Zstandard, or `zstd` as short version, is a fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios. It's backed by a very fast entropy stage, provided by Huff0 and FSE library.

Zstandard's format is stable and documented in RFC8878. Multiple independent implementations are already available. This repository represents the reference implementation, provided as an open-source dual BSD OR GPLv2 licensed **C** library, and a command line utility producing and decoding `.zst`, `.gz`, `.xz` and `.lz4` files. Should your project require another programming language, a list of known ports and bindings is provided on Zstandard homepage.

Development branch status:



Build status Build status



Benchmarks

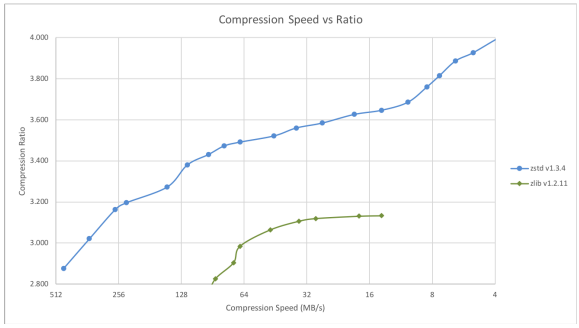
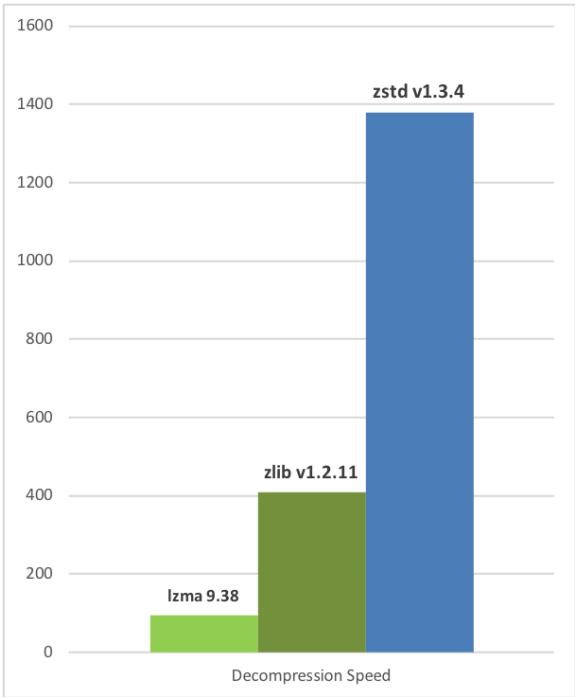
For reference, several fast compression algorithms were tested and compared on a desktop featuring a Core i7-9700K CPU @ 4.9GHz and running Ubuntu 24.04 (`Linux 6.8.0-53-generic`), using lzbench, an open-source in-memory benchmark by @inikep compiled with gcc 14.2.0, on the Silesia compression corpus.

Compressor name	Ratio	Compression	Decompress.
zstd 1.5.7 -1	2.896	510 MB/s	1550 MB/s
brotli 1.1.0 -1	2.883	290 MB/s	425 MB/s
zlib 1.3.1 -1	2.743	105 MB/s	390 MB/s
zstd 1.5.7 --fast=1	2.439	545 MB/s	1850 MB/s
quicklz 1.5.0 -1	2.238	520 MB/s	750 MB/s
zstd 1.5.7 --fast=4	2.146	665 MB/s	2050 MB/s
lzo1x 2.10 -1	2.106	650 MB/s	780 MB/s
lz4 1.10.0	2.101	675 MB/s	3850 MB/s
snappy 1.2.1	2.089	520 MB/s	1500 MB/s
lzf 3.6 -1	2.077	410 MB/s	820 MB/s

The negative compression levels, specified with `--fast=#`, offer faster compression and decompression speed at the cost of compression ratio.

Zstd can also offer stronger compression ratios at the cost of compression speed. Speed vs Compression trade-off is configurable by small increments. Decompression speed is preserved and remains roughly the same at all settings, a property shared by most LZ compression algorithms, such as zlib or lzma.

The following tests were run on a server running Linux Debian (`Linux version 4.14.0-3-amd64`) with a Core i7-6700K CPU @ 4.0GHz, using lzbench, an open-source in-memory benchmark by @inikep compiled with gcc 7.3.0, on the Silesia compression corpus.

