Embedding Compression via Spherical Coordinates

Han Xiao
Jina AI *by* Elastic
han.xiao@jina.ai

Abstract

We present a compression method for unit-norm embeddings that achieves 1.5× compression, 25% better than the best prior lossless method. The method exploits that spherical coordinates of high-dimensional unit vectors concentrate around $\pi/2$, causing IEEE 754 exponents to collapse to a single value and high-order mantissa bits to become predictable, enabling entropy coding of both. Reconstruction error is below 1e-7, under float32 machine epsilon. Evaluation across 26 configurations spanning text, image, and multi-vector embeddings confirms consistent improvement. The method requires no training and is available at https://github.com/jina-ai/jzip-compressor.

1 Introduction

Embedding vectors are fundamental to modern AI systems, powering RAG pipelines, agentic search, and multimodal retrieval. A typical embedding model produces 1024-dimensional float32 vectors, requiring 4 KB per embedding. At scale, a database of 100 million embeddings requires 400 GB. The problem is more severe for multi-vector representations, where late-interaction models like ColBERT [9] produce one embedding per token, multiplying storage by approximately 100×. While lossy quantization achieves high compression ratios, many applications benefit from high-fidelity reconstruction for embedding caches, API serialization, network transmission, and archival storage. The state-of-the-art lossless approach transposes the embedding matrix, byte-shuffles to group exponent bytes, and applies entropy coding [7, 1], but achieves only 1.2× compression because float32 mantissa bits have near-maximum entropy.

Because cosine similarity is the standard retrieval metric, most embedding models produce *unit-norm* vectors with $\|\mathbf{x}\|_2 = 1$. This constraint places embeddings on the surface of a high-dimensional hypersphere S^{d-1}, yet existing lossless methods ignore this geometric structure, while prior work using spherical coordinates has focused exclusively on lossy quantization [15, 6, 18]. Unit-norm vectors can be equivalently represented using $d - 1$ angular coordinates. The Cartesian form has values spanning ± 0.001 to ± 0.3, requiring many different IEEE 754 exponents. The angular form concentrates around $\pi/2 \approx 1.57$ [3], collapsing exponents to a single value and making high-order mantissa bits predictable. This combination of spherical transformation with entropy coding has not been explored.

Contribution. We present a compression method that converts Cartesian to spherical coordinates before byte shuffling and entropy coding, achieving 1.5× compression across 26 embedding configurations. The spherical transform introduces bounded reconstruction error below 1e-7, under float32 machine epsilon, which preserves retrieval quality perfectly. For a ColBERT index of 1 million documents, this reduces storage from 240 GB to 160 GB. The method requires no training and applies to text, image, and multi-vector embeddings.

Preprint.

2 Related Work

2.1 Lossless Floating-Point Compression

IEEE 754 Float32. A float32 number consists of 1 sign bit, 8 exponent bits, and 23 mantissa bits, encoding the value $x = (-1)^s \times 2^{e-127} \times (1 + m/2^{23})$. The exponent determines magnitude, with values near 1.0 having exponent ≈ 127 and values near 0.01 having exponent ≈ 120. The mantissa encodes precision within that magnitude range. For lossless compression, the exponent byte represents 25% of each float32 value while the mantissa represents the remaining 75%.

The HPC community developed byte shuffling [1], which reorders array bytes to group all exponent bytes together, improving entropy coding because bytes at the same position across different floats often share similar values. ZipNN [7] applies this technique to neural network weights, achieving 33% compression on BF16 by separating exponent and mantissa bytes before applying zstd [5]. FCBench [4] benchmarks lossless floating-point compression methods across scientific domains. For float32 embeddings, the mantissa comprises 3/4 of the data with near-maximum entropy of $\sim$7.3 bits/byte. Even with perfect exponent compression, the theoretical limit is Ratio $\leq 4/(0.25 \cdot 0 + 0.75 \cdot 1) = 1.33\times$. Current methods achieve $\sim$1.25$\times$, approaching this bound. Our method exceeds this limit by also reducing mantissa entropy through value concentration.

Recent work has identified that trained neural network *weights* exhibit natural exponent concentration due to heavy-tailed dynamics of stochastic gradient descent. ECF8 [17] shows that model weights follow α-stable distributions, leading to exponent entropy of 2 to 3 bits, and achieves up to 26.9% lossless compression on FP8 model weights. DFloat11 [19] exploits similar properties for BF16 weights. These methods target model parameters, where exponent concentration arises naturally from training dynamics.

