LTL learning on GPUs

Mojtaba Valizadeh[1, 2] Nathanaël Fijalkow[3] Martin Berger[2, 4]

[1] University of Sussex

[2] Neubla UK Ltd

[3] CNRS, LaBRI and Universite de Bordeaux

[4] Montanarius Ltd

Montreal, 26 July 2024

LTL_f learning in a nutshell

- **Input:** Two sets P and N of finite traces over a fixed alphabet.
- **Output:** An LTL_f formula ϕ that is
  - **sound**: all traces in P are accepted by ϕ, all traces in N are rejected by ϕ;
  - **minimal**: meaning no strictly smaller sound formula exists.

LTL_f learning in a nutshell

- **Input:** Two sets P and N of finite traces over a fixed alphabet.
- **Output:** An LTL_f formula ϕ that is
  - **sound**: all traces in P are accepted by ϕ, all traces in N are rejected by ϕ;
  - **minimal**: meaning no strictly smaller sound formula exists.

Bottom-up enumeration solves this in-the-small, but scaling is **unsolved**

LTL$_f$ learning in a nutshell

- **Input:** Two sets P and N of finite traces over a fixed alphabet.
- **Output:** An LTL$_f$ formula ϕ that is
  - **sound**: all traces in P are accepted by ϕ, all traces in N are rejected by ϕ;
  - **minimal**: meaning no strictly smaller sound formula exists.

Bottom-up enumeration solves this in-the-small, but scaling is **unsolved**

Approximate LTL$_f$ learning: formula should be not too far from minimal

Problem with GPU programming

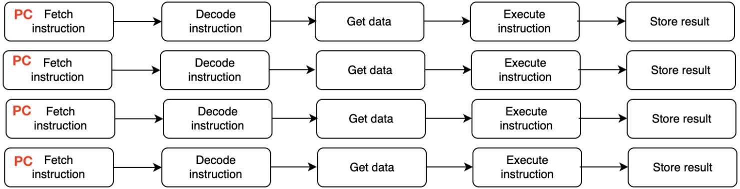
Problem with GPU programming

Hard

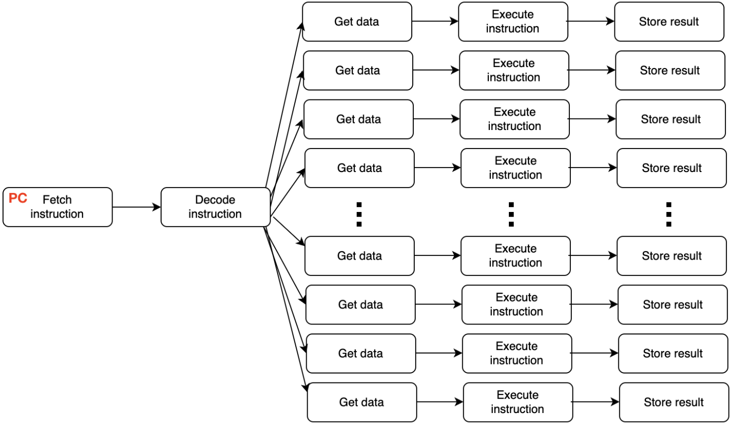
Summary: what makes program GPU-friendly?

- Minimise data movement
- Minimise data-dependent branching
- Maximise parallelism (and avoid synchronisation between threads)
- Maximise lock-step parallelism (SIMD)

CPU (highly idealised)



GPU (highly idealised)



LTL = linear temporal logic

Linear temporal logic (LTL) is widely used in industrial verification.

LTL is a modal logic for specifying properties of finite or infinite traces / strings.

LTL over finite traces (aka LTL_f) is (semantically) a strict subsystem of regular expressions ($\approx$ "aperiodic" regular expressions)

LTL, linear temporal logic

LTL formulae over $\Sigma = \{p_1, ..., p_n\}$ are given by the following grammar.

$$\phi \quad ::= \quad p_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid \phi \,\mathsf{U}\, \phi$$

We assume a simple cost$(\cdot)$ function that gives the cost of each formula. (E.g. size)

LTL$_f$ semantics: satisfaction relation

Let *tr* be a **finite** trace and ϕ a formula.

