Réseau de Contrainte Ternaire pour une Propagation Efficace de Bornes sur GPU

Journées Francophones de Programmation par Contraintes (JFPC 2025)

Pierre Talbot
pierre.talbot@uni.lu
https://ptal.github.io

2nd July 2025

University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

Motivation

- 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?

State of the Art: Combinatorial Optimization on GPU

Very scarce literature, usually:

- **Heuristics**: often population-based algorithms[1].
- **Limited set of problems**[2]
- **Limited GPU parallelization**: offloading to GPU specialized filtering procedures[3,4].
- **cuOpt**: new MILP solver—relaxation on GPU, search on CPU[5].

No general-purpose constraint solver on GPU.

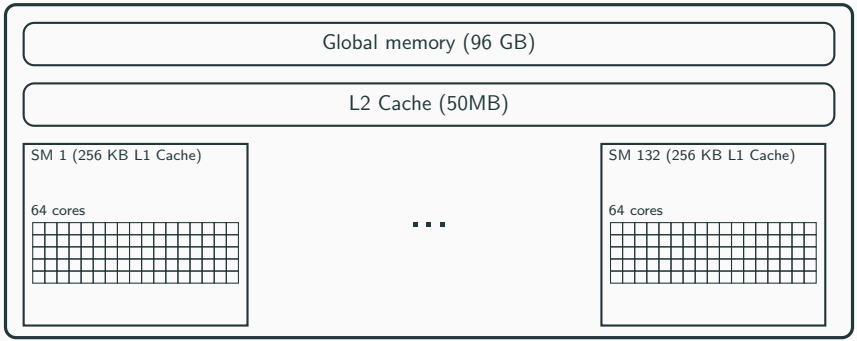
[1] A. Arbelaez and P. Codognet, *A GPU Implementation of Parallel Constraint-Based Local Search*, PDP, 2014.
[2] Jan Gmys. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. INFORMS Journal on Computing, 2022.
[3] F. Campeotto et al., *Exploring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

Our Contributions

- **First general constraint solver fully executing on GPU (propagation + search).**
  ⇒ **General**: Support MiniZinc and XCSP3 constraint models.
  ⇒ **Simple**: interval-based constraint solving + backtracking search (no global constraints, learning, restart, event-based propagation, ...).
  ⇒ **Efficient?**: Almost on-par with Choco (21% better, 30% worst, 49% equal).
  ⇒ **Open-source**: Publicly available on `https://github.com/ptal/turbo`.
- **Ternary constraint network**: representation of constraints suited for GPU architectures.

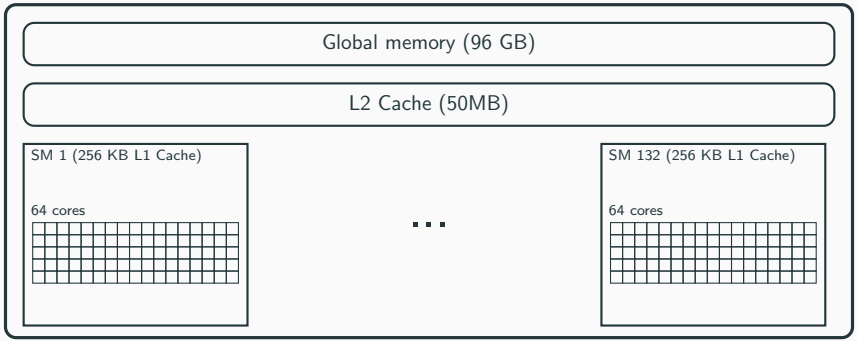
Overview

(Simplified) Architecture of the GPU Nvidia H100



8448 cores grouped in 132 streaming multiprocessors (SM) of 64 cores each.

(Simplified) Architecture of the GPU Nvidia H100



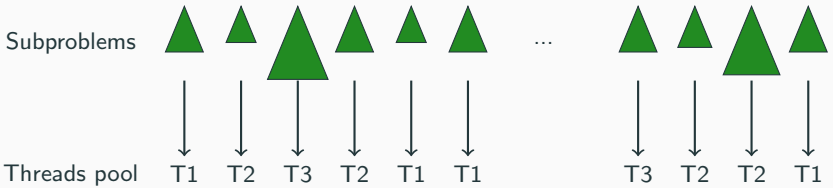
8448 cores grouped in 132 streaming multiprocessors (SM) of 64 cores each.

