A GPU-based Constraint Programming Solver

Pierre Talbot
University of Luxembourg
pierre.talbot@uni.lu

Abstract

Machine learning has tremendously benefited from graphics processing units (GPUs) to accelerate training and inference by several orders of magnitude. However, this success has not been replicated in *general and exact combinatorial optimization*. Our key contribution is to propose a general-purpose discrete constraint programming solver fully implemented on GPU. It is based on integer interval bound propagation and backtracking search. The two main ingredients are (1) *ternary constraint network* optimized for GPU architectures, and (2) an *on-demand* subproblems generation strategy. Our constraint solving algorithm is significantly simpler than those found in optimized CPU constraint solvers, yet is competitive with sequential solvers in the MiniZinc 2024 challenge.

Code — https://github.com/ptal/turbo/tree/aaai2026

1 Introduction

Although graphics processing units (GPUs) have accelerated training and inference of machine learning algorithms by several orders of magnitude (Krizhevsky, Sutskever, and Hinton 2012), this success has not been replicated in *general and exact combinatorial optimization*. GPU-accelerated combinatorial optimization has been applied to metaheuristics algorithms (Arbelaez and Codognet 2014; Essaid et al. 2019; Tang, Tian, and Ha 2022; Huang et al. 2024), and to exact optimization limited to specific problems (Gmys et al. 2016; Gmys 2022; Abbas and Swoboda 2022). A recent breakthrough in linear programming (LP) on GPU has been made possible by using a new LP solving algorithm called *primal-dual hybrid gradient method*, which essentially relies on matrix multiplication and addition operations (Chambolle and Pock 2011; Applegate et al. 2021). This method has been used with success to solve mixed integer linear programming in the NVIDIA library cuOpt (NVIDIA Corporation 2025b). We pursue this strand of research by focusing on constraint programming which supports non-linear constraints.

Constraint programming is a general and exact method based on constraint propagation and backtracking search (Lecoutre 2009). There have been a few attempts to accelerate constraint programming on GPU, either by limiting the expressiveness of the constraint language to variables with small domains (Dovier et al. 2021), or by accelerating only the costly constraints while everything else is performed on the CPU (Campeotto et al. 2014; Tardivo et al. 2023, 2024). GPU-based propagation of linear constraints was proposed in (Sofranac, Gleixner, and Pokutta 2022) in the context of LP preprocessing, and of continuous constraints in (Zhang, Feng, and Cagan 2023).

We recently introduced a new lock-free and formally correct model of computation based on concurrent constraint programming (CCP), called *parallel CCP* (PCCP) (Saraswat, Rinard, and Panangaden 1991; Talbot, Pinel, and Bouvry 2022). This work showcases a proof-of-concept constraint solver on GPU, where the propagation algorithm is implemented within PCCP. However, the implementation is not optimized for GPU architectures and only supports a few constraints applied to a scheduling problem. In particular, the representation of constraints leads to uncoalesced memory accesses, load imbalance, thread divergence and unbounded stack; all of which are detrimental to GPU efficiency (Hijma et al. 2023). Those issues are exacerbated when considering a general discrete constraint solver supporting many constraints.

We address those issues by proposing Turbo: a general and exact discrete constraint programming solver optimized for GPU architectures. It is based on integer interval bound propagation and backtracking search—both entirely implemented on GPU. Our approach decomposes a constraint network into a *ternary constraint network* (TCN) with a reduced number of operators, which is propagated in parallel following the PCCP model. This ternary representation allows us to optimize propagators on GPU architectures (Section 3). Over the 100 instances of the MiniZinc 2024 challenge, the median increase is 4.8x in the number of variables and 4.5x in the number of constraints, although 12 instances remain more than 100x larger.

The hardware restrictions of GPU led us to design a solver substantially simpler than an optimized CPU constraint solver. In particular, it does not implement global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency, although most of those optimizations are viewed as essential in modern constraint solvers (Lecoutre 2009; Ohrimenko, Stuckey, and Codish 2009). The key components of Turbo are:

- Parallel bound consistency on TCN using a propagation loop similar to AC1—the first one used in constraint programming (Mackworth 1977).
- Full recomputation to restore the state (Schulte 1999).
- A novel variant of embarrassingly parallel search, based on (Malapert, Régin, and Rezgui 2016), to distribute search nodes across GPU blocks *on-demand* (Section 4).

Despite its unsophisticated algorithm, Turbo is general and competitive with sequential solvers in the MiniZinc 2024 challenge (Stuckey et al. 2014). In comparison to Choco (Prud'homme and Fages 2022), we find the same objective value on 48% of the instances, better ones on 23% and worse ones on 29%. In comparison to the best in class OR-Tools solver (Perron and Didier 2025), we are equal on 31% of the instances, better on 11% and worse on 58%. We give a full experimental evaluation in Section 5.

2 Background

CUDA Programming Model

We overview the components of the NVIDIA Hopper GPU architecture and CUDA programming model that are useful to understand the following sections. To illustrate this section, we take the example of the NVIDIA H100 GPU which is used in the experiments. The H100 has a total of 8448 integer cores which are grouped in 132 streaming multiprocessors (SMs) consisting of 64 cores each. It has a *global memory*—the main memory of the GPU—of 96GB and a L2 cache of 50MB shared among all SMs. Each SM has its own L1 cache of 256KB, from which 227KB can be explicitly used by the programmer—a part called *shared memory*.

The CUDA programming model follows the hierarchical structure of the hardware. A *block* is a group of threads executing on a single SM, and the *grid* is the set of all blocks executing on a single GPU. In the following, we set the size of a block to 256 threads. In code, we have access to three special variables: `threadIdx.x` is the index of a thread relative to a block, `blockIdx.x` is the index of the block relative to the grid and `blockDim.x` is the number of threads per block. A *warp* further subdivides a block into groups of 32 threads; therefore each block has 8 warps. To achieve parallelism, threads within a warp should execute—as much as possible—the same instructions over multiple data (SIMD). Different warps can execute different instructions in parallel. A function executing on GPU is called a *kernel*. For a more exhaustive presentation of CUDA and GPU programming, we refer the reader to e.g. (NVIDIA Corporation 2025a; Wen-mei W. Hwu and David B. Kirk and Izzat El Hajj 2023).

We now overview the two main challenges in CUDA programming that are not encountered in CPU programming.

