Comparative Performance of GPU, SIMD and OpenMP Systems for Raw Template Matching in Computer Vision

Juan Mendez
Departamento de Informática y Sistemas. Universidad de Las Palmas de Gran Canaria
35017, Las Palmas, Spain
jmendez@dis.ulpgc.es

Javier Lorenzo
SIANI. Universidad de Las Palmas de Gran Canaria.
35017, Las Palmas, Spain
jlorenzo@iusiani.ulpgc.es

Modesto Castrillon
SIANI, Universidad de Las Palmas de Gran Canaria.
35017, Las Palmas, Spain
mcastrillon@iusiani.ulpgc.es

ABSTRACT
Template matching is a traditional technique of Computer Vision whose advantages and disadvantages are known. However, advances in computer hardware allow computing it effectively with the use of SIMD instruction set, GPUs or multi-core systems. The computation of that low-level primitive in sub millisecond scale would improve high theoretical methods if they are used with high efficient primitives. This paper presents the comparative results of basic template matching by using SIMD instructions, multi-core systems and multi-GPU implementations. The results of this study will show that the high-specialized instruction in modern releases of SIMD and the use of multi-core systems outperforms the implementations based on GPUs for small mask size due to memory transfer cost. However, for big mask size GPU and SIMD systems have similar performance.

Keywords
Computer Vision, Template Matching, Parallel Computing, GPU, Multi-Core Systems.

1. INTRODUCTION
Template Matching is a Computer Vision procedure focused on the detection of local features in a image that seems or resembles similar properties than a small part of the image or mask. This is a general definition and many different approaches implement the concept by using different paradigms. The main drawback of template matching is that it implies a "wasteful" exploring of the image for searching a local area very similar to the mask. This requires the *sliding* of the mask across the entire picture and the computation of some measure of similarity or distance for each position.

Computer Vision applications have two opposite constraints. The first is related to the randomness of the data that requires the use of higher-level theoretical procedures. These methods allow robust procedures that deal efficiently with the enormous variability of the data and provide stable results. The second constraint is related to the computational efficiency that tends to carry out real-time performance to fit the application needs and can be useful in real world problems. The progresses in Computer Vision deal to advances in both directions. The equilibrium between both viewpoints determines the most useful procedures for a defined state of the advances in the computer technology and in theoretical methods.

The evolution of computer hardware can revalue some traditional and simple procedures because they will be computed faster than in older implementations. These fast and simple procedures can be used as basic results, which are the first level of intermediate results, in more elaborated and complex procedures. That is, they can be considered primitives rather than complete procedures. Raw Template Matching is a traditional Computer Vision technique with advantages and drawbacks, but it can be computed very efficiently in modern computer hardware as Graphics Processor Unit (GPU) and Single Instruction Multiple Data (SIMD) arithmetic units in multi-Core systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Modern computer ranging from desktop to rack servers have multiple cores as well as some GPU in standard hardware configuration. Also many processors, such as the Intel/AMD series, include vector units that allow advanced SIMD instructions. Thus, no additional cost is needed in order to have high performance computer hardware. However, these units, that are normally unused, are difficult to integrate in standard programming code unless no special programming libraries are used. That is a very common tendency in many branches of computing where hardware advances are faster than programming techniques, or programming practices. The introduction of special libraries such as OpenCV [Bra08a] provides to the user the advantage of high computer performance and hides its complexity. But OpenCV library is a bit conservative and does not take advantage of all the features that the hardware can provide. The best performance using SIMD is achieved when the assembler code is used. Developing a complete application in assembler level is not a good idea, but coding only small pieces of high efficient primitives can be a good option in some special cases. Although programming GPU is difficult, the use of this specialized hardware is useful only if their programming is hidden in specialized libraries for small but high efficient pieces of the software used for Computer Vision.

The main idea of advanced tracking systems such as Lukas-Kanade procedure [Luk81a] is based on the assumption that small changes in image motion such as the brightness constancy and spatial coherence of small areas in the image motion can be detected; that is, for short time intervals, small masks can be used to detect the motion of real world objects. The similarity between mask and local image area can be computed by using different features and classification strategies and methods, but the one based on the similarity or distance between the raw data at pixel level is the simplest of all.

