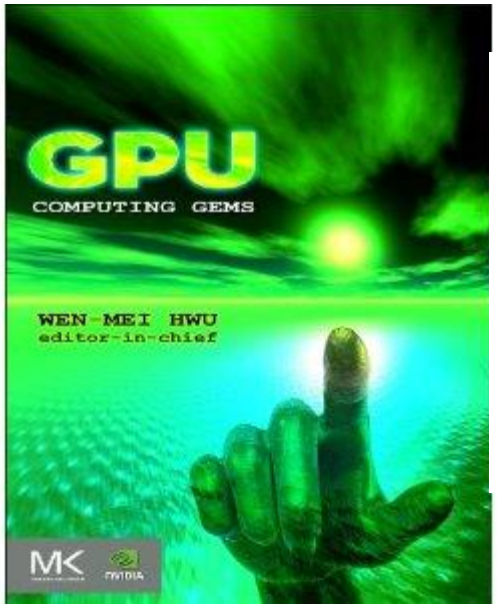
Case Study 1 – Background Subtraction

Alptekin Temizel

atemizel@metu.edu.tr

Reference Material

- Wen-mei W. Hwu (Editor), “GPU Computing Gems”, ISBN-10: 0123849888, ISBN-13: 978-0123849885.



CHAPTER

34

Experiences on Image and Video Processing with CUDA and OpenCL

Alptekin Temizel, Tugba Halici, Berker Logoglu, Tugba Taskaya Temizel, Fatih Omruuzun, Ersin Karaman

Adaptive Background Subtraction (1)

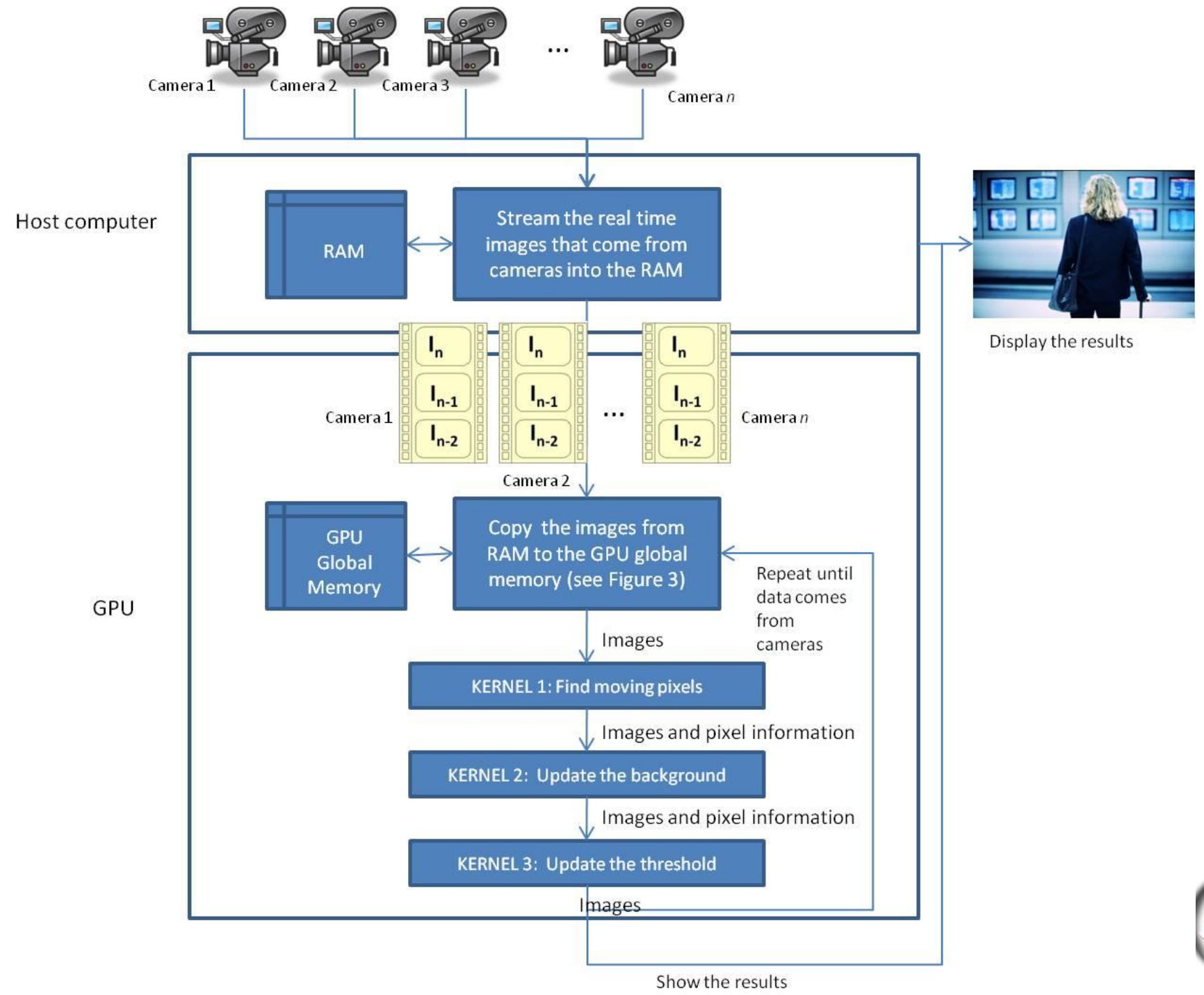
- Adaptive background subtraction algorithm can be used to identify moving or newly emerging objects from subsequent video frames.
- This information can then be used to enable tracking and deducting higher level information which is valuable for automated detection of suspicious events in real time.

Adaptive Background Subtraction (2)

- We selected the algorithm described in [1] because this method is easy to implement and cost effective and produces effective results.
- The algorithm is embarrassingly parallel in nature as the same independent operation is applied to each pixel in a frame.

[1] R. Collins, A. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y. Tsin, D. Tolliver, N. Enomoto, and O. Hasegawa, “A System for Video Surveillance and Monitoring: VSAM Final Report”, Technical Report CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, May, 2000.

Adaptive Background Subtraction (3)


Camera 1
Camera 2
Camera 3
...
Camera n
Host computer
RAM
Stream the real time images that come from cameras into the RAM
Display the results
Camera 1
Camera 2
Camera n
GPU
GPU Global Memory
Copy the images from RAM to the GPU global memory (see Figure 3)
Repeat until data comes from cameras
Images
KERNEL 1: Find moving pixels
Images and pixel information
KERNEL 2: Update the background
Images and pixel information
KERNEL 3: Update the threshold
Images
Show the results

Algorithm

- The algorithm consists of three main stages:
  - Finding Moving Pixels
  - Updating the Background B_n
  - Updating Thresholds T_n
- With each new image, moving pixels are found and Bn and Tn are updated according to following rules.

Finding Moving Pixels

$$|I_n(x,y) - I_{n-1}(x,y)| > T_n(x,y) \text{ and } |I_n(x,y) - I_{n-2}(x,y)| > T_n(x,y)$$

- Here, each pixel of the last two images are compared against the corresponding pixels of the current image.
- The algorithm keeps an adaptive threshold for each pixel and the pixel is said to be moving if the absolute difference is higher than this threshold.

Updating the Background

- Background is updated with each new image according to the following equation:

$$B_{n+1} = \begin{cases} \alpha B_n(x,y) + (1-\alpha) I_n(x,y) & \text{, if } (x,y) \text{ is not moving} \\ B_n(x,y) & \text{, if } (x,y) \text{ is moving} \end{cases}$$

- If the pixel is moving, background is updated using a time constant , otherwise it is kept the same.
- 0< α<1 determines how the new information updates older observations.
- An α value closer to 1 means that the older observations have higher weight and the background is updated more slowly.

Updating the Threshold

- Threshold is updated with each new image according to the following equation:

$$T_{n+1}(x,y) = \begin{cases} \alpha T_n(x,y) + (1-\alpha)(c|I_n(x,y) - B_n(x,y)|) & \text{, if } (x,y) \text{ is not moving} \\ T_n(x,y) & \text{, if } (x,y) \text{ is moving} \end{cases}$$

- Similar to the background update, if the pixel is not moving, threshold is updated using the time constant , otherwise it is kept the same.
- The threshold *c* is a real number greater than one, analogous to the local temporal standard deviation of the intensity.

