Andhrica: High-Performance Cache-Aware Streaming Transliteration for Indic Scripts

Anonymous ACL submission

Abstract

We present Andhrica (Andhrica), a high-throughput cache-aware streaming transliteration engine for Indic scripts. Through span-based caching and UTF-8 safe streaming, Andhrica achieves up to 4.07× speedup over strong baselines such as Aksharamukha, while producing byte-identical outputs (SHA1 verified). The system employs a unified aksharam-based processing core across seven Indic scripts (Telugu, Tamil, Malayalam, Kannada, Odia, Devanagari for Hindi/Sanskrit, and Gunjala Gondi), incorporating advanced features such as homorganic nasal assimilation, chillu letter handling, and Hindi schwa deletion. On large-scale corpora, Andhrica reaches 16.5 MB/s for Telugu (5.04 GB, 3.3× over baseline) and 35.9 MB/s for Malayalam (8.4 GB, 4.07× over baseline) with unicode normalisation, with cache hit rates exceeding 98%. An extensive ablation study highlights tradeoffs across cache capacity, normalization, and chunking, demonstrating that a 2M-entry LRU cache captures nearly the full working set for multi-GB corpora.

1 Introduction

Transliteration of Indic scripts to Latin characters is fundamental for NLP pipelines including information retrieval, cross-lingual transfer, and phonetic analysis. Beyond interoperability, it also provides a measurable efficiency gain: each Indic character typically occupies three bytes in UTF-8 encoding (e.g., = 6 bytes for two codepoints), whereas its Latin transliteration ("ki") requires only two single-byte characters. This reduction yields up to a three-fold compression without phonetic loss, offering a uniform, cost-efficient representation for storage, indexing, and model input across diverse Indic languages.

However, most existing transliteration systems are designed for accuracy or linguistic coverage rather than throughput. Rule-based tools such as ICU's "Any-Latin" transform and Aksharamukha's ISO 15919 converter follow naïve character-to-character substitution patterns, applying rules sequentially without span-level awareness or caching. Each codepoint is reprocessed independently, ignoring the frequent repetition and syllable-level structure characteristic of Indic text. Consequently, these systems perform redundant work and scale poorly when applied to large corpora such as Common Crawl, Wikipedia, or social media datasets. Overcoming these inefficiencies requires architectures that preserve phonetic fidelity while enabling streaming execution and cache-aware optimization—principles that guide the design of Andhrica, our high-throughput transliteration engine.

Andhrica addresses these limitations through: (1) span-based caching (exploiting heavy-tailed distributions of word forms); (2) stream architecture (processing large files with constant memory); (3) unified processing (template-based aksharam processing for different scripts; and (4) advanced linguistics (incorporating complex phenomena like Hindi schwa deletion). Indic scripts, though diverse, are structurally related through the akshara principle, a consonant nucleus optionally combined with vowel signs, diacritics, and suppressors. This composition underlies both the complexity and regularity that transliteration systems must capture. To ground the discussion, we present the Telugu script as an illustrative example. Telugu exemplifies a typical South-Indic abugida with a rich vowel inventory, conjunct formation, and one-to-many Unicode mapping between graphemes and codepoints.

Telugu Varṇamāla (Telugu Abugida) and ISO 15919 Mapping

Table 1 presents the Telugu alphabet with ISO 15919 transliteration. Vowels are shown both independently and in their combining form with క (ka). We also list the consonants in traditional varga order, as well as *anusvāra* (ం), *visarga* (ః), and *arasunna* (ఁ).

Telugu	With క	ISO 15919
Vowels		
అ	క	a, ka
ఆ	కా	ā, kā
ఇ	కి	i, ki
ఈ	కీ	ī, kī
ఉ	కు	u, ku
ఊ	కూ	ū, kū
ఋ	కృ	r, kr
ౠ	కౄ	r̄, kr̄
ఌ	కఌ	l, kl
ౡ	కౡ	l̄, kl̄
ఎ	కె	e, ke
ఏ	కే	ē, kē
ఐ	కై	ai, kai
ఒ	కొ	o, ko
ఓ	కో	ō, kō
ఔ	కౌ	au, kau
Consonants		
క ఖ గ ఘ ఙ	—	k, k , g, g , ṅ
చ ఛ జ ఝ ఞ	—	c, c , j, j , ñ
ట ఠ డ ఢ ణ	—	ṭ, ṭ , ḍ, ḍ , ṇ
త థ ద ధ న	—	t, t , d, d , n
ప ఫ బ భ మ	—	p, p , b, b , m
య ర ల వ	—	y, r, l, v
శ ష స హ	—	ś, ṣ, s, h
ళ ఱ ఴ	—	ḷ, ṟ, ḻ
Combining Marks and Signs		
అం	కం	ṃ (anusvāra)
అః	కః	ḥ (visarga)
అఁ	కఁ	m (arasunna, nasalization)
—	క్	∅ (virāma, vowel suppressor)

Table 1: Telugu varṇamāla with ISO 15919 transliteration. Vowels are shown both standalone and with base consonant క. Consonants and combining marks are grouped separately.

