Optimising Memory Management for Belief Propagation in Junction Trees using GPGPUs

Filippo Bistaffa, Alessandro Farinelli, Nicola Bombieri
Department of Computer Science
University of Verona
Verona, Italy
{filippo.bistaffa,alessandro.farinelli,nicola.bombieri}@univr.it

***Abstract*—Belief Propagation (BP) in Junction Trees (JT) is one of the most popular approaches to compute posteriors in Bayesian Networks (BN). Such approach has significant computational requirements that can be addressed by using highly parallel architectures (i.e., General Purpose Graphic Processing Units) to parallelise the message update phases of BP. In this paper, we propose a novel approach to parallelise BP with GPGPUs, which focuses on optimising the memory layout of the BN tables so to achieve better performance in terms of increased speedup, reduced data transfers between the host and the GPGPU, and scalability. Our empirical comparison with the state of the art approach on standard datasets confirms significant improvements in speedups (up to +594%), and scalability (as our method can operate on networks whose potential tables exceed the global memory of the GPGPU).**

Keywords—Belief Propagation on Junction Trees, GPGPUs.

I. Introduction

Bayesian Networks represent a powerful tool to perform inference on uncertain data, and they have been used in a wide range of applications in artificial intelligence and related fields. A crucial task for BN is to compute the posterior probability given observed events (i.e., evidence). A popular approach to perform exact inference is to compile the BN in a Junction Tree and then run Belief Propagation, a message passing algorithm that updates the belief of each node in the JT propagating the acquired evidence [1]. Exact inference on BN is known to be computationally hard (i.e., it is NP-hard) and while BP over JT is one of the most popular approaches to perform exact inference, the computation associated with the message update procedures increase dramatically with the variables' domain size and the clique size of the JT (i.e., such procedure is time and space exponential in the size of the largest clique). Such computational complexity is a crucial issue hindering the use of exact inference for BN when evidence collection and belief update must be performed in real-time or when the inference task is a subroutine for other algorithms [2]. To deal with such computational complexity, researchers developed different approaches ranging from cutset conditioning [3] to iterative or loopy BP [4].

In this perspective, the recent development of highly parallel architectures such as General Purpose Graphic Processing Units hold great promises to significantly reduce the run time of computational intensive algorithms by exploiting parallel execution. The use of parallel architectures for BP has been investigated before by [5], who focused on machines with distributed memory, while the use of pointer jumping has been introduced by [6]. On the other hand, the use of GPGPUs in basic sum-product algorithms has been investigated by [7], later developed by [8] exploiting node level parallelism.

In a recent work, [9] proposes an approach to parallelise the BP message update phases by using GPGPUs. Here, we also study the parallelisation of BP over JT using GPGPUs, but, in contrast to such previous work, we focus on optimising the memory layout of the BN tables so to achieve better performance in terms of increased speedup, reduced data transfers between the host and the GPGPU, and scalability. Specifically, we propose a novel approach to parallelise BP on GPGPU which is based on preprocessing potential tables so to obtain full memory coalescence, which is a crucial component for efficient memory management in GPGPUs.

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

- It proposes an algorithm to preprocess potential tables by organising the columns in a specified order (i.e., placing the shared variables in the most significant positions), thus achieving full memory coalesced accesses in the message update phases. We formally analyse such algorithm proving its correctness and giving the worst case computational complexity.
- It proposes an implementation of the GPGPU kernel that exploits the organisation of the tables specified before. Specifically, we show that such an arrangement enables pipelined data transfers from the host to the GPGPU (hence optimising transfer time) and it allows the use of highly efficient routines for crucial parts of the BP message update algorithm (reduction and scattering).
- It empirically compares the proposed approach to [9] on the same dataset. Our results show significant improvements, achieving speedups at least 59% higher than the alternative method and reaching peaks of +594% in this gain. Moreover, our method can scale up to networks whose potential tables do not fit in the global memory of the GPGPU.

The rest of the paper is organised as follows: Section II illustrates the background on BP on JT and outlines some previous approaches to this problem. Section III provides a method to obtain an appropriate representation of potential tables in memory, so as to achieve an high GPGPU computational throughput through optimal memory accesses. Section IV discusses our empirical evaluation and Section V concludes the paper.

978-1-4799-7615-7/14/$31.00 ©2014 IEEE

II. BACKGROUND

The purpose of this section is threefold. First, in Section II-A we define the theoretical concepts and the algorithms related to BP. Second, Section II-B outlines the features of GPGPU architectures, used to implement our highly-parallel approach to BP. Finally, Section II-C discusses previous works for BP on GPGPUs, focusing on the recent work by [9], used as a reference in our empirical evaluation.

