

FlashTrie: A GPU-Accelerated Constrained Beam Search for Generative Retrieval

Dakshitha Anandakumar¹, Anurag Mukkara², Wenxiang Hu¹, M. Akash Kumar¹,
Jiusheng Chen¹, Qiang Lou¹, Jian Jiao¹, Ting Ye¹

¹Microsoft, Redmond, WA, USA ²NVIDIA, Santa Clara, CA, USA

Correspondence: danandakumar@microsoft.com

Abstract

Constrained decoding is essential in generative retrieval, where document identifiers generated directly from a query must exactly match a predefined library of valid IDs. At scale, decoding is often constrained using a trie with beam search but most implementations run on CPU. Limited parallelism then makes trie traversal and candidate validation a serving bottleneck as beam width grows.

We present **FlashTrie**, which addresses this limitation by optimizing constrained beam search on GPUs. It introduces an integer-aware succinct trie layout that uses bit compression to reduce memory footprint while keeping the full index in GPU high-bandwidth memory, reducing memory stalls, and a cooperative CUDA kernel that performs beam expansion, validation, and pruning entirely on-device without per-step host orchestration. GPU-native parallel primitives further replace CPU-style irregular lookup and heap maintenance, improving warp utilization and reducing divergence.

Together, these designs significantly reduce decoding latency and increase throughput while preserving retrieval quality. On a library of 800M valid identifiers, the bit-compressed layout stores the trie in 3.1 GB. With beam widths up to 1000, FlashTrie reduces mean trie-search latency to under 2 ms, achieving up to 24× speedup over a highly optimized multi-threaded CPU baseline. These improvements enable FlashTrie to scale beam sizes by up to 5× in latency-critical applications such as sponsored search. In a large-scale online A/B experiment on a commercial search engine, it delivers a statistically significant +0.71% revenue lift, enabling real-time constrained decoding at a scale previously impractical within our production serving budget.

1 Introduction

Generative retrieval (Tay et al., 2022; Metzler et al., 2021) reframes document retrieval as a sequence-to-sequence task, mapping a query directly to an identifier (docID), replacing dual-encoder indexing (Karpukhin et al., 2020). This paradigm is attractive for large-scale search and recommendation systems because it can, in principle, avoid expensive retrieval pipelines while enabling compact end-to-end modeling. In practice, however, online serving is constrained by strict latency budgets, and the decoding procedure becomes the dominant bottleneck. Autoregressive (AR) (Sutskever et al., 2014) decoders are sequential and costly, while non-autoregressive (NAR) (Gu et al., 2018; Sun and Yang, 2020) decoders regain parallelism but their per-position independence frequently produces invalid identifiers, necessitating constrained decoding to restrict generation to valid outputs (Ziems et al., 2023; Pradeep et al., 2023). Constrained decoding is therefore not merely an implementation detail but a functional requirement of practical generative retrieval systems. However, enforcing these constraints efficiently over billion-scale retrieval inventories remains challenging. Existing search structures and validation mechanisms to guarantee valid outputs can introduce substantial overhead that dominates end-to-end serving latency.

Prior Work and Challenges. Generative retrieval produces document IDs (Tay et al., 2022), entity names (Cao et al., 2021), semantic IDs (Wang et al., 2022; Penha et al., 2025), or indexed substrings (Bevilacqua et al., 2022). Constraint enforcement has been implemented through trie-based prefix traversal for explicit identifier sets (Cao et al., 2021; Ye et al., 2025; Su et al., 2026), FM-index lookup (Bevilacqua et al., 2022), lexical beam search (Hokamp and Liu, 2017; Anderson et al., 2017), grammar-based token masking (Willard and Louf, 2023; Dong et al., 2025),

A preprint of this work appeared as arXiv:2607.10044.

and predicate-logic constraints (Lu et al., 2021).

Classical constraint indexes include compressed tries such as Patricia (Morrison, 1968), double-array tries (Aoe, 1989), minimal acyclic FSAs (Daciuk et al., 2000), and succinct structures such as MARISA (Yata, 2011), which combines Patricia compression with a LOUDS topology (Jacobson, 1989). While effective for static string collections and CPU traversal, these designs target character-level keys and map poorly to GPUs, leading to irregular accesses, warp divergence, and reduced hardware efficiency (Merrill et al., 2012).

Modern runtimes optimize GPU model execution (Wang et al., 2021; Dao et al., 2022; Kwon et al., 2023), but constraint enforcement often requires additional data structures and traversal logic outside the model’s core computation. Recent systems accelerate it through structured generation (Willard and Louf, 2023; Dong et al., 2025), vectorized trie traversal (Su et al., 2026), or parallel prefix verification (Ye et al., 2025). Structured-generation frameworks compile symbolic grammars into token masks, whereas our setting requires explicit identifier inventories containing up to hundreds of millions of token sequences and sparse top- K prefix lookup. We also evaluate exact GPU hashing (Alcantara et al., 2011) as an alternative indexing strategy.

Complementary work accelerates parallel decoding (Zeng et al., 2024), scales NAR identifier vocabularies (Valluri et al., 2025), or optimizes recommendation inference (Lin et al., 2025). These methods modify model-side candidate generation rather than constraint-index traversal and could be combined with FlashTrie.

This motivates three key questions: (i) can succinct tries be adapted to token-level vocabularies without sacrificing space efficiency; (ii) can constrained beam search be executed entirely on GPU; and (iii) can such systems meet production latency constraints without degrading retrieval quality?

Our Contribution. We present **FlashTrie**, a GPU-native constrained decoding framework for generative retrieval that jointly optimizes constraint representation and decoding computation. Building on succinct tries, we redesign MARISA for integer-token vocabularies using a bit-compressed layout that reduces index size and keeps the constraint structure resident in GPU high-bandwidth memory (HBM). We further introduce GPU-friendly search and sort primitives- parallel trie child matching and

heap-free beam selection, to replace pointer-heavy traversal and reduce divergence. Constrained beam search is executed fully on-device via a cooperative multi-step CUDA kernel that performs expansion, validation, and pruning without repeated kernel re-launch overhead across decoding steps. Two-level parallelism, beam-parallel and top- K -parallel execution, exploits hierarchical GPU concurrency, utilizing processing units called streaming multiprocessors (SMs) (NVIDIA Corporation, 2024, 2020) across the device and maintaining high occupancy across the device. FlashTrie scales constrained decoding to billion-scale constraint libraries on GPU, a regime previously feasible only in offline settings, while achieving up to $24\times$ speedup over optimized CPU baselines and reducing trie-search latency to under 3 ms p95. In production A/B testing on a large-scale sponsored-search system, these latency gains enable a $3\times$ wider beam within the same serving budget, which recovers more valid identifiers and yields a +0.71% revenue lift. Code is available at [flash-trie](#).

2 Design and Implementation

We consider constrained decoding with a trie $\mathcal{T}$ over vocabulary $\mathcal{V}$. FlashTrie applies to both AR and NAR decoding; we focus on NAR in experiments because larger branching factors make constraint checking more expensive. At decoding step t the base model emits top- K candidates $(x_{t,i}, \ell_{t,i})$ with $\ell_{t,i} = \log p(x_{t,i} \mid x_{<t})$; trie-constrained beam search keeps the top- B valid prefixes under length-normalized log-probability (Wu et al., 2016) (Equation 2, Appendix F). For comparison, we use two CPU baselines built on the same LOUDS+tail/link trie skeleton (Yata, 2011) (Appendix B).

While prior work has primarily focused on trie representation and compression, our results suggest that lookup algorithms within a fixed trie layout can have an equally important impact on decoding performance. Accordingly, FlashTrie co-designs the constraint index, lookup procedure, and beam-search execution strategy for GPU architectures.

2.1 CPU Baselines: MARISA-Int and MARISA-Opt

MARISA-Int extends open-source MARISA (Yata, 2011) from character-level keys to 32-bit integer tokens and adds a minimal beam-search layer, while preserving the original build pipeline. It runs single-threaded on CPU.

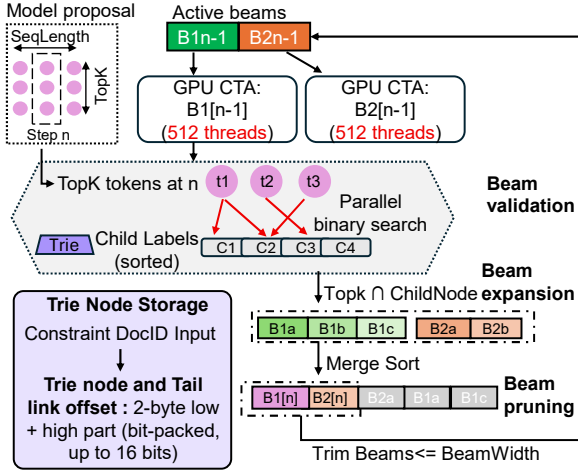


Figure 1: Overview of the FlashTrie pipeline. At each decoding step, active beam states are mapped to thread blocks (CTAs) and processed in parallel across the threads of a CTA for trie-based validation and beam expansion all executed on-device. **(Inset)** FlashTrie extends Narrow-LOUDS representation to integer tokens stored in compact structured arrays.

MARISA-Opt is a stronger CPU baseline: adds trie traversal with prefix-state caching, a multi-threaded worker pool, two-pointer merge between sorted proposals and child labels ($O(K+\Delta)$ per beam), and a bounded min-heap for top- B pruning ($O(C \log B)$). Design details and per-operation complexity are in Appendices C and D.

2.2 FlashTrie: GPU-Resident Constrained Beam Search

FlashTrie redesigns both the trie data structure and the decoding algorithm for GPU execution. The result is a GPU-resident constraint index that preserves succinct-trie space efficiency while reducing memory stalls and exposing hierarchical parallelism. We introduce each systems term below by what it does for decoding; Appendix A collects them for reference.

Narrow LOUDS for integer tokens A succinct trie represents tree topology as compact bit arrays with constant-time rank/select operations, allowing navigation through arithmetic on bit positions rather than pointer traversal; LOUDS (Jacobson, 1989) is the canonical encoding and FlashTrie introduces **Narrow-LOUDS**, a token-aware extension of the LOUDS trie. For constrained decoding, the resulting index is compact enough to reside entirely in GPU memory. Whereas published MARISA (Yata, 2011) targets 8-bit alphabets, Narrow-LOUDS supports 32-bit integer to-

kens by splitting both token labels and suffix indices into a 16-bit low part stored in the main array and a bit-packed high part materialized only when needed. This roughly halves the dominant array, enabling billion-scale libraries to fit in GPU HBM while improving memory locality by reducing amount of data fetched during traversal (Figure 1; Section 4.2).

Kernel Execution and Parallelism. FlashTrie executes all (T) decoding steps in one cooperative kernel, avoiding per-step host (CPU) launches and keeping intermediate states on the GPU. GPUs organize computation hierarchically into groups of threads called cooperative thread arrays (CTAs), which execute on SMs. FlashTrie maps this hierarchy onto beam search: queries in a batch are distributed across SMs. Each active beam is assigned to a 512-thread CTA, whose threads validate its (K) proposals in parallel using binary search over sorted trie children. After expansion, a grid-wide parallel selection retains the top-(B) hypotheses. This exposes (BK) child lookups per step while replacing the CPU baseline’s sequential expansion and heap pruning.

At each decoding step, FlashTrie exposes two levels of parallelism. First, beam-parallel expansion: all B active hypotheses are expanded at once rather than one after another. Second, proposal-parallel validation within each beam: the K candidate tokens of each hypothesis are validated simultaneously via binary search over sorted trie children. The full T -step search executes in a single kernel launch (Algorithm 1), exposing BK independent child lookups per decoding step.

After expansion, FlashTrie replaces the CPU heap with an append-only candidate buffer and parallel top- B selection, eliminating lock contention and pointer chasing while keeping intermediate state on-device (Appendix E).