$$tr, i \models \phi$$

LTL$_f$ semantics: satisfaction relation

Let *tr* be a **finite** trace and ϕ a formula.

$$tr, i \models \phi$$

- $tr, i \models p$ if $p \in tr(i)$
- ...
- $tr, i \models \mathrm{X}\phi$, if $tr, i+1 \models \phi$,
- $tr, i \models \mathrm{F}\phi$, if there is $i \leq j < \mathrm{len}(tr)$ with $tr, j \models \phi$,

LTL$_f$ semantics: language of a formula

Each ϕ induces languages:

$$\mathrm{Lang}(\phi, i) = \{tr \mid tr, i \models \phi\} \qquad \mathrm{Lang}(\phi) = \mathrm{Lang}(\phi, 0)$$

LTL_f -learning by program synthesis on a GPU.

Program synthesis

Algorithmic generation of syntactic entities from specifications.

Dominant flavours:

- PBE (= programming by example), where the input is a set of examples
- PBF (= programming by formula), where the input is a logical formula

Program synthesis

Naive algorithm: bottom-up enumeration

```
def enumerate(P, N):
    cost = 0
    while true:
        phi = next_formula(cost)
        if phi satisfies (P, N):
            return with phi
        cost += 1
```

aka generate-and-filter, guess-and-check, ...!

```
if enum(0) satisfies (P, N) then return enum(0);

if enum(1) satisfies (P, N) then return enum(1);

if enum(2) satisfies (P, N) then return enum(2);

if enum(3) satisfies (P, N) then return enum(3);

if enum(4) satisfies (P, N) then return enum(4);

if enum(5) satisfies (P, N) then return enum(5);

...
```

```
if enum(0) satisfies (P, N) then return enum(0)
   PAR
if enum(1) satisfies (P, N) then return enum(1)
   PAR
if enum(2) satisfies (P, N) then return enum(2)
   PAR
if enum(3) satisfies (P, N) then return enum(3)
   PAR
if enum(4) satisfies (P, N) then return enum(4)
   PAR
if enum(5) satisfies (P, N) then return enum(5)
   PAR
...
```

Embarrassingly parallel!

Branch-free algorithms and data-structures: search space

We are going a (refined variant of) bottom-up enumeration, so each step needs to be **fast**!

Branch-free algorithms and data-structures: search space

Search space	**Representation**	**Data Structure**	**Issue**
Formula	Tree	Pointers	Slow, redundant
Language	$\Sigma^* \to \mathbb{B}$	–	Infinite
Language up to $P \cup N$	$(P \cup N) \to \mathbb{B}$	Bitvector	Non-compositional
Language up to $cl(P \cup N)$	$cl(P \cup N)) \to \mathbb{B}$	Bitvector	More space

Branch-free algorithms and data-structures: search space

Need to prove $\phi \vDash (P, N)$ so quotient formulae by equality up to $\mathrm{Lang}(\phi) \cap (P \cup N)$

Branch-free algorithms and data-structures: search space

Need to prove $\phi \vDash (P, N)$ so quotient formulae by equality up to Lang$(\phi) \cap (P \cup N)$

$(P \cup N) \to \mathbb{B}$ is (isomorphic to) bitvector, assuming a fixed **total order** on $P \cup N$.

$$\mathbf{1}_L : \quad P \cup N \quad \to \quad \mathbb{B}$$

Branch-free algorithms and data-structures: search space

Need to prove $\phi \models (P, N)$ so quotient formulae by equality up to Lang$(\phi) \cap (P \cup N)$

$(P \cup N) \rightarrow \mathbb{B}$ is (isomorphic to) bitvector, assuming a fixed **total order** on $P \cup N$.

$$\mathbf{1}_L : \quad P \cup N \quad \rightarrow \quad \mathbb{B}$$

Positive = $\{1, 011, 1011, 11011\}$
Negative = $\{\epsilon, 10, 101, 0011\}$

Bitvector representation of ϕ

For trace *tr* of length *n* satisfaction is isomorphic to bitvector *bv* of same length.

$$bv(i) = \begin{cases} 1 & tr, i \vDash \phi \\ 0 & \text{else} \end{cases}$$

Bitvector representation of ϕ

For trace *tr* of length *n* satisfaction is isomorphic to bitvector *bv* of same length.

