Lobster: A GPU-Accelerated Framework for Neurosymbolic Programming

ANONYMOUS AUTHOR(S)

Neurosymbolic programs combine deep learning with symbolic reasoning to achieve better data efficiency, interpretability, and generalizability compared to standalone deep learning approaches. However, neurosymbolic learning frameworks are significantly hindered in scalability compared to modern deep learning frameworks. This limitation stems from challenges involved in performing differentiable symbolic reasoning efficiently compared to tensor computations which are more readily amenable to GPU acceleration. As a result, existing neurosymbolic learning frameworks implement an uneasy marriage between a highly scalable, GPU-accelerated neural component with a slower symbolic component that runs on CPUs.

We propose Lobster, a unified framework for harnessing GPUs in an end-to-end manner for neurosymbolic learning. Lobster provides a theoretical foundation and a practical implementation for mapping a general neurosymbolic language based on Datalog to the GPU programming paradigm. Lobster supports discrete, probabilistic, and differentiable modes of reasoning on GPU hardware through a versatile library of provenance semirings. We demonstrate that Lobster programs can solve interesting problems spanning the domains of natural language processing, image processing, program reasoning, bioinformatics, and planning. On a suite of 8 applications, Lobster achieves an average speedup of 5.3x over Scallop, a state-of-the-art neurosymbolic framework, and enables scaling of neurosymbolic solutions to previously infeasible tasks.

CCS Concepts: • **Theory of computation → Probabilistic computation**; • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: neurosymbolic programming, GPUs, logic programming, Datalog

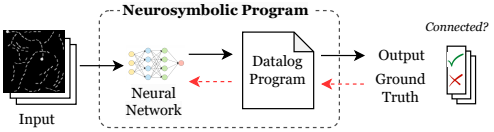
1 Introduction

Modern programming methods are based on two distinct paradigms: logical reasoning and deep learning. Logical reasoning excels at problems with clearly defined rules and structured data, such as sorting a list of numbers or finding a shortest path in a graph. In contrast, deep learning is well suited in contexts where logical reasoning approaches become intractable, particularly for problems involving noisy, complex, and high-dimensional data—such as detecting objects in an image or parsing natural language text.

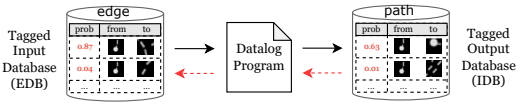
Many problems across different domains demand the complementary capabilities of these two paradigms. Neurosymbolic programming [7] is an emerging approach that enables solving such problems by suitably decomposing the computation between a neural network and a logic program. The resulting *neurosymbolic programs* have been demonstrated to achieve better data efficiency, interpretability, and generalizability compared to standalone deep learning approaches.

Recent frameworks such as DeepProbLog [24], Scallop [23], and ISED [40] have enhanced the programmability and accessibility of neurosymbolic applications. Figure 1a illustrates a neurosymbolic program for solving a binary image classification problem [43]. The logic program is specified in Datalog [2], a declarative language supported by Scallop. Using a differentiable Datalog engine, gradients can be back-propagated through the program to train the neural network, thereby enabling automatic learning of relevant image features without manual engineering.

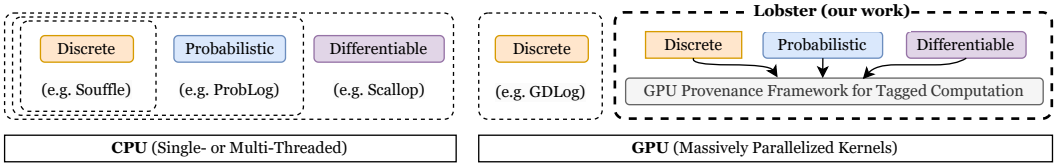
Despite their benefits, neurosymbolic programs using these frameworks incur significant computational overhead during training and inference. The scalability challenges stem primarily from managing probabilistic data and tracking additional information to maintain end-to-end differentiability. For instance, the logic program in the example in Figure 1b takes as input a graph represented by the `edge` relation, and outputs its transitive closure represented by the `path` relation. Each tuple in these relations is associated with a probability. Further, computing the probability



(a) An example neurosymbolic program.



(b) The input and output for the Datalog program.



(c) High-level comparison between Lobster and existing frameworks.

Fig. 1. Example of a neurosymbolic program using Datalog and frameworks for neurosymbolic programming.

of each path tuple must take into account all possible ways to derive it from the edge tuples, and their associated probabilities obtained from the neural network. Since it is often intractable to perform exact probabilistic reasoning, approximated probabilistic inference is employed, but does not fundamentally address scalability. Differentiability further complicates the problem by requiring to track each input's contribution to the output, increasing space and time complexity due to the extra book-keeping required for gradients.

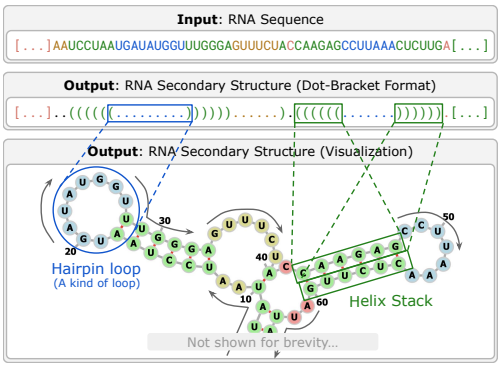
In this paper, we propose Lobster, a GPU-accelerated framework designed to enhance the scalability of neurosymbolic programming. The core innovation of Lobster lies in efficiently mapping an expressive subset of Datalog—a declarative logic programming language shown to be effective in neurosymbolic contexts [23]—onto GPU architectures, for different modes of reasoning: discrete, probabilistic, and differentiable. This subset includes computationally-intensive operations like join and recursion, enabling levels of scalability unattainable with single- or multi-threaded CPUs.

In Figure 1c, we compare Lobster with existing logic programming frameworks. While various engines exist for discrete [34], probabilistic [14], and differentiable [23] settings, they are limited to CPU runtimes with single- or multi-threading. GDLog [37, 42] enables Datalog evaluation on GPUs, accelerating queries for tasks like static analysis [6] and social media analytics [36]. However, GDLog supports only discrete reasoning with a restricted front-end, limiting its generality. In contrast, Lobster is designed for general neurosymbolic queries with multiple reasoning modes within a unified, GPU-accelerated provenance semirings framework [23].

We summarize the core contributions of this paper as follows:

- We introduce Lobster, the first GPU-accelerated neurosymbolic programming framework.
- We propose the APM language for parallel relational reasoning, and show how to compile conventional relational algebra operations to APM.
- We implement a full-fledged compiler and runtime system for Lobster using Rust and CUDA.
- We evaluate Lobster on an extensive set of discrete, probabilistic, and differentiable benchmarks from the literature, showing that Lobster consistently outperforms prior systems, including specialized ones. Lobster achieves speedups of >100× over the closest related work, Scallop.

The rest of the paper is organized as follows. We first give an illustrative overview of Lobster in Section 2 using a real-world neurosymbolic application. Next, we describe Lobster's core compiler and runtime (Section 3), followed by various optimizations (Section 4). Implementation details are discussed in Section 5, and experimental results in Section 6.



(a) Illustration of the RNA folding problem. We visualize the output RNA secondary structure in the bottom. The grey arrow indicates the direction in which the indexes of nucleotides are increasing.

$$\text{(Structure Token)} \quad \tau \quad ::= \quad H_l \mid H_r \mid L_l \mid L_r \mid L_u \mid E_u$$

(b) The set of structure tokens (terminal symbols) for our CFG, including helix (H), loop (L), and external loop (E). The subscripts l (left), r (right), and u (unpaired) denote the structural role of the corresponding nucleotide within each substructure.

$$\begin{aligned}
\text{(Paired Substr.)} \quad \mathcal{P} &::= \mathcal{L} \mid \mathcal{H} \\
\text{(Helix Stack)} \quad \mathcal{H} &::= H_l \cdot \mathcal{P} \cdot H_r \\
\text{(Ext. Unpaired)} \quad \mathcal{E} &::= E_u \mid \mathcal{E} \cdot E_u \\
\text{(Int. Unpaired)} \quad \mathcal{U} &::= L_u \mid \mathcal{U} \cdot L_u \\
\text{(Loop Body)} \quad \mathcal{B} &::= \mathcal{U} \mid \mathcal{P} \cdot \mathcal{U} \mid \mathcal{B} \cdot \mathcal{P} \mid \mathcal{B} \cdot \mathcal{U} \\
\text{(Loop)} \quad \mathcal{L} &::= L_l \cdot \mathcal{B} \cdot L_r \\
\text{(RNA SS)} \quad \mathcal{R} &::= \mathcal{P} \mid \mathcal{E} \mid \mathcal{R} \cdot \mathcal{R}
\end{aligned}$$

(c) The complete CFG used to parse RNA secondary structures ($\mathcal{R}$).

Fig. 2. RNA folding problem and the CFG that can parse RNA structures.

2 Illustrative Overview

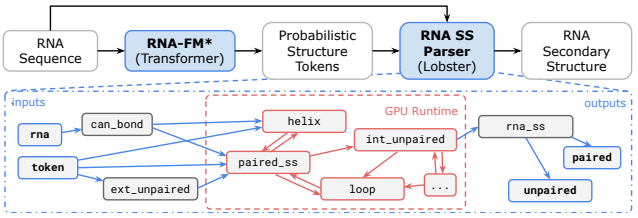
We illustrate Lobster by developing a neurosymbolic solution to the problem of RNA Secondary Structure (SS) prediction, or RNA folding. The problem is central to understanding RNA molecule functionality, with applications in gene regulation, drug discovery, and synthetic biology. Furthermore, it highlights scalability challenges we wish to address because it involves a substantial logic program for performing probabilistic parsing using a context-free grammar (CFG).

