

---

## The FastPFOR C++ library : Fast integer compression



### What is this?

A research library with integer compression schemes. It is broadly applicable to the compression of arrays of 32-bit integers where most integers are small. The library seeks to exploit SIMD instructions (SSE) whenever possible.

This library can decode at least 4 billions of compressed integers per second on most desktop or laptop processors. That is, it can decompress data at a rate of 15 GB/s. This is significantly faster than generic codecs like gzip, LZO, Snappy or LZ4.

It is used by the zsearch engine as well as in GMAP and GSNAP. DuckDB derived some of their code from this library. It has been ported to Java, C# and Go. The Java port is used by ClueWeb Tools.

Apache Lucene version 4.6.x uses a compression format derived from our FastPFOR scheme.

### Python bindings

- We have Python bindings: <https://github.com/searchivarius/PyFastPFor>

### Myths

Myth: SIMD compression requires very large blocks of integers (1024 or more).

Fact: This is not true. Our fastest scheme (SIMDBinaryPacking) works over blocks of 128 integers. Another very fast scheme (Stream VByte) works over blocks of four integers.

Myth: SIMD compression means high speed but less compression.

Fact: This is wrong. Some schemes cannot easily be accelerated with SIMD instructions, but many that do compress very well.

### Working with sorted lists of integers

If you are working primarily with sorted lists of integers, then you might want to use differential coding. That is you may want to compress the deltas instead of the integers themselves. The current

---

library (fastpfor) is generic and was not optimized for this purpose. However, we have another library designed to compress sorted integer lists:

<https://github.com/lemire/SIMDCompressionAndIntersection>

This other library (SIMDCompressionAndIntersection) also comes complete with new SIMD-based intersection algorithms.

There is also a C library for differential coding (fast computation of deltas, and recovery from deltas):

<https://github.com/lemire/FastDifferentialCoding>

## Other recommended libraries

- Fast integer compression in Go: <https://github.com/ronanh/intcomp>
- High-performance dictionary coding <https://github.com/lemire/dictionary>
- LittleIntPacker: C library to pack and unpack short arrays of integers as fast as possible <https://github.com/lemire/LittleIntPacker>
- The SIMDComp library: A simple C library for compressing lists of integers using binary packing <https://github.com/lemire/simdcomp>
- StreamVByte: Fast integer compression in C using the StreamVByte codec <https://github.com/lemire/streamvby>
- MaskedVByte: Fast decoder for VByte-compressed integers <https://github.com/lemire/MaskedVByte>
- CSharpFastPFOR: A C# integer compression library <https://github.com/Genbox/CSharpFastPFOR>
- JavaFastPFOR: A java integer compression library <https://github.com/lemire/JavaFastPFOR>
- Encoding: Integer Compression Libraries for Go <https://github.com/zhenjl/encoding>
- FrameOfReference is a C++ library dedicated to frame-of-reference (FOR) compression: <https://github.com/lemire/FrameOfReference>
- libvbyte: A fast implementation for varbyte 32bit/64bit integer compression <https://github.com/cruppstahl/libv>
- TurboPFor is a C library that offers lots of interesting optimizations. Well worth checking! (GPL license) <https://github.com/powturbo/TurboPFor-Integer-Compression>
- Oroch is a C++ library that offers a usable API (MIT license) <https://github.com/ademakov/Oroch>

## Reference and documentation

For a simple example, please see

`example.cpp`

in the root directory of this project.

Please see:

- 
- Daniel Lemire, Nathan Kurz, Christoph Rupp, Stream VByte: Faster Byte-Oriented Integer Compression, Information Processing Letters 130, 2018. <https://arxiv.org/abs/1709.08990>
  - Daniel Lemire and Leonid Boytsov, Decoding billions of integers per second through vectorization, Software Practice & Experience 45 (1), 2015. <http://arxiv.org/abs/1209.2137>  
<http://onlinelibrary.wiley.com/doi/10.1002/spe.2203/abstract>
  - Daniel Lemire, Leonid Boytsov, Nathan Kurz, SIMD Compression and the Intersection of Sorted Integers, Software Practice & Experience 46 (6), 2016 <http://arxiv.org/abs/1401.6399>
  - Jeff Plaisance, Nathan Kurz, Daniel Lemire, Vectorized VByte Decoding, International Symposium on Web Algorithms 2015, 2015. <http://arxiv.org/abs/1503.07387>
  - Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, Ji-Rong Wen, A General SIMD-based Approach to Accelerating Compression Algorithms, ACM Transactions on Information Systems 33 (3), 2015. <http://arxiv.org/abs/1502.01916>

This library was used by several papers including the following:

- Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, Steven Swanson, An Experimental Study of Bitmap Compression vs. Inverted List Compression, SIGMOD 2017 <http://db.ucsd.edu/wp-content/uploads/2017/03/sidm338-wangA.pdf>
- P. Damme, D. Habich, J. Hildebrandt, W. Lehner, Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses), EDBT 2017 <http://openproceedings.org/2017/conf/edbt/paper-146.pdf>
- P. Damme, D. Habich, J. Hildebrandt, W. Lehner, Insights into the Comparative Evaluation of Lightweight Data Compression Algorithms, EDBT 2017 <http://openproceedings.org/2017/conf/edbt/paper-414.pdf>
- G. Ottaviano, R. Venturini, Partitioned Elias-Fano Indexes, ACM SIGIR 2014 <http://www.di.unipi.it/~ottavian/files>
- M. Petri, A. Moffat, J. S. Culpepper, Score-Safe Term Dependency Processing With Hybrid Indexes, ACM SIGIR 2014 <http://www.culpepper.io/publications/sp074-petri.pdf>

It has also inspired related work such as...