According to Brunelli [Bru09a], the main drawback of Template Matching is its high computational cost, which has two distinct sources. The first one is the necessity of using multiple templates to capture the variability exhibited by the appearance of complex objects. The second one is related to the size of templates: the higher the resolution, the heavier the computational requirements. Raw template matching is much more simple if compared with advanced matching that incorporates geometric invariance [Ull04a] [Kin07a], of feature characterization rotation invariant of mask based on moments of Hu and Zernike [Teh88a]. In the case of advances tracking applications, template matching must be used along with higher level procedures such as Kalman and Particle filter [For02a]. Template Matching is also a basic tool used in video encoding, where the correspondence points between successive frames in video image, eg. in MPEG video compression, implies the detection of image block of 16x16 pixels in previous frames [Sha01a]. However, in video compression the required matching is carried out in a narrow area bounding the block.

This paper presents a comparative study of implementation of raw template matching based on modern technologies using GPU and SIMD architectures that allows the computation of template matching of small masks in few milliseconds. The basic results of raw template matching are presented as well as its efficient implementation in the more modern release of SIMD instruction set. The details of the implementation in GPU are also presented, and finally the comparative results of both approaches by using multiple cores with OpenMP [Cha08a] parallel programming.

2. RAW TEMPLATE MATCHING

The easiest way to achieve raw template matching is by using a similarity or distance measure between a local area of the image and a mask or template. If the matching is based on a distance measure, it is necessary to find the minimum or minima. A widespread used distance measure in different branches of Mathematics and Computer Science is the one based on a vector norm, e.g. as the based on the Minkowski metric [Bru09a] [Har01a]. The matching result $\boldsymbol{R}$ can be computed from the image data $\boldsymbol{D}$ and the template or mask $\boldsymbol{M}$ as:

$$\boldsymbol{R}_p = \left[\| \boldsymbol{D}(x+u, y+v) - \boldsymbol{M}(u,v) \|_p\right]^p$$

where $\| A \|_p$ is the L_p norm. Examples of very used norms in Mathematics and Computer Vision are: L_1, L_2 and L_∞. The definition of template matching using multichannel images and masks in these cases is the following:

$$R_1(x,y) = \sum_c \sum_{u,v} |D_c(x+u, y+v) - M_c(u,v)|$$

$$R_2(x,y) = \sum_c \sum_{u,v} |D_c(x+u, y+v) - M_c(u,v)|^2$$

$$R_\infty(x,y) = \max_{c,u,v} |D_c(x+u, y+v) - M_c(u,v)|$$

The L_2, norm which generates the template matching R_2, is used by OpenCV Library [Bra08a], although it has not the lowest computational cost. OpenCV is originally based on performance primitives of Intel/AMD processors, but these systems are better suited to compute efficiently the L_1 norm, which is based on the computation of Sum of Absolute

Differences (SAD). Also NVIDIA GPUs have basic support for computing the SAD primitive. The SAD for two arrays is defined as: $SAD(\boldsymbol{A},\boldsymbol{B}) = \sum_i |A_i - B_i|$. Although the use of L_1 is no advantageous in general purpose programming, some special hardware makes it the best choice. However, nowadays the special SIMD hardware of Intel processors is of such a common and widely use that we can call it as general purpose hardware.

Instead of using a coordinate system placed on the center of the template mask, we will use upper-left corner centered coordinates. It is more advantageous to deal with memory alignment, because it plays a main role in efficient memory accesses. The real position of the detection of the mask can be obtained after the minimum detection by using a simple offset to the mask center. We will use the offset evaluation of the match for a mask of dimension $S_x \times S_y$defined as:

$$R(x,y) = \sum_{u=0}^{S_x-1} \sum_{v=0}^{S_y-1} |D(x+u, y+v) - M(u,v)|$$

Multi-channel template matching, e.g. in RGB images, can be obtained simply by adding the results obtained in the previous equation that was applied in every image channel and mask. For a $N_x \times N_y$ image, the border band, which is usually located bounding the image, is moved to the right and low of the image. The right null band is $S_x - 1$ width and low band is $S_y - 1$ high. In the previous Equation the (x, y) values run in the intervals: $x \in [0, N_x - S_x + 1]$ and $y \in [0, N_y - S_y + 1]$. Usually, the border band is set to null value, but to avoid any problem with the minimum computation we decide to set it to the highest numeric positive value in its internal representation. After the local detection of the minimum matching value, its location must be offset by $(S_x/2, S_y/2)$. Template matching in 2D has advantages related to the data alignment, and it also can be computed by row 1D oriented matching:

$$R(x,y) = \sum_{u=0}^{S_x-1} \left[\sum_{v=0}^{S_y-1} |D_{y+v}(x+u) - M_v(u)| \right]$$

That can be computed by using the following general 1D template matching:

$$B(x) = \sum_{u=0}^{S-1} |A(x+u) - C(u)|$$

The use of Region of Interest (ROI) reduces significantly the computational cost because the template search can be reduced to a fraction of the image area. The ROI usage for tracking requires the implementation of a strategy for updating the ROI according to the detected trajectory by using a predictive filter such as the one based on Kalman or Particle Filters. In this paper we have computed the worst case, when the maximal ROI extends to the entire image. This option allows us to obtain an upper bound of the tracking computational time.

