Data Coding and Compression, FIT, BUT

Huffman and RLE Compression of Grayscale Images

David Mihola
xmihol00

12th of May, 2024

DISCLAIMER

Sections 1, 2 and 3 conform to the recommended length of the documentation and contain sufficient information for grading. Further sections expand the first 3 sections and are dedicated to interested readers.

Moreover, the supplied `Makefile` uses shell commands to determine availability of specific instructions on a target system, see the `README.md` for more details. Both the compilation and execution was thoroughly tested on the `merlin.fit.vutbr.cz` server, therefore no issues should be expected.

1 Problem Formulation

The overall goal of this project is to compress and decompress grayscale images with the use of Huffman coding and run length encoding (RLE) lossless compression techniques. Moreover, the compression/decompression algorithms shall support the 4 following modes:

- **static:** where the input data are perceived as 1 dimensional,
- **adaptive:** where the input data are perceived as 2 dimensional and the input is further partitioned into smaller blocks of constant size for adaptive serialization,
- **static with a model:** where a context model is applied during the regular static compression/decompression,
- **adaptive with a model:** where a context model is applied during the regular adaptive compression/decompression,

Last but not least, apart from achieving the best possible compression rates, the speed of the compression and decompression shall be greatly considered.

1.1 Huffman Coding

Huffman code is an optimal variable length prefix code. The symbol codes are derived from their probability distribution using a binary tree. The tree nodes are constructed from bottom up, starting with the symbol probabilities as initial nodes, such that 2 nodes with the smallest probability are merged into a new node with the sum of theirs probabilities, until only a single node remains. Codes are then assigned by a top to bottom traversal of the tree, where at each branching the current code is extended by 0 bit for the left branch and 1 bit for the right branch or vice versa. Meaning that the length of a code for a symbol is solely determined by its depth in the constructed tree.
When symbols at each depth of the tree are sorted, the depth property allows to derive the symbol codes so that there exists just a single possible assignment of each code to a symbol. Such codes are called canonical Huffman codes. Meaning that only the depth of each symbol must be transferred from the encoder to the decoder.

1.2 Run Length Encoding

Run length encoding (RLE) is a lossless compression technique, in which repeating symbols (runs) are encoded as the value of the symbol[1] followed by the number of its repetitions.
There are several methods how to encode runs of symbols. From encoding a number of repetitions of each symbol, even if it is an isolated occurrence, to using triplets with the <`triplet indicating symbol, repeated symbol, number of repetitions`> format, to encoding runs only of the most frequent symbol using a specific binary code[2], to encoding only runs longer than or equal to 3 symbols as a triplet of the 3 same symbol codes followed by the number of repetitions.

[1] raw, e.g. ASCII, or encoded, e.g. with a Huffman code
[2] RLE0 technique

The last named approach is particularly useful for input data in which all possible symbols are exhausted and no triplet indicating symbol is available. This is in general the case, when symbols represent values of pixels in an image.

2 Implementation Details

The implemented algorithms were from the beginning designed to be multi-threaded and with a great attention to performance, i.e. compression ratio may be in some cases negatively affected to enable faster computation.
The compression consists of the following higher level operations:

1. Application of a context model on the input data if compression with a model is active. Pixel difference model[3] is used in this implementation.
2. Adaptive serialization of the input data if adaptive compression is activate.
3. Computation of symbol frequencies, i.e. unnormalized symbol probabilities.
4. Construction of a Huffman tree, tree balancing and assignment of Huffman codes to symbols.
5. Encoding of the serialized input data with the derived Huffman codes and RLE.

Inversely, the decompression is composed of the following higher level operations:

1. Reconstruction of Huffman codes for the compressed data.
2. Decoding of the compressed data, i.e. reversal of the Huffman coding and RLE.
3. Deserialization of the decoded data if adaptive decompression is active.
4. Reversal of the pixel difference model if decompression with model is active.

The following sections further expand on some of the higher level operations listed above.

2.1 Adaptive Serialization

Horizontal zig-zag, vertical zig-zag, major diagonal zig-zag and minor diagonal zig-zag traversals of 16x16 sub-patches of the input data were used. Zig-zag traversals, i.e. where paths are not cut off at the block edges but they alternate directions, should be preferred for images compression over regular traversals thanks to their ability to better preserve the locality of neighboring pixels. However, the code has been designed to facilitate an easy replacement of the used traversals by simply providing a different set of indices specifying the desired traversal pattern[4], may it better suite the expected inputs. The Table 6 captures the frequencies of the currently used traversals on a set of benchmark images.