3 Experimental Setup

We evaluate on 13,000 retrieval requests from a production NAR model with a 2.2M-token vocabulary. Each request generates top- K proposals ($K \in \{100, \dots, 1000\}$, beam width = K) over $T = 8$ decoding steps. The constraint trie is built with 800M keyword sequences and is resident on a single A100 80GB GPU paired with an AMD EPYC 7V13 host. For MARISA-Opt, we use 8 CPU workers to saturate the 8-core allocation; full hardware isolation details in Appendix F.

Algorithm 1 FLASHTRIE beam search. One persistent GPU kernel executes all T steps on device, exposing beam-parallel expansion and proposal-parallel validation. Kernel details: Appendix E.

Require: trie $\mathcal{T}$, initial state s_0 , beam width B , length T , per-step LM proposals $\text{TOPK}(s, K)$

```

1:  $cur \leftarrow \{s_0\}$ 
2: for  $t = 0$  to  $T-1$  do
3:    $next \leftarrow \emptyset$ 
4:   for each  $s \in cur$  in parallel do
5:     {(i) beam-parallel}
6:      $(\tau_{1:K}, \rho_{1:K}) \leftarrow \text{TOPK}(s, K)$ 
7:     for  $k = 1$  to  $K$  in parallel do
8:       {(ii) proposal-parallel}
9:        $u \leftarrow \text{CHILDLOOKUP}(\mathcal{T}, s.v, \tau_k)$ 
10:       $\{O(\log \Delta)$  binary search $\}$ 
11:      if  $u \neq \perp$  and  $s.\sigma + \rho_k > \theta_{\text{sent}}$  then
12:        append  $(s, u, \rho_k)$  to  $next$ 
13:        {lock-free}
14:      end if
15:    end for
16:  end for
17:  barrier;  $cur \leftarrow \text{TOP-B}(next)$ 
18:  {on-device}
19: end for
20:
21: return  $\text{BACKTRACE}(cur)$ 

```

Table 1: Unconstrained NAR decoding with post-hoc filtering against the full 800M-identifier inventory; per-query means. Valid and Distinct are counts per query, Invalid is a rate. The Constrained column is definitional rather than measured: trie traversal returns the full beam and emits each identifier at most once, so its valid and distinct counts both equal K .

K	Unconstrained + filter			Constrained IDs/query (= K)
	Invalid (%)	Valid (n)	Distinct (n)	
100	80.8	19.2	18.3	100
600	87.0	78.1	69.2	600
1000	88.6	114.0	97.2	1000

We measure per-request latency (mean, p50/p90/p95/p99), throughput under batched serving ($b \in \{1, 4, 8\}$ queries), on-disk index size and wall-clock build time, and Precision@100/200 scored by a transformer teacher (Section 4.6). Latency timing spans the full request path (C++ entry to result return, including GPU stream synchronization). All timings average 10 passes over the full 13,000-request set after one warm-up discard. Details in Appendix F.

4 Results

4.1 Effectiveness of Constrained Decoding

To understand the role of constrained decoding at large identifier scales, we first measure the yield of

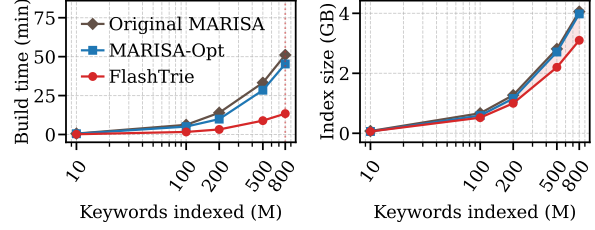


Figure 2: Trie construction scaling on 10M–800M keyword sequences, (Left) Build time, (Right) Size of constraint index. At 800M keys, FlashTrie builds in 13.4 min and uses 3.1 GB, versus 45.3 min / 4 GB for MARISA-Opt and 51.1 min / 4 GB for MARISA-Int.

unconstrained decoding followed by post-hoc filtering. We run this baseline on the same NAR top- K proposal grid used throughout the paper ($T=8$, $|\mathcal{V}| \approx 2.2\text{M}$) and validate decoded candidates against the full 800M-identifier inventory. Both methods share the same proposal distribution, scoring and length handling, differing only in whether the trie constrains expansion.

Table 1 shows that unconstrained decoding recovers only a small fraction of valid identifiers. At $K=1000$, it yields 114 valid candidates and 97.2 distinct identifiers per query after duplicate collapse, compared with 1000 identifiers returned by constrained traversal. Increasing beam width does not close this gap: valid yield falls from 19.2% at $K=100$ to 11.4% at $K=1000$, while duplicate rates increase from 4.7% to 14.7%. The invalid rate also grows sharply with identifier length, from 71% at length 2 to 94%, 98% and 99–100% at lengths 3, 4 and 5–8, biasing survivors toward short identifiers (Appendix G).

Post-hoc filtering also requires validating candidates against the same 800M-identifier inventory used during constrained decoding, but only after computation has already been spent on prefixes that would have been pruned earlier. We note that structured semantic identifiers have been observed to benefit less from constrained decoding (Penha et al., 2025). Our inventory instead consists of flat token sequences drawn from a much sparser combinatorial space, in which chance validity is correspondingly less likely.

4.2 Trie Build Time and Index Size

FlashTrie reduces both trie construction time and on-disk index size as the constraint set grows (Figure 2). Across 10M–800M sequences, the widening gap reflects a shift in the dominant cost: at small scales, fixed LOUDS and tail-construction over-

heads dominate (Appendix B), whereas at larger scales the per-edge link machinery and traversal bookkeeping become the bottleneck (Appendix D). MARISA-Int and MARISA-Opt remain close because they retain the same full-width node storage layout, while FlashTrie reduces per-node work by narrowing link offsets and splitting each 32-bit label into a compact base field plus packed high bits (Section 2.2). As the corpus grows, these savings compound, yielding increasingly larger gains in both build time and index size. Even at a large constraint index such as 800M, FlashTrie builds the trie $3.4\times$ faster than MARISA-Opt while reducing index size by 22%. The resulting index occupies only 3.1 GB, allowing it to reside entirely in GPU HBM, eliminating host-device transfers during decoding and enabling deployment on resource-constrained GPUs like A100 MIG.

4.3 Latency Results

Next, we evaluate whether FlashTrie’s GPU parallelism and fully GPU-resident constraint index maintain low trie-search latency as beam width increases. Figure 3 shows that latency grows slowly with beam width on GPU, remaining below 2 ms mean latency even at $K = 1000$. In contrast, MARISA-Opt exhibits substantially poorer scaling, with mean latency increasing from 9.03 ms to 46.30 ms over the same sweep. Consequently, FlashTrie achieves $16\times - 24\times$ mean speedup, with the performance gap widening as beam width increases. Slight latency dips near $K \approx 500$ and 800 arise from changes in the thread capacity of the on-device top- B sorter (Appendix E).

MARISA-Int (Section 2.1) is infeasible at billion-keyword scale (>3 h for a single $K=1000$ query on a 10-query subsample), so we adopt MARISA-Opt as the primary CPU comparator; full measurements are in Appendix J.

4.4 Runtime Breakdown

To profile decoding cost, we instrument the cooperative kernel and measure GPU runtime across four phases (Figure 1): beam expansion, validation, pruning and grid sync. Figure 4 (left) shows the dominant cost shifts with beam width: at $K=100$, expansion dominates ($\approx 66\%$) as trie traversal is the bottleneck. As K grows, expansion remains inexpensive because parallel beams share upper trie nodes. In contrast, the number of surviving candidates scales with $B \times K$, increasing validation and pruning cost; at $K=1000$, validation becomes dom-

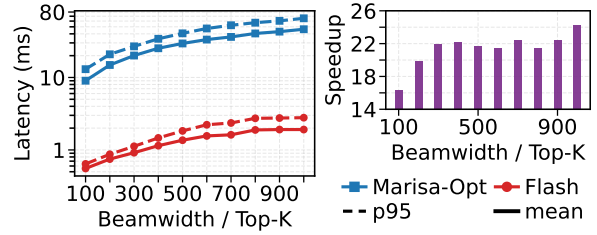


Figure 3: Per-request trie-search latency and GPU speedup vs. beam width K at $b=1$ ($|\mathcal{V}|=2.2\text{M}$, 800M-key trie). **(Left)** Mean (solid) and p95 (dashed) latency. CPU mean rises from 9.0 ms ($K=100$) to 46.3 ms ($K=1000$), with p95 65.5 ms and p99 76.7 ms at the top end. GPU mean stays sub-2 ms across the sweep (0.55–1.91 ms), with p95 ≤ 2.79 ms and p99 ≤ 3.31 ms. **(Right)** Mean GPU speedup climbs from $16\times$ to $24\times$. Full per-percentile table: Appendix H.

inant ($\approx 43\%$ of total runtime). Pruning remains stable at 15–19%, while sync overhead stays small ($\leq 6\%$, decreasing to 3.5%), confirming that the persistent kernel removes per-step host orchestration with negligible barrier overhead. Measurement details are in Appendix E.4.

4.5 Throughput Scaling with Batch Size

In Figure 4 (right), the throughput curves reflect different batching regimes. FlashTrie’s throughput rises steeply from $b=1$ to $b=8$, peaking at 1,621 q/s for $K=1000$. This is $70\times$ higher than MARISA-Opt at its own peak batch size. MARISA-Opt scales only modestly with batch size because its CPU thread pool is quickly saturated: once all workers are occupied, extra queries mostly queue (Section 2.1). FlashTrie benefits more from batching because each query is mapped to a GPU block with many threads, so larger batches better fill the device’s execution resources.

4.6 Retrieval Quality

Table 2 shows that retrieval quality improves with beam width. $P@100$ increases from 0.53 at $K=100$ to 0.78 at $K=1000$, and $P@200$ increases from 0.52 to 0.69. Gains taper after $K \approx 700$, but the trend remains monotonic overall at the operating scale of interest. Because the underlying model is unchanged, these gains stem from exploring more candidate prefixes during search rather than improved modeling. A wider beam preserves plausible prefixes that would otherwise be pruned before their continuations are scored.

FlashTrie matches MARISA-Opt to within 0.001 across all settings, indicating that GPU execu-

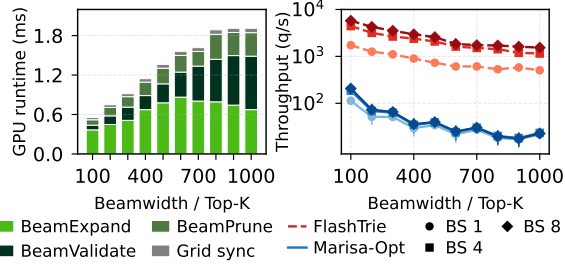


Figure 4: **(Left)** GPU runtime breakdown of FlashTrie’s cooperative beam-search kernel vs. beam width K ($b=1$, averaged over 13,000 requests). As K grows, the bottleneck shifts from trie traversal (beam expansion) to candidate handling (beam validation). **(Right)** Throughput (mean q/s) vs. K , per batch size. FlashTrie peaks at $b=8$ for all K (6,623 q/s at $K=100$, 1,621 q/s at $K=1000$), while MARISA-Opt flattens earlier. At each system’s own best batch size, FlashTrie delivers $32\times$ – $71\times$ more throughput than MARISA-Opt.

Table 2: Retrieval precision at cutoffs 100 and 200 on the 13K-request dataset, scored by a Transformer cross-encoder (Mohankumar et al., 2023). MARISA-Opt and FlashTrie agree within 0.001 for every K (collapsed into one column). Prec@200 at $K=100$ equals Prec@100 due to a truncated denominator.