2.2 Spherical and Polar Coordinate Methods

Spherical Coordinates. A d-dimensional vector $\mathbf{x}$ can be represented using spherical coordinates consisting of a radius $r = \|\mathbf{x}\|_2$ and $d-1$ angles $\theta_1, \ldots, \theta_{d-1}$. The first $d-2$ angles are computed as $\theta_i = \arccos\left(x_i / \sqrt{\sum_{j=i}^{d} x_j^2}\right)$ for $i = 1, \ldots, d-2$, each lying in $[0, \pi]$. The final angle uses $\arctan 2$ to preserve quadrant information, with $\theta_{d-1} = \arctan 2(x_d, x_{d-1}) \in [-\pi, \pi]$. For unit-norm vectors where $r = 1$, the radius can be omitted, leaving $d-1$ angles to represent d Cartesian coordinates.

Several recent works employ polar or spherical coordinates for compression, but all use lossy quantization rather than lossless encoding. Trojak and Witherden [15] use spherical polar coordinates for lossy compression of 3D vectors in computational physics, achieving 1.5$\times$ compression by quantizing angles to fixed bit-widths. This method is limited to 3D and employs deliberate precision loss. PCDVQ [18] uses polar coordinate decoupling for vector quantization of LLM weights, separately clustering direction and magnitude with codebooks to achieve 2-bit quantization. PolarQuant [6] transforms KV cache embeddings to polar coordinates and quantizes the resulting angles, exploiting that angles after random preconditioning have bounded distributions. Both PCDVQ and PolarQuant target lossy compression of model internals such as weights and KV caches, not lossless compression of embedding outputs.

2.3 Lossy Vector Compression

Product quantization [8] achieves 4 to 32$\times$ compression by partitioning vectors into subspaces and quantizing independently. Binary and scalar quantization [14] offer simpler alternatives, while learned codebooks [16] push compression further. These lossy methods achieve higher ratios than lossless approaches but introduce reconstruction error. Our work targets applications requiring high-fidelity reconstruction with bounded error.

3 Method

Figure 1 and Algorithm 1 present the compression pipeline: convert to spherical coordinates, transpose to group same-angle values, byte shuffle to separate exponents, and compress with zstd. Decompres-

Table 1: Comparison with related floating-point compression methods.

Method	Domain	Lossless?	Polar?	Unit-norm?	Mechanism
ECF8 [17]	LLM weights	Exact (FP8)	No	No	Natural concentration
DFloat11 [19]	LLM weights	Exact (BF16)	No	No	Natural concentration
EFloat [2]	Embeddings	Lossy	No	No	Variable-length encoding
PolarQuant [6]	KV cache	Lossy	Yes	No	Angle quantization
PCDVQ [18]	LLM weights	Lossy	Yes	No	Codebook quantization
Trojak [15]	3D physics	Lossy	Yes	No	Angle quantization
Ours	Embeddings	**ϵ-bounded**	**Yes**	**Yes**	Geometric transform

sion reverses these steps. The spherical transform is mathematically invertible, but floating-point transcendental functions introduce bounded reconstruction error. Using double precision for intermediate calculations keeps this error below 1e-7, under float32 machine epsilon of 1.19e-7. This preserves cosine similarity to 1e-7 and does not affect retrieval quality as shown in Table 2. Implementation is in Appendix A.

The spherical transform provides compression by concentrating IEEE 754 exponents and high-order mantissa bits. Unit-norm vectors in $\mathbb{R}^d$ lie on the hypersphere S^{d-1}, so $d-1$ angles suffice instead of d Cartesian coordinates. Cartesian coordinates of unit-norm embeddings scale as $1/\sqrt{d}$, spanning values in $[0.001, 0.3]$ for typical dimensions and requiring 22 to 40 different exponents. The first $d-2$ spherical angles, by contrast, are bounded to $[0, \pi]$ and concentrate around $\pi/2 \approx 1.57$ in high dimensions [3]; Figure 2 shows this empirically for `jina-embeddings-v4` embeddings. This concentration collapses exponents to a single dominant value of 127, reducing exponent entropy from 2.6 to 0.03 bits/byte for `jina-embeddings-v4`, with similar patterns across models validated in Appendix D.

Exponent compression alone would yield only $\sim$1.1$\times$ in practice (the 1.33$\times$ theoretical limit assumes zero exponent bits, but entropy coding has overhead and exponents retain $\sim$0.03 bits). The additional gain comes from the high-order mantissa byte: when angles cluster around $\pi/2 \approx 1.5708$, the IEEE 754 mantissa bits encoding the fractional part also become predictable. Empirically, the high-order mantissa byte entropy drops from 8.0 to 4.5 bits, contributing $\sim$11% additional savings beyond exponents. Together, exponent and mantissa concentration yield the observed 1.5$\times$ compression.