$\Rightarrow$ **Oversubscribe** (to hide memory latency): 1024 threads per SM

135168 threads running in parallel!

On CPU: Embarrasingly Parallel Search (EPS)[6]

Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).



⇒ **Other approach:** portfolio approach (e.g., different search strategy on the *same problem*) as seen in Choco and OR-Tools.

Each thread works on its own copy of the problem.

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

Programming Challenges on GPU

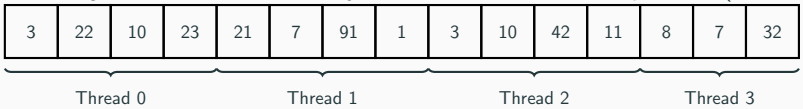
One subproblem per thread is inefficient because:

- **Memory**: suppose each thread needs 1MB (small CSP), then 1GB per SM is constantly moving from global memory to registers.
  $\Rightarrow$ Threads competing for cache, slow access to global memory.
- **Single instruction, multiple threads (SIMT)**: each consecutive 32 threads should execute the same instructions to avoid thread divergence.
- **Memory coalescence**: the way to access the data is important (factor 10).

Programming Challenges on GPU

One subproblem per thread is inefficient because:

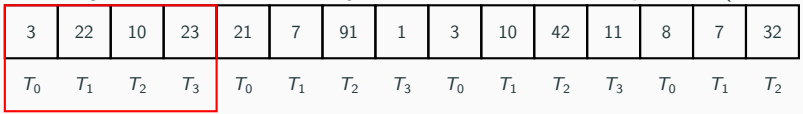
- **Memory**: suppose each thread needs 1MB (small CSP), then 1GB per SM is constantly moving from global memory to registers.
  $\Rightarrow$ Threads competing for cache, slow access to global memory.
- **Single instruction, multiple threads (SIMT)**: each consecutive 32 threads should execute the same instructions to avoid thread divergence.
- **Memory coalescence**: the way to access the data is important (factor 10).



Programming Challenges on GPU

One subproblem per thread is inefficient because:

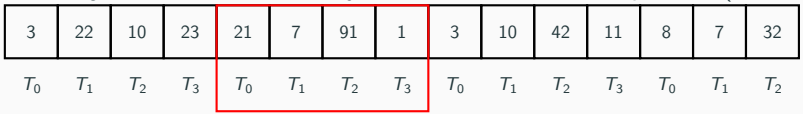
- **Memory**: suppose each thread needs 1MB (small CSP), then 1GB per SM is constantly moving from global memory to registers.
  $\Rightarrow$ Threads competing for cache, slow access to global memory.
- **Single instruction, multiple threads (SIMT)**: each consecutive 32 threads should execute the same instructions to avoid thread divergence.
- **Memory coalescence**: the way to access the data is important (factor 10).



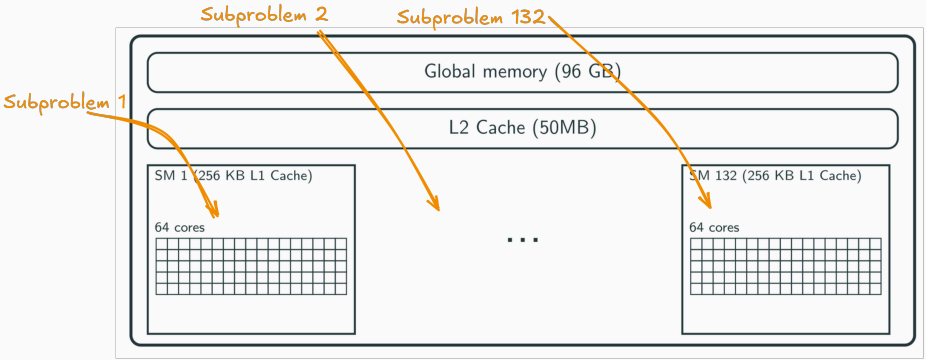
Programming Challenges on GPU

One subproblem per thread is inefficient because:

- **Memory**: suppose each thread needs 1MB (small CSP), then 1GB per SM is constantly moving from global memory to registers.
  $\Rightarrow$ Threads competing for cache, slow access to global memory.
- **Single instruction, multiple threads (SIMT)**: each consecutive 32 threads should execute the same instructions to avoid thread divergence.
- **Memory coalescence**: the way to access the data is important (factor 10).



