TPLP: Page 1–19.

doi:10.1017/S1471068425100197

GPU Accelerated Compact-Table Propagation

ENRICO SANTI, AGOSTINO DOVIER AND ANDREA FORMISANO
DMIF, University of Udine, Udine, Italy
(*e-mails:* santi.enrico@spes.uniud.it, agostino.dovier@uniud.it, andrea.formisano@uniud.it)

FABIO TARDIVO
Dept CS, New Mexico State University, Las Cruces, NM, USA
(*e-mail:* ftardivo@nmsu.edu)

submitted 24 July 2025; revised 24 July 2025; accepted 27 July 2025

Abstract

Constraint Programming developed within Logic Programming in the Eighties; nowadays all Prolog systems encompass modules capable of handling constraint programming on finite domains demanding their solution to a constraint solver. This work focuses on a specific form of constraint, the so-called table constraint, used to specify conditions on the values of variables as an enumeration of alternative options. Since every condition on a set of finite domain variables can be ultimately expressed as a finite set of cases, Table can, in principle, simulate any other constraint. These characteristics make Table one of the most studied constraints ever, leading to a series of increasingly efficient propagation algorithms. Despite this, it is not uncommon to encounter real-world problems with hundreds or thousands of valid cases that are simply too many to be handled effectively with standard CPU-based approaches. In this paper, we deal with the Compact-Table (CT) algorithm, the state-of-the-art propagation algorithms for Table. We describe how CT can be enhanced by exploiting the massive computational power offered by modern Graphics Processing Units (GPUs) to handle large Table constraints. In particular, we report on the design and implementation of GPU-accelerated CT, on its integration into an existing constraint solver, and on an experimental validation performed on a significant set of instances.

Keywords: constraint logic programming, table constraint, GPU parallelism

1 Introduction

Table constraints play a crucial role in combinatorial problem solving, as they provide an explicit and efficient way to represent all allowed combinations of values for the variables they involve (Demeulenaere *et al.* 2016). Their flexibility makes them widely applicable in different domains such as scheduling, configuration, and in every context where constraints on variables must be explicitly enumerated. Additionally, planning problems often found in constraint (logic) programming, can be conveniently formulated as Constraint Satisfaction Problems (CSP) over finite domains (Dovier *et al.* 2010; 2011; 2013) where state transitions or action scheme can be expressed as table

constraints. The relevance of such a constraint has led to the development of several efficient algorithms (such as STR2, STR3, MDD4R, and Compact-Table) to enforce the generalized arc consistency (GAC) (Lecoutre 2011; Perez and Régin 2014; Lecoutre *et al.* 2015).

Following recent works in which parts of constraint solvering have been delegated to Graphics Processing Units (GPUs) (Tardivo *et al.* 2024a; 2023; 2024b; Travasci *et al.* 2025), with this paper we are pursuing this project going toward a more exhaustive parallel implementation of constraint logic programming (Dovier *et al.* 2022).

In this paper, we recall an efficient GAC algorithm for the table constraint, Compact-Table (CT) and we describe different implementations of GPU-accelerated propagators based on CT inserting them in MiniCP, a simple and extendable constraint solver (Michel *et al.* 2021).

The paper is organized as follows: Section 2 recalls some background on the Table constraint, the CT propagator and the MiniCP constraint solver. Section 3 describes an implementation of the serial propagator. Based on such an implementation, in Section 4 different GPU-accelerated propagators are described. Section 5 reports on the experiments we run to validate the parallel implementations and discusses the obtained results. In Section 6 we draw our conclusions.

Source code, tests instances, and results are available from https://clp.dimi.uniud.it/sw.

2 CSPs and the table constraint

A CSP is a triple $P = \langle X, D, C \rangle$, where X is a finite set of variables, D is set of (finite) domains for the variables, and C is the set of *constraints* over X. For each $x \in X$ its domain is denoted by $\mathsf{dom}(x)$. A constraint $c \in C$ involves a set of variables (its scope) $\mathsf{var}(c) = \{x_1, \ldots, x_n\} \subseteq X$ and induces a relation $rel(c) \subseteq \mathsf{dom}(x_1) \times \cdots \times \mathsf{dom}(x_n)$.

A *solution* of P is an assignment σ of variables to values in their domains that satisfies all the constraints in C, namely such that $\sigma(X) \in rel(c)$ for all $c \in C$. A constraint c such that $\mathsf{var}(c) = \{x_1, \ldots, x_n\}$ is GAC if and only if for all $x_i \in \mathsf{var}(c)$ and all $a \in \mathsf{dom}(x_i)$ there is a tuple $(a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n) \in rel(c)$. In such case we say the value a is supported. Conversely, if a is not supported, a cannot belong to any solution and therefore it can be eliminated from $\mathsf{dom}(x_i)$. Constraint propagation algorithms iteratively removes unsupported values until the GAC is enforced (or unsatisfiability, namely not existence of a solution, is detected).

Several kinds of constraints on finite domain variables have been introduced and studied (Rossi *et al.* 2006). Our focus in this paper is on the table (a.k.a., extensional) constraint. The table constraint can theoretically simulate all the other constraints since it explicitly lists the tuples of values of the relation (Verhaeghe *et al.* 2017a). Table 1a shows an example of a table constraint involving three variables x_1, x_2, x_3, all having $\{1, 2, 3, 4\}$ as domain. The constraint specifies all the admissible tuples $\tau_1, \ldots, \tau_5$ of values.

Compact-Table (CT) is an efficient algorithm to enforce GAC on table constraints (Demeulenaere *et al.* 2016). While its base version allows to express only positive and

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Table 1. *A table constraint c with 5 tuples on the variables x_1, x_2, x_3 with domain $\{1, 2, 3, 4\}$ (a), the corresponding static supports matrix (b), and a possible value of currTable (c)*

	x_1	x_2	x_3
τ_1	3	1	1
τ_2	1	2	3
τ_3	2	3	3
τ_4	1	4	1
τ_5	3	4	3

(a)

	τ_1	τ_2	τ_3	τ_4	τ_5
$[x_1, 1]$	0	1	0	1	0
$[x_1, 2]$	0	0	1	0	0
$[x_1, 3]$	1	0	0	0	1
$[x_1, 4]$	0	0	0	0	0
$[x_2, 1]$	1	0	0	0	0
$[x_2, 2]$	0	1	0	0	0
...	...	...	...	...	...
$[x_3, 4]$	0	0	0	0	0

(b)

	τ_1	τ_2	τ_3	τ_4	τ_5
	0	1	0	1	0

(c)

Algorithm 1: enforceGAC()

input : c a table constraint, with $\mathsf{var}(c) = \{x_1, \ldots, x_n\}$
1 $s^{val} \leftarrow \{i \mid |\mathsf{dom}(x_i)| \neq lastSizes[i]\}$;
2 **for** $i \in s^{val}$ **do** $lastSizes[i] \leftarrow |dom(x_i)|$;
3 $s^{sup} \leftarrow \{i \mid |\mathsf{dom}(x_i)| > 1\}$;
4 *updateTable*();
5 **if** $currTable = \vec{0}$ **then** *fail*() ;
6 *filterDomains*();

exhaustive (i.e., non-short) tables,[1] extensions have been introduced by Verhaeghe *et al.* (Verhaeghe *et al.* 2017a; Verhaeghe *et al.* 2017b). In what follows we describe serial and parallel implementations of CT. Although these implementations can also be used for short tables, in this paper we limit ourselves to the case of positive and exhaustive tables.