2.1 Problem Definition

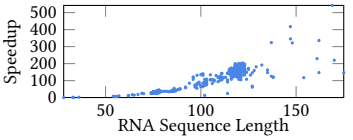
Given an RNA sequence of nucleotides ({A, C, G, U}), the goal is to determine its secondary structure, which specifies whether each nucleotide at position i pairs with a nucleotide at position j, or remains unpaired. A primary way to represent this output is through "dot-bracket" notation, where unpaired nucleotides appear as dots, and paired nucleotides as matched brackets (Figure 2a). For the structure to be valid, all "brackets" must be properly matched, forming a well-structured output.

A neurosymbolic approach is a natural fit due to the symbolic yet non-deterministic nature of the problem. RNA molecules in nature can fold into multiple valid structures depending on environmental conditions. However, certain principles remain consistent, such as structural constraints that enforce matching pairs and pairing rules that allow only specific nucleotide pairs (A-U, C-G, and A-G). In our solution, the neural component makes data-driven predictions about the potential structural roles of nucleotides—such as whether a nucleotide resides within a helix-stack or an internal loop. The symbolic component, on the other hand, uses these predictions to parse the RNA sequence into a well-formed secondary structure. Without neurosymbolic, a purely neural approach would struggle to infer long-range dependencies, and a purely symbolic approach would have trouble adapting to specific datasets and distributions.

Our solution employs a CFG with its terminal symbols, or structure tokens, illustrated in Figure 2b. The neural network produces a probability distribution over these structure tokens for each nucleotide. The symbolic component then processes this sequence of distributions to identify the "most probable parse" that satisfies the CFG (Figure 2c). To generate the final dot-bracket notation, we extract all paired substructures $\mathcal{P}$ from the parsed syntax tree represented by $\mathcal{R}$. By employing a fine-tuned RNA-FM model [8]—a foundation model for RNA sequence embedding—as the neural



(a) Our neurosymbolic pipeline for predicting RNA secondary structure. The main parsing loop is computed on the GPU.



Overall (475 Seqs)	Scallop	Lobster
Time	13 hours	**6 minutes**
Speedup	1×	**146×**

(b) The speedup of Lobster across different RNA sequence lengths relative to Scallop.

```
1  type Nucleotide = A | C | G | U                          // enum type for nucleotides
2  type StructureToken = Hl | Hr | Ll | Lr | Lu | Eu        // enum type for structure tokens
3
4  type rna(idx: usize, nuc: Nucleotide)        // input RNA seq (index mapped to nucleotide)
5  type token(idx: usize, tok: StructureToken)    // probabilistic tokens extracted by RNA-FM
6
7  rel can_bond = {(A, U), (U, A), ...}      // facts for nucleotide pairs that can be bonded
8  rel paired_ss(i, j) = loop(i, j) or helix(i, j)           // Rule for paired-substructures
9  rel helix(i, j) =                                          // Rule for helix stack
10   rna(i, x_i) and rna(j, x_j) and can_bond(x_i, x_j) and   // i,j must be bondable
11   token(i, Hl) and paired_ss(i + 1, j - 1) and token(j, Hr)   // production rule for helix
12 // ... other rules for parsing RNA are omitted for brevity
```

(c) Lobster program snippet that parses the RNA secondary structure according to the CFG in Figure 2c.

Fig. 3. The overall pipeline, symbolic program, and the acceleration result of our solution.

component, this neurosymbolic solution achieves a 92.6% F1 score on the ArchiveII dataset [39], surpassing established methods such as MxFold2 [33] which scores 88.9%.

2.2 Scalability and Programmability Challenges

Our neurosymbolic solution poses a significant scalability challenge. While the neural component can utilize modern hardware accelerators like GPUs and TPUs, the symbolic component currently runs on CPUs alone. The symbolic engine must derive a set of all possible predicted structures and their associated probabilities. As the length of the input sequence increases, the number of possible parses and the size of their associated weights also grows exponentially, leading to a combinatorial explosion in the number of required computations.

While custom GPU implementations have been developed for these algorithms [50], it demands specialized knowledge (e.g., parallel programming in CUDA) and hinders domain experts from focusing on functionality. Even for the problem of RNA folding, such experts have designed diverse CFGs tailored to specific biological contexts, making it impractical to implement custom GPU solutions for each CFG variant. A general purpose framework is therefore desirable for making accelerated GPU computation widely accessible.

2.3 Our Approach

We present the high-level pipeline of our approach in Figure 3a. We specify the symbolic component as a Datalog program which is partially shown in Figure 3c. Datalog offers an intuitive interface for defining data types (lines 1-2) and relation types (lines 4-5) which we use to specify the inputs. Notice that the RNA sequence of nucleotides is encoded as a binary relation between indices (`usize`, unsigned integers) and the nucleotides.

Our program contains recursive rules, which are key for modeling the hierarchical structure of our CFG. Line 8 defines the rule for paired substructures ($\mathcal{P}$), which could be either an internal loop ($\mathcal{L}$) or a helix-stack ($\mathcal{H}$). Lines 9-11 define the rule for helix-stacks ($\mathcal{H}$). Assuming that i-th and j-th nucleotides can be bonded (line 10), the subsequence $[i, j]$ can be parsed as a helix stack ($\mathcal{H}$) if the i-th and j-th nucleotide represent the left (L_l) and right (L_r) side of a helix-stack, while the enclosed subsequence $[i + 1, j - 1]$ forms a paired-substructure (line 11). Notice how closely these declarative rules mirror the corresponding CFG rules depicted in Figure 2c.

With the declarative high-level language, Lobster offers a convenient abstraction to hide the underlying GPU runtime from users. Figure 3a shows the dependency graph of the relations needed to compute RNA secondary structures. Lobster off-loads the main recursion computation involving the inter-dependent relations to the GPU runtime for maximum acceleration.

The probabilistic reasoning semantics is also abstracted away. The `token` relation in Figure 3c (line 5) contains probabilistic facts extracted by the underlying neural network. For example, a fact `0.97::token(16, Hl)` represents that the 16-th nucleotide in the RNA is predicted to be the left element of a helix-stack, with the probability of 0.97. All the derived facts, such as `helix(i, j)`, will carry probabilities computed from a concrete subsequence of structure tokens. This is manifested by the underlying provenance framework within Lobster. In our solution, we use a probabilistic provenance called `top-1-proof` which instruments the program to carry the most-likely parse for any given subsequence. The probabilistic reasoning semantics and the provenance framework are both accelerated on the GPU by Lobster, which we elaborate upon in Section 3.

2.4 Our Results

We advance the state-of-the-art by solving neurosymbolic programming's scalability challenges via GPU acceleration. Figure 3b shows the speedup of Lobster over the CPU baseline, Scallop, on a subset of the ArchiveII dataset [39]. Scallop suffers noticeable performance drops as sequence length increases, while Lobster scales much more gracefully, parsing all 475 sequences in 6 minutes. Speedups of 100× are easily achieved on the median RNA sequence length of 120. This efficiency gain is enabled by mapping a significant fragment of Datalog to the GPU programming paradigm, which entails careful design decisions involving how to represent relations, how to parallelize relational operators, and how to schedule computation.

How to Represent Relations? Lobster uses a column-oriented layout in order to make optimal use of the GPU memory hierarchy. This choice makes sense in the context of GPU acceleration as column-oriented layouts are cache-friendly. Concretely, this means some of the simpler operations in the RNA task, such as `unioning` the relation `loop` with the relation `helix` to create the `paired_ss` relation, can reach close to 100% utilization of the GPU memory bus. This choice also suits the context of executing Datalog programs. For instance, the probabilistic parsing algorithm is compiled to a series of query operations consisting of relational operations like *join* and *project* which focus on specific columns. As a result, columnar data allows more natural algorithm implementations. We discuss the memory layout further in Section 5.

How to Parallelize Relational Operators? Beyond memory layout, efficient algorithms are necessary to improve the performance of symbolic computations. In Lobster, the key insight is that Datalog programs are compiled to a core set of relational queries, and each of these queries can be individually parallelized to improve their performance on potentially massive inputs. For example, the rule on line 9 of Figure 3c is compiled to a query that includes joining the entire set of currently predicted structures against the input sequence. The size of the input to this join grows exponentially as the program iterates, so executing the join with data-parallelism with respect to its input is critical. We describe how our compiler and runtime expose this parallelism in Section 3.

(Predicate)	ρ		
(Projection Fn.)	α		
(Selection Fn.)	β		
(Expression)	ϵ	::=	$\rho \mid \pi_\alpha(\epsilon) \mid \sigma_\beta(\epsilon)$
		$\mid$	$\epsilon_1 \bowtie_n \epsilon_2 \mid \epsilon_1 \cup \epsilon_2$
		$\mid$	$\epsilon_1 \times \epsilon_2 \mid \epsilon_1 \cap \epsilon_2$
(Rule)	ψ	::=	$\rho \leftarrow \epsilon$
(Stratum)	ϕ	::=	$\{\psi_1, \ldots, \psi_n\}$
(Program)	$\bar{\phi}$	::=	$\phi_1; \ldots; \phi_n$

Fig. 4. The RAM language.

