GPU-Accelerated Propagation for the Stable Marriage Constraint*

Stefano Travasci[1], Fabio Tardivo[2] and Andrea Formisano[1,3,*]

[1]University of Udine, DMIF, Italy
[2]New Mexico State University, Dept. of Computer Science, Las Cruces, NM, USA
[3]Gruppo Nazionale per il Calcolo Scientifico – INdAM, Italy

Abstract

The Stable Marriage Problem (SMP) consists of finding a *stable* matching between two equally sized disjoint sets, typically referred to as *men* and *women*, based on individual preference lists. A matching is stable if no man and woman prefer each other over their assigned partners. The Gale–Shapley algorithm is the classical polynomial-time solution to this problem. In Constraint Programming (CP), the Stable Marriage Constraint (SMC) encapsulates this problem as a global constraint, with a consistency-enforcing propagator derived from an extended version of the Gale–Shapley algorithm. In this paper, we present a GPU-accelerated propagator for the SMC and its integration into a CP solver. Experimental results against the sequential version demonstrate the potential of GPU acceleration in handling large instances of the stable marriage constraint.

Keywords

Stable Marriage, Global Constraint, Constraint Programming, Parallel Programming, GPU Computing

1. Introduction

The *Stable Marriage Problem (SMP)*, also known as the *stable matching problem*, is the problem of finding a stable matching (i.e., a bijection) between two sets of equal size, given an ordered list of preferences for each element of the sets [1]. These sets are usually called men and women, thus the name *stable marriage*. Each person has a strictly ordered preference list, ranking all individuals of the opposite sex from most preferred to least preferred. A matching is *unstable* if it contains two couples (m_i, w_i) and (m_j, w_j) such that m_i prefers w_j to w_i, and w_j prefers m_i to m_j. In such a situation, m_i and w_j break their current engagements to *elope* together [2]. Couples like (m_i, w_j) are said to be *blocking pairs*. A matching is *stable* if it contains no blocking pairs. If a man m and a woman w are engaged in some stable matching, the pair (m, w) is said to be a *stable pair*. It is customary to say that an instance has size n if it involves n men and n women. Note that such an instance actually requires $2n^2$ elements to represent all preference lists. Table 1 shows an example of size $n = 5$.

The Gale–Shapley algorithm (Section 2) is the most prominent algorithm for solving SMP. It can be shown that this algorithm always terminates in time $O(n^2)$ producing a stable matching. Consequently, each instance of SMP admits at least one solution [1].

Several variations of SMP exist. In the SMP *with indifference*, the preference lists are not strictly ordered, allowing ties between elements [3]. The preference lists may be incomplete [1], meaning that some partners are deemed unacceptable and that some couples are not admissible. Variations also exist in which each element of one set may be matched with more members of the other set, i.e., the solution sought for is a *many-to-many* relation. A similar problem is the *stable roommates problem*, in which there is only one set and the preferences are referred to the other members of the such [4]. Generalizations have been proposed that involve three sets of elements [5, 6]. We refer the reader to [7]

CILC 2025: 40th Italian Conference on Computational Logic, June 25–27, 2025, Alghero, Italy

* Research partially supported by Interdepartment Project on AI (Strategic Plan of UniUD–22-25), and by Unione europea-Next Generation EU, missione 4 componente 2, project MaPSART "Future Artificial Intelligence (FAIR)"

* Corresponding author.

travasci.stefano@spes.uniud.it (S. Travasci); ftardivo@nmsu.edu (F. Tardivo); andrea.formisano@uniud.it (A. Formisano)
0000-0001-5693-6033 (S. Travasci); 0000-0003-3328-2174 (F. Tardivo); 0000-0002-6755-9314 (A. Formisano)

m_0	w_3	w_4	w_1	w_2	w_0
m_1	w_0	w_1	w_3	w_2	w_4
m_2	w_0	w_4	w_3	w_2	w_1
m_3	w_2	w_1	w_4	w_3	w_0
m_4	w_0	w_1	w_4	w_2	w_3

(a) Men's preference lists

w_0	m_1	m_3	m_4	m_0	m_2
w_1	m_2	m_3	m_0	m_4	m_1
w_2	m_0	m_2	m_1	m_4	m_3
w_3	m_0	m_3	m_2	m_1	m_4
w_4	m_1	m_4	m_0	m_3	m_2

(b) Women's preference lists

Table 1
Instance of SMP of size 5. The matching $\{(m_3, w_1), (m_0, w_4), (m_4, w_3), (m_2, w_0), (m_1, w_2)\}$ is stable, while the blocking pairs (m_1, w_0), (m_4, w_0), and (m_4, w_1) make the matching $\{(m_3, w_2), (m_4, w_4), (m_0, w_3), (m_2, w_0), (m_1, w_1)\}$ unstable.

for a survey in SMP and its variants. Combinations of the mentioned variations are possible, and give rise to problems of varying complexity. For example, while SMP with either incomplete lists or ties can be solved in polynomial time, admitting both of them makes the problem NP-hard [8]. Moreover, while SMP always admits a solution, this is not the case for some of its variants or combinations. Clearly, whenever SMP is considered as part of a larger CSP model, the presence of other constraints may compromise the existence of solutions.

SMP and its variants find wide application in various fields such as computer science, mathematics, biology and physics [9]. SMP variants have been applied also in economics to study markets and monetary exchange models [10] or to relate social networks and job markets [11]. Systems based on SMP have been used to various solve real-word problems, for example, to assign medical students to hospitals by the National Resident Matching Program in the US [7], to match students to high schools [12], to solve the problems of college admissions, medical resident allocation, and optimal assignment of sailors to billets, to mention some among many (cf., for instance, [13, 14, 15] and the references therein).

The bursting advent of GPGPU (General Purpose computing on Graphics Processing Units) has recently driven the growth of GPU-computing and, more broadly, heterogeneous computing across numerous application domains. In the field of Computational Logic, research has shown that the massive parallelism and extreme computational power provided by GPUs can be used to accelerate reasoning and problem-solving tasks more effectively than traditional CPU-oriented solvers. For instance, approaches to SAT- and ASP-solving have been put forth in [16, 17, 18, 19], while [20, 21, 22] describe encouraging outcomes when using GPUs for CSP-solving. In keeping with this research, in this paper we introduce a GPU-accelerated propagator for the Stable Marriage constraint. As far as we are aware, this is the first study to use GPUs to speed up the propagation of such a global constraint.

The paper is organized as follows. In Section 2 we briefly review the Gale–Shapley algorithm which is the basis for the serial propagator described in Section 3. A GPU-accelerated implementation of the propagator is detailed in Section 4. Experimental comparison with a serial version of the propagator is reported on in Section 5. Finally, we draw our conclusions in Section 6.