K	100	200	300	400	500	600	700	800	900	1000
P100	.53	.68	.72	.74	.75	.76	.77	.77	.78	.78
P200	.52	.46	.56	.61	.64	.65	.67	.66	.68	.69

tion remains in parity with the CPU implementation. More importantly, in latency-critical serving (e.g., sponsored search; Section 4.9), MARISA-Opt becomes impractical beyond roughly $K \geq 200$ as beam width grows. Thus, the quality gains from larger beam widths are operationally realizable only with FlashTrie, which keeps constrained decoding within tight online latency budgets where large- K MARISA-Opt cannot.

4.7 Ablations: Isolating Trie and Search-Algorithm Contributions

Section 4.3 combines three effects: execution substrate (CPU vs. GPU), trie-based constraint checking, and inner-loop child-search algorithm. Because a direct CPU MARISA implementation requires hundreds of seconds per request even at $K=100$ (Appendix J), we focus here on three GPU-resident ablations that isolate the latter two while sharing FlashTrie’s cooperative kernel, scoring rules, thresholds, and length normalization.

Per-Position Token (PPT) performs depth-only filtering by intersecting top- K model tokens with

Table 3: Per-request latency (ms) for three GPU ablations vs. FlashTrie at $b=1$; p95 speedup is ablation p95/FlashTrie p95. PPT removes trie constraints; Hash-map replaces the succinct trie with an exact separate-chaining hash map over trie transitions; Linear-probe keeps the trie but replaces binary search with linear scan in the child range.

K	PPT on GPU			Hash-map on GPU			Linear-probe on GPU		
	Mean	p95	Sp.	Mean	p95	Sp.	Mean	p95	Sp.
100	0.57	0.79	$1.2\times$	1.74	2.21	$3.4\times$	110.2	134.1	$209\times$
300	1.90	2.69	$2.4\times$	3.14	5.15	$4.6\times$	123.9	149.1	$133\times$
500	3.54	5.20	$2.8\times$	4.44	7.69	$4.2\times$	128.6	155.7	$85\times$
700	6.98	10.30	$4.4\times$	5.39	9.54	$4.0\times$	141.1	178.8	$76\times$
900	9.73	14.36	$5.2\times$	7.07	11.02	$4.0\times$	151.2	194.3	$71\times$
1000	11.28	16.65	$6.0\times$	7.59	11.51	$4.1\times$	154.3	198.7	$71\times$

a set of global tokens valid at depth t , then returns all matches regardless of beam prefix. Because this depth-level token set is shared, the matched set is computed once and reused across beams. Hash-map replaces Narrow-LOUDS with an exact hash table over trie transitions (state, token) $\rightarrow$ child. Linear-probe keeps the LOUDS trie unchanged but swaps binary search for a shared-memory linear scan over child labels. Implementation details and the full per- K sweep are in Appendix K.

Table 3 yields three observations. (i) A permissive GPU baseline (PPT) is fast but not constraint-equivalent. (ii) An exact hash map over trie transitions is $3.4\times$ – $4.6\times$ slower than FlashTrie, while consuming ≈ 31 GB versus 3.1 GB for Narrow-LOUDS ($10\times$). Hence, the hash index does not fit the 10 GB A100 MIG slice that serves the production A/B test of Section 4.9, whereas FlashTrie does. Succinctness is thus not merely a space optimisation, it is what makes the index both cache-resident and deployable (Appendix K.2). (iii) Replacing binary search with a linear scan degrades p95 latency by $71\times$ – $209\times$, demonstrating that child-lookup design is a first-order determinant of decoding performance.

4.8 Public-Dataset Reproducibility

To reproduce the latency claim on public artifacts, we synthesise a parallel workload from NQ-Open queries (Kwiatkowski et al., 2019; Lee et al., 2019), the GENRE-KILT BART-large checkpoint (Cao et al., 2021; Lewis et al., 2020), and the ~ 6 M KILT Wikipedia titles (Petroni et al., 2021) as the constraint library. We sweep beam width $K \in \{100, 200, \dots, 1000\}$ over all 3,600 NQ-Open validation queries ($T=16$ decoding positions) and measure per-request trie-search latency on the same

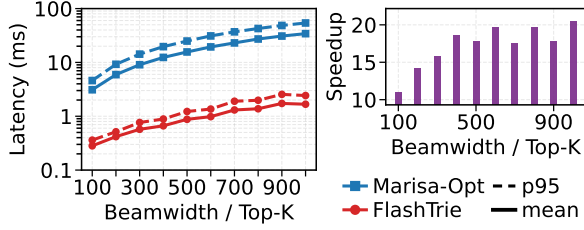


Figure 5: Latency on the NQ+GENRE public workload (3,600 queries, $T=16$, ~ 6 M-title trie) vs. beam width K . GPU mean latency rises from 0.28 ms ($K=100$) to 1.67 ms ($K=1000$); MARISA-Opt mean grows from 3.11 ms to 34.23 ms. Mean speedup: $11.0\times$ to $20.5\times$.

hardware as Section 3.

Figure 5 shows that FlashTrie reproduces the internal-dataset speedup trend (Figure 3) on entirely public artifacts. The $p99$ speedup increases from $12.8\times$ to $22.7\times$, indicating both lower and more predictable tail latency across the full K -sweep. A detailed noise-injection ablation studying the effect of cross-position incoherence (NAR-style proposal grids) is in Appendix L.

4.9 Online A/B Testing

We validate our offline findings with extensive online A/B testing on live traffic of a prominent commercial search engine. The goal is to determine whether the larger beam widths enabled by FlashTrie yield meaningful production gains in user engagement and revenue. The experiment runs for 16 consecutive days across multiple countries, with English and non-English results reported separately in Table 4.

Both arms deploy the same NAR generative retrieval model (Valluri et al., 2025) on an A100 MIG 10 GB slice. In control, production MARISA-Opt at $B=200$ is already near the online limit (≈ 25 ms $p95$; $B>200$ violates latency constraints). The treatment swaps MARISA-Opt trie stage for FlashTrie and uses the recovered latency headroom to increase beam width to $B_{\text{trt}}=600$, while keeping the model and downstream ranking stack unchanged. FlashTrie sustains $B=600$ at 15/17 ms ($p50/p95$), exploring $3\times$ more candidates within the same serving budget.

The treatment produces statistically significant improvements in revenue (0.71% revenue lift), ad coverage, impressions, and clicks. Such gains are meaningful in a mature, heavily optimized production system. Revenue gains suggest the retrieved candidates correspond to higher-value matches, while defect rates, measured using offline relevance

models, remain nonsignificant, indicating that the additional candidates are not of lower-quality.

Table 4: Online A/B test (Δ) on a commercial search engine for FlashTrie vs. CPU MARISA-Opt, reported by language segment. All metrics are statistically significant ($\alpha=0.05$). See Appendix I for the full protocol.

Language	Revenue	Coverage	Impression	Clicks	Latency
English	+0.71%	+0.22%	+0.21%	+0.17%	-32%
Non-EN	+0.68%	+0.25%	+0.37%	+0.20%	-32%

Since the underlying model and its top- K proposals are unchanged, we attribute these gains to the $3\times$ wider beam enabled by FlashTrie’s latency reduction (-32%) that narrower beams discard. Hence, this improves retrieval quality ($P@100$ rises from 0.53 to 0.78; Section 4.6) and allows more relevant candidates to reach downstream ranking.

5 Discussion

FlashTrie exemplifies a broader accelerator-era pattern: data structures traditionally implemented on CPUs can become effective on GPUs when their access patterns are redesigned for parallel execution. Our central finding is that constrained decoding need not remain a CPU bottleneck. A static succinct trie can be made GPU-native when lookup and pruning are structured for batched execution and high device utilization. This reduces decoding latency even at wider beam widths, translating to lower end-to-end inference latency under strict serving constraints. This additional headroom is important because retrieval quality is often limited not only by the model but by how much of the constrained search space decoding can afford to explore. In our online deployment, FlashTrie translated wider beams into measurable gains in user engagement and revenue without degrading ad quality. It enabled retrieval-quality operating points that were previously unreachable under production latency budgets. The constraint mechanism itself is largely domain-agnostic, operating over arbitrary tokenized integer sequences via prefix-constrained child lookup. Our public GENRE experiment (Section 4.8) gives preliminary evidence that the approach transfers beyond ad keywords, reproducing the same qualitative trend with a different model, constraint trie, and an entity-linking task. Together, these findings suggest GPU-native constraint indexing is a promising direction for scaling constrained decoding in large-vocabulary generative retrieval.

Limitations

The current implementation requires the full constraint trie to reside within a single GPU’s VRAM. At 800M keywords the trie occupies 3.1 GB on the A100 80 GB, leaving ample headroom for co-located weights; for larger libraries a natural extension is to shard subtrees across GPUs over NVLink/NVSwitch and aggregate candidates per step.

FlashTrie’s batched scheduler allocates a fixed SM partition to each query (Section 4.5), avoiding cross-slot synchronisation but capping multi-query parallelism, dynamic or work-stealing alternatives that preserve persistent-kernel coherence are a natural next step.

All experiments use an NVIDIA A100 80 GB PCIe GPU. The persistent-kernel and cooperative-groups patterns are supported on all CUDA Compute Capability ≥ 7.0 devices, but quantitative characterisation on H100/B100 remains open.

Retrieval-ID and product-catalog settings remain architectural extensions that we have not evaluated empirically, and we regard them as promising future applications rather than validated deployment domains.

References

- Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2011. [Real-time parallel hashing on the GPU](#). *ACM Transactions on Graphics*, 30(6):154:1–154:9.
- Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. 2017. Guided open vocabulary image captioning with constrained beam search. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- Jun-ichi Aoe. 1989. [An efficient digital search algorithm by using a double-array structure](#). *IEEE Transactions on Software Engineering*, 15(9):1066–1077.
- Michele Bevilacqua, Giuseppe Ottaviano, Patrick Lewis, Wen-tau Yih, Sebastian Riedel, and Fabio Petroni. 2022. [Autoregressive search engines: Generating substrings as document identifiers](#). In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. 2021. [Autoregressive entity retrieval](#). In *International Conference on Learning Representations (ICLR)*.
- Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. 2000. [Incremental construction of minimal acyclic finite-state automata](#). *Computational Linguistics*, 26(1):3–16.
- Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. [FlashAttention: Fast and memory-efficient exact attention with IO-awareness](#). In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2025. [XGrammar: Flexible and efficient structured generation engine for large language models](#). In *Proceedings of the Eighth Conference on Machine Learning and Systems (MLSys)*.
- Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. 2018. [Non-Autoregressive Neural Machine Translation](#). In *International Conference on Learning Representations*.
- Chris Hokamp and Qun Liu. 2017. [Lexically constrained decoding for sequence to sequence generation using grid beam search](#). In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, Vancouver, Canada. Association for Computational Linguistics.
- Guy Jacobson. 1989. *Succinct Static Data Structures*. Ph.D. thesis, Carnegie Mellon University.
- Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. [Natural questions: A benchmark for question answering research](#). *Transactions of the Association for Computational Linguistics*, 7:453–466.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient memory management for large language model serving with PagedAttention](#). In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*.
- Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 6086–6096.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020.

- BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7871–7880.
- Xinyu Lin, Chaoqun Yang, Wenjie Wang, Yongqi Li, Cunxiao Du, Fuli Feng, See-Kiong Ng, and Tat-Seng Chua. 2025. [Efficient inference for large language model-based generative recommendation](#). In *International Conference on Learning Representations (ICLR)*.
- Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. [Neuro-Logic decoding: \(un\)supervised neural text generation with predicate logic constraints](#). In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics.
- Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. [Scalable GPU graph traversal](#). In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 117–128.
- Donald Metzler, Yi Tay, Dara Bahri, and Marc Najork. 2021. Rethinking search: Making domain experts out of dilettantes. *SIGIR Forum*, 55(1):1–27.
- Akash Kumar Mohankumar, Bhargav Dodla, K Gururaj, and Amit Singh. 2023. [Unified generative and dense retrieval for query rewriting in sponsored search](#). In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM)*, pages 4483–4489.
- Donald R. Morrison. 1968. [PATRICIA—practical algorithm to retrieve information coded in alphanumeric](#). *Journal of the ACM*, 15(4):514–534.
- NVIDIA Corporation. 2020. NVIDIA A100 tensor core GPU architecture. <https://resources.nvidia.com/en-us-tensor-core/nvidia-ampere-architecture-whitepaper>. Accessed: 2026-06-17.
- NVIDIA Corporation. 2024. CUDA C++ programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Accessed: 2026-06-17.
- Gustavo Penha, Edoardo D’Amico, Marco De Nadai, Enrico Palumbo, Alexandre Tamborrino, Ali Vardasbi, Max Lefarov, Shawn Lin, Timothy Heath, Francesco Fabbri, and Hugues Bouchard. 2025. [Semantic IDs for joint generative search and recommendation](#). *arXiv preprint arXiv:2508.10478*.
- Fabio Petroni, Aleksandra Piktus, Angela Fan, Patrick Lewis, Majid Yazdani, Nicola De Cao, James Thorne, Yacine Jernite, Vladimir Karpukhin, Jean Maillard, Vassilis Plachouras, Tim Rocktäschel, and Sebastian Riedel. 2021. [KILT: A benchmark for knowledge intensive language tasks](#). In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2523–2544. Association for Computational Linguistics.
- Ronak Pradeep, Kai Hui, Jai Gupta, Honglei Zhuang, Adam Lelkes, Jimmy Lin, Donald Metzler, and Vinh Tran. 2023. Understanding generative retrieval at scale. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- Zhengyang Su, Isay Katsman, Yueqi Wang, Ruining He, Lukasz Heldt, Raghunandan Keshavan, Shao-Chuan Wang, Xinyang Yi, Mingyan Gao, Onkar Dalal, Lichan Hong, Ed Chi, and Ningren Han. 2026. [Vectorizing the trie: Efficient constrained decoding for LLM-based generative retrieval on accelerators](#). *arXiv preprint arXiv:2602.22647*.
- Zhiqing Sun and Yiming Yang. 2020. An EM approach to non-autoregressive conditional sequence generation. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Yi Tay, Vinh Q. Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, Tal Schuster, William W. Cohen, and Donald Metzler. 2022. [Transformer memory as a differentiable search index](#). In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Ravisri Valluri, Akash Kumar Mohankumar, Kushal Dave, Amit Singh, Jian Jiao, Manik Varma, and Gaurav Sinha. 2025. [Scaling the vocabulary of non-autoregressive models for fast generative retrieval](#). In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, KDD ’25, page 1409–1420, New York, NY, USA. Association for Computing Machinery.
- Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. [LightSeq: A high performance inference library for transformers](#). In *Proceedings of the 2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Industry Track (NAACL-Industry)*.
- Yujing Wang, Yingyan Hou, Haonan Wang, Ziming Miao, Shibin Wu, Hao Sun, Qi Chen, Yuqing Xia, Chengmin Chi, Guoshuai Zhao, Zheng Liu, Xing Xie, Hao Allen Sun, Weiwei Deng, Qi Zhang, and Mao Yang. 2022. [A neural corpus indexer for document retrieval](#). In *Advances in Neural Information Processing Systems (NeurIPS)*.
- Brandon T. Willard and Rémi Louf. 2023. [Efficient guided generation for large language models](#). *arXiv preprint arXiv:2307.09702*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, and 12 others. 2016. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Susumu Yata. 2011. MARISA-trie: Matching algorithm with recursively implemented StorAge. <https://github.com/s-yata/marisa-trie>.

Haotian Ye, Himanshu Jain, Chong You, Ananda Theertha Suresh, Haowei Lin, James Zou, and Felix X. Yu. 2025. *Efficient and asymptotically unbiased constrained decoding for large language models*. *arXiv preprint arXiv:2504.09135*.

Hansi Zeng, Chen Luo, and Hamed Zamani. 2024. *Planning ahead in generative retrieval: Guiding autoregressive generation through simultaneous decoding*. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Noah Ziems, Wenhao Yu, Zhihan Zhang, and Meng Jiang. 2023. *Large language models are built-in autoregressive search engines*. In *Findings of the Association for Computational Linguistics (ACL Findings)*.

Appendix

A Glossary of Systems Terminology

FlashTrie sits between NLP decoding and GPU systems, so the paper uses vocabulary from both. The terms introduced inline in Section 2.2 are collected in Table 5 for reference; readers familiar with GPU programming can skip this section.

B Trie Background: LOUDS with Tail/Link Compression

All three systems share the same LOUDS backbone (Yata (2011)): a $2n+1$ -bit topology vector with constant-time rank/select, plus per-node bases_labels and terminal_flags_. What differs across systems is not the core topology but auxiliary build-time structures (Patricia/TAIL, link flags, and cache) and the search algorithm used on top of this layout (Table 6).

Patricia single-child compression and the TAIL buffer. Patricia compression folds maximal single-child chains into a head node plus an offset into tail_; link nodes store offsets, not labels. Without suffix sharing, this is usually a loss for

Table 5: Systems terminology used in the main text, defined by its role in constrained decoding.

Term	Role in constrained decoding
Succinct trie	Trie stored as a bit array near the information-theoretic minimum rather than as pointer-linked nodes; what shrinks the 800M-identifier index to 3.1 GB and makes it fit on the GPU
LOUDS, rank/select	The canonical succinct encoding (one bit per child slot in level order) and the two constant-time primitives that replace pointer dereferences, so a child move is arithmetic on bit positions (Appendix B)
HBM	The GPU’s on-board DRAM; holding the whole trie there means a lookup never crosses PCIe to host memory
Kernel	One program launched from the CPU and run by many GPU threads; each launch has fixed overhead, so launching per decoding step is costly
Cooperative kernel	A kernel that synchronizes device-wide between phases, running all T steps in one launch with beam state on device
SM	Streaming multiprocessor, one of 108 execution units on an A100; the unit across which batched queries are distributed
CTA (thread block)	512 threads resident on one SM with shared on-chip memory; holds one active beam and validates its candidate tokens
Warp	32 threads executing in lockstep; the granularity at which divergence (threads branching apart, serialising each path) and coalescing (adjacent addresses served in one transaction) are decided
Occupancy	Fraction of thread slots actually resident; determines how much memory latency can be hidden
Beam-parallel / proposal-parallel	Expanding all B hypotheses at once, and checking all K proposals of a hypothesis at once (one per thread); at $B=K=1000$, up to 10^6 concurrent child lookups
MIG slice	Hardware-isolated 10 GB partition of an A100; the production serving unit, so a larger index cannot be deployed (Section 4.9)

wide labels. The net benefit depends on suffix deduplication, which can be summarized by:

$$\text{folded}(k, w, \alpha) \approx w + \alpha kw + O(\log |\text{tail}_|).$$

Folding beats expansion iff $\alpha kw + w < kw$, i.e.,

$$\alpha < 1 - \frac{1}{k}. \quad (1)$$

In byte alphabets this criterion often holds; in wide 32-bit token alphabets suffix sharing is much weaker and the supporting metadata becomes relatively expensive. This is why MARISA-Opt disables Patricia/TAIL in our integer-token setting (Section 2.1). Recursive TAIL (num_tries) helps only when the same criterion holds at each level; we use num_tries=1.

The `link_flags_` bit-vector and `cache_`. `link_flags_` tags whether `bases_` holds a label or a TAIL offset; its rank index and high-bit arrays scale with the number of link nodes. When Patricia is disabled, these structures largely collapse. MARISA’s optional `cache_` prefetch table speeds common top-of-trie transitions but adds build cost and index bytes. These component-level tradeoffs are summarized in Table 6; for full implementation details, see Yata (2011).

C CPU Baselines: Full Design Details and System Contrast

This appendix gives the full design description for the two CPU baselines summarised in Section 2.1 and the axis-by-axis contrast between all three systems (Table 6).

C.1 MARISA-Int

MARISA-Int extends the open-source MARISA codebase (Yata, 2011) with 32-bit integer-token sequences, reusing the original character-oriented storage layout and traversal routines without modifying the build pipeline. Since upstream MARISA provides only single-sequence operations (e.g., lookup, predictive_search), MARISA-Int adds a minimal beam-search layer. Each candidate is evaluated by repeatedly invoking these routines on the full prefix from the root, performing up to two root-to-node traversals per (beam, proposal) pair. No trie state is reused across steps, and pruning is implemented via `std::partial_sort` over a flat vector. The search is single-threaded, and all operations execute on the host CPU.

C.2 MARISA-Opt

MARISA-Opt builds on MARISA-Int while retaining the LOUDS-based trie layout and on-disk format, but introduces a set of implementation optimizations that make it a competitive CPU baseline. These include compact link-offset encoding, stateful traversal, multi-threaded execution, efficient per-beam expansion, and incremental top-*B* pruning (full per-operation derivations in Appendix D). Comparing FlashTrie against this optimized baseline ensures that performance gains reflect on-device parallelism rather than unoptimised CPU execution.

MARISA-Opt reduces index size by splitting link offsets across the per-node 32-bit slot and a secondary array, storing only the low-order 8 bits in

the slot and packing the remaining bits compactly. The secondary array is allocated only when needed, avoiding unnecessary overhead. Token labels themselves remain in full in the per-node slot; only the link offsets are split. On CPU, MARISA-Opt does not narrow the per-node slot itself, the slot is still 32 bits, so at each link node 24 of those 32 bits are now unused padding.

For integer-token alphabets, MARISA-Opt further removes structures whose benefit is specific to byte-alphabet inputs (Appendix B). First, Patricia single-child link compression is disabled in the build path: every input token becomes its own LOUDS node, no link nodes are emitted, and the recursive trie-of-tails terminates after the first level regardless of the configured `num_tries`. As a consequence, `extras_`, the TAIL buffer, and the rank/select index over `link_flags_` are all empty in the on-disk index, and the link-offset split encoding described above is exercised only by character workloads. Second, the build-time prefetch cache table (`cache_`, used to amortize top-of-trie descents at lookup time) is skipped at construction and is absent from the on-disk index. Third, the in-process multikey quicksort over `Vector<Key>` is replaced by an external sort `-u -V` Unix pipeline that delivers unique, lexicographically ordered token sequences to the builder, so duplicate keys are removed before any trie work begins.

The combined effect is that on a 32-bit alphabet, where each TAIL token would still occupy 4 bytes and each TAIL reference would carry non-trivial `extras_` and `link_flags_` overhead, the bytes that Patricia compression would remove from `bases_` are smaller than the bytes its supporting structures would add. Disabling the compression therefore shrinks the on-disk index and shortens construction simultaneously; this is the dominant driver of MARISA-Opt’s index-size and build-time advantage over MARISA-Int in Section 4.2.

Unlike MARISA-Int, which performs repeated root-to-node traversals at each decoding step, MARISA-Opt propagates trie state across beams. Each extension requires a single LOUDS navigation step from the cached node, making per-candidate cost independent of prefix length.

Work is parallelized across beams using a fixed-size CPU worker pool, while per-beam expansion is performed using a linear-time two-pointer merge between sorted proposals and sorted trie children. This reduces matching cost from $O(K\Delta)$ to $O(K+\Delta)$ with sequential memory ac-

cess. Threshold-based pruning is applied during expansion to discard low-probability candidates early.

Instead of end-of-step sorting, MARISA-Opt maintains a shared bounded min-heap of size B , reducing pruning cost from $O(C \log C)$ to $O(C \log B)$, where C is the number of candidates generated per step and B is the beam width, while keeping the top- B beams continuously updated.

C.3 Axis-by-Axis System Contrast

D CPU Trie Operations

The CPU trie operations used by MARISA-Int and MARISA-Opt are inherited from the MARISA framework and are not contributions of this work; we describe them in prose for completeness so that the GPU adaptations in Appendix E have a precise reference point.