$$bv(i) = \begin{cases} 1 & tr, i \models \phi \\ 0 & \text{else} \end{cases}$$

Example for tr = “squeegee” and atomic proposition g:

ϕ	bv
g	00000100
Xg	00001000
XXg	00010000
XXXg	00100000
XXXXg	01000000
XXXXXg	10000000
XXXXXXg	00000000

Bitvector representation of ϕ

For trace *tr* of length *n* satisfaction is isomorphic to bitvector *bv* of same length.

$$bv(i) = \begin{cases} 1 & tr, i \models \phi \\ 0 & \text{else} \end{cases}$$

Example for *tr* = "squeegee" and atomic proposition *g*:

ϕ	*bv*
g	00000100
Xg	00001000
XXg	00010000
XXXg	00100000
XXXXg	01000000
XXXXXg	10000000
XXXXXXg	00000000

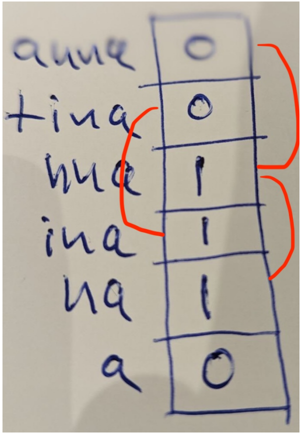
Note:

- Bitvector is suffix-closed
- Bitshifts only implements X branch-free
- Bitshifts are machine instructions
- Bitshifts assume locality

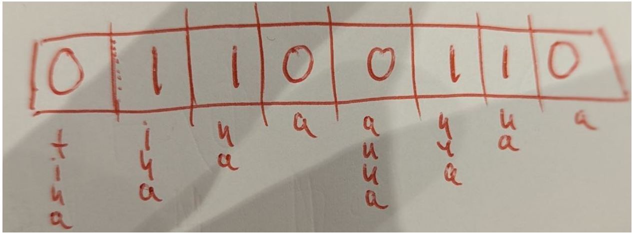
Bitvector representation of ϕ

Suffix closure of $P \cup N = \{an\underline{na}, ti\underline{na}\}$ is $\{tina, ina, na, a, anna, nna\}$

Irredundant



Redundant



- Preserves locality
- Allows compositional construction of (representations of) LTL_f formulae
- Space/time trade-off

Branch-free X

Let *bv* represent formula ϕ. Want bitvector for Xϕ.

Branch-free X

Let *bv* represent formula ϕ. Want bitvector for Xϕ.

```
def branchfree_X(bv):
    return bv << 1
```

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

```
def branchfree_F(bv):
    L = len(bv)
    for i in range(log(L)+1):
        bv |= bv << 2**i
    return bv
```

Branch-free F

Let *bv* represent formula ϕ. Want bitvector for Fϕ.

```
def branchfree_F(bv):
    L = len(bv)
    for i in range(log(L)+1):
        bv |= bv << 2**i
    return bv
```

```
def branchfree_F(bv):  // Assume length(bv) == 64
        bv |= bv << 1
        bv |= bv << 2
        bv |= bv << 4
        bv |= bv << 8
        bv |= bv << 16
        bv |= bv << 32
    return bv
```

Branch-free U

```
def branchfree_U(bv1, bv2):
    L = len(bv1)
    for i in range(log(L)+1):
        bv2 |= bv1 & (bv2 << 2**i)
        bv1 &= bv1 << 2**i
    return bv2
```

```
def branchfree_U(bv1, bv2):  // Assume length(bv1) == 64
        bv2 |= bv1 & (bv2 << 1)
        bv1 &= bv1 << 1
        bv2 |= bv1 & (bv2 << 2)
        bv1 &= bv1 << 2
        bv2 |= bv1 & (bv2 << 4)
        bv1 &= bv1 << 4
        bv2 |= bv1 & (bv2 << 8)
        bv1 &= bv1 << 8
        bv2 |= bv1 & (bv2 << 16)
        bv1 &= bv1 << 16
        bv2 |= bv1 & (bv2 << 32)
    return bv2
```

Complexity

Theorem

Algorithm implements the LTL_f semantics branch-free in $O(\log n)$ time (n trace length), assuming bitwise boolean operations and shifts by powers of 2 have costs.

