Constraint propagation on GPU: A case study for the AllDifferent constraint

FABIO TARDIVO, *Department of Computer Science, New Mexico State University, Las Cruces, 88003, New Mexico, USA.*
E-mail: ftardivo@nmsu.edu

AGOSTINO DOVIER, *Department of Math, Computer Science, University of Udine, Udine, 33100, Italy.*
E-mail: agostino.dovier@uniud.it

ANDREA FORMISANO, *Department of Math, Computer Science, University of Udine, Udine, 33100, Italy.*
E-mail: andrea.formisano@uniud.it

LAURENT MICHEL, *Department of Computer Science, Engineering, University of Connecticut, Storrs, Connecticut, 6269-4155, USA.*
E-mail: ldm@uconn.edu

ENRICO PONTELLI, *Department of Computer Science, New Mexico State University, Las Cruces, 88003, New Mexico, USA.*
E-mail: epontell@nmsu.edu

Abstract

The *AllDifferent* constraint is a fundamental tool in Constraint Programming. It naturally arises in many problems, from puzzles to scheduling and routing applications. Such popularity has prompted an extensive literature on filtering and propagation for this constraint. This paper investigates the use of General Processing Units (GPUs) to accelerate filtering and propagation. In particular, the paper presents an efficient parallelization of the *AllDifferent* constraint on GPU, along with an analysis of different design and implementation choices and evaluation of the performance of the resulting system on several benchmarks.

Keywords: Constraint propagation, AllDifferent, parallelism, GPU computing

1 Introduction

Constraint Programming (CP) is a declarative paradigm used to model and solve combinatorial problems. Problems are modeled using a set of variables, each provided with a set of possible values (the domain of the variable), and a set of constraints that characterize the feasible solutions. Dedicated *constraint solvers* are used to process the problem models and identify solutions. Thanks to the MiniZinc Challenge [32], an annual competition among solvers, the CP language MiniZinc [27] has emerged as a de-facto standard modeling language for the CP community.

Vol. 33, No. 8, Published by Oxford University Press 2023.

Advance Access Publication on 6 June 2023 https://doi.org/10.1093/logcom/exad033

Traditional constraint solvers operate by alternating two stages: non-deterministic variables assignment and constraint propagation. Once a value has been assigned to a variable, constraint propagation eliminates values from domains of other variables that are incompatible in any solution with the assignment that has just been made. Alternative assignments are typically explored through backtracking.

The effectiveness of constraint propagation is heavily dependent on how the problem is modeled. For example, it is frequently possible to model the same problem using either a collection of elementary (e.g. binary or ternary) constraints or a single constraint involving many variables (i.e. a *global* constraint). Global constraints have the advantage of capturing a complex relationship between many variables, typically allowing a more extensive level of propagation. The impact of propagation on the structure of the search tree explored by a constraint solver can be significant—indeed, the optimized propagation of global constraints is the subject of many studies [31].

The *AllDifferent* constraint, which requires all variables in the constraint to be assigned a distinct value, naturally arises in many problems, from puzzles to scheduling and routing applications. Such popularity has prompted extensive studies on the propagation of this global constraint. There are different algorithms to propagate the *AllDifferent* constraint, each with a different trade-off between propagation strength and computational cost [43]. The most popular approach is the one by Régin [29].

The use of parallelism to enhance performance of CP solvers has been extensively investigate, with emphasis on CPU-based multi-threaded solutions [14]. These approaches usually involve either (i) decomposing and distributing the problem or (ii) dividing and distributing the search space exploration. Such methods are prone to issues such as load imbalance and synchronization overhead.

Recently, the use of Graphical Processing Units (GPUs) has led to extensive performance benefits in several areas of AI, such as Machine Learning. Relatively, more limited work has been done in exploring the use of GPUs for logic-based AI [6, 7], e.g. [4] for SAT, [8, 9] for ASP and [3] for CP.

In this paper, we present a GPU-accelerated propagator for the *AllDifferent* constraint. To the best of our knowledge, this is the first work on accelerating the propagation of a global constraint using GPUs.

Our contributions are as follows: an analysis of Regin's algorithm to identify which parts are amenable of parallelization using a GPU; the comparison of alternative parallelization approaches; the integration of both the MiniZinc support and our GPU-accelerator propagator in a lightweight constraint solver [26]; and a comparison between the standard propagator and our GPU-accelerated version. Results on large instances of the Travelling Salesman Problem, N-Queens and the Langford's Problem demonstrate encouraging speedup.

The rest of the paper is organized as follows: Section 2 gives an introduction to CP, the Regin's algorithm for *AllDifferent*, related works and the use of GPU for general computation. Section 3 describes the parallelization process, the integration in a constraint solver and the implementation details of the final algorithm. In Section 4, we describe the benchmarks used to test the GPU-accelerated propagators, analyse its scalability and present the final results. Finally, Section 5 summarizes the paper and gives some directions for future works.

2 Background

2.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem (CSP)* can be described by a triple $P = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{V} = \{V_1, \ldots, V_n\}$ is a finite set of variables, $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ is a finite set of sets, called *domains,*

and $\mathcal{C}$ is a set of *constraints* on the variables $\mathcal{V}$. The domain $\mathcal{D}_i$ captures the allowable values for the variable V_i. Every *constraint* $c \in \mathcal{C}$ is defined over a subset $var(c) \subseteq \mathcal{V}$ called *scope* of c. Assume $var(c) = \{V_{i_1}, \ldots, V_{i_m}\}$, then c is a *relation* on $\mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_m}$, namely $c \subseteq \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_m}$. A *solution* is an assignment $\sigma : \mathcal{V} \longrightarrow \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_n$ such that:

- for $i = 1, \ldots, n$: $\sigma(V_i) \in \mathcal{D}_i$ and
- for all c in $\mathcal{C}$, if $var(c) = \{V_{i_1}, \ldots, V_{i_m}\}$, then $\langle \sigma(V_{i_1}), \ldots, \sigma(V_{i_m}) \rangle \in c$.

In this paper, we focus on CSPs on finite domains, i.e. each $\mathcal{D}_i$ is a finite set. Whenever clear from the context, we will use syntactic sugars for commonly understood constraints (e.g. $V_3 < 2 \cdot V_5$). We will use the term *global constraint* to refer to constraints that define relationships between a non-fixed number of variables.

Given a CSP P, a *constraint solver* looks for one or more solutions of P. A typical solver alternates two types of processes in the search for solutions: (i) constraint propagation and (ii) non-deterministic choices. The latter step is used to select the next variable to be assigned and to select non-deterministically a value to be given to the variable (drawn from its current domain). Constraint propagation makes use of the constraints to remove from the domains of the variables values that can be proved not to belong to any solution compatible with the assignments made thus far. The choice of the variable is typically fast compared to the cost of constraint propagation.

2.2 Constraint Propagation

Constraint propagation is a fundamental technique in CP. As mentioned, it is used to reduce the search space by identifying and removing inconsistent values from the domains of the variables. The propagation of a single constraint is derived from the specific semantics of such constraint. For example, the propagation of the constraint $V_i \leq V_j$ removes from D_j all values smaller than k, when k is assigned to V_i. Similarly, when h is assigned to V_j, it removes from D_i all values bigger than h.

