Breaking Down Memory Walls

Francesc Alted <francesc@blosc.org>
August 9th, 2018

Year after year we continue to see a trend where CPUs get faster and faster in comparison with the evolution of memory speed. As a consequence, modern CPUs are underutilized and memory buses are often saturated, so why not take some of the storage operations out of memory and put them on the CPU caches?

In this installment we will see how to implement computational kernels on top of data structures that are cache- and compression-friendly, and we will examine how they perform on a range of different CPU architectures.

For demonstration purposes, I will run a simple task: summing up an array of values using the *super-chunk* data container in the Blosc2 library. While this seems trivial, it exposes a couple of properties that are important for our discussion:

1. This is a memory-bounded task.
2. It is representative of many aggregation/reduction algorithms that are routinely used out in the wild.

Datasets Considered

In this study I have chosen 2 different datasets:

1. **Synthetic**: This has been chosen so that compression, and especially decompression, reaches top compression and top speeds on Intel and ARM architectures.
2. **Real**: The one chosen is the result of a regional reanalysis covering the European continent, and in particular, the precipitation data of a certain region. Computing the aggregation of this data is representative of a catchment average of precipitation over a drainage area.

Caveat: In an attempt to make the task of compressing the synthetic dataset not totally trivial, rather than using e.g. a dataset of zeroes, I have chosen a monotonically ascendent series of integers that enforces the need for the shuffle filter (so that the codec to compress/decompress can run faster and more efficiently). This is still representative of a compression pipeline in Blosc2.

OpenMP: High performance for multi-processor systems

When running high performance multi-threaded computing tasks, OpenMP is the library that should be tried first. OpenMP is a series of so-called 'pragmas' that can be used to annotate existing code in C/C++ or Fortran for producing parallel applications. The complete code that benchmarks the sum aggregation is here.

This simple snippet implements the summing algorithm with OpenMP :

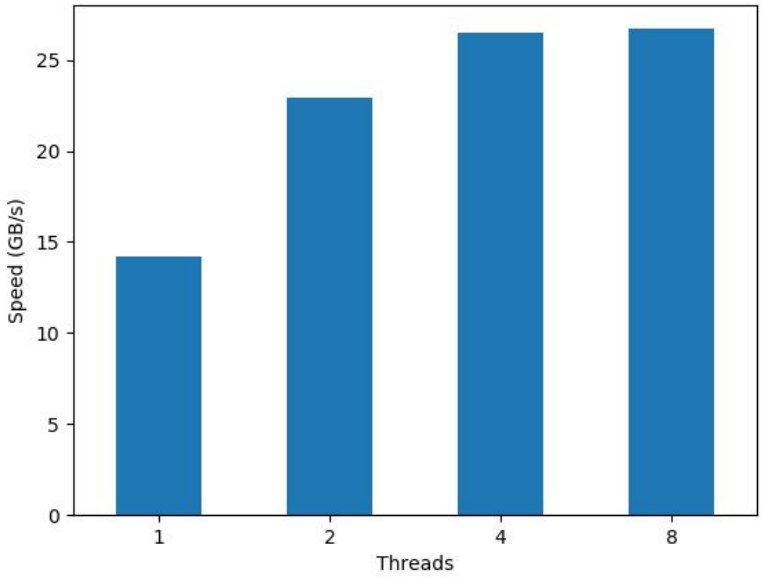
```
#pragma omp parallel for reduction (+:sum)
for (i = 0; i < N; i++) {
  sum += udata[i];
}
```

The pragma tells the compiler that it should try to parallelize the loop, and that there is a shared variable that is used for the reduction (namely 'sum'). In this case, this is enough to achieve peak performance on multi-core processors. I then run this task:

```
$ OMP_NUM_THREADS=4 OMP_PROC_BIND=spread ./sum_openmp
```

The `OMP_NUM_THREADS` environment variable specifies the number of threads that OpenMP should use. The `OMP_PROC_BIND=spread` variable tells OpenMP to not move threads among cores; rather, a sparse distribution across cores (respecting affinity policy and core nesting level) should be used. This normally gives optimal speed in this aggregation scenario.

On our reference server (Intel Xeon E3-1245 v5 4-Core processor @ 3.50GHz with hyper-threading) I get the following results for summing an array of 100 million integers (int64). I plot performance in GB/s, as this makes more sense for these kind of memory-bound problems:



So, although threading certainly improves performance, this increase is far from being linear, which is a clear sign that this computation is memory-bound. Interestingly enough, using several threads generally increases memory bandwidth in many systems, which most likely accounts for the performance improvement, rather than the increased computing capacity of multiple cores.

Just to give you an idea of how idle the CPU is during this simple task, if we replace the actual dataset by something that does not require to travel from memory, like:

```
#pragma omp parallel for reduction (+:sum)
for (i = 0; i < N; i++) {
  // sum += udata[i];
  sum += i;  // no need to transfer data from memory
}
```

With that, the speed of the aggregation can be accelerated by more than 5x. That means that, when the dataset has to be transmitted from the memory to the CPU, only 1 clock cycle every 5 is used for performing the aggregation; or put in another words, more that 80% of the CPU is idle when this task is running.

The central idea here is to put these idle CPU clock cycles to do some useful work (like decompressing data chunks) that can benefit our computation.

Operating with compressed datasets

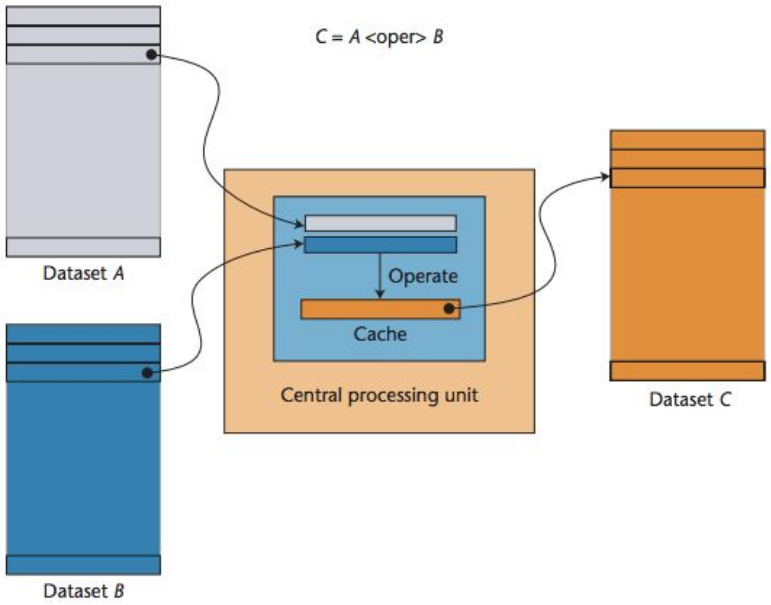
Now, let's see what happens when we perform the same operation, but using compressed data. To do this efficiently, we need:

1. A data container that supports on-the-fly compression.

2. A blocking algorithm that leverages the caches in CPUs.

For the data container we use the *super-chunk* data structure provided by Blosc2. These super-chunks host the data in chunks, so we shall iterate the operations over the complete chunks. And most importantly, in order to allow maximum efficiency when performing multi-threaded operations, the size of each chunk should fit in non-shared caches (namely, L1 and L2 in modern CPUs). This optimization avoids concurrent access to bus caches as much as possible, thereby allowing dedicated access to data caches in each core.

The algorithm I will implement on top of the super-chunk follows the principles of the blocking computing technique: for every chunk, we bring it to the CPU, decompress it (so that it stays in cache), run all the necessary operations on it, and then proceed to the next chunk.



Here's how the super-chunk is constructed:

```
schunk = blosc2_new_schunk(cparams, dparams);
```

```
for (nchunk = 0; nchunk < NCHUNKS; nchunk++) {
  for (i = 0; i < CHUNKSIZE; i++) {
    chunk_buf[i] = udata[i + nchunk * CHUNKSIZE];
  }
  blosc2_append_buffer(schunk, isize, chunk_buf);
}
```

Essentially we fill a chunk buffer one at a time, then we append it to the super-chunk (`schunk`) for compression and storage (`blosc2_append_buffer`). Note that we could have used pragmas to accelerate the build, but this is not our main goal, so I decided to keep it simple.

And finally, here's the algorithm for performing the sum of the compressed dataset:

```
compressed_sum = 0;
nchunks_thread = NCHUNKS / nthreads;
#pragma omp parallel for private(nchunk) reduction (+:compressed_sum)
for (j = 0; j < nthreads; j++) {
  dctx[j] = blosc2_create_dctx(dparams);
  for (nchunk = 0; nchunk < nchunks_thread; nchunk++) {
    blosc2_decompress_ctx(dctx[j],
                          schunk->data[j * nchunks_thread + nchunk],
                          (void*)(chunk[j]), isize);
    for (i = 0; i < CHUNKSIZE; i++) {
      compressed_sum += chunk[j][i];
    }
  }
}
```

Here we can see 3 different loops. The first loop deals with different threads, the second traverses the different chunks that are embedded in the super-chunk, and the inner loop calculates the actual sum per chunk. The OpenMP pragma is basically the same one we used in the uncompressed data sum example, but it is used only in the outer loop (for coarse grain parallelism). The important difference with the uncompressed loop example is that we introduced the two outer loops to 1) traverse the chunks and 2) take advantage of different decompression contexts (`dctx`) in every thread, so as to permit completely independent operations. This increased complexity is typical when dealing with data in **blocks** (or chunks), the common technique for optimizing algorithms introduced above.

As already stated, choosing a chunk size (`CHUNKSIZE`) that would fit in the non-shared cache is critical for best performance. How do we calculate what the chunk size should be? We will use a chunk size of 4,000 elements in our next experiments. This number is determined in the following way:

- We need two buffers, one for the source data and one for the destination data.
- Blosc2 needs a third internal buffer to keep intermediate results derived from the shuffle filter for decompression.
- We are using 64-bit integers for this exercise.
- So the final working chunk size that will fit in the caches will be 8 bytes (for a 64-bit integer) * 4,000 elements * 3 caches ~= 100,000 bytes. This should fit comfortably in L2 caches on most modern CPU architectures (256 KB or even higher).

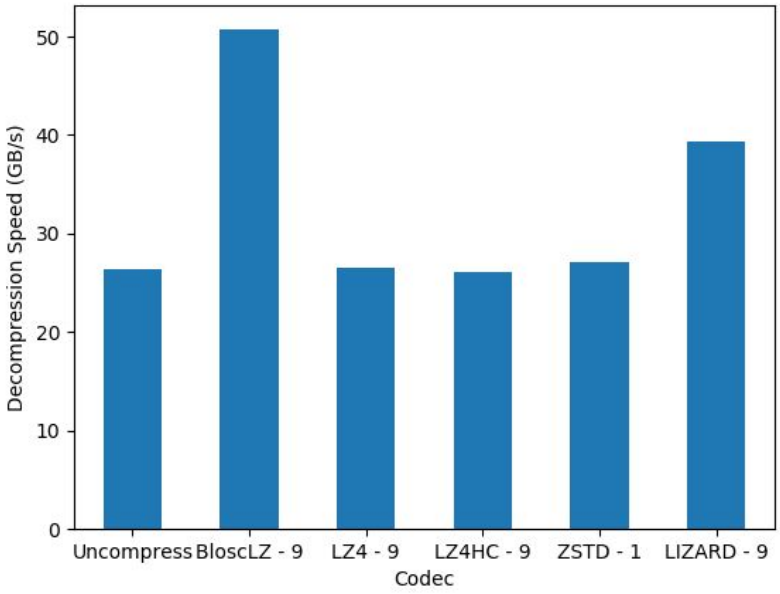
Please note that this size is critical, and might require some additional fine-tuning for different architectures (see notes for specific CPU experiments).

