Turbo: Design of a Constraint Programming Solver on GPU

Pierre Talbot
pierre.talbot@uni.lu

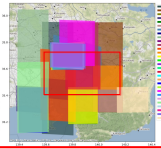
8th April 2025

University of Luxembourg

Constraint Programming

- **Declarative paradigm**: specify your problem and let the computer solves it for you.
- **Many applications**: scheduling, bin-packing, hardware design, satellite imaging, ...
- **Constraint programming** is one approach to solve such combinatorial problems.
- Other approaches include SAT, linear programming, SMT, MILP, ASP,...

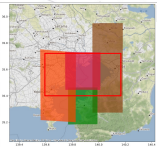
Satellite image mosaic



After a query with certain parameters:

N = 30 satellite images

Find the cover with the minimum number of images (NP-Hard)



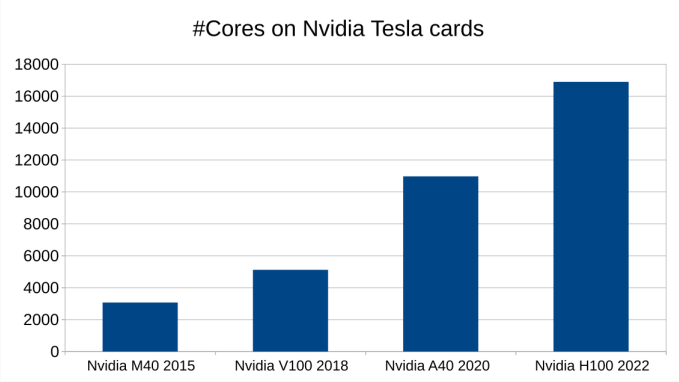
Build the mosaic

1

[1] Combarro et al., Constraint Model for the Satellite Image Mosaic Selection Problem, CP 2023

Why CP on GPU?

CPU clock speed is stagnating, GPU #cores is increasing quickly each year.



Easy speed-up: same code but faster.

Why CP on GPU?

"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin." [a]

[a] The Bitter Lesson, Rich Sutton, 2019,
http://www.incompleteideas.net/IncIdeas/BitterLesson.html

Why CP on GPU?

- Machine learning (deep learning, reinforcement learning, ...) has seen tremendous speed-ups (e.g. 100x, 1000x) by using GPU.
- Some (sequential) optimizations on CPU are made irrelevant if we can explore huge state space faster.

Can we replicate the success of GPU on machine learning applications to combinatorial optimization?

cuOpt: Nvidia Routing Optimization

Record-Breaking NVIDIA cuOpt Algorithms Deliver Route Optimization Solutions 100x Faster

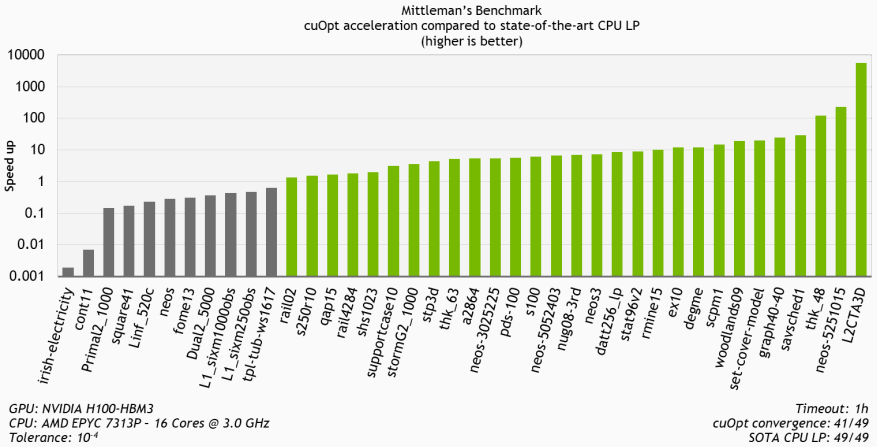
Mar 20, 2024

+20 Like Discuss (0)

By Akif Çördük, Piotr Sielski and Moon Chung

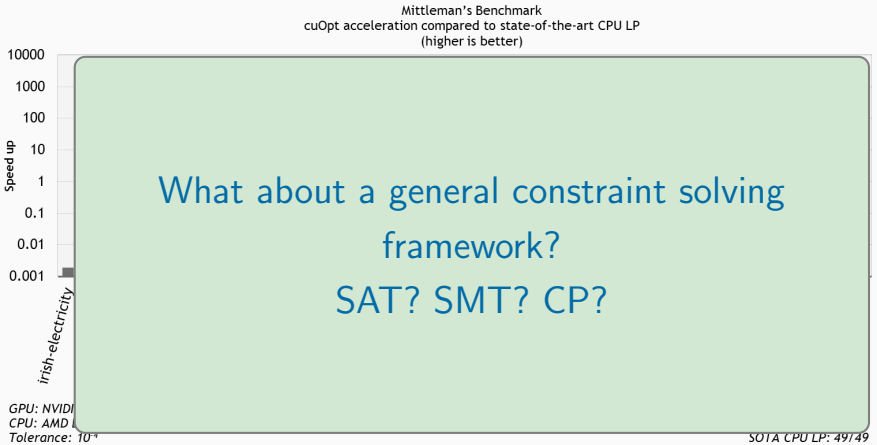
Cool, but specialized combinatorial algorithm for routing.

cuOpt: Nvidia Linear Programming Solver



Better! But only for linear constraints over continuous domain.

cuOpt: Nvidia Linear Programming Solver



What about a general constraint solving framework?