When the domains of a variable is reduced, constraints involving such a variable may be used to reduce the domains of other variables. This leads to a chain of propagations that terminates when it is not possible to remove further values (i.e. a fix-point is reached). A simple way to implement such a mechanism is by using a queue containing all constraints to be propagated. A constraint is added to the queue when the domain of at least one variable in its scope is changed.

Algorithms used to propagate constraints are called *filtering* algorithms and are subject to a trade-off between computational complexity and the amount of values they can remove. A strong filtering algorithm is one that leads to *hyper-arc consistency* [31]. An m-ary constraint c on the variables $var(c) = \{V_{i_1}, \ldots, V_{i_m}\}$ is *hyper-arc consistent (HAC)* if for all $j = 1, \ldots, m$ it holds that:

$$(\forall a_j \in \mathcal{D}_{i_j})(\exists a_1 \in \mathcal{D}_{i_1}) \cdots (\exists a_{i-1} \in \mathcal{D}_{i_{j-1}}) \\ (\exists a_{i+1} \in \mathcal{D}_{i_{j+1}}) \cdots (\exists a_m \in \mathcal{D}_{i_m})(\langle a_1, \ldots, a_m \rangle \in c).$$

A CSP is *hyper-arc consistent* if all constraints in $\mathcal{C}$ are HAC. In case of binary constraints (i.e. $m = 2$) the HAC property reduces to *arc consistency.* The time complexity of naive algorithms for achieving HAC is exponential in m.

It is common practice to simplify constraints involving many variables into collections of constraints involving a smaller number of variables (e.g. 2 or 3). For example, a constraint like $X + 2Y + 3U < 4V + Z$ can be translated to the simpler constraints $A < B, A = X + C, C = 2Y + 3U, B = 4V + Z$. Notice that this type of translation may lead to a reduced filtering capability during constraint propagation, since the HAC property is guaranteed only for the simple constraints. However, constraints can be rewritten in different ways, differently affecting the effectiveness of the

propagation step. For instance, for the above constraint one can more efficiently exploit a built-in constraint capable of handling sums of linear terms. The given constraint can be first rewritten as $X+2Y+3U-4V-Z<0$ and then, by using the scalar product global constraint, as the equivalent $[X,Y,U,V,Z]\cdot[1,2,3,-4,-1]<0$.

The CP literature has explored a number of dedicated algorithms to handle propagation for specific types of constraints. In what follows, we focus on the global constraint *AllDifferent*.

2.3 AllDifferent

The *AllDifferent* global constraint deals with a list of variables (of any length) and aims at ensuring that all of them are assigned pairwise different values in the solution. Even though $\mathit{AllDifferent}(x_1,\ldots,x_n)$ admits exactly the same set of solutions as the set of binary constraints $\{x_i\neq x_j\ :\ 1\leq i<j\leq n\}$, arc consistency applied to the individual binary constraints delivers a weaker filtering of the domains than considering the original global constraint. A comprehensive review on the *AllDifferent* constraint is out of the scope of this work, interested readers can refer to [43] for a review of the different propagation algorithms, to [13, 48] for various improvements and to [33] for a multi-thread and distributed implementation.

Régin's well-known algorithm [29] for *AllDifferent* is based on a bipartite graph representation of the constraint that matches variables with values. In general, a bipartite graph $G(N_1\cup N_2,E)$ is defined over two disjoint sets of nodes N_1 and N_2 and $E\subseteq N_1\times N_2$ are undirected edges. A *matching* of a bipartite graph is a set of edges $M\subseteq E$ such that no two distinct edges share a node. A maximum matching is a maximum cardinality matching. The Hopcroft–Karp algorithm [17] for computing a maximum matching in a bipartite graph has $O(\sqrt{n}\cdot|E|)$ running time, while the Ford–Fulkerson algorithm, which reduces the problem to a maximum flow, has time complexity $O(n\cdot|E|)$ [12], where $n=|N_1|+|N_2|$.

A *directed graph (digraph)* $G(N,A)$ pairs a set of nodes N with a set of arcs $A\subseteq N\times N$, i.e. a set of directed edges. A *path* $x_0,x_1,\ldots,x_m$ is a sequence of nodes such that $(x_i,x_i+1)\in A$ for $i=0,\ldots,m-1$. If $x_m=x_0$, the path is called a cycle. A *Strongly Connected Component (SCC)* M of G is a maximal subset of N such that, for all pairs $u,v\in M$, there is a path $u=x_0,x_1,\ldots,x_m=v$. It follows that there are no cycles with edges between different SCCs. The set of SCCs forms a partition of the nodes of the digraph. Tarjan's algorithm can be used to efficiently compute the SCCs of any digraph in $O(|N|+|A|)$ time [36].

Before discussing the GPU-based implementation, let us review the steps adopted in the propagation of the *AllDifferent* constraint. In particular, consider the constraint applied to n variables, i.e.

$$\mathit{AllDifferent}(x_1,\ldots,x_n).$$

Consider the following preliminary definitions. Given a bipartite graph $G(N_1\cup N_2,E)$ and a matching M of G, the *residual digraph* from G and M is a directed graph $R(N_R,A_R)$ built as follows (see Figure 1):

1. The matching M is used to define the set of arcs A_1 that directs the edges of E

$$\begin{aligned} A_1 \quad = \quad & \{(x,d)\ :\ x\in N_1, d\in N_2, \{x,d\}\in E\setminus M\}\cup \\ & \{(d,x)\ :\ x\in N_1, d\in N_2, \{x,d\}\in M\} \end{aligned}$$

   Namely, for each matching edge, there is an arc from value to variable and for each non-matching edge, the arc is directed from variable to value.

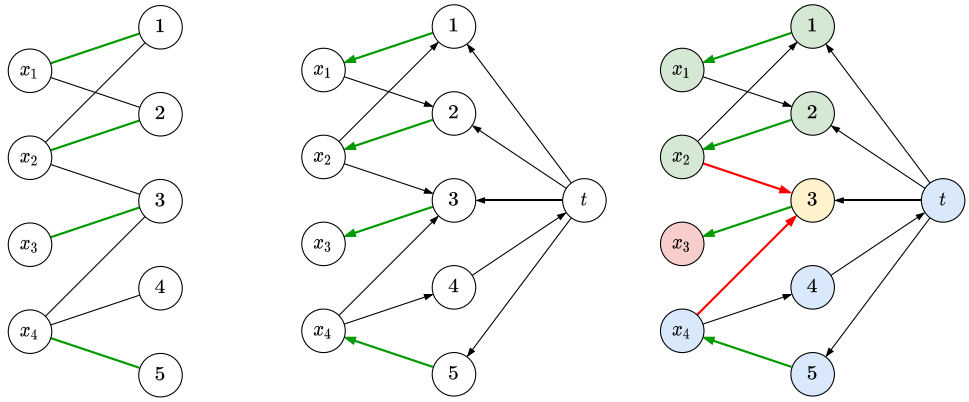

FIGURE 1. Quick overview of Regin's algorithm on x_1, x_2, x_3, x_4 where $D_1 = \{1, 2\}$, $D_2 = \{1, 2, 3\}$, $D_3 = \{3\}$, $D_4 = \{3, 4, 5\}$. In (a), we highlight in green the maximum match (step 1). In (b) it is pictured the residual graph (step 3). In (c), we paint with a different color the various SCCs (step 4); the arcs that will be removed are pictured in red (step 5).