ECF8 [17] and DFloat11 [19] exploit *natural* exponent concentration in model weights arising from training dynamics, whereas our method *creates* exponent concentration through a deterministic geometric transformation, as summarized in Table 1.

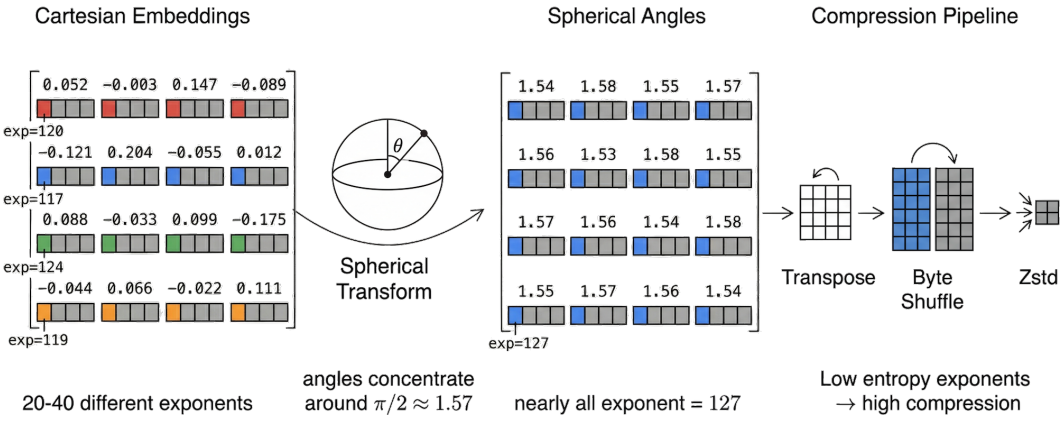

Figure 1: Compression pipeline. Cartesian coordinates span diverse magnitudes with 20 to 40 different exponents, shown in varied colors. The spherical transform produces angles concentrated around $\pi/2 \approx 1.57$, collapsing nearly all exponents to 127, shown in uniform color. Transpose groups same-position angles across vectors, byte shuffle separates exponent bytes, and zstd compresses the low-entropy exponent stream.

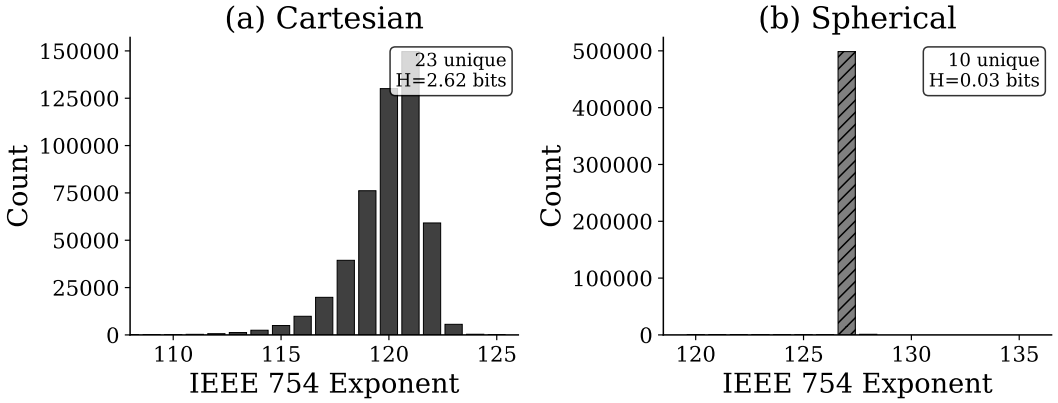

Figure 2: IEEE 754 exponent distribution for `jina-embeddings-v4` (2048d). (a) Cartesian coordinates span 23 exponent values; (b) spherical angles concentrate around exponent 127 with 99.7% frequency.

Algorithm 1 Spherical Embedding Compression

Require: Embedding matrix $\mathbf{E} \in \mathbb{R}^{n \times d}$ (float32, unit-norm)

$\mathbf{\Theta} \leftarrow \text{CartesianToSpherical}(\mathbf{E})$ *// $\mathbf{\Theta} \in \mathbb{R}^{n \times (d-1)}$*

$\mathbf{\Theta}^T \leftarrow \text{Transpose}(\mathbf{\Theta})$ *// Group same-angle values*

$\mathbf{B} \leftarrow \text{ByteShuffle}(\mathbf{\Theta}^T)$ *// Separate exponent/mantissa bytes*