[3] Values of neighbouring pixels in 1 dimensionally perceived input data are subtracted from each other.
[4] The `traversals_indices.py` script was used to generate the currently used sets of indices.

2.2 Huffman Tree Construction and Huffman Code Assignment

The computed symbol frequencies are clamped to the maximum[5] of $2^{24} - 2$ and packed into a 32-bit data structure, where the 8 least significant bits (LSBs) represent the symbol value. The Huffman tree is constructed using an algorithm described in [2] adopted to 512-bit registers. The built tree is then rebalanced to have a maximal depth of 16 with an algorithm inspired by [1].
Moreover, it is ensured that there are at most 32 unique prefixes across the different depths of the tree by iteratively rounding the number of symbols at each depth. The rounding factor increases[6] after each unsuccessful iteration to ensure convergence. This allows to perform vectorized mask decoding of the assigned Huffman codes, see section 2.3 for more details.
Furthermore, symbols at their final depths are sorted in an ascending order based on their values. 256-bit registers are used for each occupied depth as bitmaps, where bits are set to 1 for each symbol appearing at the respective depth, i.e. radix sort with a radix of 256 is used. Such populated depth bitmaps are then finally used to assign canonical Huffman codes to symbols, but also to transfer the Huffman tree to the decoder. For the latter case, the bitmaps are compressed followingly:

- Further 16-bit bitmap is used to indicate which depths are occupied.
- Unseen symbols are placed to the 0 depth, which is otherwise unused.
- The most populated depth is not transmitted, its content can be fully recovered by bitwise xoring a fully populated depth bitmap with all the others.
- Only non 0 bytes of each depth bitmap are transferred, with an additional prepended 32-bit bitmap indicating which and how many bytes of the 256-bit bitmap follow.

See the Table 7 for achieved sizes on a set of benchmark images, 128 bytes[7] can be used as a baseline for comparison.

2.3 Huffman Codes Reconstruction and Decoding

During the decompression and reconstruction the Huffman tree as well as the canonical Huffman codes, several additional steps are taken:

1. Huffman code prefixes are assigned with the depth at which they appear in the reconstructed tree, their bit length and bit length of codes that they are prefixing.
2. They are sorted by their length in descending order.
3. The prefixes are aligned and placed to the most significant bits (MSBs) of 16-bit lanes of a 512-bit register in their sorted order starting from the most significant lane.
4. Another 512-bit masking register is created for each prefix such that each lane of this register contains a 16-bit mask with MSBs set to 1 according to the respective prefix length.

[5] Value of $2^{24} - 1$ is used to represent an unseen symbol, the 8 LSBs are all set to 1 as well.

[6] At iteration N N LSBs are cleared from a variable representing the number of symbols at a specific depth and the cleared symbols are moved to lower occupied depth, with its capacity being increased accordingly.

[7] Depth encoded on 4-bits for each of 256 symbols.

5. A 32 item lookup table with entries packed to 32 bits is created for each prefix containing its length, length of codes that it is prefixing, its suffix shift and an offset to a table of symbols created during this process as well.

The decoding then comprises of the following steps:

1. First 16 bits of the yet undecoded input are broadcasted to each 16-bit lane of a 512-bit register.
2. This register is bitwise xored and bitwise anded with the prefixes and masks registers.
3. Each 16-bit lane is compared to 0 and the results are gathered to a 32-bit bitmap.
4. Leading zeros are counted in the resulting 32-bit bitmap[8] and the count is used as an index to the previously created 32 item lookup table retrieving the respective code length, prefix length, suffix shift and the base symbol table offset for the encoded symbol.
5. The 16 encoded bits are copied, left shifted by the prefix length and right shifted by the suffix shift obtaining the suffix value.
6. The base symbol table offset plus the suffix value are used as an index to the table of symbols to retrieve the encoded symbol.
7. Finally, the code length is used to shift the yet undecoded input and the process can be repeated.

This method of decompressing aims to reduce the number of L1 cache misses in comparison to another fast method using a lookup table of 2^{16} 16-bit entries[9] for immediate lookup with the raw compressed data.

