{CUDA}: Set Constraints on GPUs.

Published
DOI:10.13137/2464-8728/33310

Rend. Istit. Mat. Univ. Trieste
Volume 52 (2020), 1–27 (electronic preview)
DOI: 10.13137/0049-4704/xxxxx

{CUDA}: Set Constraints on GPUs[1]

Agostino Dovier, Andrea Formisano,
Enrico Pontelli, and Fabio Tardivo

Dedicated to our advisor and mentor Eugenio G. Omodeo

Abstract. *Set constraints have been introduced in declarative programming languages in the Nineties as a consequence of a broader research on programming with sets and on computable set theory. General Purpose Graphics Processing Units (GPUs), originally developed for graphical purposes (e.g., for high definition video games), emerged recently as a powerful and cheap parallel architecture, widely available in most desktops and laptops computers. This paper presents a constraint solver on set constraints and its parallel implementation on GPUs.*

Keywords: Computable Set Theory, Constraint Logic Programming, Parallelism.
MS Classification 2010: 68T27 Logic in artificial intelligence, 68N17 Logic programming, 68W10 Parallel algorithms in computer science.

1. Introduction

There is no doubt that the notations and the concepts underlying set theory provide powerful instruments to address challenges in computational modeling and resolution of complex problems. Set notations are common in most modeling languages—e.g., ranging from the "old" Z language [2, 35] to the more modern constraint-based modeling languages like Minizinc [37]. The concepts of sets are the foundation of the formal as well as intuitive semantics of many programming languages—e.g., the operational semantics of Answer Set Programming [29, 30] is best intuitively described in terms of constraints over sets of atoms. This raises the natural question of how one could directly *compute with sets.*

During the Eighties, and beyond, we witnessed the development and growth of a community of researchers, initially originating from the Courant Institute at New York University, focused on the exploration of theoretical and practical aspects of *computable set theory* [24]. These theoretical results provided

[1]The research pursued in this paper is partially supported by Indam GNCS grants and by Uniud PRID ENCASE.

the foundations of a wealth of research efforts, many exploring the integration of different classes of sets as native and first-class citizens of programming languages. In the area of imperative programming languages, the work on languages like SETL [33], and more recently JSetL [31], explored the benefits of native set data structures to support modeling and embed non-determinism in the imperative computation. Grounded in seminal work in the field of deductive databases (e.g., [1, 3]), computable set theory found a natural avenue of expression within the logic programming paradigm. Initially, these concepts inspired natural extensions of traditional Horn clause logic, e.g., as in the LPS proposal [28], the desiderata expressed in [34], and the complex logic language proposed in [27].

A common thread in these seminal efforts is the work of Eugenio G. Omodeo. Omodeo represents one of those rare researchers who has been able to link the theoretical foundations of computable set theory, as in [5], to the practical aspects of sets in programming languages, as in [14]. His foundational work represents the inspiration of generations of logic programming researchers and offers the building blocks for the concepts presented in this paper.

Researchers working on embedding computational aspects of set theory in programming languages, especially in logic programming, soon realized that the inherent non-determinism of set operations is better accommodated by a constraint-based framework, leading to different constraint logic programming frameworks based on sets [14, 25, 18]. Resolution of constraints over sets leads to complex computational challenges, as explored in the studies on set unification [23] and disunification [17], and have been parametrically extended to multisets and other data structures [20, 19]. Research in constraint solving over sets can be, in broad strokes, separated along two complementary strands. The efforts described in [14, 15, 13, 18, 21, 22, 16] offer very general approaches to set constraints, enabling complexities such as nested sets, partially defined sets, intensional set constructors, and even hypersets. These approaches provide very general modeling instruments, at the price of high computational costs. Recent efforts, such as those in [8, 7], have taken advantage of imperative programming features and a wealth of optimizations to allow the use of such general constructs in solving practical challenges (e.g., verification of security properties [6]).

The complementary direction is exemplified by the work on set constraints by Gervet [25, 32], which restricts the focus on simpler forms of sets (e.g., finite, non-nested) with the advantage of enabling more efficient forms of propagation and resolution. In particular, the work by Gervet explores modeling of problems using *intervals of sets*, applying propagation on the corresponding $\subseteq$-lattice. The latter approach provides effective modeling capabilities coupled with efficient computational mechanisms.

Minizinc has emerged over the years as one of the most popular modeling languages in the area of constraint programming. It is the default language adopted by the constraint programming community, where it is used in the international constraint solvers challenge organized yearly since 2008 [36]. A Minizinc program is compiled into a flat (unfolded) version called Flatzinc that the various solvers participating to the competition should be capable of interpreting. Flatzinc is a sort of *Assembly* for constraint programming. Among the constraint domains Minizinc is capable of dealing with there are finite-domains and sets in the style proposed by Gervet [25].

The overarching goal of this paper is to advance the state of the art in efficient resolution of the set constraints found in Minizinc. We aim to demonstrate the potential of parallelism to enhance efficiency and scalability of set constraint resolution. In particular, we propose to explore the use of *General Purpose Graphics Processing Units (GPUs)* in managing Minizinc set constraints—i.e., finite sets of integers, ranging over clearly defined $\subseteq$-lattices. GPUs support fine grained parallelism, particularly suitable for the manipulation of regular data structures (e.g., matrices). GPUs have been demonstrated to be effective in various relevant areas, such as constraint reasoning, logic programming, and satisfiability (e.g., [4, 9, 11, 12, 10]). We illustrate how different forms of propagation for the different set constraints of Minizinc can be mapped to GPU computations; we provide experimental assessments of the parallel performance realized in a prototype solver that is available for download at `http://clp.dimi.uniud.it/sw/`.

2. The set interval calculus

The *set interval calculus* [25] deals with subsets of a domain set X. In this paper, we focus on finite sets.

Let us consider the lattice $\mathcal{D} = (\wp(\mathcal{X}), \subseteq)$. The lattice is *bounded* by the least element $\emptyset$ and by the greatest element X. Given $s, t \in \mathcal{D}$ with $s \subseteq t$, the *set interval* $[s, t]$ is defined as $[s, t] = \{z \in \mathcal{D} : s \subseteq z \wedge z \subseteq t\}$. [1] Let us observe that $[\emptyset, \mathcal{X}] = \mathcal{D}$. Moreover, $|[s, t]| = 2^{|t \setminus s|}$.

A set $C \subseteq \mathcal{D}$ is *convex* if for every pair $x, y \in C$ it holds that $x \cap y \in C$ and $x \cup y \in C$. The closure on pairs of elements is sufficient to guarantee the closure on any finite number of elements. Since we are dealing with finite sets only, this implies that, if C is convex, for any set $S \subseteq C$ it holds that $\bigcap_{s \in S} s \in C$ and $\bigcup_{s \in S} s \in C$.

In Figure 1 we give examples of set intervals and in particular on how memberhip or not membership of a single element can be used to split the set intervals into two set intervals.

LEMMA 2.1. *Let X be a finite set and $m \subseteq M \subseteq X$. Then $[m, M]$ is convex.*

[1] For simplicity, we denote with $x \in \mathcal{D}$ an element $x \in \wp(\mathcal{X})$, namely a subset of $\mathcal{X}$.

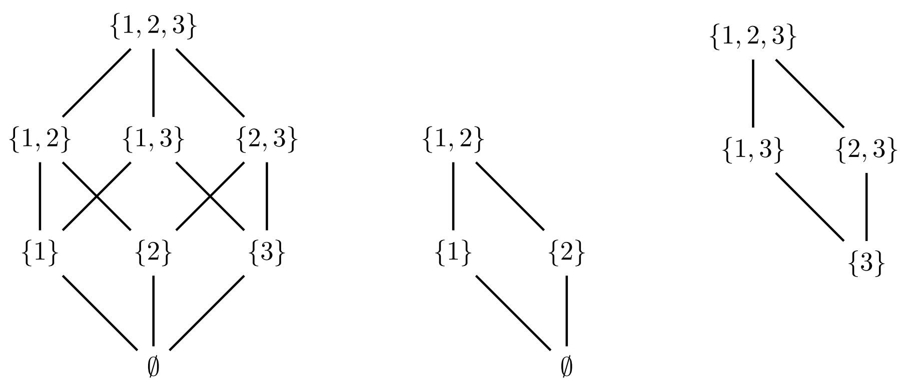

Figure 1: From left to right, the lattice $\mathcal{D} = (\wp(\{1,2,3\}), \subseteq)$ and its sublattices consisting of the sets that do not contain the number 3 and of those containing the number 3. They are all convex and, precisely, the set intervals $[\emptyset, \{1,2,3\}], [\emptyset, \{1,2\}], [\{3\}, \{1,2,3\}]$.

Proof. Let us consider the set interval $[m, M]$. Let $s, t \in [m, M]$. By definition, $m \subseteq s \subseteq M$ and $m \subseteq t \subseteq M$, thus $m \subseteq s \cap t \subseteq M$ and $m \subseteq s \cup t \subseteq M$, namely they belong to the set interval, and this proves that it is convex. $\square$

Let us observe that there are convex sets that cannot be represented as set intervals; for example, $\{\emptyset, \{1\}, \{2,3\}, \{1,2,3\}\}$.