SAT? SMT? CP?

Better! But only for linear constraints over continuous domain.

Combinatorial Optimization on GPU

Very scarce literature, usually:

- **Heuristics**: often population-based algorithms[2].
- **Limited set of problems**[3]
- **Limited GPU parallelization**: offloading to GPU specialized filtering procedures[4,5].
- **Limited expressivity**: solver with max 256 set variables[6].

For CP-based approach: not general and no proof of correctness.

[2] A. Arbelaez and P. Codognet, *A GPU Implementation of Parallel Constraint-Based Local Search*, PDP, 2014.
[3] Jan Gmys. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. INFORMS Journal on Computing, 2022.
[4] F. Campeotto et al., *Exploring the use of GPUs in constraint solving*, PADL, 2014
[5] F. Tardivo et al., *Constraint propagation on GPU: A case study for the AllDifferent constraint*, Journal of Logic and Computation, 2023.
[6] A. Dovier et al., *CUDA: Set Constraints on GPUs*, 2022.

Where is the Challenge?

A constraint solver on GPU is difficult because:

- Embarassingly parallel (one subproblem per thread) is not efficient on GPU due to limited cache per thread.
- Constraints share variables $\Rightarrow$ shared-memory parallelism $\Rightarrow$ synchronization and efficiency issues.

Our Contributions

Theoretical Foundation

We propose a new parallel model of computation:

- **Correct**: formal proofs of correctness w.r.t. the parallel load/store operations on memory[7].
- **Simple and lock-free**: no mutex, no complicated atomic primitives, just barriers.
- **Expressive**: a process algebra based on *concurrent constraint programming* which supports $\mathbb{Z}, \mathbb{R}, \mathcal{P}(\mathbb{Z}), \ldots$ domains.

[7] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

Our Contributions

Theoretical Foundation

We propose a new parallel model of computation:

- **Correct**: formal proofs of correctness w.r.t. the parallel load/store operations on memory[7].
- **Simple and lock-free**: no mutex, no complicated atomic primitives, just barriers.
- **Expressive**: a process algebra based on *concurrent constraint programming* which supports $\mathbb{Z}, \mathbb{R}, \mathcal{P}(\mathbb{Z}), \ldots$ domains.

Turbo: First general constraint solver fully executing on GPU.

- **General**: Support MiniZinc and XCSP3 constraint models (currently only $\mathbb{Z}$ variables).
- **Simple**: Solving algorithm from 50 years ago.
- **Efficient?**: Almost on-par with Choco (23% better, 28% worst, 49% equal).
- **Open-source**: Publicly available on `https://github.com/ptal/turbo`.

[7] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

What is in This Presentation?

On the menu:

1. Constraint Programming (CP)
2. Towards a **correct** GPU constraint solver.
3. Towards an **efficient** GPU constraint solver.

Constraint Programming

Constraint Network

Constraint Network

Let X be a finite set of variables and C be a finite set of constraints. A *constraint network* is a pair $P = \langle d, C \rangle$ such that $d \in X \to Itv$ is the *domain* of the variables where Itv is the set of intervals.

Example

$$\langle \{x \mapsto [0, 2], y \mapsto [2, 3]\}, \{x \leq y - 1\} \rangle$$

A solution is $\{x \mapsto 0, y \mapsto 2\}$.

Propagator Function

Interval Propagator

An interval propagator is a function $p_c : \mathbf{D} \to \mathbf{D}$ where $c \in C$ is a constraint and $\mathbf{D}$ the set of all functions $X \to Itv$. Let $d \in \mathbf{D}$, then a propagator is reductive ($p_c(d) \leq d$), monotone ($d \leq d' \Rightarrow p_c(d) \leq p_c(d')$) and sound (does not remove solutions).

Example

The propagator for an equality constraint is:

$$p_{x=y}(d) \triangleq d[x \mapsto d(x) \cap d(y), y \mapsto d(x) \cap d(y)]$$

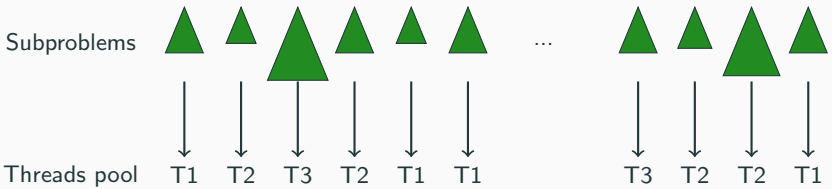
Suppose $d = \{x \mapsto [1,2], y \mapsto [0,5]\}$, then $p_{x=y}(d) = \{x \mapsto [1,2], y \mapsto [1,2]\}$.