Memory Access Coalescing A thread reading a value from the global memory triggers a 32, 64 or 128-byte memory transaction. If all threads in a warp read contiguous 4-byte integers, the 32 integers can be retrieved in a single 128-byte memory transaction, a situation called *memory access coalescing*. In the worst case, the 32 integers are spread more than 128 bytes apart from each other, leading to 32 memory transactions. The difference in efficiency between coalesced and uncoalesced accesses is usually up to an order of magnitude, and therefore it is a crucial optimization in CUDA programming.

Thread Divergence The threads of a warp execute one common instruction at a time. Consider the program:

```
if(threadIdx.x % 32 < 16) P else Q
```

Half of the threads in each warp execute `P` and the other half execute `Q`. The two halves are executed sequentially: there are at most 16 threads active at the same time since `P` and `Q` are not on the same execution path and thus have no common instruction.

Constraint Programming

In the following, we consider constraint programming over integer variables only. Let X be a finite set of variables and C be a finite set of constraints. For each constraint $c \in C$, let $scp(c) \subseteq X$ be the set of free variables of c, called its *scope*—for instance, $scp(x < y) = \{x, y\}$. Without loss of generality, we represent the domain of variables using intervals. Let $I = \{[\ell, u] \mid \ell \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, \ell \leq u\} \cup \{\bot\}$ be the set of intervals ordered by inclusion with a special element $\bot$ representing the empty interval. We define $lb([\ell, u]) \triangleq \ell$ and $ub([\ell, u]) \triangleq u$ to extract the lower and upper bounds. A *constraint network* is a pair $P = \langle d, C \rangle$ such that $d \in X \to I$ is the *domain function*.We denote $\mathbf{D}$ the set of all domain functions $X \to I$ ordered pointwise ($d \leq d' \Leftrightarrow \forall x \in X,\ d(x) \subseteq d'(x)$). An *assignment* is a map $asn : X \to \mathbb{Z}$, and we denote the set of all assignments by $\mathbf{Asn}$. The set of solutions of a constraint is given by $rel(c) \subseteq \mathbf{Asn}$. The set of solutions of a constraint network is:

$$\begin{aligned} sol(d, C) \triangleq \{&asn \in \mathbf{Asn} \mid \\ &\forall c \in C,\ asn \in rel(c) \land \forall x \in X,\ asn(x) \in d(x)\} \end{aligned}$$

An interval propagator is a function $p_c : \mathbf{D} \to \mathbf{D}$ where $c \in C$ is a constraint. 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 ($sol(d, \{c\}) \subseteq sol(p_c(d), \{\})$). It is also complete on singleton intervals: whenever $\forall x \in scp(c),\ \exists v \in \mathbb{Z},\ d(x) = [v, v]$, then $sol(d, \{c\}) \supseteq sol(p_c(d), \{\})$.

Constraint propagation consists in finding the greatest fixpoint of a set of propagators $\{p_1, \ldots, p_n\}$ over a domain $d \in X \to I$. As long as the propagators are executed fairly, their order of execution does not matter and the same greatest fixpoint is always eventually reached (Apt 1999). This fact has been used to design various *propagation algorithms* to accelerate the computation of the fixpoint (Schulte and Stuckey 2008; Tack 2009). As constraint propagation is sound but incomplete in general, it must be interleaved with a search procedure. Let $split \in \mathbf{D} \to \mathcal{P}(\mathbf{D})$ be a strictly reductive ($\forall d \in \mathbf{D},\ \forall d' \in split(d),\ d' < d$) and sound and complete ($\forall d \in \mathbf{D}, \bigcup\{sol(d', C) \mid d' \in split(d)\} =$

Algorithm 1: Propagate-and-search

```
function $minimize(d, C, best, z)$
    $d(z) \leftarrow d(z) \cap [-\infty, lb(best(z)) - 1]$
    $propagate(d, C)$
    if $isasn(d)$ then
        $best \leftarrow d$
    else if $\neg isbot(d)$ then
        $\forall d' \in split(d),\ minimize(d', C, best, z)$
    end if
end function
```

$sol(d, C)))$ branching procedure. We introduce next a standard optimization algorithm based on the *propagate-and-search* constraint solving algorithm. Algorithm 1 finds a solution $best \in \mathbf{D}$ which minimizes the value of the variable $z \in X$. It relies on the following functions:

- $propagate(d, \{c_1, \ldots, c_n\})$ computes the greatest fixpoint of $p_{c_1} \circ \ldots \circ p_{c_n}$ below d.
- $isasn(d) \triangleq \forall x \in X,\ \exists v \in \mathbb{Z},\ d(x) = [v, v]$
- $isbot(d) \triangleq \exists x \in X,\ d(x) = \bot$

Here and thereafter, we always pass parameters by reference. The function $isasn$ tests if d maps only to interval singletons (assignment) and $isbot$ tests whether a variable has an empty domain, in which case we must backtrack. It is well-known that the algorithm `minimize` is a sound and complete solving procedure, see e.g. (Lecoutre 2009; Tack 2009). The result holds even in the presence of infinite intervals as long as they become finite after a finite number of propagation steps. We rely on full restoration: the state is restored by reapplying all branching decisions from the root node before propagation (Schulte 1999). In this paper, we do not discuss further state restoration and assume *split* returns modified copies of the domain.

Parallel Constraint Programming There is a long history in parallel constraint solving going back as early as logic programming (Gupta et al. 2001). The two main directions are on parallelizing propagation and search. It has proved difficult to parallelize propagation efficiently due to the sequential interdependencies among propagators (Gent et al. 2018). In fact, most modern constraint programming solvers focus on parallelizing search, while keeping propagation sequential. Perron (1999) and Schulte (2000) parallelize search using a shared queue of nodes among threads; it is the approach of the solver Gecode (Schulte, Tack, and Lagerkvist 2020).

Instead of sharing the work queue, *embarrassingly parallel search* (EPS) (Malapert, Régin, and Rezgui 2016) and *cube-and-conquer* in SAT solvers (Heule, Kullmann, and Marek 2017) decompose the problem *a priori* into enough subproblems such that there is no subproblem taking much longer to be solved than another. It is efficient and easy to implement as threads require very few or no communication. To achieve load balancing, EPS suggests to decompose the problem into $30 \times T$ subproblems where T is the number of threads. They show their approach to scale linearly (tested up to 512 threads).

Parallel portfolio is another approach where the same problem is explored in parallel from different angles, such as different $split$ functions. It is the current parallel approach in OR-Tools (Perron and Didier 2025) and Choco (Prud'homme and Fages 2022).