LEMMA 2.2. *Let $\mathcal{D} = (\wp(\mathcal{X}), \subseteq)$ be a lattice, and $C \subseteq \mathcal{D}$ be a convex subset of $\mathcal{D}$. Let $x \in \mathcal{X}$. Then $C_1 = \{s \in C : x \in s\}$ and $C_2 = \{s \in C : x \notin s\}$ are convex.*

Proof. If x belongs to all elements of C then $C_1 = C$ and $C_2 = \emptyset$. Then C_1 is convex by hypothesis, and C_2 is trivially convex. Assume this is not the case and let $s, t \in C_1$ and, hence, in C. Since C is convex $s \cap t \in C$ and $s \cup t \in C$. By definition of C_1, $x \in s$ and $x \in t$, thus $x \in s \cap t$ and $x \in s \cup t$. Then $s \cap t$ and $s \cup t$ are in C_1.
If x belongs to no element of C then $C_1 = \emptyset$ and $C_2 = C$. Then C_2 is convex by hypothesis, and C_1 is trivially convex. Assume this is not the case and let $s, t \in C_2$ and, hence, in C. Since C is convex $s \cap t \in C$ and $s \cup t \in C$. By definition of C_2, $x \notin s$ and $x \notin t$, thus $x \notin s \cap t$ and $x \notin s \cup t$. Then $s \cap t$ and $s \cup t$ are in C_2. $\square$

COROLLARY 2.3. *Given a set interval $[m, M]$ of $\mathcal{D}$ and $x \in M \setminus m$, then $\{s \in [m, M] : x \in s\} = [m \cup \{x\}, M]$ and $\{s \in [m, M] : x \notin s\} = [m, M \setminus \{x\}]$.*

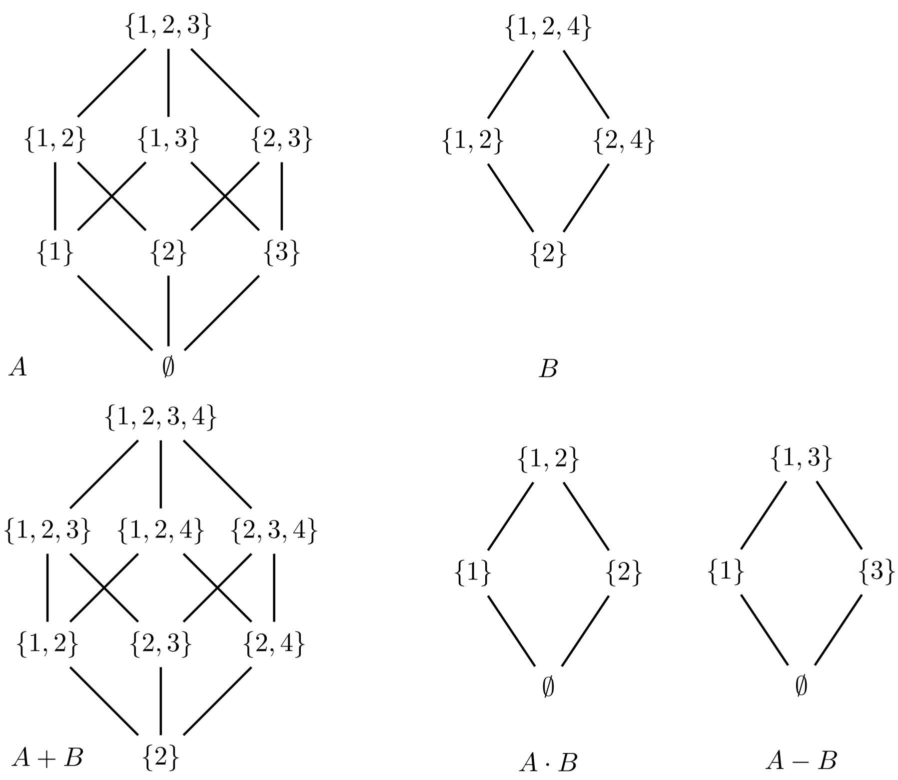

Figure 2: The set intervals $A = [\emptyset, \{1, 2, 3\}]$ and $B = [\{2\}, \{1, 2, 4\}]$ and the three operations applied to them

Definition 2.4. *Let us define the following binary operations $+, \cdot, -$ on set intervals. Let $A = [m_A, M_A]$ and $B = [m_B, M_B]$ be two set intervals.*

$$\begin{array}{rcccl} A + B & = & [m_A, M_A] + [m_B, M_B] & = & [m_A \cup m_B, M_A \cup M_B] \\ A \cdot B & = & [m_A, M_A] \cdot [m_B, M_B] & = & [m_A \cap m_B, M_A \cap M_B] \\ A - B & = & [m_A, M_A] - [m_B, M_B] & = & [m_A \setminus M_B, M_A \setminus m_B] \end{array}$$

In Figure 2 we give examples of the applications of the just defined operations on set intervals. The operations $+, \cdot, -$ are not $\cup, \cap, \setminus$, but they will be used later in the propagation of constraints involving the corresponding operator.

Lemma 2.5. *Let $A = [m_A, M_A]$ and $B = [m_B, M_B]$ two set intervals.*

1. *$r \in A \, op \, B$ if and only if there are $s \in A$ and $t \in B$ such as $r = s \, \hat{op} \, t$, where op and $\hat{op}$ are $+, \cdot, -$, and $\cup, \cap, \setminus$, respectively.*

2. $A \cap B = \{s : s \in A \wedge s \in B\} = [m_A \cup m_B, M_A \cap M_B]$.

3. $A \cup B = \{s : s \in A \vee s \in B\}$ *is not guaranteed to be a set interval.*

4. $A \setminus B = \{s : s \in A \wedge s \notin B\}$ *is not guaranteed to be a set interval.*

Proof. 1. $op = +$($\leftarrow$) Let $s \in A$ and $t \in B$. Then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq t \subseteq M_B$. By monotonicity, it holds that $m_A \cup m_B \subseteq s \cup t \subseteq M_A \cup M_B$, namely $r = s \cup t \in A + B$.
($\rightarrow$) If $r \in A + B$ then $m_A \cup m_B \subseteq r \subseteq M_A \cup M_B$. Let $s = r \cap M_A$ and $t = r \cap M_B$. Observe that $r = s \cup t$. Then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq t \subseteq M_B$.

$op = \cdot$($\leftarrow$) Let $s \in A$ and $t \in B$. Then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq t \subseteq M_B$. By monotonicity, it holds that $m_A \cap m_B \subseteq s \cap t \subseteq M_A \cap M_B$, namely $r = s \cap t \in A \cdot B$.
($\rightarrow$) If $r \in A \cdot B$ then $m_A \cap m_B \subseteq r \subseteq M_A \cap M_B$. Let $s = r \cap M_A$ and $t = r \cap M_B$. Observe that $r = s \cup t$. Then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq t \subseteq M_B$.

$op = -$($\leftarrow$) Let $s \in A$ and $t \in B$. Then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq t \subseteq M_B$. Since $t \subseteq M_B$ and $m_A \subseteq s$, we have that

$$m_A \setminus M_B \subseteq m_A \setminus t \subseteq s \setminus t$$

Since $m_B \subseteq t$ and $s \subseteq M_A$, we have that

$$s \setminus t \subseteq s \setminus m_B \subseteq M_A \setminus m_B$$

and thus $r = s \setminus t \in A - B$.
($\rightarrow$) If $r \in A - B$ then $m_A \setminus M_B \subseteq r \subseteq M_A \setminus m_B$. Thus, $r = (m_A \setminus M_B) \cup u$ with u disjoint from $m_A \setminus M_B$. Let us observe that, since $r \subseteq M_A \setminus m_B$, we have that $u \subseteq M_A$ and it is disjoint from m_B. Let $s = m_A \cup u$ and $t = M_B \setminus u$. Then $r = s \setminus t$. Now, $m_A \subseteq s$ by definition, and, as observed, $s \subseteq M_A$, thus $s \in A$. As far as t is concerned, $t \subseteq M_B$ by definition. Since u is disjoint from m_B then $M_B \setminus u \supseteq m_B$, thus $t \in B$.

2. If $s \in A$ and $s \in B$ then $m_A \subseteq s \subseteq M_A$ and $m_B \subseteq s \subseteq M_B$. Thus $m_A \cap m_B \subseteq s \subseteq M_A \cap M_B$.
3. Consider $A = [\{1\}, \{1\}]$ and $B = [\{2\}, \{2\}]$. $A \cup B = \{\{1\}, \{2\}\}$ which is not a set interval.
4. Consider $A = [\emptyset, \{1, 2\}]$ and $B = [\{1\}, \{1\}]$. $A \setminus B = \{\emptyset, \{2\}, \{1, 2\}\}$ which is not a set interval.

$\square$

3. Syntax

In this section, we review some specific fragments of the syntax of the Minizinc constraint programming language; in particular, we will focus on two basic sorts: `int` and `set`.

3.1. The sort `int`

Constants of sort `int` are integer numbers in $\mathbb{Z}$. A variable X of sort `int` is defined as a finite domain variable ranging on a domain D_X. D_X is typically initially expressed as an interval $x..y$ with $x, y \in \mathbb{Z}$ and $x \leq y$. The domain can can also be defined in an extensional manner by enumerating its elements; for example, the statement

$$\texttt{var}\,\{1,2,5\} : X;$$

describes a variable X whose domain is $\{1, 2, 5\}$. Variables and constants of sort `int` can be used to build arithmetic expressions using the common arithmetic operators $+, -, *, /, \texttt{mod}$.

