

# Fast GPU-based Space-Time Correlation for Activity Recognition in Video Sequences

Mahsan Rofouei, Maryam Moazeni, Majid Sarrafzadeh  
University of California Los Angeles  
{mahsan, mmoazeni, majid}@cs.ucla.edu

## Abstract

*Action recognition is becoming an important component of many computer vision applications such as video surveillance, video indexing and browsing. However most of the space time approaches to action recognition are very computationally expensive which prevents us from using them in real-time applications. This paper describes how Graphic Processing Units (GPUs) can be used in the field of action recognition to speed up this process. We implement a space-time behavior based correlation scheme on NVIDIA Quadro FX 5600 GPU and gain a 50x speedup over its counterpart CPU implementation.*

## 1. Introduction

Recognizing human action is becoming an increasingly important component in many computer vision applications. Video indexing and browsing, video surveillance, gesture and gait recognition are some of such applications. Recently a number of approaches have promised to detect and classify motion using spatial-temporal features of video segments. These approaches attempt to perform recognition through video features that are robust to appearance.

Space-time approaches to action recognition include the work of [6, 7, 8]. Distribution of Space-time gradients collected from a video clip is used in [8] to detect single actions at any given time. [6] examines intensity variations in small video patches to characterize motion and identify similarity between two different intensity patterns. In [7] by integrating the use of local space-time features with SVM they present a method for action recognition.

While these spatial-temporal approaches have demonstrated their effectiveness and accuracy in human action detection and classification, most of them require very large amounts of computation in both

extracting features and searching for similarities. For instance, applications like video search by example require online correlation similarity computation which is very time consuming and impossible in using these approaches for real-time applications.

Fortunately, most of these algorithms consist of lots of completely independent processes to be performed on each pixel of the video sequences. This property motivates the idea of using several simple processor units in parallel to fulfil the task much faster. In order to demonstrate this idea, we will use GPU's computational power for speeding-up one of the powerful and reliable approaches of this kind.

General-purpose computing on graphics processing units supports a broad range of applications including video compression, image processing, medical imaging, physical simulation and other scientific domains. Video processing applications are well-suited candidates for further exploration in the area of GPGPU (General-Purpose computing on Graphics Processing Units). Recently, there have been several contributions exploiting the SIMD model of GPUs for accelerating video related algorithms. In the context of video compression, an implementation of Block Matching motion estimator on graphics hardware is presented in [1] which its computational speed increased with a factor of 10 when compared to its CPU implementation. Additionally, a significant improvement in speed, processor load and algorithmic quality was observed when adding the results to an MPEG Encoding implementation [1]. In another work done in the computer vision domain, an implementation of KLT feature tracking and SIFT feature extraction algorithms are ported to GPU [2]. In both cases, computation is divided between the CPU and GPU to overcome the restrictions of the GPUs computational model. These two algorithms exploit the SIMD computational model of GPUs and results in considerable speedup. [3] is another work in implementing feature tracking algorithms on graphics

processors. In this work, it is shown that the GPU-based algorithm allows an order of magnitude more features to be tracked in real time compared to the CPU-based algorithm.

Prior to the emergence of GPGPU, FPGAs were popular for accelerating video processing applications. A comparison of GPU and FPGA approaches is presented in [4] with case studies including primary color correction and 2D convolution. The comparison concludes that video processing algorithms with high memory usage are recognized to be not suited for acceleration on GPU. In contrast, limitation of FPGA is the area requirement for complex arithmetic functions, floating point arithmetic and storage for filters of large mask sizes utilized in most video processing algorithms. On the other hand, rapid improvements of modern GPU have resulted in significant speedup by comparing successive GPU models.

A further comparison of GPU and FPGA implementation undertaken in [5] is demonstrated with case studies including Monte Carlo simulation. The comparison concludes that a GPU solution outperforms an FPGA solution. It is clear from various results presented in different domains that any conclusion regarding this issue depends solely on the specific application.

The rest of the paper is organized in the following way. Section 2 describes the method in [6] for action recognition. We will focus on this method in this paper and explore how it can be moved to GPU for acceleration. Section 3 first gives an overview on GPU architecture and then discusses our GPU implementation of the space time behavior based correlation. Experimental results are presented in section 4 and finally conclusions are drawn in section 5.