2. The Extended Gale–Shapley Algorithm

The Gale–Shapley algorithm [23], also known as *deferred acceptance algorithm*, can be seen as a sequence of proposals from the men to the women. A man who is not already engaged proposes to his most preferred woman to whom he has not proposed yet. The woman accepts the proposal if she is single, or if it comes from a man she prefers more than her current partner. Starting with all individuals unmatched, the algorithm iteratively proceeds until every man is engaged. This process creates a so-called *man-optimal* stable matching, in which each man is engaged to his most preferred woman that he can have. Note that a man-optimal stable matching is *woman-pessimal*, meaning that each woman is matched with her least preferred man that she can have [1]. By switching the roles of men and women so that the women propose, the algorithm yields a woman-optimal and man-pessimal solution [23].

Algorithm 1: The extended Gale–Shapley algorithm (from [2])

```
mark each person as free
while there is a free man $m$ do
    let $w$ be the first woman in $m$'s list
    if $w$ is engaged to some man $p$ then mark $p$ as free
    mark $m$ and $w$ as engaged to each other
    for each successor $m'$ of $m$ in $w$'s list do
        delete $m'$ from the preference list of $w$
        delete $w$ from the preference list of $m'$
```

Several algorithms have been proposed for finding a balanced solution, based on different optimization criteria [7].

Consider the procedure outlined above. We can observe that if woman w receives a proposal from man m, then w's partner in the final stable matching (which is man-optimal/woman-pessimal) cannot be less preferred than m. Hence, each m' successor of m in w's list cannot be paired with w in any solution. Consequently, we can safely remove m' from w's preference list and w from that of m'. This observation leads to an improved algorithm called *extended Gale-Shapley algorithm* [1], shown in Algorithm 1.

Note that in this algorithm proposals are always accepted. Moreover, when the algorithm ends, each man is engaged with the first remaining woman in his list, and each woman with her last remaining man. These final lists are known as the man-oriented Gale-Shapley lists (MGS-lists). Intersecting these lists with those obtained by executing the algorithm with men and women switched, produces the *GS-lists*. Every partner a person p can have in any stable matching occurs in p's GS-list.

3. CSP and the Stable Marriage Constraint

A constraint satisfaction problem (CSP) is a triple $P = \langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables, D is the set of (finite) domains for the n variables, and C is the set of *constraints* over X. Let $\mathsf{dom}(x)$ denote the domain of the variable x. A constraint c on a set of variables $\mathsf{var}(c) = \{x_{i_1}, \ldots, x_{i_h}\} \subseteq X$ describes a subset $\mathsf{rel}(c) \subseteq \mathsf{dom}(x_{i_1}) \times \cdots \times \mathsf{dom}(x_{i_h})$.

A solution of P is an assignment $\sigma = [x_1/v_1, \ldots, x_n/v_n]$ of values to the variables in X that satisfies each $c \in C$, i.e., assuming $\mathsf{var}(c) = \{x_{i_1}, \ldots, x_{i_h}\}$, it holds that $\langle v_{i_1}, \ldots, v_{i_h}\rangle \in \mathsf{rel}(c)$. An assignment is locally consistent w.r.t. a set $Y \subseteq X$ if it satisfies each $c \in C$ such that $\mathsf{var}(c) \subseteq Y$. A CSP on X is *k-consistent* [24] if, given any set $Y = \{x_{i_1}, \ldots, x_{i_{k-1}}\} \subseteq X$ of $k - 1$ variables, for each assignment $[x_{i_1}/v_{i_1}, \ldots, x_{i_{k-1}}/v_{i_{k-1}}]$ which is locally consistent w.r.t. Y, for any k-th variable x_{i_k} there exists a value v_{i_k} such that $[x_{i_1}/v_{i_1}, \ldots, x_{i_k}/v_{i_k}]$ is locally consistent w.r.t. $Y \cup \{x_k\}$.

The *Stable Marriage Constraint (SMC)* modelling a given SMP P was introduced in [2]. Enforcing 2-consistency of such constraint corresponds to the application of the extended Gale–Shapley algorithm on P. That is, each solution for the SMC describes a stable matching of P.

Let us briefly describe the data structures and the basic functions introduced by [2] to deal with a SMC involving n men and n women.

As we will see, neither engaged couples nor the single/engaged status of men/women are explicitly stored. Instead, the propagator works by removing from the domains the same values that Algorithm 1 would remove from the preference lists. This is achieved by examining the changes in the domains (i.e., removals of elements) and removing the pairs that can be safely eliminated as a result of these modifications, following the same logic as the extended Gale-Shapley algorithm. Note that the removal of elements from domains can also occur as a result of propagations of other constraints (present in the CSP model at hand), i.e. by actions "external" to the SMC propagator.

The preference lists are stored into two 2-dimensional arrays (i.e. matrices), mpl and wpl. Hence, the array $mpl[i]$ stores the preference list for the the i-th man m_i, so that $mpl[i][j]$ is the j-th preference

Algorithm 2: Stable marriage propagation

```
1  Function deltaMin (i)
2      $\ell \leftarrow mpl[i][\mathsf{getMin}(x[i])]$
3      $\mathsf{setMax}(y[\ell], wPm[\ell][i])$
4      for $k \leftarrow xlb[i]$ to $\mathsf{getMin}(x[i]) - 1$ do
5          $j \leftarrow mpl[i][k]$
6          $\mathsf{setMax}(y[j], wPm[j][i] - 1)$
7      $xlb[i] \leftarrow \mathsf{getMin}(x[i])$

8  Function deltaMax (j)
9      for $k \leftarrow \mathsf{getMax}(y[j]) + 1$ to $yub[j]$ do
10         $i \leftarrow wpl[j][k]$
11         $\mathsf{remVal}(x[i], mPw[i][j])$
12     $yub[j] \leftarrow \mathsf{getMax}(y[j])$

13 Function removeValue (i,a,isMan)
14     if $isMan$ then
15         $j \leftarrow mpl[i][a]$
16         $\mathsf{remVal}(y[j], wPm[j][i])$
17     else
18         $j \leftarrow wpl[i][a]$
19         $\mathsf{remVal}(x[j], mPw[j][i])$

20 Function init ()
21     for $i \leftarrow 0$ to $n - 1$ do  $\mathsf{deltaMin}(i)$

22 Function inst (i,isMan)
23     if $isMan$ then
24         for $k \leftarrow xlb[i]$ to $\mathsf{getVal}(x[i]) - 1$ do
25             $j \leftarrow mpl[i][k]$
26             $\mathsf{setMax}(y[j], wPm[j][i] - 1)$
27         $j \leftarrow mpl[i][\mathsf{getVal}(x[i])]$
28         $\mathsf{setVal}(y[j], wPm[j][i])$
29         for $k \leftarrow \mathsf{getVal}(x[i]) + 1$ to $n - 1$ do
30             $j \leftarrow mpl[i][k]$
31             $\mathsf{remVal}(y[j], wPm[j][i])$
32     else
33         for $k \leftarrow 0$ to $\mathsf{getVal}(y[i]) - 1$ do
34             $j \leftarrow wpl[i][k]$
35             $\mathsf{remVal}(x[j], mPw[j][i])$
36         $j \leftarrow wpl[i][\mathsf{getVal}(y[i])]$
37         $\mathsf{setVal}(x[j], mPw[j][i])$
38         for $k \leftarrow \mathsf{getVal}(y[i]) + 1$ to $yub[i]$ do
39             $j \leftarrow wpl[i][k]$
40             $\mathsf{remVal}(x[j], mPw[j][i])$
```