Given two arithmetic expressions ℓ and r, we can define primitive constraints of the form $\ell \;\; op \;\; r$. Predicate symbols that can be used as *op* are $=, <, \leq$. Primitive constraints can be combined with Boolean operators to build complex constraints on finite domains. Their negation can be expressed in general as $\texttt{not}\ (\ell \;\; op \;\; r)$; however, $\ell \,!\!= r$ can be used as a syntactic sugar for $\texttt{not}\ (\ell = r)$, $r < \ell$ for $\texttt{not}\ (\ell \leq r)$, and $r \leq \ell$ for $\texttt{not}\ (\ell < r)$.

The sort `int` is sufficiently expressive to allow us to encode NP complete problems even using only conjunctions of primitive constraints and without making use of arithmetic operators. For instance, the instance of the 3-coloring problem on a graph with:

nodes: $\{1, 2, 3, 4, 5\}$ and edges: $\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 5\}, \{3, 5\}\}$

can be expressed as

$$X_1 \neq X_2 \land X_1 \neq X_3 \land X_1 \neq X_4 \land X_2 \neq X_5 \land X_3 \neq X_5$$

where all variables have domains $\{1, 2, 3\}$, representing the three colors.

3.2. The sort `set`

The sort `set` is populated by *set terms*. The empty set $\emptyset$ (denoted by $\{\}$) is a set term. An extensional set $\{e_1, \ldots, e_n\}$, where each e_i is an arithmetic expression of sort `int`, is a set term as well. Without loss of generality, we require $e_i \in \mathbb{Z}$ or e_i to be a variable of sort `int`. In concrete syntax, if the set is an interval $\{x, x+1, x+2, \ldots, y\}$ it can be represented as $x..y$. An extensional set without variables is said to be *ground*.

A variable S of sort `set` is assigned a finite domain domain D_S that can be defined in the same way as the domain for `int` variables, i.e., either as an interval $x..y$ or as an enumeration of values. Its value ranges on the *subsets* of D_S. For instance, the variable defined as `var set of` $\{1,2,5\} : S$; is allowed to assume the eight values $\emptyset, \{1\}, \{2\}, \{5\}, \{1,2\}, \{1,5\}, \{2,5\}, \{1,2,5\}$. A variable of sort set is a set term.

A predefined ordering on sets based on a lexicographic ordering of the sorted set form is assumed in Minizinc; for example, $\{1,2\}$ is in sorted set form while $\{2,1\}$ is not. However, it holds that $\{1,2\} = \{2,1\}$.

Set terms can be used to build set expressions using the common set operators $\cup, \cap, \setminus$; the concrete syntax used to represent these operators is `union`, `intersect`, and `diff`.

Given two set expressions ℓ and r, primitive constraints over ℓ and r are of the form $\ell \ \mathit{op} \ r$. Predicate symbols that can be used as *op* are $=, \subseteq$—in concrete syntax: $=$, `subset`. Moreover, a primitive constraint $X \in S$, where X is of sort `int` and S is a set term, can be used.

Primitive constraints can be combined with Boolean operators to build complex set constraints. Negation of primitive constraints can be expressed using `not` (with the usual syntactic sugar $!=$ for the negation of equality). Let us observe that the sort `set` is sufficiently expressive to encode NP-complete problems using only conjunctions of equality constraints. For instance, the following instance of SAT:

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_2 \vee \neg X_3 \vee X_4) \wedge (\neg X_1 \vee \neg X_2)$$

can be encoded as (where N_i takes the role of $\neg X_i$)

$$\begin{array}{l}
\{X_1, N_1\} = \{0,1\} \wedge \{X_2, N_2\} = \{0,1\} \wedge \\
\{X_3, N_3\} = \{0,1\} \wedge \{X_4, N_4\} = \{0,1\} \wedge \\
\{X_1, N_2, X_3, X_4, 0\} = \{0,1\} \wedge \{X_2, N_3, X_4, 0\} = \{0,1\} \wedge \{N_1, N_2, 0\} = \{0,1\}
\end{array}$$

This approach was originally presented in [14] for the constraint logic programming language $\{\log\}$—where the encoding can be captured by a single equation (thanks to the ability of nesting sets):

$$\begin{array}{l}
\{\{X_1, N_1\}, \{X_2, N_2\}, \{X_3, N_3\}, \{X_4, N_4\}, \\
\quad \{X_1, N_2, X_3, X_4, 0\}, \{X_2, N_3, X_4, 0\}, \{N_1, N_2, 0\}\} = \{\{0,1\}\}
\end{array}$$

Note that the nesting of sets is not allowed in Minizinc.

Additional operators that can be used in the set-based constaint language include cardinality operators (`card`(s) for a set expression s) and operators to determine the minimum/maximum element of a set expression (`min`(s) and `max`(s)).

3.3. Set operations in Minizinc

Let us summarize the built-in operations for sets supported by Minizinc. Their negation can be written by anticipating `not`.

- **set of** x**..**y Returns the set $\{x, \ldots, y\}$
- X **in** S Enforces that $X \in S$
- s **subset** t (or, equivalently, t **superset** s) States that $s \subseteq t$
- $s = t$ Set equality, equivalent to s **subset** t and t **subset** s.
- s **intersect** t Returns the set $s \cap t$
- s **union** t Returns the set $s \cup t$
- s **diff** t Returns the set $s \setminus t$. Let us observe that if u is assigned to the "universe" set, then u `diff` s defines $\bar{s}$.
- s **symdiff** t Returns $s \triangle t = (s \setminus t) \cup (t \setminus s)$
- **array_intersect(**v**)** Returns the intersection of the sets in array v (unary intersection)
- **array_union(**v**)** Returns the union of the sets in array v (unary union)
- **card(**s**)** Returns the cardinality of the set s.
- **max(**s**)/min(**s**)** Returns the maximum/minimum (value of the elements) of the set s

4. Constraint solving

A *constraint satisfaction problem* (briefly, a CSP) is a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains for the variables. We denote as $D_X \in \mathcal{D}$ the domain of the variable $X \in \mathcal{X}$. Moreover, $\mathcal{C}$ is a set of constraints on subsets of variables of $\mathcal{X}$ and a constraint is a relation on Cartesian products of subsets of $\mathcal{D}$. In other words, a k-ary constraint c over variables $X_1, \ldots, X_k$ is a relation $c \subseteq D_{X_1} \times \cdots \times D_{X_k}$. For instance, a binary constraint c on the variables X, Y is a relation $c \subseteq D_X \times D_Y$. All pairs (tuples) in c are said to satisfy the constraint c. A *solution* to a CSP is an assignment $\sigma : \mathcal{X} \longrightarrow \cup_{D \in \mathcal{D}} D$ such that for all $X \in \mathcal{X}$ it holds that $\sigma(X) \in D_X$ and all constraints in $\mathcal{C}$ are satisfied by the assignment.

Constraint propagation is a fixpoint procedure that allows us to remove elements from the domains of variables which cannot appear in any solution of the CSP. In a typical constraint solving procedure, constraint propagation alternates with non-deterministic variable assignments, until either a solution is found (i.e., all the variables have been assigned a value) or one of the domain becomes empty. The latter case indicates the unsatisfiability of the constraint. Constraint propagation allows us to prune the search tree, reducing its overall size, and it is repeated at each node of the tree; thus, any speed-up in its implementation immediately impacts the performance of the overall resolution procedure.

In this paper, we focus on the possible performance improvements that can be obtained by exploiting the *Single Instruction Multiple Threads (SIMT)* parallelism supported by modern *General-Purpose Graphical Processing Units (GPUs)* and exploited through the use of programming paradigms like CUDA. In particular, we are interested in using CUDA to improve performance of constraint propagation for set constraints.

Let us start with a quick review of some general definitions related to constraint propagation. Constraint propagation is primarily based on the notions of *arc* or *bounds* consistency for binary constraints, and on *generalized arc/bounds consistency* for *global* constraints dealing with more than two variables (organized as lists of variables). A binary constraint c on the variables X and Y is said to be *arc consistent* if

- for every element $x \in D_X$ there is an element $y \in D_Y$ such that $(x, y) \in c$, and
- for every element $y \in D_Y$ there is an element $x \in D_X$ such that $(x, y) \in c$

Namely, every element of one of the two domains is *supported* by at least one element of the other domain. In order to obtain arc consistency, we need to repeatedly remove elements in the domains which are not supported. In the worst case, obtaining arc consistency of a single constraint requires time $O(|D_X| \cdot |D_Y|)$. The process of achieving arc consistency is repeated for each constraint as part of a fixpoint procedure, until no further additional domain reductions are possible.

If the domains D_X and D_Y are large, an approximated version of the above rule is often used, which focuses exclusively on the 'bounds' of the domains. The notion of bound depend on the constraint system considered. In the case of finite domains, the domains D_X and D_Y could be approximated by the intervals $\min(D_X)..\max(D_X)$ and $\min(D_Y)..\max(D_Y)$. In the case of sets, by the lower bound and the upper bound of the set interval. The fact that $\subseteq$ does not induce a total order makes this approximation, in a sense, weaker than the one of finite domains.

A binary constraint c on the variables X and Y where $m_X = \min D_X, M_X = \max D_X, m_Y = \min D_Y, M_Y = \max D_Y$, is said to be *bounds consistent* if

1. $(\exists b \in m_Y..M_Y)\,((m_X, b) \in c)$ and $(\exists b \in m_Y..M_Y)\,((M_X, b) \in c)$,
2. $(\exists a \in m_X..M_X)\,((a, m_Y) \in c)$ and $(\exists a \in m_X..M_X)\,((a, M_Y) \in c)$.

Namely, the bounds of the two domains are *supported* by at least one point within the bounds of the other domain.