2.4 Run Length Encoding and Decoding

Sequences of repeating symbols are encoded by compressing the same symbol 3 times and following it by the number of its repetitions decreased by 3. The number of repetitions is encoded with a sequence of 4-bit[10] chunks, where the MSB set to 0 indicates that the sequence ends. The remaining 3 bits of each chunk are interpreted as a binary value shifted left by the chunk index multiplied by 3, i.e. the original binary number is scattered to the chunks from LSB to MSB.

3 Results

The performance of the implemented algorithm both in terms of achieved compression ratio and the speed of compression/decompression is captured in Table 1. Furthermore, more detailed analysis of the compression ratio and compression/decompression speed is captured in sections 6 and 7 respectively.

[8] The first non 0 bit represents the position with the longest prefix match.

[9] 8 bits for symbol and 8 bits for code length

[10] The best performing chunk size on a benchmark dataset. The code facilitates an easy adjustment of the chunk size from 2 to 16 bits by rewriting a constant.

	Entropy	Average compressed bits per pixel				Compression time [μs]				Decompression time [μs]			
		S	**A**	**S + M**	**A + M**	**S**	**A**	**S + M**	**A + M**	**S**	**A**	**S + M**	**A + M**
df1h.raw	8.0000	8.0000	2.0082	0.0006	0.0084	2482	2172	1755	2051	1160	2767	1950	2428
df1hvx.raw	4.5140	2.9643	2.3313	1.1136	0.8899	1974	2204	1892	2203	2387	3050	2303	2871
df1v.raw	8.0000	0.0707	2.0082	0.0395	0.0290	1702	2186	1717	2047	1885	2803	1972	2384
hd01.raw	3.8255	3.4270	3.3723	3.1140	3.0714	2114	2323	2128	2424	2402	3187	2431	3233
hd02.raw	3.6359	3.2513	3.1966	2.6170	2.5915	2073	2312	2108	2380	2414	3153	2434	3207
hd07.raw	5.5769	4.7741	4.7143	4.5352	4.4717	2155	2436	2233	2481	2485	3316	2475	3354
hd08.raw	4.2108	3.9344	3.8388	2.9772	2.9272	2091	2358	2131	2452	2413	3310	2459	3315
hd09.raw	6.6211	7.4313	7.3494	5.2992	5.2257	2375	2654	2389	2552	2614	3383	2629	3399
hd12.raw	6.1657	6.1020	5.9931	4.8539	4.7689	2258	2525	2290	2533	2450	3346	2541	3383
nk01.raw	6.4729	7.4592	7.4553	6.0477	6.0526	2327	2606	2361	2662	2614	3670	2668	3718

Table 1: Performance of the compression and decompression on a benchmark dataset

S stands for static compression, **A** stands for adaptive compression and **+ M** stands for compression with implemented difference model. **Average compressed bits per pixel** are displayed for a single-threaded compression, i.e. the best possible. Both **Compression time** and **Decompression time** are means across 100 executions on an Intel® Xeon® Gold 6240 Processor running at 2.6 GHz with a number of threads that achieved the best mean time, e.i. usually 4 or 8 threads, as more threads cause diminishing returns with too much parallelism on such small inputs (512x512 pixels).

4 Multi-Threading

Parallelism is brought to the application via the use of OpenMP pragmas and library function calls. All the higher level operations listed in the section 2 apart from the Huffman codes computation and reconstruction can be parallelized. OpenMP parallel for cycles or data distribution based on thread numbers are used to decompose the task on the side of the compressor. On the decompressor side, however, the algorithm must react to the number of threads used for compression. Therefore, OpenMP tasks are used to drive the parallelism there. This allows the compression to be performed with a different number of threads than the decompression[11].
If for some reason the application shall not utilize multi-threading, it can be easily compiled without OpenMP support and the algorithms will execute sequentially omitting all thread management, i.e. possibly faster than a single-threaded execution with OpenMP support. See the `README.md` for more details.

5 Backward Compatibility

The use of the aforementioned vector registers is achieved with specialized Intel® Advanced Vector Extensions 512 instructions. These instructions may not be available on all machines, definitely not on machines from other manufacturers. Therefore, a backward compatible implementation is also available.
This implementation rebalances the Huffman tree at the compressor side such that each populated depth has exactly a power of 2 symbols larger or equal to the previous depth, which is closer to the root of the tree. Meaning that all prefixes will start with a sequence of 1 bits ended with a 0 bit. The decompressor can then use a similar algorithm as described in the section 2.3 with the difference that instead of using the vector instructions, the index to the 32 item lookup table[12] is directly computed by counting the leading 1 bits in the yet undecoded input. This approach also ensures that a vector instruction enabled decompressor will be able to decompress such compressed files. It must be, however, noted that this solution, although it may seem even more elegant, does not deliver nearly as good compression ratios, because the symbol probabilities/frequencies can be quite severely shifted by the tree rebalancing. The `README.md` states, how this mode can be enforced during compilation.