Propagate and Search

function $\textsc{solve}(d, \{c_1, \ldots, c_n\})$
  $d \leftarrow \mathbf{gfp}_d\ p_{c_1} \circ \ldots \circ p_{c_n}$
  if $\forall x \in X,\ d(x) = [v, v]$ **then return** $\{d\}$
  else if $\exists x \in X,\ d(x) = \emptyset$ **then return** $\{\}$
  else
    $\langle d_1, \ldots, d_n \rangle \leftarrow \texttt{split}(d)$
    return $\bigcup_{i=0}^{n} \texttt{solve}(d_i, C)$
  end if
end function

Embarrasingly Parallel Search (EPS)[8]

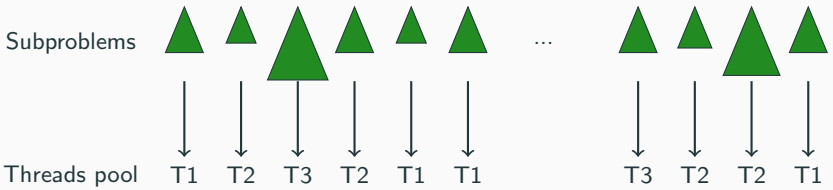
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



[8] A. Malapert et al., 'Embarrassingly Parallel Search in Constraint Programming', JAIR, 2016

Embarrasingly Parallel Search (EPS)

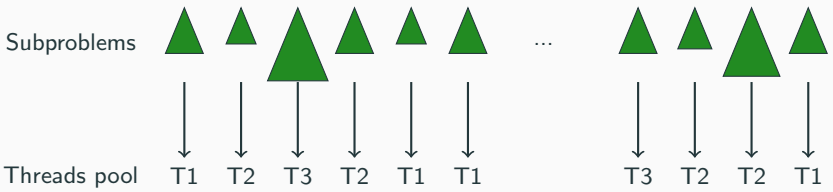
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



⇒ **Other approach:** In modern solvers (e.g., Choco, OR-Tools), they use a portfolio approach (e.g., different *split* strategy on the *same problem*).

Embarrasingly Parallel Search (EPS)

- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



On GPU architectures, 1 subproblem per thread is not efficient (limited cache).
$\Rightarrow$ Need to parallelize propagation: **gfp** $p_{c_1} \circ \ldots \circ p_{c_n}$.

Towards a Correct GPU Constraint Solver

Concurrent Constraint Programming (CCP)

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[8].
Processes interact by *asking* and *telling* constraints into a *shared grow-only constraint store*.

$\langle Program \rangle ::= (t(x_1, \ldots, x_n) \text{ :- } P)+$ (process definition)

$\langle P, Q, \ldots \rangle ::=$ **when** c **then** P *ask operator*

|	**tell** c *tell operator*
|	$P \parallel Q$ *parallel statement*
|	$\exists x,\ P$ *hiding operator*
|	$t(x_1, \ldots, x_n)$ *call*

[8] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Concurrent Constraint Programming (CCP)

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[8].
Processes interact by *asking* and *telling* constraints into a *shared grow-only constraint store*.

$\langle Program \rangle ::= (t(x_1, \ldots, x_n) \text{ :- } P)+$ (process definition)

$\langle P, Q, \ldots \rangle ::=$ **when** c **then** P *ask operator*

| **tell** c *tell operator*

| $P \parallel Q$ *parallel statement*

| $\exists x,\ P$ *hiding operator*

| $t(x_1, \ldots, x_n)$ *call*

Example

$$\exists x, y,\ (\textbf{when } x \neq 0 \textbf{ then tell}(y < x)) \parallel \textbf{tell}(x > 100)$$

A process can work with *partial information*: we do not need to know the exact value of x.

[8] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Concurrent Constraint Programming (CCP)

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[8].
Processes interact by *asking* and *telling* constraints into a *shared grow-only constraint store*.

$\langle Program \rangle ::= (t(x_1, \ldots, x_n) \text{ :- } P)+$ (process definition)

$\langle P, Q, \ldots \rangle ::=$ **when** c **then** P *ask operator*
| **tell** c *tell operator*
| $P \parallel Q$ *parallel statement*
| $\exists x,\ P$ *hiding operator*
| $t(x_1, \ldots, x_n)$ *call*

Clean theoretical framework: every program is a closure operator, unique fixpoint.

Can we rely on CCP to implement **gfp** $p_{c_1} \parallel \ldots \parallel p_{c_n}$?

