# Constraints Propagation on GPU: A Case Study for AllDifferent*

Fabio Tardivo[1,*], Agostino Dovier[2,4], Andrea Formisano[2,4], Laurent Michel[3] and Enrico Pontelli[1]

[1]*New Mexico State University, Las Cruces, NM, USA*
[2]*CLPLab - DMIF - Università di Udine, Udine, Italy*
[3]*Department of Computer Science and Engineering, University of Connecticut, Storrs, CT, USA*
[4]*GNCS-INdAM, Gruppo Nazionale per il Calcolo Scientifico, Roma, Italy*

**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. Motivated by the benefits that GPUs offer to other branches of AI, this paper investigates the use of GPUs to accelerate filtering and propagation. In particular, we present an efficient parallelization of the *AllDifferent* constraint on GPU; we analyze different design and implementation choices and evaluates the performance of the resulting system on medium to large instances of the Travelling Salesman Problem with encouraging results.

**Keywords**
Constraint Propagation, AllDifferent, Parallelism, GPU computing

## 1. Introduction

*Constraint programming (CP)* is a declarative paradigm to modeling and solving combinatorial problems. Users model a problem using a set of variables, each of them 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 [1], an annual competition among solvers, the constraint programming language MiniZinc [2] has emerged as a de-facto standard modeling language for the CP community.

Traditional constraint solvers work by alternating two stages: non-deterministic variables assignment and constraint propagation. Once a value has been assigned to a variable, constraint

*CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy*

*Research partially supported by Fondazione Friuli/Università di Udine project on Artificial Intelligence for Human Robot Collaboration and by projects INDAM GNCS-2020 *NoRMA* and INDAM GNCS-2022 InSANE (CUP_E55F22000270001). Laurent Michel was partially supported by the Synchrony Chair in Cybersecurity.

*Corresponding author.

ftardivo@nmsu.edu (F. Tardivo); agostino.dovier@uniud.it (A. Dovier); andrea.formisano@uniud.it (A. Formisano); ldm@uconn.edu (L. Michel); epontell@nmsu.edu (E. Pontelli)
0000-0003-3328-2174 (F. Tardivo); 0000-0003-2052-8593 (A. Dovier); 0000-0002-6755-9314 (A. Formisano); 0000-0001-7230-7130 (L. Michel); 0000-0002-7753-1737 (E. Pontelli)

propagation eliminates all 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 propagation of global constraints is the subject of many studies and optimizations [3].

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 [4]. The most popular approach is the one by Régin [5].

Recently, branches of AI like Machine Learning have obtained huge benefits from the use of GPUs to speed up their tasks. Relatively more limited work has been done in exploring the use of GPUs for logic-based AI, e.g., [6] for SAT, [7] for ASP, and [8] for CP. For additional references, please see the recent surveys on parallelism in constraint and logic programming [9, 10, 11].

In this paper, we present a GPU-accelerated propagator for the *AllDifferent* constraint and its implementation within a simple constraint solver compatible with the MiniZinc language. Our contributions are: 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 [12]; and a comparison between the standard propagator and our GPU-accelerated version. Results on medium to large instances of the Travelling Salesman 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* and the use of GPU for general computation. Section 3 describes the parallelization process, the implementation details of the final algorithm, and the integration in a constraint solver. In Section 4, we describe the benchmarks used to test the GPU-accelerated propagators and their 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 $\mathcal{V} = \{V_1, \dots, V_n\}$ is a finite set of variables, $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_n\}$ 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 $var(c) \subseteq \mathcal{V}$. 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., $V_3 < 2V_5$). 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: **(1)** constraint propagation and **(2)** 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 a solution. The choice of the variable is typically fast compared to the cost of constraint propagation.

During constraint propagation, constraints are placed in a queue for processing — i.e., to filter the domains of the variables involved in the constraints. A general property one could consider during propagation is *hyper-arc consistency* [3]. 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:

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

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 complexity of naive algorithms for obtaining HAC is exponential in $m$. It is also 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 $A < B, A = X + C, C = 2Y + 3U, B = 4V + Z$. However, this type of translations may lead to a reduced filtering capability during constraint propagation, since the HAC property is guaranteed only for the simple constraints. In the example above, a built-in constraint capable of handling sums of linear terms can be more efficient. We can first rewrite the constraint as $X + 2Y + 3U - 4V - Z < 0$ and then use the scalar product to write it equivalently as: $[X, Y, U, V, Z] \cdot [1, 2, 3, -4, -1] < 0$.