Consider a table constraint c expressed by a set of t rows (tuples) and $n = |\mathsf{var}(c)|$ columns. Let $\tau_1, \ldots, \tau_t$ be the tuples of c, and $\mathsf{var}(c) = \{x_1, \ldots, x_n\}$. In CT, c is represented by two data structures, a Boolean matrix and a Boolean array, both stored as sequences of bits:

supports: A static Boolean matrix of size $(|\mathsf{dom}(x_1)| + \cdots + |\mathsf{dom}(x_n)|) \times t$, for each variable x_i and each value $v \in \mathsf{dom}(x_i)$, the cell $([v, a], j)$ is set to 1 iff $\tau_j[i] = v$.
currTable: A Boolean array of size t indicating, for each tuple τ_j, if $\bigwedge_{i=1}^{n} \tau_j[i] \in \mathsf{dom}(x_i)$.

A tuple τ_i is *valid* if and only if $\forall x_j \in var(c)$, $\tau_i[j] \in dom(x_j)$. During the search process the domains of the variables are reduced. The bitset *currTable* keeps track of domain reductions by setting the i-th bit to 1 if the i-th tuple is valid, or to 0 otherwise (cf., Table 1).

[1] A Table c is *positive* if its tuples list the allowed values for $var(c)$. A table explicitly listing the disallowed tuples is said *negative*. A table is *short* if more than a one domain value can be specified in each cell.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Algorithm 2: updateTable()

```
   input : supports, currTable, dom, Δ, s^val
1  for i ∈ s^val do                    // For all variables with domain reduced by the last step
2      mask ← 0⃗;                                                    // Reset a mask of t bit
3      if |Δ_{x_i}| < |dom(x_i)| then                  // Reset-based update is advantageous
4          for a ∈ Δ_{x_i} do                        // Collect indexes of tuples to be removed
5              mask ← mask | supports[x_i, a];
6          mask ← ∼mask;                                                // Complement the mask
7      else                                          // Incremental update is preferable
8          for a ∈ dom(x_i) do                      // Collect indexes of supported tuples
9              mask ← mask | supports[x_i, a];
10     currTable ← currTable & mask;                 // Update currTable applying the mask
11     if currTable = 0⃗ then break;
```

The iterative constraint propagation algorithm uses few additional data structures:

- s^{val}: A list of the indexes of variables with domain reduced by the last iteration.
- s^{sup}: A list of the indexes of the variables such that $|\mathsf{dom}(x)| > 1$.
- Δ: for each $x \in \mathsf{var}(c)$, Δ_x is the set of values removed from $\mathsf{dom}(x)$ in the last iteration.
- *lastSizes*: storing the current sizes of the domains: $lastSizes = [|\mathsf{dom}(x_1)|, \ldots, |\mathsf{dom}(x_n)|]$.

For a given table constraint c, Algorithm 1 enforces GAC first by initializing the data structures described earlier (lines 1–3), and then by calling Algorithms 2 and 3. Assuming that for each variable x_i the values in Δ_{x_i} have been removed from $\mathsf{dom}(x_i)$, Algorithm 2 updates *currTable* by collecting the bits corresponding to all (still) valid tuples. To minimize the number of operations, a choice is made between a *reset-based update* and an *incremental update*. The number of bitset operations required by the two options depends on the size of Δ_{x_i} (see the paper by Demeulenaere *et al.* 2016) for a detailed description). The choice is made in line 3: If $|\Delta_{x_i}|$ is smaller than the current domain size of x_i, *mask* is used (lines 4–6) to collect bits corresponding to tuples that have become unsupported (because of the removed elements in Δ_{x_i}). Alternatively, in lines 8–9, the bits that will compose the updated *currTable* are gathered by inspecting $\mathsf{dom}(x_i)$. After the update of *currTable* Algorithm 3 filters the current domains using the outcome of Algorithm 2.

Enhancements, not discussed here, such as the usage of residues (Demeulenaere *et al.* 2016), can be applied to improve efficiency of the filtering step.

2.1 The MiniCP solver and GPU

Since the late Eighties several efficient constraint solvers have been developed for Constraint (Logic) Programming. For the scope of this paper we have decided to use

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Algorithm 3: filterDomains()

```
   input : supports, currTable, dom, lastSize
1  for i ∈ s^sup do                                  // For all variables with non-singleton domain
2      for a ∈ dom(x_i) do                           // Check the support for a in a tuple of the table
3          if currTable & supports[x_i, a] = 0⃗ then   // If not supported remove the value
4              dom(x_i) ← dom(x_i) \ {a}
5      lastSize[i] ← |dom(x_i)| ;                    // Update the sizes
```

MiniCP (Michel *et al.* 2021), a relatively new and open-source solver devised to ease development of extensions. In particular, our extensions build on MiniCPP (Gentzel *et al.* 2020), a C++ implementation of MiniCP that can easily include CUDA C++ code (Tardivo *et al.* 2024a; 2023; 2024b). New global constraints can be integrated in MiniCP by creating a class under `/fzn-minicpp/global_constraints/` which extends the `Constraint` class. Such a class, which will serve as the propagators for the new global constraints, must include, alongside a constructor, a procedure, *propagate()*. The *propagate()* method implements the propagation algorithm for the new constraint. To provide a simple mechanism to use the GPU-accelerated propagators, we rely on the MiniZinc annotation introduced by (Tardivo *et al.* 2023). Specifically, a table constraint annotated with `::gpu` (i.e., a constraint declared as: `constraint table(...)::gpu;` in the input model) is propagated using the GPU-accelerated implementation in place of the standard one.

3 Serial implementation of CT propagation

We briefly outline a serial implementation of the Compact-Table propagator described in Section 2, conceived to be integrated in MiniCP as a new global constraint. We will refer to this implementation simply as CT. As mentioned, while enforcing GAC, the (serial) Algorithm 2 gathers the set of valid tuples by performing either a *reset-based* update or an *incremental update*. MiniCP provides no built-in functions to keep track of the domain changes, so, to avoid the computation of Δ we opted to use only the reset-based strategy, in all our implementations. Indeed, the different update strategies were introduced to minimize the number of computations needed to update the *_currentTable*, at the cost of calculating and maintaining information about the previous domains. Our objective is to implement a GPU propagator, calculating and copying Δ from host to device can be less effective than using a simpler strategy, offloading the additional computation steps to the device.

Data structures: A specific class *Table* stores the table instance at hand and defines the methods for enforcing GAC. MiniCP itself manages all the data related to the variables. The main data structures used closely match those mentioned in Section 2:

- *_currTable*: A bitset implementing the *currTable* described in Section 2.
- *_currTableSize*: An integer storing the number of words of *_currTable*.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

- *_supports*: The compact support table, namely a Boolean matrix stored as an array of 32-bit words. Each word of *_supports* compactly stores 32 Boolean values.
- *_supportSize*: An integer storing the number of rows of the support matrix, $\_supportSize = \sum_{x \in \mathsf{var}(c)} |\mathsf{dom}(x)|$.
- *_s_val*, *_s_sup*: Lists of integers (variable indexes), corresponding to s^{val} and s^{sup}.
- *_supportJmp*: An array of integers. For each variable x_i, *_supportJmp*$[i]$ is the index of the first cell in *_supports* where a value for x_i occurs. For example, for the constraint in Table 1 we have $\_supportJmp[0] = 0$, $\_supportJmp[1] = 3$, and $\_supportJmp[2] = 6$. Note that *_supportJmp* is a constant vector and it is used to speed up the accesses to *_supports*.
- *_variablesOffsets*: A vector of integers of length n. It contains the first value in the initial domain of each variable. It is used in accessing the rows of *_supports*. As an example let x_i be a variable with a domain $dom(x_i) = [90, 120]$, then $\_variablesOffsets[i] = 90$.

Let us remark that the data structure *lastSizes* used in Algorithm 1 is not used since we rely on the MiniCP built-in method *changed*() to populate *_s_val*. This method returns *true* when invoked on a variable whose domain changed during the previous iteration. Moreover, since we adopt the reset-based update approach, we do not use the structure Δ.