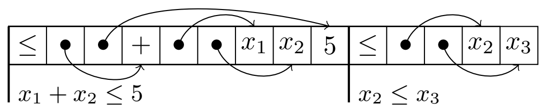
We refer to (Hamadi and Sais 2018) for a more exhaustive presentation across combinatorial communities.

3 Bound Propagation on GPU

We perform bound propagation in parallel, within a block and in-place (without copying d) using the function $propagate^{blk}(d, \{c_1, \ldots, c_n\})$. The superscript blk emphasizes the fact parallel propagation is done within a single block. We proposed a correct and lock-free $propagate^{blk}$ algorithm (Talbot, Pinel, and Bouvry 2022), but it does not optimize the representation of constraints for GPUs. In this section, we provide a better representation of constraints for parallel propagation on GPU.

The Challenge It is possible to construct an infinite number of constraints, take for example the sequence $\langle x_1 = x_2, x_1 = x_2 \odot_1 x_3, x_1 = x_2 \odot_1 x_3 \odot_2 x_4, \ldots\rangle$ for a sequence of arithmetic operators $\langle \odot_1, \odot_2, \ldots\rangle$. Therefore, we cannot implement a different propagator function for each possible constraint. Historically, solvers have limited their constraint languages in order to avoid implementing too many different propagators (Van Hentenryck 1989; Codognet and Diaz 1996; Benhamou and Older 1997; Sam-Haroud and Faltings 1996). More complex constraints must be rewritten into supported constraints automatically or by the user. However, the decomposition of constraints into primitive ones increase the number of auxiliary variables and propagators. It has been shown to be detrimental to the solver performance (Schulte and Tack 2013; Correia and Barahona 2013). A solution to this issue is to implement a propagator as an interpreter over the abstract syntax tree (AST) of the constraint. The propagator recursively traverses the AST to evaluate each node and compute the domain of each subexpression of the constraint. It is called a *view-based propagator* in recent work (Schulte and Tack 2013; Correia and Barahona 2013), but a similar technique was already used in the HC-4 consistency algorithm (Benhamou et al. 1999). Modern constraint solvers such as Choco and OR-Tools implement view-based propagation using inheritance to represent the AST, and subtype polymorphism to evaluate the tree.

On GPU architectures, the view-based representation of propagators leads to uncoalesced memory accesses, load imbalance, thread divergence and unbounded stack; all of which are detrimental to GPU efficiency (Hijma et al. 2023). To understand those issues, we depict the representation in memory of the AST of two constraints (the dots and arrows represent pointers):



To achieve parallelism, we wish to propagate those two constraints in parallel. Both threads start by reading their respective symbol $\leq$, however, the symbols are not contiguous in memory and therefore loads are not coalesced. The problem is exacerbated when considering a warp (32 threads) which generates additional non-coalesced accesses. Furthermore, the different shapes of the constraints lead to load imbalance and thread divergence.

Our Solution We solve those issues by rewriting the constraint network into a *ternary constraint network* (TCN), that is, a constraint network with only constraints of arity 3 such as $x = y + z$ and $b = (x \leq y)$ with $\{x, y, z, b\} \subseteq X$.

Definition 1. A ternary constraint network $\langle d, C \rangle$ is a constraint network such that each $c \in C$ is of the form $x = y \odot z$ where $x, y, z \in X$ are variables and $\odot \in OP = \{+, *, /, mod, min, max, =, \leq\}$[1].

The constraint language considered is sufficient to support all instances of the 2024 MiniZinc challenge. Unary constraints of the form $x = k$, $x \leq k$ and $x \geq k$ where $x \in X$ and $k \in \mathbb{Z}$ are directly represented as domains of the variables. The lack of subtraction is justified by the relational semantics of constraints as we can rewrite $x = y - z$ into $y = x + z$ without loss of precision. We provide a complete transformation of a constraint network into a TCN and its preprocessing in (Talbot 2025), along with the proof that a TCN has exactly the same set of solutions than the initial constraint network; this poses no particular challenge. Note that the global constraints are decomposed in primitive constraints before this transformation and using the MiniZinc compiler in the experiments. We now discuss how TCN addresses the issues mentioned above.

Coalesced Memory Accesses Propagators for ternary constraints are represented by a structure of 16 bytes containing 3 variable's indexes and the operator kind.

```
struct Bytecode {
  int op; // each operator has a code.
  int x;
  int y;
  int z;
};
```

The propagators' data is contained in an array of *bytecodes* `Bytecode* bc`, and the threads load contiguous bytecodes. As a warp contains 32 threads and a memory transaction is at most 128-byte wide, we need only 4 memory transactions ($32 * 16/128$) to load all the bytecodes within a warp. Furthermore, we achieve almost optimal load balancing as no code path takes much longer than any other—all operations consist of fast interval arithmetic.

We could argue that more memory transactions are needed as we need 4 integer loads per bytecode, hence requiring 4 memory transactions to load `op`, then 4 to load `x`, and so on, leading to 16 transactions. We avoid this behavior by performing a vectorized load of 4 integers in a single PTX instruction (CUDA assembly code). To achieve that, it is enough to cast `Bytecode` into the special `int4` CUDA type as follows:

[1]We follow MiniZinc semantics by using truncated integer division and modulus.

```
__device__ Bytecode load(int i) {
  int4 b4 = reinterpret_cast<int4*>(bc)[i];
  return *reinterpret_cast<Bytecode*>(&b4);
}
```

Using the CUDA profiler, we could verify this function is indeed compiled into a single PTX instruction `ld.global.v4.s32`, without additional temporary variable. Note that a larger representation of constraints (e.g. 20 bytes) would prevent us from vectorizing the loads as 16 bytes is the maximum supported size.

Once the bytecode `b` is loaded, we must load the domains of the three variables `b.x`, `b.y` and `b.z` into local variables and propagate according to the operator `b.op`. We note that the loads of the variables' domains might not be coalesced as propagators do not necessarily access variables contiguous in memory.

In the case of small problems, we can overcome the coalescence issue by storing the variables' domains (or the full TCN) in shared memory. It an order of magnitude faster to load the variables' domains from shared memory than from global memory. Note that memory coalescence becomes irrelevant in this case as other memory access rules apply to shared memory, but we do not further optimize those accesses as it is not currently a performance bottleneck.

