GPU-Accelerated Constraint-Based Causal Structure Learning for Discrete Data

Christopher Hagedorn* Johannes Huegle*

Abstract

Learning the causal structures from high-dimensional observational data is an omnipresent challenge in data science. State-of-the-art methods for constraint-based Causal Structure Learning (CSL) apply conditional independence (CI) tests to determine the underlying causal structures. In the context of discrete data, each CI test requires calculating the marginals over contingency tables based on the respective observations. This calculation leads to long overall execution times.

In our work, we propose a parallel execution strategy tailored for constraint-based CSL on a Graphics Processing Unit (GPU) to accelerate the execution for discrete data. Hence, we introduce the `gpuPC` algorithm that performs all CI tests on a GPU and extends the existing parallel execution strategy for constraint-based CSL by calculating the marginals over contingency tables within units of threads. Further, `gpuPC` implements explicit memory management to handle the corresponding auxiliary data structures in GPU memory.

An experimental evaluation shows that `gpuPC` scales well even for higher-dimensional settings, with auxiliary data structures exceeding on-chip memory. In particular, running on NVIDIA Tesla V100 hardware `gpuPC` outperforms a GPU baseline by factors of up to 45.6 and further outperforms existing parallel CPU-based implementations running on 40 cores by a factor of 62.1.

Keywords: Causal structure learning, GPU, Bayesian Networks, PC algorithm, Discrete data

1 Introduction

Learning the causal structures from observational data is a ubiquitous challenge in data science. In particular, in high-dimensional settings in which the complexity of underlying causal structures impede a dedicated examination of single direct cause and effect relationships, e.g., via randomized control trials [19]. For example in car manufacturing, where causal structure learning has the goal to derive the interaction mechanisms between thousands of factors involved in the production process to add decision support for the production operator [7].

In the context of CSL, causal relationships are encoded in a Directed Acyclic Graph (DAG), a causal graphical model that defines the basis of a theoretical framework for causal reasoning [6, 15, 18, 27]. Methods for learning causal structures from observational data, i.e., the estimation of the DAG, build upon score-based and constraint-based, as well as, hybrid approaches. In this work, we focus on constraint-based approaches, which utilize CI tests to determine the undirected skeleton of the DAG, which is also called adjacency search, in a first step. In a subsequent step, the undirected edges are oriented through the repeated application of deterministic orientation rules.

The PC algorithm proposed by Spirtes et al. [27] is a well-known constraint-based approach that provides the basis for several extensions each tackling different - sometimes domain-specific constraints, e.g., for order-independence of variables [5] or under the incorporation of latent variables [4]. The algorithm's long execution, which is polynomial under the assumption of a sparse DAG, has raised interest in parallel execution on modern multi-core CPUs [13, 14, 21, 25]. Furthermore, extensions utilizing GPUs have been proposed for additional speedup [20, 22, 29]. One major limitation of the GPU-based algorithms for constraint-based CSL is their restriction to multivariate normal distributed data. In these settings, the algorithms leverage characteristics of the particular CI test to achieve significant speedup compared to CPU-based versions and tailor specific optimizations, such as, sharing of intermediate data structures, e.g., see cuPC-S [29]. For multivariate normal distributed data, the necessary calculations are built upon a pre-calculated correlation matrix to avoid access to the observational data within each CI test.

In contrast, constraint-based CSL from discrete distributed data requires the calculation of the specific contingency table and corresponding marginals from the observational data for each CI test. Pre-calculation of the corresponding auxiliary data structures for all CI tests is impractical given the limited available memory on a GPU. Hence, the application of existing GPU-

*Chair for Enterprise Platform and Integration Concepts, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany,{firstname.lastname}@hpi.de

based approaches in the context of discrete distributed data is not straightforward.

Therefore, we propose gpuPC, an extension of the order-independent version of the PC algorithm, PC-stable [5], under the consideration of the in-level parallelization strategies for constraint-based CSL [9, 25]. gpuPC extends the existing parallel execution strategy for GPU-based CSL [20, 29] by calculating the marginals over contingency tables within units of threads, according to the Single Instruction Multiple Threads (SIMT) execution model. Using explicit memory management gpuPC switches between the use of shared and global memory and even reduces parallelism if required, to handle the auxiliary data structures, which temporarily reside in GPU memory.

The remainder of the paper is organized as follows. Section 2 reviews preliminaries on constraint-based CSL and parallel execution strategies for the PC algorithm, both on Central Processing Unit (CPU) and GPU. We provide detail on our proposed gpuPC algorithm and its implementation in Section 3, which is followed by an experimental evaluation, comparing a GPU-accelerated baseline and existing CPU-based parallel implementations in Section 4. In Section 5, we discuss related work on parallel execution of constraint-based CSL. We conclude our work in Section 6.

2 Causal Structure Learning

In this section, we introduce the concepts that define the basis of constraint-based CSL, the PC algorithm, and details on its parallel execution. Further, we describe the requirements implied by the discrete data setting.