Procedures: It suffices specializing the class `Constraint` to extend MiniCP. We implemented a new class *Table* whose constructor allocates and initializes all the data structures described earlier. It also checks if each variable in the constraint is supported by at least one tuple, otherwise unsatisfiability is reported. The method *propagate*() corresponds to three methods *enforceGAC*(), *updateTable*(), and *filterDomains*() that, except for the indexed access to *supports*, based on *_supportJmp*, are one-to-one implementations of Algorithms 1–3.

4 GPU-accelerated CT propagators

The main engine of MiniCP performs the Constraint-Based Search in the usual manner, essentially by alternating search and propagation phases, possibly involving backtracking steps. Intuitively, whenever the GAC enforcing procedure is triggered, the corresponding method is responsible to start the propagation and report the results to the main engine. The possibility of extending MiniCP with a new propagator is transparent to the way it is implemented. Indeed, provided that the propagator adheres to the interface with the solver, there is no requirement on how its computation is performed. So, the same mechanism used to add a serial propagator can be used to plug in a GPU-empowered method. Clearly, this method is also responsible to transparently offload the computation on the GPU and to suitably manage the related data transfers.

Recall that each computation on the device is described as a collection of concurrent threads, each executing the same function. Such a function is called *kernel*, in CUDA terminology. Threads are hierarchically organized in equally sized blocks that are in turn structured in a grid (see Figures 1, 2, and 3).

We implemented parallel counterparts of *updateTable*() and *filterDomains*(), composing them into three variants of the propagator:

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

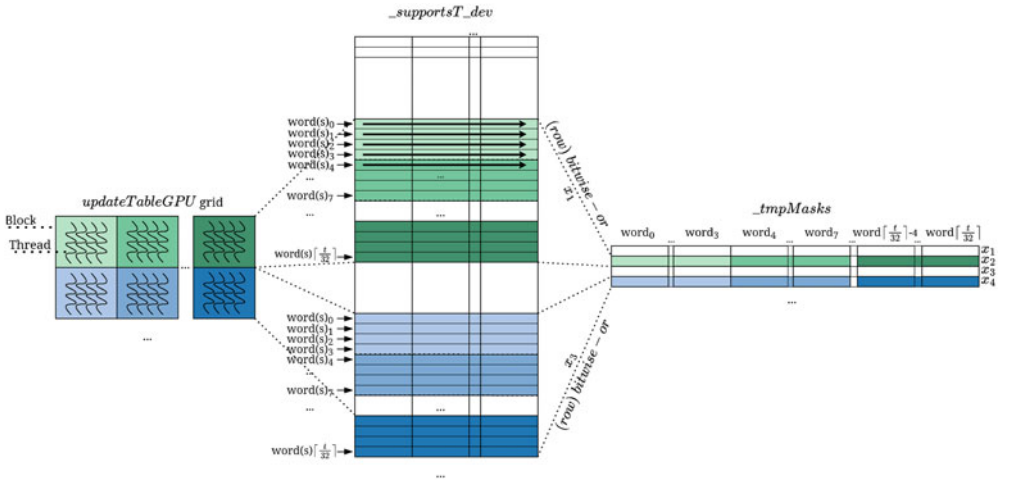

Fig 1. Overview of the parallel reduction done on $\_supportsT\_dev$, where $\_s\_val = \{x_1, x_3, \ldots\}$. Different colors indicate which words are accessed by the threads in the block. Arrows depict the direction of the parallel reductions and $\lceil \frac{t}{32} \rceil$ is a shorthand for $\_currTableSize$ (see Section 2).

- CT^{u}_{CU}: only *updateTable*() is computed on the device.
- CT^{f}_{CU}: only *filterDomains*() is computed on the device.
- CT^{uf}_{CU}: both update and filtering are computed on the device.

All implementations extend the class $Table$ described earlier. The constructors of the derived classes allocate and initialize the data structures on the GPU. When propagation is triggered, relevant data is transferred to the device, the computation is run, and the results are retrieved.

4.1 The propagator CT^{u}_{CU}

In this propagator the execution flow of *enforceGAC*(), closely matches its serial counterpart. First, the contents of $\_currTable$ and $\_s\_val$ are copied to the device, together with the variables' domains. The kernels responsible for updating the current table, that is *updateTableGPU*() and *reduce*(), are subsequently launched, see Figure 4. The outcome of this step is the same mask computed by the original *updateTable*() procedure, but it is written in GPU's global memory. It is then copied back to the host and combined bitwise with $\_currTable$. Finally, the filtering procedure on the host is invoked.

The propagator CT^{u}_{CU} uses some auxiliary structures that do not have a counterpart on the serial side. They are:

- $\_supportsT\_dev$: An array of integers storing the transposed support matrix. It is used by the kernel *updateTableGPU*() for updating the current table. The transpose is exploited to make threads of the same warp access adjacent memory words.
- $\_vars\_dev$: An array of $\lceil \_supportSize/32 \rceil$ integers. It stores the concatenation of the bitmaps representing the different variable domains. It is the counterpart of an host side variable ($\_vars\_host$) storing variable domains. The content of $\_vars\_host$ is copied to the device at each propagation.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

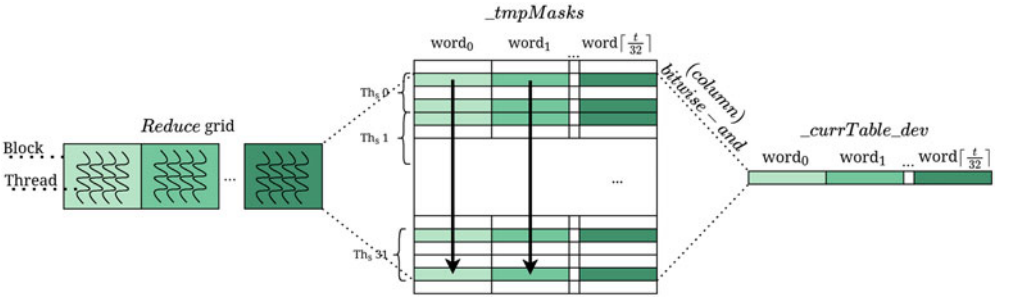

Fig 2. Overview of the parallel reduction done on *_tmpMasks* by *reduce*(). Different colors of the blocks indicate which words are accessed by the threads in the block. Arrows show the direction of the parallel reductions while $\lceil \frac{t}{32} \rceil$ is a shorthand for *_currTableSize* (see Section 2).

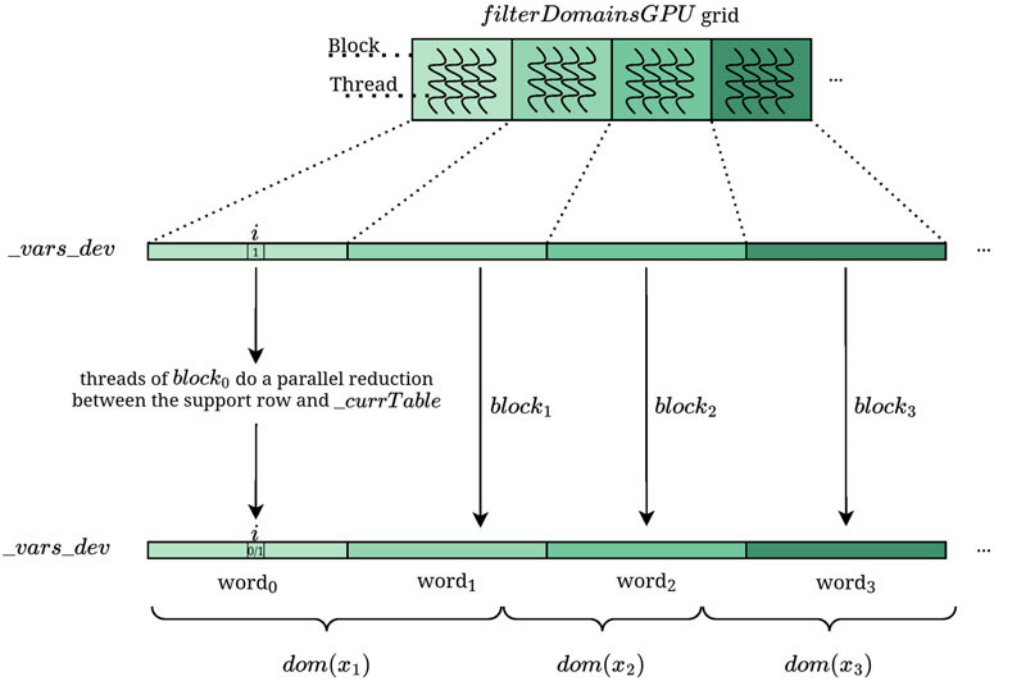

Fig 3. Overview of the data each block considers in the kernel *filterDomainsGPU*().

