

# Real-time 3D Video Processing Using Multi-stream GPU Parallel Computing

Kenia Picos, Víctor H. Díaz-Ramírez, Juan J. Tapia

Instituto Politécnico Nacional, CITEDI  
Avenida del Parque 1310, Mesa de Otay, Tijuana, B. C., México  
kpicos@citedi.mx, vdiazr@ipn.mx, jtapiaa@ipn.mx

**Abstract.** This work presents a real-time video processing algorithm for 3D scenes using a graphics processor. The processing is based on parallel computing using concurrent kernels. The proposed algorithm processes individual pixels of each pair of input stereo images to obtain an anaglyph image for each frame. To reduce the computational time, a concurrent kernel implementation using POSIX threads and CUDA streams is utilized. Also, an asynchronous streams execution is used to increase overlapping of the video processing implementation. The obtained results are presented and discussed in terms of speedup and execution time.

**Keywords:** graphics processing units, parallel computing, CUDA streams, image and video processing.

## 1 Introduction

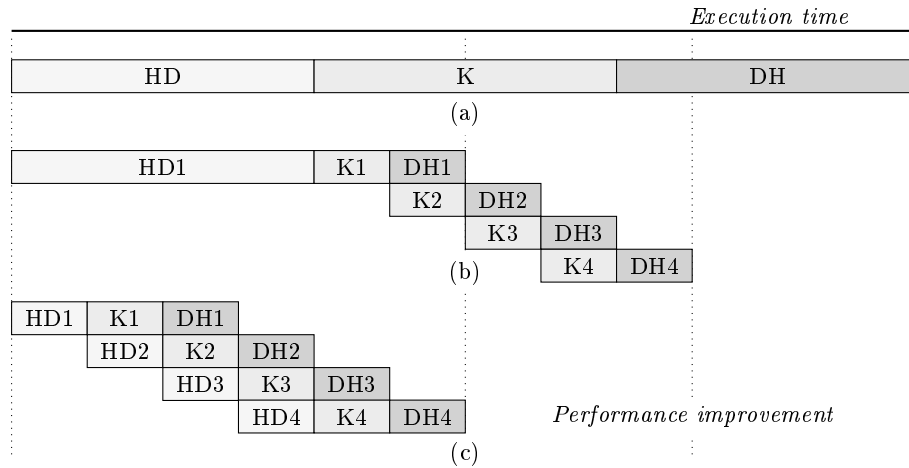
Nowadays, the use of parallel computing to solve image and video processing issues in real-time has been rapidly growing [1,7]. Due to the increasing popularity of modern processor designs based on hierarchical memory structure, the use of graphics processing units (GPU) has increased significantly due to their great performance and runtime efficiency [13]. In the state-of-art of image and video processing, there are several algorithms that has been implemented in a parallel platform with an unified device architecture [14]; which results in an increase of the execution performance of the algorithm. In recent applications, a sequential synchronization of the loop iterations within the algorithm is not enough to take full advantage of the benefits of the parallel architecture, by obtaining only a little performance increase with respect to a sequential execution. This is because there are some time gaps in the use of the processing cores, due to synchronization issues. To overcome this shortcoming, an asynchronous algorithm can be used [5]. In this scenario, data communication and the effective use of processing cores are executed asynchronously, avoiding unnecessary waiting times due synchronization, and allowing overlapping in the execution of scheduled tasks [2]. In recent years, a considerable progress has been made within the context of 3D video processing applications. Nowadays, 3D video is one of the latest innovations in information technology [10]. Current

hardware/software technology uses stereo video data to monitor functions as anaglyph display, disparity depth maps computation, and high quality signal processing. In a framework of computer vision technologies, the extraction of 3D object information from 2D images can be carried out by employing modern 3D techniques, such as shape from shading, motion tracking, stereo vision, and many others. Applications such as 3D television and cinema consist on a pair of 2D image sequences which are processed to give us a 3D perception of a scene [9]. A special monitor or glasses are needed to recreate this perception. A 3D video data can be easily generated with a stereo camera, so every observation is interactively changed. However, there are basic limitations that constrain the viewing direction, produce errors when the object presents pose changes, and the execution of the algorithm is inefficient. The last issue can be alleviated by using streams in a GPU implementation.

In the present work, a multi-stream GPU implementation for real-time video processing is proposed. A real-time 3D video processing algorithm is implemented on a GPU with parallel computing using multiple streams. We are focused on improving the performance of 3D video processing, particularly by taking advantage of the asynchronous memory of the GPU. Results obtained with computer simulations using a graphics processor are presented and discussed in terms of computational throughput, and runtime efficiency. The paper is organized as follows. A brief study of multi-stream computation is presented in section 2. The proposed real-time 3D video processing implementation is described in section 3. In section 4, results obtained with the proposed approach are presented and discussed. Finally, section 5 summarizes our conclusions.

