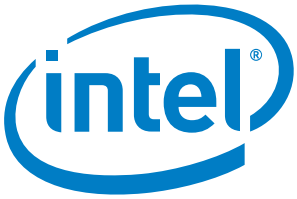


White Paper

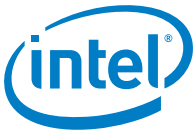
James T Kukunas
Vinodh Gopal
Jim Guilford
Sean Gulley
Arjan van de Ven
Wajdi Feghali

IA Architects
Intel Corporation

High Performance ZLIB Compression on Intel® Architecture Processors

April, 2014

330343-002



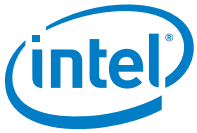
Executive Summary

The need for lossless data compression has grown significantly as the amount of data collected, transmitted, and stored has exploded in recent years. Enterprise applications and storage, such as web servers and databases, are processing this data and the computational burden associated with compression puts a strain on resources. To help alleviate the burden, we introduce an optimized industry standard DEFLATE implementation that can be used in common libraries such as Zlib.

This paper describes a high performance implementation of Zlib compression on Intel processors. Because the performance of compression implementations is data dependent, we use an industry standard data set to base our comparisons. We demonstrate substantial performance gains for all nine levels of Zlib compression, with comparable compression ratios to the baseline (except for level-1). We also introduce a new level-1 that provides significantly greater performance at the cost of some loss in compression ratio.

*Our high performance Zlib compression implementation is ~**1.8X** as fast as the latest available version of Zlib compression (1.2.8) for the default level 6 compression, on the Intel® Core™ i7 processor 4770 processor (Haswell).*

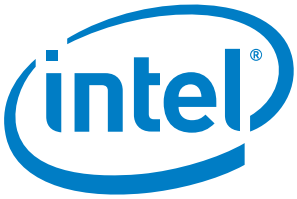
The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.



Contents

Figures

Tables



Overview

This paper describes a fast implementation of Zlib compression on Intel processors. All nine levels of compression have been improved, with a new level-1 that has much greater performance with some loss in compression ratio. The rest of the levels do not change the compression ratio, but provide substantial performance gains. Our implementation maintains ABI compatibility with Zlib, and thus functions as a drop-in replacement.

DEFLATE Compression

Background

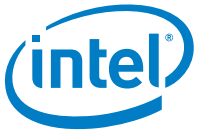
DEFLATE is an IETF standard described in RFC 1951. Aside from this “raw” format, DEFLATE can be also encoded as a Zlib stream, described in RFC 1950, or a GZIP file, described in RFC 1952.

The DEFLATE compressed data format consists of a series of blocks, corresponding to successive blocks of input data. Each block is compressed using a combination of the LZ77 [3] algorithm and Huffman coding [2]. The LZ77 algorithm finds repeated substrings and replaces them with backward references (relative distance offsets). The LZ77 algorithm can use a reference to a duplicated string occurring in the same or previous blocks, up to 32K input bytes back.

A compressed block can have either static (fixed codes defined in the standard) or dynamic Huffman codes. Each dynamic block consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part and the compressed payload. The compressed data consists of a series of elements of two types: literal bytes and pointers to replicated strings, where a pointer is represented as a pair <length, backward distance>. The DEFLATE format limits distances to 32K bytes and lengths to 258 bytes, with a minimum length of 3.

Each type of token or value (literals, distances, and lengths) in the compressed data is represented using a Huffman code, using one code tree for literals and lengths and a separate code tree for distances.

Tradeoffs between compression ratio and performance are encapsulated in nine compression levels, with level one representing the fastest performance and level nine representing the best compression ratio.



Our Implementation

We developed a highly optimized implementation for all nine levels of Zlib compression that works efficiently on data buffers of different types and sizes. We refer to this implementation in this paper as "**Zlib-new**." The optimizations focus on improved hashing, the search for the longest prefix match of substrings in LZ77 processing, and the Huffman code flow.

The original Zlib implementation consists of two primary DEFLATE strategies, DEFLATE_fast and DEFLATE_slow. The DEFLATE_fast strategy, used for levels one through three, is a greedy algorithm whereas the DEFLATE_slow strategy, used for levels four through nine, is a more exhaustive algorithm. Since multiple levels utilize the same strategy, the per-level tradeoffs between compression ratio and performance are controlled by per-level parameters, such as limiting the number of hash entries to search.