(Scalar Immediate)	n		
(Register)	id		
(Scalar Operator)	f	::=	`size` | `last` | `overhead`
(Scalar Expression)	s	::=	id | $f(s_1, \ldots, s_n)$ | n | $[s_1; \ldots; s_n]$
(Operator)	op	::=	`alloc` | `eval` | `gather` | $\texttt{gather}_{\texttt{if}}$
		$\mid$	`join` | `build` | `count` | `append`
		$\mid$	`copy` | `repeat` | `mult` | `clamp`
		$\mid$	`scan` | `sort` | `unique` | `merge`
		$\mid$	`store` | `load`
(Instruction)	e	::=	$op\langle\cdot\rangle(s_1, \ldots, s_n)$
(Block)	b	::=	$e_1 \ldots e_n$
(Program)	p	::=	$\texttt{lfp}(b)$

Fig. 5. The APM language.

How to Schedule Computation? Finally, the choice of which portions of the symbolic computation benefit from GPU acceleration is not obvious: while high in throughput, GPUs exhibit higher latency than CPUs, meaning they are not ideal for short-lived operations. Consequently, Lobster allows offloading only certain portions of a computation to the GPU to avoid undesirable overheads. An example can be seen at the bottom of Figure 3a, where pre- and post-processing steps that do not require heavy compute are executed on the CPU, but execution of the main parsing loop is offloaded to the GPU. We describe the implementation details in Section 4.

3 Compiler and Runtime

Lobster focuses on accelerating the Datalog back-end with GPU hardware. We assume an existing Datalog compiler is capable of taking in a user-level program and producing a mid-level program based on relational algebra. From there, Lobster further compiles it down to a program that can be executed by the GPU. In this section, we describe APM, a low-level sequential language with parallelized kernels related to general relational computation. We also present the compilation process from the mid-level relational algebra language to APM.

3.1 Background

Relational Algebra Machine. We start by describing our compiler's source language, the Relational Algebra Machine (RAM), which is based on the familiar language of *Relational Algebra* for expressing database queries [2]. The abstract syntax of RAM is shown in Figure 4. At a high level, executing a relational algebra program $\bar{\phi}$ means sequentially executing all the strata $\phi_1, \ldots, \phi_n$. Within each stratum, rules are iteratively executed against an input extensional database (EDB) until a fix-point is reached. The resulting materialized facts comprise the intensional database (IDB). Each rule $\rho \leftarrow \epsilon$ consists of a target relation ρ and a query ϵ. This query is a dataflow graph with potentially many sources but only one sink. The operators in the graph are a core fragment of relational algebra operators, comprising project (π), select (σ), and join ($\bowtie$) as well as three set operators, union ($\cup$), product ($\times$), and intersect ($\cap$). Note that π and σ allow taking arbitrary projection or selection functions, while the join operation $\bowtie$ accepts the number of columns to perform join on. For this section, we focus on accelerating a single recursive stratum.

Provenance Semirings. For relational algebra programs to be incorporating differentiable or probabilistic reasoning, prior works [23, 24] have shown that each fact can be tagged to carry additional information such as probabilities or boolean formulas. More generally, *provenance semirings* [18] enable programmable semantics that allow tags from an arbitrary semiring. Formally speaking, a provenance semiring T is a 5-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ where T is the space of tags (Figure 6a).

$$
\begin{array}{rccl}
\text{(Tag)} & t & \in & T \\
\text{(False)} & \mathbf{0} & \in & T \\
\text{(True)} & \mathbf{1} & \in & T \\
\text{(Disjunction)} & \oplus & : & T \times T \to T \\
\text{(Conjunction)} & \otimes & : & T \times T \to T
\end{array}
$$

(a) The provenance semiring structure.

Provenance	T	$\mathbf{0}$	$\mathbf{1}$	$\oplus$	$\otimes$
Bool	$\{\bot, \top\}$	$\bot$	$\top$	$\vee$	$\wedge$
Max-Min-Prob	$[0, 1]$	0	1	max	min
Top-k-Proofs	Φ	$\{\}$	$\{\mathbf{0}\}$	$\vee_k$	$\wedge_k$

(b) Example provenance semirings used in the literature.

Fig. 6. Provenance semiring structure and some examples of provenance semirings.

$\oplus$ and $\otimes$ dictate how tags are combined through disjunction and conjunction operations. In Figure 6b, we show a few provenance semirings used in the literature [14, 18, 20] for discrete reasoning and approximated probabilistic reasoning. Specifically, a tag can be a boolean formula $\phi \in \Phi$ represented in disjunctive normal form (DNF) under set notation. Here, the boolean variables v will be references to facts in the input databases, often represented as integers. With probability $\Pr(v)$ attached, one might perform top-k filtering on proofs to avoid blow-up of the boolean formulas. In order to support the discrete, probabilistic, and differentiable modes of reasoning, Lobster implements 7 commonly used provenance semirings, which we elaborate in Section 3.4.

GPU Computation Model. Traditionally, GPU programming is much like C programming: an unbounded set of programs can be expressed, even ones that map poorly to the underlying hardware. We aim to alleviate this problem by taking the implicit guidelines of the GPU programming model and making them explicit in the design of APM. To that end, we now explore important restrictions of GPU programming, and we tie these restrictions to design considerations of APM.

(1) **Lockstep Execution** While GPUs have thousands of cores available for parallel computation, these cores are not as flexible as CPU cores. Specifically, GPU cores implement a single instruction, multiple data (SIMD) paradigm, in which a set of 32 threads (known as a *warp*) must execute the same set of instructions while operating over separate thread registers.
(2) **Allocation** Allocating GPU memory while GPU code is executing has negative performance implications. Therefore, data structures commonly used in database systems that rely on pointer chasing like B-Trees and Tries are non-starters in programs wishing to execute on GPUs. Instead, data structures like sorted arrays, which use large contiguous blocks of memory and can pre-allocate enough memory for their use up-front, are preferred.
(3) **Coalesced Memory** In GPUs, memory accesses are fastest when threads within a warp access consecutive memory locations, a pattern known as coalesced memory access. As such, a columnar representation for relational tables helps ensure maximum utilization of GPU memory bandwidth by ensuring the common path of per-column memory operations results in coalesced accesses.

3.2 APM: A Language for Parallel Machines

APM is a low-level, assembly-style procedural language that explicitly exposes allocations and is composed solely of instructions which permit massively parallel execution, resulting in APM programs being easy to execute on GPUs. All registers in APM are vector registers that store an arbitrarily large but non-resizable and single-type collection of values. APM instructions take a destination register as their first argument, a source register as their second argument, and any additional arguments as successive parameters after that. The top level of an APM program is a loop which executes until a fix-point is reached, which we discuss further in Section 3.5. We now illustrate more features of the APM language by examining some example instructions taken from the sample APM program in Figure 7.

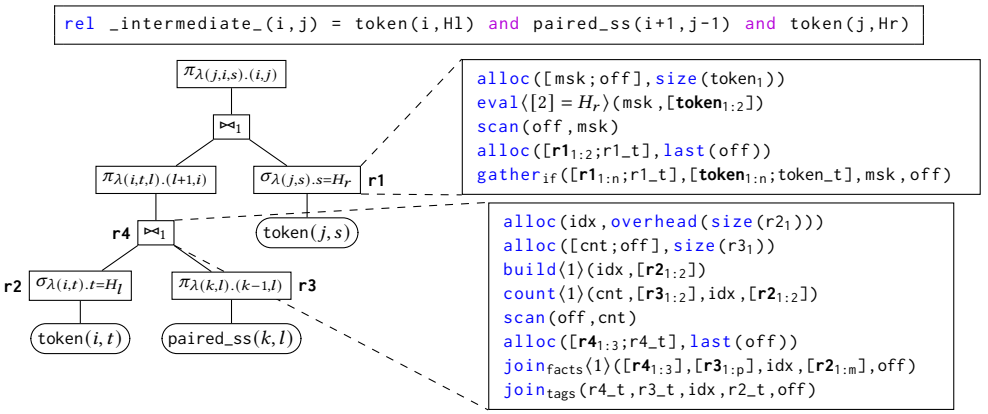


Fig. 7. In this example, we compile a part of the rule shown in Figure 3c (line 11). The code block on the top shows the Datalog rule, while bottom-left illustrates the abstract syntax tree of the RAM program compiled from it. We expand the node **r1** and **r4** on the right to show their low-level APM code.

(1) The most common instruction in APM is `alloc(n, s)`, which makes the register named n live with size s. Many uses of `alloc` can be found in Figure 7, for example to allocate the output registers [**r1**$_{1:2}$; r1_t]. Forcing allocations to be made explicit is desirable since it ensures programs which require allocating inside GPU execution are disallowed. Further, since APM lacks aliasing and backwards jumps, registers can be freed as soon as no later instruction references them, an analysis that can be done statically.
(2) Another useful instruction is `gather`$_{\texttt{if}}$`(dest,source,mask,offset)`, which copies each row of `source` for which the corresponding row of `mask` is true and copies it to the row in `dest` indicated by `offset`. `gather`$_{\texttt{if}}$ can be found in the compiled version of the `select` operation in Figure 7. This instruction is common in APM programs since there are no conditionals (indeed, there is no control flow at all besides a top level loop). Allowing conditionals allows warps to diverge, which negatively impacts performance and therefore is undesirable for APM.
(3) The operator `eval`$\langle\alpha\rangle$`(dest, source)` is used to evaluate expressions in both select and project relational algebra queries, and it is also a good example of *staged evaluation* in Lobster. When a program is being compiled to APM, some arguments can be passed at compile time to allow specialization in the resulting program. For example, α is the expression that `eval` evaluates, but this expression must be known at compile time and is baked into the program. The benefit of this is that the execution of `eval` is even faster. Figure 7 shows an example of an `eval` instruction run with the predicate $[2] = H_r$ which checks the second tuple element for equality with the constant H_r.