## **2 Multi-stream GPU Computation**

The Compute Unified Device Architecture (CUDA) is applied in hardware and software for managing operations in a GPU as a data parallel computing device using an extension to C programming language. This architecture was designed to exploit massive parallelism in graphics processors, and to use concurrent streams for single or multiple GPUs, asynchronous data transfers, and simultaneous kernel executions on a single device. CUDA creates `stream0` by default with the execution of one GPU [11]. Kernel invocations and data transfers are lined up on a stream and processed sequentially in the order they were queued [15]. The creation of multiple streams of execution can perform more work per unit time and make applications to run faster. The tasks are queued on each device (GPU), which can potentially increase the application performance by the number of GPUs installed in the system. Kernel executions can overlap data transfers and data computation to yield real-time processing and reduce the overall runtime on one or more GPUs. In order to improve efficiency on a single device, concurrent kernel executions can be carried out using multiple streams [11]. In contrast to synchronous kernel executions, asynchronous kernel executions are not coordinated within the processor, in where the transmission of messages are allowed to be performed in an unspecified order [4]. Miellou and



**Fig. 1.** Concurrent execution of an algorithm using streams. (a) Sequential version. (b) Concurrent version for device-to-host asynchronous memory copy. (c) Concurrent version for host-to-device and device-to-host asynchronous memory copy

Baudet [5] formalized asynchronous algorithms, and described a generalization of an approximation method which includes an inherent parallelism. The main advantages of asynchronous executions are: a) bottleneck reduction, b) synchronization penalty reduction, and c) convergence improvement [5].

### 3 Real-time 3D Video Processing Using Streams

Concurrent applications can be supported from device by creating a CUDA context for each GPU used in the system. This context can be determined by `cudaStream_t stream[Ns]`, where  $N_s$  is the total number of streams. This context includes driver information and capability, such as streams, events, virtual address space, and blocks of memory [12]. With the intercommunication of the context, the device can handle multiple tasks and multiple GPU applications in order to expand parallelism [8]. Several image and video processing applications exploit concurrency through streams. A contribution of this work is the use of a multi-stream approach for implementing stereo disparity estimation and anaglyph video frame generation on GPU in a more efficient way. A sequence of commands that are executed in a sequential order are called streams. These streams are often called by different host threads or POSIX threads (pthreads). The main feature of streams, is that commands can be executed in any order or concurrently. In the present work, we divide the input image in several fragments, and process each fragment asynchronously in a different CUDA stream. This asynchronous behavior allows overlapping of task executions on the device. Fig. 1 shows a stream creation process. A stream is specified by a param-

eter that defines a sequence of kernel invocations including a host-to-device (HD) and device-to-host (DH) memory copy using `cudaMemcpyAsync()` function [11]. The creation of streams could be described with a specific CUDA expression `cudaStreamCreate(&stream[i])`, where  $i$  is the number of the current stream. These streams are allocated in a host array in the page-locked memory with `cudaMallocHost(&array, size)`. A kernel call queues the function invocation on the stream related with the current device. In general, a CUDA context for multiple stream execution [12] could be configured by the expression `kernel<<<blocks, threads, 0, stream[i]>>>(parameters)`. For concurrent executions within a device multiple streams are required, that is, multiple kernels can be executed concurrently on a single device [3].

The proposed implementation has been developed in a CPU/GPU architecture running on Linux OS with multi-core host processor (Intel i7) and an NVIDIA graphics processor GeForce GTX780 with 3.5 compute capability. An algorithm with pthreads and CUDA streams for 3D video processing is implemented. The main steps related to the present work consists on a) video capture, b) video processing, and c) video display. In Algorithm 1, we introduce a CUDA/OpenCV interoperability for video capture and display. Pthreads and streams creation is specified in Algorithm 2. Here, memory is copied asynchronously from host to device in order to compute the kernel that is described in Algorithm 3. Then, the memory is copied back from device to host. The anaglyph video frame is generated and displayed in the host using OpenCV framework.