of the i-th man m_i. Analogously, women's preference lists are stored in wpl. The constraint makes use of two arrays of variables x and y. The domains of these variables are initially set to $[0, n-1]$ and the value of each variable $x[i]$ (resp., $y[i]$) represents the partner of the man m_i (resp., the woman w_i). More specifically, a value ℓ for a variable $x[i]$ represents the ℓ-th preference in m_i's preference list, i.e., an assignment of ℓ to $x[i]$ describes the match (i,$mpl[i][\ell]$), coupling m_i and $w_{mpl[i][\ell]}$. To easily switch between preferences in the domains and their corresponding people, two additional matrices mPw and wPm are used to store the *inverse preference lists.* Then, $mPw[i]$ is m_i's inverse preference list and the value $mPw[i][j]$ is the index of w_j in m_i's preference list, and similarly for wPm. Two additional arrays of integers, yub and xlb, are also used. The former stores the previous upper bounds of the domains of the variables in y, that is the upper bounds they had when the previous propagation ended. Similarly, xlb contains the previous lower bounds of the variables in x. The values in yub are initialized to $n-1$, the ones in xlb to 0.

The original propagation algorithm is specified in terms of a set of functions to call when some variable changes [2]. In our implementation (cf., Algorithm 2) we extended the code of removeValue() and inst() so that both men and women are processed, contrary to what was stated in their original descriptions, where only men are treated. While the modification of inst() does not affect the correctness of the algorithm, the change in removeValue() is indeed necessary. The functions are:

deltaMin(i). Called when the minimum of dom$(x[i])$ increases (i.e., either m_i has been rejected by its preferred partner or the minimum has been changed externally to the propagator). The new favourite partner is w_ℓ (line 2 in Algorithm 2), and the function updates the maximum of dom(w_ℓ) to remove all men w_ℓ likes less that m_i (line 3). Moreover, in lines 4–6, for each value k that has been removed from dom$(x[i])$ the corresponding woman j is determined and the maximum of dom$(y[j])$ is updated to exclude men less preferred that $x[i]$. Finally, in line 7, the lower bound $xlb[i]$ for m_i is updated.

deltaMax(j). Called when the maximum of dom(w_j) decreases. The woman m_j is removed from the domain of each man m_i that has been removed from dom(w_j) (lines 9–11 of Algorithm 2). The upper bound $yub[j]$ for w_j is updated accordingly (line 12).

removeValue$(i, a, isMan)$. Called when a value a is removed from dom$(x[i])$ (resp., dom$(y[i])$) and a is not the minimum (resp., maximum) of the domain of the man m_i (resp., woman w_i). Then, m_i (resp., w_i) is removed from the domain of the person w_j (resp., m_j) corresponding to the value a.

inst$(i, isMan)$. This function is called when a variable is bound to a specific value, namely, a match between two persons i and j is created. It handles the two symmetric cases in which i represents either a man or a woman. Whenever i is engaged with a partner j, the function propagates the decision imposing that j is engaged to i (lines 28 and 37).

init$()$. This function is called once, when the propagator is initialized. It consists in a call to deltaMin(i) for each man m_i. It is somewhat analogous to setting all the men free so that they will make proposals at the start of the extended Gale-Shapley algorithm.

These functions make use of some auxiliary simple operations on domains. Namely, those to get the upper/lower value of a dom(x) (i.e., getMin(x) and getMax(x)); to get/set the value of x (i.e., getVal(x) and setVal(x, v)); to decrease the maximum of dom(x) to v if $v <$ getMax(x) (i.e., setMax(x, v)); and to remove a value v from dom(x) (i.e., remVal(x, v)). We omit their description (see [2] for details).

The authors of [2] suggest how the functions in Algorithm 2 can be integrated into a hypothetical CSP-solver to implement the propagation procedure. We will return to this point in Section 4.1, while describing the integration of the GPU-accelerated propagator for SMC into MiniCPP.

3.1. The MiniCP Solver and its Extension Mechanism

Among several efficient CSP-solvers, for the purpose of this paper we have chosen MiniCP [25], an open source solver designed to ease the development of extensions. In particular, our work is based on MiniCPP [26], a C++ re-engineering of MiniCP supporting CUDA code integration. Previous successful attempts [20, 21, 22] to extend MiniCPP with GPU-based modules strongly motivated this choice.

The main engine of MiniCPP performs the constraint-based search in the usual manner, by alternating search and propagation phases, possibly involving backtracking steps. Whenever a consistency-enforcing procedure is triggered, the corresponding method is responsible for starting the propagation and reporting the results to the main engine.

The possibility of extending MiniCPP with a new propagator is transparent to the way it is implemented, it suffices that it complies with the solver interface. Indeed, new constraint propagators can be easily integrated by extending the `Constraint` class of MiniCPP. To handle a new constraint, the extension must provide a constructor (to allocate and initialize the main data structures) and two procedures, post() and propagate(). The former is called by MiniCPP whenever a constraint is added to the model to set the conditions under which the propagation algorithm, i.e., the propagate() method, is called. This latter method is responsible to transparently offload the computation on the GPU and to suitably manage the related data transfers.

Note that MiniCPP provides backtrackable data structures, meaning that when the solver backtracks it reverts them to their values at the choice point to which the search backtracks, undoing any change made in the meantime. This can be achieved with an apposite class provided by the solver. As we will see, we exploit this feature in implementing our parallel propagator.

As concerns the extension of the input syntax for the CSP-solver, we first introduced the definition

`stable_matching(`$\langle men\rangle, \langle women\rangle, \langle menPrefLists\rangle, \langle womenPrefLists\rangle$`)`

to represent a SMC in the input MiniZinc models. Then, we relied on the MiniZinc annotation mechanism to select the propagator to be used [21]. In detail, when a constraint is annotated with `::gpu`, it is propagated using the GPU-accelerated implementation. Figure 1 shows a fragment of a MiniZinc encoding for the SMP in Table 1, using the syntax extension just described.

```
int: n = 5;
array [1..5] of var 0..n-1: men;
array [1..5] of var 0..n-1: women;
array [int, int] of 0..n-1: pm = [|3,4,1,2,0|0,1,3,2,4|0,4,3,2,1|2,1,4,3,0|0,1,4,2,3|];
array [int, int] of 0..n-1: pw = [|1,3,4,0,2|2,3,0,4,1|0,2,1,4,3|0,3,2,1,4|1,4,0,3,2|];
constraint stable_matching(men, women, pm, pw) ::gpu;
```

Figure 1: Fragment of a MiniZinc encoding for the SMP in Table 1

4. The Parallel Propagator: SMP on GPU

As mentioned, to extend MiniCPP with a new constraint, it suffices to specialize the class `Constraint`. Our serial and parallel implementations are available at https://clp.dimi.uniud.it/sw, and fully documented [27]. In what follows, we focus on the GPU-accelerated version, in particular on the propagate() method. For the convenience of the reader, a brief introduction to GPU computing and CUDA [28] is provided in appendix A.