The two main problems involved in the computation of template matching in 2D and 1D are arithmetic and memory access. The arithmetic is concerning to the computation of SAD, which is not cheap if no special hardware is available. Memory access is a less evident problem, but it is more important in modern computers because their performance is mainly related to the pattern of memory access. The sliding of the mask across the entire image requires that all the different memory alignments patterns must be used. The sliding u value between $B(x)$ and $A(x+u)$ is the cause of many of the low performance issues in computer applications because it is very important in memory access efficiency.

3. SIMD-BASED TEMPLATE MATCHING

The efficient implementation of template matching in modern computer architectures requires a revision of some specialized instructions of the machine code of some popular microprocessors. Unfortunately, those instructions are not used by the compilers to translate the user code written in high level languages as C/C++ to machine code. This implies that the user must code the parts of the software dealing with the specialized instructions using assembly language or inline embedded assembly. In this section only some guidelines of the specialized instructions are shown with the aim of being useful for researchers and developers in Computer Vision.

Early versions of SIMD in Intel/AMD processors incorporate a SAD instruction called *psadbw* in the first SSE (Streaming SIMD Extension) release of the MMX (Multi Media eXtension) instruction set. It uses the MMX registers of 64 bits, allowing computing the SAD for 8 bits unsigned integer data. This instruction allows improving the arithmetic part of the array matching but does not solve the problems related to the sliding of the mask array across the data array. Multiple unaligned data read were needed to perform and achieve the whole matching.

A recent SSE extension, that use 128 bit registers, has included the *mpsadbw* instruction [Int09a] that solves this problem by including in-hardware sliding computation, which avoids that the user must design a code with unaligned data load. This improvement introduced in SIMD release SSE4.1 allows higher

performance, but at the time of this study, it is not included yet in all computer vision libraries, e.g. OpenCV. This instruction computes multiple and sliding SAD for 4 bytes masks including an immediate additional argument (*imm8*) that controls the selection of the group of 4 bytes defining the mask and also controlling the sliding option. The *mpsadbw* instruction requires three arguments: the source register containing 16 byte mask data, s(0-15), the result register that initially contains 16 byte data from an image row, d(0-15), and finally one byte register, imm8, to control the instruction mode. The result is obtained as 8 unsigned short array r(0-7).

This instruction is extremely important for modern HDTV codecs, and allows an 8x8 block difference to be computed in fewer than seven cycles [Kua07a], but also is very useful in general template matching of small mask on the whole image. This instruction implements the computation of the SAD for 4 unsigned char integer values and the in-hardware computation of the sliding of the 4 bytes across the data register. If $imm8(2) = a \in \{0,1\}$ and $imm8(0-1) = b \in \{0,1,2,3\}$, it computes for $i = 0, \ldots, 7$

$$r(4a + i) = \sum_{j=0}^{3} |d(4a + i + j) - s(4b + j)|$$

Computing the whole sliding of a mask across a data array will require a more complex arrangement. Therefore, to compute a full matching of a data array with a mask of suitable size multiple of 4 bytes, we have designed a computational arrangement, which is shown in Figure 1, as an useful chart that allows an easy implementation for Computer Vision developers and researchers. In this chart, Data and Result are the arrays involved in a general 1D matching. The first array is 8 bits unsigned integer and the second array 16 bits unsigned integers, also it is used the 8 bit unsigned int Mask array, whose length is multiple of 4. In Intel architecture the SSE instructions for loading and storing data are penalized if the memory reference is not aligned to 16 bytes. The goal of the arrangement shown in the Figure 1 is the computation of Result(0-7) and Result(8-15) for different mask sizes. Data are read from memory in 16 bytes block such as the CPU can read Data(0-15) and Data(16-31). However, it can be read Data(8-23) in unaligned way by incurring in efficiency penalties. To avoid that, it can be obtained Data(8-23) by using register instructions from the aligned Data(0-15) and Data(16-31), which can be read without penalties. Data contained in the first row are directly read and the contained in the second row are obtained from the previous by using register operations.

