RELATIVE DISTANCE METHOD FOR LOSSLESS IMAGE COMPRESSION ON PARALLEL ARCHITECTURES
A New Approach for Lossless Image Compression on GPU

Luca Bianchi, Riccardo Gatti, Luca Lombardi
Computer Vision Lab, University of Pavia, Via Ferrata 1, Pavia, Italy

Luigi Cinque
Department of Computer Science, University of Roma "La Sapienza", Via Salaria 113, Roma, Italy

Keywords: Parallel, Lossless image compression, CUDA, GPU, Relative distance.

Abstract: Computer graphics and digital imaging have finally reached the goal of photorealism. This comes however with a huge cost in terms of memory and CPU needs. In this paper we present a lossless method for image compression using relative distances between pixel values belonging to separate and independent blocks. In our approach we try to reach a good balance between execution time and image compression rate. In a second step, by considering the parallel characteristics of this algorithm (and nonetheless the trend of multi-core processor), a parallel version of this algorithm was implemented using Nvidia CUDA architecture.

1 INTRODUCTION

Computer graphics and digital imaging have finally reached the goal of photorealism. However such result has been paid with huge costs in terms of memory requisites and CPU time, especially for large size images. By the way, there are always a lot of redundancies in a sequence of bits representing a non-compressed image. Finding them and converting sequences of bits into a shorter form is the type of approach of data compression algorithm like LZ (Ziv, 1977). The Relative Distance Algorithm (RDA) presented in this paper, uses a similar approach by removing redundancy of data and translating the sequence of image's bits in a shorter form. The compression provided by RDA is lossless.

RDA first divides the image into separate blocks and then elaborates them independently, both in the compression and decompression phase (in this paper it is proposed the use of an 8x8 pixel window). Thanks to this characteristic it is quite easy porting it to a multi-processor architecture just by making every single processor working on a different block in parallel.

Multi-core architectures are becoming more and more common even on consumer markets. Every new processor is equipped with at least two cores (Q series of Intel and AMD Phenom has up to 4 cores; IBM Cell processor has 9 cores). GPUs evolution has been even faster: for example a recent Nvidia G80 chipset can be considered up to a 16 processor device (each one with SIMD capabilities). Classic algorithms that used to work on a single processor machine need therefore to be arranged to take advantage of all this new computational capabilities. Many efforts have been done so far to optimize the execution of lossless compression algorithms on different kinds of parallel architectures: LOCO-I Jpeg-LS (Ferretti, 2004), EREW-PRAM block matching (Cinque, 2007), with Huffman and Arithmetic Coding on PRAM model (Howard, 1996) and with Burrows-Wheeler Transform on NUMA system (Gilchrist, 2007).

As already noticed RDA is born with parallel concepts clear in mind. To show the real advantages of a parallel implementation we compared the execution time between a single-core and a multi-core architecture. We have chosen a G80 graphic card by Nvidia supporting CUDA for drawing a comparison.

2 RELATIVE DISTANCE ALGORITHM (RDA)

The Relative Distance algorithm is a lossless method for compression that uses local operators to find a way to rearrange data in a different mode. The main idea is to analyze the local chromatic characteristics of an image and compress them according to their peculiarities. This is made in O(N) time in the sequential form of the algorithm. The algorithm operates separately on every single channel (RGB) of the image. Of course it can be applied to single-channel images too.

Let’s consider for example a non-compressed 24 bit bitmap image. In a lot of images it can be observed that the representation of the colour (RGB) follows local features. For example in a picture of a landscape we can identify some areas with minimal colour change (e.g the sky) and areas with large colour difference in particular near the edge of objects. In general it can be assumed that the difference between all pixel values in a small area is usually small (Storer, 1997).

The first step is to divide every single channel of the image in blocks of dimensions *n* x *n* (with *n* = 8 every block is made of 64 values in the range [0, 255]). For each block is then applied RDA and estimated the compression rate (see details in next section). Then the block is split in 4 blocks of 4x4 pixels and again the RDA is applied for the new ones to check if this operation is able to produce a better compression. The data produced by the algorithm are saved in a single stream that contains all results of every single block in sequence. Two different types of header (formally called DR64 and DR16) are used to identify 8x8 blocks from 4x4 blocks sequences. It’s important to note that the header of each compressed block contains enough information to evaluate the block size. This is very helpful during read operations. Some compression rates showing the capabilities of Relative Distance approach are reported in section 4.