- *_tmpMasks*: An array of integers of size $\_currTableSize \times n$. It is a linearized representation of the support matrix, in which each row (of size *_currTableSize*) is associated to a variable. Namely, for each variable $x_i \in \_s\_val$, the row $\_tmpMasks[i]$ stores the *bitwise-OR* of the rows $\_supports[x_i, a]$ for all $a \in dom(x_i)$. Such a structure stores the result of the kernel *updateTableGPU*() and is subsequently processed by a *reduce*() kernel (see below).

As concerns the kernel *updateTableGPU*(), the grid launched has $\lceil \_currTableSize/4 \rceil \times |\_s\_val|$ blocks. That is, a block of 128 threads for every four (32-bit) words used by *_currTable* and for each changed variable is launched. Note that the division of work between blocks/threads (and thus between streaming

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

multiprocessors of the GPU) is dynamic and scales with the size of the problem (i.e. the number of rows in the table as well as the scope size).

Each row of the grid is related to a variable reduced during the last iteration, that is the i-th row of the grid is related to the i-th variable in *_s_val*, assume it to be x_j. Each block in the row is responsible for updating four (32-bit) words of the table mask accessing the *transposed* supports related to x_j. Each block of 128 threads can be seen as a small sub-grid of 32×4 threads, where each of the 32 rows considers a (different) slice of $dom(x_j)$. Each row of this sub-grid computes, via a *bitwise-OR*, the partial mask that needs to be added to *_currTable*. Such a mask, computed in shared memory is then copied in global memory in the i-th row of *_tmpMasks* (see Figure 1).

Once *updateTableGPU*() is terminated, the kernel *reduce*() is launched. It considers all and only the rows of *_tmpMasks* related to changed variables (i.e. those in *_s_val*) and performs a *bitwise-AND* operation along the columns. To this aim, a grid of *_currTableSize* blocks of 32 threads each is launched. Each block processes a different column of *_tmpMasks*. Similarly to *updateTableGPU*(), also for the *reduce*() kernel the division of work is dynamic and scales with the number of rows in the table.

Consider the i-th block of the *reduce* grid, each thread of such a block looks at a word (same column) in different rows of *_tmpMasks* performing a *bitwise-AND* operation. Once all the rows have been considered, all the threads of the same block perform a parallel reduction (applying a *bitwise-AND*) to produce a single 32-bit word per block (see Figure 2). After this kernel is executed, the obtained mask is copied back to the host.

4.2 The propagator CT^f_{CU}

In the case of the propagator CT^f_{CU}, the update of *_currTable* is carried out by the host while the filtering procedure is carried out on the device by means of the *filterDomainsGPU*() kernel. Such a kernel (see Figure 3) is responsible for providing the host, for each variable in the scope, the values to remove from their respective domains. The initialization and data structures used are the same as for CT^u_{CU}, with the exception of a new array *_vars_to_remove_host* on the host side. Such an array, of size $\lceil$*_supportSize*$/32\rceil$, stores the result of the *filterDomainsGPU*() kernel.

As concerns the *filterDomainsGPU*() kernel, the number of blocks launched is $\lceil$*_supportSize*$/32\rceil$, each consisting of 32 threads. The division of work is thus dynamic and depends on the domain size of the variables in the scope. Each block is responsible for processing a single 32-bit word of the variable domains stored in *_vars_dev*. All threads in the same block check whether the same value is still present in the domain of the variable x_j considered. If such is the case, a parallel reduction on the 32 threads is launched on the respective support row, computing a *bitwise-AND* with the current table. After the parallel reduction, the i-th bit of a shared word in the block is set to 1 if the value considered must be removed from the variable, or to 0 otherwise. The kernel thus computes an array of size $\lceil$*_supportSize*$/32\rceil$, in which the bit in position *_supportJmp*$[j] + i -$ *_variablesOffsets*$[j]$ is set to 1 if and only if the i-th value of the variable x_j must be removed from its domain. Each block is responsible for writing one word in *_vars_dev*. Such an array is then copied back to the host into *_vars_to_remove_host* and the values are removed from the domains of the respective variables.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

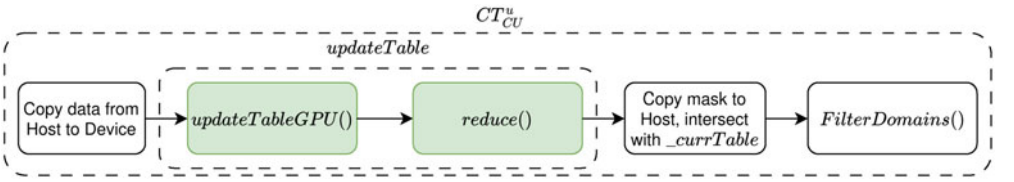

Fig 4. Overview of the execution flow of CT^u_{CU}. Kernels are depicted by green activities.

As an example consider a table with 1000 rows and suppose the scope has two variables, x_1 and x_2, respectively with $dom(x_1) = \{100, \ldots, 196\}$ and $dom(x_2) = \{1, \ldots, 100\}$. A total of 7 blocks, of 32 threads each, are launched. Consider now the second block (index 1), all 32 threads of such a block check the 32 bits responsible for keeping track of whether, for the variable x_1, the values in the set $\{132, \ldots, 163\} = \{\_variableOffsets[0] + 32, \ldots, \_variableOffsets[0] + 63\}$ are still present in $dom(x_1)$.

4.3 The propagator CT^{uf}_{CU}

This last implementation combines the kernels designed for CT^u_{CU} and CT^f_{CU}.

The kernel *filterDomainsGPU*() is called after *updateTableGPU*() and *reduce*(), so that both the table update and the domain filtering are carried out by the device. The data structures used are those introduced in both CT^u_{CU} and CT^f_{CU}.

5 Experiments

We report the computational behavior of the serial CT and the two parallel propagators CT^f_{CU} and CT^{uf}_{CU} and on a comparison with Gecode, the default solver of Minizinc.

CT^u_{CU} results are omitted both because this implementation sometimes underperforms relative to the serial counterpart, and to avoid shifting the focus away from the main results, and because of limited space. One reason why CT^u_{CU} on some instances does not keep up with the serial implementation is that the host function *updateTable*() is very efficient and the two kernels *updateTableGPU*() and *reduce*(), while being faster than it, cannot overcome alone the overhead introduced by frequently moving data and control to/from the GPU.

The choice of using MiniZinc and not use CLP(FD) was driven by the availability of a compiler from MiniZinc to FlatZinc and MiniCP supporting FlatZinc. For this reason we encoded our tests in MiniZinc. Such a language is widely adopted in the constraint programming community and used in the MiniZinc Challenges (Stuckey *et al.* 2014). For completeness, we recall that also compilers from CLP languages to FlatZinc have been developed (Cipriano *et al.* 2008).