EXAMPLE 4.1: For instance, let us consider the constraint $X = 2Y$ betweeen the finite-domain variables such that $D_X = 0..5$ and $D_Y = 0..3$. By updating D_X into $D'_X = 0..4$ and D_Y into $D'_Y = 0..2$ we reach bounds consistency. Let us remark that the points 1 and 3 in D'_X are not supported by points in D'_Y and they should be eliminated if we wish to obtain arc consistency.

Let us consider the constraint $S \subseteq T$ between set variables with set interval domains $D_S = [\{1\}, \{1, 2, 3\}]$ and $D_T = [\{0\}, \{0, 1, 2\}]$. $\{1\}$ in D_S is supported by $\{0, 1, 2\}$ in D_T. Similarly, $\{0, 1, 2\}$ in D_T is supported by $\{1\}$ in D_S. The other two bounds are not supported and bounds consistency can be obtained by updating $D'_S = [\{1\}, \{1, 2\}]$ and $D_T = [\{0, 1\}, \{0, 1, 2\}]$.

However, in the case of constraints on set variable, the non linearity of the order might prevent us in mantaining bounds consistency using set intervals for representing sets. For instance, consider the constraint $X \in S$ where $D_X = 1..2$ and $D_S = [\{0\}, \{0, 1, 2\}]$. The two bounds of D_X are supported by $\{0, 1, 2\}$. Similarly $\{0, 1, 2\}$ in D_S is supported (either by 1 or by 2). Instead, the bound $\{0\}$ of D_S is supported by no points of D_X. However, by removing $\{0\}$ from the set interval we'd obtain $D_S = \{\{0, 1\}, \{0, 2\}, \{0, 1, 2\}\}$ which is no longer a set interval.

Bounds consistency affects only the bounds, it can be implemented faster, but it reduces the effectiveness of pruning:

- Arc consistency generates a smaller search tree but with a larger computation time at each node, while
- Bounds consistency generates a larger search tree in a faster manner at each node.

In practice, removal of unsupported values is delayed in the lower parts of the search tree. The NP hardness of solving a CSP on these domains guarantees that we can find examples in which the first technique performs better and others in which it performs worse.

In the case of integer domains, one can adopt both bounds consistency and arc consistency by, e.g., storing domains using bitmaps (see Section 5.1). As pointed out by Gervet [26], the case of set domains does not allow us to deal with explicit representation of domains due to the intrinsic combinatorial explosion in their sizes and thus we are forced to implement a weak form of bounds consistency. As shown in the example above, even mantaining bounds consistency would lead us outside set interval representation.

4.1. Arc consistency and Integer constraints

Ensuring arc and bounds consistency of binary constraints on finite domains is one of the most studied problems in constraint programming and all constraint solvers implements extremely fast solvers. We will not enter here into much details. We just focus on one example that clarifies the effectiveness of strength of constraint propagation (and the polynomial limits).

EXAMPLE 4.2: Let us consider this self-contained fragment of Minizinc code, where a vector of n finite domains variables x with domain $[1, n]$ is constrained such as to ensure that $x[i] < x[i+1]$ for every i.

```
array [1..n] of var 1..n: x;
constraint forall(i in 1..n-1)( x[i] < x[i+1] );
```

Enforcing arc consistency deterministically produces the unique answer $x[i] = i$ for all i. All constraints in the first step and progressively some of them are not considered in the successive step, and in overall a quadratic computation is generated. Just to give a taste, on a common desktop Core i7, 2.30 GHz, Win 10, for $n = 4k, 8k, 12k, 16k, 20k$ running times are roughly $0.6s, 1.4s, 2.5s, 4.9s, 5.5s$ with the default solver of Minizinc 2.5.3.

4.2. Consistency and set constraints

We will make use of the following notation (possibly subscripted):

- Lower case letters $x, y, z, \dots$ to denote integer numbers
- Upper case letters $X, Y, Z, \dots$ to denote integer variables
- Upper case letters $A, B, C, S, T, \dots$ to denote set variables
- Lower case letters $a, b, c, s, t, \dots$ to denote sets.

For a set variable S, we denote with $\perp_S = \bigcap_{s \in D_S} s$ the greatest lower bound (glb) of S and with $\top_S = \bigcup_{s \in D_S} s$ the least upper bound (lub) of S. These two values represent the extremes of the domain if viewed as a set interval. Note that these values might not be part of the domain of S if it is not convex.

We denote with $'$ the new values of the domains. Most rewriting rules can introduce empty domains: if a domain becomes *empty*, then propagation ends with a failure. A set interval $[m, M]$ is empty when $m \not\subseteq M$ (i.e., when there is $x \in m$ such that $x \notin M$).

If a constraint is satisfied by all values of the domain, the constraint is *freezed*, i.e., it will not be considered in future propagations in the same branch of the search tree.

For $U \in \{A, B, C\}$ the corresponding set interval domain is denoted as $D_U = [m_U, M_U]$. We analyze the propagation in general (first) and its fast encoding if the domains are (set) intervals.

Set equality and inequality.

- $A = B$: $D'_A = D'_B = D_A \cap D_B$.

  Interval case: $D'_A = D'_B = D_A \cap D_B = [m_A \cup m_B, M_A \cap M_B]$

- $A \neq B$: if $D_A \cap D_B = \emptyset$ then the constraint is satisfied.

  If $|D_A| = 1$ then $D'_B = D_B \setminus D_A$ and if $|D_B| = 1$ then $D'_A = D_A \setminus D_B$. Let us observe that one (or both) between D'_A or D'_B might become empty.

  Otherwise no domain update is applied.

  Interval case: If $D_A \cap D_B = [m_A \cup m_B, M_A \cap M_B] = \emptyset$ (emptyness can be checked just analyzing the interval bounds) then the constraint is satisfied. This is the unique test in this case. Let us observe that the singleton case cannot be implemented in intervals. For instance, if $D_A = [\{1\}, \{1\}], D_B = [\emptyset, \{1, 2\}]$ then $D'_B = \{\emptyset, \{2\}, \{1, 2\}\}$ which is not an interval.

Membership.

- $X \in A$: $D'_X = D_X \cap \top_A$, $D'_A = \{s \in D_A : s \cap D_X \neq \emptyset\}$.
  Interval case: $D'_X = D_X \cap M_A$. If $D'_X \subseteq m_A$ then the constraint is satisfied. If $D'_X \cap m_A \neq \emptyset$ then $D'_A = D_A$ (if X is assigned in m_A all sets in D_A will be ok). Otherwise, if $D'_X = \{x_1, \ldots, x_k\} \subseteq M_A \setminus m_A$, a complete propagation could be obtained by setting $D'_A = [m_A \cup \{x_1\}, M_A] \cup \cdots \cup [m_A \cup \{x_k\}, M_A]$, namely a union of intervals. Since we avoid dealing with collections of intervals, we can restrict this case to $k = 1$, namely:
  If $D'_X = \{x_1\}$ and $x_1 \notin m_A$ then $D'_A = [m_A \cup \{x_1\}, M_A]$ else $D'_A = D_A$

  Example: $D_X = \{1, 2, 3\}, D_A = [\{4\}, \{1, 2, 4\}] = \{\{4\}, \{1, 4\}, \{2, 4\}, \{1, 2, 4\}\}$. $D'_X = \{1, 2, 3\} \cap \{1, 2, 4\} = \{1, 2\}$. Arc consistency will lead us to $D'_A = \{\{1, 4\}, \{2, 4\}, \{1, 2, 4\}\} = [\{1, 4\}, \{1, 2, 4\}] \cup [\{2, 4\}, \{1, 2, 4\}]$. However, since D'_A is a non-convex subset domain we prefer keeping $D'_A = D_A$.

- $X \notin A$. $D'_X = D_X \setminus \bot_A$, $D'_A = \{s \in D_A : D_X \setminus s \neq \emptyset\}$.
  Interval case: $D'_X = D_X \setminus m_A$. If $D'_X = \{x_1\}$ and $x_1 \in M_A$ then $D'_A = [m_A, M_A \setminus \{x_1\}]$ else $D'_A = D_A$.
  Example: $D_X = \{2, 3, 4\}, D_A = [\{2, 3\}, \{2, 3, 4, 5\}]$. $D'_X = \{2, 3, 4\} \setminus \{2, 3\} = \{4\}$, $D'_A = [\{2, 3\}, \{2, 3, 5\}]$.

Set Inclusion.

- $A \subseteq B$. $D'_A = \{s \in D_A : (\exists t \in D_B)(s \subseteq t)\}$, $D'_B = \{t \in D_B : (\exists s \in D_A)(s \subseteq t)\}$.
  Interval case: $D'_A = [m_A, M_A \cap M_B]$ and $D'_B = [m_A \cup m_B, M_B]$.
  Example: $D_A = [\{2, 3, 4\}, \{1, 2, 3, 4, 5\}]$, $D_B = [\{2, 3, 5\}, \{2, 3, 4, 5, 6\}]$. $D'_A = [\{2, 3, 4\}, \{2, 3, 4, 5\}]$, $D'_B = [\{2, 3, 4, 5\}, \{2, 3, 4, 5, 6\}]$.

Set operations and extensional sets.