With this set of primitives, APM is expressive enough to support compiling a wide range of RAM queries down to APM in a way that is fully adapted to GPU computation model. We now elaborate on the RAM to APM compilation process.

3.3 Compiling RAM to APM

The process of compiling RAM to APM involves flattening a DAG (the RAM program) into a sequential list of instructions (the APM program). This flattening is implemented by providing a RAM-to-APM translation rule for each RAM operator, and then mapping this operator over a pre-order traversal of the DAG. This translation function is provided in Figure 8. Importantly, translation proceeds in the presence of a translation context F_T, also known as the EDB, which

$\pi_\alpha(\epsilon)$ Project	**let** $m = \text{width}(\alpha)$ **in**
	`alloc([O1:m;Ot],size(A1))`
	`eval⟨α⟩ ([O1:m],[A1:n;At])`
	`copy(Ot,At)`

$\sigma_\beta(\epsilon)$ Select	`alloc(Mask,size(A1))`
	`alloc(Offset,size(A1))`
	`eval⟨β⟩ (Mask,[A1:n])`
	`scan(Offset,Mask)`
	`alloc([O1:n;Ot],last(Offset))`
	`gather_if([O1:n;Ot],[A1:n;At],`
	`Mask,Offset)`

$\epsilon_1 \bowtie_n \epsilon_2$ Join	`alloc(Index, overhead(size(A1)))`
	`alloc([Count;Offset], size(B1))`
	`build⟨n⟩(Index, [A1:m])`
	`count⟨n⟩(Count, [B1:p],Index,[A1:m],)`
	`scan(Offset,Count)`
	`alloc([O1:m+p-n;Ot],last(Offset))`
	`join_facts⟨n⟩([O1:m+p-n;Ot],[B1:p;Bt],`
	`Index,[A1:m;At],Offset)`
	`join_tags([O1:m+p-n;Ot],[B1:p;Bt],`
	`Index,[A1:m;At],Offset)`

$\epsilon_1 \cup \epsilon_2$ Union	`alloc([O1:n;Ot],`
	`size(A1)+size(B1))`
	`append([O1:n;Ot],`
	`[A1:n;At],[B1:n;Bt])`

$\epsilon_1 \times \epsilon_2$ Product	`alloc([O1:n+m;Ot],size(A1)*size(B1))`
	`copy([O1:n],[A1:n])`
	`repeat([On+1:n+m],[B1:m])`
	`alloc(Lt, size(A1))`
	`alloc(Rt, size(B1))`
	`repeat(Rt,Bt)`
	`mult(Ot,At)`

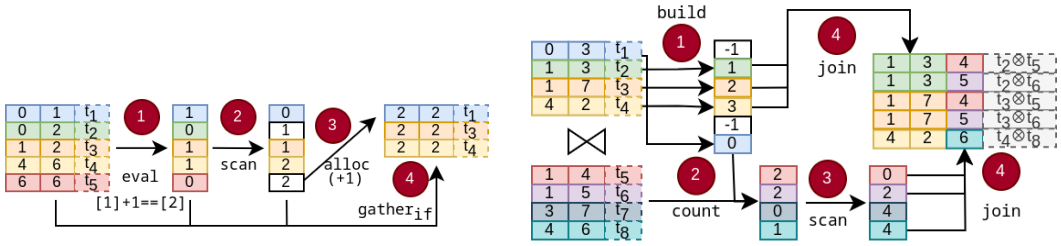
$\epsilon_1 \cap \epsilon_2$ Intersect	`alloc(Index, overhead(size(A1)))`
	`alloc([Count;Mask;Offset], size(B1))`
	`build⟨n⟩(Index, [A1:n])`
	`count⟨n⟩(Count, [B1:n],Index,[A1:n],)`
	`clamp(Mask, Count)`
	`scan(Offset,Mask)`
	`alloc([O1:n;Ot],last(Offset))`
	`gather_if([O1:n;Ot],[A1:n;At],`
	`Mask,Offset)`

Fig. 8. Functions implementing per-operator translation. Via renaming and shadowing, we assume that prior to executing instructions for a unary operator *op*, the input relation facts will be in the registers `[A1:n]` and input tags in the register `At`. Likewise with binary operators and the registers `[B1:m]` and `Bt`. All operators guarantee their output is written to the registers `[O1:r]` and output tags to register `Ot`.

contains schema necessary for applying the translations and the provenance for using the proper tag operations. We now examine two of these translation rules in detail to give examples of why the translation to APM is challenging but makes the resulting programs amenable to GPU execution.

Select. Compiling a selection operation $\sigma_\beta(\epsilon)$ for parallel GPU execution presents many challenges: for example, the location in memory of each output fact is dependent on how many prior facts evaluated to true under β. This means materializing the result relation requires coordination: there is a dependency between materializing fact i and materializing fact j for $i < j$. While this fact seems to necessitate a fully sequential execution, all hope is not lost—selection can be re-framed as a multi-step process that consists of only massively parallel operations, as illustrated in Figure 9a. First, a mask is computed by evaluating α in parallel across each fact. Second, a *scan* (also referred to as prefix sum) operation is applied to the mask. While it seems scan would have similar sequential dependencies as the naive select, it has been well studied in the literature and massively parallel single-pass, work-efficient algorithms are available [28]. Finally, a parallel conditional gather operation writes input values to the output at the location indicated by the scan, conditioning on the mask being true for that row. Importantly, the result of the scan operation also communicates the exact size of the output relation that needs to be allocated, meaning over-committing to large allocations is unnecessary and frivolous out-of-memory errors can be avoided. For an example usage of applying the `select` compilation rules, see Figure 7, which features a `select` operation that checks for equality against a constant.

Join. Potentially the most important relational operator, join ($\bowtie_n$) forms the computational core of most Lobster programs, so it is important to find an efficient implementation. Unfortunately, it is also more challenging than `select` for two reasons: (1) whereas each input fact in select can produce 0 or 1 output facts, with join each input fact can compute 0 to n output facts and (2) rather than evaluating a predicate to determine membership in the output relation, join requires a



(a) Parallel select ($\sigma_{[1]+1==[2]}$) operation.

(b) Parallel join ($\bowtie_1$) operation.

Fig. 9. An illustration of the parallelism present after translating RAM operations to APM instructions. Each of the four numbered operations must be completed sequentially, but each operation easily permits parallelism.

Algorithm 1: The `join` APM operator as an imperative procedure

Data: join width p, left relation input width $n \geq p$, right relation input width $m \geq p$, source registers [$\mathbf{L}_{1:n}$;Lt] and [$\mathbf{R}_{1:m}$;Rt], destination registers [$\mathbf{D}_{1:n+m-p:Dt}$;,] hash index H, offset table F.

Result: Destination register post-condition:

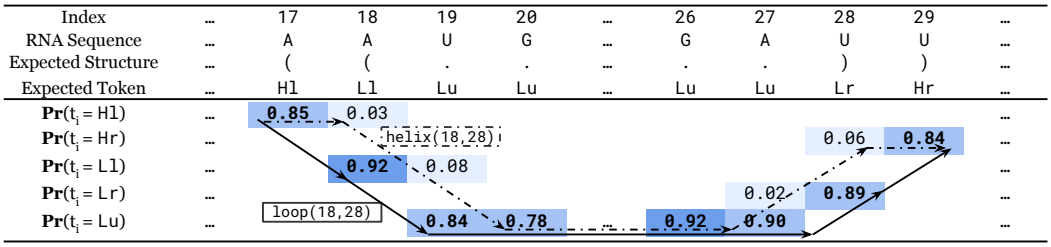
$(x_{1:n}) \in [\mathbf{L}_{1:n}] \wedge (y_{1:m}) \in [\mathbf{L}_{1:m}] \wedge (x_{1:p}) = (y_{1:p}) \Rightarrow (x_{1:n}; y_{p:m}) \in [\mathbf{O}_{1:n+m-p}]$

```
1  for i → |R1| do in parallel                  /* One thread for each row of the right relation */
2      row_r ← [R_{1:m}][i] ;                   /* Read this thread's row from the right relation */
3      position ← hash(row_r) mod |H| ;                                  /* Hash table lookup */
4      index ← H[position];
5      while index occupied do                /* Stopping condition: when the hash table lookup fails */
6          row_l = [L_{1:n}][index];
7          if row_l = row_r then              /* An equality check implements collision resolution */
8              D[F[i]] ← (row_l; row_r);
9              Dt[F[i]] ← Rt[i] ⊗ Lt[index];
10         position ← (position + 1) mod |H| ;                          /* Hash table linear probing */
```

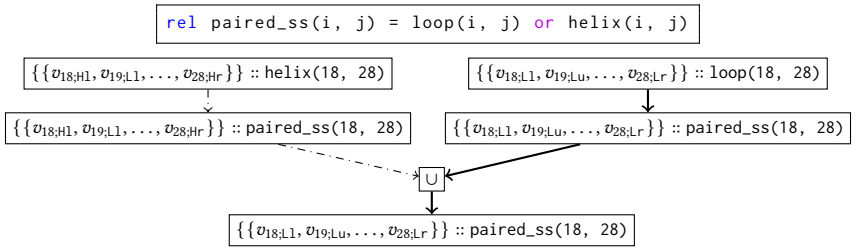
query be performed on the table being joined against. To overcome these obstacles, Lobster takes inspiration from the hash-join algorithm commonly employed by CPU databases. The process is illustrated in Figure 9b. First, a hash-table is built in parallel over the left input table. For details on the implementation of a parallel GPU hash table, see Section 5.1. Next, execution proceeds similarly to `select`: instead of building a mask via evaluating a predicate, a mask is built by checking membership in the hash table. Additionally, this mask has multiplicity, since a value may be in the hash table multiple times. This mask then undergoes a scan to calculate the output index locations. Finally, the $\texttt{join}_{\texttt{facts}}$ and $\texttt{join}_{\texttt{tags}}$ APM instructions can be issued to materialize the output in parallel—the scan result tells each thread which row to start writing at, and the mask provides the number of values to write. For full details of the $\texttt{join}_{\texttt{facts}}$ and $\texttt{join}_{\texttt{tags}}$ APM instructions, see Algorithm 1. A concrete usage of lowering `join` to APM can be seen in Figure 7, which features the compilation of a join over two binary relations.

