Accelerating Exact and Approximate Inference for (Distributed) Discrete Optimization with GPUs⋆

Ferdinando Fioretto · **Enrico Pontelli** ·
William Yeoh · **Rina Dechter**

Abstract *Discrete optimization* is a central problem in artificial intelligence. The optimization of the aggregated cost of a network of cost functions arises in a variety of problems including *Weighted Constraint Programs* (WCSPs), *Distributed Constraint Optimization* (DCOP), as well as optimization in stochastic variants such as the tasks of finding the *most probable explanation* (MPE) in *belief networks*. Inference-based algorithms are powerful techniques for solving discrete optimization problems, which can be used independently or in combination with other techniques. However, their applicability is often limited by their compute intensive nature and their space requirements.

This paper proposes the design and implementation of a novel inference-based technique, which exploits modern massively parallel architectures, such as those found in Graphical Processing Units (GPUs), to speed up the resolution of exact and

---

Ferdinando Fioretto
Industrial and Operations Engineering, University of Michigan
Ann Arbor, MI, USA
E-mail: fioretto@umich.edu

Enrico Pontelli
Computer Science, New Mexico State University
Las Cruces, NM, USA
E-mail: epontell@cs.nmsu.edu

William Yeoh
Computer Science and Engineering, Washington University in St. Louis
St. Louis, MO, USA
E-mail: wyeoh@wustl.edu

Rina Dechter
School of Information and Computer Science, University of California, Irvine
Irvine, CA, USA
E-mail: dechter@ics.uci.edu

approximated inference-based algorithms for discrete optimization. The paper studies the proposed algorithm in both centralized and distributed optimization contexts.

The paper demonstrates that the use of GPUs provides significant advantages in terms of runtime and scalability, achieving up to two orders of magnitude in speedups and showing a considerable reduction in execution time (up to 345 times faster) with respect to a sequential version.

Keywords GPU · WCSP · MPE · DCOP · (Mini-)Bucket Elimination · (A)DPOP

1 Introduction

The importance of constraint optimization is outlined by the impact of its application in a wide range of domains, such as supply-chain management (e.g., [55,33]), roster scheduling (e.g., [1,11]), combinatorial auctions (e.g., [59]), bioinformatics (e.g., [2, 13,24]), multi-agent systems (e.g., [21]) and probabilistic reasoning (e.g, [50]).

In *Constraint Satisfaction Problems (CSPs)*, the goal is to find a value assignment for a set of variables that satisfies a set of constraints [4,56]. The assignments satisfying the problem constraints are called *solutions*. In *Weighted Constraint Satisfaction Problems (WCSPs)* the goal is that of finding an optimal solution, given a set of preferences expressed by means of cost functions [61,60,8].

When the problems involve uncertainty, we recur to the notion of *belief* or *Bayesian* networks (BNs) [50]. This framework aims at modeling natural phenomena and exogenous uncertainty through probabilistic reasoning. In BNs the goal is that of answering queries given partial beliefs under conditions uncertainty. Common tasks over BNs include finding the *most probable explanation* (MPE), also known as *maximum a posteriori hypothesis* (MAP), that is finding the assignment with largest probability to all unobserved variables given some observed variables [16]. Belief networks are widely applied to a variety of applications, such as, diagnosis [43] and linkage analysis [31,32].

When resources are distributed among a set of autonomous agents and communication among the agents is restricted, WCSPs take the form of *Distributed Constraint Optimization Problems (DCOPs)* [47,53,67]. In this context, agents coordinate their value assignments to minimize the overall sum of resulting constraint costs. DCOPs have been employed to model various distributed optimization problems, such as meeting scheduling [45,66], resource allocation [23,68], power network management problems [38,34,30], and coordination of appliances in smart homes [29,57]. We will refer to WCSPs and DCOPs as *discrete optimization problems*.

Algorithms to solve discrete optimization problems can be classified as *exact* and *approximated*. Exact algorithms are guaranteed to find optimal solutions. However, since solving WCSPs and DCOPs is NP-hard [56], optimally solving these problems results in prohibitive runtime and/or use of resources, such as memory or network load. In contrast, approximated algorithms trade solution optimality for shorter runtime and a more efficient use of the available resources.

Furthermore, discrete optimization algorithms can adopt two main paradigms: *search* or *inference*. *Search-based methods* rely on the use of non-deterministic branching rules to explore different value assignments to variables. These rules are

applied recursively until all problem variables are assigned. This process defines a search tree (typically traversed in a depth-first fashion), which has the advantage of requiring only polynomial space. However, the practical efficiency of these methods relies on their ability to prune redundant or sub-optimal subtrees. *Inference-based methods* are inspired from dynamic programming (DP) techniques. These methods apply a sequence of transformations to reduce the problem size at each step while preserving its semantics. A well known inference-based approach is *Bucket Elimination* (BE) [15]. BE iterates over the variables of the problem, reducing the size of the problem at each step, by replacing a variable and its related cost functions with a single new function, derived by optimizing over the possible values of the eliminated variable. The *Dynamic Programming Optimization Protocol (DPOP)* [53] is one of the most efficient inference-based DCOP solvers, and it can be seen as a distributed version of BE, where agents exchange newly introduced cost functions via messages.

The importance of inference-based approaches arises in several optimization fields including constraint programming [4,56]. For example, several *propagators* adopt DP-based techniques to establish constraint consistency. For instance, **(1)** the *knapsack* constraint propagator proposed by Trick applies DP techniques to establish arc consistency on the constraint [65]; **(2)** the propagator for the *regular* constraint establishes arc consistency using a specific digraph representation of the DFA, which has similarities to dynamic programming [51]; **(3)** the *context free grammar* constraint makes use of a propagator based on the CYK parser that uses DP to enforce generalized arc consistency [54].

The main drawback of inference-based methods, including BE and DPOP, is that each transformation may introduce cost functions with large arities, requiring exponential time and space in a key structural parameter of a problem, called *induced width*. While inference-based approaches may not always be appropriate to solve discrete optimization problems, as their time and space requirements may be prohibitive, they may be very effective in problems with particular structures, such as problems where their underlying primal graphs have small induced widths or distributed problems where the number of messages is crucial for performance, despite the size of the messages. Additionally, approximated inference methods can be effectively used to derive lower bounds, which are important components of branch and bound algorithms, as they can be used to prune parts of the search space by detecting *dominated* solutions—i.e., solutions whose cost can provably not be lower than the best cost found so far. *Mini-Bucket Elimination (MBE)* is an approximated variant of BE that can be used for this purpose.

Recent developments on external-memory algorithms have shown that the use of large secondary data storage can be effective to extend the applicability of memory intensive approaches [22,44,37,63]. However, the computational solving runtime remains a bottleneck.

To contrast this background, we note that the structure exploited by inference-based approaches in constructing solutions makes it suitable to exploit a novel class of massively parallel platforms that are based on the *Single Instruction Multiple Thread* (SIMT) paradigm—where multiple threads may concurrently operate on different data, but are all executing the same instruction at the same time. The SIMT-based paradigm is widely used in modern *Graphical Processing Units (GPUs)* for general

purpose parallel computing. Several libraries and programming environments (e.g., the *Compute Unified Device Architecture* (CUDA)) have been made available to allow programmers to exploit the parallel computing power of GPUs.

In this paper, we propose the design and implementation of both an exact and an approximated inference-based algorithm that exploits parallel computation using GPUs to solve WCSPs and DCOPs. Our proposal aims at employing GPU hardware to speed up the inference process, thus providing an alternative way to enhance the performance of inference-based discrete optimization approaches.

This paper makes the following contributions: **(1)** We propose a novel design and implementation of a centralized and a distributed exact inference-based algorithm, inspired by BE and DPOP, to optimally solve WCSPs and DCOPs, which harnesses the computational power offered by parallel platforms based on GPUs; **(2)** We introduce an approximated version of the GPU-based inference-based algorithm, inspired by MBE; **(3)** We report an extensive empirical analysis that shows significant improvements in performance with respect to the sequential CPU-based algorithms, reporting an average speedup of two order of magnitude; and **(4)** We show the generality of our approach through empirical evaluations on three different GPU architectures, all providing significant speedups. While the description of the techniques proposed in this paper focus on discrete optimization tasks, they also applies to other key problems in probabilistic graphical models, such as, MPE.

2 Background: Weighted Constraint Satisfaction Problems

A *weighted constraint satisfaction problem* (WCSP) [40,61] is a tuple $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, where $\mathbf{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables, $\mathbf{D} = \{D_{x_1}, \ldots, D_{x_n}\}$ is a set of finite domains for the variables in $\mathbf{X}$, with D_{x_i} being the set of possible values for the variable x_i, and $\mathbf{C}$ is a set of *weighted constraints* (or *cost functions*). A weighted constraint $f_i \in \mathbf{C}$ is a function that maps tuples defined on the set of variables relevant to f_i into $\mathbb{R}_+ \cup \{\infty\}$, where ∞ is a special value denoting that a given combination of values is not allowed. The set of variables relevant to f_i is referred to as the *scope* of f_i, and denoted as $\mathbf{x}^i \subseteq \mathbf{X}$. Formally, $f_i : \Pi_{x_j \in \mathbf{x}^i} D_{x_j} \to \mathbb{R}_+ \cup \{\infty\}$.[1] A *solution* is a value assignment for a subset ρ of variables from $\mathbf{X}$ that is consistent with their respective domains; i.e., it is a partial function $\theta : \mathbf{X} \to \bigcup_{i=1}^n D_{x_i}$ such that, for each $x_j \in \mathbf{X}$, if $\theta(x_j)$ is defined (i.e., $x_j \in \rho$), then $\theta(x_j) \in D_{x_j}$. The ***cost*** of an assignment ρ is the sum of the evaluation of the constraints involving all the variables in ρ. A solution is *complete* if it assigns a value to each variable in $\mathbf{X}$ and has finite cost (i.e., different from ∞). We will use the notation σ to denote a complete solution, and, for a set of variables $\mathbf{V} = \{x_{i_1}, \ldots, x_{i_h}\} \subseteq \mathbf{X}$, $\sigma_{\mathbf{V}} = \langle \sigma(x_{i_1}), \ldots, \sigma(x_{i_h}) \rangle$ is the projection of σ to the variables in $\mathbf{V}$, where $i_1 < \cdots < i_h$. The goal of a WCSP is to find a complete solution σ^* with minimal cost, i.e.,

$$\sigma^* = \operatorname*{argmin}_{\sigma \in \Sigma} \sum_{f_i \in \mathbf{C}} f_i(\sigma_{\mathbf{x}^i}), \tag{1}$$

---

[1] For simplicity, we assume that tuples of variables are built according to a predefined ordering.

where Σ is the *state space*, defined as the set of all possible complete solutions.

Given a WCSP P, $G_P = (\mathbf{X}, E_{\mathbf{C}})$ is the **primal graph** of P, where $\{x, y\} \in E_{\mathbf{C}}$ iff $\exists f_i \in \mathbf{C}$ such that $\{x, y\} \subseteq \mathbf{x}^i$. Given an ordering o on $\mathbf{X}$, we say that a variable x_i has lower **priority** w.r.t. a variable x_j, denoted $x_i \prec_o x_j$, if x_i precedes x_j in o.

Definition 1 (Induced Graph, Induced Width [16]) Given the primal graph G_P and an ordering o on its nodes, the *induced graph* G_P^* on o is the graph obtained from G_P by connecting nodes, processed in descending order of priority, to all their preceding neighbors. Processing a node x_i results in the addition of edges connecting pairs of preceding neighbors of x_i. Given a graph and an ordering of its nodes, the *width* of a node is the number of edges connecting it to its preceding nodes in the ordering. The induced width w_o^* of G_P is maximum width over all nodes of G_P^* along the ordering o.
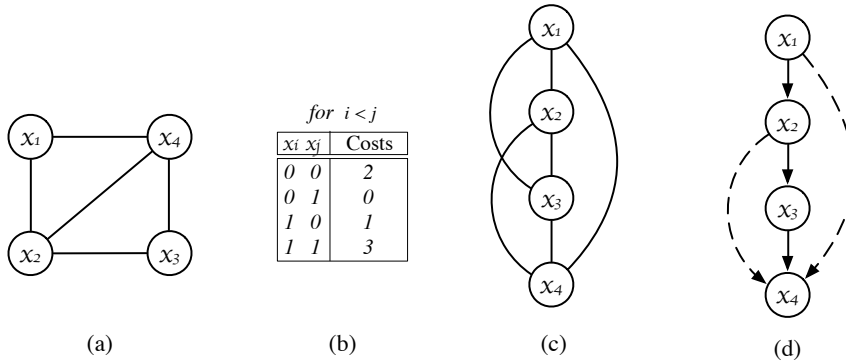

Fig. 1: Example of a WCSP : **(a):** Primal graph. **(b):** Cost functions. **(c):** A possible induced graph. **(d):** A possible pseudo-tree.

Example 1 Fig. 1(a) illustrates the primal graph of a simple WCSP instance with 4 binary variables, x_1, x_2, x_3, and x_4, and 5 constraints, $f(x_1, x_2)$, $f(x_1, x_4)$, $f(x_2, x_3)$, $f(x_2, x_4)$, $f(x_3, x_4)$. Fig. 1(b) illustrates the constraints costs of the WCSP, which associate a cost value for each combination of values for the variables in the scope of the constraints. Fig. 1(c) shows the induced graph G_P^* obtained along the ordering $o = \langle x_1, x_2, x_3, x_4 \rangle$. Its induced width is 3.

Definition 2 (Pseudo-tree [17]) Given a primal graph G_P and an ordering o on its nodes, a *DFS pseudo-tree* arrangement for G_P is a rooted directed tree $T = \langle \mathbf{X}, E_T \rangle$ of G_P such that if $f_i \in \mathbf{C}$ and $\{x, y\} \subseteq \mathbf{x}^i$, then x and y appear in the same branch of T. The root of T is the node associated to the variable with lower priority in o. Edges of G_P that are *in* (resp. *out of*) E_T are called *tree edges* (resp. *backedges*). The tree edges connect parent-child nodes, while backedges connect a node with its *pseudo-parents* and its *pseudo-children*.

Example 2 Fig. 1(d) shows one possible pseudo-tree $T = \langle \mathbf{X}, E_T \rangle$ associated to the primal graph shown in Fig. 1(a), with $E_T = \{f(x_1, x_2), f(x_2, x_3), f(x_3, x_4)\}$, and order $o = \langle x_1, x_2, x_3, x_4 \rangle$. The node labeled x_1 is the root node; it has a pseudo-child, node x_4. The node labeled x_4 has two pseudo-parents nodes: x_2 and x_1. The solid lines describe tree edges, while the dotted lines represent backedges.

Definition 3 (Projection) The *projection* of a cost function f_i on a set of variables $\mathbf{V} \subseteq \mathbf{x}^i$ is a new cost function $f_{i|\mathbf{V}} : \mathbf{V} \to \mathbb{R}_+ \cup \{\infty\}$, such that for each possible assignment $\theta \in {''}_{x_j \in \mathbf{V}} D_{x_j}$, $f_{i|\mathbf{V}}(\theta) = \min\limits_{\sigma \in \Sigma, \sigma_{\mathbf{V}} = \theta} f_i(\sigma_{\mathbf{x}^i})$.[2]

In other words, $f_{i|\mathbf{V}}$ is constructed from the tuples of f_i, removing the values of the variable that do not appear in $\mathbf{V}$ and removing duplicate values by keeping the minimum cost of the original tuples in f_i.

Definition 4 (Concatenation) Let us consider two assignments θ', defined for variables V, and θ'', defined for variables W, such that for each $x \in V \cap W$ we have that $\theta'(x) = \theta''(x)$. Their *concatenation* is an assignment $\theta' \cdot \theta''$ defined for $V \cup W$, such as for each $x \in V$ (resp. $x \in W$) we have that $\theta' \cdot \theta''(x) = \theta'(x)$ (resp. $\theta' \cdot \theta''(x) = \theta''(x)$).

We define two operations on cost functions:

- The **aggregation** of two functions f_i and f_j, is a function $f_i + f_j : \mathbf{x}^i \cup \mathbf{x}^j \to \mathbb{R}_+ \cup \{\infty\}$, such that $\forall \theta' \in {''}_{x_k \in \mathbf{x}^i} D_{x_k}$ and $\forall \theta'' \in {''}_{x_k \in \mathbf{x}^j} D_{x_k}$, if $\theta' \cdot \theta''$ is defined, then we have that

$$(f_i + f_j)(\theta' \cdot \theta'') = f_i(\theta') + f_j(\theta'').$$

- The **elimination** of a variable $x_j \in \mathbf{x}^i$ from a function f_i, denoted as $\pi_{-x_j}(f_i)$, produces a new function with scope $\mathbf{x}^i \setminus \{x_j\}$, and defined as the projection of f_i on $\mathbf{x}^i \setminus \{x_j\}$, i.e.,

$$\pi_{-x_j}(f_i) = f_{i|\mathbf{x}^i \setminus \{x_j\}}.$$

While the aggregation and elimination operators are defined on summation and minimization, respectively, for discrete optimization problems, several tasks in belief networks can be solved by using variants of the aggregation and elimination operators [16].