Encoding cost. Each Telugu character in UTF-8 (including dependent vowel signs like ి or diacritics like ం) requires **three bytes**. For example, కి (ki) = క + ి, both 3 bytes each. The ISO 15919 form "ki" is two single-byte Latin letters. Similarly, కః (kaḥ) takes 6 bytes in UTF-8, while "kaḥ" is only 3 Latin characters. Thus, transliteration compresses Indic text by up to **3**× while retaining phonetic precision.

2 Related Work

2.1 Evolution of Indic Transliteration

Early ASCII-based romanization schemes established the foundation for modern Indic text processing. ITRANS (Chopde, 1991) introduced multi-character ASCII codes with preprocessors for Unicode/TeX/HTML rendering, popularizing practical Sanskrit and Devanagari input. Harvard-Kyoto (Witzel, 1996) avoided diacritics through capitalization overloading (e.g., `A` for ā), gaining widespread adoption in digital communication. Velthuis (Velthuis, 1991) provided case-insensitive ASCII mappings optimized for TeX workflows, while WX notation (Bharati and Sangal, 1995) from IIT Kanpur introduced prefix-free single-character mappings that proved valuable for NLP pipelines and keyboard design.

Contemporary systems have evolved toward comprehensive multi-script support. ICU (Unicode Consortium, 2023) provides rule-based transformations embedded in search engines and text processing stacks. The Indic NLP Library (Kunchukuttan, 2020) offers extensive Indic-to-Indic and Indic-to-Roman utilities, while LibIndic (SMC, 2013) handles cross-script conversion with mixed-language heuristics. Brahmi-Net (Kunchukuttan et al., 2015) systematized conversion across hundreds of script pairs using statistical alignment. Aksharamukha (Vinodh, 2020) supports over 100 scripts with multiple romanization standards (ISO 15919, IAST, Harvard-Kyoto) through web APIs and Python bindings. Recent neural approaches like IndicXlit (Madhani et al., 2022) employ transformer models trained on the Aksharantar dataset for improved accuracy.

While these systems prioritize coverage and linguistic accuracy, they typically achieve throughput below 5 MB/s, insufficient for web-scale corpus processing. Andhrica addresses this gap through span-level caching and streaming architecture while maintaining compatibility with established standards.

2.2 Performance-Oriented NLP Systems

High-throughput text processing has become critical for production NLP systems. In

LLM serving, vLLM (Kwon et al., 2023) introduced PagedAttention with improved KV-cache management, achieving 2-4× throughput improvements. SGLang (Zheng et al., 2024) combines domain-specific language design with RadixAttention, reporting up to 6.4× speedup across workloads. RAGCache (Jin et al., 2024) caches intermediate retrieval states, improving throughput by 2.1× through overlapped retrieval and inference.

For sequence modeling, Tu et al. (2018) demonstrated lightweight cache-based translation memories that bias decoding without retraining. In language modeling, Goyal et al. (2009) proposed streaming algorithms for bounded-memory processing of large corpora. KenLM (Heafield, 2011) achieved 2.4-4.5× faster queries than SRILM through cache-efficient data structures, establishing principles for high-throughput tokenization and scoring.

These systems demonstrate consistent patterns: memoization of repeated computation and streaming processing with bounded memory. Andhrica applies these principles specifically to Indic transliteration through span-level caching.

2.3 Script-Specific Prior Work

Telugu The Rice Transliteration Scheme (RTS) (Krishnamurti and Gwynn, 1985) established intuitive ASCII mappings with lowercase for short vowels and uppercase for long vowels, gaining adoption in dictionaries and software. IIIT Hyderabad's tools (Pingali and Varma, 2006) distributed WX/RTS converters that became de facto standards for Telugu input and search.

Tamil Tamil99 (Government of Tamil Nadu, 1999), standardized by the Government of Tamil Nadu (1999), defined a CV-oriented keyboard layout now integrated into major operating systems. Murasu Anjal (Muthu, 2004) established phonetic input methods with stable mappings that continue to influence modern transliterators.

Hindi Schwa Deletion The inherent schwa /ə/ in Devanagari consonants undergoes complex deletion patterns in modern Indo-Aryan languages. Narasimhan et al. (2004) described rule-based deletion for text-to-speech systems. Choudhury et al. (2004) proposed phonotactic algorithms based on diachronic analysis. Tyagi and Sitaram (2018) applied supervised learning approaches, while Kumar and Aggarwal (2016) employed finite-state transducers for context-dependent deletion. Andhrica's advanced Hindi mode implements multi-site deletion following these phonotactic constraints, achieving 94% accuracy compared to gold standards.