Unlike the extended Gale-Shapley algorithm, the parallel implementation of the propagator for SMC proceeds by processing the proposals of all free men simultaneously. This is done by launching a thread for every free man. During the first execution of the propagator, all men are considered free. In subsequent iterations, the free men are those whose domain minimum has been modified, corresponding to the situation where their most preferred woman is no longer available. Each thread, independently, makes the proposal for the man it was assigned. Such proposal may cause a previously engaged man to break up and become free. The thread that handled the proposal will then take responsibility for the newly freed man and begin making proposals on his behalf. If a proposal does not free any man, the thread terminates. Note that each proposal can free at most one man, since a woman can be engaged to at most one man. Therefore, in this operation, the total number of free men cannot increase, since each proposal either reduces that number or leaves it unchanged. Consequently, at each propagator run, it is sufficient to start a thread for each currently free man.

If a missing value a is found during the processing of a free man, the SMC propagator must pretend that the proposal was made anyway, accepted, and the engagement was immediately broken. This is necessary because a pair removed externally (i.e., by another constraint or the solver) could still be a blocking pair (cf., Section 1), so the pairs it blocks must be removed from the domains. In presence of such a value a, not only the man who made the proposal remains free, but also another man may be freed. In this situation the total number of free men could increase.

For values in the men's domains removed by the propagator that are beyond the minimum, the corresponding values in the women's domains are all greater than the current woman's maximum. When, instead, a value is removed from outside the propagator, the corresponding value in the woman's domain may be smaller than the domain maximum. In this case, the woman would have accepted the proposal, forming a pair (m, w) that would be a blocking pair for all matchings in which m is engaged to a woman he likes less than w, and w is engaged to a man she likes less than m. The missing value in the domain of m was found because the new minimum is now greater than it, so m will necessarily be engaged to a woman he prefers less than w. It follows that w cannot be engaged with a man she likes less than m, otherwise the resulting matching would be unstable. The values of the men that w likes less than m must be removed from her domain. Since this removal changes the woman's maximum, it could potentially free up another man. To deal with this problem, the additional men freed this way are stored in an array, and the propagator is launched again to process them.

Note that, by applying the strategy just described, the parallel implementation runs only the necessary number of threads and does not require any synchronization between them.

Data structures: The propagator uses the same data structures described in Section 3 for the preference and inverse-preference lists (stored in simple matrices) and the domains (represented as bitmaps). Some additional data structures are also used. In particular:

- Four backtrackable arrays old_min_men, old_max_men, old_min_women and old_max_women, store the old bounds the variables had after the latest propagation. (These arrays play the role of xlb and yub seen in Section 3.)
- The arrays $stack\_mod\_men$ and $stack\_mod\_women$ store the lists of the men and the women whose domains have been modified in any way since the latest propagation. The variables $length\_men\_stack$ and $length\_women\_stack$ store their lengths. Note that values that are missing from the domains (for example, as a consequence of previous propagation of other constraints, or because they were not in the domain from the beginning) are treated as modifications of the domains, suitably initializing these arrays.
- The array $stack\_mod\_min\_men$ collects the men whose minimum was changed. The integer $length\_min\_men\_stack$ stores their number.
- As mentioned, when handling an externally removed value, a thread might end up freeing an additional man. This man is added to an array $new\_stack\_mod\_min\_men$, whose current length is $new\_length\_min\_men\_stack$.

The propagator kernels: The method propagate() essentially corresponds to the execution, managed by the host, of the following three phases:

1. Enforcing domains coherency: If a value was removed from a person's domain, that person is also removed from the domain of the person represented by that value.
2. The actual enforcing of the stable marriage constraint: This is where the strategy previously described is implemented. As previously mentioned, this step may be repeated several times
3. Update of auxiliary data structures (see below).

It is possible that an empty domain is detected in phase 2. Then, phase 3 will act accordingly to pass the information to the solver (which will either backtrack or declare unsatisfiability).

More specifically:

Phase 1. This step aims to restore domain consistency of the variables for men and women. As mentioned, when domain changes occur outside the SMC propagator, it is possible for a partner a to be removed from the domain of a person b, but not vice versa. This kernel $make\_domains\_coherent$ restores consistency, if necessary, by removing b from the domain of a. This removal could modify the minimum of a man's domain. In this case, such man is added to $stack\_mod\_min\_men$. The kernel is run with $length\_men\_stack + length\_women\_stack$ threads, one for each person whose domain was modified (and listed in $stack\_mod\_men$ and $stack\_mod\_women$). Each thread simply scans the domain associated to its person looking for missing values and acts as explained.

Phase 2. This step performs the core part of the propagation by enforcing the constraint logic. After an initialization step performed on the host, a grid of $length\_min\_men\_stack$ threads executing the kernel apply_sm_constraint() is launched (see Algorithm 3). In this execution each thread is assigned a man retrieved from the array $stack\_mod\_min\_men$ (line 2). Each thread starts a loop to process elements of the man's domain, starting from the value $old\_min\_men[m]$. The array old_min_men keeps track of the latest values of the men's domains that were processed. In line 4 each thread retrieves the first element to be checked. It may be the case that the current domain is empty (this is discovered by checking whether w_index is beyond the domain boundary), then the thread exits (line 5). Otherwise two scenarios are possible. Either w_index is in the domain and then it will be processed (lines 6–17), or it has been removed and the missing value must be handled (lines 18–25) as explained earlier.

In the first case, the thread/man makes a proposal to the woman w corresponding to w_index (lines 7–9). In line 9 an atomic access is performed to update the partner of w (i.e., to set $max\_women[w]$ to the index m_val of m in w's preference list). There may be different threads/men concurrently trying to make this update. Therefore, to ensure that only the proposer who is preferred over the current partner and all other proposers succeeds, we use an atomic instruction that coherently updates $max\_women[w]$ and returns its old value p_val (i.e., the index of w's previous partner in her preference list).

