This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

An Efficient Approach for Accelerating Bucket Elimination on GPUs

Filippo Bistaffa, Nicola Bombieri, and Alessandro Farinelli

***Abstract*—Bucket elimination (BE) is a framework that encompasses several algorithms, including belief propagation (BP) and variable elimination for constraint optimization problems (COPs). BE has significant computational requirements that can be addressed by using graphics processing units (GPUs) to parallelize its fundamental operations, i.e., composition and marginalization, which operate on functions represented by large tables. We propose a novel approach to parallelize these operations with GPUs, which optimizes the table layout so to achieve better performance in terms of increased speedup and scalability. Our approach allows us to process incomplete tables (i.e., tables with some missing variables assignments), which often occur in several practical applications (such as the ones we consider in our dataset). Finally, we can process tables that are larger than the GPU memory. Our approach outperforms the state-of-the-art technique to parallelize BP on GPUs, achieving better speedups (up to +466% with respect to such parallel technique). We test our method on a publicly available COP dataset, measuring a speedup up to 696.02× with respect to the sequential version. The ability of our technique to process large tables is crucial in this scenario, in which most of the instances generate tables larger than the GPU memory, and hence they cannot be solved with previous GPU techniques related to BE.**

***Index Terms*—Belief propagation (BP), bucket elimination (BE), constraint optimization problem (COP), graphics processing unit (GPU), junction tree (JT).**

I. Introduction

DYNAMIC programming (DP) [1] is a well-known method used to solve complex problems by exploiting their optimal substructure property, i.e., the possibility of efficiently decomposing the original problem into smaller subproblems and then construct the global optimal solution.

Several popular DP algorithms used in many different fields have been shown to fall under a general framework called bucket elimination (BE) [2]. Some examples are Fourier and Gaussian elimination for linear equalities and inequalities [3], adaptive consistency for constraint satisfaction problems (CSPs) [4], directional resolution for propositional satisfiability [5], belief propagation (BP) [6], DP for combinatorial optimization [7], variable elimination for constraint optimization problems (COPs) [2] and distributed COPs [8]. More precisely, BE operates by propagating messages over a series of buckets, i.e., sets of constraints (or functions) that depend on a given variable. Such messages, encoded by functions in tabular form, are constructed by means of two fundamental operators, $\oplus$ and $\Downarrow$, often referred as composition and marginalization. One of the main advantages of BE is its generality, as such an algorithm can be applied to solve problems in several different fields by changing the fundamental mathematical operations[1] used by these two operators.

While the number of exchanged messages is usually rather small (i.e., equal to the number of input variables minus one), the computation of the messages themselves represents the most computationally intensive task of the entire algorithm, whose complexity can be precisely predicted and it is exponential with respect to a single parameter called induced width of the graph representing the initial problem [2]. For this reason, having a very efficient algorithm to perform such a computation is of utmost importance when dealing with realistic problems, which often require BE to exchange very large messages (i.e., up to 60 GB in our experiments).

In recent years, graphics processing units (GPUs) have been successfully used to speedup the computation in different cybernetic applications that feature a high level of parallelism, achieving performance improvements of several orders of magnitude [10] in fields including computer vision [11], human–computer interaction [12], and artificial intelligence [13]. In this paper, we are interested in developing a high-performance GPU framework that allows us to deal with the computational effort inherent in the message passing phase of several BE-based algorithms, so to favor its integration in the development of cybernetic solutions, especially in fields like BP, decision making, and scheduling [14]. To this end, our main objective is to devise a solution that fulfils three key requirements. First, since BE is a general algorithm that can be applied to several problems, our framework should be likewise general to allow a wide adoption among different domains, i.e., BP and COPs. Second, our approach should be able to achieve a high computational throughput, by means of optimized memory accesses to avoid bandwidth bottlenecks, a careful load-balancing to

Manuscript received December 17, 2015; revised March 24, 2016; accepted July 17, 2016. This paper was recommended by Associate Editor Y. Tan.

The authors are with the Department of Computer Science, University of Verona, Verona 37134, Italy (e-mail: filippo.bistaffa@univr.it; nicola.bombieri@univr.it; alessandro.farinelli@univr.it).

This paper has supplementary downloadable multimedia material available at http://ieeexplore.ieee.org provided by the authors. This includes a PDF file, which shows proofs of propositions not included in the manuscript due to space constraints. The total size of the file is 1 MB.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCYB.2016.2593773

[1] BE for COPs uses + as composition and max as marginalization. Formally, the mathematical concept that describes these operations is the commutative semiring, discussed in detail by Aji and McEliece [9].

2168-2267 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

fully exploit the available computational power, and the adoption of well-known parallel primitives [15], [16] to reduce the CPU workload to the minimum. Third, our solution should tackle large-scale real-world problems, and, hence, it should not be limited by the amount of GPU global memory.

In this context, the work of Zheng *et al.* [17] (later improved by Zheng and Mengshoel [18]) represents, to the best of our knowledge, the only approach that tackles the high degree of parallelism of the BE composition and marginalization operators (specifically devised for BP) using GPUs. Specifically, they exploit the fact that such operators execute the same mathematical operation over a large amount of input data and, therefore, they are suitable for the single instruction multiple data model adopted by these architectures [10]. However, this approach has several drawbacks that hinder its applicability in general real-world scenarios. In fact, it accesses input data in a nonregular way by means of an indexing table, which introduces additional overhead and causes the lack of coalescence and data locality in memory transfers. Furthermore, as a consequence of this lack of regularity, input tables must fit entirely into the GPU global memory, since threads may need to access data that could be anywhere in such tables. Therefore, this method is not applicable to instances whose tables exceed this limitation. Finally, since this approach has been specifically devised for BP, where tables are complete (i.e., they include a row for every possible assignment of the variables in their scope), it is unable to cope with real-world problems with incomplete tables, e.g., COPs, weighted CSPs (WCSPs) [14] and graph-constrained coalition formation [19], where some variable assignments are unfeasible, i.e., they violate some sort of hard constraint. Notice that, even if it is possible to model incomplete tables as complete ones by representing unfeasible assignments as explicit rows with special values (i.e., $-\infty$ for a maximization task), this approach is not practical since it results in tables whose size is not manageable.

Against this background, we propose a novel GPU parallel approach for the BE message passing phase that achieves the objectives set above. In more detail, we advance the state-of-the-art in the following ways.

1) We propose an algorithm to preprocess input tables by organizing the columns in a specified order, thus achieving full memory coalesced accesses in the message passing phases. Such preprocessing phase can cope with both complete tables and incomplete ones, and can be realized using fully parallelizable operations. We formally analyze such algorithm proving its correctness and giving the worst case computational complexity.
2) We propose an implementation of the GPU kernel that exploits the table layout specified before. We show that such an arrangement enables pipelined data transfers from the host to the GPU (hence optimizing the transfer time) and it allows the use of highly efficient routines for crucial parts of the BE message passing phase (i.e., $\oplus$ and $\Downarrow$ operators). Our method is not limited by the amount of GPU memory, as our data layout allows us to process large tables by splitting them into manageable chunks that meet the memory capabilities of the GPU.
3) We empirically evaluate our approach on two particular BE realizations, i.e., BP and COP, adopting standard datasets in both cases. On the one hand, we compare our approach against the one proposed by Zheng and Mengshoel [18] on the same dataset. Our results show significant improvements, by achieving speedups at least 56% higher than the alternative method (reaching peaks of +466%). On the other hand, we employ our method to solve standard, publicly available WCSPs datasets [14], by modeling them as COPs and then by adopting the BE algorithm to compute their solution. The tests show that our method results in a speedup of 696× with respect to the serial approach. We prove the importance of handling tables that exceed the GPU global memory in this scenario, which would otherwise be unsolvable given the dimension of the instances.

II. Background

The purpose of this section is threefold. First, in Section II-A, we define the theoretical concepts and the algorithms related to BE and BP on junction trees (JTs). An in-depth discussion of such algorithms can be found, respectively, in [2] and [6]. Second, Section II-B outlines the features of GPU architectures, used to implement our highly-parallel approach. Finally, Section II-C discusses previous works related to BE on GPUs.

A. Bucket Elimination

BE [2] is a unifying algorithmic framework, which generalizes DP to accommodate algorithms for many complex problem-solving and reasoning techniques. BE usually accepts inputs in the form of a knowledge-base theory and a query encoded by several functions or relations over subsets of variables (e.g., clauses for propositional satisfiability, constraints, or conditional probability matrices for belief networks). In this paper, we will focus on the resolution of constraint networks (CNs), which consist of a set $X = \{x_1, \ldots, x_n\}$ of n variables such that $x_1 \in D_1, \ldots, x_n \in D_n$, where D_i represents the domain of the variable x_i, together with a set of m constraints $\{C_1, \ldots, C_m\}$, denoting the variables simultaneous legal assignments. Nonlegal assignments are denoted as unfeasible. In this paper, we are interested in the version of BE that computes the optimal solution for COPs (Algorithm 1) [2]. COPs can model several realistic problems [8] such as WCSPs [14].

Definition 1: A COP is a CN augmented with a set of functions. Let $F_1, \ldots, F_l$ be l real-valued functional components defined over the scopes $Q_1, \ldots, Q_l$, $Q_i \subseteq X$, let $\bar{a} = (a_1, \ldots, a_n)$ be an assignment of the variables, where $a_i \in D_i$. The global cost function F is defined by $F(\bar{a}) = \sum_{i=1}^{l} F_i(\bar{a})$, where $F_i(\bar{a})$ means F_i applied to the assignments in $\bar{a}$ restricted to the scope of F_i. Solving the COP requires to find $\bar{a}^* = (a_1^*, \ldots, a_n^*)$, satisfying all the constraints, such that $F(\bar{a}^*) = \max_{\bar{a}} F(\bar{a})$ [or $F(\bar{a}^*) = \min_{\bar{a}} F(\bar{a})$, in case of a minimization problem].