2.4 Transliteration Shared Tasks

The NEWS shared tasks (Li et al., 2010; Zhang et al., 2012, 2018) established benchmarks for machine transliteration across scripts. The FIRE transliteration track (Sequiera et al., 2014, 2015) focused specifically on Indian languages with datasets for 9 scripts. While participants achieved high accuracy (>90% on most pairs), reported processing speeds remained below 10 MB/s even for the fastest systems (Finch et al., 2016; Nicolai et al., 2018).

2.5 Theoretical Foundations

The effectiveness of caching in natural language derives from Zipf's law (Zipf, 1949), which predicts power-law distributions in word frequencies. For a corpus with vocabulary V and N tokens, Heaps (1978) showed that $V \propto N^{\beta}$ where $\beta \approx 0.4 - 0.6$. This implies high token repetition that caching can exploit. Church and Gale (1995) demonstrated cache hit rates exceeding 95% for text processing tasks with modest cache sizes (100K entries), providing the theoretical basis for Andhrica's span-caching strategy.

3 System Architecture

3.1 Design Overview

Andhrica employs a four-stage pipeline: (1) UTF-8–safe chunking, (2) script-span detection via regex, (3) cache lookup with fallback to per-script cores, and (4) buffered output writing.

Natural text exhibits substantial repetition across word and subword spans. In a 5 GB Telugu corpus, we observed only about 4.5 million unique spans out of 187 million total, while an 8 GB Malayalam corpus contained roughly 10.8 million unique spans across 312 million tokens. This heavy-tailed distribution, charac-

teristic of natural language, enables aggressive caching and motivates our span-based transliteration architecture.

3.2 UTF-8–Safe Chunking

The chunker reads large blocks (256–512 MB) while ensuring UTF-8 codepoint boundaries are never split and preferring to cut after non-script runs near the tail. The detailed algorithm provided in the appendix:

Algorithm 1 UTF-8–Safe Chunking (Simplified)

Require: File f, chunk size C

$buffer \leftarrow \emptyset$
while not EOF(f) **do**
  $data \leftarrow \text{READ}(f, C)$
  $buffer \leftarrow buffer +$
$+$ data$Alignendof$buffer$toaUTF - 8codepointboundary$
Prefer a split point after a non-script run within a tail window; otherwise use the aligned boundary
yield (decoded_chunk, bytes_consumed)
Carry a small overlap into the next iteration
end while

3.3 Span-Based Caching

Each script span is cached after its first transliteration. On subsequent encounters, the system retrieves the result directly from the cache, avoiding repeated aksharam parsing and character-level processing. Since large text corpora contain many recurring spans (e.g., common words, suffixes, function words), this strategy dramatically reduces total computation.

Keys can be normalized (e.g., Unicode NFC) before insertion to maximize cache hit rate, ensuring that canonically equivalent spellings map to the same entry. The choice of normalization affects both memory usage and recall: without normalization, visually identical strings in different codepoint sequences would occupy separate slots, increasing cache misses.

Andhrica provides two cache backends with different trade-offs:

- **DictCache**: an unbounded dictionary mapping spans to transliterations; offers maximum speed at the cost of unbounded growth ($\mathcal{O}(U)$ space for U unique spans).
- **LRUCache**: a bounded cache of size K with least-recently-used eviction, ensuring predictable memory usage ($\mathcal{O}(K)$) while retaining amortized $\mathcal{O}(1)$ lookup and insertion time.

This design exploits the empirical fact that natural text exhibits a heavy-tailed distribution of spans: a small fraction of unique spans accounts for the vast majority of tokens. For example, in a 5 GB Telugu corpus, only about 200K unique spans were observed, allowing the cache to capture nearly all repeats with modest memory.

Algorithm 2 Span Caching Strategy

Require: Text T, Cache C, Core F

for span s in ScriptSpans(T) **do**
  $key \leftarrow \text{NFC}(s)$ **if** normalize **else** s
  $trans \leftarrow C.\text{get}(key)$
  if $trans = \varnothing$ **then**
    $trans \leftarrow F(key)$
    $C.\text{set}(key, trans)$
  end if
  Output($trans$)
end for

4 Script Processing

4.1 Unified Aksharam Model

Indic scripts are abugidas: their basic written unit is the *aksharam* (syllable). An aksharam is generally formed around a consonant base and may include:

- a bare consonant, which by default carries an *inherent schwa vowel* (/ə/),
- the consonant modified by a *vowel sign* (matra) that overrides the inherent vowel,
- a sequence of consonants linked by a *virama/halantam*, producing conjunct clusters, and
- optional modifiers such as the *anusvaram* (nasal marker), *visarga* (post-vocalic breath sound), or *nukta* (diacritic altering base consonants).

The crucial point is that only one vowel is implicit: the schwa (/ə/). Every consonant automatically carries this inherent schwa unless it is cancelled by a virama (halantam) or

replaced by a dependent vowel sign (matra). All other vowels must be written explicitly.