$\mathbf{C} \leftarrow \text{ZstdCompress}(\mathbf{B})$

return Header $\|\mathbf{C}$

4 Evaluation

4.1 Experimental Setup

Table 2 compares compression methods on `jina-embeddings-v4` embeddings. Standard compressors and scientific floating-point compressors [13, 12, 11] all achieve under 1.10×. Trans+Shuffle+Zstd achieves 1.20× by grouping exponent bytes for better entropy coding and serves as our baseline for subsequent experiments, representing the best lossless method. Although our method introduces bounded error, we compare against lossless baselines because our error is below 1e-7, under float32 machine epsilon of 1.19e-7, making reconstructed values indistinguishable at float32 precision. Mantissa truncation variants illustrate the trade-off between compression ratio and reconstruction error.

Table 2: Baseline comparison (`jina-embeddings-v4`, 7600 vectors, 2048d). Sizes in MB.

Method	Size	Ratio	Max Err	Mean Err	Cos Max Err
Raw float32	59.38	1.00×	0	0	0
gzip -9	55.14	1.08×	0	0	0
brotli -11	54.52	1.09×	0	0	0
zstd -19	55.05	1.08×	0	0	0
npz	55.14	1.08×	0	0	0
fpzip	54.11	1.10×	0	0	0
zfp	58.99	1.01×	0	0	0
SZ3	55.03	1.08×	0	0	0
Trans+Shuffle+Zstd (Baseline)	49.57	1.20×	0	0	0
Baseline+Truncate 5 bits	42.23	1.47×	9e-7	2e-8	2e-6
Baseline+Truncate 6 bits	40.30	1.55×	2e-6	5e-8	5e-6
Baseline+Truncate 7 bits	38.40	1.62×	4e-6	9e-8	1e-5
Spherical (Ours)	**37.59**	**1.58×**	**9e-8**	**2e-8**	**2e-7**

The spherical transform requires $O(nd)$ operations using backward cumulative summation for partial norms; our C implementation achieves over 1 GB/s for the transform alone. With zstd level 1, total pipeline throughput reaches 487 MB/s encoding and 605 MB/s decoding while maintaining the same compression ratio. See Appendix B.

4.2 Main Results

Table 3 presents results across 26 configurations: 20 text models on 7600 AG News samples, 3 image models on CIFAR-10, and 3 multi-vector ColBERT models. All embeddings are unit-normalized.

Table 3: Compression results across 26 embedding configurations. Sizes in MB.

Model	Dim	Raw	Baseline	Ours	Ratio	Impr.
Text Embeddings						
MiniLM	384	11.13	9.37	7.43	1.50×	+26.0%
E5-small	384	11.13	9.10	7.31	1.52×	+24.5%
GTE-small	384	11.13	9.19	7.29	1.53×	+26.0%
BGE-base	768	22.27	18.60	14.61	1.52×	+27.3%
E5-base	768	22.27	18.19	14.31	1.56×	+27.2%
GTE-base	768	22.27	18.33	14.30	1.56×	+28.2%
MPNet	768	22.27	18.76	14.56	1.53×	+28.9%
Nomic-v1.5	768	22.27	18.57	14.58	1.53×	+27.4%
EmbedGemma-300m	768	22.27	18.72	14.82	1.50×	+26.3%
`jina-code-embeddings-0.5b`	896	25.98	21.89	17.07	1.52×	+28.2%
`jina-embeddings-v3`	1024	29.69	24.95	19.81	1.50×	+26.0%
`jina-clip-v2`	1024	29.69	24.97	20.03	1.48×	+24.6%
BGE-large	1024	29.69	24.85	19.36	1.53×	+28.4%
E5-large	1024	29.69	24.32	18.94	1.57×	+28.4%
mE5-large	1024	29.69	24.32	18.91	1.57×	+28.6%
GTE-large	1024	29.69	24.34	18.85	1.58×	+29.0%
Qwen3-Embed-0.6B	1024	29.69	24.94	19.52	1.52×	+27.8%
BGE-M3	1024	29.69	24.91	19.38	1.53×	+28.6%
`jina-code-embeddings-1.5b`	1536	44.53	37.48	28.40	1.57×	+32.0%
`jina-embeddings-v4`	2048	39.06	32.44	24.61	1.59×	+31.8%
Multimodal Image						
`jina-clip-v1`	768	5.86	4.90	3.88	1.51×	+26.5%
`jina-clip-v2`	1024	7.81	6.52	5.22	1.50×	+24.9%
`jina-embeddings-v4`	2048	15.63	12.95	9.84	1.59×	+31.6%
Multi-Vector ColBERT						
`jina-embeddings-v4` ColBERT	128	27.70	22.69	18.82	1.47×	+20.5%
`jina-colbert-v2`	1024	243.22	202.96	160.48	1.52×	+26.5%
BGE-M3 ColBERT	1024	239.89	197.87	154.13	1.56×	+28.4%