[11] If the number of threads used for compression is smaller than for decompression, some threads on the decompressor side will not be utilized.

[12] Maximum of 16 indices will be used in this case.

6 Data Analysis

	Uncompressed	Compressed file size [B]				Compression ratio				Space savings			
	file size [B]	min	max	mean	std	min	max	mean	std	min	max	mean	std
df1h.raw	262144	262145	262145	262145.00	0.00	1.00	1.00	1.00	0.00	-0.00	-0.00	-0.00	0.00
df1hvx.raw	262144	97135	97221	97163.50	32.15	2.70	2.70	2.70	0.00	0.63	0.63	0.63	0.00
df1v.raw	262144	2316	2408	2346.83	34.37	108.86	113.19	111.72	1.61	0.99	0.99	0.99	0.00
hd01.raw	262144	112295	112491	112357.17	74.33	2.33	2.33	2.33	0.00	0.57	0.57	0.57	0.00
hd02.raw	262144	106540	106726	106599.17	69.98	2.46	2.46	2.46	0.00	0.59	0.59	0.59	0.00
hd07.raw	262144	156437	156639	156501.67	76.38	1.67	1.68	1.68	0.00	0.40	0.40	0.40	0.00
hd08.raw	262144	128923	129119	128985.00	73.61	2.03	2.03	2.03	0.00	0.51	0.51	0.51	0.00
hd09.raw	262144	243510	243630	243550.67	44.23	1.08	1.08	1.08	0.00	0.07	0.07	0.07	0.00
hd12.raw	262144	199950	200138	200011.00	71.05	1.31	1.31	1.31	0.00	0.24	0.24	0.24	0.00
nk01.raw	262144	244424	244512	244455.17	32.50	1.07	1.07	1.07	0.00	0.07	0.07	0.07	0.00

Table 2: Static compression without a model

	Uncompressed	Compressed file size [B]				Compression ratio				Space savings			
	file size [B]	min	max	mean	std	min	max	mean	std	min	max	mean	std
df1h.raw	262144	65804	65916	65841.33	41.87	3.98	3.98	3.98	0.00	0.75	0.75	0.75	0.00
df1hvx.raw	262144	76391	76473	76419.33	30.79	3.43	3.43	3.43	0.00	0.71	0.71	0.71	0.00
df1v.raw	262144	65804	65916	65841.33	41.87	3.98	3.98	3.98	0.00	0.75	0.75	0.75	0.00
hd01.raw	262144	110505	110711	110572.17	78.04	2.37	2.37	2.37	0.00	0.58	0.58	0.58	0.00
hd02.raw	262144	104746	104944	104809.83	74.67	2.50	2.50	2.50	0.00	0.60	0.60	0.60	0.00
hd07.raw	262144	154477	154689	154545.67	79.96	1.69	1.70	1.70	0.00	0.41	0.41	0.41	0.00
hd08.raw	262144	125789	125991	125854.33	76.99	2.08	2.08	2.08	0.00	0.52	0.52	0.52	0.00
hd09.raw	262144	240826	240934	240860.67	40.21	1.09	1.09	1.09	0.00	0.08	0.08	0.08	0.00
hd12.raw	262144	196382	196614	196456.33	87.83	1.33	1.33	1.33	0.00	0.25	0.25	0.25	0.00
nk01.raw	262144	244296	244388	244326.17	34.48	1.07	1.07	1.07	0.00	0.07	0.07	0.07	0.00

