Hasso-Plattner-Institut für Digital Engineering
Enterprise Platforms and Integration Concepts

Parallel Execution of Causal Structure Learning on Graphics Processing Units

Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
in der Wissenschaftsdisziplin *Praktische Informatik*

eingereicht an der
Digital Engineering Fakultät
der Universität Potsdam

von

Christopher Hagedorn geb. Schmidt

Betreuer:
Prof. Dr. h.c. mult. Hasso Plattner

Gutachter:
Prof. Dr. Jakob Runge
Prof. Thuc Le, PhD

Potsdam, 19. Dezember 2022

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-59758
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-597582

Abstract

Learning the causal structures from observational data is an omnipresent challenge in data science. The amount of observational data available to Causal Structure Learning (CSL) algorithms is increasing as data is collected at high frequency from many data sources nowadays. While processing more data generally yields higher accuracy in CSL, the concomitant increase in the runtime of CSL algorithms hinders their widespread adoption in practice. CSL is a parallelizable problem. Existing parallel CSL algorithms address execution on multi-core Central Processing Units (CPUs) with dozens of compute cores. However, modern computing systems are often heterogeneous and equipped with Graphics Processing Units (GPUs) to accelerate computations. Typically, these GPUs provide several thousand compute cores for massively parallel data processing.

To shorten the runtime of CSL algorithms, we design efficient execution strategies that leverage the parallel processing power of GPUs. Particularly, we derive GPU-accelerated variants of a well-known constraint-based CSL method, the PC algorithm, as it allows choosing a statistical Conditional Independence test (CI test) appropriate to the observational data characteristics.

Our two main contributions are: (1) to reflect differences in the CI tests, we design three GPU-based variants of the PC algorithm tailored to CI tests that handle data with the following characteristics. We develop one variant for data assuming the Gaussian distribution model, one for discrete data, and another for mixed discrete-continuous data and data with non-linear relationships. Each variant is optimized for the appropriate CI test leveraging GPU hardware properties, such as shared or thread-local memory. Our GPU-accelerated variants outperform state-of-the-art parallel CPU-based algorithms by factors of up to 93.4× for data assuming the Gaussian distribution model, up to 54.3× for discrete data, up to 240× for continuous data with non-linear relationships and up to 655× for mixed discrete-continuous data. However, the proposed GPU-based variants are limited to datasets that fit into a single GPU's memory. (2) To overcome this shortcoming, we develop approaches to scale our GPU-based variants beyond a single GPU's memory capacity. For example, we design an out-of-core GPU variant that employs explicit memory management to process arbitrary-sized datasets. Runtime measurements on a large gene expression dataset reveal that our out-of-core GPU variant is 364 times faster than a parallel CPU-based CSL algorithm. Overall, our proposed GPU-accelerated variants speed up CSL in numerous settings to foster CSL's adoption in practice and research.

Acknowledgements

I am truly grateful for my time at the *Enterprise Platform and Integration Concepts* (EPIC) research group where I was able to gain valuable experience, pursue my research interests, and become part of an exceptional team.

I want to sincerely thank my supervisor Prof. Hasso Plattner for supporting me to do research in a field of my interest. Prof. Plattner and his chair representatives, Dr. Matthias Uflacker and Dr. Michael Perscheid provided helpful guidance and support for this thesis and throughout my time at the Hasso Plattner Institute. My sincere thanks also go to SAP for funding my research.

I want to thank all students who worked under my supervision on the topic of causal structures learning as part of their master's theses, research seminars, and master's projects, or as student assistants.

My colleagues from the EPIC research group, especially Dr. Markus Dreseler, Johannes Huegle, Dr. Jan Kossmann, Keven Richly, and Dr. Rainer Schlosser provided critical feedback and were available for open discussions and always motivated me to accept challenges and explore new ideas. Thank you for the collaboration and friendship!

I would like to thank my family, especially Manfred, Petra, Gunhilde, Alexander, and Thomas, for their everlasting support and for sparking my interest in technology at a young age. Finally, I am deeply grateful for the unconditional support from my wife Christiane, her encouragement and advice, as well as the motivation I experienced from her.

Zusammenfassung

Das Lernen von kausalen Strukturen aus Beobachtungsdatensätzen ist eine allgegenwärtige Herausforderung im Data Science-Bereich. Die für die Algorithmen des kausalen Strukturlernens (CSL) zur Verfügung stehende Menge von Beobachtungsdaten nimmt zu, da heutzutage mit hoher Frequenz Daten aus vielen Datenquellen gesammelt werden. Während die Verarbeitung von höheren Datenmengen im Allgemeinen zu einer höheren Genauigkeit bei CSL führt, hindert die damit einhergehende Erhöhung der Laufzeit von CSL-Algorithmen deren breite Anwendung in der Praxis. CSL ist ein parallelisierbares Problem. Bestehende parallele CSL-Algorithmen eignen sich für die Ausführung auf Mehrkern-Hauptprozessoren (CPUs) mit Dutzenden von Rechenkernen. Moderne Computersysteme sind jedoch häufig heterogen. Um notwendige Berechnungen zu beschleunigen, sind die Computersysteme typischerweise mit Grafikprozessoren (GPUs) ausgestattet, wobei diese GPUs mehrere tausend Rechenkerne für eine massive parallele Datenverarbeitung bereitstellen.

Um die Laufzeit von Algorithmen für das kausale Strukturlernen zu verkürzen, entwickeln wir im Rahmen dieser Arbeit effiziente Ausführungsstrategien, die die parallele Verarbeitungsleistung von GPUs nutzen. Dabei entwerfen wir insbesondere GPU-beschleunigte Varianten des PC-Algorithmus, der eine bekannte Constraint-basierte CSL-Methode ist. Dieser Algorithmus ermöglicht die Auswahl eines – den Eigenschaften der Beobachtungsdaten entsprechenden – statistischen Tests auf bedingte Unabhängigkeit (CI-Test).