Compression ranges from 1.47× to 1.59× across all 26 configurations, representing a 20 to 32% improvement over baseline. Results are consistent across text, image, and multi-vector embeddings, indicating that compression derives from the unit-norm constraint rather than modality-specific patterns. For ColBERT indices with 50 to 100 embeddings per document, a 1M document collection compresses from approximately 240 GB to 160 GB. Entropy analysis and ablation studies appear in Appendix D and E.

5 Conclusion

We presented a compression method for unit-norm embeddings achieving 1.5× compression by exploiting the concentration of spherical angles around $\pi/2$ in high dimensions. Using double precision for intermediate calculations, reconstruction error is below 1e-7, under float32 machine epsilon, preserving retrieval quality perfectly while achieving $10\times$ lower error than mantissa truncation at the same compression ratio. The method applies to text, image, and multi-vector embeddings without training or codebooks. Our approach fills the gap between lossless compression at 1.2× and lossy quantization at 4× or higher, targeting applications such as embedding caching, serialization

protocols, and archival storage. For applications requiring bit-exact reconstruction, lossless methods remain necessary; for retrieval workloads where approximate similarity suffices, lossy quantization achieves higher ratios.

References

[1] Francesc Alted. Blosc: A blocking, shuffling and lossless compression library. `https://www.blosc.org`, 2010.

[2] Rajesh Bordawekar, Bulent Abali, and Ming-Hung Chen. Efloat: Entropy-coded floating point format for deep learning. *arXiv preprint arXiv:2102.02705*, 2021.

[3] Tony Cai, Jianqing Fan, and Tiefeng Jiang. Distributions of angles in random packing on spheres. *The Journal of Machine Learning Research*, 14(1):1837–1864, 2013.

[4] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. Fcbench: Cross-domain benchmarking of lossless compression for floating-point data. *Proceedings of the VLDB Endowment*, 17(6):1418–1431, 2024.

[5] Yann Collet. Zstandard - fast real-time compression algorithm. `https://facebook.github.io/zstd/`, 2016.

[6] Insu Han, Praneeth Kacham, Amin Karbasi, Vahab Mirrokni, and Amir Zandieh. Polarquant: Quantizing kv caches with polar transformation. *arXiv preprint arXiv:2502.02617*, 2025.

[7] Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Or Ozeri, Ilias Ennmouri, Michal Malka, Sang (Peter) Chin, Swaminathan Sundararaman, and Danny Harnik. Zipnn: Lossless compression for ai models. In *IEEE CLOUD*, pages 186–198, 2025.

[8] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[9] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 39–48, 2020.

[10] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. Matryoshka representation learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 30233–30249, 2022.

[11] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data*, 9(2):485–498, 2023.

[12] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014.

[13] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[14] Aamir Shakir, Tom Aarsen, and Sean Lee. Binary and scalar embedding quantization for significantly faster and cheaper retrieval. `https://huggingface.co/blog/embedding-quantization`, 2024.

[15] Will Trojak and Freddie D. Witherden. Inline vector compression for computational physics. *Computer Physics Communications*, 258:107562, 2021.

[16] Théophane Vallaeys, Matthew J. Muckley, Jakob Verbeek, and Matthijs Douze. Qinco2: Vector compression and search with improved implicit neural codebooks. In *ICLR*, 2025.

[17] Zeyu Yang, Tianyi Zhang, Jianwen Xie, Chuan Li, Zhaozhuo Xu, and Anshumali Shrivastava. To compress or not? pushing the frontier of lossless genai model weights compression with exponent concentration. *arXiv preprint arXiv:2510.02676*, 2025.

[18] Yuxuan Yue, Zukang Xu, Zhihang Yuan, Dawei Yang, Jianglong Wu, and Liqiang Nie. Pcdvq: Enhancing vector quantization for large language models via polar coordinate decoupling. *arXiv preprint arXiv:2506.05432*, 2025.

[19] Tianyi Zhang, Yang Sui, Shaochen Zhong, Vipin Chaudhary, Xia Hu, and Anshumali Shrivastava. 70% size, 100% accuracy: Lossless llm compression for efficient gpu inference via dynamic-length float. *arXiv preprint arXiv:2504.11651*, 2025.