2.1 Bucket Elimination

Bucket Elimination (BE) [15,16] is a complete inference algorithm that can be used to find the optimal solutions of a WCSP. Algorithm 1 illustrates its pseudocode. BE operates in the following two phases:

---

[2] For simplicity, we also use θ to represent the tuple $\langle \theta(x_{i_1}), \ldots, \theta(x_{i_h}) \rangle$ where $\{x_{i_1}, \ldots, x_{i_h}\}$ is the domain of θ.

Algorithm 1: BUCKET ELIMINATION

```
/* Variable Elimination Phase                                        */
```
1 **for** $i \leftarrow n$ **downto** 1 **do**
2     $B_i \leftarrow \{f_j \in \mathbf{C} \mid x_i \in \mathbf{x}^j \wedge i = \min\{k \mid x_k \in \mathbf{x}^j\}\}$
3     $\hat{f}_i \leftarrow \pi_{-x_i}\left(\sum_{f_j \in B_i} f_j\right)$
4     $\mathbf{X} \leftarrow \mathbf{X} \setminus \{x_i\}$
5     $\mathbf{C} \leftarrow (\mathbf{C} \setminus B_i) \cup \{\hat{f}_i\}$
```
/* Value Assignment Phase                                            */
```
6 **for** $i \leftarrow 1$ **to** n **do**
7     $x_i \leftarrow d_i$ s.t. $d_i \in D_{x_i}$ and d_i is the best extension of $x_1, \ldots, x_{i-1}$ w.r.t. B_i
8 **return** $\hat{f}_1$

- *Variable Elimination Phase*. BE operates from the highest to lowest priority variable. When operating on variable x_i, it creates a bucket B_i, which is the set of all cost functions that involve x_i as the highest priority variable in their scope (line 2). The algorithm then computes a new cost function $\hat{f}_i$ by aggregating the functions in B_i and eliminating x_i (line 3). Thus, x_i can be removed from the set of variables $\mathbf{X}$ to be processed (line 4) and the new function $\hat{f}_i$ replaces in $\mathbf{C}$ all the cost functions that appear in B_i (line 5). Thus, a bucket B_i contains both the original WCSP functions as well as the functions placed in it during the variable elimination process. In this work, we refer to the $\hat{f}_i$ functions as the ***bucket functions***.
- *Value Assignment Phase*. Once the variable with the lowest priority has been processed, the algorithm considers variables in increasing order of priority. For each variable x_i, it generates an optimal assignment by selecting a value $d_i \in D_{x_i}$ that minimizes the cost of the functions in B_i given the assignments of all the other variables appearing in the scope of the functions in B_i.

As a byproduct, and without additional overhead, BE can compute the number of consistent solutions of the problem (see [16], for details). The time and space complexity of BE is exponential on the induced width of the underlying primal graph, which captures the maximum arity of the $\hat{f}_i$ functions (line 3).

Example 3 In our WCSP example of Fig. 1, during the *Variable Elimination Phase*, BE operates, in order, on the variables x_4, x_3, x_2, and x_1. When x_4 is processed, the bucket $B_4 = \{f(x_1, x_4), f(x_2, x_4), f(x_3, x_4)\}$ is generated, and highlighted in Fig. 2(a)(top) by red edges. The resulting bucket function $\hat{f}_4$ is shown in Fig. 2(a)(bottom), where the rightmost column shows the values for x_4 after its elimination. BE, hence, updates the sets $\mathbf{X} = \{x_1, x_2, x_3\}$ and $\mathbf{C} = \{f(x_1, x_2), f(x_2, x_3), \hat{f}_4\}$, as shown in the primal graph of Fig. 2(b)(top), where the function $\hat{f}_4$ is displayed as a dotted line. When x_3 is processed, $B_3 = \{f(x_2, x_3), \hat{f}_4\}$, and $\hat{f}_3$ is shown in Fig. 2(b)(bottom). Thus, $\mathbf{X} = \{x_1, x_2\}$ and $\mathbf{C} = \{f(x_1, x_2), \hat{f}_3\}$, as shown in Fig. 2(c)(top). Next, x_2 is processed, and $B_2 = \{f(x_1, x_2), \hat{f}_3\}$, and $\hat{f}_2$ is illustrated in Fig. 2(c)(bottom). Thus, $\mathbf{X} = \{x_1\}$ and $\mathbf{C} = \{\hat{f}_2\}$, as shown in Fig. 2(d)(top). Lastly, the algorithm processes x_1, sets $B_1 = \{\hat{f}_2\}$, and $\hat{f}_1$ is minimized when $x_1 = 0$, as shown in Fig. 2(d)(bottom). Next, BE starts the *Value Assign-*
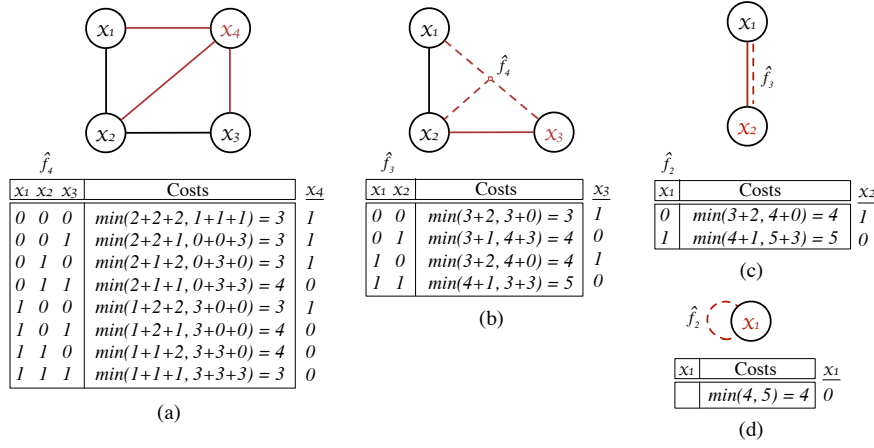
Fig. 2: Bucket Elimination steps for the WCSP of Fig. 1 .

Algorithm 2: MINI-BUCKET ELIMINATION(z)

```
   /* Variable Elimination Phase                                        */
 1 for i ← n downto 1 do
 2 │   B_i ← {f_j ∈ C | x_i ∈ x^j ∧ i = min{k | x_k ∈ x^j}}
 3 │   Let {B_{i_1}, ..., B_{i_m}} be a partition of B_i s.t. |⋃_{f_j ∈ B_{i_k}} x^j| ≤ z, for each k = 1, ..., m
 4 │   foreach k ∈ {1, ..., m} do
 5 │   │   f̂_{i_k} ← π_{-x_i}(∑_{f_j ∈ B_{i_k}} f_j)
 6 │   │   C ← (C \ B_{i_k}) ∪ {f̂_{i_k}}
 7 │   X ← X \ {x_i}
   /* Value Assignment Phase                                            */
 8 for i ← 1 to n do
 9 │   x_i ← d_i s.t. d_i ∈ D_{x_i} and d_i is the best extension of x_1, ..., x_{i-1} w.r.t. B_i
10 return f̂_1
```

---

ment Phase, which operates, in order, on the variables x_1, x_2, x_3, and x_4. First, it selects the value that minimizes $\hat{f}_1$, ($x_1 = 0$). Thus, it processes x_2, and selects the value $x_2 = 1$, as it minimizes $\hat{f}_2$ when $x_1 = 0$, as illustrated in Fig. 2(c)(bottom). Similarly, when BE processes x_3, it selects the value $x_3 = 0$, as it minimizes $\hat{f}_3$ when $x_1 = 0$ and $x_2 = 1$, illustrated in Fig. 2(b)(bottom). Finally, BE processes the last variable x_4 and assigns it the value 1, since it minimizes $\hat{f}_4$ when $x_1 = 0, x_2 = 1$, and $x_3 = 0$, illustrated in Fig. 2(a)(bottom). Thus, $\sigma^* = \langle 0, 1, 0, 1 \rangle$ is an optimal solution to the problem, with cost 4.

2.2 Mini-Buckets

The memory complexity and time complexity of BE depend on the arity of the functions $\hat{f}$ produced during the variable elimination step. Such requirements can quickly become infeasible for problems with large induced widths. To overcome this limitation, Dechter and Rish proposed an incomplete version of the Bucket Elimination [19]. The *Mini-Bucket Elimination* (MBE) is an approximated version of the BE that allows one to bound the arity of the functions $\hat{f}_i$ generated during the Variable Elimination Phase. Its pseudocode is illustrated in Algorithm 2. Similarly to BE, MBE operates in the following two phases:

- **Variable Elimination Phase**. As in BE, during the variable elimination phase, MBE operates on the problem variables in decreasing order of priority. However, rather than creating a single bucket function $\hat{f}_i$ whose scope is the union of the scope of each function in the bucket B_i, it partitions B_i in a set of m "mini"-buckets $\{B_{i_1}, \ldots, B_{i_m}\}$, such that the size of the scope of the bucket function $\hat{f}_{i_k}$, obtained by aggregating the functions in B_{i_k}, is bounded by a parameter z, for each $k \in \{1, \ldots, m\}$ (line 3). Thus, MBE considers each mini-bucket independently, and computes m new bucket functions $\hat{f}_{i_k}$, by aggregating the functions in B_{i_k} and eliminating x_i (line 5). These functions replace in $\mathbf{C}$ all the functions that appear in B_{i_k} (line 6), and the set of variables is updated as in BE (line 7).
- **Value Assignment Phase**. This phase is analogous to that of BE (lines 8–9).

Consider the elimination step for a variable $x_i \in \mathbf{X}$. Since:

$$\sum_{k=1}^{m} \left[\pi_{-x_i} \left(\sum_{f_j \in B_{i_k}} f_j \right) \right] \leq \pi_{-x_i} \left(\sum_{f_j \in B_i} f_j \right)$$

eliminating x_i using mini-buckets produces a lower bound on the optimal cost for the bucket B_i. Thus, MBE produces a lower bound on the optimal solution cost. Running the Value Assignment Phase might hence return a sub-optimal solution, whose evaluation will be an upper bound on the optimal solution cost.

The time and space complexity of MBE is exponential on the maximal arity of the aggregated functions in the mini-buckets (line 13), and thus it is bounded by the parameter z.

Example 4 Consider the WCSP of Fig. 1 solved with MBE using $z = 1$. As in BE, during the *Variable Elimination Phase* MBE operates, in order, on the variables x_4, x_3, x_2, and x_1. When x_4 is processed, the bucket $B_4 = \{f(x_1, x_4), f(x_2, x_4), f(x_3, x_4)\}$—illustrated by the red edges in Fig. 3(a) top— would result in aggregated bucket function whose arity is 3, and thus exceeds the maximal arity allowed. Thus, MBE creates a partition $\{B_{4_1}, B_{4_2}, B_{4_3}\}$ for B_4, whose sets consists of the functions, respectively, $f(x_1, x_4), f(x_2, x_4)$, and $f(x_3, x_4)$. The resulting functions $\hat{f}_{4_1}, \hat{f}_{4_2}$, and $\hat{f}_{4_3}$ have arity 1, as illustrated in Fig. 3(a) bottom. Then, MBE updates the sets $\mathbf{X}$ to $\{x_1, x_2, x_3\}$ and $\mathbf{C}$ to $\{f(x_1, x_2), f(x_2, x_3), \hat{f}_{4_1}, \hat{f}_{4_2}, \hat{f}_{4_3}\}$, as shown in the primal graph of Fig. 3(b) top. When x_3 is processed, $B_3 = \{f(x_2, x_3), \hat{f}_{4_3}\}$, marked red in Fig. 3(b) top, and the
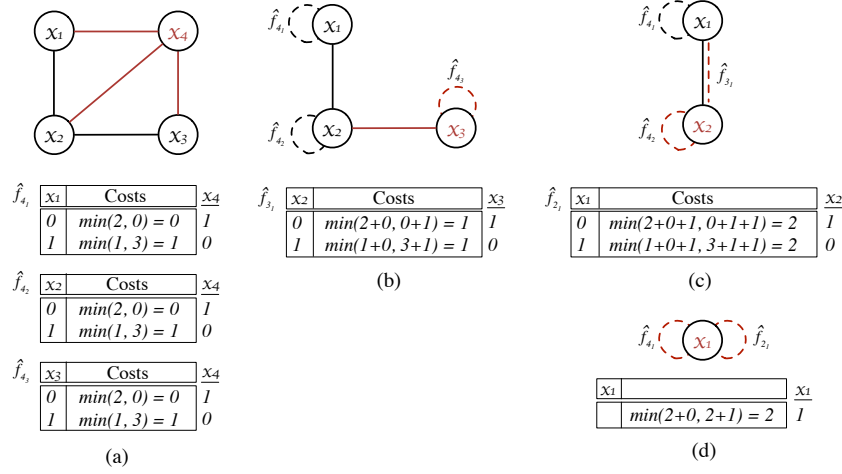
Fig. 3: Mini-Bucket Elimination steps for the WCSP of Fig. 1 :

mini-bucket $B_{3_1} = B_3$. The resulting bucket function $\hat{f}_{3_1}$ is shown in Fig. 3(b) bottom. Thus, $\mathbf{X} = \{x_1, x_2\}$ and $\mathbf{C} = \{f(x_1, x_2), \hat{f}_{4_1}, \hat{f}_{4_2} \hat{f}_{3_1}\}$. Next, x_2 is processed; $B_2 = B_{2_1} = \{f(x_1, x_2), \hat{f}_{4_2}, \hat{f}_{3_1}\}$, and $\hat{f}_{2_1}$ is illustrated in Fig. 3(c) bottom. Thus, $\mathbf{X} = \{x_1\}$ and $\mathbf{C} = \{\hat{f}_{4_1}, \hat{f}_{2_1}\}$. Lastly, the algorithm processes x_1, sets $B_1 = B_{1_1} = \{\hat{f}_{4_1}, \hat{f}_{2_1}\}$, and $\hat{f}_{1_1}$ is minimized when $x_1 = 1$, as shown in Fig. 3(d) bottom. The *Value Assignment Phase* is analogous to the process carried by BE, except that when processing variable x_4 MBE assigns it the value 1, since it minimizes $\hat{f}_{4_1} + \hat{f}_{4_2} + \hat{f}_{4_3}$ when $x_1 = 0, x_2 = 1$, and $x_3 = 0$ (Fig. 2(a) bottom). Thus, $\sigma^* = \langle 0, 1, 0, 1 \rangle$ is the reported solution to the problem, with a lower bound cost of 2.

3 Background: Belief Networks and Most Probable Explanation

A **belief network** (BN) [50] is a tuple $\langle \mathbf{X}, \mathbf{D}, \mathbf{P} \rangle$, where $\mathbf{X} = \{x_1, \ldots, x_n\}$ is a set of ordered variables defined over finite domains $\mathbf{D} = \{D_{x_1}, \ldots, D_{x_n}\}$, with o an ordering of the variables in $\mathbf{X}$, and $\mathbf{P}$ is a set of *conditional probability tables (CPTs)*. A CPT $f_i = \{\Pr(x_i|pa_i)\}$ of $\mathbf{P}$ denotes the join probability of x_i with respect to the variables in pa_i, and $pa_i \subseteq \{x_j \in \mathbf{X} | x_i \prec_o x_j\}$ is the set of variables with higher priority of x_i in the ordering o, also called *parent* variables of x_i. A BN B represents the probability distribution over the variables in $\mathbf{X}$:

$$\Pr_B(\sigma) = \prod_{i=1}^{n} \Pr(x_i|pa_i),$$

where σ is a complete assignment for the variables in $\mathbf{X}$. The *scope* of a CPT $f_i \in \mathbf{P}$ is the set $\mathbf{x}^i = \{x_i\} \cup pa_i$. An **evidence set** σ_E is an assigned subset of variables $E \subseteq \mathbf{X}$.

A BN B is represented through a directed acyclic graph $G_B = (\mathbf{X}, E_P)$, where $(y, x_i) \in E_P$ iff $\exists f_i \in \mathbf{P}$ such that $y \in pa_i$. In other words, E_P is the set of all directed arcs from each parent variable of x_i to x_i, for every $x_i \in \mathbf{X}$. The primal graph of a BN is called **moral graph** and it connects any two variables appearing in the same CPT.
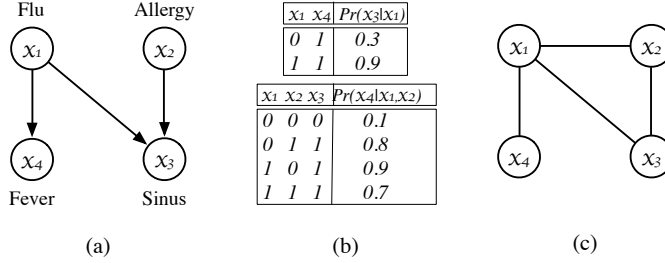

x_1	x_4	$Pr(x_3|x_1)$
0	1	0.3
1	1	0.9

x_1	x_2	x_3	$Pr(x_4|x_1,x_2)$
0	0	0	0.1
0	1	1	0.8
1	0	1	0.9
1	1	1	0.7

(a)   (b)   (c)

Fig. 4: Example of a Belief Network : **(a):** Belief Network: $Pr(x_4, x_3, x_2, x_1) = Pr(x_4|x_1)Pr(x_3|x_1, x_2)Pr(x_2)Pr(x_1)$. **(b):** conditional probability tables. **(c):** Its moral graph.