Wir leisten in dieser Doktorarbeit zwei wissenschaftliche Hauptbeiträge: (1) Um den Unterschieden in den CI-Tests Rechnung zu tragen, entwickeln wir drei GPU-basierte, auf CI-Tests zugeschnittene Varianten des PC-Algorithmus. Dadurch können Daten mit den folgenden Merkmalen verarbeitet werden: eine Variante fokussiert sich auf Daten, die das Gaußsche Verteilungsmodell annehmen, eine weitere auf diskrete Daten und die dritte Variante setzt den Fokus auf gemischte diskret-kontinuierliche Daten sowie Daten mit nicht-linearen funktionalen Beziehungen. Jede Variante ist für den entsprechenden CI-Test optimiert und nutzt Eigenschaften der GPU-Hardware wie beispielsweise "Shared Memory" oder "Thread-local Memory" aus. Unsere GPU-beschleunigten Varianten übertreffen die modernsten parallelen CPU-basierten Algorithmen um Faktoren von bis zu 93,4× für Daten, die das Gaußsche Verteilungsmodell annehmen, bis zu 54,3× für diskrete Daten, bis zu 240× für kontinuierliche

Daten mit nichtlinearen Beziehungen und bis zu 655× für gemischte diskret-kontinuierliche Daten. Die vorgeschlagenen GPU-basierten Varianten sind dabei jedoch auf Datensätze beschränkt, die in den Speicher einer einzelnen GPU passen. (2) Um diese Schwachstelle zu beseitigen, entwickeln wir Ansätze zur Skalierung unserer GPU-basierten Varianten über die Speicherkapazität einer einzelnen GPU hinaus. So entwerfen wir beispielsweise eine auf einer expliziten Speicherverwaltung aufbauenden Out-of-Core-Variante für eine einzelne GPU, um Datensätze beliebiger Größe zu verarbeiten. Laufzeitmessungen auf einem großen Genexpressionsdatensatz zeigen, dass unsere Out-of-Core GPU-Variante 364-mal schneller ist als ein paralleler CPU-basierter CSL-Algorithmus.

Insgesamt beschleunigen unsere vorgestellten GPU-basierten Varianten das kausale Strukturlernen in zahlreichen Situationen und unterstützen dadurch die breite Anwendung des kausalen Strukturlernens in Praxis und Forschung.

Contents

1

Introduction

Data have become an essential resource and driver for economic success [50], while advances in digital technology have led to a higher volume of data [270]. This is, on the one hand, due to data from existing systems being emitted at a higher frequency, and on the other hand, due to the technological progress in incorporating data from additional data sources to map the complex system from the real world in greater detail.

For example, driven by the Internet of Things (IoT) in modern manufacturing, data are collected from robots, sensors, and other smart devices. These data are emitted at a high frequency and in fine granularity, resulting in hundreds of gigabytes of data. Thus, the captured data make it possible to monitor every step of today's highly complex and automatized production line [261].

A further example is genetic research, where modern RNA sequencing methods [255] enable the collection of comprehensive gene expression data. The large-sized gene expression data help address biomedical challenges, such as drug design [54]. Summarizing, we observe the following two trends:

- **Trend (T1) amount of data**: Higher data emission rates, more fine-grained data, and additional data emitting devices increase the entire amount of data collected.
- **Trend (T2) complexity of systems**: Additional data-emitting devices, e.g., sensors in production lines, lead to an increasing number of individual data sources. While this allows for detailed models of the observed systems, considering each data source as a separate variable increases the complexity of the models.

As a result of these two trends, existing computational methods for knowledge discovery, e.g., data analysis or data-driven decision support, stemming, for example, from the fields of Machine Learning (ML) [164], or Deep Learning (DL) [127] must process extensive amounts of raw data to derive meaningful information. Processing more data generally increases the runtime of computational methods for knowledge discovery. The increased runtime gives rise to research in efficiently processing algorithms for data analysis using modern heterogeneous computing systems, particularly leveraging the parallel computing power of GPUs [11, 57, 105, 127, 138, 205].

Besides quickly processing extensive amounts of raw data, the quality of derived information and the interpretability of learned models [209] are important for value creation from the raw data [270]. Methods from the domain of ML or

DL focus on learning patterns in the raw data. The detected patterns are used to solve a number of tasks, such as classifications [24, 103] or predictions [141]. In contrast, methods from the field of Causal Structure Learning (CSL) [91] go beyond correlation-based analysis and focus on discovering the causal mechanisms of the system. The discovery of causal mechanisms enables an understanding of cause and effect relationships between variables in a system based on the observed raw data [188]. Therefore, CSL enables a deeper understanding of the data, particularly its data-generating process.

1.1 Learning Cause and Effect Relationships

In a broader sense, methods of CSL belong to the field of knowledge discovery. Knowledge discovery can be defined as the task of extracting information from data [62]. The extracted information represents knowledge, which is deduced through knowledge discovery methods. Further, the extracted knowledge is generally assumed to be more implicit, practical, as well as simpler than the data itself [62]. Methods from the field of CSL concentrate on discovering the causal mechanisms of the system, i.e., the true Causal Graphical Model (CGM), which produced the raw data. Knowing the causal relationships, at least partially, allows for identifying causal effects or applying causal inference [74, 149, 196].

CSL is based on the theoretical framework of causality as introduced by Pearl [188]. The theoretical framework of causality builds upon the Markov condition, which connects graphical models representing a set of variables with the joint probability distribution over the variables [251]. Hence, if—given a set of variables—the graphical model is compatible with the joint probability distribution, the statistical independence information coincides with the graphical d-separation [272]. In this context, methods of CSL infer the CGM based upon statistical concepts, e.g., statistical Conditional Independence tests (CI tests) or scoring functions [142]. Methods using CI tests belong to the class of constraint-based methods of CSL. While widely used in practice, constraint-based methods, such as the well-known PC algorithm [251], struggle with significantly long execution times in the two following settings:

- **Setting (S1) high-dimensional data**: The computational complexity of constraint-based algorithms, such as the PC algorithm, is exponential to the number of variables within the CGM in the worst case [251]. Additional data characteristics such as the number of data samples, or in the case of discrete data, the number of classes within the variables, impact the execution time of CI tests [190].
- **Setting (S2) complex distributional characteristics**: The choice for the appropriate CI test is directly determined by the joint probability distribution of the variables within the raw data [42]. Hence, a higher complexity within the raw data, e.g., due to non-linear relationships or mixed discrete-continuous data, yields choices of computationally expensive CI tests.

In these settings, the computational demand results in extensive runtimes of hours or days, e.g., for gene expression data [123], which hinder the application of CSL in practice [150]. However, research publications show the value of CSL in different domains [74, 82, 101, 117, 151].

For example, in the field of genetic research, CSL provides a method for inference of Gene Regulatory Networks (GRNs) based on gene expression data [65]. Thus, the learned CGM provides a graphical representation of the GRN [74] and supports drug design or diagnostics [54, 206]. However, gene expression data is often high-dimensional, resulting in long runtimes of hours or days [123] or requiring feature selection techniques to reduce the number of variables at the risk of information loss.