Child enumeration. Locating the children of a node v is two rank/select operations on `louds_`. The position of v 's first child within `louds_` is $\text{select}_0(v) + 1$, and the corresponding child node index is $\text{select}_0(v) + 1 - v - 1$ (subtracting the LOUDS header bits seen so far). The number of children Δ is the distance from that position to the next 0-bit; on CPU this is a sequential scan over `louds_`, and on GPU it is replaced by `WARPNEXTUNSET` (Appendix E). Labels of the Δ children sit contiguously at `bases_[base_node..base_node+ $\Delta$ )` in sorted order.

FINDCHILD. Given a parent v and a target token τ , MARISA does a binary search over the Δ child labels. The only subtlety is that when a child u is a link node (`link_flags_[u] = 1`), its `bases_` slot holds a TAIL offset rather than a token, so the binary-search comparison must peek into `tail_` at that offset to recover the first label of the suffix; the comparison itself is then ordinary integer comparison. This costs one extra memory indirection per visited link node and is the reason TAIL traversal must read at most one tail byte per binary-search step rather than the entire suffix. Per-call cost is $O(\log \Delta)$ comparisons, each $O(1)$ amortized.

LOOKUP and predictive search. A whole-key lookup is L successive FINDCHILD calls followed by a check that the resulting node carries a terminal flag; the key identifier is then $\text{rank}_1(\text{terminal_flags_}, v) - 1$. Common prefix search and predictive search extend this loop with

an additional in-order walk of the subtree rooted at the deepest matching node, paying $O(R)$ for R returned keys on top of the $O(L \log \Delta)$ descent. All of these operations execute serially on CPU and are the per-beam unit of work that FlashTrie's expansion kernel parallelises in Appendix E.

E GPU Implementation Details

FlashTrie replaces the serial FINDCHILD/LOOKUP loop of Appendix D with a cooperative-kernel pipeline that keeps the trie, the beam state, and the top- K output resident on the device for the entire query. This appendix documents the pieces that genuinely differ from the CPU baseline and are not adequately specified by prose alone: tail-suffix traversal, warp-cooperative LOUDS scanning, and large- B top- K selection. Bookkeeping aspects, memory residency, output serialization, and per-batch allocation, are described in prose since they involve no concurrency subtleties. Table 7 summarises per-operation time complexity for the GPU primitives used below alongside the CPU counterparts; the $1/P$ factor is the per-thread reduction the GPU contributes, with $P=512$ for per-CTA expansion and the full grid for sort passes.

Memory architecture. GPU memory is partitioned into three classes. The *static trie* (LOUDS bit-vectors, label arrays, tail buffer, rank/select indices) lives in CUDA Unified Memory and is prefetched on demand; this lets a single index back many concurrent search slots without per-slot duplication. *Per-batch search state*, three frontier arenas, a backtrace arena for parent pointers, and a beam-history buffer used during output reconstruction, is device-resident, allocated once at `init_tbs_gpu` time, and reused across requests so that no `cudaMalloc` appears on the hot path. *Per-request I/O* (input top- K tokens and log-probabilities, output beam sequences and normalised scores) is held in pinned host buffers paired with device mirrors, enabling asynchronous DMA that overlaps with kernel execution.

E.1 Tail Traversal

Beam expansion on the GPU runs in two passes per decoding step, both driven by the cooperative kernel in Algorithm 1. The first pass handles beams that are already *inside* a tail suffix (`s.in_link`): such a beam has exactly one valid continuation, the next token along the tail chain, so the work is to (i) read that token from `tail_` at `s.link_offset`, (ii)

Table 6: Design and implementation contrast across the three systems. All three share the LOUDS+tail/link trie skeleton (Appendix B); they differ in build-time storage choices and in where/how the beam search is executed. The *Improves* column tags whether each axis primarily affects search latency (**LAT**), build time (**BUILD**), or trie-on-disk size (**SIZE**). Per-operation time complexity is reported separately in Table 7. Background on Patricia single-child compression, the TAIL buffer, the link_flags_ bit-vector, and the prefetch cache_ table referenced in the rows below is given in Appendix B.

Aspect	MARISA-Int	MARISA-Opt	FlashTrie	Improves
bases_ slot type	Vector<UInt32>, 4 B/entry	Vector<UInt32>, 4 B/entry	Vector<UInt16>, 2 B/entry	SIZE
Label storage at non-link nodes	full 32-bit token in bases_	full 32-bit token in bases_; label not split	low 16 bits in bases_; high bits in extras_ via split_label	SIZE
Link-offset storage at link nodes	full pointer in bases_;	full pointer in bases_	low 16 bits in bases_; high bits packed in link_extras_;	SIZE
num_tries (trie-of-tails recursion depth)	1–7, default 1	1–7, default 1	1 only	BUILD SIZE
Recursive TAIL block (tail_)	built; tokens stored uncompressed as uint32, one offset entry per link node	empty (no link nodes produced, so the recursive build_next_trie returns immediately and tail_ is never populated)	built; tokens stored as uint32, references compressed via bit-packed link_extras_	BUILD SIZE
link_flags_ rank/select index	full bit-vector with rank/select over true link positions	identically zero across all nodes; rank/select index collapses to constant overhead	full bit-vector with rank/select over true link positions	SIZE
Build-time prefetch cache (cache_ table)	built; size set by cache_level, persisted to disk to amortize top-of-trie descents at lookup time	skipped (reserve_cache, per-node cache<T>, and fill_cache all bypassed)	built	BUILD SIZE
Build-time key sort and dedup	in-process multikey quicksort over Vector<Key>; duplicate inputs not removed	in-process multikey quicksort over Vector<Key>; deduplication done outside	in-process multikey quicksort over Vector<Key>; deduplication done outside	BUILD
Beam-search residency	host RAM	host RAM	GPU HBM (CUDA Unified Memory, eager-prefetched)	LAT
Trie cursor primitive in beam search	lookup then predictive_search from root; up to two full root-to-node walks per candidate, no state carried across steps	predictive_find_children: one LOUDS step from a cached node_id	extend_beam_binary_child_search: one GPU LOUDS step; state in device buffers	LAT
Parallelism axis	none (single thread serialises both beams and proposals)	across parent beams (proposals serial within a beam)	across parent beams and across proposals within a beam	LAT
Workers / dispatch unit	1 CPU thread	8 std::thread workers; one parent beam per pool job	$\lfloor 108/b \rfloor$ CTAs $\times$ 512 threads/CTA per slot	LAT
Per-beam inner loop	linear scan over K proposals; trie re-walked per candidate	two-pointer merge between sorted child labels and sorted top- K proposals	one thread per proposal; parallel binary search against the sorted child-label array	LAT
Top- B pruning structure	std::partial_sort over a flat std::vector of beam states	mutex-protected std::priority_queue bounded heap	on-device block-merge-sort; registers/shared memory	LAT
Batched-call mechanism	sequential per query	shared FIFO of (query, beam) jobs across 8 workers; step barrier	b cooperative kernels on b CUDA streams; disjoint SM partitions; no cross-slot sync	LAT

Table 7: Per-operation time complexity. L = key length, Q = query length, R = result count, Δ = node out-degree, T = decoding steps, B = beam width, K = top- K proposals per step, P = GPU parallelism, BS = Beam Search.

Operation	Time
rank ₁ (i)	$O(1)$
select _b (k)	$O(1)$
WARPNEXTUNSET	$O(\Delta/32)$
Lookup	$O(L \log \Delta)$
Common prefix search	$O(Q)$ total
Predictive search	$O(Q + R)$
MARISA-Opt BS	$O(T \cdot B \cdot (K + \Delta))$
FlashTrie BS	$O\left(T \frac{BK}{P} \log \Delta + T \frac{B \log B}{P}\right)$
MergeSort top- B	$O(B \log B/P)$

scan the top- K proposals for a match, (iii) apply token- and sentence-level log-prob thresholds, and (iv) emit the resulting beam either to selected (if the proposal closes the chain on a terminal node) or to next. Because the continuation is unique, no binary search is needed and each beam costs $O(K/P)$ work across the CTA’s P threads; this is EXTENDINLINK in Algorithm 2.

The second pass handles beams sitting on ordinary LOUDS nodes. For each such beam the kernel first computes the child range $[base_node, base_node + \Delta)$ using select₀ and WARPNEXTUNSET, then has each thread test one top- K proposal in parallel by binary-searching the sorted child labels. Each comparison transparently peeks into tail_ when the visited child is a link node, exactly as in CPU FINDCHILD. Surviv-

ing proposals become new beams that are routed to selected or next on the same terminal-/link-status rules as the first pass; this is EXTENDNODE in Algorithm 3. Splitting the work into two passes keeps the warps that are walking tail chains from interfering with the warps that are doing wider parallel child search, which keeps lane utilisation high in both modes.

Algorithm 2 ExtendInLink (device; one CTA per parent beam)

```

1:  $\tau_{\text{next}}, \text{end}$   $\leftarrow$ 
    $\text{tail.next\_token}(s.\text{link\_offset})$ 
2: for  $k = \text{threadIdx.x}$  to  $K - 1$  step
    $\text{blockDim.x}$  do
3:   if  $\text{topk\_id}[k] \neq \tau_{\text{next}}$  then
4:     continue
5:   end if
6:    $\rho \leftarrow \text{topk\_logp}[k]$ 
7:   if  $\rho \leq \theta_{\text{tok}}$  or  $s.\sigma + \rho \leq \theta_{\text{sent}}$  then
8:     break
9:   end if
10:   $s' \leftarrow \text{BEAMSTATE}(\text{parent}=s, \text{token}=\tau_{\text{next}})$ 
11:   $s'.\sigma \leftarrow s.\sigma + \rho$ 
12:   $s'.\text{link\_offset} \leftarrow s.\text{link\_offset} + 1;$ 
    $s'.\text{end\_link} \leftarrow \text{end}$ 
13:  if  $\text{end}$  then
14:    if  $\text{is\_terminal}(s'.v)$  then
15:       $\text{selected.push}(s')$ 
16:    end if
17:  else
18:     $\text{next.push}(s')$ 
19:  end if
20:  break
21: end for

```

E.2 Warp-Level LOUDS Navigation

The inner step of child enumeration, finding the next 0-bit in louds_ starting from a given position, can span many 64-bit words when a node has high out-degree, and a sequential per-thread scan leaves 31 of the 32 lanes of a warp idle. FlashTrie uses a warp-ballot loop in which each lane checks one of the next 32 words simultaneously and the warp votes via __ballot_sync; the position of the first word containing a 0-bit is recovered by ffs on the ballot mask, and the final intra-word bit position by another ffs on the negated word. Asymptotically this turns the $O(\Delta)$ inner loop into $O(\Delta/32)$ memory accesses. Algorithm 4 gives the exact word-level masking and ballot pattern; the lane

Algorithm 3 ExtendNode (device; one CTA per parent beam)

```

1: Compute  $\text{child}$   $\text{range}$ 
    $[base\_node, base\_node + deg)$  via LOUDS
    $\text{select}_0$  and WARPNEXTUNSET()
2: if  $deg = 0$  then
3:   return
4: end if
5: for  $k = \text{threadIdx.x}$  to  $K - 1$  step
    $\text{blockDim.x}$  do
6:    $\tau \leftarrow \text{topk\_id}[k]; \rho \leftarrow \text{topk\_logp}[k]$ 
7:   if  $\rho \leq \theta_{\text{tok}}$  or  $s.\sigma + \rho \leq \theta_{\text{sent}}$  then
8:     continue
9:   end if
10:  Binary-search sorted child labels in
    $[base\_node, base\_node + deg)$  for  $\tau$ ; peek
   into  $\text{tail}$  when the visited child is a link
   node
11:  if not found then
12:    continue
13:  end if
14:   $s' \leftarrow \text{BEAMSTATE}(\text{parent}=s, \text{token}=\tau);$ 
    $s'.\sigma \leftarrow s.\sigma + \rho; s'.v \leftarrow \text{matched child}$ 
15:  Route  $s'$  to selected or next per terminal-
   and link-status (same rules as Algorithm 2)
16: end for

```

arithmetic is what prose cannot specify precisely.