Synthetic Dataset

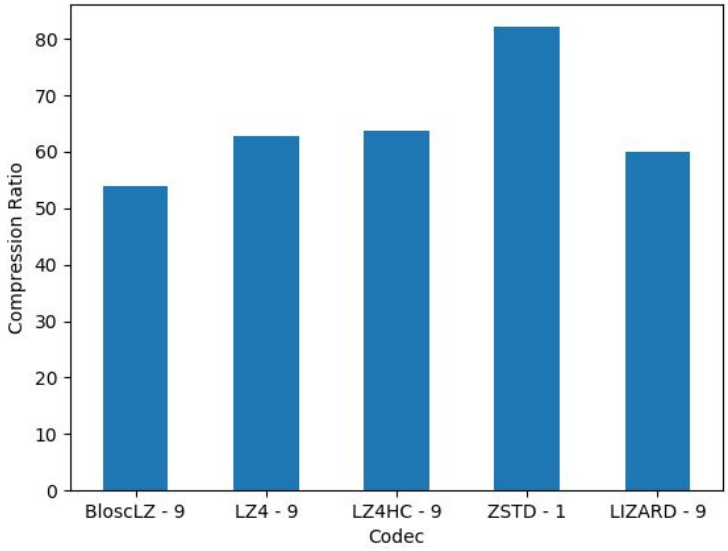
Let's start with the synthetic dataset and see how our compressed sum algorithm performs compared to the first uncompressed example. Since not all codecs inside Blosc2 (BloscLZ, LZ4, LZ4HC, Zstd and Lizard) nor CPUs are created equal (Intel, AMD and ARM have a large number of different implementations), we will measure how different configurations perform when running exactly the same computation. Some surprises are in store.

Choosing the Compression Codec

Blosc (both version 1 and version 2) supports different compression codecs. For Blosc2 the current set is BloscLZ, LZ4, LZ4HC, Zstd and Lizard (also Snappy, mainly for backward compatibility with Blosc1). Interestingly enough, my experience is that the different codecs expose different compression ratios and speeds when using different datasets or different CPU architectures. For example, for the synthetic dataset, BloscLZ usually decompresses the fastest, as can be seen in our reference platform (Intel Xeon E3-1245) when using 8 threads (for maximum performance):



As regards to compression ratio, other codecs are generally more capable than BloscLZ:

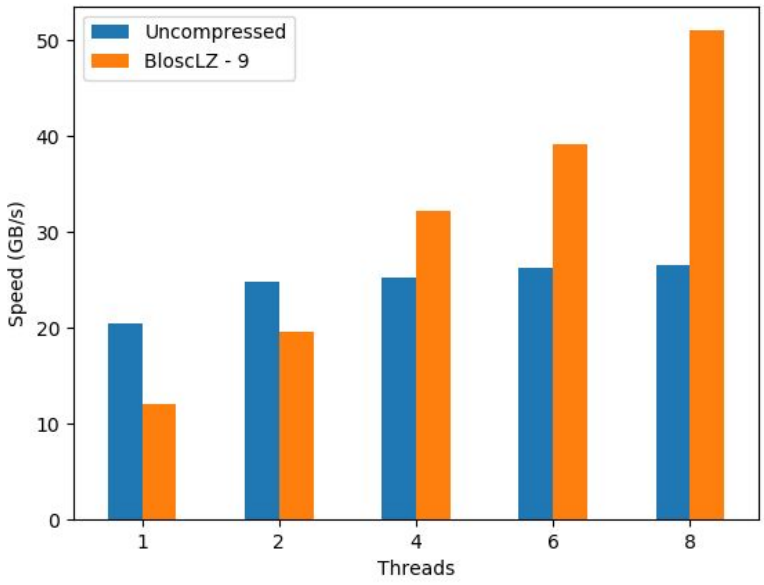


However, for this study we are more interested in having the best decompression speed, and this is why we are going to use BloscLZ with compression level 9 for all the benchmarks with synthetic data (incidentally, for the real data section things are not so easy and we will need to carefully choose our codec per every CPU as we will see).

Results on Different CPUs for the Synthetic Dataset

Reference: Intel Xeon E3-1245 v5 4-Core processor @ 3.50GHz

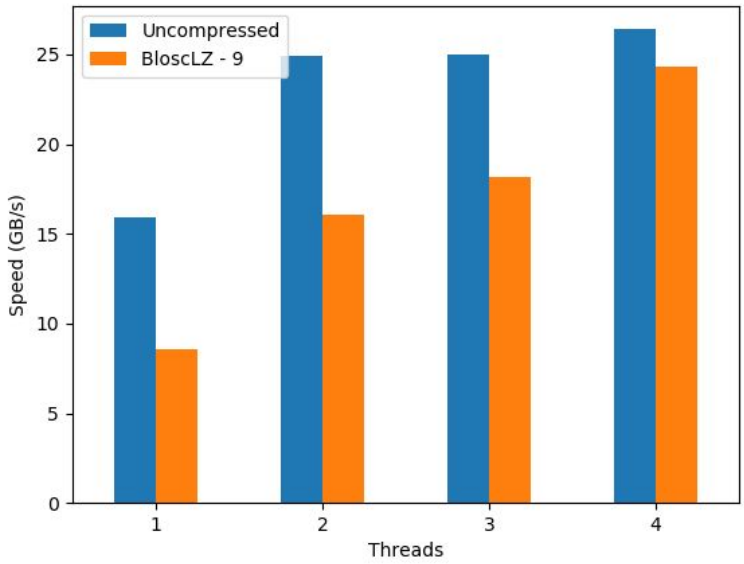
This is a mainstream, somewhat 'small' processor for servers that has an excellent price/performance ratio. Its main virtue is that, due to its small core count, the CPU can be run at considerably high clock speeds which, combined with a high IPC (Instructions Per Clock) count, delivers considerable computational power. These results are a good baseline reference point for comparing other CPUs packing a larger number of cores (and hence, lower clock speeds). Here it is how it performs:



We can see the poor scalability of the uncompressed dataset (although it does reach the respectable speed of 26 GB/s throughput), compared to using more threads on a compressed dataset. The compressed dataset scales much better, even beyond 4 threads. Considering that this CPU has 4 physical cores, it is evident that Intel devised a good hyperthreading implementation on this CPU, reaching a speed of 51 GB/s, and almost doubling the speed for the uncompressed dataset.