In Algorithm 1, the $\oplus$ operator is instantiated in lines 11 and 13 with the relational join and the join sum operation, respectively. Moreover, these lines instantiate the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 BE COP(CN, $F_1, \dots, F_l$)

1: Partition $\{C_1, \dots, C_m\}$ and $\{F_1, \dots, F_l\}$ into n buckets
2: **for all** $p \leftarrow n$ **down to** 1 **do**
3: **for all** $C_k, \dots, C_g$ over scopes $X_k, \dots, X_g$, and **for all** $F_h, \dots, F_j$ over scopes $Q_h, \dots, Q_j$, in bucket p **do**
4: **if** $x_p = a_p$ **then**
5: $x_p \leftarrow a_p$ in each F_i and C_i
6: Put each F_i and C_i in appropriate bucket
7: **else**
8: $U_p \leftarrow \bigcup_i X_i - \{x_p\}$
9: $V_p \leftarrow \bigcup_i Q_i - \{x_p\}$
10: $W_p \leftarrow U_p \cup V_p$
11: $C^p \leftarrow \pi_{U_p}\left(\bowtie_{i=1}^{g} C_i\right)$
12: **for all** tuples t over W_p **do**
13: $F^p(t) \leftarrow \Downarrow_{a_p \mid (t,a_p) \text{ satisfies } \{C_1,\dots,C_g\}} \bigoplus_{i=1}^{j} F_i(t, a_p)$
14: Place F^p in the latest lower bucket with a variable in W_p, and C^p in the latest lower bucket with a variable in U_p
15: Assign maximising values for the functions in each bucket
16: **return** $F(\bar{a}^*)$ and $\bar{a}^*$

$\Downarrow$ operator with the project and the maximization operations, respectively. This difference is due to the fact that constraints are relations [2], as they contain one row per legit assignment of their input variables (while excluding the unfeasible ones). On the other hand, a cost function F_i also specifies a value for each row, representing the cost of that particular variable assignment. In the context of the scope of this paper, it is crucial to note that the presence of constraints in all these problems inherently makes some variable assignments unfeasible. As a consequence, all the rows in the exchanged messages corresponding to such assignments can be dropped, greatly reducing the memory requirements of the algorithm. This is not a mere performance optimization, but it is often necessary to achieve a manageable size of the tables. In fact, the size of such tables is exponential in the induced width of the COP [2], which usually makes this approach not practical if all the rows are explicitly represented in the tables (e.g., assigning $-\infty$ to the unfeasible rows). In order to better understand our contribution to the GPU parallelization of BE, in the following sections we provide a brief description of how composition and marginalization are realized in Algorithm 1.

1) Composition: We now discuss how the $\oplus$ composition operator is implemented in Algorithm 1 by the join sum, an operation closely related to the inner join of relational algebra. For the remainder of this thesis, tables are represented according to Definition 2. Moreover, if L is a tuple of elements, we refer to its kth element with $L[k]$. We adopt the zero-based numbering convention, i.e., tuples start at index 0.

Definition 2: A table $T_i = \langle Q_i, d_i, R_i, \phi_i \rangle$ is defined by the following.

1) $Q_i \subseteq X$, a tuple of variables called the scope of T_i.
2) d_i, a tuple of natural numbers such that $d_i[k] = D_j$ is the size of the domain $Q_i[k] = x_j$, where $k \in \{1, \dots, |Q_i|\}$.
3) R_i, a tuple of rows: in particular, each row $R_i[k]$ is a tuple of natural numbers, defining a particular assignment of the variables in Q_i, where $k \in \{1, \dots, |R_i|\}$.
4) ϕ_i, a tuple representing the actual values of the function, one for each row $R_i[k]$: in particular, $\phi_i[k]$ is the value associated to the variable assignment represented by $R_i[k]$, where $k \in \{1, \dots, |R_i|\}$.

T_1

x_1	x_3	x_5	x_8	ϕ_1
0	1	0	1	α_0
1	0	0	1	α_1
1	1	0	1	α_2
0	1	0	0	α_3
0	0	0	1	α_4
1	1	1	1	α_5

$\oplus$

T_2

x_1	x_2	x_3	x_4	x_6	x_{10}	ϕ_2
1	0	0	1	1	0	β_0
1	0	1	1	1	0	β_1
0	1	0	0	1	1	β_2
1	1	0	1	0	1	β_3
0	0	0	1	1	0	β_4
1	1	1	1	1	1	β_5

Fig. 1. Original tables T_1 and T_2.

Suppose we want to compute the join sum between T_1 and T_2 (shown in Fig. 1), respectively, associated to two tuples of variables $Q_1 = \langle x_1, x_3, x_5, x_8 \rangle$ and $Q_2 = \langle x_1, x_2, x_3, x_4, x_6, x_{10} \rangle$, with $Q_1 \cap Q_2 = \langle x_1, x_3 \rangle$ representing the shared variables between T_1 and T_2. Notice that, as previously stated, some variable assignments are missing in T_1 and T_2, i.e., the unfeasible assignments.

A row in T_1 matches a row in T_2 if all the shared variables have the same values in both the rows (matching rows have been highlighted with the same color in Fig. 1). It is important to note that this is a many-to-many relationship, because multiple rows in the first table can match multiple rows in the second table.

x_1	x_3	x_5	x_8	ϕ_1
1	0	0	1	α_1

matches

x_1	x_2	x_3	x_4	x_6	x_{10}	ϕ_2
1	0	0	1	1	0	β_0
1	1	0	1	0	1	β_3

For instance because they all have $x_1 = 1$ and $x_3 = 0$. Thus, the result table will have a row for each couple of matching rows in the input tables. In the above example, the corresponding rows in the result table $T_1 \oplus T_2$ will be as follows.

x_1	x_3	x_5	x_8	x_2	x_4	x_6	x_{10}	$\phi_\oplus$
1	0	0	1	0	1	1	0	$\alpha_1 + \beta_0$
1	0	0	1	1	1	0	1	$\alpha_1 + \beta_3$

These resulting rows are obtained combining the second row of T_1 and, respectively, the first and the fourth rows of T_2. They both have the same values for the shared variables ($x_1 = 1$ and $x_3 = 0$). The values of the nonshared variables (i.e., x_5 and x_8 for T_1, and x_2, x_4, x_6, and x_{10} for T_2) are copied from the corresponding matching rows. Hence, in the above example, $x_5 = 0$ and $x_8 = 1$ for both the resulting rows (since there is only one matching row in T_1), and $x_2 = 0$, $x_4 = 1$, $x_6 = 1$, and $x_{10} = 0$ for the first resulting row (since it results from the match with the first matching row in T_2), and so on. Thus, the variable set of the resulting table is the union of the variable sets of the input tables. Finally, the values of the resulting rows are obtained summing the values of the corresponding matching rows, i.e., $\alpha_1 + \beta_0$ and $\alpha_1 + \beta_3$. Is it easy to see that if n rows in T_1 match m rows in T_2, they will result in $n \cdot m$ rows in the resulting table (Fig. 2).

2) Marginalization: The second fundamental operation, which implements the $\Downarrow$ marginalization operator in Algorithm 1, is the maximization. Suppose that, as a result of the inner join sum operation at line 13 of Algorithm 1, we obtain the table T in Fig. 3.

Now, suppose that $x_p = x_8$. Then, Algorithm 1 requires to maximize such table marginalizing out x_8, i.e., removing the column corresponding to x_8. As a result of this removal, some

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

x_1	x_3	x_5	x_8	x_2	x_4	x_6	x_{10}	$\phi_{\oplus}$
0	0	0	1	1	0	1	1	$\alpha_4 + \beta_2$
0	0	0	1	0	1	1	0	$\alpha_4 + \beta_4$
1	0	0	1	0	1	1	0	$\alpha_1 + \beta_0$
1	0	0	1	1	1	0	1	$\alpha_1 + \beta_3$
1	1	0	1	0	1	1	0	$\alpha_2 + \beta_1$
1	1	0	1	1	1	1	1	$\alpha_2 + \beta_5$
1	1	1	1	0	1	1	0	$\alpha_5 + \beta_1$
1	1	1	1	1	1	1	1	$\alpha_5 + \beta_5$

Fig. 2. Join sum result $T_{\oplus}$.

x_1	x_3	x_5	x_8	ϕ
0	0	0	0	α_0
0	0	0	1	α_1
1	0	0	0	α_2
1	0	0	1	α_3
1	1	0	1	α_4
1	1	1	1	α_5

Fig. 3. Initial table T.

x_1	x_3	x_5	$\phi_{\Downarrow}$
0	0	0	$\max(\alpha_0, \alpha_1)$
1	0	0	$\max(\alpha_2, \alpha_3)$
1	1	0	α_4
1	1	1	α_5

Fig. 4. Maximization result.

rows may now be equal considering the remaining columns (e.g., $R[1]$ and $R[2]$ both contain $\langle 0, 0, 0\rangle$ in the first three columns, as well as $R[3]$ and $R[4]$, which contain $\langle 1, 0, 0\rangle$). Since one cannot have duplicate rows, the maximization operations computes a single row that, as a value, stores the maximum of the original values.[2] The final result of the maximization of T can be seen in Fig. 4.

