ECAI 2016
G.A. Kaminka et al. (Eds.)

© 2016 The Authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/978-1-61499-672-9-125

CUBE: A CUDA Approach for Bucket Elimination on GPUs

Filippo Bistaffa[1] and **Nicola Bombieri**[2] and **Alessandro Farinelli**[3]

Abstract. We consider Bucket Elimination (BE), a popular algorithmic framework to solve Constraint Optimisation Problems (COPs). We focus on the parallelisation of the most computationally intensive operations of BE, i.e., join sum and maximisation, which are key ingredients in several close variants of the BE framework (including Belief Propagation on Junction Trees and Distributed COP techniques such as ActionGDL and DPOP). In particular, we propose CUBE, a highly-parallel GPU implementation of such operations, which adopts an efficient memory layout allowing all threads to independently locate their input and output addresses in memory, hence achieving a high computational throughput. We compare CUBE with the most recent GPU implementation of BE. Our results show that CUBE achieves significant speed-ups (up to two orders of magnitude) w.r.t. the counterpart approach, showing a dramatic decrease of the runtime w.r.t. the serial version (i.e., up to 652× faster). More important, such speed-ups increase when the complexity of the problem grows, showing that CUBE correctly exploits the additional degree of parallelism inherent in the problem.

1 INTRODUCTION

Bucket Elimination (BE) [9] is a general algorithmic framework that adopts Dynamic Programming (DP) to incorporate many reasoning techniques. In this paper, we focus on the version of BE that solves Constraint Optimisation Problems (COPs), a general class of problems that can be used to model several optimisation scenarios [8]. BE operates on functions in tabular form by means of two fundamental operations, i.e., *join sum* and *maximisation*, which are the most computationally intensive tasks of the entire algorithm. Such operations are also the key ingredients in several close variants of the BE framework, including Belief Propagation (BP) on Junction Trees [15], and Distributed COP techniques such as ActionGDL [24] and DPOP [20].

Nevertheless, in many large COP instances BE may result in prohibitive computation requirements (both in memory and runtime), as its computational complexity is exponential in the *induced width* of the graph representation of the problem [9]. For this reason, several works in the constrained optimisation literature have tried to deal with this complexity adopting various approaches. On the one hand, Dechter [7] proposed Mini-Bucket Elimination, an approximate version of BE with limited memory requirements and reduced computation. On the other hand, a recent strand of literature [17, 18] has investigated the use of AND/OR search trees, proposing several heuristic approaches and bounding methods to reduce the search space.

In this paper we propose the use of parallel architectures to speed-up the computation associated to COP solution techniques. In particular, in recent years, many computationally intensive applications have successfully employed Graphics Processing Units (GPUs), achieving speed-ups of several orders of magnitude [10]. Parallelisation has also been investigated to speed-up search-based approaches for COP on multi-core CPUs [19], but the application of these techniques to GPUs is difficult for several reasons. On the one hand, general depth-first search is known to be difficult to parallelise [21], especially on highly parallel architectures such as GPUs. Moreover, the use of branch-and-bound may result in heavily unbalanced search trees, requiring complex techniques to balance the workload among the threads [19]. Such techniques are not effective on GPUs, where load balancing is crucial to achieve a high computational throughput.

Against this background, in this paper, we investigate the use of GPUs for BE, motivated by the above discussion, by previous works that successfully parallelised DP on GPUs [13, 6, 23, 5], and by the work of Fioretto et al. [11], who recently proposed a GPU approach for BE. Specifically, we propose CUBE (CUda Bucket Elimination), providing a highly parallel implementation for the join sum and maximisation operations associated to BE. CUBE proposes a novel methodology for the parallelisation of such operations, which is specifically designed to consider two fundamental aspects of the GPU algorithmic design: thread independence and memory management. This allows CUBE to achieve significant speed-ups with respect to previous approaches and specifically to Fioretto et al. [11].

Our work opens future research developments in the field of constraint optimisation as it provides valuable techniques that can help improving the performance, apart from BE itself, in other algorithmic frameworks that adopt join sum and maximisation as subroutines. These operations represent the key ingredients of several solution techniques for COPs that have been proposed to overcome the memory requirements of BE, such as Mini-Bucket Elimination [7] (which adopts the same join sum and maximisation operations discussed in this paper), and the AND/OR search-based approaches proposed by Marinescu and Dechter [17, 18], in which Mini-Bucket heuristics are used to guide the search.

In more detail, this paper advances the state-of-the-art in the following ways:

- We propose a computational model for the join sum and maximisation operations where each thread is completely independent from the others. More specifically, in our approach each thread retrieves its input data and performs the necessary computations avoiding any interaction with the other threads. This allows us to significantly reduce sequential computation, hence fully exploiting the computational capabilities of the GPU.

[1] University of Verona, Italy, email: filippo.bistaffa@univr.it
[2] University of Verona, Italy, email: nicola.bombieri@univr.it
[3] University of Verona, Italy, email: alessandro.farinelli@univr.it

- We avoid unnecessary, expensive memory accesses by proposing a technique that allows threads to locate their input data only on the base of their own ID. We take advantage of the *data reuse* pattern inherent in the join sum and the maximisation operations by caching the input data in the *shared memory*, i.e., the fastest form of memory in the GPU hierarchy [10]. Bandwidth efficiency is also ensured by the high spatial locality inherent in our data representation, achieved through a technique recently proposed by Bistaffa et al. [5] for BP.
- We compare CUBE to the approach proposed by Fioretto et al. [11] on the same experimental settings, i.e., we used the same dataset and the same baseline sequential benchmark (i.e., FRODO [16]). Our results show that CUBE is up to $652\times$ faster than FRODO and that the speed-up obtained by CUBE is up to two orders of magnitude higher than the other GPU approach. In contrast to the approach proposed by Fioretto et al. [11], the speed-ups achieved by CUBE increase when the complexity of the problem grows, thus showing that CUBE correctly exploits the degree of parallelism inherent in the problem. This improvement allows us to compute solutions for COP instances that could not be tackled by previous BE approaches in a reasonable amount of time, showing that our approach is a viable method for real-world problems.