## 2. Space-time behavior based correlation

Shechtman et al have introduced a space-time behavior based correlation approach to measure similarity between two video segments [6]. In this approach a small video template is correlated against a larger video sequence. Using space-time intensity information in two video segments, a behavioral correlation volume is computed. Peaks in this correlation volume indicate similar motions and activities despite differences in appearance and texture.

ST-patches are defined as small Space-Time patches (e.g.  $7 \times 7 \times 3$ ) in video sequences. Intensity patterns are computed within these ST-patches. For each point in the video sequence a small space-time patch is centered around it and compared against its

corresponding ST-patch in the template video to compute a correlation score. These scores of similarity are added to find a global correlation score for each point in the video sequence. Figure 1 demonstrates how patches are compared between the template and video sequence.

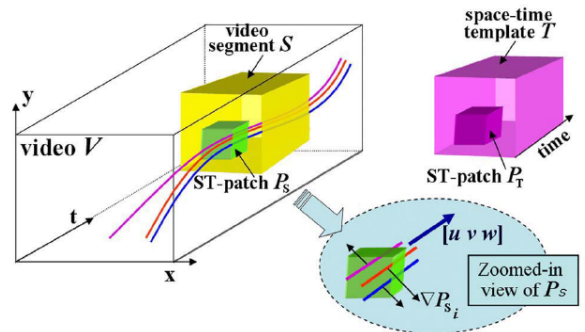
If ST-patches are chosen small enough all intensity lines within it can be assumed to orient in a single direction and move with a single uniform motion [6]. Therefore computing gradients of intensities in both space and time within a patch and stacking them up in a matrix named G, will be perpendicular to the orientation of the patch (u, v, w). In the below formula, n is the number of pixels with patch P and (u, v, w) is the orientation of the patch.

$$\begin{bmatrix} P_{x1} & P_{y1} & P_{t1} \\ P_{x2} & P_{y2} & P_{t2} \\ \dots & \dots & \dots \\ P_{xn} & P_{yn} & P_{tn} \end{bmatrix}_{n \times 3} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix}_{n \times 1} \quad (1)$$

Hence, for a patch with consistent motion i.e, parallel equal intensity lines, matrix M, called the Gram matrix of G, is a  $3 \times 3$  matrix that can be shown as below:

$$M = G^T G = \begin{bmatrix} \sum P_x^2 & \sum P_x P_y & \sum P_x P_t \\ \sum P_y P_x & \sum P_y^2 & \sum P_y P_t \\ \sum P_t P_x & \sum P_t P_y & \sum P_t^2 \end{bmatrix} \quad (2)$$

Therefore, calculating the rank of matrix M is sufficient to investigate the motion consistency in the patch. For patches with consistent motion, this matrix is rank deficient (  $\text{rank}(M) \leq 2$  ). Therefore patches for which the corresponding M matrix has rank of 3 imply no coherent motion or multiple independent motions.



**Fig. 1 ST-patches in video from [6]**

Similarly, the consistency of motions in two ST patches from two different videos can be checked by simply concatenating the Gram matrices and investigating the rank of the resulting  $M_{12}$  matrix. Note that actual intensity values of the pixels are not important here and only gradients are taken into consideration.

Furthermore, the same approach can be used between two different patches. Two ST-patches are considered consistent if there exists a common  $(u, v, w)$  that satisfies equation (1) for matrix  $G_{12}$  which contains all space-time intensity gradients for both ST-Patches. Similar to the above approach matrix  $M_{12}$  which contains pure intensity information can be used to detect motion consistencies between two ST-patches by checking its rank.

It is explained in [6] why the rank-increase  $\Delta r$  between  $M_{12}$  and its  $2 \times 2$  upper-left minor  $M_{12}^\diamond$  is a good metric for defining whether  $P_1$  and  $P_2$  are motion consistent.  $\Delta r$  being zero implies consistency while value one reveals inconsistency. The below formula shows rank increase in terms of eigen vectors.

$$\Delta r = \frac{\lambda_2 \cdot \lambda_3}{\lambda_1^\diamond \cdot \lambda_2^\diamond} \quad (3)$$

A good match between the video template  $T$  and a video segment is a match which maximizes the number of motion consistent patches as well as minimizing the number of local inconsistent matches. The measure below addresses these two issues.

$$m_{12} = \frac{\Delta r_{12}}{\min(\Delta r_1, \Delta r_2) + \varepsilon} \quad (4)$$

Therefore, using the above formula a global consistency metric for a video segment and a template is defined as  $C(T, S) = \frac{1}{N} \sum \frac{1}{m_{12}}$ , where  $n$  is the number of

space-time pixels in  $S$ . Hence, a correlation volume can be computed by sliding  $T$  against the video sequence in which the peaks determine similarity. As mentioned above, calculating different values in the correlation volume is completely independent and can be done in parallel.