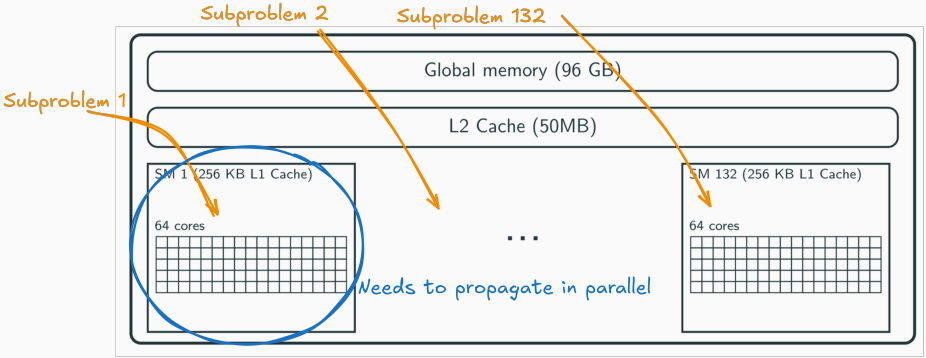
One subproblem per SM[7] with EPS.



Less memory transfer, L1 cache per subproblem.

[7] More precisely, one subproblem per block, a block is running on a single SM. Several blocks can be scheduled on the same SM.

Parallel Propagation

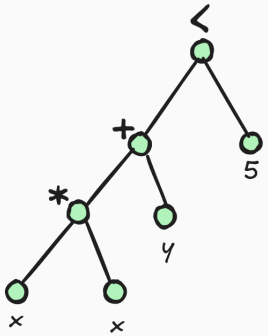


We proposed a correct model of lock-free parallel propagation, but lacked efficiency[8].

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

Ternary Constraint Network

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 Constraint Network

CSP $\langle X, D, \{c_1, \ldots, c_n\}\rangle$ where each c_i is of the form `x = y <op> z` with:

- $x, y, z \in X$ (no constant),
- $op \in \{+, /, *, mod, min, max, \leq, =\}$.

Expressive enough to support all problems of MiniZinc competitions 2022–2024.

Ternary Constraint Network

CSP $\langle X, D, \{c_1, \ldots, c_n\}\rangle$ where each c_i is of the form `x = y <op> z` with:

- $x, y, z \in X$ (no constant),
- $op \in \{+, /, *, mod, min, max, \leq, =\}$.

Expressive enough to support all problems of MiniZinc competitions 2022–2024.

Example

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

```
x = t1 + y                    equivalent to t1 = x − y
ZERO = (t1 = TWO)             equivalent to false ⇔ (t1 = 2)
```