5.1 The benchmark suite

We report here on the experimentation we conducted using three collections of instances. The instances of the first set involve a single table constraint and a single linear equation on *some* of the variables in the scope of the table constraint, as in the following example:

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Table 2. *Overview of table constraint features in the first test set*

Benchmark	Variables	Max Domain Size	Tuples
LIN_B	80–150	600	5000–10000
LIN_EB	100–200	800	5000–15000

```
var 1..10 : x1; var 1..10 : x2; var 1..10 : x3;  var 1..10 : x4;
var 10..20 : x5;
array [int,int] of 1..10: t = [|1,1,1,1 | 1,2,1,2| 2,3,2,3|...|];
constraint table([x1,x2,x3,x4], t) :: gpu;
constraint 2*x1+3*x2+4*x3=120;
solve satisfy;
```

Note that we admit variables in the linear constraint not occurring in the table constraint. We generated two batches of instances, called *LIN_B* and *LIN_EB* resp., differing in the number of variables and tuples (see Table 2). The (synthetic) instances of this benchmark are based on a simple encoding of the bounded knapsack problem. More specifically, in the bounded Knapsack problem there are k items and a number representing the capacity of the knapsack. Each item is associated with a weight, a profit, and a maximum number of times the object can be included in the knapsack. The goal is to select the number of instances of each item maximizing the profit which can fit the knapsack (Assi and Haraty 2018). This problem can be modeled using two constant arrays (*profit* and *weight*) and a variable array representing, for each object, the number of times it is put in the knapsack. A table describing all possible knapsack configurations (i.e., the variables in the scope are the elements of the variable array) is defined. Additionally, a constraint requiring the weights of the objects considered to fit the knapsack is added (this is the linear equation mentioned earlier).

The tables in the *LIN_B* and *LIN_EB* tests may not involve all variables declared in the instance and do not fix a specific solving strategy, so such tests may trigger different numbers of propagations when solved by different solvers (e,g., Gecode and CT^{uf}_{CU}). In order to have a fair comparison between our implementations and Gecode, we introduced a second set of instances where both the linear equation and the table constraint involve all variables occurring in the instance. Moreover, in each solver, the solving strategy has been fixed as follows:

```
solve :: int_search(
  [x0, x1, x2, ..., x_n],   % The variables in the scope
  input_order,              % Variable selection strategy
  indomain_max,             % Value selection strategy
  complete                  % Search completeness
) satisfy;
```

Finally, we experimented with a set of instances of the *Orienteering problem (OP)* or *Single-Agent Path Planning with Reward Maximization* (Bock and Sanità 2015). The OP seeks a path for an agent starting and ending at specified locations, imposing a bound D on the length of the path and maximizing the reward accumulated by visiting the various sites along the path, up to a threshold C. The problem can be formalized as a problem on

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

an undirected graph $G = \langle V, E \rangle$, with nodes $V = \{v_1, \ldots, v_k\}$ and edges E, such that, for each edge (v_i, v_j), $w_{i,j}$ denotes its length and for each node v_i, π_i is a profit value associated to v_i. Given such $G = \langle V, E \rangle$, a starting and an ending nodes $v_s, v_e \in V$, a length bound $D \in \mathbb{R}$, a set $\Pi = \{\pi_1, \ldots, \pi_k\}$ of profit values for nodes, and a profit bound $C \in \mathbb{R}$, an instance of OP asks to determine a path $P = \langle (v_{p_1}, v_{p_2}), (v_{p_2}, v_{p_3}), \ldots, (v_{p_k-1}, v_{p_k}) \rangle$ (with $v_{p_1} = v_s$ and $v_{p_k} = v_e$) such that $\sum_{(v_{p_i}, v_{p_j}) \in P} w_{p_i,p_j} \leq D$. Such a path must also maximize the profit collected by visiting the nodes, subject to the bound C, namely, $(\sum_{(v_{p_i}, v_{p_j}) \in P} \pi_{p_i}) + \pi_{p_k} \leq C$.

We used a collection of instances of OP in which G is a 4-connected grid (a graph in which each node is connected to at most four other nodes) with $w_{ij} = 1$ for each $(v_i, v_j) \in E$. Hence, a path can exit a node in one among (at most) four possible directions. We did two additional changes, w.r.t. the original definition of OP:

- Each node can either contain a non-negative profit or be an obstacle, in which case it is not traversable by any path.
- The profit corresponding to any node in the path of the agent is collected once.

To model this problem, we use $\lceil \frac{D}{2} \rceil$ table constraints, each describing agent position and accumulated reward across two consecutive time-steps. If G is a grid of size $r \times s$, then each table contains $2 \times r \times s$ columns. In each row, only two cells are assigned values between 1 and D, representing agent position and collected reward. Instances with grid size from 7×7 to 10×10 have been generated (see Figure 5).

5.2 Results

We now report the results considering the *solve time* statistic provided by MiniZinc on the instances described in Section 5.1. Such tests have been executed on both CT^{f}_{CU} and CT^{uf}_{CU} implementations on a system equipped with an Intel Core i7-13700KF at base clock 3.4 GHz (up to 5.4 GHz in boost), 128 GB of DDR4 RAM and a NVIDIA GeForce RTX 4090 with 16K CUDA cores, grouped into 128 streaming multiprocessors, running at 2.23 GHz (up to 2.52 GHz). On the software side, the system runs Ubuntu 24.04 and CUDA 12.7. All tests have been executed with 15 minutes timeout. Barplots summarizing the serial (CT) and CUDA solve times are now presented. The same results in tabular form are available in the appendix. Such tables show also the number of propagation performed for each test, allowing to calculate additional metrics, such as the performance gain in terms of propagations per second. Propagation times of some tests are analyzed more in detail in Section 5.2.1.

From Figures 6 and 7 it can be observed that the kernels responsible for the update step scale well, taking advantage of the computational power of the GPU. Indeed, for all sufficiently large instances, CT^{uf}_{CU} outperformed CT^{f}_{CU}, achieving an average speedup of 2.88 compared to 2.11 on LIN_B instances, and 4.35 compared to 2.52 on LIB_EB. Figure 5 shows an overview of the solving times and speedups for the *Orienteering* test set. Notice that while such models involve several other constraints in addition to multiple table instances, the speedups remain consistent with those of *LIN_B* and *LIN_EB* instances.

Kernel execution profiling showed that during the tests performed, the GPU capabilities were not fully exploited. In all the tests, the average GPU utilization was roughly 45%. This is because the amount of work offloaded to the GPU is often not enough to

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

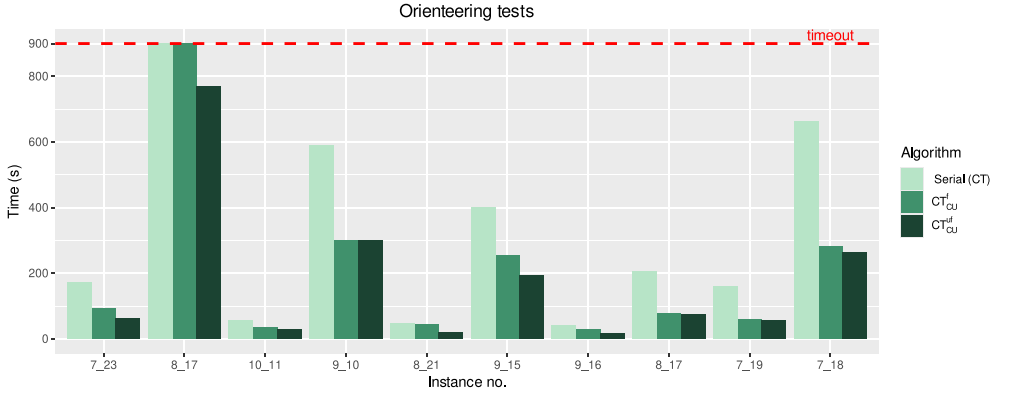

Fig 5. Barplot comparing serial and CUDA solve times for the *OR* instances. The grids considered are squared, the instance named r_C is an *OR* instance with a $r \times r$ grid and profit bound C.

