+bx] != 1)
{
  for(i=row; i<row+B; i++)
  {
     for(j=col; j<col+B; j++)
     {
        localSum +=  frame_dct[i*strideFrame+j];
     }
  }
 }
 dct_sum[by*strideDctSum+bx] = localSum;
```



Results of the Overall System





Speedup

	160x120	320x240	640x480
Quadro 2000	4.41	7.34	11.41
GTX 470	4.66	10.60	18.44

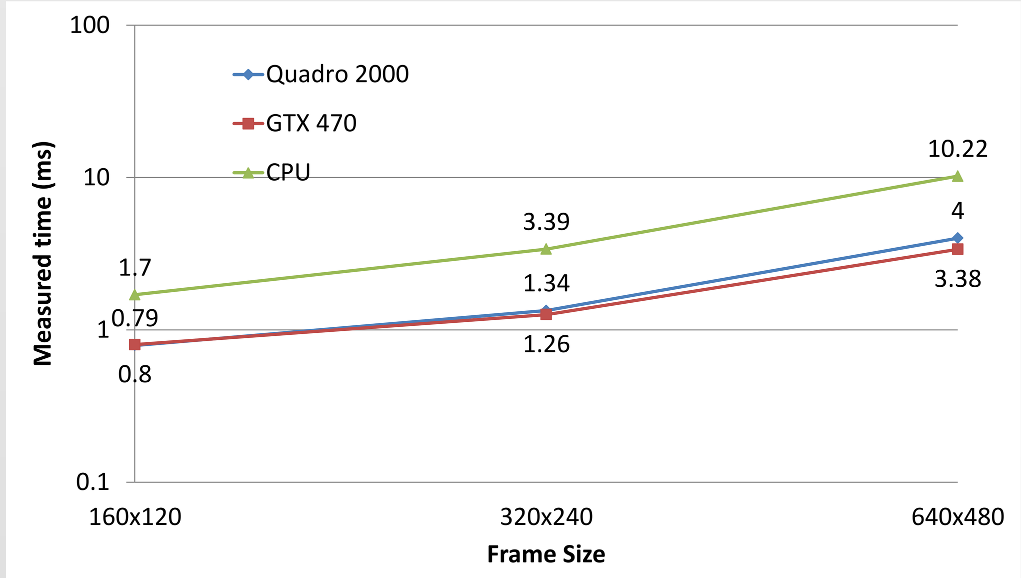
Number of cameras supported

	160x120	320x240	640x480
Quadro 2000	379	172	61
GTX 470	400	249	99
CPU	86	23	5

Results of Used Algorithms

Experiments are performed on a PC having Intel Core i7 CPU and 3.5 GB usable RAM. The GPU algorithms are tested with Quadro 2000 and NVIDIA GTX 470.

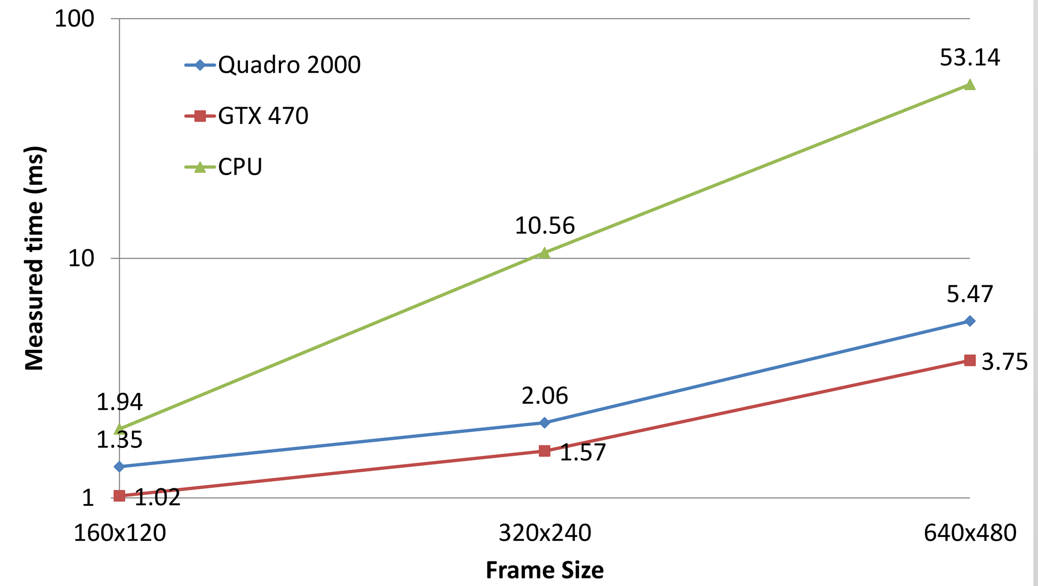
Camera Covered Detection



Speedup

	160x120	320x240	640x480
Quadro 2000	2.15	2.53	2.56
GTX 470	2.13	2.69	3.02

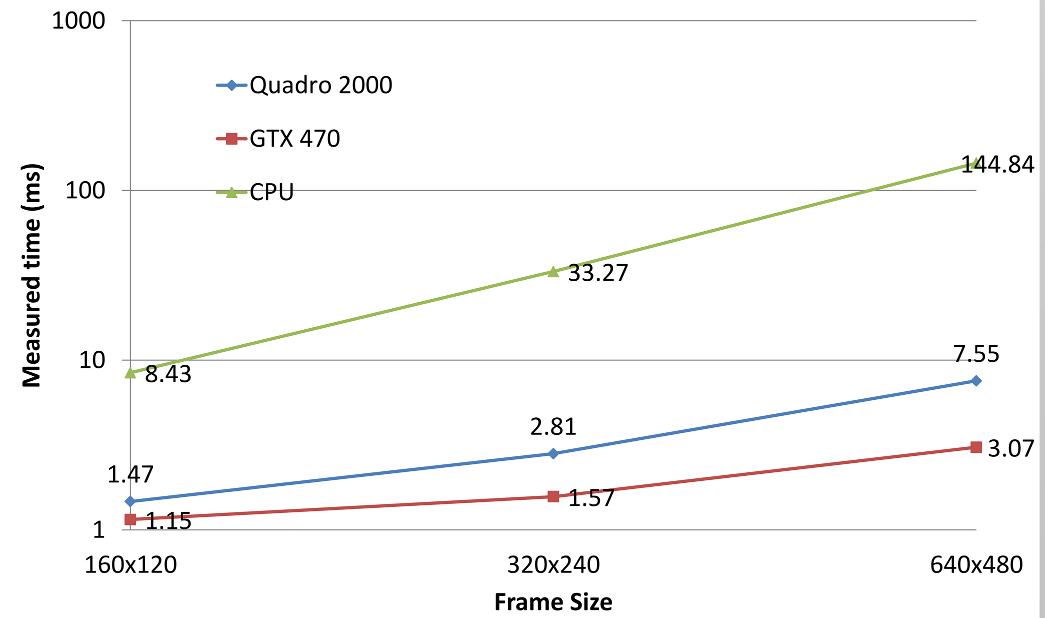
Camera Moved Detection



Speedup

	160x120	320x240	640x480
Quadro 2000	1.44	5.13	9.71
GTX	1.90	6.73	14.17

Camera Out of Focus Detection



Speedup

	160x120	320x240	640x480
Quadro 2000	5.73	11.84	19.18
GTX 470	7.33	21.19	47.18

References

[1] Saglam, A., & Temizel, A. (2009). Real-Time Adaptive Camera Tamper Detection for Video Surveillance. *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance AVSS '09*, (pp. 430-435).
[2] Temizel, A., Halıcı, T., Loğoğlu, B., Taşkaya Temizel, T., Ömrüuzun, F., & Karaman, E. (2011). Experiences on Image and Video Processing with CUDA and OpenCL. In *GPU Computing Gems 1, NVIDIA.* Elsevier.
[3] Teke, M. (2011). *Satellite Image Processing on GPU Technical Report.* Retrieved from http://www.ii.metu.edu.tr/coursewebsites/quda/mteke/Satellite_Image_Proc_GPU_Paper.pdf
[4] Harris, M. (2008). *Optimizing Parallel Reduction in CUDA.* Retrieved from http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf
[5] NVIDIA Corporation. (2010, August). Fermi Compatibility Guide for CUDA Applications. Retrieved from http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Compatibility_Guide.pdf
[6] Obukhov, A., & Kharlamov, A. (2008, October). *Discrete Cosine Transform for 8x8 Blocks with CUDA.* Retrieved from http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/dct8x8/doc/dct8x8.pdf

This research is funded by Ministry of Science, Industry and Technology SAN-TEZ program grant number 00542.STZ.2010-1.