Compression Speed vs Ratio	Decompression Speed
	

A few other algorithms can produce higher compression ratios at slower speeds, falling outside of the graph. For a larger picture including slow modes, click on this link.

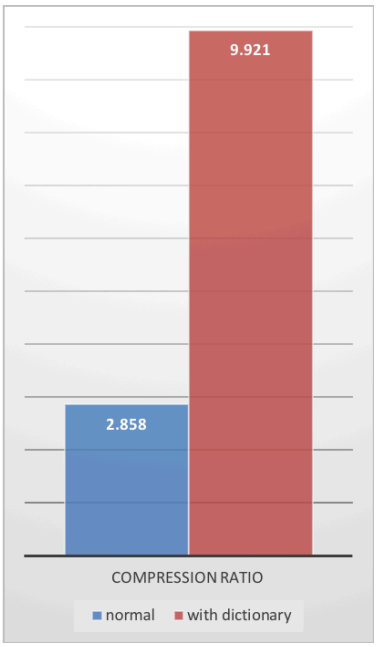
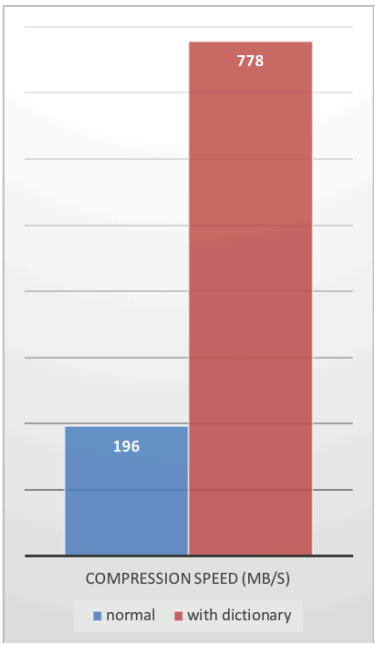
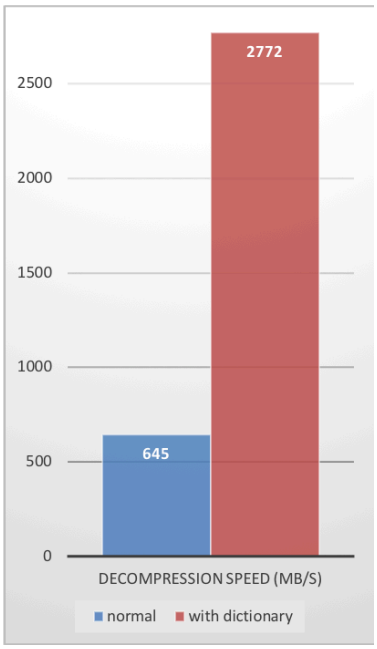
The case for Small Data compression

Previous charts provide results applicable to typical file and stream scenarios (several MB). Small data comes with different perspectives.

The smaller the amount of data to compress, the more difficult it is to compress. This problem is common to all compression algorithms, and reason is, compression algorithms learn from past data how to compress future data. But at the beginning of a new data set, there is no “past” to build upon.

To solve this situation, Zstd offers a **training mode**, which can be used to tune the algorithm for a selected type of data. Training Zstandard is achieved by providing it with a few samples (one file per sample). The result of this training is stored in a file called “dictionary”, which must be loaded before compression and decompression. Using this dictionary, the compression ratio achievable on small data improves dramatically.

The following example uses the `github-users` sample set, created from github public API. It consists of roughly 10K records weighing about 1KB each.

Compression Ratio	Compression Speed	Decompression Speed
		

These compression gains are achieved while simultaneously providing *faster* compression and decompression speeds.

Training works if there is some correlation in a family of small data samples. The more data-specific a dictionary is, the more efficient it is (there is no *universal dictionary*). Hence, deploying one dictionary per type of data will provide the greatest benefits. Dictionary gains are mostly effective in the first few KB. Then, the compression algorithm will gradually use previously decoded content to better compress the rest of the file.