Example 5 Fig. 4(a) illustrates a simple belief network with 4 binary variables: x_1, x_2, x_3 and x_4, representing respectively the observations for a patient to have *flu*, *allergy*, *fever*, and *sinus infection*, and 4 CPTs, each associated to a node and its parent nodes. For example, the CPT table illustrated in Fig. 4(b) (top), describes the probability the patient has fever given that she/he does has have *flu*. The conditional probability for $x_4 = 0$ is implied since the probabilities need to sum up to 1. Similarly, the CPT table of Fig. 4(b) (bottom) describes the probabilities the patient has sinus infection for each combination of outcomes for flu and allergy. The BN represents the joint probability distribution:

$$\forall x_1, x_2, x_3, x_4, \quad Pr(x_4, x_3, x_2, x_1) = Pr(x_4|x_1)Pr(x_3|x_1, x_2)Pr(x_2)Pr(x_1).$$

Its moral graph is illustrated in Fig. 4(c).

One of the main tasks posed over belief networks is that of finding the **maximum probably explanation** (MPE). Given a BN B, and an evidence set E, finding the MPE correspond to finding a complete assignment for the variables of B that has the maximal probability given the evidence E. More formally, the goal of the MPE is that of finding a complete assignment σ^* such that:

$$\sigma^* = \underset{\sigma \in \Sigma}{\mathrm{argmax}} \prod_{f_i \in \mathbf{P}} \Pr(x_i|\sigma_{pa_i}, E) \tag{2}$$

where σ_{pa_i} is the projection of σ to the variables in pa_i.

This problems can be solved with small variations of the Bucket Elimination algorithm presented in section 2.1. Bucket Elimination can be adapted to solve MPE

tasks on belief networks where the min *operator* in the projection within the elimination operation is substituted by max *operator* and the *summation* in the aggregation operator is substituted by the *product* (for more details we refer the reader to [16]).

4 Background: Distributed Constraint Optimization Problems (DCOPs)

In a *Distributed Constraint Optimization Problem* (DCOP) [47,53,67], the variables, domains, and cost functions of a WCSP are distributed among a collection of *agents*. A DCOP is defined as $\langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{A}, \alpha \rangle$, where $\mathbf{X}, \mathbf{D}$, and $\mathbf{C}$ are defined as in a WCSP, $\mathbf{A} = \{a_1, \ldots, a_p\}$ is a set of *agents*, and $\alpha : \mathbf{X} \to \mathbf{A}$ maps each variable to one agent. Following common conventions, we assume that α is a bijection: Each agent controls exactly one variable. Thus, we will use the terms "variable" and "agent" interchangeably and assume that $\alpha(x_i) = a_i$. In DCOPs, solutions are defined as for WCSPs, and many solution approaches emulate those proposed in the WCSPs literature. For example, ADOPT [47] is a distributed version of *Iterative Deepening Depth First Search*, and DPOP [53] is a distributed version of BE. The main difference is in the way the information is shared among agents. Typically, a DCOP agent knows exclusively its domain and the functions involving its variable. It can communicate exclusively with its neighbors (i.e., agents directly connected to it in the primal graph[3]), and the exchange of information takes the form of messages. Given a DCOP P, and a DFS pseudo-tree T for the primal graph G_P, we use $N(a_i) = \{a_j \in \mathbf{A} \mid \{x_i, x_j\} \in E_{\mathbf{C}}\}$ to denote the **neighbors** of agent a_i; and $sep(a_i)$ to denote the **separator** of agent a_i, which is the set of ancestor agents that are constrained (i.e., they are linked in G_P) with agent a_i or with one of its descendant agents in the pseudo-tree T.

Example 6 Fig. 1(a–b) illustrate an example of a DCOP instance with 4 agents, a_i ($i \in \{1 \ldots, 4\}$), each controlling one variable, x_i. The problem variables, domains and constraints are analogous to those of the WCSP of Example 1. Fig. 1(d) shows one possible pseudo-tree for the DCOP instance, where the agents a_1 and a_2 have one pseudo-child: a_4. The dotted lines represent backedges.

4.1 Dynamic Programming Optimization Protocol (DPOP)

DPOP [53] is a dynamic programming based DCOP algorithm that is composed of three phases:

- **Pseudo-tree Generation Phase**. In this phase the agents coordinate the construction of a pseudo-tree, realized through existing distributed pseudo-tree construction algorithms [35].
- *UTIL Propagation Phase*. Each agent, starting from the leaves of the pseudo-tree, computes the optimal sum of costs in its subtree for each value combination of variables in its separator set. The agent does so by aggregating the costs of its

---

[3] The *primal graph* of a DCOP is equivalent to that of the corresponding WCSP.

functions with the variables in its separator and the costs in the *UTIL* messages received from its child agents, and then eliminating its own variable.

- *VALUE **Propagation Phase***: Each agent, starting from the root of the pseudo-tree, determines the optimal value for its variable. The root agent does so by choosing the value of its variable from its *UTIL* computations—selecting the value with the minimal cost. It sends the selected value to its children in a *VALUE* message. Each agent, upon receiving a *VALUE* message, determines the value for its variable that results in the minimum cost given the variable assignments (of the agents in its separator) indicated in the *VALUE* message. Once determined, such assignment is further propagated to the children via *VALUE* messages.

Example 7 In our example problem, after coordinating to construct the pseudo-tree (Fig. 1(d)), agent a_4, being the leaf of the pseudo-tree, starts the *UTIL* propagation phase, by computing the optimal cost for each value combination of variables x_1, x_2, and x_3 (Fig. 2(a)(bottom)), and sending the costs to its parent agent a_3 in a *UTIL* message. Upon receiving the *UTIL* messages from each of its children, agents a_3 and a_2 follow an analogous process. When the root agent a_1 receives the *UTIL* message from each of its children, it computes the minimum cost of the entire problem, and starts the *VALUE* propagation phase. It selects the value for its variable that minimizes the problem cost (Fig. 2(d)(bottom)) and sends this value down to the pseudo-tree to its child, a_3, in a *VALUE* message. Upon receiving a *VALUE* message from its parent, each agents follows the same process.

The time and the space complexities of DPOP are dominated by the *UTIL Propagation Phase*, which is exponential in the size of the largest separator set $sep(a_i)$ for all $a_i \in \mathbf{A}$. The other two phases require a polynomial number of linear sized messages (in the number of variables of the problem), and the complexity of the local operations is at most linear in the size of the domain [53].

Observe that the *UTIL Propagation Phase* of DPOP emulates the *Variable Elimination Phase* of BE in a distributed context [10]. Given a pseudo-tree and its ordering o, the *UTIL* message generated by each DPOP agent a_i is equivalent to the aggregated and projected function $\hat{f}_i$ in BE when x_i is processed according to the ordering o.

4.2 Approximate Distributed Pseudotree Optimization

Analogously to how DPOP emulates BE in the distributed context, the *Approximate Distributed Pseudotree Optimization* (ADPOP) algorithm emulates MBE to solve DCOPs [52]. ADPOP has the same three phases as DPOP, and given a pseudo-tree and its ordering o, the content of the *UTIL* messages generated by each ADPOP agent a_i is equivalent to the bucket functions $\hat{f}_{i_j}$ ($j \in \{1, \ldots, i_m\}$) in MBE when x_i is processed according to the ordering o.

The complexity of ADPOP is exponential in the input parameter z, while its *VALUE Propagation Phase* has the same order complexity of the *VALUE Propagation Phase* in DPOP.
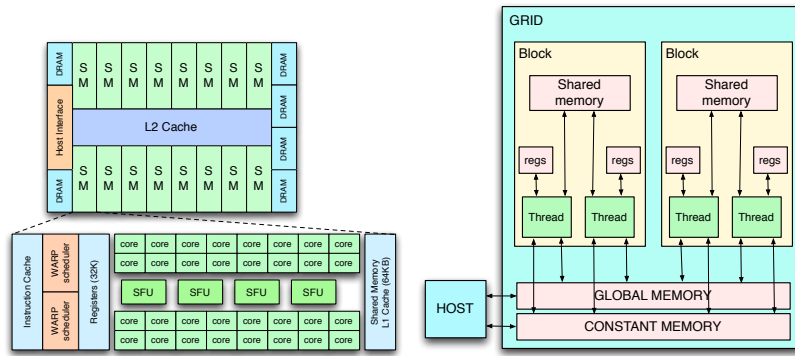
Fig. 5: Fermi Hardware Architecture (left) and CUDA Logical Architecture (right)

5 Background: Graphical Processing Units (GPUs)

Modern *Graphics Processing Units* (GPUs) are massive parallel architectures, offering thousands of computing cores and a rich memory hierarchy to support graphical processing (e.g., DirectX and OpenGL APIs). NVIDIA's *Compute Unified Device Architecture* (CUDA) [58] aims at enabling the use of the multiple cores of a graphic card to accelerate *general purpose* (non-graphical) applications by providing programming models and APIs that enable the full programmability of the GPU. The computational model supported by CUDA is *Single-Instruction Multiple-Threads* (SIMT), where multiple threads perform the same operation on multiple data points simultaneously.

A GPU is constituted by a series of *Streaming MultiProcessors* (SMs), whose number depends on the specific characteristics of each class of GPU. For example, the Fermi architecture provides 16 SMs, as illustrated in Fig. 5(left). Each SM contains a number of computing cores, each of which incorporate an ALU and a floating-point processing unit. Fig. 5(right) shows a typical CUDA logical architecture. A CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the ***host***) and parts meant for parallel execution on the GPU (referred as the ***device***). A parallel computation is described by a collection of ***GPU kernels***, where each kernel is a function to be executed by several ***threads***. When mapping a kernel to a specific GPU, CUDA schedules groups of threads (***blocks***) on the SMs. In turn, each SM partitions the threads within a block in ***warps***[4] for execution, which represents the smallest work unit on the device. Each thread instantiated by a kernel can be identified by a unique, sequential, identifier (T_{id}), which allows to differentiate both the data read by each thread and code to be executed.

---

[4] A warp is typically composed of 32 threads.
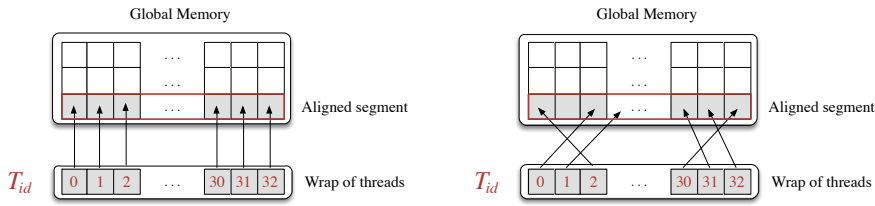
Fig. 6: Coalesced (left) and scattered (right) data access patterns.

5.1 Memory Organization

GPU and CPU are, in general, separate hardware units with physically distinct memory types connected by a system bus. Thus, in order for the device to execute some computation invoked by the host and to return the results back to the caller, a data flow needs to be enforced from the host memory to the device memory and vice versa. The device memory architecture is quite different from that of the host, in that it is organized in several levels differing to each other for both physical and logical characteristics.

Each thread can utilize a small number of *registers*,[5] which have thread lifetime and visibility. Threads in a block can communicate by reading and writing a common area of memory, called ***shared memory***. The total amount of shared memory per block is typically 48KB. Communication between blocks and communication between the blocks and the host is realized through a large ***global memory***. The data stored in the global memory has global visibility and lifetime. Thus, it is visible to all threads within the application (including the host), and lasts for the duration of the host allocation.

Apart from lifetime and visibility, different memory types have also different dimensions, bandwidths, and access times. Registers have the fastest access memory, typically consuming zero clock cycles per instruction, while the global memory is the slowest but largest memory accessible by the device, with access times ranging from 300 to 600 clock cycles. The shared memory is partitioned into 32 logical banks, each serving exactly one request per cycle. Shared memory has an extremely small access latency, provided that multiple thread memory accesses are mapped to different memory banks.

5.2 Bottlenecks and Common Optimization Practices

While it is relatively simple to develop correct GPU programs (e.g., by incrementally modifying an existing sequential program), it is nevertheless challenging to design an efficient solution. Several factors are critical in gaining performance. In this section, we discuss a few common practice that are important for the design of efficient CUDA programs.

---

[5] In modern devices, each SM allots 64KB for registers space.

Memory bandwidth is widely considered to be an important bottleneck for the performance of GPU applications. Accessing global memory is relatively slow compared to accessing shared memory in a CUDA kernel. However, even if not cached, global accesses covering a contiguous 128 Bytes data are fetched at once. Thus, most of the global memory access latency can be hidden if the GPU kernel employs a *coalesced* memory access pattern. Fig. 6(left) illustrates an example of coalesced memory access pattern, in which aligned threads in a warp accesses aligned entries in a memory *segment*, which results in a single transaction. Thus, coalesced memory accesses allow the device to reduce the number of fetches to global memory for every thread in a warp. In contrast, when threads adopt a *scattered* data accesses (Fig. 6(right)), the device serializes the memory transaction, drastically increasing its access latency.

Data transfers between the host and device memory is performed through a system bus, which translates to slow transactions. Thus, in general, it is convenient to store the data onto the device memory. Additionally, batching small memory transfers into a large one will reduce most of the per-transfer processing overhead [58].

The organization of the data in data structures and data access patterns play a fundamental role in the efficiency of the GPU computations. Due to the computational model employed by the GPU, it is important that each thread in a warp executes the same branch of execution. When this condition is not satisfied (e.g., two threads execute different branches of a conditional construct), the degree of concurrency typically decreases, as the execution of threads performing separate control flows can be serialized. This is referred to as *branch divergence*, a phenomenon that has been intensely analyzed within the *High Performance Computing* (HPC) community [36, 14, 20].

6 GPU-based (Distributed) Bucket Elimination (GPU-(D)BE)

Our *GPU-based (Distributed) Bucket Elimination* (GpuBE) framework, extends BE and MBE (DPOP and ADPOP, respectively) by exploiting GPU parallelism within the *aggregation* and *elimination* operations. These operations are responsible for the creation of the functions $\hat{f}_i$ in BE and $\hat{f}_{i_k}$ in MBE (lines 3 and 5 of Algorithms 1 and 2, respectively) and the *UTIL* tables in DPOP and ADPOP (*UTIL Propagation Phase*), and they dominate the complexity of the algorithms. Thus, we focus on the details of the design and the implementation relevant to such operations. The key observation that allows us to parallelize these operations is that the computation of the cost for each value combination in a bucket function is independent of the computation in the other combinations. The use of a GPU architecture allows us to exploit such independence, by concurrently exploring several value combinations of the bucket function, computed by the aggregation operator, as well as concurrently eliminating out variables.

Due to the equivalence of BE (resp. MBE) and DPOP (resp. ADPOP), we will refer to the *bucket* functions $\hat{f}$ and *UTIL tables* resulted by the aggregation and elimination operations of Algorithms 1 and 2, as well as variables and agents, interchangeably.

6.1 GPU Data Structures

In order to fully capitalize on the parallel computational power of GPUs, the data structures need to be designed in such a way to limit the amount of information exchanged between the CPU host and the GPU device, minimizing the accesses to the (slow) device global memory, while ensuring that the data access pattern enforced is coalesced. To do so, we store into the device global memory exclusively the minimal information required to compute the bucket functions, which are communicated to the GPU once at the beginning of the computation of each bucket or mini-bucket. This allows the GPU kernels to communicate with the CPU host exclusively to exchange the results of the aggregation and elimination processes.

We introduce the following concept:

Definition 5 (Bucket-table) A *bucket-table* is a 4-tuple, $T = \langle \mathbf{S}, \mathbf{R}, \chi, \prec \rangle$, where:

- $\mathbf{S} \subseteq \mathbf{X}$, is a list of variables denoting the *scope* of T.
- $\mathbf{R}$ is a list of tuples of values, each tuple having length $|S|$. Each element in this list (called *row* of T) specifies an assignment of values for the variables in $\mathbf{S}$ that is consistent with their domains. We denote with $\mathbf{R}[i]$ the tuple of values corresponding to the i-th row in $\mathbf{R}$, for $i = \{1, \ldots, |\mathbf{R}|\}$.
- χ is a list of length $|R|$ of cost values corresponding to the costs of the assignments in $\mathbf{R}$. In particular, the element $\chi[i]$ represents the cost of the assignment $\mathbf{R}[i]$ for the variables in $\mathbf{S}$, with $i = \{1, \ldots, |\mathbf{R}|\}$.
- $\prec$ denotes an ordering relation used to sort the variables in the list $\mathbf{S}$. In turn, the value assignments, and cost values, in each row of $\mathbf{R}$ and χ, respectively, obey to the same ordering.

As a technical note, a bucket table T is mapped onto the GPU device to store exclusively the cost values χ, not the associated variables values. We assume that the rows of $\mathbf{R}$ are sorted in lexicographic order—thus, the i-th entry $\chi[i]$ is associated with the i-th permutation $\mathbf{R}[i]$ of the variable values in $\mathbf{S}$, in lexicographic order. This strategy allows us to employ a simple perfect hashing to efficiently associate row numbers with variables' values. We will elaborate on this topic in Section 6.3. Additionally, all the data stored on the GPU global memory is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*.