Column containing Result(0-7) defines the intermediate results that must be added to get the result and from which they are obtained. For instance, to compute Result(0-7) by using the smaller mask of 4, we must compute by using the imm8 value of 000 in the Data(0-15) according to the description of the instruction. When we want the same result but for a mask size of 8, we must obtain the previous result (using 000 in Data(0-15)) and add this to the intermediate result by using 010 applied to Data(8-23). For each row related to a mask size, intermediate results are organized form right to left and successively computed from Data(0-15), Data(8-23), Data(16-31), Data(24-39) and Data(32-47). Each cell in the sub-rows corresponds to each mask size when it is computed by using the defined {imm8} datum. Although the arrangement can be extended to bigger masks of size $4 \times N$, we only have included the cases from 4 to 32.

4. TEMPLATE MATCHING IN GPU

A GPU has many processors or cores that can be suitably arranged to fit better for a specific problem. The advantage of GPUs is the massive number of cores, e.g. 2x240 in a NVIDIA GTX 295, but its drawback is the access to memory of such big number of cores. The memory is organized in different types: global, constant, texture and shared, but each type is accessed by using a single port, therefore the serial part and the bottleneck of the algorithms programmed in GPU is the memory access. According the Amdahl's law this is the factor that limits the efficiency of this massive parallel system.

Figure 1. Computational arrangement for fast pattern matching. Data inboxes is the *imm8* value

The main decision in the GPU programming is the design criterion of how memory will be used. We have decided to place the mask data in constant memory and the image and the result data in global memory. The image is constant in the algorithm but it does not fit in the small constant memory of the GPU. The CUDA programming methodology [Nvi09a] allows arranging the processors as a grid of threads. In our problem, the grid of threads is configured by assigning a thread to each result pixel at (x, y) excluding the border band; that is, each thread is involved in the computation of a $R(x, y)$ result. To reduce the negative effects of memory access, the mask data are placed in constant memory because this type of memory is cached and is smaller in size than the mask data. Also, we take advantage of broadcast access to that memory type because for (u, v) values of the mask indexes all the threads access to the same $M(u, v)$, which takes advantage of the broadcast access to constant memory.

In the developed code, the access to global memory is coalescent but unaligned due to the sliding. This would require the access to two consecutive data block in the same half ward, depending on the sliding value of v. To avoid the decreasing in performance of unaligned access, NVIDIA documentation [Nvi09a] suggests the use of texture memory instead of global memory, but we have not experimented any increasing in the performance when allocating the image data in texture memory and the mask in constant memory. At this point, the provided results are the related to the data $\boldsymbol{D}$ being placed in global memory, with coalescent but unaligned access, and the mask $\boldsymbol{M}$ being in constant memory with cached and broadcast access. To achieve the computation of the SAD, the unsigned integer version of the CUDA function *sad(a,b,c)* was used, where *a* becomes the value of the sliding image in global memory, *b* the value of the mask in constant memory and *c* the value of the serialized computation of the result.

To increase the efficiency in GPU, streamed calls to memory transfer and kernel launches have been used by splitting the image in several areas, non overlapping in result data and overlapping in image data. The interleaving between data transfer and kernel computation allows hiding the time used in data transfer between device and host. For this streamed asynchronous data transfer, the optional pinned memory allocation was used. The last included improvement is the use of this methodology to feed data and kernel launches in the two GPU contained in the same graphic card.

5. RESULTS

To test the implementations of raw template matching, images of 640×480 pixel have been used. Mask sizes range from 4×4 to 32×32. Both image and mask are single-channel with a pixel data of 8 bit unsigned int. The test computer is a Core2 Quad Q8300 with 4 GB of RAM memory and one NVIDIA GTX 295 graphic card. The Operating System is Windows XP and the code has been written in C++ in Visual Studio. Computational times are obtained by using performance counters of Windows, the reported values are the average on one hundred runs. Table 1 contains the results for four different implementations. The first one is C plain with low performance, but that is used as the baseline to provide the speedup for higher performance

implementations. The second implementation uses SIMD instruction in the 1D matching in which is based in the 2D. Also, for these two implementations the use of multiple cores is included by using OpenMP [Cha08a] parallelism. The 2D matching is implemented by row oriented 1D matching primitive, an *omp parallel for* directive of OpenMP is used for the row loop. The static schedule strategy is used, so the computation of rows is assigned to each core at the thread forking. Four threads are used for this four core system. The speedup (Column Sp) reported is the SIMD implementation with OpenMP in relation to the serial C plain one. Core based parallelism is more effective when 32×32 mask is used because the latencies of the threads forking are less relevant in bigger tasks than in the lowest associated to the small 4×4 mask due to the effectiveness of threads level parallelism is greatly dependent on the computational grain size. A remarkable speedup value of 184 is achieved for 16×16 mask when the four core of the system are used and also their high specialized SIMD vector units.