3.4 Provenance Semiring Framework

Lobster employs a GPU-accelerated provenance semiring framework with 7 implemented semirings covering discrete, probabilistic, and differentiable modes of reasoning. Tags in GPU are stored in a separate, row-major, register which we refer to as `Rt`. Since the tags may store boolean, floating



(a) We illustrate two plausible ways of parsing the subsequence: one more probable (solid arrow) and one less probable (dashed arrow). The difference is whether $[18, 28]$ is parsed into a helix stack or a loop.



(b) According to the rule shown on top, there are two ways to derive the fact `paired_ss(18, 28)`. Notice how the facts carry the tag of top-1-proof, where a proof represent a concrete trace of structure tokens for the parse. Since we only keep the top-1 proof, only the proof on the right is propagated through the union ($\cup$) operation because it has a higher probability. Note that boolean variable such as $v_{18;\texttt{Ll}}$ denotes the variable corresponding to the probability $\Pr(t_{18} = \texttt{Ll})$.

i	j	empty	len	top-1-proof						
…	…	…	…	…	…	…	…	…	…	…
17	29	0	13	17;Hl	18;Hl	…	27;Lu	28;Lr	29;Hr	X
18	28	0	11	18;Hl	19;Lu	…	28;Lr	X	X	X
…	…	…	…	…	…	…	…	…	…	…

(c) The memory layout for `paired_ss` relation on GPU, where `i` and `j` are the two columns for the relation and the rest is the table for tags. The `empty` bit is to represent whether there exists a proof, while the `len` represents the number of literals within the proof. Similar to (b), we use `18;Ll` to denote the variable ID (an integer) for the corresponding probability.

Fig. 10. An illustration of the top-1-proof provenance in action. Consider the RNA sequence adapted from our motivating example in Fig. 2a. In (a), we show the probability of each position i being predicted as one structured token (e.g. `Hl`) by the underlying neural network. While (b) shows the derivation process of the top-1-proof for the fact `paired_ss(18,28)`, (c) illustrates the corresponding memory layout after derivation.

point, and even complex data structures like dual-numbers and boolean formulae, we need each provenance to specify a fixed size for each tag. Specifically, Lobster supports unit, max-min-prob, add-mult-prob, top-1-proof, and the differentiable versions of the probabilistic semirings.

Without loss of generality, we illustrate the top-1-proof semiring in Figure 10. It is a special case of top-k-proofs proposed in [20] and is sufficiently effective in practice. In general, the proof tracks one conjunction of the corresponding boolean variables for facts used to derive the current fact. During disjunction, the provenance picks the more likely one from the two proofs by computing the probabilities of the two proofs. For conjunction, the provenance merges the two proofs while ensuring that no conflict is seen. Figure 10c further details the memory layout for the top-1-proof semiring, where we use extra `empty` and `len` fields to track the structure of the proof. Note that in

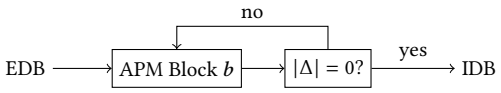


Fig. 11. Illustration of the least fix-point iteration of APM program $\texttt{lfp}(b)$.

this formulation, the size limit for a proof needs to be specified beforehand. In practice, we set the limit to 300 which is sufficient for all evaluated benchmarks that use top-1-proof semiring.

3.5 Evaluating APM Programs

While former sections describe the evaluation of APM program translated from individual RAM expressions, we must enter least fix-point iteration (`lfp`) in order for APM to support recursive relational programs. As shown in Figure 11, executing the APM program $p = \texttt{lfp}(b)$ means keep executing the main computation block b until no new fact is derived, denoted by $|\Delta| = 0$. In practice, Lobster implements semi-naive evaluation, a strategy that subsumes naive iterated rule applications by always considering an equal or smaller number of facts during each iteration. Succinctly, semi-naive evaluation involves tracking a frontier of recently discovered facts, and only applying rules to frontier facts. This avoids the redundant computation of applying rules to stale facts that are known a priori to not produce new facts. Concretely, each relation is partitioned into three sets of facts: delta facts (those that are computed during the current iteration), recent facts (those that were computed in the previous iteration), and stable facts (all other facts). After each iteration, the recent facts are merged with the stable facts and the delta facts become the recent facts. We note that this iterative process is sequential and not parallelized.

4 Optimizations

Section 3 gives a theoretical underpinning for an expressive and performant compiler and runtime. However, these ideas can be further optimized for more efficient execution and better mating with the neurosymbolic paradigm. In this section, we discuss some of these optimizations while omitting a full redefinition of the translation rules and semantics for the optimized variant of Lobster.

4.1 Batched Evaluation

A key component of deep learning is grouping samples into *batches* of samples that can be processed by the model in a single pass. To make Lobster truly pair well with deep learning as an end-to-end neurosymbolic tool, it should be aware of batching and able to process batches effectively. Surprisingly, batching is a straightforward extension of the existing semantics. Given a program $\overline{s}$ that needs to be evaluated against a batch of three databases EDB_1, EDB_2, and EDB_3, we seek a database EDB^* such that evaluating $\overline{s}$ against EDB^* is equivalent to evaluating $\overline{s}$ against each of EDB_1, EDB_2, and EDB_3. We can construct EDB^* without adding new constructs to APM or RAM by taking each relation $\rho_n \in EDB^n$ and pre-pending the "sample tag" n in a new column at the front of the relation's schema. This tag represents which sample the fact resides in. After execution, each IDB fact will have a sample tag which can be used to disambiguate results into per-sample databases that are returned to the user. The tagging process is illustrated in Figure 12.

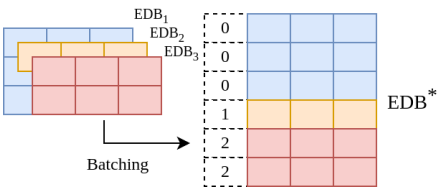


Fig. 12. How Lobster incorporates batched input data via sample tagging. Distinct colors represent distinct samples within a batch.

Some desirable ramifications that naturally arise from this framing of batching are (1) facts from separate batches cannot be joined together, so long as the width of each join operator is extended by one to include the batch tag; (2) parallelizing over each element of the batch is implicit, as the runtime already parallelizes over the rows of a relation; and (3) the additional memory footprint is minimal, since batches are seldom larger than 256 samples, meaning sample tags only take one byte of memory.

4.2 RAM Optimization Passes

While the query planning pass that converts a Datalog program to RAM is out of scope for this paper, the RAM program nonetheless has an impact on the performance of the resulting APM program. Therefore, we describe a set of RAM rewrite rules that result in a RAM program that translates to a more efficient APM program.

Inlining. Given a non-recursive rule $q \leftarrow s$ that updates relation q with the result of query s, we can rewrite all other rules to replace any usages of relation s with the query q. This has an outsized impact on APM performance, due to the high cost of incorporating new facts into IDB relations under fix-point evaluation. Inlining reduces the total number of relations, which improves performance. As always, a heuristic must be used to set a threshold for when inlining is useful. In practice, Lobster inlining relations with at most two use-sites that are the target of a single non-recursive update is a reasonable approach with a positive impact on performance.

Join index selection. Executing a `join` operation requires building a hash index on either the left or right input, which naturally leads to the question of which is preferable for optimal performance. The main insight is that if either the left or right input is part of the EDB rather than the IDB, the index should be built for that input. This is because EDB relations are constant, so the hash table can be built once and re-used during each iteration of the fix-point loop. Datalog programs are said to be "linear recursive" if each join has at most one IDB input, and we find that in practice nearly all the programs we consider are linear recursive and therefore can utilize hash indices which are built once and reused.

5 Implementation

We build Lobster with a mixture of Rust, C++, and CUDA, reusing the front-end and query planner of Scallop to limit the scope of implementation. Lobster comprises approximately 2,000 new lines of Rust code and 9,000 new lines of CUDA/C++. We now discuss implementation details that fall outside the scope of the theory of Lobster's core compiler and runtime, yet are of practical interest and importance for implementing Lobster.