Minimize Thread Divergence We identify three optimizations to minimize thread divergence. Firstly, we sort lexicographically the bytecodes array on $\langle op, y, x, z \rangle$. It reduces divergence as the propagators with the same operators are executed together. Furthermore, we sort the variables to reduce the number of memory transactions—the y, x, z order led to the best results. A second optimization is to choose a small set of operators OP. It makes the propagation code shorter, therefore triggering more inlining and requiring less instruction loads. Although OP contains 8 distinct operators, the propagator function can diverge in 13 locations due to the operators $\leq$ and $=$ being possibly negated (e.g. $0 = (y = z)$ models $y \neq z$) and reified. For the few problems using division, there might be additional divergence points as propagation depends on the sign of the intervals. Finally, we represent constants as variables as ternary constraints do not have constants. It enables us to represent ternary constraints with only 16 bytes as shown above, but also to avoid diverging in the leaves of the AST where one thread could load a constant and another a variable.

Warp-centric Fixpoint A warp propagates a group of 32 contiguous constraints $\{p_{c_k}, \dots, p_{c_{k+31}}\}$ before moving on to the next group. Since each constraint is propagated only once, there is no guarantee we reach a local fixpoint—an issue that is further exacerbated after sorting the constraints, as contiguous constraints are then more likely to share variables. Because bytecodes and variable domains are already resident in caches or registers, it is advantageous to reuse them by reaching a local fixpoint before proceeding. This approach, known as *warp-centric programming*, improves temporal locality (Hijma et al. 2023).

Motivated by this observation, we propose a new warp-centric fixpoint formulation:

$$\begin{aligned} &\mathit{wpropagate}(d, \{c_1, \ldots, c_n\}) \triangleq \\ &\quad \mathbf{gfp}_d\ (\lambda x.\mathbf{gfp}_x\ p_{c_1} \| \ldots \| p_{c_{32}}) \quad \| \ldots \| \\ &\qquad (\lambda x.\mathbf{gfp}_x\ p_{c_{n-32}} \| \ldots \| p_{c_n}) \end{aligned}$$

Unlike what this mathematical formalization suggests, our implementation shares the variables' domains across warps. In practice, domains are reloaded at each iteration to allow warps to exchange information during their local fixpoint computations. Although this mathematical definition differs from the implementation, it is simpler for presentation purposes and was shown equivalent in (Talbot, Pinel, and Bouvry 2022).

The warp-centric fixpoint loop yields an average speed-up of approximately 10% in the number of nodes explored per second. Correctness is established by the following proposition.

Proposition 1. *The following two greatest fixpoints are equal:*

$$\mathbf{gfp}_d\ p_{c_1} \| \ldots \| p_{c_n} = \mathit{wpropagate}(d, \{c_1, \ldots, c_n\})$$

Proof. The warp-centric fixpoint is merely a scheduling strategy. As shown in (Talbot, Pinel, and Bouvry 2022), the parallel greatest fixpoint is independent of the scheduling strategy, provided that the strategy is fair. The proposed warp-centric scheduling is fair, since every constraint is eventually propagated. Therefore, both fixpoint formulations compute the same greatest fixpoint. □

4 Search on GPU

The Challenge The most successful kind of parallelism used in constraint programming solvers consists in dividing the search among the threads (e.g. assigning one thread per subproblem), or repeating it differently in a portfolio approach (e.g., using different *split* functions) (Gent et al. 2018). In both cases, the threads are working on a local copy of the problem. Due to its loose requirements on communication among threads, EPS is a promising approach to parallelize search on GPUs. However, it must be adapted to reduce its memory footprint.

Indeed, as revealed in the experiments, the decomposition of some instances into TCN can significantly increase the number of variables and constraints, leading to problems where the domain function d can be as large as 56MB[2]—although the average is 2MB and median is 110KB. Therefore, it is not reasonable to solve one subproblem per thread as it takes 68GB (256 threads-per-block $\times$ 132 blocks $\times$ 2MB) on average to store the domains of variables. When performing parallel propagation within a block, we can reduce the memory footprint to 264MB (132 $\times$ 2MB) on average and up to 7.4GB in extreme cases. EPS introduces another challenge by generating *a priori* the subproblems which take 8GB (30 $\times$ 264MB) on average and 222GB (30 $\times$ 7.4GB) for the largest instance. This is without accounting for the necessary solver's internal data structures for, e.g., backtracking. Furthermore, we wish our solver to run on commodity GPUs which have more limited global memory.

[2] On *yumi-dynamic* model with the `p_5_GG_GGG_yumi_grid_setup_5_5_zones.dzn` data file.

Algorithm 2: Propagate-and-search on a single block

```
function $\mathit{minimize}^{blk}(d, C, opt)$
    $d(opt.z) \leftarrow d(opt.z) \cap [-\infty, opt.ub - 1]$
    $\mathit{propagate}^{blk}(d, C)$
    if $\mathit{isasn}^{blk}(d)$ then
        $opt.update(d)$
    else if $\neg \mathit{isbot}^{blk}(d)$ then
        $\forall d' \in \mathit{split}^{blk}(d),\ \mathit{minimize}^{blk}(d', C, opt)$
    end if
end function
```

Our Solution For those reasons, we avoid storing all subproblems *a priori* and propose the *dive and solve* algorithm, a form of lazy EPS, generating subproblems *on-demand*. Dive and solve consists of two phases: a backtrack-free *diving phase*, where a block dives in the search tree until it reaches the next subproblem to be solved, and a *solving phase* where the subproblem is solved by a single block using parallel propagation and backtrack search. The algorithm is entirely ran on the GPU—both diving and solving—without returning to the CPU between phases. For conciseness, we mix code and mathematical notation, for instance, we use $\mathbf{D}$ for the type of the domain function. We describe the solving phase, then the diving phase, and combine them to obtain the *divesolve* algorithm.

Solving phase The solving algorithm is very similar to *minimize* introduced earlier. The main difference is to share the best objective upper bound found so far among the blocks running in parallel. We rely on the following C++ data structure to store the objective variable $z \in X$, the best solution found so far by the block $\mathit{best} \in \mathbf{D}$ and the best upper bound $ub \in \mathbb{Z}$ found so far among all blocks:

```
struct Opt {
  X z;              // objective variable
  atomic<int>& ub; // global best bound
  D best;           // block best solution
  void update(const D& d) {
    best = d;
    ub.fetch_min(lb(best(z)));
  }
};
```

The function $\mathit{minimize}^{blk}(d, C, opt)$ (Algorithm 2) searches for a solution minimizing $opt.z$ and below the current global upper bound $opt.ub$. In the following algorithms, the functions annotated by blk are executed in parallel by a block, while individual statements are executed by one thread (as if guarded with `if(threadIdx.x == 0)`). When a block reaches a solution, it calls $opt.update(d)$ to update its best solution and the global upper bound. In the other direction, each block constrains the objective variable with $opt.ub$ before propagating—instead of using $\mathit{best}(z)$ as in the sequential version. The functions isasn^{blk}, isbot^{blk}

and *split*^{blk} leverage block-parallelism, but due to limited space and because iterating an array in parallel is standard, we do not show them.