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

## 2.2. 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 *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 (see, e.g., [4]).

Régin's well-known algorithm [5] 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 vertices $N_1$ and $N_2$ and $E \subseteq \{\{u, v\} \,:\, u \in N_1, v \in N_2\}$ 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 vertex. A maximum matching is a maximum cardinality matching. The Hopcroft–Karp algorithm [13] 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 $O(n \cdot |E|)$ [14], where $n = |N_1| + |N_2|$.

A *directed graph (digraph)* $G(N, A)$ pairs a set of vertices $N$ 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 vertices 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 vertices of the digraph. Tarjan's algorithm can be used to efficiently compute the SCCs of any digraph in $O(|N| + |A|)$ time [15].

Before discussing the GPU-based implementation, it is perhaps useful to briefly review the steps adopted in the propagation for the *AllDifferent* constraint. In particular, consider the constraint applied to $n$ variables, i.e.

$$\textit{AllDifferent}(x_1, \ldots, x_n)$$

Consider the following preliminary definitions. Given a bipartite graph $G(N_1 \cup N_2, E)$ and a matching $M$ of $G$, the *residual graph* 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 $E$

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

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

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

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 *AllDifferent*$(x_1, \ldots, x_n)$. The algorithm constructs a bipartite graph $G = (N_1 \cup N_2, E)$ where:

- $N_1 = \{x_1, \ldots, x_n\}$,

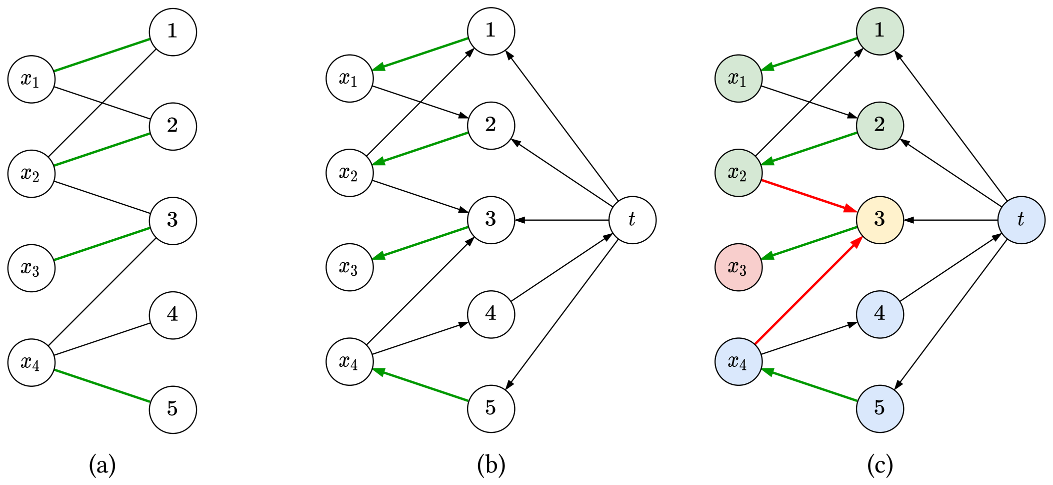

**Figure 1:** Quick overview of Regin's algorithm on $x_1, x_2, x_3, x_4$ where $D_1 = \{1, 2\}$, $D_2 = \{1, 2, 3\}$, $D_3 = \{3\}$, $D_4 = \{3, 4, 5\}$. In (a) is highlighted the maximum match of step 1. In (b) is pictured the residual graph of step 3. In (c) are highlighted the SCCs of step 4, each with a different color, and in red the arcs considered in step 5.

- $N_2 = \bigcup_{i \in 1..n} \mathcal{D}_i$, 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 is 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 has complexity $O(1)$ since is just a check. Step 3 has complexity $O(|N_1|+|N_2|+|E|)$ as described below. 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 [16]. Our implementation mitigates the cost of step 1 using an incremental approach as is traditionally done.

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 analyzing a single maximum matching [17].

