Compressing floating point data with Gorilla

Miikka Koskinen

Helsinki Python Meetup, June 2025

Agenda

- Part 1: Introduction
  - Who am I?
  - What is Gorilla?
  - What is time series data?
- Part 2: Floats
  - How do floating point numbers work?
- Part 3: Gorilla
  - How does the Gorilla algorithm work?
- Part 4: Is it any good?
  - Should you use Gorilla?

Part 1:
Introduction

My name is Miikka Koskinen

- I’m a software engineer
- Independent consultant at Jacksnipe Consulting Oy
  - Before that: Oura, Metosin, ZenRobotics, Futurice
- Interested in the problems you get when you have a lot of data
- Tech I use: Python, Rust, AWS

Read my blog at https://quanttype.net/

Gorilla

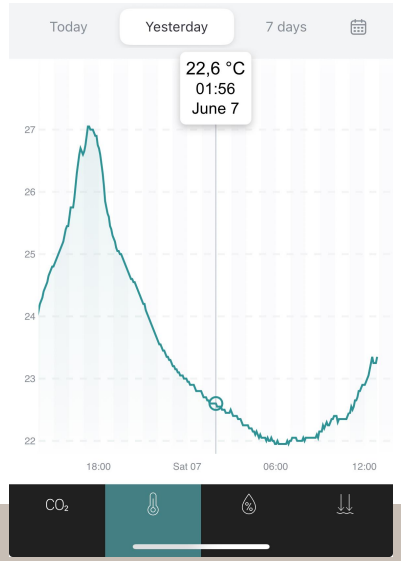
- Gorilla is/was a timeseries database developed by Facebook for monitoring their services
  - Described in a paper published in 2015.
  - Pelkonen, Franklin, Teller, Cavallaro, Huang, Meza, and Veeraraghavan: *Gorilla: A Fast, Scalable, In-Memory Time Series Database*.
- Includes a simple, clever scheme for compressing floating point data

Time series data

When you collect a series of measurements over time, you get time series data.

Examples: network latency, software memory usage, heartbeat, room temperature, humidity, electricity usage, location, stock price

Time(DD/MM/YYYY H:mm:ss) timestamp	Carbon dioxide(ppm) int64	Temperature(°C) double	Relative humidity(%) int64	Atmospheric pressure(hPa) double
2025-06-07 16:01:32	750	25.7	41	1000.7
2025-06-07 16:06:32	687	25.6	41	1000.7
2025-06-07 16:11:32	648	25.6	40	1000.8
2025-06-07 16:16:32	616	25.8	40	1000.8
2025-06-07 16:21:32	609	26.2	39	1000.8



Part 2:
Floats

What are floats used for

Floats solve two problems:

- How to represent non-integral numbers like 1.25?
- How to represent large or small numbers like 77 000 000 000 000 000 000 or 0.000 000 000 000 62?

Python uses floats, too! Specifically IEEE-754 binary64 i.e. double precision floats

```
>>> type(1.25)
<class ‘float’>
```

Float decomposition

Floats work by decomposing numbers like this:

$$x = (-1)^{sign} * 2^{exponent} * 1.significand$$

Where sign is 1 bit, exponent 11 bits and significand 53 bits. Example:

$1.25 = (-1)^0 * 2^0 * 1.01_2$ (sign = 0, exponent = 0, significand = 1.01_2)

Because decimal numbers can't be perfectly decomposed to binary floats, you get this:

```
>>> 0.1 + 0.2

0.30000000000000004
```

Demo

Go to https://float.exposed/0x3ff0000000000000

Part 3:
Gorilla

The main insights in Gorilla

Gorilla consists of two “tricks”:

- xorring the consecutive values
- Encoding the results carefully

xor: exclusive or

- Xor is a bitwise operator that yields 1 when exactly one of the inputs is 1. A truth table:

	0	1
0	0	1
1	1	0

- In Python, you can xor two bools or integers with the ^ operator

```
>>> 1 ^ 0
1
>>> 0b110 ^ 0b011
5
>>> f"0b{_:b}"  # show the result as a bit string
'0b101'
```

The xor trick: it zeroes out the bits that are the same

Input	Binary representation
20.5	`01000000001101001000000000000000000000000000000000000000000000`
21.0	`01000000001101010000000000000000000000000000000000000000000000`
20.5 ^ 21.0	`00000000000000011000000000000000000000000000000000000000000000`

