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

Pierre Talbot
Université du Luxembourg

pierre.talbot@uni.lu

Résumé

L'apprentissage automatique a grandement bénéficié des processeurs graphiques (GPU) pour accélérer l'entraînement et l'inférence. Cependant, ce succès n'a pas été reproduit pour l'optimisation combinatoire générale et exacte. Dans cet article, nous proposons de paralléliser sur GPU un solveur de programmation par contraintes, qui est une méthode générale et exacte basée sur la propagation de contraintes et le retour sur trace. Nous proposons un algorithme efficace de propagation de bornes d'intervalles entiers sur GPU pour résoudre des problèmes de contraintes discrètes. Notre approche décompose un réseau de contraintes en un réseau de contraintes ternaire optimisé pour les architectures GPU. Notre solveur de contraintes est significativement plus simple qu'un solveur de contraintes CPU optimisé, tout en étant compétitif par rapport aux solveurs séquentiels sur les problèmes du MiniZinc Challenge 2024.

Mots-clés

programmation par contraints, GPU, solveur parallèle, cohérence de bornes

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. We explore GPU-accelerated constraint programming, a general and exact method based on constraint propagation and backtracking search. We propose an efficient integer interval bound propagation algorithm on GPUs for solving discrete constraint problems. Our approach decomposes a constraint network into a ternary constraint network optimized for GPU architectures. Our constraint solver is significantly simpler than an optimized CPU constraint solver, yet is competitive with optimized sequential solvers in the MiniZinc 2024 challenge.

Keywords

constraint programming, GPU, parallel solver, bound consistency

1 Introduction

L'apprentissage automatique a grandement bénéficié des processeurs graphiques (GPU) pour accélérer l'entraînement et l'inférence de plusieurs ordres de grandeur, par exemple dans la classification d'images [23]. Cependant, ce succès n'a pas été reproduit pour l'*optimisation combinatoire générale et exacte*. L'optimisation combinatoire parallélisée sur GPU a été appliquée aux algorithmes de métaheuristiques [4, 16, 42, 22], ainsi qu'à l'optimisation exacte limitée à des problèmes spécifiques [18, 19, 1]. Une percée récente dans la programmation linéaire (PL) sur GPU a été rendue possible en utilisant un algorithme de résolution de PL plus exotique appelé *méthode hybride gradient primal-dual*, qui repose essentiellement sur des opérations de multiplication et d'addition matricielles [12, 2]. Cette méthode a été utilisée avec succès pour résoudre des programmes linéaires mixtes entiers dans la bibliothèque NVIDIA cuOpt.

La programmation par contraintes est une méthode générale et exacte basée sur la propagation de contraintes et le retour sur trace [24]. Il n'existe que peu de tentatives pour paralléliser sur GPU la programmation par contraintes, soit en limitant l'expressivité du langage de contraintes aux variables à petits domaines [15], soit en accélérant uniquement les contraintes coûteuses tandis que tout le reste est exécuté sur le CPU [10, 43, 44]. Récemment, un nouveau modèle parallèle de calcul appliqué à la programmation par contraintes sur GPU a été proposé dans [41]. Ils prouvent formellement que la propagation de contraintes peut être parallélisée en place sans verrous et avec peu d'exigences matérielles, tant que les domaines satisfont une structure de treillis et que les propagateurs sont des fonctions monotones. Bien que la formalisation proposée soit générale et pour l'optimisation exacte, le prototype ne prend en charge que quelques contraintes appliquées à un problème d'ordonnancement. Un travail similaire spécialisé dans la propagation des domaines pour la PL a été proposé dans [38], mais repose sur des opérations atomiques de minimum et maximum.

L'implémentation proposée dans [41] est simple mais non optimisée pour les architectures GPU. Leur représentation des contraintes entraîne des accès mémoire non coalescés, un déséquilibre de charge, une divergence de threads et une pile non bornée ; autant d'éléments nuisibles à l'efficacité sur GPU [21]. Ces problèmes sont exacerbés lorsque l'on

considère un solveur de contraintes discret et général supportant de nombreuses contraintes (Section 3).
Pour relever ces défis, nous proposons un algorithme efficace de propagation de bornes d'intervalles entiers sur GPU. Le solveur proposé est général et peut être utilisé avec les formats de spécification de contraintes standard MiniZinc [27] et XCSP3 [9]. Notre approche décompose un réseau de contraintes en un *réseau de contraintes ternaire* avec un nombre réduit d'opérateurs (Section 4). Cette représentation ternaire nous permet d'optimiser les propagateurs sur GPU (Section 3). Comme la taille de la décomposition est jusqu'à 1000 fois plus grande que le réseau de contraintes initial, nous appliquons plusieurs techniques de prétraitement connues pour réduire sa taille (Section 5). L'augmentation médiane est de 4,8x en nombre de variables et de 4,3x en nombre de contraintes, bien que quatre problèmes (sur 90) soient restés plus de 100x plus grands.
Notre solveur de contraintes est nettement plus simple qu'un solveur de contraintes optimisé sur CPU. En particulier, on n'implémente pas de contraintes globales, d'apprentissage de nogoods, de génération paresseuse de clauses, de stratégies de redémarrage, de propagation événementielle, de restauration d'état basé sur le *trailing* ou recalcul, ni d'algorithme de cohérence de domaine. La plupart de ces optimisations sont considérées comme essentielles dans les solveurs de contraintes modernes [24]. L'architecture GPU nous a conduits à concevoir un solveur nettement plus simple :

- Cohérence de bornes en parallèle en utilisant une boucle de propagation similaire à AC1 — le premier algorithme de consistence utilisé en programmation par contraintes [25].
- Recalcul complet du nœud racine pour restaurer l'état [35].
- Recherche massivement parallèle [26] pour distribuer les nœuds de l'arbre de recherche sur les différents multiprocesseurs de flux du GPU (Section 2).

Malgré son algorithme peu sophistiqué, notre solveur est compétitif par rapport aux solveurs séquentiels sur les problèmes du *MiniZinc Challenge 2024*. En comparaison avec Choco [33], nous trouvons la même valeur objective sur 49% des instances, une meilleure valeur sur 23% et une pire sur 28%. En comparaison avec le meilleur solveur OR-Tools [32], nous sommes égaux dans 32% des cas, meilleurs dans 11% et moins bons dans 57%.
Dans son célèbre billet de blog *The Bitter Lesson*, Rich Sutton affirme que "les méthodes générales exploitant la puissance de calcul sont finalement les plus efficaces, et de loin" [39]. Cet article peut être considéré comme une première étape explorant cette thèse dans le contexte de la programmation par contraintes.

2 Contexte

2.1 Modèle de programmation CUDA

Nous présentons les composants de l'architecture GPU NVIDIA Hopper et le modèle de programmation CUDA afin de comprendre les sections suivantes. Pour illustrer cette section, nous prenons l'exemple du système *NVIDIA GH200 Grace Hopper Superchip* [29] utilisé dans les expériences. Le GH200 combine un processeur Grace Arm avec 72 cœurs *Arm Neoverse V2* et un GPU H100 connecté par une bande passante élevée. Le H100 possède un total de 16896 cœurs regroupés en 132 multiprocesseurs de flux (SM) composés chacun de 128 cœurs. Il dispose d'une *mémoire globale* — la mémoire principale du GPU — de 96GB et d'un cache L2 de 50MB partagé entre tous les SM. Chaque SM possède son propre cache L1 de 256KB.
Le modèle de programmation CUDA suit la structure hiérarchique du matériel. Un *bloc* est un groupe de threads s'exécutant sur un seul SM. Dans la suite, nous fixons la taille d'un bloc à 256 threads, mais elle peut parfois varier en fonction de l'application (jusqu'à 1024 threads). Un *warp* divise un bloc en groupes de 32 threads, chaque bloc contenant donc 8 warps. Pour maximiser le parallélisme d'un warp, les threads doivent exécuter — autant que possible — les mêmes instructions sur des données multiples (SIMD). Les différents warps peuvent exécuter des instructions différentes en parallèle. Le pilote NVIDIA planifie les warps sur les SM en fonction, entre autres, de la disponibilité des données : lorsqu'un warp attend des données depuis la mémoire globale, un autre warp peut être planifié pour masquer la latence des données. Une fonction s'exécutant sur GPU est appelée un *kernel*. Pour une présentation plus exhaustive de CUDA et de la programmation GPU, nous renvoyons le lecteur à [30, 48].
Nous présentons maintenant trois défis de la programmation CUDA qui ne sont pas rencontrés dans la programmation CPU.