Algorithm 3: The kernel function apply_sm_constraint() (simplified)

```
1  get thread's unique $ID$
2  $m \leftarrow stack\_mod\_min\_men[ID]$    /* get the (first) man assigned to this thread */
3  loop forever
4      $w\_index \leftarrow old\_min\_men[m]$    /* get the next value/woman to process */
5      if $\mathsf{dom}(m) = \emptyset$ then return
6      else if $w\_index \in \mathsf{dom}(m)$ then    /* the value/woman is still in $m$'s domain */
7          $w \leftarrow mpl[m][w\_index]$    /* get $w$ corresponding to $w\_index$ */
8          $m\_val \leftarrow wPm[w][m]$    /* get $m$'s value $m\_val$ in $w$'s domain */
9          $p\_val \leftarrow \mathsf{atomicMin}(max\_women[w], m\_val)$    /* the best proposal succedes */
10         if $p\_val < m\_val$ then    /* $w$ prefers another man and rejects the proposal */
11             $old\_min\_men[m] \leftarrow w\_index + 1$
12             continue    /* proceed to the next iteration of the loop */
13         else if $p\_val = m\_val$ then    /* $w$ is already engaged to $m$ */
14             return
15         else    /* $p\_val > m\_val$: the proposal has been accepted */
16             remove $w$ from the domains of all the man following $m$ in her list
17             $m \leftarrow wpl[w][p\_val]$    /* set $m$ to be the man previously engaged with $w$ */
18     else    /* the value/woman has been removed from $m$'s domain */
19         $old\_min\_men[m] \leftarrow w\_index + 1$
20         $w \leftarrow mpl[m][w\_index]$    /* get $w$ corresponding to $w\_index$ */
21         $m\_val \leftarrow wPm[w][m]$    /* get $m$'s value $m\_val$ in $w$'s domain */
22         $\mathsf{atomicMin}(max\_women[w], m\_val-1)$    /* remove men following $m$ in $w$'s list */
23         remove $w$ from the domains of the men who were just removed from her list
24         if a man $p$ was freed and is not in $new\_stack\_mod\_min\_men$ then    /* prepare for the */
25             add $p$ to $new\_stack\_mod\_min\_men$    /* next call to apply_sm_constraint() */
```

The proposal may be rejected (line 10). This may happen because another concurrent thread removed w_index from m's domain (because, such thread made a proposal to w on behalf of a man m' that w prefers to m). It follows that a value that was thought to be in the domain of m has to be considered as missing. The thread simply moves on to the next value in m's domain (after having updated $old\_min\_men[m]$ appropriately in line 11 to trigger the next iteration in the loop).

If m is already engaged with w (line 13), no man has been freed, so the thread terminates.

If the proposal is accepted (that is, m_val is smaller than w's previous maximum and it becomes the new maximum), then w is removed from the domains of all the men following m in w's preference list, up to and including w's old maximum (line 16). If a man has been freed, the thread will treat such man as its newly assigned man (the update of m in line 17 will trigger another iteration of the loop).

In the case in which w_index is a missing value (the test in line 6 fails because it has been removed from $\mathsf{dom}(m)$, as mentioned above), the thread must remove from the domain of the woman associated with w_index all men following m in her list/domain (line 22). If this step actually removes some men from her domain, the woman must be deleted from their domains too (for brevity this is shown succinctly in line 23 of Algorithm 3). The value of $old\_min\_men[m]$ is updated accordingly (line 19) to avoid processing again the same woman. If w's maximum was actually modified (line 22), the old maximum must be added to $new\_stack\_mod\_min\_men$ (lines 24–25), so that apply_sm_constraint() can be launched again and handle him appropriately. Note that this step is simplified in Algorithm 3. The real implementation avoids multiple additions of the same elements to $new\_stack\_mod\_min\_men$ by exploiting some auxiliary data structures and using atomic accesses to such data.

When apply_sm_constraint() ends the host retrieves the updated data structures. In particular, the array $new\_stack\_mod\_min\_men$ is inspected. If some men occur in it, the host prepares for another call to the kernel to process them. This process is repeated until no free men exist.

Phase 3. The aim of this phase is to coherently update in parallel all elements of the arrays old_max_women, old_max_men, and old_min_women. This task is performed by the kernel function finalize_changes(). The grid is composed of n threads. The i-th thread processes the i-th man and the i-th woman. The i-th thread updates the i-th woman's values in old_max_women and in old_min_women, and the i-th man's value in old_max_men. When looking for the current bounds to put in old_max_men (resp., old_min_women), the thread scans the domain starting from the corresponding bound stored in max_men (resp., min_women). For lack of space, we omit the complete code of finalize_changes().

4.1. Integration into MiniCPP

In order to enable the integration of the functions of Algorithm 2 into a generic CSP-solver, the authors of [2] assume that constraint propagation is managed with a stack where the constraints to propagate are collected. Such constraints are extracted and propagated one by one, possibly causing some variable domains to shrink. This, in turn, causes all the constraints the variable is involved in to be added to the stack. The propagation process ends when there are no more constraints to propagate or a domain becomes empty.

MiniCPP has such design, but embodies trade-offs that prioritize simplicity over extensive feature support. One example is that it does not track which event (i.e., bound change or element removal) caused the insertion into the stack. Because of this, when the propagator for SMC is invoked, it checks all men's and women's variables for changes. When a change is found, the method to be called is added to an internal queue. Then, the queue is processed, executing all the needed methods. These, in turn, may cause further additions to the internal queue.

Another trade-off concerns how domain changes are tracked. MiniCPP offers various methods to check whether a variable has been modified (e.g., changed() and changedMin()), but these refer to the last time the variable was backed up (i.e., at the end of the previous propagation phase), not to the last time the variable was actually modified. Whenever a propagator is run twice between two of these time points, variables marked as modified before the first run will continue to be considered as modified before the second run. Additionally, when a propagator changes one of the variables it monitors, it is re-scheduled and will see that variable as modified. Consequently, using the built-in methods would result in redundant calls to the propagator. To avoid this, we exploited backtrackable arrays whose implementation is built in MiniCPP. In particular, the arrays x_old_sizes and y_old_sizes store the dimensions of the men's and women's domains after the previous propagation of the SMC. By comparing their current sizes with their old ones, it is possible to detect whether a variable has been modified since the latest execution of the propagator.

5. Experimental Evaluation

To compare the performance of the serial and parallel implementations, we randomly generated 200 instances of size $n = 2400$. Each instance includes two SMCs: One modelling the standard SMP, and another with the roles of men and women swapped. Our experiments focus on propagation time and aim to identify the conditions under which each propagator performs more effectively.

The machine used in these tests is equipped with an Intel Core i7-13700KF CPU (with 16 cores and 24 threads in Hyper-Threading, CPU clock rate 5.40GHz, 30MB cache), 128GB of RAM, and an NVIDIA GeForce RTX 4090 GPU (compute capability 8.9, 24GB VRAM, 16384 CUDA cores, clock rate 2.57GH).