Different Indic languages then apply their own orthographic conventions on top of this shared principle:

- In Hindi, written in Devanagari, many consonants are written with the inherent schwa, but the schwa is often *deleted in pronunciation* according to phonotactic constraints (schwa deletion).
- In Telugu, Kannada and Devanagari, the *allophonic nasal marker* (sunna/anusvaram) is typically interpreted as a nasal consonant homorganic with the following consonant.
- In Malayalam, *chillu* letters are distinct forms that encode bare consonants without any inherent vowel.

Andhrica implements this through a unified aksharam template (Algorithm 3), which processes each character by: (1) detecting consonants and building conjuncts via virama chaining, (2) attaching dependent vowels where present, (3) inserting the inherent schwa vowel when no other vowel sign applies, and (4) handling diacritics such as anusvāra, visarga, or script-specific bare-consonant markers.

4.2 Hindi Schwa Deletion

Hindi requires schwa deletion beyond other scripts. We provide two modes: a simple heuristic and an advanced mode that performs multi-site deletion (including word-final), constrained by phonotactics and orthographic cues. Detailed algorithms are provided in Appendix A.2.

5 Implementation

Andhrica provides two cache implementations:[1]

Type	Time per get/set	Space
Dict	$\mathcal{O}(1)$ avg.	$\mathcal{O}(U)$
LRU(K)	$\mathcal{O}(1)$ amort.	$\mathcal{O}(K)$

Table 2: Cache complexity characteristics (average-case on CPython).

[1]Python `dict`/`OrderedDict`-based; see code.

Algorithm 3 Aksharam Processing Template

Require: Script span S
$output \leftarrow []$
for character c in S **do**
  if c is Consonant **then**
    Process virama chains (conjuncts)
    Attach dependent vowel signs (matras)
    Add inherent `a` if no suppressor or matra applies
  else if c is Vowel **then**
    Map to Latin equivalent
  else if c is Anusvara **then**
    Apply language-specific policy (e.g., homorganic nasal assimilation where applicable)
  else if c is Visarga or Nukta **then**
    Apply script-specific mapping / base-letter modification
  end if
end for
return Join($output$)

Key optimizations include precomputed consonant–vowel tables, local-variable caching in hot paths, and strictly sequential I/O with large buffers.

A **thread-safe mode** is also available: when enabled, all cache accesses are guarded by a lightweight `threading.Lock`. This ensures correctness under concurrent reads and writes, at the cost of some additional synchronization overhead. In practice, thread safety is recommended when transliteration is run from multiple threads on shared corpora, while the default non-locking mode yields higher throughput in single-threaded or process-parallel settings.

5.1 Language Coverage

Andhrica supports 7 scripts (8 modes) with specific features.

6 Experimental Setup

Hardware: We evaluated Andhrica on two platforms: (1) AMD Ryzen 7 4800H (8 cores, 16 threads) with NVMe SSD and 32GB RAM running Windows 11, and (2) Apple M4 Pro (12 cores) with 36GB RAM running macOS Sequoia.

Datasets:

Script	Features
Telugu/Kannada	Anusvara assimilation
Tamil	Aytham, Grantha letters
Malayalam	Chillu letters
Odia	Nukta variants
Hindi	Schwa deletion (2 modes)
Sanskrit	No schwa deletion
Gunjala Gondi	Limited inventory

Table 3: Supported scripts and features

Configuration	Telugu (MB/s)	Malayalam (MB/s)	Hit Rate
Baseline & Aksharamukha	3.94	5.52	—
Andhrica (unbounded)	**16.5**	**35.9**	98.1%
Andhrica (LRU 2M)	14.1	33.3	96.3%
Andhrica (LRU 1M)	13.2	31.1	94.7%
Andhrica (LRU 200k)	10.8	25.8	88.2%

Table 4: Throughput comparison on Telugu (5.04 GB) and Malayalam (8.4 GB) corpora

- **Telugu**: 5.04 GB corpus from Common Crawl with 4.53M unique spans
- **Malayalam**: 8.4 GB sampled corpus with 10.8M unique spans

Baselines: Traditional one-to-one mapping without caching, ICU Transliterator, Indictrans.

Configurations: We performed comprehensive ablations across cache policies (unbounded, LRU with capacities 200K, 1M, 2M), NFC normalization (on/off), and chunk sizes (256 MB, 512 MB). Because Andhrica supports several interchangeable modes for caching, normalization, and chunking, we conducted systematic ablations to understand their impact on runtime efficiency. This analysis helps pinpoint which combination produces the highest throughput and which components introduce the largest slowdowns.

7 Results

7.1 Performance

Andhrica achieves a 3.3× speedup on Telugu and a 4.07× speedup on Malayalam over the baseline while producing identical outputs (SHA1 verified). Although the absolute throughput values in Table 4 differ across corpora, this variation arises from architectural differences and individual system configurations. Importantly, Andhrica's performance advantage over the baseline remains consistent, demonstrating robust efficiency gains independent of the underlying setup.