Intel i5 2-Core (6267U) processor @ 2,9 G

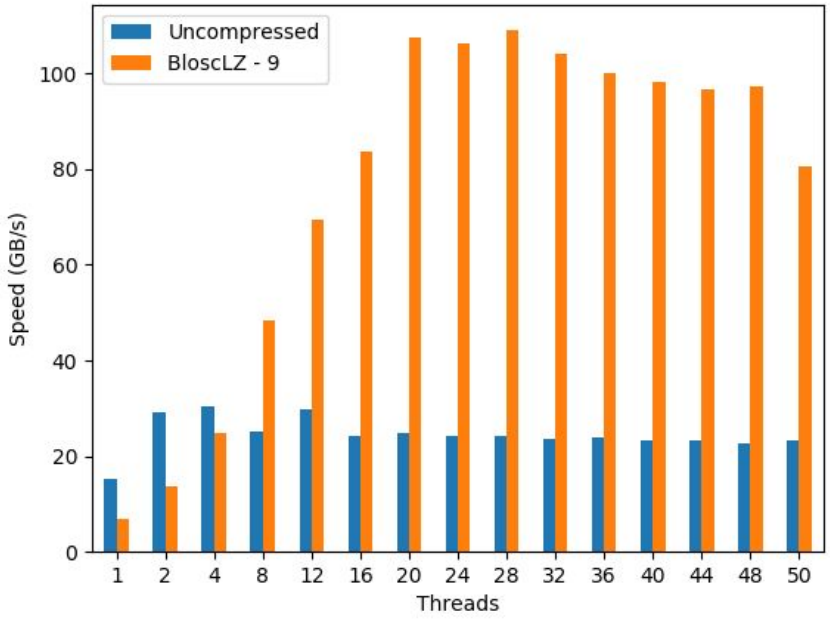
This is the processor included on my MacBook Pro (2016 model). It has just 2 cores with hyperthreading, but its clock speed is relatively high:



In this case, the bandwidth for reading memory is high (26 GB/s); at 24 GB/s, decompression is close to this mark. Hyperthreading is doing a good job here, although admittedly not as good as in our reference (probably because hyperthreading on server CPUs has more chance to shine than on laptop CPUs, but this is just a guess).

AMD EPYC 7401P 24-Core Processor @ 2.0GHz

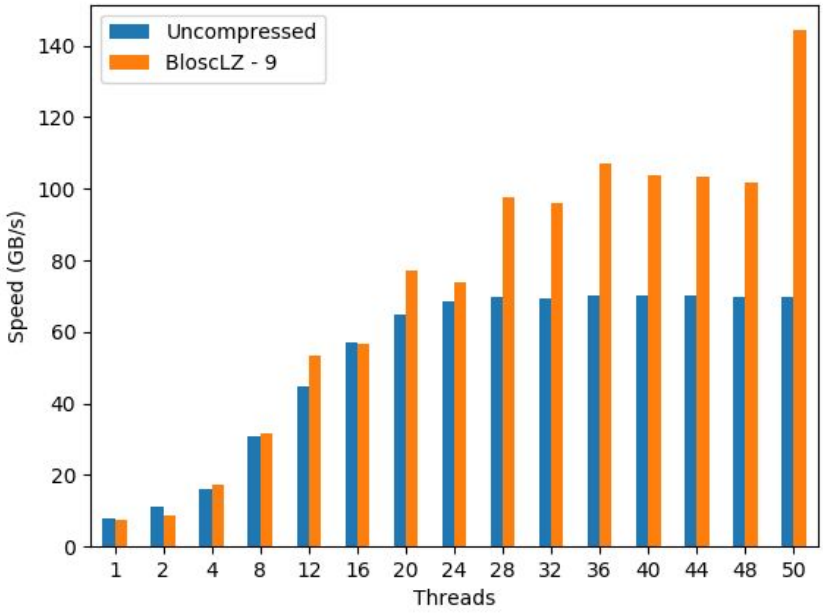
This CPU implements one of the most powerful architectures ever created by AMD. It packs 24 physical cores, although internally they are split into 2 blocks with 12 cores each. Here is how it behaves:



At about 30 GB/s, the highest speed for uncompressed data is reached when using 4 threads. Beyond that, there is no scalability at all. On the other hand, the compressed algorithm can exceed 100 GB/s at 20 threads, although adding more threads does not increase throughput. This a good example of how the new generation of multi-core architectures takes advantage of compression to deliver better performance.

Intel Scalable Gold 5120 2x 14-Core Processor @ 2.2GHz

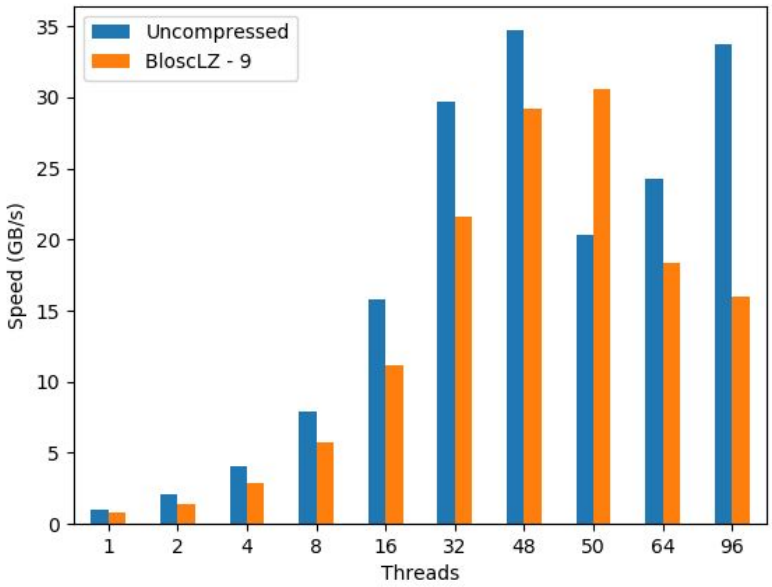
Here we have one of the latest and most powerful CPU architectures developed by Intel. We test it in a machine with 2 CPUs, each containing 14 cores. Here's it how it performed:



We reach the highest speed for the uncompressed operation with 24-50 threads, almost achieving the impressing speed of 70 GB/s; this is the first time that I see such a high speed for operations on uncompressed data on a real system. This boost is most likely due to the 6 high-speed memory access channels per CPU. On the other hand, the compressed dataset reaches its peak performance with 50 threads (no idea on why 50 is a magic number here), exceeding the impressive mark of 144 GB/s (another first!), consistently beating the performance of the uncompressed dataset starting at 20 threads.