Parameters, frameworks and architectures used in the experiments

Parameters		Values
CPU		Intel i7 920, 2.66 GHz, Windows 7 Professional (32 Bit)
GPUs	**GPU1**	NVIDIA Quadro FX 5800, 240 Cores, Processor Clock: 1.3 GHz Memory: (512bit), 4GB
	GPU2	NVIDIA GeForce GTX 285 , 240 cores, Processor Clock: 1.476 GHz Memory Clock: 1.242 GHz (512bit), 1 GB
	GPU3	NVIDIA GeForce 9800GT, 112 cores, Processor Clock: 1.5 GHz Memory Clock: 900 MHz (256bit), 1 GB
	GPU4	ATI HD 5750, 720 Stream Processing Units, Engine Clock: 700 MHz Memory Clock: 1.15 GHz, 1 GB
Programming Architectures		CPU applications implemented using Open MP 2.0, CUDA C/OpenCL driver version 197.13, C++ using Visual Studio 2008 CUDA SDK 3.0 ATI Catalyst Display Driver 10.5, v8.732.0.0 ATI Stream SDK 2.1 OpenCV 2.0

Implementation

- For the *single kernel case*, all three steps of the algorithm are integrated in a single kernel.
- For the *multi-kernel case*, three separate kernels comprised of individual steps, namely; finding moving pixels, updating the background and updating thresholds are used. These three kernels are then run successively.
- In the experiments, we use 4 different image sizes: 160x120, 320x240, 640x480 and 720x576.
- Number of threads and block size are set to 64 and 512, respectively.

Finding the moving pixels

$|I_n(x,y) - I_{n-1}(x,y)| > T_n(x,y)$ and $|I_n(x,y) - I_{n-2}(x,y)| > T_n(x,y)$

```
__global__ void FindMovingPixels(unsigned char *In, unsigned
  char *In_1, unsigned char *In_2, unsigned char *Tn,
  __global char *MovingPixelMap)
{
  long int i = blockDim.x * blockIdx.x + threadIdx.x;
  if( abs(In[i]-In_1[i]) > Tn[i] &&   abs(In[i]-In_2[i]) >
  Tn[i] ) {
     MovingPixelMap[i] = 255;
   }
   else  {
     MovingPixelMap[i] = 0;
   }
}
```

Updating the background

$$B_{n+1} = \begin{cases} \alpha B_n(x,y) + (1-\alpha) I_n(x,y) & \text{, if } (x,y) \text{ is not moving} \\ B_n(x,y) & \text{, if } (x,y) \text{ is moving} \end{cases}$$

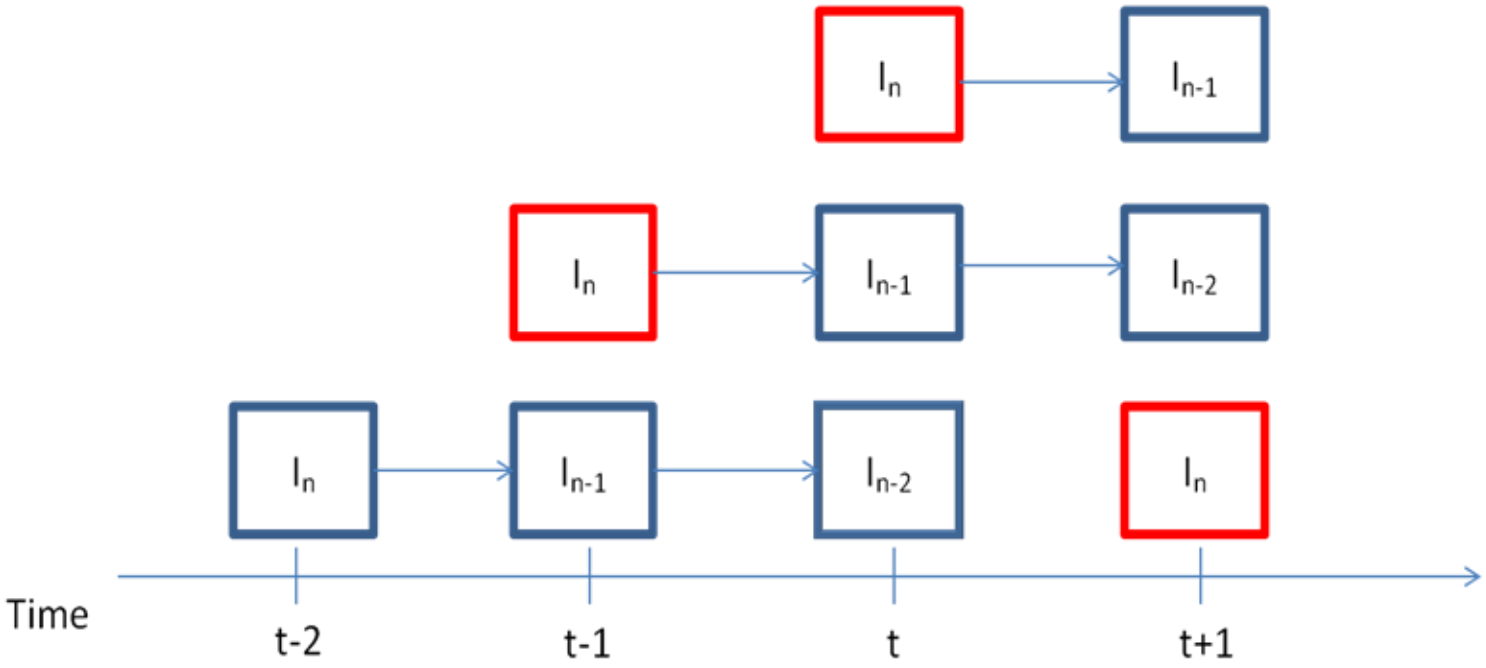
```
__global__ void updateBackgroundImage(unsigned char *In,
  unsigned char * MovingPixelMap, unsigned char *Bn)
{
  long int index = blockDim.x * blockIdx.x + threadIdx.x;
  if(MovingPixelMap [i] == 0)
  {
     Bn[i]=0.92*Bn[i] + 0.08*In[i];
  }
}
```

Updating the threshold

$$T_{n+1}(x,y) = \begin{cases} \alpha T_n(x,y) + (1-\alpha)(c|I_n(x,y) - B_n(x,y)|) & \text{, if } (x,y) \text{ is not moving} \\ T_n(x,y) & \text{, if } (x,y) \text{ is moving} \end{cases}$$

```
__global__ void updateThresholdImage(__global const char *In,
   __global char *Tn, __global char *MovingPixelMap, __global char
   *Bn)
 {
        long int i = blockDim.x * blockIdx.x +        threadIdx.x;
        int minTh = 20;
        float th = 0.92* Tn[i] + 0.24 * abs(In[i] - Bn[i]);
        if(moving_pixel_map[i]==0) {
              if(th>minTh) {
                 Tn[i] = th;
              }
             else {
                 Tn[i] = minTh;
              }
       }
}
```

Swapping Buffers



Swapping Buffers

```
void SwapBuffers()
{
    unsigned char *tempMem;
    tempMem = GPU_In_2;
    GPU_In_2 = GPU_In_1;
    GPU_In_1 = GPU_In;
    GPU_In = tempMem;
    CopyFromHostToDevice(GPU_In,CPU_In,sizeof(In));
}
```

Memory Use

- The images reside in global memory.
- We did not consider using shared memory since each pixel is accessed only once.
- Coalescing of the image data is not an issue for this algorithm as threads work on subsequent pixels.