Algorithm 4 WarpNextUnset (device; 32-thread warp)

```

1:  $w \leftarrow \lfloor i/64 \rfloor; \text{word} \leftarrow B[w] \mid \sim(\sim 0 \ll$ 
    $(i \bmod 64))$ 
2: {mask bits below  $i$ }
3: if  $(\sim \text{word}) \neq 0$  then
4:   return  $w \cdot 64 + \text{ffs}(\sim \text{word}) - 1$ 
5: end if
6:  $w \leftarrow w + 1; \text{lane} \leftarrow \text{threadIdx.x} \bmod 32$ 
7: repeat
8:    $\text{ballot} \leftarrow \text{ballot}((\sim B[w + \text{lane}]) \neq 0)$ 
9:   if  $\text{ballot} = 0$  then
10:     $w \leftarrow w + 32$ 
11:   else
12:     $w \leftarrow w + \text{ffs}(\text{ballot}) - 1; \text{break}$ 
13:   end if
14: until false
15: return  $w \cdot 64 + \text{ffs}(\sim B[w]) - 1$ 

```

E.3 Top- K Selection on the GPU

After expansion the candidate buffer next is trimmed to the top- B hypotheses by length-normalized score. FlashTrie switches between two implementations depending on B .

Block merge sort ($B \leq 1024$). For small B the entire selection fits in a small number of sort-blocks of size $2B$. Each CTA sorts one block in descending order of $\hat{\sigma}$ using CUB’s BlockMergeSort, then the sorted blocks are combined by a tournament-merge tree in $\lceil \log_2(n/2B) \rceil$ rounds with a grid-wide barrier between rounds. After the final merge round the first B entries are the top- B beams. Total work is $O(n \log n / P)$ for the initial sort and $O(B \log(n/B) / P)$ for the merge rounds, with P the grid-wide thread count; see Algorithm 5.

Algorithm 5 BlockMergeSort Top- K (device; all blocks)

```

1:  $remaining \leftarrow \lceil |cand| / (2B) \rceil$ 
2: {number of sort-blocks}
3: while  $remaining \geq 1$  do
4:   {Phase 1: sort each block of  $2B$  elements}
5:   for all block  $b$  assigned to this CTA do
6:     Load  $2B/P$  elements per thread (pad with  $-\infty$  if OOB)
7:     BLOCKMERGESORT( $thread\_data$ ) (descending by  $\hat{\sigma}$ )
8:     write back to  $sort\_state$ 
9:   end for
10:  if  $remaining = 1$  then
11:    break
12:  end if
13:  sync( $\mathcal{G}$ )
14:  {Phase 2: tournament merge}
15:   $remaining \leftarrow \lceil (remaining - 1) / 2 \rceil + 1$ 
16:  sync( $\mathcal{G}$ )
17: end while
18: {Phase 3: collect top- $B$  entries into  $cand[0 \dots B - 1]$ }
```

Output serialization. The final selected buffer is descending-sorted by $\hat{\sigma}$ but the wire format expects ascending order, so GENERATERESULT writes the i -th selected beam into slot $n_o - 1 - i$ of the output arrays. Per-beam token sequences are reconstructed by walking the beam_history parent pointers in reverse, with a CUB BlockScan converting per-beam lengths into the end-offsets that index the flat tokens

buffer. The on-wire layout for n_o beams totalling n_{tok} tokens is an 8-byte num_beams header followed by three contiguous arrays: $8n_o$ bytes of logp_norm (float64), $8n_o$ bytes of sent_offset (uint64), and $4n_{tok}$ bytes of tokens (uint32); sequence i occupies tokens[sent_offset[$i-1$] : sent_offset[i]].

Per-batch GPU memory budget. A single search slot occupies roughly 320 MB of GPU memory, dominated by the persistent device-resident state: three frontier arrays at 56 MB each, a 56 MB beam history, six 8 MB sort-state arrays, a ~ 128 MB device-side output buffer paired with pinned host memory, and a ~ 12 MB input buffer similarly paired. Top- K scratch is small (3×0.5 MB) and the trie struct shell is a few hundred bytes; the trie arrays themselves live in Unified Memory and range from 50 to 500 MB depending on the constraint set, charged once across all slots rather than per-slot. A four-slot batch therefore uses ≈ 1.3 GB plus the trie, which fits comfortably alongside a generation-model weights footprint on a single 80 GB device.

E.4 Runtime Breakdown Methodology

Because beam search runs as a single persistent cooperative kernel (Section 2.2), we measure its internal phase breakdown with lightweight on-device instrumentation rather than separate kernel launches. Under a compile-time flag, a single grid-leader thread reads the GPU cycle counter (clock64) along its timeline. We time the wall-clock span of each phase region, bracketed by the grid barriers (grid.sync) that already separate the phases; because each region ends at a barrier, its span includes the slowest CTA, i.e. the phase’s critical-path time across the grid rather than the leader’s own work alone. Cycle counts are converted to milliseconds via the SM clock rate and averaged over the 13,000-request workload.

The four buckets map to kernel regions as follows. *Expansion* is the per-proposal binary search that locates a candidate token among a parent’s children, including the dependent LOUDS/tail memory accesses at each search step (a single tail read for link-interior beams). *Validation* is per-candidate admission: the per-token and cumulative log-probability threshold tests, the terminal-node check, and the append into the candidate buffer. *Pruning* is the top- B selection (parallel mergesort) that enforces the beam width. *Sync* is the residual:

inter-phase grid barriers outside the work regions, frontier setup/save, result serialization, and buffer swaps. The expansion and validation regions are interleaved per proposal, so we report their combined wall span split by the leader thread’s measured compute ratio; pruning and sync are measured directly. The four phases sum to total kernel time by construction.

We report *proportions* rather than absolute times: the instrumented build incurs mild register pressure that inflates absolute latency, so all latency numbers in Section 4.3 come from a separate, non-instrumented build. For reference, the measured mean per-phase times at $b=1$ range from 0.34/0.07/0.08/0.03 ms (expansion/validation/pruning/sync) at $K=100$ to 0.66/0.79/0.35/0.07 ms at $K=1000$, with total kernel time under 2 ms across the full sweep.

F Experimental Setup: Full Protocol

This appendix fills in the measurement details abbreviated in the main body’s Experimental Setup (Section 3).

Deployment envelope. All systems run in one Docker container on $1 \times A100$ 80 GB with an AMD EPYC 7V13 host and 60 GB host RAM, NUMA-pinned to cores 0–7 of socket 0 via `numactl -cpunodebind=0 -membind=0 taskset -c 0-7`. The latency window is a C++ entry-point timer covering H2D, kernel, and D2H with stream synchronisation.

Comparability of the timing window. The CPU systems’ timer closes when the search call returns; the FlashTrie timer closes after `cudaStreamSynchronize`. All three systems share the same C++ entry-point boundary and exclude only language-binding marshalling, so reported latencies are directly comparable. NUMA-pinning to a single socket also removes cross-socket traffic on the H2D path, so FlashTrie’s speedup is reported under the same deployment-realistic envelope as the CPU baselines.

Length-normalised scoring. Beams are ranked by the Google-style length-normalised log-probability used in Section 2 (Wu et al., 2016):

$$s_{\text{norm}}(\mathbf{x}_{0:d}) = \left(\sum_{j=0}^{d-1} \ell_{j, \sigma(j)} \right) \cdot \left(\frac{6}{5+d} \right)^\alpha, \quad (2)$$

where α controls the normalisation strength and $\sigma(j)$ indexes the selected candidate at step j .

Per-request latency sampling. Each (K, b) configuration is benchmarked over the full 13,000-request dataset at batch size 1. One warm-up pass is discarded; FlashTrie additionally runs a short GPU warm-up before the measurement window opens. The next 10 passes are timed and pooled, producing 130,000 samples per configuration from which the mean and the 50th/90th/95th/99th percentiles are computed. The across-run standard deviation is reported as a variance bar in figures and as the $\pm$ value in Table 9.

Throughput sweep. For the throughput evaluation (Figure 4, Section 4) we sweep $b \in \{1, 4, 8\}$ at $K \in \{100, 300, 500, 700, 1000\}$. For each (K, b) the dataset is run as $\lceil 13000/b \rceil$ batched calls per pass, and per-pass throughput is total queries divided by total wall-clock duration measured with the same C++ entry-point timer window used for latency. One warm-up pass is discarded and four passes are timed; reported Θ is the mean over those four passes, with the across-pass standard deviation drawn as the error band in Figure 4.

Precision@K scoring. For the retrieval-quality evaluation (Section 4.6) each request’s top- K trie-admitted output is scored by a transformer teacher model adapted from (Mohankumar et al., 2023) that takes the request and a candidate keyword sequence as input and returns a single relevance score. The teacher is identical across all three systems (MARISA-Opt, FlashTrie, and PPT) so that any precision differences are attributable to the candidate set, not to the scorer. Precision@ $K=100$ and Precision@ $K=200$ are reported as the macro-average over the 13,000 requests.

G Unconstrained Decoding with Post-Hoc Filtering

This appendix gives the protocol and the length breakdown behind the baseline of Section 4.1.

Protocol

The unconstrained path reuses the proposal grid of Section 3 unchanged: the same 13,000 requests, $T=8$ decoding positions, and a 2.2M-token vocabulary. It runs the identical beam search with the trie-validity test removed, so a beam may extend by any of its top- K proposals, while scoring, length normalisation and the log-prob thresholds stay fixed. Each decoded sequence is then looked up in the full 800M-identifier inventory, and a candidate counts

Table 8: Invalid rate of unconstrained NAR decoding by decoded identifier length, over the same 13,000-request set.

Identifier length	2	3	4	5–8
Invalid (%)	71	94	98	99–100

as valid only on an exact full-sequence match, the same criterion the trie enforces incrementally. *Valid* counts matching rows, *Distinct* counts unique identifiers after collapsing duplicates, and the invalid rate is one minus valid rows over K . All figures are per-query means over the request set.

Two properties of the comparison are worth making explicit. First, constrained traversal cannot emit a duplicate: a root-to-terminal trie path corresponds to exactly one identifier, so its valid and distinct counts coincide and both equal the beam width. Duplication is a cost borne only by the unconstrained path. Second, the unconstrained path is not cheaper per step in any way that offsets its shortfall. It performs the same expansion and the same top- B selection and merely omits the validity test, so the lower yield is not purchased with lower decode cost.

Invalid Rate by Identifier Length

Table 8 shows that validity decays steeply with identifier length. This is what per-position independence predicts: NAR positions are produced without conditioning on one another, and a candidate is valid only if all ℓ positions simultaneously agree on a single inventory entry, so the probability of validity falls off geometrically in ℓ .

The consequence for retrieval matters more than the rate itself. Because the surviving identifiers concentrate at lengths 2–3 while the inventory is dominated by longer sequences, post-hoc filtering does not simply return fewer identifiers, it returns a systematically biased subset of them. The shortfall in Table 1 therefore understates the loss in inventory coverage, and the bias cannot be removed by widening the beam, since a wider beam samples more densely from the same length-skewed distribution.

H Per-Percentile Latency Table

Table 9 gives the full per-percentile breakdown of the per-request trie-search latency reported in Figure 3: mean, p50, p90, p95 and p99 in milliseconds, for every beam width K , computed as the median

across 10 runs of 13,000 queries each.

I Online A/B Test Protocol

This appendix gives the operational protocol of the production A/B test summarised in Section 4.9.

Sample-ratio-mismatch (SRM) check. A sample-ratio-mismatch check passed throughout the flight, confirming the realised treatment:control event ratio is consistent with the configured allocation after eligibility filtering. This rules out exposure-bias confounds before reading the metric lifts.