## 2.3. GPGPU with CUDA

General-Purpose computing on Graphics Processing Units (GPGPU) is the use of a Graphics Processing Unit (GPU) to speed up computations traditionally handled by the Central Processing

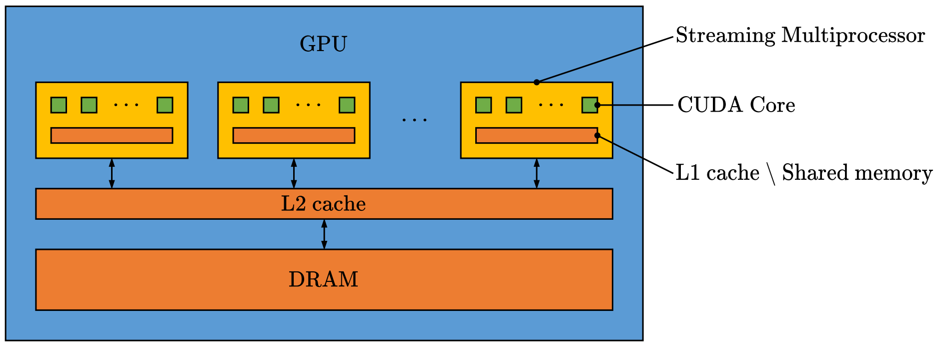

**Figure 2:** Simplified GPU architecture

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 [18]. CUDA has been successfully used to accelerate computations in a variety of domains, such as physics, bioinformatics, and machine learning [19].

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),* (variant of SIMD) 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 Fig. 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.

The first step in this process consists of selecting an existing constraint solver suitable to host a GPU-enabled *AllDifferent*. Initially, we had a look at the fastest solvers compatible with the MiniZinc language and, thus, we selected OR-Tools [20], JaCoP [21], and Gecode [22] respectively Gold and Silver medal of the last MiniZinc challenge [23]. We realized soon that their efficiency is also due to several optimizations that makes the code difficult to modify.

Our choice converged on *MiniCP* [12], 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* [24], a C++ implementation of MiniCP, suitable to host C++ CUDA kernels. With minor optimizations, MiniCPP provides a performance that is comparable to that of other solvers (e.g., JaCoP) for several classes of problems.

To use MiniCPP as a base solver, it was necessary to implement the support for the MiniZinc language, using the FlatZinc skeleton parser provided by Gecode, and implement all the standard integer and Boolean constraints. Moreover, we created the necessary definitions to allow the MiniZinc compiler to recognize the MiniCPP's native *AllDifferent* as a global constraint, thus avoiding its decomposition in a collection of binary constraints.

The following subsection describes the implementation of the parallel version of the *AllDifferent* filtering algorithm on GPU and how it has been integrated in the solver. We enabled the FlatZinc parser to recognize the annotation $::\mathrm{gpu}$ to instruct the solver to propagate the annotated constraint using the GPU algorithm in place of the standard CPU algorithm.

## 3.1. Parallelization

The key components of the filtering algorithm for *AllDifferent* propagation are *(1)* the computation of a Maximum Matching in a bipartite graph (MM) and *(2)* the computation of the Strongly Connected Components (SCC) of a directed graph (see Section 2.2).

Before discussing the parallelization process we introduce some preliminary definitions. *Breadth-First Search* (BFS) is a graph traversal algorithm that explores the graph's vertices in the order of their distance from a source vertex $s$. Given a graph $G = (V, E)$ and a source vertex $s \in V$, BFS systematically traverses the edges in such a way that all vertices at distance $k$ from $s$ are discovered before any vertices at distance $k + 1$. By BFS is possible to find all the $v \in V$ reachable from $s$. In case of digraph, the *forward reachability* of a vertex $s$ is defined as the set of nodes $F$ such that exists a path from $s$ to any $v \in F$. Similarly the *backward reachability* of a vertex $s$ is the set of nodes $B$ such that for any $v \in B$ exists a path from $v$ to $s$. Forward / backward reachability can be expressed as a binary matrix where the element at coordinates $(i, j)$ is 1 if and only if vertex $i$ reaches / is reachable from vertex $j$.