Cavium ARMv8 2x 48-Core

We are used to seeing ARM architectures powering most of our phones and tablets, but seeing them performing computational duties is far more uncommon. This does not mean that there are not ARM implementations that cannot power big servers. Cavium, with its 48-core in a single CPU, is an example of a server-grade chip.

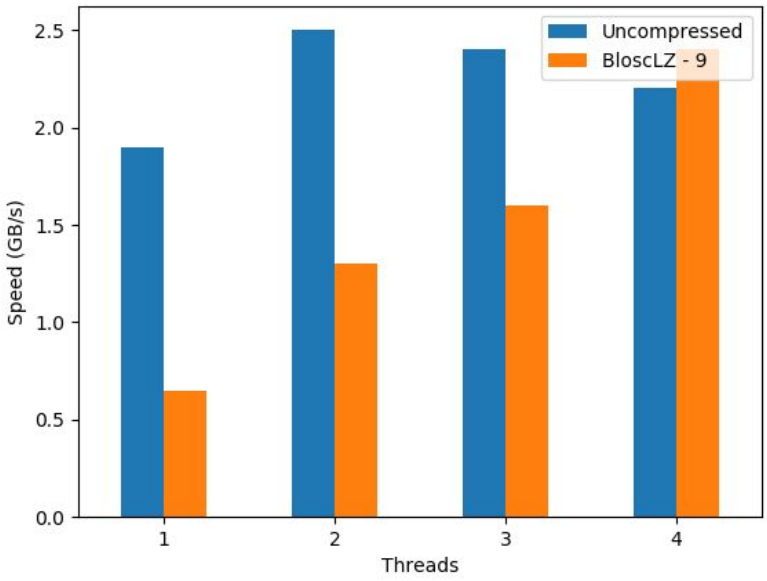


Here we see that for the uncompressed data the memory bandwidth scales quite linearly, reaching a 35 GB/s peak at 48 threads, and then follows a funny pattern. The performance for compressed data scales quite well also, reaching a peak of 30 GB/s at 50 threads, quite close to the uncompressed data algorithm which is an interesting fact for a supposedly 'weak' architecture like ARM. It turns out that Blosc2 has support for NEON, the specific SIMD instruction set for ARM CPUs, which it accounts for such high decompression speeds. While these figures cannot compete with modern Intel/AMD architectures yet (see above), they show that the ARM architecture can be a serious contender.

[As an aside, Cavium has just announced the availability of its new ThunderX2 CPU, which, with 32 high-performance ARMv8 cores and, most importantly, an impressive 8-channel DDR4 memory subsystem, may be able to rival even the most powerful architectures by Intel and AMD. Unfortunately, I have no access to this chipset If you do, I'd be interested in seeing your results from these tests run on this CPU.]

Raspberry Pi 3b+ 4-Core @ 1,4 GHz

Finally, and to close this CPU overview for the synthetic dataset, we will test one of the most popular and recent computer systems: the Raspberry Pi. In this case I tested the model 3b+, which sports a 4-core Cortex-A53 ARM CPU with NEON support.



In this case performance for uncompressed data peaks at 2.5 GB/s when using 2 threads, and does not seem to scale well after that; however, the performance with compressed data does, reaching approximately the same speed (2.4 GB/s) when using 4 threads. The fact that such a low-power CPU can execute decompression operations so effectively seems to contradict the general belief that you need a speedy CPU to break down the memory wall by using compression. However, provided that these CPUs are coupled with low memory-bandwidth subsystems, they can still decompress at par speeds than memory.

Incidentally, even though 2.5 GB/s may seem a small figure compared with e.g. our reference Intel Xeon platform (with a peak of 26 GB/s and 51 GB/s with compression), it is important to stress out that, at about 5 Watts with maximum load, the Raspberry Pi consumes a lot less power than a regular server (~100 Watts for our reference). And although the raw memory bandwidth per Watt is actually beneficial for ARM (0.5 vs 0.25 GB/(s * Watt)), compression actually equalizes this balance for both architectures (0.5 GB/(s * Watt)).

Note: For best performance in the Raspberry Pi, the total size of the dataset was reduced to 10 millions of elements and the chunksize was reduced to 1,000 elements (for a total of 24,000 bytes of working size, counting the internal buffer). This is probably due to fact that the L2 cache in this CPU is shared (remember that using non-shared caches for hosting buffers for blocks is critical for performance).

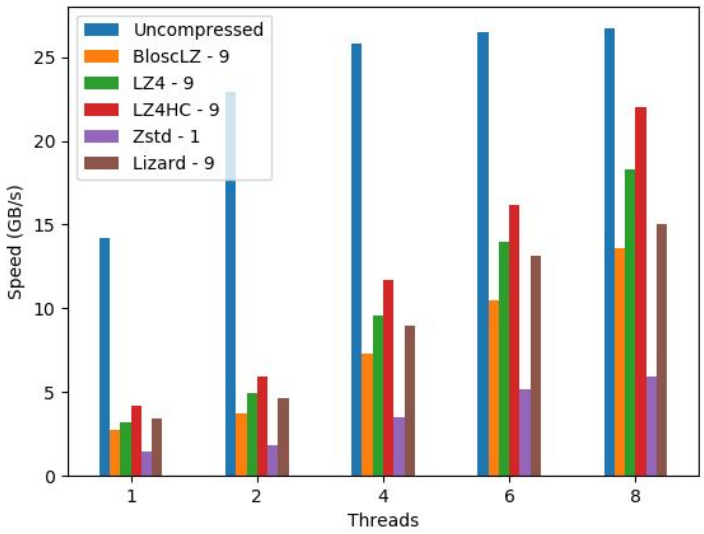
Precipitation Dataset

We have seen how compression can help in doing operations with synthetic, well-compressible data, but indeed using real data would be more interesting. However, there is all kinds of real data out there, and among them, I am going to use one that is of common use in my work: precipitation measurements. Other datasets will result in different results, so your mileage may vary.

