A GPU Accelerated Error-Bounded Lossy Compressor

cuSZ

development group

Jiannan Tian, Dingwen Tao, Sheng Di, Franck Cappello

contributors (alphabetic)

Cody Rivera, Jon Calhoun, Kai Zhao, Megan Hickman, Robert Underwood, Xin Liang

created on May 7, 2020, last compiled on May 24, 2020

Contents

1 user guide

1.1 introduction

1.1.1 workflow

1.1.2 about compression ratio

1.2 set up

1.2.1 requirements

- NVIDIA GPU with Kepler, Maxwell, Pascal, Volta, or Turing microarchitectures
- CUDA 9.1+ and GCC 7+ (recommended: CUDA 10.1 + GCC 8)
- CMake 3.11+

1.2.2 download

```
git clone git@github.com:hipdac-lab/cuSZ.git
```

1.2.3 compile

```
cd cuSZ
cmake CMakeLists.txt      # using cmake to compile cuSZ for {1,2,3}-D, with Huffman codec
make
```

1.3 run

Commands cusz or cusz -h are for instant instructions.

1.3.1 cuSZ as a compressor

basic use

The basic use cuSZ is given below.

```
./cusz -f32 -m r2r -e 1.23e-4.56 -i ./data/sample-cesm-CLDHGH -D cesm -z -x
        ^   ~~~~~~  ~~~~~~~~~~~~~~  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~  ~~~~~~~  ^   ^
        |    mode    error bound          input datum file           demo    |   |
       dtype                                                         datum  zip unzip
```

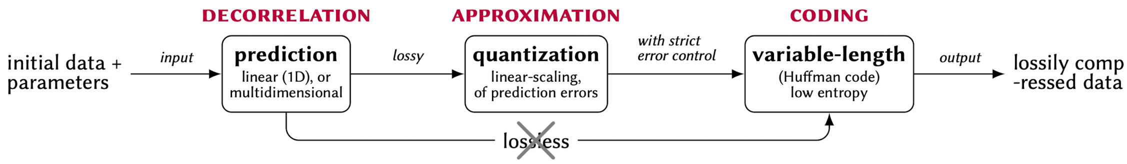

Figure 1.1: Error-control worflow.

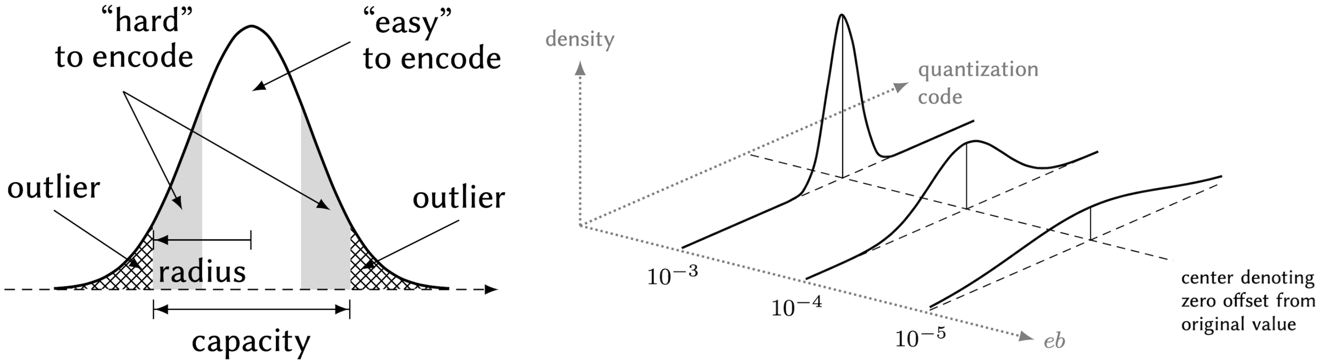

Figure 1.2: (left) classification in prediction error distribution, (right) smaller error bound linear-scales the prediction error distribution.

`-D cesm` specifies preset dataset for demonstration. In this case, it is CESM-ATM, whose dimension is 1800-by-3600, following y-x order. To otherwise specify datum file and input dimensions arbitrarily, we use `-2 3600 1800`, then it becomes

```
./cusz -f32 -m r2r -e 1.23e-4.56 -i ./data/sample-cesm-CLDHGH -2 3600 1800 -z -x
```

To conduct compression, several input arguments are **necessary**,

- `-z` or `--zip` to compress
- `-x` or `--unzip` to decompress
- `-m` or `--mode` to speciry compression mode. Options include `abs` (absolute value) and `r2r` (relative to value range).
- `-e` or `--eb` to specify error bound
- `-i` to specify input datum file
- `-D` to specify demo dataset name or `-{1,2,3}` to input dimensions

tuning

There are also internal a) quant. code representation, b) Huffman codeword representation, and c) chunk size for Huffman coding exposed. Each can be specified with argument options.

- `-Q` or `--quant-rep` or `--bcode-bitwidth <8|16|32>` to specify bincode/quant. code representation. Options 8, 16, 32 are for `uint8_t`, `uint16_t`, `uint32_t`, respectively. (Manually specifying this may not result in optimal memory footprint.)
- `-H` or `--huffman-rep` or `--hcode-bitwidth <32|64>` to specify Huffman codeword representation. Options 32, 64 are for `uint32_t`, `uint64_t`, respectively. (Manually specifying this may not result in optimal memory footprint.)
- `-C` or `--huffman-chunk` or `--hcode-chunk [256|512|1024|...]` to specify chunk size for Huffman codec. Should be a power-of-2 that is sufficiently large. (This affects Huffman decoding performance *significantly*.)

extension and use scenarios

preprocess Some application such as EXAFEL preprocesses with binning [1] in addition to skipping Huffman codec.

disabling modules Also according to EXAFEL, given binning and `uint8_t` have already result in a compression ratio of up to 16, Huffman codec may not be expected in a real-world use scenario. In such cirumstances, `--skip huffman` can be used.