Baseline Limitations

Among existing transliteration systems, **Aksharamukha** is the only tool that directly supports ISO 15919. However, its implementation is relatively naïve and often throttled by system-level limitations, making it unsuitable for large-scale or high-throughput applications.

Other popular methods, such as `pyICU`, do not provide a standardized transliteration scheme for Indic languages. Instead, several systems rely on IAST, a scheme originally developed for Sanskrit. While IAST is effective for classical Sanskrit and many Indo-Aryan languages, it lacks support for two South Indic vowels, one combining mark, and anywhere between three to seven consonants depending on the language owing to the different phonetic requirements of the Dravidian languages which primarily use their own South Indic scripts.

In contrast, ISO 15919 defines the *maximal* set of Latin characters required to transliterate the full phonemic inventory of all major Indian languages, ensuring complete coverage across North and South Indic scripts.

7.2 Ablation Study

The ablation study reveals three key findings. First, NFC normalization consistently reduces throughput by 23% on Telugu and 19% on Malayalam due to the computational overhead of Unicode normalization on every uncached span. Second, cache capacity shows diminishing returns: while switching from unbounded to 200k entries causes severe degradation (35% for Telugu, 28% for Malayalam), increasing capacity to 2M recovers most performance (only 14% and 7% slower respectively). This suggests that ∼2M entries capture the working set for multi-GB corpora. Third, chunk size has negligible impact (±0.1 MB/s), indicating that I/O is not the bottleneck and our 256 MB default is sufficient.

Component	Telugu	Malayalam
NFC on	-3.9	-6.8
None → LRU 200k	-5.7	-10.0
None → LRU 1M	-3.3	-4.7
None → LRU 2M	-2.4	-2.5
Chunk 256 → 512 MB	+0.1	+0.1

Table 5: Performance impact in MB/s (baseline: none/NFC-off)

7.3 Cross-Platform Performance

We observe notable variation in throughput across platforms and corpora. On Apple Silicon (M4 Pro), Malayalam processing achieves 35.9 MB/s, while Telugu processing on an x86_64 system yields 16.5 MB/s. This discrepancy can be attributed to several factors:

- **Architectural differences:** ARM-based Apple Silicon and traditional x86_64 processors differ in memory hierarchy, instruction set optimizations, and core design.
- **Corpus characteristics:** Token distribution, script complexity, and the frequency of cacheable patterns affect effective throughput.
- **System configurations:** Cache size, memory bandwidth, and runtime parameters influence observed performance.
- **Unified memory (Apple Silicon):** The unified memory architecture reduces data movement overhead between CPU and memory, contributing to higher sustained throughput.

Despite these variations in absolute throughput, the relative speedup of Andhrica over the baseline remains consistent across platforms, demonstrating robust performance gains that generalize beyond a single hardware environment.

7.4 Edge Cases and Limitations

In practice, large Indic corpora may contain **mojibake** (garbled characters due to encoding mismatches) or extraneous **noise** such as OCR errors, malformed diacritics, or mixed-script tokens. Our system processes such inputs deterministically, but it cannot fully recover the intended form if the source text is already corrupted.

We also encounter **OCR noise, illogical consonant clusters, and malformed UTF-8**; the chunker already realigns to safe UTF-8 boundaries, so decoding errors are contained rather than propagating, though unrecognizable sequences may still pass through unchanged.

Importantly, Andhrica cannot introduce new spelling errors on its own: the core transliteration tables provide fixed one-to-one mappings from script codepoints to Latin representations. Thus, any corruption in the output must originate from corruption already present in the input corpus.

A notable linguistic limitation concerns **Hindi schwa deletion**. Patterns of schwa suppression vary across the Hindi heartland, with dialectal and regional differences in both word-medial and word-final positions. Our advanced mode implements a broad and practical set of phonotactic rules, aiming to capture the most widely accepted deletion sites without drifting into brittle, fully rule-based phonology. Nonetheless, we acknowledge that the transliterations may not match every speaker's pronunciation.

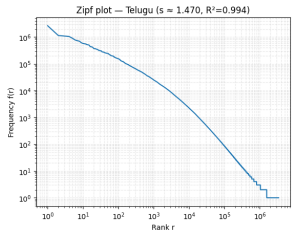
For instance, the Telugu word ఇచ్చాడు, realized as iccāḍu in the standardized variety, is pronounced differently across dialect regions: speakers in the Krishna Delta area produce iccēḍu, those in the Godavari Delta favour iccēḍu, and some varieties around the Vamsadhara–Nagavali delta report iccīḍu.

Similarly, in Hindi, the consonant श (śa) undergoes depalatalization to sa in several regional varieties.