Regroupement des accès mémoire. Un thread lisant une valeur depuis la mémoire globale déclenche une transaction mémoire de 32, 64 ou 128 octets. Si tous les threads d'un warp lisent des entiers de 4 octets contigus, les 32 entiers peuvent être récupérés en une seule transaction mémoire de 128 octets, ce qui est appelé *regroupement des accès mémoire* (memory access coalescing). Dans le pire des cas, les 32 entiers sont répartis sur plus de 128 octets, ce qui conduit à 32 transactions mémoire. La différence d'efficacité entre les accès coalescés et non coalescés est généralement d'un ordre de grandeur, ce qui en fait une optimisation cruciale dans la programmation CUDA.

Divergence des threads. Les threads d'un warp exécutent une instruction commune à la fois. Considérons le programme suivant (où `threadIdx.x` est l'indice du thread par rapport à son bloc) :

```
int warpIdx = threadIdx.x % 32;
if(warpIdx < 16) P
else Q
```

La moitié des threads de chaque warp exécute P et l'autre moitié exécute Q. Les deux moitiés sont exécutées séquentiellement : au plus 16 threads sont actifs en même temps, car P et Q ne sont pas sur le même chemin d'exécution et n'ont donc aucune instruction commune. Dans la mesure du possible, ce code doit être réécrit pour éviter l'instruction conditionnelle en fusionnant P et Q.

Manque d'abstraction. En général, un kernel représente un algorithme parallélisé sur GPU, tel que un algorithme de tri ou la multiplication matrice-vecteur creuse (SpMV). Le CPU orchestre le flux principal de l'application en exécutant les kernels et les transferts de mémoire lorsque nécessaire. Le support de bibliothèque pour développer des kernels plus complexes est limité et, en particulier, la majeure partie de la bibliothèque standard C++ est inaccessible au sein d'un kernel. Une autre limitation est l'absence de primitives d'entrée/sortie (seul `printf` est supporté). Ainsi, le code CUDA typique reste relativement bas niveau avec une programmation de style C, notamment avec des pointeurs. Il est donc difficile d'écrire un logiciel de grande envergure tel qu'un solveur de contraintes dans un seul kernel. Néanmoins, la plupart de la syntaxe et de la sémantique de C++20 est supportée et une nouvelle bibliothèque standard compatible CUDA (`libcu++`) est en cours de développement par NVIDIA [31], bien qu'elle manque encore de nombreuses structures de données utiles comme `vector`. En attendant, nous avons développé une bibliothèque de structure de données compatible CUDA, appelé *cuda-battery* et disponible en ligne [1].

2.2 Programmation par Contraintes

Dans la suite, nous considérons la programmation par contraintes sur des variables entières uniquement. Soit X un ensemble fini de variables et C un ensemble fini de contraintes. Pour chaque contrainte $c \in C$, nous notons par $scp(c) \subseteq X$ sa *portée*, c'est-à-dire l'ensemble des variables libres de c. Un *réseau de contraintes* est une paire $P = \langle \rho, C \rangle$ telle que $\rho \in X \to \mathcal{P}(\mathbb{Z})$ est le *domaine* des variables et $\forall c \in C,\ scp(c) \subseteq X$. Une *affectation* est une fonction $asn : X \to \mathbb{Z}$, et nous notons l'ensemble de toutes les affectations par $\mathbf{Asn}$. L'ensemble des solutions d'une contrainte est donné par $rel(c) \subseteq \mathbf{Asn}$. L'ensemble des solutions d'un réseau de contraintes est :

$$sol^\flat(\rho, C) \triangleq \{asn \in \mathbf{Asn} \mid \forall c \in C,\ asn \in rel(c) \land \forall x \in X,\ asn(x) \in \rho(x)\}$$

Sans perte de généralité, nous représentons le domaine des variables à l'aide d'intervalles. Soit $I = \{[\ell, u] \mid \ell \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{\infty\}, \ell \leq u\} \cup \{\bot\}$ l'ensemble des intervalles ordonnés par inclusion avec un élément spécial $\bot$ représentant l'intervalle vide. Dans la suite, nous manipulons exclusivement des intervalles, et donc la fonction de domaine est définie par $d \in X \to I$. Nous notons $\mathbf{D}$ l'ensemble de toutes les fonctions de domaine $X \to I$, ordonné point par point ($d \leq d' \Leftrightarrow \forall x \in X,\ d(x) \subseteq d'(x)$). Soit $[\ell, u]$ un intervalle, l'ensemble des valeurs dans l'intervalle est donné par $\gamma([\ell, u]) \triangleq \{v \in \mathbb{Z} \mid \ell \leq v \leq u\}$.
Un propagateur d'intervalle est une fonction $p_c : \mathbf{D} \to \mathbf{D}$ où $c \in C$ est une contrainte. Pour $d \in \mathbf{D}$, un propagateur est réducteur ($p_c(d) \leq d$), monotone ($d \leq d' \Rightarrow p_c(d) \leq p_c(d')$) et correct ($sol(d, \{c\}) \subseteq sol(p_c(d), \{\})$). Il est également complet pour les intervalles singletons : lorsque $\forall x \in scp(c),\ \exists v \in \mathbb{Z},\ d(x) = [v, v]$, alors $sol(d, \{c\}) \supseteq sol(p_c(d), \{\})$.

1. https://github.com/lattice-land/cuda-battery

La propagation de contraintes consiste à trouver le plus grand point fixe d'un ensemble de propagateurs $\{p_1, \ldots, p_n\}$ sur un domaine $d \in X \to I$. Tant que les propagateurs sont exécutés de manière équitable, leur ordre d'exécution n'a pas d'importance et le même plus grand point fixe est toujours atteint [3]. Ce fait a été utilisé pour concevoir divers *algorithmes de propagation* afin d'accélérer le calcul du point fixe [36, 40].
Comme la propagation de contraintes est correcte mais incomplète en général, elle doit être entrelacée avec une procédure de recherche. Soit $split \in \mathbf{D} \to \mathcal{P}(\mathbf{D})$ une function de branchement strictement réductrice ($\forall d \in \mathbf{D},\ \forall d' \in split(d),\ d' < d$), correcte et complète ($\forall d \in \mathbf{D}, \bigcup\{sol(d', C) \mid d' \in split(d)\} = sol(d, C)$), nous définissons l'algorithme de résolution par *propagation et recherche* comme suit.

function 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 $\exists x \in X,\ d(x) = \bot$ **then return** $\{\}$
  else
    $\langle d_1, \ldots, d_n \rangle \leftarrow$ `split`(d)
    return $\bigcup_{i=0}^{n}$ `solve`(d_i, C)
  end if
end function

Le plus grand point fixe paramétrique $\mathbf{gfp}_x\ f$ d'une fonction f est équivalent à $\mathbf{gfp}\ (f \circ \lambda y.x \cap y)$, c'est-à-dire que nous commençons le calcul du point fixe à partir de l'élément x. Il est bien connu que l'algorithme `solve` est une procédure de résolution correcte et complète, voir par exemple [24, 40]. Le résultat reste valable même en présence d'intervalles infinis tant qu'ils deviennent finis après un nombre fini d'étapes de propagation.

Programmation par contraintes parallèle. Le seul type de parallélisme ayant fait ses preuves dans les solveurs de programmation par contraintes consiste à diviser la recherche entre les threads (par exemple, en assignant un thread par sous-problème $d_1, \ldots, d_n$), ou à la répéter différemment dans une approche portfolio (par exemple, en utilisant différentes fonctions *split*) [17]. Dans les deux cas, les threads travaillent sur une copie locale du problème. Cela est efficace et facile à implémenter car les threads nécessitent peu de synchronisation et de communication, comme illustré par l'algorithme de recherche massivement parallèle (EPS) [26].