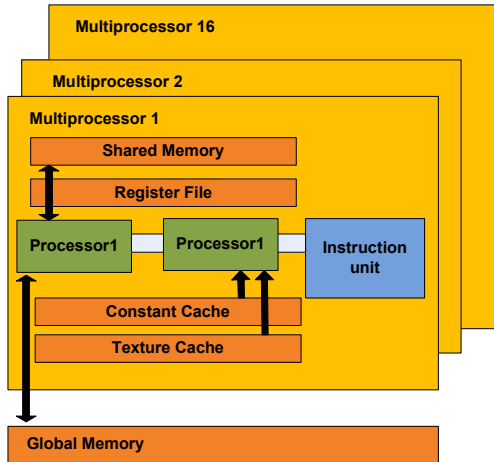
### 3. GPU acceleration

#### 3.1. GPU architecture

The NVIDIA Quadro FX 5600 is used as our main experimental platform. Quadro FX 5600 is a large set of processor cores with the ability to directly access a global memory, which allows a more flexible programming model than the previous generation of GPUs. FX 5600 is based on the G80 graphics processing unit architecture first introduced in NVIDIA'S GeForce 8800 GTS and GTX graphics cards. It has 128 processor cores supporting the Single Program Multiple Data (SPMD) programming model, which is more general and flexible than the programming models supported by previous generations of GPUs. This model allows data-parallel algorithms to be well suited for this architecture. CUDA was introduced by NVIDIA as a set of development tools to ease the developments on this architecture.

In this section we discuss the architectural features of G80, which are most relevant to our implementation. CUDA is a C-compiler that allows programmers to code algorithms in a data-parallel programming model [10]. CUDA gives developers access to the native instruction set and memory of the massively parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA'S GPUs effectively become powerful and programmable architectures like today'S CPUs. Unlike CPUs however, GPUs have a parallel Many-Core Architecture; each core capable of running thousands of threads. CUDA treats GPU as a coprocessor that executes data-parallel functions, so called kernels functions. The source program is divided into host (CPU) and kernel (GPU) code, which are then compiled by the host compiler and NVIDIA'S compiler (nvcc).

Figure 2 demonstrates Quadro'S architecture. The G80 GPU consists of 16 streaming multiprocessors (SMs), each containing eight streaming processors (SPs). Each SM has 8,192 registers and 16KB of on-chip memory that are shared among all threads assigned to the SM. The threads on a given SM'S cores execute in SIMD fashion, with the instruction unit broadcasting the current instruction to the eight cores. Each core has a single arithmetic unit that performs single-precision floating-point arithmetic and 32-bit integer operations.



**Fig. 2 Architecture of Quadro FX 5600**

The Quadro has 76.8 GB/s of bandwidth to its off-chip global memory. Bandwidth to off-chip memory is quite high, but can be saturated if many threads request access within a short period of time. This bandwidth can be achieved only if accesses to the memory are contiguous 16-words lines; therefore, it is very important to follow the right access pattern to get maximum memory bandwidth. In order to reduce the application's demand for off-chip memory bandwidth, there are several on-chip memories that can be employed to exploit the data locality and data sharing. As mentioned earlier, each SM has a 16KB shared memory for data that is either written and reused or shared among threads. The Quadro has a 64 KB, off-chip constant memory. The constant memory space is cached, therefore, a read from constant memory costs one memory read from global memory only on a cache miss, otherwise it only costs a single read from the constant cache. Each SM has 8 KB of constant memory cache. Finally, for read-only data that is shared by many threads but not necessarily accessed simultaneously by all threads, the off-chip texture memory and the on-chip texture caches exploit 2D data locality to substantially reduce memory latency.

The batch of threads that executes the kernel on the G80 is organized as a grid of thread blocks. Each kernel creates a single grid, which consists of many thread blocks. Each thread block is assigned to a single SM for the duration of its execution. The blocks that are processed by one multiprocessor in one batch are referred to as active blocks.