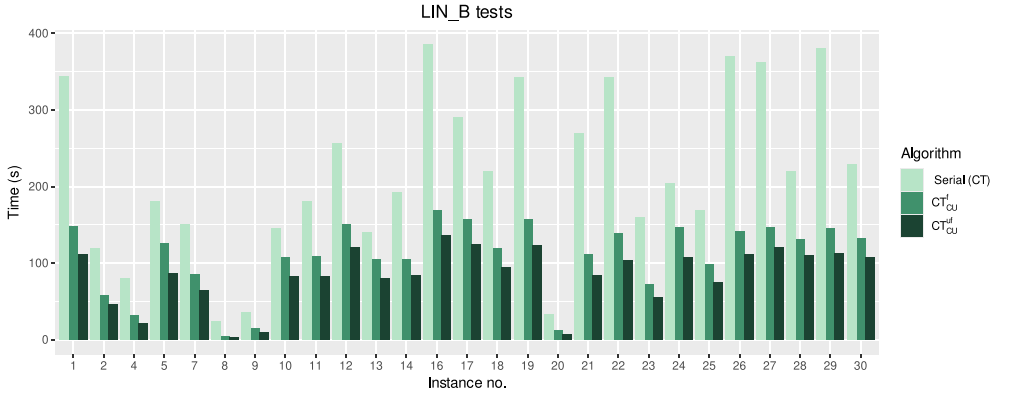

Fig 6. Barplot comparing serial and CUDA solve times for the first batch of instances. Test instances where all implementations timed out are omitted.

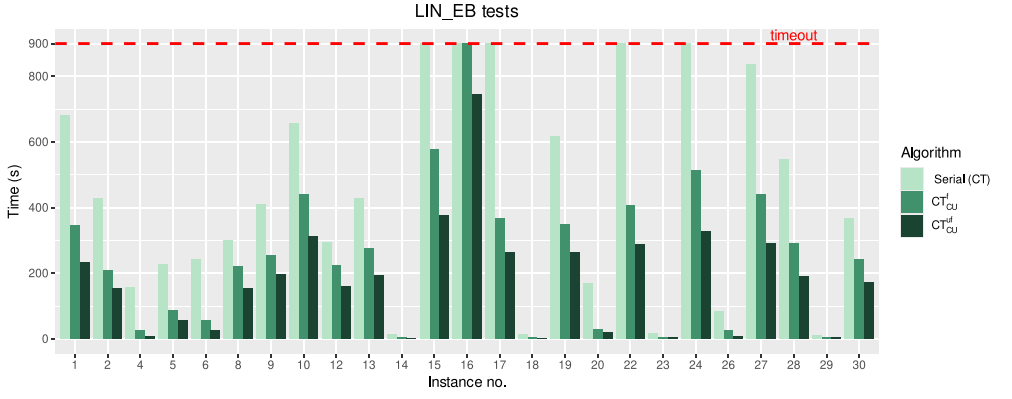

Fig 7. Barplot comparing serial and CUDA solve times for the second batch of instances. Test instances where all implementations timed out are omitted.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

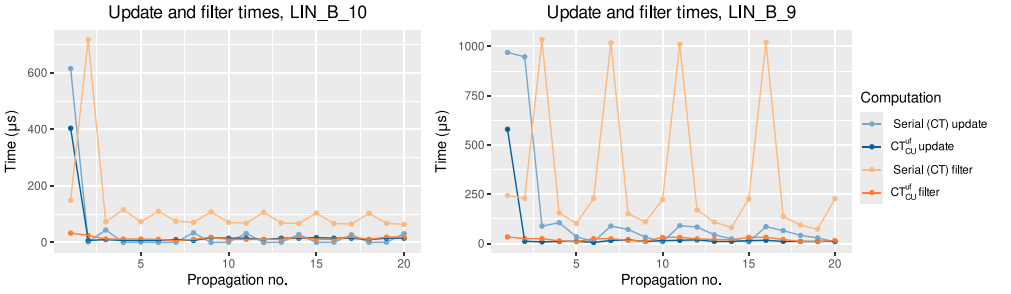

Fig 8. Series plots of the first 20 CT and CT^{uf}_{CU} propagation times for $TEST\_B\_10$ and $TEST\_B\_9$. The times are grouped for the update and filter procedures.

saturate all streaming multiprocessors. (To saturate the capabilities of the RTX 4090, much larger instances would be needed, involving much larger tables.) This supports the fact that the good performance achieved using the RTX 4090 is also expected when using GPUs equipped with less streaming multiprocessors.

Note that forcing a finer division of work for the sole purpose of using all available streaming multiprocessors often results in performance degradation. In such cases, in fact, required work synchronization and additional data exchange generate overhead that outweighs the benefits of parallel computation.

As concerns memory transfers, the time taken by memory copies from the host to the device and vise versa is significant when compared to the pure processing time. In part, this is due to the fact that the CT algorithm is very efficient. On average, on LIN_B and LIN_EB instances, the copy of data to the device can take up to 50% of the kernel time, while the copy of the results back to the host can take up to 80% of the kernel execution time.

5.2.1 CT^{uf}_{CU} propagations in detail

In this section we briefly analyze the behavior of CT^{uf}_{CU} executed on the first set of instances. We do so by analyzing the first 20 propagations of LIN_B_9 and LIN_B_10, which represent the two most representative behaviors observed in experimenting with the LIN_B and the LIN_EB instances. In the first scenario, s^{sup} has a stable size, making the serial filtering times quite constant. In contrast, the second case is characterized by high variability in the size of s^{sup}, leading to spikes in the serial filtering times, see Figure 8. In both scenarios however, the times recorded for the filtering kernel remain almost constant, this is due to the fact that the size of the grid launched is independent from $|s^{sup}|$. As it can be seen from Figure 8, concerning the first kind of instances, the filtering process is the procedure leading to the most sensible improvements when parallelized.

For what concerns the procedure responsible for updating $currTable$, Figure 8 compares the sum of the times of kernels $udpateTableGPU()$ and $reduce()$ (in $CUDA\ update$) with those of $updateTable()$. Such times, when paired with a small number of propagations, do not lead to a significative difference. Moreover, the sum of $udpateTableGPU()$ and $reduce()$ times, due to the grids not presenting a constant size, present the same time trends as the serial counterpart.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

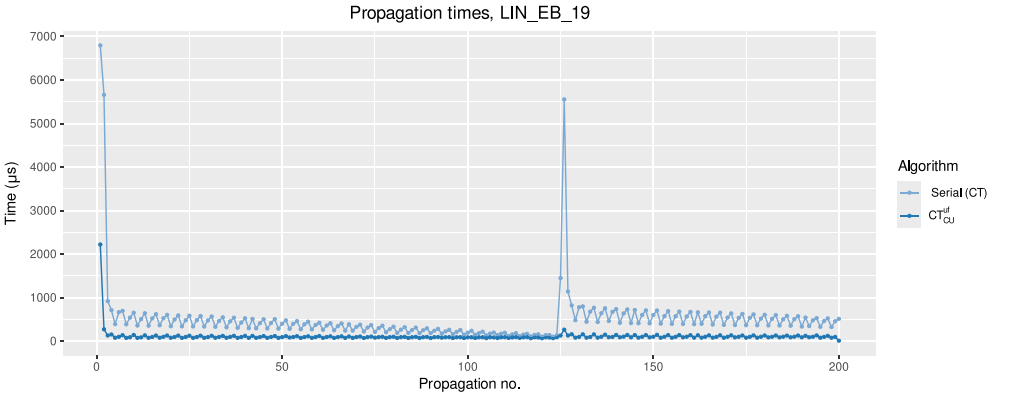

Fig 9. Series plot of the first 200 CT and CT^{uf}_{CU} propagation times for $TEST\_EB\_19$.

