An Incremental Valiant Recognizer for GPU-accelerated General Context-free Prefix Validation

Anonymous ACL submission

Abstract

Constrained decoding systems are often built on context-free parsers intended for programming languages. These parsers either degrade into $O(n^3)$ time-complexity or fail entirely if the grammar is not carefully engineered to keep properties such as determinism and non-ambiguity. There is thus a need to design parsers that efficiently handle non-determinism and ambiguity, while simultaneously being incremental so that they can be coupled with the token-based predictions of large language models. Inspired by prior work, we derive an incremental Valiant prefix recognizer, which still has $O(n^3)$ complexity but allows for acceleration with only a fraction of GPU resources (stream multiprocessors). Our parser shows robust efficiency in complex context-free grammars while other parsers crash or degrade. At the same time, we remain empirically competitive in restrictive grammar classes such as LALR.

1 Introduction

Constrained decoding systems typically face a trade-off when selecting a context-free parser. One option is to adopt parsers from the LL or LR families, which consistently achieve $O(n)$ complexity but require users to invest substantial engineering effort in crafting restrictive grammars. Alternatively, systems may employ more general parsers such as Earley, which can achieve $O(n)$ complexity when the grammar happens to be unambiguous and deterministic, but degrade to $O(n^3)$ otherwise.

This limitation poses fewer problems in the traditional use case of parsing programming languages, as such languages are typically designed with these parsing restrictions in mind and have update cycles spanning at least a year, allowing ample time for careful grammar engineering. However, LLM systems iterate at a much faster pace, and researchers have increasingly sought to constrain diverse outputs including reasoning traces, molecular structures, and privacy-sensitive contents (Banerjee et al., 2025; Loula et al., 2025; Ugare et al., 2025). This calls for stronger general-case performance.

The Valiant recognizer (Valiant, 1975) achieves sub-cubic general context-free recognition by connecting CYK parsers with Boolean matrix multiplication. Although a practical sub-cubic algorithm has yet to emerge, GPUs can provide strong acceleration. Azimov and Grigorev (2018) demonstrates a GPU-accelerated Valiant recognizer for graph database query that outperforms GLL, a generalized top-down parsing algorithm.

However, the Valiant recognizer cannot be applied to constrained decoding, as it lacks the valid-prefix property—the ability to parse incrementally and reject invalid continuations as soon as the first violating element is encountered. To address this limitation, our technical contributions include:

- Deriving a Valiant valid-prefix recognizer.
- Demonstrating that incremental parsing substantially reduces memory footprint and improves CUDA L1/SMEM access patterns.
- Implementing a constrained decoder that empirically performs competitively in restricted grammars and outperforms prior work significantly in complex grammars.

2 Related Work

There has been multiple work on improving the efficiency of constrained decoders. A large fraction of these works focuses on the alignment between LLM tokens and grammar terminals (Dong et al., 2025; Moskal et al., 2025; Ugare et al., 2024; Beurer-Kellner et al., 2024; Anonymous, 2025). Work on improving the internals of existing parsers can also be found (Sun et al., 2025).

General context-free parsing can be achieved by extending the traditional LL and LR parsers (Scott and Johnstone, 2010; Tomita, 1985). Alternatively,

chart-based parsers such as CYK (Cocke, 1969; Younger, 1967; Kasami, 1965) and Earley (Earley, 1970) can be used. Unlike LL and LR parsers, general context-free parsers can be extended to mildly-context-sensitive parsers, which is relevant to efforts in the constrained decoding community to cover more complex constraints (Ugare et al., 2025; Nagy et al., 2025).

Parallelization of chart-based parsers is inspired by the connection between the CYK parser and matrix multiplication (Valiant, 1975). Bernardy and Claessen (2015) shows that a sparsely encoded chart can reduce complexity from $O(n^3)$ to $O(log^2 n)$ in practical cases. A practical implementation on GPUs is used to find context-free paths in a graph database (Azimov and Grigorev, 2018).

3 Background

3.1 Constrained Decoding

Large language models produce a probability distribution over their vocabulary to express preferences for different output tokens across different time steps. Constrained decoders seek to modify this distribution to make the eventual output text conform to a specification, often expressed by a formal grammar, such as a context-free grammar.

3.2 Context-free Grammar

Given a finite set alphabet Σ, a context-free grammar defines a language $L \subseteq \Sigma^*$ using a quadruple $G = (N, \Sigma, P, S)$ where:

- Σ is a finite set of terminal symbols, which we will write in lower case,
- N is a finite set of nonterminal (variable) symbols, which we will write in upper case,
- $S \in N$ is the start symbol, and
- $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of production rules, written $A \to \alpha$ with $A \in N$ and $\alpha \in (N \cup \Sigma)^*$.

We use $A \Rightarrow^* \alpha$ to denote that symbol sequence α is derivable from non-terminal A by repeated application of production rules.