GPUs have very limited resources available to each thread. Thus, the more resources consumed by each thread, less number of threads can be active simultaneously which results in tremendous performance loss. Although, by using more resources

in each thread we can increase the performance of each thread individually, but it eventually reduces the degree of parallelism.

### 3.2. Space-time behavior based correlation on GPU

The approach discussed in section 2, can be used in content based video search where a user enters a query (template video) containing a certain behavior and looks for similar actions in a big database of videos. This process consists of two parts: first, computing the  $M$  matrices for the pixels of both query and video independently and second computing joint  $M12$  matrices which are simply the summation of two  $M$ 's and calculating the correlation volume. Since the first stage for large video sequences in the database is independent of the query, it can be done offline and stored ahead of time in the database (e.g. whenever a new video is added to the database). But computing Gram matrices for query and also the second stage has to be online. Note that the size of matrix  $M$  is independent of patch sizes and is always  $3 \times 3$ . Therefore a  $3 \times 3$  matrix needs to be stored for each pixel. Then different queries can be used either for indexing the database videos or real-time search. Finding the Gram matrix for the query and the  $M12$  matrix however need to be performed online. The Gram matrix of the query can be computed very quickly due to the small size of the query. [6] provides details on how to approximate the computation of rank increase of  $M12$ .

Computing the correlation surface for a given query is computationally very expensive and highly parallel. We use the graphic processing unit for this task. Each of the Gram matrices of each pixel of video and template are needed. The already computed Gram matrices are transferred to GPU device memory. As described in 3.2, access to GPU global and texture memory is relatively slower than shared memory. Therefore having to access these memories for 18 entries (9 for the video sequence and 9 for template) per  $M12$  computation would come at a great communication cost. On the other hand, the size of the shared memory is limited to 16KB per streaming multiprocessor. Therefore a good memory structure is needed to minimize access time.

Since the template video is small, and is needed for computing the  $M12$  matrix of each of the video pixels. We store a copy of the Gram matrices of query pixels in each of the streaming multiprocessor's shared

memory. This way we guarantee fast access to them while computing correlations.

Once patches are used to compute the Gram matrices for all the pixels we do not need to treat the video as 3D data and the M12 matrix can be computed independently. Therefore, in our implementation we process the video data, frame by frame. During the online process, we compute the Gram matrices of the query and transfer them to the GPU device memory at the beginning. Then we send the Gram matrices of all the pixels of one video frame to the GPU and invoke a kernel. This kernel will compute the correlations for one video frame and send the results back to the CPU. This way there is a possibility of overlapping the communication delay of writing data to GPU with work needed to be done on its previous frame, which we will discuss further on.

To further benefit from shared memory each video frame is divided to small blocks that would fit into the shared memory. Each kernel is responsible for computing the correlations for each of these blocks. The number of threads per kernel is equal to the number of elements of the block. In the beginning each block thread transfers its own Gram matrix to the shared memory. One possible implementation for computing correlations is using all the block threads to compute the correlation of one video location with the query simultaneously and compute the block correlations one by one. This method however, needs an extra reduction operation on values resulting from correlating the video location with each of the query locations. However, each thread being in charge of computing the correlation of one video location with the query omits the needs of an extra reduction step and results in a better speedup. Our implementation uses the second approach described above. We transfer and save the query pixels into shared memory block by block. Then each thread block computes the correlation of one video location with each block of the query.

Once the correlations of all the video locations are computed, we need to find the maximum correlation location in the whole video. Every kernel finds a local maximum of the block using a reduction operation and sends the data back to the CPU. Finding the global maximum is the job of CPU. CPU is in charge of computing the maximum among the local maximums per frame and in the whole video segment.

Since frames are treated independently the process of finding the maximum among the local maximums of the previous frame can be done on the CPU while the GPU is finding local maximums of the next frame. This further helps the acceleration of the whole process.

The above approach describes the exhaustive search implementation of the algorithm. In the exhaustive search approach the correlation of every video pixel with the query is computed. However in most of the cases, we know that the correlation volume is smooth and we can find the maximum using a hierarchical procedure. Hence in the hierarchical implementation, we start the calculation of the correlations on a sparse grid and then proceed by improving the resolution around the peak.

We implemented the hierarchical version both in C and in CUDA. For implementing the iteration stages of the hierarchical approach we invoked multiple kernels. The first kernel finds the maximum in a sparse grid of the data and sends the results back to the CPU. A second kernel is invoked to search for the maximum around the previous stage maxima. This time the data we need to send per frame is relatively small since we only need the data around the peak.