To have a direct comparison between the filtering and update steps performed on GPU and their serial counterparts, the times considered in Figure 8 do not include the copy and retrieval of the data nor the time taken to dump the domains in *_vars_host*.

Finally, we discuss the times spent by the overall propagations in CT and CT^{uf}_{CU}. For what concerns the serial propagations the time only includes the update of *_currTable* and filtering. On the other hand, for CT^{uf}_{CU} the time accounts for the kernels, the dump of the domains into *_vars_host*, the copy and retrieval of the data. Figure 9 shows the timings of the first 200 propagations of LIN_EB_19. Such an instance has been chosen since already in early propagations it presents a significant backtracking phase. As can be seen, once the valid tuples and domains start to reduce, the serial propagation times become very close to the CT^{uf}_{CU} ones. In some instances requiring many propagations toward the bottom part of the search tree, it can also happen that the serial propagations run faster than the parallel ones. In Figure 9 it can be noticed that around propagation no. 125 a substantial backtracking phase takes place. Such an event leads the domains and valid tuples in the table to increase again, advantaging CT^{uf}_{CU} over the serial propagator.

5.3 CT^{uf}_{CU} and gecode

We now present some of the results obtained by comparing CT^{uf}_{CU} against Gecode (version 6.3.0). Both solvers rely on a single thread (host side) with no parallelization features enabled. The comparison involves the second set of instances of Section 5.1. The instances have a number of variables between 100 and 150, a maximum domain size of 2000, and between 10000 and 15000 tuples in the table. The number of propagations for the tests in Table 3 ranges between 35000 and 85000. On some instances we noticed a slightly different number of propagations in favor of MiniCP. This difference is less than 2% and is likely due to CT^{uf}_{CU} relying on the built-in method *changed*() and not using *lastSizes* to update s^{val}. However, the number of explored nodes and failures is always the same for both solvers. Note that while the instance features (such as number of tuples or domain sizes) are comparable with those of LIN_EB instances, the number of propagations (see also the tabular data in the Appendix) result very different. This is due to the fact that all variables are now involved in the table constraint. Table 3 reports an excerpt of the instances we used, along with the solve times obtained by Gecode and CT^{uf}_{CU}.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Table 3. *The solving times (in seconds) for gecode and* CT^{uf}_{CU}

Instance	Solve times (s)		Speedup	Instance	Solve times (s)		Speedup
	Gecode	CT^{uf}_{CU}			Gecode	CT^{uf}_{CU}	
Ge_1	37.2	23.9	1.55	Ge_11	35.0	17.1	2.04
Ge_2	27.6	15.4	1.79	Ge_12	44.3	27.1	1.63
Ge_3	33.6	16.6	2.02	Ge_13	29.1	13.7	2.13
Ge_4	35.6	16.7	2.13	Ge_14	45.2	20.0	2.26
Ge_5	42.3	20.3	2.07	Ge_15	28.5	14.9	1.91
Ge_6	30.7	15.7	1.95	Ge_16	27.2	14.6	1.87
Ge_7	31.3	17.9	1.74	Ge_17	36.3	16.1	2.25
Ge_8	40.2	17.1	2.35	Ge_18	28.9	19.3	1.50
Ge_9	37.6	20.9	1.80	Ge_19	26.5	14.8	1.80
Ge_10	34.3	21.9	1.57	Ge_20	34.1	18.0	1.88

6 Conclusions

In this paper we presented a GPU parallel version of the propagation algorithm for the table constraint, a fundamental built-in constraint for Constraint Logic Programming languages. The massive parallelism offered by GPU computing allows such implementations to scale up as the CSP instance size grows. Such scaling was especially noticeable in the case of CT^{uf}_{CU} (Section 5.2). We have experimentally observed the speedup provided by the CUDA parallel implementations when compared to the serial counterpart. Such implementations, in particular CT^{uf}_{CU}, have been shown to be capable of achieving a significant speedup also against a state-of-the-art solver such as Gecode which relies on more sophisticated propagation techniques.

Acknowledgments

Research partially supported by Interdepartment Project on AI (Strategic Plan UniUD–22-25), and by Unione europea-Next Generation EU, missione 4 componente 2, project MaPSART "Future Artificial Intelligence (FAIR)," PNRR. A.Dovier and A.Formisano are members of GNCS-INdAM, Gruppo Nazionale per il Calcolo Scientifico.

References

ASSI, M. AND HARATY, R. A. 2018. *A survey of the knapsack problem*. In *International Arab Conference on Information Technology, ACIT 2018*, IEEE, Werdanye, Lebanon, 1–6.

BOCK, A. AND SANITÀ, L. 2015. *The capacitated orienteering problem. Discrete Applied Mathematics* 195, 31–42.

CIPRIANO, R., DOVIER, A. AND MAURO, J. 2008. *Compiling and executing declarative modeling languages to Gecode*. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9–13 2008, Proceedings*, M. G. DE LA BANDA and E. PONTELLI, Eds. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Vol. 5366, 744–748.

DEMEULENAERE, J., HARTERT, R., LECOUTRE, C., PEREZ, G., PERRON, L., RÉGIN, J. AND SCHAUS, P. 2016. *Compact-table: Efficiently filtering table constraints*

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

with reversible sparse bit-sets. In *Proc. of CP'16*, M. Rueher, Ed. Lecture Notes in Computer Science, Springer, Cham, Vol. 9892, 207–223.

DOVIER, A., FORMISANO, A., GUPTA, G., HERMENEGILDO, M. V., PONTELLI, E. AND ROCHA, R. 2022. *Parallel logic programming: A sequel*. *Theory and Practice of Logic Programming* 22, 6, 905–973.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2010. *An investigation of multi-agent planning in CLP*. *Fundamenta Informaticae* 105, 79–103.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2011. Perspectives on logic-based approaches for reasoning about actions and change. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, M. Balduccini and T. C. Son, Eds. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Vol. 6565, 259–279.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2013. *Autonomous agents coordination: Action languages meet CLP(FD) and Linda*. *Theory and Practice of Logic Programming* 13, 2, 149–173.

GENTZEL, R., MICHEL, L. AND VAN HOEVE, W. J. 2020. *HADDOCK: A language and architecture for decision diagram compilation*. In *Proc. of CP'20*, H. Simonis, Ed. Lecture Notes in Computer Science, Springer, Cham, Vol. 12333, 531–547.

LECOUTRE, C. 2011. *STR2: Optimized simple tabular reduction for table constraints*. *Constraints* 16, 4, 341–371.

LECOUTRE, C., LIKITVIVATANAVONG, C. AND YAP, R. H. C. 2015. *STR3: A path-optimal filtering algorithm for table constraints*. *Artificial Intelligence* 220, 1–27.

MICHEL, L., SCHAUS, P. AND VAN HENTENRYCK, P. 2021. *MiniCP: a lightweight solver for constraint programming*. *Mathematical Programming Computation* 13, 133–184.

PEREZ, G. AND RÉGIN, J. 2014. *Improving GAC-4 for table and MDD constraints*. In *Proc. of CP 2014*, B. O'Sullivan, Ed. Lecture Notes in Computer Science, Springer, Cham, Vol. 8656, 606–621.

ROSSI, F., VAN BEEK, P. AND WALSH, T. 2006. Handbook of constraint programming. In *Foundations of Artificial Intelligence*, Vol. 2, Elsevier, The Netherlands.

STUCKEY, P. J., FEYDY, T., SCHUTT, A., TACK, G. AND FISCHER, J. 2014. *The minizinc challenge 2008–2013*. *AI Magazine* 35, 2, 55–60.

TARDIVO, F., DOVIER, A., FORMISANO, A., MICHEL, L. AND PONTELLI, E. 2023. *Constraint propagation on GPU: A case study for the alldifferent constraint*. *Journal of Logic and Computation* 33, 8, 1734–1752.