2 BACKGROUND

In this section, we first provide a brief introduction to the BE algorithm (Section 2.1), while Section 2.2 discusses previous works related to BE on GPUs. Section 2.3 describes the main features of GPUs, and Section 2.4 discusses the table memory layout employed by CUBE [5].

2.1 Bucket Elimination

Bucket Elimination (BE) [9] is a general algorithmic framework that adopts DP to incorporate many reasoning techniques. The input of BE is given as a knowledge-base theory 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 work, we focus on *Constraint Networks* (CN), following the definitions provided by Dechter [9].

Definition 1 *A* Constraint Network *(CN) consists 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* $\{C_1, \ldots, C_m\}$.

Definition 2 *A* constraint C_i *is a relation defined on a set* $X_i = \{x_{i_1}, \ldots, x_{i_h}\}$ *of* h *variables, called the* scope *of the constraint, such that* $X_i \subseteq X$*. Such a relation denotes the variables simultaneous legal assignments. Non-legal assignments are denoted as* unfeasible*. Notice that* C_i *is a subset of the Cartesian product* $D_{i_1} \times \cdots \times D_{i_h}$.

In this work we focus on the version of BE that solves COPs, and specifically on Algorithm 1 detailed by Dechter [9]. COPs are a general class of problems, which can be used to model several optimisation scenarios [8].

Definition 3 *A* Constraint Optimisation Problem *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* $\bar{\mathrm{a}} = (\mathrm{a}_1, \ldots, \mathrm{a}_n)$ *be an assignment of the variables, where* $\mathrm{a}_i \in D_i$*. The* global cost function F *is defined by* $F(\bar{\mathrm{a}}) = \sum_{i=1}^{l} F_i(\bar{\mathrm{a}})$*, where* $F_i(\bar{\mathrm{a}})$ *means* F_i *applied to the assignments in* $\bar{\mathrm{a}}$ *restricted to the scope of* F_i*. Solving the COP requires to find* $\bar{\mathrm{a}}^* = (\mathrm{a}_1^*, \ldots, \mathrm{a}_n^*)$*, satisfying all the constraints, such that* $F(\bar{\mathrm{a}}^*) = max_{\bar{\mathrm{a}}} F(\bar{\mathrm{a}})$ *(or* $F(\bar{\mathrm{a}}^*) = min_{\bar{\mathrm{a}}} F(\bar{\mathrm{a}})$*, in case of a minimisation problem).*

Algorithm 1 BUCKETELIMINATIONCOP ($\mathrm{CN}, F_1, \ldots, F_l, o$)

1: Partition $\{C_1, \ldots, C_m\}$ and $\{F_1, \ldots, F_l\}$ into n buckets according to o
2: **for all** $p \leftarrow n$ down to 1 **do**
3: **for all** $C_k, \ldots, C_g$ over scopes $X_k, \ldots, X_g$, and **for all** $F_h, \ldots, F_j$ over scopes $Q_h, \ldots, Q_j$, **in** bucket p **do**
4: **if** $x_p = \mathrm{a}_p$ **then**
5: $x_p \leftarrow \mathrm{a}_p$ in each F_i and C_i
6: Put each F_i and C_i in appropriate bucket
7: **else**
8: $U_p \leftarrow \bigcup_i X_i \quad \{x_p\}$
9: $V_p \leftarrow \bigcup_i Q_i \quad \{x_p\}$
10: $W_p \leftarrow U_p \cup V_p$
11: $C_p \leftarrow \pi_{U_p} (\bowtie_{i=1}^{g} C_i)$
12: **for all** tuples t over W_p **do**
13: $H_p\ (t) \leftarrow \Downarrow_{\mathrm{a}_p:(t,\mathrm{a}_p) \text{ satisfies } \{C_1,\ldots,C_g\}} \bigoplus_{i=1}^{j} F_i\ (t, \mathrm{a}_p)$
14: Place H_p in the latest lower bucket mentioning 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** $F(\bar{\mathrm{a}}^*)$, i.e., the optimal cost computed in the first bucket and $\bar{\mathrm{a}}^*$, i.e., the optimal assignment

BE operates on the basis of a *variable ordering* o, which is used to partition the set of functions into n sets $B_1, \ldots, B_n$ called *buckets*, each associated to one variable of the COP. In particular, each function F_i is placed in the bucket associated to the last bucket that is associated with a variable in Q_i, i.e., the scope of F_i. Figure 2 shows the buckets corresponding to the example COP in Figure 1, adopting the ordering $o = \langle x_1, x_3, x_2, x_5, x_4, x_6 \rangle$.

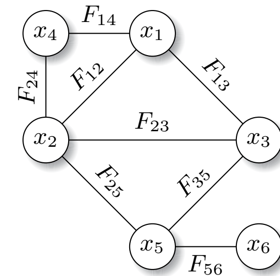

Figure 1: Example COP.

Then, buckets are processed from last to first (top to bottom), by means of two fundamental operations, i.e., *join sum* (denoted as $\oplus$) and *maximisation* (denoted as $\Downarrow$). Specifically, all the cost functions in B_p, i.e., the current bucket, are *composed* with the $\oplus$ operation, and the result is the input of a $\Downarrow$ operation. Such operation removes x_p (i.e., the variable associated to B_p) from the table, and produces a new function H_p that does not involve x_p, which is then placed in the last bucket that is associated to a variable appearing in the scope of the new function.

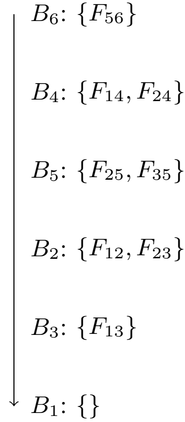

Figure 2: Initial buckets.