Latency-timing boundary. End-to-end latency is measured from request ingress at the retrieval service to emission of the final top- k candidate set, inclusive of trie traversal, beam expansion, scoring, and post-processing. Control and treatment share the entire ads downstream stack; they differ only in the retrieval-stage trie back-end and the beam configuration the back-end can sustain within the 30 ms end-to-end latency SLA.

Constraint library and back-end configurations.

The production constraint library indexes hundreds of millions of docIDs. The control arm runs production CPU MARISA-Opt at $B_{\text{ctrl}}=200$, top- $k=300$, with a 3-thread pool matched to the per-worker MIG-slice CPU-core allocation.

Metric definitions. Columns of Table 4: *Revenue* is total advertiser revenue for the arm; *Coverage* is the fraction of queries for which a sponsored ad is shown; *Impression* is Impression Yield (IY), the expected value per impression opportunity; *Clicks* is total ad clicks; *Latency* is the end-to-end window defined above. The defect guardrails reported as nonsignificant in Section 4.9 are the Defect-ML classifier and Feedback-Based-Suppression (FBS) signals on Ads Selection.

J MARISA-Int Feasibility Measurements

MARISA-Int is included only as a feasibility check (not a primary performance baseline). At billion-key scale it is too slow for a full 13K sweep, so we report a fixed 10-query subsample (seed 42) and omit $K \geq 500$ after observing a single $K=1000$ query at 10,997 s (≈ 3.05 h).

Table 10 reports mean/p50/p90/p95 on the subsample. p99 is omitted at $N=10$. The large mean-median gap reflects high per-query variation in traversal depth and root branching.

Table 9: Per-request trie-search latency (ms), full percentile breakdown. Each cell is the median across 10 runs of 13,000 queries each. **Lower is better.**

K	MARISA-Opt (CPU)					FlashTrie (GPU)				
	Mean	p50	p90	p95	p99	Mean	p50	p90	p95	p99
100	9.0	8.9	11.8	13.0	15.1	0.6	0.5	0.6	0.6	0.7
200	14.9	14.4	19.2	20.9	23.6	0.7	0.7	0.8	0.9	0.9
300	20.1	19.8	25.2	26.9	30.0	0.9	0.9	1.1	1.1	1.2
400	25.3	25.1	32.2	34.2	38.3	1.1	1.1	1.4	1.5	1.6
500	29.5	29.3	38.2	40.6	45.4	1.4	1.4	1.7	1.8	2.0
600	33.4	33.0	44.2	47.4	53.9	1.6	1.5	2.0	2.2	2.6
700	36.3	35.9	48.8	52.6	60.4	1.6	1.6	2.2	2.4	2.8
800	40.5	38.4	52.7	57.0	66.5	1.9	1.8	2.5	2.7	3.2
900	42.9	40.6	55.9	60.5	70.5	1.9	1.9	2.5	2.8	3.3
1000	46.3	43.8	60.6	65.5	76.7	1.9	1.8	2.5	2.8	3.3

Table 10: MARISA-Int per-request latency on a fixed 10-query random subsample of the 13,000-request workload (seed 42, identical query indices across K).

K	Latency (s)			
	Mean	p50	p90	p95
100	394.5	37.6	1,049.7	1,809.4
200	764.6	492.0	2,392.3	2,597.4
300	1,498.9	947.7	4,333.6	5,134.4
400	2,028.9	1,329.3	4,789.9	6,391.6

Latency grows roughly linearly with K , consistent with MARISA-Int’s serial per-beam execution model, and confirms it is unsuitable as a main comparator at this operating point (hence MARISA-Opt in Section 2.1).

K Ablations: Full Details

This appendix gives the construction, full per- K measurements, and analysis for the three GPU ablations of Section 4.7, together with two additional sequential variants that were measured but failed to scale. Together with the main-text comparison against MARISA-Opt, these results decompose the FlashTrie speedup into three contributions: GPU residency and kernel fusion (PPT vs. MARISA-Opt), the LOUDS-trie structure for path validity (FlashTrie vs. PPT and vs. the hash map), and binary search at the inner per-key loop (FlashTrie vs. the per-key linear probe).

Shared protocol. Every variant below runs in a cooperative kernel that is otherwise a clone of the FlashTrie kernel: the same grid-wide barrier per decoding step, the same 512-thread CTAs with one CTA per parent beam, the same proposal-parallel inner loop, and the same log-prob thresholds, length normalisation, append-only candidate

buffer and on-device top- B selection. *Only the constraint lookup differs.* All variants are measured on the 13,000-request workload and the hardware of Appendix F at $b=1$ with beam width = K , using a 20-iteration GPU warm-up and one discarded warm-up pass followed by timed passes; percentiles are computed over the per-request wall-clock timings and averaged across passes. PPT and the per-key linear probe were measured at all ten beam widths, the hash map at every other K .

K.1 Per-Position Token (PPT)

For each decoding depth $d \in [0, T)$, we collect the set of all token IDs that appear at position d in any keyword of the constraint library and store them as a sorted, deduplicated array on the GPU. A flat (valid_tokens, valid_offsets, valid_counts) layout is used so that the depth- d alphabet is contiguous in device memory. At search time, for each beam at depth d , every top- K proposal token is tested against valid_tokens[d] via a single lower_bound binary search; tokens that pass are admitted into the next beam, tokens that fail are pruned.

Formally, PPT admits a token $x_{t,i}$ at depth d iff $x_{t,i}$ appears at position d in some keyword; it does not require that the prefix $x_{0:d}$ extended by $x_{t,i}$ corresponds to a valid path in the constraint library. The candidate set produced by PPT is therefore a strict superset of the trie-valid set, and PPT performs strictly less work per step than FlashTrie. It is consequently not constraint-equivalent, and cannot answer whether a succinct trie is needed once the constraint index is GPU-resident.

K.2 Hash-Map Constraint Index

To answer that sharper question we replace the trie with an *exact* hash map over trie transitions and hold everything else fixed.

Index construction. Every transition of the LOUDS trie is materialised as one hash-map entry, keyed by the (state, token) pair that it consumes:

$$\text{key}(s, x) = ((s \ll 32) \mid x) + 1, \quad (3)$$

where x is a 32-bit token id and s is a 32-bit state. A state is either a LOUDS node id or, for a position interior to a tail link, the link offset tagged with a high bit ($s = 0 \times 80000000 \mid \text{offset}$), so a single map covers both the node-to-node and the in-link transitions that the trie kernel reaches through two separate code paths. The $+1$ keeps the key non-zero without disturbing the tag bit. Each entry stores the 64-bit key, the transition payload the kernel needs (child node id, link offset, and the `is_link`, `end_link` and `is_terminal` flags), and a chain successor index: 32 bytes in total, laid out as an array of structures in global memory.

Collisions are resolved by separate chaining, the layout of `std::unordered_map`: an array of bucket heads, each an index into a flat entry pool linked by a `next` field. The bucket count is the entry count rounded up to the next odd integer, giving a load factor of 1.0 and forcing a true integer modulo rather than a bit mask, and the hash is the identity modulo that bucket count. Because full 64-bit keys are stored and compared, the map is exact: it has no false positives and admits precisely the trie-valid candidate set, so the ablation is constraint-equivalent to FlashTrie by construction. The map is built on the host in a single trie traversal, serialised to a flat file, and uploaded to the device with one memory copy per array; there is no device-side build.

Kernel integration. Where FlashTrie binary searches the parent’s sorted child labels in $\mathcal{O}(\log \Delta)$, the hash variant computes one modulo and walks the corresponding bucket chain. One probe is issued per (beam, proposal) pair, so a decoding step issues $B \times K$ probes, exactly the number of binary searches the trie kernel performs.

Memory footprint. A hash map stores one entry per transition and therefore cannot share structure between sibling or ancestor edges the way a trie does. The LOUDS bit-vector encodes the shape of the trie implicitly, so a node costs a handful of bits beyond its label, whereas the map must spell out every (state, token) pair explicitly and additionally pay for a bucket-head array and a per-entry chain pointer. On the 800M-keyword

library the map holds 926,855,329 transitions over 790,943,908 trie nodes, which at 32 bytes per entry is 27.6 GiB of entry pool plus 3.5 GiB of bucket heads: ≈ 31 GB resident, against 3.1 GB for the Narrow-LOUDS index at the same scale (Section 4.2), a factor of 10.0. Beyond the deployment consequence noted in Section 4.7, the map would demand a full A100 80 GB, where it would still occupy $\approx 40\%$ of HBM before any model weights or per-slot working memory (Appendix E).

K.3 Linear-Search Variants

Within the trie kernel itself, the per-parent child lookup is performed by a binary search over the parent’s sorted child labels. To isolate the contribution of this algorithmic choice, as distinct from GPU residency, kernel fusion, and the LOUDS layout, we replaced the binary-search inner loop with three successively weaker variants. They differ only in how each parent beam locates the children whose labels appear in the top- K proposal set:

- **Two-pointer merge (vanilla MARISA port).** Faithful single-thread port of the CPU MARISA child-extension loop: one CTA per parent, one thread per CTA walks the parent’s sorted children and the sorted top- K in lockstep, $\mathcal{O}(\text{degree} + K)$ per parent. The remaining 511 threads of the CTA idle. Across-parent parallelism is preserved (one CTA per parent, many CTAs per SM), matching the CPU pattern of multi-threading across parents.
- **Per-child parallel linear scan.** Block-parallel baseline that distributes children across the CTA’s threads: thread t handles children $t, t + \text{blockDim}, \dots$ and for each assigned child performs an $\mathcal{O}(K)$ linear scan through the top- K array to test membership. Per-parent work $\mathcal{O}(\text{degree} \cdot K / \text{blockDim})$ in wall-clock, but every thread re-issues trie metadata loads (`select0`, `rank1`, label decode) for its assigned children with no sharing.
- **Per-key linear probe (FlashTrie inner loop with binary $\rightarrow$ linear).** Identical parallelisation to FlashTrie, one thread per top- K key, striding `blockDim`, but each thread linearly scans the parent’s sorted children for its assigned key (with sorted-array early exit) instead of binary searching. To make this

Table 11: Full per- K sweep for PPT and the per-key linear probe (companion to Table 3). Speedup is ablation p95 divided by FlashTrie p95 at the same K . **Lower latency is better.**

K	PPT			Linear-probe		
	Mean	p95	Sp.	Mean	p95	Sp.
100	0.57	0.79	1.23×	110.2	134.1	209×
200	1.16	1.61	1.85×	119.6	144.2	166×
300	1.90	2.69	2.40×	123.9	149.1	133×
400	2.73	3.97	2.71×	126.8	152.8	104×
500	3.54	5.20	2.84×	128.6	155.7	85×
600	5.57	8.22	3.69×	135.5	169.3	76×
700	6.98	10.30	4.37×	141.1	178.8	76×
800	8.22	12.15	4.45×	145.6	186.3	68×
900	9.73	14.36	5.21×	151.2	194.3	71×
1000	11.28	16.65	5.98×	154.3	198.7	71×

competitive at all, the CTA cooperatively decodes each parent’s child labels once into a shared-memory cache, after which all 512 threads scan the cache. Per-parent work $O(\text{degree} + K \cdot \text{degree}/\text{blockDim})$ in arithmetic, with all global-memory trie loads amortised across the topk loop.

Both fully sequential variants fail to scale to the full benchmark at production beam widths, and are reported only as a cautionary data point for naive GPU ports of the textbook MARISA traversal. The two-pointer merge, although algorithmically the lowest in arithmetic complexity per parent, uses only 1/512 of each CTA’s threads; the warm-up pass alone takes several minutes at $K=100$. The per-child parallel scan keeps every thread busy but pays with redundant per-thread re-decoding of child labels and an $O(K)$ inner scan per child, and a single 13K-query pass at $K=100$ failed to complete within seven hours. Only the per-key linear probe admits a meaningful comparison, since it shares FlashTrie’s parallelisation scheme exactly and differs only in whether the inner per-key search over the cached children is binary or linear. Its full per- K sweep is given alongside PPT in Table 11.