Table 3: Adaptive compression without a model

	Uncompressed file size [B]	Compressed file size [B]				Compression ratio				Space savings			
		min	max	mean	std	min	max	mean	std	min	max	mean	std
df1h.raw	262144	19	157	62.67	52.19	1669.71	13797.05	6848.90	4559.57	1.00	1.00	1.00	0.00
df1hvx.raw	262144	36490	36558	36513.67	25.32	7.17	7.18	7.18	0.00	0.86	0.86	0.86	0.00
df1v.raw	262144	1295	1473	1377.33	69.84	177.97	202.43	190.74	9.70	0.99	1.00	0.99	0.00
hd01.raw	262144	102040	102244	102105.50	76.54	2.56	2.57	2.57	0.00	0.61	0.61	0.61	0.00
hd02.raw	262144	85755	85919	85808.83	62.42	3.05	3.06	3.05	0.00	0.67	0.67	0.67	0.00
hd07.raw	262144	148610	148810	148675.33	75.78	1.76	1.76	1.76	0.00	0.43	0.43	0.43	0.00
hd08.raw	262144	97558	97734	97613.67	66.28	2.68	2.69	2.69	0.00	0.63	0.63	0.63	0.00
hd09.raw	262144	173643	173757	173682.33	42.07	1.51	1.51	1.51	0.00	0.34	0.34	0.34	0.00
hd12.raw	262144	159052	159236	159112.00	69.28	1.65	1.65	1.65	0.00	0.39	0.39	0.39	0.00
nk01.raw	262144	198172	198276	198206.00	38.69	1.32	1.32	1.32	0.00	0.24	0.24	0.24	0.00

Table 4: Static compression with implemented difference model

	Uncompressed file size [B]	Compressed file size [B]				Compression ratio				Space savings			
		min	max	mean	std	min	max	mean	std	min	max	mean	std
df1h.raw	262144	275	413	318.67	52.19	634.73	953.25	838.97	120.45	1.00	1.00	1.00	0.00
df1hvx.raw	262144	29160	29224	29183.00	24.02	8.97	8.99	8.98	0.01	0.89	0.89	0.89	0.00
df1v.raw	262144	949	1085	1021.00	58.15	241.61	276.23	257.46	14.96	1.00	1.00	1.00	0.00
hd01.raw	262144	100644	100860	100713.50	81.69	2.60	2.60	2.60	0.00	0.62	0.62	0.62	0.00
hd02.raw	262144	84917	85097	84975.50	67.60	3.08	3.09	3.08	0.00	0.68	0.68	0.68	0.00
hd07.raw	262144	146528	146744	146597.00	82.09	1.79	1.79	1.79	0.00	0.44	0.44	0.44	0.00
hd08.raw	262144	95918	96096	95975.33	66.94	2.73	2.73	2.73	0.00	0.63	0.63	0.63	0.00
hd09.raw	262144	171235	171359	171276.00	46.74	1.53	1.53	1.53	0.00	0.35	0.35	0.35	0.00
hd12.raw	262144	156268	156486	156338.33	82.31	1.68	1.68	1.68	0.00	0.40	0.40	0.40	0.00
nk01.raw	262144	198330	198430	198364.33	37.31	1.32	1.32	1.32	0.00	0.24	0.24	0.24	0.00

Table 5: Adaptive compression with implemented difference model

	Without a model				With implemented difference model			
	Horizontal	**Vertical**	**Minor diagonal**	**Major diagonal**	**Horizontal**	**Vertical**	**Minor diagonal**	**Major diagonal**
df1h.raw	0	1024	0	0	1024	0	0	0
df1hvx.raw	540	484	0	0	484	332	0	208
df1v.raw	1024	0	0	0	1022	0	0	2
hd01.raw	695	214	36	79	741	177	35	71
hd02.raw	679	216	49	80	716	192	51	65
hd07.raw	381	127	182	334	447	158	152	267
hd08.raw	499	209	123	193	560	222	106	136
hd09.raw	200	246	239	339	263	244	228	289
hd12.raw	369	182	172	301	441	170	162	251
nk01.raw	780	146	22	76	891	90	11	32

Table 6: Applied zig-zag block traversals for adaptive compression

	Static compression without a model [B]	**Adaptive compression without a model [B]**	**Static compression with implemented difference model [B]**	**Adaptive compression with implemented difference model [B]**
df1h.raw	–	3	8	8
df1hvx.raw	54	54	93	93
df1v.raw	3	3	20	20
hd01.raw	80	80	85	85
hd02.raw	71	71	90	90
hd07.raw	86	86	81	81
hd08.raw	60	60	143	143
hd09.raw	83	83	94	94
hd12.raw	53	53	77	77
nk01.raw	93	93	87	87