---

**Algorithm 1** Video Capture Algorithm
 

---

```

1: procedure VIDEOCAPTURE(frame)                                ▷ input *.mp4 3D video file
2:   Capturing 3D video frames
3:    $fps = \text{capture properties}$ 
4:   Memory allocation in GPU
5:   while  $frame \neq 0$ , do                                       ▷ video capture is opened
6:     Separate  $L(x, y)$  and  $R(x, y)$  from frame
7:      $L_b = L_g = 0$ 
8:      $R_r = 0$ 
9:     Image=VideoProcessing( $L, R$ );                               ▷ processing kernel in gpu
10:  end while
11:  FreeCudaMem();                                               ▷ free memory allocation
12: end procedure

```

---

## 4 Results

In this section, results obtained with the proposed algorithm for 3D video processing are presented and discussed in terms of computational performance and throughput. The algorithm was implemented using parallel computing with pthreads and CUDA streams. In each frame, a pthread executes several CUDA

**Algorithm 2** Video Processing Algorithm

---

```

1: procedure VIDEOPROCESSING(L,R)                                ▷ input L, R video channels
2:   Pthread Create and Execute
3:   for  $i \leftarrow 1, Nd$ , do                                  ▷ Nd=number of devices
4:     for  $i \leftarrow 1, Ns$ , do                                  ▷ Ns=number of streams
5:       cudaStreamCreate(&stream[i]);
6:     end for
7:     for  $i \leftarrow 1, Ns$ , do
8:        $j = (i * S * Nd) + (tid * S)$ ;                            ▷ j is an offset
9:       cudaMemcpyAsync(dL+j, L+j, size, host→device, stream[i]);
10:      cudaMemcpyAsync(dR+j, R+j, size, host→device, stream[i]);
11:      kernel<<<grid,thread,stream[i]>>>(dL+j, dR+j, dI+j);
12:      cudaMemcpyAsync(I+j, dI+j, size, device→host, stream[i]);
13:    end for                                                    ▷ Stream destroy
14:  end for                                                        ▷ Pthread Join
15: end procedure

```

---

streams. The maximum number of streams executed per thread depends on the graphics processor architecture. A sample of video frames used in our experiments are illustrated in Fig. 2. Note that CUDA kernel has been used for processing input images, in order to obtain an anaglyph image for each frame. Fig. 2(a) and 2(b) shows the left and right channels of the input sequence. Fig. 2(c) shows the resultant anaglyph images obtained with the GPU. In the anaglyph images, different disparities are computed to present the displayed content. It can be seen that the red-cyan disparity depends on the distance of objects with respect of both channels. The disparity  $d$  is obtained from the sum of absolute differences (SAD), as follows:

$$SAD(x, y, d) = \sum_{i, j \in W(x, y)}^N |L(i, j) - R(i - d, j)|. \quad (1)$$

The intensity difference of both channels  $L$  and  $R$  is calculated for each pixel in a square window  $W(x, y)$  with origin at  $(i, j)$ -th pixel [6]. The area-based disparity is obtained from the sum of squared differences (SSD) of both channels, given by [6]

$$SSD(x, y, d) = \sum_{i, j \in W(x, y)}^N |L(i, j) - R(i - d, j)|^2. \quad (2)$$

The accuracy of the disparity map depends on the window size because there is a correspondence with the probability of matched points. In our experiments the size of the window where  $W$  is  $11 \times 11$  pixels. Note that the size of the window has a direct impact on the computational load of the proposed method. The kernel function used in Algorithm 3 is implemented with 1 to 8 pthreads and with 1 to 8 streams in each pthread. The input images are fragmented in rows and they are processed accordingly with the current thread and stream execution.