Composition and marginalization operators are also employed by more modern versions of BE, e.g., bucket-tree elimination (BTE) proposed by Kask *et al.* [20], hence our contributions are valuable also in the context of these newer algorithms. Here we focus on BE since it was the first version of these message-passing techniques to tackle constrained optimization, and its performance is generally comparable with BTE, which, in turn, is optimized for some specific problems, i.e., singleton-optimality problems.

Notice that DP (and in particular BE) is not the only approach to solve COPs, which can also be tackled with DFS-based approaches [21]. While they represent important techniques in the context of constrained optimization, we do not focus on these algorithms in the context of this paper, since DFS is known to be difficult to parallelize [22].

We now discuss BP on JTs, which is a close variation of BE [2] that is also based on composition and marginalization operators (i.e., scattering and reduction, respectively).

3) Belief Propagation on Junction Trees: BP on JTs [6] is used to propagate inference on a Bayesian network (BN), a representation of a joint distribution over a set of n random variables X, structured as a directed acyclic graph whose vertices are the random variables and the directed edges represent dependency relationships among the random variables. The propagation of beliefs (or posteriors) runs over a JT, generated from a BN by means of moralization and triangulation [6].

[2]If we marginalize out x_i, we maximize over up to d_i rows.

Every vertex N_i of the JT contains a set $Q_i \subseteq X$ of random variables that forms a maximal clique in the moralized and triangulated BN, each associated to a potential table represented by $T_i = \langle Q_i, d_i, R_i, \phi_i\rangle$ according to Definition 2. In standard BP, R_i contains all the possible variable assignments over their domains, hence $|R_i| = |\phi_i| = \prod_{k=1}^{|Q_i|} d_i[k]$. Assuming that T_i and T_j are potential tables associated to adjacent vertices in the JT, we associate a separator table $\text{Sep}_{ij} = \langle Q_{ij}, d_{ij}, R_{ij}, \phi_{ij}\rangle$ to the edge (N_i, N_j), whose scope Q_{ij} is represented by the shared variables between the two tables, i.e., $Q_{ij} = Q_i \cap Q_j$.

BP on JTs is then invoked whenever we receive new evidence for a particular set of variables $\mathcal{E} \subseteq X$, updating the potential tables associated to the BN in order to reflect this new information. To this end, a two-phase procedure is employed: first, in the evidence collection phase, messages are collected from each vertex N_i, starting from the leaves all the way up to a designated root vertex. Then, during evidence distribution, messages are sent from the root to the leaves. In both phases, each recursive call comprises a MESSAGEPASS procedure, which implements the propagation of beliefs between the potential tables T_i and T_j in two steps.

1) *Reduction:* The potential table Sep_{ij} is updated to Sep^*_{ij}. In particular, each row of Sep^*_{ij} is obtained summing the corresponding rows of T_i, i.e., the ones with a matching variable assignment. Reduction implements the $\Downarrow$ marginalization operator of BE, which is achieved with a summation in this case.
2) *Scattering:* T_j is updated with the new values of Sep^*_{ij}, i.e., every row of T_j is multiplied for the ratio between the corresponding rows in Sep^*_{ij} and Sep_{ij}. Following Zheng and Mengshoel [18], we assume that $(0/0) = 0$. Scattering implements the composition with the product operation, and it corresponds to the $\oplus$ operator of BE.

In both BE and BP on JTs, it is easy to see that the message passing phase require several independent computations spanning over multiple rows of the tables, suggesting a multithreaded algorithm in which such degree of parallelism can be exploited by means of GPUs.

B. GPU Architecture

GPUs are designed for compute-intensive, highly parallel computations. To this end, more transistors are devoted to data processing rather than data caching and flow control. These architectures are especially well-suited for problems that can be expressed as data-parallel computations where data elements are mapped to parallel processing threads. The GPU (device) is mainly employed to implement compute-intensive parts of an application, while control-dominant computations are performed by the CPU (host). In our approach, the GPU is programmed using the CUDA programming framework [10], which requires the definition of a kernel, a particular routine executed in parallel by thousands of threads on different inputs. Threads are organized in thread blocks, sharing fast forms of storage and synchronization primitives. On the other hand, physical and design constraints limit the number of threads per block, since all the threads of a block are expected to reside on the same streaming multiprocessor (SM) and must

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

share limited memory resources. Memory management is a crucial aspect in the design of efficient GPU algorithms, since memory accesses are particularly expensive and have a significant impact on performance. As a consequence, memory accesses must be carefully devised to achieve high computational throughput (see Section V-A). In what follows, we discuss previous approaches for BE-related algorithms that can be found in the parallel computing literature.

C. Related Work

BP on JTs represents a well-known inference algorithm, which has received significant attention in the parallel computing literature due to its high computational demands. In particular, Xia and Prasanna [23] proposed a distributed approach that combat this by decomposing the initial JT into a set of subtrees, and then they perform the evidence propagation in the subtrees in parallel on a cluster. In this paper, we also focus on exploiting parallel architectures for BP on JTs, but, in contrast, we aim at parallelizing the single propagation operation, which is the most computationally intensive task of the entire algorithm.

The most recent work addressing the parallelization of BP on GPUs is presented by Zheng *et al.* [17]. In particular, the authors pursue the same goal tackled by our approach discussed in Section III, i.e., parallelize the atomic operations of propagation so that it could be embedded in different algorithms. On the other hand, such an approach proposes a different parallelization strategy, devising a 2-D parallelism, in which an higher level element-wise parallelism is stacked on top of a lower level arithmetic parallelism, to better exploit the massive computational power provided by modern GPUs. In particular, element-wise parallelism is achieved by computing each of the $|R_{ij}|$ reduction-and-scattering operations in parallel, which require $|R_{ij}|$ mapping tables (one per row of Sep_{ij}) to allow each concurrent task to correctly locate its input data from the corresponding potential tables. On the other hand, arithmetic parallelism represents the multithreaded computation of each reduction-and-scattering operation, by means of well known parallel algorithms that can be found in [10].

Although this approach represents a significant contribution to the state-of-the-art, there are some drawbacks that hinder its applicability. In particular, the proposed memory layout is not optimized for GPUs, for three main reasons.

1) Threads need to access data in sparse and discontinuous memory locations using an additional indexing table, breaking coalescence, and drastically reducing the throughput of memory transfers (Fig. 8). Coalescence is crucial and it should be exploited in order to reduce memory accesses to the global memory, improving the compute-to-memory ratio and achieving a greater computational throughput.
2) Since input data is organized in a discontinuous pattern rather than in continuous chunks, it is mandatory to transfer the entire potential tables to the global memory of the GPU before starting the computation of the BP algorithm, hindering two desirable properties: a) this approach is not applicable to potential tables that do not fit into global memory, since the sparsity of the data prevents any possibility of splitting them into smaller parts and b) since the computation cannot be started before the entire input data has been copied to the GPU, the cost of memory transfers cannot be amortized by means of technologies like NVIDIA CUDA streams.

Moreover, the authors devise this technique for BP, where tables are complete (i.e., they include a row for every possible assignment of the variables in their scope). Thus, this approach cannot be applied to problems in which tables are incomplete, e.g., COPs and WCSPs.

In the context of COPs, the only work that specifically focuses on the implementation of the BE algorithm for many-cores architectures is the one by Fioretto *et al.* [24], in which the authors devise an algorithm to realize the join sum and the maximization operations (referred as aggregate and project) on GPUs, by exploiting the high degree of parallelism inherent in these operations. Although this approach represents a significant contribution to the state-of-the-art, there are some drawbacks that hinder its applicability. First, the indexing of the tables is executed by using a minimal perfect hash function, i.e., a hash function that maps n keys to n consecutive integers, which can be easily adopted as the indices of such keys. Although minimal perfect hash functions can be used in parallel by different threads to index the input, their construction is inherently sequential, since the index of a key depends on the indices assigned to the previously considered keys [25]. This aspect reduces the efficiency of this approach especially on big instances, as shown by our experiments in Section VI-B.

To overcome these limitations, we propose a better way to tackle the GPU computation of the message-passing phase of BE. In particular, we first present a technique that improves the parallelization of BP with complete tables (already published in [26]), and then we extend it in order to be applicable to the more general case of incomplete tables.

III. Processing Complete Tables

In this section, we detail our contribution to the GPU computation of messages in BP, exploiting the fact that, in such problem, tables are complete. In particular, we first discuss how we preprocess potential tables in order to index their rows efficiently and achieve coalesced memory accesses. Then, this table layout is exploited by the actual CUDA kernel, which executes the actual message passing phase of BP through highly efficient routines.

A. Table Preprocessing

Suppose we have to propagate new evidence from the potential table T_1 to the potential table T_2, respectively, associated to two tuples of variables $Q_1 = \langle x_3, x_2, x_1 \rangle$ and $Q_2 = \langle x_5, x_4, x_1 \rangle$, with the shared variables $Q_{12} = Q_1 \cap Q_2 = \langle x_1 \rangle$. We assume that x_1, x_3, and x_5 are binary variables, while x_2 and x_4 can assume three values. In the approach by Zheng and Mengshoel [18], each row of the separator table Sep_{12} is assigned to a different block of threads, which are responsible of the reduction of the rows of T_1 with a matching

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

T_1

x_3	x_2	x_1	ϕ_1
0	0	0	α_0
0	0	1	α_1
0	1	0	α_2
0	1	1	α_3
0	2	0	α_4
0	2	1	α_5
1	0	0	α_6
1	0	1	α_7
1	1	0	α_8
1	1	1	α_9
1	2	0	α_{10}
1	2	1	α_{11}

T_2

x_5	x_4	x_1	ϕ_2
0	0	0	β_0
0	0	1	β_1
0	1	0	β_2
0	1	1	β_3
0	2	0	β_4
0	2	1	β_5
1	0	0	β_6
1	0	1	β_7
1	1	0	β_8
1	1	1	β_9
1	2	0	β_{10}
1	2	1	β_{11}

Fig. 5. Original tables.

