High Speed Lossless Image Compression

Hendrik Siedelmann[1,2(✉)], Alexander Wender[1], and Martin Fuchs[1]

[1] University of Stuttgart, Stuttgart, Germany
hendrik.siedelmann@googlemail.com
[2] Heidelberg University, Heidelberg, Germany

Abstract. We introduce a simple approach to lossless image compression, which makes use of SIMD vectorization at every processing step to provide very high speed on modern CPUs. This is achieved by basing the compression on delta coding for prediction and bit packing for the actual compression, allowing a tuneable tradeoff between efficiency and speed, via the block size used for bit packing. The maximum achievable speed surpasses main memory bandwidth on the tested CPU, as well as the speed of all previous methods that achieve at least the same coding efficiency.

1 Introduction

For applications which need to process large amounts of image data such as high speed video or high resolution light fields, the I/O subsystem can easily become the bottleneck, even when using a RAID0 or SSDs for storage. This makes compression an attractive tool to widen this bottleneck, increasing overall performance. As lossy compression incurs signal degradation and may interfere with further processing, we will only consider lossless compression in the following. While dictionary based compression methods like `lzo` [18] can reach a high bandwidth, the compression ratio of such generic compression methods is quite low compared to dedicated image compression methods. However, research in lossless image compression has mainly been concerned with the maximization of the compression ratio, and even relatively fast image compression schemes like `jpeg-ls` are not able to keep up with the transfer rates of fast I/O configurations.

In this article, we present a simple lossless image compression scheme, which makes use of the SIMD instructions of modern CPUs to achieve extremely high performance. Our implementation allows a configurable tradeoff between speed and compression and exceeds the memory transfer speeds of modern CPUs in its fastest configuration, allowing incorporation of lossless compression into many areas where compression was not feasible before. Our specific use case, which motivated this work, is an example of such an application: recording a very large and dense light field data set (several terabytes in size), requiring a continuous compression bandwidth of 360 MiB/s, a rate which the used I/O configuration was not able to guarantee on its own.

J. Gall et al. (Eds.): GCPR 2015, LNCS 9358, pp. 343–355, 2015.
DOI: 10.1007/978-3-319-24947-6_28

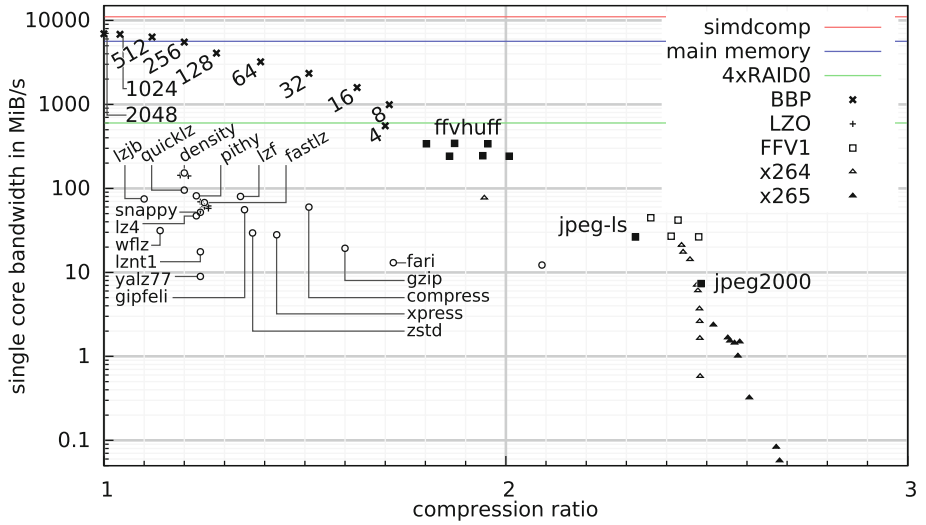

Fig. 1. Bandwidth vs. compression ratio for all tested methods, showing single core results on an Intel® Core™ i7-2600 CPU on the "blue sky" test sequence from the SVT data set [12]. "BBP" denotes our method for several block sizes from 2048 to 4 bytes. An example 4xRAID0 configuration and main memory bandwidth are included for comparison. The `simdcomp` method is the C implementation of [14] and can, in general, not compress image data, but is shown for bandwidth comparison. For `lzo`, and `ffvhuff` all configurations are shown, as well as all available presets for `x264` and `x265`.