[8] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Challenges of CCP

- **Concurrent but not parallel**: lack connection with parallel hardware.
- Rely on the (abstract) notion of *constraint system*: good for mathematics, but not for implementation.

  $\Rightarrow$ e.g. **ask** can be an intractable operation depending on the constraint system.
- **Unbounded number of variables**: parallelism + garbage collection is difficult to achieve efficiently.

Parallel Concurrent Constraint Programming (PCCP)

We propose a variant of CCP suited for parallel execution on GPU.[9]

- **Parallel**: a process = a thread.
- **Correct**: equivalence between denotational and (parallel) operational semantics.
- **Lattice type**: each variable has an explicit domain, instead of being a logical entity.
- **No recursion**: finite number of variables (known at compile-time).
- **Lock-free**: processes are executed without locks, even if they share variables.

[9] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Syntax of Parallel Concurrent Constraint Programming (PCCP)

Syntax of PCCP

Let $x, y, y_1, \ldots, y_n \in \mathit{Vars}$ be variables, L a lattice, f a monotone function, and b a Boolean variable of type $\langle \{\mathit{true}, \mathit{false}\}, \Leftarrow \rangle$:

$\langle P, Q \rangle ::= \texttt{if } b \texttt{ then } P$	*ask statement*
$\mid \quad x \leftarrow f(y_1, ..., y_n)$	*tell statement*
$\mid \quad \exists x{:}\mathrm{L},\ P$	*local statement*
$\mid \quad P \parallel Q$	*parallel composition*

From Constraints to PCCP Programs

Let's x, y be variables, $k \in \mathbb{Z}$ a constant and Itv the interval lattice.
We define a function $[\![\varphi]\!]$ compiling a logical formula φ into a PCCP program.

- Constraint $[\![x + k \leq y]\!] \triangleq x \leftarrow f_x(k, x, y) \parallel y \leftarrow f_y(k, x, y)$ with

$$f_x(k, [x_\ell, x_u], [y_\ell, y_u]) \triangleq [-\infty, y_u - k]$$
$$f_y(k, [x_\ell, x_u], [y_\ell, y_u]) \triangleq [x_\ell + k, \infty]$$

From Constraints to PCCP Programs

Let's x, y be variables, $k \in \mathbb{Z}$ a constant and *Itv* the interval lattice.
We define a function $[\![\varphi]\!]$ compiling a logical formula φ into a PCCP program.

- Constraint $[\![x + k \leq y]\!] \triangleq x \leftarrow f_x(k, x, y) \parallel y \leftarrow f_y(k, x, y)$ with

$$f_x(k, [x_\ell, x_u], [y_\ell, y_u]) \triangleq [-\infty, y_u - k]$$
$$f_y(k, [x_\ell, x_u], [y_\ell, y_u]) \triangleq [x_\ell + k, \infty]$$

- Existential $[\![\exists x,\ \varphi]\!] \triangleq \exists x : \mathit{Itv},\ [\![\varphi]\!]$
- Conjunction $[\![c_1 \wedge c_2]\!] \triangleq [\![c_1]\!] \parallel [\![c_2]\!]$
- Equivalence:

$$\begin{aligned}
&[\![\varphi \Leftrightarrow \psi]\!] \triangleq \\
&\quad \texttt{if } \mathit{entailed}(\varphi) \texttt{ then } [\![\psi]\!] \\
\parallel &\quad \texttt{if } \mathit{entailed}(\psi) \texttt{ then } [\![\varphi]\!] \\
\parallel &\quad \texttt{if } \mathit{entailed}(\neg\varphi) \texttt{ then } [\![\neg\psi]\!] \\
\parallel &\quad \texttt{if } \mathit{entailed}(\neg\psi) \texttt{ then } [\![\neg\varphi]\!]
\end{aligned}$$

with $\mathit{entailed}(x + k \leq y) \triangleq \lceil x \rceil + k \leq \lfloor y \rfloor$

Example of Parallel Propagation

Let's consider $[\![\exists x : \mathit{Itv},\ \exists y : \mathit{Itv},\ x + 3 \leq y \wedge x + 6 \leq z]\!]$

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$x \leftarrow [-\infty, y_u - 3]$
$\|\|$ $y \leftarrow [x_\ell + 3, \infty]$
$\|\|$ $x \leftarrow [-\infty, z_u - 6]$
$\|\|$ $z \leftarrow [x_\ell + 6, \infty]$

Example of Parallel Propagation

Let's consider $[\![\exists x : \mathit{Itv},\ \exists y : \mathit{Itv},\ x + 3 \leq y \wedge x + 6 \leq z]\!]$

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$
\begin{array}{ll}
 & x \leftarrow [-\infty, y_u - 3] \\
\| & y \leftarrow [x_\ell + 3, \infty] \\
\| & x \leftarrow [-\infty, z_u - 6] \\
\| & z \leftarrow [x_\ell + 6, \infty]
\end{array}
$$

Issue 1: data race? Parallel update of the same integer x_u.