T_1^p

x_1	x_3	x_2	$p(\phi_1)$
0	0	0	α_0
0	0	1	α_2
0	0	2	α_4
0	1	0	α_6
0	1	1	α_8
0	1	2	α_{10}
1	0	0	α_1
1	0	1	α_3
1	0	2	α_5
1	1	0	α_7
1	1	1	α_9
1	1	2	α_{11}

T_2^p

x_1	x_5	x_4	$p(\phi_2)$
0	0	0	β_0
0	0	1	β_2
0	0	2	β_4
0	1	0	β_6
0	1	1	β_8
0	1	2	β_{10}
1	0	0	β_1
1	0	1	β_3
1	0	2	β_5
1	1	0	β_7
1	1	1	β_9
1	1	2	β_{11}

Fig. 6. Preprocessed tables.

variable assignment and the subsequent scattering on matching rows in T_2. In Fig. 5, rows associated to different blocks of threads have been marked in different colors, i.e., white and gray for $x_1 = 0$ and $x_1 = 1$, respectively. The organization of input data provided by these tables is undesirable for GPU architectures. In fact, threads responsible of the computation of white rows cannot access consecutive memory addresses, as their data is interleaved with gray rows, thus breaking memory coalescence. Moreover, even if the computation of white rows requires half of the input data, its sparsity forces us to transfer the entire tables to the global memory before starting the algorithm. We propose to solve these issues by means of a preprocessing phase, in which rows associated to the same row in Sep_{12} (i.e., rows of the same color, in the above example) are stored in consecutive addresses in the corresponding potential tables, as shown in Fig. 6. Threads responsible of white rows execute coalesced memory accesses, and start the computation while gray rows are still being transferred to the GPU. Each block of threads easily retrieves its input data without any costly mapping table, in contrast with the approach by Zheng and Mengshoel [18].

Consider $T_1^p = \langle Q_1^p, d_1^p, R_1^p, \phi_1^p \rangle$ in Fig. 6, resulting from a permutation σ of Q_1 in which the variables in Q_{12} are brought to the most significant[3] (MS) positions in $Q^p = \sigma(Q)$. In this way, we can assure that rows with the same assignment of the variables in Q_{12} form a contiguous chunk of memory.

Our table representation by means of ordered tuples imposes that d^p, R^p, and ϕ^p are coherently defined, to guarantee the equivalence to the original table. While the former can be easily obtained by applying σ to d, the computation of R^p can be avoided, therefore only ϕ^p requires a particular dissertation, which is covered in the following sections.

1) Table Indexing: Since in any table $T = \langle Q, d, R, \phi \rangle$, R contains all the possible variable assignments, we can avoid storing R in memory. In fact, since the order of variables is fixed, given any row $r = R[k]$, k can be computed with

$$k = \sum_{i=1}^{|Q|-1} \left(r[i] \underbrace{\prod_{j=i+1}^{|Q|} d[j]}_{\mathcal{D}[i]} \right) + r[|Q|] = \sum_{i=1}^{|Q|-1} (r[i] \cdot \mathcal{D}[i]) + r[|Q|] \quad (1)$$

where $r[i]$ represents the value assumed by the variable $Q[i]$ in r. Each $\mathcal{D}[i]$ represents the product of all the elements starting from position $i+1$ in d, hence we refer to the tuple $\mathcal{D}$ as the exclusive postfix product of d. Such an operation can be seen as a variation of the standard exclusive prefix sum operation, in which the result is computed by summing all the elements up to $i-1$. We define $\mathcal{D}[|Q|] \coloneqq 1$ (the identity element for the product), similarly to the definition of the first element of the exclusive prefix sum as 0.

On the other hand, each $r[i]$ can be retrieved from k as $r[i] = \lfloor k/\mathcal{D}[i] \rfloor \bmod d[i]$. Thus, R can be dropped from our representation in memory, hence, as previously claimed, the computation of R^p is unnecessary. For a better understanding, let r with $Q = \langle x_1, x_2, x_3 \rangle$ and $d = \langle 2, 16, 10 \rangle$.

$r =$

x_1	x_2	x_3	ϕ
1	10	7	v_{267}

From (1), r is in position $k = 1 \cdot d[2] \cdot d[3] + 10 \cdot d[3] + 7 = 267$ in ϕ. Moreover, $x_1 = 1 = \lfloor 267/\mathcal{D}[1] \rfloor \bmod d[1]$, $x_2 = 10 = \lfloor 267/\mathcal{D}[2] \rfloor \bmod d[2]$ and $x_3 = 7 = \lfloor 267/\mathcal{D}[3] \rfloor \bmod d[3]$.

As mentioned before, to maintain a coherent representation of the preprocessed table $T^p = \langle Q^p, d^p, R^p, \phi^p \rangle$, the values in ϕ must be correctly permuted into ϕ^p.

2) Table Reordering: This section will cover our approach to achieve the column reordering detailed in Section III-A. As mentioned before, we do not store R, since each row $r \in R$ can be retrieved from its index with the above detailed technique, hence the computation of R^p will not be covered. On the other hand, for any $\phi[k]$ at index k in ϕ it is necessary to compute its index k^p in the preprocessed table T^p to compute ϕ^p.

A naive approach would require to apply the permutation σ on each row $r = R[k]$, which comprises three steps: 1) for each k, compute the corresponding variable assignment $\langle r[1], \ldots, r[i], \ldots, r[|Q|] \rangle$; 2) apply σ on the now available sequence of $r[i]$; and 3) obtain k^p using (1). Since each of the three above mentioned steps has a complexity of $O(|Q|)$, such approach requires $O(3|\phi||Q|)$.

In what follows, we show a more efficient approach to calculate k^p. For simplicity, we first explain how to compute the index resulting from swapping the variables at positions i and j. Then, we provide an algorithm to compute k^p by means of a sequence of swaps.

Proposition 1: Given $T = \langle Q, d, R, \phi \rangle$ and $T^s = \langle Q^s, d^s, R^s, \phi^s \rangle$, where Q^s and d^s has been, respectively, obtained

[3]Variables are listed from the MS to the LS.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 2 Reorder Table($\phi, \mathcal{S}$)

1: **for all** $k \in \{1, \ldots, |\phi|\}$ **do in parallel**
2: $k^p \leftarrow k$
3: **for all** $\langle a_i, b_i \rangle \in \mathcal{S}$ **do** {For every swap in $\mathcal{S}$}
4: $k^p \leftarrow f(k^p, a_i, b_i)$ {Proposition 1}
5: SWAP($Q[a_i], Q[b_i]$) {Swap variables}
6: SWAP($d[a_i], d[b_i]$) {Swap variable domains}
7: $\phi^p[k^p] \leftarrow \phi[k]$ {Write $\phi[k]$ in position k^p of ϕ^p}
8: **return** ϕ^p

swapping $Q[i]$ with $Q[j]$ and $d[i]$ with $d[j]$ (with $i > j$), ϕ^s is a permutation of ϕ, i.e., $\phi[k] = \phi^s[k']$ and k' is

$$k' = r[1]\cdot d[2]\cdots d[i]\cdots d[j]\cdots d[|Q|]+\cdots \quad (2a)$$
$$+\ r[i]\cdot d[j+1]\cdots d[j]\cdots d[|Q|] \quad (2b)$$
$$+\ r[j+1]\cdot d[j+2]\cdots d[|Q|]+\cdots+r[i-1]\cdot d[j]\cdots d[|Q|] \quad (2c)$$
$$+\ r[j]\cdot d[i+1]\cdots d[|Q|] \quad (2d)$$
$$+\ r[i+1]\cdot d[i+2]\cdots d[|Q|]+\cdots+r[|Q|]. \quad (2e)$$

Then, $k' = f(k, i, j)$ can also be calculated as

$$k' = \overbrace{k - k \bmod \mathcal{D}[j-1]}^{(2a)} + \overbrace{\mathcal{D}[j]\cdot d[j]/d[i]\cdot \lfloor k/\mathcal{D}[i] \rfloor \bmod d[i]}^{(2b)}$$
$$+\ \overbrace{k \bmod \mathcal{D}[i]}^{(2e)} + \underbrace{d[j]/d[i]\cdot \overbrace{(k \bmod \mathcal{D}[j] - k \bmod \mathcal{D}[i-1])}^{(2c')}}_{(2c)}$$
$$+\ \underbrace{\mathcal{D}[i]\cdot \lfloor k/\mathcal{D}[j] \rfloor \bmod d[j]}_{(2d)}.$$

Proof: The proof is in the supplementary material. ■

Proposition 1 is then used to reorder any potential table T according to the layout detailed in Section III-A. More formally, let $\mathcal{S} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ be a sequence of n swaps, each represented by an ordered[4] pair of positions $\mathcal{S}[i] = \langle a_i, b_i \rangle$, so that we permute Q into $\sigma(Q)$ (moving the desired subset of variables to the MS positions) by means of the sequence of i swaps of the variables in positions a_i and b_i, as described in Proposition 1. Then, ϕ^p is computed with Algorithm 2.