Figure 3 shows the execution of BE on the previous example. In particular, if a bucket, say B_4, contains more than one F_i, such functions are first composed with $\oplus$ and then the corresponding variable (i.e., x_4) is maximised out. In Figure 3, we represent these two subsequent operations by means of the compact notation $\Downarrow\oplus$. In the case of B_4, the result of $\Downarrow\oplus$ is a function $h_4(x_1, x_2)$ without x_4, which is placed in B_2. By operating in such a way, we can guarantee that the resulting function in the first bucket (i.e., $H_3(x_1)$ in Figure 3) contains only the first variable in o, i.e., x_1, since all the remaining ones have been maximised out during the previous steps. Hence, we compute the optimal assignment for x_1 as the one that maximises $H_3(x_1)$, and propagate such assignment back to the second bucket. Then, we proceed in the same way as before, computing the optimal assignment for the corresponding variable, and propagating the result until all buckets have been processed. Such process terminates when the optimal assignment for all variables has been computed.

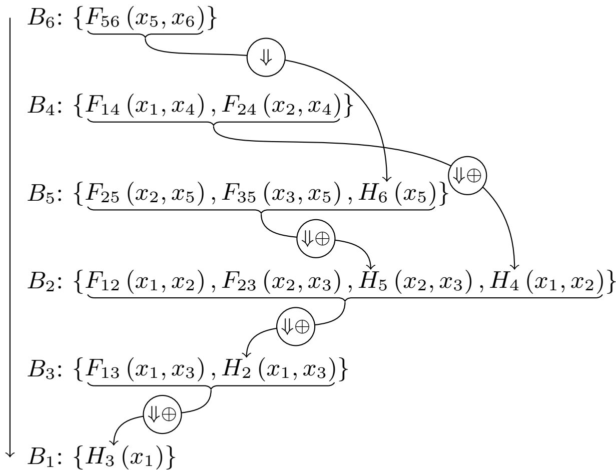

Figure 3: BE execution.

Dechter [9] proves that the computational complexity of the BE algorithm is directly determined by the ordering o.

Proposition 1 *The complexity of BE is time and space exponential in $w^*(o)$, the induced width of the problem given the variable ordering o, i.e., $O\left(m \cdot k^{w^*(o)}\right)$, where k bounds the domain size and m is the number of constraints.*

As a consequence, it is of utmost importance to adopt a variable ordering o that minimises the induced width $w^*(o)$. Unfortunately, the task of computing such ordering is NP-complete [9], and, for this reason, a greedy procedure (Algorithm 2) [9] is usually adopted to compute a variable ordering of acceptable quality.

Algorithm 2 GREEDYORDERING $(\text{CN}, metric(\cdot))$

1: **for all** $k \leftarrow n$ down to 1 **do**
2: $\quad x^* = \arg\min_{x_i \in X} metric(x_i)$
3: $\quad o[k] \leftarrow x^*$
4: $\quad$ Introduce edges in CN between all neighbours of x^*
5: $\quad$ Remove x^* from CN
6: **return** o

Notice that Algorithm 2 can be parametrised with different $metric(\cdot)$ functions, that evaluate each node on the basis of different properties. The most commonly used are the *min-degree* heuristic (in which $metric(x_i)$ is the number of neighbours of x_i) and the *min-fill* heuristic (in which $metric(x_i)$ is the number of edges that need to be added to the graph due to the elimination of x_i).

2.2 Related Work

To the best of our knowledge, the only work that specifically focuses on the implementation of the BE algorithm for many-cores architectures is the one by Fioretto et al. [11], in which the authors devise an algorithm to realise the join sum and the maximisation 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 [3], 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 [2]. This aspect reduces the efficiency of this approach especially on big instances, as shown by our experiments in Section 4. In contrast, our focus on thread independence and memory management allows us to obtain better speed-ups that increase when growing the size of the instances.

This is possible thanks to the preprocessing technique proposed by Bistaffa et al. [5], which exploits the improved table layout achieved with such preprocessing to implement an efficient GPU version of the base operations of Belief Propagation on Junction Trees (BP on JTs) [15], i.e., *reduction* and *scattering*.

Nonetheless, their approach cannot be directly applied to BE. On the one hand, the join sum operation is fundamentally different from both reduction and scattering, as the reduction corresponds to the maximisation in BE, and scattering computes a completely different output than the join sum. On the other hand, Bistaffa et al. [5] do not adopt the current state-of-the-art technique to implement the reduction operation (i.e., *segmented reduction*) and, hence, their approach can suffer from a reduced computational throughput.

2.3 GPUs

GPUs are designed for compute-intensive, highly parallel computations. These architectures perform particularly well on problems that can be modelled as data-parallel computations where data elements correspond to parallel processing threads, as they are designed on the basis of the Single Instruction Multiple Data (SIMD) model [10]. We program the GPU using the NVIDIA CUDA framework, which requires the definition of particular functions, called *kernels*, executed in parallel by thousands of threads on different inputs. Threads are grouped into thread *blocks*. Threads of the same block share fast forms of storage and synchronisation primitives. Memory plays a crucial role in the design of efficient GPU algorithms. In fact, modern GPUs contain very fast but small-size memories (i.e., registers, cache and *shared memory*), intended to assist high performance computations, stacked above a slower but larger memory (i.e., *global memory*), suitable to hold large amounts of data. Accessing global memory is particularly expensive, and should be reduced as much as possible. To do that, a common practice suggests to exploit data locality, i.e., transferring small portions of frequently used data from global to shared memory and to complete all the computational tasks that use such data before accessing to new one. This allows minimising global memory accesses. Such transfers should be executed in order to have consecutive threads fetching data from consecutive memory addresses, which is denoted as memory *coalescing* (Figure 4). Coalesced accesses are related to the principle of locality of information and they allow the hardware to combine multiple transfers between global and shared memory into a single transaction. In contrast, sparse data results in poor memory performance (Figure 5).