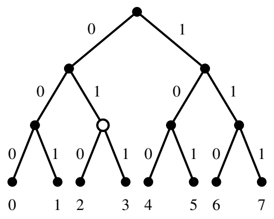
Diving phase Let the root node be at depth 0 of the tree and the subproblems be the nodes at depth $D \in \mathbb{Z}$. We index the subproblems from 0 to $2^D - 1$. The blocks solve the subproblems in order from 0 to $2^D - 1$, hence starting with the first `blockDim.x` subproblems. We introduce three operations to manage the assignment of subproblems to blocks.

The *next operation* atomically increments a shared counter `nextsub` $\in \mathbb{Z}$ and returns its value. When a block has completly explored the subtree of its current subproblem, it requests the next subproblem, called the *target*. Since the next operation is atomic, each block explores different subproblems and all subproblems are explored.

The *dive operation* allows a block to reach its target subproblem from root. After each call to `split`, this operation decides whether to take the left or right branch.

The *skip operation* allows a block to skip some subproblems when it encounters a satisfiable or unsatisfiable node along the path to its target subproblem.

The implementation of those operations essentially relies on bitwise arithmetic and two atomic operations. The main observation is that the binary representation of a subproblem index is also the path in the search tree to reach that subproblem. As an example, we annotate the following tree of depth $D = 3$ with the binary representation of each path.



Suppose we have two blocks running concurrently. They start by solving the first and second subproblems. The first block finishing will start solving the third subproblem. Let `target` $= 2$ be a variable storing the index of the third subproblem. Its binary representation is 0...0010, but only the last 3 bits 010 are of interest. They represent the path from the root to the target:

$$\underset{\uparrow}{0}10 \quad \text{left branch}$$
$$0\underset{\uparrow}{1}0 \quad \text{right branch}$$
$$01\underset{\uparrow}{0} \quad \text{left branch}$$

The `dive` operation consists in moving this pointer from left to right and extracting the corresponding bit.

Now, let us suppose the white node on the path to 010 is unsatisfiable. It means that both 010 and 011 must be skipped since they have a common unsatisfiable ancestor. In general, if we detect (un)satisfiability along a path $b_1 \dots \underset{\uparrow}{b_i} \dots b_D$, we can skip all subproblems prefixed by $b_1 \dots b_i$. To achieve it, it is sufficient to increment the integer represented by $b_1 \dots b_i$, and set all bits b_k with $i < k \leq D$ to 0. On our example, the path to the failed intermediate node is $01 = 1$, thus incrementing it gives $10 = 2$. By completing 10 with one 0, we obtain $100 = 4$ which is the path to the fifth subproblem, effectively skipping the failed subtree.

Algorithm 3: Dive on a single block

```
function $dive^{blk}(d, C, path)$
    while $path.rd > 0$ do
        $propagate^{blk}(d, C)$
        if $isasn^{blk}(d) \lor isbot^{blk}(d)$ then
            $path.skip()$
            return
        else
            $d \leftarrow splitdive^{blk}(d)[path.dive()]$
        end if
    end while
end function
```

The following `Path` structure contains all information to explore all subproblems (in parenthesis the initialization values). All attributes are local to each block but for `nextsub` which is shared among blocks. The depth `D` is initialized to $\lceil \log_2($`blockDim.x` $\times 300)\rceil$. Because we are exploring more nodes due to bound propagation, we use 10 times more subproblems than EPS (which uses $T \times 30$).

```
struct Path {
  int D;
  atomic<uint32_t>& nextsub;       (blockDim.x)
  uint32_t target;                 (blockIdx.x)
  int rd;                          (D)

  void next(){
    target = nextsub++;
    rd = D;
  }
  int dive(){
    --rd;
    return (target & (uint32_t{1}<<rd))>>rd;
  }
  void skip() {
    nextsub.fetch_max(
      ((target >> rd) + uint32_t{1}) << rd);
  }
};
```

The variable `rd` maintains the remaining depth before reaching the target subproblem. In `dive`, the remaining depth is used to efficiently extract the bit indicating whether to turn left or right in the current layer. In `skip`, it is used to obtain the path to the subtree to be skipped (`target` $\gg$ `rd`), increment it and then completing it with zeros using $\ll$ `rd`.

The diving phase is described in Algorithm 3. We skip subproblems whenever we hit a leaf node—either a solution or failed node. We write *splitdive*^{blk} the splitting function used in the diving phase to emphasize the fact it can be different from the one used in *minimize*^{blk}. The function *dive*^{blk} is backtrack-free as only one branch returned by *splitdive*^{blk} is ever considered—note that we use array indexing $splitdive^{blk}(d)[i]$ where $i \in \{0, 1\}$ to retrieve the first or second branch.

Algorithm 4: Dive-and-solve on a single block

```
function $divesolve^{blk}(d, C, path, opt)$
    while $path.target < 2^D$ do
        $d' \leftarrow d$
        $dive^{blk}(d', C, path)$
        if $path.rd = 0 \lor isasn(d')$ then
            $minimize^{blk}(d', C, opt)$
        end if
        $path.next()$
    end while
end function
```

The blocks sharing a common subpath to their targets will independently propagate the same nodes, resulting in duplicated computations. However, as revealed in the experimental evaluation, the time taken to dive is insignificant in comparison to the overall solving time, and it avoids time-consuming and tricky synchronization among blocks.

An important property of the diving phase is to always generate the same tree leaves regardless of what happens in $minimize^{blk}$. For example, modifying the objective variable during diving could modify the set of subproblems, and new subproblems on already visited paths could be generated. Similarly, $splitdive^{blk}$ must be functional and must not contain an internal state, and therefore rules out dynamic search strategies such as *wdeg/dom* (Boussemart et al. 2004) during diving.

Dive and Solve The function $divesolve^{blk}$ (Algorithm 4) is ran by each block until no subproblem is left to be solved ($path.target \geq 2^D$). If $dive^{blk}$ reaches a subproblem, or if it finds a solution along the way, we call $minimize^{blk}$. In the first case, it solves the subproblem using a backtracking algorithm and parallel propagation. In the second case, it updates the best upper bound and best solution found so far, and returns immediately. We reinitialize d' to the root node d each time we solve a new subproblem.