Représentation des propagateurs. Il est possible de construire un nombre infini de contraintes, prenons par exemple la séquence $\langle x_1 = x_2, x_1 = x_2 \odot_1 x_3, x_1 = x_2 \odot_1 x_3 \odot_2 x_4, \ldots \rangle$ où chaque $\odot_i$ est un opérateur arithmétique quelconque. Par conséquent, il est impossible de créer statiquement un propagateur par contrainte possible. Historiquement, les solveurs ont limité leurs langages de contraintes afin d'éviter la mise en œuvre d'un trop grand nombre de propagateurs [45, 13, 8]. Les contraintes plus complexes doivent être réécrites en contraintes prises en charge, soit automatiquement, soit par l'utilisateur. L'approche des *indexicaux* est un moyen particulièrement

concis de spécifier des propagateurs à l'aide de quelques constructions primitives [11, 46].
Cependant, la décomposition des contraintes en contraintes primitives augmente le nombre de variables auxiliaires et de propagateurs. Il a été démontré que cela nuit aux performances du solveur [37, 14]. L'idée clé est donc d'implémenter un propagateur en tant qu'interpréteur sur l'arbre syntaxique abstrait (AST) de la contrainte. Le propagateur parcourt récursivement l'AST pour évaluer chaque nœud et calculer le domaine de chaque sous-expression de la contrainte. On parle de *propagateur basé sur les vues* dans les travaux récents [37, 14], mais une technique similaire avait déjà été utilisée dans l'algorithme de cohérence HC-4 [7].
Les solveurs de contraintes modernes tels que Choco et OR-Tools implémentent la propagation basée sur les vues en utilisant l'héritage pour représenter l'AST, et le polymorphisme de sous-type pour évaluer l'arbre. Pour éviter la surcharge causée par le sous-typage dynamique, le polymorphisme paramétrique a également été utilisé [14, 37], mais cela présente l'inconvénient que le solveur doit être recompilé pour chaque nouveau problème de contrainte traité.

3 Propagation Efficace de Bornes sur GPU

La représentation basée sur les vues entraîne des accès mémoire non coalescés, un déséquilibre de charge, une divergence de threads et une pile non bornée ; autant d'éléments nuisibles à l'efficacité sur GPU [21].
Pour comprendre ces problèmes, nous considérons que les propagateurs sont stockés dans un tableau de pointeurs de la forme `Propagator** p` où `Propagator` est une interface. Comme nous considérons des propagateurs peu coûteux en calcul pour des contraintes arithmétiques simples (et non des contraintes globales), une stratégie parallèle consiste à exécuter un propagateur par thread $\mathbf{gfp}_d\ p_{c_1} \| \ldots \| \ p_{c_n}$. Il est garanti que le même plus grand point fixe que celui calculé en Section 2 est finalement atteint [41].
Les lectures mémoires initiales sont coalescées lorsque deux threads accèdent à `p[0]` et `p[1]`. Cependant, comme `*p[0]` et `*p[1]` ne sont pas nécessairement contigus en mémoire, les lectures futures peuvent être non coalescées. Il est possible d'améliorer la situation en aplatissant la représentation des propagateurs dans un tableau de nœuds, de manière similaire à ce qui est utilisé pour représenter les AST dans les compilateurs [47]. Le nœud racine des deux propagateurs sont stockés de manière contigües, puis les deux nœuds de gauche, et ainsi de suite. Lorsque deux propagateurs ont des AST de formes différentes, nous devons stocker la longueur des enfants, ou pour maintenir la coalescence mémoire, utiliser un remplissage similaire au format ELL pour les matrices creuses [6]. Mais cela conduit toujours à un déséquilibre de charge car certains threads peuvent devenir inactifs, et le temps d'exécution d'un groupe de propagateurs parallèles est borné par le plus lent d'entre eux. De plus, si les opérandes des nœuds ne sont pas du même type (par exemple, une addition et une multiplication), les threads divergent et l'évaluation des nœuds devient séquentielle. Enfin, lors de l'évaluation de l'AST, des variables locales sont créées et empilées pour stocker les intervalles des sous-expressions. Les fonctions récursives empêchent le compilateur CUDA de faire l'*inlining* et de calculer automatiquement la taille de la pile.
Notre solution pour résoudre ces problèmes consiste à réécrire le réseau de contraintes en un *réseau de contraintes ternaire*, c'est-à-dire un réseau de contraintes ne contenant que des contraintes d'arité 3 telles que $x = y + z$ et $b = (x \leq y)$. Nous introduisons la *forme normale ternaire* (TNF), qui est une représentation optimisée des contraintes ternaires pour GPU.

Définition 3.1. *Une contrainte est en TNF si elle est de la forme* $x = y \odot z$ *où* $x, y, z \in X$ *sont des variables et* $\odot \in OP = \{+, *, /, mod, min, max, =, \leq\}$.

Les contraintes unaires de la forme $x = k$, $x \leq k$ et $x \geq k$ où $x \in X$ et $k \in \mathbb{Z}$ ne sont pas représentées comme des contraintes mais directement comme des domaines des variables.
La définition de l'ensemble OP pourrait être différente. Par exemple, la contrainte $x = min(y, z)$ peut être réécrite sous la forme $-x = max(-y, -z)$, ce qui, une fois décomposé en contraintes ternaires, donne $\exists x', \exists y', \exists z',\ x' = max(y', z') \wedge x' = zero - x \wedge y' = zero - y \wedge z' = zero - z$, avec $d(zero) = [0, 0]$.
Cela permet de réduire la taille de OP, mais introduit trois nouvelles variables et contraintes. Comme nous le verrons, les opérateurs min et max sont particulièrement importants car ils permettent aussi d'encoder la disjonction et la conjonction, et il est donc préférable de les conserver.
L'absence de soustraction est justifiée par la sémantique relationnelle des contraintes, car on peut réécrire $x = y - z$ sous la forme $y = x + z$ sans perte de précision.
Ce petit ensemble d'opérateurs est suffisamment expressif pour résoudre des instances générales de MiniZinc sur variables entières. Une question ouverte est de savoir s'il existe un meilleur ensemble d'opérateurs OP pour l'exécution sur GPU.
Nous montrons en Section 4 comment réécrire un réseau de contraintes en un réseau de contraintes ternaire utilisant uniquement des contraintes en TNF. Nous discutons maintenant comment la TNF répond aux quatre problèmes mentionnés précédemment.

Coalescence des accès mémoire. Un propagateur implémentant une contrainte en TNF peut être représenté par une structure de 16 octets contenant les indices de trois variables et le type d'opérateur.

```
struct Bytecode {
  OP op; // OP est une énumération entière.
  int x;
  int y;
  int z;
};
```

Les propagateurs sont stockés dans un tableau `Bytecode* bc`, et chaque thread d'un warp charge en mémoire des propagateurs contigus. Un warp contient 32 threads et une transaction mémoire couvre 128 octets. Par conséquent, seules 4 transactions mémoire sont nécessaires ($32 * 16/128$) pour lire tous les propagateurs d'un warp.

De plus, nous obtenons un bon équilibrage de charge, car aucun chemin d'exécution n'est significativement plus long qu'un autre — toutes les opérations sont définies par de l'arithmétique sur intervalles, et donc rapides.

On pourrait argumenter que davantage de transactions mémoire sont nécessaires car il faut lire 4 entiers par propagateur, nécessitant ainsi 4 transactions pour `op`, puis 4 pour `x`, et ainsi de suite, menant à 16 transactions. Nous évitons ce comportement en effectuant un chargement vectorisé de 4 entiers en une seule instruction PTX (le langage assembleur de CUDA). Pour cela, il suffit de convertir `Bytecode` en type spécial CUDA `int4` comme suit :

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

À l'aide du profileur CUDA, nous avons pu vérifier que cette fonction est bien compilée en une seule instruction PTX `ld.global.v4.s32`, sans variable temporaire supplémentaire.

Notons qu'une représentation plus grande des propagateurs (par exemple, 20 octets) nous empêcherait de vectoriser les lectures, car 16 octets est la taille maximale supportée.