2. A new sink node $t \notin N_1 \cup N_2$ is added: $N_R = N_1 \cup N_2 \cup \{t\}$.
3. The matching M is used to define the set of arcs between t and the nodes in N_2

$$\begin{aligned} A_2 \quad = \quad & \{(d, t) \ : \ d \in N_2, (\nexists x \in N_1)(\{d, x\} \in M)\} \cup \\ & \{(t, d) \ : \ d \in N_2, (\exists x \in N_1)(\{d, x\} \in M\} \end{aligned}$$

4. Finally, the set of arcs A_R is defined as $A_R = A_1 \cup A_2$.

Let us now review the algorithm to propagate $\mathit{AllDifferent}(x_1, \ldots, x_n)$. The algorithm constructs a bipartite graph $G = (N_1 \cup N_2, E)$ where:

- $N_1 = \{x_1, \ldots, x_n\}$,
- $N_2 = \bigcup_{i \in 1..n} \mathcal{D}_i$, where $\mathcal{D}_i$ is the domain of the variable x_i, and
- $E = \{\{x_i, d\} \mid i \in 1..n \wedge d \in \mathcal{D}_i\}$

The algorithms proceeds as follows (see also Figure 1):

1. Find a maximum matching M for $G(N_1 \cup N_2, E)$.
2. If $|M| < n$, then the constraint will be unsatisfiable.
3. Otherwise, construct the *residual digraph* $R(N_R, A_R)$ from G and M.
4. Compute the strongly connected components of R.
5. For every variable x_i, remove from its domains all the values d such that there exists an arc $(x_i, d) \in A_R$ or $(d, x_i) \in A_R$ that is not in M and connects two distinct SCCs.

In our implementation, we use the Hopcroft–Karp's algorithm for step 1, with a time complexity $O(\sqrt{|N_1| + |N_2|} \cdot |E|)$. Step 2 can be performed with complexity $O(1)$, since it is a simple check. Step 3 has complexity $O(|N_1| + |N_2| + |E|)$ (see R construction above). In step 4, we use the Tarjan's algorithm with complexity $O(|N_1| + |N_2| + |A|)$. Finally, step 5 has time complexity $O(|A|)$ since it scans all the arcs. In practice, the computational time can be reduced using several optimizations

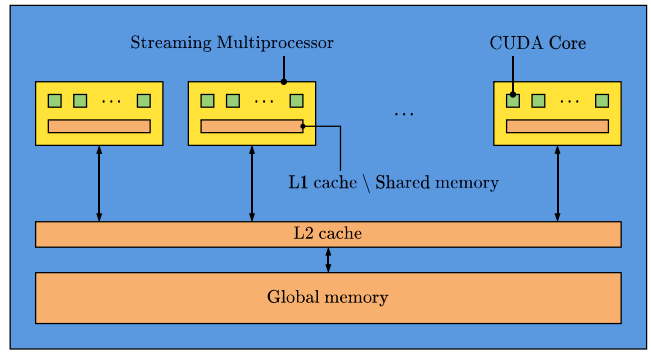

FIGURE 2. Simplified GPU architecture.

[13]. Our implementation mitigates the cost of step 1 using an incremental approach as described in [13].

Correctness of the procedure follows from a theorem by Berge that characterize the edges that belongs to some but not to all maximum matchings by just analysing a single maximum matching [2].

2.4 GPU Computing with CUDA

General-Purpose Computing on Graphics Processing Units (GPGPU) is the use of a Graphics Processing Units (GPU) to speed up computations traditionally handled by the Central Processing Unit (CPU). NVIDIA introduced the *Compute Unified Device Architecture (CUDA),* a general-purpose programming library that allows programmers to ignore the underlying graphical concepts in favor of high-performance computing concepts. It has been successfully used to accelerate computations in a variety of domains, such as physics, bioinformatics and machine learning [41].

The architecture of a GPU (Figure 2) includes a main memory (DRAM), typically off-chip and used as global memory, an L2 cache and an array of *Streaming Multiprocessors (SM)*. Each SM contains a small amount of on-chip fast memory, used as L1 cache or scratchpad memory (the *Shared memory*), and several *CUDA cores,* responsible for the actual execution of instructions. The architecture is designed to enable rapid context switching of lightweight threads.

The underlying conceptual model for parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT),* where the same instruction is executed by different threads, while data and operands may differ from thread to thread. A CUDA program includes parts meant for execution on the CPU (the *host*) and parts meant for parallel execution on the GPU (the *device*). Typically, the host code transfers data to the device memory (DRAM in Figure 2), starts parallel computations on the device and retrieves the results from device memory. The CUDA library supports interaction, synchronization and communication between host and device. Each device computation is described as a collection of concurrent threads, each executing the same function (called a *kernel,* in CUDA terminology). Each thread is executed by a CUDA core; these threads are hierarchically organized in *blocks* of threads, assigned to SMs. The threads in a block assigned to an SM execute the same instruction on different data. In case of control flow divergence among the threads within a block, their execution is serialized. Device global memory is accessible by all threads, whereas threads of the same block may access the high-throughput shared memory.

3 Design and Implementation

In this section, we explore the development of a constraint solver, which supports parallel propagation of *AllDifferent* on GPUs. Its code is open-source and is freely available online [34].

The first step in this process consists of selecting an existing constraint solver suitable to host a GPU-enabled *AllDifferent*. We looked at the fastest solvers of the recent MiniZinc challenges [38] and selected OR-Tools [28], JaCoP [21] and Gecode [37]. We realized soon that their efficiency is also due to several optimizations that make unsuitable to the modifications required to exploit parallelism.

Our choice converged on *MiniCP* [26], a lightweight solver specifically designed to enable research and exploration of diverse search and propagation methodologies. MiniCP provides a comprehensive documentation and a clean design. In particular, our research builds on *MiniCPP* [16], a C++ implementation of MiniCP, suitable to host C++ CUDA kernels. With minor optimizations, MiniCPP proved comparable to Jacop in terms of search time when they explore the same search tree on a collection of benchmarks.

To facilitate the use of MiniCPP we extend it with a front-end for the MiniZinc language. A MiniZinc model is a text file describing a constraint problem in a high-level, solver-independent way. The model is compiled (i.e. 'flattened') into a low-level format, called FlatZinc. A FlatZinc file is a text file that specifies variables, domains and constraints in an easy-to-parse way. A solver compatible with MiniZinc takes as input a FlatZinc file, creates the variables, domains and constraints specified in it and solves the problem.

Different solvers have different capabilities (e.g. a propagator for a specific constraint) and MiniZinc can take advantage of that if properly configured [39]. If a solver does not provide a propagator for a specific constraint, the flattening process decompose such constraint into basic constraints. MiniZinc allows us to enrich variables and constraints with annotations (i.e. tags) to provide additional information. Such annotations are not part of the model itself and a solver is free to either exploit them or ignore them.

We enable MiniCPP to read FlatZinc files using the skeleton parser provided by Gecode. Moreover, we create the necessary configuration files to inform the MiniZinc compiler of the MiniCPP's *AllDifferent* propagator, and to allow the custom `::gpu` annotation (see Listing ??). In this way, when MiniCPP parses an *AllDifferent* constraint annotated with `::gpu`, it uses the GPU algorithm in place of the standard CPU algorithm.

```
include "alldifferent.mzn";
include "minicpp.mzn";
...
constraint alldifferent(...) ::gpu;
...
```

LISTING 1.1: MiniZinc annotation for selecting the GPU propagator.