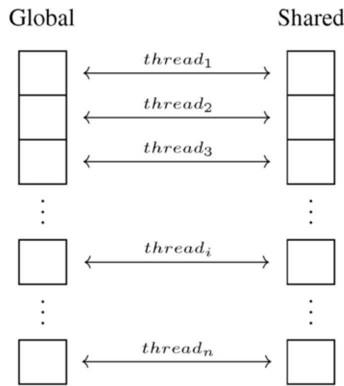

Figure 4: Coalesced accesses.

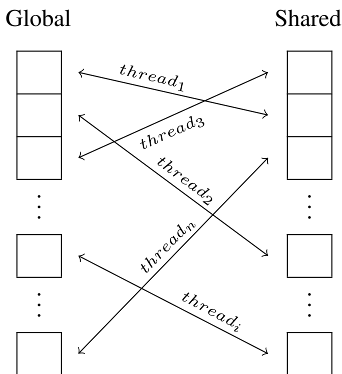

Figure 5: Uncoalesced accesses.

2.4 Preprocessing Tables

BE, as well as BP on JTs, operates on functions in tabular form considering groups of rows having the same assignments of the *shared* variables between two tables. As an example, both tables in Figure 6 contain x_1, thus BE (and, in particular, the join sum and maximisation operations) will operate on groups having the same value for x_1 (coloured in white and grey). Bistaffa et al. [5] notice that, in general, the arrangement of rows may suffer from poor data locality (i.e., white and grey groups are interleaved in Figure 6), reducing the efficiency of computations associated to BP. Thus, they propose a preprocessing approach for tables to achieve the row arrangement shown in Figure 7. Such arrangement allows optimised memory accesses and it enables tables to be split into smaller chunks (which are now in consecutive memory addresses), so to handle tables that may not fit into the GPU global memory. This table layout enables the GPU to execute *coalesced* loads, grouping several memory accesses and improving the computational throughput (Figure 5).

T_1

x_3	x_2	x_1	ϕ^1
0	0	0	α_1
0	0	1	α_2
0	1	0	α_3
0	1	1	α_4
0	2	0	α_5
0	2	1	α_6
1	0	0	α_7
1	0	1	α_8
1	1	0	α_9
1	1	1	α_{10}
1	2	0	α_{11}
1	2	1	α_{12}

T_2

x_5	x_4	x_1	ϕ^2
0	0	0	β_1
0	0	1	β_2
0	1	0	β_3
0	1	1	β_4
0	2	0	β_5
0	2	1	β_6
1	0	0	β_7
1	0	1	β_8
1	1	0	β_9
1	1	1	β_{10}
1	2	0	β_{11}
1	2	1	β_{12}

Figure 6: Original tables.

T_1^p

x_1	x_3	x_2	$p(\phi^1)$
0	0	0	α_1
0	0	1	α_3
0	0	2	α_5
0	1	0	α_7
0	1	1	α_9
0	1	2	α_{11}
1	0	0	α_2
1	0	1	α_4
1	0	2	α_6
1	1	0	α_8
1	1	1	α_{10}
1	1	2	α_{12}

T_2^p

x_1	x_5	x_4	$p(\phi^2)$
0	0	0	β_1
0	0	1	β_3
0	0	2	β_5
0	1	0	β_7
0	1	1	β_9
0	1	2	β_{11}
1	0	0	β_2
1	0	1	β_4
1	0	2	β_6
1	1	0	β_8
1	1	1	β_{10}
1	1	2	β_{12}

Figure 7: Preprocessed tables.

3 BE ON GPUS

This section presents CUBE, a GPU implementation of the joint sum and maximisation operations of BE.

3.1 Join Sum on GPUs

We first discuss the implementation of the join sum operation on GPUs. Such operation, denoted as $\oplus$, is very similar to the *join* of relational algebra, in which the output table contains one row for each couple of rows of the input tables that have a matching assignment of the shared variables. In the case of the join sum, the value of each row is given by the sum of the values of the corresponding input rows. To better explain how the join sum works, we consider the tables T_1^p and T_2^p in Figure 7. In what follows, we denote as *group* a set of rows that all have the same assignment over the shared variables, or, more intuitively, the same colour.

In order to achieve a full parallelisation of the join sum, we adopt the *gather* paradigm [14], in which each thread is responsible of the computation of exactly one element of the output. Such a paradigm offers many advantages w.r.t. the counterpart approach, i.e., the *scatter*[4] paradigm, in which each thread is associated to one element of the input and contributes to the computation of many output elements. In fact, scatter-based algorithms have a reduced degree of parallelism since they often require atomic primitives (which inherently serialise parts of the computation) to avoid having multiple threads concurrently operating on the same output. As previously discussed, only the array ϕ is stored in memory, since we assume that tables are *complete*[5] and, hence, it is not necessary to store the variable assignment part. Therefore, we only discuss on how we compute the array ϕ of the output table $T_i \oplus T_j$, denoted as $\phi_\oplus$. We map one GPU thread t to each element of $\phi_\oplus$, denoted as $\phi_\oplus[t]$.[6]

Our main goal is that each thread should be capable of computing the indices of its input rows in T_1^p and T_2^p in a closed form only on the base of its own ID t, with the aim of avoiding unnecessary memory accesses to the input data. To achieve this, we now introduce some background concepts needed to explain our indexing approach. First, notice that the number of rows in each group is equal to the number of all the possible assignments of the non-shared variables in the scope of the table, i.e., the product of the domain sizes of such variables. In particular, each group in T_1^p consists of 6 rows, as $|D_2| \cdot |D_3| = 6$, and the same applies to T_2^p, i.e., $|D_4| \cdot |D_5| = 6$. Since the join sum operation associates each of these 6 rows in T_1^p to each of the 6 matching rows in T_2^p, the corresponding group in the output table will contain $|D_2| \cdot |D_3| \cdot |D_4| \cdot |D_5| = 36$ rows. In general, it is easy to verify that, if $X_i = \{x_{i_1}, \dots, x_{i_h}\}$ and $X_j = \{x_{j_1}, \dots, x_{j_k}\}$ are the scopes of the input tables T_i and T_j, the output table $T_i \oplus T_j$ (where the $\oplus$ operator represents the join sum) contains a number of rows equal to

$$\left(\prod_{x_a \in X_i \cap X_j} |D_a| \right) \cdot \underbrace{\left(\prod_{x_b \in X_i \quad X_j} |D_b| \right) \cdot \left(\prod_{x_c \in X_j \quad X_i} |D_c| \right)}_{rows(X_i, X_j)} . \tag{1}$$