A. Belief Propagation in Junction Trees

A BN is a compact representation of a joint distribution over a set of random variables $\mathcal{Z}$, 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 derived graph J called *junction tree*, generated from a BN by means of moralisation and triangulation [1]. Every vertex of the junction tree contains a subset of the random variables that forms a maximal clique in the moralised and triangulated BN, each associated to a potential table represented by $T = \langle \mathcal{X}, d, \mathcal{R}, \phi \rangle$ such that:

- $\mathcal{X} \subseteq \mathcal{Z}$ is an ordered tuple defining the variables of the clique associated to T.
- d is an ordered tuple of natural numbers such that each d_i is the size of the domain of variable $\mathcal{X}_i$.[1]
- $\mathcal{R}$ is an ordered tuple of *rows*: in particular, each row $r = \mathcal{R}_i$ is, in turn, an ordered tuple of natural numbers (such that $\forall r_i \in r, 1 \leq r_i \leq d_i$), defining a particular assignment of the variables in $\mathcal{X}$.
- ϕ is an ordered tuple representing the actual potential values of the table, one for each row $\mathcal{R}_i$: in particular, ϕ_i is the potential value associated to the variable assignment represented by $\mathcal{R}_i$.

Notice that, in standard BP, $\mathcal{R}$ contains all the possible variable assignments over their domains, hence $|\mathcal{R}| = |\phi| = \prod_{i=1}^{|\mathcal{X}|} d_i$. Assuming that T_i and T_j are potential tables associated to adjacent cliques in the junction tree, we associate a separator S_{ij} to the edge connecting T_i and T_j, which indicates the shared variables between the two tables, i.e., $\mathcal{S} = \mathcal{X} \cap \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ denotes the tuples of variables of T_i and T_j respectively. BP is then invoked whenever we receive new evidence for a particular set of variables $\mathcal{E} \subseteq \mathcal{Z}$, requiring to update 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 C_i, starting from the leaves all the way up to a designated root vertex (Algorithm 1).

Algorithm 1 COLLECT (J, C_i)

```
1: for all C_j child of C_i do
2:     MESSAGEPASS (C_i, COLLECT (J, C_j))
3: return C_i
```

Then, during *evidence distribution*, messages are distributed from the root R to the leaves (Algorithm 2).

[1]If a is an ordered tuple, a_i refers to its ith element.

Algorithm 2 DISTRIBUTE (J, R)

```
1: for all C_j child of R do
2:     MESSAGEPASS (R, C_j)
3:     DISTRIBUTE (J, C_j)
4: return C_i
```

In both phases, each recursive call comprises a *message passing* procedure, which implements the actual propagation of beliefs between the potential tables T_i and T_j associated to cliques C_i and C_j. More specifically, such operation involves two steps:

1) **Reduction**: the potential table S_{ij} is updated to S^*_{ij}. In particular, each row of S^*_{ij} is obtained summing the corresponding rows of T_i, i.e., the ones with a matching variable assignment.
2) **Scattering**: T_j is updated with the new values of S^*_{ij}, i.e., every row of T_j is multiplied for the corresponding $R_{ij} = S^*_{ij}/S_{ij}$. Following [9], we assume that $0/0 = 0$.

It is easy to see that both these steps require several independent computations spanning over multiple rows of the considered tables, suggesting a multi-threaded algorithm in which such degree of parallelism can be exploited by means of GPGPUs, i.e., general purpose graphics processing units.

B. GPGPU Architecture

GPGPUs 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 GPGPU (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 GPGPU 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 organised in thread *blocks*, sharing fast forms of storage and synchronisation 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 share limited memory resources. Memory management is a crucial aspect in the design of efficient GPGPU 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 III-B1). In what follows, we provide some background on previous approaches to BP on GPGPUs that can be found in literature.

C. Related Work

The most recent solution technique for BP on GPGPUs is presented by [9], in which the authors propose a way to parallelise the atomic operations of propagation, so that it could be embedded in different algorithms. The authors

devise a *two-dimensional 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 GPGPUs. In particular, element-wise parallelism is achieved by computing each of the $|S|$ reduction-and-scattering operations in parallel, which require $|S|$ *mapping tables* (one per row of S) to allow each concurrent task to correctly locate its input data from the corresponding potential tables. On the other hand, arithmetic parallelism represents the multi-threaded computation of each reduction-and-scattering operation, by means of well known parallel algorithms that can be found in literature [11].

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 suitable for a GPGPU programming model, for two main reasons:

- GPGPU threads need to access data in sparse and discontinuous memory locations by means of an additional indexing table, breaking coalescence and drastically reducing the throughput of memory transfers (Figure 4). Coalescence is a crucial feature that should be exploited in order to reduce memory accesses to the global memory, hence improving the compute-to-memory ratio and achieving a greater computational throughput.
- Since input data is organised in a discontinuous pattern rather than in continuous chunks, it is mandatory to transfer the entire potential tables to the global memory of the GPGPU before starting the computation of the BP algorithm, hindering two desirable properties: i) 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 ii) since the computation cannot be started before the entire input data has been copied to the GPGPU, the cost of memory transfers cannot be amortised by means of technologies like NVIDIA CUDA streams.

To overcome these limitations, we propose a better way to organise potential tables in memory, hence devising a better approach to BP on GPGPUs.

III. GPGPU Message Computation

In this section, we describe our contribution to the computation of BP on GPGPUs. In particular, we first introduce an appropriate representation of potential tables in memory, detailing how we can preprocess original tables to maximise the performance through optimal data transfers and memory accesses. Such representation is then exploited by the actual BP algorithm computed by means of a CUDA kernel.