3.3 Chomsky Normal Form

Each context-free grammar can be converted into equivalent Chomsky Normal Form (CNF; Chomsky, 1959), where the right side of each production rule either has two non-terminals or one terminal, and only the start symbol is allowed to generate the empty string ϵ. This conversion is also called binarization, an essential step for CYK-like parsers.

Algorithm 1 CYK Algorithmn, row-by-row

Input: Terminal sequence $t_0 \dots t_{n-1}$

Base case: row $= 0$
$TABLE$[row][col][V] $\Leftrightarrow$
  $\exists$ rule $V \to t_{\text{col}}$ in CNF grammar
Recursion: row $= 1 \dots n-1$
$TABLE$[row][col][V] $\Leftrightarrow$
  $\exists$ rule $V \to AB$ in CNF grammar, s.t.
    $\exists$ mid $\in \mathbb{N}$, s.t. mid $<$ row and
      $TABLE$[mid][col][A] and
      $TABLE$[row $-$ mid $-$ 1]
        [col $+$ mid $+$ 1][B]

4 The Incremental Valiant Prefix Recognizer

We describe the well known CYK/Valiant recognizer for context free grammars in section 4.1, modify it to run incrementally in section 4.2, extend it to recognize valid prefixes in section 4.3, and further improve performance in section 4.4.

4.1 Canonical CYK/Valiant

The CYK parser is a bottom-up, dynamic programming parser, as defined in algorithm 1. It holds the invariant that V is at row and col (i.e., $TABLE$[row][col][V] $= true$) if and only if $V \Rightarrow^* t_{col} \dots t_{col+row}$.

The Valiant Recognizer re-frames the same algorithm as boolean matrix operations. We use the two names interchangeably.

4.2 Incremental CYK/Valiant

Algorithm 1 works only if the entire terminal sequence is observed. However, in constrained decoding, we are given a parser state for prefix $t_0 \dots t_{n-1}$ and a new terminal t_n, and need to compute parser state for $t_0 \dots t_{n-1} t_n$ quickly.

Bernardy and Claessen (2015) proves that a CYK parser can be executed incrementally. To implement such an incremental parser, we build on the observation that $TABLE[row][col]$ depends on $TABLE[r][c]$ if and only if $r + c \leq row + col$ and $c \geq col$. In other words, a cell depends on cells within the triangle tipped by itself, as illustrated in figure 1.

Taking advantage of this property, instead of addressing cells using $[row, col]$, we address cells as

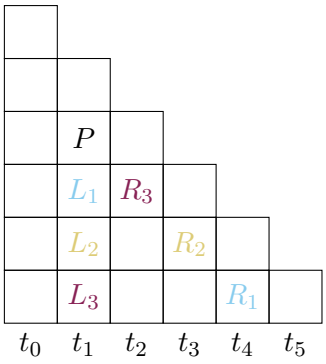

Figure 1: All potential children of the parent cell P at (3, 1) fall in a triangle below it: the cells (r, c) where $r + c \leq 3 + 1$ and $c \geq 1$. All potential children follow this pattern: L_1 at (2, 1), R_1 at (0, 4), L_2 at (1, 1), R_2 at (1, 3), L_3 at (0, 1), and R_3 at (2, 2).

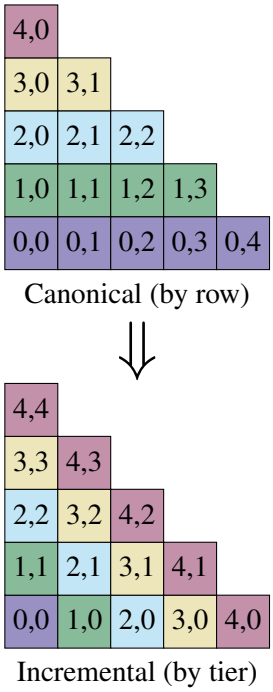

Figure 2: Conversion from canonical (row, col) addressing system to incremental $(tier, row)$ addressing system, where $tier = row + col$. Cells processed in the same loop are in the same color. The canonical approach processes row by row, while the incremental approach processes tier by tier.

$[tier, row]$ as shown in figure 2, and re-arrange the canonical CYK (algorithm 1) into an incremental one (algorithm 2). Notice that each incremental step adds one tier, and each tier is never written to after calculated. This enables the zero-copy branching or rollback property typically desired by constrained decoders.