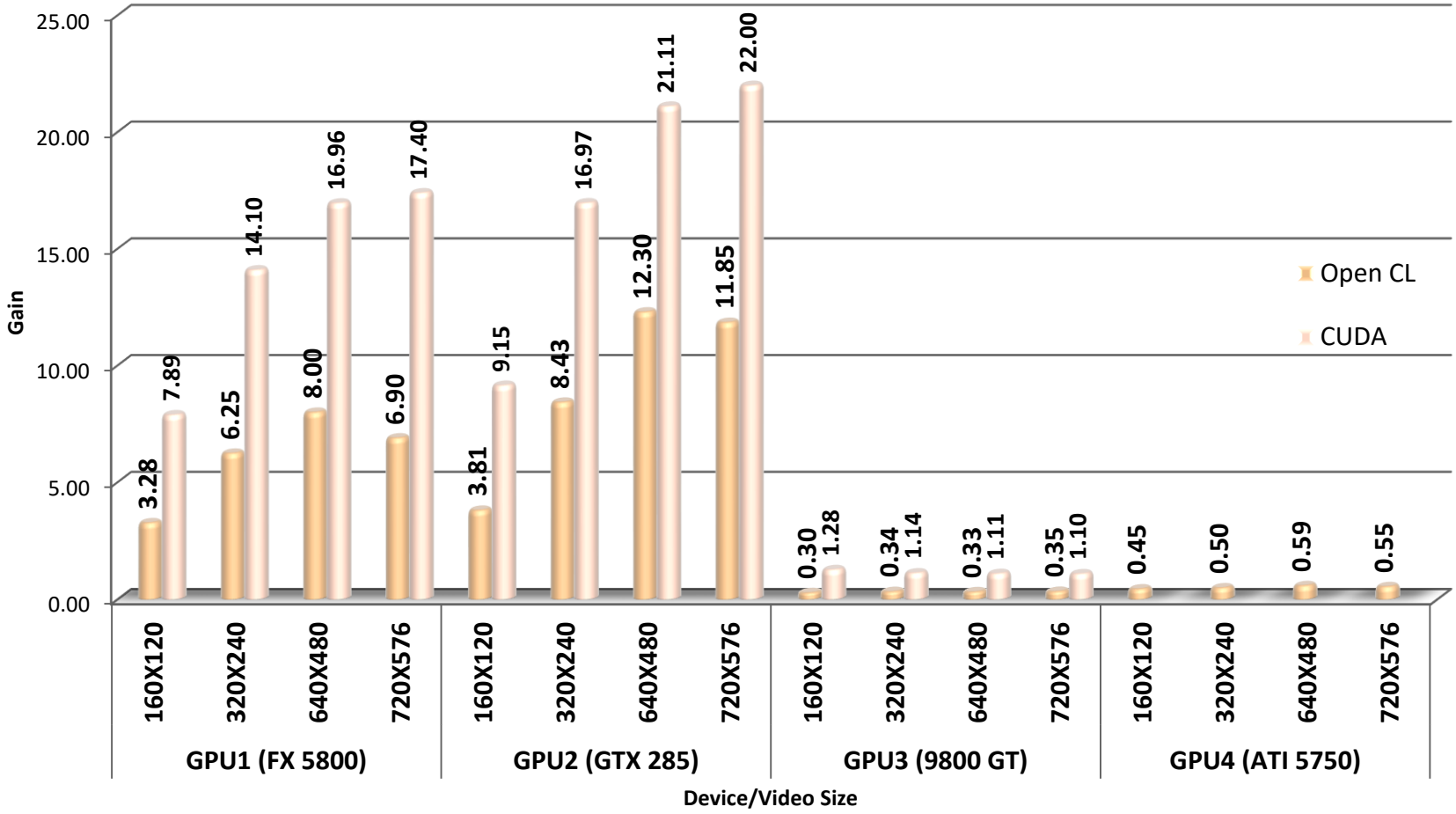
Measuring Performance

- In order to measure the performances, we calculate the maximum number of cameras that can be processed in real time.
- We assume a real-time constraint of 25 frames per second;
  - hence we calculate the number of images that can be processed in 40 ms to find out the number of cameras supported in real-time.

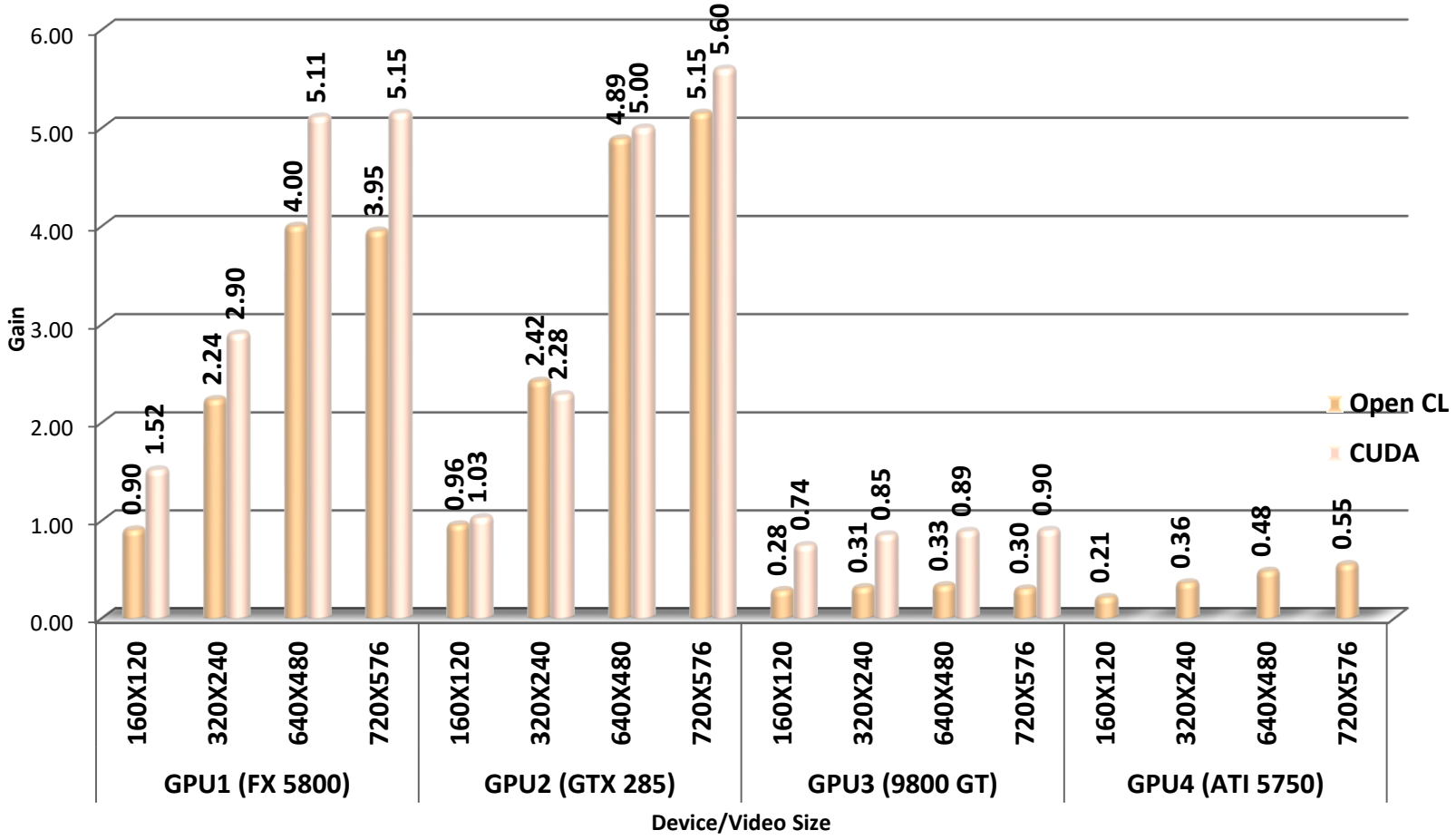
Single kernel only, performance gains with and without I/O operations

- In this experiment, we aim to measure how much speedup we get using various image resolutions on different GPU architectures compared to CPU.
- We first find the maximum number of cameras supported with different image resolutions on Intel i7 920 architecture using OpenMP to utilize all four cores of this processor.
- Then, we measure the number of supported cameras on different GPU architectures. The performance gain is calculated by dividing the GPU results to the CPU results.

Without I/O operations



I/O Included


6.00
5.00
4.00
3.00
2.00
1.00
0.00
Gain
0.90
1.52
2.24
2.90
4.00
5.11
3.95
5.15
0.96
1.03
2.42
2.28
4.89
5.00
5.15
5.60
0.28
0.74
0.31
0.85
0.33
0.89
0.30
0.90
0.21
0.36
0.48
0.55
Open CL
CUDA
160X120
320X240
640X480
720X576
GPU1 (FX 5800)
GPU2 (GTX 285)
GPU3 (9800 GT)
GPU4 (ATI 5750)
Device/Video Size