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 $\mathcal{X} = \langle x_3, x_2, x_1 \rangle$ and $\mathcal{Y} = \langle x_5, x_4, x_1 \rangle$, with the separator $\mathcal{S} = \mathcal{X} \cap \mathcal{Y} = \langle x_1 \rangle$. We assume that x_1, x_3 and x_5 are binary variables, while x_2 and x_4 can assume 3 values. In the approach proposed by [9], each row of the separator table S_{12} is assigned to a different block of threads, which are responsible of the reduction of the rows of T_1 with a matching variable assignment and the subsequent scattering on matching rows in T_2. In Figure 1, rows associated to different blocks of threads have been marked in different colours, i.e., white and grey for $x_1 = 0$ and $x_1 = 1$ respectively. The organisation of input data provided by these tables is undesirable for GPGPU architectures. In fact, threads responsible of the computation of white rows cannot access consecutive memory addresses, as their data is interleaved with grey rows, 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 S_{12} (i.e., rows of the same colour, in the above example) are stored in *consecutive* addresses in the corresponding potential tables, as shown in Figure 2. Threads responsible of white rows execute coalesced memory accesses, and start the computation while grey rows are still being transferred to the GPGPU. Furthermore, each block of threads easily retrieves its input data without the need of any costly mapping table, unlike [9].

Consider $T_1^p = \langle \mathcal{X}^p, d^p, \mathcal{R}^p, \phi^p \rangle$ in Figure 2, resulting from a particular permutation σ of $\mathcal{X}$, in which the variables in $\mathcal{S}$ are brought to the *most significant*[2] positions in $\mathcal{X}^p = \sigma(\mathcal{X})$. In this way, we can assure that rows with the same assignment of the variables in $\mathcal{S}$ form a contiguous chunk of memory.

T_1

x_3	x_2	x_1	ϕ
0	0	0	a_1
0	0	1	a_2
0	1	0	a_3
0	1	1	a_4
0	2	0	a_5
0	2	1	a_6
1	0	0	a_7
1	0	1	a_8
1	1	0	a_9
1	1	1	a_{10}
1	2	0	a_{11}
1	2	1	a_{12}

S_{12}

x_1	ϕ
0	b_1
1	b_2

T_2

x_5	x_4	x_1	ϕ
0	0	0	c_1
0	0	1	c_2
0	1	0	c_3
0	1	1	c_4
0	2	0	c_5
0	2	1	c_6
1	0	0	c_7
1	0	1	c_8
1	1	0	c_9
1	1	1	c_{10}
1	2	0	c_{11}
1	2	1	c_{12}

Fig. 1: Original Tables

T_1^p

x_1	x_3	x_2	ϕ^p
0	0	0	a_1
0	0	1	a_3
0	0	2	a_5
0	1	0	a_7
0	1	1	a_9
0	1	2	a_{11}
1	0	0	a_2
1	0	1	a_4
1	0	2	a_6
1	1	0	a_8
1	1	1	a_{10}
1	1	2	a_{12}

S_{12}^p

x_1	ϕ^p
0	b_1
1	b_2

T_2^p

x_1	x_5	x_4	ϕ^p
0	0	0	c_1
0	0	1	c_3
0	0	2	c_5
0	1	0	c_7
0	1	1	c_9
0	1	2	c_{11}
1	0	0	c_2
1	0	1	c_4
1	0	2	c_6
1	1	0	c_8
1	1	1	c_{10}
1	1	2	c_{12}

Fig. 2: Preprocessed Tables

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

[2] Variables are listed from the most significant to the least significant.

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

$$k = \sum_{i=1}^{|\mathcal{X}|-1} \Bigg(r_i \underbrace{\prod_{j=i+1}^{|\mathcal{X}|} d_j}_{D_i} \Bigg) + r_{|\mathcal{X}|} = \sum_{i=1}^{|\mathcal{X}|-1} (r_i \cdot D_i) + r_{|\mathcal{X}|} \tag{1}$$

where r_i represents the value assumed by the variable $\mathcal{X}_i$ in r. Notice that the tuple D is the *exclusive postfix product* of d, hence we define $D_{|\mathcal{X}|} \coloneqq 1$. On the other hand, each r_i can be retrieved from k as $r_i = \lfloor k/D_i \rfloor \bmod d_i$. As a consequence, $\mathcal{R}$ can be dropped from our representation of potential tables in memory, hence, as previously claimed, the computation of $\mathcal{R}^p$ is unnecessary. For a better understanding, consider the row r with $\mathcal{X} = \langle x_1, x_2, x_3 \rangle$, and $d = \langle 2, 16, 10 \rangle$:

$r =$

x_1	x_2	x_3	ϕ
1	10	7	v_{267}

From Equation 1, r will be in position $k = d_2 \cdot d_3 + 10 \cdot d_3 + 7 = 267$ in ϕ. Moreover, it is easy to verify that $x_1 = 1 = \lfloor 267/D_1 \rfloor \bmod d_1$, $x_2 = 10 = \lfloor 267/D_2 \rfloor \bmod d_2$ and $x_3 = 7 = \lfloor 267/D_3 \rfloor \bmod d_3$.