TARDIVO, F., DOVIER, A., FORMISANO, A., MICHEL, L. AND PONTELLI, E. 2024a. Constraint propagation on GPU: A case study for the cumulative constraint. *Constraints* 29, 192–214.

TARDIVO, F., MICHEL, L. AND PONTELLI, E. 2024b. CP for bin packing with multi-core and GPUs. In 30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain, P. Shaw, Ed. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 307, 28:1–28:19.

TRAVASCI, S., TARDIVO, F. AND FORMISANO, A. 2025. GPU-accelerated propagation for the stable marriage constraint. In Proceedings of the 40th Italian Conference on Computational Logic, Alghero, Italy, June 25-27, 2025, D. Guidotti, L. Pandolfo and L. Pulina, Eds. CEUR Workshop Proceedings. CEUR-WS.org, Aachen. Vol. 4003. https://ceur-ws.org/Vol-4003

VERHAEGHE, H., LECOUTRE, C., DEVILLE, Y. and SCHAUS, P. 2017a. Extending compact-table to basic smart tables. In Proc. of CP'17, Beck, J. C., Ed. Lecture Notes in Computer Science, Springer, Cham, Vol. 10416, 297–307.

VERHAEGHE, H., LECOUTRE, C. and SCHAUS, P. 2017b. Extending compact-table to negative and short tables. In Proc. of AAAI'17, S. Singh and S. Markovitch, Eds. AAAI Press, San Francisco, California, USA, 3951–3957.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Table 4. *MiniZinc solve times, number of propagations and speedup for CT^{f}_{CU} and CT^{uf}_{CU} on the first batch of tests. A barplot summing up the table is presented in Figure 6*

	Solve times (sec)			Speedup		
Instance	Serial	CT^{f}_{CU}	CT^{uf}_{CU}	CT^{f}_{CU}	CT^{uf}_{CU}	Propagations (10^6)
LIN_B_1	344.6	147.6	111.4	2.33	3.09	3.72
LIN_B_2	119.1	58.4	46.7	2.04	2.55	2.02
LIN_B_3	T.O.	T.O.	T.O.	-	-	-
LIN_B_4	80.5	32.1	21.7	2.51	3.71	0.62
LIN_B_5	181.4	125.7	87.2	1.44	2.08	3.43
LIN_B_6	T.O.	T.O.	T.O.	-	-	-
LIN_B_7	150.8	86.0	64.5	1.75	2.34	2.59
LIN_B_8	24.5	5.2	2.8	4.70	8.61	0.07
LIN_B_9	35.5	15.2	10.2	2.34	3.47	0.33
LIN_B_10	145.5	108.1	83.2	1.35	1.75	2.99
LIN_B_11	181.3	109.0	83.2	1.66	2.18	3.15
LIN_B_12	256.9	151.2	121.4	1.70	2.12	4.53
LIN_B_13	140.7	104.6	80.1	1.34	1.76	3.33
LIN_B_14	192.5	105.7	84.4	1.82	2.28	2.77
LIN_B_15	T.O.	T.O.	T.O.	-	-	-
LIN_B_16	385.7	168.8	135.9	2.28	2.84	4.71
LIN_B_17	290.4	156.9	125.3	1.85	2.32	4.72
LIN_B_18	219.6	119.2	94.2	1.84	2.33	2.89
LIN_B_19	342.9	156.8	124.0	2.19	2.76	4.00
LIN_B_20	32.7	12.5	7.7	2.61	4.24	0.22
LIN_B_21	269.8	112.0	84.0	2.41	3.21	2.61
LIN_B_22	342.5	139.2	104.1	2.46	3.29	3.32
LIN_B_23	160.6	72.7	55.5	2.21	2.89	1.58
LIN_B_24	204.9	146.8	107.1	1.40	1.91	4.27
LIN_B_25	168.9	98.9	75.5	1.71	2.24	2.73
LIN_B_26	370.9	142.1	111.6	2.61	3.32	3.36
LIN_B_27	362.8	147.0	120.6	2.47	3.01	3.58
LIN_B_28	220.1	130.7	110.5	1.68	1.99	4.40
LIN_B_29	380.5	145.4	113.0	2.62	3.37	3.28
LIN_B_30	229.6	133.0	108.3	1.73	2.12	3.21

Appendix A Results of the experiments

In this section we report the tables summarizing the performance of the tested implementations across LIN_B and LIN_EB batches. Such results are summarized in Figures 6 and 7 and included here in detail for completeness.

Solve times entries in Tables 4 and 5 presenting a $T.O.$ value indicate that such a test timed out after 15 minutes.

Let CT_{CU} denote either CT^{f}_{CU} or CT^{uf}_{CU}, its relative speedup is computed as $\frac{serial\ sovle\ time}{CT_{CU}\ solve\ time}$. Table 5 presents some instances where the serial solver timed out but CT_{CU} did not. In such scenarios a lower bound on the achievable speedup is reported, calculated by imposing *serial solve time* to the 15 minute lower bound, that is $\frac{900}{CT_{CU}\ solve\ time}$.

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press

Table 5. *MiniZinc solve times, number of propagations and speedup for CT^{f}_{CU} and CT^{uf}_{CU} on the second batch of tests. A barplot summing up the table is presented in Figure 7*

	Solve times (sec)			Speedup		
Instance	Serial	CT^{f}_{CU}	CT^{uf}_{CU}	CT^{f}_{CU}	CT^{uf}_{CU}	Propagations (10^6)
LIN_EB_1	681.5	347.2	231.7	1.96	2.94	6.84
LIN_EB_2	429.7	208.3	153.9	2.06	2.79	3.96
LIN_EB_3	T.O.	T.O.	T.O.	-	-	-
LIN_EB_4	157.1	26.5	9.0	5.93	17.44	0.08
LIN_EB_5	226.8	86.3	57.7	2.63	3.93	1.87
LIN_EB_6	243.0	55.3	25.9	4.39	9.38	0.48
LIN_EB_7	T.O.	T.O.	T.O.	-	-	-
LIN_EB_8	300.1	220.5	154.3	1.36	1.94	5.56
LIN_EB_9	410.2	253.9	196.7	1.62	2.09	6.09
LIN_EB_10	656.6	439.7	313.8	1.49	2.09	13.22
LIN_EB_11	T.O.	T.O.	T.O.	-	-	-
LIN_EB_12	293.6	223.6	160.8	1.31	1.83	5.68
LIN_EB_13	427.8	276.6	194.6	1.55	2.20	5.85
LIN_EB_14	13.3	4.6	3.0	2.89	4.46	0.07
LIN_EB_15	T.O.	578.1	375.4	> 1.56	> 2.40	14.04
LIN_EB_16	T.O.	T.O.	744.2	-	> 1.21	13.19
LIN_EB_17	T.O.	366.4	262.2	> 2.46	> 3.43	8.92
LIN_EB_18	12.2	3.9	2.5	3.10	4.81	0.06
LIN_EB_19	617.6	348.4	262.3	1.77	2.35	6.97
LIN_EB_20	168.8	29.0	19.5	5.82	8.67	0.28
LIN_EB_21	T.O.	T.O.	T.O.	-	-	-
LIN_EB_22	T.O.	405.9	286.8	>2.22	>3.14	6.45
LIN_EB_23	16.6	5.2	3.4	3.20	4.95	0.07
LIN_EB_24	T.O.	513.3	328.8	>1.75	>2.74	10.33
LIN_EB_25	T.O.	T.O.	T.O.	-	-	-
LIN_EB_26	84.1	24.5	6.7	3.40	12.53	0.13
LIN_EB_27	837.5	440.6	289.8	1.90	2.89	7.34
LIN_EB_28	545.7	290.2	192.0	1.88	2.84	5.57
LIN_EB_29	12.0	4.7	3.4	2.62	3.59	0.08
LIN_EB_30	366.1	241.2	171.0	1.52	2.14	4.84

https://doi.org/10.1017/S1471068425100197 Published online by Cambridge University Press