**Computing a Maximum Matching on an GPU.** There are several approaches to solving such problems on GPU. For maximum matching there are implementations based on the auction [25], push-relabel [26], and the BFS [27] algorithms.
The auction algorithm works as an auction where persons compete for objects by raising their prices. It alternates bidding and assignment phases until all the person have been assigned to an object. The bidding and assignment phases are offloaded on the GPU, where bids and assignments are computed in parallel.
The push-relabel approach solves the maximum matching reducing it to a flow problem. The

initial bipartite graph $G = (N_1 \cup N_2, E)$ is modified by adding two nodes $s$ and $t$, the first connected to all the $n \in N_1$ and the second reached by all the $n \in N_2$. The resulting graph is seen as a flow network such that:

- through an edge can pass 0 or 1 unit of flow
- the node $s$ produces $N_1$ units of flow and the node $t$ can receive $N_1$ units of flow
- the sum of the ingoing flow in a node must be equal to the sum of the outgoing flows

The problem is to find which edges to use to move the maximum amount of flow from $s$ to $t$. The push-relabel algorithm 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.
The BFS algorithms are based on the Hopcroft-Karp algorithm and make use of a GPU-accelerated parallel BFS to find the augmenting paths in place of the standard Depth First Search.

We started our study with a push-relabel algorithm based on [26]. We choose it because its more studied than the other approaches and it does not assume the existence of a matching. Despite our optimization efforts, offloading the calculation of MM on the GPU does not pay off: the algorithm does not scale [28] and is slower than on the CPU since the solvers can quickly calculate MM incrementally [16]. Moreover, for large instances, the cost of MM is negligible compared with the cost of SCCs. In the end, the computation of the maximum matching was kept on the CPU.

**Strongly Connected Components on a GPU.** For SCCs, the majority of the GPU implementations [29, 30] ultimately make use of forward/backward reachability [31]. The literature about SCCs on GPU considers enormous sparse graphs with millions of nodes. The trend is to use, as a fundamental step, a parallel BFS to calculate forward/backward reachability. This scenario does not fit our context where (1) a major constraint leads to a dense graph of hundreds/thousands of nodes and (2) we aim for low latency and BFS notoriously suffers from load imbalance [32]. The first observation direct us to calculate 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. The identifier of the SCC of the node $i$ is the minimum $j$ such that $C(i, j) = 1$.

The second observation made us look for alternatives to BFS. Let $G(V, E)$ be a graph, the standard algorithms to calculate the reachability matrix are:

- Matrix multiplication, with complexity $O(|V|^{2.8} log_2(|V|))$ [33].
- Warshall algorithm, with complexity $O(|V|^3)$ [34].

We explored the parallelization of both of these approaches (see Section 3.2) and chose Warshall algorithm.

## 3.2. Implementation Details

**Data transfer.** To begin with, we had to decide which data to transfer to the GPU. We opted to transfer the residual graph as a binary adjacency matrix. Preliminary tests showed that it is better than transferring only the domains and the match necessary to generate the graph's adjacency matrix. This is because, in our case, most of the transmission's cost is in the initialization phase than in the actual data transfer. Moreover, the build of the adjacency matrix requires many sparse memory accesses, and the CPU is sensibly faster than the GPU for such tasks.

**Matrix representation.** We choose to represent the adjacency matrix as a bit matrix for its several benefits. It enables the use of bitwise operations, it can be initialized by dumping the domains' internal representation, and it minimizes the amount of data to transfer. In preliminary tests, we also explored the use of other representations suited for hardware accelerated matrix multiplication (i.e., Tensor Core), and for sparse matrices. In both cases, the matrix multiplication performs worst than an ad-hoc matrix multiplication encoded by us using bitwise operations. In the first case, the penalty come from the more general algorithm and dealing with floating-point numbers, while in the second case it comes from the fast increasing density of the matrices and dealing with integer numbers.

**Matrix multiplication.** After the choice of the representation, we focused on matrix multiplication. We tested two of the state-of-the-art GPU linear algebra libraries, namely cuBLAS [35] and cuBool [36]. The performances were, at best, matching the classic algorithm on CPU.

Due to these poor results, we focused on binary matrix multiplication on GPU. There are efficient implementations for multi-GPUs matrix multiplication [37] and to speedup Binarized Neural Network [38], but their code is tailored to their use case. Thus, we decided to implement binary matrix multiplication from scratch. We used a few known techniques in the design:

- tiled matrix multiplication to optimize cache usage
- data arrangement to optimize memory access, and
- bitwise operation to speed up the computation

The result was a simple but efficient binary matrix multiplication, with performances slightly worst than [37] for squared matrices of a few thousand rows.