1.1 Applications of Lossless Image Compression

Compression is always a tradeoff of processing resources against bandwidth and storage. While lossy compression provides high compression ratios, its application is limited to areas where the distortion of the signal due to lossy compression does not pose a problem. But in some cases losses are not acceptable. Examples for this include reference data sets for image processing or sophisticated image processing, like superresolution. In this case, lossless compression may still provide an overall performance increase due to bandwidth savings as well as reduced storage requirements. However lossless compression provides less compression, which reduces the gain obtained from using it. In many capture scenarios, the required bandwidth for on the fly compression, before initial storage, imposes a hard constraint. This is addressed by the proposed scheme, as compression bandwidth can be adjusted to speeds that surpass memory transfer speeds, a hard limit for any hardware configuration.

2 Related Work

In the following, we briefly discuss lossless compression methods that focus on high speed, see Fig. 1 for an overview of evaluated methods. The methods are

ordered by the dimensionality of the compression scheme, starting with generic 1D compression algorithms. Any compression method with lower dimensionality may be used to compress data with a higher dimensionality at reduced efficiency, as correlation in the missing dimensions cannot be exploited. On the other hand, lower dimensionality can lead to lower algorithmic complexity and therefore higher speed. While the method discussed in this paper is a pure image compression method, we will regard and evaluate compression from the basis of compressing a video stream, which allows the inclusion of advanced lossless video compression methods like `AVC` and `HEVC` which allow higher compression ratios by exploiting the 3D correlation. Also applications for high bandwidth image processing like light field, or high speed video capture, may allow the use of video compression methods, making an evaluation based on video data adequate.

2.1 Dictionary and 1D Compression

General purpose compression methods like `deflate` (`zlib`/`gzip`) [10] or `bzip2` [23] achieve good compression ratios for most types of inputs, but are quite slow, with a maximum bandwidth of around 30 MiB/s on an off-the-shelf Intel Core i7-2600 CPU. Faster methods, including `lzo`, `lz4` and `gipfeli` [9,15,18] are more directly based on the original `LZ77` compression scheme [28] and provide a bandwidth of up to 152 MiB/s for our configuration, see Fig. 1. This performance has led to the utilization for lossless 4 K image transmission [11], but compared to image based methods the compression ratio is rather poor, as dictionary based methods are not well suited to the task of image coding. Specialized methods adapted for specific tasks, like integer or floating point compression, provide a much higher speed of several gigabytes per second, by using a simple bit packing approach [8,14] and exploiting the SIMD instructions provided by modern CPU architectures [14]. However, those methods operate on 32 bit integers or 64 bit floating point values and are thus not directly applicable to the compression of 8 bit image data. We have developed an algorithm which uses SIMD bit packing for the actual compression, combining it with a prediction scheme and small block sizes, allowing efficient compression of image data. See Sect. 3 for details.

2.2 Image Compression

Dedicated lossless image compression methods like `jpeg-ls` [27] peak at around 25MiB/s, with modifications reaching 75MiB/s on an Intel® Core™ i7-920 processor at 2.67 GHz [26]. The `jpeg-ls` codec was standardized in 1999 and is still widely used as the baseline for the evaluation of lossless image compression methods. To our knowledge there are no significantly faster methods available, even though the gap to the fastest 1D compression method [14] amounts to more than two orders of magnitude. Later works mainly concentrate on increasing compression efficiency at reduced speed, which is not the focus of this work.