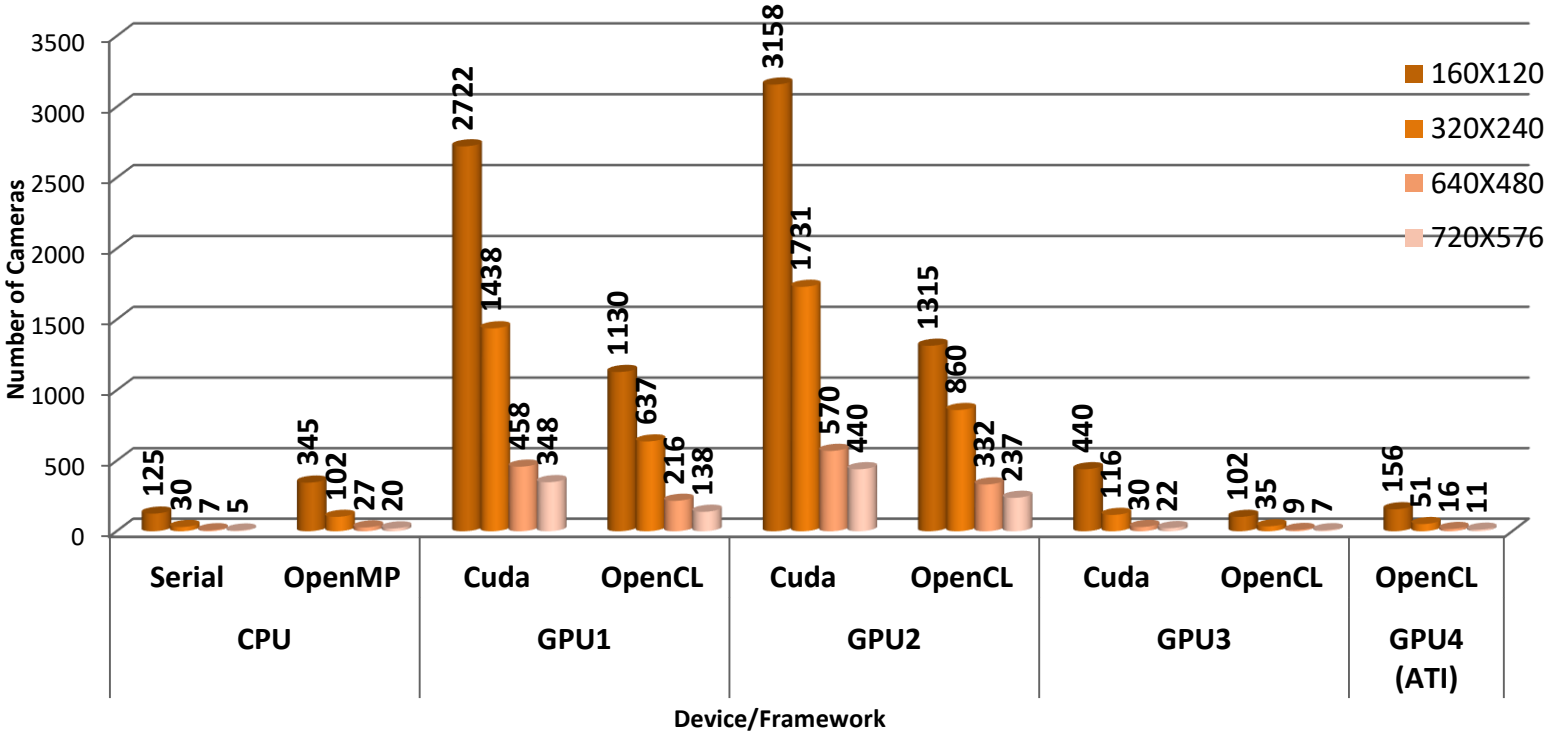
Analysis

- For the case where I/O time is included, even though CUDA has still higher performance, the gain decreases and the performance gets closer to OpenCL version.
- This is due to the I/O operation time dominating over the processing time and making the effect of processing time less significant for the overall results.
- While up to 22x increase could be observed when I/O time is not included, this drops to 5.6x when the I/O time is included.
- This drop could be remedied to a degree by overlapping the memory copy operations with the processing as will be described later

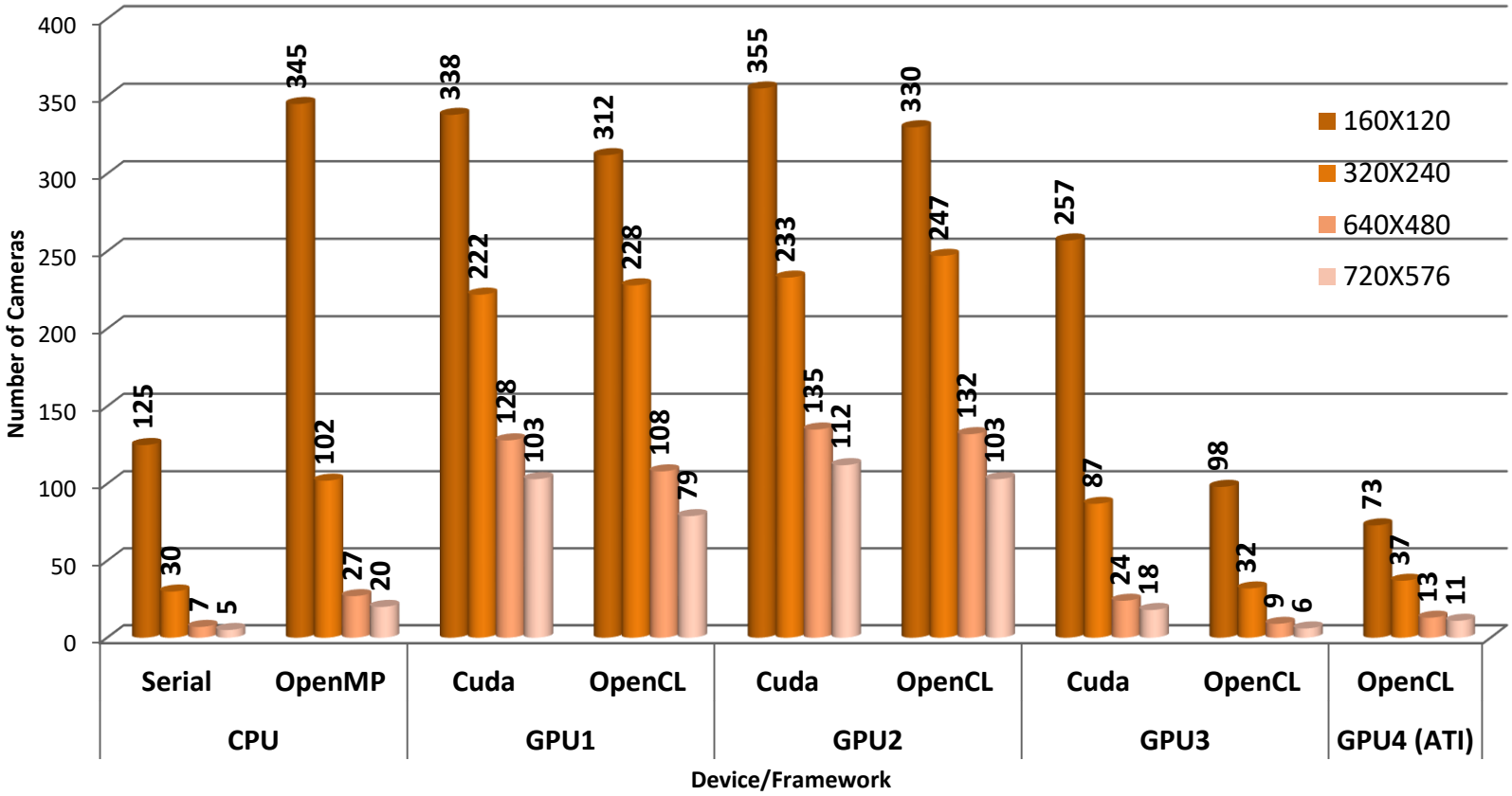
Analysis

- An important parameter is the maximum number of cameras that can be supported for real-time operation.
- These experiments have been conducted with single kernel version to get the best performance for all devices used in the experiment.
- Even though the I/O not included cases give the theoretically highest performance, it is not realistic as the data needs to be copied from the host to the device memory.
- In order to make a fair comparison, the measurements should include the time spent for data transfer from host to device.