Algorithm 2 Incremental Valiant Step

Input: $TIER[0 \ldots n-1]$ for terminal $t_0 \ldots t_{n-1}$.
Input: new terminal t_n
1: $TIER$[n][0][V] $\Leftrightarrow \exists$ CNF rule $V \rightarrow t_i$
2: **for** row in **range**(1, n+1):
3: $TIER$[n][row][V] $\Leftrightarrow$
4: $\exists$ CNF rule $V \rightarrow AB$ s.t.
5: $\exists$ mid $<$ row, s.t.
6: $TIER$[n-row+mid][mid][A] and
7: $TIER$[n][n $-$ mid $-$ 1][B]

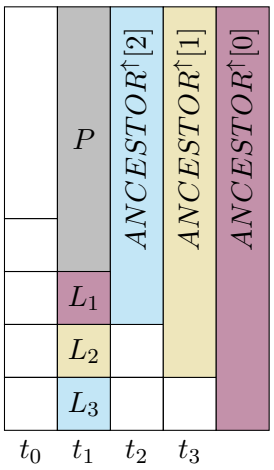

Figure 3: Visually, $ANCESTOR^{\uparrow}$ is the union of all possible future cells in the same column. For each cell to act as left child, any paired right cell will always come from the same column. For example, if a future parent P's left child comes from L_2, its right child must come from $ANCESTOR^{\uparrow}[1]$. Note also that any future parent that has a concrete cell as its left child must have a future cell as its right child, as per CYK indexing (see figure 1).

4.3 Valid Prefix Recognition

Unlike the often used LR/LL/Earley parsers, algorithm 2 will process invalid prefixes (i.e., prefixes that can never be generated from the start symbol) without complaining. This makes it unusable in a constrained decoder where such prefixes should be excluded. So we need to further modify it to compute a new tier type named **ANCESTOR**$^{\uparrow}[j][V]$, which we visualize in figure 3 and formalize as algorithm 3.

Theorem 1. ***ANCESTOR***$^{\uparrow}[j][V]$ *if and only if terminal sequence* $t_{n-j} \ldots t_{n-1}$ *is a valid prefix of non-terminal* V

See appendix A for the formal proof. Intuitively and visually, $ANCESTOR^{\uparrow}$ is the union of all possible future cells in the same column in the CYK

Algorithm 3 $ANCESTOR^{\uparrow}$ algorithm

Input: $TIER[0 \dots n-1]$ for terminal $t_0 \dots t_{n-1}$
Input: $FIRST$ = FIRST set of CNF grammar
Input: hypothetical terminal $\hat{t}_n$

$ANCESTOR^{\uparrow}[0] = FIRST^{-1}[\hat{t}_n]$
FOR row in **RANGE**(1, n+1):
  # Case 1: Left child is a concrete cell
  $ANCESTOR^{\uparrow}[row][V] = True$ if
    $\exists$ CNF rule $V \to AB$ s.t.
      $\exists$ mid $<$ row, s.t.
        $TIER$[n-row+mid][mid][A] and
        $ANCESTOR^{\uparrow}$[n $-$ mid $-$ 1][B]
  # Case 2: Left child is an ancestor cell
  $ANCESTOR^{\uparrow}[row][W] = True$ if
    $\exists V$ s.t. $ANCESTOR^{\uparrow}[row][V] \wedge$
      $FIRST[V][W]$

Algorithm 4 $ANCESTOR^{\downarrow}$ algorithm

Input: $TIER[0 \dots n-1]$ for terminal $t_0 \dots t_{n-1}$
Input: $FIRST$ = FIRST set of CNF grammar

$START$ non-terminal $\in ANCESTOR^{\uparrow}[0]$
FOR row in **RANGE**(n, -1, -1):
  # Case 1: parent in a future cell
  $ANCESTOR^{\uparrow}[row][V] = True$ if
    $\exists V$ s.t.
      $ANCESTOR^{\uparrow}[row][W]$ and
      $FIRST[V][W]$
  # Case 2: left child in a concrete cell
  $ANCESTOR^{\downarrow}[row][V] = True$ if
    $\exists$ CNF rule $V \to AB$ s.t.
      $\exists$ mid $<$ n $-$ row, s.t.
        $TIER$[n-row-1][mid][A] and
        $ANCESTOR^{\downarrow}[row + mid + 1][V]$

table. For a future parent to exist, CYK dictates that it must have a left child in the same column as itself, either from an existing cell or a future one. If the left child is from an existing cell, the right child must be from a future cell, or it would not cover enough terminals. If the left child is from a future cell, the right child must be from a column to the left of n, which is unrestricted by the existing prefix. We only need to prove the existence of the left child's membership in the current $ANCESTOR^{\uparrow}$ column. To check if $\hat{t}_n$ is a viable next terminal, we simply check if the start symbol is in $ANCESTOR \uparrow [n]$.

Definition 1 (FIRST set)**.** *Given a context-free grammar G, non-terminal V and W,*

> ***FIRST***$[V][W]$ *and* ***FIRST***$^{-1}[W][V]$ *if and only if* $V \Rightarrow^* W \dots$

The algorithm for computing $FIRST$ is well established, and is equivalent to computing the transitive closure of the relationship $\{<V, A> \mid \text{rule } V \to A \cdots \in G\}$

4.4 Beyond Valid Prefix Property

When relying on the valid prefix property, a constrained decoder would have to run the parser for each individual hypothetical terminal. Here we show that, with slight modification, algorithm 3 can instead parse top-down and find all possible next terminals within one run. We compute $ANCESTOR^{\downarrow}$ using algorithm 4.