2.3 Video Compression

State of the art video standards like `HEVC`, as well as its predecessor `AVC`, include lossless profiles which provide high compression ratios. The `ffvhuff` coder from

the ffmpeg library [3] based on HuffYUV [24], which uses the jpeg-ls predictor together with simple Huffmann coding, obtains a speed of 343 MiB/s, the highest speed for any video codec we evaluated, see Fig. 1. Another interesting compressor is the ffv1 coder [17] at 44 MiB/s, also from ffmpeg, which combines a jpeg-ls style predictor with a context adaptive entropy coder, based on similiar principles as the one from AVC, with context adaption over several frames. The ffv1 coder achieves a compression ratio and speed competitive to AVC, see Fig. 1.

2.4 GPU Aided Compression

There have been various efforts to improve the performance of different compression techniques by using the massively parallel computation capabilities of modern GPUs. For 1D compression a text based method has been ported to the GPU by Ozsoy et al. [21] demonstrating a speedup of up to 2.21x for the GPU solution. Floating point compression based on bit packing has also been shown to reach a speedup of 5x [20], but at the cost of a reduced compression efficiency and in comparison to a CPU version which does not make use of SIMD for bit packing, like implemented in [14].

An approach for GPU based predictive lossless image coding has been presented by Kau and Chen, showing a speedup of up to 5x with a combined system utilizing GPU and CPU [13], however absolute performance is quite low at less than 1 MiB/s. The cuj2k library, implementing jpeg2000 on the GPU provides similar performance to a parallel jpeg2000 implementation for the CPU [2]. The work presented by Treib et al. [25], implementing a simple wavelet based compressor, provides results with slightly worse compression than jpeg2000 but with a compression speed of up to 700 MiB/s, which is significantly higher than previous approaches.

The difficulties in porting standard image compression methods to the GPU are analyzed by Richter and Simon [22], specifically for jpeg and jpeg2000. They conclude that especially the entropy coding part proves difficult for modern GPUs, together with the codestream build-up. Their observations include also the case where a highly optimized CPU implementation outperforms a GPU based approach. These considerations have led us to the conclusion that a carefully implemented CPU image compression algorithm that takes advantage of modern SIMD instruction sets may already provide a significant boost in lossless compression speed. Another advantage of the CPU only implementation is that the GPU remains free for image based processing tasks, for which it is better suited. However, there still is the opportunity for further research to investigate the potential of a GPU based implementation, but a CPU implementation is needed for a thorough comparison.

3 Compression Scheme

From the above methods, we found the SIMD based integer compression method to be the most promising approach for a fast compression scheme, due to the

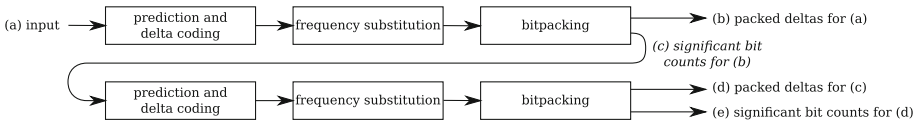

Fig. 2. Flow chart illustrating the compression procedure. An input stream (a) is processed with delta coding, frequency substitution and bit packing, see Sects. 3.2 to 3.5, producing a packed output stream (b), as well as the number of significant bits per block (c). The number of significant bits can change slowly from block to block, therefore (c) is fed through the same compression procedure to produce the final output streams (d) and (e) which, together with (b), compose the compressed output.

demonstrated high performance. However, bit packing is not directly possible with image data. A prediction scheme enables decorrelation of neighboring pixels and allows bit packing of the residuals, see Sect. 3.2, but the scheme still has to overcome several hurdles compared to the bit packing method using 32 bit integers from [14]:

Less Latitude: For image data, the bit depth is normally 8 bit compared to at least 32 bit in database indices, so each additional bit needed for the encoding has four times the impact on the overall compression performance.