- **Union.** $A = B \cup C$. $D'_A = D_A \cap \{s \cup t : s \in D_B, t \in D_C\}$, $D'_B = \{s \in D_B : (\exists t \in D_C)(s \cup t \in D_A)\}$, $D'_C = \{s \in D_C : (\exists t \in D_B)(s \cup t \in D_A)\}$.
  Interval case: Note that, in order to be satisfiable, we need to have that $m_B \subseteq M_A$ and $m_C \subseteq M_A$; $D'_A = [m_A \cup m_B \cup m_C, M_A \cap (M_B \cup M_C)]$
  $D'_B = [m_B \cup (m_A \setminus M_C), M_B \cap M_A]$ and $D'_C = [m_C \cup (m_A \setminus M_B), M_C \cap M_A]$.
  Example: $D_A = [\{1,2\}, \{1,2,3,4,7,8\}]$, $D_B = [\{1,3\}, \{1,2,3,6,9\}]$, $D_C = [\{2,4\}, \{2,4,5,10\}]$. $D'_A = [\{1,2,3,4\}, \{1,2,3,4\}]$, $D'_B = [\{1,3\}, \{1,2,3\}]$, $D'_C = [\{2,4\}, \{2,4\}]$.
- **Intersection.** $A = B \cap C$: $D'_A = D_A \cap \{s \cap t : s \in D_B, t \in D_C\}$, $D'_B = \{s \in D_B : (\exists t \in D_C)(s \cap t \in D_A)\}$, $D'_C = \{t \in D_C : (\exists s \in D_B)(s \cap t \in D_A)\}$.
  Interval case: $D'_A = [m_A \cup (m_B \cap m_C), M_A \cap M_B \cap M_C)]$
  $D'_B = [m_A \cup m_B, M_B]$, $D'_C = [m_A \cup m_C, M_C]$
- **Difference.** $A = B \setminus C$: $D'_A = D_A \cap \{s \setminus t : s \in D_B, t \in D_C\}$, $D'_B = \{s \in D_B : (\exists t \in D_C)(s \setminus t \in D_A)\}$, $D'_C = \{s \in D_C : (\exists t \in D_B)(s \setminus t \in D_A)\}$.
  Interval case: $D'_A = [m_A \cup (m_B \setminus M_C), M_A \cap (M_B \setminus m_C)]$
  $D'_B = [m_B \cup m_A, M_B \cap (M_A \cup M_C)]$, $D'_C = [m_C \cup (m_B \setminus m_A), M_C]$.
  Example: $D_A = [\{1,2\}, \{1,2,3,4,5,7,8\}]$, $D_B = [\{1,2,3,4\}, \{1,2,3,4,5,6\}]$, $D_C = [\{4\}, \{4,6,9,10\}]$. $D'_A = [\{1,2,3\}, \{1,2,3,5\}]$,
  $D'_B = [\{1,2,3,4\}, \{1,2,3,4,5\}]$, $D'_C = [\{4\}, \{4,6,9,10\}]$.
- $A = \{x_1, \ldots, x_m, X_1, \ldots, X_n\}$ where $x_i \in \mathbb{Z}$ for $i = 1, \ldots, m$. Let $R_D = \{x_1, \ldots, x_m\} \cup D_{X_1} \cup \cdots \cup D_{X_n}$. Then $D'_A = D_A \cap R_D$. $D'_{X_i} = D_{X_i} \cap D_A$.
  Interval case: $D'_A = [m_A \cup \{x_1, \ldots, x_n\}, M_A \cap R_D]$, $D'_{X_i} = D_{X_i} \cap M'_A$

Negative constraints. These constraints are handled via pre-processing, since their explicit handling would lead us to introduce disjunctions.

- $A \not\subseteq B$. This constraint is replaced by $N \in A, N \notin B$ where N is a fresh variable and $D_N = M_A$.
- $A \neq B \, op \, C$ where *op* is $\cup, \cap, \setminus$. Write it as $A \neq N, N = B \, op \, C$ where N is a fresh variable with $D_N = \mathcal{D}$.
- $A \neq \{x_1, \ldots, x_m, X_1, \ldots, X_n\}$ is replaced by $A \neq N$,
  $N = \{x_1, \ldots, x_m, X_1, \ldots, X_n\}$, where N is a fresh variable with $D_N = \mathcal{D}$.

LEMMA 4.3. *Rewriting rules are sound and complete.*

Proof. Rewriting rules for the set representation are exactly the definition of arc consistency in the domain of sets (i.e., using the definitions of set equality, membership, inclusion, and of the set operations). Let us focus on the rules applied to interval domains.

Set equality. If $A = B$ then the two domains should be the same. This is ensured assigning to both of them their intersection $D_A \cap D_B = [m_A \cup m_B, M_A \cap M_B]$ (Lemma 2.5). In the negative case if the intersection is empty then the constraint is simply true. Nothing else is implemented.

Membership. $X \in A$, implies that X should be member of any possible set assigned to A. Thus, we remove from D_X values "external" to the interval: $D'_X = D_X \cap M_A$. If $D'_X \subseteq m_A$, the constraint is trivially satisfied. If $D'_X = \{x\}$ and $x \in M_A \setminus m_A$ we have to remove from D_A all the sets that do not contain x. This is made setting $D'_A = [m_A \cup \{x\}, M_A]$ (and then the constraint is satisfied).

$X \notin A$, implies that X cannot be a member of m_A (hence of all sets of the interval). Thus $D'_X = D_X \setminus m_A$. If D'_X contain one element outside M_A there is always a solution satisfying the constraint. Instead, if $D'_X = \{x\}$ and $x \in M_A \setminus m_A$ we can restrict the interval to $D'_A = [m_A, M_A \setminus \{x\}]$

Set Inclusion. $D_A = [m_A, M_A]$ and $D_B = [m_B, M_B]$. Then $D'_A = [m_A, M_A \cap M_B]$ and $D'_B = [m_A \cup m_B, M_B]$. (Rule I1 in [25]).

Union. $A = B \cup C$. The domain of A should be intersected with that of $B \cup C$ which is computed with the $+$ operation. $D'_A = D_A \cap (D_B + D_C) = [m_A, M_A] \cap [m_B \cup m_C, M_B \cup M_C] = [m_A \cup m_B \cup m_C, M_A \cap (M_B \cup M_C)]$ (see Lemma 2.5). For the converse direction, we can safely remove from D_B and D_C all sets containing points that are not in M_A, thus: $D'_B = [m_B, M_B \cap M_A], D'_C = [m_C, M_C \cap M_A]$

Intersection. $A = B \cap C$. The domain of A should be intersected with that of $B \cap C$ which is computed with the $\cdot$ operation. $D'_A = D_A \cap (D_B \cdot D_C) = [m_A, M_A] \cap [m_B \cap m_C, M_B \cap M_C] = [m_A \cup (m_B \cap m_C), M_A \cap M_B \cap M_C)]$ (see Lemma 2.5) For the converse direction, we can safely remove from D_B and D_C all sets that do not contain the points of m_A, thus: $D'_B = [m_B \cup m_A, M_B], D'_C = [m_C \cup m_A, M_C]$.

Difference. $A = B \setminus C$. The possible sets for A should be intersected with those that can be generated by $B \setminus C$. Then, $D'_A = D_A \cap (D_B - D_C) = [m_A, M_A] \cap [m_B \setminus M_C, M_B \setminus m_C] = [m_A \cup (m_B \setminus M_C), M_A \cap (M_B \setminus m_C)]$ (see Lemma 2.5). For the converse direction, let us observe that D_C can contain sets of arbitrary size as long as they have elements not in D_B (the semantics of the set difference is rather asymmetric). Therefore, its upper bound M_C is not updated.
Instead, if an element x belongs to all sets in D_A (hence, in m_A) it must belong to all sets of B. This can be achieved by 'enlarging' the bottom of the set interval: $m'_B = m_B \cup m_A$.
If an element x belongs to all sets of B (hence, in m_B) and it occurs in

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$D_X = 0..15 = FFFF$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	$D_Y = 0..7 = FF00$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	$D'_X = 0..6 = FE00$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	$D'_Y = 1..7 = 7F00$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Figure 3: The two integer variables X and Y such that there is a constraint $X < Y$ have initial domains D_X and D_Y and get the new domains D'_X and D'_Y. Domains are represented as bitmaps (succinctly written in HEX) and as intervals

no sets of A (hence it is not in M_A), then it must be element of all sets of C and thus in m_C: $m'_C = m_C \cup (m_B \setminus M_A)$.
If an element x belongs to some sets in D_B (hence, in M_B) and it does not occur in sets of D_A (hence is is not in M_A), then either there is some set in D_C that contains it or it must be removed from M_B. This can be achieved as $M'_B = M_B \setminus ((M_B \setminus M_A) \setminus M_C) = M_B \cap (M_A \cup M_C)$.

Extensionally defined set. By the set extensionality principle, $x \in A$ iff $x \in \{x_1, \ldots, x_m, X_1, \ldots, X_n\}$, namely $x = x_i$ for some $i \in \{1, \ldots, m\}$ or $x \in D_{X_j}$ for some $j \in \{1, \ldots, n\}$. Thus, it is safe to remove from D_A all elements that are not allowed in the extensionally defined set. Removing the known elements leaves the domain an interval. Similarly, we can remove from D_{X_j} all elements that are not in D_A.

Negated constraints. of $A \neq Exp$ iff an only if there is a N such that $N = Exp$ and $A \neq N$ (equality axioms). Moreover, by definition of $\subseteq$, $A \not\subseteq B$ iff there is an element $N \in A$ such that $A \notin B$. This justifies the rewriting.

□

5. Toward a GPU-based CSP-solver

In this section we describe the main traits of a prototypical solver for CSPs over set constrains of the forms described in Section 4.2.