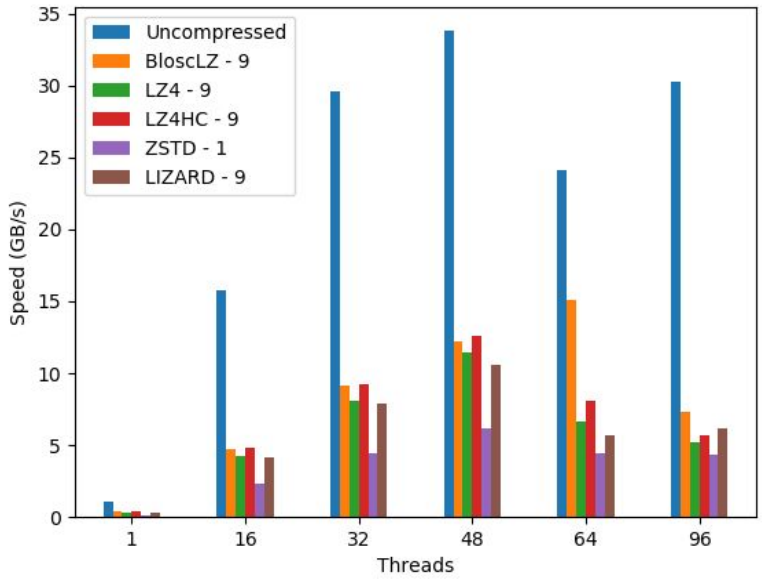
Caveat: For the sake of easy reproducibility, for building the 100 million dataset I have chosen a small geographical area with a size of 150x150 = 22,500 elements and reused these repeatedly so as to fill the final dataset completely. As the size of the chunks that we are using in this section are still 4,000 elements and the super-chunk, as it is configured here, does not use redundancies from other chunks, the results obtained in this section can be safely extrapolated to an actual dataset made of real data (bar some small differences). Also, the data type for the elements in this dataset is a *float32* (single precision floating point), quite usual in geo-sciences, and not an *int64* as in the synthetic dataset.

Choosing the Compression Codec

When determining the best codec for the precipitation dataset, it turns out that they behave quite differently both in terms of compression and speed. This is quite usual, and is the reason why you should always try to find the best codec for your case. Here you have how the different codecs behaves for our precipitation dataset in terms of decompression speed for our reference platform (Intel Xeon E3-1245):

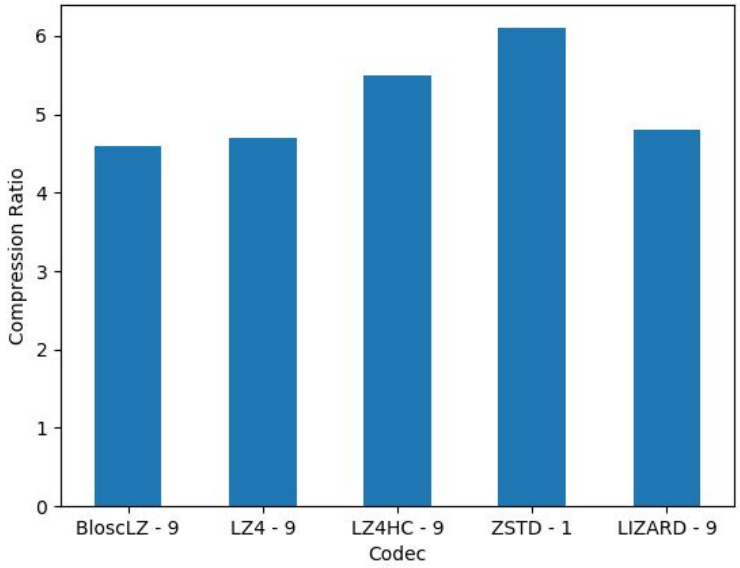


In this case LZ4HC is the codec that decompress faster for any number of threads, and the one selected for the reference platform. And here are the results for the Cavium CPU:



In this case is BloscLZ the one that achieves best *absolute* decompression speed. So what I did was to select the codec that could decompress faster for some *arbitrary* number of threads. The selected codec for every platform will be conveniently specified in the plots below.

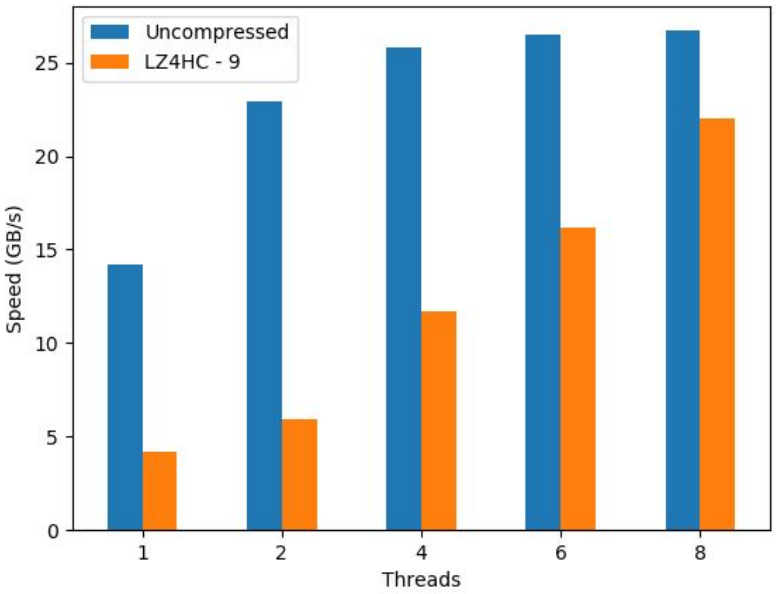
For completeness, here there are the compression ratios achieved by the different codecs for the precipitation dataset:



Although there are significant differences in the compression ratio, these usually come at the cost of compression/decompression time. Moreover, we are mainly interested in decompression speed, so we will use the latter as the only important parameter for codec selection.