A Python Implementation

```python
import numpy as np, zstandard as zstd

def compress(x, level=19):  # x: (n, d) unit-norm float32
    n, d = x.shape
    xd = x.astype(np.float64)  # Double precision for transforms
    ang = np.zeros((n, d-1), np.float64)
    for i in range(d-2):
        r = np.linalg.norm(xd[:, i:], axis=1)
        ang[:, i] = np.arccos(np.clip(xd[:, i] / r, -1, 1))
    ang[:, -1] = np.arctan2(xd[:, -1], xd[:, -2])
    ang = ang.astype(np.float32)  # Store as float32
    shuf = np.frombuffer(ang.T.tobytes(), np.uint8).reshape(-1, 4).T.tobytes()
    return np.array([n, d], np.uint32).tobytes() + zstd.compress(shuf, level)

def decompress(blob):
    n, d = np.frombuffer(blob[:8], np.uint32)
    ang = np.frombuffer(zstd.decompress(blob[8:]), np.uint8).reshape(4, -1).T
    ang = np.frombuffer(ang.tobytes(), np.float32).reshape(d-1, n).T
    ang = ang.astype(np.float64)  # Double precision for inverse
    x, s = np.zeros((n, d), np.float64), np.ones(n, np.float64)
    for i in range(d-2):
        x[:, i] = s * np.cos(ang[:, i])
        s *= np.sin(ang[:, i])
    x[:, -2], x[:, -1] = s * np.cos(ang[:, -1]), s * np.sin(ang[:, -1])
    return x.astype(np.float32)
```

B Compression Speed

The reference Python implementation achieves 21 MB/s encoding due to an $O(nd^2)$ loop for partial norm computation. Our C implementation reduces this to $O(nd)$ by precomputing partial squared norms via backward cumulative summation:

$$r_i^2 = \sum_{j=i}^{d-1} x_j^2 = r_{i+1}^2 + x_i^2, \quad r_{d-1}^2 = x_{d-1}^2 \tag{1}$$

This eliminates redundant computation while maintaining numerical precision through double-precision accumulation. The spherical transform achieves over 1000 MB/s, a 50$\times$ speedup.

Table 4 shows throughput across zstd compression levels. The compression ratio remains nearly constant at 1.50–1.52$\times$ regardless of zstd level because the spherical transform already concentrates exponents into a low-entropy distribution. Higher zstd levels provide negligible benefit while reducing encoding speed by 100$\times$. We recommend level 1 for most applications: it achieves 487 MB/s encoding with the same 1.52$\times$ compression ratio as level 19.

Table 4: Throughput vs. zstd level (768d, 100 MB, single-threaded CPU). Compression ratio is constant because the spherical transform already minimizes exponent entropy.

Level	Size (MB)	Ratio	Enc (MB/s)	Dec (MB/s)
1	65.7	1.52×	487	605
3	65.8	1.52×	400	609
5	66.1	1.51×	285	600
7	66.2	1.51×	258	583
9	66.3	1.51×	218	587
11	66.4	1.51×	143	581
13	66.4	1.51×	56	573
15	66.3	1.51×	44	557
17	65.7	1.52×	10	620
19	66.6	1.50×	7	555
21	66.5	1.50×	5	564

C Formal Analysis

Theorem 1 (Exponent Bound for Bounded Intervals)**.** For any value $\theta \in [a, b]$ with $0 < a \leq b$, the IEEE 754 exponent satisfies $\mathcal{E}(\theta) \in \{e_{\min}, \ldots, e_{\max}\}$ where $e_{\min} = \lfloor \log_2 a \rfloor + 127$ and $e_{\max} = \lfloor \log_2 b \rfloor + 127$. The number of distinct exponents is at most $\lfloor \log_2 b \rfloor - \lfloor \log_2 a \rfloor + 1$.

Proof. The IEEE 754 exponent function is $\mathcal{E}(x) = \lfloor \log_2 |x| \rfloor + 127$ for $x \neq 0$. For $\theta \in [a, b]$, we have $\log_2 a \leq \log_2 \theta \leq \log_2 b$, so $\lfloor \log_2 \theta \rfloor \in \{\lfloor \log_2 a \rfloor, \ldots, \lfloor \log_2 b \rfloor\}$. □

Corollary 2 (Spherical Angle Exponents)**.** For spherical angles $\theta_i \in [\epsilon, \pi]$: if $\epsilon \geq 0.5$, at most 3 exponents; if $\epsilon \geq 0.125$, at most 5 exponents; if $\epsilon \geq 2^{-k}$, at most $k + 2$ exponents.

Corollary 3 (Entropy Upper Bound)**.** For spherical angles with at most k distinct exponents, the exponent entropy is bounded by $H(\mathcal{E}_{\text{sph}}) \leq \log_2 k$ bits.