5.1. Internal representation of CSPs

We restrict integer domains to subsets of $0..k$ and set-domains to subsets of the set interval $[\emptyset, \{0, 1, 2, \dots, k\}]$ for a given k. See for instance Fig. 3, where, for the sake of simplicity, we set $k = 15$. Notice that, for practical reasons, in the concrete implementation it is convenient to choose $k = 32 * h - 1$, for $h > 0$. This because domains of integer variables are represented as bitmaps of $k + 1$ bits. Each of such bitmap is stored in memory as a sequence of h `unsigned int`. A domain for a variable of sort set is represented as a set interval $[m, M]$ where m and M are both represented as a bitmap of $k + 1$ bits. (The actual value of h is a parameter that can be set by the user.)

Each integer variable is internally referred as a natural number. In the current implementation we bound the number of variables in a CSP to be less that 256, hence each integer variable can be referenced by using a single byte. The same bound/representation is adopted for set variables (hence, there can be at most 256 set variables in a CSP).

Each constraint is internally represented as an `unsigned int`, whose 4 bytes encode the components of the constraint. For instance, consider a constraint c of the form $A_i \mathit{rel} A_j \mathit{op} A_h$, where A_i, A_j, A_h are set variables, *rel* is a relator (i.e., $=$,$\neq$,$\subseteq$,...), and *op* is a set operator (i.e., $\cup$,$\cap$,...). Then, c is compiled into a word of 4 bytes. Let $r(c)$ be such a word. The leftmost byte of $r(c)$ contains a code representing *rel* and *op*, while the remaining three bytes, positionally, encode the three variables (that, as mentioned, are identifiable by single bytes). Such a representation is also adopted for those constraints involving two variables (e.g., of the form $A_i \mathit{rel} A_j$) and integer variables (e.g., $X_i \in A_j$). Also in these cases, the information on the kind of constraint in encoded in the leftmost byte (and the rightmost byte is ignored).

A slightly more complex representation is adopted for constraints c of the form $A_i = \{x_1, \dots, x_m, X_1, \dots, X_n\}$, because of the arbitrary number of elements that may occur in them. An auxiliary array `Exts` of integers is used to store contiguously (the integers representing) $x_1, \dots, x_m$,$X_1, \dots, X_n$, for each constraint of this form. As before the first byte of $r(c)$ encodes the kind of constraint. The second byte stores i, the index of the variable A_i of the l.h.s. of the constraint. The third and fourth bytes of $r(c)$ store the initial positions of $x_1, \dots, x_m$ and of $X_1, \dots, X_n$ in `Exts`, respectively.

Notice that the set of constraints is accessed at each propagation step (see below). The fact that each constraint is compactly represented by a single memory word, makes it possible for parallel CUDA threads to access uniformly the constraint representations, maximizing the bandwidth in memory transfers. The same advantage is obtained for accesses to domains: thanks to the uniform way in which they are represented, all threads concurrently accessing domain extensions, perform essentially the same amount of work. This helps in optimizing thread occupancy.

Algorithm 1: Host code of the CSP-solver (simplified)

procedure CPCS($\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP)
$CSPs = \emptyset$ `/* empty collection of` $CSPs$ `*/`
`InputCompilation`($\mathcal{X}, \mathcal{D}, \mathcal{C}$, $CSPs$) `/* generate internal representation of the input` CSP `*/`
while *$CSPs$ is not empty and no solution has been found* **do**
  select a not empty subset bs of $CSPs$
  foreach *each b in bs* **do in parallel** `/* a CUDA kernel processes` bs `in parallel */`
    remove b from $CSPs$
    `Propagation`(b) `/* update domains of` b `until fixpoint */`
    `CheckSatisfiability`(b, *Status*) `/* check outcome of propagation */`
    if *Status* $==$ *SOLVED* **then** `StoreSolution`() `/* solution found */`
    else if *Status* $\neq$ *UNSAT* **then** `/* select a variable and split its domain and */`
      `DomainSplit`(b, $CSPs$) `/* add the generated problems to` $CSPs$ `*/`
    end
if *exist solutions* **then** output solutions
else return unsatisfiable

5.2. Solving Procedure

As mentioned, to solve a specific instance of a Constraint Satisfaction Problem, the solver proceeds by alternating constraint propagation and (possibly, non-deterministic) variable assignment. This process implicitly searches a solution space that can be thought as tree-shaped. Each node corresponds to a (partially solved) CSP, while each edge corresponds to the updates in the CSP caused by a variable assignment (and the consequent propagations).

Before entering into the details of the procedure, we add here a few remarks on the main features of the parallel architecture employed. GPUs are designed to execute a very large number of concurrent threads on multiple data (the parallel model is known as *Single-Instruction Multiple-Thread (SIMT)*). Each GPU has a number of computing cores *physically* grouped in a collection of so-called *Streaming MultiProcessors (SMs)*. Concurrent threads are scheduled on the SMs and executed in sets of 32, called *warps*. Threads in the same warp are expected (but not forced) to follow the same program address. If this condition is guaranteed, parallelism is maximized, otherwise the *thread divergence* forces serialization and the overall performance decreases.

Threads are *logically* grouped in *blocks* that are organized as a 3D grid (the built-in 3D access was introduced to support the graphical applications of GPUs). A typical CUDA program includes parts meant for execution on the CPU (the *host*) and parts meant for parallel execution on the GPU (the *device*). A *kernel* is a (C) procedure launched by the CPU and running on GPU and its parallel execution is organized by setting the number of blocks and the number of threads per block that will be exploited. The host program contains instructions for device data initialization, grids/blocks/threads configuration, kernel launch, and retrieval of results. GPUs also exhibit a hierarchical memory organization. The threads in the same block share data using high-throughput on-chip shared memory organized in *bank*s of equal dimension. Threads of different blocks can only share data through the off-chip global memory.

To take full advantage of GPU architecture, one has to: distribute the workload among the cores to maximize GPU *occupancy* (exploit all available device resources) and minimize *thread divergence*. Existing serial or parallel solutions need to be substantially re-engineered to become profitably applicable in the context of GPUs.

Going back to the implementation we are presenting, our CUDA-based constraint solver combines two level of parallelism. First, different CSPs are solved in parallel by different CUDA blocks of the same CUDA kernel. This consists in following different paths in the solution space. The paths are guaranteed to be disjoint by the domain-splitting mechanism (see below). Second, each CSP is processed by the CUDA threads of a block, operating in parallel on its constraints and domains.

Algorithm 1 shows the (simplified) code of the solving procedure. After compiling the input CSP (line 2), a collection of *active CSPs* is allocated in the GPU's global memory and is initialized as a set containing the unique generated internal representation. A loop (starting in line 3) is performed until unsatisfiability is detected or a solution is found (actually, a number n of solutions can be required by using a command-line option). This part of the execution is performed on CPU (host). In line 4 a subset bs of the current collection of active CSPs is selected. The cardinality of this subset is specified by a command-line option and determines the number of problems that are processed concurrently by the CUDA blocks: the set CSPs represents a "pool of problems" to be solved: each block picks a different problem b from CSPs in order to solve it. Hence, the inner loop (lines 5–11) is executed in parallel by each CUDA block (on a different problem b) on the GPU (device). Each block performs constraint propagation on b until a fixpoint is reached (line 7). This procedure requires repeated accesses and updates of variable domains. To improve performance it is executed by exploiting the fast shared memory available on chip: initially, the threads of the block perform a coalescent access to global memory to retrieve the representations of domains. These representations are stored in shared memory in order to speed up the subsequent accesses/updates performed during the propagation loop. During each propagation loop constraints are applied to the corresponding variable domains using the rules described in the previous sections.

Each constraint c is processed by ℓ threads —the simple choice is $\ell = 1$, but any value $\ell = 2^i$, for $0 \leq i \leq 5$ is possible (recall that a warp is made of 32 threads). The ℓ threads also access the domains of the variables occurring in c and, if needed, updates their bitmaps. In performing the set-operations on bitmaps, the different `unsigned int` composing the same bitmap to be updated/accessed are updated/accessed in parallel by the different ℓ threads of the same warp. This allows a better exploitation of SIMT-parallelism and reduces thread divergence. To avoid race conditions in concurrent accesses, updates of bitmaps use atomic operations, that, operating on shared memory, involve little loss in performance.

Let us observe that since each block is scheduled on a different streaming multiprocessor (SM) of the GPU, its execution is independent from those of other blocks. The GPU scheduler is enabled to assign problems/blocks to different SMs, balancing work load and maximizing GPU usage.

When a fixpoint is reached the threads of a block perform a satisfiability check (line 7). If the CSP at hand, say b, turns out to be unsatisfiable then it is removed. If b is in solved form (each variable domain is a singleton) the solution is stored. Otherwise, b is still active and a decision step has to be performed in order to "shrink" a variable domain. At this point a variable X is heuristically chosen, for instance by identifying the most constrained one (alternative heuristics are possible).

- If X is a variable of sort `int` with domain D_X, then the domain is partitioned in two sets and two problems are put in the pool of active problems in place of b. This opens two branches in the visit of the solution space. There may be different ways in which D_X is partitioned. For instance, if $D_X = \{x_1, x_2, \ldots, x_n\}$ ($x_1 < x_2 < \cdots < x_n$) the two sub-domains $D'_X = \{x_1\}$ and $D''_X = \{x_2, \ldots, x_n\}$ can be considered. Alternatively, D_X could be split in two disjoint sub-domains of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.
- If X is a variable of sort `set` with domain D_X where $\top_X \setminus \bot_X = \{x_1, x_2, \ldots, x_n\}$ then two new problems will replace b. These new problems differ on the domain of X, namely, the two domains will be such that $D'_X = \{s \in D_X : x_1 \in s\}$ and $D''_X = \{s \in D_X : x_1 \notin s\}$.