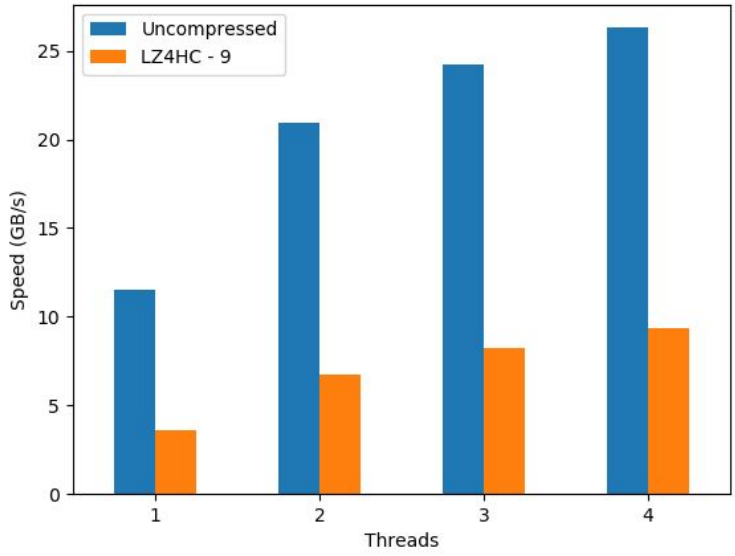
Results on Different CPUs for the Precipitation Dataset

Reference: Intel Xeon E3-1245 v5 4-Core processor @ 3.50GHz



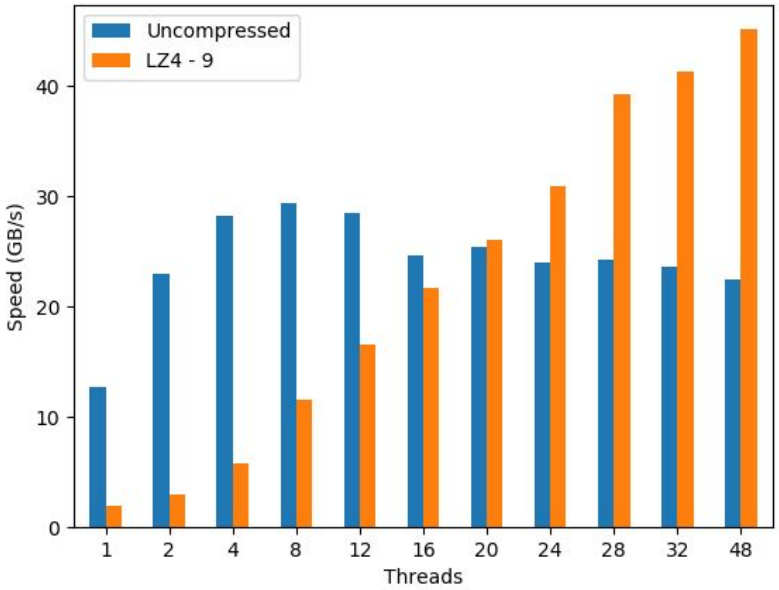
Although reaching less speed, here we see a behaviour that is similar to the synthetic data scenario: nice scalability on the compressed dataset even using hyperthreading. The performance peak for the compressed precipitation dataset (22 GB/s) is really close to the uncompressed one (27 GB/s); quite an achievement for a CPU with just 4 physical cores.

Intel i5 2-Core (6267U) processor @ 2,9 G



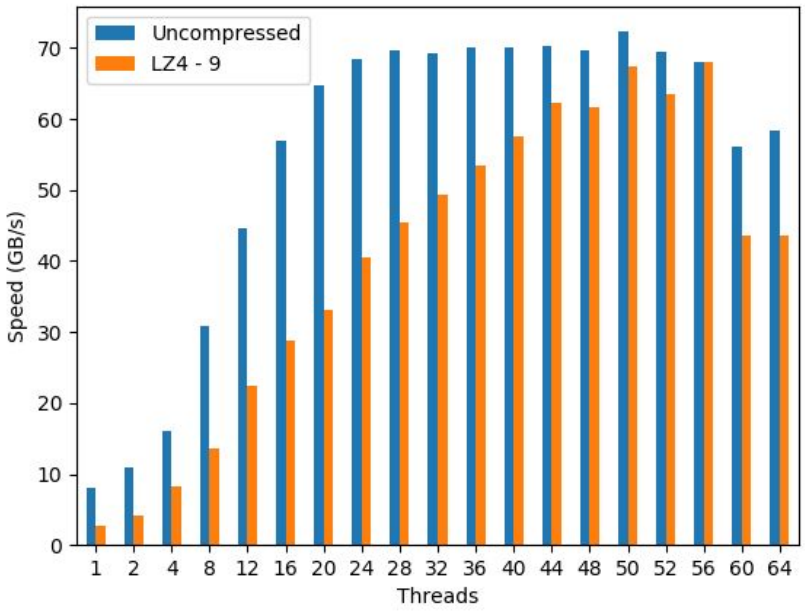
In the case of this laptop CPU we see that scalability for the compressed dataset is not as good as for our reference platform, and performance for the compressed dataset (a bit more of 9 GB/s) is quite far from the uncompressed one (a solid 26 GB/s figure). However, we should not forget that we are using the CPU with less number of cores (2) from all of our analysis.

AMD EPYC 7401P 24-Core Processor @ 2.0GHz



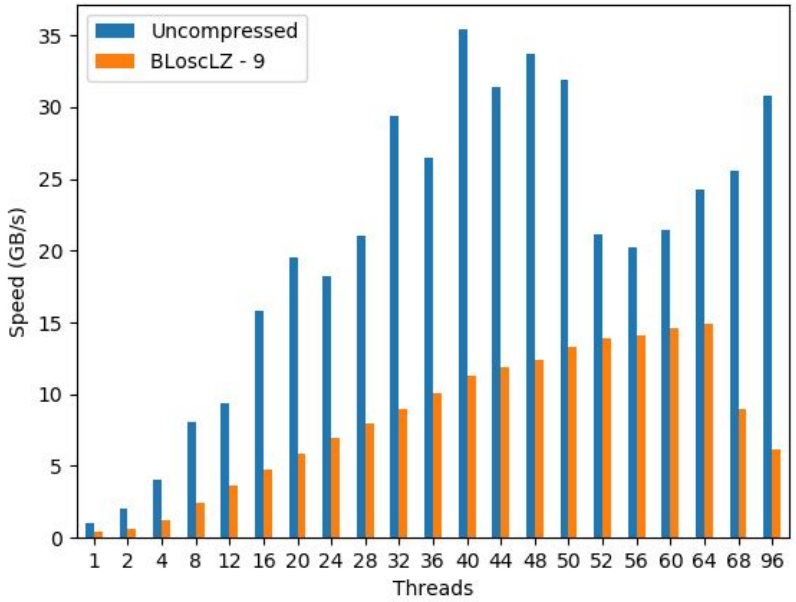
The EPYC behaviour for the compressed precipitation dataset is pretty amazing: it shows nice scalability through the whole range of cores in the machine --even when using hyperthreading--, reaching the best performance (45 GB/s) at precisely 48 threads, and well above the maximum performance achieved by the uncompressed dataset (30 GB/s).