	1 Core/ 1 Thread		4 Cores / 4 Threads		
Mask	Cp	SIMD	Cp	SIMD	Sp
4x4	11.1	0.7	2.7	0.6	18.5
8x8	33.1	0.8	8.8	0.2	165.5
12x12	66.4	2.4	27.2	0.9	73.8
16x16	110.6	3.4	29.0	0.6	184.3
24x24	231.3	6.5	60.0	1.3	177.9
32x32	388.6	10.5	101.0	2.3	168.9

Table 1. Time in msec. for Template Matching in 640x480 images. C plain (Cp), SIMD and Speedup (Sp) for the 4 Cores case are included.

Table 2 contains the results for the implementations using one GPU. It contains the host to device (HtoD) data transfer, that in this case means, the transfer of mask from host memory to constant memory and the image transfer to global memory. This table also includes the kernel computation and finally the device to host (DtoH) transfer of the result to host memory. The result is coded as unsigned int which is better for the SAD computation but increases the data transfer time. However, this decision is not relevant in bigger mask sizes. Speedup columns are related to the kernel part (Sp1) or the total time (Sp2) over the C plain simple case. Figure 2 shows the graphical representation of these values. In the 4×4 case, data transfer is the critical subtask, while for the 32×32 case the kernel computation is highly more significant than the data transfer.

The final test that has been carried out includes other computational advantages of the GPU. The first one is the use of the second GPU included in the GTX 295 graphic card by means of splitting the whole image in two parts and by loading each part to each GPU. This methodology is extended by splitting the image in many parts and transferring each one to the GPUs in different threads context. This is implemented by thread forking in OpenMP in a number of threads defined by the user and using the *cudaSetDevice* function to select the device. Pinned memory is used to accomplish streamed asynchronous memory transfer and kernel launches. Table 3 contains the results for several mask size, and Figure 3 illustrate the results for 32×32 mask by using synchronous and asynchronous memory transfer. The 2 thread case, which uses a task in each GPU, is the best result followed by the 4 threads, which includes two tasks in each CPU. The asynchronous case that hides part of the memory transfer cost is the best case, but it cannot overtake the SIMD implementation.

Mask	HtoD	Kernel	DtoH	Total	Sp1	Sp2
4x4	0.4	0.3	1.0	1.7	37	6
8x8	0.4	0.7	1.0	2.1	47	16
12x12	0.4	1.5	1.0	2.9	44	23
16x16	0.4	2.7	1.0	4.1	41	27
24x24	0.4	5.8	1.0	7.2	40	32
32x32	0.4	10.1	1.0	11.5	38	34

Table 2. Results in msec. for Template Matching in one GPU. Speedup 1 (Sp1) is Kernel time and Speedup 2 (Sp2) is Total time, both related to 1 Core C plain case.

6. CONCLUSIONS

Modern computer hardware allows the computation of raw template matching in few milliseconds for small mask sizes. The use of ROI can reduce this computational cost to sub milliseconds scale. This fact is an opportunity to revalorize the developing of template matching applications. The use of many cores or many GPUs are different options to consider, but nowadays, the many-core approach, which includes specialized vector SIMD instructions, is more computational efficient because it is high specialized in SAD arithmetic and in-hardware sliding of mask across data array.

Technological improvements are in the line of increasing the number of cores in host. This trend is not well suited for small masks because they do not use effectively the OpenMP parallelism, but in bigger masks they can greatly increase the performance of

template matching procedures. The evolution of GPU architectures, e.g. the new NVIDIA Fermi architecture, will introduce full global memory cached access that can improve the kernel part of the GPU procedure as well as an increase in the number of involves cores. However, this can be improved only in the performance of big mask sizes whereas in smaller ones it will depend on the increase of the bandwidth of memory transfer between the host and the graphic card.