As a second example, CSL can be used in the manufacturing industry to determine the root causes of failures during production by understanding the causal relationships between sensor readings obtained along the assembly lines [82, 151]. Integrating the learned causal structures within production monitoring tools provides data-driven decision support for effective troubleshooting [101]. In this domain, the data have complex distributional characteristics. Sensor readings are often continuous, while fault and status messages or configuration parameters are discrete. These complex distributional characteristics require appropriate CI tests that cause long runtimes [214, 260], or the data must be transformed, potentially yielding wrong independence decisions [150] to apply fast CI tests.

Thus, tackling the challenge of learning the causal relationships from high-dimensional (S1) and complex (S2) raw data in acceptable runtime is of utmost relevance, given the two current trends to collect more data (T1) from increasingly complex systems (T2).

1.2 The Role of GPUs in Modern Heterogeneous Computing Systems

Modern computing systems shift from purely CPU-centric computing to heterogeneous computing [93]. Traditionally, CPUs served the computational demand in computing systems. While computational demand increased over the years, advances in CPU hardware led to improved performance of CPUs that could fulfill the computational demand [47, 165]. However, the power wall and thermal dissipation power are limiting factors in single-core [92] and multi-core CPUs [93]. To stay within the thermal dissipation power limits, multi-core CPUs would turn off idle cores, giving rise to the term dark silicon [55]. To overcome the limitation of dark silicon, processor architects focus on energy-efficient cores that target specific applications [58].

With heterogeneous computing in mind, a series of application-specific processing chips have been devised; for example, data processing chips in the domain of processing analytical workloads of databases [1, 281], Field-Programmable Gate Arrays (FPGAs) for accelerating cloud search [197], Tensor Processing Units (TPUs) for ML and DL [107], or GPUs for image processing [113]. From a system's architecture point of view, these processing chips are seen as accelerator devices, i.e., co-processors, with application-tailored processing units and separate on-chip memory. In heterogeneous computing systems, GPUs are of particular interest, as they have been increasingly adopted into applications other than image processing, e.g., DL [127], database management systems [17], mining cryptocurrencies [9], or graph processing [276]. This development gave rise to the term General-Purpose Graphics Processing Unit (GPGPU) [220]. One major driver for the GPU's adoption in these applications is the device's efficiency for

compute-intensive and data-parallel processing concerning the energy consumption and the achievable Floating Point Operations per Second (FLOPS) [113]. Typically, GPUs provide several thousand compute cores for parallel processing compared to CPUs, which have up to dozens of compute cores. However, GPUs have several restrictions that stem from their design as a co-processor with dedicated on-chip memory and their focus on parallel processing. Hence, for the efficient use of GPUs the following three challenges must be addressed:

- **Challenge (C1) on-chip memory size**: GPUs require data to reside in their on-chip memory for processing. Commonly, the on-chip memory has a capacity of several dozens of GBs compared to the system's Dynamic Random Access Memory (DRAM), which can store multiple TBs of data.
- **Challenge (C2) data transfer**: In heterogeneous computing systems, GPUs are commonly considered as separate hardware accelerators, i.e., co-processors. Thus, data must be transferred between a GPU and a CPU on a dedicated interconnect. This indirection poses a potential bottleneck, e.g., due to low data transfer rates of the interconnect.
- **Challenge (C3) execution model**: The GPU follows the Single Instruction Multiple Threads (SIMT) execution model [136]. Algorithms executed on the GPU need to adhere to the execution model and define tasks for parallel execution accordingly to exploit the GPU's full potential.

Existing research shows that if these challenges are addressed when designing algorithms for execution on GPUs, a speedup is achievable over CPU-based algorithms [127, 138]. Therefore, we argue that leveraging the parallel computing capabilities of GPUs for data-parallel processing of CSL algorithms is of high interest to address the CSL algorithms' computational demand.

1.3 Research Questions

Current methods for discovering the causal relationships within observational data struggle with long execution times given high-dimensional data (P1) and complex distributional characteristics within the data (P2) [123, 214]. Across industries, enterprises collect high amounts of data (T1) from increasingly complex systems (T2) [270]. Thus, applying constraint-based CSL becomes more challenging for these companies due to increased execution times, of hours, days, or even weeks. Parallel execution of algorithms for constraint-based CSL, notably the PC algorithm, on multi-core CPUs helps to reduce the long execution times by factors linearly related to the number of CPU cores used [123, 224, 234]. However, physical hardware constraints limit the number of cores on a CPU, and thus the achievable speedup. In contrast to dozens of cores on a CPU, GPUs are typically equipped with several thousand computational cores for massively parallel data processing. Further, compared to other co-processors such as TPUs or FPGAs, GPUs are common in many modern heterogeneous computing systems, e.g., see the adoption of GPUs in supercomputers[1]. Hence, we investigate the utilization of GPUs to achieve additional speedup for constraint-based CSL in real-world settings (P1 & P2). While GPUs are suited to providing

[1] https://www.top500.org/

additional speedup when solving data-parallel problems, employing GPUs introduces unique challenges (C1, C2 & C3). Therefore, in this thesis we focus on the following research goal and in that context two distinct research questions:

Research Goal: *Design efficient execution strategies for constraint-based CSL algorithms that leverage the parallel processing power of GPUs to provide fast runtimes in case of high-dimensional data.*

- **Research Question 1 (RQ1):** *How can we improve the runtime of constraint-based CSL on a GPU?*
  Given their availability in modern heterogeneous computing systems and massively parallel data processing power, we argue that GPUs are well suited to improve the runtime of constraint-based CSL. In this context, a parallel execution strategy and a definition of tasks for parallel execution within constraint-based CSL are required, which map to the unique GPU hardware and its SIMT execution model (C3). Thereby, the definition of tasks must reflect the characteristics of different CI tests for complex relationships within the data and different data distributions. Foremost, this requires the design of new algorithms of constraint-based CSL with different CI tests tailored for execution on a GPU and efficient implementations of these algorithms.
- **Research Question 2 (RQ2):** *How can we scale GPU-accelerated constraint-based CSL to arbitrarily large datasets?*
  We argue that high-dimensional datasets can exceed the memory capacity of a single GPU, either as the memory size of the dataset itself or as the memory demand of auxiliary data structures of the CSL algorithm is too high. In this context, a problem decomposition is required that can cope with the limited on-chip memory capacity of a single GPU (C1) while retaining efficient execution on a single or multiple GPUs. Hence, the problem decomposition must result in tasks that expose enough parallelism to saturate all GPUs' compute units while not exceeding the on-chip memory capacity of each GPU. Further, data transfer costs must be minimized (C2), and communication via interconnects with different bandwidth or latency properties should be reflected.

