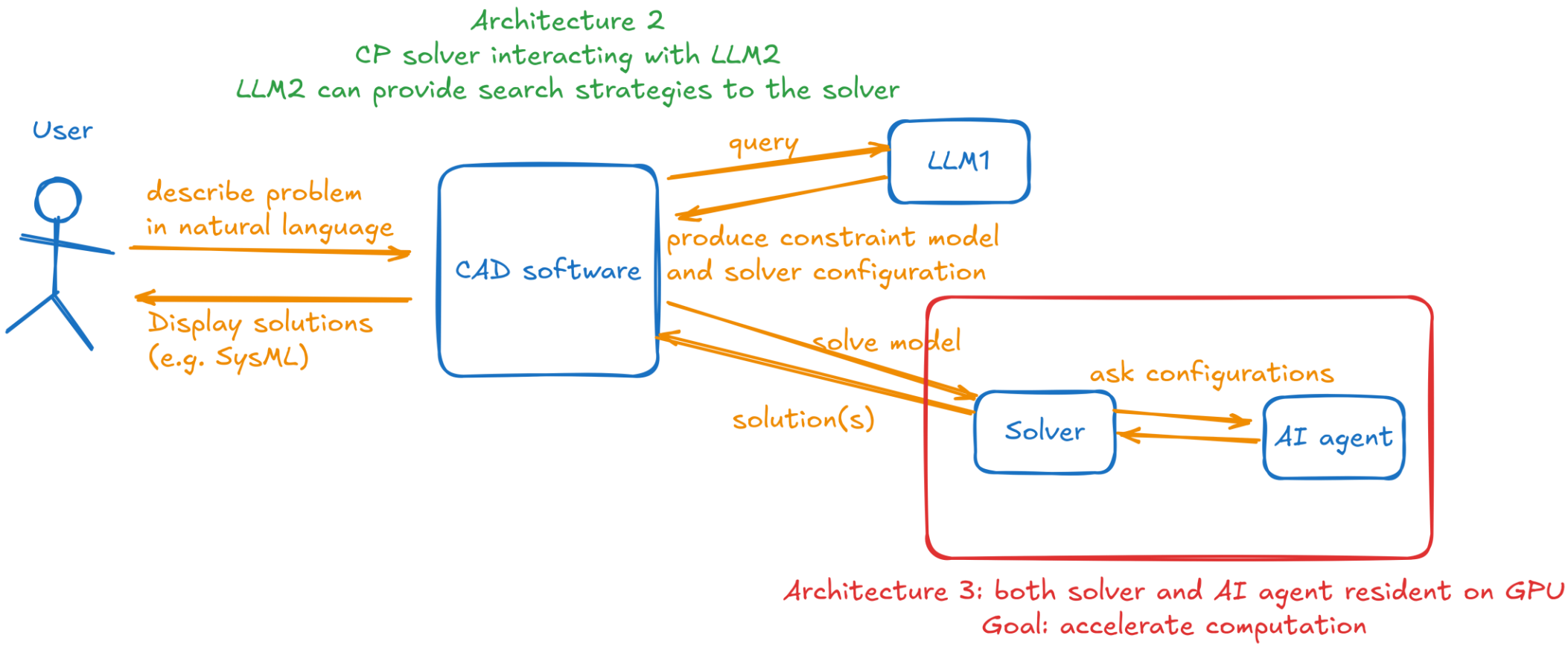
Towards a Neuro-Symbolic Constraint Solver

Turbo 1.3 and Beyond!

NEURESA Workshop 2026
Pierre Talbot, University of Luxembourg
8 July 2026

Neuro-Symbolic Architecture (NEURESA)



Challenge of Auto-configurability

- For a LLM to configure the solver, we need….
  - To expose various solving options (command-line arguments)

```
ptalbot@pierre-machina:~/repositories/lattice-land/turbo$ ./build/gpu-release-local/turbo
usage: ./build/gpu-release-local/turbo [-t 2000] [-a] [-n 10] [-i] [-f] [-s] [-v] [-p <i>] [-arch <cpu|hybrid|gpu|barebones>] [-p 48] [-or 48] [-sub 12] [-stack 100] [-fp <ac1|wac1>] [
-wac1_threshold 0] [-eps_var_order <input_order|first_fail|anti_first_fail|smallest|largest>] [-eps_value_order <min|max|split|reverse_split>] [-seed 0] [-network_analysis] [-cutnodes
0] [-disable_simplify] [-force_ternarize] [-globalmem] [-version 1.0.0] [xcsp3instance.xml | fzninstance.fzn]
```