Intel Scalable Gold 5120 2x 14-Core Processor @ 2.2GHz



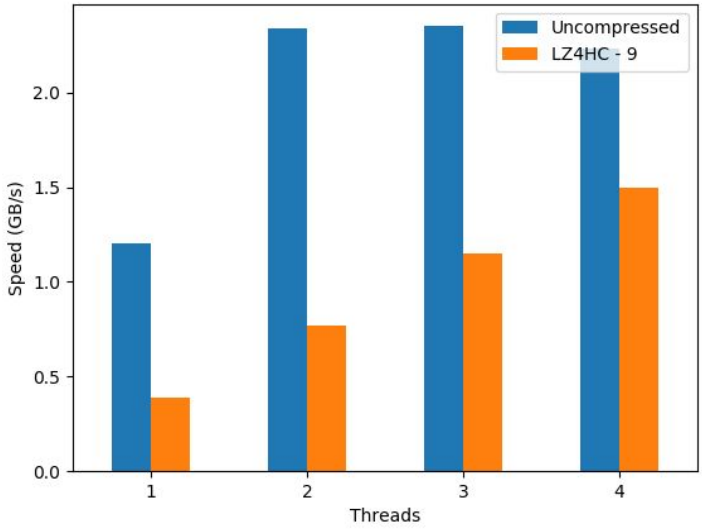
In this case, the Intel Scalable also shows a nice scalability on the compressed dataset, all the way up to 56 threads (which is expected provided the 2x 14-core CPUs with hyperthreading); this is a remarkable feat for a bandwidth beast like this. In absolute terms, the compressed dataset reaches a performance (68 GB/s) that is very close to the uncompressed one (72 GB/s).

Cavium ARMv8 2x 48-Core



Again, we see a nice scalability for the compressed dataset. It is curious to see how the performance peaks at 64 threads and then drops significantly after that point (even if the CPU still has enough cores to continue the scaling). At 15 GB/s, the best compressed mark is less than half of the best uncompressed figure (34 GB/s), but still in the same order.

Raspberry Pi 3b+ 4-Core @ 1,4 GHz



On its hand, the Raspberry shows a nice scalability throughout all the cores when processing the compressed dataset. However, the absolute compressed performance (1.5 GB/s) is still a bit far from the uncompressed one (2.4 GB/s). Probably another performance oriented CPU from ARM (like the Cortex A-7x instead of the A-53 here) would change this picture significantly.

Conclusions for the Precipitation Dataset

As expected, computational performance with real data is below of that when using a synthetic dataset, but we can observe two important points here:

- The loss in speed is of the *same order of magnitude* than for the synthetic dataset, not more.
- Performance for the compressed dataset scales very well on the number of threads (and in a similar way than with synthetic data). With that, and provided the right CPU and amount of dedicated threads, the performance of compressed computations can be on par (or even surpass, like the AMD EPYC case) of that of uncompressed data.

What We Learned

From all we have seen, we can draw the following conclusions:

1. Modern CPUs can use their idle computational power to alleviate the pressure that many applications put on their cache and memory subsystem. The simple aggregation task above is representative of a memory-bounded computation where compression can actually help to reduce stress on memory buses.

2. The systems that benefit the most from compression are those with relatively low memory bandwidth and CPUs with many cores. In particular, the EPYC architecture is a good example of this kind of system, where we have shown how compression can ameliorate the performance as compared to an uncompressed dataset.

3. ARM architectures, with their low power-consumption, theoretically one can pack many cores in the same CPU quite easily (Cavium being a nice example of this). In that sense, ARM will probably become one of the best suited platforms for leveraging compression in the future.

4. The appropriate codec to use within Blosc2 for maximum performance can vary depending on the dataset and the CPU used. Having a way to automatically discover not only the best codec, but also the best compression level and other compression parameters, would be a nice addition to the Blosc2 library.

5. And last, but not least, even if compression cannot always help in improving performance, it does normally improve memory and disk utilization, which is always a welcome feature in these Big Data ages. And the fact that it also puts less stress on memory buses could lead to better overall performance for other applications in the system (but that depends a good deal on what these other applications are).

We have seen that, with a basic understanding of the cache and memory subsystem, and by using appropriate compressed data structures, like the super-chunk in Blosc2, we can easily produce code that enables modern CPUs to perform operations on compressed data at a speed that approaches the speed of the same operations on uncompressed data (and sometimes exceeding it). The takeaway message is that, by leveraging these knowledge and techniques, we can achieve computational speeds that can be on par with (or even surpass) traditional, uncompressed computations, while saving precious amounts of memory and disk space.

Final thoughts

To conclude, it is interesting to remember here what Linus Torvalds said back in 2006 (talking about the git system that he created the year before) :

> [...] git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful.
> [...] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more

> important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Of course, we all know how drastic Linus can be in his statements, but I cannot agree more on how important is to adopt a data-driven view when designing our applications. But I'd go further and say that, when trying to squeeze the last drop of performance out of modern CPUs, data containers need to be structured in a way that leverages the characteristics of the underlying CPU, as well as to facilitate the application of the blocking technique (and thereby allowing compression to run efficiently). Hopefully, installments like this can help us explore new possibilities to break down the memory wall that bedevils modern computing.

Appendix: Software used

For reference, here it is the software that has been used for this blog entry.

- OS: Linux Ubuntu 18.04 (except MacOSX 10.13.4 for the i5 CPU and Raspbian April 2018 for the Raspberry Pi)
- Compiler: GCC 7.3.0
- C-Blosc2: 2.0.0a6.dev (2018-05-18)

Thanks

I’d like to thank all the people that suggested flaws and ways to improve this document during the different drafts. Scott Prater for coming with great advices on improving my writing style, Dirk Schwanenberg for pointing out to the precipitation dataset and for providing the script for reading it, and Robert McLeod, J. David Ibáñez and Javier Sancho for suggesting general improvements.