As mentioned before, to maintain a coherent representation of the preprocessed table $T^p = \langle \mathcal{X}^p, d^p, \mathcal{R}^p, \phi^p \rangle$, the values in ϕ must be correctly permuted into ϕ^p, as detailed in the following section.

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 $\mathcal{R}$, since each row $r \in \mathcal{R}$ can be retrieved from its index with the above detailed technique, hence the computation of $\mathcal{R}^p$ will not be covered. On the other hand, for any ϕ_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 $r = \mathcal{R}_k$, which comprises 3 steps: for each k, compute the corresponding variable assignment $\langle r_1, \dots, r_i, \dots, r_{|\mathcal{X}|} \rangle$, apply σ on the now available sequence of r_i and, finally, obtain k^p using Equation 1. Since each of the 3 above mentioned steps has a complexity of $O(|\mathcal{X}|)$, such approach requires $O(3|\phi||\mathcal{X}|)$ to reorder the entire table.

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 $T = \langle \mathcal{X}, d, \mathcal{R}, \phi \rangle$ and $T^s = \langle \mathcal{X}^s, d^s, \mathcal{R}^s, \phi^s \rangle$, where $\mathcal{X}^s$ and d^s has been respectively obtained swapping $\mathcal{X}_i$ with $\mathcal{X}_j$ 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' equal to:

$$\begin{aligned}
k' = {} & r_1 \cdot d_2 \cdots d_i \cdots d_j \cdots d_{|\mathcal{X}|} + \cdots && (2a) \\
& + r_i \cdot d_{j+1} \cdots d_j \cdots d_{|\mathcal{X}|} && (2b) \\
& + r_{j+1} \cdot d_{j+2} \cdots d_j \cdots d_{|\mathcal{X}|} + \cdots + r_{i-1} \cdot d_j \cdots d_{|\mathcal{X}|} && (2c) \\
& + r_j \cdot d_{i+1} \cdots d_{|\mathcal{X}|} && (2d) \\
& + r_{i+1} \cdot d_{i+2} \cdots d_{|\mathcal{X}|} + \cdots + r_{|\mathcal{X}|} && (2e)
\end{aligned}$$

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

$$\begin{aligned}
k' = {} & \overbrace{k - k \bmod D_{j-1}}^{(2a)} + \overbrace{D_j \cdot {}^{d_j}/_{d_i} \cdot \lfloor k/D_i \rfloor \bmod d_i}^{(2b)} + \overbrace{k \bmod D_i}^{(2e)} \\
& \underbrace{+ {}^{d_j}/_{d_i} \cdot \overbrace{(k \bmod D_j - k \bmod D_{i-1})}^{(2c')}}_{(2c)} + \underbrace{D_i \cdot \lfloor k/D_j \rfloor \bmod d_j}_{(2d)}
\end{aligned}$$

Proof: In the following proof, we use the following properties, which can be demonstrated by means of basic algebraic procedures:

$$k = \sum_{h=1}^{|\mathcal{X}|} r_h \cdot D_h \implies \sum_{h=l}^{|\mathcal{X}|} r_h \cdot D_h = k \bmod D_{l-1} \tag{3}$$

$$k = \sum_{h=1}^{|\mathcal{X}|} r_h \cdot D_h \implies r_h = \lfloor k/D_h \rfloor \bmod d_h \tag{4}$$

From Equation 1 we can easily verify that $k' = (2a) + (2b) + (2c) + (2d) + (2e)$. Similarly, k can be written as:

$$\begin{aligned}
k = {} & \overbrace{r_1 \cdot D_1 + \cdots + r_{j-1} \cdot D_{j-1}}^{(2a)} + r_j \cdot D_j + \overbrace{r_{i+1} \cdot D_{i+1} + \cdots + r_{|\mathcal{X}|}}^{(2e)} \\
& + \overbrace{r_{j+1} \cdot D_{j+1} + \cdots + r_{i-1} \cdot D_{i-1}}^{(2c')} + r_i \cdot D_i
\end{aligned}$$

To prove the correctness of Proposition 1, we must show that:

- $(2a) = k - k \bmod D_{j-1}$
- $(2b) = D_j \cdot {}^{d_j}/_{d_i} \cdot \lfloor k/D_i \rfloor \bmod d_i$
- $(2c) = {}^{d_j}/_{d_i} \cdot (k \bmod D_j - k \bmod D_{i-1})$
- $(2d) = D_i \cdot \lfloor k/D_j \rfloor \bmod d_j$
- $(2e) = k \bmod D_i$.