D Entropy Analysis

Table 5 compares the byte-level entropy of Cartesian versus spherical representations across 11 embedding models spanning 384 to 1024 dimensions.

Table 5: Entropy comparison between Cartesian and Spherical representations in bits/byte.

		Total Entropy		Cartesian Exponent		Spherical Exponent	
Model	**Dim**	Cartesian	Spherical	Entropy	Unique	Entropy	Unique
MiniLM	384	7.35	6.58	2.61	41	0.10	13
E5-small	384	7.34	6.58	2.51	23	0.10	9
GTE-small	384	7.37	6.55	2.61	26	0.13	11
MPNet	768	7.38	6.50	2.65	33	0.06	12
BGE-base	768	7.37	6.51	2.54	27	0.06	14
E5-base	768	7.35	6.51	2.36	25	0.05	12
GTE-base	768	7.37	6.51	2.60	25	0.08	15
Nomic-v1.5	768	7.37	6.51	2.55	26	0.05	9
BGE-large	1024	7.37	6.47	2.54	28	0.05	11
E5-large	1024	7.36	6.48	2.40	24	0.04	14
GTE-large	1024	7.37	6.48	2.48	26	0.05	11
Average	—	7.36	6.52	2.53	28	0.07	12
Reduction	—	0.84 bits/byte		2.46 bits/byte		—	

The entropy measurements validate our theoretical framework. Corollary 2 predicts that spherical angles use fewer exponents than Cartesian coordinates. Cartesian coordinates span multiple orders of magnitude, requiring 23 to 41 unique exponents, while spherical angles bounded to $[0, \pi]$ use

only 9 to 15. Corollary 3 bounds exponent entropy at $\log_2 k$ bits for k unique exponents. With $k \approx 12$, this gives $\log_2 12 \approx 3.6$ bits. The observed 0.04 to 0.13 bits/byte is 30 to 90× lower than this bound because angles concentrate around $\pi/2$ in high dimensions [3], producing highly non-uniform exponent distributions dominated by exponent 127.

The exponent entropy reduction from 2.53 to 0.07 bits/byte accounts for most of the compression gain. Exponent bytes comprise 25% of float32 data, saving $0.25 \times 2.46 \approx 0.61$ bits/byte. The observed total entropy reduction of 0.84 bits/byte exceeds this because byte shuffling also improves mantissa compression when exponents are concentrated.

E Ablation Studies

E.1 Matryoshka Dimension Ablation

Modern embedding models support Matryoshka representations [10], where embeddings can be truncated to lower dimensions while preserving semantic quality. Table 6 tests how compression varies with dimension for the *same model* at different truncation levels, isolating the effect of dimensionality.

Table 6: Matryoshka ablation: Same model at different truncation dimensions (2000 vectors). Sizes in KB.

Model	Dims	Raw	Baseline	Ours	Ratio	Impr.
`jina-embeddings-v3`	64	500	425	360	1.39×	+18.1%
`jina-embeddings-v3`	128	1000	848	703	1.42×	+20.7%
`jina-embeddings-v3`	256	2000	1689	1379	1.45×	+22.5%
`jina-embeddings-v3`	512	4000	3377	2730	1.47×	+23.7%
`jina-embeddings-v3`	768	6000	5066	4029	1.49×	+25.7%
`jina-embeddings-v3`	1024	8000	6753	5344	1.50×	+26.4%
`jina-clip-v2`	64	500	423	358	1.40×	+18.2%
`jina-clip-v2`	128	1000	846	703	1.42×	+20.4%
`jina-clip-v2`	256	2000	1687	1383	1.45×	+22.0%
`jina-clip-v2`	512	4000	3373	2711	1.48×	+24.4%
`jina-clip-v2`	768	6000	5058	4027	1.49×	+25.6%
`jina-clip-v2`	1024	8000	6740	5364	1.49×	+25.7%
`jina-embeddings-v4`	128	1000	773	614	1.63×	+20.6%
`jina-embeddings-v4`	256	2000	1549	1202	1.66×	+22.4%
`jina-embeddings-v4`	512	4000	3104	2323	1.72×	+25.2%
`jina-embeddings-v4`	1024	8000	6208	4555	1.76×	+26.6%
`jina-embeddings-v4`	2048	16000	12406	8774	1.82×	+29.3%

The improvement increases with dimension: from 18% at 64d to 29% at 2048d. Higher dimensions have $d - 1$ angles versus d Cartesian coordinates, so the fraction of data benefiting from exponent concentration grows toward unity as $(d - 1)/d \to 1$. The angle concentration phenomenon [3] also strengthens with dimension, reducing entropy further. Text, multimodal, and vision-language models show consistent behavior, with `jina-embeddings-v4` achieving the highest ratios due to its 2048-dimensional output.