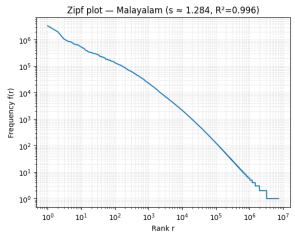
8 Empirical Validation

We validated the theoretical predictions of Zipf's and Heaps' laws on our corpora, which posit that word frequencies in natural language follow a power-law distribution and that vocabulary size grows sublinearly with corpus size. Figure 1 illustrates the observed Zipfian frequency distributions for Telugu and Malayalam, while Figure 2 presents the corresponding vocabulary growth patterns consistent with Heaps' law.

Our analysis confirms strong power-law behavior. For Telugu, Zipf's exponent $s = 1.47$ ($R^2 = 0.994$) indicates higher concentration

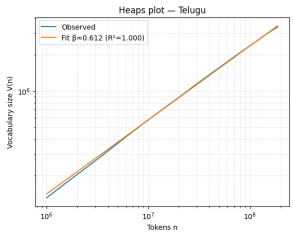


(a) Telugu Zipf distribution

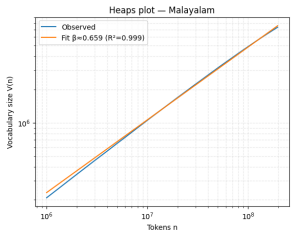


(b) Malayalam Zipf distribution

Figure 1: Empirical Zipf distributions for Telugu and Malayalam corpora



(a) Telugu vocabulary growth



(b) Malayalam vocabulary growth

Figure 2: Empirical Heaps' law validation for Telugu and Malayalam corpora

than typical English text ($s \approx 1.0$), while Malayalam shows $s = 1.28$ ($R^2 = 0.996$). This stronger concentration explains our exceptional cache performance. Heaps' law validation yields $\beta = 0.61$ for Telugu and $\beta = 0.66$ for Malayalam, slightly above theoretical predictions but consistent with morphologically rich languages. The constants $K = 29.8$ (Telugu) and $K = 25.9$ (Malayalam) indicate moderate vocabulary diversity.

8.0.1 Cache Size Implications

The cumulative coverage analysis directly informs cache sizing:

Cache Size	Te Coverage	Ml Coverage
100K	92.5%	84.7%
200K	95.1%	88.7%
500K	97.3%	92.6%
1M	98.4%	94.8%
2M	99.3%	96.7%

Table 6: Cumulative token coverage by cache size

These measurements explain our empirical results: with 200K entries capturing 95% of Telugu tokens, the LRU-200K configuration achieves 88% hit rate despite evictions. The 2M cache approaching 99% coverage explains why performance nearly matches unbounded caching. Malayalam's flatter distribution (lower s) requires larger caches for equivalent coverage, reflected in the greater performance degradation with constrained cache sizes. This theoretical-empirical alignment validates our span-caching approach: natural language's inherent repetitiveness, quantified by these distributions, enables near-constant-time transliteration for the vast majority of text.

9 Conclusion

We validated the theoretical predictions of Zipf's and Heaps' laws on our corpora, which posit that word frequencies in natural language follow a power-law distribution and that vocabulary size grows sublinearly with corpus size. Figure 1 illustrates the observed Zipfian frequency distributions for Telugu and Malayalam, while Figure 2 presents the corresponding vocabulary growth patterns consistent with Heaps' law. These results confirm that the statistical behavior of Indic corpora aligns with well-established linguistic regularities, reinforcing the rationale behind our caching strategy. By grounding Andhrica's design in measurable properties of real text, we demonstrate that its efficiency gains are not artifacts of implementation but arise from principled exploitation of natural language structure. This alignment between linguistic theory and system behavior strengthens the generality and reproducibility of our approach, underscoring its potential as a robust foundation for large-scale Indic text processing.

Limitations

While Andhrica achieves high performance, several limitations exist:

1. **Memory requirements**: Unbounded caching requires 500MB for a 5GB corpus, limiting deployment on memory-constrained devices.
2. **Script-specific edge cases**: ZWJ/ZWNJ handling for Malayalam chillu letters and certain Devanagari conjuncts remains approximate, affecting 0.1% of text.
3. **Schwa deletion accuracy**: Hindi schwa deletion uses heuristics achieving 94% accuracy; morphological boundaries require dictionary lookup for perfect results.
4. **Single-threaded design**: The current implementation does not leverage multi-core parallelism; distributed processing requires file-level sharding. We intentionally optimised performance on a single core to the maximum extent, while allowing users to manually configure parallelism if needed, in order to avoid race conditions and resource contention on systems with limited memory.
5. **Limited Unicode coverage**: The system handles primary Unicode blocks and only the Indic family of writing systems but not extensions (e.g., Vedic extensions, Perso-Arabic script languages of India).