Table 7: Compressed Huffman tree sizes in bytes

7 Performance Analysis

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	4063.66	183.44	3053.22	547.61	2615.22	200.48	2477.06	212.09	2651.40	219.48	13961.96	11473.40
df1hvx.raw	3005.82	137.20	2512.10	975.12	2046.40	154.87	1962.40	181.17	2100.52	162.77	11815.76	10517.39
df1v.raw	2202.20	132.18	1823.42	139.02	1661.02	144.02	1713.36	138.47	1934.66	172.79	11162.30	10579.77
hd01.raw	2949.50	148.29	2348.86	186.09	2128.62	140.06	2027.68	138.91	2152.98	157.87	11511.98	11094.16
hd02.raw	2929.92	175.79	2293.06	140.24	2145.44	162.37	2060.00	171.85	2200.00	174.62	14578.90	11713.34
hd07.raw	3411.50	153.97	2668.62	180.63	2278.68	187.11	2174.16	186.48	2284.70	184.68	13214.12	11494.07
hd08.raw	3145.90	168.67	2574.44	182.94	2203.20	163.22	2111.48	162.85	2217.52	162.09	12803.02	11300.56
hd09.raw	3952.78	245.45	2932.64	171.54	2504.34	154.45	2367.56	175.22	2437.26	172.54	10370.74	10076.15
hd12.raw	3478.12	200.76	2744.64	169.64	2355.86	166.73	2246.08	177.54	2342.92	170.50	15494.04	12469.59
nk01.raw	3747.36	199.44	2873.94	164.02	2536.40	435.64	2362.14	190.79	2458.32	214.06	16009.74	13376.36

Table 8: Performance static compression without a model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	4281.88	156.52	2964.32	134.35	2399.54	152.59	2171.28	170.70	2252.26	203.57	14203.72	11250.28
df1hvx.raw	4518.30	140.53	3113.54	137.55	2555.22	631.27	2159.52	166.14	2284.64	206.69	13161.34	11185.83
df1v.raw	4423.54	141.47	3001.76	156.62	2424.48	221.66	2185.74	179.27	2225.70	181.01	11477.02	10398.39
hd01.raw	4734.32	169.14	3237.40	166.83	2627.02	220.56	2347.22	155.28	2410.00	212.54	10801.14	10341.85
hd02.raw	4684.04	167.54	3242.12	164.84	2626.26	214.99	2301.26	169.20	2309.54	185.43	13630.88	11150.64
hd07.raw	5170.56	171.57	3553.64	173.61	2706.60	146.34	2444.62	196.55	2459.78	193.58	14122.86	11666.26
hd08.raw	5030.32	512.61	3500.68	164.61	2668.48	204.18	2362.56	216.34	2447.44	284.87	10161.82	9593.68
hd09.raw	5656.56	186.44	3834.78	205.19	2983.30	188.07	2592.58	184.19	3099.42	2845.51	15238.36	11792.53
hd12.raw	5249.28	188.52	3603.88	168.36	2832.66	181.16	2531.94	202.35	2555.46	214.32	11920.02	10451.98
nk01.raw	5679.84	543.28	3819.80	336.91	2969.56	204.47	2609.54	189.74	2619.60	210.40	12953.74	11935.36

Table 9: Performance adaptive compression without a model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	2332.02	121.38	1917.66	231.87	1800.42	166.89	1778.20	186.79	2002.94	215.18	12465.30	12377.45
df1hvx.raw	2719.34	147.74	2150.54	132.63	1995.54	168.65	1922.74	164.30	2080.66	172.10	13320.42	10996.21
df1v.raw	2312.96	132.18	1885.12	155.48	1748.82	159.82	1937.64	1089.92	1921.54	180.77	12225.24	11105.32
hd01.raw	3030.22	151.48	2382.28	167.78	2204.40	174.51	2114.96	179.31	2273.68	194.54	11242.12	10227.06
hd02.raw	2955.64	125.82	2367.38	164.57	2169.02	164.20	2101.48	171.75	2284.32	185.73	8950.98	8689.87
hd07.raw	3660.96	152.67	2803.52	150.07	2403.96	170.32	2296.02	213.64	2346.96	190.31	13764.04	11355.15
hd08.raw	3275.30	152.43	2673.96	149.64	2287.06	171.76	2216.24	405.33	2304.62	187.42	11429.18	10205.85
hd09.raw	4110.78	181.88	3027.14	158.44	2612.46	434.16	2399.18	207.70	2418.70	194.14	10728.06	10172.71
hd12.raw	3756.90	147.37	2895.96	165.68	2541.62	507.14	2281.94	187.10	2404.24	273.07	12482.74	10940.00
nk01.raw	3705.18	211.52	2824.96	193.68	2418.74	173.87	2357.72	233.29	2506.36	232.27	15084.36	11506.03