The first parameter of interest is the number of free men at the beginning of phase 2 of the method propagate(). The design of the instances is such that all men are free in the first propagation, while in the following propagations either two or no men are free. Figure 2 shows the distribution of the propagation times in these three different situations. When there are no or only a few free men, the serial propagator outperforms the parallel one, with median execution times of 0.44 ms and 0.61 ms, respectively. In contrast, the parallel version yields median times of 1.36 ms and 8.07 ms under the same conditions. The significant difference between these two parallel timings is due to an optimization

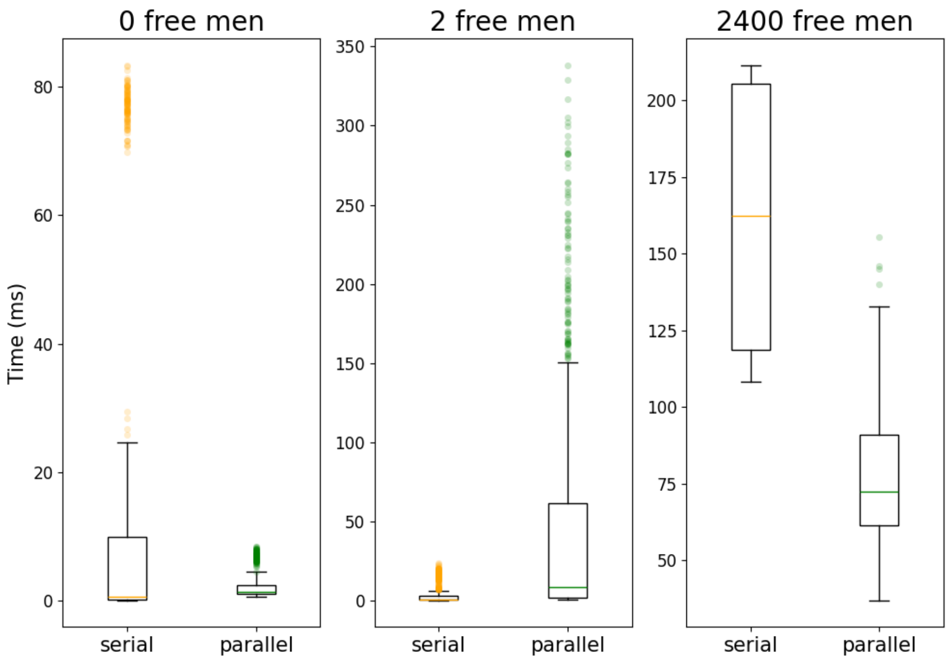

Figure 2: Box-plots showing the propagation times for different numbers of free men

that avoids launching the kernel for the second phase when there are no free men. When all men are free, the parallel propagator tends to be faster, with a median propagation time of 72.34 ms, compared to 162.34 ms for the serial version.

Figure 3 compares the execution times of the first 100 propagations for each of the three cases, for both serial and parallel versions. When there are exactly two free men, the serial propagator consistently outperforms the parallel one. Conversely, when all men are free, the parallel propagator shows a clear performance advantage. Interestingly, when no free men remain, the results show that the parallel version performs better in many instances, but there is no clear winner.

The total number of modified variables is another useful parameter to consider in order to characterize the class of instances in which the parallel version performs better. The instances are designed such that this number varies only when there are no free men. Figure 4 shows how the propagation time varies with respect to the number of modified variables. For illustrative purposes, the plot displays only a subset of data points, selected to uniformly cover the full range of the number of modified variables. The serial propagator is faster when the number of modified variables is relatively small, while the parallel propagator becomes more efficient as the number of modified variables increases. Note that the parallel propagation time remains nearly constant across all instances, demonstrating strong scalability. This behavior is mainly because of the optimization that skips the launch of the kernel for the second phase. More specifically, in the case of few free men, the limited parallelization is not enough to compensate the overhead of launching the kernel, and may even prove detrimental due to the small number of threads launched. Having skipped the second phase, the efficiency of the propagator depends only on the performances of the other phases. Moreover, a larger number of modified variables allows for greater parallelization of the first phase, resulting in a corresponding increase in performance.

A final parameter we considered is the total domain size, computed as the sum of the sizes of all domains. Specifically, we are interested in the old sizes, the domain sizes after the previous propagation of the constraint. Each modified domain is scanned from its old lower bound to its old upper bound, so the old size provides an estimate of the number of values the propagators may need to examine.

Figure 5 shows the results, with points selected to uniformly cover the x-axis. The case for $n = 2400$ free men is omitted, as the total domain size at the first propagation step is always $n * n * 2$, resulting in a plot that resembles the rightmost one in Figure 2. When there are no free men, the parallel propagator