3.1 Parallelization

The key components of the filtering algorithm for *AllDifferent* propagation are (see Section 2.3):

1. the computation of a Maximum Matching in a bipartite graph (MM), and
2. the computation of the Strongly Connected Components (SCCs) of a directed graph.

TABLE 1. Average time to calculate MM and SCCs (in milliseconds) for the Travelling Salesman Problem (a), N-Queens (b) and Langford's Problem (c)

Instance	SCCs	MM	Instance	SCCs	MM	Instance	SCCs	MM
kroA200	0.012	0.003	200	0.046	0.008	2_100	0.014	0.003
rat575	0.028	0.005	600	0.364	0.056	2_300	0.039	0.010
dsj1000	0.052	0.014	1000	1.009	0.195	2_500	0.369	0.080
d2103	0.249	0.046	2000	4.271	0.712	2_1000	1.464	0.387
rl5934	35.845	9.109	6000	108.793	9.358	2_3000	34.134	3.870

Maximum Matching The results of the empirical study in [13] suggest that improving the computation of the SCCs is more beneficial than improving the incremental computation of MM. To verify such claim, we analysed the solving process of some instances of three problems: Travelling Salesman Problem, N-Queens and Langford's Problem (a description of these problems, the instances and the experimental setup are given in Section 4). We considered instances of increasing size and set 1 hour timeout. The results are reported in Table 1 and confirm that most of the computation time is spent in computing the SCCs. As a consequence of this observation, in our approach, we decided to keep the computation of MM on the CPU and exploit the GPU to speed up the computation of the SCCs.

For the sake of completeness, we report the main approaches for computing the maximum matching on GPU: Breadth-First Search (BFS) [5], auction [44] and push-relabel [47] algorithms.

Breadth-First Search Such approaches are based on the Hopcroft–Karp algorithm [17] and make use of a GPU-accelerated parallel BFS to find the augmenting paths.

Auction algorithm It works as an auction where agents compete for object by raising their prices. It alternates bidding and assignment phases until all agents have been assigned an object. The bidding and assignment phases are offloaded on the GPU, where bids and assignments are computed in parallel.

Push-relabel algorithm It solves the maximum matching reducing it to a maximum flow problem. It alternates push operations where flow is pushed through an edge, and relabel operation to mark the nodes with an excess of ingoing flow. Such alternation is repeated until no nodes, except t, have an excess of ingoing flow. The push and relabel phases are offloaded to the GPU where each node is processed in parallel.

We also explored the use of a GPU-accelerated push-relabel algorithm to speedup the computation of MM but it does not scale [18] and it is slower than the CPU incremental approach [13].

Strongly Connected Components. The majority of the approaches to calculate the SCCs on GPUs [1, 23] exploits, as a fundamental step, a parallel BFS to compute forward/backward reachability [11]. Moreover, these approaches mainly focuses on the case of huge sparse graphs with millions of nodes. This scenario does not fit well in our context, where we are faced with these specific aspects:

1. a major constraint leads to a dense graph of hundreds/thousands of nodes
2. we aim for low latency and GPU implementations of BFS notoriously suffers from load imbalance [42].

Because of (a), we decided to compute SCCs using forward reachability as follows. Let A be the adjacency (binary) matrix of the graph, namely $A(i,j) = 1$ iff there is an edge between node i and node j. Then:

1. Compute the forward reachability matrix F from A.
2. Transpose F to obtain the backward reachability matrix B.
3. Create a matrix C such that $C(i,j) = F(i,j) \cdot B(i,j)$. That is, $C(i,j) = 1$ if and only if there is a cycle containing node i and node j.
4. Set an identifier of each SCC as the minimum node i in that SCC (i.e. the SCC of the node i is identified by the minimum j such that $C(i,j) = 1$).

To fulfill the requirement (b), about optimal latency and load balance, we considered possible alternatives to BFS. In particular, we considered the following algorithms exploitable to compute the reachability matrix of a graph $G(V,E)$:

Matrix multiplication This approach starts with a Boolean matrix $M_0 = I + A$ where I is the identity matrix and A is the adjacency matrix of the graph. Then it performs the multiplication $M_{i+1} = M_i M_i$ for $i = 1, \ldots, \lceil \log_2(|V|) \rceil$. The matrix M_i represents the nodes reachable in 2^i steps, hence, to calculate the reachability matrix are necessary $\lceil \log_2(|V|) \rceil$ matrix multiplications,[1] for a total computational complexity of $O(|V|^{2.8} \lceil \log_2(|V|) \rceil)$ [10].

Warshall algorithm This algorithm is the core of the best known Floyd–Warshall algorithm that computes the shortest path between all the pairs of nodes. It starts from a Boolean matrix $M = I + A$ where I is the identity matrix and A is the adjacency matrix of the graph. Then it updates M for $|V|$ times. The k-th update modifies each element of the current Boolean matrix by putting $M(i,j) = M(i,j) \vee (M(i,k) \wedge M(k,j))$ (for each $1 \leq i,j \leq |V|$). This procedure has total computational complexity $O(|V|^3)$ [46].

ALGORITHM 1 Naive matrix multiplication algorithm.

Data: A, B squared matrices of size $n \times n$
Result: $C = AB$

```
for i = 1 to n do
    for j = 1 to n do
        C(i, j) = 0;
        for k = 1 to n do
            C(i, j) = C(i, j) + A(i, k) · B(k, j)
```

[1] The elements of the matrices are considered Boolean values, and $+, \cdot$ are replaced with $\vee, \wedge$.

ALGORITHM 2 Warshall algorithm.

Data: I, A identity matrix and adjacency matrix of size $n \times n$
Result: M reachability matrix

$M = I + A$
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $M(i, j) = M(i, j) \vee (M(i, k) \wedge M(k, j))$

The Warshall algorithm and the matrix multiplication have similar structures (see Algorithms 1 and 2), but since matrix multiplication is a fundamental operation in many different fields it is more studied and optimized. For example, the loops in the matrix multiplication algorithm are often rearranged to obtain better performances by processing the matrices according to their memory layout (i.e. row-major or column-major).

To determine which approach is the best choice in our context, we compared our GPU-accelerated Warshall algorithm (see Section 3.2) with state-of-the-art GPU-accelerated implementations of the binary matrix multiplication [19, 22], and with cuBLAS [40], the GPU-accelerated dense linear algebra library provided by NVIDIA.

The plot in Figure 3 shows the time spent to perform a single matrix multiplication and to perform the Warshall algorithm on random matrices of increasing size. It can be noted that one implementation of binary matrix multiplication complete faster than the Warshall implementation. However, we have to consider that in order to compute SCCs, the matrix multiplication have to be repeated up to $\lceil \log_2(|V|) \rceil$ times (possibly, less if a fixpoint is reached). This makes the Warshall algorithm more convenient and, consequently, we opted for it.

3.2 Implementation Details

Let us describe the main design choices we made concerning the data representation and the parallel Warshall algorithm.