For convenience, we define the function $rows$ to denote the number of rows in each group of the output table induced by the scopes X_i and X_j. Formally, $rows : 2^X \times 2^X \to \mathbb{N}$, where 2^X denotes the powerset of X.

Algorithm 3 JOINSUMGPU(t, X_i, X_j)

1: $g \leftarrow \lfloor \frac{t}{rows(X_i, X_j)} \rfloor$ {Output group t belongs to}
2: $idx \leftarrow t \bmod rows(X_i, X_j)$ {ID of t within g}
3: $\#_i \leftarrow \prod_{x_b \in X_i \quad X_j} |D_b|$ {# of rows associated to g in T_i}
4: $\#_j \leftarrow \prod_{x_c \in X_j \quad X_i} |D_c|$ {# of rows associated to g in T_j}
5: $\gamma \leftarrow g \cdot \#_i + \lfloor \frac{idx}{\#_j} \rfloor$ {Input row in T_i}
6: $\delta \leftarrow g \cdot \#_j + idx \bmod \#_j$ {Input row in T_j}
7: $\phi_\oplus[t] \leftarrow \phi_i[\gamma] + \phi_j[\delta]$ {Compute and store output}

Algorithm 3 summarises the approach we propose to compute the join sum of two tables T_i and T_j, which is executed in parallel by each thread to index the input tables and to compute each row of the output table. As a first step, each thread t identifies which group it belongs to (Line 1), by dividing its index t for the number of rows in each output group, i.e., $rows(X_i, X_j)$. Specifically, t operates within the gth group. Furthermore, t computes its index idx relative to the first row of its group in Line 2.

Then, to compute the indices γ and δ of its two input rows, t first calculates $\#_i$ and $\#_j$, representing the number of rows of each group in T_i and T_j respectively, by multiplying the sizes of the domains of the non-shared variables in each table (Lines 3 and 4).

A further inspection of Lines 5 and 6 reveals how Algorithm 3 organises the $rows(X_i, X_j)$ elements of the gth output group among the corresponding GPU threads. It associates the first $\#_j$ rows of such group to the first row of the gth group in T_i, and each of these threads is then associated to each of the $\#_j$ rows of the gth group in T_j. This pattern is then repeated for the second row of the gth group in T_i, and so on for all the $\#_i$ rows of the gth group in T_i (Figure 8). The offsets $g \cdot \#_i$ and $g \cdot \#_j$ ensure the selection of the gth group in T_i and T_j, as they represent the total number of rows in the g groups that precede the gth one in each input table.

Example 1 *For a better understanding, we show how Algorithm 3 computes the row at index 59 of* $T_1^p \oplus T_2^p$. *Such a row would be computed by the thread* $t = 59$, *associated to the index* $idx = 23$ *of the output group* $g = 1$, *i.e., the grey one. In fact, as introduced earlier in this section,* $rows(X_1, X_2) = 36$. *It is easy to verify that* $\#_i = \#_j = 6$. *Then, t computes the indices of its input rows in* T_1^p *and* T_2^p, *i.e.,* $\gamma = 6 + 3 = 9$ *and* $\delta = 6 + 5 = 11$. *Hence,* $t = 59$ *computes the element at index 23 of the output grey group, i.e., the one associated to the line at index 3 of the grey group in* T_1^p *and the last line of the grey group in* T_2^p, *as represented by* γ *and* δ.

Note that the only input required by each thread executing Algorithm 3 is its own ID t, since X_i and X_j are equal and known in advance by all threads. t does not determine *which* operations are executed (as they are equal for all threads), but only *where* the input data is located. For these reasons, Algorithm 3 fits perfectly the SIMD model adopted by GPU architectures. In addition, Algorithm 3 does not contain any branching instruction, which would cause a phenomenon called *divergence*, which reduces the degree of parallelism by forcing the serialisation of threads executing different branches of the program [12], hence limiting its computational throughput.

Finally, Algorithm 3 relies on a *data reuse* pattern, as each row of T_i is the input of $\#_j$ output elements and, symmetrically, each row of T_j is the input of $\#_i$ output elements. We avoid expensive accesses to the GPU global memory[7] by first transferring each coloured group

[4] Even if this paradigm shares the same name with the *scattering* phase of BP, it refers to a completely different concept.

[5] A table T_i with the scope X_i is *complete* if it contains all the possible assignments over the domains of the variables in X_i. We represent *unfeasible* rows as ∞ values.

[6] We adopt the *zero-based* convention, i.e., arrays start at index 0.

[7] Global memory, in which the data is initially stored, is the slowest type of memory of the GPU hierarchy [10].

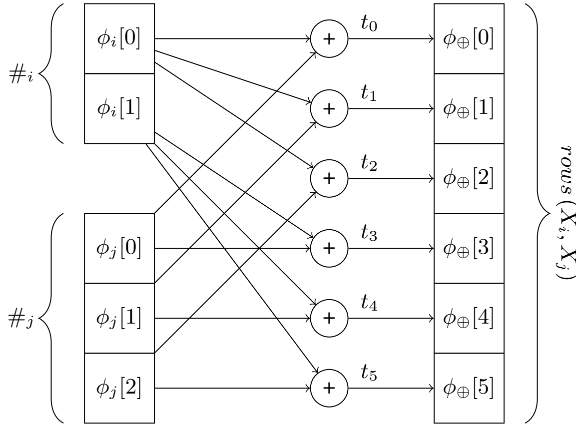

Figure 8: Join sum output computation.

to the shared memory, which allows threads to fetch data roughly $100\times$ faster [10]. Notice that the use of the shared memory is possible only because we represent the input data with the table layout discussed in Section 2.4, in which coloured groups are in small, contiguous chunks of memory. Since GPUs only have tens of KB of shared memory available, it is not possible to achieve the same benefits with the original tables (Figure 6), which should be transferred *in toto*, possibly exceeding the hardware capabilities of the GPU.