Theorem 2. ***ANCESTOR***$^{\downarrow}[j][V]$ *if and only if* $t_0 \dots t_{n+j-1}$ *stays a valid prefix after appended with a string derivable by* V

The proof of correctness of $ANCESTOR^{\downarrow}$ is symmetrical to that of $ANCESTOR^{\uparrow}$, and thus omitted. The set of all viable next terminals is $\cup_{V \in ANCESTOR^{\downarrow}} FIRST[V]$.

5 Implementation on GPU

5.1 Parallelization

Algorithm 3 and algorithm 4 each contain two synchronization points: one after each case within the loop body. Parallelization therefore occurs within each case.

For the case where the left child is an ancestor cell, the FIRST set is encoded as ancestor/descendant pairs, allowing straightforward parallelization along the pairs dimension. For the case where the left child is a concrete cell, the natural choice is to parallelize along the dimension of CNF grammar rules. Additionally, there is an opportunity to parallelize across rows, as illustrated in figure 4. We distribute different CNF grammar rules across threads and employ hardware bit-wise parallelization across rows.

5.2 Exploiting Chart Sparsity

Letting one thread handle all rows has the additional benefit of skipping computation when the right child (in bottom-up parsing) or parent (in top-down parsing) evaluates to false. This allows partial exploitation of chart sparsity.

5.3 Memory Occupancy and Access Patterns

An unexpected benefit of incremental parsing is that, once a tier is computed, it is subsequently

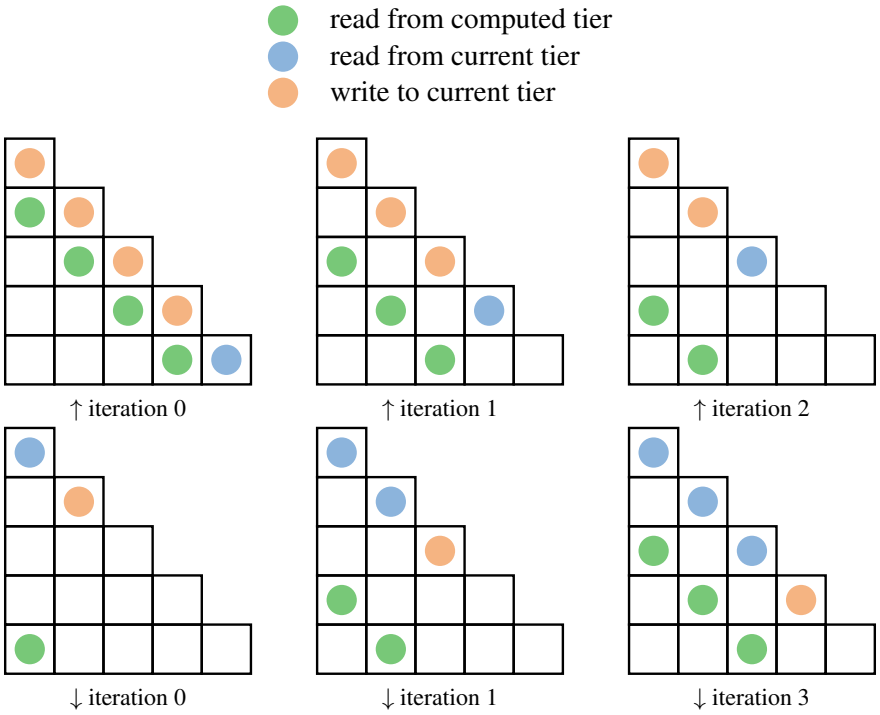

Figure 4: Access patterns. At each iteration, a different already computed tier is fetched as the left child. All the reads and writes across different rows can be accessed concurrently within iteration, which we exploit using hardware bit-wise parallelism.

accessed only as a left child. This limits the number of non-terminals that must be stored in the persistent parser state. We further reduce storage requirements by producing only right-leaning rules during binarization: a rule $X \rightarrow A\ B\ C$ is decomposed into $X \rightarrow A\ X'$ and $X' \rightarrow B\ C$. With this optimization, the number of stored non-terminals in our SQL grammar decreases from 3219 to 103.

CUDA partitions L1 cache and shared memory into 4 and 32 banks, respectively. Bank conflicts occur when threads within the same warp access different addresses in the same bank. Since non-terminals in computed tiers are accessed as left children and each thread processes a different grammar rule, we sort grammar rules by their left child non-terminal. Given that there are only 103 left-child non-terminals but over 3000 normalized grammar rules, threads within each warp of 32 will most likely access either the same address (triggering a broadcast) or neighboring addresses (avoiding conflicts).