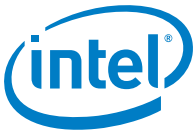
Our optimized Zlib-new implementation introduces two additional strategies, DEFLATE_quick and DEFLATE_medium. Our DEFLATE_quick strategy, used as a replacement for level 1, is based on ideas from igzip [10]. Since Zlib level 1 is generally chosen for cases where performance is more important than compression ratio, our strategy sacrifices a small amount of compressibility to achieve much greater performance. Our new DEFLATE_medium strategy is designed to maintain comparable compressibility with improved performance. The mapping of our strategies to Zlib levels can be described as:

DEFLATE_quick:	1
DEFLATE_fast:	2 – 3
DEFLATE_medium:	4 – 6
DEFLATE_slow:	7 – 9

We also use the CRC32 instruction improving both hash performance and quality. This impacts the speed of the hashing operating itself, but also improves the string-matching time, due to the good quality of the hash.

Improved Hashing

Zlib uses a hash table to quickly look up the location of previous occurrences of a given string in the sliding history window. Because the minimum DEFLATE match length is three, the original Zlib implementation hashes elements as triplets.



For levels 1 through 5, Zlib-new hashes elements as quadruplets, thus each hash entry corresponds to a match of four or above. This has the effect of shortening the hash chain and increasing the quality of each match, at the opportunity cost of matches of length three. To maintain the superior compressibility of levels 6 through 9, our Zlib-new does not diverge from the original hashing elements as triplets.

On systems supporting SSE4.2, we leverage the CRC32 instruction to provide a fast and high quality hash function.

New DEFLATE Quick Strategy

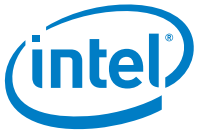
DEFLATE_quick is designed specifically for level 1, where maximum performance is favored over a high compression ratio.

DEFLATE_quick limits its hash chain search to the first entry. This takes advantage of the fact that we hash quadruplets instead of triplets for level 1, and thus any match will be at least of length four. Since the first hash entry corresponds to the most recent prior occurrence we will also have the shortest distance, which best leverages Huffman encoding.

The fixed Huffman trees defined in the DEFLATE specification are always used. Typically during the DEFLATE loop Zlib tallies each literal and distance pointer. To finish compressing a block, a dynamic Huffman tree is computed based on the tallies and compared with the standard fixed Huffman tree. The block is then compressed using the tree which provides the best encoding. By forcing the use of the fixed tree, we trade off potential gains in compression ratio for the much faster ability to output compressed data immediately.

Because the exact Huffman tree is known during the DEFLATE_quick inner loop, we use pre-computed lookup tables for generating the Huffman distance and length codes. Constructing the Huffman code for each distance-length pair typically involves four to six table lookups and three branches. By utilizing lookup tables, generating the Huffman code only requires two table lookups and zero branches. While the two lookup tables are larger than the previous six tables, they still fit within the CPU data cache. To facilitate this, we limit the history window to 8 KB.

An optimization in the implementation of DEFLATE_quick is to use an inlined string comparison function that leverages the PCMPESTRI SSE4.2 instruction.



New DEFLATE Medium Strategy

DEFLATE_medium is designed to strike a balance between the compression ratio of the DEFLATE_slow strategy, and the performance of the DEFLATE_fast strategy.

The DEFLATE_slow strategy leverages lazy match evaluations to improve compression ratio. When a match is found at position p, the position and length are temporarily stored while the next position, p + 1, is evaluated. If p + 1 does not provide a better match, the match at p is committed. If a better match is found at p + 1, then the element at position p is outputted as a literal. The match at position p + 1 is now temporarily stored and the position p + 2 is evaluated.

This more exhaustive method used by DEFLATE_slow requires a significant number of searches and constitutes the majority of time spent in the higher Zlib levels. The DEFLATE_fast strategy eschews lazy match evaluations and skips forward by the match length after each match is found. DEFLATE_medium is designed to be a mixture of the two.