For the same reason, CUBE is capable of processing tables that are larger than the GPU global memory. In fact, our approach exploits the proposed table layout by splitting large tables into manageable chunks that can be processed independently. Specifically, 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. We also take into account the space constraints deriving from the use of shared memory (see Section 2.3), by enforcing that single coloured chunks of data can fit in such memory.

Figure 9 shows an example in which the input is processed in three different pieces by the kernels K_1, K_2 and K_3. Notice that the independence among such computations can be exploited by enabling a *pipelined* execution model, in which each kernel K_i starts processing as soon as its input chunk has been transferred to the GPU. Nonetheless, consumer NVIDIA GPUs have only one channel that can be used for data transfers, preventing the parallelisation of the transfers from the host to the device with the ones from the device to the host. However, more advanced GPUs (e.g., NVIDIA Tesla) feature an additional transfer channel, enabling a full pipeline (Figure 10).

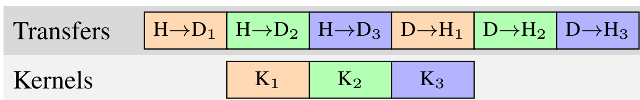

Figure 9: Pipeline with one transfer channel.

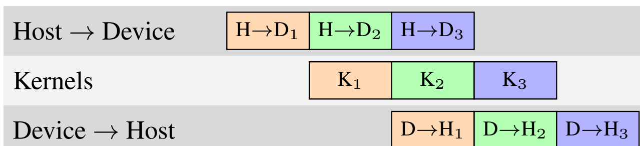

Figure 10: Pipeline with two transfer channel.

3.2 Maximisation on GPUs

Maximisation can be seen as a particular case of the relational algebra *project* operation. In the case of BE, maximisation operates by removing the variable associated to the current bucket from the input table T_i. As a consequence, the resulting table contains D_p copies of each unique assignment of the variables in its scope, i.e., X_i $\{x_p\}$. Maximisation then maps each unique assignment to the maximum of the D_p values mentioned above. For example, if we want to compute the maximisation of T_1 (Figure 6) by removing x_3, we first obtain the table shown in Figure 11, in which each unique variable assignment is highlighted with a different colour (here $D_p = D_3 = 2$). The final output is computed as shown in Figure 12. Figures 11 and 12 highlight the high degree of parallelisation inherent in the maximisation operation, as each coloured group can be processed independently from the others. The maximisation of the D_p values within each coloured group can be realised with a *reduction* operation, which can be efficiently implemented on the GPU by means of a well-known parallel algorithm [10].

x_2	x_1	ϕ^1
0	0	α_1
0	1	α_2
1	0	α_3
1	1	α_4
2	0	α_5
2	1	α_6
0	0	α_7
0	1	α_8
1	0	α_9
1	1	α_{10}
2	0	α_{11}
2	1	α_{12}

Figure 11: T_1 without x_3.

x_2	x_1	$m(\phi^1)$
0	0	$\max(\alpha_1, \alpha_7)$
0	1	$\max(\alpha_2, \alpha_8)$
1	0	$\max(\alpha_3, \alpha_9)$
1	1	$\max(\alpha_4, \alpha_{10})$
2	0	$\max(\alpha_5, \alpha_{11})$
2	1	$\max(\alpha_6, \alpha_{12})$

Figure 12: Maximisation output.

Nonetheless, Figure 11 also highlights the poor data locality of this table layout (similar to the one in Figure 6), which causes the same issues discussed in Section 2.4. To overcome these problems, we preprocess the input table to achieve the row arrangement shown in Figure 13. In particular, we aim at placing each coloured group in consecutive memory locations, so to achieve a better data locality and improve the efficiency of the maximisation operation. This is equivalent to moving x_p to the last column, and is implemented in CUBE with Bistaffa et al.'s technique by considering as *shared* all the variables in the scope of the table minus x_p.

This table layout enables an efficient GPU algorithm to compute the final output of the maximisation operation, i.e., $m(\phi_1)$ in the above example. In general, the array ϕ of the output table can be

x_2	x_1	x_3	$p(\phi^1)$
0	0	0	α_1
0	0	1	α_7
0	1	0	α_2
0	1	1	α_8
1	0	0	α_3
1	0	1	α_9
1	1	0	α_4
1	1	1	α_{10}
2	0	0	α_5
2	0	1	α_{11}
2	1	0	α_6
2	1	1	α_{12}

Figure 13: T_1 after the preprocessing.

computed with a *segmented reduction* algorithm [22], a well-known GPU primitive that differs from the standard reduction in that the latter operates on the entire set of input elements (e.g., it computes the maximum over the entire length of the input array), while the former operates on several fractions of the input data, i.e., the coloured groups in our case. The use of the segmented reduction allows an improved computational throughput w.r.t. the approach proposed by Bistaffa et al. [5], in which the authors implement the same operation by manually devising a series of small standard reductions that can lead to a low GPU utilisation when the coloured segments are small.

Finally, we further increase the efficiency of CUBE with two improvements. On the one hand, we avoid unnecessary data transfers between the host and the device when computing the maximisation operation. In particular, since the BE algorithm always applies the maximisation operation on the result of the join sum operation, we can avoid to transfer the join sum result (produced on the GPU memory) from the GPU to the CPU and directly run the maximisation on the GPU, hence saving two data transfers. On the other hand, if the tables are particularly small, we execute both the join sum and the maximisation on the CPU, since the overhead of the transfers to the GPU would hinder the benefits of parallelisation.