The xor trick: it zeroes out the bits that are the same

Input	Binary representation
20.5	`01000000001101001000000000000000000000000000000000000000000000000`
21.0	`01000000001101010000000000000000000000000000000000000000000000000`
20.5 ^ 21.0	`00000000000000011000000000000000000000000000000000000000000000000`

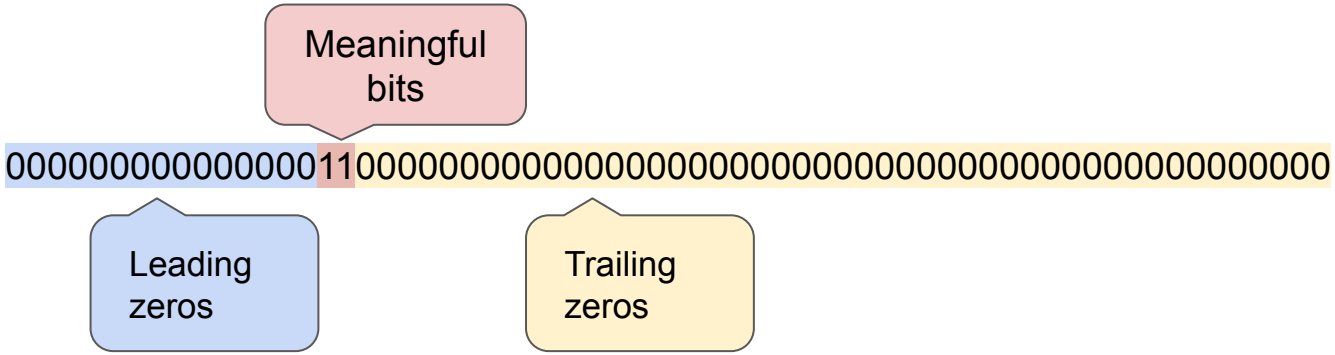
The xor trick: xor is reversible!

Input	Binary representation
20.5	01000000001101001000000000000000000000000000000000000000000000
21.0	01000000001101010000000000000000000000000000000000000000000000
20.5 ^ 21.0	00000000000000011000000000000000000000000000000000000000000000
(20.5 ^ 21.0) ^ 21.0	01000000001101001000000000000000000000000000000000000000000000

The xor trick: xor is reversible!

Input	Binary representation
20.5	`0100000000110100100000000000000000000000000000000000000000000000`
21.0	`0100000000110101000000000000000000000000000000000000000000000000`
20.5 ^ 21.0	`0000000000000001100000000000000000000000000000000000000000000000`
(20.5 ^ 21.0) ^ 21.0	`0100000000110100100000000000000000000000000000000000000000000000`

Terminology for the xor result



Encoding it carefully

- Store the first value as-is. For each value after the first, calculate the xor with the previous value and store that.

- How to store the xor result:
  - If the result is all zeros, emit bit ‘0’
  - If the meaningful bits fit within the block of previous meaningful bits, then emit ‘10’ and the meaningful bits
  - Otherwise, emit ‘11’, then the number of leading zeros (5 bits, value 0-31), then the number of meaningful bits (6 bits, value 1-64), and finally the meaningful bits

Demo

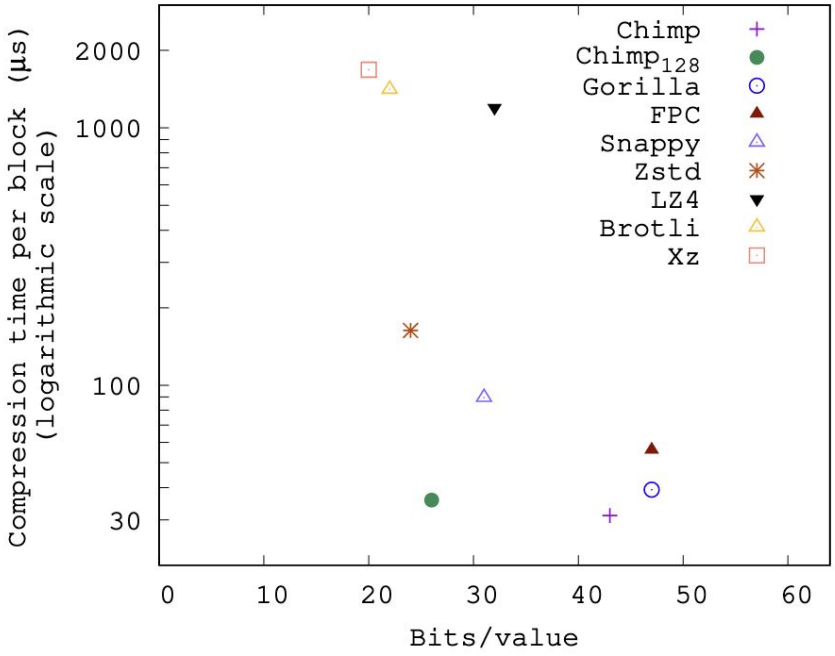
Go to
http://localhost:5001/?floats=20.5%0D%0A21.0%0D%0A21.0%0D%0A21.2%0D%0A21.1%0D%0A20.9&step=1