1.4 Contributions

The following section describes the contributions made to reach our research goal and answer the research questions listed (RQ1 & RQ2). This thesis focuses on the PC algorithm, a well-known constraint-based CSL method widely used in practice [150]. Many extensions of the PC algorithm exist to which the results of this thesis can be applied. The individual contributions are detailed below.

1.4.1 Deriving Tasks for Parallel Execution Within the PC Algorithm

As a first contribution, we derive tasks for parallel execution within the context of the PC algorithm using Foster's methodology [61] (Section 4.1). The tasks for parallel execution have different granularities to manage key characteristics stemming from various CI tests. The theoretical considerations address the research question (RQ1) and have influenced several of our publications.

1.4.2 GPU-Accelerated PC Algorithm Covering a Variety of Data Distributions

We contribute three GPU-accelerated algorithms and their implementation for constraint-based CSL to support CI tests with different characteristics, covering a range of data distributions (see Sections 4.2, 4.3, 4.4). We aim to reach our research goal with the introduced algorithms, particularly by addressing the first research question RQ1. In particular, we introduce a GPU-accelerated algorithm for data following the Gaussian distribution model [226] (Section 4.2). This algorithm leverages the GPU's shared memory to synchronize results within thread blocks allowing for early termination. Also, the algorithm relies upon a precomputed correlation data structure to reduce data access. In an experimental evaluation using gene expression datasets, our GPU-accelerated algorithm can outperform existing CPU-based implementations by factors of up to 93.4×.

Further, we present our GPU-accelerated algorithm for discrete data [81] (Section 4.3) that calculates the marginals over contingency tables within units of threads, managing the corresponding auxiliary data structures in GPU global memory. Measuring the execution time on well-known discrete benchmark datasets, we find that our algorithm is up to 6.5 times faster than a naive GPU-based algorithm and up to 54.3 times faster than a CPU-based implementation running in parallel on 40 CPU cores.

Additionally, we propose a GPU-accelerated algorithm for mixed discrete-continuous data and data with non-linear relationships [83] (Section 4.4). The proposed algorithm computes the mutual information based on the nearest neighbors using a pipeline approach mapping each data sample to an individual GPU thread. Furthermore, the algorithm leverages the undefined order of thread block execution to realize a GPU kernel computing local permutations. In an experimental evaluation on synthetic data, we find that depending on the parameter setting, our algorithm is up to 240 times faster for continuous data with non-linear relationships and up to 655 times faster for mixed discrete-continuous data than a parallel CPU-based algorithm running on 8 cores. The material addresses the first research question RQ1 and was published in the following papers:

[81] Hagedorn, C.; Huegle, J.: *GPU-Accelerated Constraint-Based Causal Structure Learning for Discrete Data*. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 2021, pp. 37–45.

[83] Hagedorn, C.; Lange, C.; Huegle, J.; Schlosser, R.: *GPU Acceleration for Information-theoretic Constraint-based Causal Discovery*. In *Proceedings of The KDD'22 Workshop on Causal Discovery*. PMLR, 2022, pp. 30–60.

[226] Schmidt, C.; Huegle, J.; Uflacker, M.: *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)*. ACM, 2018, pp. 19:1–19:10.

The author of this thesis is the first author of the three publications. Johannes Huegle contributed several ideas and detailed sections regarding the theoretical background of causal graphical models and causal structure learning. Constantin

Lange supported the baseline implementation of the paper [83]. Further, the co-authors improved the material and its presentation.

1.4.3 Scaling the GPU-Accelerated PC Algorithm Beyond a Single GPU's Memory Capacity

Furthermore, we contribute concepts and implementations for the execution of constraint-based CSL when the memory capacity of a single GPU is exceeded given large problem sizes (see Chapter 5). The devised concepts and implementations help to reach our research goal, mainly as they address the second research question RQ2. In systems with only one GPU, we propose to split the problem into small-sized blocks, which fit into GPU global memory [225]. This block-based approach with explicit memory management employs a technique to overlap data transfer and GPU kernel execution. In the case of predominantly random data access, we avoid costly stalls due to page faults in a naive Unified Memory (UM)-based version that implicitly manages memory. A high-dimensional gene expression dataset for which data structures exceed a single GPU's memory capacity is used for an experimental evaluation. Under the simplifying assumption that this data follows the Gaussian distribution model, we find that our approach is a factor of 29.6× faster than a naive UM-based GPU approach and is faster than a parallel CPU-based algorithm by a factor of 364×.

For multi-GPU systems, we extend the block-based approach to perform CSL on multiple GPUs [80]. We demonstrate that the proposed multi-GPU approach remains unaffected by differences in the underlying inter-GPU interconnect performance, which can affect a naive, implicitly memory-managed version. In an experimental evaluation, we find that in the multi-GPU setting, our explicit memory-managed approach is a factor of 2.17× faster than the naive UM-based approach. Both approaches address the second research question RQ2. The work was published in the following two papers:

[80] Hagedorn, C.; Huegle, J.: *Constraint-Based Causal Structure Learning in Multi-GPU Environments*. In *Proceedings of the LWDA 2021 Workshops: FGWM, KDML, FGWI-BIA, and FGIR*. CEUR-WS.org, 2021, pp. 106–118.

[225] Schmidt, C.; Huegle, J.; Horschig, S.; Uflacker, M.: *Out-of-Core GPU-Accelerated Causal Structure Learning*. In *Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2020, pp. 89–104.

The thesis author is the first author of both papers. In the first paper, the thesis author implemented the implicitly memory-managed approach. Siegfried Horschig contributed ideas to the block-based approach, which he supported implementing. Johannes Huegle detailed the background section on constraint-based CSL. The co-authors improved the material and its presentation. In the second paper, the thesis author extended the block-based approach to a multi-GPU system and implemented this approach. Johannes Huegle improved the material and its presentation.

1.4.4 Complementary Contributions

In addition to the main contributions covered within this thesis, the thesis author conducted further research in the field of hardware acceleration for CSL.

First, the thesis author proposed a parallel variant of the PC algorithm, which incorporates a load-balancing algorithm using dynamic task mapping at runtime. The work focuses on multi-core CPU systems and effectively mitigates load imbalances faced by existing parallel variants of the PC algorithm with static task distribution. Compared to the existing parallel variants of the PC algorithm, the introduced load-balanced variant achieves a speedup of up to factor 2.4× on a range of gene expression datasets. The material was published in the following paper:

[224] Schmidt, C.; Huegle, J.; Bode, P.; Uflacker, M.: *Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data*. In *Proceedings of The 2019 KDD Workshop on Causal Discovery*. PMLR, 2019, pp. 59–77.

Second, the thesis author proposed an improved GPU-accelerated correlation coefficient computation, leveraging shared memory, which shows execution time gains of up to 58% over variants not using shared memory. The calculation of correlation coefficients is a commonly applied preprocessing step for constraint-based CSL under the assumption that data follows the Gaussian distribution model. With this preprocessing step, each CI test can directly access its required correlation coefficients and skip processing the observational data. The material was published in the following technical report:

[223] Schmidt, C.; Huegle, J.: *Towards a GPU-Accelerated Causal Inference*. In *HPI Future SOC Lab – Proceedings 2017*. Universitätsverlag Potsdam, 2020, pp. 187–194.

Furthermore, the thesis author contributed to the following research in CSL. One research work focuses on implementing a python package containing the thesis author's GPU-accelerated CSL algorithms [16]. Other research publications address the benchmarking of CSL algorithms [98, 99], the application of CSL in the manufacturing domain [82, 101, 102], and causal discovery from mixed data using an information-theoretic approach [100].

[16] Braun, T.; Hurdelhey, B.; Meier, D.; Tsayun, P.; Hagedorn, C.; Huegle, J.; Schlosser, R.: *GPUCSL: GPU-Based Library for Causal Structure Learning*. In *2022 International Conference on Data Mining, ICDM 2022 – Workshops*. IEEE, 2022, pp. 1228–1231.

[82] Hagedorn, C.; Huegle, J.; Schlosser, R.: *Understanding Unforeseen Production Downtimes in Manufacturing Processes Using Log Data-Driven Causal Reasoning*. In *Journal of Intelligent Manufacturing* 33(7), 2022: pp. 2027–2043.

[98] Huegle, J.; Hagedorn, C.; Böhme, L.; Pörschke, M.; Umland, J.; Schlosser, R.: *MANM-CS: Data Generation for Benchmarking Causal Structure Learning from Mixed Discrete-Continuous and Nonlinear Data*. In *WHY-21 @ NeurIPS*. WHY-21, 2021, pp. 1–15.

[99] Huegle, J.; Hagedorn, C.; Perscheid, M.; Plattner, H.: *MPCSL – A Modular Pipeline for Causal Structure Learning*. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 2021, pp. 3068–3076.

[100] Huegle, J.; Hagedorn, C.; Schlosser, R.: *A kNN-based Non-Parametric Conditional Independence Test for Mixed Data and Application in Causal Discovery*. In *ECML-PKDD 2023, accepted*. 2023.

[101] Huegle, J.; Hagedorn, C.; Uflacker, M.: *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 (ICJAI)*. IJCAI, 2020, pp. 5246–5248.

[102] Huegle, J.; Hagedorn, C.; Uflacker, M.: *Unterstützte Fehlerbehebung durch kausales Strukturwissen in Überwachungssystemen der Automobilfertigung*. In *Software Engineering 2021 Satellite Events, Lecture Notes in Informatics (LNI)*. Gesellschaft für Informatik, 2021, pp. 1–2.

Beyond the topic of CSL, the thesis author also published and contributed to research on hardware acceleration in other domains, such as database management systems, or business applications.

[222] Schmidt, C.; Dreseler, M.; Akin, B.; Roy, A.: *A Case for Hardware-Supported Sub-Cache Line Accesses*. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2018, pp. 1–3.

[227] Schmidt, C.; Uflacker, M.: *Workload-Driven Data Placement for GPU-Accelerated Database Management Systems*. In *BTW 2019 – Workshopband*. Gesellschaft für Informatik, 2019, pp. 91–94.

[230] Schwarz, C.; Schmidt, C.; Hopstock, M.; Sinzig, W.; Plattner, H.: *Efficient Calculation and Simulation of Product Cost Leveraging In-Memory Technology and Coprocessors*. In *The Sixth International Conference on Business Intelligence and Technology*. IARIA, 2016, pp. 12–18.

1.5 Outline

This thesis is organized as follows: Chapter 2 provides the theoretical and technical background that lays the foundation for this thesis. This chapter includes a formal description of causality in the context of CSL and an introduction to GPU hardware and programming. We discuss related literature on parallel CSL,

on overcoming a single GPU's memory limit, and multi-GPU computing in heterogeneous computing systems in Chapter 3. Chapter 4 presents the contributed algorithms for GPU-accelerated constraint-based CSL for data following different distribution models. This chapter includes a formal deduction of parallel tasks within the PC algorithm. The parallel tasks are used to derive variants of the PC algorithm tailored to the characteristics of different CI tests considering the GPU hardware. For the proposed variants of the PC algorithm, we include implementation details. In Chapter 5, we propose algorithms that cope with the limited memory capacity of a single GPU and execute on multiple GPUs. Chapter 6 presents an experimental evaluation of the previously described algorithms. The experimental evaluation covers data from real-world use cases to illustrate the applicability in practice. Additionally, we use synthetic data to demonstrate our proposed algorithms' performance gains, scalability, and limitations in a broader range of settings. Lastly, we discuss the results of our experimental evaluation. Chapter 7 describes the limitations of our proposed GPU-accelerated CSL algorithms and their experimental evaluation. Further, we detail opportunities for future work before we conclude this thesis.

2

Background

This chapter provides the necessary theoretical and technical background for the remainder of this thesis.

In Section 2.1, we explain the fundamental concept of Causal Graphical Models (CGMs). CGMs provide a graphical representation to describe causal relationships between a given number of variables, using Direct Acyclic Graphs (DAGs) [90, 188, 194, 251]. Provided with data for a given set of variables, Causal Structure Learning (CSL) enables the derivation of the causal relationships between these variables and consolidates the learned causal relationships to the CGM. In Section 2.2, we introduce CSL and the main techniques to learn the causal structures, respectively constraint-based CSL (see Section 2.2.1), score-based CSL (see Section 2.2.2), hybrid CSL (see Section 2.2.3) and discuss differences in the techniques (see Section 2.2.4). Note that the data used for CSL is often assumed to be observational data only. Yet, extensions of CSL for mixtures of observational and interventional data also exist [110].

We detail one representative algorithm, the PC algorithm [251], which is the main subject of this thesis, in Section 2.3. The PC algorithm is a well-known representative of the class of constraint-based CSL methods, which has attracted numerous extensions or variations in research, e.g., incorporating algorithmic improvements regarding the accuracy, stability, or the relaxation of assumptions [19, 22, 35, 148, 201, 203, 207, 246, 251, 258], focusing on fast execution leveraging hardware for parallel execution [80, 81, 83, 106, 123, 146, 147, 169, 224, 225, 226, 287, 290], or considering different data distributions [39, 87].