2.1 Compression of a Single Block

Let’s consider a single 8x8 block of the image. A pixel matrix **P** is defined by this block as follows:

$$\mathbf{P}' = \mathbf{P} - (M_{8x8} \cdot \min_p) \quad (1)$$

$$\min_p = \min_{i=0..63} p_i \quad (2)$$

$$p'_i = p_i - \min_p \quad (3)$$

In equation (1) every element of M is 1. Every p’$_i$ represents the distance between the element *i* from min$_p$. The last step consists in finding the necessary bits to represent each p’$_i$:

$$\max_{\log 2} = \max_{i=0..63}\left[\log_2(p'_i)\right] \quad (4)$$

The block is then reconstructed with the values of p’$_i$ coded with max$_{\log2}$ bits. This value is formally known as “block distance”.

If $0 \le \max_{\log2} \le 2$ (the maximum distance is in [0, 4]) the algorithm proceed with DR64 header recording. Otherwise the block is divided into 4 sub blocks of 4x4 pixels and the algorithm loops back for each new subset. In this way the algorithm checks if a smaller relative distance (and therefore a better compression) can be obtained through a smaller subdivision. Then the results are compared and the best type of coding (DR64 for one 8x8 block or DR16 for four 4x4 blocks) is chosen. If both types of coding do not provide advantages, the block is leaved as it is and a special header (NC64) is used to identify it among the non-compressed block.

All the information gathered for each block 8x8 of the starting image is then recorded in a binary file. In this file the ordered sequence of all blocks can be found starting from the upper left corner to the lower right corner of the image.

2.2 Header DR64

DR64 header contains the specifics of compression of all the 64 values of an 8x8 block. The header looks like this:

1	XXX (3 bit)	Min$_p$ (8bit)

where the three x values contain the result of equation (4) and Min$_p$ contains the minimum value of the block. The first bit is set to “1” to recognize the start of the header. Therefore DR64 length is 12 bits. After that there are the distances of all 64 pixel belonging to the block expressed with max$_{\log2}$ bit.

Table 1 resumes the necessary bits for recording a block depending on the minimum value found. The necessary bits for a block are calculated with (5) and compression ratio with (6) where uncompressed block size is 512 bits (64x8).

$$\text{necessary bits} = (\text{DR64 bits}) + \text{distance} * 64 \quad (5)$$

$$\text{c.ratio} = \text{uncompressed block size} / \text{nec. bits} \quad (6)$$

2.3 Header DR16

In DR16 header, as already mentioned, there is the need to record the information of four different blocks, each one with its own minimum value and relative distances. DR16 header is used only if it provides a better compression (requires less bits)

Table 1: Dr64 distances coding and compression rates.

Distance	Coded Distance	Necessary Bits	Compression Ratio
0	000	12	42.73
1	001	76	6.73
2	010	140	3.65
3	011	204	2.51
4	100	268	1.91
5	101	332	1.54

than DR64. Table 2 resumes all the possible combination of DR16 header signatures. The correct coding of DR16 is guaranteed by the uniqueness of the code created. DR16 header looks like this:

0	d_{b1}	d_{b2}	d_{b3}	d_{b4}
Min_1 (8 bit)				
Min_2 (8 bit)				
Min_3 (8 bit)				
Min_4 (8 bit)				

where d_{bi} fields (the distance of each sub-block) have different type of values according to Table 2. The first bit is set to "0" for recognize DR16 from DR64. Min_i contains the minimum value for each 4x4 block. The total size of the 8x8 block is the sum of the size of the four 4x4 block each one compressed with different compression ratio.

Table 2: d_{bi} fields in DR16 header.