Data representation. We choose to represent the residual graph's adjacency matrix as a bit matrix because it enables the use of bitwise operations and it can be initialized by dumping the domains' internal representation. Moreover, preliminary tests showed that transfer the bit matrix to the GPU is more efficient than transfer the domains, the match, and generate the matrix on the GPU. This is because, in our case, most of the transmission cost regards the initialization phase than the actual data transfer. The SCCs are returned to the CPU as an array of integers where the *i*-th element is the identifier of the SCC containing the *i*-th node.

Warshall Algorithm. The majority of the GPU implementations of the Warshall algorithm that use an adjacency matrix representation [20, 25] is based on a tiled version of the algorithm [45]. Such an algorithm was developed to maximize CPU's cache utilization, and it is particularly efficient in exploiting GPU's shared memory. As the Warshall algorithm, its tiled version starts from an adjacency matrix of size $n \times n$ and iteratively updates it to obtain the reachability matrix. First, it

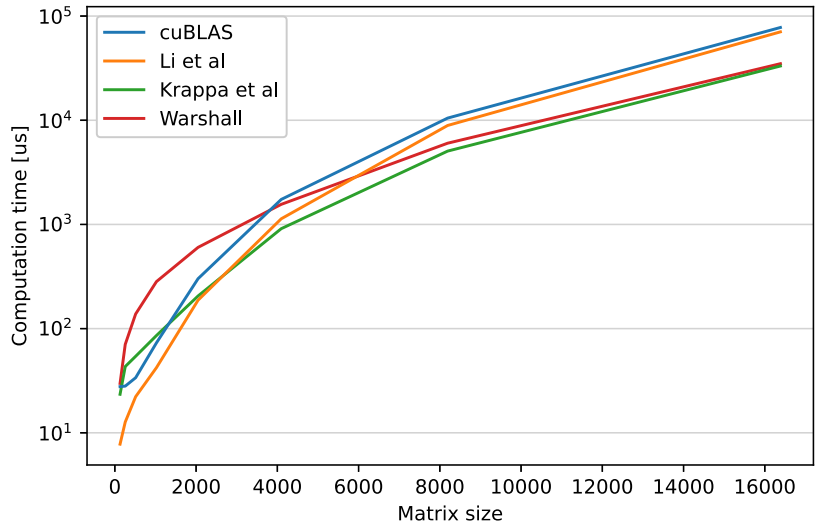

FIGURE 3. Comparison between GPU-accelerated matrix multiplication and GPU-accelerated Warshall algorithm.

splits the $n \times n$ matrix in $t \times t$ tiles,[2] each one identifying a submatrix T_{rc}, for $1 \leq r, c \leq \frac{n}{t}$. Then it updates the entire matrix $\frac{n}{t}$ times. An update (step, in the following) is performed in three phases, performed in sequence, each one updating a different set of tiles and accessing the tiles updated by previous phases/steps.

Let us describe the three phases of each step (see also Figure 4). At step $s \in [1, \frac{n}{t}]$, the phases work as follow:

Phase 1: This phase updates T_{ss}, the s-th tile of the main diagonal (Figure 4a). Each element $T_{ss}(i,j)$ (for $1 \leq i,j \leq t$) of T_{ss} is updated as follows:
$T_{ss}(i,j) = T_{ss}(i,j) \vee (T_{ss}(i,k) \wedge T_{ss}(k,j))$ for all $k \in \{1, \ldots, t\}$.
Notice that this phase only depends on the values of elements of T_{ss}.

Phase 2: This phase updates the tiles of the s-th row and s-th column, excluding T_{ss} (Figure 4b). More specifically, a tile in the s-th row and c-th column (for $c \neq s$) is updated by putting:
$T_{sc}(i,j) = T_{sc}(i,j) \vee (T_{ss}(i,k) \wedge T_{sc}(k,j))$, for $1 \leq i,j,k \leq t$.
Similarly, a tile of the s-th column and r-th row (for $r \neq s$) is updated as follows:
$T_{rs}(i,j) = T_{rs}(i,j) \vee (T_{rs}(i,k) \wedge T_{ss}(k,j))$, for $1 \leq i,j,k \leq t$.
Notice that this phase only reads T_{ss} (the tile updated in the previous phase) and the tile being updated.

Phase 3: This phase modifies all the remaining tiles (Figure 4c). A tile T_{rc} (for $c, r \neq s$) is updated in this manner:
$T_{rc}(i,j) = T_{rc}(i,j) \vee (T_{sc}(i,k) \wedge T_{rs}(k,j))$, for $1 \leq i,j,k \leq t$.
As before, this phase uses the outcome of previous phases to complete the step.

[2] If n is not a multiple of t it is increased to $t\lceil \frac{n}{t} \rceil$, and the matrix is padded with 0s.

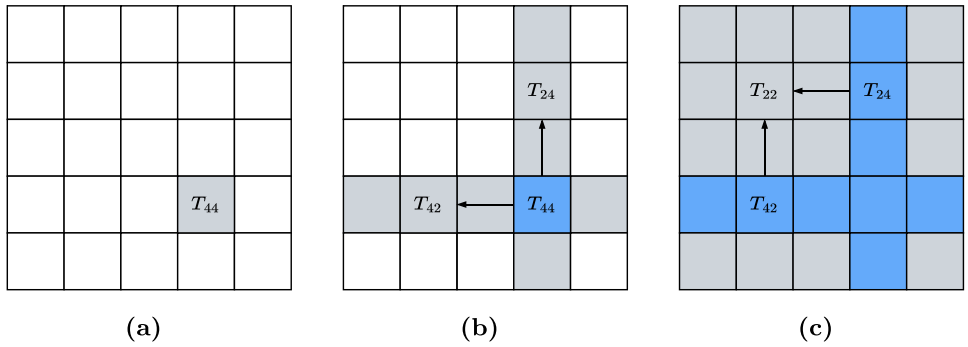

FIGURE 4. Illustration of the three phases of the 4-th step ($s = 4$ in Section 3.2) of the tiled Warshall algorithm on a matrix divided in 5×5 tiles. The tiles updated in each phase are highlighted in grey, while the tiles read (and already processed in previous phases) are colored in blue. Updates direction are denoted by arrows.

Note that the updates performed by phase 2 are independent from each other. Hence, all tiles of the s-th row and of the s-th column can be updated in parallel. Similarly, all tiles updated in phase 3 can be computed in parallel. These independent updates map well into the GPU computational model: tiles can be processed in parallel by different CUDA blocks. The parallelization of phase 2 and phase 3 are illustrated in Figure 5a and Figure 5b, respectively, where the mapping of tiles to CUDA blocks is shown.

Moreover, the update of each $t \times t$ tile can be parallelized by using t CUDA threads within a block: each of these threads is responsible for computing the elements of one row of the tile.

Considering the specific GPU we used and version of CUDA we employed, the most convenient value for t turned out to be 128. With such a choice for t, the GPU is allowed to read each row using a single memory access, as one `uint4` ($32 \cdot 4$ bits) and manipulating it by executing two 64-bit operations.

4 Experiments and Analysis

We compared our GPU-accelerated *AllDifferent* propagator (i.e. MiniCPP-GPU) with the (CPU) one present in MiniCPP. It is expected that a GPU implementation is advantageous on large instances, as the setup overhead would otherwise overshadow the benefits of the parallel execution. We chose three different benchmarks based on the *AllDifferent* constraint:

Travelling Salesman Problem Given a list of n cities and the distances between them, this problem asks to find the shortest possible route that visits each city exactly once and returns to the origin city. The Travelling Salesman Problem is suitable for our purposes because it can be modeled using a *Circuit* constraint, which internally makes use of *AllDifferent*. Moreover, there is an established set of large instances available [30].