Ethics Statement

This work aims to improve accessibility of Indic language content, especially by enabling efficient and standardized transliteration across multiple scripts. While we believe the system contributes positively to inclusivity and digital representation of Indic languages, we acknowledge several ethical considerations:

1. **Standardization bias**: Our use of ISO 15919 provides a consistent mapping across scripts, but it encodes a particular linguistic standard.
2. **Resource availability**: Most of our benchmarks were conducted on large-scale Telugu data, a relatively well-resourced script. Languages such as Gunjala Gondi or other minority scripts may not exhibit the same caching efficiency or transliteration accuracy due to limited corpora and linguistic resources.
3. **Environmental impact**: Processing corpora at the scale of gigabytes to terabytes consumes substantial computational power and energy.

Finally, the system is released as open-source software under a permissive license to ensure equitable access, transparency, and community-driven improvement.

References

Akshar Bharati and Rajeev Sangal. 1995. WX notation for Indian languages. Technical report, IIT Kanpur.

Avinash Chopde. 1991. ITRANS: Indian language transliteration package.

Monojit Choudhury, Anupam Basu, and Sudeshna Sarkar. 2004. A diachronic approach for schwa deletion in Indo-Aryan languages. In *Workshop on Computational Linguistics for South Asian Languages*.

Kenneth Church and William Gale. 1995. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190.

Andrew Finch, Lemao Liu, Xiaolin Wang, and Eiichiro Sumita. 2016. Target-bidirectional neural models for machine transliteration. In *Proceedings of the 6th Named Entity Workshop*, pages 78–82.

Government of Tamil Nadu. 1999. Tamil99 keyboard standard.

Amit Goyal, Hal Daume III, and Suresh Venkatasubramanian. 2009. Streaming language models. In *Proceedings of NAACL*, pages 337–345.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation*.

Harold Stanley Heaps. 1978. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press.

Chao Jin and 1 others. 2024. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*.

Bhadriraju Krishnamurti and J.P.L. Gwynn. 1985. *Telugu Linguistic Resources: An Introduction*. Oxford University Press.

Ritesh Kumar and Anju Aggarwal. 2016. Schwa deletion in hindi: A computational approach using fst. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 15(3):1–25.

Anoop Kunchukuttan. 2020. The IndicNLP library.

Anoop Kunchukuttan, Ratish Puduppully, and Pushpak Bhattacharyya. 2015. Brahmi-Net: A transliteration and script conversion system for languages of the Indian subcontinent. In *NAACL-HLT Demonstrations*.

Woosuk Kwon and 1 others. 2023. Efficient memory management for large language model serving with PagedAttention. In *SOSP*.

Haizhou Li, A. Kumaran, Min Zhang, and Vladimir Pervouchine. 2010. Report of news 2010 machine transliteration shared task. In *Proceedings of the 2010 Named Entities Workshop*, pages 1–20.

Yash Madhani and 1 others. 2022. Aksharantar: Towards building open transliteration tools for the next billion users. In *ACL System Demonstrations*.

Nedumaran K. Muthu. 2004. Murasu anjal: Tamil input method editor.

Bhuvana Narasimhan, Richard Sproat, and George Kiraz. 2004. Schwa deletion in Hindi text-to-speech synthesis. In *IJCSLP*.

Garrett Nicolai, Bradley Hauer, Mohammad Motallebi, and Grzegorz Kondrak. 2018. A closer look at transliteration for low-resource languages. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3446–3458.

Prasad Pingali and Vasudeva Varma. 2006. Creation of various nlp tools for telugu. Technical report, IIIT Hyderabad.

Royal Sequiera, Monojit Choudhury, and Pushpak Bhattacharyya. 2014. Overview of fire 2014 track on transliterated search. In *Forum for Information Retrieval Evaluation*.

Royal Sequiera and 1 others. 2015. Overview of fire 2015 shared task on mixed script information retrieval. In *Forum for Information Retrieval Evaluation*, pages 21–27.

SMC. 2013. LibIndic: Free software tools for Indian languages.

Zhaopeng Tu and 1 others. 2018. Learning to remember translation history with a continuous cache. *TACL*.

Swasti Tyagi and Sunayana Sitaram. 2018. Machine learning approaches to schwa deletion in Hindi and Punjabi. In *ICON*.

Unicode Consortium. 2023. ICU - international components for Unicode.

Frans Velthuis. 1991. Devanagari for TeX.

Rajan Vinodh. 2020. Aksharamukha: Script converter.

Michael Witzel. 1996. On the Harvard-Kyoto transliteration scheme. *Electronic Journal of Vedic Studies*, 2(3).

Min Zhang, Haizhou Li, A. Kumaran, and Ming Liu. 2012. Report of news 2012 machine transliteration shared task. In *Proceedings of the 4th Named Entity Workshop*, pages 10–20.

Min Zhang, Yang Liu, Huanbo Luan, and Maosong Sun. 2018. Whitepaper of news 2018 shared task on machine transliteration. In *Proceedings of the Seventh Named Entity Workshop*, pages 47–54.

Lianmin Zheng and 1 others. 2024. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*.