Large Block Size: While SIMD on x86 has a width of 16 bytes, the implementation in [14] uses block sizes of 128 integers, respectively 512 bytes. Image data has a much higher variance compared to database indices, especially considering the smaller range for 8 bit data. Therefore, a smaller block size is required for efficient image compression, resulting in an increased overhead for signaling significant bit counts, which is especially problematic considering the greater impact of this overhead. Also, constant per block computations like signaling and branching depending on significant bit counts, have a higher impact on processing times with smaller block sizes.

Missing SIMD Instructions: On x86 many instructions that are available for processing 32 bit integers, as for example shifts, are not available in byte variants and have to be replaced by a more expensive combination of 32 bit shift and mask operations.

Size Increase Due to Delta Coding: In contrast to database indices, pixels in an image are not sorted by value and deltas between consecutive pixels therefore require one extra sign bit, increasing the compressed size by 12.5 % for naive delta coding.

3.1 Overview

Our approach, see Fig. 2, revolves around the concept of bit packing, which is the compression of integers by storing only the significant bits for each input value. To facilitate vectorization and to avoid excessive overhead due to the coding of significant bit counts, we apply bit packing to whole blocks of bytes, with larger

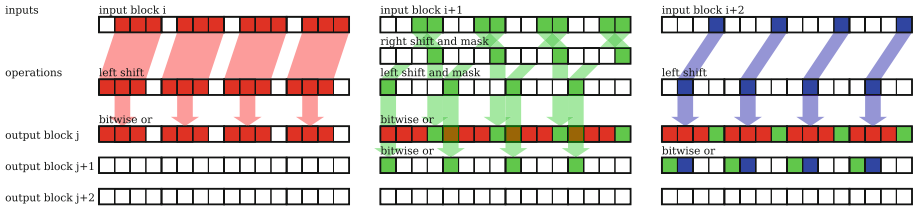

Fig. 3. Illustration of the interleaved bit packing scheme. For clarity the depicted blocks have a size of four samples with four bits each, instead of 8 bits as in the actual implementation. Bits are represented by boxes, significant bits are colored, white blocks have a value of zero. The packing routine distinguishes two cases: either there is enough space to store the significant bits, as is the case with block i and i+2, or the block has to be split between the current and next output block, as with block i+1 in the illustration. The colored arrows denote executed operations. Please note that the operations are always performed with the whole SIMD width and not per element.

block sizes resulting in higher performance at reduced compression efficiency. We use a very simple prediction scheme, trading coding efficiency for higher speed: We refer to this whole scheme as block-wise bit packing (`BBP`). Compared to Huffmann or arithmetic coding, bit packing cannot adapt to the distribution of symbols. Therefore, we employ *frequency substitution* in order to encode more frequent values, i.e. values with a higher probability of occurrence, with less significant bits, see Sect. 3.4. As bit packing results in a variable length code which is not prefix free, and correct decoding thus requires that the number of significant bits is known to the decoder via external means, the significant bit lengths have to be signaled separately. The need to store the number of significant bits is another reason, apart from vectorization, for the division into blocks. As the block size approaches one byte, the overhead of signaling the number of significant bits outweighs the increase in coding efficiency. To reduce the efficiency loss at all block sizes, which occurs due to the overhead for signaling significant bit lengths, the whole scheme is also applied to the data stream which signals those bit lengths, exploiting the correlation between successive blocks.

3.2 Prediction in 1 Dimension

Delta coding can be regarded as the most simple form of prediction based compression: each sample is predicted to have the same value as the last one, so calculation of the residual simplifies to the calculation of the difference between the two samples. A problem with such prediction schemes is the necessity to calculate a prefix sum when decoding. While parallel calculation of the prefix sum is possible, performance is still reduced. The integer coding by Lemire et al. [14] avoids the problem by using a fixed offset of 16 Bytes, which the authors report to increase the average delta by a factor of four.