N-Queens Given a $n \times n$ chessboard, this problem asks to place n non-attacking queens on the board. Such problem can be easily modeled using with three *AllDifferent* constraints and is an

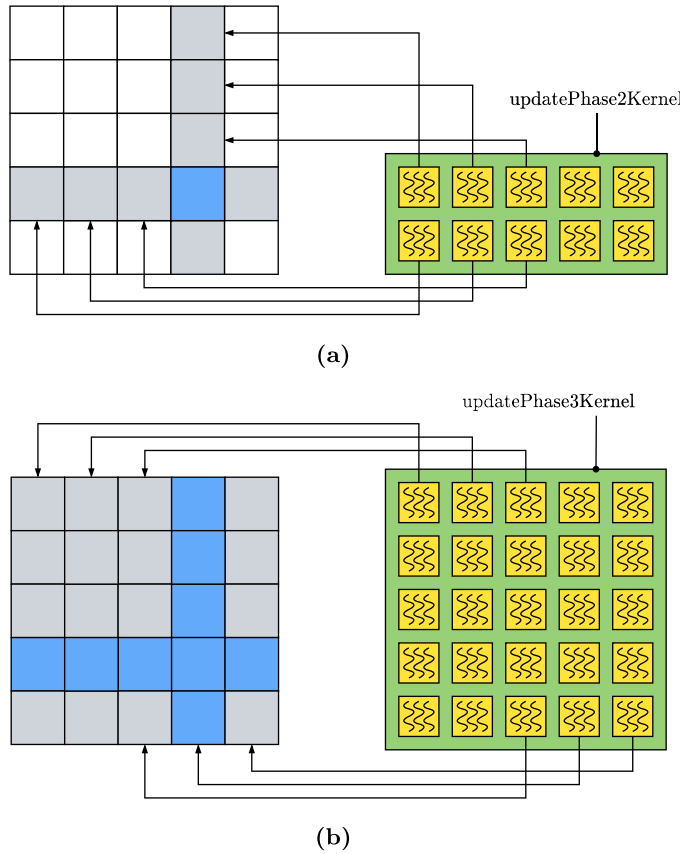

FIGURE 5. Parallelization of phase 2 (top) and phase 3 (bottom) of the tiled Warshall algorithm on GPU. The arrows show the mapping between matrix tiles (left) and CUDA blocks (right). The blocks responsible for tiles already processed (in blue) do not perform operations.

established test [13, 15, 24, 48] often used in benchmarks. Moreover, it is very easy to generate large instances by simply increasing the value of n.

Langford's Problem Given k copies of digits $1, \ldots, m$ this problem consist to arrange them so that any two consecutive copies of digit d are separated by d other digits. We choose this popular benchmark [13, 15, 48] in our experimentation because its model contains an *AllDifferent* constraint involving $k \cdot m$ variables, hence, one can obtain arbitrarily large instances by increasing k and m.

To give an effective and concise presentation, we defined two, partially overlapping, sets of (sizes of) instances:

- big-size: instances with 100, 200, 400, 600, 800, 1000 variables in the scope of the *AllDifferent* constraint(s);
- huge-size: instances with 1000, 2000, 4000, 6000, 8000, 10000 variables in the scope of the *AllDifferent* constraint(s).

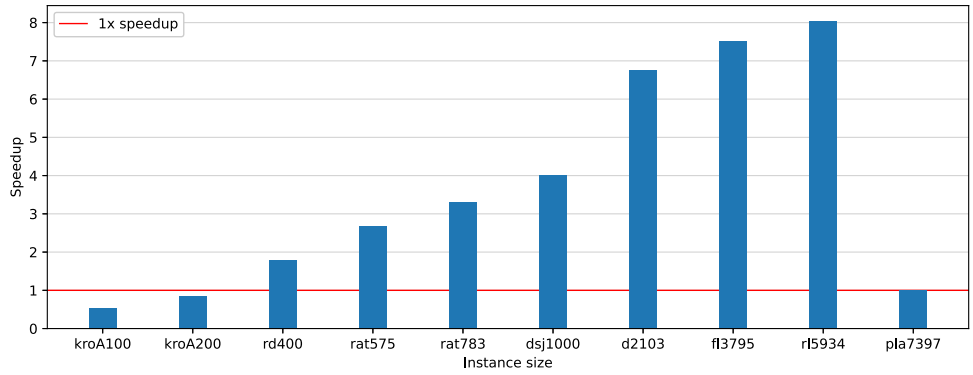

FIGURE 6. Speedup for the Travelling Salesman Problem.

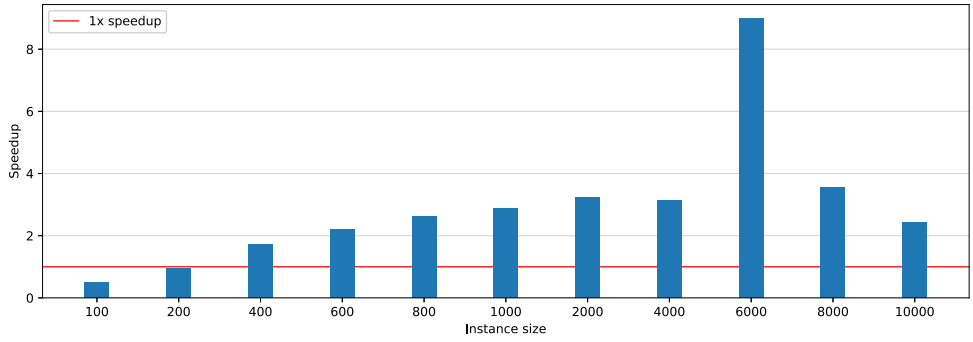

FIGURE 7. Speedup for the N-Queens.

Instances of the N-Queens and the Langford's problems where generated for each size of these sets by selecting values for the parameters n and k, m. For the Travelling Salesman Problem, we selected from [30] the instances with the size closest to the required one. Note that, we did not find in [30] any instance of size 10000. The models, instances and benchmark scripts are available [35].

To perform the tests in a reasonable amount of time, we limited the search process to 1 hour and we focus on the amount of work effectively done by each algorithm/implementation. The experiments are executed on a system equipped with an Intel Core i7-10700K, 32GB of RAM and a GeForce RTX 3080 running Ubuntu 22.04 and CUDA 11.8.

The results are illustrated in Figures 6, 7 and 8. The bars represent how faster MiniCPP-GPU explores the search tree compared to MiniCPP. In detail, the speedup is defined as the quantity

$$\frac{\text{Speed MiniCPP-GPU}}{\text{Speed MiniCPP}}$$

where the speed is the ratio between the number of explored nodes and the search time.

Figures 6, 7 and 8 show how the speedup varies with respect to the size if the instance (namely, the number of variables in the *AllDifferent* constraint). In Figures 7 and 8, the x-axes indicate the size of the instance (cf., the two classes of sizes described earlier, namely, big-size and huge-size instances).