2.1 Preliminaries In the framework of causal inference, a direct causal relationship between two variables V_i to V_j is represented trough a direct edge $V_i \rightarrow V_j$ within a Causal Graphical Model (CGM) [18, 27]. In this sense, let a graph $\mathcal{G}$ be defined as a pair $\mathcal{G} = (\mathbf{V}, \mathbf{E})$, with a finite vertex set $\mathbf{V} = (V_1, \dots, V_N)$ corresponding to variables V_i, $i = 1, \dots, N$, and an edge set $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$. An edge $(V_i, V_j) \in \mathbf{E}$ is called directed, i.e., $V_i \rightarrow V_j$, if $(V_i, V_j) \in \mathbf{E}$ but $(V_j, V_i) \notin \mathbf{E}$. If both $(V_i, V_j) \in \mathbf{E}$ and $(V_j, V_i) \in \mathbf{E}$ the edge is called undirected, i.e., $V_i - V_j$. In this context, a graph $\mathcal{G}$ where all directed edges are replaced by undirected edges is called the skeleton $\mathcal{C}$ of $\mathcal{G}$. Further, if there exists an edge (V_i, V_j) within the skeleton $\mathcal{C}$ of $\mathcal{G}$, then the two vertices V_i and V_j, are called adjacent. The adjacency set $adj(\mathcal{G}, V_i)$ of the vertex $V_i \in \mathbf{V}$ in $\mathcal{G}$ are all vertices $V_j \in \mathbf{V}$ that are directly connected to V_i by an edge in the skeleton of $\mathcal{G}$. We speak of a DAG if all edges $\mathbf{E}$ of $\mathcal{G}$ are directed and $\mathcal{G}$ does not contain any cycle. Besides its information on direct causal relationships depicted in $V_i \rightarrow V_j$, a DAG entails information about the conditional independencies of the variables through the application of the d-separation criterion on the vertices [18]. Using the d-separation criterion, two variables $V_i, V_j \in \mathbf{V}$ are conditionally independent given a set of variables $\mathbf{S} \subset \mathbf{V} \setminus \{V_i, V_j\}$, denoted by $V_i \perp\!\!\!\perp V_j \mid \mathbf{S}$ if and only if the vertices V_i and V_j are d-separated by the set $\mathbf{S}$ in the corresponding DAG. Methods for CSL are built upon this relation to conditional independencies in the probability distribution P. Therefore, it needs to be assumed that causal sufficiency, i.e., the incorporation of all relevant variables in the DAG, and causal faithfulness, i.e., that the probability distribution P satisfies the d-separation criterion for the DAG, hold. Note, that the same conditional independence information can be described by multiple DAGs that form a Markov equivalent class that can be described uniquely by a Complete Partially Directed Acyclic Graph (CPDAG) [2, 3]. Thus, the goal of CSL is to estimate the equivalence class of the DAG $\mathcal{G}$ based on the information about conditional independencies in the probability distribution P of the involved variables [5, 8, 26].

2.2 Constraint-based CSL Methods for constraint-based CSL build upon the previously addressed concept of d-separation to estimate the skeleton $\mathcal{C}$ by an adjacency search of the involved variables as a common first step that is then extended to the CPDAG through the repeated application of deterministic orientation rules [5, 8, 10, 26]. Our work focuses on the GPU-accelerated execution of the PC algorithm under the consideration of the GPU hardware characteristics, where the parallel execution of threads follows the SIMT execution model [11]. To introduce the parallel computing paradigm of constraint-based CSL, we sketch the adjacency search of the order-independent version of the PC algorithm [5] in Algorithm 1, which sets the foundation for parallel versions on the CPU and GPU.

Starting with a complete undirected skeleton $\mathcal{C}$ the PC algorithm conducts CI tests with an increasing size l of the separation set $\mathbf{S}$ of adjacent vertices to subsequently remove edges $V_i - V_j$ from the skeleton $\mathcal{C}$ for which the variables V_i and V_j are determined as being independent given $\mathbf{S}$. Hence, the algorithm only needs to query CI tests of vertices V_i and V_j given separation sets $\mathbf{S}$ with size $l = 0$ up to the maximum size of the adjacency sets of the vertices in the underlying DAG $\mathcal{G}$, i.e., up to $\max_{V_i \in \mathbf{V}} |adj(\mathcal{G}, V_i) \setminus \{V_j\}|$. This results in a polynomial complexity for sparse underlying true DAGs $\mathcal{G}$ which makes the algorithm computationally feasible even in high-dimensional settings [8].

Algorithm 1 Adjacency search of PC algorithm
Input: V
Output: $\mathcal{C}$, **Sepset**

```
1: Start with fully connected skeleton $\mathcal{C}$ and $l = -1$;
2: repeat
3:    $l = l + 1$;
4:    for all Variables $V_i$ in $\mathcal{C}$ do
5:       Let $a(V_i) = adj(\mathcal{C}, V_i)$;
6:    end for
7:    repeat
8:       Select adjacent pair $V_i, V_j$ in $\mathcal{C}$
         with $|a(V_i) \setminus \{V_j\}| \geq l$;
9:       repeat
10:         Choose $\mathbf{S} \subseteq a(V_i) \setminus \{V_j\}$ with $|\mathbf{S}| = l$;
11:         Perform $T(V_i, V_j | \mathbf{S})$;
12:         if $\mathrm{p}(V_i, V_j | \mathbf{S}) \geq \alpha$ then
13:            Delete edge $V_i - V_j$ from $\mathcal{C}$;
14:            Save $\mathbf{S}$ in Sepset;
15:         end if
16:      until edge $V_i - V_j$ is deleted in $\mathcal{C}$ or all
         $\mathbf{S} \subseteq a(V_i) \setminus \{V_j\}$ with $|\mathbf{S}| = l$ were chosen
17:   until all adjacent vertices $V_i$ and $V_j$ in $\mathcal{C}$ such
      that $|a(V_i) \setminus \{V_j\}| \geq l$ were considered
18: until each adjacent pair $V_i, V_j$ in $\mathcal{C}$ satisfy $|a(V_i) \setminus \{V_j\}| \leq l$
19: return $\mathcal{C}$, Sepset
```

For every level $l = 0, \ldots, \max_{V_i \in \mathbf{V}} |adj(\mathcal{G}, V_i) \setminus \{V_j\}|$ the adjacency sets $a(V_i) = adj(\mathcal{C}, V_i)$ of variables V_i with respect to the current skeleton $\mathcal{C}$ are computed and stored to allow for order-independence (see lines 4–6). This enables parallel execution within each level l, without synchronization while processing the current level l [10]. Hence, existing CPU-based approaches process the adjacent pairs V_i, V_j in $\mathcal{C}$ within each level (see lines 7–17) in parallel [10, 25]. Based on the GPU hardware characteristics GPU-accelerated versions extend the parallel execution strategy through nested parallelism [20, 29]. In detail, while processing each adjacent pair V_i, V_j in parallel, the corresponding CI tests (see lines 9–16) are executed in parallel, too. This parallel execution strategy is mapped to the GPU execution model, by processing adjacent pairs V_i, V_j in separate thread blocks, and using groups of threads within each thread block to perform the corresponding CI tests. The returned skeleton $\mathcal{C}$ and the separation sets **Sepset** are the basis for the application of deterministic orientation rules to extend $\mathcal{C}$ to the corresponding CPDAG [5, 8, 10, 26].

2.3 Discrete Distribution Model In this work, we consider discrete distributed data where the hypotheses on conditional independence need to be examined between random variables V_i and V_j given a set of variables $\mathbf{S}$ with respective discrete domains $\mathcal{V}_i$, $\mathcal{V}_j$, and $\mathcal{S}$. In this context, $V_i \perp\!\!\!\perp V_j \mid \mathbf{S}$ holds true if for all $(v_i, v_j, \mathbf{s}) \in \mathcal{V}_i \times \mathcal{V}_j \times \mathcal{S}$ we have that $P(V_i = v_i, V_j = v_j | \mathbf{S} = \mathbf{s}) = P(V_i = v_i | \mathbf{S} = \mathbf{s}) \cdot P(V_j = v_j | \mathbf{S} = \mathbf{s})$, where v_i, v_j and $\mathbf{s}$ are the corresponding realization values and vector of realization values, respectively. Classically, related statistical hypothesis testing for CI for this discrete distribution model, e.g., the well-known Pearson's χ^2 test of independence is based upon the properties of marginals over contingency tables [1, 15].

Hence, we consider a statistical hypothesis test $T(V_i, V_j | \mathbf{S})$ for conditional independence $V_i \perp\!\!\!\perp V_j \mid \mathbf{S}$ and denote $T(V_i, V_j)$ if $\mathbf{S} = \emptyset$, i.e., in case of testing for $V_i \perp\!\!\!\perp V_j$. Furthermore, let $N_{v_i v_j \mathbf{s}}$ denote the frequency of samples where $V_i = v_i$, $V_j = v_j$ and $\mathbf{S} = \mathbf{s}$ in the data such that $N_{v_i v_j \mathbf{s}}$ reflects the corresponding entry in the contingency table. Moreover, let $N_{v_i + \mathbf{s}} = \sum_{\mathcal{V}_j} N_{v_i v_j \mathbf{s}}$ denote the marginal with respect to V_i and similarly for $N_{+ v_j \mathbf{s}}$, $N_{v_i v_j +}$, $N_{++\mathbf{s}}$, such that N_{+++} is equal to the total sample size n. Finally, let $|\mathcal{V}_i|$, $|\mathcal{V}_j|$, and $|\mathcal{S}|$ denote the size of the domains of $\mathcal{V}_i$, $\mathcal{V}_j$, and $\mathcal{S}$, respectively. Under the null hypothesis of conditional independence, i.e., $V_i \perp\!\!\!\perp V_j \mid \mathbf{S}$ the expected frequency where $V_i = v_i$, $V_j = v_j$, and $\mathbf{S} = \mathbf{s}$ is given by $E_{v_i v_j \mathbf{s}} = \frac{N_{v_i + \mathbf{s}} \cdot N_{+ v_j \mathbf{s}}}{N_{++\mathbf{s}}}$ and the actual frequency is $N_{v_i v_j \mathbf{s}}$. CI tests $T(V_i, V_j | \mathbf{S})$ build upon the examination of an overall discrepancy between these two quantities for all cells of the contingency tables. For example, Pearson's χ^2 test of independence is based upon the test statistic

$$\chi^2 (V_i, V_j | \mathbf{S}) = \sum_{\mathcal{V}_i \mathcal{V}_j \mathcal{S}} \frac{\left(N_{v_i v_j \mathbf{s}} - E_{v_i v_j \mathbf{s}}\right)^2}{E_{v_i v_j \mathbf{s}}},$$

which is equal to zero whenever $E_{v_i v_j \mathbf{s}}$ is equal to zero. Under the null hypothesis, the test statistic $\chi^2 (V_i, V_j | \mathbf{S})$ is asymptotically distributed as χ^2_{df} with $df = (|\mathcal{V}_i| - 1)(|\mathcal{V}_j| - 1) \cdot |\mathcal{S}|$ degrees of freedom. Hence, the corresponding p-value $\mathrm{p}(V_i, V_j | \mathbf{S})$ can be calculated as $1 - \mathrm{F}(\widehat{\chi}^2)$ where F is the cumulative distribution function of χ^2_{df} and $\widehat{\chi}^2$ the calculated statistic derived of the marginals over the observed frequencies in the corresponding contingency table. Hence, given the significance level α, we reject the null-hypothesis $V_i \perp\!\!\!\perp V_j | \mathbf{S}$ against the two sided alternative $V_i \not\perp\!\!\!\perp V_j | \mathbf{S}$ if for the corresponding p-value it holds that $\mathrm{p}(V_i, V_j | \mathbf{S}) \leq \alpha$.

Besides the previously introduced Pearson's χ^2 test, a number of methods for CI testing in the discrete distribution model have been proposed, e.g., permutation testing [28] or methods based on the stochastic complexity [16]. These methods incorporate, in one way or another, the examination of an overall discrepancy between the marginals over contingency tables [1, 15], the contribution of this work applies to those settings, too.

3 gpuPC: GPU-Accelerated Causal Structure Learning for Discrete Distributed Data

In the following, we describe the gpuPC algorithmand its implementation using the parallel computing platform and programming model CUDA [17].

3.1 CUDA Programming Model Within CUDA, the code of functions to be executed on a GPU is organized in kernels. These kernels are launched specifying the number of threads, each executing the code sequentially. The threads operate in units, so-called warps, with each thread performing the same operation. Warps commonly have a size of 32 threads. Diverging branches within warps are executed by all threads, using no-ops for masking threads of different branches. Further, threads are grouped into blocks. Each block is mapped onto a Streaming Multiprocessor (SM) of the GPU. Thus, allowing for threads within a block to access on-chip shared memory and fast synchronization. For referencing threads inside CUDA code, each thread and each block have a three-dimensional id, `threadIdx.(x,y,z)` and `blockIdx.(x,y,z)`, which we abbreviate with tx or bx. In addition to on-chip shared memory, accessible within blocks only, GPUs also provide global memory accessible across all blocks. For example, recent NVIDIA GPU generations provide High Bandwidth Memory (HBM) with of up to 32 GB of size, as global memory. Thus, limiting the amount of data that can be accessed during kernel execution.

3.2 Implementation of gpuPC The proposed gpuPC algorithm follows the general outline of the cupc algorithm [29]. Hence, gpuPC starts a separate kernel for each level l, that requires different concepts for level $l = 0$ and for higher level $l \geq 1$. Before a kernel for a higher level $l \geq 1$ is invoked, the current skeleton $\mathcal{C}$ is compacted to $\mathcal{C}_c$ [29] and an explicit memory management step is executed.

Due to the discrete distribution model gpuPC adopts a fine-grained parallel execution strategy. In particular, while processing each CI test of adjacent pairs V_i, V_j in parallel, the calculation of the marginals over contingency tables is done in parallel, too. In practice, gpuPC uses threads, in multiples of warps, within a thread block to jointly calculate the marginals over contingency tables. Further, gpuPC the warps within a thread block are split to operate on multiple CI tests in parallel. Lastly, each block operates on an adjacent pair V_i, V_j. While gpuPC targets the execution on a single GPU, an extension to run on multiple GPUs is future work, e.g., using a block-based approach [22].

3.3 Kernel for Level Zero In level $l = 0$, the separation set is empty and for each adjacent pair of variables V_i, V_j, only a single independence test $T(V_i, V_j)$ is performed. Hence, parallel execution is straightforward performing all independence tests in parallel. In contrast to existing GPU-accelerated implementations [20, 22, 29], which map the independence tests to threads and blocks, gpuPC performs each single independence test in a separate block and uses the threads within each block, denoted by δ, to jointly calculate the marginals over contingency tables, see Algorithm 2.

Algorithm 2 Kernel for level zero using shared memory
Input: D, α, V, $\mathcal{C}$, **Sepset**, Category Counts $\cup_{\mathbf{V}} |\mathcal{V}_i|$
Output: $\mathcal{C}$, **Sepset**
# of blocks: $N \times N$
# of threads per block: δ

```
1: $i = bx, j = by$
2: if $bx < by$ then
3:     Initialize $N_{v_i v_j}$,$N_{+v_j}$,$N_{v_i+}$ in shared memory
4:     for $k = tx; k < n; k = k + \delta$ do
5:         $atomicAdd(N_{v_i v_j}[D[V_i][k] \times |\mathcal{V}_i| + D[V_j][k]], 1)$
6:     end for
7:     syncthreads
8:     if $tx == 0$ then
9:         Calculate $N_{+v_j}$,$N_{v_i+}$
10:        Perform $T(V_i, V_j)$
11:        if $p(V_i, V_j) \leq \alpha$ then
12:            Delete edge $V_i - V_j$ from $\mathcal{C}$
13:            Sepset$(V_i, V_j) = -1$
14:        end if
15:    end if
16: end if
```

The kernel is launched with # of blocks set to # of pairs of V_i, V_j, and # of threads per block determined by δ. At first, the indices of the pair of variables V_i, V_j are determined (see line 1). Under the assumption of reasonable small category counts $|\mathcal{V}_i| \times |\mathcal{V}_j|$, the contingency table $N_{v_i v_j}$ and marginals N_{+v_j}, N_{v_i+} are stored in shared memory (see line 3). Otherwise, these auxiliary data structures are allocated in global memory by a primary thread $tx = 0$. All threads within a block jointly calculate the contingency table $N_{v_i v_j}$, required to perform $T(V_i, V_j)$ (see lines 4-6). Note, gpuPC assumes no missing observations, hence n is the same for all V_i. After synchronizing the threads (see line 7), a primary thread with $tx = 0$ calculates the marginals N_{+v_j}, N_{v_i+}, and performs $T(V_i, V_j)$ according to the method described in Section 2. In the case of $p(V_i, V_j) \leq \alpha$, the edge between V_i, V_j is deleted in $\mathcal{C}$ and the corresponding **Sepset** entry is marked (see lines 11-14). Note, that the kernel implementation presented in

Algorithm 2 assumes that enough memory is available to process all pairs of V_i, V_j in parallel within separate blocks. Further, data D is assumed to be organized in a columnar fashion, allowing for aligned access of the threads within a warp, while calculating $N_{v_i v_j}$.

3.4 Kernel for Arbitrary Level $l \geq 1$ Algorithm 3 outlines the implementation of the kernel for arbitrary level $l \geq 1$. When launched, the kernel receives a predetermined amount of scratch space in global memory to store the auxiliary data structures. In detail, the algorithm uses `cudaMallocManaged` to allocate memory for the scratch space, which is used within kernels for level $l \geq 1$. With recent GPU generations that allow to over-allocate memory, exceeding the amount provided on-chip, processing of datasets that require auxiliary data structures exceeding on-chip memory is enabled. Yet, for high-dimensional data, the explicit memory management step determines a factor F that limits the parallel execution of the kernel, to constrain the memory demand. Hence, the kernel is launched with $\frac{N}{\mathsf{F}} \times \frac{N}{\beta}$ blocks, and the code shown in Algorithm 3 is executed F times for each block. In the regular setting, the kernel is launched with # of blocks set to $N \times \frac{N}{\beta}$ blocks and # of threads set to $\delta \times \gamma \times \beta$. In detail, within each thread block β pair of variables V_i, V_j are processed. For each of the pairs of variables V_i, V_j, γ CI tests are conducted in parallel using δ threads to jointly calculate the marginals over contingency tables.

At the beginning of the kernel, the indices i, j are determined and the frequently accessed adjacencies $adj(\mathcal{C}_c, V_i)$ are copied into shared memory $a_s(V_i)$ (see lines 1–4). Next, all possible combinations of separation sets are iterated (see lines 5 – 37). At first, a parallel combination function is called to determine the current separation set separation $S_{1..l}$, see [29]. Next, the memory for the auxiliary data structures in the scratch space is set to 0 by a primary thread. After synchronization, δ threads jointly calculate the contingency tables from data D. In this step, all variables V_k in $S_{1..l}$ have to be considered (see lines 14–17). Using the calculated contingency table $N_{v_i v_j \mathbf{s}}$, the δ threads jointly calculate the marginals $N_{+v_j\mathbf{s}}$,$N_{v_i+\mathbf{s}}$ and $N_{++\mathbf{s}}$ (see lines 20–22). In a subsequent step, each thread calculates a part of the statistic derived from the marginals of the corresponding contingency table, which is stored in the array *local_statistic* in shared memory (see lines 24–26). In a final step, the primary thread $tx = 0$ performs the CI test for the pair of adjacent variables V_i, V_j given $S_{1..l}$, based on the sum over *local_statistic*. In the case of $\mathrm{p}(V_i, V_j|S_{1..l}) \leq \alpha$, the edge between V_i, V_j is deleted and the corresponding **Sepset** entry is marked (see lines 31–34). Additionally, a flag is set to enable an early termination of the loop over the possible separation sets. Note, in case of small values of $\max_{\mathbf{V}}(|\mathcal{V}_i|)$, i.e., 2, the auxiliary data structures could be kept in shared memory, or thread local memory for faster execution times. Yet, for larger values of $\max_{\mathbf{V}}(|\mathcal{V}_i|)$ the auxiliary data structures exceed the available space of these memory fast memory options.

Algorithm 3 Kernel for arbitrary level $l \geq 1$
Input: D, α, $\mathbf{V}$, $\mathcal{C}$, $\mathcal{C}_c$, **Sepset**, $\cup_{\mathbf{V}}|\mathcal{V}_i|$
Output: $\mathcal{C}$, **Sepset**
# of blocks: $N \times \frac{N}{\beta}$
# of threads per block: $\delta \times \gamma \times \beta$

```
 1: $i = bx$
 2: Let $a_s(V_i) = adj(\mathcal{C}_c, V_i)$ in shared memory
 3: $pos = by \times \beta + tz$
 4: $j = a_s[pos]$
 5: for $k = ty; t < \binom{|a_s(V_i)|-1}{l}; t = t + \gamma$ do
 6:     $Pos_{1..l} = parallel\_comb(|a_s(V_i)| - 1, l, k)$
 7:     $S_{1..l} = a_s[Pos_{1..l}]$
 8:     if $tx == 0$ then
 9:         set memory for $N_{v_i v_j \mathbf{s}}$,$N_{+v_j\mathbf{s}}$,$N_{v_i+\mathbf{s}}$,$N_{++\mathbf{s}}$
            to 0 in scratch space
10:     end if
11:     syncthreads
12:     for $g = tx; g < n; g = g + \delta$ do
13:         $sum\_sep = 0, cat\_sep = 1$
14:         for all $s$ in $S_{1..l}$ do
15:             $sum\_sep = sum\_sep + (D[s][g] \times cat\_sep)$
16:             $cat\_sep = cat\_sep \times |\mathcal{V}_s|$
17:         end for
18:         $atomicAdd(N_{v_i v_j \mathbf{s}}[sum\_sep \times |\mathcal{V}_i| \times |\mathcal{V}_j| +$
            $D[V_i][g] \times |\mathcal{V}_j| + D[V_j][g]], 1)$
19:         syncthreads
20:         for $g = tx; g < |\mathcal{V}_i| \times |\mathcal{V}_j| \times |\mathbf{S}|; g = g + \theta$ do
21:             Calculate $N_{+v_j\mathbf{s}}$,$N_{v_i+\mathbf{s}}$,$N_{++\mathbf{s}}$
22:         end for
23:         syncthreads
24:         for $g = tx; g < |\mathbf{S}|; g = g + \theta$ do
25:             Calculate $local\_statistic[tx]$
26:         end for
27:         syncthreads
28:         if $tx == 0$ then
29:             $Sum(local\_statistic)$
30:             Perform $T(V_i, V_j|S_{1..l})$
31:             if $\mathrm{p}(V_i, V_j|S_{1..l}) \leq \alpha$ then
32:                 Delete edge $V_i - V_j$ from $\mathcal{C}$
33:                 Sepset$(V_i, V_j) = S_{1..l}$
34:             end if
35:         end if
36:     end for
37: end for
```

4 Experimental Evaluation

In this section, we evaluate the proposed gpuPC algorithm for discrete data within two sets of experiments. The first experiments compare gpuPC to a GPU-accelerated baseline implementation that follows the parallel execution strategy proposed by cupc [29]. In the second experiments compare gpuPC and the GPU-accelerated baseline to existing parallel implementations of the PC algorithm executed on a CPU.

4.1 Experimental Setup The experiments are conducted on an enterprise-grade server with 2 Intel ® Xeon ® Gold 6148 CPU with 20 cores each, which is equipped with an NVIDIA V100 card, with 32 GB of HBM. The GPU card is connected via PCI-E 4.0. Furthermore, the server is equipped with 1.5 TB of RAM, allowing to keep all data in memory during the execution of the experiments. The operating system is an Ubuntu 18.04 and the NVIDIA driver version 410.79 is installed with CUDA version 9.1.

For the experiments, we compare gpuPC to a GPU-accelerated baseline using the same implementation of the CI test and follows the parallel execution strategy proposed by cupc-E [29], which we call disc-cupc. Similar to cuPC, disc-cupc uses the threads within a thread block to process γ CI tests for β edges. Further, starting $N \times \frac{N}{\beta}$ blocks ensures that each edge is processed. In contrast to gpuPC, the implementation of disc-cupc does not support any explicit memory management and each thread allocates global memory for the auxiliary data structures inside the kernel. For both GPU-accelerated implementations configuration parameters for the parallel execution have been tuned on several synthetic settings. The parameters for disc-cupc are in accordance to the parameters of cupc-E [29]. Hence, we set $\gamma = 32$ and $\beta = 2$. For gpuPC we set $\delta = 64$, $\gamma = 2$ and $\beta = 1$.

Further gpuPC is compared to CPU implementations from the R-packages bnlearn [23] and parallelPC [9]. The package bnlearn provides an implementation of the CI test written in C. Yet, the adjacency search is parallelized using the R library parallel, which results in the launch of multiple R processes. The package parallelPC provides a parallel implementation using the R library parallel and is entirely written in R. Thus, it avoids the overhead of mapping R data structures to C data structures.

Measurements comparing GPU- and CPU-based versions include all data transfer between devices. Furthermore, all implementations are called from their R interface. Thus, measurements include times for data copies from R to C or CUDA. If not stated differently, we repeat each experiment run at least 10 times and present the median runtime. Further, we set the tuning parameter α to 0.01, which is common in application [5].

For the first set of experiments, we investigate the performance under the dimensions relevant to the PC algorithm considering the characteristics of a CI test for discrete data and its execution on a GPU. Thus, we examine the number of vertices N, the number of observations n, the maximum category count $\max_{\mathbf{V}}(|\mathcal{V}_i|)$, the number of CI tests per edge T, and a decay factor d. The decay factor d describes the percentage of edges that are removed equally distributed from skeleton $\mathcal{C}$ within each level l of the adjacency search. We consider T and d as a proxy for the impact of edge deletion from the underlying $\mathcal{C}$ to ensure reproducibility and traceability of experiments over levels. Furthermore, varying the number of T allows to investigating how the implementations cope with load imbalance due to a different number of CI tests per edge [21].

In the second set of experiments, we consider discrete distributed datasets, sampled from benchmark Bayesian networks from the bnlearn repository [24] with varying characteristics, see Table 1. In addition to high-dimensional datasets, e.g., LINK and MUNIN, that provide ample opportunity for parallel execution, we also consider small-dimensional datasets, e.g., ALARM and ANDES with varying sample size.

Dataset	N	n	$\max_{\mathbf{V}}(\lvert\mathcal{V}_i\rvert)$
ALARM	37	200,000	4
ANDES	223	20,000	2
LINK	724	20,000	4
MUNIN	1,041	20,000	21

Table 1: Characteristics of datasets, generated based on benchmark Bayesian networks from the bnlearn repository. N - number of vertices, n - number of observations, $\max_{\mathbf{V}}(|\mathcal{V}_i|)$ - maximal category count.

4.2 Comparing GPU-based Implementations in Synthetic Settings We investigate the scalability of gpuPC in five different dimensions. We report the measured speedup compared to the baseline disc-cupc for the different experiment runs in Table 2.

For a varying N, which corresponds to a varying number of CI tests, i.e., larger N imply more CI tests, the performance difference remains fairly steady. As the number of vertices N doubles, the actual execution times quadruple, which follows the polynomial complexity [8]. For a varying $\max_{\mathbf{V}}(|\mathcal{V}_i|)$, larger values lead to higher memory demand of the auxiliary data structures. Correspondingly, the measured execution times increase with larger $\max_{\mathbf{V}}(|\mathcal{V}_i|)$. Yet, the speedup remains similar, with an exception for $\max_{\mathbf{V}}(|\mathcal{V}_i|) = 2$.

Dimensions	Parameter settings / Speedup measurements				
N	1K	2K	4K	8K	
	4.2	4.38	4.1	2.97	
$\max_{\mathbf{V}}(\lvert\mathcal{V}_i\rvert)$	2	4	8	10	20
	3.6	7.67	7.5	6.4	∞
n	10K	50K	100K	500K	1M
	8.4	29.1	37.7	45.5	45.6
T	1	4	16	32	128
	9.4	9.7	6.25	4.25	4.36
d	0.9	0.75	0.5		
	8.2	12.0	∞		

Table 2: Speedup of gpuPC compared to disc-cupc, based on median execution times of 10 runs. For each experiment all dimensions except the one of interest are fixed, with the following values $N = 1,000$, $\max_{\mathbf{V}}(|\mathcal{V}_i|) = 4$, $n = 10,000$, $T = 32$, $d = 0.9$. Note, that ∞ indicates that disc-cupc ran out of memory.

For $\max_{\mathbf{V}}(|\mathcal{V}_i|) > 10$, `disc-cupc` fails due to memory requirements exceeding available on-chip memory, while `gpuPC` remains operating. Yet, the measured execution times of `gpuPC` degrade; due to a limited parallel execution and poorer caching behavior given large auxiliary data structures. For a varying n, the difference in the parallel execution strategies yields the largest performance gap between the two implementations. `gpuPC` is up to factors of 45 faster than `disc-cupc` for $n \geq 500K$, while it is only a factor of 8.4 faster for $n = 10K$. The measured execution times for `gpuPC` increase linear with n for $n \geq 500K$. For smaller n the increase is sublinear. For a varying T, which reflects settings of load imbalance, the speedup is fairly constant for $T \geq 32$. Yet, for $T < 32$ the speedup is higher for smaller values of T. The results are in line with the fine-granular execution strategy of `gpuPC`, which is better suited for situations of load imbalance. In particular, in `disc-cupc` at least 32 CI tests per edge are conducted in parallel due to γ and the warp size of 32, which explains the overhead for $T < 32$. For a varying d, a smaller decay factor d indicates more remaining edges after each level. Hence, more CI tests are conducted per level. Besides, a higher level may be reached leading to larger separation sets with higher memory demand for the auxiliary data structures. Therefore, `disc-cupc` fails for $d = 0.5$. Further, we observe that the performance gap increases with smaller d, as `gpuPC` better handles both a larger number of CI tests, as well as, higher-order CI tests.

In summary, `gpuPC` is faster compared to `disc-cupc` in all considered dimensions. In particular, when considering the number of observations n, `gpuPC` is faster by over an order of magnitude. Furthermore, `gpuPC` scales for datasets with high memory demand for auxiliary data structures, while `disc-cupc` result in errors. Yet, to handle datasets with a high memory demand for auxiliary data structures, e.g., with $\max_{\mathbf{V}}(|\mathcal{V}_i|)$ of 20, `gpuPC` restricts the degree of parallelism, leading to reduced execution time performance. Note, the restriction of the degree of parallelism occurs with a lower memory demand, in case of a GPU with lower available memory.