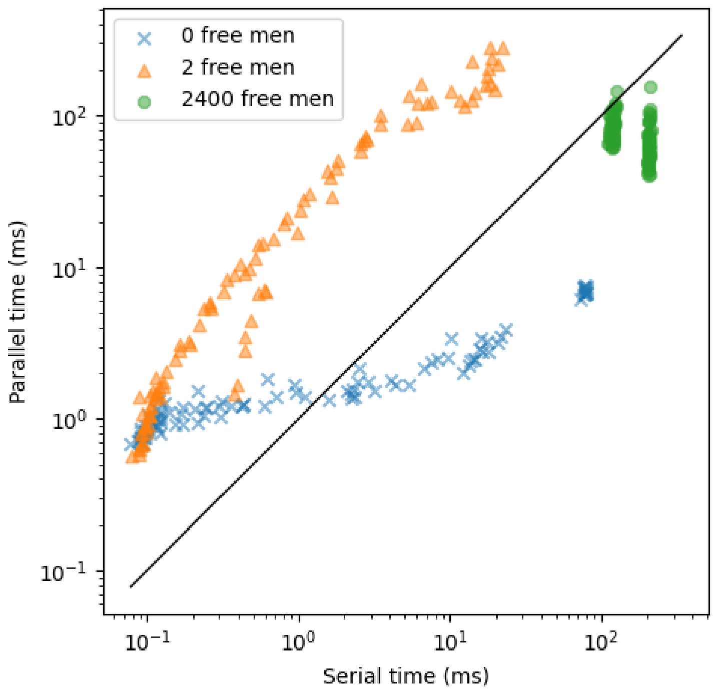

Figure 3: Comparison of the propagation times for the two propagators

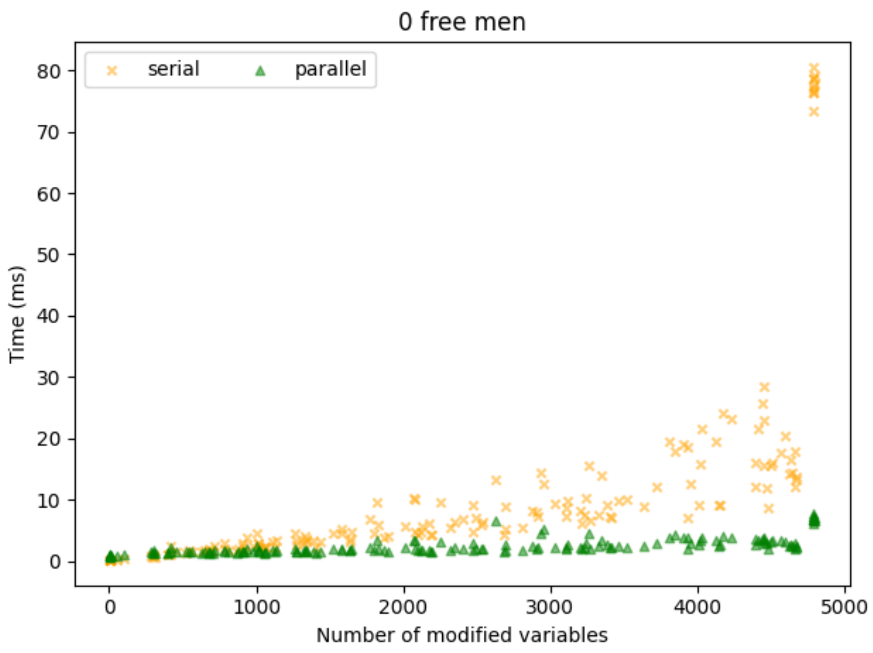

Figure 4: Correlation between propagation times and the number of modified variables

scales significantly better than the sequential. This is because larger total domains involve more work, which can be effectively parallelized, helping to offset the various overheads. In contrast, when there are two free men, the sequential propagator performs better.

Overall, the performance of the parallel propagator aligns with expectations. The more free men there are, the more advantageous it becomes. A notable exception is the case with no free men, where the parallel propagator can skip launching the kernel that executes the second phase. In this scenario,

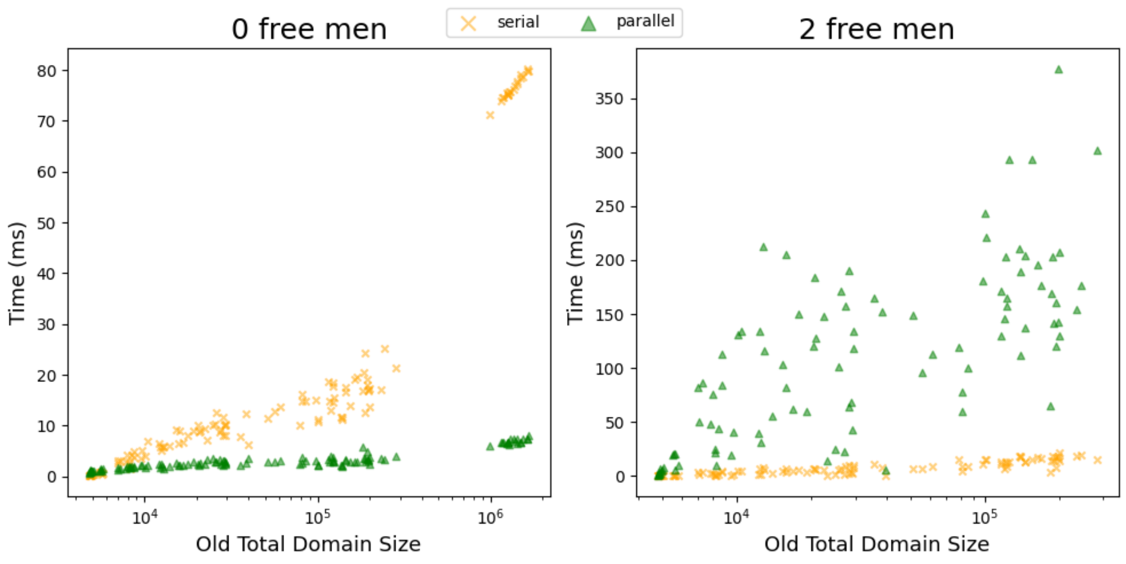

Figure 5: Correlation between propagation time and old total domain sizes

the parallel propagator demonstrates good scalability with respect to both the number of modified variables and the total domain size.

It is worth noting that the second phase is the most complex step of the parallel propagator. Consequently, inefficiencies in this phase have a greater impact on the overall propagation time than any performance gains achieved in the other phases.

6. Conclusions

This paper revisits the Stable Marriage constraint from a parallel perspective and demonstrates that CSP-solvers can effectively leverage the computational power of modern GPUs.

It presents a GPU-accelerated implementation of the constraint and compares it against a sequential version. The results demonstrate that the GPU-accelerated propagator offers better scalability, enabling solvers to handle larger instances more efficiently.

As future work, an interesting extension would be a hybrid propagator capable of automatically switching between sequential and GPU-accelerated propagators based on performance considerations. This mechanism could be implemented using insights from our analysis and determine threshold parameters specific to the machine on which the hybrid propagator runs.

Another valuable direction would be to extend the propagator to its symmetrical version. In this case, proposals are made by both men and women, and enforcing the SMC produces the GS-Lists instead of the MGS-Lists. This enables more values to be removed during propagation, at the cost of an increased computational workload. The use of a GPU in this context is especially beneficial, as it allows for both faster and stronger propagation.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] D. Gusfield, R. W. Irving, The Stable marriage problem - structure and algorithms, Foundations of computing series, MIT Press, Cambridge, MA, USA, 1989.

[2] C. Unsworth, P. Prosser, An n-ary constraint for the stable marriage problem, in: Proc. of the Fifth Int. Workshop on Modelling and Solving Problems with Constraints, 2005.

[3] R. W. Irving, Stable marriage and indifference, Discret. Appl. Math. 48 (1994) 261–272.

[4] R. W. Irving, An efficient algorithm for the "stable roommates" problem, J. Alg. 6 (1985) 577–595.

[5] D. E. Knuth, Mariages Stables et leurs relations avec d'autres problèmes combinatoires, Les Presses de l'Université de Montréal, 1976.

[6] C. Ng, D. S. Hirschberg, Three-dimensional stable matching problems, SIAM J. Discret. Math. 4 (1991) 245–252.

[7] K. Iwama, S. Miyazaki, A survey of the stable marriage problem and its variants, in: Proc. of ICKS'08, 2008, pp. 131–136.

[8] K. Iwama, D. F. Manlove, S. Miyazaki, Y. Morita, Stable marriage with incomplete lists and ties, in: J. Wiedermann, P. van Emde Boas, M. Nielsen (Eds.), Proc. of ICALP'99, volume 1644 of *LNCS*, Springer, 1999, pp. 443–452.

[9] E. M. Fenoaltea, I. B. Baybusinov, J. Zhao, L. Zhou, Y.-C. Zhang, The stable marriage problem: An interdisciplinary review from the physicist's perspective, Physics Reports 917 (2021) 1–79.

[10] P. Biró, F. Klijn, Matching with couples: a multidisciplinary survey, Int. Game Theory Review 15 (2013).

[11] E. Arcaute, S. Vassilvitskii, Social networks and stable matchings in the job market, in: S. Leonardi (Ed.), Proc. of WINE'09, volume 5929 of *LNCS*, Springer, 2009, pp. 220–231.