**Warshall.** Then, we focused on the Warshall algorithm. Despite sharing a similar nested loops structure with the matrix multiplication, its nesting order is stricter and does not allows the same optimizations. The majority of the GPU implementations that use an adjacency matrix representation [39, 40] are based on a blocked version of the Warshall algorithm from [41]. Such an algorithm was developed to maximize the CPU's cache utilization, and it is particularly efficient to exploit the GPU's shared memory. As the Warshall algorithm, this blocked version starts from an adjacency matrix of size $n \times n$ and iteratively updates it to obtain the reachability matrix. It begins by dividing the matrix in tiles of size $t \times t$ and then performing $n/t$ steps. Each step $s$ is made of three phases (see Figure 3), each one updating a specific set of tiles according to their dependency:

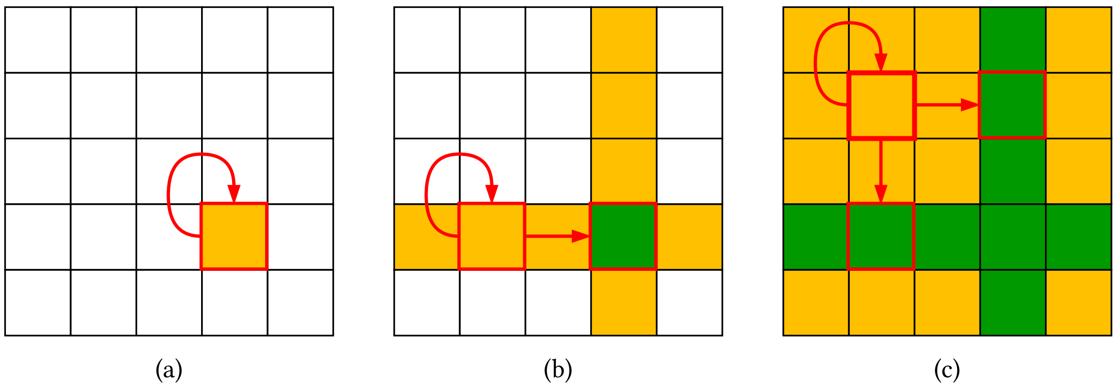

**Figure 3:** Illustration of the 4-th step of the blocked Warshall algorithm on a matrix divided in 25 tiles. The tiles updated in each phase are highlighted in yellow, while the tiles already processed are colored in green. Tiles dependencies are illustrated in with red arrows.

**Phase 1** : This phase consider the $s$-th tile of the main diagonal, named $D$, and update it using the equation $D(i,j) = D(i,j) \vee (D(i,k)) \wedge D(k,j))$ for $0 \leq k < t$.

**Phase 2** : This phase consider the tiles of the $s$-th row and $s$-th column excluding the $D$ tile. A generic tile of the $s$-th row, named R, is updated using the equation $R(i,j) = R(i,j) \vee (D(i,k)) \wedge R(k,j))$. A tile of the $s$-th column, named C, is updated using the equation $C(i,j) = C(i,j) \vee (C(i,k)) \wedge D(k,j))$.

**Phase 3** : This phase consider all the remaining tiles. The equation to update a generic tile $T$ in position $(r,c)$ is $T(i,j) = T(i,k) \vee (R(i,k)) \wedge C(k,j))$ where $R$ is the tile in position $(r,s)$ an $C$ is the tile in position $(s,c)$.

Each tile within a phase can be updated independently, allowing parallelization of phases 2 and 3. These independent updates map well into the GPU computational model, where each tile of size $t \times t$ is managed by a block of $t$ threads. In this way, the $i$-th thread updates the $i$-th row of the tile considering every $0 \leq k < t$. We paid particular attention to optimizing phase 3 since it involves most of the tiles. Unlike phase 2, where the update of the $i$-th row $R(i,*)$ depends on the $k$-th row $R(k,*)$, in phase 3, the update of the $i$-th row $T(i,*)$ does not depends on other lines of the tile. This fact makes possible to avoid threads synchronization, sensibly reducing the computational time. Preliminary benchmarks show that the most convenient tile size is $t = 128$. Such dimension allows reading each row in one memory access as one `uint4` (32*4 bit) and manipulating it by two 64-bit operations. Preliminary comparisons with the matrix multiplication approach reported similar performance. However, in such tests, the matrix multiplication approach performed only a few of the $log_2(|V|)$ iterations. In the end, we chose as our approach the blocked Warshall algorithm because its runs in a slightly less amount of time as a good case of the matrix multiplication approach.