Minimisation de la divergence des threads. Une fois le bytecode `b` en mémoire, nous devons lire le domaine des trois variables `b.x`, `b.y` et `b.z` dans des variables locales et propager en fonction de l'opérateur `b.op`. Nous remarquons que les lectures des domaines des variables peuvent être non-coalescés, car les propagateurs n'accèdent pas nécessairement à des variables contigües en mémoire. Nous laissons ouverte la question de savoir comment coalescer les lectures des domaines des variables, et si cela est possible ou non.

Une première optimisation consiste à trier lexicographiquement le tableau de bytecodes selon $\langle op, y, x, z\rangle$. Cela réduit la divergence car les propagateurs ayant les mêmes opérateurs sont exécutés ensemble. De plus, nous trions les variables afin de réduire le nombre de transactions mémoire — l'ordre y, x, z a donné les meilleurs résultats.

Une deuxième optimisation est de choisir un petit ensemble d'opérateurs OP afin de minimiser la divergence des threads, mais aussi de rendre le code plus court, ce qui favorise davantage l'inlining et nécessite moins de chargements d'instructions. Bien que OP contienne 8 opérateurs distincts, la fonction de propagation peut diverger en 13 points en raison des opérateurs $\leq$ et $=$ pouvant être niés (par exemple, $0 = (y = z)$ modélise $y \neq z$) et réifiés. Nous effectuons une analyse des opérateurs utilisés dans différents problèmes de contraintes en Section 6.

Une troisième optimisation consiste à représenter les constantes comme des variables. Cela permet de représenter les contraintes ternaires avec seulement 16 octets comme montré précédemment, mais aussi d'éviter la divergence dans les feuilles de l'AST où un thread pourrait charger une constante et l'autre une variable.

Sans récursion. Étant donné que les contraintes ont une représentation régulière, les propagateurs peuvent être implémentés comme des fonctions sans récursion en utilisant quelques instructions `switch` sur `b.op`. Cela permet au compilateur CUDA de réaliser un inlining agressif, notamment pour les fonctions d'arithmétique sur intervalles.

Point fixe par warp. Les propagateurs exécutés dans un warp traitent souvent des contraintes partageant des variables. Ce regroupement est presque accidentel et dû à la manière dont nous écrivons les modèles de contraintes, mais aussi à la décomposition de chaque contrainte qui est spatialement proche dans le tableau de bytecodes. Avec cette observation en tête, nous proposons une nouvelle formulation de point fixe par warp :

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

Chaque warp atteint un point fixe local avant de passer aux 32 propagateurs suivants. D'un point de vue algorithmique, cela réduit le nombre total d'appels aux propagateurs. D'un point de vue architecture matériel, cela permet de réutiliser le bytecode déjà chargé dans le cache et les registres. Il est important de noter que contrairement à ce que suggère la formalisation mathématique ci-dessus, dans notre implémentation, les domaines des variables sont partagés entre les warps. En effet, nous lisons les domaines à chaque itération pour permettre aux warps d'échanger des informations pendant le calcul de leur point fixe local.

Bien que cette définition mathématique diffère de l'implémentation, elle est plus simple à utiliser dans notre contexte et a été démontrée équivalente [41]. La boucle de point fixe par warp est montrée plus efficace en Section 6. Elle est également correcte, comme démontré par la proposition suivante.

Proposition 3.1. *Les deux plus grands points fixes suivants sont égaux :*

$$\begin{aligned}\mathbf{gfp}_d\ p_{c_1} \| \ldots \| \ p_{c_n} = \quad &\mathbf{gfp}_d\ (\lambda x.\mathbf{gfp}_x\ p_{c_1} \| \ldots \| \ p_{c_{32}})\\ &\| \ldots\\ &\| \ (\lambda x.\mathbf{gfp}_x\ p_{c_{n-32}} \| \ldots \| \ p_{c_n})\end{aligned}$$

Démonstration. L'ordre dans lequel les propagateurs mettent à jour la mémoire dépend de la stratégie de planification des threads sous-jacente. Cependant, il a été démontré que le plus grand point fixe parallèle est toujours le même, indépendamment de la stratégie de planification [41]. Le point fixe par warp impose essentiellement une contrainte sur la stratégie de planification en exécutant des groupes de 32 propagateurs jusqu'à ce qu'un point fixe local soit atteint. Par conséquent, il existe une stratégie de planification des propagateurs pour $\mathbf{gfp}_d\ p_{c_1} \| \ldots \| \ p_{c_n}$

qui correspond à celle utilisée pour le point fixe par warp. Puisque le plus grand point fixe est indépendant de la stratégie de planification, les deux points fixes doivent être égaux. □

4 Réseau de Contrainte Ternaire

Soit $\langle d \in X \to I, C\rangle$ un réseau de contraintes. Nous introduisons la fonction $tcn(d, C) = \langle d', C'\rangle$ qui réécrit un réseau de contraintes en un réseau de contraintes ternaire où chaque contrainte est en TNF. Le langage de contraintes considéré pour C est suffisant pour prendre en charge 95 instances entières du MiniZinc Challenge 2024—les 5 restantes utilisant des variables ensemblistes non supportées pour le moment. La fonction tcn ne fait qu'ajouter des nouvelles variables et par conséquent nous avons $dom(d) \subseteq dom(d')$ où dom est le domaine de la fonction d.

Pour plus de clarté, nous notons $(\!|\, b \,?\, X : Y \,|\!)$ la fonction qui retourne X si l'expression b est vraie et Y sinon. Nous utilisons également l'expression *let* $\mathbf{let}\ x = E\ \mathbf{in}\ E'$ qui associe le résultat de l'évaluation de l'expression E à la variable x, et retourne l'évaluation de l'expression E' où E' peut utiliser x. Cela s'étend naturellement au cas où E retourne un tuple de valeurs, par exemple $\mathbf{let}\ x, y = E\ \mathbf{in}\ E'$.

Nous définissons d'abord plusieurs fonctions pour étendre un réseau de contraintes avec une nouvelle variable $x \notin dom(d)$, et pour mettre à jour une variable déjà présente dans d :

$$
\begin{aligned}
&extend(d, x, [\ell, u]) = (d \cup \{x \mapsto [\ell, u]\}, x)\\
&extend(d, x) = extend(d, x, [-\infty, \infty])\\
&extend(d) = extend(d, fresh(d))\\
&extend_{\mathbb{B}}(d) = extend(d, fresh(d), [0, 1])\\
&extend_{co}(d, k) =\\
&\quad \mathbf{let}\ c = \texttt{\_\_CONSTANT\_}(\!|\, k < 0 \,?\, \texttt{m}k : k \,|\!)\ \mathbf{in}\\
&\quad (\!|\, c \in dom(d) \,?\, (d, c) : extend(d, c, [k, k]) \,|\!)\\
&update(d, x, [\ell, u]) =\\
&\quad \{y \mapsto (\!|\, x = y \,?\, [\ell, u] \cap d(y) : d(y) \,|\!) \mid y \in dom(d)\}
\end{aligned}
$$

La fonction $fresh(d)$ retourne un nom de variable x tel que $x \notin dom(d)$. Nous utilisons la fonction $extend_{\mathbb{B}}$ pour créer une nouvelle variable booléenne. Afin d'éviter de créer plusieurs variables représentant une même constante, nous générons un nom unique pour chaque constante. Par exemple, `__CONSTANT_5` est le nom de la variable correspondant à la constante 5, et `__CONSTANT_m1` pour la constante -1. Nous supposons qu'aucun nom de variable dans le réseau de contraintes initial n'est de cette forme. La fonction $update(d, x, [\ell, u])$ intersecte le domaine $[\ell_x, u_x]$ d'une variable x avec un intervalle $[\ell, u]$, où l'intersection d'intervalles est définie par $[\ell_x, u_x] \cap [\ell, u] \triangleq [\max\{\ell_x, \ell\}, \min\{u_x, u\}]$.