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

Bytecode Representation

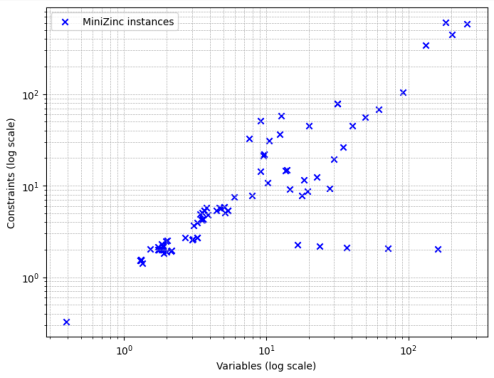
The ternary form of a propagator holds on 16 bytes:

```
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 TCN: increase in number of propagators and variables.

Benchmark on the MiniZinc Challenge 2024 (95 instances)



The **median increase** of variables is 4.46x and propagators is 4.85x.
The **maximum increase** of variables is 258x and propagators is 607x.

Benchmarking

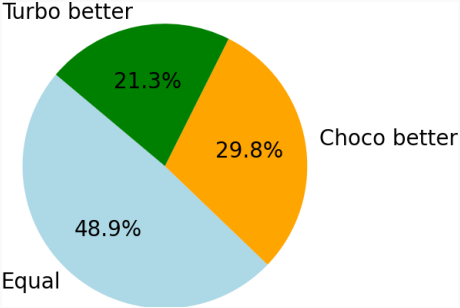
Experimental Evaluation

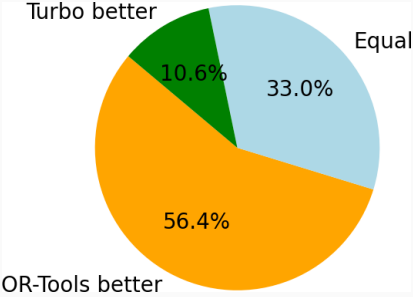
On 95 instances of the MiniZinc 2024 competition.
5 instances discarded (yumi-static) due to out of memory error with TCN.
Timeout 20 minutes, CPU 64 cores, GPU H100.

solver	MiniZinc score	#Optimal
Or-Tools 9.9 (64 threads)	312.9	80
Choco 4.10.18 (64 threads)	233.0	43
Choco 4.10.18 (free search)	159.5	32
Or-Tools 9.9 (fixed search)	130.2	36
Choco 4.10.18 (fixed search)	56.6	25
Turbo 1.2.8 (fixed search)	52.3	20

1-to-1 Comparison

Comparison of the best objective values found.





Conclusion

Conclusion

Turbo: General-purpose GPU constraint solver

- 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).
- Many possible optimizations to improve the efficiency, but need to be redesigned for GPU.

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

Conclusion

Turbo: General-purpose GPU constraint solver

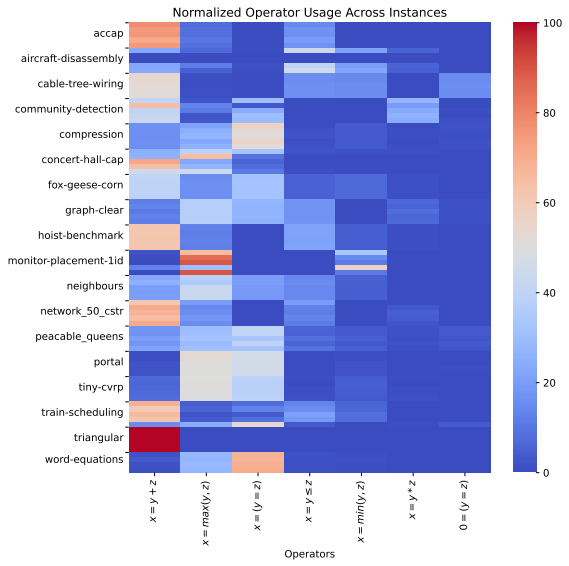
- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).
- Many possible optimizations to improve the efficiency, but need to be redesigned for GPU.

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

But...

- Still lagging behind CP+SAT solvers, and SAT learning is inherently sequential...
- There is hope, apparently, search is not dead!

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

Lock-free Parallel Propagation

Example of Parallel Propagation[9]

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$\boxed{x = [-\infty, \infty]}$

Propagators:

$$\begin{array}{ll} & x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

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

Example of Parallel Propagation[9]

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$x = [-\infty, ?]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$

$x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

Issue: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

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

Example of Parallel Propagation[9]

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$x = [-\infty, 4]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$ $x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

Issue: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

$\Rightarrow$ **Solution**: fixpoint + fair scheduling + strict updates (`if (v < x.ub) { x.ub = v; }`).

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

GPU Fixpoint Algorithm

```
__device__ void fixpoint(Store& d, Props* props, int n) {
  __shared__ bool has_changed = true;
  // Keep going until no variable domain is modified.
  while(has_changed) {
    __syncthreads(); has_changed = false; __syncthreads();
```

GPU Fixpoint Algorithm

```
__device__ void fixpoint(Store& d, Props* props, int n) {
  __shared__ bool has_changed = true;
  // Keep going until no variable domain is modified.
  while(has_changed) {
    __syncthreads(); has_changed = false; __syncthreads();
    // Execute all propagators (similar to AC1)
    for(int i = threadIdx.x; i < n; i += blockDim.x) {
      has_changed |= props[i].propagate(d);
    }
    __syncthreads();
  }
}
```

GPU Challenges

- Coalesced memory accesses of the propagator representation `props[i]`.
- Avoiding divergence in `propagate`.