- T. D. Wu, Bitpacking techniques for indexing genomes: I. Hash tables, Algorithms for Molecular Biology 11 (5), 2016. <http://almob.biomedcentral.com/articles/10.1186/s13015-016-0069-5>

## License

This code is licensed under Apache License, Version 2.0 (ASL2.0).

## Software Requirements

This code requires a compiler supporting C++11. This was a design decision.

---

It builds under

- clang++ 3.2 (LLVM 3.2) or better,
- Intel icpc (ICC) 13.0.1 or better,
- MinGW32 (x64-4.8.1-posix-seh-rev5)
- Microsoft VS 2012 or better,
- and GNU GCC 4.7 or better.

The code was tested under Windows, Linux and MacOS.

## Hardware Requirements

We require an x64 platform.

To fully use the library, your processor should support SSSE3. This includes almost every Intel or AMD processor sold after 2006. (Note: the key schemes require merely SSE2.)

Some specific binaries will only run if your processor supports SSE4.1. They have been purely used for specific tests however.

## Building with CMake

You need cmake. On most linux distributions, you can simply do the following:

```
1  git clone https://github.com/lemire/FastPFor.git
2  cd FastPFor
3  mkdir build
4  cd build
5  cmake ..
6  cmake --build .
```

It may be necessary to set the CXX variable. The project is installable (`make install` works).

To create project files for Microsoft Visual Studio, it might be useful to target 64-bit Windows (e.g., see <http://www.cmake.org/cmake/help/v3.0/generator/Visual%20Studio%2012%202013.html>).

## Multithreaded context

You should not assume that our objects are thread safe. If you have several threads, each thread should have its own IntegerCODEC objects to ensure that there is no concurrency problems.

---

## Why C++11?

With minor changes, all schemes will compile fine under compilers that do not support C++11. And porting the code to C should not be a challenge.

In any case, we already support 3 major C++ compilers so portability is not a major issue.

## What if I prefer Java?

Many schemes cannot be efficiently ported to Java. However some have been. Please see:

<https://github.com/lemire/JavaFastPFOR>

## What if I prefer C#?

See CSharpFastPFOR: A C# integer compression library <https://github.com/Genbox/CSharpFastPFOR>

## What if I prefer Go?

See Encoding: Integer Compression Libraries for Go <https://github.com/zhenjl/encoding>

## Testing

If you used CMake to generate the build files, the `check` target will run the unit tests. For example , if you generated Unix Makefiles

```
1 make check
```

will do it.

## Simple benchmark

```
1 make codecs
2 ./codecs --clusterdynamic
3 ./codecs --uniformdynamic
```

---

## Optional : Snappy

Typing “make allallall” will install some testing binaries that depend on Google Snappy. If you want to build these, you need to install Google snappy. You can do so on a recent ubuntu machine as:

```
1 sudo apt-get install libsnappy-dev
```

## Processing data files

Typing “make” will generate an “inmemorybenchmark” executable that can process data files.

You can use it to process arrays on (sorted!) integers on disk using the following 32-bit format: 1 unsigned 32-bit integer indicating array length followed by the corresponding number of 32-bit integer. Repeat.

( It is assumed that the integers are sorted.)

Once you have such a binary file somefilename you can process it with our inmemorybenchmark:

```
1 ./inmemorybenchmark --minlength 10000 somefilename
```

The “minlength” flag skips short arrays. (Warning: timings over short arrays are unreliable.)

## Testing with the Gov2 and ClueWeb09 data sets

As of April 2014, we recommend getting our archive at

<http://lemire.me/data/integercompression2014.html>

It is the data was used for the following paper:

Daniel Lemire, Leonid Boytsov, Nathan Kurz, SIMD Compression and the Intersection of Sorted Integers, arXiv: 1401.6399, 2014 <http://arxiv.org/abs/1401.6399>

## I used your code and I get segmentation faults

Our code is thoroughly tested.

One common issue is that people do not provide large enough buffers. Some schemes can have such small compression rates that the compressed data generated will be much larger than the input data.

---

### **Is any of this code subject to patents?**

I (D. Lemire) did not patent anything.

However, we implemented varint-G8UI which was patented by its authors. DO NOT use varint-G8UI if you want to avoid patents.

The rest of the library *should be* patent-free.

### **Funding**

This work was supported by NSERC grant number 26143.