Example of Parallel Propagation

Let's consider $[\![\exists x : \mathit{Itv},\ \exists y : \mathit{Itv},\ x + 3 \leq y \wedge x + 6 \leq z]\!]$

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$x \leftarrow [-\infty, y_u - 3]$
$\|\quad y \leftarrow [x_\ell + 3, \infty]$
$\|\quad x \leftarrow [-\infty, z_u - 6]$
$\|\quad z \leftarrow [x_\ell + 6, \infty]$

Issue 1: data race? Parallel update of the same integer x_u.
$\Rightarrow$ **Solution**: Use atomic load and store!

Example of Parallel Propagation

Let's consider $[\![\exists x : \mathit{Itv},\ \exists y : \mathit{Itv},\ x + 3 \leq y \wedge x + 6 \leq z]\!]$

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$
\begin{array}{ll}
 & x \leftarrow [-\infty, y_u - 3] \\
\| & y \leftarrow [x_\ell + 3, \infty] \\
\| & x \leftarrow [-\infty, z_u - 6] \\
\| & z \leftarrow [x_\ell + 6, \infty]
\end{array}
$$

Issue 1: data race? Parallel update of the same integer x_u.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Example of Parallel Propagation

Let's consider $[\![\exists x : \mathit{Itv},\ \exists y : \mathit{Itv},\ x + 3 \leq y \wedge x + 6 \leq z]\!]$

Memory:

$x = [1..7]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$
\begin{array}{ll}
 & x \leftarrow [-\infty, y_u - 3] \\
\| & y \leftarrow [x_\ell + 3, \infty] \\
\| & x \leftarrow [-\infty, z_u - 6] \\
\| & z \leftarrow [x_\ell + 6, \infty]
\end{array}
$$

Issue 1: data race? Parallel update of the same integer x_u.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? x_u can be equal to 4 or 7 depending on the order of execution.

Solution: Use a fixpoint loop

While something is changing we reexecute all the propagators!

Memory:

$x = [1..4]$
$y = [1..10]$
$z = [1..10]$

Propagators:

$$
\begin{array}{ll}
\texttt{fp}\ (& \\
 & x \leftarrow [-\infty, y_u - 3] \\
\mid\mid & y \leftarrow [x_\ell + 3, \infty] \\
\mid\mid & x \leftarrow [-\infty, z_u - 6] \\
\mid\mid & z \leftarrow [x_\ell + 6, \infty] \qquad) \\
\end{array}
$$

Solution: Use a fixpoint loop

While something is changing we reexecute all the propagators!

Memory:

$x = [1..4]$
$y = [1..10]$
$z = [1..10]$

Need to add a condition for progress!
We write in the memory only if the value is strictly lower.

$x \leftarrow v$ iff $v < x$

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $\mathit{Store} = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let Proc be the set of all PCCP processes.

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let $Proc$ be the set of all PCCP processes.

Denotational Semantics

We define a function $\mathcal{D} : Proc \to (Store \to Store)$:

$$\mathcal{D}(x \leftarrow f(y_1, .., y_n)) \triangleq \lambda s.s[x \mapsto s(x) \sqcap f(s(y_1), .., s(y_n))]$$
$$\mathcal{D}(\texttt{if } b \texttt{ then } P) \triangleq \lambda s.(\!| \, s(b) \overset{?}{\circ} \mathcal{D}(P)(s) \, \overset{:}{\circ} \, s \, |\!)$$
$$\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$$

Executing the program: **gfp** $\mathcal{D}(P)$.

Sequential Computation = Parallel Computation

We obtain the same result if we execute P in parallel or if we replace all parallel $\|$ by a sequential operator ; (a transformation we write $\mathtt{seq}\ P$) defined as follows:

$$\mathcal{D}(P\ ;\ Q) \triangleq \mathcal{D}(Q) \circ \mathcal{D}(P)$$

Let **fix** f be the set of fixpoints of a function f.

Theorem (Equivalence Between Sequential and Parallel Operators)

fix $\mathcal{D}(\mathtt{seq}\ P) =$ **fix** $\mathcal{D}(P)$

Did We Cheat Using Math?...

OK Ok, we use a "parallel operator", but it is not really executed in parallel on a machine?
Also: $\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$, by unfolding this mathematics definition we have:

$$\begin{aligned} \mathcal{D}(P \parallel Q)(S) &= (\mathcal{D}(P) \sqcap \mathcal{D}(Q))(S) \\ &= \mathcal{D}(P)(S) \sqcap \mathcal{D}(Q)(S) \end{aligned}$$

What's the problem?

Did We Cheat Using Math?...