5.4 Tensor Memory Accelerator

The NVIDIA Hopper architecture includes a specialized hardware component called the Tensor Memory Accelerator (TMA), which asynchronously fetches data from global memory, freeing CUDA cores for computation. We attempted to use TMA for loading left children—the dominant factor in the $O(n^2)$ per-step complexity. However, this resulted in a slowdown due to additional bookkeeping overhead. We attribute this to the relatively short sequence lengths in our experiments and the effectiveness of the optimizations described above.

6 Alignment With Tokenizer

LLM tokens can span multiple grammar terminals. For example, the Llama-3 token `");` could become the tail of $STRING$, followed by LP and $SEMI$ in an SQL grammar. This necessitates searching multiple steps beyond the already committed terminals, and correct handling is critical to the speed of constrained decoders. Dedicated literature exists on this topic (Beurer-Kellner et al., 2024; Dong et al., 2025), and state-of-the-art approaches can be found in works such as LLGuidance (Moskal et al., 2025) and PSC (Anonymous, 2025).

Efficient alignment of grammar and tokenizer is out of scope for this paper. Instead, we further develop on the zero-copy property discussed in section 4.2 and derive a simple but GPU friendly tree search.

6.1 Parallel Terminal Tree Search

Given a pair of a context-free grammar and an LLM tokenizer, the worst-case search tree is finite due to the finite number of terminals and tokens. This allows pre-computation and pre-allocation of the search tree skeleton, enabling efficient GPU kernels.

We encode the search tree as a flat array, where each element is one possible terminal sequence $t_{lead} \dots t_{tail}$ that can appear within a token. For example, with the full-sized official SQL grammar containing 167 terminals and Llama 3 tokenizer containing 128256 tokens, the reachability of all nodes in the search tree can be encoded as an array of 307692 booleans.

6.1.1 Storage of Parser State

At each node in the search tree, we make a branch of parser state from the parent node then commit a new terminal. With zero-copy, each node only requires one additional parser tier (i.e. $\mathbf{TIER_{t_n \dots t_{n-1} t'_n \dots t'_m}}$).

CUDA requires all memory to be allocated before kernel launch. Although tempting, allocating for every parser tier for every search tree node will exceed the memory limit on larger grammars. So we employ breadth first search (BFS) up to a given depth, and switch to depth first search (DFS) once the search tree is sufficiently developed. Thus, only BFS searched tiers require dedicated parser tier allocation. DFS searched nodes will use a per-worker stack of tiers.

6.2 From Terminal Search Tree to Token Mask

Since a terminal can consist of multiple tokens, it is conventional to track progress within terminals with deterministic finite state automata (DFA). Each token would be assigned multiple tuples $<q_{start}, t, q_{end}>$ indicating that for terminal t the token would transit DFA from state q_{start} to q_{end}.

We expand this notation to account for tokens that span multiple terminals, $<q^{lead}, <t_{lead}, \dots, t_{tail}>, q^{tail}>$ For example, the token “”);” would take a $STRING$ DFA in its penultimate state, complete it by consuming the first character (“”), then complete a LP DFA (“)”), and finally stay on a $SEMI$ DFA (“;”) in its final state.

A token is allowed as a prediction from the large language model (i.e., a token is unmasked) if it has one transition tuple $<q^{lead}, <t_{lead}, \dots, t_{tail}>, \_>$ such that the DFA for t_{lead} is currently in state q^{lead}, and $t_{lead} \dots t_{tail}$ is marked as a viable terminal during the tree search introduced in section 6.1.

7 Experiments

7.1 Data and Grammar

Spider Text-to-SQL is a widely used text-to-SQL dataset. We test on a normal SQL syntax, and a spaced-operators (sp-op) syntax that forces space after symbols “+”, “-”, “*”, “/”, “,” and “;”‘. Both grammars are LALR(1). The spaced-operators grammar better isolates parser performance from token-parser alignment performance, as tokenizers contain some extremely grammar-unfriendly tokens. For example, the Qwen 3 tokenizer contains a token where the “-” character is repeated 98 times, which could be interpreted as many unary minus signs by the grammar. Matching such unrealistically long math expressions would require a more sophisticated search strategy than our simple GPU-friendly tree search.

UPenn TreeBank is a small English grammar obtained from the production rules used in 300 samples from UPenn TreeBank. We measure performance by parsing the exact same 300 samples. To simplify lexer construction, we parse part-of-speech sequences instead of word sequences. This grammar is 1000 lines long and highly ambiguous. We will release both the grammar and the data upon publication to allow future work to replicate our settings.

7.2 Baselines

We compare against three constrained decoding systems across various underlying parsers.

XGrammar (Dong et al., 2025) emulates a push down automata (PDA), which provides general context-free recognition. However, due to the non-deterministic nature of PDA, time complexity of the system is unclear. XGrammar reduces search scope within tokenizer vocabulary by detecting tokens that can be determined without running the underlying parser.

LLGuidance (Moskal et al., 2025) uses an Earley parser, allowing $O(n)$ complexity in optimal grammars and $O(n^3)$ in the general case. To search through the tokenizer vocabulary efficiently, it employs a trie of tokens at the character level.