Quick look at code!

https://github.com/miikka/python-gorilla

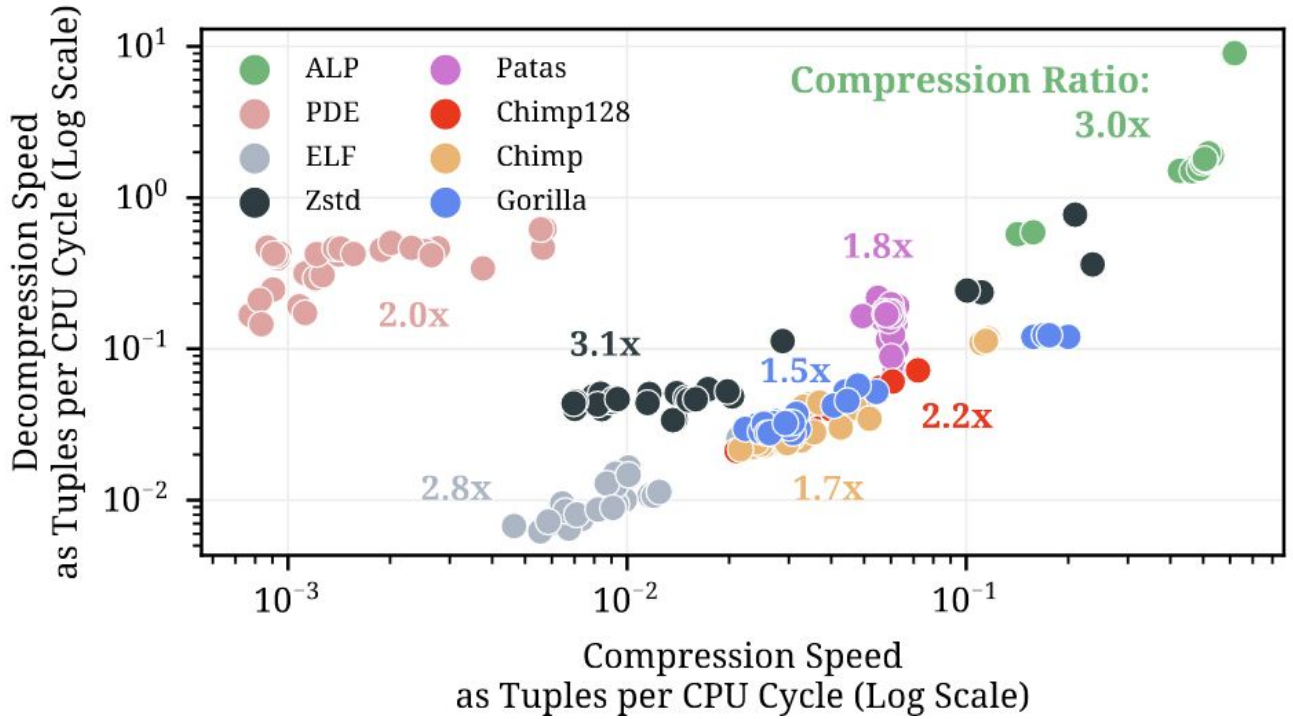
Part 4:
Is it any good?

Benchmark results



Source: Liakos, Papakonstantinopoulou, and Kotidis. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. PVLDB, 15(11): 3058 - 3070, 2022.

Benchmark results



Source: Afroozeh, Kuffó, and Boncz. 2024. ALP: Adaptive Lossless floating-Point Compression. In Proceedings of The 2024 International Conference on Management of Data (SIGMOD '24).

You can compress any data with Gorilla!

The algorithm itself does not require the data to be valid floats. It just uses tricks that happen to work well on floats. Might not work the best on everything!

In conclusion

- In time series data, the data points are typically similar to each other
- If you xor two similar floats, the 1 bits are typically in the middle
- Gorilla was nice back in the day but now you should use ALP

Code is at: https://github.com/miikka/python-gorilla

Thank you!

psst. read my blog at https://quanttype.net/ / follow me on LinkedIn / Mastodon