Previous implementations are $O(n^2)$ or worse

Main loop (1)

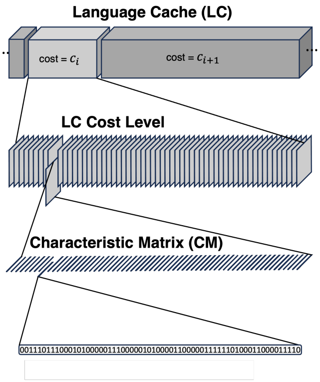
```
language_cache = []

def enum(p, n, cost):
    if (p, n) can be solved with Atom then return Atom
    language_cache.append([Atom])
    for c in range(cost(Atom)+1, cost(overfit(p, n))):
        language_cache.append([])
        for op in [F, U, G, X, And, Or, Not]:
            handleOp(op, p, n, c, cost)
    return overfit(p, n)
```

Main loop (2)

```
def handleOp(op, p, n, c, cost):
    match op:
        case F:
            for all phi in language_cache(c-cost(F)):
                phi_new = branchfree_F(phi)
                if phi_new |= (p, n): then exit(phi_new)
                if phi_new is unique in language_cache:
                    language_cache[c].append(phi_new)
        case U:
            ...
```

Redundancies of syntax



We cache bitvectors (representing formulae) in a (read-only) language cache. Why?

Problem: LTL_f operators don't reserve uniqueness.

If bv_1 represents ϕ and bv_2 represents ψ, both are unique, i.e., have not been seen before, then it is **not** guaranteed that the bitvector representing $\phi \, \mathsf{U} \, \psi$ is unique.

Uniqueness check of newly constructed (representation of) formula. Using fast hashing library. **Most expensive part of search.**

Density conjecture

Conjecture

Density of unique formulae among all formulae is 0

Explosive growth of language cache main scaling limit. Two solutions

- Relaxed uniqueness check
- Divide-and-conquer

Relaxed uniqueness

(Pseudo-)Randomly reject **unique** representations of formulae from language cache

Relaxed uniqueness

(Pseudo-)Randomly reject **unique** representations of formulae from language cache

This is sound: if we find ϕ that satisfies (P, N) we are done

But might increase size of returned formula.

I ♡ exponential algorithms

Divide & conquer

Relaxed uniqueness checks, and bitvector representation are not enough to improve on our scalability issue w.r.t. memory.

Divide & conquer

If (P, N) is too big, split (P, N), into disjoint (P_i, N_j) for $i, j = 1, 2$, such that

$$P = P_1 \cup P_2 \qquad N = N_1 \cup N_2$$

Learn recursively:

- $\phi_{11} = \text{synth}(P_1, N_1)$
- $\phi_{12} = \text{synth}(P_1, N_2)$
- $\phi_{21} = \text{synth}(P_2, N_1)$
- $\phi_{22} = \text{synth}(P_2, N_2)$

Combine all into $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$

Not guaranteed to be minimal

Divide & conquer

Interesting variant: probabilistic sampling from (P, N) effective.

Two sources of losing minimality

- Divide & conquer
- Relaxed uniqueness checks

For small (P, N) we don't need those and our algorithm learns minimal formula.

Benchmarks

Existing benchmarks are too easy, essentially all solved within measurement threshold.

We made various new benchmarks. Please use them.

Comparison with SOTA (Scarlet)

(# P, # N)	Our impl. Time (Cost)	Scarlet Time (Cost)
$(2^3, 2^3)$	0.31s (12)	1532.85s (19)
$(2^4, 2^4)$	0.32s (12)	1463.67s (17)
$(2^5, 2^5)$	0.36s (12)	2867.47s (17)
$(2^6, 2^6)$	0.34s (12)	5691.98s (17)
$(2^7, 2^7)$	0.63s (20)	OOM
$(2^8, 2^8)$	0.95s (19)	OOM
$(2^9, 2^9)$	0.72s (19)	OOM
$(2^{10}, 2^{10})$	1.09s (19)	OOM
$(2^{11}, 2^{11})$	1.32s (19)	OOM
$(2^{12}, 2^{12})$	1.66s (19)	OOM
$(2^{13}, 2^{13})$	2.46s (19)	OOM
$(2^{14}, 2^{14})$	4.62s (20)	OOM
$(2^{15}, 2^{15})$	8.35s (19)	OOM
$(2^{16}, 2^{16})$	15.52s (19)	OOM
$(2^{17}, 2^{17})$	30.49s (19)	OOM