Method	SQL			SQL (SP-OP)[5]			TREEBANK		
	Cmplx.	Init	Per-mask	Cmplx.	Init	Per-mask	Cmplx.	Init	Per-mask
XGrammar (Dong et al., 2025)	?	1s	278319μs[1]	?	1s	269841μs[1]	?	1s	1e7μs[2]
LLGuidance (Earley; Moskal et al., 2025)	$O(n)$	1s	770μs	$O(n)$	1s	646μs	$O(n^3)$	1s	✗[3]
PSC (LALR; Anonymous, 2025)	$O(n)$	4770s	1μs	$O(n)$	4770s	1μs	✗[4]	✗	✗
Ours (Incremental Valiant)	$O(n^3)$	40s	8670μs	$O(n^3)$	40s	492μs	$O(n^3)$	17s	248μs

Table 1: Performance comparison across different grammars.

[1] Estimated from 30 samples(totals at 10 min). [2] Estimated from 5 samples (totals at 30 min). [3] LLGuidance aborts at ∼20 tokens; Before termination, step time between 10-20th token costs 18682 μs on average, comparing to 250 μs of ours in figure 6. [4] LALR is believed to be too restrictive for this grammar. Our attempt to initialize an LALR parser did not terminate after 3 hours. [5] SQL(sp-op) inserts spaces after unary operator such as minus (−) and semicolon (;). LLM tokenizer contains tokens such as 96 "−", which cause problem with our simplistic implementation of tokenizer/parser alignment.

PSC (Anonymous, 2025) shows that in a deterministic push down automata based parser, the set of valid next terminal sequences (to a finite length) can be computed using the parser's stack and a finite state transducer. Consequently, the number of parser calls is independent of LLM vocabulary size.

7.3 Environment and Implementation

We run experiments on a cloud VM with Intel Xeon Platinum 8480+ CPU and Nvidia H100 PCIe GPU. GPU kernels are implemented in CUDA and the tokenizer used is from Qwen 3.

7.4 Results

Our main results are summarized in table 1, which reports the complexity of the underlying parser, the constrainer initialization time, and the per-mask (i.e., per-token) overhead. In the experiments on the LALR SQL grammar, our system outperforms XGrammar significantly in per-mask overhead. When we experiment on the SQL(sp-op) grammar variant to control for the naively implemented tokenizer/constrainer alignment, we empirically outperform the Earley-based LLGuidance, despite its theoretically superior complexity. Our system still underperforms the LALR-based PSC, which takes advantage of the properties of LALR grammars, while giving up the ability to parse more general context free grammars.

In the experiments on the highly ambiguous grammar found in the TreeBank, our system shows robust efficiency without degradation. XGrammar is the only other system that runs successfully on this grammar, but 40,000 times slower in average per-mask overhead, with in some cases a single token mask taking minutes to compute. The Earley-based LLGuidance aborts at ∼20 tokens. Before termination, the step time between the 10th and 20th token costs 18682 μs on average, much slower than the 250 μs of ours. LALR parsers like PSC are commonly believed to be inherently incapable of handling complex grammars such as those found in the TreeBank. We still attempted to initialize an LALR parser using the grammar, and the construction did not terminate after 3 hours of wait. These results suggest that our parser allows enforcing a broader range of constraints that were previously too complex to be practical.

In terms of initialization time, our parser (40s/17s) is slower than XGrammar (1s/1s) and Earley-based LLGuidance (1s/1s), but still faster than PSC (4770s/✗). Our initialization process uses mostly un-optimized Python and Numba code with redundant features for other projects. XGrammar and LLGuidance on the other hand use C and rust implemented initializations. Thus, there is significant room for improvement in the intialization of our parser.

GPU Occupancy Figure 5 shows that our constrainer only requires a fraction of stream multiprocessors (SM) on a GPU. Most of the efficiency is achieved using 16 to 32 out of over 100 available SM on an H100. In fact, each of our parser steps only uses 1 single SM, with work distributed across SMs only at the terminal trie level as described in section 6.1. The SQL grammar is more sensitive to the number of workers due to the larger number of LLM tokens containing multiple grammar terminals. This suggests our approach has the capacity to process larger batches concurrently with minimal addition to latency.

Scaling Figure 6 shows the growth of overhead per token, which has a theoretical bound of $O(n^2)$.