Initial tests highlight that offloading the computation on GPU is not convenient when the bipartite graphs are small (i.e. $|V| \leq 200$). For these cases we created a single procedure that performs steps 1, 4 without performing steps 2, 3.

## 4. Results and Analysis

It is expected that a GPU implementation would provide benefits on instances that are sufficiently large, as the setup overhead would otherwise overshadow the benefits of the parallel execution. We chose as benchmark the Travelling Salesman Problem (TSP) because it can be simply modeled using a Circuit constraint, that internally makes use of the *AllDifferent* constraint, there is an established set of large benchmarks [42], and it is a fundamental problem for routing applications.

We select about 80 instances from the TSPLib with 100 to 10,000 cities and convert them into the MiniZinc format. We solve them using the MiniCP's native *AllDifferent* propagator as well as our GPU-accelerated propagator. All benchmarks have a timeout of 10 minutes and are executed on a system equipped with an Intel Core i7-10700K, 32GB of DDR4 RAM, and GeForce RTX 3080 running Ubuntu 21.04 and CUDA 11.4.

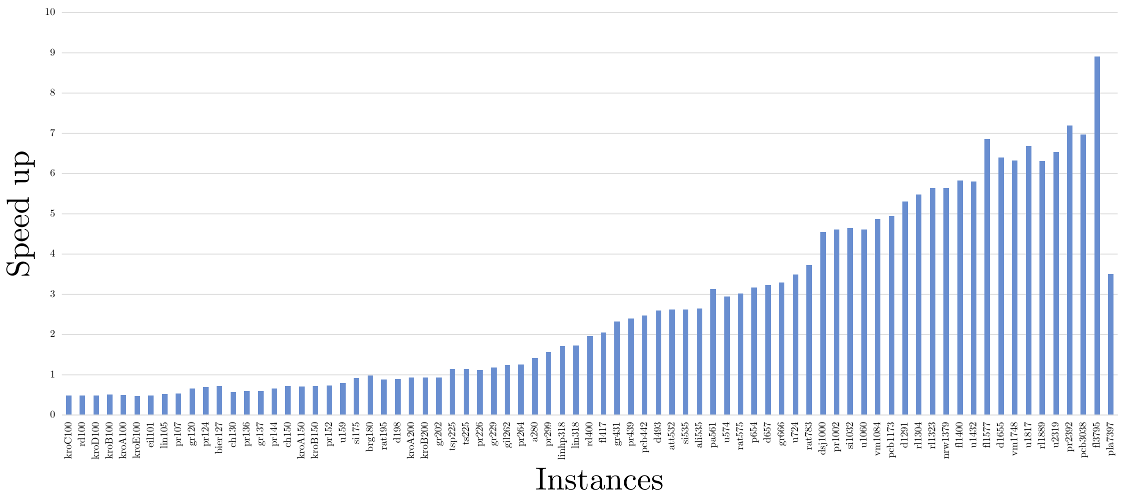

**Figure 4:** Results of the TSPLib benchmarks. On the horizontal axis there are the instances sorted by increasing size. The vertical values indicate the speedup of the GPU in terms of explored nodes.

Figure 4 illustrates the results of the benchmarks. The speedup is calculated as ratio between the GPU search speed and the CPU search speed. The search speed is the number of visited nodes over the search time. As expected from preliminary tests, the benefits of our approach start to be visible when a constraint involves several hundreds of variables. The plot shows an increasing speedup as the size of the instance increases, except for the largest instance. Such behavior is due to the large time between two GPU-accelerated propagations, reducing the opportunities to speed up the computation. To verify such explanation we increased the timeout to 30 minutes and obtained a speedup of approximately 7 times.

## 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 medium to large instances of the Travelling Salesman Problem and obtained speedups up to 7 times in terms of explored nodes. Unlike other parallel approaches, our method is immediately usable since modern PCs often have a GPU.