4.3 Comparing GPU-based and CPU-based Implementations on Benchmark Bayesian Networks

In this experiment, we compare the measured execution times of the adjacency search for all four implementations, see Table 3. Note, we executed the CPU-based implementations running on 40 cores to achieve a fairer comparison.

Dataset	parallelPC	bnlearn	disc-cupc	gpuPC
ALARM	579.54 s	14.71 s	0.95 s	0.26 s
ANDES	187.24 s	20.78 s	1.41 s	0.38 s
LINK	16,510.31 s	141.65 s	12.93 s	2.28 s
MUNIN	110,740.5 s	273.79 s	97.45 s	14.99 s

Table 3: Execution time in seconds for the adjacency search of bnlearn and parallelPC running on 40 cores on CPU, disc-cupc and gpuPC running on GPU using different benchmark datasets.

Comparing the two CPU-based implementations, a significant difference in execution times becomes visible. The implementation from `bnlearn` is faster by a factor of up to 400 compared to `parallelPC`. We assume the difference of execution is a result of the efficient C implementation in `bnlearn` and overhead in the R implementation of `parallelPC`. Both GPU-accelerated implementations are faster than the CPU-based implementations. `disc-cupc` achieves a speedup ranging between 2.8 to 15.5 compared to `bnlearn`. `gpuPC` has the fastest execution times on all datasets, achieving a speedup of 18.3 to 62.1 compared to `bnlearn` and is faster by a factor of up to 6.5 compared to `disc-cupc`. There is no difference in the observed pattern of the measured execution times for small- or high-dimensional data, as well as for data with a larger number of observations.

5 Related Work

In high-dimensional settings, the large number of CI tests performed during the adjacency search within the PC algorithm has raised a significant interest in a parallel , e.g., see [9, 13, 14, 20, 21, 22, 25, 29].

The R-package `bnlearn` implements several constraint-based CSL algorithms, including the PC algorithm and its extensions. All implementations follow a generalized framework for parallel execution introduced by Scutari [25] that proposes to process the pairs of variables to be tested for independence within each level in parallel. In contrast to `bnlearn`, `gpuPC` targets a GPU as an execution device. While `gpuPC` also parallelize over the pairs of variables, it extends to have threads within units jointly compute the contingency tables, adding a layer of parallel execution.

Madsen et al. [14] propose two parallel implementations for discrete distributions. One implementation is based on the concept of Balanced Incomplete Block Designs [12] to distribute the computations of CI tests to threads in a shared memory system. The second version is designed for datasets with a large number of observations. In this case, the contingency tables are computed in parallel. Furthermore, their implementation handles higher-order CI tests through the ranking of edges according to a test score to perform the most promising CI tests first. Also, they limit higher-order CI tests to a separation set size of three. In contrast `gpuPC` runs on a GPU and uses a combination of the proposed parallel execution scheme. While threads within units jointly compute the contingency tables, multiple of these are executed in parallel on the SMs of the GPU.

Further, Thuc et al. [9] propose a parallel version implemented in the R-Package `ParallelPC`, showing performance improvements on multivariate normal distributed gene expression data. Schmidt et al. [21] study the impact of load balancing mechanisms in a similar setting. They apply a dynamic task distribution mechanism to achieve speed-up compared to static task distribution approaches. Despite potential performance gain, `gpuPC` does not include dynamic task distribution, given that this mechanism is costly to implement on a GPU and the impact of load imbalance for discrete distributions is small [25]. Further, the finer-grained parallel execution strategy copes better with load imbalance.

Existing GPU-accelerated approaches focus on multivariate normal distributed data. Schmidt et al. [20] provide an implementation with GPU kernels performing unconditioned independence tests and CI tests with a separation set of size 1, only. Arguing that only a few higher-order tests for independence have to be conducted in the gene expression data, used in their evaluation. They extend their work, with an out-of-core approach that works beyond GPU device memory limitations [22]. They split data into smaller chunks and use CUDA streams to overlap data transfer, kernel execution on the GPU, and splitting of the data on the CPU reducing overhead. In contrast `gpuPC` focuses on discrete data distributions and is not limited by a separation set size. Currently, it is limited by the device memory, with regards to vertices N and data D. Yet, to overcome this shortcoming the out-of-core approach [22] could be integrated into `gpuPC` in future work.

Zarebavani et al. propose `cupc`, a generalized implementation conducting all CI tests on the GPU parallelizing over pairs of variables and separation sets respectively [29]. Further, they share intermediate results within the CI tests, i.e., the pseudo-inverse, to reduce computation. Within their evaluation, they report a speedup of several orders of magnitude compared to CPU-based implementations. Yet, they compare to a parallel version written entirely in the R-language, which showed similar runtimes as a single-threaded C++ version. Within `gpuPC`, we utilize a similar scheme for parallel execution as `cupc`, yet, units of threads jointly work on the computation of the contingency table and marginal. Furthermore, in our evaluation we compare to a parallel implementation completely written in C++ using up to 40 cores, providing ample opportunity for fast parallel execution on a CPU.

6 Summary

In this work, we propose a parallel execution strategy tailored for constraint-based CSL on a GPU to accelerate the execution for discrete data. Therefore, we introduce the `gpuPC` algorithm, which performs the adjacency search of the PC algorithm and the CI tests in parallel on the GPU. In particular, `gpuPC` incorporates the data access characteristics a of CI test for discrete data through a fine-grained parallel execution strategy, which calculates the marginals over contingency tables within units of threads. Further, `gpuPC` implements explicit memory management to handle the corresponding auxiliary data structures in GPU memory, even in cases that exceed on-chip memory.