6.2 Algorithm Overview

Algorithm 3 illustrates the pseudocode of GpuBE, where z is an input parameter denoting the maximal mini-bucket size to be processed. We use the following notations:

- Starred line numbers denote those instructions that are executed concurrently by both the CPU and the GPU.
- The symbols $\leftarrow$ and $\Leftarrow$ denote sequential and parallel (i.e., multiple GPU threads) operations, respectively.
- If a parallel operation requires a copy from host (device) to device (host), we write $\overset{D \leftarrow H}{\Leftarrow}$ ($\overset{H \leftarrow D}{\Leftarrow}$). Host to device (device to host) memory transfers are performed immediately before (after) the execution of the GPU kernel.

Algorithm 3: GPUBE(z)

/* Variable Ordering Phase (Pseudo-Tree Construction)   */

1   $\bar{\mathbf{X}} \leftarrow$ Sort $\mathbf{X}$ w.r.t. $\prec_T$ ordering

   /* Variable Elimination Phase   */

2   **for** $i \leftarrow n$ **downto** 1, *with* $x_i \in \bar{\mathbf{X}}$ **do**

3      $B_i \leftarrow$ CPU::CONSTRUCTBUCKET$(\mathbf{C}, x_i, z)$

4   Let $\{B_{i_1}, \ldots, B_{i_m}\}$ be a partition of B_i s.t. $\left| \bigcup_{f_j \in B_{i_k}} \mathbf{x}^j \right| \leq z$, for each $k = 1, \ldots, m$

5   **foreach** $k \in \{1, \ldots, m\}$ **do**

6         $T_{i_k} = \langle B_{i_k}, \mathbf{R}_{i_k}, \chi_{i_k}, \prec_T \rangle \Leftarrow$ GPU::RESERVE$(|\mathbf{R}_{i_k}|)$

7   **foreach** $f_j \in B_{i_k}$ **do**

8* $T_j = \langle \mathbf{x}^{f_j}, \mathbf{R}_j, \chi_j, \prec_T \rangle \overset{D \leftarrow H}{\Longleftarrow}$ GPU::RESERVE$(|\mathbf{R}_j|)$

9* $T_{i_k} \Leftarrow$ GPU::AGGREGATE(T_{i_k}, T_j)

10         $\hat{f}_{i_k} \overset{H \leftarrow D}{\Longleftarrow}$ GPU::ELIMINATE(T_{i_k}, x_i)

   /* Variable Assignment Phase   */

11   **for** $i \leftarrow 1$ **to** n, *with* $x_i \in \bar{\mathbf{X}}$ **do**

12      $x_i \leftarrow$ CPU::FINDBESTASSIGNMENT$(x_1, \ldots, x_{i-1})$

13   **return** $\hat{f}_1$

     GpuBE is composed of three phases: **(1)** *Variable Ordering*, **(2)** *Variable Elimination*, and **(3)** *Variable Assignment*. Let us consider $N(x_i) = \{x_j \in \mathbf{X} \mid \{x_i, x_j\} \in E_{\mathbf{C}}\}$, defined analogously as for the agents' case. During the first phase (line 1), the problem variables are sorted according to a pseudo-tree ordering relation; in particular, we apply the following heuristics in the construction of the pseudo-tree: $x_i \prec_T x_j$ *iff* $|N(x_i)| < |N(x_j)|$, for every $x_i, x_j \in \mathbf{X}$. For the distributed case, this phase is identical to that of (A)DPOP, where the agents coordinate the construction of a pseudo-tree, using an off-the-shelf message-passing algorithm [35].

     In the second phase, the algorithm processes each variable, in descending order, according to the relation $\prec_T$, and proceeds as in (M)BE:

- The function CPU::CONSTRUCTBUCKET constructs the bucket B_i as illustrated in Algorithm 1, line 2. The algorithm proceeds in creating a partition of this bucket, if required (i.e., if $z < w^*$). This phase differs slightly in the distributed case, where each agent, upon receiving a new bucket function from its descendant agents, inserts it into its bucket set B_i.

- For each mini-bucket B_{i_k} $(k = 1, \ldots, m)$, GpuBE determines and reserves the amount of global memory to be assigned to each associated bucket-table T_{i_k} (line 6). After the GPU::RESERVE function is invoked, a space sufficient to store the bucket-table is allocated, and its cost values χ_{i_k} are initialized to 0.

- Thus, GpuBE aggregates the bucket-table T_j associated to each function f_j in the mini-bucket with the bucket-table T_{i_k} (lines 7–9). To do so, it first creates a bucket-table T_j that encodes the cost values of the bucket function f_j, reordering them, if necessary, according to the order on its scope specified by the pseudo-tree relation $\prec_T$ (line 8). This procedure requires a memory transfer from the CPU host to the GPU device global memory. Then, it adds the values χ_j of the aggregating bucket-table T_j into the corresponding entries of the bucket-table T_{i_k} (line 9). We will

further discuss the details of this function, as well as the other kernel functions, in the next sections.

- Finally, the algorithm invokes a GPU call to eliminate the variable x_i from the bucket-table T_{i_k}, thereby constructing the bucket function $\hat{f}_{i_k}$, which is, finally, copied back to the CPU host memory (line 10).

In the distributed case, each agent processes lines 3–6 in parallel without prior coordination. Starting from the leaves of the pseudo-tree, the agents build their *UTIL* messages containing the bucket functions (lines 5–10), and send them to their parents. Thus, each agent waits to receive the *UTIL* messages from all of its children before performing the aggregation and elimination operations (lines 7–9 and line 10, respectively) for each mini-bucket. By the end of this phase (line 10), the root agent knows the overall cost for each values of its variable x_i. Thus, it chooses the value that results in the minimum cost, and it starts the third phase by sending to each child agent the value of its variable x_i.

In the centralized case, when space is not a concern, there is no need of copying the bucket tables back to the host, after the variable elimination step (line 10). Thus, two memory transfer transactions are avoided for each variable being processed.

In the third phase, the algorithm proceeds analogously to as done in (M)BE. For the distributed case, the agents select the values for their variables that minimize their bucket functions costs, given the assignments of their ancestor agents, and send them in *VALUE* messages to their children. These operations are repeated by every agent receiving a *VALUE* message until the leaf agents are reached.

While we described the case in which the underlying problem primal graph is connected, our implementation allows us to handle disconnected graphs. This is done by solving the sub-problems in each connected subgraph independently from other subproblems, and retrieving the problem cost by aggregating the costs stored in the root of each pseudo-tree associated to the connected graphs.

6.3 GPU-based Constraint Aggregation

We now describe the implementation of the constraint aggregation GPU kernel. This operation, takes as input two bucket-tables: T_{i_k} and T_j, and aggregates the cost values in χ_j to those of χ_{i_k} for all the corresponding assignments of the shared variables in the scope of the two bucket-tables. We refer to T_{i_k} and T_j as to the *output* and *input* bucket-tables, respectively.

Consider the example in Fig. 7, the cost values χ_j of the input bucket-table T_j (right) are aggregated to the cost values χ_{i_k} of the output bucket-table T_{i_k} (left)—which where initialized to 0. The rows of the two tables with identical value assignments for the shared variables x_2 and x_3 are shaded with the same color.

To optimize performance of the GPU operations and to avoid unnecessary data transfer to/from the GPU global memory, we only transfer the list of cost values χ for each bucket-table that need to be aggregated, and employ a simple perfect hashing function to efficiently associate row numbers with variables' values. This allows us to compute the indices of the cost vector of the input bucket-table relying exclusively
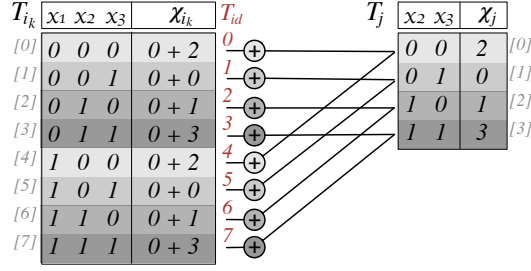
Fig. 7: Example of aggregation of two tables on GPU.

on the information of the thread ID and, thus, avoiding accessing the scope $\mathbf{S}$ and assignment vectors $\mathbf{R}$ of the input and output bucket-tables.

We now discuss how this process can be efficiently handled on the GPU kernels. Let $T^{out} = \langle \mathbf{S}^{out}, \mathbf{R}^{out}, \chi^{out}, \prec^{out} \rangle$ be the output bucket-table, whose scope is $\mathbf{S}^{out} = \{x_1^{out}, \ldots, x_m^{out}\}$. Let $T^{in} = \langle \mathbf{S}^{in}, \mathbf{R}^{in}, \chi^{in}, \prec^{in} \rangle$ be the input bucket-table, whose scope is $\mathbf{S}^{in} = \{x_1^{in}, \ldots, x_s^{in}\}$, and such that $\mathbf{S}^{in} \sqsubseteq \mathbf{S}^{out}$, where $A \sqsubseteq B$ denotes that A is a subsequence of B, and with $s \leq m$. Additionally, let $x_m^{out} = x_s^{in}$, i.e., the last variable of the input and output bucket-table scopes coincides. The latter is the variable to be eliminated; We will explain this design choice in the next section, where we will discuss the variable elimination process on a GPU. Finally, let $\phi_{out} : \mathbb{N} \to \mathbb{N}$ be a mapping from input bucket-table scope variables indexes to output bucket-table scope variable indexes, such that $\phi_{out}(i) = j$ *iff* $x_i^{in} = x_j^{out}$. For instance, in our example of Fig. 7, $\phi_{out}(0) = 1$, as the variable $\mathbf{S}_j[0] = \mathbf{S}_{i_k}[1] = x_2$. Hence, given a row index r_{out} for the output bucket-table χ^{out}, the corresponding row index r_{in} associated to the input bucket-table cost array χ^{in} is given by:

$$r_{in} = \sum_{k=1}^{s-1} \left[\underbrace{\left(\prod_{j=k+1}^{s} |D_{x_j^{in}}| \right)}_{mul[k]} \cdot \left(\underbrace{\lfloor \frac{r_{out}}{\prod_{j=\phi_{out}(k)+1}^{m} |D_{x_j^{out}}|} \rfloor}_{div[k]} \bmod \underbrace{|D_{x_k^{in}}|}_{mod[k]} \right) \right] + r_{out} \bmod \underbrace{|D_{x_s^{in}}|}_{mod[s]} \quad (3)$$

Each term in the summation of Equation (3) represents the contribution of the k-th variable's value in $\mathbf{R}^{out}[r_{out}]$, as an offset to the index r_{in} in the array $\mathbf{R}^{in}$.

The vectors mul, div, and mod are data structures employed to compute efficiently the r_{in} indices on the GPU. The values $mul[k]$, $div[k]$, and $mod[k]$ (and $mod[s]$) can be efficiently computed in $O(s)$, $O(n)$, and $O(1)$, respectively, for each $k = \{1, \ldots, s-1\}$, and copied onto the GPU global memory with one copy transaction—we allocate them as a single mono-dimensional array.

In order to exploit the highest degree of parallelism offered by the GPU device, we **(1)** map one GPU thread T_{id} to one element of the output bucket-table r_{out} and **(2)** adopt the ordering relation $\prec_T$ for each input and output bucket-table processed.

Procedure GPU::AGGREGATE(T_{i_k}, T_j)

1 $r_{i_k} \leftarrow$ the thread's entry ID (T_{id})
2 $r_j \leftarrow 0$ /* holds the value of the index entry of χ_j */
3 $s \leftarrow |\mathbf{S}_j|$
4 $\langle mul, div, mod \rangle \leftarrow$ COPYTOSHAREDMEMORY()
5 **for** $\ell \leftarrow (1 \ldots s-1)$ **do**
6 $\quad \lfloor \quad r_j \leftarrow r_j + mul[\ell] \cdot (\lfloor \frac{r_{i_k}}{div[\ell]} \rfloor) \% mod[\ell]$
7 $r_j \leftarrow r_j + (r_{i_k} \% mod[s])$
8 $\chi_{i_k}[r_{i_k}] \leftarrow \chi_{i_k}[r_{i_k}] + \chi_j[r_j]$

Adopting such techniques allows each thread to be responsible of performing exactly two reads and one write from/to the GPU global memory. Additionally, the ordering relation enforced on the bucket-tables allows us to exploit the locality of data and to encourage coalesced data accesses. As illustrated in Fig. 7, this paradigm allows threads (whose IDs are identified in red by their T_{id}'s) to operate on contiguous chunks of data and, thus, minimizes the number of actual read (from the input bucket-table, on the right) and write (onto the output bucket-table, on the left) operations from/to the global memory performed by a group of threads with a single data transaction.[6]

The constraint aggregation GPU kernel is described in Procedure Gpu::Aggregate, which is computed in parallel by a number of threads equal to the number of rows of the output bucket-table. Each thread identifies its row index r_{i_k} within the output bucket-table cost values array χ_{i_r} based on its thread ID (line 1), and it initializes a variable that will contain the input bucket-table row index to 0 (line 2). It then copies into the shared memory the static entities mul, div, and mod associated to the aggregation of the the bucket-tables being processed (line 4). A further inspection to the Gpu::Aggregate procedure reveals how it makes use of the auxiliary data structures above to efficiently implement the *hash function* of equation (3), and retrieve the entry index of the input bucket-table associated to the variables value permutation of the output bucket-table $\mathbf{R}_{i_k}[r_{i_k}]$ (lines 5–7). Finally, the instruction in line 8 aggregates the corresponding input bucket-table value to the output bucket-table $\chi_{i_k}[r_{i_k}]$.

Note that this algorithm highly fits the SIMT paradigm adopted by GPUs; the thread ID and the auxiliary mul, div, and mod arrays are used to retrieve and update all the data necessary to compute the output bucket-table. Additionally, the accesses to the global memory are minimized, as the auxiliary arrays are copied into the shared memory.

We illustrate the above process in the following example.

Example 8 Consider the operation of aggregating the input bucket-table T_j with the bucket-table T_{i_k} of Fig. 7 corresponding, respectively, to the bucket-table representing the constraint f_{23} and the bucket-table $\hat{f}_3$ (before eliminating the variable x_3) in Fig. 2(b). With the Equation (3) notation, $s = 2$, $m = 3$ and, thus, the index k of the

---

[6] Accesses to the GPU global memory are cached into cache lines of 128 Bytes, and can be fetched by all requiring threads in a warp.

summation ranges from 1 to $s - 1 = 1$. Therefore:

$$mul[0] = \prod_{j=2}^{2} |D_{x_j}| = 2 \qquad div[0] = \prod_{j=\phi_{out}(1)+1=2}^{2} |D_{x_j}| = 2$$

$$mod[0] = |D_{x_2}| = 2 \qquad mod[1] = |D_{x_3}| = 2$$

Therefore, the mapping from the thread IDs (or, equivalently, the output bucket-table row indices r_{i_k}) to the input bucket-table row indices r_j is:

$$T_{id} = 0 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{0}{2} \rfloor) + 0 \bmod 2 = 0$$
$$T_{id} = 1 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{1}{2} \rfloor) + 1 \bmod 2 = 1$$
$$T_{id} = 2 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{2}{2} \rfloor) + 2 \bmod 2 = 2$$
$$T_{id} = 3 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{3}{2} \rfloor) + 3 \bmod 2 = 3$$
$$T_{id} = 4 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{4}{2} \rfloor) + 4 \bmod 2 = 0$$
$$T_{id} = 5 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{5}{2} \rfloor) + 5 \bmod 2 = 1$$
$$T_{id} = 6 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{6}{2} \rfloor) + 6 \bmod 2 = 2$$
$$T_{id} = 7 \quad \Rightarrow \quad r_j = 2 \cdot (\lfloor \tfrac{7}{2} \rfloor) + 7 \bmod 2 = 3$$

As a technical detail, the bucket-tables are created and processed so that the variables in their scope are sorted according to the order $\prec_T$. This means that the variables with the highest priority appear first in the scope list, while the variable to be eliminated always appear last. We will see, in the next section, that such detail allows us to efficiently encode the elimination operation on the GPU.
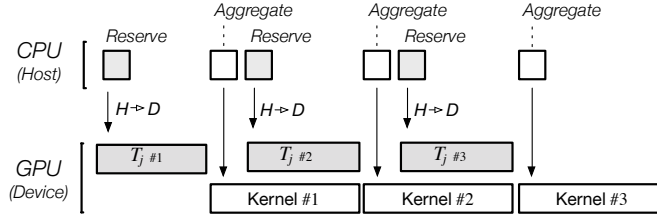
Fig. 8: Concurrent computation between host and device.