La fonction $tcn(d, C) = \langle d', C'\rangle$ réécrit chaque contrainte en TNF :

$$
\begin{aligned}
&tcn(d, \{\}) = \langle d, \{\}\rangle\\
&tcn(d, \{c_1, c_2, \ldots, c_n\}) =\\
&\quad \mathbf{let}\ d, T = tcn(d, \{c_2, \ldots, c_n\})\ \mathbf{in}\\
&\quad \mathbf{let}\ d, U, x = tnf(d, c_1)\ \mathbf{in}\\
&\quad \langle update(d, x, [1, 1]), T \cup U\rangle
\end{aligned}
$$

Chaque contrainte $c \in C$ est réécrite en un ensemble T de contraintes en TNF, éventuellement avec de nouvelles variables dans d, à l'aide de la fonction tnf. La variable x est une variable booléenne qui réifie la contrainte c. Comme les contraintes doivent toutes être satisfaites, nous devons fixer cette variable x à 1 pour activer la contrainte, ce qui est fait en mettant à jour son domaine.

Nous introduisons la fonction récursive $tnf(d, t) = (d', T, x)$ qui réécrit un terme ou une contrainte t en un ensemble T de contraintes en TNF. Le résultat de l'expression t (ou son statut de réification s'il s'agit d'une contrainte) est stocké dans la variable x. Nous considérons d'abord les cas de base (variables et constantes) et les contraintes unaires (négation arithmétique $-$, appartenance à un ensemble $\in$ et valeur absolue abs).

Les variables sont simplement retournées, car toute variable apparaissant dans une contrainte doit déjà être présente dans d. Une nouvelle variable est créée pour chaque constante distincte k en utilisant $extend_{co}$ défini précédemment. Les contraintes unaires sont réécrites en utilisant des contraintes ternaires équivalentes.

Pour l'appartenance à un ensemble, nous avons besoin de la fonction $itvs(S)$ qui transforme un ensemble S en un ensemble d'intervalles. Par exemple, $itvs(\{1, 2, 3, 5\}) = \{[1, 3], [5, 5]\}$. La stratégie de réécriture est toujours la même : nous réécrivons d'abord les paramètres de la fonction ou du prédicat, puis nous assemblons les résultats pour réécrire l'expression courante.

$$
\begin{aligned}
&tnf(d, x) = (d, \{\}, x)\\
&tnf(d, -t) = tnf(d, 0 - t)\\
&tnf(d, t \in S) =\\
&\quad \mathbf{let}\ d, T, x = tnf(d, t)\ \mathbf{in}\\
&\quad \mathbf{let}\ d = update(d, x, [\min S, \max S])\ \mathbf{in}\\
&\quad \mathbf{let}\ d, U, y =\\
&\qquad tnf(d, \textstyle\bigvee_{[\ell,u]\in itvs(S)}(x \geq \ell \wedge x \leq u))\ \mathbf{in}\\
&\quad (d, T \cup U, y)\\
&tnf(d, k) =\\
&\quad \mathbf{let}\ d, x = extend_{co}(d, k)\ \mathbf{in}\\
&\quad (d, \{\}, x)\\
&tnf(d, abs(t)) =\\
&\quad \mathbf{let}\ d, T, x = tnf(d, t)\ \mathbf{in}\\
&\quad \mathbf{let}\ d, U, y = tnf(d, 0 - x)\ \mathbf{in}\\
&\quad \mathbf{let}\ d, V, z = tnf(d, max(x, y))\ \mathbf{in}\\
&\quad \mathbf{let}\ d = update(d, z, [0, \infty])\\
&\quad tnf(d, T \cup U \cup V, z)
\end{aligned}
$$

La réécriture des opérateurs binaires $\odot \in OP$ est factorisée dans une fonction unique. Lorsque l'opérateur est booléen

$(\leq, =)$, le résultat est stocké dans une variable booléenne, sinon dans une variable entière.

Les autres opérateurs peuvent être réécrits en expressions utilisant les opérateurs OP. En particulier, comme la soustraction est l'inverse de l'addition, nous avons $x = y - z \Leftrightarrow y = x + z$. Ce n'est pas le cas pour la multiplication et la division sur les entiers, c'est pourquoi nous conservons les deux opérateurs. Nous réécrivons l'opérateur $\neq$ en utilisant la négation de l'égalité. Comme nous pouvons être dans un contexte réifié, nous ne pouvons pas simplement ajouter la contrainte TNF $zero = (x = y)$, c'est pourquoi nous déléguons la réécriture à l'équivalence logique.

$$
\begin{aligned}
&tnf(d, t_1 \odot t_2) = \\
&\quad \textbf{let } d, x = \\
&\qquad (\!| \odot \in \{\leq, =\} \; ?\; extend_{\mathbb{B}}(d) : extend(d) |\!) \textbf{ in} \\
&\quad \textbf{let } d, T, y = tnf(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, z = tnf(d, t_2) \textbf{ in} \\
&\quad (d, T \cup U \cup \{x = y \odot z\}, x)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 \neq t_2) = \\
&\quad \textbf{let } d, zero = extend_{co}(d, 0) \textbf{ in} \\
&\quad tnf(d, zero \Leftrightarrow t_1 = t_2)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 - t_2) = \\
&\quad \textbf{let } d, x = extend(d) \textbf{ in} \\
&\quad \textbf{let } d, T, y = tnf(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, z = tnf(d, t_2) \textbf{ in} \\
&\quad (d, T \cup U \cup \{y = x + z\}, x)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 \geq t_2) = tnf(d, t_2 \leq t_1) \\
&tnf(d, t_1 > t_2) = tnf(d, t_2 \leq t_1 - 1) \\
&tnf(d, t_1 < t_2) = tnf(d, t_1 \leq t_2 - 1)
\end{aligned}
$$

La dernière étape consiste à compiler les connecteurs logiques en contraintes TNF. Les formules logiques peuvent apparaître dans des expressions arithmétiques, par exemple $(x \neq y \wedge x \neq z) + (y \neq z) \geq 1$, et les variables entières dans les formules logiques, par exemple $((x + w) \vee y) = z$.

Dans un contexte booléen, nous interprétons la valeur d'un intervalle $[\ell, u]$ comme étant $true$ si et seulement si $0 \notin [\ell, u]$, et $false$ si $\ell = u = 0$. Cependant, si la contrainte est réifiée, nous devons associer $true$ à 1 et non à n'importe quelle valeur. Comme nous compilons $\vee$ en min et $\wedge$ en max, nous devons nous assurer que le résultat des connecteurs logiques se situe dans l'intervalle $[0, 1]$.

La fonction $booleanize$ mappe le domaine d'une expression apparaissant dans un contexte booléen à une variable booléenne. Comme optimisation, nous ne connectons x à une variable booléenne que si ce n'est pas déjà une variable booléenne, c'est-à-dire si son domaine ne se situe pas dans l'intervalle $[0, 1]$. Cette fonction est ensuite utilisée pour réécrire tous les connecteurs logiques.

Il est à noter que nous ne pouvons pas directement réécrire t_1 xor t_2 en $t_1 \neq t_2$, sinon des expressions telles que $[1, 1]$ xor $[2, 2]$ seraient évaluées à $true$, ce qui, selon notre sémantique, n'est pas correct car à la fois $[1, 1]$ et $[2, 2]$ doivent être associés à $true$.