E.2 Scale and Domain Ablation

Table 7 tests robustness across batch sizes and text domains.

The improvement is stable at 24 to 26% across batch sizes from 100 to 7600 vectors, indicating applicability from single API responses to large vector databases. Compression is also consistent across text domains including news, product reviews, medical abstracts, and legal documents, confirming that the method exploits geometric properties rather than domain-specific patterns.

Table 7: Scale ablation (MiniLM-384d, varying N) and domain ablation (2000 vectors each). Sizes in KB.

Ablation	N	Raw	Baseline	Ours	Ratio	Impr.
Scale (AG News, MiniLM-384d)						
N=100	100	150	128	103	1.46×	+24.2%
N=500	500	750	635	505	1.49×	+25.6%
N=1000	1000	1500	1266	1011	1.48×	+25.2%
N=2000	2000	3000	2529	2004	1.50×	+26.2%
N=5000	5000	7500	6308	5051	1.49×	+24.9%
N=7600	7600	11400	9588	7605	1.50×	+26.0%
Domain (N=2000, MiniLM-384d)						
News (AG News)	2000	3000	2529	2004	1.50×	+26.2%
Reviews (Amazon)	2000	3000	2523	2022	1.48×	+24.8%
Medical (PubMed)	1000	1500	1266	1016	1.48×	+24.6%
Legal (ECtHR)	2000	3000	2509	2008	1.49×	+25.0%

E.3 Geometric Distribution Analysis

Our compression exploits the geometric constraint that unit-norm embeddings lie on a hypersphere. To understand which geometric properties drive compression, Table 8 tests various distributions including uniform random points on the sphere, clustered points using von Mises-Fisher distributions with varying concentration, sparse vectors, orthogonal vectors, and real embeddings.

Table 8: Compression across geometric distributions (2000 vectors, 768d). vMF = von Mises-Fisher; κ = concentration parameter.

Distribution	Avg Cos-Sim	Baseline	Spherical	Impr.
Uniform on sphere	0.000	1.19×	1.50×	+20.9%
vMF clustered (κ=50, 5 clusters)	0.001	1.18×	1.50×	+20.9%
vMF moderate (κ=10)	0.000	1.19×	1.50×	+20.9%
vMF concentrated (κ=100)	0.016	1.18×	1.50×	+20.9%
vMF tight (κ=1000)	0.472	1.18×	1.50×	+21.0%
Orthogonal vectors	0.000	1.18×	1.50×	+20.9%
Sparse (10% nonzero)	0.000	5.83×	7.22×	+19.3%
BGE-base (real)	0.048	1.19×	1.52×	+21.7%

The improvement is consistent at 20 to 21% regardless of how vectors are distributed on the sphere. Uniform random points, tightly clustered points with κ=1000 and average cosine similarity of 0.47, orthogonal vectors, and real embeddings all achieve nearly identical compression ratios. This confirms that the spherical transform exploits a geometric property of bounded angles that is independent of the distribution. Sparse vectors already compress well with the baseline due to zero values, but spherical still adds 19% improvement.

E.4 Reduced Precision: BF16 Embeddings

Many inference pipelines use BF16 to reduce GPU memory. BF16 retains the same 8 exponent bits as float32 but reduces mantissa bits from 23 to 7. When BF16 embeddings are stored as float32, the lower 16 mantissa bits are zeros, creating a distinct compression profile. Table 9 compares compression methods on BF16 embeddings.

The baseline achieves 2.91× on BF16-as-float32 because the 16 zero mantissa bits per float compress almost perfectly. The spherical transform is *not recommended* for BF16 inputs: the transcendental functions (arccos, arctan2) produce full float32 outputs, destroying the zero-bit pattern and yielding only 1.50×, worse than baseline. For BF16-as-float32 storage, the baseline Trans+Shuffle+Zstd remains optimal. The spherical method targets native float32 embeddings where no such zero-bit structure exists.

Table 9: BF16 vs float32 compression (BGE-base, 768d, 2000 vectors). Sizes in MB.

Format	**Method**	**Size**	**Ratio**	**vs Baseline**
Float32	Baseline (Trans+Shuffle+Zstd)	4.92	1.19×	—
Float32	Spherical (Ours)	3.90	1.50×	+26%
BF16-as-float32	Baseline (Trans+Shuffle+Zstd)	2.11	2.91×	—