To fully capitalize on the use of the GPU, we exploit an additional level of parallelism, achieved by running GPU kernels and CPU computations concurrently (lines 8–9 of Algorithm 3). This is possible when the T_j bucket-tables can be partitioned in multiple chunks. Fig. 8 illustrates this operation. After transferring the first bucket-table chunk ($T_{j\ \#1}$) into the device memory, the process starts the execution of the Gpu::Aggregate() kernel, which operates on this portion of the bucket table (called *Kernel* $_{\#1}$ in Fig. 8). Thus, the control immediately returns to the CPU host, which enforces the next data transfer onto the device memory, through a call to a GPU::RESERVE($T_{j\ \#2}$). A host-device synchronization point is imposed after each memory transfer (except the first one), to ensure that no overlapping Gpu::Aggregate() GPU kernels are enforced.
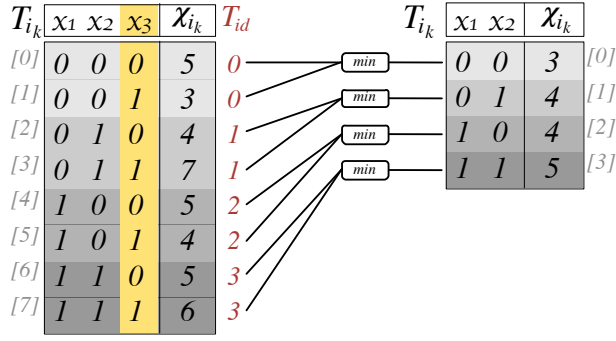
Fig. 9: Example of aggregation of two tables on GPU.

6.4 GPU-based Variable Elimination

We now describe the implementation of the variable elimination GPU kernel. This operation takes as input a bucket-table T_{i_k} and a variable $x_i \in \mathbf{S}_{i_k}$ and removes this variable from the bucket-table's scope, optimizing over its cost rows. As a result, the output bucket-table rows list the unique assignments for the value combinations of $\mathbf{S}_{i_k} \setminus \{x_i\}$ in the input bucket-table $\mathbf{R}_{i_k}$ which minimizes the costs values for each $d \in D_{x_i}$.

Fig. 9 illustrates this process, where the variable x_3 is eliminated from the bucket-table T_{i_k}. The column being eliminated is highlighted yellow in the input bucket-table. The different row colors identify the unique assignments for the remaining variables x_1, x_2, and exposes the high degree of parallelization that is associated to such operation. To exploit this level of parallelization, we adopt a paradigm similar to that employed in the aggregation operation on GPU, where each thread is responsible of the computation of a single output element.

Procedure GPU::ELIMINATE(T_{i_k}, x_i)

1 $r_{i_k} \leftarrow$ the thread's entry ID (T_{id})
2 $r_j \leftarrow r_{i_k} \cdot |D_{x_i}|$ /* holds the value of the index entry of χ_{i_k} */
3 $c^* \leftarrow \chi_{i_k}[r_j]$
4 **for** $\ell \leftarrow (1 \ldots |D_{x_i}|-1)$ **do**
5 $\quad \lfloor \quad c^* \leftarrow \min\{c^*, \chi_{i_k}[r_j + \ell]\}$
6 $\chi_{i_k}[r_{i_k}] \leftarrow c^*$

The variable elimination GPU kernel is described in Procedure Gpu::Eliminate, which is computed in parallel by a number of threads equal to the number of rows of the output bucket-table. Each thread identifies its row index r_{i_k} within the output bucket-table cost values χ_{i_k} (line 1), given its thread ID. It hence sets an input row index r_j to the value of the first χ_{i_k} input bucket-table row to analyze (line 1), and it stores in c^* its associated cost. Note that, as the variable to eliminate is listed last

in the scope of the bucket-table, it is possible to retrieve each unique assignment for the projected output bucket table, simply by offsetting r_{i_k} by the size of D_{x_i}. Additionally, all elements listed in $\chi_{i_k}[r_j], \ldots, \chi_{i_k}[r_j + |D_{x_i}|]$ differ exclusively on the value assignment to the variable x_i (see Fig. 9). Thus, the GPU kernel evaluates the input bucket-table cost values associated to each element in the domain of x_i, by incrementing the row index r_j, $|D_{x_i}| - 1$ times, and chooses the minimum cost value (lines 4–5). At last, it saves to the associated output row the best cost found (line 6).

Note that each thread reads $|D_{x_i}|$ adjacent values of the vector χ_{i_k}, and writes one value in the same vector. Thus, this algorithm **(1)** perfectly fits the SIMT paradigm, **(2)** minimizes the accesses to the global memory as it encourages a coalesced data access pattern, and **(3)** uses a relatively small amount of global memory, as it recycles the memory area allocated for the input bucket-table, to output the cost values for the output bucket-table.

The ordering $\prec_T$ adopted by the bucket-tables makes this procedure effective, by forcing the variables to be eliminated to be always listed as last. Additionally, we note that reordering the bucket-tables scope may be necessary exclusively when constructing the bucket-table associated to the constraints in **C**. Indeed, the bucket-tables constructed by the algorithm preserve this ordering over their scope, since all the problem variables are processed according to the same ordering relation $\prec_T$, guaranteeing that the variables being eliminated are those with lower priority with respect to $\prec_T$. Therefore, no reordering will be required in the bucket functions during the process.

Finally, to reduce the memory transfer time, in addition to the technique described in the previous section, we unrolled the for-loop in lines 7–9 of Algorithm 3. Doing so allows us to process all the bucket-tables within a mini-bucket in a single GPU kernel and to copy them to the device using a single transaction.

7 Theoretical Analysis

We report below a theoretical analysis on the runtime and memory complexity of our GpuBE(z) algorithms. For the distributed case, we report results on the network load and messages size complexity provided by the proposed algorithms. The *network load* and *messages size* are defined, respectively, as the total number of messages exchanged by the agents and as the size of the largest message exchanged by the agents during problem resolution. Since our algorithms rely on an inference-based procedure, the agent's complexity (i.e., the maximal number of operations performed by the agents while solving the problem) is equivalent to the size of the largest message exchanged. In turn, the latter corresponds to the memory complexity of the algorithm. We use *GpuBE(w^*)* and *GpuDBE(w^*)* to denote our GPU versions of BE and DPOP, respectively, and *GpuBE(z)* and *GpuDBE(z)* to denote our GPU versions of MBE and ADPOP, respectively, with mini-bucket size z.

Theorem 1 *For a problem P, given an ordering $\prec_T$ on the primal graph G_P, the (mini-)bucket tables (resp. UTIL messages) constructed by GpuBE(z) (resp. the GpuDBE(z) agents) are identical to those constructed by (M)BE (resp. the (A)DPOP agents), for $z \leq w^*$.*

Proof The proof follows from the observation that GpuBE(z) and (M)BE are executed on the same induced graph G_P^*. Thus, the problem variables are processed in the same order by both versions of the algorithms—lines 1 and 6 of Algorithm 1 (1 and 8 of Algorithm 2) for (M)BE, and lines 2 and 11 of Algorithm 3 for GpuBE(z). Analogously, in GpuDBE(z) and (A)DPOP, agents operate on the same pseudo-tree ordering.

For the centralized case, during the *Variable Elimination Phase*, the bucket construction and mini-bucket partitioning operations of GpuBE(z) (lines 3–4 of Algorithm 3) are identical to those of MBE (lines 2–3 of Algorithm 2). For each mini-bucket B_{i_j} in MBE, the operations to create the bucket function $\hat{f}_{i_k}$ are identical in both algorithms: the effect of invoking the Gpu::Aggregate(T_{i_k}, T_j) routine, in GpuBE, for each bucket-table T_{i_k}, corresponding to the bucket function f_{i_k} (lines 7–9 of Algorithm 3), is analogous to the aggregation operations performed in MBE: $F = \sum_{f_j \in B_{i_k}} f_j$ (line 5 of Algorithm 2, in parenthesis), and the effect of the Gpu::Eliminate(T_{i_k}, x_i) routine, which projects the variable x_i onto the scope of T_{i_j}, produces the bucket function $\hat{f}_{i_k}$, which in turn correspond to the elimination operation performed by MBE: $\pi_{-x_i}(F)$ (line 5 of Algorithm 2). For the distributed cases, both ADPOP and GpuDBE(z) agents perform the same operations described above—during the *UTIL Propagation Phase*—and populate the *UTIL* messages they send to their parent. The equivalence between the Variable Elimination and *UTIL Propagation Phases* of BE and DPOP, with the respective phases in GpuBE(w^*) and GpuDBE(w^*), respectively, follows from the process described above differing exclusively in that partitioning B_i produces a single bucket with the same functions as those listed in B_i.

The operations performed during the *Variable Assignment Phases* for (M)BE and GpuBE(z) (lines 5-7, Algorithm 1, for BE , lines 8–9, Algorithm 2, for MBE, and lines 11–12, Algorithm 3, for GpuBE(z)) are identical. Additionally, the variables are processed in the same order in both algorithms. Thus, the solution assignment for the problem variables returned by (M)BE and GpuBE are identical. Similarly, for the distributed case, (A)DPOP and GpuDBE(z) agents perform the same *VALUE* Propagation phase. □

Corollary 1 *For a given $z \leq w^*$, the time and memory (message size) requirements of Gpu(D)BE(z) are, in the worst case, in $O(d^{z+1})$, and $O(d^z)$, respectively, where $d = \max_{x_i \in \mathbf{X}} D_{x_i}$.*

Proof This result follows from the equivalence of the *Variable Elimination Phases* of (M)BE and GpuBE(z), and of the *UTIL Propagation Phases* of (A)DPOP and GpuDBE(z). During these phases, the construction of the (mini)-buckets requires to save, in the worst case, all possible combinations for the value assignments of the bucket-function with bounded arity z. Thus, they require $O(d^z)$ space. Similarly, for the distributed case, due to the equivalence of (A)DPOP and GpuDBE(z), the largest message exchanged by the agents has size $O(d^z)$.

Additionally, the total amount of operations (or, equivalently, bucket-tables rows) that can be processed in parallel during the GPU-based Constraint Aggregation and GPU-based Variable Elimination steps, is bounded by a constant value which de-

pends on the GPU card characteristic. Thus, the time complexity of GpuDBE(z) is in $O(d^{z+1})$. $\square$

Corollary 2 *The network load required for GpuDBE(z) is equivalent to the network load required by (A)DPOP.*

Proof This result follow from the equivalence of DPOP with GpuDBE(w^*) and ADPOP(z) with GpuBE(z) (Theorem 1). Since (A)DPOP requires each agent to send one *UTIL* message to its parent and one *VALUE* message to each of its children, there are a total of $n-1$ *UTIL/VALUE* messages exchanged—one through each tree-edge of the pseudo-tree T_P. Thus, the network load required by (A)DPOP and GpuDBE(z) is in $O(n)$. $\square$

Corollary 3 *Gpu(D)BE is correct and complete.*

Proof The correctness and completeness of GPU-(D)BE(w^*) follow from the correctness and completeness of BE [15] and DPOP [53], and Theorem 1. $\square$

8 Experimental Results

In this section, we evaluate our GPU implementations of BE and MBE (GpuBE) as well as our GPU implementations of DPOP and ADPOP (GpuDBE) and compare them with their CPU counterparts.[7]

Experiments for GpuDBE and (A)DPOP are conducted using a multi-agent DCOP simulator that simulates the concurrent activities of multiple agents, whose actions are activated upon receipt of a message. All algorithms use the same variable ordering in the centralized case and pseudo-tree in the distributed case. Performance of the centralized algorithms are evaluated using the algorithms' wallclock runtime, while the performance of distributed algorithms are evaluated using the *simulated runtime* metric [64]. The main focus of the evaluation is on runtime and speedup achieved by the GPU implementations with respect to their CPU counterparts. Additionally, to compare the quality of the solution bounds reported by the incomplete algorithms, we also report the best solution quality found within the given time limits by *toulbar2* [3], an optimized, exact centralized solver for WCSPs. Toulbar2 is a state-of-the-art solver that uses a depth-first branch-and-bound process to identify a minimum cost assignment and employs the notion of *soft local consistency* to prune the search space using the problem lower bound.

Our experiments are conducted on an *AMD Opteron 6276* with a 2.3GHz CPU and is equipped with a GPU device *GeForce GTX TITAN* with 14 multiprocessors, 2688 cores, with a clock rate of 837MHz, and 6GB of global memory.

We performed our experiments on both randomly generated instances on different networks topologies and on standard WCSP benchmarks.[8] We first analyze the

---

[7] Our source code is available at https://github.com/nandofioretto/GpuBE, and https://github.com/nandofioretto/GpuDBE

[8] Downloadable from http://costfunction.org/en/benchmark/ and http://graphmod.ics.uci.edu/group/Repository

runtimes of the CPU and GPU versions of BE and DPOP on randomly generated instances, where we report the runtimes and lower bounds of the GPU and CPU versions of MBE and ADPOP at varying of the bucket size z. Then, to ensure that the speedups are not due to a specific GPU device configuration, we compare the CPU and GPU speedups achieved on 3 distinct GPU architectures, characterized by different clock rates, number of SMs, and memory sizes. Finally, we report the solving time and lower bounds of our GpuBE on an extensive set of WCSP benchmarks to verify the generality of the speedups across different domains. Each solver has 1-hour timeout of wallclock time in the centralized case and a 1-hour timeout of simulated time in the distributed case. Additionally, they have a memory limit of 32GB to solve each problem instance. Results are averaged over all instances. If a solver fails to solve an instance is due to either memory limits (labeled *oom*) or timeout (labeled *oot*).

8.1 Binary Random Networks

The instances for each binary network topology are generated as follows:
- **Random:** We create an n-node network, whose density p_1 produces $\lfloor n(n-1)p_1 \rfloor$ edges in total. We do not bound the tree-width, which is based on the underlying graph and randomly generated.
- **Scale-free:** We create an n-node network based on the *Barabasi-Albert model* [6]. Starting from a connected 2-node network, we repeatedly add a new node, randomly connecting it to two existing nodes. In turn, these two nodes are selected with probabilities that are proportional to the numbers of their connected edges. The total number of edges is $2(n-2)+1$.
- **Grid:** We create an n-node network arranged as a rectangular grid, where each internal node is connected to four neighboring nodes, while nodes on the grid perimeter are connected to three neighboring nodes unless they are at the corner of the grid, in which case they are connected to two neighboring nodes.

We generate 50 instances for each topology, ensuring that the underlying graph is connected. The cost functions are generated using random integer costs in $[0, 100]$, and the constraint tightness (i.e., ratio of entries in the cost table that have a cost of ∞) p_2 is set to 0.5 for all experiments. We set the following as default parameters: For the random and scale-free topology, $n = 10$, $d = \max_{D_i \in \mathbf{D}} |D_i| = 10$, and $p_1 = 0.3$, and for the grid topology, $\sqrt{n} = 10$.

Tables 1–3 report the runtime, in seconds, for random, scale-free, and grid topologies, respectively, varying the number of variables (resp. agents) for the centralized (resp. distributed) algorithms, the size of the variables domains, and the constraint tightness of the primal graph. The first four (three) columns of Table 1, (2 and 3) describe the problem setting adopted for each experiment. The induced width w^* is averaged across all instances. All other columns report the average runtime and GPU vs. CPU speedup in parenthesis. We make the following observations:
- The GPU-based inference-algorithms are consistently faster that their CPU counterparts, with speedups of up to 307x. Only two exceptions arise for the random networks, where in the small instances with $n = 10$, $d = 5$, $p_1 = 0.3$, and $n = 10$,

Problem				BE			DPOP		
n	d	p_1	w^*	CPU	GPU	speedup	CPU	GPU	speedup
10	10	0.3	2.9	0.019	0.002	10.5	0.007	0.001	7.20
11	10	0.3	3.2	0.031	0.002	13.6	0.013	0.001	13.0
12	10	0.3	3.6	0.069	0.003	25.7	0.028	0.001	28.5
13	10	0.3	4.3	0.413	0.005	79.4	0.210	0.002	116
14	10	0.3	4.4	0.631	0.006	98.6	0.214	0.002	134
15	10	0.3	5.3	4.190	0.026	158	1.609	0.009	187
16	10	0.3	5.8	32.29	0.189	171	9.848	0.049	202
17	10	0.3	6.4	65.41	0.328	200	28.14	0.138	204
18	10	0.3	7.5	206.1	0.944	218	103.0	0.483	213
19	10	0.3	8.0	602.1	2.541	237	470.2	2.019	233
20	10	0.3	8.5	675.3	3.145	215	508.9	2.160	236
10	5	0.3	3.0	0.001	0.002	0.56	0.001	0.001	0.80
10	10	0.3	2.9	0.019	0.002	10.5	0.007	0.001	7.20
10	25	0.3	2.8	0.227	0.004	55.4	0.092	0.001	92.3
10	50	0.3	2.9	24.81	0.095	262	13.99	0.048	291
10	100	0.3	2.9	67.59	0.220	308	35.22	0.118	299
10	10	0.2	2.0	0.001	0.001	0.62	0.001	0.001	0.94
10	10	0.3	2.9	0.019	0.002	10.5	0.007	0.001	7.20
10	10	0.4	3.8	0.094	0.002	40.7	0.042	0.001	42.5
10	10	0.5	4.5	0.525	0.005	105	0.234	0.002	130
10	10	0.6	5.4	3.378	0.019	176	1.941	0.011	176
10	10	0.7	5.9	14.86	0.072	205	10.00	0.053	189
10	10	0.8	6.7	56.23	0.246	228	31.29	0.147	213
10	10	0.9	7.6	72.32	0.312	232	42.47	0.201	211

Table 1: Random networks.

$d = 10$, $p_1 = 0.2$, the GPU versions of the algorithms are slower than their CPU counterparts.

- The speedup increases with the problem size. In particular, the speedup increases with increasing induced width and with increasing domain size of the problem variables. Both these factors influence the size of the bucket-tables to be processed.[9] This observation corroborates the effectiveness of the GPU parallelism exploited in the construction of these tables.