4 EMPIRICAL EVALUATION

The main goals of the empirical analysis are: i) to evaluate the parallel speed-up that CUBE achieves w.r.t. a sequential version of BE, ii) to compare CUBE against the most recent approach to parallelise BE on GPU, i.e., the work by Fioretto et al. [11], and iii) to evaluate the scalability of our approach w.r.t. the size of the problem.

Following Fioretto et al. [11], we considered 3 different CN topologies: i) *random networks* with a graph density of 0.3, ii) *scale-free networks* generated with the Barabási-Albert model [1] using $m = 2$, and iii) *2-dimensional square grid networks*, in which internal nodes are connected to four neighbours, while nodes on the edges (resp. corners) are connected to two (resp. three) neighbours. Each function F_i is generated using uniformly distributed random integer values in $[0, 100]$ and the constraint tightness (i.e., ratio of entries in such tables different from ∞) is set to 0.5 for all experiments. Domain size is 5 for all experiments. We compared both GPU approaches with FRODO [16], a standard sequential COP solver also adopted by Fioretto et al. as baseline benchmark. In particular, within FRODO we employ the DPOP algorithm [20].

To ensure a fair comparison, we run all the algorithms on the same instances and adopting the same variable ordering, i.e., the one produced by FRODO. We consider the entire execution time for all the algorithms, including data transfers.[8] All our experiments are run on a machine with a 3.10GHz processor, 16 GB of memory and a NVIDIA Tesla K40. CUBE is implemented in CUDA.[9] For Fioretto et al.'s approach we use the authors' implementation.

4.1 Experimental Results

Figures 14–16 show the speed-up of both GPU approaches w.r.t. FRODO when increasing the number of variables in the CN. Each data point in the plots represents the average over 20 random instances of the ratio between the runtime required by the GPU approach and FRODO's runtime.

The results show that CUBE allows a dramatic runtime reduction w.r.t. to FRODO, by computing the solution at least one order of magnitude faster than the sequential approach in every experiment. In particular, CUBE is, on average, 530× faster than FRODO when considering the biggest instances in our experiments (i.e., random networks with $n \geq 30$ and scale-free and grid networks with $n \geq 70$), by reaching a maximum speed-up of $652\times$.

More important, such speed-ups increase when the complexity of the problem grows, thus confirming the scalability of CUBE, which correctly exploits the additional degree of parallelism inherent in the problem. In contrast, the speed-up of the approach by Fioretto et al. decreases when the size of the problem increases.

Finally, the results show that the speed-up saturates after a certain number of variables (25 for random networks, 70 for scale-free networks, and 36 for grid networks). This saturation happens when the GPU reaches a full occupancy and it runs the maximum number of threads (i.e., 30720 for our GPU model). After that, the hardware forces blocks of threads to run sequentially, hence limiting the maximum speed-up.

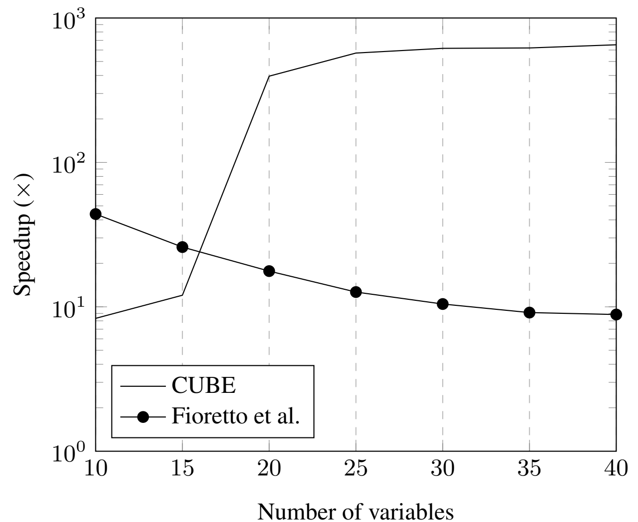

Figure 14: Speed-up on random networks.

[8] We measured that, on average, data transfers take approximately 20% of the entire CUBE runtime.

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

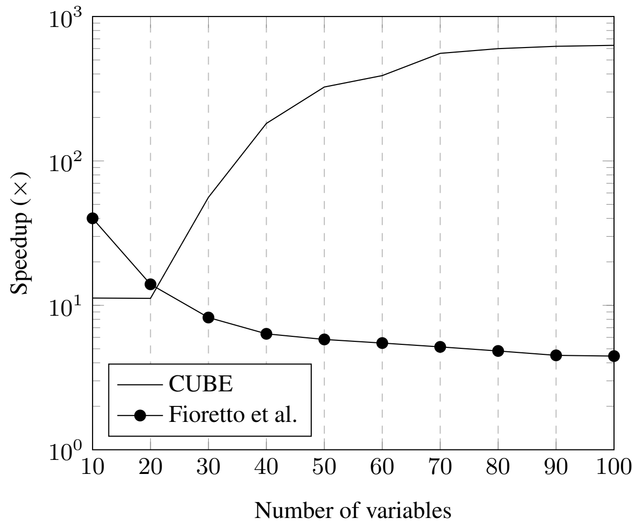

Figure 15: Speed-up on scale-free networks.