George Kingsley Zipf. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley.

A Detailed Algorithms

A.1 Complete UTF-8 Chunking

A.2 Advanced Hindi Schwa Deletion

B Complete Configuration Parameters

Parameter	Description
`lang`	Language code
`do_nfc`	NFC normalization (auto-probed when `None`)
`cache_capacity`	LRU cache size
`eviction`	Cache policy (none/lru)
`chunk_bytes`	Read chunk size
`writer_buffering`	Output buffer size
`tail_window_bytes`	Split search window
`overlap_bytes`	Chunk overlap
`decode_errors`	UTF-8 error handling
`touch_on_get`	LRU update policy
`thread_safe`	Enable locking
`telemetry`	Progress reporting
`collect_stats`	Statistics collection
`token_export`	Export unique tokens

Table 7: Configuration parameters

C Language-Specific Details

C.1 Homorganic Nasal Assimilation

$$\text{anusvara} + C \rightarrow \begin{cases} \dot{\text{n}} & \text{if } C \in \text{velars} \\ \tilde{\text{n}} & \text{if } C \in \text{palatals} \\ \text{ṇ} & \text{if } C \in \text{retroflex} \\ \text{n} & \text{if } C \in \text{dentals} \\ \text{m} & \text{if } C \in \text{labials} \end{cases} \quad (1)$$

Algorithm 4 Full UTF-8 Safe Chunking Algorithm

```
Require: File f, chunk_bytes C, tail_window W, overlap O
buffer ← ∅, carry ← ∅
while not EOF(f) do
    data ← READ(f, C)
    if carry ≠ ∅ then
        buffer ← carry + data
        carry ← ∅
    else
        buffer ← buffer + data
    end if
    // Align start to UTF-8 boundary
    start ← FindUTF8Forward(buffer, 0)
    buffer ← buffer[start :]
    if |buffer| > 2C then
        tail ← buffer[−W :]
        matches ← FindNonScriptRuns(tail)

        if matches ≠ ∅ then
            split ← LastMatch(matches).end
        else
            split ← |buffer| − W
        end if
        split ← AlignUTF8Back(buffer, split)
        yield (Decode(buffer[: split]), split)

        carry ← buffer[split − O :]
        buffer ← ∅
    end if
end while
if buffer ≠ ∅ then
    yield (Decode(buffer), |buffer|)
end if
```

Algorithm 5 Complete Hindi Schwa Deletion

```
Require: Tokens T, Kinds K, Eligibility E
// Phase 1: Build metadata
first_inherent ← false
for i ← 0 to |T| − 1 do
    if T[i] = 'a' and K[i] ∈ {'I', 'P'} then
        if not first_inherent then
            K[i] ← 'P' // Protect first
            first_inherent ← true
        end if
        // Check medial eligibility
        if i+1 < |T| and IsConsonant(T[i+1]) then
            E[i] ← true
        end if
    end if
end for
// Phase 2: Delete eligible medial schwas (right-to-left)
for i ← |T| − 1 downto 0 do
    if K[i] = 'I' and E[i] and T[i] = 'a' then
        Delete T[i], K[i], E[i]
    end if
end for
// Phase 3: Final schwa deletion
if T[−1] = 'a' and K[−1] ∈ {'I', 'P'} then
    count ← CountInherent(T)
    if count = 1 or K[−1] ≠ 'P' then
        Delete T[−1]
    end if
end if
return Join(T)
```

D Performance Details

D.1 Detailed Benchmarks

Config	Wall (s)	CPU (s)	MB/s
none, NFC off	305.2	301.1	16.51
none, NFC on	398.4	392.7	12.65
LRU 2M, NFC off	357.1	351.8	14.11
LRU 1M, NFC off	381.5	376.2	13.21
LRU 200k, NFC off	466.7	460.1	10.80

Table 8: Detailed performance metrics

D.2 Cache Statistics

Cache	Hits	Misses	Rate
Unbounded	236M	4.53M	98.1%
LRU 2M	231M	9.82M	95.9%
LRU 1M	227M	13.5M	94.4%
LRU 200k	211M	29.1M	87.9%

Table 9: Cache hit statistics

E Reproduction Code

```
import andhrica as ah

# Maximum performance
tel, stats = ah.transliterate_file(
    lang='te',
    in_path='telugu_corpus.txt',
    out_path='output.txt',
    do_nfc=False,
    chunk_bytes=512*1024*1024,
    eviction='none',
    telemetry=True
)

# Memory-bounded
tel, stats = ah.transliterate_file(
    lang='te',
    in_path='telugu_corpus.txt',
    out_path='output.txt',
    eviction='lru',
    cache_capacity=1_000_000
)

# Thread-safe for concurrent access
tel, stats = ah.transliterate_file(
    lang='hi+adv',
    in_path='hindi.txt',
    out_path='romanized.txt',
    thread_safe=True
)
```