In contrast to the database indexes they address, image data is correlated in two dimensions. Normally, this is exploited to improve compression efficiency by

using a 2D predictor, such as the median edge predictor in `jpeg-ls`. We exploit the two-dimensional correlation to accelerate performance by coding the delta between vertical pixel neighbors, followed by the packing of horizontal blocks using SIMD. The correlation in the vertical direction provides residuals that are smaller than the actual sample values, while horizontal correlation means that a horizontal block tends to group samples with similar significant bit lengths, reducing efficiency loss due to block-wise handling.

Additionally, this gives a large flexibility for the layout of the input data. Because of the in-memory layout as continuous chunks of memory, the vertical prediction can be implemented with simple pointer arithmetic. The address of the previous line is calculated using the address of the current line minus a fixed offset, which is normally the width of the image. If the input data consists of interleaved samples, e.g. RGB images or raw Bayer patterns, then the offset may be adjusted, so that prediction is always executed from the same sample type, avoiding costly preprocessing steps for format conversion.

3.3 Modulo Delta Coding

As the difference between predicted and observed value may be anywhere between -255 and 255, it cannot be coded naively inside an 8 bit range. To avoid the necessity of expanding the coding range to 9 bit, which would halve the effective SIMD width and waste one bit of space, we utilize modular arithmetic over $\mathbb{Z}/256\mathbb{Z}$.

3.4 Frequency Substitution

The prediction residuals from delta coding tend to follow a two sided geometric distribution [16,19], where small differences are very common. While small values are well suited for bit packing, the use of non-saturated wrapping arithmetics maps small differences, like -1, to potentially large values (255 in this case), which requires all 8 bits for encoding. To compensate for this effect, we apply a substitution, ordered by the minimal absolute difference: 0 stays 0, $-1 \equiv 255$ maps to 1, $-255 \equiv 1$ maps to 2, $-2 \equiv 254$ maps to 3, etc. This mapping is identical to the scheme often used when mapping signed to unsigned integers in the context of universal codes, as used by `jpeg-ls` and `AVC` when using Golomb codes for the entropy stage, just applied with respect to the implied modulo operation.

This mapping describes a triangle function with a slope of 2, where the first slope maps to even values and the second slope to odd values. This function is suitable for SIMD implementation, leading to high performance.

3.5 Block-Wise Interleaved Bit Packing

The packing uses a vertical layout, where a block of n bytes is interleaved into a block of the same size, with unused bits remaining at the same position in

every byte. Consecutive blocks are interleaved into the unused bits for each byte in a block, until no unused bit remains, which leads to the write out of the current block and allocation of the next one, see Fig. 3 for a visualisation of the procedure. Compared to a computed jump to one of the different bit packing routines, depending on unused and required bits, as implemented in [14], we only branch over the block full condition and process any bit combination using the same code, with computed shifts and masks. This single branch is the only one within the compression loop, leading to low misprediction rates. This approach results in a lower performance penalty for small block sizes, which are necessary for efficient image compression.

4 Evaluation

The following sections will outline the conditions of the evaluation, which was performed on an Intel® Core™ i7-2600 Processor running at 3.4 GHz.

4.1 Implementations

Our implementation for the introduced method is executed in C, making extensive use of compiler intrinsics for SIMD operations as well as constant propagation and link time optimizations to realize portable, modular and easily extended code. As SSE3 instructions, released in 2006, have become quite ubiquitous on the x86 platform, we have targeted it in the evaluations presented in this article (AVX, which became available in 2011, focused on floating point operations and was therefore not considered). To allow a more detailed insight into the actual implementation, our code – including non-vectorized C fallback and an unoptimized implementation for the ARM NEON SIMD extension – will be released online under an open source license.