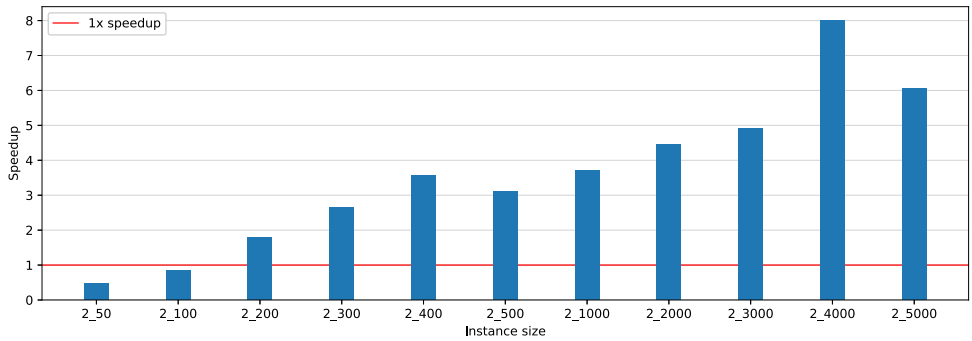

FIGURE 8. Speedup for the Langford's Problem.

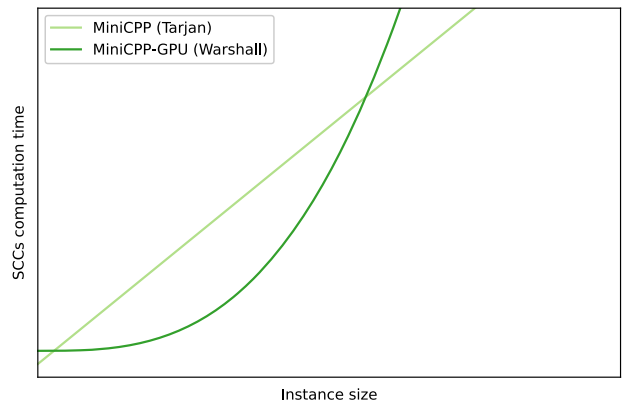

FIGURE 9. Qualitative representation of the computational time of MiniCPP ($O(n)$) and MiniCPP-GPU ($O(n^3)$).

For the Travelling Salesman Problem (Figure 6), the x-axis reports the name of the instance, as retrieved from [30], which includes the number of variables of the constraint (e.g. instance `fl3793` involves 3793 variables). The y-axes report the speedup as previously described.

It can be seen that in all three sets of experiments the speedup increases as the instance size increases, up to a maximum (8x in these experiments) and then decreases. This can be explained by considering the fact that we are using the GPU to accelerate an algorithm that has $O(n^3)$ time complexity, while Tarjan algorithm used in MiniCPP exposes linear complexity. Figure 9 depicts this situation. The shape of the curve depends on the hardware computational power and the problem/instance characteristics. However, the picture suggests that it is not convenient to offload the computation of the SCCs when there are too few variables (in our experiments this happens for instances having less than 300–400 variables). On the other hand, there is a point where the cubic complexity of the algorithm is not compensated by the speed of the parallel hardware. In our experiments the point of diminishing returns occurs around 6000–8000 variables. Note that this range, from 300 to 8000 variables, identifies a relevant class of problems. For this class the use of GPUs is significantly advantageous.

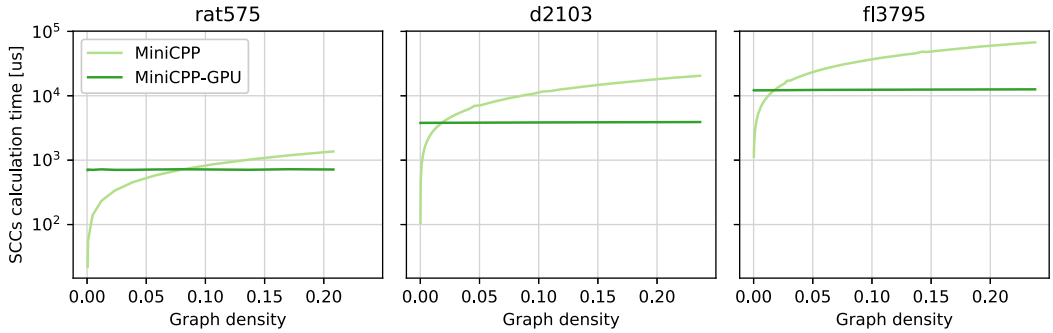

FIGURE 10. Evolution of SCCs' calculation time while the search is moving down in the search tree.

A less trivial circumstance where the GPU offload is inconvenient is when the search reaches the lower part of the search tree. At this stage, several edges of the original bipartite graph do not exist anymore because the relative variables have been fixed during the previous part of the computation. Having many variables already assigned allows MiniCPP to quickly calculate the SCCs by traversing the remaining few edges. On the opposite, the MiniCPP-GPU algorithm has to work on the entire (sparse) adjacency matrix. To emphasize the phenomenon, we selected `rat575`, `d2103`, `fl3795` among the instances of the Travelling Salesman Problem and timed the SCCs calculation on both MiniCPP and MiniCPP-GPU. The results are illustrated in Figure 10. Note the logarithmic scale and the fact that because the graphs are bipartite their *density* $\frac{|E|}{|V|\cdot|V|}$ cannot be larger than 0.25. Considering the progress of the computation, each of the three plots has to be read from right to left, since the density of the graphs decreases during the search. As expected, the CPU become faster as the graphs become less dense, while the GPU exhibits constant calculation time.

5 Conclusion and Future Works

Motivated by the benefits that GPUs offer in terms of computational power, we designed and implemented a GPU-accelerated propagator for the *AllDifferent* constraint. We described the process of developing such a propagator, which challenges we encountered, and the motivations behind the main implementation choices. The propagator has been integrated into an existing solver. We tested our implementation on large instances of different benchmarks and obtained speedups up to 8 times in terms of number of nodes explored in the unit of time. Unlike other parallel approaches that run on a cluster of PCs or are tailored to a specific application, our method is easy accessible thanks to the MiniZinc compatibility and the common presence of a GPU in modern PCs. Despite the performances of our approach can be improved in the lower part of the search tree, the empirical results show that GPGPU can be successfully applied in CP.

The analysis suggests that the application of GPUs to accelerate constraint propagation is limited by two factors: the amount of useful parallelizable work and the CPU-GPU communication overhead. As future work, we plan to investigate these limitations. The first limitation can be mitigated by parallelizing algorithms with strong filtering, which usually have high computational complexity. By utilizing GPUs, we aim to achieve fast computation and robust filtering, in contrast with the current trend of sacrificing filtering strength for low computational complexity. The second limiting factor can be mitigated by offloading multiple constraints simultaneously to the GPU. The aim is

to improve overall efficiency by overlapping data communication and computation, which would be particularly beneficial for small and medium-sized problems where the GPU hardware is not fully utilized.

Acknowledgements

Agostino Dovier and Andrea Formisano are partially supported by Interdepartmental Project on AI (Strategic Plan UniUD–22-25) and by INdAM-GNCS projects CUP E55F22000270001 and CUP E53C22001930001. Laurent Michel is partially supported by Synchrony. Enrico Pontelli and Fabio Tardivo are supported by NSF grants 2151254, 1914635 and 1757207.