DEFLATE_medium skips forward by the match length after a match is found and supports a variant of lazy match evaluation. After a match is found at position p and of length l, rather than evaluating p + 1, DEFLATE_medium checks for a match at position p + l + 1. If a new match is found, DEFLATE_medium scans backwards from the second match, at p + l + 1, to determine whether the second match can be improved. If so, it adjusts the first and second matches accordingly.

Consider the example presented in Figure 1.

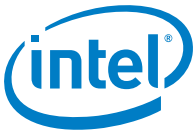
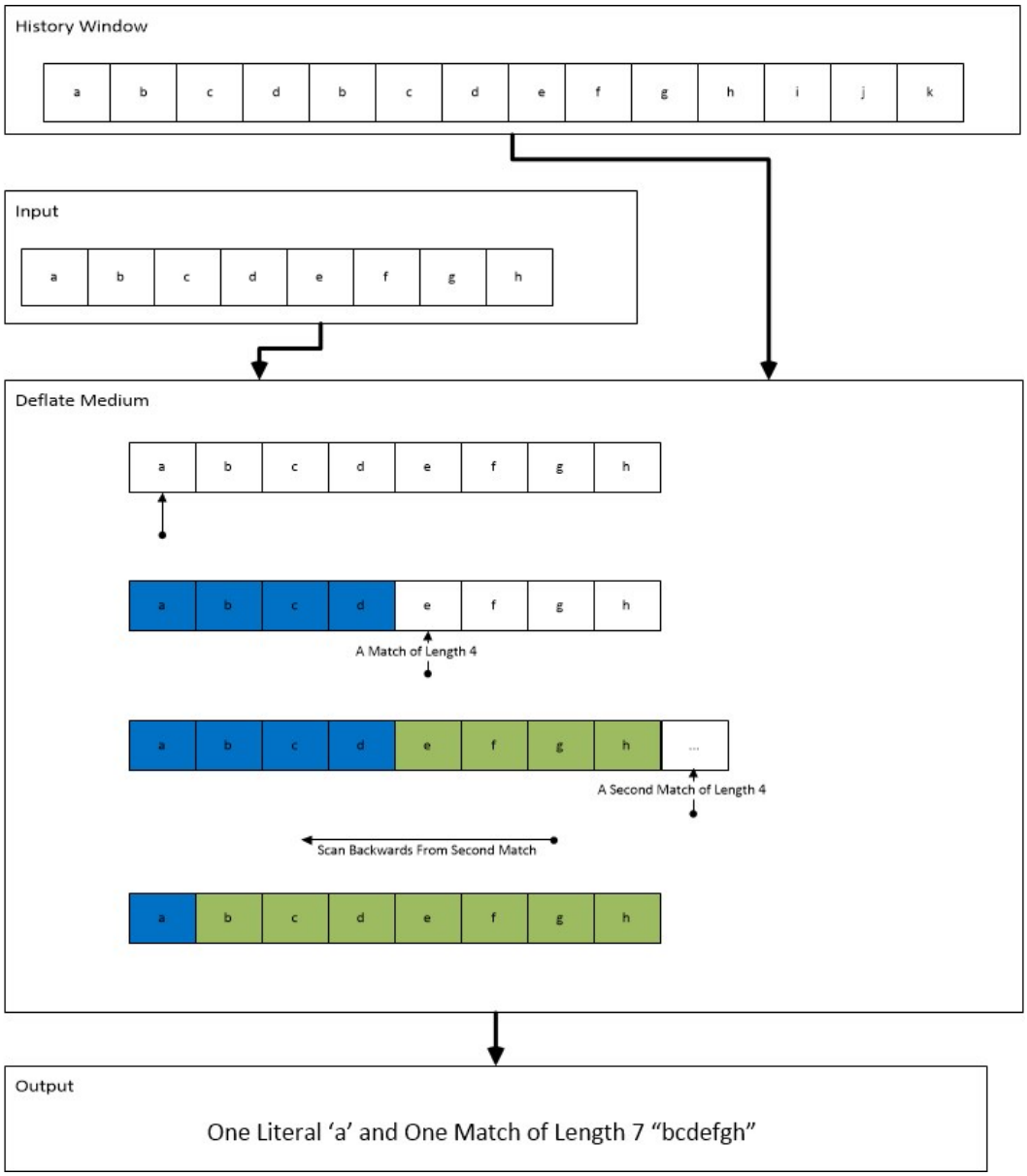
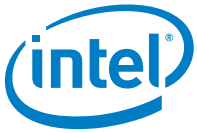


Figure 1: DEFLATE Medium Example



SSE Saturated Subtractions for Hash table shifting

To facilitate lookups in the sliding window, Zlib utilizes a hash table. Because this hash table entries store offsets into the window, the hash table contents must be shifted when the window contents are shifted. Zlib performs this operation in bulk whenever the window is shifted down to make room for more input.