The implementations for the image and video compression methods were taken from the `ffmpeg` library [3], except the `AVC` (libx264 [6]), `HEVC` (libx265 [1]) and `jpeg2000` encoders (libopenjpeg [4]), which are provided by external libraries, but are still integrated within `ffmpeg`. The implementation of Lemire et al. [14] was also evaluated to show the performance possible when executing pure bit packing, although no compression could be achieved on the used data set. To test the dictionary methods the open source squash library [5] was used, which incorporates a broad range of generic compression methods, using the respective reference implementations.

4.2 Method

To minimize the effect of memory transfers for the fastest methods, the compression was executed on chunks of 64 KiB, which allows the whole compression to take place within the 256 KiB L2 cache of the Intel® Core™ i7-2600, for which we measured a bandwidth of 25 GiB/s. Timings were obtained for each coder, by reading the file in chunks, measuring 16 repeated runs of the coder in

one go, to avoid the influence of the execution time of the timing syscall on the measured bandwidth. Measuring only a single iteration resulted in a bandwidth up to 30 % lower for the fastest coders. Note that the results of the video compression methods were obtained by running `ffmpeg` in benchmark mode, thus including memory transfer and management overhead. However extra measurement of the pure decoding performance of `ffmpeg` indicate an overhead of less than 3 % for the fastest measured video compression method.

4.3 Data Set

The results shown in Fig. 1 were obtained by executing the benchmark described in Sect. 4.1 on the uncompressed "blue sky" test sequence from the SVT data set [12]. The specific sequence was selected because it is easy enough as a compression challenge that all methods achieved at least minimal compression, while still being a recording of a natural scene, containing noise and other artifacts, which make compression more challenging than computer generated imagery. The scene was converted to grayscale as this was the only format compatible with all tested implementations. More complex scenes caused some coders, specifically `lzo`, to fall back into a non-compressing mode with a much higher bandwidth of nearly 1.7 GiB/s, see Table 1. While such a mode is a useful fallback if compression is not possible, such a behavior does not provide useful data for this evaluation. Please refer to the supplemental materials for a more extensive list of benchmark results for several different sources.

4.4 Results

A comparison of the performances of all evaluated compression algorithms can be found in Fig. 1. Positions in the plot show the relative performance of methods against each other, with faster methods towards the top and better compressing methods to the right. The evaluation of the performance of the different compression methods can be summarized as follows:

- Even the fastest generic compression methods are dominated by the dedicated image and video compression methods, as they achieve relatively low compression ratios without higher speed.
- Most of the methods optimized for image or video compression can provide better compression ratios, but at a significantly lower bandwidth.
- Our implementation can provide a very high bandwidth of over 6 GiB/s, much faster than previous methods, while it is also able to provide a compression ratio approaching that of the previously fastest dedicated image compression methods.

Regarding encoding speed, the closest contender to our method is `ffvhuff`, which is several times slower, but achieves slightly higher compression of up to 1:2.0. The two video coders `x264` and `x265` produce the highest compression at up to 1:2.48 and 1:2.68 respectively, but at a noticeable penalty in encoding

speed. The `jpeg-ls` coder can not keep up with this performance but is still notable as the relatively high compression ratio of 1:2.32 is achieved by a pure image coder, which is unable to exploit the temporal correlation.

4.5 Minor Findings

The performance of the different compression methods depends to a varying degree on the specific data set, please see Table 1 for benchmark results of a few coders on different sources, more are available in the supplemental materials. While a detailed analysis of the speed variability was out of scope for this work, the basic pattern seems to be that the more complex a method the higher the dependency of the performance on the input characteristics. Specifically the relatively simple method implemented in this work, as well as the method in [14] and `ffvhuff` show a moderate dependency on the input characteristics, while most video compression methods and the dictionary based methods show a more complex behavior. The most variability was observed by the `lzo` coder for which performance varied by a factor of ten depending on the input.