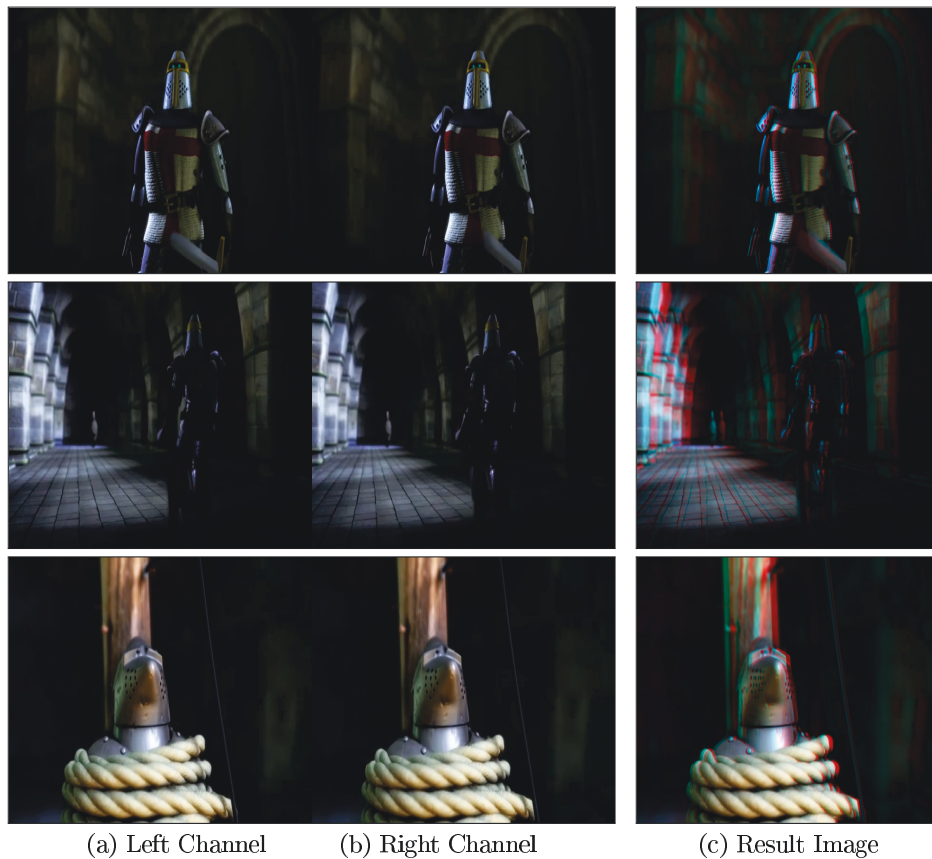
---

**Algorithm 3** Kernel function in GPU


---

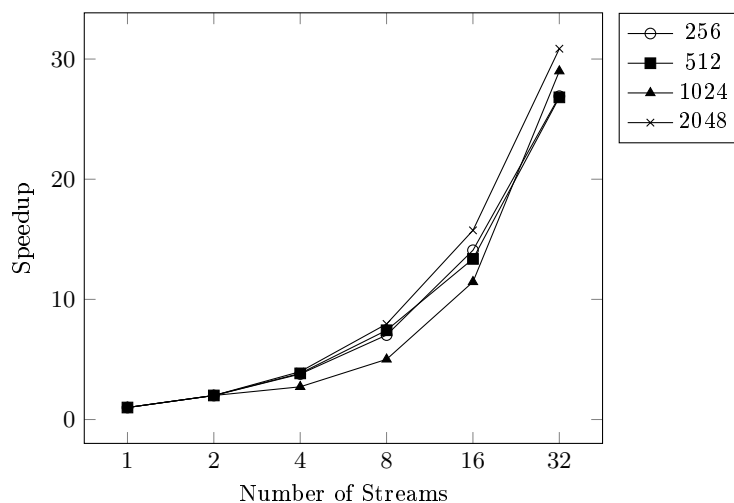
```
1: procedure KERNEL<<<GRIDS, THREADS, STREAM[i]>>>(L, R, Image)
2:    $x = \text{threadIdx}.x + \text{blockIdx}.x * \text{blockDim}.x;$ 
3:    $y = \text{threadIdx}.y + \text{blockIdx}.y * \text{blockDim}.y;$ 
4:    $tid = x + y * \text{blockDim}.x * \text{gridDim}.x;$ 
5:    $\text{Image}[tid] = (L[tid] + R[tid]);$ 
6: end procedure
```

---



**Fig. 2.** Left (a) and right (b) channels. (c) GPU result of anaglyph image at different frames of the sample video sequence (Video sequence from © RedStar Studios)

 3D red-cyan glasses are recommended



**Fig. 3.** Performance of the proposed algorithm for different image sizes

The number of grids and threads per block executed concurrently in the kernel depend on the number of streams and pthreads, given by

$$S = (N_x \times N_y) \times CH / (N_d \times N_s). \quad (3)$$

where  $S$  is the size of the fragment of the image processed on the current kernel. For each frame, the image fragment has the same size  $S$ , and no fragment overlaps with another.  $N_x \times N_y$  and  $CH$  are the size of the input image and number of channels (3 RGB color channels), respectively. The number of pthreads and the number of streams are represented by  $N_d$  and  $N_s$ , respectively. The index of a fragment is assigned according the order of call of POSIX and stream argument; however the frames are asynchronously processed. The performance of the algorithm given in terms of the number of threads and streams used in the image partition, is shown in Table 1. It can be seen that by increasing the number of streams per pthread, the speedup increases. The runtime of the kernel execution is 70% off the overall processing executions. The achieved occupancy is the ratio of active warps to the maximum active warps per processor, given by almost 50%. The sample video sequence contains 2000 frames of a set of stereo images with  $1024 \times 1024$  pixels, and 3 color channels (RGB). Fig. 3 shows the relation of the performance in terms of speedup and number of streams. Observe that with different image sizes, the performance increases more than 30 times when more streams are used.