There are many ways to extend and improve this work: implementing on GPU propagators for other global constraints, exploring their usage in Constraint-Based Local Search, etc. Our next step will be focusing on problems containing multiple *AllDifferent* constraints. In such cases it is possible to process the *AllDifferent* constraints in parallel. This promises significant speedups even for small to medium problems, where the GPU propagation is still less efficient than the CPU version.

## References

[1] P. J. Stuckey, R. Becket, J. Fischer, Philosophy of the MiniZinc challenge, Constraints 15 (2010) 307–316. doi:10.1007/s10601-010-9093-0.

[2] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: Principles and Practice of Constraint Programming – CP 2007, volume 4741 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 529–543. doi:10.1007/978-3-540-74970-7_38.

[3] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006. doi:10.1016/s1574-6526(06)x8001-x.

[4] W. J. van Hoeve, The alldifferent constraint: A survey, 2001. URL: https://arxiv.org/abs/cs/0105015.

[5] 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, American Association for Artificial Intelligence, USA, 1994, p. 362–367.

[6] A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli, CUD@SAT: SAT solving on GPUs, Journal of Experimental & Theoretical Artificial Intelligence 27 (2014) 293–316. doi:10.1080/0952813x.2014.954274.

[7] A. Dovier, A. Formisano, E. Pontelli, F. Vella, A GPU implementation of the ASP computation, in: Practical Aspects of Declarative Languages, Springer International Publishing, 2016, pp. 30–47. doi:10.1007/978-3-319-28228-2_3.

[8] F. Campeotto, A. D. Palù, A. Dovier, F. Fioretto, E. Pontelli, Exploring the use of GPUs in constraint solving, in: Practical Aspects of Declarative Languages, Springer International Publishing, 2014, pp. 152–167. doi:10.1007/978-3-319-04132-2_11.

[9] I. P. Gent, I. Miguel, P. Nightingale, C. Mccreeh, P. Prosser, N. C. A. Moore, C. Unsworth, A review of literature on parallel constraint solving, Theory and Practice of Logic Programming 18 (2018) 725–758. doi:10.1017/s1471068418000340.

[10] A. Dovier, A. Formisano, E. Pontelli, Parallel answer set programming, in: Handbook of Parallel Constraint Reasoning, Springer International Publishing, 2018, pp. 237–282. doi:10.1007/978-3-319-63516-3_7.

[11] A. Dovier, A. Formisano, G. Gupta, M. V. Hermenegildo, E. Pontelli, R. Rocha, Parallel

logic programming: A sequel, Theory and Practice of Logic Programming (2022) 1–69. doi:10.1017/s1471068422000059.

[12] L. Michel, P. Schaus, P. V. Hentenryck, MiniCP: a lightweight solver for constraint programming, Mathematical Programming Computation 13 (2021) 133–184. doi:10.1007/s12532-020-00190-7.

[13] J. E. Hopcroft, R. M. Karp, A nˆ5/2 algorithm for maximum matchings in bipartite graphs, in: 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971, IEEE Computer Society, 1971, pp. 122–125. doi:10.1109/SWAT.1971.1.

[14] L. R. Ford, D. R. Fulkerson, Maximal flow through a network, Canadian Journal of Mathematics 8 (1956) 399–404. doi:10.4153/cjm-1956-045-5.

[15] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (1972) 146–160. doi:10.1137/0201010.

[16] I. P. Gent, I. Miguel, P. Nightingale, Generalised arc consistency for the AllDifferent constraint: An empirical survey, Artificial Intelligence 172 (2008) 1973–2000. doi:10.1016/j.artint.2008.10.006.

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

[18] NVIDIA Team, CUDA toolkit documentation, (n.d.). URL: https://developer.nvidia.com/cuda-zone.

[19] NVIDIA Team, GPU accelerated applications, (n.d.). URL: https://www.nvidia.com/en-us/gpu-accelerated-applications.

[20] L. Perron, V. Furnon, OR-Tools, (n.d.). URL: https://developers.google.com/optimization/.

[21] K. Kuchcinski, R. Szymanek, JaCoP, (n.d.). URL: https://github.com/radsz/jacop.

[22] Gecode Team, GECODE, (n.d.). URL: https://www.gecode.org.

[23] MiniZinc Team, The MiniZinc challenge, (n.d.). URL: https://www.minizinc.org/challenge.html.