- But:
  - so far, not so many… most of them are irrelevant (e.g. -ast, -v, …)
  - **adding more functionalities?**
    (Wei? Anisa? Hakan? Yi-Nung? Clément? Ching-Yao? Me?)

Hardware Restriction

- A lot of functionalities mean **more code.**
  - More memory used.
  - More registers used (can decrease occupancy as shown by Wei).
  - Long compilation time, even not compiling anymore… bug nvcc in earlier version of Turbo.

We need more functionalities to leverage LLM, but more code decrease efficiency….

Solutions

- **Pre-compile many kernels** specialized to a subset of options.
  - Wei already shows the feasibility with the fixpoint algorithm option as a template parameter.
  - Combinatorial explosion?
- **KISS principle**: avoid integrating functionalities for which the performance improvement is low, or complex without big improvement.
- **Code as data?**
  - Instead of hard-coding the options, provide small building blocks that can be assembled through an interpreted configuration language.
  - The interpreter is compiled and optimized inside the kernel (keep it small and expressive).
  - Programs can be created by the LLM on-the-fly without recompilation.

Configurability Language

- **Not easy to design**: what are the building blocks?
  - Within CP, many work on search strategy language, nothing really worked/was adopted.
  - Within GPU programming, also quite hard, only language for specific tasks, e.g. Halide for image processing, GRAFS for graph processing, …
- **But a fun and interesting research direction.**
  - Building blocks: lattice data structure, monotone functions, fixpoint engine.
  - Combinators: Product of lattices, sequencing fixpoint engines, …

Example

```
use ItvAD<int> as zitv;
use ItvAD<float> as fitv;
use Octagon<int> as zoct1;
use Octagon<int> as zoct2;

gfp (
  gfp (zitv; fitv);
  gfp zoct1;
  gfp zoct2;
)
```

Connection to the model

```
vars int x[10] in zitv, zoct;
vars int y[20] in zitv;
..

constraint x[0] = x[1] + x[2] in zoct;
constraint x[0] = x[4] * x[5] + 12 in zitv;
...
```

Pushing further: fixpoint scheduler

```
use ItvAD<int> as zitv;
use Octagon<int> as zoct;

use scheduler AC1 as ac1;
use scheduler WAC1 as wac1;
use scheduler GaussSiedel as gs;

gfp(gs) (
 gfp(wac1) zitv;
 gfp(ac1) zoct;
)
```

But first… we need the building blocks

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26, C++29 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26, C++29

Language
- Preprocessor – Comments
- ASCII chart
- Basic concepts
  - Keywords
  - Names (lookup)
  - Types (fundamental types)
  - The `main` function
  - Modules (C++20)
  - Contracts (C++26)
- Expressions
  - Value categories
  - Evaluation order
  - Operators (precedence)
  - Conversions – Literals
  - Constant expressions
- Statements
  - `if` – `switch`
  - `for` – range-`for` (C++11)
  - `while` – `do`-`while`
- Declarations – Initialization
- Functions – Overloading
- Coroutines (C++20)
- Classes (unions)
- Templates – Exceptions
- Freestanding implementations

Standard library (headers)

Named requirements

Language support library
- Program utilities
  - Signals – Non-local jumps

Diagnostics library
- Assertions – System error (C++11)
- Exception types – Error numbers
- `basic_stacktrace` (C++23)
- Debugging support (C++26)

Memory management library
- Allocators – Smart pointers
- Memory resources (C++17)

Metaprogramming library (C++11)
- Type traits – `ratio`
- `integer_sequence` (C++14)
- Reflection (C++26)