Algorithm 5 shows a CUDA kernel that takes a TCN $\langle d, C\rangle$ shared among all blocks and two arrays $paths$ and $opts$ of size $B \in \mathbb{Z}$ containing the data of each block $i < B$. The number of blocks per SM is chosen using CUDA occupancy calculator, which is usually 4 blocks per SM on our benchmark. We reduce the number of blocks suggested if we exceed the available global memory.

Theorem 2. *The algorithm $divesolve^{grid}$ is sound and complete for any number of blocks and subproblems depth D.*

Proof. We have shown in (Talbot, Pinel, and Bouvry 2022) the propagation fixpoints of $minimize^{blk}$ and $minimize$ on any $\langle d, C\rangle$ are the same. The blocks can only communicate through $opt.ub$, which might prune some solutions of a subproblem. However, since we are seeking for the smallest value of z, it is safe to prune all subtrees where $z \geq opt.ub$: a block has already found a solution with $z = opt.ub$.

Next, we must show that S covers all nodes at depth D of the search tree. Since `path.next()` atomically increment `nextsub`, $divesolve^{blk}$ must enumerate all subproblems from 0 to $2^D - 1$. If $dive^{blk}$ reaches a target subproblem, all solutions are preserved since $propagate^{blk}$ and $splitdive^{blk}$ are both sound and complete. If an intermediate node along a path is a solution or a failure, then all subproblems $s_j, \dots, s_k$ below that node can be safely skipped. The function *skip* returns the next subproblem s_{k+1}. As all subproblems s_i with $i < k$ have been assigned to some blocks, we can update `nextsub` to $k + 1$ using an atomic maximum operation.

Since $divesolve^{blk}$ covers all subproblems, and because $minimize^{blk}$ is sound and complete, there must be one subproblem containing the best solution. Therefore, $divesolve^{grid}$ is sound and complete. □

Algorithm 5: Dive-and-solve across blocks

```
function $divesolve^{grid}(d, C, B, paths, opts)$
 for parallel $blockIdx.x \in \{0, \dots, B-1\}$ do
  $divesolve^{blk}(d, C, paths[blockIdx.x], opts[blockIdx.x])$
 end for
 $syncgrid()$                (wait for all blocks to terminate)
end function
```

5 Experimental Evaluation

We evaluate our approach on the 100 instances of the 2024 MiniZinc Challenge (Stuckey et al. 2014). For all instances, the MiniZinc model is converted into a simpler format called FlatZinc as it is done during the MiniZinc challenge. We remove one instance detected unsatisfiable during the conversion from MiniZinc to FlatZinc, and one instance from the problem *yumi-dynamic* too large to be decomposed into TCN[3]. We run the experiments on the H100 GPU described in Section 2. Unless specified otherwise, we rely on the search strategies specified in the MiniZinc models for both $split^{blk}$ and $splitdive^{blk}$ for all experiments. We set the timeout for each instance to 20 minutes; it includes the FlatZinc and TCN decompositions, preprocessing, diving and solving. All experiments are run on Turbo v1.2.8[4] with all the optimizations described in this paper.

Analysis of Ternary Constraint Networks As an absolute reference, we take the constraint networks generated by the MiniZinc to FlatZinc conversion for Choco, which does not decompose global constraints. We measure the impact of the decomposition into *flat constraint networks* (FCN)—which is FlatZinc without any global constraint including element constraints (e.g. `array_int_element`)—and the subsequent TCN decomposition. The average preprocessing time within Turbo (TCN decomposition and preprocessing) is 11.06s with a standard deviation of 31.53s. The median

[3]`p_9_GSGS_GSGSG_yumi_grid_setup_7_7_-zones.dzn`. Among the 41 solvers' configurations (all categories) participating in the challenge, 15 crashed on this instance, and only 4 could find a solution.

[4]Code is available here: https://github.com/ptal/turbo/tree/aaai2026

Instances	CN	Variables				Constraints			
		average	median	stddev	max	average	median	stddev	max
all (96)	FCN	15.79x	1.76x	40.11x	273.15x	84.82x	3.35x	312.46x	1981.40x
	TCN	64.77x	4.74x	211.97x	1316.34x	152.43x	4.46x	602.61x	3912.41x
w/o yumi-dynamic (92)	FCN	8.78x	1.74x	18.41x	111.62x	24.62x	2.97x	69.88x	525.40x
	TCN	23.42x	4.45x	50.74x	336.28x	37.75x	4.34x	115.71x	731.63x

Table 1: Increase in size of FCNs and TCNs relative to Choco constraint networks on 96 instances.

	TCN	View-based
Nodes per second	103036	14623
FP iterations per node	13.9	6.1
Propagators mem. (MB)	0.69	10.33
Variables mem. (KB)	286.5	76.6

Table 2: TCN vs view-based representation of propagators.

Strategy	#idle	average	median	stddev
user-defined	26	85%	95%	19%
first-fail	21	85%	96%	20%
random	10	83%	91%	27%

Table 3: Impact of split strategy during diving on load balancing.

time is 0.63s and 14/98 instances take more than 10 seconds to be converted and preprocessed (maximum is 179.61s).

On Table 1, we notice the decomposition into FCN is already costly, averaging an increase in variables of 15.79x and constraints of 84.82x. Further decomposing in TCN yields 4 times more variables and 2 times more constraints than FCN on average. Nonetheless, the TCN median increase is about 4.5x in both variables and constraints, which is a small increase considering TCNs only contain ternary constraints. The standard deviation indicates a high variability due to outliers, especially due to the instances *yumi-dynamic* which contain many global constraints. Removing these instances divides by about 3 the average variables increase and by 4 the constraints increase.

Perspective 1 The element constraint is crucial on some instances, in particular those with `table` global constraints. As its decomposition substantially increases the size of the network, it would be interesting to support the element constraint directly, not only to improve propagation, but also for its compact representation. More generally, we could revisit the decomposition of global constraints for TCN.

Fixpoint Computation We claimed in Section 3 that propagators for TCN are more efficient on GPU than view-based propagators. To validate our claim, we evaluate both TCN and view-based propagators on GPU on a restricted set of 20 small to large instances (1 for each problem class). On Table 2, we compare their raw efficiency in terms of nodes per second and fixpoint iterations per node. TCN propagation explores around 7x more nodes per seconds on average than view-based propagation. Interestingly, although the TCN decomposition is larger, the bytecode representation is so compact that it uses significantly less memory than view-based propagators. However, the store of variables is larger.