Plainly, in both cases, alternative heuristics are possible, even involving partitioning in more than two sub-problems. Moreover, a choice must be made when a decision can be made considering both an integer variable and a set variable. In the current implementation, we always split in two and give priority to integer variables.

Once the new problems are generated (working in shared memory) the block stores them back in global memory, so that other blocks can process them.

Notice that, the way in which a problem b is replaced in *CSP* by two (or more) new problems corresponds to performing *domain cloning*. There is not an explicit management of a stack of choice points. Moreover there is no imposed order on the active problems in *CSPs*. Hence, any strategy can be adopted in selecting *bs* and in assigning blocks to active problems in *bs* (cf., lines 4–5 in Algorithm 1). This, in combination with the selection of the number of blocks that are launched at each iteration of the loop (line 5), permits to explore in parallel different multiple paths in the solution space implementing different search strategy. For example, on the one hand, always launching a single block operating on the last generated problem implements a depth-first visit. On the other hand, launching $|CSPs|$ blocks implements breadth-first search. Clearly, all intermediate strategies are possible.

Threads per block	Blocks in propagation	Time to solution	Speedup
32	1	29.24	1
32	2	14.89	1.96
32	4	8.09	3.61
32	8	3.80	7.70
32	16	1.95	15.02
32	32	1.04	28.23
32	64	0.62	47.05
32	128	0.39	75.40
32	256	0.28	103.54
32	512	0.29	101.36
32	1024	0.27	106.74
64	1	27.57	1
64	2	14.10	1.96
64	4	7.07	3.90
64	8	3.61	7.65
64	16	1.85	14.90
64	32	1.00	27.56
64	64	0.59	46.78
64	128	0.40	69.75
64	256	0.37	73.98
64	512	0.32	85.32
64	1024	0.28	97.60
128	1	26.90	1
128	2	13.71	1.96
128	4	6.91	3.89
128	8	3.50	7.68
128	16	1.82	14.76
128	32	0.95	28.45
128	64	0.57	47.55
128	128	0.54	49.49
128	256	0.46	58.03
128	512	0.41	65.01
128	1024	0.38	70.44
256	1	27.13	1
256	2	13.80	1.97
256	4	6.91	3.92
256	8	3.52	7.72
256	16	1.78	15.21
256	32	0.99	27.50
256	64	0.97	28.05
256	128	0.81	33.47
256	256	0.70	38.99
256	512	0.65	41.60
256	1024	0.64	42.16

Table 1: Performance of the solver for different configuration parameters for the instance $\text{Chain}_{(8,9)}$ of the CSP described in Figure 4

5.3. The solver at work

The GPU-based solver described so far is a prototype still under development. Only simple basic heuristics have been implemented in the decision step, in domain partitioning, and in the selection of active problems to be processed. Much work has to be done in fully exploiting computing capabilities supported by modern GPUs, such as Volta's new independent thread scheduling, L2 cache management, warp-level communication, cooperative groups, etc. Nevertheless, the current implementation exhibits promising performance and scalability properties. As a witness of this claim, we report here the outcome of just

Threads per block	Blocks	$\mathrm{Comb}_{(5,3,6)}$ Solving time	Problems per second	Speedup	$\mathrm{Comb}_{(6,2,5)}$ Solving time	Problems per second	Speedup
32	1	99.90	13437	1.00	96.04	13649	1.00
32	2	54.13	24791	1.84	51.20	25596	1.88
32	4	29.83	44961	3.35	27.20	48158	3.53
32	8	15.64	85648	6.37	14.47	90425	6.63
32	16	8.55	156487	11.65	7.92	164965	12.09
32	32	5.41	246666	18.36	4.73	275110	20.16
32	64	3.26	407085	30.30	2.86	453147	33.20
32	128	2.42	546783	40.69	2.19	590055	43.23
32	256	2.03	650867	48.44	1.87	688423	50.44
32	512	1.74	756059	56.27	1.67	771193	56.50
32	1024	1.52	865939	64.44	2.07	622775	45.63
64	1	89.29	15034	1.00	83.04	15786	1.00
64	2	48.13	27883	1.85	43.68	30004	1.90
64	4	25.67	52241	3.47	22.69	57717	3.66
64	8	14.02	95555	6.36	11.61	112610	7.13
64	16	7.42	180307	11.99	6.24	209142	13.25
64	32	4.28	311362	20.71	3.83	339300	21.49
64	64	2.95	449484	29.90	2.72	477109	30.22
64	128	2.37	560089	37.25	2.10	616241	39.04
64	256	1.96	673715	44.81	1.75	737537	46.72
64	512	1.72	767465	51.05	1.63	791213	50.12
64	1024	1.52	865702	57.58	1.41	910306	57.67
128	1	83.70	16037	1.00	77.38	16942	1.00
128	2	44.55	30123	1.88	40.71	32187	1.90
128	4	23.75	56469	3.52	21.15	61916	3.65
128	8	12.43	107742	6.72	10.80	121075	7.15
128	16	6.86	194859	12.15	5.82	224044	13.22
128	32	4.05	328799	20.50	3.64	357031	21.07
128	64	2.86	464232	28.95	2.56	506638	29.90
128	128	2.91	456052	28.44	2.05	631459	37.27
128	256	1.88	703497	43.87	1.77	729807	43.08
128	512	1.69	781315	48.72	1.58	813127	47.99
128	1024	1.46	900412	56.15	1.42	900697	53.16
256	1	76.42	17563	1.00	71.33	18378	1.00
256	2	40.77	32911	1.87	37.32	35111	1.91
256	4	21.41	62629	3.57	19.20	68200	3.71
256	8	11.07	120935	6.89	10.00	130737	7.11
256	16	6.13	217853	12.40	5.52	236077	12.85
256	32	3.75	355279	20.23	3.53	367977	20.02
256	64	2.69	493805	28.12	2.56	506269	27.55
256	128	2.21	598025	34.05	2.01	643814	35.03
256	256	1.85	712850	40.59	1.70	757656	41.23
256	512	1.63	807943	46.00	1.68	765378	41.65
256	1024	1.48	886061	50.45	1.44	889006	48.37

Table 2: Performance of the solver for different configuration parameters for two instances of the CSP $\mathrm{Comb}_{(m,t,n)}$ described in Figure 5 We report the time spent to find all solutions for $\mathrm{Comb}_{(5,3,6)}$ and to detect unsatisfiability of $\mathrm{Comb}_{(6,2,5)}$.

some sets of experiments involving significant example. The first CSP we used is formulated as in Figure 4, where `n` and `m` are integer parameters. (We denote instances of this problem by $\mathrm{Chain}_{(n,m)}$.)

We run experiments selecting different values for `n` and `m` and varying both the number of threads in each block and the number of blocks launched by the solver. We used different GPU, obtaining comparable results. We report on the experiments run on a server running Ubuntu 20.04.2 equipped with an Nvidia GeForce GTX 1060 with 6GB of RAM, 10 SMs, 1280 cores, compute capability 6.1, CUDA Driver v. 11.2, GPU frequency 1.5 GHz.

Table 1 shows the results obtained for `n=8` and `m=9`. The first column reports the number of threads composing each block. The second column reports the number of problems/blocks run in parallel (namely, the cardinality of the set bs, described earlier). The third column shows the time in seconds

```
%%% Variables: each x[i]  and each delta[j] is a set variable:
array [0..m-1] of var set of 1..n: x;
array [0..m-1] of var set of 1..n: delta;
%%% each reps[i] is an int variable:
array [0..m-1] of var 1..n: repr;

%%% Constraints:
constraint forall(i in 0..m-2)
        ( x[i] subset x[i+1]);
constraint forall(i in 0..m-2)
          ( delta[i] = x[i+1] diff x[i] /\ repr[i] in delta[i]);
```

Figure 4: Encoding of instances $\text{Chain}_{(m,n)}$

needed by the solver to solve the CSP. Finally, the fourth column reports the speedup obtained by using different number of blocks w.r.t. the run which uses a single block. This last set of data shows the impact on performance of visiting multiple paths of the solution space, in parallel. The best improvement is 106x, obtained launching 1024 parallel blocks, for the case of 32 threads-per-block. On the other hand, rising the number of threads-per-block seems to have a negative impact on performance. The best performance is obtained using a number of threads close to the number of constraints of the CSP. In this case, choosing 32 threads-per-block represents the best configuration (notice that a block must include at least one warp, namely, 32 threads). This is because each constraint is processed by 1 thread (i.e., the parameter ℓ described earlier is set to 1 in these experiments) and running more threads than the number of constraints only introduces overhead in their management. The Gecode 6.3.0 solver of Minizinc with `input_order` and `indomain_min` search heuristics, running on a faster Windows 10 Desktop with 3.60 GHz i7 CPU finds the first solution to the instance of the table in 5.6 s.

Results in line with those obtained for the $\text{Chain}_{(m,n)}$ instances, have been obtained for other CSPs. As an example we report in Table 2 the performance of the solver for two instances of the CSP $\text{Comb}_{(m,t,n)}$ described in Figure 5. Given the integer values `m`, `t`, and `n`, solving this CSP consists in finding, if possible, `m` pairwise distinct subsets of $\{0, ..., n-1\}$ such that the intersection of any pair of them is a set of exactly `t` elements.