The tuple of swaps $\mathcal{S}$ required to move $|Q_{12}|$ variables to the MS positions of the tables T_1 and T_2 is computed as follows. Consider T_1 and let us assume that s shared variables (with $0 \leq s \leq |Q_{12}|$) are already within the first $|Q_{12}|$ positions of the corresponding variable tuple Q_1. Then, it is sufficient to swap the $|Q_{12}|-s$ shared variables with index greater than $|Q_{12}|$ with the nonshared ones which are placed within the first $|Q_{12}|$ positions. On the other hand, table T_2 can be preprocessed by swapping each shared variable $Q_2[h]$ with $Q_2[k]$ such that $Q_2[h] = Q_1[k]$ for $k \in \{1, \ldots, |Q_{12}|\}$. This algorithm ensures the same order of the variables in Q_{12} in both tables.

As an example, we reorder the row $R_1[10] = \langle 1, 2, 0 \rangle$ in position $k = 10$ of T_1 in Fig. 5 and compute its index k^p in T_1^p. In this case, the desired order is obtained with $\mathcal{S}[1] = \langle 3, 1 \rangle$ and $\mathcal{S}[2] = \langle 3, 2 \rangle$,[5] i.e., by swapping $Q[3] = x_1$ with $Q[1] = x_3$, then swapping $Q[3] = x_3$ with $Q[2] = x_2$. Initially, $d[3] = d[1] = 2$, $\mathcal{D}[3] = 1$ and $\mathcal{D}[1] = 6$. Then, applying Proposition 1 to the row with index $k = 10$ results in $(2a) = (2e) = 0$ (since there are no variables before x_3 and after x_1), $(2b) = 6 \cdot 2/2 \cdot 0 = 0$, $(2c) = 2/2(10 \bmod 6 - 10 \bmod 2) = 4$, and $(2d) = 1 \cdot 1 = 1$, hence $f(10, 3, 1) = 5$, meaning that α_{10} would have index 5 after $\mathcal{S}[1]$. To compute its final index, we apply $\mathcal{S}[2] = \langle 3, 2 \rangle$. At this point $Q_1 = \langle x_1, x_2, x_3 \rangle$, $\mathcal{D}[3] = 1$, $\mathcal{D}[2] = d[3] = 2$ and $d[2] = 3$, hence $(2c) = (2e) = 0$ (since there are no variables after x_3 and between x_2 and x_3). On the other hand, $(2a) = 5 - 5 \bmod 6 = 0$, $(2b) = 2 \cdot 3/2 \cdot 1 = 3$ and $(2d) = 1 \cdot 2 = 2$, thus $\phi^p[5] = \alpha_{10}$ (see T_1^p).

Algorithm 2 provides a method to rearrange any couple of potential tables T_i and T_j such that the variables of their separator are moved to the MS positions (see Section III-A). We now analyze the impact of this preprocessing phase on the overall performance of the algorithm, by showing how it is more efficient than the naive approach mentioned above.

3) Computational Complexity:

Proposition 2: Algorithm 2 has a time complexity of $O(|\phi||\mathcal{S}|) \leq O(|\phi||Q_{12}|/2) < O(|\phi||Q|)$.

Proof: The proof is in the supplementary material. ■

In our experimental evaluation, we performed the variable ordering with an average of $|\mathcal{S}| = 3$ swaps, resulting in an improvement of an order of magnitude with respect to the naive approach, which, in contrast, requires tens of operations for each row. It is important to note that this preprocessing phase is done once for all, while compiling the BN in the corresponding JT. In fact, the acquisition of new evidence does not change the structure of the network itself, hence we can avoid to reorder each potential table at each BP by storing and updating the couple of corresponding reordered tables for each separator. Even if Algorithm 2 does not reorder ϕ in-place, the additional space required to store ϕ^p is amortized by discarding the original table, since it is not needed in any subsequent phase of the algorithm. Furthermore, each iteration of the external loop (lines 1–7) of Algorithm 2 is independent and can be computed in parallel. As a consequence, the worst-case time complexity of the parallel version of Algorithm 2 is $O(|\phi||\mathcal{S}|/t)$, where t is the number of threads. Given a JT $= (V, E)$, our algorithm needs to store a couple of potential tables for each separator. Since threads can index input rows on-the-fly, mapping tables can be avoided. Thus, the memory requirements are $O(2 \cdot |E|)$. In contrast, the approach proposed by Zheng and Mengshoel [18] maintains one potential table for each clique, but it needs two mapping tables for each separator table. Hence, it requires $O(V + 2 \cdot |E|)$ tables.

B. GPU Kernel Implementation

In our approach to BP on GPUs, each block of threads is responsible for one element of the separator table, which is associated to a corresponding group of rows in potential tables. Such high-level organization of the computation allows us to carry out the entire reduction and scattering stages within a single thread block, hence avoiding any costly interblock synchronization structure. On one hand, the performance of our algorithm clearly benefits from the lack of interdependence among different blocks, which would reduce the overall

[4] We assume that, for every pair $\langle a_i, b_i \rangle$, $a_i > b_i$.

[5] Swapping x_3 and x_2 is not necessary since neither of them belongs to Q_{12}, but it has been included in our example to better explain the algorithm.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

computation parallelism. On the other hand, since the size of thread blocks has an intrinsic limit imposed by the hardware architecture (e.g., 2048 threads in Kepler GPUs), the proposed organization may serialize part of the workload if the number of rows to manage exceeds such limit. Nevertheless, such an issue is not problematic in our test cases, since the above mentioned case rarely verifies. In fact, in our experimental evaluation, each block has to reduce an average of 14 elements,[6] hence allowing a full parallelization.

If the serialization is small (i.e., each thread has to reduce and scatter few rows), the effect on the overall performance is negligible. This is due to the fact that the task is computed extremely efficiently in thread-private memory space using registers. In what follows, we explain the actual implementation of the above mentioned concepts in detail.

1) Reduction: Once the input data is in the shared memory, the kernel starts the reduction phase that, in our approach, is implemented with the NVIDIA CUB library[7] by means of a block reduce raking algorithm. The algorithm consists of three steps: 1) an initial sequential reduction in registers (if each thread contributes to more than one input), in which warps other than the first one place their partial reductions into shared memory; 2) a second sequential reduction in shared memory, in which threads within the first warp accumulate data by ranking across segments of shared partial reductions; and 3) a final reduction within the raking warp based on the Kogge–Stone algorithm [27] produces the final output. This scheme is particularly efficient, since it involves a single synchronization barrier after the first phase it incurs zero bank conflicts[8] for primitive data types. On newer CUDA architectures (e.g., NVIDIA Kepler), such implementation exploits shuffle instructions, which are a new set of primitives provided by the CUDA programming language. Shuffle instructions enable threads within the same warp to exchange data through direct register accesses, hence avoiding shared memory accesses and improving the computational throughput of the algorithm.

In particular, such scheme is collectively performed by the block of threads associated to a particular element of the separator table, in order to compute its updated value as the sum of the corresponding rows of the first potential tables, i.e., the ones with a matching variable assignment. Once the reduction of the entire chunk has been completed, the first thread computes the value of R_{ij} assigned to the considered block, which serves as input for the subsequent scattering phase of BP.

2) Scattering: The final stage of BP consists of the scattering operation, which performs the actual update of T_2^p by means of R_{ij} computed in the above mentioned phase. The implementation of such operation benefits from the proposed memory layout, since it is realized with maximum parallelism and computational throughput. Each row of T_2^p is assigned to one thread, which multiplies its current value for R_{ij}, computed in the reduction phase. Once the kernel has been executed by all blocks, the propagation of belief has completed the inclusion of new evidence in T_2^p, which can be finally transferred back to the CPU memory.

Notice that our method of computing the message passing phase of BP does not affect the semantic of the algorithm, hence the results computed by our techniques are correct.

It is important to note that all the techniques described in the current section are based on the fact that, in BP on JTs, tables are complete, i.e., they contain all the possible variable assignments. Therefore, this approach cannot be directly applied to solve COPs with BE, where, as explained in Section II-A, constraints make some variable assignments unfeasible and, hence, tables are incomplete. To overcome this issue, we propose a generalized approach able to process incomplete tables, as explained in the following section.

IV. Processing Incomplete Tables

In this section, we elaborate our contribution to the parallelization of the BE algorithm to solve COPs. We propose a novel method to implement the join sum and the maximization operations (line 13 of Algorithm 1) on GPUs, to tackle their computational complexity and speedup the execution of BE.

A. Table Preprocessing

In order to explain our approach, we contextualize it on the example introduced in Section II-A1 (Fig. 1). The goal is to rearrange the rows of these tables so to have the same final placement discussed in Section III-A, since the current data organization suffers from very poor data locality.

In particular, the preprocessing of these tables aims at achieving two fundamental requirements: 1) rows of the same color should be in consecutive memory addresses, to have full coalescence in memory accesses and to reduce the sparsity of data and 2) colored groups should be in the same order (considering the set of shared variables) in both tables, to locate them efficiently when computing the join sum result. This is required since tables can be incomplete. We achieve these objectives by means of Algorithm 3. The first step of the preprocessing phase requires to move the $|Q_{12}|$ columns corresponding to the shared variables to the $|Q_{12}|$ least significant[3] (LS) places. Notice that this step is an embarrassingly parallel [10] task, and it can be trivially divided among $|R|$ threads, each independently processing a single row. Subsequently, the algorithm reorders R and ϕ by means of an LSD radix sort algorithm [15] (implemented with the NVIDIA CUB library).[7] It is not necessary to adopt a radix sort algorithm in this phase (as every sorting algorithm that operates on the basis of the LS $|Q_{12}|$ places would work). However, we decide to use such an algorithm since it can be parallelized very efficiently [15]. As a final step, the algorithm remove the nonmatching groups of rows (white rows in Fig. 1), since they do not generate any output row in the result table, obtaining the preprocessed tables T_1^p and T_2^p in Fig. 7. Notice that none of these three steps require to have an entire table stored in the global memory, thus it is possible to easily split

[6]This is the average, over all BNs, of the ratio between the average potential table size and the average separator table size (see Table I).