$$
\begin{aligned}
&booleanize(d, t) = \\
&\quad \textbf{let } d, T, x = tnf(d, t) \textbf{ in} \\
&\quad \textbf{if } \neg(d(x) \leq [0, 1]) \textbf{ then} \\
&\qquad \textbf{let } d, U, b = tnf(d, x \neq 0) \textbf{ in} \\
&\qquad (d, T \cup U, b) \\
&\quad \textbf{else } (d, T, x)
\end{aligned}
$$

$$tnf(d, \neg t) = tnf(d, t = 0)$$

$$tnf(d, t_1 \Rightarrow t_2) = tnf(d, \neg t_1 \vee t_2)$$

$$
\begin{aligned}
&tnf(d, t_1 \wedge t_2) = \\
&\quad \textbf{let } d, b = extend_{\mathbb{B}}(d) \textbf{ in} \\
&\quad \textbf{let } d, T, b_1 = booleanize(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, b_2 = booleanize(d, t_2) \textbf{ in} \\
&\quad (d, T \cup U \cup \{b = min(b_1, b_2)\}, b)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 \vee t_2) = \\
&\quad \textbf{let } d, b = extend_{\mathbb{B}}(d) \textbf{ in} \\
&\quad \textbf{let } d, T, b_1 = booleanize(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, b_2 = booleanize(d, t_2) \textbf{ in} \\
&\quad (d, T \cup U \cup \{b = max(b_1, b_2)\}, b)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 \Leftrightarrow t_2) = \\
&\quad \textbf{let } d, b = extend_{\mathbb{B}}(d) \textbf{ in} \\
&\quad \textbf{let } d, T, b_1 = booleanize(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, b_2 = booleanize(d, t_2) \textbf{ in} \\
&\quad (d, T \cup U \cup \{b = (b_1 = b_2)\}, b)
\end{aligned}
$$

$$
\begin{aligned}
&tnf(d, t_1 \; xor \; t_2) = \\
&\quad \textbf{let } d, T, b_1 = booleanize(d, t_1) \textbf{ in} \\
&\quad \textbf{let } d, U, b_2 = booleanize(d, t_2) \textbf{ in} \\
&\quad \textbf{let } d, V, b = tnf(d, b_1 \neq b_2) \textbf{ in} \\
&\quad (d, T \cup U \cup V, b)
\end{aligned}
$$

Nous prouvons maintenant que chaque réseau de contraintes $\langle d, C \rangle$ est équivalent à sa réécriture $tcn(d, C)$, c'est-à-dire qu'il possède exactement le même ensemble de solutions. Comme nous introduisons de nouvelles variables, nous devons restreindre les solutions à celles définies sur les variables initiales. Soit $asn : X \to \mathbb{Z}$ une affectation et Y un sous-ensemble de X, sa restriction $asn|_Y$ est définie par $\{y \mapsto asn(y) \mid y \in Y\}$. Nous étendons la restriction à l'ensemble des affectations A comme suit : $A|_Y = \{asn|_Y \mid asn \in A\}$.

Proposition 4.1 (Correction et complétude)**.** *Soit $\langle d, C \rangle$ un réseau de contraintes défini sur l'ensemble de variables X, alors $sol(d, C) = sol(tcn(d, C))|_X$.*

Proposition 4.2 (Unicité)**.** *De plus, nous avons une bijection entre les deux ensembles de solutions : $|sol(d, C)| = |sol(tcn(d, C))|$.*

Ces affirmations peuvent être prouvées par induction structurelle sur la formule, montrant qu'à chaque étape l'ensemble des solutions est préservé.

5 Prétraitement du Réseau de Contrainte Ternaire

Nous suivons des techniques de prétraitement standards que nous spécialisons pour les réseaux de contraintes ter-

naires [34, 28]. Le prétraitement a un impact substantiel sur les performances comme le montrent les expériences. L'objectif du prétraitement est essentiellement de supprimer des variables et des contraintes. Avant de présenter notre algorithme de prétraitement, nous définissons une structure pour détecter les variables équivalentes.

Nous cherchons à trouver une partition E de X telle que chaque composante Y de E représente un ensemble de variables équivalentes. Plus précisément, toutes les paires de variables $x, y \in Y$ sont connectées par une contrainte d'égalité $x = y$. Nous écrivons $[x]_E \in E$ pour désigner la classe d'équivalence de x dans E. Nous supposons que les variables sont totalement ordonnées (par exemple, par un indexage) et nous choisissons $\min Y$ comme variable représentative de la classe d'équivalence Y. Les classes d'équivalence sont découvertes par diverses techniques de prétraitement. Initialement, nous supposons que les variables sont toutes distinctes, ce qui est donné par la partition $init(X) \triangleq \{\{x\} \mid x \in X\}$. Nous ajoutons une égalité de variables $x = y$ en supprimant les deux classes d'équivalence $[x]_E$ et $[y]_E$ de E, puis en ajoutant leur union dans la partition. Le domaine d'intervalle d'une variable $x \in X$ dans une classe d'équivalence Y est l'intersection des domaines de toutes les variables de Y, défini comme $d_E(x) \triangleq \bigcap_{y \in [x]_E} d(y)$.

Nous appliquons sept fonctions de prétraitement :

- *Propagation* pour réduire le domaine des variables.
- *Simplification algébrique* pour éliminer les contraintes dont les solutions peuvent être directement exprimées dans le domaine des variables. Elle détecte également les équivalences entre les variables et affine la partition E.
- *Élimination des sous-expressions communes* pour détecter les contraintes ternaires ayant la même expression à droite de l'égalité. Par exemple, considérons $a = y + z$ et $b = y + z$, la procédure élimine l'une des deux contraintes et fusionne les classes d'équivalence de a et b.
- *Fusion des domaines* des variables dans les classes d'équivalence. Cela est particulièrement utile pour la propagation qui ne connaît pas les classes d'équivalence.
- *Élimination des contraintes redondantes* pour supprimer les contraintes qui sont déjà impliquées par d. Par exemple, si $d(x) = [1, 2]$ et $d(y) = [2, 3]$, alors la contrainte ternaire $1 = (x \leq y)$ est toujours vraie, indépendamment de l'évolution de d.
- *Renommage des variables* pour utiliser une variable unique par classe d'équivalence.
- *Élimination des variables inutiles* pour supprimer les variables qui ne sont dans le domaine d'aucune contrainte. Nous conservons les variables ayant un domaine vide afin de pouvoir détecter l'insatisfiabilité.

Nous donnons un exemple de l'algorithme de prétraitement sur le réseau de contraintes $\langle d, \{x = y + z, w = y + z, x = (y = z)\}\rangle$ avec $d(x) = [0, 1]$, $d(w) = [1, 2]$ et toutes les autres variables ayant pour domaine $[-\infty, \infty]$.

La propagation est incapable de supprimer des valeurs des domaines et la simplification algébrique ne détecte aucune contrainte d'égalité puisque x pourrait être égal à 0. Ensuite, l'élimination de sous-expressions communes détecte l'égalité $x = w$, modifie la partition en $\{\{x, w\}, \{y\}, \{z\}\}$ et supprime la contrainte $x = (y = z)$.

La propagation reste inefficace, mais cette fois la simplification algébrique est capable de détecter l'égalité $y = z$ car $d_E(x) = d(x) \cap d(w) = [1, 1]$, et la partition est mise à jour en $\{\{x, w\}, \{y, z\}\}$. Ensuite, la simplification algébrique réécrit la contrainte $w = y + z$ en $w = y * 2$. Cela permet à l'étape de propagation de réduire le domaine de y à $\bot$, détectant ainsi effectivement que le problème est insatisfiable.

À ce stade, le réseau de contraintes est $\langle d, \{w = y * 2\}\rangle$ et le prétraitement termine. Dans cette situation, le problème a été complètement résolu, et détecté comme insatisfiable, par le prétraitement.

6 Évaluation Expérimentale

Nous évaluons notre approche sur 90/100 instances du MiniZinc Challenge 2024. Nous supprimons 5 instances du problème *harmony* contenant des variables d'ensemble (actuellement non prises en charge) et 5 instances supplémentaires du problème *yumi-dynamic* pour lesquelles notre décomposition est trop volumineuse pour être résolue en parallèle sur GPU (nécessite plus de 96GB). Les 90 instances restantes peuvent être résolues avec notre approche. Pour toutes les instances, le modèle MiniZinc est converti en un format plus simple appelé FlatZinc, comme cela est fait pendant le MiniZinc Challenge. Il y a une instance qui est détectée comme insatisfiable lors de la conversion en FlatZinc et que nous supprimons des expériences lorsqu'elle n'est pas pertinente.

Notre solveur GPU, Turbo v1.2.8 [2], est exécuté sur le superprocesseur NVIDIA GH200 décrit dans la Section 2. Choco v4.10.18 et OR-Tools v9.9 sont exécutés sur un processeur AMD Epyc ROME 7H12 avec 64 cœurs cadencés à 2,6GHz. Nous fixons la limite de temps pour chaque instance à 20 minutes.