The hash shifting iterates over the hash table, subtracting the shift amount from each hash entry index. A check for underflow must also be performed in case the hash entry points to content too old for the history buffer.

The original Zlib implementation performed this operation entry-by-entry, operating on two bytes at a time. This operation can be accelerated by leveraging SSE to operate on eight entries, sixteen bytes, at a time. The branch for underflow can also be avoided by utilizing the SSE instruction PSUBUSW that performs word subtraction with unsigned saturation.

Faster CRC calculations of fragmented data buffers

The GZIP file format requires the computation of the 32-bit CRC of the uncompressed data. Traditional CRC implementations use table-lookups to process one or four bytes of the input at a time. It is more efficient to use the PCLMULQDQ instruction to process 16 bytes at a time [4]. Due to the PCLMULQDQ latency, actually processing 64-byte chunks at a time gives greater efficiency. This approach requires the 64-bytes of result to be combined into 16-bytes of state and then reduced to a 4-byte final CRC.

It is advantageous to do the CRC calculation at the same time the data is being copied from the user buffer to the internal buffer.

One characteristic of the compression environment is that the entire buffer is generally not available at the same time, but rather blocks (with some arbitrary block size) are passed in until the final block. Rather than reducing the state from 64-bytes to 4 bytes at the end of every block, it is more efficient to maintain 64-bytes of CRC state until the very end and only do the reduction once.

The algorithm is:

1. Reduce by 512 bits until the remaining data is less than 1024 bits in size.
2. Perform 0…3 128-bit reductions until the remaining data is less than 640 bits in size.
3. Shift the block boundaries by adding zeros on the left to make the buffer 640-bits in size
4. Do a single 128-bit reduction, so that the final size is 512 bits.

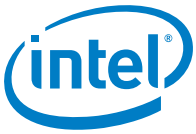
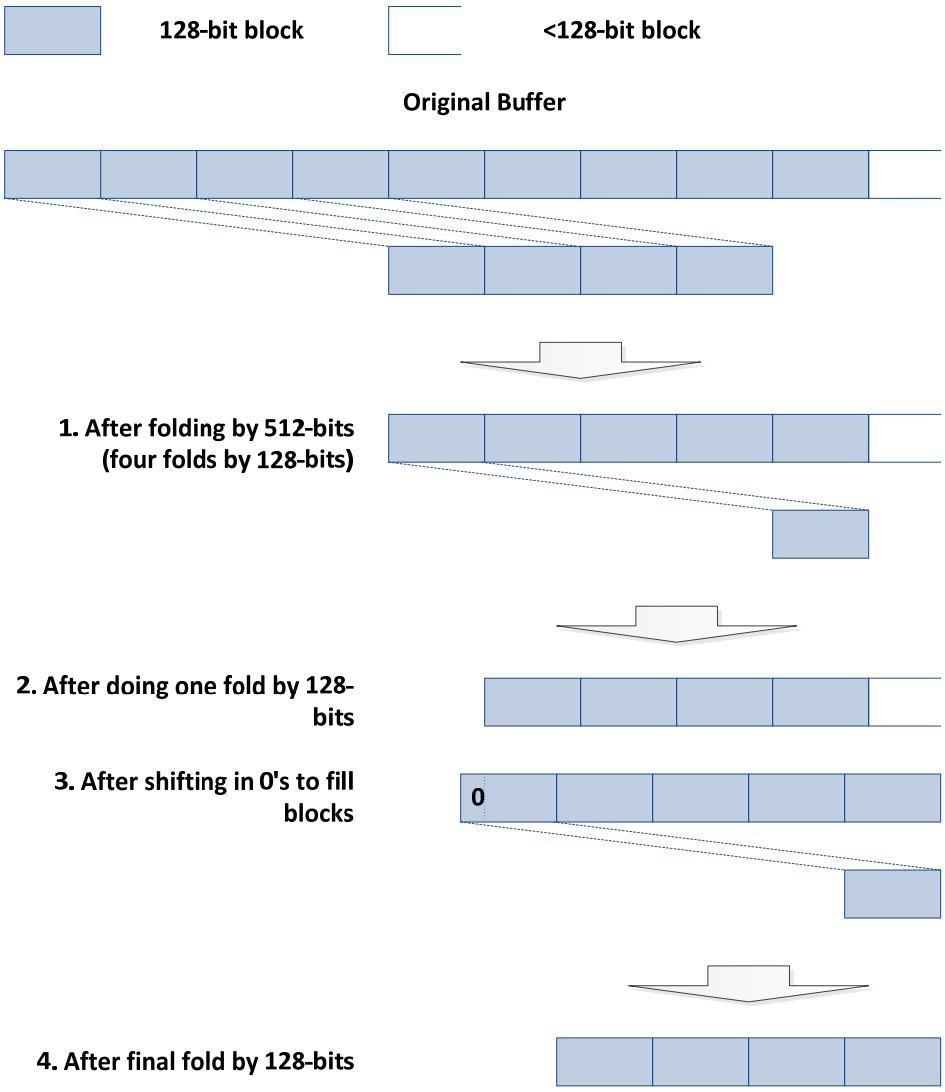
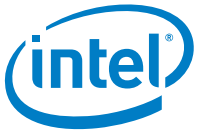