K.4 Results

Table 11 gives the complete per- K sweep for PPT and the inner-loop linear probe. Hash-map mean and p95 values are already reported in Table 3; Table 12 provides its full percentile and throughput breakdown. Run-to-run variation for the hash variant is small: the standard deviation of the mean is ≤ 0.012 ms and of p95 is ≤ 0.032 ms at every K .

Table 12: Hash-map ablation, full percentile breakdown (ms) and throughput on the 13,000-request workload, $b=1$. Compare with the FlashTrie columns of Table 9. **Lower is better** for latency.

K	Mean	p50	p90	p95	p99	QPS
100	1.74	1.64	1.93	2.21	3.33	575.4
300	3.14	2.79	4.22	5.15	8.30	318.1
500	4.44	3.96	6.28	7.69	12.35	225.4
700	5.39	4.75	7.81	9.54	14.45	185.6
900	7.07	5.61	9.02	11.02	16.26	141.4
1000	7.59	5.86	9.44	11.51	16.90	131.8

K.5 Discussion

Why constant-time lookup loses on a GPU. Despite an expected $O(1)$ lookup, and despite a load factor of 1.0 that makes the average chain a single hop, the hash variant is $3.1\times\text{--}4.0\times$ slower than FlashTrie in mean and $4.8\times\text{--}5.1\times$ slower at p99, and its throughput falls from 575 to 132 queries per second across the sweep. Three properties of the GPU memory system account for the gap. First, locality: the threads of a warp probe consecutive proposal tokens, whose keys differ only in the low 32 bits and therefore land in unrelated buckets, so one warp’s probes touch as many distinct cache lines as it has active threads. FlashTrie’s binary search instead walks a contiguous, sorted child range, and successive steps of the search hit lines that are already resident. Second, dependent loads: a bucket head is an index rather than a value, so even a one-hop lookup costs two serialised global accesses whose latency cannot be hidden when every thread misses simultaneously. Third, footprint: at ≈ 31 GB the entry pool exceeds the cache hierarchy by orders of magnitude, so nearly every probe reaches DRAM, whereas the 3.1 GB narrow-LOUDS arrays keep the upper trie levels, which every beam traverses at every query, resident in L2. The memory and latency disadvantages are thus two consequences of one cause: the map stores transitions individually instead of encoding them structurally.

This ablation therefore separates the value of the succinct layout from the value of GPU residency more cleanly than PPT does. The hash variant is GPU-resident, kernel-fused, constraint-exact, and asymptotically cheaper per lookup, and it still loses by roughly $4\times$ in latency and an order of magnitude in memory. On a GPU, constant-time lookup is the wrong objective; coalesced access over a bounded footprint is the right one.

Why the hash map overtakes PPT at large K .

The hash variant is slower than PPT below $K=700$ and faster above it (Table 3), which is expected rather than anomalous: PPT has the cheaper lookup of the two but is permissive, so it forwards a strict superset of the trie-valid candidates, and the validation and pruning stages that consume those candidates dominate runtime at large beam widths (Section 4.4). Past $K \approx 700$ the cost of PPT’s larger candidate set outweighs its cheaper per-token test. The two ablations are therefore not ordered by lookup cost alone, and only the hash variant is constraint-equivalent to FlashTrie.

Why the inner search algorithm matters. At $K=100$ the per-key linear probe has a mean per-request latency of 110.2ms against FlashTrie’s 0.55ms, indicating that binary search at the inner loop contributes a two-orders-of-magnitude factor on top of the GPU residency, kernel fusion, and shared-memory caching that it shares with FlashTrie. The two discarded variants illustrate a common failure mode of porting CPU succinct-trie code to the GPU. The CPU MARISA loop is fast because modern x86 cores are individually fast and MARISA-Opt multi-threads across parents. Both properties evaporate on the GPU: an SM’s individual thread is far weaker than a CPU core, and the across-parents axis alone is not wide enough to amortise the trie metadata cost when each parent’s inner loop is sequential. Only by holding the parallelisation scheme fixed and varying the inner-loop algorithm can the contribution of the search algorithm itself be measured cleanly, and that contribution is large.

L NQ + GENRE Workload Construction and Realistic-Threshold Analysis

The three components introduced in Section 4.8 are pinned for exact reproduction: NQ-Open validation (**3,600 questions**, free-form input matching serving shape), the public GENRE-KILT checkpoint (facebook/genre-kilt, BART-large with a $\sim 50k$ token vocabulary, ungated), and the deduplicated KILT Wikipedia titles (**$\sim 6M$ tokenized titles**, BART-tokenized so the model’s emitted token IDs and the trie’s edge labels share one ID space, avoiding train/serve vocabulary mismatch).

GENRE is autoregressive: a vanilla top- K beam roll-out produces a proposal grid where each position’s top- K is conditioned on the previous-step argmax, so the grid is internally coherent and the

trie does little admissibility filtering. To simulate the cross-position incoherence of production NAR grids, we use a two-stage pipeline.

Stage 1: AR top- K extraction. For each NQ-Open query we run GENRE over $T=16$ decoding positions, emitting at each position t the top- $K=600$ tokens with their log-softmax scores. The result is a per-query (T, K) grid of token IDs and log-probs, the same data shape FlashTrie and MARISA-Opt consume on internal traffic.

Stage 2: head + noise-tail replacement. For a chosen head size $r \in \{0, 9, 18, 150, 300, 600\}$ we keep the top- r AR-emitted tokens at each position (the “coherent head”) and replace ranks $[r, K)$ with tokens sampled from the position-stratified marginal distribution of the trie corpus (the empirical frequency of each token at position t across all tokenized Wikipedia titles). Noise tokens inherit the AR model’s tail log-probs at the occupied ranks so the resulting grid is indistinguishable from a real model emission to a downstream beam scorer. The knob r interpolates between $r=0$ (pure NAR-style proxy, maximum trie work) and $r=K$ (pure AR, minimum trie work).

Alternative noise designs (uniform-vocab, uniform-position alphabet, and history-conditioned sampling) were rejected because they either under-stress the trie or reintroduce AR coherence. Position-stratified marginal sampling preserves per-position plausibility while remaining cross-position independent. We keep AR tail log-probs to preserve realistic score decay under θ_{tok} , θ_{sent} , and length normalization.

All experiments use the deployment envelope of Appendix F. The constraint trie is built once over the tokenized KILT corpus and reused across all r . For the r -sweep, beam-score thresholds are set permissively ($\theta_{\text{tok}}=-20$, $\theta_{\text{sent}}=-200$, length-norm exponent 5.0) so that the trie’s admissibility check, not the scorer, drives all pruning. Results are reported at $K=600$, which sits inside the beam-width range of the internal sweep (Figure 3) and has the densest r -grid in the released artifacts. The latency measurement protocol (Section 3) is reused verbatim.

GPU Overhead Floor

The GPU curve flattening in Table 13 is explained by a structural overhead floor (kernel launch, grid barriers, fixed top- B selection passes, and fixed-size H2D/D2H I/O). CPU has less fixed overhead, so its latency keeps scaling with reduced trie work

Table 13: Mean trie-search latency (ms) at $K=600$ on NQ+GENRE vs. head size r ($r=0$: NAR-style proxy, max trie work; $r=600$: GENRE verbatim). Speedup = CPU/GPU mean. **Lower latency, higher speedup are better.**

r	0	9	18	150	300	600
MARISA-Opt (ms)	41.2	40.5	40.2	37.6	33.7	20.4
FlashTrie (ms)	5.7	5.2	3.6	2.8	2.7	2.4
Speedup (CPU/GPU)	7.3	7.8	11.1	13.3	12.5	8.4

while GPU saturates once work drops below this floor.

Realistic-Threshold Operating Point

Table 13 uses permissive thresholds to stress trie work. Here we evaluate realistic production thresholds (Section 4.9), where scorer-side pruning is stronger under the same 30 ms SLA.

We keep all other NQ+GENRE settings fixed (3,600 queries, $K=600$, $T=16$, length norm 5.0, identical warmup) and use the deployed threshold pair. Table 14 reports mean latency and CPU/GPU speedup across the r sweep.

At realistic thresholds and worst-case $r=0$, MARISA-Opt drops from 41.19 ms (permissive) to 8.95 ms mean and FlashTrie from 5.66 ms to 1.54 ms, showing scorer-side pruning reduces absolute work for both systems. Relative advantage remains multi- $\times$: $5.8\times$ at $r=0$, typically $7.4\text{--}7.6\times$ for $r \in \{9, 150, 300\}$ (Table 14).

Across the r sweep, CPU latency still tracks workload coherence, while GPU latency stays in a narrow band because fixed cooperative-kernel overhead dominates once useful trie work is small. This preserves FlashTrie’s core value at production thresholds: lower and more stable latency under workload variance.

Retrieval Quality on NQ+GENRE

Sections 4.8 and the sweeps above establish the latency claim on public artifacts; here we check that constraint semantics are also preserved. Following KILT convention we report R@100 rather than P@ k , because NQ-Open typically has a single gold title per query, which makes precision at a cutoff dominated by the cutoff itself rather than by retrieval behaviour.

MARISA-Opt and FlashTrie return *identical* results at every beam width (Table 15). This is the public-artifact counterpart of the internal agreement reported in Section 4.6 (Table 2), and it states

Table 14: Per-request mean trie-search latency (ms) at $K=600$ under realistic production thresholds on the NQ + GENRE public workload, across the head-size sweep $r \in \{0, 9, 18, 150, 300, 600\}$. MARISA-Opt is the CPU baseline (8 worker threads, NUMA-local pinning); FlashTrie is the GPU implementation on a single A100 80 GB. The **Speedup** column reports CPU mean / GPU mean per row. Compared to the permissive-threshold sweep in Table 13, score-driven pruning ahead of the trie reduces CPU mean latency by $\approx 5\times$ and GPU mean by $\approx 2\text{--}4\times$, while the relative CPU-to-GPU mean speedup remains multi- $\times$ throughout ($3\text{--}8\times$). **Lower is better for latency; higher is better for speedup.**

r	MARISA-Opt	FlashTrie	Speedup
0	8.95	1.54	$5.8\times$
9	8.61	1.13	$7.6\times$
18	8.45	2.83	$3.0\times$
150	7.22	0.95	$7.6\times$
300	6.78	0.91	$7.4\times$
600	6.70	1.40	$4.8\times$

Table 15: Retrieval quality (R@100) on the NQ+GENRE public workload (3,600 NQ-Open validation queries, $\sim 6\text{M}$ -title KILT trie) vs. beam width K . The two systems agree exactly at every K , confirming that GPU execution is constraint-equivalent to the CPU baseline. **Higher is better.**

K	100	600	1000
MARISA-Opt (R@100)	0.5369	0.5805	0.6194
FlashTrie (R@100)	0.5369	0.5805	0.6194

the intended property directly: FlashTrie changes the cost of constrained beam search, not its semantics. Quality improves monotonically with beam width, from 0.5369 at $K=100$ to 0.5805 at $K=600$ and 0.6194 at $K=1000$, an absolute gain of 0.083 ($+15\%$ relative) across the sweep, matching the mechanism described in Section 4.6.

The improvement is smaller than at production scale, as expected: the $\sim 6\text{M}$ -title library and $\sim 50\text{k}$ BART vocabulary give far lower branching than the 800M-keyword, 2.2M-token production setting, so fewer plausible prefixes are lost to early pruning and there is correspondingly less search error for a wider beam to recover. The public workload therefore validates the latency and constraint-equivalence claims directly, while the magnitude of the quality gain remains scale-dependent.