Other module skipping for use scenarios are in development.

1.3.2 cuSZ as an analytical tool

`--dry-run` or `-r` in place of `-a` and/or `-x` enables dry-run mode to get PSNR. This employs the feature of dual-quantization that the decompressed data is guaranteed the same with prequantized data.

[1] A current binning setting is to downsample a 4-by-4 cell to 1 point.

1.3.3 example

1. run a 2D CESM demo at 1e-4 relative to value range

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-cesm-CLDHGH -D cesm -z -x
```

2. alternatively, to use full option name,

```
./cusz -f32 --mode r2r --eb 1e-4 --input ./data/sample-cesm-CLDHGH \
    --demo cesm --zip --unzip
```

3. run a 3D Hurricane Isabel demo at 1e-4 relative to value range

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-hurr-CLOUDf48 -D huricanne -z -x
```

4. run CESM demo with 1) `uint8_t`, 2) 256 quant. bins,

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-cesm-CLDHGH -D cesm -z -x \
    -d 256 -Q 8
```

5. in addition to the previous command, if skipping Huffman codec,

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-cesm-CLDHGH -D cesm -z -x \
    -d 256 -Q 8 --skip huffman  # or ‘-X/-S huffman‘
```

6. some application such as EXAFEL preprocesses with binning [2] in addition to skipping Huffman codec

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-cesm-CLDHGH -D cesm -z -x \
    -d 256 -Q 8 --pre binning --skip huffman    # or ‘-p binning‘
```

7. dry-run to get PSNR and to skip real compression or decompression; `-r` also works alternatively to `--dry-run`

```
./cusz -f32 -m r2r -e 1e-4 -i ./data/sample-cesm-CLDHGH -D cesm --dry-run   # or ‘-r‘
```

1.4 project management

1.4.1 TODO List

Please refere to *Project Management page*.

1.4.2 `changelog`

May, 2020

perf [need review] decrease memory footprint by merging `data` and `ourlier` during compression
feature add `--skip huffman` and `--verify huffman` options
feature add binning as preprocessing, `--pre binning` or `-p binning`
prototype use `cuSparse` to transform `outlier` to dense format
feature add `argparse` to check and parse argument inputs
refactor add CUDA wrappers (e.g., `mem::CreateCUDASpace`)

April, 2020

feature add concise and detailed help doc
deploy `sm_61` (e.g., P1000) and `sm_70` (e.g., V100) binary
feature add dry-run mode
refactor merge cuSZ and Huffman codec in driver program
perf 1D `PdQ` (and reverse `PdQ`) `blockDim` set to 32, throughput changed from 2.7 GBps to 16.8 GBps
deploy histograming, 2013 algorithm supersedes naive 2007 algorithm by default
feature add communication of equivalance calculation
feature use cooperative groups (CUDA 9 required) for canonical Huffman codebook
perf faster initializing shared memory for PdQ, from 150 GBps to 200 GBps

[2] A current binning setting is to downsample a 4-by-4 cell to 1 point.

feature add Huffman inflating/decoding
refactor merge {1,2,3}-D cuSZ
feature set 32- and 64-bit as internal Huffman codeword representation
feature now use arbitrary multiple-of-8-bit for quant. code
feature switch to canonical Huffman code for decoding

March, 2020

perf tuning thread number for Huffman deflating and inflating
feature change freely to 32bit intermediate Huffman code representation
demo add EXAFEL demo
feature switch to faster histograming

February, 2020

demo SDRB suite metadata in `SDRB.hh`
feature visualize histogram (`pSZ`)
prototype add CPU pSZ, p for prototyping
milestone `PdQ` for compression, Huffman encoding and deflating

1.5 reference

[1] Gómez-Luna, Juan, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. "An optimized approach to histogram computation on GPU." Machine Vision and Applications 24, no. 5 (2013): 899-908.
[2] Barnett, Mark L. "Canonical Huffman encoded data decompression algorithm." U.S. Patent 6,657,569, issued December 2, 2003.

1.6 acknowledgement

This R&D was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem. This repository was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and also supported by the National Science Foundation under Grant No. 1948447.

1.7 license

Contact: Dingwen Tao (dingwen.tao@ieee.org), Jiannan Tian(tian.jn09@gmail.com)

2 development

2.1 parallelism

2.2 API (draft)

Structure

```
cuSZ driver program
│
├── DryRun
│   ├── c_lorenzo_1d1l
│   ├── c_lorenzo_2d1l
│   └── c_lorenzo_3d1l
│
├── Dual-Quant
│   ├── c_lorenzo_1d1l
│   ├── c_lorenzo_2d1l
│   └── c_lorenzo_3d1l
│
├── reversed Dual-Quant
│   ├── x_lorenzo_1d1l
│   ├── x_lorenzo_2d1l
│   └── x_lorenzo_3d1l
│
...
```

```
...
│
├── HuffmanEncode
│   ├── 0.GetFrequency
│   ├── 1.InitHuffTreeAndGetCodebook
│   ├── 2.GetCanonicalCode
│   ├── 3.EncodeFixedLen
│   └── 4.Deflate
│
├── HuffmanDecode
│   └── Decode
│       └── InflateChunkwise
│
└── verification
```

2.2.1 `Dual-Quant` and reversed `Dual-Quant`

2.2.2 histogramming

2.2.3 Huffman codec