Pour assurer l'équité, nous créons les sous-problèmes à résoudre en parallèle en suivant la même stratégie de recherche que celle utilisée pour la résolution. Cela permet à notre algorithme parallèle d'explorer un arbre de recherche similaire aux solveurs séquentiels, ce qui nous désavantage car cela conduit à des blocs inactifs dans 20% des instances en raison d'un déséquilibre de charge. La propagation et la recherche sont entièrement exécutées sur le GPU — le CPU est limité à l'analyse, au prétraitement et à l'affichage du résultat final. La création des sous-problèmes est incluse dans la limite de temps, ainsi que les décompositions FlatZinc et TCN.

Analyse des réseaux de contrainte ternaire. Comme nous ne prenons en charge aucune contrainte globale, la conversion de MiniZinc en FlatZinc est déjà assez coûteuse. La Table 1 montre l'augmentation moyenne, médiane, l'écart-type et l'augmentation maximale du nombre de variables et

2. Disponible sur Github : https://github.com/ptal/turbo/tree/v1.2.8

	Variables				Contraintes			
	moyenne	**médiane**	**écart-type**	**max**	**moyenne**	**médiane**	**écart-type**	**max**
FlatZinc	9.42x	1.86x	18.63x	111.62x	24.94x	2.95x	67.91x	486.87x
TCN	53.61x	7.97x	151.61x	1133.22x	76.68x	6.21x	265.88x	1837.19x
Prétraité	22.06x	4.76x	50.48x	344.62x	36.39x	4.33x	115.18x	746.09x

TABLE 1 – Augmentation de la taille des réseaux de contraintes par rapport aux réseaux de contraintes Choco sur 89 instances.

Solveur	**TCN**	**Autre**	**Égal**
Choco	22.5%	28.1%	49.4%
OR-Tools	11.2%	57.3%	31.5%
Choco-64	6.7%	71.9%	21.3%
OR-Tools-64	0%	80.9%	19.1%

FIGURE 1 – Comparaison entre deux solveurs de l'état de l'art et le réseau de contraintes ternaire prétraité avec point fixe par war psur 89 instances.

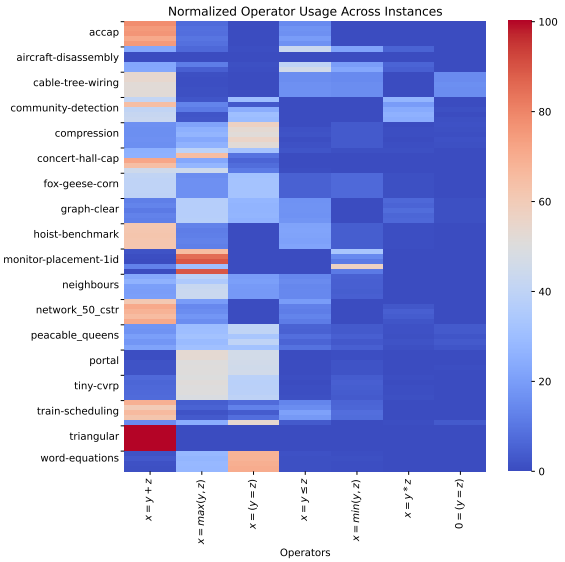

FIGURE 2 – Carte thermique des opérateurs utilisés dans les réseaux de contraintes ternaires prétraités sur 89 instances, regroupés par classes de problèmes.

de contraintes pour un modèle MiniZinc complètement décomposé (sans contraintes globales), après décomposition en TCN (Section 4) et après prétraitement (Section 5).

Sur 63 des 89 instances, la décomposition FlatZinc est inférieure à un ordre de grandeur par rapport au réseau de contraintes Choco, tant en nombre de variables qu'en nombre de contraintes. Après l'application de la décomposition TCN, nous augmentons encore de 5 fois le nombre de variables et de 3 fois le nombre de contraintes en moyenne. Heureusement, l'étape de prétraitement réduit cette augmentation à 2,5 fois pour les variables et 1,5 fois pour les contraintes en moyenne, bien qu'elle ne contienne que des contraintes ternaires. Le temps moyen de prétraitement est de 24,22s avec un écart-type de 96,91s. Le temps médian est de 0,91s et seulement 11 instances prennent plus de 10 secondes pour être prétraitées.

Sur la Figure 2, nous analysons l'utilisation des opérateurs à travers les instances. De toute évidence, la divergence de threads n'est plus la principale préoccupation puisque les instances n'utilisent pas une grande variété d'opérateurs. En particulier, il n'y a aucune instance avec des opérations de modulo et de division (ou elles sont simplifiées après le prétraitement). En raison du prétraitement, toutes les contraintes d'égalité non réifiées $1 = (x = y)$ sont supprimées. La contrainte $x = y \leq z$ n'existe que dans un contexte réifié et $0 = y \leq z$ (modélisant $y > z$) et $1 = y \leq z$ ont complètement disparu.

Cela est dû à la décomposition FlatZinc et TCN, où $y \leq z$ est réécrit en $y - z \leq 0$ par la décomposition FlatZinc, puis réécrit en $y = x + z \wedge x \leq 0$, ce qui permet de gérer directement $\leq$ et $>$ dans le domaine des variables.

Les contraintes en TNF les plus utilisées sont l'addition, le maximum (qui est utilisé pour encoder la disjonction) et l'égalité réifiée. De manière intéressante, bien que présente dans 5 classes de problèmes, la décomposition de la contrainte globale *all-different* en $\mathcal{O}(n^2)$ contraintes de la forme $0 = (x = y)$ ne semble pas constituer un goulet d'étranglement.

Analyse de TCN pour la propagation sur GPU. Afin de valider notre décomposition TCN sur GPU, nous la comparons à une représentation traditionnelle des propagateurs basée sur les vues [37, 14], également exécutée sur GPU. Nous effectuons nos benchmarks sur un ensemble restreint de 20 instances, allant de petites à grandes tailles (au moins une pour chaque classe de problème), afin d'économiser des ressources.

Dans la Table 2, nous comparons l'efficacité brute en termes de nœuds par seconde et d'itérations de point fixe par seconde. Nous désactivons le prétraitement lors de la comparaison entre la propagation basée sur bytecode et celle basée sur les vues car nous ne disposons pas du même

	Basé sur Bytecode TCN Prétraité	Basé sur Bytecode Propagation	Basé sur Vues TCN Prétraité	Basé sur Vues Propagation
Nœuds par seconde	103036	79884	24276	14623
Itérations de point fixe par seconde	1429274	1379686	291438	88944
Itérations de point fixe par nœud	13.9	17.3	12	6.1
Mémoire des propagateurs (MB)	0.69	0.91	15.82	10.33
Mémoire des variables (KB)	286.5	397.3	286.5	76.6

TABLE 2 – Analyse de l'efficacité de la représentation en bytecode (prétraitée) de TCN, de la représentation basée sur les vues de TCN et de la représentation basée sur les vues de contraintes arbitraires. La moyenne est utilisée pour agréger les résultats. FP signifie point fixe.

algorithme de prétraitement pour les deux. La propagation basée sur bytecode explore environ 5 fois plus de nœuds par seconde en moyenne que la propagation basée sur les vues. Comme elle converge également près de 3 fois plus lentement, cela signifie que les propagateurs TCN pourraient bénéficier d'un tri plus intelligent pour accélérer la convergence, offrant ainsi la possibilité de tripler le nombre de nœuds explorés par seconde. La propagation basée sur les vues sur GPU bénéficie également de la représentation TCN, bien qu'elle soit clairement surpassée par la propagation basée sur bytecode.

De manière intéressante, bien que la décomposition TCN soit plus volumineuse, la représentation en bytecode est si compacte qu'elle utilise beaucoup moins de mémoire que les propagateurs basés sur les vues. Cependant, le stockage des variables est plus volumineux.

Analyse des algorithmes de point fixe. Nous comparons le point fixe par warp (WAC1) introduit dans la Section 3 avec la boucle de point fixe AC1. Globalement, WAC1 explore en moyenne 10% de nœuds en plus par rapport à AC1 et trouve une meilleure borne sur 12% des problèmes tout en étant égal sur les autres. De plus, en termes de vitesse de convergence—mesurée par le nombre de propagateurs appelés pour atteindre le point fixe—WAC1 est 30% plus rapide en moyenne et converge plus rapidement sur 83 des 89 instances. Par conséquent, il est raisonnable de conclure que WAC1 est une optimisation utile.