5.1 Hash Table Design

A crucial component of Lobster's GPU-accelerated join algorithm is the existence of a lock-free, GPU hash table that supports parallel insertions and lookups. While many implementations are possible, ours is most directly inspired by previous work in [37]. Namely, we adopt open-addressing with linear probing for collision resolution, enabling a contiguous memory representation with no indirection. Unlike prior work, Lobster supports reasoning over relations with arbitrary width, so rather than implementing `join` with a hash table that stores fact data directly, we build hash *indices* that map back to the corresponding row of the source table for collision resolution. While this necessitates an additional random memory access to resolve collisions, it decouples the time and space complexity of the join from the width of the input relations, meaning Lobster's `join` algorithm scales with respect to the number of columns being joined on, rather than the total width of the input relations.

5.2 Bytecode Interpreter for Expression Evaluation

Projection operations are necessary and pervasive in Datalog programs, yet often do not account for as significant a portion of the runtime as other operators like `join` due to their algorithmic simplicity. Nonetheless, optimizing them does contribute to overall performance and therefore their implementation in Lobster is non-trivial. Most notably, there are two separate code paths for compiling the kernel that evaluates projection operations. If it is a "pure" project expression that permutes or subsets the columns of the relation, the expression can be evaluated as a series of columnar memory copies. More interestingly, if the expression is not "pure" and contains arithmetic or comparison of tuple elements, the expression is compiled to bytecode for a simple stack machine, and each GPU thread executes this bytecode program against one fact with a small (on the order of tens of bytes) fixed-size stack residing in shared memory. This results in improved performance compared to trying to evaluate the expression by tree walking.

5.3 Scheduling Stratum Offloading

In addition to defining data-parallel implementations of relational algebra operators, a key source of speedup for Lobster is managing the granularity and data movement associated with those computations. Relations in Lobster begin life in CPU memory, and there is notable latency involved in transferring them to GPU memory via a relatively narrow PCIe connection. Once a relation lives in GPU memory, it is advantageous to continue performing computation on the GPU side.

Lobster's GPU placement strategy starts with the longest-running stratum, which Lobster can identify either via a profiling run or static heuristics. From that longest-running stratum, we expand forwards and backwards in the static data-dependency graph to encompass adjacent strata as well, until we encounter a stratum that contains one of the small handful of operators that Lobster does not yet support (relying instead on the CPU implementation via Scallop), or the size of the stratum's inputs and outputs is small. Adopting a min-cut-like approach to GPU scheduling avoids spending excessive time in CPU-GPU transfers.

5.4 Arena Allocation

Lobster is purpose-built for neurosymbolic applications, which enables optimizations that are advantageous but which don't necessarily make sense for general purpose symbolic engines. For example, neurosymbolic applications often run in a training loop that processes an entire sample of data before clearing all data from memory and processing the next sample. After profiling Lobster, it becomes apparent that GPU memory allocations contribute a small yet non-negligible portion of total runtime.

Combining these observations, we outfit Lobster with an arena allocator: an allocation strategy that pre-allocates a large chunk of memory and then (1) fulfills `malloc` calls by bumping a pointer within that memory region and (2) fulfills `free` operations with a no-op. Arena allocators are beneficial because allocation and deallocation are essentially free, but they are troublesome because they don't reclaim memory except by freeing all allocated objects. However, we know a priori that after a training sample is processed we can clear all memory, which means Lobster is a perfect fit for an arena allocator for many of its workloads. We implement the arena allocation strategy and see a speedup of 17% on the PacMan-Maze benchmark and 7% on the Pathfinder benchmark.

6 Evaluation

We empirically evaluate Lobster with the goal of answering the following research questions:

RQ1 To what extent does Lobster improve performance in the training pipeline?
RQ2 To what extent does Lobster improve performance in the inference pipeline?

Table 1. Characteristics of benchmark tasks.

Task	Input	Logic Program	Kind	#Rules
Pathfinder	Image	Check if two dots are connected by a sequence of dashes.	Diff.	2
PacMan-Maze	Image	Plan optimal next step by finding safe path from actor to goal.	Diff.	14
HWF	Images	Parse and evaluate formula over recognized symbols.	Diff.	13
CLUTTR	Text	Deduce kinship by recursively applying composition rules.	Diff.	3
Prob. Static Analysis	Code	Compute alarms with severity via probabilistic static analysis.	Prob.	28
RNA SSP	RNA	Parse an RNA sequence according to a context-free grammar.	Prob.	28
Points-to Analysis	Code	Perform Andersen-style pointer analysis for C.	Disc.	10
Transitive Closure	Graph	Compute transitive closure of a directed graph.	Disc.	2

RQ3 What is the cost of Lobster's fully general reasoning compared to specialized CPU and GPU Datalog engines?

RQ4 How does the performance of Lobster scale with increasing problem size?

In the following sections, we introduce the benchmark tasks (Section 6.1) and the chosen baselines (Section 6.2) and then proceed to answer **RQ1** to **RQ4** in Section 6.3 through Section 6.6. All benchmarks are run on a machine with two 20-core Intel Xeon CPUs, four GeForce RTX 2080 Ti GPUs, and 768 GB RAM, with the exception of the Points-to Analysis and Transitive Closure benchmarks, which have higher VRAM requirements and were run on a machine with two 24-core Intel Xeon CPUs, eight NVIDIA A100 GPUs each with 80 GB VRAM, and 1.5 TB RAM.

6.1 Benchmarks

We evaluate Lobster on a suite of eight benchmark tasks summarized in Table 1. Since Lobster is built on a flexible framework of provenance semirings, it supports differentiable, probabilistic, and discrete reasoning. Correspondingly, we pick tasks across each of these reasoning modes to better illustrate the tradeoffs inherent in providing this flexibility. The tasks span diverse application domains: natural language processing (CLUTRR), image processing (Pathfinder and HWF), program reasoning (Prob. Static Analysis and Points-to Analysis), bioinformatics (RNA SSP), planning (PacMan-Maze), and graph databases (Transitive Closure).

The table describes each task's input, the functionality of the logic program, kind of reasoning involved ("Diff." for differentiable, "Prob." for probabilistic, and "Disc." for discrete), the lines of code (LoC) of the logic program specified in Datalog, and the number of rules. The tasks requiring differentiable reasoning are taken from Scallop's evaluation (although we omit some tasks that do not have an obvious notion of scalability), the probabilistic reasoning tasks are crafted by us for problems from the literature, and the discrete reasoning tasks mirror the evaluation of GDLog.

We next briefly describe additional characteristics of each of the tasks.

Pathfinder. This task, originally proposed in [43] and discussed briefly in Section 1, requires reasoning over an image to determine if two dots are connected by a sequence of dashes. The symbolic program extends the classic transitive closure problem to include differentiable reasoning, with the model predicting a connectivity graph and the symbolic program computing reachability over that graph. This task requires the baseline feature set to demonstrate Lobster's utility: differentiable reasoning, parallel joins, and recursion.

PacMan-Maze. In this task, a neurosymbolic reinforcement learning agent aims to solve a 2D maze given only an image of the maze. The neural portion executes a convolutional neural network to predict where in the maze enemies are and the symbolic portion plans a safe path to the goal.

This program can trivially be scaled up by increasing the size of the maze. In our formulation, it is a curriculum learning problem, in which the agent first learns in a 5-by-5 maze and then training moves to a 20-by-20 grid, with the goal of showing generalization. The Lobster program uses the `diff-top-1-proofs` provenance.

HWF. The Handwritten Formula (HWF) [22] task requires parsing and evaluating a formula of handwritten digits and operators, given supervision only on the final value. The dataset consists of formulas of varying length, meaning naive parallelism strategies like processing batch elements separately will fall short due to work imbalances. Further, the symbolic program requires support for floating point data and floating point arithmetic operations. The Lobster program uses the `diff-top-1-proofs` provenance.

CLUTRR. This is a natural language reasoning task about family kinship relations [38]. The input contains a natural language (NL) passage about a family. Each sentence in the passage hints at kinship relations. The goal is to infer the relationship between a given pair of characters. The target relation is not stated explicitly in the passage and it must be deduced through a reasoning chain. The most difficult problem in the evaluation dataset requires reasoning through a chain of length 10. The Lobster program uses the `diff-top-1-proofs` provenance.

Prob. Static Analysis. This benchmark extends the well-studied task of static program analysis with probabilistic inputs. Specifically, inputs are annotated with probabilities to reflect the system's confidence. These probabilities are propagated to the output and used to rank results in order to decrease the visibility of false positives [41]. This task uses the `minmaxprob` provenance.

RNA SSP. This task, discussed in depth in Section 2, concerns RNA Secondary Structure Prediction. It is evaluated on a set of 475 sequences of length 28 to 175 within the ArchiveII [39] dataset. Lobster uses the `diff-top-1-proofs` provenance to capture the concrete trace of structure tokens.

Points-to Analysis. This benchmark is a traditional Andersen-style points-to static analysis, where variable assignment and pointer dereferences are represented as input facts. Lobster utilizes discrete reasoning and is run on a variety of program graphs.

Transitive Closure. This benchmark computes the reachability of nodes in a graph using discrete reasoning. We evaluate on a set of graphs used by the authors of GDLog, which includes social circle graphs like ego-Facebook [32].

6.2 Baselines

There are two orthogonal axes along which a baseline makes an interesting comparison to Lobster: if it also features GPU acceleration or if it also allows advanced (differentiable or probabilistic) reasoning. As Lobster is the only framework that implements both these features, we compare against a suite of baselines that collectively implement the same feature set as Lobster.