[7]Available at http://nvlabs.github.io/cub.

[8]If multiple memory accesses map to the same memory bank, the accesses are serialized and split into as many separate conflict-free requests as necessary, thus decreasing the effective bandwidth.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 3 Preprocess Incomplete(T_1, T_2)

1: Move shared variables in Q_1 and Q_2 to the $|Q_{12}|$ LS places
2: Sort R_1, R_2, ϕ_1, ϕ_2 using a LSD radix sort on the $|Q_{12}|$ LS places
3: Remove row groups that do not match between T_1 and T_2

T_1^p

x_5	x_8	x_1	x_3	ϕ_1^p
0	1	0	0	α_4
0	1	1	0	α_1
0	1	1	1	α_2
1	1	1	1	α_5

$\oplus$

T_2^p

x_6	x_{10}	x_2	x_4	x_1	x_3	ϕ_2^p
1	1	1	0	0	0	β_2
1	0	0	1	0	0	β_4
1	0	0	1	1	0	β_0
0	1	1	1	1	0	β_3
1	0	0	1	1	1	β_1
1	1	1	1	1	1	β_5

Fig. 7. Final preprocessed tables.

the input tables into manageable chunks meeting the memory capabilities of the GPU and preprocess them.[9]

We now discuss how it is possible to exploit this row layout to index these tables and have multiple thread efficiently locate their input to compute the join sum in parallel on the GPU.

B. Join Sum GPU Computation

Join sum implements the composition operation of BE. Our method adopts a gather paradigm [10], in which each thread is responsible for the computation of exactly one element of the output. Such a paradigm offers many advantages with respect to the counterpart approach, i.e., the scatter paradigm, in which each thread is associated to one element of input and contributes to the computation of many output elements. Scatter-based approaches have a reduced independence of the operations and they require atomic primitives (which serialize parts of the computation) to avoid race conditions, when multiple input elements concurrently modify the same output element.

In our particular case, one thread computes one particular output row at index i (i.e., both $R[i]$, the variable assignment part, and $\phi[i]$, the value part), on the basis of the two input rows associated, which can be identified as explained below. First, we compute the number of rows in each colored group for T_1 and T_2. As a result, we obtain a tuple $\mathcal{H}$ such that $\mathcal{H}[i]$ is the number of rows of the ith colored group. This operation can be seen as the computation of the histogram of the rows of the tables, which is a well-know primitive that can be parallelized very efficiently. In the above example, $\mathcal{H}_1 = \langle 1, 1, 2\rangle$ and $\mathcal{H}_2 = \langle 2, 2, 2\rangle$. These histograms are also useful to compute the number of rows of the result table, a crucial information when we have to allocate the exact amount of memory to store the result. Each group of output rows has a number of elements equal to the product of the numbers of rows of the corresponding input groups. Hence, the histogram of the result table, namely $\mathcal{H}_\oplus$, is computed as the element-wise product[10] (denoted as $*$) of the input histograms. It is easy to verify that $\langle 1, 1, 2\rangle * \langle 2, 2, 2\rangle = \langle 2, 2, 4\rangle$ is the histogram of the result table in Fig. 2. The sum of the values of such histogram is the total number of rows of the result table.

These histograms also allow each thread to efficiently locate its input rows, as well as the index of the output row it is responsible for, by indexing the colored groups in T_1 and T_2. As a first step, we compute the exclusive prefix sum of the input and output histograms, which can be done very efficiently on the GPU [16] and, in our case, it is implemented with the NVIDIA CUB library. Given an histogram $\mathcal{H}$, its exclusive prefix sum $\mathcal{P}$ is a tuple in which each element $\mathcal{P}[i]$ represents the index of the first row of the ith colored group. With these data structures, each thread can compute its row in the join sum result. Algorithm 4 represents the actual kernel function executed by the GPU, which receives as inputs the histograms of T_1 and the output histogram (i.e., $\mathcal{H}_1$ and $\mathcal{H}_\oplus$),[11] as well as the corresponding prefix sum tuples (i.e., $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_\oplus$). The variable assignment and the value parts of the output table are, respectively, denoted as $R_\oplus$ and $\phi_\oplus$.

It is important to note the absence of divergence in Algorithm 4, thanks to the fact that the only branch instruction (line 3) is used to limit the number of running threads to the amount needed, i.e., $\mathcal{H}_\oplus[bx]$. For the sake of clarity, we made a number of simplifications in Algorithm 4. First, here we do not explicitly mention the use of shared memory, which is used to exploit the data reuse[12] inherent to the join sum operation, so to avoid unnecessary memory accesses to the global memory. The properties of these memory transfers between shared and global memory are discussed in Section V-A. Furthermore, we assume that each colored group of rows is computed by exactly one block of threads. In contrast, our actual implementation realizes a dynamic load balancing by assigning the appropriate number of groups to each block, in order to achieve a higher GPU occupancy and computational throughput. This number is determined by the amount of shared memory of the GPU and the maximum number of threads per block. It is possible that a single group of rows is larger than the available shared memory: this case is managed by splitting such a group into a number of subgroups, once again with the objective of maximizing the GPU occupancy.

Algorithm 4 Join Sum Kernel($\mathcal{H}_1, \mathcal{H}_\oplus, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_\oplus$)

1: $bx \leftarrow$ block ID
2: $tx \leftarrow$ thread ID
3: **if** $tx < \mathcal{H}_\oplus[bx]$ **then**
4: $\quad idx_1 \leftarrow \mathcal{P}_1[bx] + tx \bmod \mathcal{H}_1[bx]$
5: $\quad idx_2 \leftarrow \mathcal{P}_2[bx] + \lfloor tx/\mathcal{H}_1[bx] \rfloor$
6: $\quad idx_\oplus \leftarrow \mathcal{P}_\oplus[bx] + tx$
7: $\quad$ Compute the join sum of input rows at indices idx_1 and idx_2 in T_1 and T_2
8: $\quad$ Store the results in $R_\oplus[idx_\oplus]$ and $\phi_\oplus[idx_\oplus]$

C. Maximization GPU Computation

In this section, we describe how we implement the maximization (which corresponds to the marginalization operation

[9] If it is necessary to sort a table larger that the GPU global memory, it is possible to split it into chunks, sort each of them (using the above mentioned radix sort algorithm), and then merge the sorted chunks (adopting the merge sort algorithm) on the CPU.

[10] Element-wise product is an embarrassingly parallel operation.

[11] We do not explicitly provide $\mathcal{H}_2$ to the kernel, since this information is implicitly included in $\mathcal{H}_1$ and $\mathcal{H}_\oplus$, i.e., $\mathcal{H}_2[i] = \mathcal{H}_\oplus[i]/\mathcal{H}_1[i]$.

[12] Blue rows in Fig. 2 both refer to the same input row in T_1, hence both the corresponding threads can reuse the same input data.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 5 Maximization Kernel($\mathcal{H}, \mathcal{P}$)

1: $tx \leftarrow$ thread ID
2: **if** $tx < |\mathcal{H}|$ **then**
3: $idx \leftarrow \mathcal{P}[tx]$
4: $R_{\max}[tx] \leftarrow R[idx]$ without the column in the MS place
5: $\phi_{\max}[tx] \leftarrow$ max of the $\mathcal{H}[tx]$ values starting at $\phi[idx]$

of BE) on GPUs, exploiting the data layout discussed in Section IV-A. In particular, we adopt the same preprocessing phase detailed by Algorithm 3, with the sole difference that the set of shared variables is represented by all the variables in the scope of the table, excluding the one we want to marginalize. Intuitively, this corresponds to move such a variable to the MS place. In this case, if we compute the histogram $\mathcal{H}$ and the exclusive prefix sum $\mathcal{P}$ as previously described, we are able to index the groups of rows that must be considered when computing the maximum for the output, denoted by $R_{\max}$ and $\phi_{\max}$ for the variable assignment and the value part. Algorithm 5 implements the maximization kernel.

Algorithm 5 shows a simplified version of our actual implementation, in which we generate the appropriate number of threads and blocks on the basis of the size of the input. In contrast with the join sum operation, we do not have any data reuse (i.e., each input row is accessed by exactly one thread), hence the use of shared memory is not necessary. As a final performance remark, notice that the maximization of the $\mathcal{H}[tx]$ elements at line 5 is sequentially executed by the thread. Nevertheless, this aspect has a negligible impact of the computational throughput of our approach, since $\mathcal{H}[tx]$ depends on the size of the domain of the marginalized variable[2] and, in our experiments, it is usually a small value. When the considered variable has a binary domain, line 5 collapses to one single max operation.

Notice that Algorithms 4 and 5 correctly implement the $\oplus$ and $\Downarrow$ operations of BE, hence preserving its optimality.

V. Data Transfers

In the following sections we detail how our technique allows an optimized management of data transfers, thanks to full memory coalescence and pipelining.

A. Global-Shared Memory Transfers

Memory hierarchy in GPUs follows a widely adopted design in modern hardware architectures, in which very fast but small-size memories (i.e., registers, cache, and shared memory), which are designed to assist high-performance computations, are stacked above a slower but larger memory (i.e., global memory), suitable to hold large amounts of rarely accessed data. In particular, shared memory resides on each SM and can deliver 32 bits per two clock cycles. To increase performance, it is mandatory to exploit such a low latency memory to store information that needs to be used very often. On the other hand, accessing global memory is particularly costly (400–800 clock cycles), and should be minimized to achieve a good compute-to-memory ratio.