	4x4		16x16		32x32	
Th	S	A	S	A	S	A
1	2.6	2.3	5.1	4.7	12.5	12.1
2	2.4	2.1	3.7	3.3	7.4	7.1
3	5.8	3.6	7.5	5.1	12.6	10.4
4	5.9	3.7	7.3	4.8	11.1	8.7
5	6.2	4.7	9.3	6.0	14.0	10.7
6	6.4	5.0	7.8	6.0	13.8	10.3
7	7.8	5.9	9.5	7.3	14.2	12.0
8	7.9	6.2	9.4	7.3	13.7	11.3

Table 3. Results in msec. for 2 GPUs and OpenMP using different threads number (Th) from 1 to 8. The Sync (S) and Async (A) cases are included for several mask sizes.

7. REFERENCES

[Bra08a] Bradski, G. and Kaehler, A, Learning OpenCV, Computer Vision with the OpenCV Library, O'Reilly, 2008.

[Bru09a] Brunelli, R, Template Matching Techniques in Computer Vision, Theory and Practice, John Wiley and Sons Ltd, 2009.

[Cha08a] Chapman B., Jost, G and van der Pas, R., Using OpenMP Portable Shared Memory Parallel Programming, MIT Press, 2008

[For02a] Forsyth, D.A., and Ponce, J., Computer Vision: A Modern Approach, Prentice Hall, 2002.

[Har01a] Hart, P.E., Duda, R.O., Stork D.G., Pattern Classification, John Wiley and Sons, 2001.

[Int09a] Intel 64 and IA-32 Architectures Software Developer's Manual, Vols 2A-2B, Intel Corporation, 2009

[Kin07a] Kim, H.Y, and Araujo S.A., Grayscale template-matching invariant to rotation, scale, translation, brightness and contrast, IEEE Pacific-Rim Symposium on Image and Video Technology, Lecture Notes in Computer Science, 4872:100-113, 2007.

[Kua07a] Kuah, K, Motion stimation with Intel streaming SIMD extension 4. Technical report, Intel Software Solution Group, 2007

[Luk81a] Lukas B.D. and Kanade, T., An iterative image registration technique with application to stereo vision, Proceeding of the 1981 DARPA Image Understanding Workshop, pp 121-130, 1981.

[Nvi09a] NVIDIA CUDA, Programming Guide, Version 2.3.1, NVIDIA Corporation, 2009.

[Sha01a] Shapiro, L., and Stockman G., Computer Vision, Prentice Hall, 2001.

[Teh88a] The, C.H., and Chin, R.T., On image analysis by the method of moments, IEEE Trans. on Pattern Analysis and Machine Intelligence, 10(4):496-513, 1988.

[Ull04a] Ullah, F., and Kaneko, S., Using orientation codes for rotation-invariant template matching, Pattern Recognition, 37:201-209, 2004.

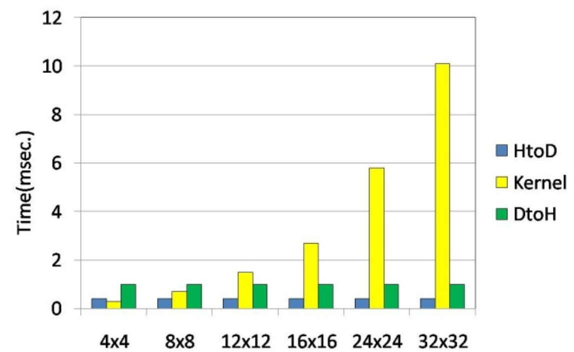

Figure 2. Kernel and Data transfer in GPU, memory copy from host to device (HtoD), device to host (DtoH) and kenel matching

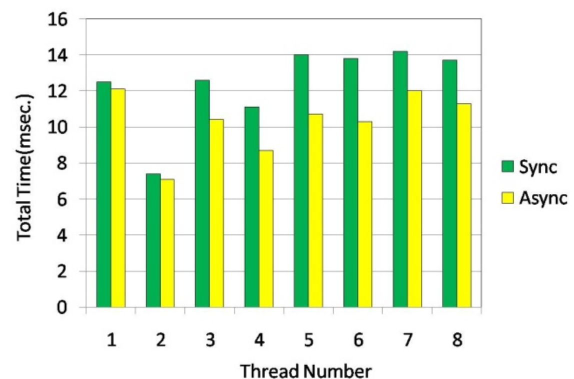

Figure 3. Total time for 2 GPUs and 32x32 mask matching. Asynchronous memory transfer was used with interleave between data transfer and kernel launches.