Perspective 2 In terms of fixpoint iterations, TCN propagation converges almost 2.5x more slowly than view-based propagation. Therefore, sorting more intelligently propagators, taking into account their dependencies (e.g. p_1 triggers p_2 which triggers p_3, etc.), could accelerate the convergence, with an opportunity to double the number of nodes explored.

Time Distribution Most of the time spent on GPU consists in computing $propagate^{blk}$. It averages 93% of the overall GPU time with a standard deviation of 18% and a median of 98%. The rest is spent in the search ($split^{blk}$, $isasn^{blk}$, $isbot^{blk}$). When comparing the diving phase against the solving phase, we find that $divesolve^{blk}$ spends an average of 0.01% diving with a standard deviation of 0.05%. Therefore, repeating propagation among blocks during diving does not incur a significant overhead.

Perspective 3 One of our current focus is to improve the parallel fixpoint algorithm. However, there is a trade-off between the time spent to schedule propagators and the time spent to execute them. So far, our attempts to design parallel dynamic scheduling algorithms increased the time spent overall, although it reduced the number of propagators called.

Diving Split The strategy $splitdive^{blk}$ is important to obtain effective load balancing, i.e. there are as few as possible idle blocks when we reach the timeout. By default, we use the same strategy than $split^{blk}$ which is the one defined in the constraint model by the user. As shown on Table 3, this strategy leads to load imbalance on 26 instances. We test two other strategies: selecting the variable with the smallest domain known as *first-fail* and random selection; the domains are split in the middle. The random selection strategy[5] has better load balancing, with 10 instances terminating with idle blocks. The average proportion of idle blocks at the end of computation is very high at around 85% for all strategies. It highlights that whenever one block becomes idle, many more typically follow.

[5]Random seed is 0, similar results with seeds 2, 42, 10000.

Model	Architecture	#SMs	INT32 cores	RAM	L1 cache	#shmem	avg NPS
Quadro RTX 5000 Max-Q	Turing (7.5)	48	3072	16GB	64KB	19	94k
RTX A5000	Ampere (8.6)	64	4096	24GB	128KB	24	207k
V100	Turing (7.0)	80	5120	16GB	128KB	24	212k
H100	Hopper (9.0)	132	8448	96GB	256KB	46	458k

Table 4: Scalability tests with GPUs of three different generations.

Solver	MZN	XCSP3	Turbo	Other
OR-Tools Par	266.7	97	0%	81.6%
Choco Par	190.8	80	6.1%	72.4%
OR-Tools	119.9	48	11.2%	58.2%
Choco	49.3	37	23.5%	28.6%
Turbo	45.8	35	na	na

Table 5: Overall performance.

Perspective 4 EPS adjusts the cutting depth depending on the unsatisfiable problems encountered. It is possible because subproblems are generated *a priori*. Future research could focus on an efficient algorithm to adjust the depth on-the-fly during diving to improve load balancing. The challenge is to achieve it without a costly synchronization overhead.

Overall Performance Turbo is almost on-par with Choco v4.10.18, but still lags behind OR-Tools v9.9 which is an hybrid solver between CP and SAT. We run both solvers on a processor AMD Epyc ROME 7H12@2.6GHz (64 cores) sequentially and in parallel. The parallel versions of each solver using 64 threads outperform Turbo.

On Table 5, we rank each solver using the MiniZinc scores, taking into account the time to find the objective value, and XCSP3 scores (Audemard et al. 2020) which does not. We also give a 1-to-1 comparison (only objective values are compared): Choco is strictly better than Turbo on 28 instances, and Turbo outperforms Choco on 23 instances. The best-in-class OR-Tools does not completely surpass us as we find better bounds on 11 instances. Turbo usually performs better on problems with few or no global constraints, although there are exceptions (e.g. Turbo is worse on *portal* which has no global constraint and is better *community-detection* which has two kinds of global constraints).

Scalability Testing We evaluate Turbo on 4 NVIDIA GPUs from different generations, all running 64 INT32 cores per SM. Table 4 gives a description of the GPUs used, the number of instances with the domains stored in shared memory (#shmem), and the average number of nodes-per-second (NPS) explored. The results show superlinear scaling in NPS w.r.t. INT32 cores, but for V100 due to an older compute capability (7.0) and more limited RAM. Superlinear scaling is mostly due to the upgraded architecture and the increased number of problems that can be stored in shared memory, effectively reducing global memory accesses.

6 Conclusion

This paper describes Turbo: a discrete constraint solving algorithm fully executing on GPU. We show that an algorithmically simple but massively parallel solver can compete with an algorithmically sophisticated but sequential CPU solver. We discussed several perspectives but believe many more can be explored such as GPU-accelerated continuous constraint solving. Moreover, as our solver fully runs on GPU, machine learning components could be integrated more efficiently as they are also usually accelerated on GPUs. In particular, neural network prediction time was identified as a bottleneck in previous attempts (Cappart et al. 2021).

Acknowledgments

I am grateful to Akif Coerduek and Alice Boucher for their valuable advice on Turbo during the Poland Open Hackathon 2024. I am also grateful to Gwen Maudet and Grégoire Danoy for our productive discussions. This work is supported by the Luxembourg National Research Fund (FNR)—COMOC Project, ref. C21/IS/16101289. We gratefully acknowledge Polish high-performance computing infrastructure PLGrid (HPC Centers: ACK Cyfronet AGH) for providing computer facilities and support within computational grant no. PLG/2025/017986.

References

Abbas, A.; and Swoboda, P. 2022. FastDOG: Fast Discrete Optimization on GPU. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 439–449. New Orleans, LA, USA: IEEE. ISBN 978-1-66546-946-3.

Applegate, D.; Díaz, M.; Hinder, O.; Lu, H.; Lubin, M.; O'Donoghue, B.; and Schudy, W. 2021. Practical large-scale linear programming using primal-dual hybrid gradient. *Advances in Neural Information Processing Systems*, 34: 20243–20257.

Apt, K. R. 1999. The essence of constraint propagation. *Theoretical computer science*, 221(1-2): 179–210.

Arbelaez, A.; and Codognet, P. 2014. A GPU Implementation of Parallel Constraint-Based Local Search. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 648–655. Torino, Italy: IEEE. ISBN 978-1-4799-2729-6.

Audemard, G.; Boussemart, F.; Lecoutre, C.; Piette, C.; and Roussel, O. 2020. XCSP3 and its ecosystem. *Constraints*.

Benhamou, F.; Goualard, F.; Granvilliers, L.; and Puget, J.-F. 1999. Revising hull and box consistency. In *Proceedings of the 1999 International Conference on Logic Programming*, 230–244. USA: Massachusetts Institute of Technology. ISBN 0262541041.