- As expected, the inference-based algorithms are unable to process instances characterized by large induced widths or large domain sizes, as the size of the bucket-tables become intractable with the memory limitations. This is evident in the scale-free and grid networks, where the solvers run out of memory for instances with $n \geq 16$ and $d \geq 50$, respectively.

- The simulated runtimes of the DCOP algorithms are consistently smaller than the wallclock runtimes of the WCSPs ones. This is due to the fact that agents in different branches of the pseudo-tree can compute their bucket-tables independently from each other.

- Finally, the speedup trends of the distributed algorithms are similar to those of the centralized algorithms.

---

[9] Recall that BE needs to process bucket-tables whose number of rows is in $O(d^{w^*})$.

Problem			BE			DPOP		
n	d	w^*	CPU	GPU	speedup	CPU	GPU	speedup
10	10	6.3	22.99	0.111	207	13.78	0.064	215
11	10	6.0	25.57	0.120	212	13.21	0.057	231
12	10	6.0	27.96	0.132	212	14.60	0.072	203
13	10	5.9	80.14	0.370	217	36.21	0.174	208
14	10	6.9	78.36	0.339	231	32.50	0.145	223
15	10	8.2	189.4	0.887	213	66.86	0.340	197
16	10	9.2	oom	oom	-	oom	oom	-
17	10	9.5	oom	oom	-	oom	oom	-
18	10	10	oom	oom	-	oom	oom	-
19	10	11	oom	oom	-	oom	oom	-
20	10	12	oom	oom	-	oom	oom	-
10	5	6.8	0.322	0.004	74.8	0.175	0.001	145
10	10	6.3	22.99	0.111	207	13.78	0.064	215
10	25	6.6	242.5	0.888	273	127.9	0.593	216
10	50	6.4	oom	oom	-	oom	oom	-
10	100	6.4	oom	oom	-	oom	oom	-

Table 2: Scale-free networks.

Problem			BE			D-BE		
$\sqrt{n}$	d	w^*	CPU	GPU	speedup	CPU	GPU	speedup
5	10	3.3	0.259	0.005	44.7	0.022	0.001	21.8
6	10	3.7	0.267	0.008	33.4	0.022	0.001	22.6
7	10	3.7	0.515	0.012	42.9	0.037	0.001	30.8
8	10	3.6	0.848	0.018	47.1	0.041	0.001	29.6
9	10	3.9	1.460	0.028	52.0	0.049	0.001	33.7
10	10	4.0	1.881	0.035	34.4	0.054	0.002	31.7
11	10	3.7	1.934	0.040	48.3	0.073	0.002	38.4
12	10	3.8	2.174	0.042	48.4	0.089	0.002	38.7
13	10	3.9	2.430	0.045	42.2	0.102	0.003	37.8
14	10	4.0	2.996	0.055	54.5	0.127	0.003	39.7
15	10	4.0	3.785	0.071	53.3	0.151	0.004	38.7
10	5	3.7	0.043	0.020	2.21	0.001	0.001	1.00
10	10	4.0	1.881	0.035	34.4	0.054	0.002	31.7
10	25	3.9	97.29	0.388	251	2.930	0.011	266
10	50	4.0	oom	oom	-	oom	oom	-
10	100	4.0	oom	oom	-	oom	oom	-

Table 3: Grid networks.

Next, we analyze the performance of the individual kernels that implement the constraint aggregation and the variable elimination processes described, respectively, in Sections 6.3 and 6.4. Figure 10 illustrates the average speedup obtained by the GPU-based constraint aggregation, and the GPU-based variable elimination with respect to their CPU-based counterparts when considering the largest bucket processed in each instance of the random, scale-free, and grid network instances. The reported average speedup for the constraint aggregation operations range from 363x (in scale
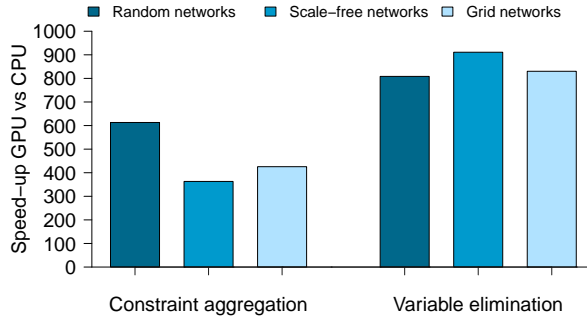
Fig. 10: Analysis of the average speedup obtained by the GPU-based constraint aggregation, and GPU-based variable elimination w.r.t. their CPU-based counterparts in the random, scale-free, and grid network instances.

free networks) to 613x (in random networks). The variable elimination operations achieve an even higher speedup, ranging from 830x (for grid networks) to 911x (for scale free networks). This is due to the high locality of data exploited by the GPU-based variable elimination kernel, which encourages coalesced data accesses, and through memory reuse, where we overwrite the input bucket-table of the variable elimination process with the resulting bucket-table from the same process.

Next, we compare our centralized and distributed versions of GpuBE with MBE [15] and ADPOP [53] at varying of the mini-bucket size $z \in \{2, \ldots, 10\}$, on binary constraint networks with *random*, *scale-free*, and *grid* topologies, using the same settings described in the previous section. The instances for each topology are generated as described above. Fig. 11(a–c) illustrate the speedup of the CPU and GPU versions of MBE, respectively, on random networks with $n = 20$, $d = 25$, $p_1 = 0.3$, on scale-free networks with $n = 20$, $d = 25$, and on grid networks with $\sqrt{n} = 10, d = 25$. The intensity of the color illustrates the solution quality of the bound returned (darker color denotes better solution quality). We make the following observations:

- The speedup obtained by the GPU vs. CPU solvers increases as the size of the mini-buckets increases. This observation is consistent with the previous observation that the speedup increases with increasing induced widths.
- The speedup saturates when $z = 7$ in all benchmarks, reporting maximal speedups of 235*x*, 274*x*, and 156*x*, for random, scale-free, and grid networks, respectively. This phenomena occurs when the maximum concurrent number of GPU threads are scheduled and executed simultaneously by all the GPU SMs—i.e., when there is enough work to saturate the GPU maximal occupancy.
- As for the previous experiment, the speedup trends of the distributed algorithms are similar to those of the centralized algorithms. The correlation[10] of the CPU

---

[10] We use the *Pearson product-moment correlation* coefficient.
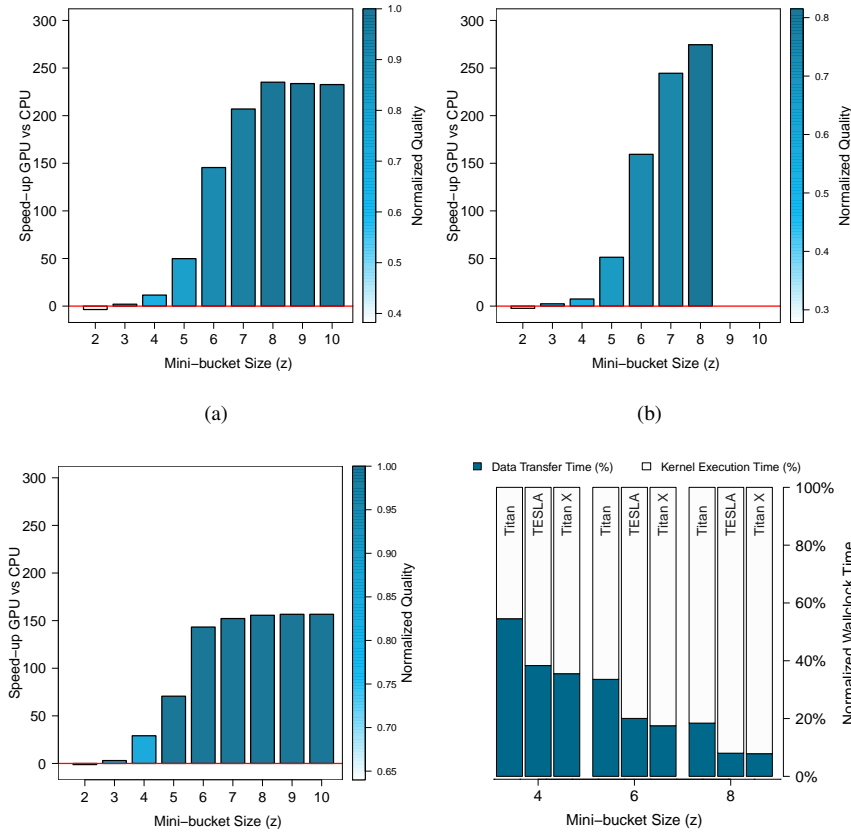
Fig. 11: (a) MBE Results on Random Networks: $n = 20, d = 25, p_1 = 0.3$; (b) MBE results on Scale-free Networks: $n = 20, d = 25$; (c) MBE results on Grid Networks: $\sqrt{n} = 10, d = 25$; (d) Normalized data allocation and transfer times (blue) vs. kernel times (white) on different GPUs.

vs. GPU speedup between the centralized and the distributed solutions are 0.93, 0.95, and 0.99, respectively for the grid, random, and scale-free network topologies.

Table 5 illustrates a comparison of the speedups obtained with three different GPU hardware configurations: *TESLA M2075*, *GeForce GTX Titan* and *GeForce GTX Titan X*, whose specifics are summarized in Table 4.[11]

Among the three GPU devices, the TESLA M2075 achieve the lowest maximal speedups, which range from 117.8*x* to 213,7*x*. Additionally, the speedup saturates when $z = 5$ for grid networks and $z = 7$ for random and scale-free networks. This is

---

[11] In all other experiments we used the GeForce GTX Titan, as this is the best, most affordable card at our disposal.

	TESLA M2075	GeForce GTX Titan	GeForce GTX Titan X
CUDA Capability	2.0	3.5	5.2
Global Memory Size	5375 MB	6137 MB	12286 MB
Number of SMs	14	14	24
Cores per SM	32	192	128
GPU Max Clock Rate	1.15 GHz	0.88 GHz	1.08 GHz
Memory Clock Rate	1566 Mhz	3004 Mhz	3505 Mhz
L2 Cache Size	786 KB	1572 KB	3145 KB
Max Number of Threads per SM	1536	2048	2048
Concurrent copy and execution	yes	yes	yes

Table 4: GPU device specifics.

	Grid			Random			Scale-Free		
z	TESLA	Titan	Titan X	TESLA	Titan	Titan X	TESLA	Titan	Titan X
2	0.22	0.85	0.49	0.21	0.28	0.60	0.27	0.43	0.57
3	1.54	3.05	2.80	1.42	1.99	3.25	1.48	2.42	1.61
4	21.9	29.1	33.0	10.7	11.6	16.1	12.2	7.46	12.0
5	117	150	232	60.6	49.8	66.4	59.3	51.3	66.8
6	117	143	237	144	145	223	163	159	285
7	117	152	235	198	207	392	208	244	435
8	118	153	241	198	235	645	211	274	627
9	115	156	238	197	234	620	oom	oom	oom
10	117	155	235	199	233	628	oom	oom	oom

Table 5: CPU vs. GPU speedup on different GPU devices.

due to the fact that this card can schedule the smallest number of cores per each SMs (32). Since each core can run concurrently a wrap (32 threads), its maximal level of concurrency is $14 \times 32 \times 32 = 14,336$ threads (and is thus the maximum number of parallel aggregation operations). In contrast, the speedup obtained by our GpuBE is the highest on the GeForce GTX Titan X—obtaining a maximal speedup of $646.9x$—and saturates when $z = 8$ in all networks. The maximum number of threads that can run concurrently on this card is $24 \times 128 \times 32 = 98,304$. The speedups obtained by our solver on the GeForce GTX Titan, used in the rest for the experiments in this paper, are larger than those obtained on the TESLA but smaller than those obtained on the GeForce GTX Titan X. This card can run up to $86,016$ threads. In addition to the number of threads than can run concurrently, the GPU clock rate and L2 cache size play a substantial role in the GPU performance.

Finally, Fig. 11(d) illustrates the time spent by the GPU devices while executing the kernel functions (in white) in contrast to the time used for memory transfers and allocations (in blue), at varying mini-bucket size $z = \{4, 6, 8\}$. These times are averaged among all instances for the three network topologies examined and are normalized with the respect to the wallclock runtime. The results show that the time spent by the device in performing actual computations increases, with the respect to the memory transfer time, as the mini-bucket size increase. Allocations and memory transfers on the Titan device are slower than on the TESLA and the GTX Titan X. Finally, these times account for the 36% to 55%, 18% to 34%, and 8% to 18% of the total time, respectively for the mini-bucket sizes $4, 6$, and 8.

8.2 WCSPs Benchmarks

We now report the evaluation of our GpuBE on the following standard WCSPs benchmarks:

- *Celar*: Radio link frequency assignment problems.
- *Coloring*: Graph coloring instances cast into minimum coloring instances.
- *Iscas89*: WCSPs derived from digital circuits.
- *Pedigree*: Instances from the genetic linkage analysis domain that is associated with the task of haplotyping.
- *Spot*: Instances of the daily photograph scheduling problem of Earth observation satellites.

Tables 6–9, tabulate the results for the above benchmarks. In each table and for each instance, we report, in order, the instance name—as appearing in the original benchmark—the number of variables n of the problem, the maximum size of their domains d, the number of constraints c, the graph density p_1, and the induced width w^* of the underlying primal graph. In each table, the top row shows the runtimes in seconds of GpuBE(z) at varying bucket size z and GpuBE. The bottom row shows the returned solutions' qualities, where for GpuBE(z), we report the lower bound it returned. When GpuBE failed to report a solution (due to memory limits), we report the solution quality found by toulbar2 (shown in parenthesis) or a dash symbol, if toulbar2 did not terminate within the time limit. The speedup of GpuBE(z) and GpuBE w.r.t. their CPU counterparts are shown in parentheses. For each instance, we vary the bucket size z from 2 to 20, and report the minimum bucket size $z_{\min}$, which is the largest constraint arity of the instance, the maximum bucket size $z_{\max} = \min\{w^z, 20\}$, where w^z is defined as the maximal bucket size that can be processed within the hardware memory limits, and the intermediate bucket sizes $z_2 = z_{\min} + \frac{1}{3}(z_{\max} - z_{\min})$ and $z_3 = z_{\min} + \frac{2}{3}(z_{\max} - z_{\min})$.

Consistent with our previous observations, the algorithms' speedups and solution qualities increase as the bucket size increases. Additionally, for several large problems instances (e.g., *scen06-24reduc—scen06reduc* in the Celar benchmark), our GPU implementation of MBE can report good lower bounds quickly (within a few seconds), whereas solving the entire problem with the most competitive soft consistency technique in toulbar2 requires from 6 to 48 minutes. For other large instances (e.g., in the Spot benchmark), we observe that toulbar2 ran out of time for the majority of the instances, while our GpuBE(z) can quickly find lower bounds, which could be used in a AND-and-OR search type as proposed by Marinescu and Dechter [46].