General utilities library
- Function objects – `hash` (C++11)
- Swap – Type operations (C++11)
- Integer comparison (C++20)
- `pair` – `tuple` (C++11)
- `optional` (C++17)
- `expected` (C++23)
- `variant` (C++17) – `any` (C++17)
- `bitset` – Bit manipulation (C++20)

Containers library
- `vector` – `deque` – `array` (C++11)
- `list` – `forward_list` (C++11)
- `inplace_vector` (C++26)
- `hive` (C++26)
- `map` – `multimap` – `set` – `multiset`
- `unordered_map` (C++11)
- `unordered_multimap` (C++11)
- `unordered_set` (C++11)
- `unordered_multiset` (C++11)

Strings library
- `basic_string` – `char_traits`
- `basic_string_view` (C++17)

Text processing library
- Primitive numeric conversions (C++17)
- Formatting (C++20) – Localization
- `text_encoding` (C++26)
- Regular expressions (C++11)
  - `basic_regex` – Algorithms
  - Default regular expression grammar
- Null-terminated sequence utilities:
  - byte – multibyte – wide

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Data-parallel types (SIMD) (C++26)
- Pseudo-random number generation
- Floating-point environment (C++11)
- `complex` – `valarray`

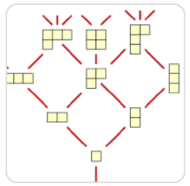
Date and time library
- Calendar (C++20) – Time zone (C++20)

Input/output library
- Print functions (C++23)
- Stream-based I/O – I/O manipulators
- `basic_istream` – `basic_ostream`
- Synchronized output (C++20)
- File systems (C++17)

Concurrency support library (C++11)

Standard libraries for lattice data structures?

The ambition of lattice-land



Lattice land

Collection of lattice-based data structures compatible with GPUs. Powering up the constraint solver Turbo!

Pinned

Customize pins

cuda-battery Public

Abstractions of memory, allocator, vector, tuple, shared_ptr, unique_ptr, bitset, variant and string working on both CPU and GPU

C++ 32 4

lala-pc Public

Abstract propagators completion (PC) is an abstract domain encapsulating the propagation component of a constraint programming solver.

C++ 3 5

lala-power Public

Abstract domains based on powerdomain constructions. It includes search tree and branch and bound abstract domains.

C++ 2

lala-octagon Public

Octagon abstract domain

C++ 1

New! Lala-interval

https://github.com/lattice-land/lala-interval

- Soon ready to integrate into Turbo.
- **Ambition**: not only designed only for Turbo! For abstract interpreters, distributed computing, parallel programming, ...

- Primitive types:
  - Instead of integers: LB<int> or UB<int> --> is your integer going up or down?
  - Instead of float: LB<float> or UB<float> --> is your float going up or down?
  - Instead of Boolean: LB<bool> or UB<bool> --> is your Boolean going from false to true, or true to false?
- Reduced products of primitive types: ZInterval, FInterval
  - Come with interval arithmetic functions (monotone functions)
  - Short term: publish a paper on the best propagator for integer division.

Next! Lala-collection

What's std::vector, std::set, ... in lattice-land???

- **Powerset construction**: P(Z) is a collection of integers, {1,2,4,6} <= {1,2,3,4,5,6}
- What if we want **P(ZInterval)** instead? {[1,2], [4,6]} <= {[1,6]} ???
  - Well-defined by Hoare/Smyth powerdomains
  - Guess what? That's what we want for PIR and symbolic constraint manipulation, e.g. {y = 9, x = 1} <= {x + y = 10, x = 1} <= {x = 1}
- **Vector of lattice elements**: our current VStore. Mathematically, the lattice of functions Z -> L,

WIP! Lala-fixpoint

- Generic fixpoint engines (GaussSeidel, AC1, WAC1, ...)
- Generic algorithm built on those fixpoint engines, e.g. min/max/BFS/...

https://github.com/lattice-land/lala-fixpoint

Conclusion

- Many possible directions, only possible to explore most of them because of your work!

Build strong foundation, no need to rush.

- Step by step, we define the building blocks and combinators.
- Once it is ready, we can propose a configurability language on top of them.
- And LLM can directly write in this language, choosing the right constraints / abstract domains / fixpoint engines for a specific problem.