Dictionary compression How To:

1. Create the dictionary

   ```
   zstd --train FullPathToTrainingSet/* -o dictionaryName
   ```

2. Compress with dictionary

   ```
   zstd -D dictionaryName FILE
   ```

3. Decompress with dictionary

   ```
   zstd -D dictionaryName --decompress FILE.zst
   ```

Build instructions

`make` is the main build system of this project. It is the reference, and other build systems are periodically updated to stay compatible. However, small drifts and feature differences can be present, since perfect synchronization is difficult. For this reason, when your build system allows it, prefer employing `make`.

Makefile

Assuming your system supports standard `make` (or `gmake`), just invoking `make` in root directory generates `zstd` cli at root, and also generates `libzstd` into `lib/`.

Other standard targets include:

- `make install` : install zstd cli, library and man pages

- `make check` : run `zstd`, test its essential behavior on local platform

The `Makefile` follows the GNU Standard Makefile conventions, allowing staged install, standard compilation flags, directory variables and command variables.

For advanced use cases, specialized flags which control binary generation and installation paths are documented in `lib/README.md` for the `libzstd` library and in `programs/README.md` for the `zstd` CLI.

cmake
A `cmake` project generator is available for generating Makefiles or other build scripts to create the `zstd` binary as well as `libzstd` dynamic and static libraries. The repository root now contains a minimal `CMakeLists.txt` that forwards to `build/cmake`, so you can configure the project with a standard `cmake -S .` invocation, while the historical `cmake -S build/cmake` entry point remains fully supported.

```
cmake -S . -B build-cmake
cmake --build build-cmake
```

By default, `CMAKE_BUILD_TYPE` is set to `Release`.

Support for Fat (Universal2) Output
`zstd` can be built and installed with support for both Apple Silicon (M1/M2) as well as Intel by using CMake's Universal2 support. To perform a Fat/Universal2 build and install use the following commands:

```
cmake -S . -B build-cmake-debug -G Ninja -DCMAKE_OSX_ARCHITECTURES="x86_64;x86_64h;arm64"
cd build-cmake-debug
ninja
sudo ninja install
```

Meson
A Meson project is provided within `build/meson`. Follow build instructions in that directory.

You can also take a look at `.travis.yml` file for an example about how Meson is used to build this project.

Note that default build type is **release**.

VCPKG
You can build and install zstd vcpkg dependency manager:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
./vcpkg install zstd
```

The zstd port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please create an issue or pull request on the vcpkg repository.

Conan
You can install pre-built binaries for zstd or build it from source using Conan. Use the following command:

```
conan install --requires="zstd/[*]" --build=missing
```

The zstd Conan recipe is kept up to date by Conan maintainers and community contributors. If the version is out of date, please create an issue or pull request on the ConanCenterIndex repository.

Visual Studio (Windows)
Going into `build` directory, you will find additional possibilities:

- Projects for Visual Studio 2008 and 2010.
  - VS2010 project is compatible with VS2012, VS2013, VS2015 and VS2017.
- Automated build scripts for Visual compiler by @KrzysFR, in `build/VS_scripts`, which will build `zstd` cli and `libzstd` library without any need to open Visual Studio solution.
- It is now recommended to generate Visual Studio solutions from `cmake`

Buck
You can build the zstd binary via buck by executing: `buck build programs:zstd` from the root of the repo. The output binary will be in `buck-out/gen/programs/`.

Bazel
You can integrate zstd into your Bazel project by using the module hosted on the Bazel Central Repository.

Testing

You can run quick local smoke tests by running `make check`. If you can't use `make`, execute the `playTest.sh` script from the `src/tests` directory. Two env variables `$ZSTD_BIN` and `$DATAGEN_BIN` are needed for the test script to locate the `zstd` and `datagen` binary. For information on CI testing, please refer to `TESTING.md`.

Status

Zstandard is deployed within Meta and many other large cloud infrastructures, to compress humongous amounts of data in various formats and use cases. It is also continuously fuzzed for security issues by Google's oss-fuzz program.

License

Zstandard is dual-licensed under BSD OR GPLv2.

Contributing

The `dev` branch is the one where all contributions are merged before reaching `release`. Direct commit to `release` are not permitted. For more information, please read CONTRIBUTING.