9 Related Work

The use of GPUs to solve difficult combinatorial problems has been explored by several proposals in different areas of constraint optimization. For instance, Meyer *et al.* [39] proposed a multi-GPU implementation of the *simplex tableau* algorithm that relies on a vertical problem decomposition to reduce communication between GPUs. In constraint programming, Arbelaez and Codognet [5] proposed a GPU-based version of the *Adaptive Search* algorithm, which explores several *large neighborhoods*

| Problem | n | d | c | p_1 | w^* | GpuBE(z) | | | | GpuBE |
						$z_{\min}$	z_2	z_3	$z_{\max}$	w^*
CELAR6-SUB0	16	44	207	0.47	7	0.116 (1.26x) 0	0.116 (13.5x) 13	0.31 (182x) 13	0.31 (182x) 13	oom (159)
CELAR7-SUB1-20	14	20	300	0.98	9	0.111 (3.37x) 20102	0.129 (42.8x) 40931	0.451 (215x) 71023	5.1 (313x) 81433	oom (132538)
CELAR6-SUB1-24	14	24	300	0.82	9	0.188 (0.59x) 0	0.122 (4.81x) 280	0.161 (71.8x) 598	0.837 (279x) 772	oom (2656)
CELAR7-SUB0	16	44	188	0.66	9	0.173 (0.63x) 0	0.157 (10.1x) 104	0.317 (187x) 10001	0.317 (187x) 10001	oom (10310)
CELAR6-SUB1	14	44	300	0.82	9	0.201 (1.25x) 0	0.317 (10.6x) 308	0.723 (171x) 626	0.723 (171x) 626	oom (2669)
CELAR7-SUB1	14	44	300	0.82	9	0.295 (1.03x) 0	0.316 (10.5x) 21523	0.593 (210x) 51123	0.593 (210x) 51123	oom (142640)
CELAR6-SUB2	16	44	353	0.77	10	0.254 (1.3x) 0	0.328 (12.6x) 231	0.82 (188x) 387	0.82 (188x) 387	oom (2746)
CELAR7-SUB2	16	44	353	0.77	10	0.407 (0.54x) 0	0.254 (16.4x) 20420	0.95 (164x) 40931	0.95 (164x) 40931	oom (173252)
CELAR6-SUB3	18	44	421	0.71	10	0.375 (1.08x) 0	0.304 (16.1x) 265	0.975 (188x) 452	0.975 (188x) 452	oom (3079)
CELAR7-SUB3	18	44	421	0.71	10	0.449 (0.97x) 0	0.46 (10.8x) 20615	1.002 (178x) 51434	1.002 (178x) 51434	oom (203460)
scen06-30reduc	81	14	399	0.11	10	0.071 (2.65x) 285	0.068 (10.3x) 690	0.325 (155x) 975	1.738 (278x) 1447	oom (2080)
scen06-30	99	14	1178	0.09	10	0.19 (1.5x) 450	0.189 (14.2x) 411	1.313 (238x) 1201	13.3 (345x) 1100	oom (2080)
CELAR6-SUB4-20	22	20	477	0.82	11	0.197 (4.09x) 494	0.232 (49.6x) 598	1.021 (225x) 732	11.29 (344x) 1359	oom (2716)
CELAR7-SUB4-22	22	22	473	0.67	11	0.221 (0.69x) 0	0.211 (3.54x) 40104	0.158 (82.4x) 60214	0.922 (286x) 31530	oom (202342)
CELAR6-SUB4reduc	20	44	149	0.77	11	0.106 (2.01x) 0	0.213 (10.9x) 44	0.357 (241x) 283	0.357 (241x) 283	oom (202342)
CELAR6-SUB4	22	44	477	0.65	1	0.387 (0.73x) 0	0.343 (17.4x) 170	1.013 (229x) 405	1.013 (229x) 405	oom (3230)
CELAR7-SUB4	22	44	477	0.65	1	0.347 (0.82x) 0	0.344 (18.1x) 30118	1.24 (188x) 31442	1.24 (188x) 31442	oom (242443)
scen06-24reduc	81	20	403	0.12	12	0.099 (4.76x) 278	0.101 (57.9x) 599	0.375 (217x) 634	4.001 (303x) 1411	oom (2857)
scen06-22reduc	81	22	404	0.12	12	0.164 (0.68x) 0	0.091 (5.87x) 453	0.122 (67.1x) 717	0.52 (243x) 793	oom (3159)
scen06-20reduc	82	24	409	0.12	12	0.203 (0.68x) 0	0.095 (7.53x) 447	0.142 (86.1x) 717	0.838 (277x) 794	oom (3163)
scen06-18reduc	82	26	409	0.12	12	0.221 (0.76x) 0	0.194 (4.7x) 458	0.303 (56x) 718	1.189 (292x) 796	oom (3263)
scen06-24	99	20	1203	0.10	12	0.25 (0.52x) 0	0.236 (3.97x) 437	0.278 (47.1x) 319	0.867 (233x) 900	oom
scen06-16reduc	82	28	409	0.12	12	0.22 (0.45x) 0	0.113 (10.4x) 458	0.235 (101x) 717	1.695 (304x) 812	oom
scen06-22	99	22	1210	0.10	12	0.271 (0.58x) 0	0.26 (4.93x) 437	0.358 (56.3x) 403	1.415 (256x) 803	oom
scen06-20	100	24	1215	0.10	12	0.306 (1.2x) 0	0.263 (6.37x) 437	0.371 (78.2x) 352	1.979 (291x) 804	oom
scen06-18	100	26	1221	0.10	12	0.352 (0.83x) 0	0.299 (7.54x) 437	0.457 (94.4x) 327	2.995 (303x) 813	oom
scen06-16	100	28	1222	0.1	12	0.36 (1.34x) 0	0.389 (7.12x) 437	0.537 (122x) 328	4.382 (317x) 813	oom
scen06reduc	82	44	409	0.12	14	0.343 (0.68x) 0	0.306 (15.1x) 137	0.787 (204x) 318	0.787 (204x) 318	oom

Table 6: Celar Benchmark: Runtime (in seconds) of GpuBE, at varying of the bucket size z and GpuBE(w^*) (top), and solution quality (bottom). The speedup of GpuBE(z) and GpuBE(w^*) w.r.t. their CPU counterparts are shown in parenthesis.

in parallel, resulting in a speedup factor of 17. Campeotto *et al.* [12] proposed a GPU-based framework that exploits both parallel propagation and parallel exploration of several large neighborhoods using local search techniques, leading to a speedup factor of up to 38. The combination of GPUs with dynamic programming has also been explored to solve different combinatorial optimization problems. For instance, Boyer *et al.* [9] proposed the use of GPUs to compute the classical DP recursion step for the knapsack problem, which led to a speedup factor of 26. Pawłowski *et al.* [49] pre-

Problem	n	d	c	p_1	w^*	GpuBE(z)				GpuBE
						$z_{\min}$	z_2	z_3	$z_{\max}$	w^*
GEOM40-2	40	2	78	0.12	5	0.004 (0.25x) 22	0.004 (0.25x) 22	0.004 (0.25x) 22	0.004 (0.25x) 22	0.004 (0.25x) 22
GEOM40-3	40	3	78	0.12	5	0.009 (0.44x) 7	0.009 (0.44x) 7	0.009 (0.44x) 7	0.009 (0.44x) 7	0.009 (0.44x) 7
GEOM40-4	40	4	78	0.12	5	0.004 (4.25x) 3	0.004 (4.25x) 3	0.004 (4.25x) 3	0.004 (4.25x) 3	0.004 (4.25x) 3
GEOM40-5	40	5	78	0.12	5	0.005 (10.8x) 1	0.005 (10.8x) 1	0.005 (10.8x) 1	0.005 (10.8x) 1	0.005 (10.8x) 1
GEOM40-6	40	6	78	0.12	5	0.011 (5.73x) 0	0.011 (5.73x) 0	0.011 (5.73x) 0	0.011 (5.73x) 0	0.011 (5.73x) 0
GEOM30a-3	30	3	81	0.2	6	0.014 (0.21x) 0	0.024 (0.5x) 10	0.011 (1.73x) 11	0.024 (0.79x) 11	0.003 (2.67x) 11
GEOM30a-4	30	4	81	0.2	6	0.028 (0.14x) 0	0.024 (2.25x) 4	0.012 (10.3x) 4	0.012 (10.2x) 4	0.003 (14.7x) 4
GEOM30a-5	30	5	81	0.2	6	0.029 (0.21x) 0	0.012 (16x) 1	0.012 (21.3x) 1	0.012 (20.2x) 1	0.004 (21.8x) 1
GEOM30a-6	30	6	81	0.2	6	0.03 (0.17x) 0	0.013 (35.3x) 0	0.028 (31.6x) 0	0.015 (68.8x) 0	0.006 (50.3x) 0
queen5-5-3	25	3	160	0.87	18	0.031 (0.19x) 6	0.034 (48.2x) 18	0.471 (203x) 23	2.899 (250x) 25	oom (29)
queen5-5-4	25	4	160	0.87	18	0.031 (1x) 0	0.038 (53.3x) 1	0.158 (169x) 4	1.355 (238x) 5	oom (12)
queen5-5-5	25	5	160	0.87	18	0.031 (2.19x) 0	0.121 (97x) 0	1.031 (247x) 0	4.659 (267x) 0	oom (0)
myciel5g-3	47	3	236	0.44	20	0.033 (2.58x) 0	0.069 (27.2x) 3	1.999 (201x) 9	11.97 (308x) 12	oom (16)
myciel5g-4	47	4	236	0.44	20	0.031 (1.03x) 0	0.045 (52.6x) 0	0.23 (143x) 0	7.852 (293x) 0	oom (4)
myciel5g-5	47	5	236	0.44	20	0.072 (0.92x) 0	0.051 (62.7x) 0	0.41 (160x) 0	6.513 (278x) 0	oom (1)
myciel5g-6	47	6	236	0.44	20	0.035 (3.57x) 0	0.069 (5.39x) 0	0.123 (91x) 0	1.636 (220x) 0	oom (0)
DSJC125.1.4	125	4	736	0.72	72	0.241 (0.62x) 0	0.168 (19.4x) 0	0.477 (96.5x) 0	3.551 (197x) 0	oom –
DSJC125.1.5	125	5	736	0.72	72	0.285 (1.1x) 0	0.17 (4.43x) 0	0.393 (38.8x) 0	1.801 (191x) 0	oom (0)
le450-5a-2	450	2	5714	0.81	344	1.725 (0.18x) 618	1.611 (1.46x) 734	2.934 (40.7x) 833	10.21 (67.3x) 878	oom –
le450-5a-3	450	3	5714	0.81	344	1.728 (0.19x) 42	1.847 (2.39x) 55	2.042 (17.6x) 57	7.207 (44.4x) 58	oom –
le450-5a-4	450	4	5714	0.81	344	1.422 (0.44x) 0	1.549 (1.29x) 0	2.088 (13.7x) 4	6.517 (68.6x) 1	oom –
le450-5a-5	450	5	5714	0.81	344	1.67 (0.98x) 0	1.844 (3.7x) 0	3.505 (39.3x) 0	10.23 (65.1x) 0	oom –

Table 7: Coloring Benchmark: Runtime (in seconds) of GpuBE, at varying of the bucket size z and GpuBE(w^*) (top), and solution quality (bottom). The speedup of GpuBE(z) and GpuBE(w^*) w.r.t. their CPU counterparts are shown in parenthesis.

sented a DP-based solution for the *coalition structure formation problem* on GPUs, reporting up to two orders of magnitude of speedup. In a recent work, Bistaffa *et al.* [7] study the parallelization of an inference-based algorithm to solve COPs using GPUs, albeit exclusively in the centralized case. Silberstein *et al.* [62] study a GPU-based kernel for the sum-product operations that arise in *marginalize a product of functions* (MPF) problems. The authors report an average speedup factor of 15 for random benchmarks and Bayesian networks and higher average speedups (up to two orders of magnitude) for log domains due to the difference in performance of the *log2f* and *exp2f* functions on the CPU and GPU.

In the distributed constraint optimization context, GPU parallelism has been applied to speed up several DCOP solving techniques. Fioretto *et al.* [28] proposed a multi-variable agent decomposition strategy to solve *general* DCOPs with complex local subproblems, which makes use of GPUs to implement a search-based and a

Problem	n	d	c	p_1	w^*	GpuBE(z)				GpuBE w^*
						z_{min}	z_2	z_3	z_{max}	
s386	172	2	172	0.04	19	0.054 (0.28x) 29	0.051 (2.55x) 29	0.053 (16.1x) 29	0.185 (71.9x) 29	0.129 (82.8x) 29
s1423	748	2	748	0.06	38	0.184 (0.12x) 231	0.182 (1.08x) 231	0.189 (6.05x) 231	0.546 (57.8x) 231	oom (231)
c499	499	2	499	0.01	42	0.133 (0.89x) 111	0.131 (1.08x) 111	0.141 (9.72x) 111	0.391 (78.1x) 111	oom (111)
c432	432	2	432	0.01	54	0.225 (0.33x) 101	0.237 (0.72x) 101	0.270 (7.93x) 101	0.622 (69.6x) 101	oom –
s1494	661	2	661	0.01	57	0.238 (0.19x) 32	0.222 (1.14x) 32	0.249 (17.1x) 32	1.285 (101x) 32	oom (32)
s1488	667	2	667	0.01	62	0.232 (0.13x) 32	0.219 (1.07x) 32	0.244 (16.7x) 32	1.268 (98.5x) 32	oom (32)
c880	880	2	880	0.04	68	0.245 (0.11x) 162	0.241 (0.69x) 162	0.295 (8.86x) 162	1.1 (91.7x) 162	oom –
s1196	561	2	561	0.01	92	0.19 (0.27x) 95	0.185 (0.93x) 95	0.305 (10.6x) 95	1.049 (101x) 95	oom (95)
s953	440	2	440	0.01	93	0.234 (0.13x) 124	0.139 (1.62x) 124	0.262 (11.2x) 124	0.781 (98.8x) 124	oom (124)
s1238	540	2	540	0.01	95	0.195 (0.28x) 95	0.184 (1.04x) 95	0.21 (16.8x) 95	0.964 (94.6x) 95	oom (95)

Table 8: Iscas-89 Benchmark: Runtime (in seconds) of GpuBE, at varying of the bucket size z and GpuBE(w^*) (top), and solution quality (bottom). The speedup of GpuBE(z) and GpuBE(w^*) w.r.t. their CPU counterparts are shown in parenthesis.

Problem	n	d	c	p_1	w^*	GpuBE(z)				GpuBE w^*
						z_{min}	z_2	z_3	z_{max}	
eye	36	21	53	0.09	2	0.048 (3.83x) 1	0.045 (3.67x) 1	0.045 (2.2x) 1	0.045 (4.13x) 1	0.006 (12.7x) 1
wijsmanguo	49	36	68	0.06	3	0.278 (2.15x) 1	0.279 (2.18x) 1	0.278 (2.21x) 1	0.278 (2.15x) 1	0.012 (18.8x) 1
cancer	49	36	68	0.06	3	0.304 (2.34x) 1	0.28 (2.91x) 1	0.291 (2.44x) 1	0.278 (2.55x) 1	0.012 (18.8x) 1
sobel	7	6	8	0.61	3	0.002 (0.5x) 0	0.002 (1x) 0	0.001 (2x) 0	0.001 (2x) 0	0.001 (1x) 0
connell	12	6	15	0.39	3	0.004 (0.25x) 0	0.004 (1.5x) 1	0.002 (2.5x) 1	0.001 (6x) 1	0.001 (4x) 1
pedck60-L2	60	10	106	0.08	5	0.017 (1.71x) 2	0.052 (78x) 2	0.052 (77x) 2	0.051 (82.6x) 2	0.02 (144x) 2
pedck60-L1	60	10	108	0.08	5	0.016 (1.56x) 2	0.033 (125x) 2	0.044 (94.1x) 2	0.033 (124x) 2	0.02 (137x) 2
pedck60-L12	60	10	108	0.08	5	0.023 (1.26x) 6	0.033 (124x) 6	0.033 (127x) 6	0.033 (126x) 6	0.02 (143x) 6
saudiarabia	37	15	43	0.16	5	0.015 (4.13x) 0	0.267 (231x) 0	0.286 (211x) 0	0.282 (210x) 0	0.187 (212x) 0
parkinson	37	15	43	0.16	5	0.015 (3.4x) 0	0.292 (206x) 0	0.265 (223x) 0	0.274 (223x) 0	0.186 (237x) 0
pedck350l3	350	10	578	0.03	24	0.092 (2.13x) 0	0.094 (7.23x) 0	0.109 (35.7x) 0	0.192 (112x) 0	oom (0)
pedck350l2	350	10	578	0.03	24	0.091 (1.02x) 0	0.131 (4.21x) 0	0.109 (32.9x) 1	0.235 (81.1x) 1	oom (1)
pedck350	350	10	580	0.03	26	0.099 (1.04x) 0	0.139 (3.89x) 0	0.117 (27.9x) 2	0.252 (114x) 2	oom (2)
sheep4r-4-3	2662	10	5021	0.00	38	1.16 (0.98x) 0	1.173 (5.83x) 0	1.551 (29.9x) 1	3.139 (110x) 1	oom (1)
sheep4r-4-2	2172	10	4026	0.00	46	0.874 (0.96x) 1	1.3 (4.11x) 0	1.145 (29.9x) 0	2.339 (110x) 0	oom (1)
sheep4r-4-1	641	10	1196	0.05	63	0.273 (0.96x) 0	0.291 (5.92x) 0	0.36 (35.7x) 0	0.782 (120x) 0	oom (0)
sheep4r-4-0	1541	10	2941	0.03	108	0.76 (0.95x) 0	0.786 (5.93x) 0	1.066 (31.3x) 0	1.066 (31.3x) 0	oom (0)
pedck1000	928	6	1736	0.09	126	0.368 (0.34x) 15	0.393 (1.32x) 15	0.399 (4.46x) 12	0.447 (20.1x) 16	oom (19)

Table 9: Pedigree Benchmark: Runtime (in seconds) of GpuBE, at varying of the bucket size z and GpuBE(w^*) (top), and solution quality (bottom). The speedup of GpuBE(z) and GpuBE(w^*) w.r.t. their CPU counterparts are shown in parenthesis.