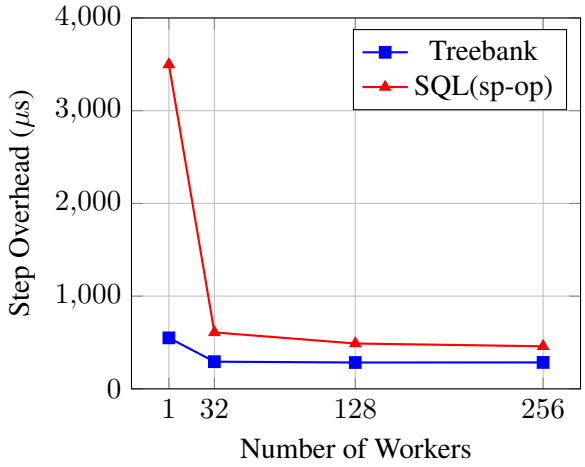

Figure 5: Each worker is a CUDA thread block that occupies at most one stream multiprocessor (out of over 100 on a single GPU). Our implementation requires only a fraction of a GPU (16 to 32 workers) to achieve most of the speed gain. The number of required workers is proportional to the extend of LLM tokens containing multiple grammar terminals.

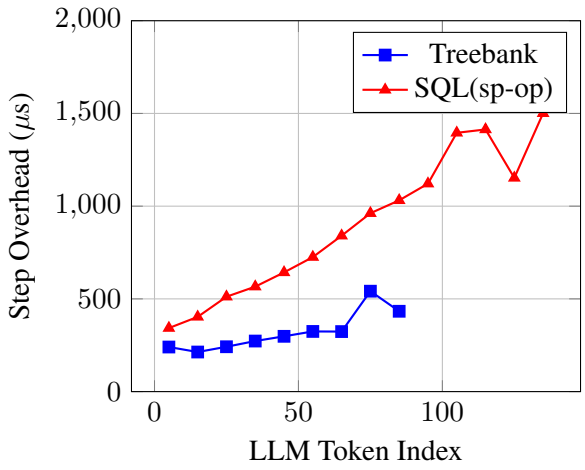

Figure 6: Although step time has theoretical growth rate of $O(n^2)$, empirical growth is more benign, likely due to the inherent and highly sparse nature of the chart. Displayed times are averages across data points from the entire benchmark set.

The empirical growth is more benign likely due to the inherent sparsity of parser charts. We also note that the constant overhead of launching a step is non-trivial, suggested by the intercept at the left-end of the curve.

Single Parser Step vs Single Constrainer Step A constrainer step might contain multiple parser steps if a LLM token contains multiple grammar tokens, such as “");” in SQL. In the case where such problems are less frequent, the constrainer step will be closer to the cost of the parser step, which we observed at approximately 100μs for $n = 100$ and $|G^{CNF}| = 3219$.

8 Conclusion

We derived a new variant of the Valiant recognizer, adding support for valid prefix recognition, a critical component for supporting the use of constrained decoders with large language models. Our parsing model is parallelizable and we show that it is efficient with only a fraction of a GPU. Our approach enables high performance context-free constrained decoding over a wider range of grammars than prior work. This is critical for expanding use cases and reducing engineering effort for users of constrained decoders with large language models, who would previously have had to work within the overly restrictive LALR formalism to get efficient context free grammar based decoding.

Limitations

Our strategy to align LLM tokens and grammar terminals is primitive and causes a slow down when tokens contain multiple terminals. Improved handling of this mismatch between the grammar and the tokenizer would yield improved parser performance.

We use dense charts in our parser. However, both theoretical work (Bernardy and Claessen, 2015) and empirical experiments on GPUs (Azimov and Grigorev, 2018) show that sparse representations of parser charts can improve run time and complexity (to $O(log^2 n)$ in some cases) of CYK-like parsers. Further research would be required to discover a version of our parser compatible with sparse charts.

Our experiments are on datasets that are associated with only two grammars. Though we believe that the SQL and TreeBank grammars are representative of useful endpoints along the spectrum of unambiguous to ambiguous grammars, a collection of grammars used in real world applications of large language models with constrained decoders would allow a deeper investigation of performance.

References

Anonymous. 2025. PSC: Efficient grammar-constrained decoding via parser stack classification. In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.

Rustam Azimov and Semyon Grigorev. 2018. Context-free path querying by matrix multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International*

Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), GRADES-NDA '18, New York, NY, USA. Association for Computing Machinery.

Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. Crane: Reasoning with constrained llm generation. *Preprint*, arXiv:2502.09061.

Jean-Philippe Bernardy and Koen Claessen. 2015. Efficient parallel and incremental parsing of practical context-free languages. *Journal of Functional Programming*, 25:e10.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding llms the right way: fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Noam Chomsky. 1959. On certain formal properties of grammars. *Information and Control*, 2(2):137–167.

John Cocke. 1969. *Programming languages and their compilers: Preliminary notes*. New York University, USA.

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. *Preprint*, arXiv:2411.15100.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Tadao Kasami. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. 2025. Syntactic and semantic control of large language models via sequential monte carlo. *Preprint*, arXiv:2504.13139.

Michał Moskal, Harsha Nori, Hudson Cooper, and Loc Huynh. 2025. Llguidance: Making structured outputs go brrr.

Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. 2025. Chopchop: a programmable framework for semantically constraining the output of language models. *Preprint*, arXiv:2509.00360.