## 4. Experimental Results

Our experiments were conducted on a PC with 4 GB RAM, Intel Core 2 Duo, 3 GHZ processor with a single NVIDIA Quadro FX 5600 graphic card. This GPU contains 16 streaming multiprocessors.

We implemented both an exhaustive and hierarchical version of the algorithm in CUDA and compared their runtimes with their counterparts in C running on the CPU. Table 1 shows the running times and speed up of the exhaustive search version of the algorithm both on CPU and GPU, while Table 2 shows these results for the hierarchical implementation. Note that all communication delays for copying data from the CPU to GPU and vice versa have been taken into account.

**Table 1. Running times for exhaustive implementation.**

Query Size	Video Size	Running Time on CPU (s)	Running Time on GPU (s)	Speed up
40 × 30 × 15	144 × 192 × 150	31160	623	50
60 × 30 × 15	144 × 192 × 200	58015	1163	49.7

**Table 2. Running times for hierarchical implementation.**

Query Size	Video Size	Running Time on CPU (s)	Running Time on GPU (s)	Speed up
40 × 30 × 15	144 × 192 × 150	436	21.8	19.9
60 × 30 × 15	144 × 192 × 200	929.2	45.4	20.43

As mentioned, Table 1 shows the results of an exhaustive search implementation, however in reality it is enough to correlate for every other pixel and every other frame, and then interpolate [6]. So for example the running times for the entries of the first row of Table 1 will be 64 minutes on CPU while only one minute and half running on the GPU. Tables 1 and 2 show a speed up of 50x in the exhaustive search implementation and 20x on the hierarchical implementation. The current running times might still be large to use in real time query-based video search but can have a huge affect in video indexing using queries.

## 5. Conclusions and future work

In this paper we have presented how graphic processing units can help speedup space time action recognition algorithms. We implemented the space time behavior based algorithm described in [6] on a Quadro FX 5600 NVIDIA GPU using CUDA programming model and achieved a 50x and 20x speedup on two different implementations of the algorithm. The space time behavior based algorithm is chosen as an example to show the amount of parallelism in activity recognition algorithms and how well they fit to GPU architecture.

In our current implementation we use the GPU to accelerate computing the correlations of video locations and query in order to find similar behavior. This is the part that needs to be computed online. Since computing the Gram matrixes of video segments and query can be done prior to searching, it does not affect the speed of computing the correlation volume. However Gram matrix calculation is also computationally expensive and can be further speeded up by the GPU.

## 6. Acknowledgement

We thank Hamed Pirsiavash from the department of Computer Science in University of California, Irvine for providing us with input data and helping in the

implementation of the algorithm in C. We also thank him for his helpful comments and reviews on the paper.

## 7. References

- [1] S. Mazaré, R. Pacalet and J.-L. Dugelay, "Using GPU for fast block-matching", *Proceedings of EUSIPCO 2006, 14th European Signal Processing Conference*, 2006.
- [2] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys and Yakup Genc, "GPU-Based Video Feature Tracking and Matching", *EDGE 2006, workshop on Edge Computing Using New Commodity Architectures*, May 2006.
- [3] J.M. Ready and C.N. Taylor, "GPU Acceleration of Real-time Feature Based Algorithms", in *2007 IEEE Workshop on Motion and Vision Computing*, Feb. 2007.
- [4] B. Cope, P. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs Redundant in the Field of Video Processing?" in *Proc. IEEE Field Programmable Technology*, vol.1, Dec.2005, pp.111–118.
- [5] L. Howes, O. Pell, O. Mencer, and O. Beckmann, "Accelerating the development of hardware accelerators," in *Proc. Workshop on Edge Computing*, North Carolina USA, 2006.
- [6] E. Shechtman and M. Irani, "Space-time behavior based correlation", in *Proc. CVPR*, 2005.
- [7] C. Schuidt, I. Laptev, and B. Caputo, "Recognizing human actions: a local SVM approach", in *ICPR*, pages III: 32-36, 2004.
- [8] L. Zelnik-Manor and M. Irani, "Event-based analysis of video", *CVPR*, 2001.
- [9] I. Laptev and T. Lindeberg, "Space-time interest points", *ICCV*, 2003.
- [10] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, V. 1.0, 06/01/2007.