A common programming pattern suggests to split input data into tiles that fit into shared memory (i.e., 48 KB of

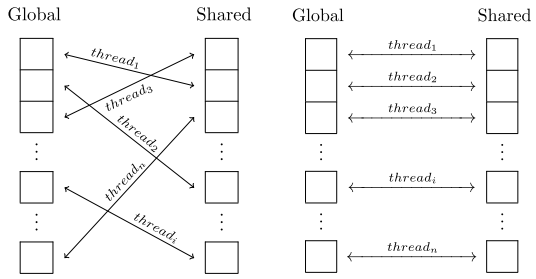

Fig. 8. Uncoalesced versus coalesced memory accesses.

information) and to complete all the computational tasks using only such data. This allows to minimize global memory accesses for each kernel execution. Coalesced accesses are the optimal way to carry out such data transfers, which is related to the principle of spatial locality of information.

Memory coalescing refers to combining multiple transfers between global and shared memory into a single transaction, so that every successive 128 bytes (i.e., 32 single precision words) can be accessed by a warp (i.e., 32 consecutive threads) in a single transaction. In general, sparse or misaligned data organization may result in uncoalesced loads, serializing memory accesses, and reducing the performance, while consecutive and properly aligned data chunks enable full memory coalescing (Fig. 8). Thanks to the previously explained preprocessing phase, the portion of input data needed by each thread block is read from global memory with fully coalesced memory accesses, since such data is already organized in consecutive addresses. The transfers are further optimized using vectorized[13] memory accesses in order to increase bandwidth, reduce instruction count, and improve latency.

B. Host-Device Data Transfers

The table layout presented in Sections III and IV allows tables to split into several data segments and threads to independently operate in each segment. This leads to a twofold improvement.

1) We devise a pipelined flow of smaller copy-and-compute operations, by amortizing the cost of CPU-GPU data transfers on the overall algorithm performance.
2) We can process tables that do not fit into global memory, by breaking them into more manageable chunks.

This allows our approach to perform BE even on problems that were intractable for previous approaches [17], [18].

1) Pipelining: The standard pattern of GPU computation requires the whole input to be transferred to the global memory before starting the kernel execution. The results are then copied back to the host memory. Such synchronous approach can be improved if the kernel can start on a partial set of input data, while the copy process is still running.

Fig. 9 shows the proposed pipelined model of computation, in which a single GPU message passing operation has been split into four stages (marked by different colors). Each computation kernel K_i executes as soon as the corresponding input

[13]Vectorized memory instructions compile to single `LD.E.128` and `ST.E.128` instructions to transfer chunks of 128 bits at a time.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

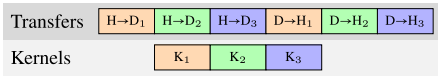

Fig. 9. Asynchronous data transfers.

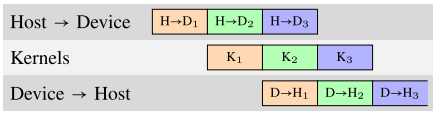

Fig. 10. Full pipeline.

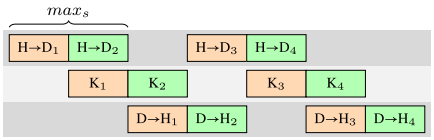

Fig. 11. Limited number of streams.

data subset has been transferred by means of $H \rightarrow D_i$. This solution applies to GPU architectures that feature only one copy engine (i.e., data between host and device can be transferred through a single channel only). Data segments for table processing are necessarily serialized, thus allowing overlapping between one kernel execution and one data transfer only. In our experiments focusing on BP, we found that, in average, this approach achieves a performance improvement of 50% with respect to synchronous data transfers. Most recent and advanced GPUs (e.g., NVIDIA Kepler) feature an additional copy engine, which enables a further degree of parallelism between data transfers and computation. On these architectures, this approach exploits the supplementary channel to overlap input and output data transfers (see Fig. 10). Such a fully pipelined model of computation achieves, in average, a performance improvement of 75% with respect to synchronous data transfers, achieved in our experiments on WCSPs.

2) Large Tables Processing: Our technique can be applied to execute BE-based algorithm even when tables do not fit into the GPU global memory, by splitting large tables into smaller data structures. In particular, this division is achieved by computing the maximum number of kernels, namely $\max_s$, which can execute at the same time without exceeding the memory capabilities of the device. In our implementation, $\max_s$ is dynamically determined at runtime as the maximum number of kernels whose total amount of input and output data can be stored into global memory. In addition, we also take into account the space constraints deriving from the use of shared memory (see Section IV-B), by enforcing that single colored chunks of data can fit in such memory. Fig. 11 shows an example, in which $\max_s = 2$. Each kernel K_i is enqueued in stream i mod $\max_s$. Transaction $H \rightarrow D_3$ cannot be scheduled in parallel with $D \rightarrow H_2$ (unlike the example of Fig. 10), as it would violate the above mentioned memory constraint. Thus, one time slot is skipped in order to complete the copy $D \rightarrow H_1$ and to free an adequate amount of memory before starting $H \rightarrow D_3$. The serialization of these two operations is a direct consequence of their execution in the same stream (i.e., stream 1). Even though the hardware constraints limit the size of data to be processed, the proposed approach allows oversized tables to be processed in multiple steps, improving scalability. As an example, 5 out of 7 instances in the considered WCSP dataset (see Section VI-B) cannot be solved without this capability of handling large tables.

VI. Experimental Results

In order to evaluate appropriately our approach presented in Sections III and IV, we conducted two different set of experiments, discussed, respectively, in Sections VI-A and VI-B. First, we discuss the results obtained executing BP on JTs (using complete tables), then we test our approach that handles incomplete tables to solve WCSPs. Our approach is implemented in CUDA[14] and all our experiments are run on a machine with an AMD A8-7600 processor, 16 GB of memory, and an NVIDIA Tesla K40.

A. BP on JTs

In this section we benchmark the approach described in Section III, i.e., CUDA-BP, which exploits the completeness of tables to achieve a better indexing of potential tables when executing BP on JTs. We compare our approach with the best approach (i.e., the SVR regression model) published by Zheng and Mengshoel [18], using the authors' implementation. We use the same BN dataset,[15] which comprises various BNs with heterogeneous structures and variable domains. We compile each BN into a JT, which is then used as input for both approaches in order to guarantee a fair comparison. Table I details some features of our JTs, i.e., the number of JT nodes resulting from their compilation and the minimum, maximum and average size of the potential and separator tables. Following Zheng and Mengshoel [18], the compilation of these networks into the corresponding JTs has been done offline, before the execution of the BP algorithm. For this reason, it has been excluded from the runtime measurements.

Table II reports the runtime in milliseconds corresponding to the following phases of the BP on JTs algorithm: 1) the total time required to complete all the reduce and scatter phases in the sequential version; 2) the total time required to preprocess all potential tables using our technique; 3) the total time required to complete the data transfers between the host and the device; 4) the total time required to complete all the reduce and scatter phases in our GPU approach; 5) our GPU speedup; 6) the total time required to complete all the reduce and scatter phases in the GPU approach by Zheng and Mengshoel [18] based on the SVR regression model; and 7) Zheng and Mengshoel's speedup [18]. Since the preprocessing phase must be done only once and can be avoided when any new evidence is received and propagated, it has not been considered in the calculation of the speedup. Moreover, we do not consider the runtimes relative to data transfers in such calculation, as such time is amortized thanks to our pipelining technique (Fig. 10). For a fair comparison, transfers are also excluded when calculating the speedup for Zheng and Mengshoel's approach [18].

[14] Available at https://github.com/filippobistaffa/CUBE.

[15] Available at http://bndg.cs.aau.dk/html/bayesian_networks.html.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
BNs

	Mildew	**Diabetes**	**Barley**	**Munin$_1$**	**Munin$_2$**	**Munin$_3$**	**Munin$_4$**	**Water**
Number of JT nodes	29	337	36	162	854	904	877	21
Max Potential size	1249280	84480	7257600	38400000	151200	156800	448000	589824
Avg Potential size	117257	29157	476133	516887	2400	3404	10102	144205
Min Potential size	336	495	216	4	4	4	4	9
Max Separator size	62464	5280	907200	2400000	6048	22400	56000	147456
Avg Separator size	3950	1698	38237	58691	204	528	1376	28527
Min Separator size	72	16	7	2	2	2	2	3

TABLE II
BP ON JTs RESULTS (TIME VALUES ARE IN MILLISECONDS)

	Mildew	**Diabetes**	**Barley**	**Munin$_1$**	**Munin$_2$**	**Munin$_3$**	**Munin$_4$**	**Water**
CPU R/S	117	219	1057	6584	54	109	315	123
Preprocessing	28	71	294	2395	24	37	106	46
Transfers	12	57	110	1464	16	39	45	15
CUDA-BP R/S	3	14	35	193	14	19	31	12
CUDA-BP Speedup	39×	15.64×	33.03×	34.11×	3.85×	5.73×	10.16×	10.25×
SVR R/S	17	34	50	648	48	38	71	14
SVR Speedup	6.88×	6.44×	21.14×	10.16×	1.12×	2.86×	4.43×	8.78×

TABLE III
WCSPs RESULTS (TIME VALUES ARE IN SECONDS)

	54	**29**	**404**	**503**	**42b**	**505b**	**408b**
Variables	67	82	100	143	190	240	200
Induced Width	11	14	19	9	18	16	24
BE Runtime	965.66	2656.72	7584.12	6347.98	31637.91	53710.41	76456.15
CUDA-BE Runtime	3.01	5.36	12.40	17.46	58.42	77.17	120.79
CUDA-BE Speedup	321.03×	495.38×	611.67×	363.63×	541.55×	696.02×	632.97×
Fioretto *et al.* Runtime	10.32	20.75	41.12	38.96	–	–	–
Fioretto *et al.* Speedup	93.57×	128.03×	184.43×	162.93×	–	–	–

In our tests, our algorithm outperforms the counterpart in the majority the instances, i.e., all except in the Water network, where runtimes are comparable. In more detail, our approach achieves speedups at least 56% higher than the counterpart in the Barley dataset (i.e., 33.03× versus 21.14×). Our best improvement with respect to the counterpart happens on the Mildew network, where our approach runs 39× faster than the CPU version, and it produces a GPU speedup that is the 466% higher than the counterpart. In general, our approach produces speedups that increase when the average potential table size increases (see Table I). In fact, we achieve speedups less than 10× only with small instances (i.e., Munin$_2$ and Munin$_3$).