Elizabeth Scott and Adrian Johnstone. 2010. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

Xintong Sun, Chi Wei, Minghao Tian, and Shiwen Ni. 2025. Earley-driven dynamic pruning for efficient structured decoding. *Preprint*, arXiv:2506.01151.

Masaru Tomita. 1985. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'85, page 756–764, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. 2025. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Syncode: Llm generation with grammar augmentation. *Preprint*, arXiv:2403.01632.

Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189–208.

A Correctness of ANCESTOR$^{\uparrow}$

A.1 Completeness

Using induction, we prove **ANCESTOR**$^{\uparrow}[j][V]$ if V has valid prefix $t_{n-j} \dots t_{n-1}$.

- **Base case:** $j = 0$
  - invaild index, there is no t_n yet, $t_n \dots t_{n-1}$ is empty string
  - every non-terminal has empty string as its valid prefix
  - we set **ANCESTOR**$^{\uparrow}[0][:] = True$
- **Induction:** $j = 1 \dots$
  - Induction Hypothesis:
    ANCESTOR$^{\uparrow}[k][V]$ if V has valid prefix $t_{n-k} \dots t_{n-1}$ for any $k < j$
    $\Rightarrow$ **ANCESTOR**$^{\uparrow}[j][V]$ if V has valid prefix $t_{n-j} \dots t_{n-1}$
  - Assume V has valid prefix $t_{n-j} \dots t_{n-1}$
  - There exists rule $V \to AB$, where
    - $A \Rightarrow^* t_{n-j} \dots t_{n-j+mid}$
    - $B \Rightarrow^* t_{n-j+mid+1} \dots$
  - Depending on mid, there are two cases
    - **Case 1:** A only generates part of or exactly entire of the committed prefix

1. formally
   · $n - j + mid \leq n - 1$
   · $A \Rightarrow^* t_{n-j} \dots t_{n-j+mid}$
   · $B \Rightarrow^* t_{n-j+mid+1} \dots t_{n-1} \dots$
2. $A \in$ **TIER**$[n - j + mid][n - j]$, by 1 and CYK
3. $B \in$ **ANCESTOR**$^{\uparrow}[j - mid - 1]$, by 1 and IH
4. **ANCESTOR**$^{\uparrow}[j][V]$, by case 1 in pseudo-code, 2, 3

∗ **Case 2:** A generates all of the committed prefix and beyond

1. formally
   · $n - j + mid > n - 1$
   · $A \Rightarrow^* t_{n-j+mid} \dots t_{n-1} \dots$
2. Since the grammar is Chomsky Normal form, it is guaranteed that $\exists W$ s.t.
   · $W \in FIRST[V]$, and
   · $W \rightarrow CD$, and
   · W belongs to case 1. (i.e. C generates part or exactly the committed prefix
3. **ANCESTOR**$^{\uparrow}[j][W]$, by IH, 2
4. **ANCESTOR**$^{\uparrow}[j][V]$, by case 2 in pseudo-code, 3

A.2 Soundness

Using induction, we prove **ANCESTOR**$^{\uparrow}[j][V]$ only if V has valid prefix $t_{n-j} \dots t_{n-1}$.

- **Base case:** $j = 0$
  - invaild index, there is no t_n yet, $t_n \dots t_{n-1}$ is empty string
  - every non-terminal has empty string as its valid prefix
  - we set **ANCESTOR**$^{\uparrow}[0][:] = True$
- **Induction:** $j = 1 \dots$
  - Induction Hypothesis:
    ANCESTOR$^{\uparrow}[k][V]$ only if V has valid prefix $t_{n-k} \dots t_{n-1}$ for any $k < j$
    $\Rightarrow$ **ANCESTOR**$^{\uparrow}[j][V]$ only if V has valid prefix $t_{n-j} \dots t_{n-1}$
  - Assume **ANCESTOR**$^{\uparrow}[j][V]$
    ∗ **Case 1:**
    1. **ANCESTOR**$^{\uparrow}[j][V]$ is set to true by condition
       · $V \Rightarrow AB$, and
       · **TIER**$[n - j + mid][n - j][A]$, and
       · **ANCESTOR**$^{\uparrow}[j - mid - 1][B]$
    2. $A \Rightarrow^* t_{n-j} \dots t_{n-j+mid}$, by CYK, 1
    3. $B \Rightarrow^* t_{n-j+mid+1} \dots t_{n-1} \dots$, by IH, 1
    4. $V \Rightarrow^* t_{n-j} \dots t_{n-1} \dots$, by 1, 2, 3

    ∗ **Case 2:**
    1. **ANCESTOR**$^{\uparrow}[j][W]$ is set to true by condition
       · **FIRST**$[V][W]$, and
       · **ANCESTOR**$^{\uparrow}[j][V]$, caused by case 1
    2. $V \Rightarrow^* t_{n-j} \dots t_n - 1 \dots$, by case 1
    3. $W \Rightarrow^* t_{n-j} \dots t_{n-1} \dots$, by property of **FIRST** set