Table 10: Performance static compression with implemented difference model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	4163.08	155.64	2944.54	121.19	2304.54	147.51	2040.54	153.65	2093.90	144.16	14088.82	11182.34
df1hvx.raw	4423.50	127.66	3080.64	165.13	2441.42	140.64	2161.74	163.57	2186.36	170.28	13178.86	10509.04
df1v.raw	4134.66	163.52	2853.38	148.17	2268.92	173.05	2017.08	138.99	2082.46	166.28	12891.58	11083.63
hd01.raw	4877.02	181.16	3373.12	137.60	2760.26	161.73	2426.94	169.44	2399.98	171.87	12995.14	12394.14
hd02.raw	4782.24	176.90	3365.40	164.77	2691.48	167.66	2443.68	379.58	2381.94	226.85	15815.10	11558.13
hd07.raw	5389.88	180.24	3735.14	167.45	2900.82	167.92	2493.66	175.09	2556.84	210.97	12696.66	11235.75
hd08.raw	5099.32	146.10	3655.52	160.59	2787.28	186.15	2420.96	193.81	2433.06	201.25	10781.98	10369.51
hd09.raw	5794.54	183.28	4014.66	191.59	3121.28	182.50	2608.02	186.28	2604.50	337.23	12132.34	11160.99
hd12.raw	5401.40	138.60	3779.34	179.41	2947.02	179.02	2575.86	199.91	2518.92	184.99	13172.76	10799.70
nk01.raw	5564.14	186.14	3828.68	192.94	3046.20	182.80	2610.24	177.62	2706.14	195.70	16678.52	12506.43

Table 11: Performance adaptive compression with implemented difference model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	1232.48	175.39	1160.89	175.61	1175.11	169.11	1195.86	182.46	1184.99	174.23	1196.60	183.10
df1hvx.raw	4012.82	279.88	3040.37	202.02	2557.86	201.12	2387.40	187.33	2489.15	185.66	11109.04	10304.31
df1v.raw	1885.20	188.59	1890.14	199.43	1926.92	205.74	1997.57	195.14	2231.04	202.68	13190.25	11062.25
hd01.raw	3481.25	219.33	2690.52	180.53	2545.33	171.93	2402.90	192.26	2478.41	194.13	9346.53	9672.91
hd02.raw	3438.67	217.24	2672.73	185.29	2530.28	199.41	2414.13	382.81	2549.88	656.71	10207.01	9979.58
hd07.raw	4609.91	211.76	3265.88	180.77	2675.19	197.92	2485.88	227.12	2616.58	529.54	11593.91	10761.62
hd08.raw	3991.00	210.17	3187.90	285.57	2667.67	201.59	2413.78	211.36	2481.07	197.63	14044.70	11484.25
hd09.raw	5572.11	222.05	3805.96	177.38	2973.45	219.10	2614.43	198.30	2629.76	179.01	11501.46	10214.78
hd12.raw	4757.48	420.57	3363.72	181.01	2771.05	231.58	2450.66	206.68	2546.79	156.73	12786.35	11194.68
nk01.raw	5561.67	238.24	3762.13	147.49	2936.76	185.32	2614.73	198.13	2699.43	316.05	12107.95	10705.93

Table 12: Performance static decompression without a model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	3121.73	187.68	2851.64	201.83	2767.50	189.23	2893.85	261.49	3233.97	416.94	12150.18	10320.50
df1hvx.raw	3827.38	250.38	3208.29	194.76	3050.83	215.69	3202.81	288.34	4190.50	409.16	13809.14	10742.01
df1v.raw	3102.34	194.10	2824.79	176.14	2803.90	282.18	3429.68	326.52	4123.47	253.57	12863.21	10163.48
hd01.raw	3770.32	215.08	3187.36	196.07	3190.67	215.51	3443.74	275.07	4267.39	238.95	13041.57	10611.45
hd02.raw	3718.72	214.64	3185.13	199.37	3153.08	200.52	3305.38	262.22	4207.61	194.74	13354.73	10691.04
hd07.raw	4914.91	285.46	3775.46	195.50	3316.98	211.13	3408.92	410.34	4292.89	237.58	12954.44	10538.48
hd08.raw	4308.66	228.44	3714.92	881.86	3310.71	296.19	3409.28	261.71	4229.90	211.71	12141.40	10027.61
hd09.raw	5822.16	178.38	4254.33	210.13	3523.65	196.02	3383.70	427.18	4237.49	347.12	12646.28	9989.34
hd12.raw	4966.21	226.74	3836.81	193.05	3346.49	218.49	3361.74	337.33	4350.21	315.35	12857.33	10315.46
nk01.raw	5940.17	192.81	4254.72	224.00	3670.96	341.23	3849.99	234.37	4409.26	299.67	13978.61	10486.50