OK Ok, we use a "parallel operator", but it is not really executed in parallel on a machine?
Also: $\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)$, by unfolding this mathematics definition we have:

$$\begin{aligned} \mathcal{D}(P \parallel Q)(S) &= (\mathcal{D}(P) \sqcap \mathcal{D}(Q))(S) \\ &= \mathcal{D}(P)(S) \sqcap \mathcal{D}(Q)(S) \end{aligned}$$

What's the problem?

We have copied the store!!
Not very "Von Neumann Architecture"-friendly.

Load/Store Operational Semantics

Contribution: An operational semantics based on atomic load and store in memory shown equivalent to the denotational semantics under the following assumptions:

(ATOM) Load and store instructions must be atomic.

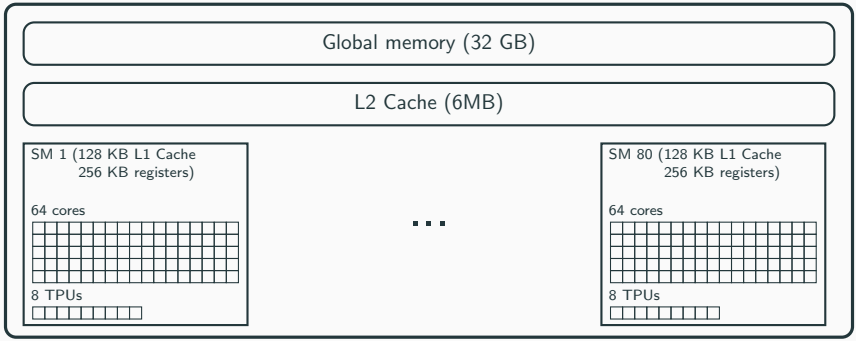
(EC) The caches must eventually become coherent.

(OTA) Values cannot appear *out-of-thin-air*.

Towards an Efficient GPU Constraint Solver

GPU Architecture

(Simplified) Architecture of the GPU Nvidia V100



5120 cores on a single V100 GPU @ 1290MHz

Whitepaper: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

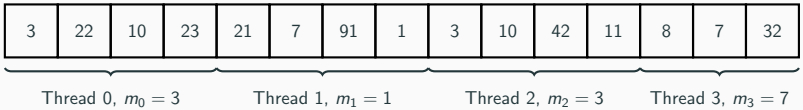
Programming Challenges

- **Memory coalescence**: the way to access the data is important (factor 10).
- **Thread divergence**: each thread within a warp (group of 32 threads) should execute the same instructions.
- **Memory allocation** (dynamic data structures): costly on GPU, everything is generally pre-allocated.
- **Other limitations**: small cache, limited number of lines of code, limited STL...

Example: Find the Minimum in an Array (CPU Style)

Each thread computes its local min (*map*), then we compute the min of all local min (*reduce*).

- Map:

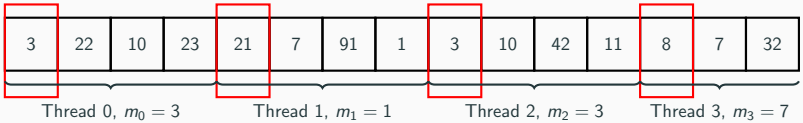


- Reduce: $min([3, 1, 3, 7]) = 1$.

Example: Find the Minimum in an Array (CPU Style)

Iteration 1:

- Map:

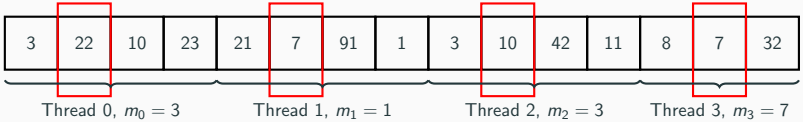


- Reduce: $min([3, 1, 3, 7]) = 1$.

Example: Find the Minimum in an Array (CPU Style)

Iteration 2:

- Map:

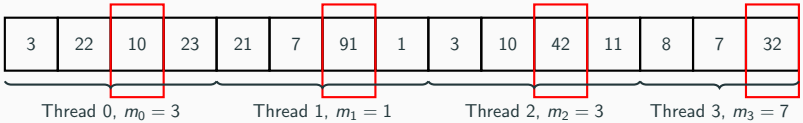


- Reduce: $min([3, 1, 3, 7]) = 1$.

Example: Find the Minimum in an Array (CPU Style)

Iteration 3:

- Map:

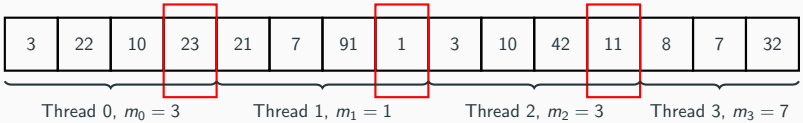


- Reduce: $min([3, 1, 3, 7]) = 1$.