References

[1] J. Barnat, P. Bauch, L. Brim and M. Ceška. Computing strongly connected components in parallel on CUDA. In *In 2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 544–555. IEEE, 2011.

[2] C. Berge. *Graphs and Hypergraphs*. Elsevier Science Ltd., GBR, 1985.

[3] F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto and E. Pontelli. Exploring the use of GPUs in constraint solving. In *Practical Aspects of Declarative Languages*, pp. 152–167. Springer International Publishing, 2014.

[4] A. Dal Palù, A. Dovier, A. Formisano and E. Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, **27**, 293–316, 2014.

[5] M. Deveci, K. Kaya, B. Uçar and Ü. V. Çatalyürek. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. In *Euro-Par 2013 Parallel Processing*. Lecture Notes in Computer Science, vol. 8097, F. Wolf and B. Mohr., eds, pp. 850–861. Springer, 2013.

[6] A. Dovier, A. Formisano, G. Gupta, M. V. Hermenegildo, E. Pontelli and R. Rocha. Parallel logic programming: a sequel. *Theory and Practice of Logic Programming*, **22**, 905–973, 2022.

[7] A. Dovier, A. Formisano and E. Pontelli. Parallel answer set programming. In *Handbook of Parallel Constraint Reasoning*, pp. 237–282. Springer International Publishing, 2018.

[8] A. Dovier, A. Formisano, E. Pontelli and F. Vella. A GPU implementation of the ASP computation. In *Practical Aspects of Declarative Languages*, pp. 30–47. Springer International Publishing, 2016.

[9] A. Dovier, A. Formisano and F. Vella. GPU-based parallelism for ASP-solving. In *Declarative Programming and Knowledge Management, DECLARE 2019, Cottbus, Germany, September 9–12, 2019, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 12057, P. Hofstedt, S. Abreu, U. John, H. Kuchen and D. Seipel., eds, pp. 3–23. Springer, 2019.

[10] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (Swat 1971)*, pp. 129–131. IEEE, 1971.

[11] L. K. Fleischer, B. Hendrickson and A. Pinar. On identifying strongly connected components in parallel. In *Lecture Notes in Computer Science*, pp. 505–511. Springer, Berlin Heidelberg, 2000.

[12] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, **8**, 399–404, 1956.

[13] I. P. Gent, I. Miguel and P. Nightingale. Generalised arc consistency for the AllDifferent constraint: an empirical survey. *Artificial Intelligence*, **172**, 1973–2000, 2008.

[14] I. P. Gent, I. Miguel, P. Nightingale, C. Mccreeh, P. Prosser, N. C. A. Moore and C. Unsworth. A

review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, **18**, 725–758, 2018.
[15] I. P. Gent and T. Walsh. CSPlib: a benchmark library for constraints. In *Principles and Practice of Constraint Programming—CP'99*, pp. 480–481. Springer, Berlin Heidelberg, 1999.
[16] R. Gentzel, L. Michel and W.-J. van Hoeve. HADDOCK: a language and architecture for decision diagram compilation. In *Lecture Notes in Computer Science*, pp. 531–547. Springer International Publishing, 2020.
[17] J. E. Hopcroft and R. M. Karp. A n5^/2 algorithm for maximum matchings in bipartite graphs. In *12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13–15, 1971*, pp. 122–125. IEEE Computer Society, 1971.
[18] P. M. Jensen, N. Jeppesen, A. B. Dahl and V. A. Dahl. *Review of Serial and Parallel Min-Cut/Max-Flow Algorithms for Computer Vision*, vol. 45, pp. 2310–2329, 2022.
[19] M. Karppa and P. Kaski. Engineering boolean matrix multiplication for multiple-accelerator shared-memory architectures. *CoRR*, abs/1909.01554, 2019.
[20] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the EUROGRAPHICS/ACM SIGGRAPH Conference on Graphics Hardware 2008, Sarajevo, Bosnia and Herzegovina, 2008*, D. Luebke and J. Owens., eds, pp. 47–55. Eurographics Association, 2008.
[21] K. Kuchcinski and R. Szymanek. *Jacop*, 2022.
[22] A. Li and S. Simon. Accelerating binarized neural networks via bit-tensor-cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems*, **32**, 1878–1891, 2021.
[23] G. Li, Z. Zhu, Z. Cong and F. Yang. Efficient decomposition of strongly connected components on GPUs. *Journal of Systems Architecture*, **60**, 1–10, 2014.
[24] A. López-Ortiz, C.-G. Quimper, J. Tromp and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9–15, 2003*, G. Gottlob and T. Walsh., eds, pp. 245–250. Morgan Kaufmann, 2003.
[25] K. Matsumoto, N. Nakasato and S. G. Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In *In 2011 IEEE International Conference on High Performance Computing and Communications*, pp. 145–152. IEEE, 2011.
[26] L. Michel, P. Schaus and P. Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Mathematical Programming Computation*, **13**, 133–184, 2021.
[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack. MiniZinc: towards a standard CP modelling language. In *Principles and Practice of Constraint Programming—CP 2007*. Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer, 2007.
[28] L. Perron and V. Furnon. *Or-Tools*, 2022.
[29] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1), AAAI '94*, pp. 362–367. American Association for Artificial Intelligence, USA, 1994.
[30] G. Reinelt. TSPLIB—A traveling salesman problem library. *ORSA Journal on Computing*, **3**, 376–384, 1991.
[31] F. Rossi, P. van Beek and T. Walsh., eds. *Handbook of Constraint Programming*. Elsevier, 2006.
[32] P. J. Stuckey, R. Becket and J. Fischer. Philosophy of the MiniZinc challenge. *Constraints*, **15**, 307–316, 2010.
[33] W. Suijlen, F. de Framond, A. Lallouet and A. Petitet. A parallel algorithm for GAC filtering

of the alldifferent constraint. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 390–407. Springer International Publishing, 2022.
[34] F. Tardivo. *Fzn-Minicpp*, 2022.
[35] F. Tardivo. *Minicpp-Benchmarks*, 2022.
[36] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, **1**, 146–160, 1972.
[37] Team Gecode *Gecode*, 2022.
[38] Team MiniZinc *The MiniZinc Challenge*, 2022.
[39] Team MiniZinc *The MiniZinc Handbook*, 2022.
[40] Team NVIDIA *Cublas*, 2022.
[41] Team NVIDIA *CUDA Toolkit Documentation*, 2022.
[42] H.-N. Tran and E. Cambria. A survey of graph processing on graphics processing units. *The Journal of Supercomputing*, **74**, 2086–2115, 2018.
[43] W. J. van Hoeve. *The Alldifferent Constraint: A Survey*, 2001.
[44] C. N. Vasconcelos and B. Rosenhahn. Bipartite graph matching computation on GPU. In *Lecture Notes in Computer Science*, pp. 42–55. Springer, Berlin Heidelberg, 2009.
[45] G. Venkataraman, S. Sahni and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *ACM Journal of Experimental Algorithmics*, **8**, 2003.
[46] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, **9**, 11–12, 1962.
[47] W. Jiadong, Z. He and B. Hong. Efficient CUDA algorithms for the maximum network flow problem. In *GPU Computing Gems Jade Edition*, pp. 55–66. Elsevier, 2012.
[48] X. Zhang, Q. Li and W. Zhang. A fast algorithm for generalized arc consistency of the alldifferent constraint. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2018.

Received 1 May 2023