Distance	Coded Distance d_b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111100
8	111101

2.4 Header NC64 (No compression)

If compression with DR64 or DR16 does not provide any benefit, the best thing to do is to leave the block 8x8 as it is. It must be preceded anyway with a special header. The format is the following:

1111	Value1	Value2	Value3	…	Value64

2.5 Pseudo Code

This section contains a pseudo code example for RDA implementation. Remember that this procedure is the same for all blocks and must be done for all the channel (RGB) of the image separately.

```
For(channel = 1 to N) {

for(block[8x8] = 1 to N){
   maxDist8x8=findMaxDist(block[8x8]);
   If (maxDist8x8 < 4) {
      recordDR64();
   }
   else {
      numBitDR64 =
         (DR64headerBit)+64*maxDist8x8;
      numBitDR16 = 0;
      for(block[4x4] = 1 to 4) {
         maxDist4x4=findMaxDist(block[4x4]);
         numBitDR16 +=
            (DR16HeaderBit)+16*maxDist4x4;
      }
      if(numBitDR16 > numBitDR64)
         recordDR64();
      else
         if(numBitDR16 < sizeNotCompressed)
            recordDR16();
         else
            recordNC64();
      }
   }
}

}
```

In this example code findMaxDist(block[]) function returns the results of equation (4). "DR64headerBit" is always 12 (according to Table 1), "DR16headerBit" depends on the distance (according to Table 2) and "sizeNotCompressed" is the size of the block without compression (8 bit * 64 pixel).

3 RDA ON CUDA

The characteristics of Relative Distance algorithm makes it very suitable for an implementation on a multi-processor machine. Its main quality is the fact that every block is totally independent from the others. Therefore we can work, both in compression and in decompression stage, on more than one block in parallel. In theory, the more processors we have the fast will be the algorithm steps. For providing some benchmark of the algorithm in a multi-processor situation, a porting for G80 Nvidia graphic cards featuring CUDA (Computer Unified Device Architecture) was realized.

3.1 CUDA Specifications

Today's GPUs offer incredible resources for both graphics and non-graphics processing. The GPUs in particular are well-suited for problems that can be expressed in a data-parallel form. CUDA architecture was introduced by Nvidia on G80 graphic card series as a data-parallel computing device for managing computations on GPU (Nvidia 2008). The GPU is considered as a device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU (or host) but it has its own DRAM (referred as device memory) separated from the CPU RAM (referred as host memory). Transfer of data between them is made by high-performance DMA access.

The device is implemented as a set of SIMD (Single Instruction Multiple Data) multiprocessors. Each multiprocessor supplies a set of local 32-bit registers and a parallel data cache (or shared memory), a read only constant cache and a texture cache that are shared between all the processors.

A function that is executed on the device as many different threads is called "kernel". In the parallel version of RDA every kernel will contain all the steps required for compressing or decompressing a single block of pixels.

3.2 Optimize RDA for CUDA

For using CUDA full computing capabilities, there is the need to write a general "kernel" that operates on different sets of input data. Every kernel is loaded in a thread and a grid of threads is then sent to the GPU multiprocessors for execution. Reference to input data could be made by using thread IDs. In this way a multiprocessor that is running a thread with ($\text{Thread}_{\text{IDx}}$, $\text{Thread}_{\text{IDy}}$) is working to the block having coordinates (X,Y). Since CUDA usually works in an almost random order, there is not the certainty that the blocks will be executed in a pre-established way. As a consequence the file containing the compressed data must be organized in an appropriate way.

In a standard sequential form of the algorithm (working on a single processor) the file is organized as a list of blocks (each one with its correct header) and the program reads, executes and writes every single block starting from the first to the last one.

In random parallel execution condition we need to read from memory the correct set of input data related to each block of coordinates (X,Y). Since all blocks have different size in bits due to different compression rate and header length, there is the need to arrange the order of the file both in compression and decompression phase. Let's consider compression stage first. The steps of the algorithm are:

1. Load uncompressed data to device memory.
2. Run RD kernel on each block.
3. Transfer results file to host memory.

During step 2 each processor has to share both input and output data. While reading input data do not create problems cause it is performed in different and independent parts of the original image, the output data have a big problem: it is impossible to know the final data size of compressed block and therefore each thread does not know where to write down results (because we are in a random parallel thread execution). A way to solve this problem is to separate each block information in different but consecutive memory location:

1. Header type bit list (0 = DR64, 1 = DR16).
2. Blocks minimum values.
3. Blocks d_p
4. Pixel in every blocks expressed with d_p bits

In this way each processor knows exactly where its results have to be written, because the length of each one of this vector can be known a priori. All these vectors are created as unsigned char vectors and then cleaned out of useless data before starting memory transfer from device to host. Vectors #2 and #3 will be created with a size of 4 * (number of image's block). In this way it is possible to write down 4 different data if a DR16 header occurs.