Scallop. Scallop features flexible reasoning with provenance semirings like Lobster, but does not feature GPU acceleration and therefore struggles to scale with problem and data complexity. However, Scallop plays an important rule in our evaluation as the only system that attempts to achieve a similar level of expressivity and flexibility as Lobster.

ProbLog. ProbLog [14] features only discrete and probabilistic reasoning, and similarly does not feature GPU acceleration or CPU multi-threading. Notably, ProbLog supports Prolog, which is more expressive than Datalog and may incur extra cost due to a more complicated computation model.

Table 2. Comparison of training time for the differentiable reasoning tasks. We note that the number shown is the total time across multiple training and evaluation epochs.

Task	Pathfinder	PacMan-Maze	HWF	CLUTRR
Scallop	41 hr.	107 hr.	125 min.	134 min.
Lobster	**32 hr.**	**6.5 hr.**	**102 min.**	**111 min.**

GDLog. GDLog [42] supports only discrete reasoning, but does feature GPU acceleration. GDLog is specialized for a different sort of workload than Lobster: it targets large batch analysis jobs that may span minutes, whereas Lobster emphasizes running the same program multiple times as a component of a neurosymbolic model. Notably, GDLog does not offer a Datalog front-end and query planner, meaning that all GDLog programs are human-written, low-level, relational algebra programs. Regardless, comparing GDLog to Lobster helps see how much performance Lobster sacrifices by not specializing to one mode of reasoning.

Souffle. Souffle [34] does not support GPU acceleration or advanced reasoning modes, but does implement a high-performance multi-core CPU Datalog engine. Therefore, comparing Lobster to Souffle helps us understand the gap between GPU parallelization and CPU parallelization.

6.3 RQ1: Lobster for Training

To evaluate the extent to which Lobster improves performance in the training pipeline, we compare the total training time of Lobster against the only viable alternative, Scallop, on the four differentiable reasoning tasks. The results are shown in Table 2. For each task, training is run until convergence rather than a pre-determined number of epochs, and therefore takes a task-specific number of epochs. However, for a given task, both Lobster and Scallop take the same number of epochs.

It is evident from the results that Lobster achieves a significant speedup in training time compared to Scallop, ranging from 1.2x for CLUTRR to 16x for PacMan-Maze. Further, the reported times correspond to end-to-end training, encompassing both neural and symbolic computation. The neural computation is already heavily optimized on GPU hardware since both Scallop and Lobster offload it to Pytorch [30]. Therefore, the bulk of the training time constitutes symbolic computation. The difference in speedups across tasks is consistent with Amdahl's Law, which dictates that the performance gain of Lobster over Scallop is limited by the fraction of the overall computation that is parallelizable using Lobster. In tasks where this fraction dominates, like Pacman-Maze, Lobster results in higher speedups. Conversely, in tasks where the unparallelized parts of the symbolic computation dominate, like CLUTRR, the speedup is comparatively modest.

Finally, despite impressive speedups over the state-of-the-art baseline Scallop, the absolute training times of Lobster still leave ample room to further optimize the symbolic computation using orthogonal relational database optimizations (e.g., view materialization) which are beyond the scope of this work.

6.4 RQ2: Lobster for Inference

While training is applicable to only differentiable reasoning tasks, inference encompasses both differentiable and probabilistic reasoning tasks. To evaluate the extent to which Lobster improves the performance of such tasks at inference time, we compare it against both Scallop and ProbLog. The results are summarized in Table 3.

To capture realistic workloads at inference time, as well as to avoid high variance in measuring the inference time for a single sample, we report times for a set of samples per task. The set is input

Table 3. Comparison of inference time for all benchmark tasks.

Kind	Task	Lobster	Scallop	ProbLog	GDLog	Souffle
Differentiable	Pathfinder	**42 sec**	65 sec	×	×	×
	Pacman	**8 min**	65 min	×	×	×
	HWF	**4.5 min**	5.5 min	×	×	×
	CLUTRR	**42 sec**	155 sec	×	×	×
Probabilistic	Prob. Static Analysis	**19 min**	88 min	TO$^+$	×	×
	RNA SSP	**5.4 sec**	795 sec	TO	×	×
Discrete	Points-to Analysis	TO$^+$	TO	TO	**17.8 sec**	206 sec
	Transitive Closure	278 sec	TO$^+$	TO$^+$	**257 sec**	2300 sec

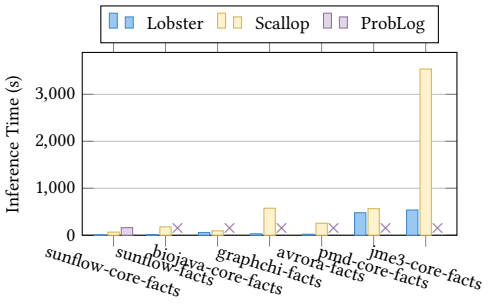


Fig. 13. Comparison of runtime on the Prob. Static Analysis task for different input programs.

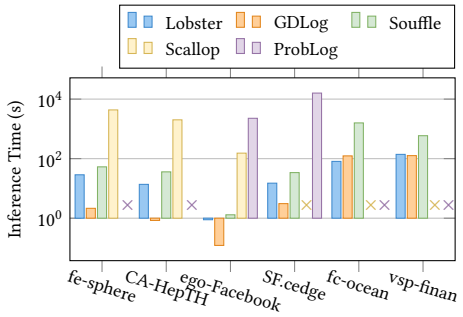


Fig. 14. Comparison of runtime on the Transitive Closure task for different input graphs.

as a single batch for each differentiable task; for each probabilistic task, it is input one sample at a time, and we report the cumulative time.

Following this setup, Pathfinder is evaluated on a set of 1216 images, the PacMan-Maze agent plays three episodes on a 15x15 grid, HWF evaluates 160 formulas of length 13, CLUTRR processes relationship graphs from 13 text passages that require long-range reasoning, probabilistic static analysis analyzes seven programs, and RNA SSP predicts the structure of 475 sequences. "TO" indicates timing out on all samples while "TO$^+$" ("partial timeout") indicates a timeout on some samples. The timeout is set to two hours per sample. "×" indicates a baseline does not have the features required to support a given task.

Lobster is consistently the most performant for all the differentiable and probabilistic tasks. ProbLog times out on both the probabilistic tasks, completing on only one of the seven programs for the Prob. Static Analysis task but timing out on even the smallest of the 475 sequences for the RNA SSP task. Since GDLog and Souffle do not support probabilistic tasks, Scallop is effectively rendered as the only viable alternative to Lobster. At its best, Lobster achieves a speedup of 146x over Scallop on RNA SSP, giving near real-time performance.

Figure 13 shows the breakdown of the runtimes (sec.) of Lobster, Scallop, and ProbLog for the Prob. Static Analysis task on the seven input programs. Using ProbLog, the analysis finishes on only one of the programs, and using Scallop, the analysis takes over 3,500 seconds in the worst case, compared to only 540 seconds for Lobster.

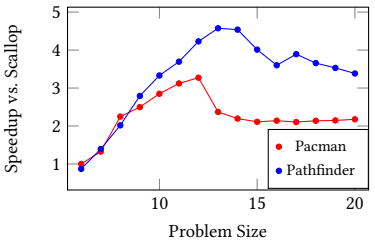


Fig. 15. Per-batch speedup displayed by Lobster over Scallop when running two benchmarks on varying problem sizes.

Task	Lobster	Scallop	Problog	GDLog	Souffle
`httpd`	OOM	TO	TO	8 sec	86 sec
`linux`	35 sec	TO	TO	1.1 sec	25 sec
`postgres`	OOM	TO	TO	8 sec	95 sec

Table 4. Comparison of runtime on the Points-to Analysis task for different programs.

6.5 RQ3: The Cost of Lobster's Flexibility

Lobster's fully general reasoning incurs a cost over CPU and GPU Datalog engines for discrete reasoning tasks. Although discrete reasoning is not Lobster's focus, measuring this cost helps establish empirical lower bounds on running time. We therefore evaluate Lobster using our two discrete reasoning tasks on a variety of different inputs: the Points-to Analysis task on three input programs (`httpd`, `linux`, and `postgres`), and the Transitive Closure task on six different input graphs. The results are shown cumulatively at the bottom of Table 3, and on individual inputs in Figure 14 and Table 4 for the two tasks.

Lobster achieves comparable performance to GDLog on the Transitive Closure task and significantly outperforms Souffle (8x speedup). Meanwhile, both Scallop and ProbLog suffer timeouts (i.e., take more than 2 hours) on a subset of the data for this task. Lastly, for the other discrete task of Points-to Analysis, Lobster suffers an out-of-memory on two of the three programs (`httpd` and `postgres`) but finishes in 35 seconds on `linux`. GDLog takes 18 seconds on average per program for this task and Souffle takes 206 seconds, but both Souffle and ProbLog suffer a timeout on all three programs. While undesirable, the most likely explanation for Lobster's memory challenges stems from the facts that features which enable flexibility in neurosymbolic reasoning like batching and semiring tags become dead weight when executing simple programs like discrete pointer analysis. A more mature implementation could attempt to minimize the impact of these features when not in use, although it seems inevitable that a more flexible solution should pay a performance penalty compared to a purpose-built system.