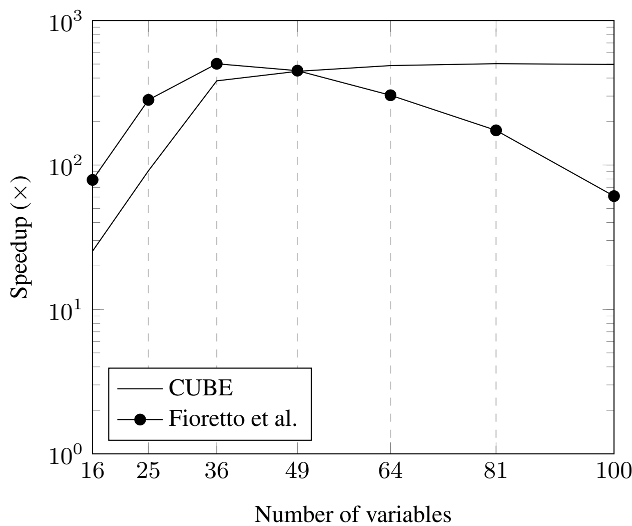

Figure 16: Speed-up on grid networks.

5 CONCLUSIONS

This paper proposes CUBE (CUda for Bucket Elimination), a high-throughput GPU implementation of the BE algorithm. Our experimental results show that CUBE outperforms the most recent GPU implementation of BE, by achieving parallel speed-ups up to two orders of magnitude higher. More important, the speed-ups achieved by CUBE increase when the complexity of the problem grows, allowing us to solve problems that could not be tackled by the sequential BE implementation in a reasonable amount of time.

Future work will aim at validating our approach on real-world COP instances, such as pedigree haplotyping problems [19] and satellite management problems [4]. We also plan to integrate our GPU techniques in other algorithmic frameworks, such as Mini-Bucket Elimination [7] (which adopts the same join sum and maximisation operations discussed in this paper), and the AND/OR search-based approaches proposed by Marinescu and Dechter [17, 18], in which Mini-Bucket heuristics are used to guide the search.

REFERENCES

[1] Réka Albert and Albert-László Barabási, ‘Statistical mechanics of complex networks’, *Reviews of modern physics*, **74**(1), 47, (2002).

[2] Dan Anthony Feliciano Alcantara, *Efficient hash tables on the GPU*, University of California at Davis, 2011.

[3] Ricardo Baeza-Yates and Patricio V. Poblete, ‘Algorithms and theory of computation handbook’, chapter Searching, CRC Press, (2010).

[4] Eric Bensana, Michel Lemaitre, and Gerard Verfaillie, ‘Earth observation satellite management’, *Constraints*, **4**(3), 293–299, (1999).

[5] Filippo Bistaffa, Alessandro Farinelli, and Nicola Bombieri, ‘Optimising memory management for belief propagation in junction trees using GPGPUs’, in *IEEE International Conference on Parallel and Distributed Systems*, pp. 526–533, (2014).

[6] Sebastián Dormido Canto, Ángel P. de Madrid, and Sebastián Dormido Bencomo, ‘Parallel dynamic programming on clusters of workstations’, *Parallel and Distributed Systems, IEEE Transactions on*, **16**(9), 785–798, (2005).

[7] Rina Dechter, ‘Mini-buckets: A general scheme for generating approximations in automated reasoning’, in *International Joint Conference on Artificial Intelligence*, pp. 1297–1303, (1997).

[8] Rina Dechter, ‘Bucket elimination: A unifying framework for reasoning’, *Artificial Intelligence*, **113**(1–2), 41–85, (1999).

[9] Rina Dechter, *Constraint processing*, Morgan Kaufmann, 2003.

[10] Rob Farber, *CUDA Application Design and Development*, Elsevier, 2012.

[11] Ferdinando Fioretto, Tiep Le, Enrico Pontelli, William Yeoh, and Tran-Cao Son, ‘Exploiting GPUs in solving (distributed) constraint optimization problems with dynamic programming’, in *Principles and Practice of Constraint Programming*, 121–139, Springer, (2015).

[12] Tianyi David Han and Tarek S Abdelrahman, ‘Reducing branch divergence in GPU programs’, in *ACM GPGPUs Workshop*, (2011).

[13] Stephen Huang, Hongfei Liu, and Venkatraman Viswanathan, ‘Parallel dynamic programming’, *Parallel and Distributed Systems, IEEE Transactions on*, **5**(3), 326–328, (1994).

[14] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to parallel computing: design and analysis of algorithms*, Benjamin/Cummings Publishing Company, 1994.

[15] Steffen L Lauritzen and David J Spiegelhalter, ‘Local computations with probabilities on graphical structures and their application to expert systems’, *Journal of the Royal Statistical Society*, 157–224, (1988).

[16] Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek, ‘FRODO 2.0: An open-source framework for distributed constraint optimization’, in *IJCAI DCR Workshop*, pp. 160–164, (2009).

[17] Radu Marinescu and Rina Dechter, ‘Dynamic orderings for AND/OR branch-and-bound search in graphical models’, *Frontiers in Artificial Intelligence and Applications*, **141**, 138, (2006).

[18] Radu Marinescu and Rina Dechter, ‘Best-first AND/OR search for graphical models’, in *AAAI Conference on Artificial Intelligence*, pp. 1171–1176, (2007).

[19] Lars Otten and Rina Dechter, ‘A case study in complexity estimation: Towards parallel branch-and-bound over graphical models’, in *Conference on Uncertainty in Artificial Intelligence*, pp. 665–674, (2012).

[20] Adrian Petcu, *A Class of Algorithms for Distributed Constraint Optimization*, Phd. thesis no. 3942, Swiss Federal Institute of Technology (EPFL), 2007.

[21] John H Reif, ‘Depth-first search is inherently sequential’, *Information Processing Letters*, **20**(5), 229–234, (1985).

[22] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens, ‘Scan primitives for GPU computing’, in *Graphics hardware*, pp. 97–106, (2007).

[23] Guangming Tan, Ninghui Sun, and Guang R Gao, ‘Improving performance of dynamic programming via parallelism and locality on multicore architectures’, *Parallel and Distributed Systems, IEEE Transactions on*, **20**(2), 261–274, (2009).

[24] Meritxell Vinyals, Juan A Rodriguez-Aguilar, and Jesús Cerquides, ‘Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law’, *Autonomous Agents and Multi-Agent Systems*, 439–464, (2011).