[12] D. Austin, The stable marriage problem and school choice, AMS Feature Column (2015). Online; accessed 4-March-2025.

[13] P. Biró, Applications of matching models under preferences, in: U. Endriss (Ed.), Trends in Computational Social Choice, AI Access, 2017, pp. 345–373.

[14] D. F. Manlove, Algorithmics of Matching Under Preferences, volume 2 of *Series on Theoretical Computer Science*, World Scientific, 2013.

[15] A. E. Roth, M. A. O. Sotomayor, Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis, Econometric Society Monographs, Cambridge University Press, 1990.

[16] A. D. Palù, A. Dovier, A. Formisano, E. Pontelli, CUD@SAT: SAT solving on GPUs, J. Exp. Theor. Artif. Intell. 27 (2015) 293–316. doi:10.1080/0952813X.2014.954274.

[17] A. Dovier, A. Formisano, E. Pontelli, F. Vella, Parallel execution of the ASP computation - an investigation on GPUs, in: M. D. Vos, T. Eiter, Y. Lierler, F. Toni (Eds.), Proc. of ICLP 2015, Technical Communications, volume 1433 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015.

[18] A. Dovier, A. Formisano, E. Pontelli, Parallel answer set programming, in: Y. Hamadi, L. Sais (Eds.), Handbook of Parallel Constraint Reasoning, Springer, 2018, pp. 237–282. doi:10.1007/978-3-319-63516-3_7.

[19] A. Dovier, A. Formisano, F. Vella, GPU-based parallelism for ASP-solving, in: P. Hofstedt, S. Abreu, U. John, H. Kuchen, D. Seipel (Eds.), Proc. of DECLARE'19, volume 12057 of *LNCS*, Springer, 2019, pp. 3–23. doi:10.1007/978-3-030-46714-2_1.

[20] F. Tardivo, A. Dovier, A. Formisano, L. Michel, E. Pontelli, Constraint propagation on GPU: a case study for the cumulative constraint, Constraints An Int. J. 29 (2024) 192–214. doi:10.1007/S10601-024-09371-W.

[21] F. Tardivo, A. Dovier, A. Formisano, L. Michel, E. Pontelli, Constraint propagation on GPU: a case study for the alldifferent constraint, J. Log. Comput. 33 (2023) 1734–1752. doi:10.1093/LOGCOM/EXAD033.

[22] F. Tardivo, L. Michel, E. Pontelli, CP for bin packing with multi-core and GPUs, in: P. Shaw (Ed.), Proc. of CP'24, volume 307 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 28:1–28:19. doi:10.4230/LIPICS.CP.2024.28.

[23] D. Gale, L. S. Shapley, College admissions and the stability of marriage, The American Mathematical Monthly 69 (1962) 9–15.

[24] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006.

[25] L. Michel, P. Schaus, P. Van Hentenryck, MiniCP: a lightweight solver for constraint programming,

Mathematical Programming Computation 13 (2021).
[26] R. Gentzel, L. Michel, W. J. van Hoeve, HADDOCK: a language and architecture for decision diagram compilation, in: H. Simonis (Ed.), Proc. of CP'20, volume 12333 of *LNCS*, Springer, 2020.
[27] S. Travasci, A GPU-based Parallel Propagator for the Stable Marriage Constraint, Master's thesis, University of Udine, Italy, 2025.
[28] NVIDIA Corporation, NVIDIA CUDA Zone, 2025. https://developer.nvidia.com/cuda-zone.
[29] NVIDIA, CUDA C++: Programming Guide, NVIDIA Press, Santa Clara, CA, 2025. Version 12.9.

A. CUDA and GPU Computing, in a Nutshell

GPU-computing can be seen as the use of a graphics processing unit (GPU) to accelerate 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 graphics concepts in favor of high-performance computing concepts [29]. It has been successfully used to accelerate computations in a variety of fields, such as physics, bioinformatics, and machine learning, to mention some among many.

The architecture of a GPU (cf., Fig. 6) includes a main memory (DRAM), typically external to the chip and used as *global memory*, an L2 cache, and an array of *streaming multiprocessors.* Each streaming multiprocessor contains a small amount of fast memory integrated into the chip, used as an L1 cache or *shared memory*, and several *CUDA cores*, responsible for the actual execution of instructions. The architecture is designed to allow fast context switching of lightweight threads.

The conceptual model underlying the parallelism supported by CUDA is the *Single-Instruction Multiple-Thread (SIMT),* in which the same instruction is executed by different threads, while data and operands can vary from thread to thread.

A CUDA program includes parts intended to execute on the CPU (also referred to as *host*) and parts intended to execute concurrently with the GPU (usually referred to as *device*). In the simplest possible interaction between host and device, the host code transfers data to the device global memory, launches a sequence of parallel computations on the device, and retrieves the results from device memory. Commands are sent to the device from the host in a *stream* and are executed in-order. Multiple streams can be used to run concurrent parallel computations on the device. The CUDA library supports interaction, synchronization, and communication between the host and the device. Each computation on the device is described as a set of concurrent threads (a *grid*, in CUDA terminology), each of which performs the same function (called a *kernel*). Each thread is executed by a CUDA core; these threads are organized hierarchically into *warps* (groups of 32 threads that execute in lock-step mode) and thread *blocks.* Each block is assigned for execution to a streaming multiprocessor. Threads in a warp/block are intended to execute the same instruction on different data. If the control flow between threads within a warp diverges, their execution is serialized. The global memory of the device is accessible to all threads, while threads in the same block can access shared memory at high speed.

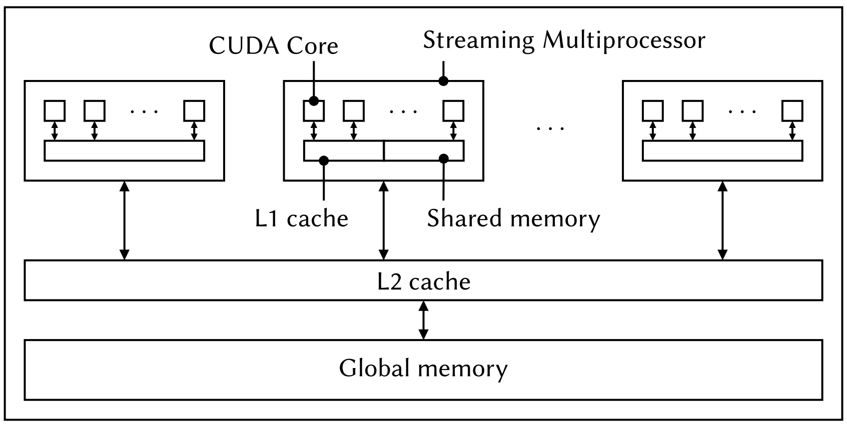

Figure 6: Simplified GPU architecture