6.6 RQ4: Scalability of Lobster

Regardless of the reasoning mode Lobster is operating in, it should provide enhanced scalability so that as problem size increases, performance degrades gracefully rather than exponentially. This is a reasonable expectation since the thousands of cores resident on a GPU are much better suited to keep up with scale than the dozens of cores on a CPU (given that the GPU cores are utilized effectively). To that end we seek to evaluate the scalability of Lobster by measuring performance versus problem size. As shown in Figure 15, our experiment analyzes the PacMan-Maze benchmark, where the problem size can be increased by increasing the grid size, and Pathfinder, where the problem size can be increased by increasing the resolution of the analysis. Both tasks use Scallop as a baseline, as it is the only other system with adequate features to support these programs.

After sweeping over the scale parameter from 5 to 20, we see that Lobster displays improved scaling vs. Scallop, with a peak speed up over Scallop of 5×, and no data points at which Lobster is slower than Scallop. Notably, after a problem size of 13, the speedups begin to go back down. This is likely because for smaller inputs the GPU is not saturated so any additional computations can be

easily serviced, but after the problem reaches a certain size this is no longer the case and increased computational burden leads to reduced performance.

7 Related Work

While there is a wealth of work on GPU-acceleration for SQL databases in both research and industry (e.g., [19, 31]), we focus our related work discussion on systems for logic programming instead of just SQL. We relate Lobster to works along three directions: accelerated Datalog engines, probabilistic and differentiable programming, and neurosymbolic methods.

High-Performance Datalog. A variety of Datalog-based systems have been built for program analysis [15, 21, 34] and even enterprise database applications [3], though these systems run exclusively on the CPU. The GDLog system [37, 42] provides a Datalog engine implemented for GPUs, but it lacks support for the probabilistic and differentiable reasoning needed for deep learning integration. Moreover, GDLog focuses on the domain of large analytics queries, which emphasizes simpler queries executed against large databases, which is not a focus for Lobster. Notably, GDLog does not come with a query planner and user-facing front-end, meaning that users need to directly interact with low-level relational algebra operations supported by the system.

Probabilistic and Differentiable Programming. *Probabilistic programming* allows programmers to model distributions and perform probabilistic sampling and inference [5, 14, 17, 44]. *Differentiable programming* systems allow programmers to write code that is differentiable and therefore amenable to use during neural network training. Symbolic and automatic differentiation [4] are commonly used in popular ML frameworks such as PyTorch and others [1, 16, 30].

Probabilistic programs are not in general differentiable and thus cannot be run during training. The differentiable programming systems described above are designed for real-valued functions and are not compatible with logic programming. Lobster, on the contrary, focuses on the differentiability of logic programs with probabilities.

Neurosymbolic Methods. The emerging domain of neurosymbolic computation combines symbolic reasoning into existing data-driven learning systems. There have been a large number of successful neurosymbolic systems across a range of machine learning domains like computer vision and natural language processing [9–13, 22, 24–27, 29, 35, 40, 45–49, 51]. Lobster builds upon the Scallop neurosymbolic programming language [20, 23], as Scallop is general enough to implement other neurosymbolic systems [10, 26, 46, 47]. However, Lobster improves upon the CPU-only Scallop by using GPU acceleration to provide higher performance and the ability to scale to larger datasets, as we demonstrated in Section 6.

8 Conclusion

We have described the design and implementation of the Lobster neurosymbolic engine. With existing engines, symbolic computation can quickly become the bottleneck when neural computations benefit from domain-specific hardware accelerators like GPUs. Lobster shows how Datalog programs can also take advantage of GPUs, providing large speedups and strong scalability over CPU-only neurosymbolic engines like Scallop.

In the future, we plan to address some of the limitations in the current Lobster implementation. For example, we can generalize existing provenances like top-k-proofs to support larger k, and add additional semirings as well. Limited GPU memory can also become a bottleneck that prevents scaling to very large problem sizes. Techniques to spill over into CPU memory or to extend APM execution to multiple GPUs could help to alleviate this, allowing Lobster to scale even further.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc.

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *ACM International Conference on Management of Data (SIGMOD).* https://doi.org/10.1145/2723372.2742796

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. 2015. Automatic Differentiation in Machine Learning: a Survey. (2015). arXiv:1502.05767

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2018).

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications.* 243–262.

[7] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. 2021. Neurosymbolic Programming. *Foundations and Trends in Programming Languages* 7, 3 (2021). https://doi.org/10.1561/2500000049

[8] Jiayang Chen, Zhihang Hu, Siqi Sun, Qingxiong Tan, Yixuan Wang, Qinze Yu, Licheng Zong, Liang Hong, Jin Xiao, Tao Shen, Irwin King, and Yu Li. 2022. Interpretable RNA Foundation Model from Unannotated Data for Highly Accurate RNA Structure and Function Predictions. arXiv:2204.00300 [q-bio.QM] https://arxiv.org/abs/2204.00300

[9] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web Question Answering with Neurosymbolic Program Synthesis. In *ACM International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3453483.3454047

[10] Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V. Le. 2020. Neural Symbolic Reader: Scalable Integration of Distributed and Symbolic Representations for Reading Comprehension. In *International Conference on Learning Representations (ICLR).*

[11] Zeming Chen, Qiyue Gao, and Lawrence S Moss. 2021. NeuralLog: Natural language inference with joint neural and logical reasoning. *arXiv preprint arXiv:2105.14167* (2021).

[12] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding Language Models in Symbolic Languages. (2022). arXiv:2210.02875

[13] William W. Cohen, Fan Yang, and Kathryn Rivard Mazaitis. 2017. TensorLog: Deep Learning Meets Probabilistic DBs. arXiv:1707.05390

[14] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. ProbLog2: Probabilistic Logic Programming. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD).* https://doi.org/10.1007/978-3-319-23461-8_37

[15] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-up in-memory datalog processing: observations and techniques. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 695–708. https://doi.org/10.14778/3311880.3311886

[16] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. In *SysML.* https://mlsys.org/Conferences/doc/2018/146.pdf

[17] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics, (AISTATS).*

[18] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *ACM Symposium on Principles of Database Systems (PODS).* https://doi.org/10.1145/1265530.1265535

[19] heavydb 2024. Heavy.AI. https://www.heavy.ai.

[20] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. 2021. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *Conference on Neural Information Processing Systems (NeurIPS).*

[21] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[22] Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. 2020. Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning. In *International Conference on Machine Learning (ICML).* https://doi.org/10.48550/arXiv.2006.06649

[23] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1463–1487.

[24] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. Deepproblog: Neural Probabilistic Logic Programming. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[25] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2021. Neural Probabilistic Logic Programming in DeepProbLog. *Artificial Intelligence* 298 (2021). https://doi.org/10.1016/j.artint.2021.103504

[26] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. (2019). arXiv:1904.12584

[27] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584* (2019).

[28] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[29] Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. 2020. Learning Reasoning Strategies in End-to-End Differentiable Proving. In *International Conference on Machine Learning (ICML)*. arXiv:2007.06477

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems (NeurIPS)*. arXiv:1912.01703

[31] pgstrom 2024. PG-Strom. https://github.com/heterodb/pg-strom.

[32] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[33] Kengo Sato, Manato Akiyama, and Yasubumi Sakakibara. 2021. RNA secondary structure prediction using deep learning with thermodynamic integration. *Nature Communications* 12, 1 (2021), 941. https://doi.org/10.1038/s41467-021-21194-4

[34] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *International Conference on Compiler Construction (CC)*. https://doi.org/10.1145/2892208.2892226

[35] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[36] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[37] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2023. Towards Iterative Relational Algebra on the GPU. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 1009–1016. https://www.usenix.org/conference/atc23/presentation/shovon

[38] Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. 2019. CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text. (2019). arXiv:1908.06177

[39] Michael F. Sloma and David H. Mathews. 2016. Exact calculation of loop formation probability identifies folding motifs in RNA secondary structures. *RNA* 22 (2016), 1808 – 1818. https://api.semanticscholar.org/CorpusID:365048

[40] Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. 2024. Data-Efficient Learning with Neural Programs. *arXiv preprint arXiv:2406.06246* (2024).

[41] Leo St. Amour and Eli Tilevich. 2024. Toward Declarative Auditing of Java Software for Graceful Exception Handling. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 90–97.

[42] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2024. Modern Datalog on the GPU. arXiv:2311.02206 [cs.DB] https://arxiv.org/abs/2311.02206

[43] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. (2020). arXiv:2011.04006

[44] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. arXiv:1809.10756

[45] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. 2019. SATNet: Bridging Deep Learning and Logical Reasoning Using a Differentiable Satisfiability Solver. In *International Conference on Machine Learning (ICML)*. arXiv:1905.12149

[46] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. 2018. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. In *International Conference on Machine Learning (ICML)*. arXiv:1711.11157

[47] Ziwei Xu, Yogesh S Rawat, Yongkang Wong, Mohan Kankanhalli, and Mubarak Shah. 2022. Don't Pour Cereal into Coffee: Differentiable Temporal Logic for Temporal Action Segmentation. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[48] Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Neurasp: Embracing neural networks into answer set programming. *arXiv preprint arXiv:2307.07700* (2023).

[49] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. 2018. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[50] Youngmin Yi, Chao-Yue Lai, and Slav Petrov. 2014. Efficient parallel CKY parsing using GPUs. *Journal of Logic and Computation* 24, 2 (2014), 375–393. https://doi.org/10.1093/logcom/exs078

[51] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2023. Parsel: A (De-)compositional Framework for Algorithmic Reasoning with Language Models. arXiv:2212.10561