Example: Find the Minimum in an Array (CPU Style)

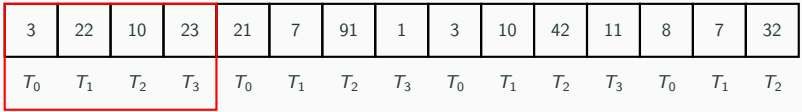
Iteration 4:

- Map:



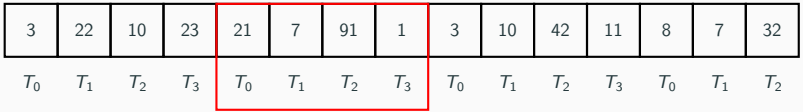
- Reduce: $min([3, 1, 3, 7]) = 1$.

Optimization: Coalesced Memory Accesses



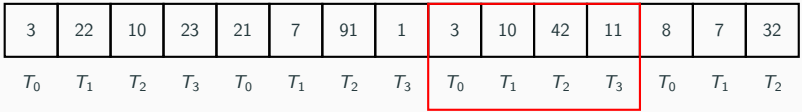
Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses



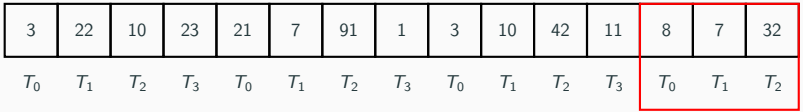
Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses



Very important: up to an order of magnitude faster (10x).

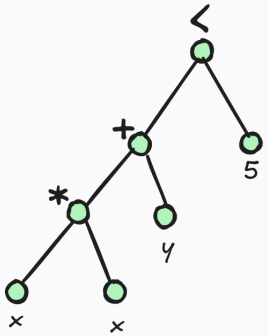
Optimization: Coalesced Memory Accesses



Very important: up to an order of magnitude faster (10x).

Towards an Efficient GPU Constraint Solver

Representation of Propagators



- Represented using `shared_ptr` and `variant` data structures.

  ⇒ Uncoalesced memory accesses.

- Code similar to an interpreter:

```
switch(term.index()) {
  case IVar:
  case INeg:
  case IAdd:
  case IMul:
  // ...
```

  ⇒ Thread divergence.

Ternary Normal Form (TNF)

We simplify the representation of constraints to ternary constraints of the form:

- `x = y <op> z` where `x,y,z` are variables.
- The operators are $\{+, /, *, mod, min, max, \leq, =\}$.

Example

The constraint $x + y \neq 2$ is represented by:

```
t1 = x + y
ZERO = (t1 = TWO)
```

where `ZERO` and `TWO` are two variables with constant values.

Bytecode Representation

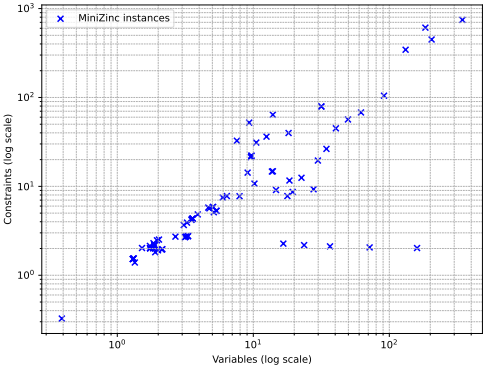
The ternary form of a propagator holds on 16 bytes:

```
struct bytecode_type {
  int op;
  int x;
  int y;
  int z;
};
```

- **Uniform representation** of propagators in memory ⇒ coalesced memory accesses.
- Limited number of operators + sorting ⇒ reduced thread divergence.

Drawback of TNF: increase in number of propagators and variables.

Benchmark on the MiniZinc Challenge 2024 (89 instances)



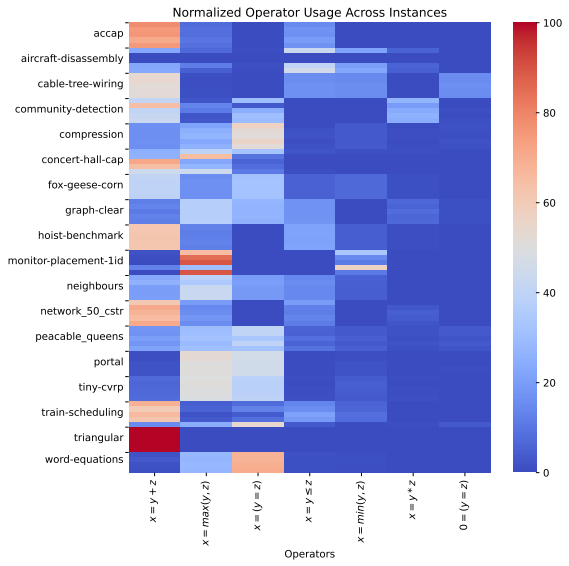
The **median increase** of variables is 4.76x and propagators is 4.33x.