[24] R. Gentzel, L. Michel, W.-J. van Hoeve, HADDOCK: A language and architecture for decision diagram compilation, in: Lecture Notes in Computer Science, Springer International Publishing, 2020, pp. 531–547. doi:10.1007/978-3-030-58475-7_31.

[25] C. N. Vasconcelos, B. Rosenhahn, Bipartite graph matching computation on GPU, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 42–55. doi:10.1007/978-3-642-03641-5_4.

[26] J. Wu, Z. He, B. Hong, Efficient CUDA algorithms for the maximum network flow problem, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 55–66. doi:10.1016/b978-0-12-385963-1.00005-8.

[27] M. Deveci, K. Kaya, B. Uçar, Ü. V. Çatalyürek, GPU accelerated maximum cardinality matching algorithms for bipartite graphs, in: F. Wolf, B. Mohr, D. an Mey (Eds.), Euro-Par 2013 Parallel Processing, volume 8097 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 850–861. doi:10.1007/978-3-642-40047-6_84.

[28] P. M. Jensen, N. Jeppesen, A. B. Dahl, V. A. Dahl, Review of serial and parallel min-cut/max-flow algorithms for computer vision, 2022. URL: https://arxiv.org/abs/2202.00418.

[29] J. Barnat, P. Bauch, L. Brim, M. Češka, Computing strongly connected components in parallel on CUDA, in: 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE, 2011, pp. 544–555. doi:10.1109/ipdps.2011.59.

[30] G. Li, Z. Zhu, Z. Cong, F. Yang, Efficient decomposition of strongly connected components on GPUs, Journal of Systems Architecture 60 (2014) 1–10. doi:10.1016/j.sysarc.2013.10.014.

[31] L. K. Fleischer, B. Hendrickson, A. Pınar, On identifying strongly connected components in parallel, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 505–511. doi:10.1007/3-540-45591-4_68.

[32] H.-N. Tran, E. Cambria, A survey of graph processing on graphics processing units, The Journal of Supercomputing 74 (2018) 2086–2115. doi:10.1007/s11227-017-2225-1.

[33] M. J. Fischer, A. R. Meyer, Boolean matrix multiplication and transitive closure, in: 12th Annual Symposium on Switching and Automata Theory (swat 1971), IEEE, 1971, pp. 129–131. doi:10.1109/swat.1971.4.

[34] S. Warshall, A theorem on Boolean matrices, Journal of the ACM 9 (1962) 11–12. doi:10.1145/321105.321107.

[35] NVIDIA Team, cuBLAS, (n.d.). URL: https://developer.nvidia.com/cublas.

[36] E. Orachyov, P. Alimov, S. Grigorev, cuBool: sparse Boolean linear algebra for Nvidia Cuda, (n.d.). URL: https://github.com/JetBrains-Research/cuBool.

[37] M. Karppa, P. Kaski, Engineering boolean matrix multiplication for multiple-accelerator shared-memory architectures, CoRR abs/1909.01554 (2019). URL: http://arxiv.org/abs/1909.01554.

[38] A. Li, S. Su, Accelerating binarized neural networks via bit-tensor-cores in Turing GPUs, CoRR abs/2006.16578 (2020). URL: https://arxiv.org/abs/2006.16578.

[39] G. J. Katz, J. T. K. Jr., All-pairs shortest-paths for large graphs on the GPU, in: D. Luebke, J. Owens (Eds.), Proceedings of the EUROGRAPHICS/ACM SIGGRAPH Conference on Graphics Hardware 2008, Sarajevo, Bosnia and Herzegovina, 2008, Eurographics Association, 2008, pp. 47–55. doi:10.2312/EGGH/EGGH08/047-055.

[40] K. Matsumoto, N. Nakasato, S. G. Sedukhin, Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system, in: 2011 IEEE International Conference on High Performance Computing and Communications, IEEE, 2011, pp. 145–152. doi:10.1109/hpcc.2011.28.

[41] G. Venkataraman, S. Sahni, S. Mukhopadhyaya, A blocked all-pairs shortest-paths algorithm, ACM Journal of Experimental Algorithmics 8 (2003). doi:10.1145/996546.996553.

[42] G. Reinelt, TSPLIB—a traveling salesman problem library, ORSA Journal on Computing 3 (1991) 376–384. doi:10.1287/ijoc.3.4.376.