In Section 2.4, we elaborate on conditional independence testing. Notably, we detail CI tests for data with the following characteristics. We consider CI tests for data assuming the Gaussian distribution model, for discrete data, and mixed discrete-continuous data and data with non-linear relationships.

Lastly, in Section 2.5, we explain Graphics Processing Units (GPUs) with a focus on unique hardware aspects (see Section 2.5.1) and the corresponding programming techniques (see Section 2.5.2). In recent years, GPUs have changed from being a processor solely for video and graphics processing to an accelerator for many compute-intensive workloads, e.g., in the High-Performance Computing (HPC) domain [56], in enterprise computing [17, 68, 230], or for Machine Learning (ML) tasks [127].

2.1 Causal Graphical Models (CGMs)

CGMs are a recognized formalization for the concept of causality [90, 142, 187, 188, 189, 251]. Let a CGM be defined as a pair $< \mathcal{G}, \mathcal{P} >$, where $\mathcal{G}$ is a DAG over a set of variables $\mathbf{V}$ and $\mathcal{P}$ is a joint distribution over $\mathbf{V}$ [251].

Graph $\mathcal{G}$, Variables $\mathbf{V}$ and Edges $\mathbf{E}$

Further, let $\mathcal{G}$ be defined as a pair $\mathcal{G} = (\mathbf{V}, \mathbf{E})$, with a finite set of N variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ and a set of edges $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$. Note, in graphical terminology, the variables are represented as vertices, and the terms are used synonymously throughout this thesis. 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., $V_i - V_j$, with $i, j = 1, \ldots, N$ and $i \neq j$.

Skeleton $\mathcal{C}$ and Adjacency Set $adj(\mathcal{G}, V_i)$

Suppose undirected edges replace all directed edges in $\mathcal{G}$. In that case, the resulting graph is called the skeleton $\mathcal{C}$ of $\mathcal{G}$. Based on the skeleton $\mathcal{C}$ of $\mathcal{G}$, we call a pair of vertices (V_i, V_j) adjacent, if there exists an edge $V_i - V_j$ in $\mathcal{C}$. Thus, the adjacency set $adj(\mathcal{G}, V_i)$ of the vertex $V_i \in \mathbf{V}$ in $\mathcal{G}$ contains all vertices $V_j \in \mathbf{V}$ that are adjacent to V_i in $\mathcal{C}$, with $i, j = 1, \ldots, N$ and $i \neq j$.

V-Structure and Completed Partially Directed Acyclic Graph (CPDAG)

Further, let a v-structure be defined as an ordered triple $(V_i, V_j, V_k), i, j, k = 1, \ldots, N, i \neq j, i \neq k, j \neq k$ with directed edges $V_i \rightarrow V_j$ and $V_k \rightarrow V_j$ in $\mathcal{G}$ and $V_k \notin adj(\mathcal{G}, V_i)$ [26]. In this case, two DAGs are equivalent if they share the same skeleton $\mathcal{C}$ and the same v-structures [189]. The corresponding equivalence class of a DAG can be described by a CPDAG [26]. A CPDAG is defined as a completed partially directed acyclic graph containing directed and undirected edges. Furthermore, the CPDAG and all DAGs in the equivalence class share the same skeleton $\mathcal{C}$ and v-structures. Additionally, there exist two DAGs for each undirected edge $V_i - V_j$ in the equivalence class, one for each directed version of the undirected edge $V_i - V_j$ [6].

Graph Concepts: Path, Descendants, Ancestors and Parents

A path in $\mathcal{G}$ of length m from V_i to V_j, denoted by $V_i \mapsto V_j$, is defined as a sequence $V_i = V_1, \ldots, V_m = V_j$ of unique vertices, with $V_k \rightarrow V_{k+1}$ for all $k = 1, \ldots, m-1$ [122]. Based upon the path definition, the descendants of a vertex V_i, denoted by $desc(V_i)$, are all vertices $V_j \in \mathbf{V} \setminus V_i$ for which there exists a path $V_i \mapsto V_j$ [121]. The non-descendants of a vertex V_i, denoted by $nondesc(V_i)$, are defined as $nondesc(V_i) = \mathbf{V} \setminus \{V_i \cup desc(V_i)\}$ [121]. Further, let the ancestors of a vertex V_i, denoted by $an(V_i)$, be defined as all vertices $V_j \in \mathbf{V} \setminus V_i$ for which there exists a path $V_j \mapsto V_i$ [121]. In that case, the parents of V_i, denoted by $pa(V_i)$, are defined as vertices $V_j \in an(V_i)$ with a direct edge to V_i [49].

D-Separation

Furthermore, let the concept of d-separation be defined as follows: Two variables $V_i, V_j \in \mathbf{V}$ are d-separated by a conditioning set of variables $S^{i,j} \subseteq \mathbf{V} \setminus \{V_i, V_j\}$, if for each path between V_i and V_j either there exists a chain $V_i \to Z \to V_j$ or a fork $V_i \leftarrow Z \to V_j$ with $Z \in S^{i,j}$ or there exists a collider $V_i \to W \leftarrow V_j$ with $W \notin S^{i,j}$ and any $desc(W) \notin S^{i,j}$.

Causal Sufficiency and Global Markov Condition

For a DAG $\mathcal{G}$, a directed edge $V_i \to V_j$ represents a causal relationship from V_i to V_j. In this context, if $\mathbf{V}$ contains all direct causal relationships of any variable in $\mathbf{V}$, $\mathcal{G}$ is causally sufficient, i.e., there are no unmeasured confounders [251]. Further, the global Markov condition defines that, for a causally sufficient DAG $\mathcal{G}$ of a CGM, every variable V_i in $\mathbf{V}$ is independent of its non-descendants in $\mathcal{G}$ conditioning on its direct causes, i.e., its parents $pa(V_i)$ in $\mathcal{G}$ [247].

Causal Faithfulness

In addition to the information on direct causal relationships, the DAG $\mathcal{G}$ implies information on the conditional independencies regarding the joint distribution $\mathcal{P}$ and the d-separation criterion on the vertices $\mathbf{V}$ of $\mathcal{G}$ [121, 142, 188, 251]. If V_i, V_j, with $i, j = 1, \ldots, N$ and $i \neq j$, are d-separated by $S^{i,j}$ and the Markov condition is satisfied, then V_i is independent of V_j conditioning on $S^{i,j}$, which we denote by $V_i \perp\!\!\!\perp V_j | S^{i,j}$. Further, we denote the matrix of conditioning sets of all pairs of variables (V_i, V_j) by Sep. Note that in the literature and throughout this thesis, the conditioning set is also referred to as a separation set. If a joint distribution entails only the conditional independencies that are implied by the d-separation condition, $\mathcal{P}$ over $\mathbf{V}$ is called faithful [142].