Figure 2: Fragmented Buffer CRC Calculation



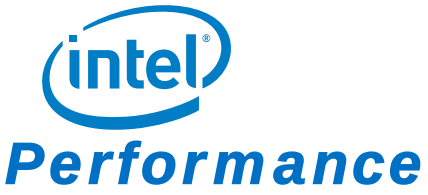


Reduce Loop Unrolling

The Zlib codebase has a long history and has previously been optimized for the platforms and compilers of yesteryear. Unfortunately, some of these optimizations do more harm than good on modern systems. One such case is the excessive loop unrolling in the Adler32 and CRC32 computations.

While loop unrolling can be a beneficial optimization, excessive loop unrolling can lead to additional register pressure and code size which actually hinders performance. Modern processors execute instructions out-of-order, effectively unrolling the loops in hardware with better scheduling of resources.

For the Adler32 computation, we reduced the unrolling factor from 16 to 8, and for CRC32 computation, we reduced the unrolling factor from 8 to 4.



The performance results provided in this section were measured on an Intel® Core™ i7 processor 4770 core (formerly Haswell), and an Intel® Core™ i7 processor 2600 core (formerly Sandy Bridge), supporting Intel® SSE 4.2 instruction-set. All test results are given in cycles/byte in order to provide an accurate representation of the micro-architecture's capabilities and to eliminate any frequency discrepancies. The tests were run with Intel® Turbo Boost Technology off, and represent the performance without Intel® Hyper-Threading Technology (Intel® HT Technology) on a **single core**. We present the results with data in memory, excluding file I/O, to reflect a variety of usage models.

Because the compression performance depends on the type and size of the compressed data, we study the performance on a set of industry-standard data sets.

We compiled the Zlib and Zlib-new implementations with the GCC Compiler with aggressive optimizations. We used version 1.2.8 of Zlib as the baseline for comparison.

The performance results represent all compute elements such as LZ String processing, the actual process of encoding the symbols, CRC, etc., excluding File IO time.

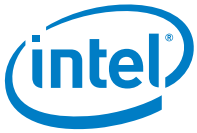
Methodology

To measure the performance of a compressor on a given file, we first read the file into a large memory buffer and make a single DEFLATE call on the entire buffer to warm up the caches. We now loop through repeated calls to DEFLATE on the entire file input memory buffer, measuring the cycles consumed by each iteration. When complete, the timings are sorted with the top and bottom quartiles discarded and the rest is averaged. This average provides a stable performance of the compressor for the particular file. The process is now repeated for each file in the given dataset.

The timing is measured using the **rdtsc**() function, which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The "TSC_initial" is the TSC recorded before the function is called. After the function is complete, the **rdtsc**() is called again to record the new cycle count "TSC_final." The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial).