Benhamou, F.; and Older, W. J. 1997. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1): 1 – 24.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, 96–97.

Campeotto, F.; Dal Palu, A.; Dovier, A.; Fioretto, F.; and Pontelli, E. 2014. Exploring the use of GPUs in constraint solving. In *International Symposium on Practical Aspects of Declarative Languages*, 152–167. Springer.

Cappart, Q.; Moisan, T.; Rousseau, L.-M.; Prémont-Schwarz, I.; and Cire, A. A. 2021. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, 3677–3687.

Chambolle, A.; and Pock, T. 2011. A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging. *Journal of Mathematical Imaging and Vision*, 40(1): 120–145.

Codognet, P.; and Diaz, D. 1996. Compiling constraints in clp(FD). *The Journal of Logic Programming*, 27(3): 185 – 226.

Correia, M.; and Barahona, P. 2013. View-based propagation of decomposable constraints. *Constraints*, 18(4): 579–608.

Dovier, A.; Formisano, A.; Pontelli, E.; and Tardivo, F. 2021. {CUDA}: Set Constraints on GPUs. 28.

Essaid, M.; Idoumghar, L.; Lepagnot, J.; and Brévilliers, M. 2019. GPU parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34(5): 497–522.

Gent, I. P.; Miguel, I.; Nightingale, P.; McCreesh, C.; Prosser, P.; Moore, N. C.; and Unsworth, C. 2018. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6): 725–758.

Gmys, J. 2022. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. *INFORMS Journal on Computing*, 34(5): 2502–2522.

Gmys, J.; Mezmaz, M.; Melab, N.; and Tuyttens, D. 2016. A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure. *Parallel Computing*, 59: 119–139.

Gupta, G.; Pontelli, E.; Ali, K. A.; Carlsson, M.; and Hermenegildo, M. V. 2001. Parallel execution of Prolog programs: a survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4): 472–602.

Hamadi, Y.; and Sais, L., eds. 2018. *Handbook of Parallel Constraint Reasoning*. Cham: Springer International Publishing. ISBN 978-3-319-63515-6 978-3-319-63516-3.

Heule, M. J.; Kullmann, O.; and Marek, V. W. 2017. Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4864–4868.

Hijma, P.; Heldens, S.; Sclocco, A.; Van Werkhoven, B.; and Bal, H. E. 2023. Optimization Techniques for GPU Programming. *ACM Computing Surveys*, 55(11): 1–81.

Huang, B.; Cheng, R.; Li, Z.; Jin, Y.; and Tan, K. C. 2024. EvoX: A distributed GPU-accelerated framework for scalable evolutionary computation. *IEEE Transactions on Evolutionary Computation*. Publisher: IEEE.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Lecoutre, C. 2009. *Constraint networks: techniques and algorithms*. Hoboken, NJ: ISTE/John Wiley. ISBN 978-1-84821-106-3.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial intelligence*, 8(1): 99–118.

Malapert, A.; Régin, J.-C.; and Rezgui, M. 2016. Embarrassingly Parallel Search in Constraint Programming. *Journal of Artificial Intelligence Research*, 57: 421–464.

NVIDIA Corporation. 2025a. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/. Accessed: 2025-11-10.

NVIDIA Corporation. 2025b. NVIDIA cuOpt. https://docs.nvidia.com/cuopt/user-guide/latest/. Accessed: 2025-11-10.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via Lazy Clause Generation. *Constraints*, 14(3): 357–391.

Perron, L. 1999. Search procedures and parallelism in constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, 346–360. Springer.

Perron, L.; and Didier, F. 2025. CP-SAT (v9.9). https://developers.google.com/optimization/cp/cp_solver/. Accessed: 2025-12-27.

Prud'homme, C.; and Fages, J.-G. 2022. Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*, 7(78): 4708.

Sam-Haroud, D.; and Faltings, B. 1996. Consistency techniques for continuous constraints. *Constraints*, 1(1): 85–118.

Saraswat, V. A.; Rinard, M.; and Panangaden, P. 1991. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, 333–352. New York, NY, USA: ACM. ISBN 0-89791-419-8.

Schulte, C. 1999. Comparing trailing and copying for constraint programming. In *In Proceedings of the International Conference on Logic Programming*, 275–289. The MIT Press.

Schulte, C. 2000. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, 41–57.

Schulte, C.; and Stuckey, P. J. 2008. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1): 2:1–2:43.

Schulte, C.; and Tack, G. 2013. View-based propagator derivation. *Constraints*, 18(1): 75–107.

Schulte, C.; Tack, G.; and Lagerkvist, M. 2020. *Modeling and Programming with Gecode*.

Sofranac, B.; Gleixner, A.; and Pokutta, S. 2022. Accelerating domain propagation: An efficient GPU-parallel algorithm over sparse matrices. *Parallel Computing*, 109: 102874.

Stuckey, P. J.; Feydy, T.; Schutt, A.; Tack, G.; and Fischer, J. 2014. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2): 55–60.

Tack, G. 2009. *Constraint Propagation – Models, Techniques, Implementation*. Ph.D. thesis, Saarland University.

Talbot, P. 2025. Decomposition and Preprocessing of Ternary Constraint Networks. *arXiv preprint arXiv:2511.11872*.

Talbot, P.; Pinel, F.; and Bouvry, P. 2022. A Variant of Concurrent Constraint Programming on GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 3830–3839.

Tang, Y.; Tian, Y.; and Ha, D. 2022. Evojax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 308–311.

Tardivo, F.; Dovier, A.; Formisano, A.; Michel, L.; and Pontelli, E. 2023. Constraint propagation on GPU: A case study for the AllDifferent constraint. *Journal of Logic and Computation*, 33(8): 1734–1752.

Tardivo, F.; Dovier, A.; Formisano, A.; Michel, L.; and Pontelli, E. 2024. Constraint propagation on GPU: A case study for the cumulative constraint. *Constraints*, 29: 192–214. Publisher: Springer.

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press. ISBN 0-262-08181-4.

Wen-mei W. Hwu and David B. Kirk and Izzat El Hajj. 2023. *Programming Massively Parallel Processors (Fourth Edition)*. Morgan Kaufmann. ISBN 9780323912310.

Zhang, G.; Feng, W.; and Cagan, J. 2023. A GPU-Based Parallel Region Classification Method for Continuous Constraint Satisfaction Problems. *Journal of Mechanical Design*, 146(4): 041705.