B. WCSPs

WCSPs involve incomplete tables, as they contain some unfeasible variable assignments. Thus, we apply the approach in Section IV. We consider the SPOT5 dataset [14], a standard dataset that models the problem of managing an Earth observing satellite as a WCSP, to maximize the importance of the captured images, while satisfying some feasibility constraints.

The main objective of these experiments is to evaluate the speedup that can be achieved adopting our parallel approach, which is compared to a sequential BE version that uses a simple implementation for the join sum and the maximization operations. Our speedup is compared with the one achieved by the approach by Fioretto *et al.* [24] (i.e., the most recent GPU implementation of the BE algorithm) considering the same sequential BE implementation. The counterpart approach is implemented using the source code provided by the authors.

Table III shows the runtime in seconds (including preprocessing and data transfers) needed to solve the instances of our reference domain by our parallel approach, i.e., CUDA-BE, compared to its sequential version, i.e., BE. Such table also reports the number of variables and the induced width of these instances. The results show that CUDA-BE provides a speedup of at least two orders of magnitude with respect to the sequential algorithm, by reaching a maximum of 696.02×. Such speedup increases consistently with the size of the instances (i.e., the induced width and the number of variables), showing that the proposed approach correctly exploits the increased amount of parallelism in bigger tables. In fact, the speedup provided by CUDA-BE monotonically increases in the first three WCSP instances (i.e., 54, 29, and 404), in which both the number of variables and the induced width increase. On the other hand, such speedup decreases in instance 503, which, despite having a larger number of variables, is characterized by a lower induced width. Notice that the induced width has a stronger influence on the complexity of the problem [2]. The ability of our method of handling large tables is crucial in this scenario. In fact, 5 out of 7 instances (i.e., 404, 503, 42b, 505b, and 408b) cannot be solved without this feature, as their tables exceed the amount of GPU memory. Finally, notice that our approach outperforms the approach by Fioretto *et al.* [24] both in terms of runtime and scalability. On the one hand, our approach is, on average, 3.21× faster than the counterpart in the solution of the first four instances. On the other hand, the counterpart approach cannot handle the three biggest instances of the SPOT5 dataset, probably due to the different representation of the tables in memory.

VII. CONCLUSION

In this paper, we considered the BE framework and we proposed an efficient and scalable highly-parallel approach

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

that is able to harness the computational power of modern GPUs by means of an appropriate data organization in memory. The proposed approach applies also to problems involving incomplete tables, i.e., tables that do not contain all variable assignments, as well as problems that do not fit into the global memory of the GPU. Furthermore, it enables pipelined data transfers between host and device, thus further improving performance. Our experimental results show that our approach outperforms the state-of-the-art approach for BP on JTs proposed by Zheng and Mengshoel [18], by obtaining speedups ranging from +56% to +466%. The tests on WCSPs confirmed the ability of our technique to achieve a high computational throughput (reaching a speedup of 696.02× with respect to the CPU version), and proving the importance of its ability to process large tables, a necessary feature to solve these instances.

Future work will look at integrating the proposed GPU techniques in other algorithmic frameworks, such as AND/OR search-based approaches [21], in which mini-BE heuristics [2] are used to guide the search.

REFERENCES

[1] S. Dasgupta, C. Papadimitriou, and U. Vazirani, *Algorithms*. Boston, MA, USA: McGraw-Hill, 2006.

[2] R. Dechter, *Constraint Processing*. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[3] V. Chandru, "Variable elimination in linear constraints," *Comput. J.*, vol. 36, no. 5, pp. 463–472, 1993.

[4] R. Dechter and J. Pearl, "Network-based heuristics for constraint-satisfaction problems," *Artif. Intell.*, vol. 34, no. 1, pp. 1–38, 1987.

[5] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[6] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *J. Roy. Stat. Soc.*, vol. 50, no. 2, pp. 157–224, 1988.

[7] U. Bertelè and F. Brioschi, *Nonserial Dynamic Programming*. New York, NY, USA: Academic Press, 1972.

[8] A. Petcu, "A class of algorithms for distributed constraint optimization," Ph.D. dissertation, Dept. Comput. Sci. Commun., École Polytechnique Fédérale De Lausanne, Lausanne, Switzerland, 2007.

[9] S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Trans. Inf. Theory*, vol. 46, no. 2, pp. 325–343, Mar. 2000.

[10] R. Farber, *CUDA Application Design and Development*. Amsterdam, The Netherlands: Elsevier, 2011.

[11] S. Balla-Arabé, X. Gao, D. Ginhac, V. Brost, and F. Yang, "Architecture-driven level set optimization: From clustering to subpixel image segmentation," *IEEE Trans. Cybern.*, no. 99, pp. 1–14, Dec. 2015.

[12] D. Berjón, G. Gallego, C. Cuevas, F. Morán, and N. García, "Optimal piecewise linear function approximation for GPU-based applications," *IEEE Trans. Cybern.*, no. 99, pp. 1–12, Oct. 2015.

[13] Y. Tan and K. Ding, "A survey on GPU-based implementation of swarm intelligence algorithms," *IEEE Trans. Cybern.*, no. 99, pp. 1–14, Nov. 2015.

[14] E. Bensana, M. Lemaitre, and G. Verfaillie, "Earth observation satellite management," *Constraints*, vol. 4, no. 3, pp. 293–299, 1999.

[15] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. IEEE IPDPS*, Rome, Italy, 2009, pp. 1–10.

[16] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. ACM GH*, San Diego, CA, USA, 2007, pp. 97–106.

[17] L. Zheng, O. J. Mengshoel, and J. Chong, "Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization," in *Proc. UAI*, Barcelona, Spain, 2011, pp. 822–830.

[18] L. Zheng and O. Mengshoel, "Optimizing parallel belief propagation in junction trees using regression," in *Proc. SIGKDD*, Chicago, IL, USA, 2013, pp. 757–765.

[19] F. Bistaffa, A. Farinelli, and S. D. Ramchurn, "Sharing rides with friends: A coalition formation algorithm for ridesharing," in *Proc. AAAI*, Austin, TX, USA, 2015, pp. 608–614.

[20] K. Kask, R. Dechter, J. Larrosa, and A. Dechter, "Unifying tree decompositions for reasoning in graphical models," *Artif. Intell.*, vol. 166, nos. 1–2, pp. 165–193, 2005.

[21] N. Flerova, R. Marinescu, and R. Dechter, "Weighted heuristic anytime search: New schemes for optimization over graphical models," *Ann. Math. Artif. Intell.*, pp. 1–52, Jan. 2016.

[22] J. H. Reif, "Depth-first search is inherently sequential," *Inf. Process. Lett.*, vol. 20, no. 5, pp. 229–234, 1985.

[23] Y. Xia and V. K. Prasanna, "Distributed evidence propagation in junction trees on clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 7, pp. 1169–1177, Jul. 2012.

[24] F. Fioretto, T. Le, E. Pontelli, W. Yeoh, and T. C. Son, "Exploiting GPUs in solving (distributed) constraint optimization problems with dynamic programming," in *Principles and Practice of Constraint Programming*. Cham, Switzerland: Springer, 2015, pp. 121–139.

[25] D. A. F. Alcantara, "Efficient hash tables on the GPU," Ph.D. dissertation, Dept. Comput. Sci., Univ. California at Davis, Davis, CA, USA, 2011.

[26] F. Bistaffa, A. Farinelli, and N. Bombieri, "Optimising memory management for belief propagation in junction trees using GPGPUs," in *Proc. IEEE ICPADS*, Hsinchu, Taiwan, 2014, pp. 526–533.

[27] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22, no. 8, pp. 786–793, Aug. 1973.

Filippo Bistaffa received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2016.

He is a Research Associate with the Department of Computer Science, University of Verona. His current research interests include combinatorial optimization problems for realistic applications and graphics processing unit computing.

Nicola Bombieri received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2008.

Since 2008, he has been a Researcher and a Professor Assistant with the Department of Computer Science, University of Verona. He has been involved in several research projects and has published over 70 papers on conference proceedings and journals. His current research interests include parallel computing, design and verification of embedded systems, and automatic generation and optimization of embedded SW.

Alessandro Farinelli received the Ph.D. degree in computer science from the University "La Sapienza," Rome, Italy, in 2005.

He is an Associate Professor with the Department of Computer Science, University of Verona, Verona, Italy. His current research interest includes theoretical and practical issues related to the development of artificial intelligent systems applied to robotics.