Total Number of Cameras Supported in Real-Time – I/O not included



Total Number of Cameras Supported in Real-Time – I/O included



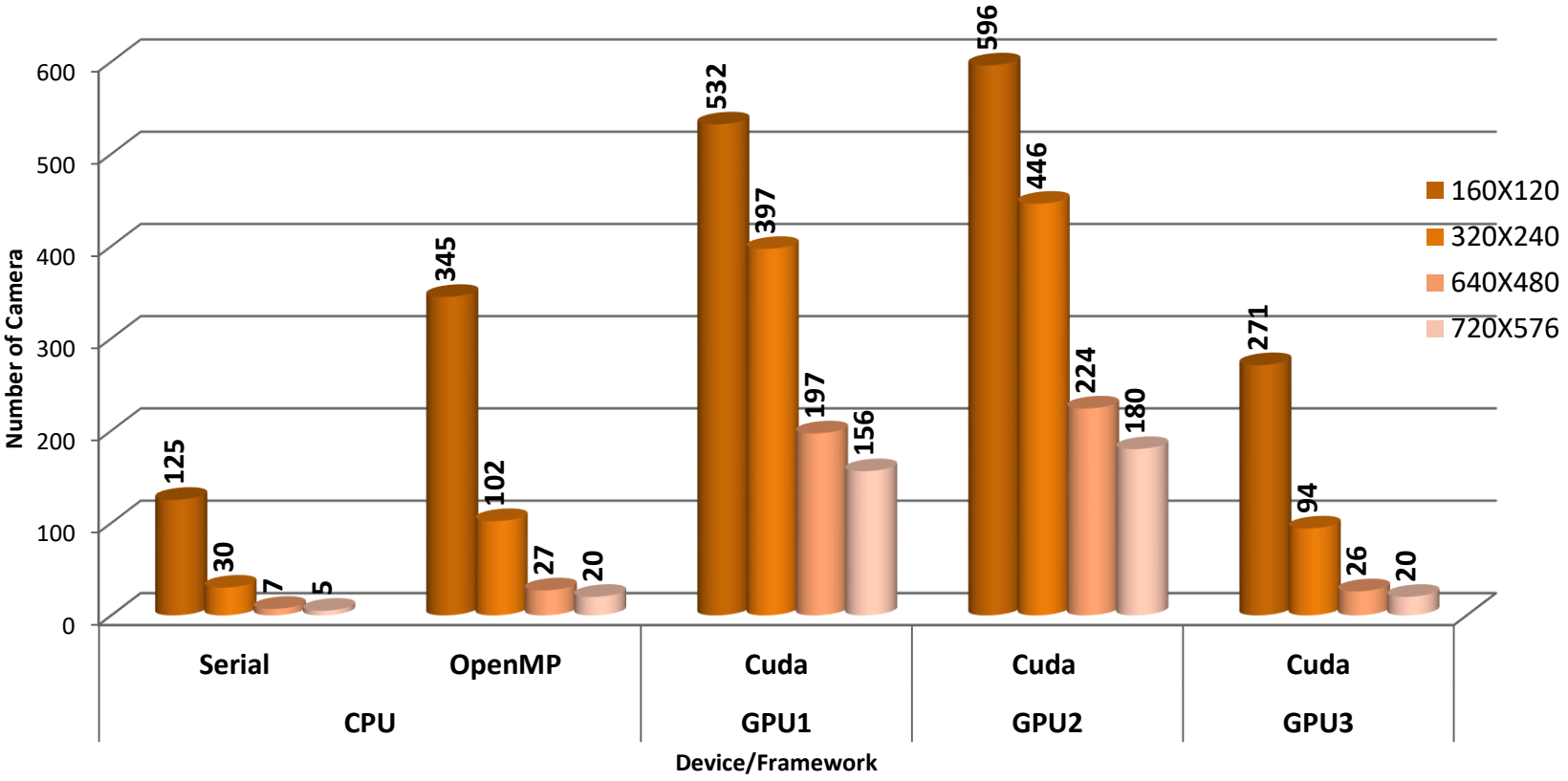
Asynchronous I/O

- As can be observed from these figures, the I/O operation during which the image data are copied from host to device have a significant adverse effect on performance.
- Hence, we implemented an asynchronous version where the memory copy and processing operations are overlapped
  - i.e. an image is processed while the next image is copied from the host *simultaneously*.

Asynchronous Version

```
cudaStream_t stream1, stream2;

cudaStreamCreate(&stream1);

cudaStreamCreate(&stream2);

cuda_process_frames<<<dimGrid,dimBlock,stream1>>>(In_D,In_1_
  D,In_2_D,Th_D,Bn_D,moving_pixel_map_D,framesize);
cudaMemcpyAsync(In_D_async,In,framesize,cudaMemcpyHostToDevi
  ce, stream2);

cudaDeviceSynchronize();

char *tmp = In_D;

In_D = In_D_async;

In_D_async = tmp;
```

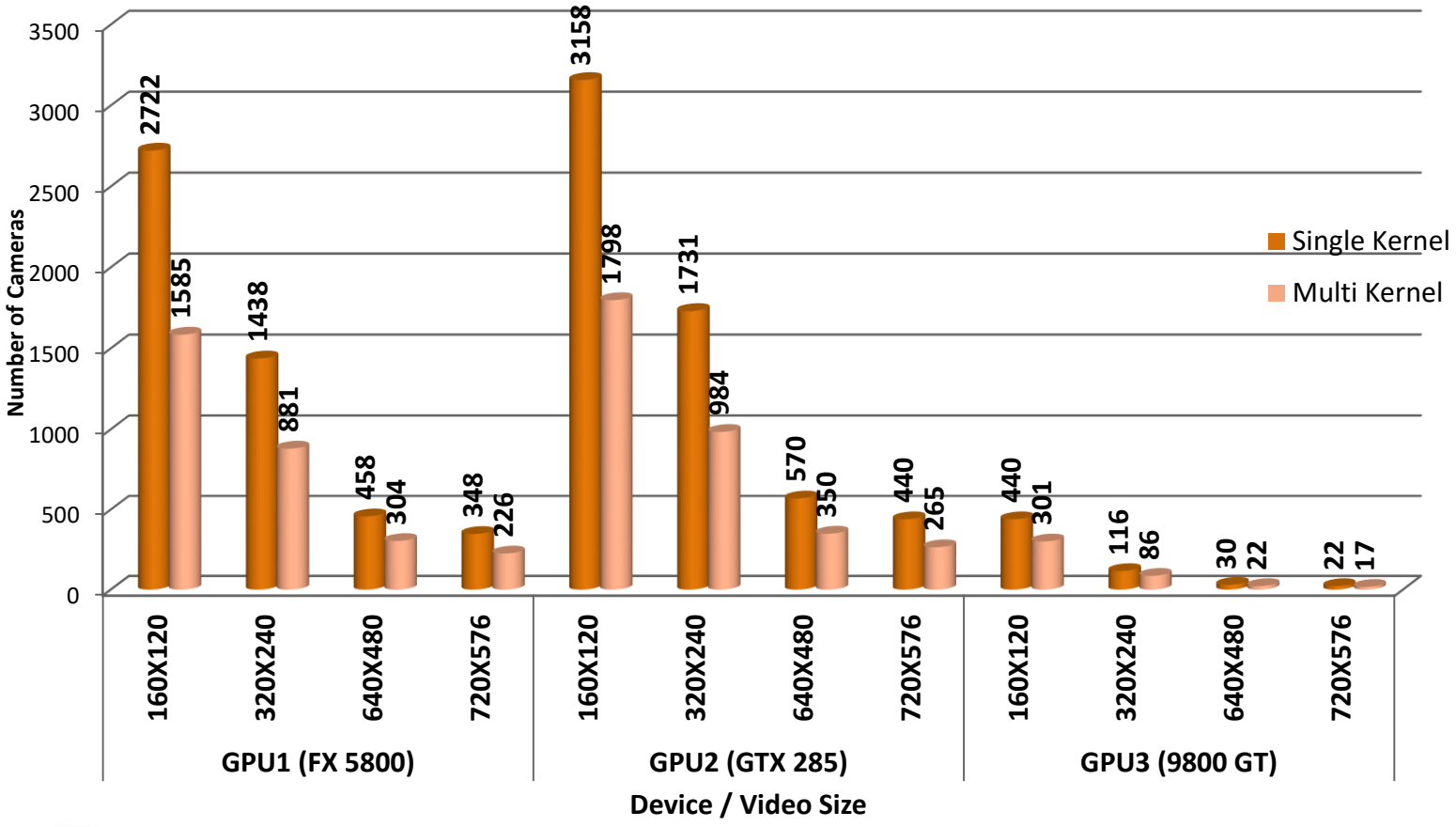
Asynchronous Version



Single Kernel vs.Multiple Kernels

- We have implemented
  - a single kernel version where all operations are gathered in a single kernel.
  - a multiple kernel version where the three main steps of the algorithms are implemented in three different kernels which are called subsequently.