A large number of such measurements are made for each file and then averaged as described above.



Note: *Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to:*
http://www.intel.com/performance/resources/benchmark_limitations.htm

Results

For the purposes of demonstrating the performance, we chose the well-known Calgary Corpus as the data-set. We show performance with a single thread in cycles/byte and compression ratio for the baseline Zlib and our improved Zlib implementation. The compression ratio is the compressed size as a percentage of the original size (i.e., lower/smaller is better).

We determine the averages based on summing the aggregate cycles and the aggregate bytes of the uncompressed files and computing the ratio. This method weights the average more towards bigger files. An alternate method would be to average the cycles/byte of each file, which weights all files evenly. However, as both methods give close performance we chose the former method.

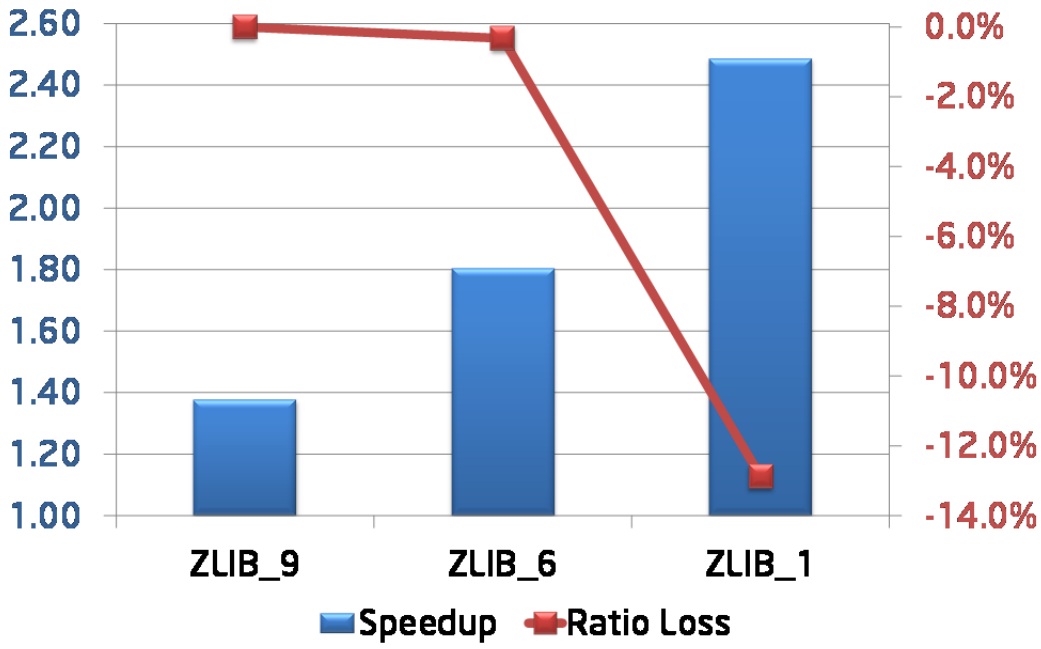
Table 1. Zlib-new Compression Performance (Cycles/Byte)

	i7-2600	i7-4770	i7-2600/i7-4770
ZLIB_1	20.46	17.66	1.16
ZLIB_6	93.86	88.10	1.07
ZLIB_9	215.23	209.51	1.03

Table 2. Zlib 1.2.8 Compression Performance (Cycles/Byte)

	i7-2600	i7-4770	i7-2600/i7-4770
ZLIB_1	47.26	43.91	1.08
ZLIB_6	166.33	158.96	1.05
ZLIB_9	296.51	288.58	1.03

Figure 3: Performance Gain and Compression ratio Loss of Zlib-new vs. Zlib



On average our optimized **Zlib-new** level-1 compression is ~**2.5X** faster than the baseline Zlib-1 on an i7-4770 processor. The large performance improvement does come at a cost in terms of compression ratio, with **Zlib-new** level-1 achieving an average of 51% compared to 38.1% achieved by Zlib-1.

The **Zlib-new** level-1 performance averages to ~**17.7 cycles/byte** on 1 i7-4770 core with a single thread.

