GPU Acceleration for Information-theoretic Constraint-based Causal Discovery

Christopher Hagedorn christopher.hagedorn@hpi.de
Constantin Lange constantin.lange@student.hpi.de
Johannes Huegle johannes.huegle@hpi.de
Rainer Schlosser rainer.schlosser@hpi.de
Enterprise Platform and Integration Concepts
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany

Editor: Thuc Duy Le, Lin Liu, Emre Kıcıman, Sofia Triantafyllou, and Huan Liu

Abstract

The discovery of causal relationships from observational data is an omnipresent task in data science. In real-world scenarios, observational data is often high-dimensional, and functional causal relationships can be nonlinear. To handle nonlinear relationships within constraint-based causal discovery, appropriate conditional independence tests (CI-tests) become necessary, e.g., non-parametric information-theory-based CI-tests. Both high-dimensional data and CI-tests for nonlinear relationships pose computational challenges.

Existing work proposes parallel processing on Graphics Processing Units (GPUs) to address the computational demand resulting from high-dimensional data, in the case of discrete data or linear relationships. We extend this idea to cover CI-tests for nonlinear relationships in our work. Therefore, we develop $\texttt{GPU}_{\texttt{CMIknn}}$, a GPU-accelerated version of an existing CI-test, which builds upon conditional mutual information (CMI) combined with a local permutation scheme. Further, we propose a version of the PC algorithm, called $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, to process multiple instances of $\texttt{GPU}_{\texttt{CMIknn}}$ on the GPU in parallel.

Experiments show that the performance of $\texttt{GPU}_{\texttt{CMIknn}}$ is mainly affected by the number of k-nearest-neighbors (knn) within the CMI estimation. Depending on the chosen number of knn, the achieved speedup of $\texttt{GPU}_{\texttt{CMIknn}}$ ranges between factors of 2.3 to 352. In causal discovery, our method $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ outperforms a single-threaded CPU version by factors of up to 1 000, a multi-threaded CPU version using eight cores by factors of up to 240, and a naive GPU version by up to a factor 3.

Keywords: GPU Acceleration, PC Algorithm, Conditional Independence Testing, High-dimensional Data

1. Introduction

The discovery of causal structures from observational data is of relevance in many domains (Spirtes et al., 2000; Rau et al., 2013; Huegle et al., 2020; Hagedorn et al., 2022), in particular when randomized control trials are not feasible due to costs, ethics, or complexity (Rubin, 2007). In real-world scenarios, frequently, observational data sets have neither purely discrete nor continuous variables but contain a mixture of discrete-continuous variables or have non-linear relationships (Malinsky and Danks, 2018). Furthermore, data

sets are often high-dimensional, leading to long execution times for causal discovery algorithms (Le et al., 2019).

Recent advances in the field of constraint-based causal structure learning (CSL) propose non-parametric conditional independence tests (CI-tests) based on information theory to handle non-linear relationships (Runge, 2018) or mixed discrete-continuous data (Huegle, 2021). These CI-tests build upon the estimation of mutual information (MI), respectively conditional mutual information (CMI) using k-nearest-neighbor (knn) estimators, e.g., see Frenzel and Pompe (2007); Vejmelka and Paluš (2008); Gao et al. (2017); Mesner and Shalizi (2021) and combine the estimator with a local permutation scheme to generate the null distribution. The computational complexity of such CI-tests is much higher, i.e., $O(n^2)$ (Runge, 2018), compared to well-known CI-tests, e.g., Fisher's z-test (Fisher, 1915) or Pearson's χ^2 test (Pearson, 1900) for continuous or discrete data. Consequently, the runtime of the individual CI-test and hence runtime of constraint-based CSL algorithms, such as the well-known PC algorithm (Spirtes et al., 2000), increase with higher computational complexity.

In this context, versions of the PC algorithm for highly parallel execution on Graphics Processing Units (GPUs) are proposed, which achieve speed-up of orders of magnitude over CPU-based approaches (Schmidt et al., 2018; Zarebavani et al., 2020; Hagedorn and Huegle, 2021a). The GPU-accelerated versions are tailored to use of common CI-tests for continuous (Schmidt et al., 2018; Zarebavani et al., 2020) or discrete data (Hagedorn and Huegle, 2021a). Each version leverages unique CI-test characteristics to achieve speed-up on a GPU. Therefore, the approaches are not easily transferred to CI-tests based on information theory.

This paper addresses the high runtime of the existing information-theoretic CI-test `CMIknn` (Runge, 2018) by employing a GPU as an execution device. Our GPU-accelerated, CUDA-based version $\texttt{GPU}_{\texttt{CMIknn}}$ computes the local permutation scheme and estimates the CMI value in separate CUDA kernels. The permutation kernel of $\texttt{GPU}_{\texttt{CMIknn}}$ leverages GPU thread block scheduling (Hennessy and Patterson, 2017) to create a random order for the permutation scheme by processing observations within multiple warps over multiple thread blocks. In the CMI estimation kernel, we apply ideas of pipelined execution (Funke et al., 2018) to utilize per-thread local memory during knn computation following a brute force approach. While generally, knn estimation using a kd-tree has lower computational complexity than using a brute force approach, the brute force approach benefits from the parallel computing capabilities of the GPU. Therefore, our proposed GPU-accelerated version implementing a brute force approach is well suited for CMI estimators requiring a smaller k, such as, $k \leq 200$. Note that the optimal choice of the parameter k within CMI estimation depends on the employed estimator. For example, for `CMIknn`, hence for $\texttt{GPU}_{\texttt{CMIknn}}$, it is suggested to set $k = 0.2 \times n$, where n is the number of observations (Runge, 2018).

We integrate $\texttt{GPU}_{\texttt{CMIknn}}$ into a custom version of the PC algorithm, which allows for parallel execution of multiple CI-tests on the GPU. Therefore, we implement parallel versions of the permutation kernel and the CMI kernel. Additionally, we adopt an execution order of CI-tests that allows reusing computed local permutations across multiple CI-tests, reducing the runtime.

We evaluate $\texttt{GPU}_{\texttt{CMIknn}}$ and our proposed version of the PC algorithm $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ in an experimental setting on synthetic data. Therefore, we consider the execution of a sin-

gle CI-test and investigate the performance regarding relevant parameters, e.g., number of observations n, number of permutations $perm$, or number of k_{CMI} nearest neighbors within CMI estimation. Second, we evaluate the performance of $\texttt{GPU}_{\texttt{CMIknn}}$ in the context of the PC algorithm. In both cases, we compare with the existing CPU-based version `CMIknn` (Runge, 2018). We find that the performance of our GPU-based version of the CI-test is mainly dominated by choice of k_{CMI}, the number of knn during CMI estimation. For values of $k_{CMI} \leq 200$, the $\texttt{GPU}_{\texttt{CMIknn}}$ outperforms its CPU counterpart by up to two orders of magnitude, mainly when keeping $k_{CMI} \leq 20$. Under the assumption of small values for k_{CMI}, $\texttt{GPU}_{\texttt{CMIknn}}$ scales well within all other considered dimensions, i.e., number of samples n, number of permutations $perm$, or size of separation set $|S|$. In the context of the PC algorithm, we observe similar behavior concerning the parameter k_{CMI}. $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` provides additional speed-up of almost a factor of 3 over a baseline non-parallel version employing $\texttt{GPU}_{\texttt{CMIknn}}$. Compared to the parallel execution of `CMIknn` on the CPU on eight cores, $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` remains faster by factors of up to 250 for $k_{CMI} = 7$ but performs similarly for $k_{CMI} = 200$.

The remainder of the paper is structured as follows: Section 2 provides background on constraint-based CSL, the concepts of information-theoretic CI-tests, and GPUs. In Section 3, we discuss existing work on GPU acceleration for CSL. We introduce our GPU-accelerated CI-test $\texttt{GPU}_{\texttt{CMIknn}}$ and its parallel implementation within the PC algorithm in Section 4. Section 5 provides an experimental evaluation of our proposed method and discusses our results. Finally, in Section 6, we summarize our work.

2. Preliminaries

This section introduces terminology in the context of constraint-based CSL, the PC algorithm, and information-theoretic-based CI-tests. Also, we outline necessary GPU hardware and execution concepts together with the associated programming model.

2.1 Constraint-based Causal Structure Learning

Within our work, we follow the well-known theory of Causal Graphical Models (CGMs) and CSL, cf. (Spirtes et al., 2000; Peters et al., 2017; Pearl, 2009). Thus, a Directed Acyclic Graph (DAG) $\mathcal{G}$ is defined by $\mathcal{G} = (\mathbf{V}, \mathbf{E})$, with N random variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ and an edge set $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$, that contains only directed edges $V_i \to V_j$ such that $\mathcal{G}$ contains no cycles. The combination of a DAG $\mathcal{G}$ and a joint probability distribution over the variables $\mathbf{V}$ defines a CGM (Pearl, 2009). Thus, a directed edge $V_i \to V_j$ in $\mathcal{G}$ represents a direct causal relationship. Further, the DAG $\mathcal{G}$ entails information about the conditional independence of the variables via the d-separation criterion. Hence, two variables $V_i, V_j \in \mathbf{V}$ are conditionally independent given a set of variables $S^{i,j} \subset \mathbf{V} \setminus \{V_i, V_j\}$ if and only if the variables V_i and V_i are d-separated by the set $S^{i,j}$.

In this context, CSL is the search for the underlying causal relationships described by $\mathcal{G}$ from i.i.d. observational data D with n samples of N variables. Thereby, methods of CSL leverage the conditional independence characteristics of the joint probability distribution over the variables, which allows estimation of the causal structures up to the Markov equivalence class of $\mathcal{G}$, which can be represented by a Complete Partially Directed Acyclic Graph (CPDAG) (Chickering, 2002). Therefore, methods of constraint-based CSL apply CI-tests

to discover the causal structures of the CGM. Note that the appropriate CI-test for a given datasset is directly determined by the datasset's probability distribution (Dawid, 1979). An efficient method for constraint-based CSL is the PC algorithm (Spirtes et al., 2000). The PC algorithm has an adjacency search and an edge orientation. The adjacency search determines the skeleton graph $\mathcal{C}$ of $\mathcal{G}$, in which all edges $E^{i,j} \in \mathbf{E}$, with $E^{i,j} = (V_i, V_j)$ are undirected. The adjacency search is an iterative algorithm running from level $l = 0$ up to $l = \max_{V_i \in \mathbf{V}} |adj(\mathcal{G}, V_i)| - 1$, where $adj(\mathcal{G}, V_i)$ returns all adjacent variables of V_i in $\mathcal{G}$. The algorithm starts with a fully connected skeleton $\mathcal{C}^0$ in level $l = 0$. Within each level, the algorithm performs CI-tests for all remaining edges $E^{i,j}$ in $\mathcal{C}^l$, for which at least one separation set $S^{i,j}$ of size l can be constructed. A separation set $S^{i,j}$ contains a combination of adjacent variables from V_i, i.e., $S^{i,j}$ that is drawn from $adj(\mathcal{C}^l, V_i) \setminus \{V_j\}$. The algorithm iteratively processes all possible separation sets $\mathbf{S}^{i,j} = \{S_1^{i,j}, \ldots, S_M^{i,j}\}$, where $M = \binom{|adj(\mathcal{C}^l, V_i) \setminus \{V_j\}|}{l}$, where the size l equals to the current level. In case the p-value computed by the CI-test $CI(V_i, V_j | S^{i,j})$ is larger than the provided significance level α, the corresponding edge $E^{i,j}$is removed from $\mathcal{C}^l$. Additionally, the corresponding separation set $S^{i,j}$ is stored, and no additional CI-tests are conducted for this edge. Hence, this inner loop can be terminated early. Further, the edge $E^{i,j}$ is not considered at any higher level. Once all edges within the current level have been processed, the algorithm continues with the subsequent level. Upon reaching the maximum level, the adjacency search returns the estimated skeleton $\mathcal{C}$ and the stored separation sets $SepSet$.

Within the edge orientation, as many edges as possible are oriented based upon deterministic rules (Spirtes et al., 2000; Colombo and Maathuis, 2014). The adjacency search constitutes the majority of the computation of the PC algorithm, given its computational complexity, which is exponential to the number of variables N in the worst case and remains polynomial even for sparse graphs (Kalisch and Bühlmann, 2007). Therefore, CPU-based parallel extensions (Le et al., 2019; Schmidt et al., 2019; Scutari, 2017) and GPU-accelerated versions (Schmidt et al., 2018; Zarebavani et al., 2020; Hagedorn and Huegle, 2021a) of the PC algorithm, focus on improving the execution time of the adjacency search.

2.2 Information-theoretic Conditional Independence Tests

In information theory, measures, such as mutual information (MI), provide a means to analyze the information flow between two systems (Hlaváčková-Schindler et al., 2007). Thus, the concept of MI encodes the shared information for two random variables V_i, V_j, and is the basis for CI-testing (Bishop, 2006). Utilizing the MI for CI-tests has two challenges. First, MI has to be estimated from the observational data, depending on the underlying data distribution (Gao et al., 2017). Second, to use MI within a CI-test requires determining a null distribution. A recently proposed information-theoretic CI-test (Runge, 2018) employs k-nearest neighbor estimation (Frenzel and Pompe, 2007; Vejmelka and Paluš, 2008) to estimate MI, respectively CMI and utilizes a local permutation-scheme (Doran et al., 2014) to determine the null distribution. The CI-test is sketched in Algorithm 1.

The algorithm starts by computing for each sample a with $a \in \{1, \ldots, n\}$ a list of nearest neighbors $knn[a]$ of size k_{perm} with $0 < k_{perm} < n$. The nearest neighbors are determined according to the distance in the dimension of the separation set $S^{i,j}$ of the current sample to all other samples of the observation data (cf. Algorithm 1 lines 1–3).

Algorithm 1 CI-test based on nearest-neighbor permutation (Runge, 2018)
Input: number of permutations $perm$, k-nearest neighbors within permutation k_{perm},k-nearest neighbors within CMI estimation k_{CMI}, observational data D, variables V_i, V_j, separation set $S^{i,j}$, number of samples n, estimator function $estimator_{knn}()$
Output: p-value p, test statistic cmi

1: **for all** $a \in \{1, \ldots, n\}$ **do**
2: $knn[a] = k\_nearest\_neighbors(k_{perm}, a, D[S^{i,j}], n)$
3: **end for**
4: **for all** $m \in \{1, \ldots, perm\}$ **do**
5: **for all** $a \in \{1, \ldots, n\}$ **do**
6: Shuffle list $knn[a]$
7: **end for**
8: Initialize empty set $used$
9: $\hat{D} = \{\}$
10: $ord = create\_random\_order(\{1, \ldots, n\})$
11: **for all** $a \in ord$ **do**
12: $x = knn[a][0]$
13: $y = 0$
14: **while** $x \in used$ & $y < k_{perm} - 1$ **do**
15: $y = y + 1$
16: $x = knn[a][y]$
17: **end while**
18: $\hat{D}[a] = D[V_i][x]$
19: $used.add(x)$
20: **end for**
21: $\hat{cmi}[m] = estimator_{knn}(\hat{D}, D[V_j], D[S^{i,j}], k_{CMI})$
22: **end for**
23: $cmi = estimator_{knn}(D[V_i], D[V_j], D[S^{i,j}], k_{CMI})$
24: $\mathrm{p} = \frac{1}{perm} \sum_{m=1}^{perm} \mathbf{1}(cmi \leq \hat{cmi}[m])$

Next, for each permutation m with $m \in \{1, \ldots, perm\}$ the values of V_i are locally permuted according to the lists of nearest neighbors $knn[a]$ with $a \in \{1, \ldots, n\}$. In this step, first, each list of nearest neighbors $knn[a]$ is shuffled, an empty set for used elements $used$ is initialized, a vector of size n for the permuted values $\hat{D}$ is initialized, and a random order is created for the n samples (cf Algorithm 1 lines 5–10). Next, the samples are iterated in the previously determined order (cf. Algorithm 1 lines 11–18). For each sample, one of its nearest neighbors is drawn from the list and placed at the sample's position in the local permutation $\hat{D}$. The drawing mechanism chooses the nearest neighbor of the current sample that a previous samples has not drawn unless it is the k^{th}-nearest neighbor. This restriction is achieved as drawn neighbors are added to the set for used elements $used$. Once the local permutation of V_i is generated, it is used to compute the CMI based on the estimator function $estimator_{knn}()$ (cf. Algorithm 1 line 21). The calculated CMI value is stored for each permutation in a list $\hat{cmi}$.

After the CMI values for all permutations are computed, the CMI value from original non-permuted samples cmi is computed (cf. Algorithm 1 line 23). Finally, the p-value p is computed as the average of the indicator function, evaluating if cmi is less or equal to the permuted CMI values $\hat{cmi}[m]$ over all permutations $m \in \{1, \ldots, perm\}$. The algorithm returns the p-value p and cmi as the test statistic.

The algorithmic template of the CI-test sketched in Algorithm 1 allows substituting the estimator. Without additional changes to the algorithm, estimators suitable for specific data characteristics can be plugged-in. For example, in the original version of the CI-test, a k-nearest neighbor estimator suitable for continuous time series data with non-linear relationships (Frenzel and Pompe, 2007) is used. In recent work, Huegle (2021) suggests employing a k-nearest neighbor estimator proposed by Gao et al. (2017) for mixed discrete-continuous data.

2.3 Graphics Processing Units

GPUs have seen a growing interest as dedicated processing units to accelerate machine learning workloads (LeCun et al., 2015). These machine workloads benefit from the ample parallel computing capabilities and specific hardware features of GPUs.

GPU hardware characteristics A GPU provides global and shared memory and has its cache hierarchy and dozens of streaming multiprocessors (SMs), with individual processing cores each. Within current GPU generations, the global memory has a capacity of up to 80 GB (Choquette et al., 2021). Data structures in global memory can be accessed by threads placed on cores across all SMs. In contrast, shared memory has a limited capacity of only up to 192 KB. Yet, it has a higher memory bandwidth than global memory. Further, data structures in shared memory are only accessible by threads placed on cores within the same SM.

GPU execution concept The GPU is a throughput-oriented device (Kirk and Hwu, 2013) that follows the Single Instruction Multiple Threads (SIMT) execution model (Lindholm et al., 2008), meaning that processing threads are grouped and execute the same scheduled instruction in lockstep. Each thread operates on a dedicated core. Commonly, 32 threads are grouped, referred to as a warp.

Programming model Programming frameworks, such as CUDA (Nickolls et al., 2008), enable efficient development for GPUs. The functions executed on a GPU are organized in kernels within CUDA code. Each kernel is launched with several CUDA threads organized in thread blocks that execute the code. Each thread block is mapped to one SM during execution, enabling access to shared memory and providing fast synchronization mechanisms for all threads within the same thread block. The number of threads and thread blocks are specified in three dimensions upon launch of the kernel. Hence, each thread has its three-dimensional ids within the kernel, i.e., `threadIdx.(x,y,z)` and `blockIdx.(x,y,z)`, abbreviated with tx, ty, tz or bx, by, bz. Furthermore, programmers can explicitly move data structures to global and shared memory for efficient memory management.

3. Related Work

We first discuss existing work on parallel execution of the PC algorithm, focusing on GPU acceleration. Second, as the information-theoretic CI-test `CMIknn` (Runge, 2018) builds upon knn estimation, we consider existing GPU-based knn searches.

3.1 Parallel Approaches of PC Algorithm

Several versions of the PC algorithm have been proposed that leverage parallel computing capabilities of modern hardware, such as multi-core CPUs (Le et al., 2019; Madsen et al., 2015; Schmidt et al., 2019; Scutari, 2017), GPUs (Hagedorn and Huegle, 2021a b; Schmidt et al., 2018, 2020; Zarebavani et al., 2020), or FPGAs (Guo and Luk, 2022), to address the computational demand of causal discovery. Most work on parallel execution on multi-core CPUs parallelizes over the edges within the CGM (Le et al., 2019; Madsen et al., 2015; Schmidt et al., 2019; Scutari, 2017). Therefore, the proposed approaches remain independent from the applied CI-test and thus can be directly applied to information-theoretic CI-tests.

In the context of GPU acceleration of the PC algorithm, such a universal approach is not feasible, as GPU hardware characteristics and the execution concept have to be considered. Thus, existing GPU-based approaches leverage CI-test characteristics to achieve speed-up and provide individual CI-test implementations targeting a specific data distribution. Also, the GPU-based algorithms provide unique kernels corresponding to specific levels l of the PC algorithm. In the case of continuous data, Schmidt et al. (2018) take advantage of pre-calculated correlation coefficients to avoid access to samples within each CI-test. Further, they compute batches of CI-test for a single edge, i.e., considering different separation sets, in parallel within the same thread block. In `cupc` (Zarebavani et al., 2020), intermediate results of CI-tests for continuous data with the same separation set are shared. In detail, `cupc` reuses computed inverse matrices required for the CI-test for continuous data. In both GPU-based versions for continuous data (Schmidt et al., 2018; Zarebavani et al., 2020) CI-tests are parallelized over the GPU threads. In contrast, in the case of discrete data, Hagedorn and Huegle (2021a) propose parallelizing the processing of individual samples within CI-tests. The authors reflect the requirement of the implemented Pearson's χ^2 test (Pearson, 1900) to compute the marginals over contingency tables from the samples for each CI-test separately.

Given that the existing GPU-based approaches are tailored to the CI-test characteristics, direct transfer to the case of an information-theoretic CI-test is not an option. Therefore, a GPU-accelerated information-theoretic CI-test requires a unique GPU-based implementation of the CI-test that is tailored to the SIMT execution model and considers the GPU memory constraints. Second, in the context of execution within the PC algorithm, a parallel execution scheme is required that takes full advantage of the parallel processing capabilities of the GPU, i.e., by processing multiple CI-tests or edges in parallel. Therefore, we propose $\texttt{GPU}_{\texttt{CMIknn}}$, a GPU-based information-theoretic CI-test, and provide an adapted version of the PC algorithm for parallel execution of multiple CI-tests.

3.2 GPU-based Approaches to knn Estimation

The CPU-based version `CMIknn` (Runge, 2018) implements the knn estimation using kd-search trees (Bentley, 1975; Friedman et al., 1977). Kd-trees are a computationally efficient option for the knn estimation, given that their computational complexity is in average $O(n \times log(n))$ when searching over all n samples. In contrast, a brute-force approach to solving the knn estimation has a complexity of $O(n^2)$. GPU-based implementations exist for knn estimation using kd-search (Garcia et al., 2010) and for knn estimation built upon brute force approaches (Gieseke et al., 2014; RAPIDS Development Team, 2018).

Commonly, performing knn searches using kd-search trees on GPU may result in poor performance due to branching and memory access inapt for GPU hardware. Gieseke et al. (2014) propose a buffer kd-tree that addresses these shortcomings. The buffer kd-tree consists of one top tree with a small height of, e.g., $h = 8$, leaf structures and buffers for each leaf of the top tree. When querying the buffer kd-tree the buffers are filled with query indices processed upon reaching a threshold. During the processing of the buffers the k nearest neighbors are determined for each query index within each buffer in parallel, using one GPU thread each. The GPU thread conducts a brute force search within the leaf structure corresponding to the query index buffer. Note the initial construction of the kd-tree and orchestration occurs on the CPU. A huge amount of queries is needed for the buffer kd-tree to be efficient (Gieseke et al., 2014).

The brute-force approach is well suited for the execution model of a GPU. (Garcia et al., 2010) propose a GPU-accelerated implementation of the brute-force approach that outperforms CPU-based versions by up to two orders of magnitude. Their approach implements two separate GPU kernels. The first kernel computes a distance matrix of size $n \times n$ between all n samples. Given that these computations are independent, the problem is embarrassingly parallel. The second kernel sorts the distances in parallel for each sample.

Despite the lower computational complexity, the construction and search of kd-trees remain challenging on the GPU. The buffer kd-tree (Gieseke et al., 2014) requires many queries to become an efficient option. Furthermore, the kd-tree requires additional storage from the limited GPU memory. Therefore, we chose a brute force approach to knn estimation within $\texttt{GPU}_{\texttt{CMIknn}}$. Thus, we build upon the general idea of Garcia et al. (2010) but make adaptions suited to our use case. In particular, we apply a pipeline execution approach (Funke et al., 2018) to keep intermediate results, e.g., the k nearest neighbors, in GPU thread local memory and avoid storing additional data structures, such as large distance matrices.

4. GPU-Accelerated Causal Discovery using Information-theoretic CI-test

This section presents $\texttt{GPU}_{\texttt{CMIknn}}$ [1], a GPU-accelerated implementation of the information-theoretic CI-test $_{\text{CMIknn}}$ (Runge, 2018). Therefore, we sketch the algorithm and provide detail on two GPU kernel implementations (see Section 4.1). The first kernel computes the local permutation by applying the nearest neighbor search on the GPU. The second kernel computes the CMI estimates based on the permuted values, again using nearest

1. Code available on GitHub: https://github.com/ChristopherSchmidt89/gpucmiknn/

Algorithm 2 $\texttt{GPU}_{\texttt{CMIknn}}$
Input: number of permutations $perm$, k-nearest neighbors within permutation k_{perm}, k-nearest neighbors within CMI estimation k_{CMI}, observational data D, variables V_i, V_j, separation set $S^{i,j}$, number of samples n, knn-based CMI estimation kernel $estimate_{CMIknn}()$
Output: p-value p

1: transferToGPU($D[V_i], D[V_j], D[S^{i,j}]$)
2: allocateOnGPU($\hat{V}_i[perm][n], used[perm][n], partial_{cmi}[perm+1]$)
3: launchOnGPU($localPermutation, \{D[S^{i,j}], D[V_i], \hat{V}_i, used, n, perm, k_{perm}\}$)
4: launchOnGPU($estimate_{CMIknn}, \{D[V_i], D[V_j], D[S^{i,j}], \hat{V}_i, partial_{cmi}, n, k_{CMI}\}$)
5: transferFromGPU($partial_{cmi}$)
6: $base_{cmi} = \digamma(k_{CMI}) - \frac{partial_{cmi}[0]}{n}$
7: $c = 0$
8: **for all** $a \in \{1, \ldots, perm\}$ **do**
9: **if** $(\digamma(k_{CMI}) - \frac{partial_{cmi}[a]}{n}) \geq base_{cmi}$ **then**
10: $c = c + 1$
11: **end if**
12: **end for**
13: p = $\frac{c}{perm}$
14: **return** p

neighbor searches. Additionally, we present a GPU-accelerated version of the PC algorithm that employs an extended version of $\texttt{GPU}_{\texttt{CMIknn}}$, in which the kernel implementations allow computing multiple CI-tests in parallel (see Section 4.2). Note the proposed algorithms allow exchanging the CMI-estimator, assuming that the CMI-estimator builds upon knn searches, e.g., see Mesner and Shalizi (2021).

4.1 GPU-Accelerated Information-theoretic CI-test: $\texttt{GPU}_{\texttt{CMIknn}}$

$\texttt{GPU}_{\texttt{CMIknn}}$ uses the GPU to accelerate the computation of the local permutation and the CMI estimate. In both computations, the knn are estimated. Despite a higher computational complexity than Kd-trees, $\texttt{GPU}_{\texttt{CMIknn}}$ implements a brute-force approach to estimate the knn, as the brute-force approach is well-suited for parallel execution on the GPU (see Section 3.2). Thus, $\texttt{GPU}_{\texttt{CMIknn}}$ parallelizes over the samples, or the samples and permutations, respectively. In both cases, each GPU thread computes the knn for one sample, respectively, one sample or permutation. During computation of the knn, $\texttt{GPU}_{\texttt{CMIknn}}$ aims to keep all intermediate data in GPU thread local memory fo efficient execution. Therefore, $\texttt{GPU}_{\texttt{CMIknn}}$ works best for small values of k_{perm} and k_{CMI}.

We sketch the overall idea of our GPU-based implementation in Algorithm 2 $\texttt{GPU}_{\texttt{CMIknn}}$ receives a series of input parameters to conduct the CI-test and outputs the p-value p. As input, the algorithm takes the observational data D, and indices of the variables V_i,V_j, and $S^{i,j}$ that point to the corresponding samples within D. Furthermore, $\texttt{GPU}_{\texttt{CMIknn}}$ requires the following input parameters: $perm$, the number of permutations, k_{perm} the number of k-nearest neighbors during local permutation, k_{CMI} the number of k-nearest neighbors during CMI estimation, n the number of samples and $estimate_{CMIknn}()$ kernel for CMI estimation.

In the beginning, the algorithm prepares the data on the GPU. Therefore, the algorithm transfers the observational data of the variables V_i,V_j, and $S^{i,j}$ to the GPU and allocates memory for the permutations of V_i $\hat{V}_i$, the intermediate CMI values $partial_{cmi}$, and the auxiliary matrix *used* (see Algorithm 2 lines 1 – 2). Next, the *localPermutation* kernel is launched on the GPU, which computes $\hat{V}_i$. After completion of the *localPermutation* kernel, the $estimate_{CMIknn}$ kernel is launched, which calculates $perm + 1$ intermediate values for the CMI stored in $partial_{cmi}$. After transfer of $partial_{cmi}$ from the GPU (see Algorithm 2 line 5), the final CMI values are determined. The applied calculation corresponds to the implementation of the $estimate_{CMIknn}()$ kernel. In Algorithm 2 and in our reference implementation, we follow the approach of `CMIknn` (Runge, 2018) that builds upon the CMI estimator by Frenzel and Pompe (2007); Vejmelka and Paluš (2008). Thus, the calculation uses the following equation:

$$cmi = \digamma(k_{CMI}) - \frac{partial_{cmi}}{n}, \tag{1}$$

where $partial_{cmi}$ is based upon the counts of points within the subspaces $\mathcal{V}_i \otimes \mathcal{S}^{i,j}$, $\mathcal{V}_j \otimes \mathcal{S}^{i,j}$ and $\mathcal{S}^{i,j}$ that are within the distances of the k-nearest neighbor taken from the joint space $\mathcal{V}_i \otimes \mathcal{V}_j \otimes \mathcal{S}^{i,j}$ (cf. Equation 5 in Runge (2018)). At first, the $base_{CMI}$ is computed based on the non-permuted case V_i,V_j,$S^{i,j}$. Next, the CMI is computed for all permutations $\{1, \ldots, perm\}$. Within the same loop, the algorithm checks if the CMI for a permutation is larger or equal than the $base_{CMI}$ and increments a counter c. Finally, the actual p-value p is computed as the sum of the indicator function over the number of permutations (see Algorithm 2 line 13). In the following, we provide detail on the *localPermutation* kernel and the implemented $estimate_{CMIknn}$ kernel.

Local Permutation Kernel The *localPermutation* kernel (see Algorithm 3 below) takes several parameters and pointer to data structures as input (see Algorithm 3 **Input**). The kernel does not return a specific result but places the *perm* local permutations of V_i in the data structure $\hat{V}_i$, which remain on GPU for further processing. The kernel is launched with β threads per thread block and $\lceil \frac{n}{\beta} \rceil$ thread blocks. The parameter β should ideally be chosen as a multiple of the GPU warp size, i.e., 32, and not exceed hardware constraints, i.e., 1024. As a default, we set $\beta = 32$. Further, upon kernel launch, shared memory of size $\beta \times \text{DIMENSION}(S^{i,j}) \times sizeof(float)$ bytes is reserved for each thread block. Note the function DIMENSION() returns the size of the separation set, i.e., the number of variables contained within $S^{i,j}$. The function $sizeof()$ returns the size of the input data type in byte. Once the kernel is launched, each GPU thread processes lines 1–29 of Algorithm 3. Given that the kernel is launched with at least n GPU threads, each thread is responsible for processing one of the n samples and computing the corresponding local permutations of its sample.

At first, two arrays of size k_{perm}, namely *sDist* and *sPos* are allocated and initialized in GPU thread local memory. These arrays will eventually store the distances and positions of the k-nearest neighbors. Further, each GPU thread loads the values from $D[S^{i,j}]$ corresponding to its sample, calculated based on the GPU thread's thread id and block id.

After this setup, all n samples are iterated in a stride of size β. The following steps are executed within this loop over the n samples. First, the values from $D[S^{i,j}]$, which remain in global memory, are loaded into shared memory (see line 5 in Algorithm 3). After

Algorithm 3 Local permutation kernel within $\texttt{GPU}_{\texttt{CMIknn}}$
Input: samples $D[S^{i,j}]$ and $D[V_i]$, data structure for permutations $\hat{V}_i$, auxiliary data structure *used*, number of samples n, number of permutations $perm$, k-nearest neighbors within permutation k_{perm}
# of blocks: $\lceil \frac{n}{\beta} \rceil$
# of threads per block: β
Shared memory: $\beta \times$ DIMENSION$(S^{i,j}) \times$ SIZEOF$(float)$

1: Initialize $sDist[k_{perm}]$ with BIG_FLOAT, $sPos[k_{perm}]$ with 0 in thread local memory
2: Initialize myS of size DIMENSION$(S^{i,j})$ in thread local memory
3: Set $myS = D[S^{i,j}][bx \times \beta + tx]$
4: **for all** $a \in \{0, \ldots, (\lceil \frac{n}{\beta} \rceil - 1)\}$ **do**
5: $S_{shared}[tx] = D[S^{i,j}][a \times \beta + tx]$ in shared memory
6: SYNCTHREADS()
7: **for all** $b \in \{0, \ldots, (\beta - 1)\}$ **do**
8: **if** $a \times \beta + b == bx \times \beta + tx$ **then**
9: continue
10: **end if**
11: $dist =$DISTMETRIC$(myS, S_{shared}[b])$
12: **if** $dist$ is smaller than any $c \in sDist$ **then**
13: Insert $dist$ in sorted order into $sDist$
14: Insert position $a \times \beta + b$ in sorted order into $sPos$
15: **end if**
16: **end for**
17: **end for**
18: CURAND_INIT()
19: **for all** $c \in \{0, \ldots, (perm - 1)\}$ **do**
20: **for all** $d \in \{(k_{perm} - 1), \ldots, 1\}$ **do**
21: $pos_{shuffled} =$ CURAND() $\bmod (d + 1)$
22: SWAP$(sPos[d], sPos[pos_{shuffled}])$
23: **end for**
24: $u = 0$
25: **while** ATOMICCAS$(used[c \times n + sPos[u]], 0, 1) \neq 0$ and $u < k_{perm} - 1$ **do**
26: $u = u + 1$
27: **end while**
28: $\hat{V}_i[c \times n + bx \times \beta + tx] = D[V_i][sPos[u]]$
29: **end for**

synchronizing the threads within the same thread block, the stride of values from $D[S^{i,j}]$ stored in shared memory is processed iteratively (see Algorithm 3 lines 7 – 16). If a value selected from the current stride corresponds to the GPU thread's sample, the iteration is skipped. Otherwise, the distance $dist$ is computed between myS, the GPU thread's sample, and $S_{shared}[b]$, the sample from the current iteration within the current stride. A distance function DISTMETRIC is used to compute $dist$. As a default $\texttt{GPU}_{\texttt{CMIknn}}$ computes the Chebyshev distance. Based upon the computed distance $dist$, the local arrays $sDist$

and $sPos$ are updated. In case the value of $dist$ is smaller than any element in $sDist$, $dist$ is inserted into $sDist$ at the position that keeps $sDist$ in order. The remaining elements are shifted accordingly, and the entry with the largest distance is removed from $sDist$. Accordingly, the array $sPos$ is updated, storing the positions of the corresponding samples in $D[S^{i,j}]$. After both loops have been executed, the k_{perm}-nearest neighbors are determined, and their positions are stored in $sPos$.

Next, the permutations $\hat{V}_i$ are computed. This step builds upon using a random number generator, e.g., from NVIDIA's $cuRAND$ library. After initializing the random number generator, each GPU thread computes the permutations corresponding to its sample (see Algorithm 3 lines 18–29). Thus, for each permutation, the following steps are executed. First, the positions within $sPos$ are randomly shuffled (see Algorithm 3 lines 20–23). Next, positions from $sPos$ are drawn until either no other GPU thread has drawn the same position (from $\{0, \ldots, n-1\}$) before, or it is the last position in $sPos$. To ensure that no other GPU thread has drawn the same position before, an atomic compare and swap operation ATOMICCAS is performed on the $used$ data structure (see Algorithm 3 line 25). Finally, the selected position from $sPos$ is used to retrieve the value from $D[V_i]$, which is used within the current permutation at the GPU thread's corresponding position (see Algorithm 3 line 28). Once all GPU threads have terminated, the data structure $\hat{V}_i$ contains the local permutations of V_i according to the k_{perm}-nearest neighbors within the $\mathcal{S}^{i,j}$ space.

CMI Estimation Kernel The $estimate_{CMIknn}$ kernel (see Algorithm 4) takes several parameters and pointer to data structures as input (see Algorithm 4 **Input**). The kernel computes partial CMI values for each permutation, stored in the list $partial_{cmi}$ on GPU upon termination. The kernel is launched with γ threads per thread block and $(perm + 1) \times \lceil \frac{n}{\beta} \rceil$ thread blocks. The parameter γ should ideally be chosen as a multiple of the GPU warp size, i.e., 32, and not exceed hardware constraints, i.e., 1024. Again, as a default, we set $\gamma = 32$. Further, upon kernel launch, shared memory of size $2 \times \gamma \times \text{DIMENSION}(S^{i,j}) \times sizeof(float)$ bytes is reserved for each thread block. Note the DIMENSION function returns the size of the separation set, i.e., the number of variables contained within $S^{i,j}$. Once the kernel is launched, each GPU thread processes lines 1–39 of Algorithm 4. Given that the kernel is launched with $n \times (perm+1)$ GPU threads, each thread is responsible for processing one of the n samples within one permutation and participates in the computation of the corresponding partial CMI value. Note that we increment $perm$ by one to handle the base CMI estimation from the non-permuted $D[V_i]$ values.

At first, the array $sDist$ of size k_{CMI} is allocated and initialized in GPU thread local memory. $sDist$ is used to store the k_{CMI}-nearest neighbors. Further, each GPU thread loads the values relevant to the sample it is processing, i.e., at position pos, into shared memory (see function LOADINTOSHARED in Algorithm 4 line 3). Apart from $D[V_j][pos]$ and $D[S^{i,j}][pos]$, the function LOADINTOSHARED either loads $D[V_i][pos]$ if $bx == 0$, or $\hat{V}_i[bx \times n + pos]$ into the shared memory $D_{shared}[tx]$.

Now, all n samples are iterated in a stride with a size corresponding to the number of threads per block γ. The following steps are executed within this loop over the n samples and compute the distance of the k_{CMI}-nearest neighbor. In the first step, values for the current stride of samples are loaded into shared memory $D_{shared}[\gamma + tx]$. The LOADINTOSHARED function is used to distinguish between values from $D[V_i]$ and its permutations $\hat{V}_i$. After

Algorithm 4 CMI estimation kernel within $\texttt{GPU}_{\texttt{CMIknn}}$
Input: samples $D[V_i]$, $D[V_j]$ and $D[S^{i,j}]$, permutations $\hat{V}_i$, data structure for partial cmi values $partial_{cmi}$, number of samples n, k-nearest neighbors within CMI estimation k_{CMI}
# of blocks: $(perm + 1) \times \lceil \frac{n}{\gamma} \rceil$
# of threads per block: γ
Shared memory: $2 \times \gamma \times (\text{DIMENSION}(S^{i,j}) + 2) \times \text{SIZEOF}(float)$

1: Initialize $sDist[k_{CMI}]$ with BIG_FLOAT in thread local memory
2: $pos = by \times \gamma + tx$
3: LOADINTOSHARED($D_{shared}[tx], bx, pos, \hat{V}_i, D[V_i], D[V_j], D[S^{i,j}]$)
4: SYNCTHREADS()
5: **for all** $a \in \{0, \ldots, (\lceil \frac{n}{\gamma} \rceil - 1)\}$ **do**
6: $pos_2 = a \times \gamma + tx$
7: LOADINTOSHARED($D_{shared}[\gamma + tx], bx, pos_2, \hat{V}_i, D[V_i], D[V_j], D[S^{i,j}]$)
8: SYNCTHREADS()
9: **for all** $b \in \{0, \ldots, (\gamma - 1)\}$ **do**
10: $dist =$DISTMETRIC($D_{shared}[tx], D_{shared}[\gamma + b]$)
11: **if** $dist$ is smaller than any $c \in sDist$ **then**
12: Insert $dist$ in sorted order into $sDist$
13: **end if**
14: **end for**
15: **end for**
16: Init counter $C_{S^{i,j}V_i}, C_{S^{i,j}V_j}, C_{S^{i,j}} = 0$
17: **for all** $a \in \{0, \ldots, (\lceil \frac{n}{\gamma} \rceil - 1)\}$ **do**
18: $pos_2 = a \times \gamma + tx$
19: LOADINTOSHARED($D_{shared}[\gamma + tx], bx, pos_2, \hat{V}_i, D[V_i], D[V_j], D[S^{i,j}]$)
20: SYNCTHREADS()
21: **for all** $b \in \{0, \ldots, (\gamma - 1)\}$ **do**
22: $dist =$DISTMETRIC($D_{shared}[tx][S^{i,j}], D_{shared}[\gamma + b][S^{i,j}]$)
23: UPDATECOUNTER($dist, sDist[k_{CMI}], C_{S^{i,j}V_i}, C_{S^{i,j}V_j}, C_{S^{i,j}}$)
24: **end for**
25: **end for**
26: ATOMICADD($partial_{cmi}[bx], \digamma(C_{S^{i,j}V_i}) + \digamma(C_{S^{i,j}V_j}) - \digamma(C_{S^{i,j}})$)

synchronization of the threads within the same thread block, each thread loops through the samples within the current stride, i.e., $b \in \{0, ..., (\gamma - 1)\}$. Within each loop iteration, the thread computes the distance $dist$ between its sample $D_{shared}[tx]$ and one sample from the current stride $D_{shared}[\gamma + b]$. The distance $dist$ is computed using the distance function DISTMETRIC, which defaults to the Chebyshev distance. If $dist$ is smaller than any value within $sDist$, i.e., the current k_{CMI}-nearest neighbors, $dist$ is inserted into the fix-sized array $sDist$ at the corresponding position to keep $sDist$ sorted. Note that the last element within $sDist$, i.e., the farthest distance, will be removed during this operation. After all n elements have been processed, the distances of the k_{CMI}-nearest neighbors are stored in sorted order $sDist$. Thus, the distance to the k_{CMI}-nearest neighbor is stored in $sDist$ at position k_{CMI}.

Next, the partial CMI values are computed, which requires counting the number of points within the distance of the k_{CMI}-nearest neighbor, i.e., within $sDist[k_{CMI}]$, for the following subspaces $\mathcal{V}_i \otimes \mathcal{S}^{i,j}$, $\mathcal{V}_j \otimes \mathcal{S}^{i,j}$ and $\mathcal{S}^{i,j}$. Accordingly, the algorithm initializes counters $C_{S^{i,j}V_i}$, $C_{S^{i,j}V_j}$ and $C_{S^{i,j}}$ for each subspace (see Algorithm 4 line 16). Again, all n samples are iterated in a stride with a size corresponding to the number of threads per block γ using the loadIntoShared function to load the appropriate samples into shared memory. For each element within the current stride, the GPU thread first computes the distances *dist* within the subspace $\mathcal{S}^{i,j}$. The updateCounter function checks if *dist* is within $sDist[k_{CMI}]$. If this check evaluates *true*, the algorithm increments $C_{S^{i,j}}$ and computes the distance within the other two subspaces, checks if these distances are within $sDist[k_{CMI}]$, and increments $C_{S^{i,j}V_i}$, $C_{S^{i,j}V_j}$ accordingly. Finally, after the n samples are processed, each GPU thread computes its partial result for its corresponding sample, i.e., $F(C_{S^{i,j}V_i}) + F(C_{S^{i,j}V_j}) - F(C_{S^{i,j}})$. The partial result is added to the partial CMI value $partial_{cmi}[bx]$ for the corresponding permutation or the original CMI estimate, i.e. if $bx == 0$ (see Algorithm 4 line 26). This addition requires an atomic operation to synchronize between threads from multiple thread blocks. Once all GPU threads terminate, the data structure $partial_{cmi}$ contains all partial CMI values for *perm* permutations and the non-permuted case.

4.2 A GPU-based PC Algorithm for Parallel Execution of GPU$_{\text{CMIknn}}$

The PC algorithm allows to plug in any CI-test to discover the causal structures. Thus, GPU$_{\text{CMIknn}}$, as sketched in Algorithm 2, can be directly applied, computing each CI-test individually on the GPU. Note, we call this version GPU$_{\text{CMIknn}}$-Single. In contrast, existing GPU-accelerated approaches to CSL assume that all CI-tests within the same level, i.e., with the same sized separation set, are computed in parallel on the GPU (Hagedorn and Huegle, 2021a; Schmidt et al., 2018; Zarebavani et al., 2020). Adopting this *one kernel launch per level* approach does not apply to GPU$_{\text{CMIknn}}$, given that two individual kernels are needed. The construction of one fused kernel that integrates the *localPermutation* kernel with the $estimate_{CMIknn}$ kernel is not considered for memory capacity reasons. In particular, reserving space for all possible CI-tests' local permutations in global memory can quickly exceed the GPU memory capacity. Besides, additional engineering is needed to compute the p-value within the kernel. Further, global synchronization across all GPU threads computing one CI-test needs to be introduced within the kernel after calculating the local permutation and computing the partial CMI values. Currently, this synchronization occurs implicitly through the separation into two kernels.

Nevertheless, we propose a version of the PC algorithm tailored to the GPU$_{\text{CMIknn}}$ CI-test's ideas, which computes multiple CI-test operations in parallel. We call this version GPU$_{\text{CMIknn}}$-Parallel. By computing multiple CI-test operations within one kernel, GPU$_{\text{CMIknn}}$-Parallel achieves speed-up using the following two optimizations. First, given that GPU$_{\text{CMIknn}}$ CI-test processes each of the n samples, respectively each of the $n \times (perm+1)$ samples, in one GPU thread, the number of GPU threads launched within a kernel may not fully utilize all available GPU cores. GPU$_{\text{CMIknn}}$-Parallel uses the idle GPU cores to compute multiple CI-tests in parallel within one kernel, which yields higher CI-test throughput. Second, GPU$_{\text{CMIknn}}$-Parallel reuses computed local permutations to avoid redundant

computations. Based on the local permutation for a given separation set, the algorithm estimates CMI values for multiple edges. In more detail, `GPU`$_{\texttt{CMIknn}}$`-Parallel` computes the local permutations for a given set of separation sets to one variable V_i in parallel at once. Afterward, these local permutations are used multiple times during CMI estimation with combinations of V_i to any of the variables $V_j \in a(V_i)$ adjacent to V_i. In Algorithm 5, we describe the algorithm for `GPU`$_{\texttt{CMIknn}}$`-Parallel`, which includes the two optimizations mentioned earlier.

Algorithm 5 `GPU`$_{\texttt{CMIknn}}$`-Parallel`: A GPU-based adjacency search of PC algorithm
Input: observational data D with n samples from $\mathbf{V}$ variables, significance level α, number of permutations $perm$, k-nearest neighbors within permutation k_{perm}, k-nearest neighbors within CMI estimation k_{CMI}, knn-based CMI estimation kernel $estimate_{CMIknn}()$
Output: estimated skeleton $\mathcal{C}^l$, separation sets $SepSet$

Start with fully connected skeleton $\mathcal{C}^{-1}$ and $l = -1$
repeat
  $l = l + 1$
  for all variables V_i in $\mathcal{C}^l$ **do**
    Let $a(V_i) = adj(\mathcal{C}^l, V_i)$;
  end for
  for all variables V_i in $\mathcal{C}^l$ with $|a(V_i)| > l$ **do**
    Compute all possible separation sets $\mathbf{S}^i$ from $a(V_i)$
    On GPU: Compute local permutations for all $\mathbf{S}^i$ with D, $perm$ and k_{perm}
    for all $S^i \in \mathbf{S}^i$ **do**
      Store local permutations in $localPerm[S^i]$
    end for
    repeat
      Choose S^i from $\mathbf{S}^i$
      On GPU: Estimate CMI for all $V_j \in a(V_i) \setminus \{S^i\}$ with D, S^i, $perm$ $estimate_{CMIknn}()$, k_{CMI} and $localPerm[S^i]$
      for all $V_j \in a(V_i)$ **do**
        Compute p based on computed CMI values
        if p $\geq \alpha$ **then**
          Delete edge $E^{i,j}$ from $\mathcal{C}^l$
          Save S^i in $SepSet$
          Remove V_j from $a(V_i)$
        end if
      end for
    until all computed S^i were chosen or $|a(V_i)| == 0$
  end for
until each V_i in $\mathcal{C}^l$ satisfies $|a(V_i)| < l$
return $\mathcal{C}^l$, $SepSet$

Our proposed GPU-based adjacency search of the PC algorithm, called `GPU`$_{\texttt{CMIknn}}$`-Parallel`, takes the common input parameters of the PC algorithm, such as observational data D or the significance level α, together with `GPU`$_{\texttt{CMIknn}}$ specific parameters, such as $perm$, k_{perm},

k_{CMI} or $estimate_{CMIknn}()$. The adjacency search outputs the estimated skeleton $\mathcal{C}^l$, and the corresponding separation sets $SepSet$. The algorithm starts with a fully connected skeleton $\mathcal{C}^0$, technically at level $l = 0$. In level $l = 0$, independence testing does not consider any separation sets, and therefore we can apply a simpler approach for the permutation following the implementation of `CMIknn` (Runge, 2018). Description of the approach for level $l = 0$ is skipped for brevity [2]. In any other level $l \geq 1$, the following steps are performed. First, for each variable $V_i \in \mathcal{C}^l$ the adjacent variables within the current version of the skeleton $\mathcal{C}^l$ are obtained (see Algorithm 5 lines 4–6). Next, for variables $V_i \in \mathcal{C}^l$ whose adjacency $a(V_i)$ has a size larger than the current level l, i.e., for which a separation set can be constructed, are iterated (see Algorithm 5 lines 7–25). All possible separation sets $\mathbf{S}^i$ are computed for V_i within this loop based on $a(V_i)$. For these separation sets, the local permutations are computed at once within one GPU kernel and stored in the data structure $localPerm$. The GPU kernel is an extended version of the $localPermutation$ kernel (see Algorithm 3), launched with additional δ thread blocks in the second grid dimension according to the number of possible separation sets, i.e., $\delta = |\mathbf{S}^i|$. In an inner-loop (see Algorithm 5 lines 13–24), one separation set S^i from $\mathbf{S}^i$ is selected within each iteration. Based on the selected separation set S^i, the partial CMI values are computed for V_i and all $V_j \in a(V_i) \setminus \{S^i\}$ at once within one GPU kernel. This GPU kernel is an extended version of the $estimate_{CMIknn}$ kernel (see Algorithm 4), which is launched with additional δ thread blocks in the third grid dimension according to the remaining adjacent variables $a(V_i)$, i.e., $\delta = |a(V_i)|$. Afterward, the p-value is computed based on the corresponding partial CMI values for each $V_j \in a(V_i)$. If $\mathrm{p} \geq \alpha$, the edge $E^{i,j}$ is removed from the current skeleton $\mathcal{C}^l$, the separation set S^i is stored in $Sepset$ at the position of $E^{i,j}$, and V_j is removed from $a(V_i)$ (see Algorithm 5 lines 18–22). Once all possible $S^i \in \mathbf{S}^i$ were chosen, or there is no adjacent variable in $a(V_i)$, the inner-loop is finished. After all variables $V_i \in \mathcal{C}^l$ were processed, the procedure is repeated with the next level $l = l + 1$. This process continues until no more separation sets with the size of the current level l can be constructed from the adjacency $a(V_i)$ for any variable $V_i \in \mathcal{C}^l$. At this point, the algorithm returns the current skeleton $\mathcal{C}^l$ and the corresponding separation sets $SepSet$.

Note that within the approach of $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, the number of possible separation sets $\mathbf{S}^i$ and the number of adjacent variables $|a(V_i)|$ can become significantly large and lead to a memory demand that exceeds the capacity of the GPU memory. For these scenarios, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ provides a blocked version, which operates on blocks of separation sets for each variable V_i. The blocked version introduces an additional loop, which performs all steps in lines 9–23 of Algorithm 5 for each block of separation sets. Similarly, a blocked version within the inner loop for the adjacent variables $V_j \in a(V_i)$ could be introduced if required.

5. Experiments

In the following section, we present results from a series of experiments to evaluate the runtime performance of $\texttt{GPU}_{\texttt{CMIknn}}$, our proposed GPU-accelerated version of the `CMIknn` CI-test (Runge, 2018), with varying parameters (see Section 5.2). Furthermore, we evaluate the runtime performance when $\texttt{GPU}_{\texttt{CMIknn}}$ is applied in constraint-based causal discovery

2. The implementation for level $l = 0$ is included in the GitHub repository

(see Section 5.3). Therefore, we compare a CPU-based version of the PC algorithm that employs `CMIknn` (Runge, 2018) with a version of the PC algorithm using $\texttt{GPU}_{\texttt{CMIknn}}$, called $\texttt{GPU}_{\texttt{CMIknn}}$`-Single`, and our proposed adaption of the PC algorithm $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`, which is optimized for the use of $\texttt{GPU}_{\texttt{CMIknn}}$. Note that the CPU-based version is executed on a single thread and using multiple CPU cores in parallel. All experiments are run following the experimental setup described as follows (see Section 5.1).

5.1 Experimental Setup

For each experiment, we conducted at least ten measurements executed on the same hardware setup. The experiments were conducted on a system with one AMD EPYC 7343 with 16 cores equipped with an NVIDIA A40 card, with 48 GB of global High Bandwidth Memory. The GPU card is connected via PCI-E 4.0. The system is equipped with 96 GB of DRAM. The operating system is Ubuntu 21.04, and the NVIDIA driver version 470.57 is installed with CUDA version 11.4. We report the median runtime in seconds of the measurement runs to reduce the impact of noise due to background operating system processes. The implementations used for the measurements are available online [3]. Note for the implementation of `CMIknn` (Runge, 2018) `tigramite` version 5.0 is used.

For the measurement runs, we utilize synthetic data, which allows us to investigate the scalability concerning several dimensions. These dimensions are, for example, the number of samples n, size of separation set $|S^{i,j}|$ or the number of variables N. We generate data for each measurement using the `MANM-CS` library (Huegle et al., 2021). In particular, we generate CGMs that contain only continuous variables and have an edge density randomly chosen between $\{0.1, \ldots, 0.5\}$. The functions associated with the edges are randomly selected from the following list: {*linear*, *quadratic*, *tanh*}. Further, the number of samples n and number of nodes N are chosen according to the requirements of the experiment setting, and for the remaining parameters of `MANM-CS` the default values are used. Note, for the CI-test evaluation experiments, we also use the generated CGMs, but explicitly measure the runtime of single CI-tests.

If not stated differently, we chose the following default values for the parameters for the examined implementations of $\texttt{GPU}_{\texttt{CMIknn}}$, $\texttt{GPU}_{\texttt{CMIknn}}$`-Single`, and $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` and `CMIknn`. We set $k_{perm} = 15$, which is slightly above the suggested range found in `CMIknn` (Runge, 2018) and use $perm = 100$ to avoid excessive experiment runtimes. Note the experiment runtime increases linearly with the number of permutations $perm$. Further, we set $\beta = 32$ and $\gamma = 32$. Therefore, each GPU thread block contains enough GPU threads to fill an entire warp. At the same time, we keep the amount of shared memory required for each GPU thread block low. For all experiments that require a significance level, we set $\alpha = 0.01$ by convention (Malinsky and Danks, 2018).

When comparing the GPU-accelerated approaches $\texttt{GPU}_{\texttt{CMIknn}}$, $\texttt{GPU}_{\texttt{CMIknn}}$`-Single`, and $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` to the CPU-based baseline `CMIknn`, the comparison focuses on two aspects. First and foremost, the comparison focuses on the difference stemming from the change in execution hardware, CPU vs. GPU. Second, the comparison considers different knn estimation approaches. For the CPU-based baseline, computational efficient kd-tree searches are employed. In contrast, the GPU-based versions are built upon brute-force

3. GitHub: https://github.com/ChristopherSchmidt89/gpucmiknn/

Table 1: Median runtime in seconds (20 CI-tests), scaling k_{CMI} for CI-tests with fixed parameters: $n = 1\,000$, $perm = 100$, $|S^{i,j}| = 1$. For percentiles see Table 6 in the Appendix.

Method	k_{CMI}									
	7	10	20	30	40	50	75	100	250	500
`CMIknn`	1.76	1.79	1.84	1.85	1.9	1.96	1.94	2.02	2.31	2.69
$\texttt{GPU}_{\texttt{CMIknn}}$	0.005	0.01	0.01	0.01	0.02	0.02	0.07	0.13	0.52	1.15

searches, which are computationally less efficient, but better suited for the parallel execution model of GPUs.

5.2 Runtime Evaluation of $\texttt{GPU}_{\texttt{CMIknn}}$

We compare the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$ to the CPU-based implementation `CMIknn` (Runge, 2018). We investigate the impact when scaling one of several parameters for the runtime evaluation. In detail, we consider the number of k_{CMI}-nearest neighbors, the number of samples n, the number of permutations $perm$, and the size of the separation set $|S^{i,j}|$. We do not evaluate the performance concerning changes in k_{perm} as k_{perm} does not impact runtime much (Runge, 2018).

Impact of k_{CMI}: According to Runge (2018) the k_{CMI}-nearest neighbors should be set to $k_{CMI} \approx \{0.1 \dots 0.2\} \times n$ to yield good power. In the context of $\texttt{GPU}_{\texttt{CMIknn}}$, the parameter k_{CMI} determines the size of arrays stored in GPU thread local memory. GPU thread local memory can yield high performance as long as the data is placed in registers, which are highly limited in size. Otherwise, performance degrades due to register spilling (Micikevicius, 2011), as data structures within GPU thread local memory are now placed within global memory. Thus, we assume that the runtime performance of $\texttt{GPU}_{\texttt{CMIknn}}$ drops while k_{CMI} is increased. Table 1 shows the median runtime in seconds for 20 CI-tests with $n = 1\,000$ samples, $perm = 100$ permutations and a separation set of size $|S^{i,j}| = 1$, when scaling k_{CMI} from $k_{CMI} = 7$ to $k_{CMI} = 500$. For the CPU-based baseline `CMIknn` that implements kd-search trees to estimate knn, we find that the runtime increases by approximately 53%. In contrast, for the GPU-based version $\texttt{GPU}_{\texttt{CMIknn}}$, which implements a brute-force approach to estimate knn, we see a runtime increase by a factor of 230. Particularly for $k_{CMI} > 50$, the runtime performance drops, which we account to register spilling. Thus, our assumption is confirmed. Comparing the CPU- and GPU-based approaches, we find that for small values of k_{CMI}, e.g., up to $k_{CMI} = 50$, the GPU-based version is up to a factor of 352 faster than the CPU-based version and remains faster by a factor of 2.3 even for $k_{CMI} = 500$. Yet, for these large values of k_{CMI} one should note that the GPU-based version is operating in parallel, while `CMIknn` is single-threaded. Based on these observations we will report the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$ with several settings of k_{CMI} for the following measurements.

Impact of number of permutations $perm$: The number of permutations $perm$ impacts the runtime of the local permutation computation and the CMI estimation. In fact, for both

Table 2: Median runtime in seconds (20 CI-tests), scaling *perm* for CI-tests with fixed parameters: $n = 1\,000$, $|S^{i,j}| = 1$. For percentiles see Table 7 in the Appendix.

Method	k_{CMI}	*perm* 50	100	250	500	1 000
`CMIknn`	200	1.18	2.39	5.98	12.47	24.61
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.004	0.01	0.01	0.02	0.04
	20	0.01	0.01	0.01	0.03	0.05
	200	0.2	0.39	0.93	1.83	3.65

Table 3: Median runtime in seconds (20 CI-tests), scaling $|S^{i,j}|$ for CI-tests with fixed parameters: $n = 1000$, $perm = 100$. For percentiles see Table 8 in the Appendix.

Method	k_{CMI}	$\lvert S^{i,j}\rvert$ 1	2	3	4	5
`CMIknn`	200	2.43	2.9	3.28	3.56	4.02
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.005	0.01	0.01	0.01	0.01
	20	0.01	0.01	0.01	0.01	0.01
	200	0.39	0.39	0.39	0.38	0.38

steps, the number of required computations increases linearly to the number of permutations *perm*. Within the $estimate_{CMIknn}$ kernel, a larger number of permutations *perm* results in launching the kernel with additional thread blocks, whereas the launch parameters for the $localPermutation$ kernel remain unaffected. In Table 2, we find the median runtime in seconds for 20 CI-tests with $n = 1\,000$ samples and a separation set of size $|S^{i,j}| = 1$ for several settings of k_{CMI} when scaling *perm* from $perm = 50$ to $perm = 1000$. For `CMIknn`, we find that the runtime increases by a factor of 20.8 from $perm = 50$ to $perm = 1\,000$, which confirms the linear increase in runtime, as *perm* is increased by a factor of 20. For $\texttt{GPU}_{\texttt{CMIknn}}$, we see that from $perm = 50$ to $perm = 1\,000$, the runtime increases below a factor of 20. While, for $k_{CMI} = \{7, 20\}$ the runtime increases up to a factor of 10, for $k_{CMI} = 200$ the runtime increases by up to factor of 18.25. We assume that the slightly lower increase in runtime is due to better utilization of the parallel computing capabilities of the GPU, given that more threads are launched during CMI estimation. Yet, for $k_{CMI} = 200$, the accesses to global memory, due to register spilling, seem to become a performance bottleneck.

Impact of separation set size $|S^{i,j}|$: The size of the separation set $|S^{i,j}|$ directly increases the number of dimensions within the knn searches during local permutation computation and CMI estimation. Higher dimensions impact the runtime of the kd-tree approach and the brute-force approach. Kd-trees generally suffer under the curse of dimensionality (Bentley, 1975). Thus we assume that the performance will drop with a larger separation set $|S^{i,j}|$ for `CMIknn`. Generally, we assume a similar behavior for $\texttt{GPU}_{\texttt{CMIknn}}$. In Table 3 we

Table 4: Median runtime in seconds (20 CI-tests), scaling n for CI-tests with fixed parameters: $|S^{i,j}| = 1$, $perm = 100$. Note $k_{CMI} = adaptive$ refers to a value dependent on the number of samples n, $k_{CMI} = 0.2 \times n$. For percentiles see Table 9 in the Appendix.

Method	k_{CMI}	n						
		100	250	500	1 000	2 500	5 000	10 000
`CMIknn`	7	0.43	0.65	1.02	1.79	4.76	9.82	20.62
	20	0.43	0.67	1.05	1.85	4.89	10.19	21.64
	adaptive	0.46	0.67	1.12	2.31	6.86	17.96	55.07
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.002	0.002	0.003	0.005	0.01	0.04	0.13
	20	0.002	0.002	0.004	0.01	0.02	0.05	0.17
	adaptive	0.002	0.004	0.04	0.39	5.7	44.88	355.77

find the median runtime in seconds for 20 CI-tests with $n = 1\,000$ samples and $perm = 100$ permutations for several settings of k_{CMI} when scaling the size of the separation set $|S^{i,j}|$ from $|S^{i,j}| = 1$ to $|S^{i,j}| = 5$. For `CMIknn` we find that the runtime increases by 65% from $|S^{i,j}| = 1$ to $|S^{i,j}| = 5$, which confirms our assumption. For $\texttt{GPU}_{\texttt{CMIknn}}$, we find for all three chosen parameters of $k_{CMI} = \{7, 20, 200\}$ that the runtime remains unaffected by the size of the separation set $|S^{i,j}|$. We observe that loading the additional dimensions into GPU thread local or shared memory does not add any costs.

Impact of number of samples n: The number of samples n has a major impact on the runtime of knn-estimation approaches (see Section 3.2). Thus, we assume that the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$ increases quadratic with an increase of the number of samples n, whereas the runtime of `CMIknn` increases approximately logarithmicconcerning to an increase of n. Furthermore, note that according to Runge (2018), the parameter k_{CMI}, which has a significant impact on the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$, depends on the number of samples n. Table 4 accounts for this dependence within the rows where $k_{CMI} = adaptive$, i.e., $k_{CMI} = 0.2 \times n$. Furthermore, in Table 4, we report the median runtime in seconds for 20 CI-tests with a separation set size of $|S^{i,j}| = 1$, and $perm = 100$ permutations for several settings of k_{CMI} when scaling the number of samples n from $n = 100$ to $n = 10\,000$. For `CMIknn`, we confirm that the runtime increases logarithmically to the number of samples. Similarly, our measurements confirm a quadratic increase in runtime for $\texttt{GPU}_{\texttt{CMIknn}}$, when $k_{CMI} = 200$. Yet, for small values of $k_{CMI} = \{7, 20\}$ we observe that the increase in runtime is less drastic. Again, we assume that for $k_{CMI} = 200$, accesses to global memory are the main bottleneck within the GPU kernel. In contrast, for the smaller values of k_{CMI} the additional GPU threads launched due to an increase of n hide some of the assumed performance degradations. Yet, for a certain number of samples n, the number of launched GPU threads exceeds the capabilities of the GPU hardware, and we observe the quadratic increase, e.g., for $n \geq 2\,500$. When comparing the CPU- and GPU-based approaches, we find a large performance gain for sample sizes of of up to $n = 1\,000$. The performance gain depends on the parameter k_{CMI}, but is within the range of factor 5.9 to factor 358. For larger values

Table 5: Speed-up over single-threaded CPU execution, median of 10 different CGMs, with $n = 1\,000$ and $perm = 100$. Significance level of PC algorithm $\alpha = 0.01$. Additional runtime measurements for selected high-dimensional settings can be found in Table 10 in the Appendix.

Method	k_{CMI}	N 10	20	30	40	50
	7	3.43	4.34	3.95	4.12	4.16
`CPU-8`	20	3.33	4.21	3.91	4.03	3.97
	200	2.36	2.81	3.44	3.35	3.88
	7	280.73	267.46	320.71	342.39	342.19
$\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$	20	190.55	176.5	216.87	258.3	234.09
	200	3.62	3.15	4.94	4.39	5.17
	7	469.07	458.64	844.69	985.83	1 002.0
$\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$	20	274.15	265.17	466.63	522.87	489.25
	200	3.7	3.48	5.09	4.68	5.59

of $n \geq 2\,500$, the GPU-based version only remains faster if the parameter k_{CMI} is fixed to a small value, e.g., $k_{CMI} = \{7, 20\}$. In this case, $\texttt{GPU}_{\texttt{CMIknn}}$ is up to a factor of 476 faster than `CMIknn`. In contrast, for the case that $k_{CMI} = adaptive$ `CMIknn` is faster by a factor of up to 6.5 for large n.

5.3 Runtime Evaluation of Adjacency Search in PC Algorithm with $\texttt{GPU}_{\texttt{CMIknn}}$

In the following, we experimentally evaluate the runtime performance of our proposed GPU-based CI-test $\texttt{GPU}_{\texttt{CMIknn}}$ used within the PC algorithm. Therefore, we compare the CPU-based CI-test `CMIknn` applied within a single-threaded and parallel CPU-based version of the PC algorithm to the two GPU-accelerated versions $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$ and $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, which are described in Section 4.2.

In Table 5 we present the measurements based on synthetic generated CGMs (see Section 5.1 for detail), increasing the number of variables N within the CGMs, keeping the number of samples fixed at $n = 1\,000$, and the number permutations at $perm = 100$. The significance level is set to $\alpha = 0.01$. We report the median speed-up for 10 CGMs for $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, and `CPU-8` over the single-threaded CPU-based version. Note `CPU-8` is the parallel CPU-based version running on 8 CPU cores.

From the measurements presented in Table 5, we make the following observations: First, the CPU-based parallel version, `CPU-8`, achieves a speed-up of roughly factor 4 over the single-threaded version, even though it utilizes 8 CPU cores. For $k_{CMI} = 200$, we observe slightly less speed-up than for small values of k_{CMI}. This effect is explained given that single CI-test runtime is higher for $k_{CMI} = 200$, which amplifies the impact of load imbalance present in parallel execution of the PC algorithm' adjacency search (Schmidt et al., 2019). For the GPU-based versions of the PC algorithm, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, we observe a high speed-up for small values of $k_{CMI} = \{7, 20\}$. Whereas for $k_{CMI} = 200$,

the achieved speed-up is within the CPU-based version `CPU-8`. Furthermore, we find that for smaller values of N, e.g., $N \leq 40$, the speed-up of $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` over the single-threaded CPU-based version increases as N increases. We assume that this additional speed-up is a results from idea behind $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` to process multiple CI-tests in parallel. For larger N, the parallel computing capabilities of the GPU are already saturated, and no additional speed-up is achieved.

Comparing $\texttt{GPU}_{\texttt{CMIknn}}$`-Single` with $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`, we observe that for $k_{CMI} = \{7, 20\}$ between factors of 1.44 to 2.93, additional speed-up is achieved with our GPU-accelerated version of the PC algorithm tailored for the $\texttt{GPU}_{\texttt{CMIknn}}$ CI-test. In contrast, for $k_{CMI} = 200$, both versions achieve a similar speed-up.

Overall, we confirm the performance gain of $\texttt{GPU}_{\texttt{CMIknn}}$ in the context of the PC algorithm over the existing CPU-based version `CMIknn` for small values of k_{CMI}, e.g., $k_{CMI} = \{7, 20\}$, as already observed in the experiments of the previous Section 5.2. In these settings, our proposed version $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` outperforms `CPU-8` by factors of up to 240. Yet, for large k_{CMI}, i.e., $k_{CMI} = 200$, `CPU-8` and both GPU versions of the PC algorithm have a similar runtime. Thus, If runtime is the main goal, we recommend choosing a small value for k_{CMI}. Although, this contradicts the recommendation for choosing k_{CMI} by Runge (2018).

Impact of k_{CMI} on quality of learned CGM: The proposed GPU-accelerated algorithms provide good runtime for small values of k_{CMI}, regardless of other parameters. Thus, we examine the impact of the parameter k_{CMI} on the structural hamming distance (SHD) (Tsamardinos et al., 2006) when discovering the CGM. The SHD allows insights into the quality of the learned CGM. Therefore, we randomly generate 50 CGMs with $N = 20, n = 1\,000$ and run the PC algorithm version $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` with $perm = 100$, $\alpha = 0.01$ and various values for k_{CMI} from $k_{CMI} = 7$ to $k_{CMI} = 200$. We set the computed SHD for $k_{CMI} = 7$ as a baseline. Then, we calculate the difference between the baseline SHD and the SHD computed for other values of k_{CMI}. Thus, values of the SHD below 0 indicate a quality improvement of the learned CGM.

In Figure 1, we report the minimum, median and maximum difference of the SHD, denoted by ΔSHD, from the 50 CGMs. We observe that the median SHD improves for up to $k_{CMI} = 30$, remains similar for up to $k_{CMI} = 75$, and deteriorates for $k_{CMI} \geq 100$, compared to the SHD calculated for $k_{CMI} = 7$. Based on this observation, one could conclude that small values of k_{CMI} are sufficient to learn the CGM, which favor the runtime improvement of our GPU-accelerated approaches. Yet, for several CGMs, there is an improvement of the SHD observable for large values of k_{CMI}. Thus, a trade-off remains between runtime and the quality of the learned CGM, based on the parameter k_{CMI}. Further research on the impact of k_{CMI} on the quality of the learned CGM would be needed.

6. Conclusion

In this work, we propose a GPU-accelerated CI-test for nonlinear relationships called $\texttt{GPU}_{\texttt{CMIknn}}$. $\texttt{GPU}_{\texttt{CMIknn}}$ is a GPU-based version of `CMIknn` (Runge, 2018), an existing CI-test using CMI combined with a local permutation scheme. In $\texttt{GPU}_{\texttt{CMIknn}}$, we leverage the parallel computing capabilities of GPUs to accelerate the local permutation computation and the CMI estimation. Furthermore, we introduce an extension of the PC algorithm that employs

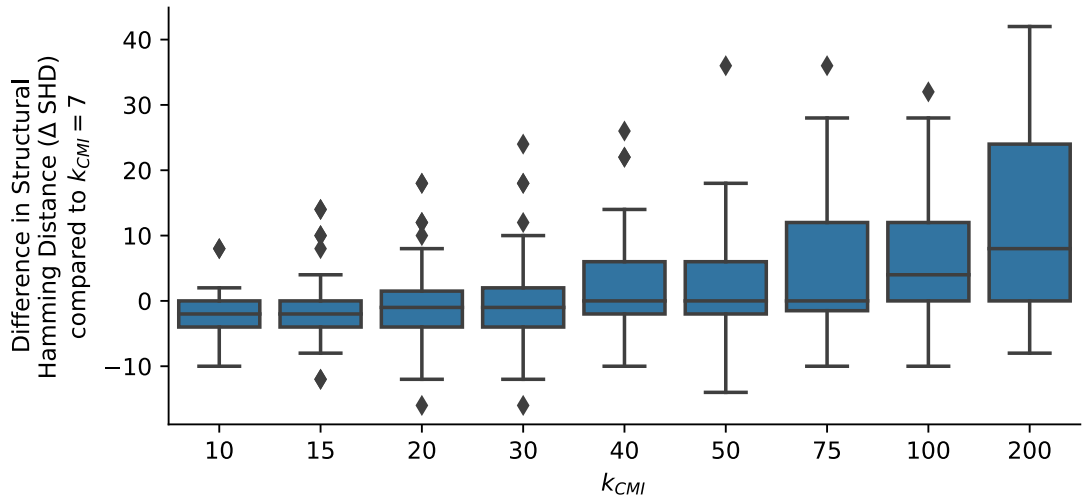

Figure 1: Development of SHD with increasing k_{CMI} as difference regarding the SHD with $k_{CMI} = 7$, computed for 50 random generated CGMs with $N = 20$, $n = 1\,000$ and algorithm parameters $perm = 100$ and $\alpha = 0.01$. $\Delta SHD < 0$ describes an improved quality.

$\texttt{GPU}_{\texttt{CMIknn}}$ to compute multiple CI-tests in parallel. The approach called $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel reuses computed local permutations to reduce the computational demand.

In our experimental evaluation, we demonstrate the runtime performance of our GPU-based approaches concerning relevant parameters of the algorithm. In particular, we find that the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$ and, respectively, $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel mainly depends on the chosen value for the parameter k_{CMI}. For small values of k_{CMI}, e.g., $k_{CMI} = 7$, $\texttt{GPU}_{\texttt{CMIknn}}$ is up to a factor 352 faster than its CPU-based counterpart. In the context of the PC algorithm, $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel outperforms a multi-threaded CPU-based version running on eight cores by up to a factor of 240. In contrast, for large values of k_{CMI},e.g.,$k_{CMI} = 200$, we find that $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel has similar runtimes to the multi-threaded CPU-based version. $\texttt{GPU}_{\texttt{CMIknn}}$ remains only up to a factor 2.3 faster than its single-threaded CPU-based counterpart for $k_{CMI} = 500$. The chosen value of k_{CMI} impacts the quality of the CI-test (Runge, 2018) and the quality of a learned CGM. Yet, our evaluation indicates that the impact of a smaller value for k_{CMI} on the quality of the learned CGM is not as strong as expected. Further research is required to determine suitable values for k_{CMI} that balance runtime improvements with loss in result quality. In this context, a more extensive experimental evaluation comparing $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel to other existing methods for causal discovery in non-linear settings (Strobl et al., 2019; Zhang et al., 2011) is left for future work.

References

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

David Maxwell Chickering. Learning Equivalence Classes of Bayesian-Network Structures. *Journal of Machine Learning Research*, 2(3):445–498, 2002.

Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 The A100 Datacenter GPU and Ampere Architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 48–50, 2021.

Diego Colombo and Marloes H Maathuis. Order-Independent Constraint-Based Causal Structure Learning. *Journal of Machine Learning Research*, 15(116):3921–3962, 2014.

A Philip Dawid. Conditional Independence in Statistical Theory. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(1):1–31, 1979.

Gary Doran, Krikamol Muandet, Kun Zhang, and Bernhard Schölkopf. A Permutation-Based Kernel Conditional Independence Test. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, UAI'14, pages 132–141. AUAI Press, 2014.

R. A. Fisher. Frequency Distribution of the Values of the Correlation Coefficient in Samples from an Indefinitely Large Population. *Biometrika*, 10(4):507–521, 1915.

Stefan Frenzel and Bernd Pompe. Partial Mutual Information for Coupling Analysis of Multivariate Time Series. *Physical Review Letters*, 99:204101, 2007.

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1603–1618. ACM, 2018.

Weihao Gao, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. Estimating Mutual Information for Discrete-Continuous Mixtures. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 5988–5999. Curran Associates Inc., 2017.

Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proceedings of the International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer K-d trees: Processing massive nearest neighbor queries on GPUs. In *Proceedings of the 31st International Conference on International Conference on Machine Learning*, volume 32 of *ICML'14*, pages 172–180. JMLR.org, 2014.

Ce Guo and Wayne Luk. *Accelerating Constraint-Based Causal Discovery by Shifting Speed Bottleneck*, pages 169–179. Association for Computing Machinery, 2022.

Christopher Hagedorn and Johannes Huegle. GPU-Accelerated Constraint-Based Causal Structure Learning for Discrete Data. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 37–45. Society for Industrial and Applied Mathematics, 2021a.

Christopher Hagedorn and Johannes Huegle. Constraint-Based Causal Structure Learning in Multi-GPU Environments. In *Proceedings of the LWDA 2021 Workshops: FGWM, KDML, FGWI-BIA, and FGIR*, volume 2993 of *CEUR Workshop Proceedings*, pages 106–118. CEUR-WS.org, 2021b.

Christopher Hagedorn, Johannes Huegle, and Rainer Schlosser. Understanding unforeseen production downtimes in manufacturing processes using log data-driven causal reasoning. *Journal of Intelligent Manufacturing*, 2022.

John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

Katerina Hlaváčková-Schindler, Milan Paluš, Martin Vejmelka, and Joydeep Bhattacharya. Causality detection based on information-theoretic approaches in time series analysis. *Physics Reports*, 441(1):1–46, 2007.

Johannes Huegle. An Information-Theoretic Approach on Causal Structure Learning for Heterogeneous Data Characteristics of Real-World Scenarios. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4891–4892. International Joint Conferences on Artificial Intelligence Organization, 2021. Doctoral Consortium.

Johannes Huegle, Christopher Hagedorn, and Matthias Uflacker. 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*, pages 5246–5248. International Joint Conferences on Artificial Intelligence Organization, 2020. Demos.

Johannes Huegle, Christopher Hagedorn, Lukas Böhme, Mats Pörschke, Jonas Umland, and Rainer Schlosser. MANM-CS: Data Generation for Benchmarking Causal Structure Learning from Mixed Discrete-Continuous and Nonlinear Data. In *WHY-21 @ NeurIPS 2021*, 2021.

Markus Kalisch and Peter Bühlmann. Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm. *Journal of Machine Learning Research*, 8:613–636, 2007.

David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 2013.

Thuc Duy Le, Tao Hoang, Jiuyong Li, Lin Liu, Huawen Liu, and Shu Hu. A Fast PC Algorithm for High Dimensional Causal Discovery with Multi-Core PCs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(5):1483–1495, 2019.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39–55, 2008.

Anders L. Madsen, Frank Jensen, Antonio Salmerón, Helge Langseth, and Thomas D. Nielsen. Parallelisation of the PC Algorithm. In *Advances in Artificial Intelligence*, pages 14–24. Springer-Verlag New York, Inc., 2015.

Daniel Malinsky and David Danks. Causal Discovery Algorithms: A Practical Guide. *Philosophy Compass*, 13(1):e12470, 2018.

Octavio César Mesner and Cosma Rohilla Shalizi. Conditional Mutual Information Estimation for Mixed, Discrete and Continuous Data. *IEEE Transactions on Information Theory*, 67(1):464–484, 2021.

Paulius Micikevicius. Local Memory and Register Spilling, 2011. URL `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 16:1–16:14. ACM, 2008.

Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2nd edition, 2009.

Karl F.R.S. Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

Jonas Peters, Dominik Janzing, and Bernhard Schölkopf. *Elements of Causal Inference - Foundations and Learning Algorithms*. Adaptive Computation and Machine Learning Series. MIT Press, 2017.

RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. URL `https://rapids.ai`.

Andrea Rau, Florence Jaffrézic, and Grégory Nuel. Joint estimation of causal effects from observational and intervention gene expression data. *BMC Systems Biology*, 7(1):111, 2013.

Donald B. Rubin. The design versus the analysis of observational studies for causal effects: Parallels with the design of randomized trials. *Statistics in Medicine*, 26:20–36, 2007.

Jakob Runge. Conditional Independence Testing based on a Nearest-Neighbor Estimator of Conditional Mutual Information. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84, pages 938–947. PMLR, 2018.

Christopher Schmidt, Johannes Huegle, and Matthias Uflacker. 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, pages 19:1–19:10. ACM, 2018.

Christopher Schmidt, Johannes Huegle, Philipp Bode, and Matthias Uflacker. Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data. In *Proceedings of Machine Learning Research*, volume 104, pages 59–77. PMLR, 2019.

Christopher Schmidt, Johannes Huegle, Siegfried Horschig, and Matthias Uflacker. Out-of-Core GPU-Accelerated Causal Structure Learning. In *Algorithms and Architectures for Parallel Processing*, pages 89–104. Springer International Publishing, 2020.

Marco Scutari. Bayesian Network Constraint-Based Structure Learning Algorithms: Parallel and Optimized Implementations in the bnlearn R Package. *Journal of Statistical Software, Articles*, 77(2):1–20, 2017.

Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, Prediction, and Search, Second Edition*. Adaptive Computation and Machine Learning. MIT Press, 2000.

Eric V. Strobl, Kun Zhang, and Shyam Visweswaran. Approximate kernel-based conditional independence tests for fast non-parametric causal discovery. *Journal of Causal Inference*, 7(1), 2019.

Ioannis Tsamardinos, Laura E. Brown, and Constantin F. Aliferis. The max-min hill-climbing bayesian network structure learning algorithm. *Machine Learning*, 65(1):31–78, 2006.

Martin Vejmelka and Milan Paluš. Inferring the directionality of coupling with conditional mutual information. *Physical Review E*, 77:026214, 2008.

Behrooz Zarebavani, Foad Jafarinejad, Matin Hashemi, and Saber Salehkaleybar. cuPC: CUDA-Based parallel PC algorithm for causal structure learning on GPU. *IEEE Transactions on Parallel and Distributed Systems*, 31(03):530–542, 2020.

Kun Zhang, Jonas Peters, Dominik Janzing, and Bernhard Schölkopf. Kernel-based conditional independence test and application in causal discovery. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, pages 804––813. AUAI Press, 2011.

Appendix

A. Runtime Evaluation of $\texttt{GPU}_{\texttt{CMIknn}}$

Table 6: Median runtime in seconds (20 CI-tests), scaling k_{CMI} for CI-tests with fixed parameters: $n = 1\,000$, $perm = 100$, $|S^{i,j}| = 1$. Extended version of Table 1 including percentiles.

Method	percentile	k_{CMI} 7	10	20	30	40	50	75	100	250	500
CMIknn	0.05	1.74	1.77	1.82	1.81	1.89	1.94	1.93	1.99	2.2	2.61
	0.5	1.76	1.79	1.84	1.85	1.9	1.96	1.94	2.02	2.31	2.69
	0.95	2.03	1.89	1.96	1.98	2.03	2.07	2.06	2.14	2.48	2.87
$\texttt{GPU}_{\texttt{CMIknn}}$	0.05	0.005	0.01	0.01	0.01	0.02	0.02	0.07	0.13	0.52	1.14
	0.5	0.005	0.01	0.01	0.01	0.02	0.02	0.07	0.13	0.52	1.15
	0.95	0.005	0.01	0.01	0.01	0.02	0.03	0.07	0.13	0.52	1.15

Table 7: Median runtime in seconds (20 CI-tests), scaling $perm$ for CI-tests with fixed parameters: $n = 1\,000$, $|S^{i,j}| = 1$. Extended version of Table 2 including percentiles.

Method	k_{CMI}	percentile	$perm$ 50	100	250	500	1 000
CMIknn	200	0.05	1.17	2.36	5.87	11.79	23.87
		0.5	1.18	2.39	5.98	12.47	24.61
		0.95	1.29	2.51	6.3	12.63	25.03
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.05	0.004	0.005	0.01	0.02	0.04
		0.5	0.004	0.01	0.01	0.02	0.04
		0.95	0.004	0.01	0.01	0.02	0.04
$\texttt{GPU}_{\texttt{CMIknn}}$	20	0.05	0.01	0.01	0.01	0.03	0.05
		0.5	0.01	0.01	0.01	0.03	0.05
		0.95	0.01	0.01	0.01	0.03	0.05
$\texttt{GPU}_{\texttt{CMIknn}}$	200	0.05	0.19	0.39	0.92	1.82	3.64
		0.5	0.2	0.39	0.93	1.83	3.65
		0.95	0.2	0.39	0.93	1.84	3.67

Table 8: Median runtime in seconds (20 CI-tests), scaling $|S^{i,j}|$ for CI-tests with fixed parameters: $n = 1\,000$, $perm = 100$. Extended version of Table 3 including percentiles.

Method	k_{CMI}	percentile	$\lvert S^{i,j}\rvert$ 1	2	3	4	5
`CMIknn`	200	0.05	2.4	2.85	3.21	3.47	3.89
		0.5	2.43	2.9	3.28	3.56	4.02
		0.95	2.57	3.04	3.46	3.76	4.18
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.05	0.005	0.01	0.01	0.01	0.01
		0.5	0.005	0.01	0.01	0.01	0.01
		0.95	0.005	0.01	0.01	0.01	0.01
$\texttt{GPU}_{\texttt{CMIknn}}$	20	0.05	0.01	0.01	0.01	0.01	0.01
		0.5	0.01	0.01	0.01	0.01	0.01
		0.95	0.01	0.01	0.01	0.01	0.01
$\texttt{GPU}_{\texttt{CMIknn}}$	200	0.05	0.39	0.38	0.38	0.38	0.38
		0.5	0.39	0.39	0.39	0.38	0.38
		0.95	0.39	0.39	0.39	0.39	0.39

Table 9: Median runtime in seconds (20 CI-tests), scaling n for CI-tests with fixed parameters: $|S^{i,j}| = 1$, $perm = 100$. Note $k_{CMI} = adaptive$ refers to a value dependent on the number of samples n, $k_{CMI} = 0.2 \times n$. Extended version of Table 4 including percentiles.

Method	k_{CMI}	percentile	n						
			100	250	500	1 000	2 500	5 000	10 000
`CMIknn`	7	0.05	0.43	0.64	1.02	1.78	4.72	9.75	20.52
		0.5	0.43	0.65	1.02	1.79	4.76	9.82	20.62
		0.95	0.55	0.76	1.13	1.92	4.89	10.02	20.79
`CMIknn`	20	0.05	0.43	0.65	1.03	1.83	4.75	10.04	21.58
		0.5	0.43	0.67	1.05	1.85	4.89	10.19	21.64
		0.95	0.55	0.78	1.15	1.99	5.05	10.41	22.03
`CMIknn`	adaptive	0.05	0.41	0.64	1.1	2.16	6.78	17.76	54.4
		0.5	0.46	0.67	1.12	2.31	6.86	17.96	55.07
		0.95	0.57	0.77	1.23	2.43	7.46	18.1	55.61
$\texttt{GPU}_{\texttt{CMIknn}}$	7	0.05	0.002	0.002	0.003	0.005	0.01	0.04	0.13
		0.5	0.002	0.002	0.003	0.005	0.01	0.04	0.13
		0.95	0.002	0.002	0.003	0.005	0.01	0.04	0.13
$\texttt{GPU}_{\texttt{CMIknn}}$	20	0.05	0.002	0.002	0.004	0.01	0.02	0.05	0.17
		0.5	0.002	0.002	0.004	0.01	0.02	0.05	0.17
		0.95	0.002	0.003	0.004	0.01	0.02	0.05	0.17
$\texttt{GPU}_{\texttt{CMIknn}}$	adaptive	0.05	0.002	0.004	0.04	0.39	5.69	44.8	355.37
		0.5	0.002	0.004	0.04	0.39	5.7	44.88	355.77
		0.95	0.002	0.004	0.04	0.39	5.72	45.04	357.36

B. Runtime Evaluation of Adjacency Search in PC Algorithm with $\texttt{GPU}_{\texttt{CMIknn}}$

Table 10: Runtime in seconds for high-dimensional sparse synthetic CGMs and selected gene expression datasets used in previous work (Schmidt et al., 2018). The sparse synthetic CGMs are generated as described in Section 5.1, with $n = 1\,000$, but have an edge density that leads to DAGs with an average degree of approximately 1.5. The algorithms' parameters are set as follows: $perm = 100$, $k_{CMI} = 7$, $k_{perm} = 15$, $\beta = \gamma = 32$, $\alpha = 0.01$. Note experiment runs longer than 24 hours were terminated and are marked with did not finish (DNF).

Method	Synthetic CGMs					NCI-60	MCC	BR51
	N							
	100	250	500	750	1 000	1 190	1 380	1 592
`CPU-8`	1 399	5 425	17 586	40 762	62 967	82 287	DNF	DNF
$\texttt{GPU}_{\texttt{CMIknn}}$`-Single`	25.5	122	446	1 028	1 715	1 867	7 545	4 677
$\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`	14.4	74.7	281	637	1 088	764	1 653	1 680