Single Kernel vs.Multiple Kernels



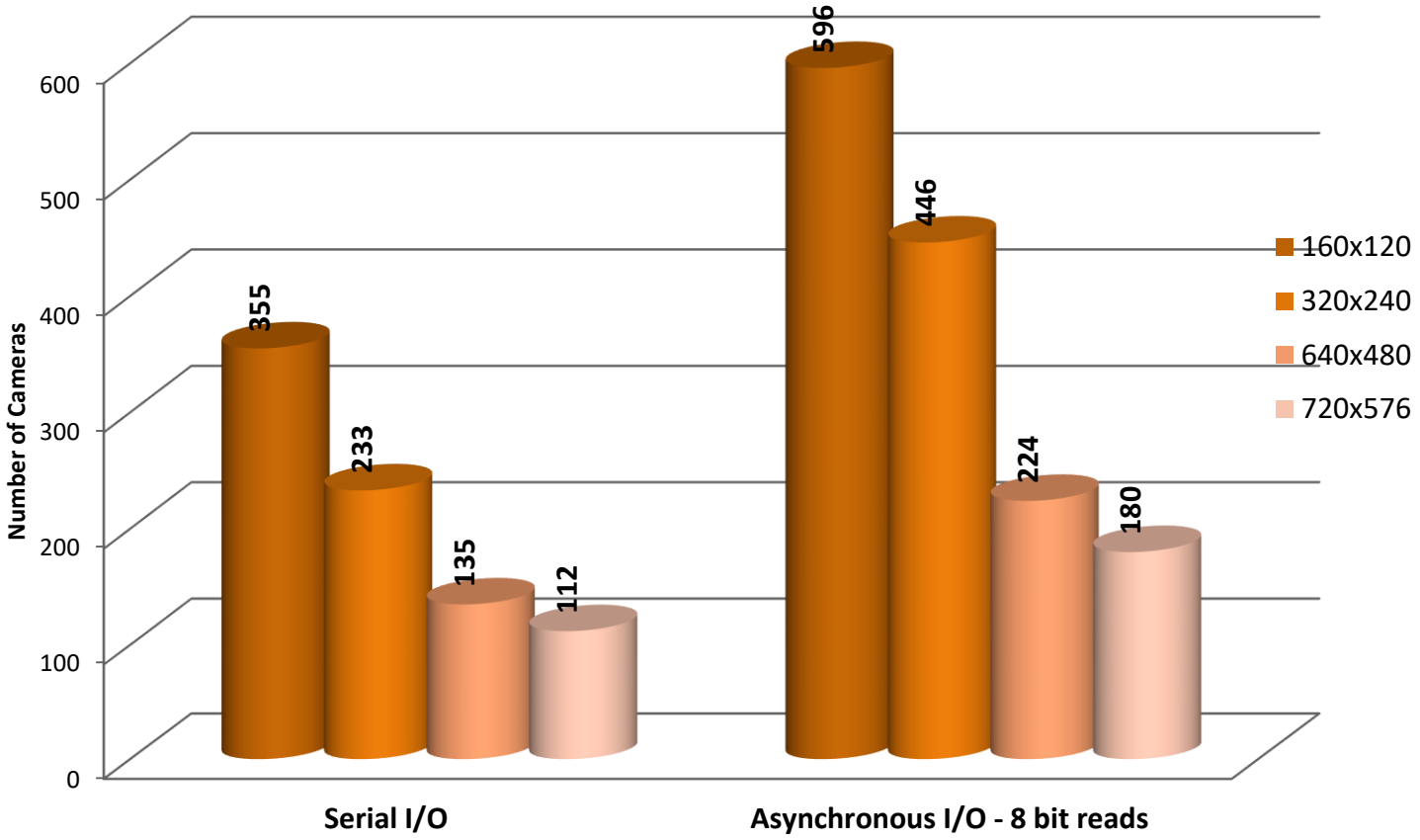
Single Kernel vs.Multiple Kernels

- Increasing the number of kernels brings a significant overhead where the performance decreases as much as 43%.
- This is due to each kernel accessing the memory independently and hence requiring multiple accesses to the same data.
- On the other hand, in the single kernel case, data is fetched once.
- OpenCL suffers more heavily compared to CUDA.

Serial I/O vs. Asynchronous I/O

- As serial I/O case doesn't reflect real performance gains and serial I/O is not the best method for achieving the best performance, an asynchronous version has been implemented.
- The results show that much higher number cameras could be processed using asynchronous calls compared to using serial I/O.
- With asynchronous calls, 68%, 91%, 66% and 61% more cameras could be processed for 160x120, 320x240, 640x480 and 720x576 resolutions respectively.

Serial I/O vs. Asynchronous I/O



32-bit access to the global memory

- In the previous experiments, we used *unsigned char* to access 8-bit image data from the global memory.
- Memory reads in GT200 is done for half-warps of 16 threads and reads are done for 32-, 64- or 128 bytes. In the previous cases, as each thread reads a byte, 16 bytes are read.
- For better utilization of global memory bandwidth, these reads have to be 32-, 64- or 128 bytes. In the context of the problem in hand, this can be achieved by first reading 4 *chars* per thread and then processing each *char,* which results in 16 * 4 = 64 bytes reads for each half-warp.

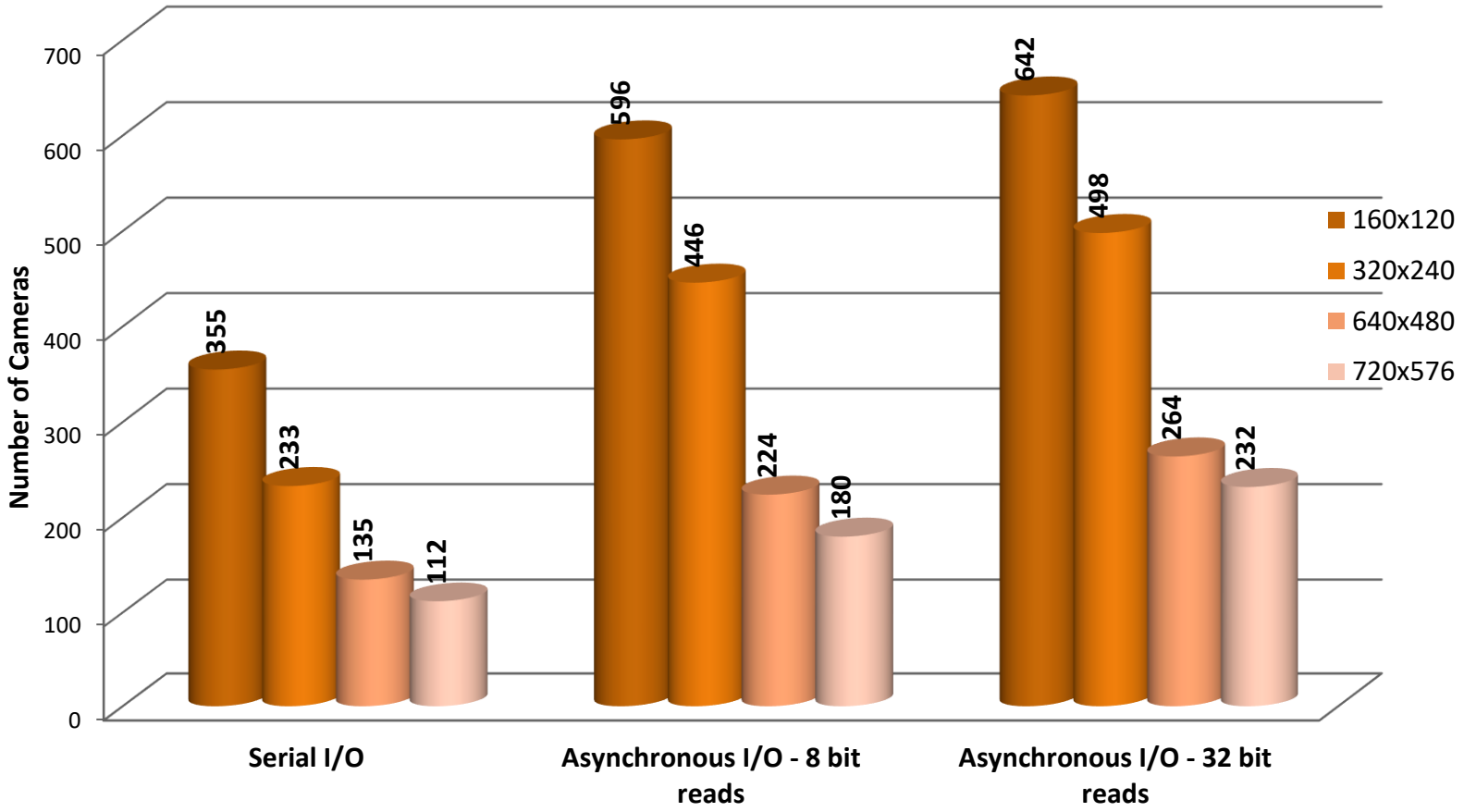
32-bit access to the global memory

- The following example demonstrates reading of data in 32-bit chunks processing 4 x 8-bit elements by each thread.
- Note that in this case only the current image In and the background Bn are loaded from the global memory and Bn is written back to the global memory. For a complete implementation all the other images In_1, In_2, Bn, Tn and MovingPixelMap should also be loaded in a similar fashion.

32-bit access to the global memory

```
__global__ void processImage(unsigned int *In, unsigned int *Bn) {
long int index = blockIdx.x * N + threadIdx.x * 4;
unsigned int packedIn = In[index]; // Read an int from the global mem.
unsigned int packedBn = Bn[index]; // Read an int from the global mem.
unsigned int bnout = 0;

for(int i=0;i<4;i++) {
  unsigned int in = (packedIn & 0x0ffu); // read a char from packed data
  int outdata = ...;//Process the data here, each thread processes 4-
  chars
  bnout += ((unsigned int)(outdata)) << (i * 8); // pack the output
     packedIn >>= 8; packedBn >>= 8;// Move to the next char
     }
  Bn[index] = bnout; // Copy the result back to the global memory
  }
```

Serial I/O vs. Asynchronous I/O



Case Study 2 – Cross Correlation

Case Study: Cross Correlation

- Correlation is a technique for measuring the degree of relationship between two variables. It is used in broad range of disciplines including image/video processing and pattern recognition.
- The relationship between the variables is obtained by the calculation of coefficients of correlation.
- Example: Comparison of images of bullet cartridges

Cross Correlation

- There are a number of different coefficients including Kendall's tau corelation coefficient and Spearman's rho correlation coefficient but the most popular and widely used one is that of Pearson's correlation coefficient (PMCC)
- It takes values between [-1,1]
  - 1: perfect match
  - 0: no match
  - -1: perfect negative correlation

Pearson's Cross Correlation

- PMCC is defined as the covariance of the variables that are to be compared, divided by their standard deviations:

$$\rho_{x,y} = \frac{cov(x,y)}{\sigma_x \sigma_y}$$

$$\rho_{x,y} = \frac{\sum_{i=0}^{n}(x_i - \mu_x)(y_i - \mu_y)}{\sqrt{\sum_{i=0}^{n}(x_i - \mu_x)^2}\sqrt{\sum_{i=0}^{n}(y_i - \mu_y)^2}}$$

$$\rho_{x,y} = \frac{n\sum_{i=0}^{n} x_i y_i - \sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i}{\sqrt{n\sum_{i=0}^{n} x_i^2 - \left(\sum_{i=0}^{n} x_i\right)^2}\sqrt{n\sum_{i=0}^{n} y_i^2 - \left(\sum_{i=0}^{n} y_i\right)^2}}$$

Implementation

- The first approach: using only the global memory of the GPU
- The second utilizes the shared memory along with the global memory.