Table 1. As this extract from our tests (see the supplemental material for all results) shows, compression ratio and speed vary a lot with the content, with some correlation between high throughput and better compression rates. The results for `lzo` indicate a special mode for non-compressible input. The best results for bandwidth and compression ratio are marked in bold for each file, the fastest and slowest result for each coder in italic. For our method block sizes of 8 and 128 bytes were selected as representative tradeoffs, please refer to Fig. 1 for an overview of all possible block sizes and their respective performance.

	Bandwidth in MiB/s					Compression Ratio				
Coder	File 1	File 2	File 3	File 4	File 5	File 1	File 2	File 3	File 4	File 5
BBP-8	975	992	*947*	*1307*	1022	1.31	1.71	1.38	5.16	1.51
BBP-128	**3993**	**4236**	***3957***	***4835***	**4199**	1.16	1.28	1.14	3.42	1.31
ffvhuff	329	343	*308*	*479*	339	1.45	**1.96**	1.39	3.47	1.59
x264-ultrafast	101	76	*72*	*206*	109	**1.49**	1.95	**1.47**	**9.17**	**1.84**
density	245	*150*	*311*	198	227	1.02	1.20	0.99	1.66	1.04
lzo1x	633	*141*	*1668*	334	427	1.02	1.21	1	3.69	1.04

5 Discussion and Future Work

As shown in Sect. 4.4, the performance achieved by our method is significantly higher than any previous method. In the fastest configuration the achievable performance surpasses the memory bandwidth, although at a considerable loss in compression efficiency, while the simple bitpacking method of Lemire et al.

[14] roughly doubles this performance, without achieving any compression for the evaluated content.

While one could expect the performance of an implementation to scale well with the SIMD width, which is effectively the level of parallism, preliminary results for the new AVX2 extensions, operating with a SIMD width of 256 bits, indicate a speedup of less than 40 % compared to the 128 bit wide SSSE3 implementation, suggesting that a large portion of the execution time is not spent on pure arithmetics, but on cache misses and branch misprediction. The expected AVX-512/AVX3 instructions will warrant further examination of this property.

In the introduced method, compression is based solely on 2D correlation. A candidate for an efficient disparity compensation based solution could be based on PatchMatch [7], which could be used to evaluate only few candidates per pixel over several iterations, propagating good solutions over the frame.

Also, the high performance of the method has implications for applications limited by main memory size or bandwidth. The compression scheme supports in-memory compression to increase available memory as well as to increase memory bandwidth, provided access patterns permit the decompression and processing to execute from within the CPU caches.

6 Conclusion

This work shows how to overcome the bandwidth limitations of previous lossless generic and image specific compression methods, while providing a reasonable compression performance. This does not only enable lossless image compression for applications which could not previously make use of it, but also provides interesting possibilities for image processing tasks, regarding memory bandwidth and utilization optimizations.

Acknowledgements. This research was financially supported by the Juniorprofessorenprogramm Baden-Württemberg.

References

1. x265 HEVC high efficiency video coding H.265 encoder (last accessed on 17 December 2014). http://x265.org/
2. JPEG 2000 on CUDA (last accessed on 27 May 2015). http://cuj2k.sourceforge.net/
3. A complete, cross-platform solution to record, convert and stream audio and video. (last accessed on 27 May 2015). https://www.ffmpeg.org
4. OpenJPEG library : an open source JPEG 2000 codec (last accessed on 27 May 2015). http://www.openjpeg.org/
5. Squash - compression abstraction library (last accessed on 27 May 2015). http://quixdb.github.io/squash