2.2 Causal Structure Learning (CSL)

Causal Structure Learning (CSL) is the task of deriving as many of the edges in the DAG $\mathcal{G}$ of a CGM as possible, given a set of data D, with $1, \ldots, n$ data samples sampled from $\mathcal{P}$. In this thesis, we consider D to be observational data, meaning that the data stems from sources other than randomized experiments [279]. For example:

- In social sciences, observational data is obtained through surveys, censuses, or administrative records [279].
- In manufacturing, observational data is generated by sensors and process messages [82, 101, 102, 134, 151].
- In genetics, gene expression samples or transcriptomic data represent observational data [125, 206].

Note there also exist methods of CSL for interventional data [282] or mixtures of observational and interventional data [89].

In order to apply CSL [251], generally, the following assumptions are made to relate the DAG $\mathcal{G}$ to the given set of observational data D with variables $\mathbf{V}$. First, the observational data is assumed to be independent and identically

distributed (i.i.d) [91]. In other terms, the stable unit treatment value assumption, explained as follows, holds. The stable unit treatment value assumption defines that the variables of one unit in the population do not causally interact with the variables of another unit of the population [212]. Further, we assume that $\mathcal{G}$ is causally sufficient and faithful and that it fulfills the global Markov condition, as defined in Section 2.1. Based on the global Markov condition and the faithfulness assumption, V_i and V_j are independent given $S^{i,j}$ if and only if V_i and V_j are d-separated by $S^{i,j}$ in $\mathcal{G}$ [142].

Following these assumptions, several techniques have been developed to learn the DAG $\mathcal{G}$ of a CGM from observational data D. A major challenge for many structure learning methods is the non-identifiability of $\mathcal{G}$ from the joint distribution $\mathcal{P}$ [49], beyond the equivalence class of $\mathcal{G}$ [251]. Therefore, methods of CSL learn a CPDAG describing the equivalence class of $\mathcal{G}$ [6, 26]. The most common techniques to CSL can be classified into the following three groups, constraint-based methods, score-based methods, and hybrid methods. For an exhaustive overview of methods for CSL, beyond the three mentioned groups, we refer the reader to the survey by Vowels et al. [275]. Constraint-based methods require the satisfaction of constraints, often considering Conditional Independence (CI) and conditional dependence constraints to hold for the observational data [142]. In this case, these methods apply statistical independence tests to determine the CI relationships in $\mathcal{G}$. Note, there also exist non-parametric constraints or constraints for specific parametric settings [118, 242, 248]. In contrast to constraint-based methods, score-based methods search for $\mathcal{G}$, given the data D, by maximizing or minimizing a score, such as the Bayesian Information Criterion (BIC) [231]. Hybrid methods combine elements from score-based methods with CI tests to learn $\mathcal{G}$ [142].

2.2.1 Constraint-Based CSL

Based on the Markov condition and the faithfulness assumption, constraint-based CSL assumes that when the CI and conditional dependence constraints of a DAG $\mathcal{G}$ hold, $\mathcal{P}$ entails only the CI relationships that are induced by the underlying true DAG [142]. Constraint-based CSL utilizes CI tests to determine the CI relationships between the set of variables $\mathbf{V}$ from observational data D.

Algorithms for constraint-based CSL, for example, `SGS` [250], `IC` [273], or `PC` [249], provide the flexibility to execute different CI tests [142]. The appropriate selection is directly determined through the underlying distribution $\mathcal{P}$ of the observational data D [42]. Choices for CI tests are, for example, the Fisher's z-test [59] for linear Gaussian distribution models, the Pearson's χ^2 test [190] in the multinomial case, or non-parametric tests [100, 157, 214, 286, 289] in the case of mixed discrete-continuous data or data with non-linear functions. Upon executing a CI test, independence for a given pair of variables is determined based on a provided significance value α. As many CI tests are conducted during constraint-based CSL, α can be seen as a tuning parameter [35]. Note that smaller values of α tend to derive sparser graphs.

The application of CI tests introduces the potential for statistical errors, such as type I errors due to calculating approximate CI or limitations in the power of the CI test [142]. While standard approaches do not consider statistical errors, extensions handling statistical errors have been proposed [33, 35, 135, 203, 252, 259, 265].

Besides statistical errors, constraint-based CSL also faces a computational challenge, as the number of CI tests grows exponentially with the number of variables N in $\mathcal{G}$. Under the assumption of a sparse underlying true DAG, common for real-world scenarios, the computational complexity is reduced to be polynomial [109]. Yet, execution times remain a challenge [123].

Note that in this thesis, we focus on global-search constraint-based CSL algorithms. Thus, we do not elaborate on any local-to-global constraint-based approaches [154, 267], which build upon the concept of Markov blankets [186].

2.2.2 Score-Based CSL

Methods of score-based CSL learn the DAG $\mathcal{G}$ or the equivalence class described by the CPDAG of the CGM by optimizing a specified score [49]. Often, these methods use a penalized likelihood score, such as the Bayesian Information Criterion (BIC) [231], to determine the DAG with an optimal score [27].

Finding the DAG is an NP-hard problem for properly defined scores [25, 28], given that the number of DAGs increases super-exponential with the number of variables [208]. Thus, methods to determine the exact DAG, e.g., using partial order covers [184], bounded treewidth [43, 53], or dynamic programming [114, 243], are feasible for up to 30 variables [49]. For larger graphs, exact searches over all possible DAGs are infeasible, and methods using heuristics, searching the CPDAG representing the equivalence class of the DAG $\mathcal{G}$, are proposed.

A well-known example is the Greedy Equivalent Search (`GES`) [27], a two-phase search procedure. In the forward phase, new edges are added to an empty DAG, and in the backward phase, edges are removed. Both addition and removal of edges are executed to optimize the score. Based on the greedy approach, this method may not yield the global optimum of the score for a limited number of data samples n from data D. Yet, if the score has certain properties, it is shown that for $n \to \infty$ the method finds the global optimum with a probability converging to 1 [27]. The number of score evaluations is exponential to the number of variables N in $\mathcal{G}$. Yet, under the assumption of a sparse underlying true DAG, it is shown that the backward phase can have polynomial complexity [29].

Several extensions have been introduced to the `GES` algorithm, e.g., to improve its performance [171, 202], to extend its applicability to incorporate interventional data [89] or to allow for latent variables [181]. Like the tuning parameter α in constraint-based methods, score-based methods use a penalty discount parameter, which penalizes the inclusion of edges and thus influences the sparseness of the learned equivalence class of the CPDAG [199].