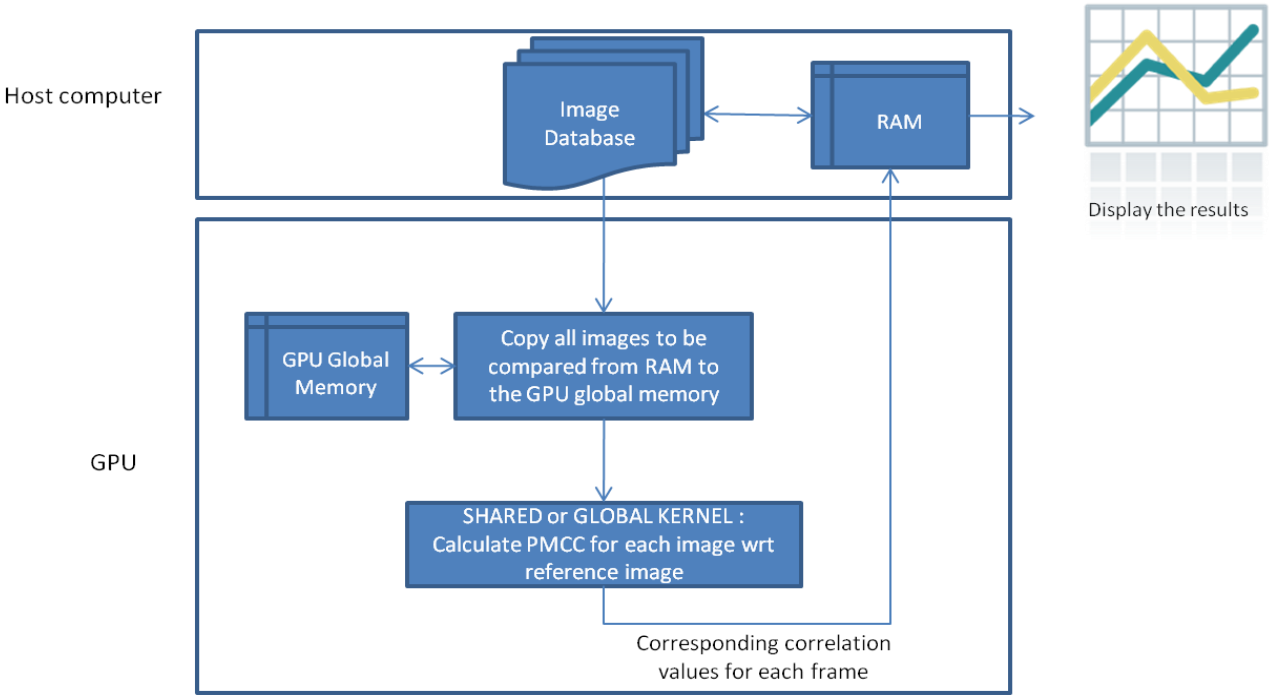
Implementation

- The GPU implementation of the algorithm significantly differs in parallelization from the background subtraction.
- In this algorithm each worker cannot be assigned to each pixel of the image since the algorithm is composed of summations of pixels.
- The jobs are rather chosen as the individual image frames to be able to fully utilize the GPU.

Implementation

- But it should be noted that this yields two constraints:
  - High number of images are needed for high GPU utilization.
  - The images to be compared should be available on the GPU memory.

Flowchart

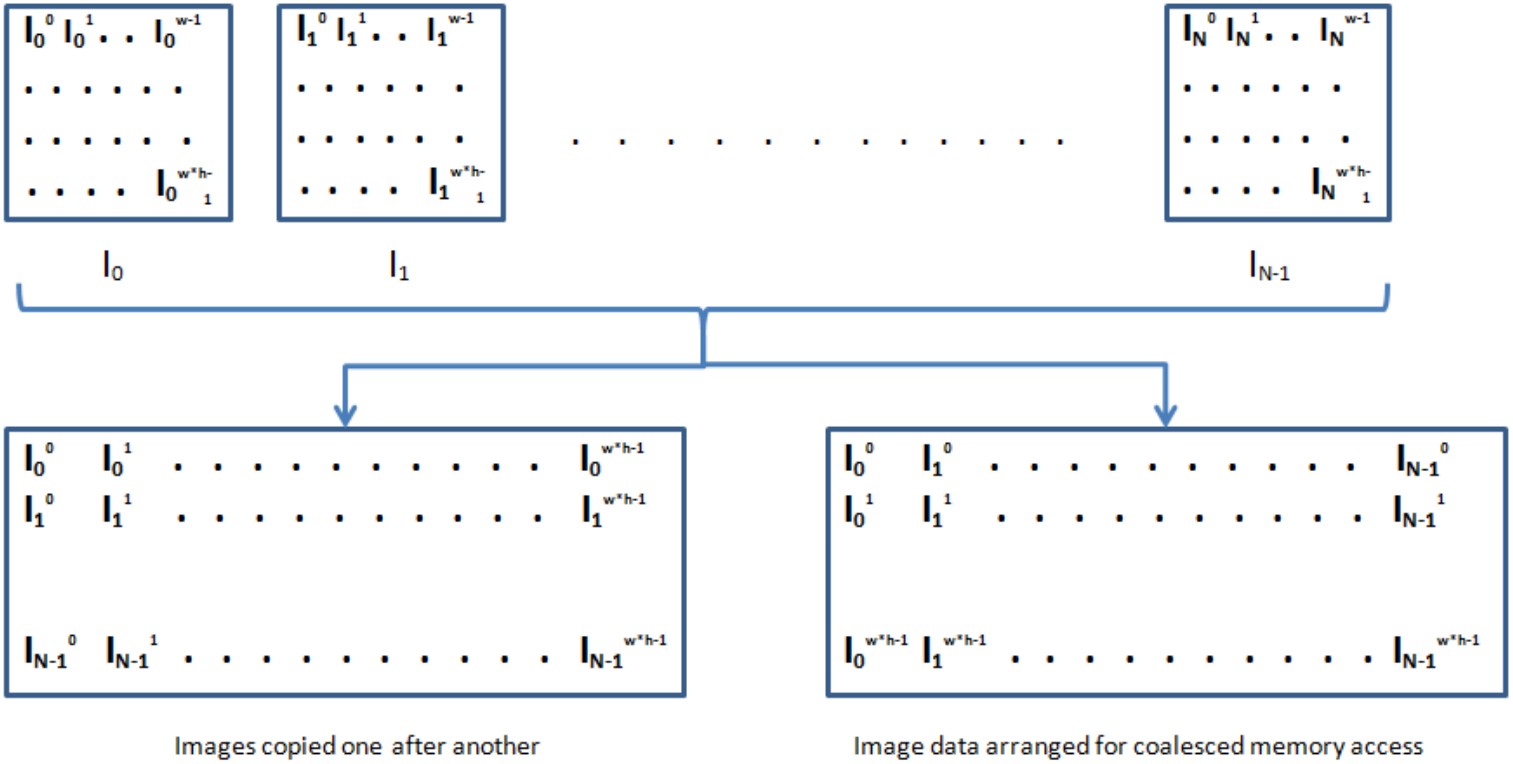


Flowchart of the calculation of Pearson's correlation coefficients

Global Memory Only

- The first approach which uses only the global memory of the GPU is rather straightforward; the given one pass version for the PMCC formula is directly implemented.
- Example kernel code is very similar to the one using the shared memory version.

Global Memory Only



Memory arrangement for coalesced memory access

N is the total number of images where *n*th image is shown with I_n, *w* and *h* are the image width and height, respectively.