6. VideoLAN - x264, the best H.264/AVC encoder (last accessed on 27 May 2015). http://www.videolan.org/developers/x264.html
7. Barnes, C., Shechtman, E., Finkelstein, A., Goldman, D.B.: PatchMatch: A randomized correspondence algorithm for structural image editing. In: Proceeding of SIGGRAPH ACM Transactions on Graphics vol. 28, no. 3, August 2009
8. Burtscher, M., Ratanaworabhan, P.: FPC: a high-speed compressor for double-precision floating-point data. IEEE Trans. Comput. **58**(1), 18–31 (2009)
9. Collet, Y.: LZ4 explained, May 2011. http://fastcompression.blogspot.co.at/2011/05/lz4-explained.html
10. Deutsch, P.: Deflate compressed data format specification version 1.3. RFC 1951, May 1996. https://tools.ietf.org/html/rfc1951
11. Gomes, R.D., Costa, Y.G.G.d., Aquino Júnior, L.L., Silva Neto, M.G.d., Duarte, A.N., Souza Filho, G.L.d.: A solution for transmitting and displaying UHD 3d raw videos using lossless compression. In: Proceedings of the 19th Brazilian Symposium on Multimedia and the Web, pp. 173–176, WebMedia 2013. ACM, New York (2013). http://doi.acm.org/10.1145/2526188.2526228
12. Haglund, L.: The SVT high definition multi format test set. Swedish Television Stockholm (2006). https://media.xiph.org/video/derf/
13. Kau, L.J., Chen, C.S.: Speeding up the runtime performance for lossless image coding on GPUs with CUDA. In: IEEE International Symposium on Circuits and Systems (ISCAS), 2013, pp. 2868–2871, May 2013
14. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. CoRR abs/1209.2137 (2012). http://arxiv.org/abs/1209.2137
15. Lenhardt, R., Alakuijala, J.: Gipfeli-high speed compression algorithm. In: Data Compression Conference (DCC), 2012, pp. 109–118. IEEE (2012)
16. Netravali, A., Limb, J.O.: Picture coding: a review. Proc. IEEE **68**(3), 366–406 (1980)
17. Niedermayer, M.: FFV1 video codec specification, August 2013. http://www1.mplayerhq.hu/michael/ffv1.html
18. Oberhumer, M.F.: oberhumer.com: LZO real-time data compression library (last accessed on 27 May 2015). http://www.oberhumer.com/opensource/lzo/
19. O'Neal, J.: Predictive quantizing systems (differential pulse code modulation) for the transmission of television signals. Bell Syst. Tech. J. **45**(5), 689–721 (1966)
20. O'Neil, M.A., Burtscher, M.: Floating-point data compression at 75 Gb/s on a GPU. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, pp. 7:1–7:7, GPGPU-4. ACM, New York (2011). http://doi.acm.org/10.1145/1964179.1964189
21. Ozsoy, A., Swany, M., Chauhan, A.: Pipelined parallel lzss for streaming data compression on GPGPUs. In: IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), 2012, pp. 37–44, December 2012
22. Richter, T., Simon, S.: Coding strategies and performance analysis of GPU accelerated image compression. Picture Coding Symp. (PCS) **2013**, 125–128 (2013)
23. Seward, J.: bzip2 and libbzip2 (1996). http://www.bzip.org
24. Togni, R.: Description of the HuffYUV (HFYU) codec, March 2003. http://multimedia.cx/huffyuv.txt
25. Treib, M., Reichl, F., Auer, S., Westermann, R.: Interactive editing of gigasample terrain fields. In: Proceeding Eurographics Computer Graphics Forum, vol. 31, no. 2, pp. 383–392 (2012). http://diglib.eg.org/EG/CGF/volume31/issue2/v31i2pp383-392.pdf

26. Wang, Z., Klaiber, M., Gera, Y., Simon, S., Richter, T.: Fast lossless image compression with 2d golomb parameter adaptation based on JPEG-LS. In: Proceedings of the 20th European Signal Processing Conference, EUSIPCO 2012, Bucharest, Romania, August 27–31, 2012, pp. 1920–1924 (2012). http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6334076
27. Weinberger, M., Seroussi, G., Sapiro, G.: The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. IEEE Trans. Image Process. **9**(8), 1309–1324 (2000)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. **23**(3), 337–343 (1977)