Bytecode-based VS Tree-based

	Bytecode-based + preprocessing	**Bytecode-based**	**Tree-based**
Nodes per second	103,036	79,884	14,623
FP iterations per second	1,429,274	1,379,686	88,944
FP iterations per node	13.9	17.3	6.1
Propagators mem. (MB)	0.69	0.91	10.33
Variables mem. (KB)	286.5	397.3	76.6

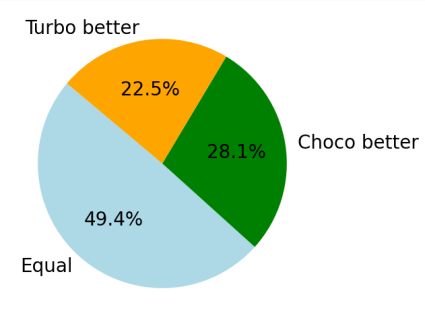
Analysis of the efficiency of the (preprocessed) bytecode representation of TNF and tree-based representation of arbitrary constraints. The mean is used to aggregate the results. FP stands for fixpoint.

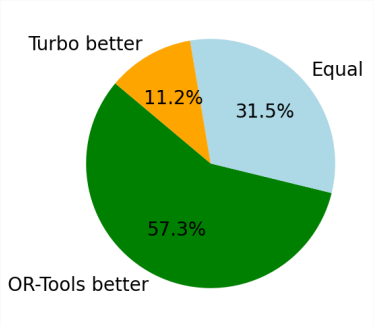
Divergence?


Normalized Operator Usage Across Instances
accap
aircraft-disassembly
cable-tree-wiring
community-detection
compression
concert-hall-cap
fox-geese-corn
graph-clear
hoist-benchmark
monitor-placement-1id
neighbours
network_50_cstr
peacable_queens
portal
tiny-cvrp
train-scheduling
triangular
word-equations
x = y + z
x = max(y, z)
x = (y = z)
x = y ≤ z
x = min(y, z)
x = y * z
0 = (y = z)
Operators
100
80
60
40
20
0

Benchmarks: Turbo vs Choco v4.18

Comparison of the best objective values found (timeout: 20 mins, GPU: H100).





What's Next?

In-Progress: Parallel Consistency Algorithm

Static Scheduling

Find a better ordering of the propagators or fixpoint scheme to reach the greatest fixpoint faster (now, the average is 14 iterations before reaching the fixpoint).

Dynamic Scheduling

- **Goal**: Avoid executing all propagators in each iteration of the fixpoint loop.
- Constraint network as a sparse graph.
- Waking up propagators using GPU-based algorithms (SpMV and scan).

In-Progress: Extension Constraint

- Instead of representing the constraints implicitly, we can list all their solutions in a "table":

$$
\begin{aligned}
&(x \geq 4 \wedge y > 1 \wedge z < 3) \\
\vee &(x = 1 \wedge y = 2 \wedge z = 3) \\
\vee &(x > 1 \wedge y > 1 \wedge z > 3)
\end{aligned}
$$

- This representation is useful but has a large and inefficient TNF decomposition.
- $\Rightarrow$ Directly support extension constraints in Turbo.

Beyond Constraint Reasoning

- Investigate if PCCP (or an extension) can be used for general parallel programming:
  $\Rightarrow$ finding minimum/maximum in an array, union-find, Floyd-Warshall, ...
- As Turbo is based on lattice theory and abstract interpretation, investigate its applicability to abstract interpretation and verification of neural network.

Conclusion

C++ Abstraction: Lattice Land Project

`lattice-land` is a collection of libraries abstracting our parallel model.

It provides various data types and fixpoint engine:

- `ZLB`, `ZUB`: increasing/decreasing integers.
- `B`: Boolean lattices.
- `VStore`: Array (of lattice elements).
- `IPC`: Arithmetic constraints.
- `GaussSeidelIteration`: Sequential CPU fixed point loop.
- `AsynchronousIteration`: GPU-accelerated fixed point loop.
- ...

```
void max(int tid, const int* data, ZLB& m) {
  m.tell(data[tid]);
}
AsynchronousIteration::fixpoint(max);
```

https://github.com/lattice-land

Conclusion: Theoretical Parallel Model of Computation

Data races occur rarely, so we should avoid working so much to avoid them.

Properties of the model

A Variant of Concurrent Constraint Programming on GPU (AAAI 2022)[10].

- Correct: Proofs that $P; Q \equiv P||Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[10] https://ptal.github.io/papers/aaai2022.pdf

Conclusion: Practical Implementation

"General methods that leverage computation are ultimately the most effective, and by a large margin."—Rich Sutton

Turbo

- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).


`https://github.com/ptal/turbo`

Conclusion: Practical Implementation

"General methods that leverage computation are ultimately the most effective, and by a large margin."—Rich Sutton

Turbo

- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).

`https://github.com/ptal/turbo`

But...

- Still lagging behind CP+SAT solvers, and SAT learning is inherently sequential...
- There is hope: ACE (a pure CP solver) show CP-only is still competitive in XCSP3 compet.