Now, it is easy to see that $(2a)$, $(2d)$ and $(2e)$ are not affected by the swap of $\mathcal{X}_i$ and $\mathcal{X}_j$: in fact, since $(2a)$ refers to all the variables before $\mathcal{X}_j$ (and, consequently, before $\mathcal{X}_i$), all the terms $D_1 \cdots D_{j-1}$ contain both d_i and d_j, hence the swap does not produce any effect. On the other hand, $(2e)$ contains neither d_i nor d_j, since it refers to all the variables after $\mathcal{X}_i$, thus $(2e)$ is not affected either. Using Property 3, we can calculate $(2a)$ as the difference between k and all the terms from $r_j \cdot D_j$ on, i.e., $(2a) = k - k \bmod D_{j-1}$, while $(2e)$ is given by $k \bmod D_i$. Finally, in $(2d)$ none of the terms $d_{i+1} \cdots d_{|\mathcal{X}|} = D_i$ contains either d_i or d_j, thus $(2d) = r_j \cdot D_i = D_i \cdot \lfloor k/D_j \rfloor \bmod d_j$.

On the contrary, $(2b)$ and $(2c)$ are affected by the swap of $\mathcal{X}_i$ and $\mathcal{X}_j$: to calculate them, we first have to calculate $(2c')$ as the difference between all the terms from $r_{j+1} \cdot D_{j+1}$ on and all the terms from $r_i \cdot D_i$ on, i.e., $(2c') = k \bmod D_j - k \bmod D_{i-1}$. Using Property 4, we can also compute $r_i = \lfloor k/D_i \rfloor \bmod d_i$ and $r_j = \lfloor k/D_j \rfloor \bmod d_j$. Finally, d_i needs to be substituted with d_j in all the terms $D_j \cdots D_{i-1}$ in $(2b)$ and $(2c)$, since $\mathcal{X}_j$ has been moved to a less significant position, taking the place of $\mathcal{X}_i$ which has been moved to a more significant one. Thus, we multiply $(2c')$ by ${}^{d_j}/_{d_i}$ to compensate for this effect and obtain $(2c)$. Equivalently, we calculate $(2b) = D_j \cdot {}^{d_j}/_{d_i} \cdot \lfloor k/D_i \rfloor \bmod d_i$ by swapping d_i with d_j in D_j. ■

The result provided in Proposition 1 is used to reorder any potential table T according to the layout detailed in Section III-A. More formally, let $\mathcal{P} = \langle \mathcal{P}_1, \dots, \mathcal{P}_n \rangle$ be a sequence of n swaps, each represented by an ordered[3] pair of positions $\mathcal{P}_i = \langle a_i, b_i \rangle$, so that we permute $\mathcal{X}$ into $\sigma(\mathcal{X})$ (moving the desired subset of variables to the most significant 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 the following algorithm:

Algorithm 3 ReorderTable($\phi, \mathcal{P}$)

```
1: for all k ∈ {1, ···, |φ|} do
2:     k^p ← k
3:     for all ⟨a_i, b_i⟩ ∈ P do                    ▷ For every swap in P
4:         k^p ← f(k^p, a_i, b_i)                    ▷ Proposition 1
5:         Swap(X_{a_i}, X_{b_i})                    ▷ Swap variables
6:         Swap(d_{a_i}, d_{b_i})                    ▷ Swap variable domains
7:     φ^p_{k^p} ← φ_k                               ▷ Write φ_k in position k^p of φ^p
8: return φ^p                                        ▷ Return φ^p
```

Notice that the sequence of swaps $\mathcal{P}$ required to move $|\mathcal{S}|$ variables to the most significant positions of the tables T_i and T_j can be computed as follows. Consider T_i and let us assume that s shared variables (with $0 \le s \le |\mathcal{S}|$) are already within the first $|\mathcal{S}|$ positions of the corresponding variable tuple $\mathcal{X}$. Then, it is sufficient to swap the $|\mathcal{S}| - s$ shared variables with index greater than $|\mathcal{S}|$ with the non-shared ones which are placed within the first $|\mathcal{S}|$ positions. On the other hand, table T_j can be preprocessed by swapping each shared variable $\mathcal{Y}_h$ with $\mathcal{Y}_k$ such that $\mathcal{Y}_h = \mathcal{X}_k$ for $k \in \{1, \dots, |\mathcal{S}|\}$. Notice that this algorithm ensures the same order of the shared variables in both tables.

Let us show the above procedure to reorder an example row of T_1 in Figure 1 and compute its index k^p in T_1^p. In this case, the desired order is obtained with $\mathcal{P}_1 = \langle 3, 1 \rangle$ and $\mathcal{P}_2 = \langle 3, 2 \rangle$[4], i.e, swap $\mathcal{X}_3 = x_1$ with $\mathcal{X}_1 = x_3$, then swap $\mathcal{X}_3 = x_3$ with $\mathcal{X}_2 = x_2$. Initially, $d_3 = d_1 = 2$, $D_3 = 1$ and $D_1 = 6$. Then, applying Proposition 1 to the row $\langle 1, 2, 0 \rangle$ with index $k = 11$ results in $(2a) = (2e) = 0$ (since there are no variables before x_1 and after x_3), $(2b) = 6 \cdot {}^2\!/_2 \cdot 0 = 0$, $(2c) = {}^2\!/_2\,(10 \bmod 6 - 10 \bmod 2) = 4$ and $(2d) = 1 \cdot 1 = 1$, hence $f(10, 1, 3) = 5$, meaning that a_{11} would have index 5 after $\mathcal{P}_1$. To calculate its final position, we apply $\mathcal{P}_2 = \langle 3, 2 \rangle$. At this point $D_3 = 1$, $D_2 = d_3 = 2$ and $d_2 = 3$, hence $(2c) = (2e) = 0$ (since there are no variables before x_3 and between x_3 and x_2). 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 $\phi_5^p = a_{11}$, as can be verified in T_1^p.