Future

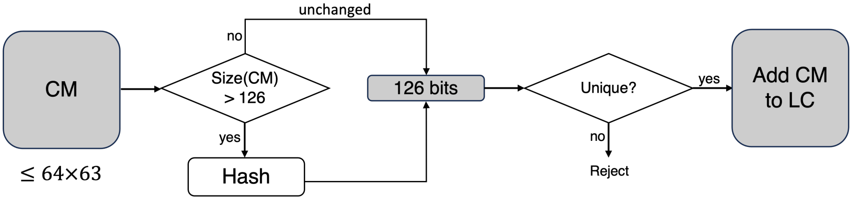
- Almost nothing we do in this paper is tied to LTL_f . Almost any program synthesis approach should be re-implemented on GPUs. In the future the performance gap between CPUs and GPUs will grow!
- Scaling to richer languages.
- Need new form of computational complexity that is predictive for modern hardware.
- Programming language support for GPUs needs improvement.

Thank you!

I ♥ GPUs

Good talk from PLDI 2024 about programming contemporary compute `https://www.youtube.com/live/66oKqvwoIv0?t=1238s`

Relaxed uniqueness



We implement (pseudo-)random decision by hashing:

We check for uniqueness not using full bitvectors, but bitvectors hashed to k bits.

Choice of $k = 126$ bits is pragmatic, in experiments there is **unexplained** phase transition at around 70 bits.

If size of bitvector is ≤ 126 then our LTL_f -learner is precise: returns minimal formula.

Density conjecture

Fix an enumeration $\#$ of all LTL$_f$ formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n\to\infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

Density conjecture

Fix an enumeration $\#$ of all LTL$_f$ formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n\to\infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

I doubt this depends on the chosen notion of cost / enumeration either.

Noisy-learning conjecture

Conjecture

If we learn with an allowed ϵ fraction of misclassified strings from (P, N), then learning becomes easier in a way that is exponential in ϵ.

Here easier means: the size of the bottom-up construction of formulae, before the first solution is hit, shrinks.

- L. Pitt, M. K. Warmuth, The minimum consistent DFA problem cannot be approximated within any polynomial. (1989)
- M. Kearns, L. Valiant, Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. (1994)

Density conjecture

Fix an enumeration $\#$ of all LTL_f formulae (resp. aperiodic languages) over Σ.

Definition

$L = \#(n)$ is **unique** if for all $i < n$, $L \neq \#(i)$. ϕ is **unique** if ϕ's language is unique.

Conjecture

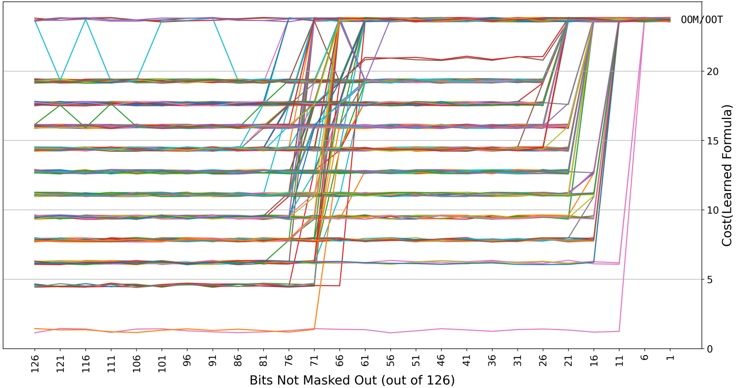
In the limit the density of unique formulae among all formulae is 0:

$$\lim_{n \to \infty} \frac{\#\{\phi \mid \mathrm{cost}(\phi) \star n, \phi \text{ unique}\}}{\#\{\phi \mid \mathrm{cost}(\phi) < n\}} = 0$$

where $\star$ ranges over $=, <$. (Mutatis mutandis for aperiodic languages)

Phase transition conjecture

Recall: relaxed uniqueness checks map candidate to *k* bits. The smaller the *k* the bigger the resulting learned formula. Experiment:



Conjecture

This is a phase transition.