Table 13: Performance adaptive decompression without a model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	2151.53	655.14	1950.85	212.02	1966.93	185.58	2058.13	205.67	2261.99	207.61	12670.15	10866.77
df1hvx.raw	3288.37	311.87	2553.81	158.21	2365.42	187.02	2303.97	186.18	2445.66	409.18	11935.37	10351.16
df1v.raw	2172.01	1005.00	1989.12	176.85	1972.00	206.61	2075.31	186.42	2302.70	219.61	11242.42	10479.65
hd01.raw	3578.31	376.04	2808.76	201.43	2584.49	199.54	2431.34	206.93	2522.50	200.82	11881.47	10571.22
hd02.raw	3503.26	204.58	2769.45	190.73	2613.48	211.71	2434.12	217.65	2521.62	217.14	11199.50	10401.82
hd07.raw	4748.04	223.87	3436.73	751.55	2739.13	291.02	2475.48	159.29	2761.47	2068.23	10047.88	9673.71
hd08.raw	4122.97	212.01	3355.73	586.85	2766.11	542.08	2459.99	213.82	2584.14	569.92	11737.22	10745.76
hd09.raw	5710.36	229.39	3929.57	942.49	2999.26	196.51	2629.73	204.29	2645.69	211.48	12068.51	10470.36
hd12.raw	4802.80	213.77	3496.98	573.24	2794.63	192.02	2541.37	201.34	2637.68	298.39	12391.65	10886.27
nk01.raw	5678.82	207.95	3875.90	180.41	3011.77	200.99	2668.53	215.21	2709.61	452.63	11866.52	10438.84

Table 14: Performance static decompression with implemented difference model measured across 100 executions on 2.6 GHz

	1 thread [μs]		2 threads [μs]		4 threads [μs]		8 threads [μs]		16 threads [μs]		32 threads [μs]	
	mean	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**	**mean**	**std**
df1h.raw	2428.10	217.30	2452.74	174.10	2538.19	190.08	3344.44	280.71	4021.81	196.58	14528.87	11058.21
df1hvx.raw	3252.83	207.24	2874.31	181.84	2871.15	192.41	3052.04	331.91	4040.22	337.04	13688.53	10380.42
df1v.raw	2384.75	191.03	2442.99	191.54	2534.66	193.12	3390.11	608.31	4080.01	212.57	12604.36	10282.27
hd01.raw	3909.10	182.72	3245.94	193.86	3233.71	194.80	3647.96	312.98	4401.65	477.45	10854.57	9207.66
hd02.raw	3821.51	196.67	3228.33	211.96	3207.27	233.23	3449.14	302.66	4369.31	398.09	12090.47	9909.67
hd07.raw	5092.46	496.90	3812.70	475.91	3354.71	258.03	3399.71	299.36	4342.84	385.00	12668.76	9972.10
hd08.raw	4416.14	184.86	3695.93	196.83	3315.51	195.40	3406.01	295.95	4313.52	275.61	13481.09	10540.48
hd09.raw	5933.21	243.66	4291.50	189.18	3608.85	238.47	3399.00	242.64	4414.08	385.44	13032.09	10123.86
hd12.raw	5087.27	212.61	3916.39	174.86	3383.42	196.42	3476.96	985.33	4381.64	406.14	12846.45	10159.08
nk01.raw	6036.24	277.62	4340.69	199.72	3718.08	569.85	3859.60	253.88	4428.66	307.16	12425.93	9440.54

Table 15: Performance adaptive decompression with implemented difference model measured across 100 executions on 2.6 GHz

References

[1] Cyan. Huffman revisited - part 3 - depth limited tree. http://fastcompression.blogspot.com/2015/07/huffman-revisited-part-3-depth-limited.html, 2015. Accessed: 2024-05-06.

[2] Guilford, J. D., and Nguyen, K. T. Fast computation of huffman codes. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-computation-of-huffman-codes.html, 2016. Accessed: 2024-05-06.