Algorithm 3 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 most significant positions, according to Section III-A. In the next section, the impact of this preprocessing phase on the overall performance of the algorithm will be analysed in detail, by showing how it is more efficient than the naive approach previously mentioned.

3) Computational Complexity:

Proposition 2: Algorithm 3 has a time complexity of $O(|\phi||\mathcal{P}|) \le O(|\phi|^{|\mathcal{S}|}\!/_2) < O(3|\phi||\mathcal{X}|)$.

Proof: The external loop (line 1) requires $|\phi|$ iterations, while the inner loop (line 3) requires $|\mathcal{P}|$ iterations, which is equal to ${}^{|\mathcal{S}|}\!/_2$ assuming the worst case of reordering all the variables in $\mathcal{S}$. Since lines 4-6 can be computed in $O(1)$, the resulting time complexity of Algorithm 3 is $O(|\phi||\mathcal{P}|) \le O(|\phi|^{|\mathcal{S}|}\!/_2)$. ■

In our experimental evaluation, we performed the variable ordering with an average of $|\mathcal{P}| = 3$ swaps, resulting in an improvement of an order of magnitude w.r.t. 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 belief propagation by storing and updating the couple of corresponding reordered tables for each separator. Even if Algorithm 3 does not reorder ϕ in-place, the additional space required to store ϕ^p is amortised by discarding the original table, since it is not needed in any subsequent phase of the algorithm. Furthermore, each iteration of the external loop of Algorithm 3 is independent and can be computed in parallel: hence, the worst-case time complexity of the parallel version of Algorithm 3 is $O({}^{|\phi||\mathcal{P}|}\!/_t)$, where t is the number of threads.

With respect to memory requirements, given a junction tree $J = (V, E)$, our algorithm needs to store a couple of potential tables for each separator, but since threads can index input rows on-the-fly, mapping tables can be avoided: thus, our memory requirements are $O(2 \cdot |E|)$. On the other hand, the approach proposed by [9] maintains one potential table for each clique, but it needs two mapping tables for each separator table: hence, the counterpart algorithm requires $O(V + 2 \cdot |E|)$ tables.

B. GPGPU Kernel Implementation

In our approach to belief propagation on GPGPUs, 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 organisation of the computation allows us to carry out the entire reduction and scattering stages within a single thread block, hence avoiding any costly inter-block synchronisation structure. On one hand, the performance of our algorithm clearly benefits from the lack of interdependence among different blocks, which would reduce the overall computation parallelism. On the other hand, since the size of thread blocks has an intrinsic limit imposed by the hardware architecture (i.e., 2048 threads in Kepler[5] GPGPUs), the proposed organisation may serialise 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 parallelisation.

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

[4]Notice that swapping x_3 and x_2 is not mandatory since neither of them belongs to the separator, but it has been included in our example to better explain the algorithm.

[5]White paper available at http://bit.ly/NtZYOi.

[6]Such quantity is the average, over all networks, of the ratio between the average potential table size and the average separator table size, reported in Table I.

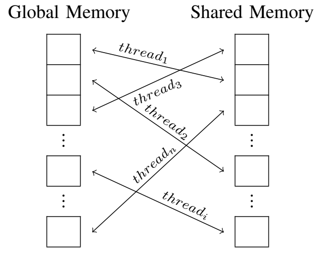

Fig. 4: Uncoalesced Memory Accesses

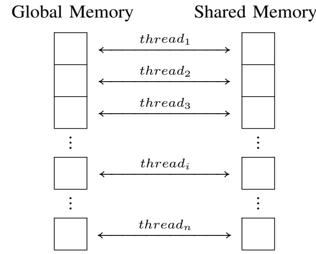

Fig. 5: Coalesced Memory Accesses

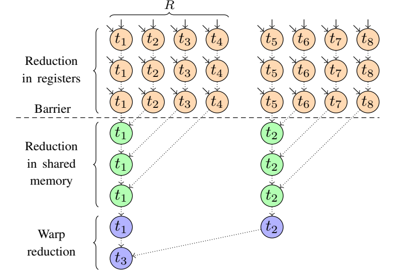

Fig. 6: Block Reduce Raking Algorithm

Moreover, if the serialisation is on a small scale (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. The following sections explain the actual implementation of the above mentioned concepts in detail, by covering our approach to memory transfers and computations.

1) Global ⇆ Shared Memory Transfers: Memory hierarchy in GPGPUs follows a widely adopted design in modern hardware architectures, in which very fast but small-size memories (i.e., registers, cache and *shared memory*), intended 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 reduced as much as possible 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 Kilobytes of information) and to complete all the computational tasks using only such data. This allows to minimise global memory accesses for each kernel execution. *Coalesced accesses* are the optimal way to carry out such data transfers, which is closely related to the principle of spatial locality of information.

More precisely, 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 organisation may result in uncoalesced loads (Figure 4), serialising memory accesses and reducing the performance, while consecutive and properly aligned data chunks enable full memory coalescing (Figure 5).

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 organised in consecutive addresses. The transfers are further optimised using *vectorised*[7] memory accesses provided by CUDA architectures to increase bandwidth, reduce instruction count and improve latency.

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

2) Reduction: Once the input data has been transferred to the shared memory, the kernel starts the reduction phase that, in our approach, is implemented with the NVIDIA CUB library[8] by means of a *block reduce raking algorithm*. The algorithm consists of three steps: i) 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, ii) 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 iii) a final reduction within the raking warp based on the Kogge-Stone algorithm [12] produces the final output.

Figure 6 shows an example reduction of 32 input values performed by a block of 8 threads $\{t_1, \ldots, t_8\}$, in which each thread operation is pictured as a circle and each input value is represented with a solid arrow, while the dotted ones represent partial results propagated among threads.

In our implementation, the definition of parameter R is managed by the aforementioned library, in order to achieve a good balance between parallelisation degree and communication among threads. This scheme is particularly efficient, since it involves a single synchronisation barrier after the first phase and it incurs zero bank conflicts[9] for primitive data types. On newer CUDA architectures (i.e., CUDA 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 belief propagation.

3) Scattering: The final stage of belief propagation 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

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

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

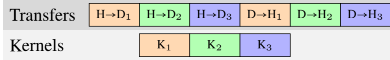

Fig. 7: Asynchronous Data Transfers

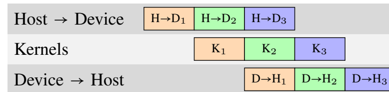

Fig. 8: Full Pipeline

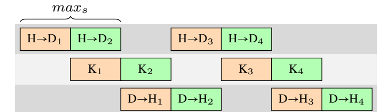

Fig. 9: Limited Number of Streams

proposed memory layout, since it is realised 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.

Since memory transactions can be very costly and have a negative impact on the overall performance of the algorithm, a significant optimisation effort should be devoted to reduce such effect. In this work, we exploit pipelining between GPGPU computation and data transfers between host and device, as described in the following sections.

C. Host ⇆ Device Data Transfers

The memory layout presented in Section III-A allows potential tables to be split into several data segments and threads to be identified to independently operate in each segment. This leads to a twofold improvement: on one hand, we can devise a pipelined flow of smaller copy-and-compute operations, by amortising the cost of CPU-GPGPU data transfers on the overall algorithm performance. On the other hand, we can process tables that do not fit into global memory, by breaking them into more manageable data structures. This allows our approach to perform BP even on networks that are intractable for the approaches in literature.

1) Pipelining: The standard pattern of GPGPU computation requires the whole input dataset to be transferred to the device 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.

Figure 7 shows the proposed *pipelined* model of computation, in which belief propagation on GPGPU has been split into four stages (marked by different colours). Each computation kernel K_i executes as soon as the corresponding input data subset has been transferred by means of $\mathrm{H{\rightarrow}D}_i$. This solution applies to GPGPU 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 serialised, thus allowing overlapping between one kernel execution and one data transfer only. In our experimental results, we found that, in average, this approach achieves a performance improvement of 50% w.r.t. synchronous data transfers. Most recent and advanced GPGPUs (e.g., NVIDIA Kepler on Tesla devices) 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 Figure 8). Such a fully pipelined model of computation achieves, in average, a performance improvement of 75% w.r.t. synchronous data transfers.

2) Large Tables Processing: The proposed technique can be applied to execute BP on BN whose potential tables do not fit into global memory. Tables are split into small data structures, by limiting the maximum number of host-device data transfers that can run concurrently on the GPGPU. More specifically, max_s is defined as the maximum number of kernels whose total amount of input and output data can be stored into global memory. Figure 9 shows an example, in which max_s = 2. Each kernel K_i is enqueued in stream $i \bmod max_s$. Transaction $\mathrm{H{\rightarrow}D}_3$ cannot be scheduled in parallel with $\mathrm{D{\rightarrow}H}_2$ (unlike the example of Figure 8), as it would violate the above mentioned memory constraint. Thus, one time slot is skipped in order to complete the copy $\mathrm{D{\rightarrow}H}_1$ and to free an adequate amount of memory before starting $\mathrm{H{\rightarrow}D}_3$. The serialisation of these two operations is a direct consequence of their execution in the same stream (i.e., stream 1). As a consequence, even though the hardware constraints limit the size of data to be transferred and processed, the proposed approach allows oversized tables to be processed in multiple steps, improving scalability.

IV. Experimental Results

Having described and analysed our GPGPU approach to belief propagation, we now present the empirical evaluation. In what follows, we first discuss the methodology we use for comparison and then present the results obtained on the same dataset used in [9].

A. Evaluation Methodology

The main goal of our empirical evaluation is to compare our algorithm with the one proposed by [9]. Our approach has been tested on the same Bayesian Networks,[10] which are related to various scenarios, with heterogeneous structures and variable domains.