We demonstrate the performance gain of `gpuPC` compared to a GPU baseline implementation and existing parallel CPU-based implementations, using discrete data sampled from benchmark Bayesian networks. Running the experiments on a multi-core system with 40 cores provide ample opportunity for parallel execution on the CPU. Nevertheless, `gpuPC` outperforms the CPU-based implementations by factors of up to 62.1. Moreover, due to the more fine-granular parallel execution strategy in combination with the explicit memory management, `gpuPC` remains operating in situations, in which auxiliary data structures exceed on-chip memory, and is up to 6.5 times faster on common benchmark datasets compared to a GPU baseline implementation.

In settings of CSL that require access to the observational data within each CI test, e.g., for discrete data, the proposed parallel execution strategy implemented in `gpuPC` outperforms existing parallel execution strategies. In the future, we are going to examine the transferability of the introduced parallel execution strategy to data distribution models that require more complex statistical methods with CI tests that, generally, follow the access patterns considered in this work.

References

[1] A. Agresti, *A Survey of Exact Inference for Contingency Tables*, Statistical Science, 7 (1992), pp. 131–153.

[2] S. A. Andersson et al., *A Characterization of Markov Equivalence Classes for Acyclic Digraphs*, Annals of Statistics, 25 (1997), pp. 505–541.

[3] D. M. Chickering, *Learning Equivalence Classes of Bayesian-Network Structures*, Journal of Machine Learning Research, 2 (2002), pp. 445–498.

[4] D. Colombo et al., *Learning High-Dimensional Directed Acyclic Graphs with Latent and Selection Variables*, Annals of Statistics, 40 (2012), pp. 294–321.

[5] D. Colombo and M. H. Maathuis, *Order-Independent Constraint-Based Causal Structure Learning*, Journal of Machine Learning Research, 15 (2014), pp. 3921–3962.

[6] D. Heckerman et al., *Learning Bayesian Networks: The Combination of Knowledge and Statistical Data*, Machine Learning, 20 (1995), pp. 197–243.

[7] J. Huegle et al., *How Causal Structural Knowledge Adds Decision-Support in Monitoring of Automotive Body Shop Assembly Lines*, in Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI Organization, 7 2020, pp. 5246–5248. Demos.

[8] M. Kalisch and P. Bühlmann, *Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm*, Journal of Machine Learning Research, 8 (2007), pp. 613–636.

[9] T. D. Le et al., *ParallelPC: An R Package for Efficient Causal Exploration in Genomic Data*, in Pacific-Asia Conference on Knowledge Discovery and Data Mining, Cham, 2018, Springer International Publishing, pp. 207–218.

[10] ———, *A Fast PC Algorithm for High Dimensional Causal Discovery with Multi-Core PCs*, IEEE/ACM Transactions on Computational Biology and Bioinformatics, 16 (2019), pp. 1483–1495.

[11] E. Lindholm et al., *NVIDIA Tesla: A Unified Graphics and Computing Architecture*, IEEE Micro, 28 (2008), pp. 39–55.

[12] A. L. Madsen et al., *A New Method for Vertical Parallelisation of TAN Learning Based on Balanced Incomplete Block Designs*, in Probabilistic Graphical Models, Cham, 2014, Springer International Publishing, pp. 302–317.

[13] ———, *Parallelisation of the PC Algorithm*, in Advances in Artificial Intelligence, New York, NY, USA, 2015, Springer-Verlag New York, Inc., pp. 14–24.

[14] ———, *A Parallel Algorithm for Bayesian Network Structure Learning from Large Data Sets*, Knowledge-Based Systems, 117 (2017), pp. 46–55.

[15] D. Margaritis, *Learning Bayesian Network Model Structure From Data*, PhD thesis, School of Computer Science, Carnegie-Mellon University, 5 2003.

[16] A. Marx and J. Vreeken, *Testing Conditional Independence on Discrete Data using Stochastic Complexity*, in Proceedings of Machine Learning Research, vol. 89, PMLR, 2019, pp. 496–505.

[17] J. Nickolls et al., *Scalable Parallel Programming with CUDA*, Queue, 6 (2008), pp. 40–53.

[18] J. Pearl, *Causality: Models, Reasoning, and Inference*, Cambridge University Press, USA, 2nd ed., 2009.

[19] D. B. Rubin, *The design versus the analysis of observational studies for causal effects: Parallels with the design of randomized trials*, Statistics in Medicine, 26 (2007), pp. 20–36.

[20] C. Schmidt et al., *Order-independent Constraint-based Causal Structure Learning for Gaussian Distribution Models Using GPUs*, in Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM '18, New York, NY, USA, 2018, ACM, pp. 19:1–19:10.

[21] ———, *Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data*, in Proceedings of Machine Learning Research, vol. 104, Anchorage, Alaska, USA, dec 2019, PMLR, pp. 59–77.

[22] ———, *Out-of-Core GPU-Accelerated Causal Structure Learning*, in Algorithms and Architectures for Parallel Processing, Cham, 2020, Springer International Publishing, pp. 89–104.

[23] M. Scutari, *Learning Bayesian Networks with the bnlearn R Package*, Journal of Statistical Software, 35 (2010), pp. 1–22.

[24] ———, *Bayesian Network Repository*, 2012. http://www.bnlearn.com/bnrepository.

[25] ———, *Bayesian Network Constraint-Based Structure Learning Algorithms: Parallel and Optimized Implementations in the bnlearn R Package*, Journal of Statistical Software, Articles, 77 (2017), pp. 1–20.

[26] P. Spirtes, *Introduction to Causal Inference*, Journal of Machine Learning Research, 11 (2010), pp. 1643–1662.

[27] P. Spirtes et al., *Causation, Prediction, and Search, Second Edition*, Adaptive Computation and Machine Learning, MIT Press, Cambridge, MA, USA, 2000.

[28] I. Tsamardinos and G. Borboudakis, *Permutation Testing Improves Bayesian Network Learning*, in Machine Learning and Knowledge Discovery in Databases, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 322–337.

[29] B. Zarebavani et al., *cuPC: CUDA-Based parallel PC algorithm for causal structure learning on GPU*, IEEE Transactions on Parallel and Distributed Systems, 31 (2020), pp. 530–542.