Table 2 also reports the number of intermediate problems (generated by decision steps, see Section 5.2) processed per second, depending on different configuration parameters of the solver. The Gecode 6.3.0 solver of Minizinc with `input_order` and `indomain_min` search heuristics, running on a Windows 10 Desktop with 3.60 GHz i7 CPU computes the two instances of the table in 1.73 s and 0.55 s, respectively.

```
%%% Variables:
int: m=5;   int: t=3;  int: n=5;
array [0..m-1] of var set of 0..n-1: sets;
array [0..m-1,0..m-1,0..t-1] of var 0..n-1: witn;

%%% Constraints:
constraint
  forall(i,j in 0..m-1 where i < j) (sets[i] != sets[j]);
constraint
  forall(i,j in 0..m-1 where i < j)
       (sets[i] intersect sets[j] = {witn[i,j,k]|k in 1..t});
constraint
  forall(i,j in 0..m-1, k in 0..t-2) (witn[i,j,k] < witn[i,j,k+1]);
constraint
  forall(i,j in 0..m-1 where i >= j, k in 0..t-1) (witn[i,j,k]=k);
```

Figure 5: Encoding of instances $\mathrm{Comb}_{(m,t,n)}$

6. Conclusions

In this paper we have presented a first attempt of exploiting the parallelism offered by the widespread hardware available inside most of our desktop and laptop computers, namely GPUs, for the research areas introduced by Eugenio G. Omodeo et al. of computable set theory and programming with sets. Precisely, we have revised the set constraints procedure originally proposed in [25] and developed an implementation in GPU using the CUDA programming paradigm. Results are interesting and experimentally proved to be scalable. As future work we would like, on the one side to include more set operations and optimize the encoding, on the other side to embed the proposal within a complete solver for the Minizinc modeling language.

References

[1] Serge Abiteboul and Stéphane Grumbach, *A rule-based language with functions and sets*, ACM Trans. Database Syst. **16** (1991), no. 1, 1–30.
[2] Jean-Raymond Abrial, Stephen A Schuman, and Bertrand Meyer, *A specification language*, On the Construction of Programs (A. M. Macnaghten and R. M. McKeag, eds.), Oxford University Press, 1980.
[3] Catriel Beeri, Shamim A. Naqvi, Oded Shmueli, and Shalom Tsur, *Set constructors in a logic database language*, J. Log. Program. **10** (1991), no. 3&4, 181–232.
[4] Federico Campeotto, Alessandro Dal Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli, *Exploring the use of GPUs in*

constraint solving, Practical Aspects of Declarative Languages - 16th International Symposium, PADL 2014, San Diego, CA, USA, January 20-21, 2014. Proceedings (Matthew Flatt and Hai-Feng Guo, eds.), Lecture Notes in Computer Science, vol. 8324, Springer, 2014, pp. 152–167.

[5] Domenico Cantone, Alfredo Ferro, and Eugenio G. Omodeo, *Computable set theory: Volume 1*, Oxford University Press, 1990.

[6] Maximiliano Cristiá and Gianfranco Rossi, *Automated Proof of Bell-LaPadula Security Properties*, Journal of Automated Reasoning **65** (2021), no. 4, 463–478.

[7] Maximiliano Cristiá and Gianfranco Rossi, {*log*}*: Set formulas as programs*, CoRR **abs/2104.08130** (2021).

[8] Maximiliano Cristiá, Gianfranco Rossi, and Claudia S. Frydman, *Adding partial functions to constraint logic programming with sets*, Theory Pract. Log. Program. **15** (2015), no. 4-5, 651–665.

[9] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli, *CUD@SAT: SAT solving on GPUs*, J. Exp. Theor. Artif. Intell. **27** (2015), no. 3, 293–316.

[10] Agostino Dovier, Andrea Formisano, and Enrico Pontelli, *Parallel answer set programming*, Handbook of Parallel Constraint Reasoning (Youssef Hamadi and Lakhdar Sais, eds.), Springer, 2018, pp. 237–282.

[11] Agostino Dovier, Andrea Formisano, Enrico Pontelli, and Flavio Vella, *A GPU implementation of the ASP computation*, Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings (Marco Gavanelli and John H. Reppy, eds.), Lecture Notes in Computer Science, vol. 9585, Springer, 2016, pp. 30–47.

[12] Agostino Dovier, Andrea Formisano, and Flavio Vella, *GPU-Based Parallelism for ASP-Solving*, Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers (Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel, eds.), Lecture Notes in Computer Science, vol. 12057, Springer, 2019, pp. 3–23.

[13] Agostino Dovier, Eugenio G. Omodeo, and Alberto Policriti, *Solvable set/hyperset contexts: II. A goal-driven unification algorithm for the blended case*, Appl. Algebra Eng. Commun. Comput. **9** (1999), no. 4, 293–332.

[14] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi, {*log*}*: A logic programming language with finite sets*, Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991 (Koichi Furukawa, ed.), MIT Press, 1991, pp. 111–124.

[15] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi, *A language for programming in logic with finite sets*, J. Log. Program. **28** (1996), no. 1, 1–44.

[16] Agostino Dovier and Carla Piazza, *The subgraph bisimulation problem*, IEEE Trans. Knowl. Data Eng. **15** (2003), no. 4, 1055–1056.

[17] Agostino Dovier, Carla Piazza, and Enrico Pontelli, *Disunification in*

ACI1 Theories, Constraints **9** (2004), no. 1, 35–91.
[18] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi, *Sets and constraint logic programming*, ACM Trans. Program. Lang. Syst. **22** (2000), no. 5, 861–931.
[19] Agostino Dovier, Carla Piazza, and Gianfranco Rossi, *A uniform approach to constraint-solving for lists, multisets, compact lists, and sets*, ACM Trans. Comput. Log. **9** (2008), no. 3, 15:1–15:30.
[20] Agostino Dovier, Alberto Policriti, and Gianfranco Rossi, *A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms*, Fundam. Informaticae **36** (1998), no. 2-3, 201–234.
[21] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi, *Constructive negation and constraint logic programming with sets*, New Gener. Comput. **19** (2001), no. 3, 209–256.
[22] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi, *Intensional sets in CLP*, Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings (Catuscia Palamidessi, ed.), Lecture Notes in Computer Science, vol. 2916, Springer, 2003, pp. 284–299.
[23] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi, *Set Unification*, Theory and Practice of Logic Programming **6** (2006), no. 6, 645–701.
[24] Alfredo Ferro, Eugenio G. Omodeo, and Jacob T. Schwartz, *Decision procedures for some fragments of set theory*, 5th Conference on Automated Deduction, Les Arcs, France, July 8-11, 1980, Proceedings (Wolfgang Bibel and Robert A. Kowalski, eds.), Lecture Notes in Computer Science, vol. 87, Springer, 1980, pp. 88–96.
[25] Carmen Gervet, *Interval propagation to reason about sets: Definition and implementation of a practical language*, Constraints An Int. J. **1** (1997), no. 3, 191–244.
[26] Carmen Gervet, *Constraints over structured domains*, Handbook of Constraint Programming (Francesca Rossi, Peter van Beek, and Toby Walsh, eds.), Foundations of Artificial Intelligence, vol. 2, Elsevier, 2006, pp. 605–638.
[27] Bharat Jayaraman and David A. Plaisted, *Programming with equations, subsets, and relations*, Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio, USA, October 16-20, 1989. 2 Volumes (Ewing L. Lusk and Ross A. Overbeek, eds.), MIT Press, 1989, pp. 1051–1068.
[28] Gabriel M. Kuper, *Logic programming with sets*, Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA (Moshe Y. Vardi, ed.), ACM, 1987, pp. 11–20.
[29] Victor Marek and Miroslav Truszczynski, *Stable models and an alternative logic programming paradigm*, The Logic programming paradigm: a 25-year perspective (K. Apt, ed.), Springer Verlag, 1999, pp. 169–181.
[30] Illka Niemela, *Logic programs with stable model semantics as a constraint programming paradigm*, Annals of Mathematics and Artificial Intelligence **25** (1999), no. 3-4, 241–273.
[31] Gianfranco Rossi, Elio Panegai, and Elisabetta Poleo, *Jsetl: a java library for supporting declarative programming in java*, Softw. Pract. Exp. **37**

(2007), no. 2, 115–149.

[32] Andrew John Sadler and Carmen Gervet, *Hybrid Set Domains to Strengthen Constraint Propagation and Reduce Symmetries*, Principles and Practice of Constraint Programming, 2004, pp. 604–618.

[33] Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edith Schonberg, *Programming with sets - an introduction to SETL*, Texts and Monographs in Computer Science, Springer, 1986.

[34] Ron Sigal, *Desiderata for Logic Programming with Sets*, Fourth National Conference on Logic Programming (GULP), 1989, pp. 127–141.

[35] John Michael Spivey, *The Z Notation: A reference manual*, International Series in Computer Science. Prentice Hall, 1992.

[36] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer, *The minizinc challenge 2008-2013*, AI Mag. **35** (2014), no. 2, 55–60.

[37] Peter J. Stuckey, Kim Marriot, and Guido Tack, *The MiniZinc Handbook*, Tech. Report `https://www.minizinc.org/doc-2.5.5/en/index.html`, Monash University, 2021.

Authors' addresses:

Agostino Dovier, Andrea Formisano
DMIF, University of Udine, Udine, Italy
Enrico Pontelli, Fabio Tardivo
Dept CS, NMSU, Las Cruces, NM, USA

Received JUNE 30, 2021
Revised Month dd, yyyy
Accepted Month dd, yyyy