Table I, taken from [9], details some features of the networks by showing the number of junction tree nodes resulting from their compilation and the minimum, maximum and average size of the potential and separator tables. In addition, we also consider the Munin_1 dataset. Table II reports the time required to preprocess all potential tables, as well as the sum of all data transfers and the runtime needed by all kernels to complete. The values are expressed in milliseconds. For a significant comparison, the claimed speedups do not take into account the time required for data transfers, but it consider only the runtime of kernel routines. Following [9], the compilation of these networks into the corresponding junction trees has been done offline, before the execution of the belief propagation algorithm. Since this phase must be done only once and can be avoided when any new evidence is received and propagated, it has not been considered. For the same reason, our preprocessing phase is not considered in the speedups, but it has been reported for completeness.

[10] All data is available at http://bndg.cs.aau.dk/html/bayesian_networks.html.

TABLE I: Bayesian Networks

	Mildew	**Diabetes**	**Barley**	**Munin$_1$**	**Munin$_2$**	**Munin$_3$**	**Munin$_4$**	**Water**
# of JT nodes	28	337	36	162	860	904	872	20
Max CPT size	4372480	190080	7257600	38400000	504000	156800	784000	995328
Min CPT size	336	495	216	4	4	4	4	9
Avg CPT size	341651	32443	512044	516887	5653	3443	16444	173297
Max SPT size	71680	11880	907200	2400000	72000	22400	112000	147456
Min SPT size	72	16	7	2	2	2	2	3
Avg SPT size	9273	1845	39318	44058	713	553	2099	26065

TABLE II: Experimental Results

	Mildew	**Diabetes**	**Barley**	**Munin$_1$**	**Munin$_2$**	**Munin$_3$**	**Munin$_4$**	**Water**
Preprocessing	40	112	444	4518	50	99	268	10
Transfers	12	57	110	1521	16	39	45	15
CPU Runtime	355	420	974	7490	210	137	473	120
GPU Runtime	3	12	29	216	8	14	29	11
GPU Speedup	118.33×	35.00×	33.59×	34.67×	26.25×	9.79×	16.31×	10.91×
SVR Runtime	17.84	33.98	45.96	-	43.73	35.18	67.14	9.47
SVR Speedup	19.90×	12.36×	21.19×	-	4.80×	3.89×	7.04×	12.67×

All our experiments have been executed on a machine with a 3.40GHz processor, 16 GB of memory and a NVIDIA GTX 780 (with one copy engine). Our results are compared to those obtained by applying the best approach (i.e., the SVR regression model) published in [9].

B. Experimental Results

In our tests, our algorithm outperforms the counterpart in the majority of the above mentioned scenarios, while achieving comparable performance in the Water network.

Our approach achieves speedups at least 59% higher than the alternative method in the Barley dataset, reaching peaks of $+594\%$ in this improvement. In fact, the GPGPU belief propagation on the Mildew network runs $118.33\times$ faster than the serial implementation. Since our approach supports pipelined memory transactions, the computation time is amortised and carried out while such data transfers are still being completed, as Figure 7 shows. Notice that our algorithm allows a further improvement (Figure 8), to be achieved employing a professional GPGPU with two copy engines.

V. Conclusions

In this paper we considered belief propagation in Bayesian Networks. We proposed an efficient and scalable highly-parallel approach that is able to harness the computational power of modern GPGPUs by means of an appropriate data organisation in memory. The proposed approach applies also to Bayesian Networks whose potential tables do not fit into the global memory of the GPGPU, and it enables pipelined data transfers between host and device, thus further improving performance. The experimental results show that our approach outperforms the one proposed by [9], which is the most recent work in this field, by obtaining speedups ranging from 59% to $+594\%$.

Future work will look at extending our approach to work on other hard optimisation problems, such as Constraint Satisfaction and Constraint Optimisation Problems [13], where similar message-passing techniques can be used.

References

[1] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," in *Readings in Uncertain Reasoning*, 1990, pp. 415–448.

[2] S. L. Lauritzen, "The em algorithm for graphical association models with missing data," in *CSDA*, 1995, pp. 191–201.

[3] B. Bidyuk and R. Dechter, "Cutset sampling for bayesian networks," in *JAIR*, 2007, pp. 1–48.

[4] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, 1988.

[5] A. V. Kozlov and J. P. Singh, "A parallel lauritzen-spiegelhalter algorithm for probabilistic inference," in *Supercomputing*, 1994, pp. 320–329.

[6] V. K. Namasivayam and V. K. Prasanna, "Scalable parallel implementation of exact inference in bayesian networks," in *ICPADS*, 2006, pp. 143–150.

[7] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *ICS*, 2008, pp. 309–318.

[8] H. Jeon, Y. Xia, and V. K. Prasanna, "Parallel exact inference on a cpu-gpgpu heterogenous system," in *ICPP*, 2010, pp. 61–70.

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

[10] NVIDIA, *NVIDIA CUDA C Programming Guide*, July 2013. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," in *Commun. ACM*, 1986, pp. 1170–1183.

[12] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," in *Computers, IEEE Transactions on*, 1973, pp. 786–793.

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