Plus important encore, cela montre qu'il existe des opportunités pour améliorer la planification des propagateurs afin d'accélérer davantage la convergence du plus grand point fixe. À ce stade, il n'est pas clair si une stratégie de planification dynamique serait efficace sur GPU en raison du surcoût lié à la planification des propagateurs en parallèle. Nous laissons cette question ouverte pour les recherches futures.

Comparaison avec Choco et OR-Tools. Dans cette dernière série d'expériences, nous comparons notre approche avec deux solveurs de l'état de l'art. Les résultats globaux sont illustrés dans la Figure 1. Malgré la simplicité de notre solveur, il n'est pas loin de Choco en mode séquentiel. Cela démontre que l'optimisation matérielle peut surpasser l'optimisation algorithmique telle que les contraintes globales et le redémarrage sur une certaine classe de problèmes. Le meilleur solveur OR-Tools ne nous dépasse pas complètement, car nous trouvons de meilleures bornes sur 10 instances.

Pour être complets, nous présentons également la version parallèle de chaque solveur en utilisant 64 threads, bien que la comparaison soit difficile en raison des stratégies de parallélisation différentes. Choco et OR-Tools utilisent tous deux une parallélisation portfolio impliquant différentes stratégies de recherche, y compris des stratégies aléatoires. Nous sommes surpassés par ces deux solveurs. Pour une meilleure évaluation, nous devrions implémenter une recherche parallèle portfolio au lieu de EPS dans notre solveur GPU, en attribuant une stratégie de recherche par bloc. Nous laissons cela pour des travaux futurs.

7 Conclusion

Selon la thèse de Rich Sutton, tout ce dont nous avons besoin, c'est de plus de puissance de calcul et d'algorithmes évolutifs basés uniquement sur la recherche et l'apprentissage [39]. Dans cet article, nous proposons un algorithme de résolution évolutif et simple, accéléré sur GPU, basé sur la consistance des bornes et la recherche. Lorsqu'il est exécuté en parallèle sur un seul GPU, notre algorithme est compétitif avec le solveur de contraintes séquentiel Choco. Par conséquent, cette expérience soutient la thèse de Rich Sutton.

Cependant, nous devons réévaluer nos résultats en tenant compte d'un solveur hybride SAT-CP tel que OR-Tools. Dans ce cas, davantage de calculs ne suffisent pas à dépasser l'apprentissage de conflits. Le problème immédiat est que l'apprentissage de conflits est intrinsèquement un processus séquentiel, et difficilement parallélisable, bien qu'il existe quelques tentatives infructueuses [20]. La question est alors : peut-on se passer de l'apprentissage de conflits ?

Lors de la compétition XCSP3 2024, en comparant les solveurs basés sur SAT (par exemple, OR-Tools et Picat) avec ACE, un solveur purement basé sur la recherche, il a été montré qu'ACE est meilleur pour trouver de bonnes bornes d'objectif, tandis que les solveurs basés sur les conflits sont meilleurs pour prouver l'optimalité [5]. Par conséquent, il est possible que même sans apprentissage de conflits, un solveur de contraintes basé sur GPU puisse devenir compétitif avec les solveurs hybridés avec SAT.

Remerciements

Ce travail est supporté par le fond national de la recherche Luxembourgeois (FNR)—project COMOC, ref. C21/IS/16101289. Nous remercions également l'infrastructure Polonaise de calcul informatique haute performance PLGrid (HPC Center : ACK Cyfronet AGH) pour nous avoir fourni l'accès aux GPU H100 (computational grant no. PLG/2025/017986).

Références

[1] Ahmed Abbas and Paul Swoboda. FastDOG : Fast Discrete Optimization on GPU. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 439–449, New Orleans, LA, USA, June 2022. IEEE.

[2] David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. Practical large-scale linear programming using primal-dual hybrid gradient. *Advances in Neural Information Processing Systems*, 34 :20243–20257, 2021.

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

[4] Alejandro Arbelaez and Philippe Codognet. A GPU Implementation of Parallel Constraint-Based Local Search. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 648–655, Torino, Italy, February 2014. IEEE.

[5] Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2024 xcsp3 competition, 2024.

[6] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, Portland Oregon, November 2009. ACM.

[7] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *Proceedings of the 1999 International Conference on Logic Programming*, page 230–244, USA, 1999. Massachusetts Institute of Technology.

[8] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1) :1 – 24, 1997.

[9] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3 : An Integrated Format for Benchmarking Combinatorial Constrained Problems. *arXiv e-prints*, November 2016.

[10] Federico Campeotto, Alessandro Dal Palu, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of GPUs in constraint solving. In *International Symposium on Practical Aspects of Declarative Languages*, pages 152–167. Springer, 2014.

[11] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. *Programming Languages : Implementations, Logics, and Programs*, 1997.

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

[13] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3) :185 – 226, 1996.

[14] Marco Correia and Pedro Barahona. View-based propagation of decomposable constraints. *Constraints*, 18(4) :579–608, October 2013.

[15] Agostino Dovier, Andrea Formisano, Enrico Pontelli, and Fabio Tardivo. {CUDA} : Set Constraints on GPUs. page 28, 2021.

[16] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. GPU parallelization strategies for metaheuristics : a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34(5) :497–522, September 2019.

[17] Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil CA Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6) :725–758, 2018.

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

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

[20] Youssef Hamadi and Lakhdar Sais, editors. *Handbook of Parallel Constraint Reasoning*. Springer International Publishing, Cham, 2018.

[21] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E. Bal. Optimization Techniques for GPU Programming. *ACM Computing Surveys*, 55(11) :1–81, November 2023.

[22] Beichen Huang, Ran Cheng, Zhuozhao Li, Yaochu Jin, and Kay Chen Tan. EvoX : A distributed GPU-accelerated framework for scalable evolutionary computation. *IEEE Transactions on Evolutionary Computation*, 2024. Publisher : IEEE.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional

neural networks. *Advances in neural information processing systems*, 25, 2012.

[24] Christophe Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/John Wiley, Hoboken, NJ, 2009.

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

[26] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly Parallel Search in Constraint Programming. *Journal of Artificial Intelligence Research*, 57 :421–464, November 2016.

[27] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc : Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming—CP 2007*, pages 529–543. Springer, 2007.

[28] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251 :35–61, October 2017.

[29] NVIDIA Corporation. NVIDIA Grace Hopper Superchip Architecture. 2024.

[30] NVIDIA Corporation. *CUDA Toolkit Documentation*, 2025. Accessed : 2025-03-04.

[31] NVIDIA Corporation. *libcu++*, 2025. Accessed : 2025-03-04.

[32] Laurent Perron and Frédéric Didier. Cp-sat, 2024.

[33] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver : A java library for constraint programming. *Journal of Open Source Software*, 7(78) :4708, 2022.

[34] Andrea Rendl. *Effective compilation of constraint models*. PhD Thesis, University of St Andrews, 2010.

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

[36] Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1) :2 :1–2 :43, December 2008.

[37] Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints*, 18(1) :75–107, January 2013.

[38] Boro Sofranac, Ambros Gleixner, and Sebastian Pokutta. Accelerating domain propagation : An efficient GPU-parallel algorithm over sparse matrices. *Parallel Computing*, 109 :102874, March 2022.

[39] Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1) :38, 2019.

[40] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.

[41] Pierre Talbot, Frédéric Pinel, and Pascal Bouvry. A Variant of Concurrent Constraint Programming on GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3830–3839, Jun. 2022.

[42] Yujin Tang, Yingtao Tian, and David Ha. Evojax : Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 308–311, 2022.

[43] Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU : A case study for the AllDifferent constraint. *Journal of Logic and Computation*, 33(8) :1734–1752, December 2023.

[44] Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU : A case study for the cumulative constraint. *Constraints*, 29 :192–214, 2024. Publisher : Springer.

[45] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[46] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3) :139–164, 1998.

[47] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. Compiling Tree Transforms to Operate on Packed Representations. pages 29 pages, 728133 bytes, 2017. Artwork Size : 29 pages, 728133 bytes ISBN : 9783959770354 Medium : application/pdf Publisher : [object Object].

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