Global Memory + Shared Memory with Coalesced Memory Access

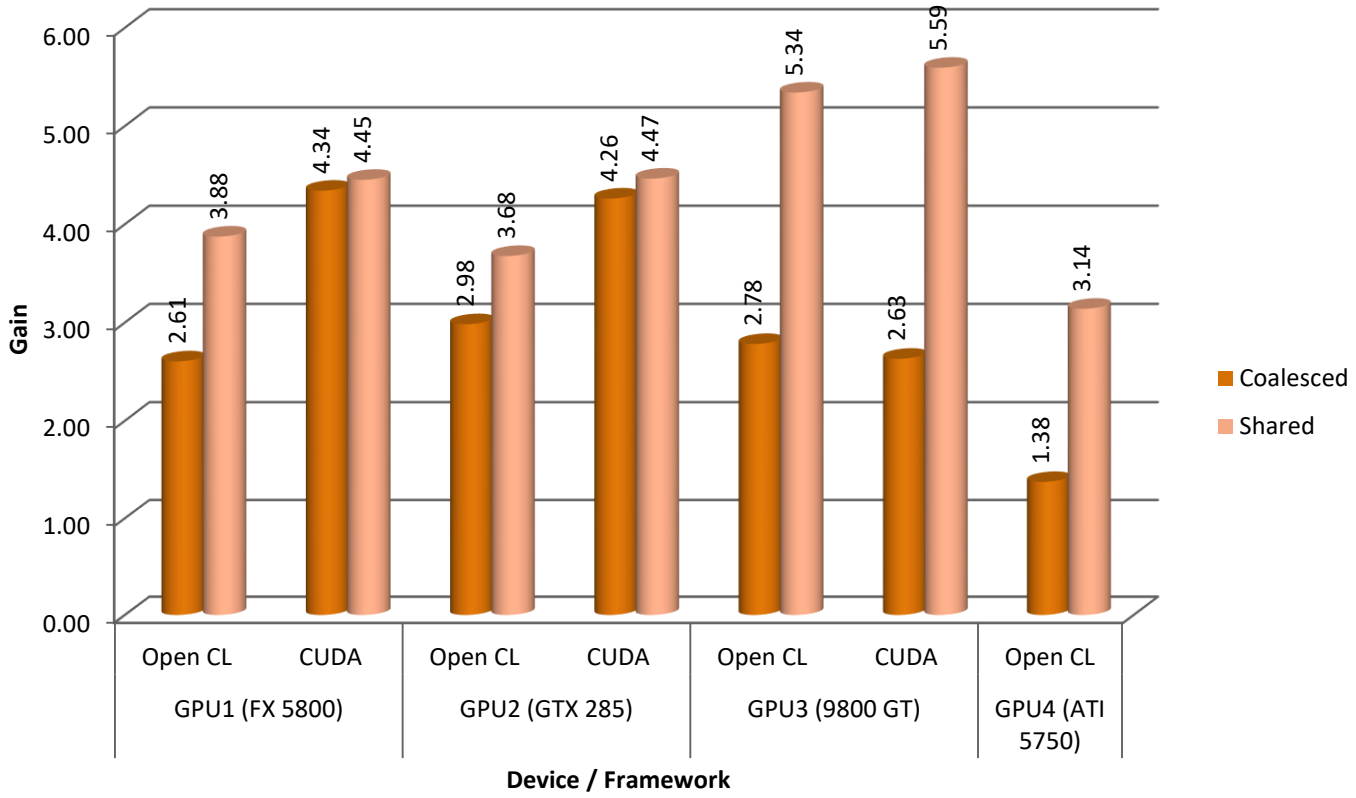
- One downside of the first approach is that every image to be compared accesses to the same memory space, since they are all compared with the same reference image.
- The same memory which is written in global memory is read over and over again.

Global Memory + Shared Memory with Coalesced Memory Access

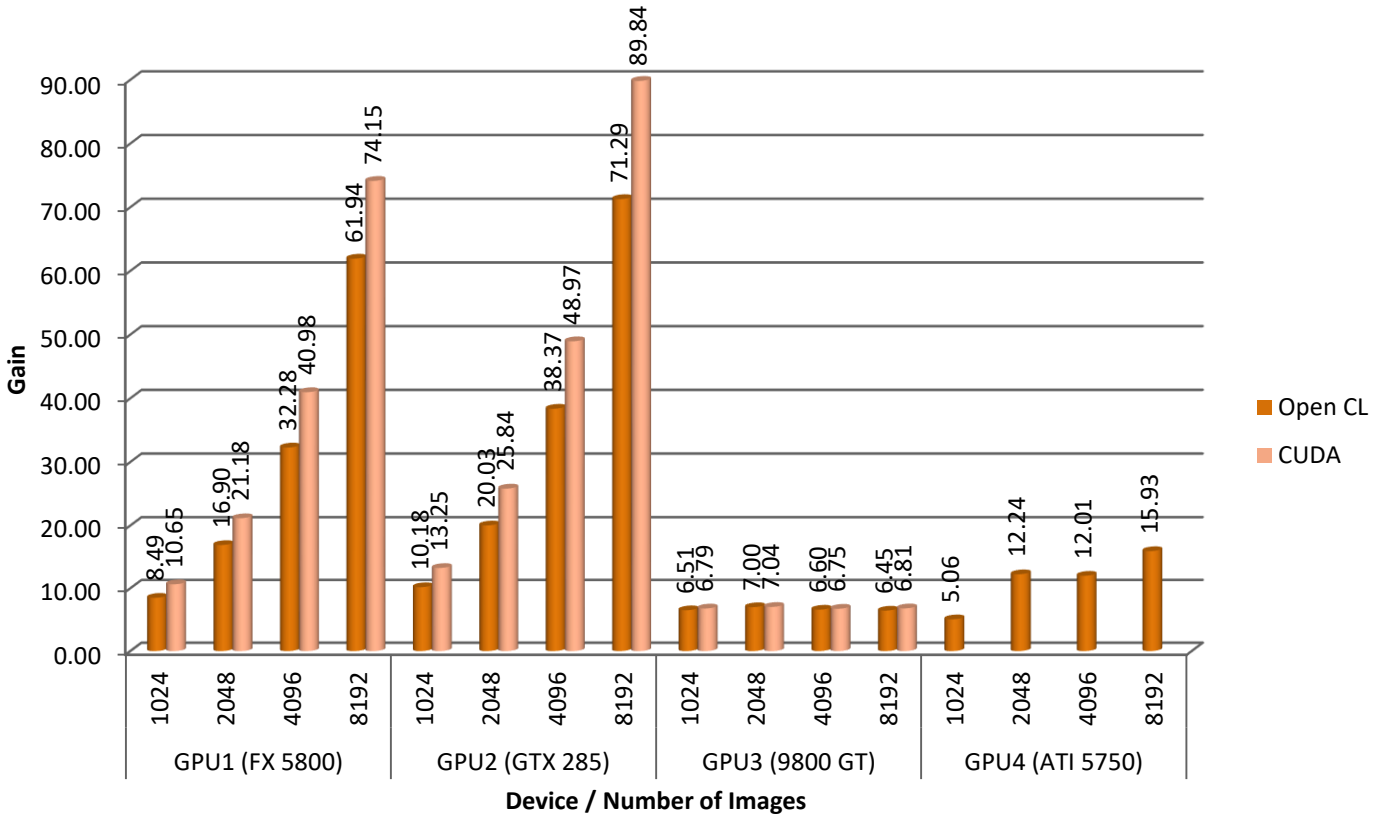
- An approach to overcome this inefficiency is using shared memory:
  - Occupy a block of shared memory
  - Calculate how many passes are required to process all the pixels in an image by dividing pixel number by the chosen shared memory block size.
  - For each pass, firstly load a block of pixel values of reference image from global to shared memory. Calculate each element within the block using the values in the shared memory when needed. Note that, the elements are added up in the each step on top of previously calculated values.
  - At the end of the passes, we end up with calculated elements, thus just putting the values in equation 3, the correlation coefficient is obtained.

```
__global__ void cudaPMCC(float *GPUcorrVal, unsigned char *GPUhostImg, unsigned char *GPUsampleImg, int ImageWidth, int
      ImageHeight, int GPUimgNumber){
int index = blockDim.x * blockIdx.x + threadIdx.x;
if(index < GPUimgNumber) {
 float sum1=0.0f, sqrSum1=0.0f, sum2=0.0f, sqrSum2=0.0f, prodSum=0.0f;
 __shared__ unsigned char temp[blockDim];
 int BLOCK_LENGTH = blockDim;
 int shotCount = (ImageWidth*ImageHeight + BLOCK_LENGTH -1)/BLOCK_LENGTH ;
 for(int shotIndex = 0 ; shotIndex < shotCount ; shotIndex += 1) {
      int shotBegin = shotIndex * BLOCK_LENGTH ;
      int shotEnd = min(shotBegin + BLOCK_LENGTH, ImageWidth*ImageHeight) ;
      int shotLength = shotEnd - shotBegin ;
      int shotRef = shotBegin + threadIdx.x ;
      temp[threadIdx.x] = (unsigned char)GPUhostImg[shotRef];
      __syncthreads();
      for(int i=0;i<shotLength;i++) {
         float val1 = temp[i];
         float val2 = GPUsampleImg[(shotBegin+i)*GPUimgNumber + index];
         sum1    += val1;
         sqrSum1 += val1*val1;
         sum2    += val2;
         sqrSum2 += val2*val2;
         prodSum += val1*val2;
      }
 }
GPUcorrVal[index] = 0;
float count = ImageWidth*ImageHeight;
float variance1 = (count*sqrSum1)-(sum1*sum1) ;
float variance2 = (count*sqrSum2)-(sum2*sum2);
float covariance = (count*prodSum)-(sum1*sum2);
float energySqr = variance1*variance2 ;
if(energySqr > 0)
GPUcorrVal[index] = covariance/sqrt(energySqr);
}
}
```

Performance gain for 320x240 images



Effect of Increasing the Number of Images



Performance gain for increasing image number over OpenMP implementation on the CPU for 160x120 image sizes.

Analysis

- As seen from the figure, with the increasing number of images the performance gain increases for GPUs with high number of multi-cores since utilization increases for those GPUs.
- As a result of efficiency, comparison of 8192 images of size 320x240 was performed 89.84 times faster than the OpenMP CPU implementation.
- Amount of RAM could affect the performance as more images could be fit in and hence processed in parallel when RAM is higher.