Problem	n	d	c	p_1	w^*	GpuBE(z) z_{min}	z_2	z_3	z_{max}	GpuBE w^*
8	8	4	15	0.28	2	0.003 (0.33x) 2	0.001 (1.0x) 2	0.002 (1.5x) 2	0.003 (1.33x) 2	0.003 (2.0x) 2
1502	209	4	411	0.01	5	0.082 (0.32x) 28042	0.039 (1.18x) 28042	0.08 (0.57x) 28042	0.079 (0.61x) 28042	0.029 (0.66x) 28042
54	67	4	271	0.15	11	0.037 (0.92x) 31	0.036 (1.86x) 32	0.036 (7.31x) 35	0.045 (53.7x) 37	0.027 (46.1x) 37
503	143	4	635	0.07	12	0.088 (0.74x) 7093	0.085 (2.88x) 9106	0.088 (18.3x) 11111	0.203 (31.9x) 11113	0.044 (59.9x) 11113
29	82	4	462	0.17	14	0.085 (0.11x) 7035	0.07 (3.71x) 8048	0.147 (97.7x) 8055	7.281 (236x) 8059	3.327 (141x) 8059
404	100	4	710	0.16	19	0.112 (0.6x) 76	0.207 (5.95x) 106	0.188 (113x) 112	0.701 (224x) 114	0.301 (124x) 114
42b	190	4	1140	0.09	19	0.455 (0.13x) 58049	0.358 (0.65x) 95050	0.276 (43.6x) 135050	12.98 (239x) 155050	oom –
505b	240	4	1716	0.06	23	0.392 (0.12x) 6106	0.484 (0.61x) 11159	0.361 (48.6x) 15204	7.531 (238x) 19244	oom –
1504	605	4	4187	0.03	25	1.047 (0.18x) 80104	0.884 (2.41x) 107175	1.221 (55.3x) 131227	12.09 (263x) 141251	oom –
408b	200	4	1843	0.10	29	0.552 (0.14x) 2104	0.309 (2.77x) 5162	0.862 (132x) 6206	5.774 (218x) 6215	oom –
42	190	4	1394	0.11	30	0.295 (0.11x) 60049	0.251 (7.53x) 97050	0.59 (56.9x) 105050	9.419 (241x) 133050	oom –
505	240	4	2242	0.11	30	0.61 (0.09x) 5103	0.547 (1.05x) 7149	0.712 (83.3x) 10178	8.757 (220x) 15211	oom –
408	200	4	2232	0.17	40	0.637 (0.12x) 2100	0.506 (2.14x) 3123	0.816 (96x) 4169	10.2 (265x) 5185	oom –
5	309	4	5621	0.19	44	1.45 (0.14x) 42	1.18 (1.09x) 53	1.879 (37x) 86	6.573 (158x) 106	oom –
412	300	4	4348	0.16	48	1.366 (0.08x) 2106	1.097 (2.05x) 2131	1.228 (41.3x) 8176	7.048 (183x) 11258	oom –
507	311	4	5732	0.18	68	1.788 (0.13x) 6114	1.379 (1.59x) 5140	1.744 (38.2x) 7194	5.343 (140x) 10226	oom –
1506	940	4	15240	0.05	77	4.82 (0.13x) 50122	3.805 (0.82x) 68152	3.929 (8.52x) 86172	16.93 (128x) 88235	oom –
28	230	4	5226	0.42	87	1.397 (0.12x) 43075	1.173 (1.89x) 52105	1.946 (50x) 76105	8.584 (226x) 88105	oom –
509	348	4	8624	0.22	92	2.82 (0.16x) 3105	2.204 (5.56x) 3217	5.664 (75.6x) 5191	10.61 (278x) 6157	oom –
414	364	4	10108	0.24	104	3.61 (0.16x) 3113	2.673 (1.95x) 3120	3.091 (16.7x) 4195	7.608 (80.7x) 6141	oom –
1401	488	4	10963	0.17	105	3.283 (0.16x) 64057	3.047 (1.62x) 61066	3.543 (13x) 66071	13.65 (157x) 87071	oom –
1403	665	4	13616	0.11	105	4.52 (0.14x) 58099	3.829 (1.63x) 71129	4.246 (15x) 74118	10.11 (69.6x) 82120	oom –
1405	855	4	18258	0.09	105	5.708 (0.14x) 58099	4.791 (2.08x) 71129	6.787 (17.3x) 74118	19.98 (74.5x) 84177	oom –
1407	1057	4	21786	0.07	105	7.018 (0.15x) 58127	6.283 (2.21x) 74180	8.931 (19x) 75164	18.15 (53.9x) 84202	oom –

Table 10: Spot Benchmark: Runtime (in seconds) of GpuBE, at varying of the bucket size z and GpuBE(w^*) (top), and solution quality (bottom). The speedup of GpuBE(z) and GpuBE(w^*) w.r.t. their CPU counterparts are shown in parenthesis.

sampling-based algorithm to speed up the agents' local subproblems resolution. Le *et al.* [42] studied a GPU accelerated algorithm in the context of stochastic DCOPs— DCOPs where the values of the cost tables are stochastic. The authors used SIMT-style parallelism on a DP-based approach, which resulted in a speedup of up to two orders of magnitude. Recently, a combination of GPUs with *Markov Chain Monte Carlo* (MCMC) sampling algorithms has been proposed in the context of solving DCOPs [27], where the authors adopted GPUs to accelerate the computation of the normalization constants used in the MCMC sampling process as well as to compute several samples in parallel, resulting in a speedup of up to one order of magnitude.

Finally, a parallelization of the AND/OR Branch-and-Bound search with mini-bucket heuristic has been presented in [41].

Differently from other proposals, our approach aims at using GPUs to exploit SIMT-style parallelism from DP-based methods to solve general, exact and approximated, WCSPs and DCOPs.

10 Conclusions and Discussions

Inference-based algorithms are powerful tools for solving discrete optimization problems. However, their applicability is limited by their high time and space requirements. Motivated by the increasing availability of GPUs, in this paper, we proposed a scheme to speed up the resolution of inference-based methods for centralized and distributed constraint optimization by exploiting SIMT-style parallelism. We introduced an exact algorithm and an approximated algorithm that are inspired by BE and MBE for WCSPs and tasks over belief networks (e.g., MPE), and by DPOP and AD-POP for DCOPs. These procedures make use of multiple threads in the GPU cards to parallelize the aggregation and elimination procedures, which are responsible for the high complexity in the inference-based approaches. Additionally, we detailed the design of the data structures adopted to process cost functions with GPUs, and of the mapping adopted to associate GPU threads to cost functions' entries, which allows us to efficiently exploit the data parallelism (SIMT) supported by GPUs.

Finally, we reported an extensive experimental evaluation of our inference-based GPU implementations on both centralized and distributed benchmarks. We showed that the use of GPUs provides significant advantages in terms of runtime and scalability, achieving speedups of up to two order of magnitude, showing a considerable reduction in runtime (up to 345 times faster) with respect to the serialized version, and that the speedups increase with the induced width of the problem and with the size of the domain of the problem's variables.

The proposed results are significant—the wide availability of GPUs provides access to parallel computing solutions that can be used to improve efficiency of WCSPs and DCOP solvers. Furthermore, GPUs are renowned for their complex architectures (multiple memory levels with very different size and speed characteristics; relatively slow cores), which often create challenges to the effective exploitation of parallelism from irregular applications. The strong experimental results indicate that the proposed algorithms are well-suited to GPU architectures.

These results hint that our approach could be exploited in the the context of a dynamic search that makes use of the mini-bucket elimination method as an heuristic to infer bounds on the solution quality, potentially allowing dynamic variable orderings. Indeed, the main drawback of this type of search (and various look-ahead methods) is that, since the heuristic is (re)computed in various nodes during search process, the time invested in the heuristic computation may not be cost-effective. Leveraging the use of GPUs to infer bounds faster during the dynamic search, may therefore produce dramatic speedup to the whole search process.

While this paper describes the applicability of our approach to (M)BE and (A)DPOP, analogous techniques can be derived and applied to other inference-based

approaches to solve discrete optimization problems (e.g., to implement the logic of inference-based propagators) and optimization on probabilistic graphical models— (e.g., in finding maximum probability explanation (MPE) in belief networks). Additionally, our work can be extended to solve *sum-product* problems, also known as *weighted counting*, *partition function*, or *probability of evidence*. Due to the difficulty of these problems the value of accelerated versions of the bucket elimination algorithms is especially important. We plan to work in this direction as future work.

We also envision this technology could open the door to efficiently enforcing higher forms of consistencies than domain consistency (e.g., *path consistency* [48], *adaptive consistency* [18], or the more recently proposed *branch consistency* for DCOPs [25]), especially when the cost functions need to be represented explicitly.

References

1. Abdennadher, S., Schlenker, H.: Nurse Scheduling using Constraint Logic Programming. In: Proceedings of the Conference on Innovative Applications of Artificial Intelligence (IAAI), pp. 838–843 (1999)
2. Allouche, D., André, I., Barbe, S., Davies, J., de Givry, S., Katsirelos, G., O'Sullivan, B., Prestwich, S.D., Schiex, T., Traoré, S.: Computational protein design as an optimization problem. Artificial Intelligence **212**, 59–79 (2014)
3. Allouche, D., de Givry, S., Nguyen, H., Schiex, T.: ToulBar2 to solve Weighted Partial Max-SAT. Tech. rep., INRA (2013)
4. Apt, K.: Principles of constraint programming. Cambridge University Press (2003)
5. Arbelaez, A., Codognet, P.: A GPU implementation of parallel constraint-based local search. In: Proceedings of the Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP), pp. 648–655 (2014)
6. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
7. Bistaffa, F., Bomberi, N., Farinelli, A.: CUBE: A CUDA approach for bucket elimination on GPUs. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), p. to appear (2016)
8. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM **44**(2), 201–236 (1997)
9. Boyer, V., El Baz, D., Elkihel, M.: Solving knapsack problems on GPU. Computers & Operations Research **39**(1), 42–47 (2012)
10. Brito, I., Meseguer, P.: Improving DPOP with function filtering. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 141–158 (2010)
11. Burke, E.K., De Causmaecker, P., Berghe, G.V., Van Landeghem, H.: The state of the art of nurse rostering. Journal of scheduling **7**(6), 441–499 (2004)
12. Campeotto, F., Dovier, A., Fioretto, F., Pontelli, E.: A GPU implementation of large neighborhood search for solving constraint optimization problems. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), pp. 189–194 (2014)
13. Campeotto, F., Palù, A.D., Dovier, A., Fioretto, F., Pontelli, E.: A constraint solver for flexible protein model. Journal of Artificial Intelligence Research **48**, 953–1000 (2013)
14. Chakroun, I., Mezmaz, M.S., Melab, N., Bendjoudi, A.: Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience **25**(8), 1121–1136 (2013)

15. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1), 41–85 (1999)
16. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
17. Dechter, R.: Reasoning with probabilistic and deterministic graphical models: Exact algorithms. Synthesis Lectures on Artificial Intelligence and Machine Learning **7**(3), 1–191 (2013)
18. Dechter, R., Pearl, J.: Network-based heuristics for constraint-satisfaction problems. Springer (1988)
19. Dechter, R., Rish, I.: Mini-buckets: A general scheme for bounded inference. Journal of the ACM **50**(2), 107–153 (2003)
20. Diamos, G.F., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., Yalamanchili, S.: SIMD re-convergence at thread frontiers. In: Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture, pp. 477–488 (2011)
21. Dovier, A., Formisano, A., Pontelli, E.: Autonomous agents coordination: Action languages meet CLP() and Linda. Theory and Practice of Logic Programming **13**(2), 149–173 (2013)
22. Edelkamp, S., Jabbar, S., Schrödl, S.: External A*. In: Advances in Artificial Intelligence: 27th Annual German Conference on AI, (KI) 2004, pp. 226–240 (2004)
23. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.: Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 639–646 (2008)
24. Fioretto, F., Dovier, A., Pontelli, E.: Constrained community-based gene regulatory network inference. ACM Trans. Model. Comput. Simul. **25**(2), 11 (2015)
25. Fioretto, F., Le, T., Yeoh, W., Pontelli, E., Son, T.C.: Improving DPOP with branch consistency for solving distributed constraint optimization problems. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 307–323 (2014)
26. Fioretto, F., Le, T., Yeoh, W., Pontelli, E., Son, T.C.: Exploiting GPUs in solving (distributed) constraint optimization problems with dynamic programming. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 121–139 (2015)
27. Fioretto, F., Yeoh, W., Pontelli, E.: A dynamic programming-based MCMC framework for solving DCOPs with GPUs. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 813–831 (2016)
28. Fioretto, F., Yeoh, W., Pontelli, E.: Multi-Variable Agent Decomposition for DCOPs. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 2480–2486 (2016)
29. Fioretto, F., Yeoh, W., Pontelli, E.: A multiagent system approach to scheduling devices in smart homes. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 981–989 (2017)
30. Fioretto, F., Yeoh, W., Pontelli, E., Ma, Y., Ranade, S.: A DCOP approach to the economic dispatch with demand response. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 981–989 (2017)
31. Fishelson, M., Geiger, D.: Exact genetic linkage computations for general pedigrees. Bioinformatics **18**(suppl 1), S189–S198 (2002)
32. Friedman, N., Linial, M., Nachman, I., Pe'er, D.: Using bayesian networks to analyze expression data. Journal of computational biology **7**(3-4), 601–620 (2000)
33. Gaudreault, J., Frayret, J.M., Pesant, G.: Distributed search for supply chain coordination. Computers in Industry **60**(6), 441–451 (2009)
34. Gupta, S., Yeoh, W., Pontelli, E., Jain, P., Ranade, S.J.: Modeling microgrid islanding problems as DCOPs. In: North American Power Symposium (NAPS), pp. 1–6. IEEE (2013)
35. Hamadi, Y., Bessière, C., Quinqueton, J.: Distributed intelligent backtracking. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), pp. 219–223 (1998)
36. Han, T.D., Abdelrahman, T.S.: Reducing Branch Divergence in GPU Programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, pp. 3:1–3:8. ACM Press, New York, NY (2011)
37. Kask, K., Dechter, R., Gelfand, A.E.: Beem: bucket elimination with external memory. arXiv preprint arXiv:1203.3487 (2012)
38. Kumar, A., Faltings, B., Petcu, A.: Distributed constraint optimization with structured resource constraints. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 923–930 (2009)
39. Lalami, M.E., El Baz, D., Boyer, V.: Multi GPU implementation of the simplex algorithm. In: Proceedings of the International Conference on High Performance Computing and Communication (HPCC), vol. 11, pp. 179–186 (2011)

40. Larrosa, J.: Node and arc consistency in weighted csp. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 48–53 (2002)
41. Lars, O., Rina, D.: And/or branch-and-bound on a computational grid. Journal of Artificial Intelligence Research ((to appear)) (2017)
42. Le, T., Fioretto, F., Yeoh, W., Son, T.C., Pontelli, E.: ER-DCOPs: A Framework for Distributed Constraint Optimization with Uncertainty in Constraint Utilities. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 605–614 (2016)
43. Lerner, U., Parr, R., Koller, D., Biswas, G., et al.: Bayesian fault detection and diagnosis in dynamic systems. In: AAAI/IAAI, pp. 531–537 (2000)
44. Lim, H., Yuan, C., Hansen, E.A.: Scaling up map search in bayesian networks using external memory. on Probabilistic Graphical Models p. 177 (2010)
45. Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 310–317 (2004)
46. Marinescu, R., Dechter, R.: Memory intensive and/or search for combinatorial optimization in graphical models. Artificial Intelligence **173**(16-17), 1492–1524 (2009)
47. Modi, P., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161**(1–2), 149–180 (2005)
48. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information sciences **7**, 95–132 (1974)
49. Pawłowski, K., Kurach, K., Michalak, T., Rahwan, T.: Coalition structure generation with the graphic processor unit. Tech. Rep. CS-RR-13-07, Department of Computer Science, University of Oxford (2104)
50. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
51. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 482–495 (2004)
52. Petcu, A., Faltings, B.: Approximations in distributed optimization. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 802–806 (2005)
53. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1413–1420 (2005)
54. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 751–755. Springer (2006)
55. Rodrigues, L., Magatao, L.: Enhancing Supply Chain Decisions Using Constraint Programming: A Case Study. In: MICAI 2007: Advances in Artificial Intelligence, vol. LNCS 4827, pp. 1110–1121. Springer Verlag (2007)
56. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
57. Rust, P., Picard, G., Ramparany, F.: Using message-passing DCOP algorithms to solve energy-efficient smart environment configuration problems. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 468–474 (2016)
58. Sanders, J., Kandrot, E.: CUDA by Example. An Introduction to General-Purpose GPU Programming. Addison Wesley (2010)
59. Sandholm, T.: Algorithm for optimal winner determination in combinatorial auctions. Artificial intelligence **135**(1), 1–54 (2002)
60. Schiex, T., Fargier, H., Verfaillie, G., et al.: Valued constraint satisfaction problems: Hard and easy problems. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) **95**, 631–639 (1995)
61. Shapiro, L.G., Haralick, R.M.: Structural descriptions and inexact matching. IEEE Transactions on Pattern Analysis and Machine Intelligence **3**(5), 504–519 (1981)
62. Silberstein, M., Schuster, A., Geiger, D., Patney, A., Owens, J.D.: Efficient computation of sum-products on gpus through software-managed cache. In: Proceedings of the 22nd annual international conference on Supercomputing, pp. 309–318. ACM (2008)
63. Sturtevant, N.R., Rutherford, M.J.: Minimizing writes in parallel external memory search. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) (2013)
64. Sultanik, E., Modi, P.J., Regli, W.C.: On modeling multiagent task scheduling as a distributed constraint optimization problem. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1531–1536 (2007)

65. Trick, M.A.: A dynamic programming approach for consistency and propagation for knapsack constraints. Annals of Operations Research **118**(1-4), 73–84 (2003)
66. Yeoh, W., Felner, A., Koenig, S.: BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. Journal of Artificial Intelligence Research **38**, 85–133 (2010)
67. Yeoh, W., Yokoo, M.: Distributed problem solving. AI Magazine **33**(3), 53–65 (2012)
68. Zivan, R., Yedidsion, H., Okamoto, S., Glinton, R., Sycara, K.: Distributed constraint optimization for teams of mobile sensing agents. Journal of Autonomous Agents and Multi-Agent Systems **29**(3), 495–536 (2015)