## 5 Conclusions

In this work, a CUDA stream-based algorithm that increases parallelism for 3D video processing was proposed. The algorithm can be efficiently used for

**Table 1.** Computing results for a sample 3D video with 1024×1024 pixels

Threads	Streams per threads	Profiler Time (%)	Kernel execution (ms)	Speedup (times)	Achieved occupancy (%)	Memory stored Throughput (MB/s)	Memory loaded Throughput (GB/s)
1	1	72.63%	15.72	1.00	47.86%	883.64	211.50
1	2	72.48%	7.86	2.00	47.95%	874.76	209.37
1	4	72.49%	3.93	3.99	47.99%	855.03	204.61
1	8	72.32%	1.97	7.98	47.91%	804.08	192.42
2	1	71.90%	7.90	1.99	47.95%	872.57	208.85
2	2	72.19%	5.77	2.72	47.99%	815.72	195.20
2	4	72.51%	3.14	5.01	47.89%	804.19	192.45
2	8	72.98%	1.03	15.31	47.22%	781.47	187.01
4	1	71.06%	3.98	3.95	48.00%	820.25	196.28
4	2	71.83%	2.02	7.79	47.06%	803.98	192.40
4	4	72.80%	1.37	11.45	47.57%	781.66	187.05
4	8	74.33%	0.54	29.22	46.73%	748.43	179.10
8	1	70.29%	2.02	7.77	47.90%	804.09	192.42
8	2	71.69%	1.03	15.20	47.57%	781.45	187.00
8	4	73.68%	0.54	28.99	46.73%	784.60	179.14

real-time 3D video processing. In our implementation, several CUDA streams are executed asynchronously in where each one is able to process a fragment of the input image to compute an anaglyph image. According to computer simulation results, the proposed system yields high effectiveness in the execution of the video processing algorithm in terms on computing performance metrics. The proposed system was implemented in a GPU by taking advantage of massive parallelism. In our implementations, the proposed system achieves a processing rate above 100 frames-per-second (fps) in 3 color channel images of 1024×1024 pixels. The proposed approach performs well with occlusions, however may introduce artificial borders artifacts for a video sequence at speed higher than 70 fps. For future work, the proposed algorithm will be optimized for scenes when the visualization of the object of interest presents light reflections.

## References

1. Bhatti, A., Nahavandi, S.: Stereo correspondence estimation using multiwavelets scale-space representation-based multiresolution analysis. *Cybern. Syst.* 39(6), 641–665 (2008)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*. The MIT Press, 3rd edn. (2009)
3. Farber, R.: *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
4. Hwu, W.W.: *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2011)



5. J. M. Bahi, S. Contassot-Vivier, R.C.: *Parallel Iterative Algorithms*. Chapman and Hall/CRC (2008)
6. Kamencay, P., Breznan, M., Jarina, R., Lukac, P., Zachariasova, M.: Improved depth map estimation from stereo images based on hybrid method. *Radioengineering* 21(1), 70–78 (2012)
7. Kim, C., Zimmer, H., Pritch, Y., Sorkine-Hornung, A., Gross, M.: Scene reconstruction from high spatio-angular resolution light fields. *ACM Trans. Graph.* 32(4), 73:1–73:12 (2013)
8. Kirk, D.B., Hwu, W.W.: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edn. (2013)
9. Malik, A.S., Choi, T.S., Nisar, H.: *Depth Map and 3D Imaging Applications: Algorithms and Technologies*. IGI Global, Hershey, PA, USA, 1st edn. (2011)
10. Matsuyama, T., Nobuhara, S., Takai, T., Tung, T.: *3D Video and Its Applications*. Springer Publishing Company, Incorporated (2012)
11. NVIDIA: *CUDA C Best Practices Guide* (2014)
12. NVIDIA: *NVIDIA CUDA Programming Guide 6.0* (2014)
13. Pacheco, P.: *An Introduction to Parallel Programming*. Morgan Kaufmann (2011)
14. Rodríguez-Sánchez, R., Martínez, J.L., Cock, J.D., Fernández-Escribano, G., Pieters, B., Sánchez, J.L., Claver, J., de Walle, R.V.: 3D high definition video coding on a GPU-based heterogeneous system. *Computers & Electrical Engineering* 39(8), 2623–2637 (2013)
15. Sanders, J., Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional (2010)