As benchmarks highlight (section 4) memory transfer is a big problem in CUDA approach and takes more than 90% of total execution time in compression phase. Since it is better making a unique large transfer instead of a set of many small transfers, image data has to be moved from and to the device memory all at once. That's the reason because memory clean out of unnecessary data is very important before starting the transfer between devices. When compressed data is available to host memory a file is written maintaining the block information division. This helps in decompression phase where the first step is to reconstruct the four vectors. In fact decompression stage operates in a very similar way:

1. Load compressed data to device memory.
2. Division of the data in 4 vectors.
3. Run RD kernel on each block.

Again, by maintaining the division between block's information every thread reads different and precise memory locations without overlap memory access.

4 RESULTS

RDA has very few and simple execution steps that can provide very fast performance even on low-end computers. For benchmarking we used a standard set of images supplied by Kodak that range between every color sample and chromatic characteristic. These images reproduce different subjects and situations and can provide a good model of a generic collection of pictures. The file size of every original picture is 1.179.702 byte. As compression ratio (c. ratio) we define the amount given by the ratio between the original size and the compressed file size. On Table 3 compression ratio of RDA is compared with compression ratio of the well-known JPEG2000-LS (Christopoulos 2000).

Table 3: Compression ratio of RDA vs JPEG2000-LS.

Image	c. ratio RDA	c.ratio JPEG2000-LS
Kodim01	1.22	2.27
Kodim02	1.55	2.54
Kodim03	1.75	3.05
Kodim04	1.52	2.51
Kodim05	1.23	2.13
Kodim06	1.40	2.45
Kodim07	1.58	2.85
Kodim08	1.19	2.10
Kodim09	1.60	2.67
Kodim10	1.57	2.65
Kodim11	1.44	2.56
Kodim12	1.63	2.83
Kodim13	1.16	1.93
Kodim14	1.31	2.28
Kodim15	1.67	2.61
Kodim16	1.53	2.77
Kodim17	1.52	2.65
Kodim18	1.32	2.08
Kodim19	1.43	2.42
Kodim20	2.10	2.95
Kodim21	1.45	2.40
Kodim22	1.43	2.23
Kodim23	1.74	2.75
Kodim24	1.40	2.35

On average RDA gives a compression ratio of 1.49 while JPEG-LS gives a compression ratio of 2.41. The main reason of this result is that RDA is not adaptive to account for high detailed areas.

4.1 Execution Time

In table 2 are reported the total execution time for an operation of compression and for an operation of decompression of images with increasing resolution size. We considered for testing an Intel Q6600 CPU and an Nvidia 8800GTX with 768M on board ram.

Table 4: Execution time x86 vs GPU.

Image Size	Intel q6600		Nvidia 8800	
	Compr.	Decompr.	Compr.	Decompr.
640x480	100 ms	100 ms	38 ms	18 ms
800x600	170 ms	160 ms	58 ms	29 ms
1024x768	250 ms	220 ms	98 ms	37 ms
1280x960	380 ms	320 ms	147 ms	55 ms
1600x1200	560 ms	480 ms	226 ms	74 ms
1920x1440	770 ms	680 ms	326 ms	101 ms

Table 5: Nvidia 8800 detailed compression benchmark.