With the Zlib default level 6 setting, **Zlib-new** compression is ~**1.8X** faster than the baseline Zlib-6 with only a tiny decrease in compression ratio.

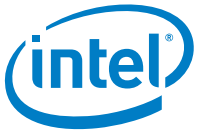


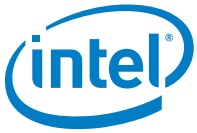
Table 3. Detailed Performance (Cycles/Byte) of Zlib-new Compression on Calgary Corpus

	ZLIB_1		ZLIB_6		ZLIB_9	
Filename	i7-2600	i7-4770	i7-2600	i7-4770	i7-2600	i7-4770
bib	21.52	18.83	87.62	81.61	157.33	153.52
book1	26.22	22.69	128.87	123.29	252.94	254.38
book2	22.21	19.30	106.70	101.57	185.73	183.36
geo	33.46	28.23	131.03	121.61	178.10	170.86
news	22.45	19.23	90.35	84.12	138.94	133.47
obj1	19.51	14.92	75.25	67.88	135.56	126.64
obj2	18.72	16.12	85.26	78.04	201.06	195.64
paper1	21.25	17.99	89.33	83.19	140.98	135.88
paper2	22.93	19.96	108.19	102.90	193.94	191.63
paper3	23.35	19.80	99.57	92.61	160.20	155.56
paper4	21.07	16.91	74.73	67.71	95.21	88.54
paper5	20.82	16.56	72.83	66.92	89.00	82.70
paper6	20.08	16.60	83.69	76.59	127.76	122.42
pic	8.19	7.33	39.30	34.83	334.71	313.29
progc	19.28	15.94	82.48	75.22	135.73	128.64
progl	14.61	12.54	76.31	70.37	194.77	190.56
progp	14.10	11.82	67.29	61.15	230.24	223.47
trans	14.63	12.75	58.66	52.65	113.72	108.19
TOTAL	20.46	17.66	93.86	88.10	215.23	209.51

Conclusion

The paper describes our fast implementation of Zlib Compression, **Zlib-new**, optimized for Intel® Processors. The **Zlib-new** level-1 i7-4770 performance averages to ~**17.7 cycles/byte** across the Calgary Corpus on a single thread. While using the Zlib default level 6 setting, **Zlib-new** compression is ~**1.8X** faster than the baseline Zlib-6.

With this release we hope to enable new usages of Zlib compression that were not possible before. Usages that require DEFLATE but were picking lower compression levels such as 1 or 2 due to performance constraints, may now be able to afford the zlib-6 default compression and benefit from the better compression ratio. The biggest advantage could be to the lower-end server market, i.e., the users who did not previously have the cycles to spare for DEFLATE.



Contributors

We thank Yann Argotti, Christophe Prigent, and Gil Wolrich for their substantial contributions to this work.

References

[1] DEFLATE Compressed Data Format Specification - RFC1951: http://www.faqs.org/rfcs/rfc1951.html

[2] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp. 1098-1101.

[3] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

[4] Fast CRC Computation for Generic Polynomials using PCLMULQDQ Instruction http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/fast-crc-computation-generic-polynomials-pclmulqdq-paper.html

[5] Zlib-new Implementation: https://github.com/jtkukunas/zlib

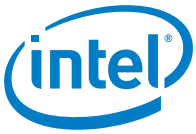
[6] RFC1950 - ZLIB Compressed Data Format Specification version 3.3 http://www.faqs.org/rfcs/rfc1950.html

[7] Calgary Corpus http://www.data-compression.info/Corpora/CalgaryCorpus/index.htm

[8] Canterbury Corpus http://corpus.canterbury.ac.nz/descriptions/

[9] Silesia Corpus http://www.data-compression.info/Corpora/SilesiaCorpus/index.htm

[10] High Performance DEFLATE Compression on Intel® Architecture Processors http://www.intel.com/content/www/us/en/intelligent-systems/wireless-infrastructure/ia-deflate-compression-paper.html



Authors

James T Kukunas, Vinodh Gopal, Jim Guilford, Sean Gulley, Arjan van de Ven, and Wajdi Feghali are IA Architects at Intel Corporation.

Acronyms

IA	Intel® Architecture
ILP	Instruction level parallelism
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions