RDVideo: A New Lossless Video Codec on GPU

Piercarlo Dondi[1], Luca Lombardi[1], and Luigi Cinque[2]

[1]Department of Computer Engineering and Systems Science, University of Pavia,
Via Ferrata 1, 27100 Pavia, Italy
[2]Department of Computer Science, University of Roma "La Sapienza",
Via Salaria 113, Roma, Italy
{piercarlo.dondi,luca.lombardi}@unipv.it,
cinque@di.uniroma1.it

Abstract. In this paper we present RDVideo, a new lossless video codec based on Relative Distance Algorithm. The peculiar characteristics of our method make it particularly suitable for compressing screen video recordings. In this field its compression ratio is similar and often greater to that of the other algorithms of the same category. Additionally its high degree of parallelism allowed us to develop an efficient implementation on graphic hardware, using the Nvidia CUDA architecture, able to significantly reduce the computational time.

Keywords: Lossless Video Coding, Screen Recording, GPGPU.

1 Introduction

Recording screen activity is an important operation that can be useful in different fields, from remote descktop applications to usability tests. In the first case is essential to reduce the amount of trasferred data, it is not strictly necessary a high quality video compression: a lossy compression and a small set of informations (e.g. the list of pressed keys or of started programs) are generally sufficient[1]. On the contrary, in the second case, it is generally useful to have a complete recording of the screen in order to study the user interaction.

There are many screen recording system on market [2] that implement different features, but all of them use traditional codec for compress the output.

In this paper we introduce a new method for lossless video coding, called RDVideo, based on Relative Distance Algorithm [3], whose peculiar characteristics make it particularly suitable for compressing screen video recording. In this field its compression ratio is similar and often greater to that of the other algorithms of the same category, like Lagarith [4], HuffYUV [5], FFV1 [6] or Alpary [7].

Additionally, its high degree of parallelism can be very useful for increasing its performances, in particular with a GPU implementation. The execution on the Nvidia CUDA architecture gives interesting results: the comparison between CPU and GPU showed the significant speed-up in elaboration time supplied by graphic hardware, both in compression and in decompression.

The paper is organized as follows: section 2 presents the description of the codec; its parallel implementation on CUDA is described in section 3; section

G. Maino and G.L. Foresti (Eds.): ICIAP 2011, Part II, LNCS 6979, pp. 158–167, 2011.

4 shows the experimental results, an analysis of compression ratio and a comparison of the performances between the sequential realization on CPU and the parallel on GPU; finally our conclusions in section 5.

2 Relative Distance Video Codec (RDVideo)

2.1 Relative Distance Algorithm (RDA)

The RDA is a lossless method for image compression that uses local operators to minimize the number of bits needed to memorize the data. The base idea of the algorithm is that, in a small area, also the color differences between pixel are generally small [8]. This subsection provides only a general description of the algorithm, with the aim of give the information needed for understanding the extension for the video. A more complete description can be found in a previous work [3]. Every channel of an image is compressed in the same way independently from the others, then we can describe the procedure for a single channel image without loss of generality.

At the beginning the image is divided in blocks of 8x8 pixels. In every block it looks for the maximum (max_p) and the minimum (min_p) pixel value; the found min_p is subtracted to every pixel p_i of the block (1).

$$p'_i = p_i - min_p \tag{1}$$

Each p'_i represents the distance between the element i from min_p − to memorize p'_i is necessary a number of bits less than or equals of that for p_i. We defined the *Relative Distance* (RD) of a block as the minimal number of bits required to represent all the p'_i of that block (2).

$$RD = \lceil log_2 (max_p - min_p + 1) \rceil \tag{2}$$

The compression is obtained coding the values p'_i with a number of bits equals to RD; the compression ratio is good only when RD is minor than 7 (RD is always included between [0, 8]), otherwise the data are saved uncompressed.

After this first analysis, every block is further split in four 4x4 sub-blocks and the RDA is applied again to each of these ones for checking if the subdivision can increase the compression ratio. Finally the block is saved with the better configuration.

Different headers identify the type of compression used (one 8x8 block, four 4x4 sub-blocks or no-compression). Every header, except in the no-compression case, contains the minimum (saved with 8 bits) and the RD (saved with 3 bits). The number of bits for a compressed block 8x8 is calculated by (3). Instead the dimension of a subdivided block is the sum of the size of the four 4x4 sub-blocks, each one compressed with different RD (4).

$$NBits_{8x8} = HBits + RD * 64 \tag{3}$$

$$NBits_{4x4} = \sum_{i=1}^{4} HBits_i + (RD_i * 16) \tag{4}$$

$$CR = UBS/NBits \tag{5}$$

The compression ratio (CR) is calculated by (5), where uncompressed block size (UBS) is 512 bits (64 pixels x 8 bits). The sequential implementation requires a time linear with the dimension of the image.

2.2 RDVideo Codec

The RDA gives the best performances when the RD is 0 or 1 (that means large homogeneous color areas, like sea or sky). So, even if generally the medium compression ratio of the algorithm is between 1.5 and 3, with a very uniform image it can grow to 10 or more (the theoretical limit is 42.73 with a monochromatic image). Clearly this situation is not very frequent, but in a video sequence consecutive frames are quite similar, in particular analyzing a screen video recording. Consider some standard tasks − e.g. a user that reads a text or copies a file from a folder to another − in all of them there are small changes between a frame and the next thus the variation of the scene depends only by the user's speed. Making the difference between two consecutive frames we can create a new more uniform image, called difference frame (D-Frame), on which we can successfully apply the Relative Distance Algorithm.

RDA is designed to work only on positive values, so before the compression, a D-Frame must be rescaled (6) adding to all pixels an offset (OF), in the two's-complement arithmetic. The result is saved on 8 bits. This operation can be inverted without loss of information using equation (7). The scale factor must have a value between [0, 256], our experiments show that the better results are achieved when it is between [90, 144].

$$DFrame_i = Frame_i - Frame_{i-1} + OF \tag{6}$$

$$Frame_i = Frame_{i-1} + DFrame_i + OF \tag{7}$$

When there is a change of scene a D-Frame does not give significant improvements. The frames that are very different from previous ones are considered intra-frames (I-Frame) and are compressed directly without other elaborations. Obviously the first frame is always an I-Frame.

The following pseudo code summarizes the main steps of RDVideo codec:

```
save(width, height, framerate);
compressedFrame = compress(frame[0]);
save(compressedFrame);
for (i = 1 to NumFrames)
{
     D-Frame = calcDifference(frame[i], frame[i-1]);
     D-FrameDim = evalCompressionRatio(D-Frame);
     I-FrameDim = evalCompressionRatio(frame[i]);
     if(D-FrameDim < I-FrameDim)
          compressedFrame = compress(D-Frame);
```

```
    else
        compressedFrame = compress(frame[i]);

    save(compressedFrame);
}
```

3 CUDA Parallel Implementation

The modern GPUs offer very high computation capabilities, that can be useful not only for 3D graphic. At the end of 2006, with chipset G80, Nvidia introduced CUDA (Computer Unified Device Architecture), a parallel computing architecture that allows to exploit the potentiality of GPU also in general purpose applications and provides a new parallel programming model and a proper instructions set [9].

3.1 CUDA Specifications

A CUDA GPU is a many core device composed by a set of multithreaded Streaming Multiprocessors (SMs), each of them contains eight Scalar Processor cores (CUDA Cores) for single precision floating-point operations, one unit for double precision, two special function units for transcendental functions and a multithreaded instruction unit. The multiprocessor employs a new architecture called SIMT (single-instruction, multiple-thread), capable to create, manage, and execute hundreds of concurrent threads in hardware with zero scheduling overhead. The GPU acts as a coprocessor for the CPU: the data on which it works must be transmitted from the CPU RAM (the host memory) to the GPU DRAM (the device memory), because the GPU cannot read or write directly on the hard disk. This is the real bottleneck of the system, a good CUDA program should minimize the data transmission.

The GPU has also other levels of memory: each multiprocessor supplies a set of local 32-bit registers and a very fast parallel data cache (or shared memory), a read only constant cache and a texture cache that are shared among all the processors. An appropriate use of all the memory levels can significantly improve the total performances of the program.

The function that is executed on the device by many different CUDA threads in parallel is called "kernel". There are two types of kernel: the global ones that are called from the host only, and the device kernel that are called from the device only. A kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block multiplied for the number of blocks. These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks. Thread blocks are executed independently, this allows to schedule them in any order across any number of cores. For this reason the code has a high degree of scalability [9].

Until the last year a GPU was able to execute only one kernel at time, this limitation is overcome by the most recent model of graphic cards that implement the new CUDA FERMI architecture [10].

3.2 RDVideo on CUDA

A CUDA implementation of RDA for a single image was discussed in a previous work [3]. The main difference with the implementation for the video concerns the amount of data to transfer from host to device, in this case very high. So the program has been redesigned, according to the characteristic of the graphic hardware, for minimizing the data transfer and to take advantage from all the memory levels of the CUDA architecture.

Compression. All the compression phases are distributed on three global and one device kernels. Only the final operation, the saving on hard disk, is entrusted to CPU. We start the description analyzing the simplest case, the compression of an I-Frame.

The GPU analyzes each block of the frame, for each of them it chooses the best type of compression (one 8x8 block or four 4x4 sub-blocks) and then compresses them as described in 2.2. The values of the current 8x8 block are saved in shared memory, in this way we can reduce the time needed for multiple accesses to the data; the use of the slower device memory is limited only for the initial reading of input data and for the final writing of the results.

In a CUDA program each block is executed in parallel with each other, there is no way to know the order in which they finish, so is very important the synchronization of the reading and the writing phases in order to avoid any kind of conflict. In this case reading phase is not critical − each processor operates on a different part of the image − more complicate is the management of the writing. The dimension of the compressed data is unknow a priori, it must allocate on the device memory enough space for each block in order to exclude multiple writing on the same memory address. For each frame the GPU has to give back to CPU all the data to be saved: the compressed values, the relative distances, the min_p arrays and a global array of flags that mark the type of compression used for each block. For definition, the last one has always a dimension equals to number of blocks of the image; more complex is determine how much space is needed for the other variables.

Taking advantage of the parallel structure of the graphic hardware we can concurrently calculate the compression for the 8x8 block and for the four 4x4 sub-blocks. The main kernel calls four times (in parallel) a device kernel, while continues the computation for the 8x8 block. The device kernel executes on its data the same operations of the global kernel and gives back the compressed dimension of its sub-block. The results of the two kind of compression are both saved in two different shared memory areas, after the evaluation of the compression ratio only the better value is stored in the output array (Fig. 1).

For the elaboration the GPU saves five RD and five min_p per block, so the system need two arrays with a size of 5*(number of blocks of the images). The dimension of all the arrays must be multiply for the number of channels in case of RGB image. For decreasing the device memory allocation, the compressed data overwrite the correspondent input data that are no longer used. This solution resolves also the issue of concurrent writings: each block writes only the same image area that has previously read.

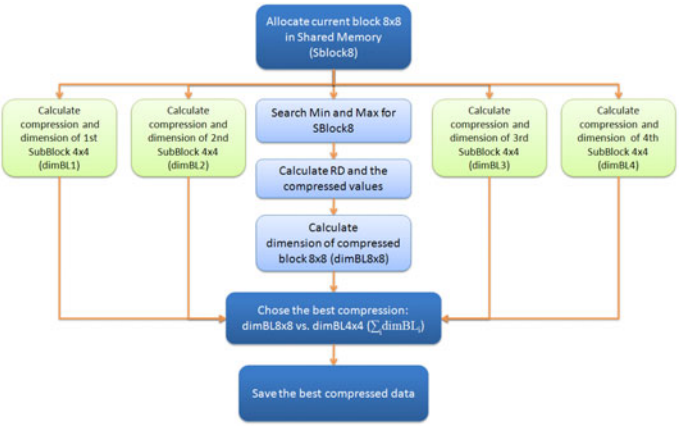

Fig. 1. Flow chart of Compression Kernel for an I-Frame

Summarizing, the steps for compressing an I-Frame are the following:

1. Allocation on device memory of 4 arrays (min_p, RD, flags, image)
2. Copy of the image from host to device
3. Launch of global kernel
4. Copy of all the arrays from device to host

Let us now examine the steps for a generic frame. Firstly a second global kernel creates the D-Frame: the difference between frames is a very simple operation that involves independent data, it fits well on CUDA and it does not required any particular optimization. The obtained D-Frame will be used only by GPU, so it is maintained on device memory and it is not copied on host.

At this point, in the sequential implementation, we check the compression ratio for D-Frame$_i$ and for frame$_i$, in order to calculate the compression only on the one that gives the best CR. This is a good choice for the CPU but not for the GPU, if we first check the dimension of frames and after compress the best one, we must transfer twice the same input data. A better procedure is making the compression during the CR evaluation: a subtraction of independent data is a very fast operation on GPU, so it is more efficient performing a few more operations if it reduces the transfers. Consequently the compression process for a D-Frame follows the nearly steps described in figure 1 for an I-Frame, but the input image is already present on GPU. In the same way works the compression of the frame$_i$.

The trasfer phase requires a specific optimization. The copy on host of both results (compressed D-Frame$_i$ and compressed frame$_i$) is a waste of time; but storing both on the device and transfering only the best one is unsuitable, because with high resolution videos there is an elevated risk to exceed the memory space.

We implement a solution that tries to reach a good balance between memory occupation and the amount of transferred data:

1. Compress D-Frame$_i$.
2. Copy on host all the results (CR and compressed data).
3. Compress frame$_i$.
4. Copy on host only the CR of frame$_i$.
5. If the CR of frame$_i$ is better copy all its data.

Statistically a D-Frame is better than the correspondent I-Frame, so in the most of cases we have only one copy from device to host.

Finally a last optimization: to create a D-Frame the current frame$_i$ and the previous frame$_{i-1}$ are needed, saving the current frame$_i$ in the texture memory it will be already available as frame$_{i-1}$ at next iteration. In addiction accessing the texture memory is faster than reading the device memory. So, making the copy before step 3 of the previous list, also the compression of the frame$_i$ becomes faster. Consequently, as mentioned at the beginning, there is a third global kernel that executes the compression using as input the texture memory rather than the device memory.

Decompression. The decompression is computationally simpler than the compression and also requires less memory transfers. Only the input data must be copied from CPU, the decompressed video can be directly showed on screen by GPU. For this task we need two type of global kernel: one for I-Frames and another for D-Frames. The first simply adds the correspondent minimum to the compressed values of every block. The second makes the same operations and also recreates the original frame$_i$ adding to the just decompressed D-Frame$_i$ the frame$_{i-1}$. To reduce the transfer rate a strategy similar to that used for compression is exploited: the decompressed frame$_i$ is saved in the texture memory to be used again in the next iteration in case of D-Frame. This copy is more efficient than the previous one: for compression the program must send the image from the host memory to the device texture memory, now it makes a copy of the frame$_i$ between two different areas of the GPU memory.

Summarizing, the decompression for a generic frame is composed by four steps:

1. Copy of compressed frame$_i$ from CPU to GPU.
2. Launch of the proper kernel (for D-Frames or for I-Frame).
3. Saving of the decompressed frame$_i$ in texture memory.
4. Visualization on the screen.

4 Experimental Results

The proposed approach is useful for different tasks, but we focused our experiments on screen video recording. In order to test the performances of our algorithm we analyzed different kind of situations, in particular two typical human-computer interactions: a statical use (like reading and writing text with an editor) in which there are very small variations in screen image; and a dynamic use (like opening and moving folders and files on screen) in which the

screen changes significantly. The tests are made on an Intel Q9300 CPU with 2.5 GHz and a Nvidia GeForce GTX 260.

The video are recorded at different resolutions (the first five at 1680x1050, the sixth at 1280x1024 and the seventh at 800x600), the duration depends on the type of interaction but it is always in the range of 20-30 sec. Sequence 1, 2, 3 and 7 present a dynamic use, as opening and closing windows and programs or copying files between folders. Test 5 and 6 record a static use, respectively writing with an editor, and playing with a casual game (Solitaire). Finally test 4 considers a particular case, that is partially dynamic and partially static: web navigation with multiple pages visualized.

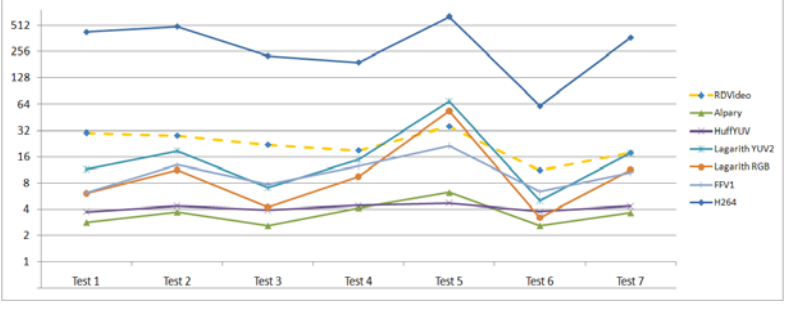

Fig. 2. Compression ratio − comparison between different codecs (RDVideo is the dotted line)

Figure 2 show a comparison of the compression ratio between our method and other standard lossless codecs. In all the videos our method has a good compression ratio, at most only the h264 [11] has a greater compression ratio.

4.1 Execution Times

Compression. For compression we considered the computation time and the total time (computation time plus the time for reading the video input and writing the compressed output). Figure 3(a) shows the medium elaboration time to compress one frame, while figure 3(b) shows the total time to compress the entire sequences. As expected, the outcomes present an evident difference of performances for the computation time in favour of the GPU (Fig. 3(a)).

It must point out that the gap between GPU and CPU is not constant but it is greater in the first five tests where the screen resolution is higher. This is a consequence of the graphic hardware architecture: the GPU has better performances when there are lot of data to analyze, with small frames there is no way to reach the full load of the stream multiprocessors, so a high resolution video fits better than a little one. However, in spite of this improvement, the total time shows a lower enhancement (Fig. 3(b)). This issue, generated by the bottleneck of the data transfer between CPU and GPU, can be overcome considering long recordings. In fact at growing of video lenght the elaboration speed up