Image Size	Nvidia 8800 – Compression	
	Memory Transfer Time	Computation Time
640x480	36.63 ms	1.36 ms
800x600	56.90 ms	1.68 ms
1024x768	96.64 ms	2.21 ms
1280x960	143.50 ms	2.99 ms
1600x1200	221.92 ms	4.34 ms
1920x1440	320.91 ms	5.93 ms

As benchmarks highlight there is a great improvement in speed between sequential and parallel execution of the algorithm. However even in the sequential form, a compression of a 1920x1440 image takes only 770 ms and its decompression takes only 680 ms. This shows clearly how this algorithm is performing well on all architectures.
On Nvidia G8800 GTX a compression needs on average 50% time less. But as table 5 point out, the total time needed for computation is only a very small part if compared to the total execution time. Regardless, on CUDA architectures memory transfers from device to host are usually slower than memory transfers from host to device and this helps out to achieve an awesome total execution time of 101 ms for a decompression of a 1920x1440 image. The reason is that there is not need to send back the decompressed image to the main system memory since it's already in the framebuffer ready for visualization on screen. On Table 6 are reported the comparisons between RDA and JPEG2000-LS parallel GPU implementation (Wong, 2007). RDA clearly gives better performances than JPEG2000-

LS especially with low resolution images. RDA can perform the compression of around 20 image/second and the decompression of around 40 image/second if the resolution is 800x600.

Table 6: Benchmark on Nvidia G80: RDA vs JPEG-LS.

Image Size	Nvidia 8800GTX – Compression	
	RDA	JPEG2000-LS
640x480	38 ms	516 ms
800x600	58 ms	560 ms
1024x768	98 ms	600 ms
1280x960	147 ms	721 ms
1600x1200	226 ms	828 ms
1920x1440	326 ms	969 ms

This results shows that RDA approach is a good candidate for real-time compression and decompression problem.

5 CONCLUSIONS

Relative distance algorithm for image compression proved to be a good compromise between sufficient compression ratio and very fast performance thanks to its algorithm structure that suits very well to multi-core parallel architectures. Future works will address the problem of reaching a better compression ratio by making better compression rates even when the image has lots of non homogeneous areas. This can be done with the use of a dynamic block size.

Moreover RDA approach will be considered for inter-frame compression of movie files. It's not difficult to find sequences where the actors are speaking in front of the camera or the scene is quite stand-still. When this happen some areas are not changing too much their colours between frames. If we define blocks not on a single image but between frames of a video sequence, RDA can be applied on them quite easily. This application will be discussed in future works.

ACKNOWLEDGEMENTS

This work has been partially supported by FIRB Project RBIN043TKY.

REFERENCES

Christopoulos C., Ebrahimi T., Skodras A., 2000. The JPEG 2000 Still Image Coding System: an Overview. In *IEEE Transactions on Consumer Electronics*. Vol. 16, num. 4, p. 1103-1127.

Cinque L., De Agostino S., 2007. A parallel decoder for Lossless Image Compression by Block Matching. In *Proceedings of the 2007 Data Compression Conference*. DCC. IEEE Computer Society, p 183-192.

Ferretti M., Boffadossi M., 2004. A parallel Pipelined Implementation of LOCO-I for JPEG-LS. In *Proceedings of the Pattern Recognition, 17th international Conference on (Icpr'04)*. Vol. 1, p. 769-772.

Gilchrist J., Cuhadar A., 2007. Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform. In *Proceedings of 21st international Conference on Advanced Networking and Applications*. AINA. IEEE Computer Society, p. 877-884.

Howard P.G., Vitter J.S., 1996. Parallel Lossless Image Compression Using Huffman and Arithmetic Coding. In *Inf. Process. Lett.*, p. 65-73.

Kodak test images: *http://r0k.us/graphics/kodak*

Nvidia Corporation, 2008. *Cuda Programming Guide, Version 2.0*.

Storer J.A., 1996. Lossless Image Compression Using Generalized LZ1-Type Methods. In *Proceedings of the Conference on Data Compression*. DCC. IEEE Computer Society, p. 290-299.

Wong T.T., Leung C.S., Heng P.A., Wang J., 2007. Discrete Wavelet Transform on Consumer-Level Graphics Hardware. In *IEEE Transaction on Multimedia*. Vol. 9, No. 3, p. 668-673.

Ziv J., Lempel A. 1977. A universal algorithm for sequential data compression. In *IEEE Trans. on Inform. Theory*. Vol. IT-23, no. 3, p. 337-343.