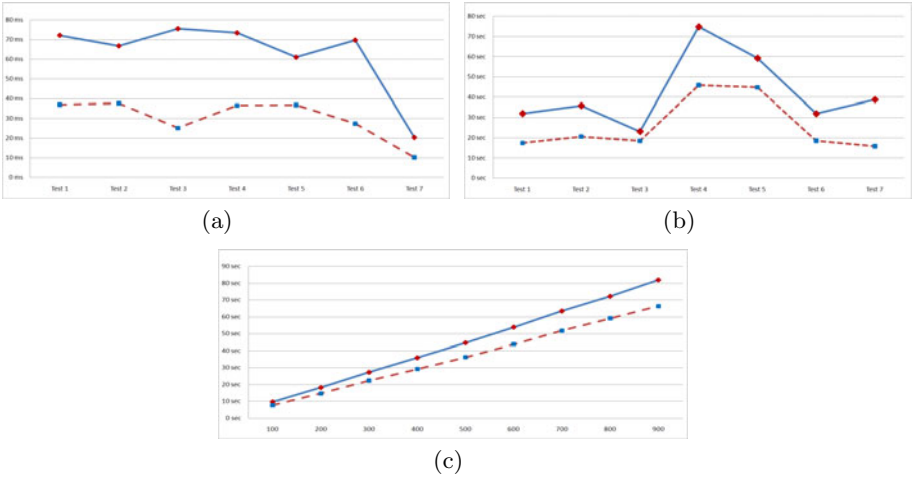

Fig. 3. Compression: (a) medium elaboration time for one frame (in ms); (b) total time for an entire sequence (in sec); (c) total compression time for the same sequence at growing of number of frames − CPU (continuous line) vs. GPU (dotted line)

supplied by graphic hardware generates a global improvement of performances that reduces, slowly but constantly, the total compression time for the GPU (Fig. 3(c)).

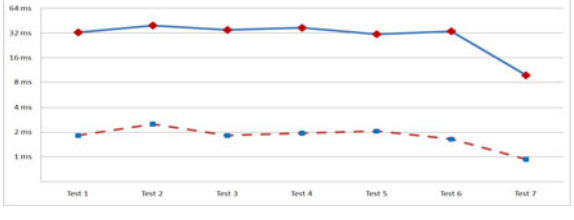

Fig. 4. Medium elaboration time for decompressing one frame: CPU (continuous lines) vs. GPU (dotted lines)

Decompression. For decompression we check only the elaboration time because the total time is function of frame rate, so necessarily they must be similar both for CPU than for GPU. Due to the reduced numbers of operations to execute, the elaboration times are lower than those of the compression, while the difference between CPU and GPU becomes greater (Fig. 4). These outcomes proves that a real-time decompression is always reachable with RDVideo codec, also for high screen resolutions video recordings.

5 Conclusions

In this paper we have presented a new lossless video coding, particularly efficient for the compression of screen video recordings. The experimental results show a compression ratio between 11 and 35 both in static and in dynamic situation, that makes our method a viable alternative in this field to other standard codecs.

The high parallelism of RDVideo codec allowed us to realize an implementation on CUDA architecture that provides a good enhancement of performances for the elaboration time in compression and especially in decompression. The tests have also proved that our method performs better with high screen resolutions and long recordings.

A future update for the codec includes the introduction of a predictor that can be used in addition to D-Frame in order to further improve the compression ratio.

References

1. Shann-Chiuen, W., Shih-Jen, C., Tzao-Lin, L.: Remote Screen Recording and Playback. In: International Conference on Systems and Networks Communications, ICSNC 2006 (2006)
2. Fast, K.: Recording Screen Activity During Usability (2002), http://www.boxesandarrows.com/archives/recording_screen_activity_during_usability_testing.php
3. Bianchi, L., Gatti, R., Lombardi, L., Cinque, L.: Relative Distance Method for Lossless Image Compression on Parallel Architecture. In: Fourth International Conference on Computer Vision Theory and Applications (VISAPP 2009), vol. 1, pp. 20–25 (Febraury 2009)
4. Lagarith Codec, http://lags.leetcode.net/codec.html
5. Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. In: Proceedings of the I.R.E., pp. 1098–1102 (September 1952)
6. FFV1 Codec, http://www1.mplayerhq.hu/~michael/ffv1.html
7. Alparysoft Lossless Video Codec, http://www.free-codecs.com/download/alparysoft_lossless_video_codec.htm
8. Storer, J.A.: Lossless Image Compression Using Generalized LZ1-Type Methods. In: Data Compression Comference (DCC 1996), pp. 290–299. IEEE, Los Alamitos (1996)
9. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. Elsevier, Amsterdam (2010)
10. Nvidia Corporation, Cuda Programming Guide, http://developer.nvidia.com/object/gpucomputing.html
11. Sullivan, G.J., Topiwala, P., Luthra, A.: The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. In: SPIE Conference on Applications of Digital Image Processing XXVII (August 2004)