Notably, recent research reformulates the traditional score-based combinatorial problem into a continuous optimization problem [293]. The continuous optimization problem is efficiently solved using standard numerical algorithms and outperforms traditional methods.

2.2.3 Hybrid CSL

Hybrid CSL methods combine score-based approaches with elements from constraint-based approaches, i.e., conditional independence testing, to leverage the advantages of the individual techniques [142]. Most hybrid methods [3, 167, 228, 229, 245, 269] reduce the search space over all possible DAGs

of a score-based approach by first applying CI tests or variable selection methods [167]. Therefore, the first step of these hybrid methods produces an estimated skeleton graph $\mathcal{C}$ or an estimated conditional independence graph $\mathcal{CIG}$, a supergraph of $\mathcal{C}$. In the next step, a score-based method, such as `GES` [27], takes $\mathcal{C}$ or $\mathcal{CIG}$ as input and outputs the estimated DAG $\mathcal{G}$ or CPDAG of the CGM.

2.2.4 Comparing CSL Approaches

Several studies and surveys compare CSL approaches from the three main groups, constraint-based, hybrid, and score-based methods [49, 74, 79, 91, 110, 199, 234, 235, 244]. Scutari et al. [235] run a set of experiments on synthetic and real-world climate data focusing on CSL without latent or missing variables. They aim to empirically determine whether any method, respectively group of methods, has a clear advantage over the others. They find that a score-based approach using a tabu-search outperforms other considered methods in most cases. However, there is no general pattern on the most suitable approach, which is in line with another empirical study [91].

The simulation study by Heinze-Deml et al. [91] shows that algorithms from the same group tend to behave similarly, forming clusters. Besides algorithms from the three main groups, the study also considers an approach based on structural equation models with additional restrictions, i.e., the LiNGAM algorithm [241] and a technique that exploits invariance properties, i.e., the backShift algorithm [211]. Further, the study includes extensions of CSL, which allow for hidden variables, handle interventional data, and cope with cycles. The empirical evaluation underlines that both constraint-based and score-based methods have unique advantages and disadvantages.

In particular constraint-based methods, such as the PC algorithm [251], have the following advantages. The algorithm allows the exchange of the applied CI test making it widely applicable given the faithfulness assumption holds [74]. The appropriate CI test for given observational data D is directly determined by the underlying distribution $\mathcal{P}$ [42]. Additionally, numerous extensions exist, for example, to handle hidden variables [36] or interventional data [143, 144, 256] or to address the parallel execution on modern multi-core CPU architectures for faster execution times [126, 146, 147, 224, 234, 254]. Yet, in the finite sample case, the application of CI tests can introduce statistical errors,and constraint-based methods determine the CPDAG of the CGM, as they cannot orient all edges in $\mathcal{G}$ [74, 79]. Despite parallel extensions targeting modern multi-core CPU architectures, long execution times remain a challenge for the application in practice, particularly for high-dimensional settings [126].

In contrast, score-based methods are less prone to propagate statistical errors than constraint-based methods [142] as they consider the entire graphical structure at once [115]. Further, applying heuristics to search for the equivalence class of the true underlying DAG as in the `GES` algorithm [27], score-based methods are often faster than constraint-based methods [235]. Yet, exact score-based methods are known to be NP-hard or NP-complete [25, 28]. In addition, it is known that greedy algorithms are P-complete [5] and thus cannot benefit significantly from parallel execution [75].

2.3 Path Consistency (PC) Algorithm

In the following, we explain the PC algorithm [249], a well-known representative of the constraint-based CSL methods. We focus on the order-independent variant, called PC-stable [35] (see Section 2.3.1), which is the foundation for parallel extensions of the PC algorithm [123, 234] (see Section 2.3.2). The PC algorithm executes two steps to determine the CPDAG describing the equivalence class of $\mathcal{G}$, given observational data D with N variables $\mathbf{V}$ [35].

In the first and most time-consuming step [169], the algorithm conducts an adjacency search utilizing CI tests to determine the skeleton $\mathcal{C}$ of the CPDAG. In addition, the first step outputs the matrix of separation sets Sep. The matrix of separation sets Sep contains a separation set $S^{i,j}$ for all independent pairs of variables (V_i, V_j) with $i, j = 1, \ldots, N$ and $i \neq j$, such that the pair (V_i, V_j) is d-separated by $S^{i,j}$. Note, in literature, the adjacency search is sometimes referred to as skeleton discovery.

In the second step, the edges within skeleton $\mathcal{C}$ are oriented. Therefore, at first, the unshielded triples, defined as a triple (V_i, V_j, V_k), with $i, j, k = 1, \ldots, N$ and $i \neq j, i \neq k, j \neq k$, for which the pairs (V_i, V_j) and (V_j, V_k) are adjacent and (V_i, V_k) is not adjacent in $\mathcal{C}$, are oriented as v-structures if and only if $V_j \notin S^{i,k}$ [251]. Next, further orientation rules, e.g., cf. [36, 109, 251], are applied to orient as many edges as possible.

2.3.1 PC-Stable

The PC-stable is an extension by Colombo et al. [35], introducing an order-independent variant of the PC algorithm's adjacency search as outlined in Algorithm 1 (see p. 18). The adjacency search of the PC-stable takes as input a vertex set $\mathbf{V}$ representing the N variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ (see Section 2.1), a significance level for each CI test α, a function for the CI test $CI(\ldots)$, and

symbol	description
$\mathcal{G}$	DAG of a CGM (see Section 2.1)
$\mathcal{C}$	skeleton graph, i.e., undirected version of $\mathcal{G}$
$\mathbf{V}$	set of vertices representing N variables $\mathbf{V} = \{V_1, \ldots, V_N\}$
D	observational data with n data samples
l	integer of current level within PC-stable
α	significance level
p	p-value
$S^{i,j}$	single separation set of two variables V_i,V_j
$\mathbf{S}^{i,j,l}$	list of all possible separation sets for V_i,V_j in level l
Sep	matrix of separation sets of dimension $N \times N$ containing $S^{i,j}$ that determined $(V_i$,$V_j)$ independent
$a(V_i)$	set containing the adjacent variables of V_i
$adj(\mathcal{C}, V_i)$	adjacency set of V_i given $\mathcal{C}$
$CI(\ldots)$	function of the CI test
$\max_{i=1,\ldots,N}\{\lvert adj(\mathcal{G}, V_i)\rvert\} - 1$	maximum level l reachable within PC algorithm given $\mathcal{G}$

Table 2.1: Notation table for CGM and PC-stable.

Algorithm 1 Adjacency search of PC-stable algorithm according to [35]
Input: Vertex set $\mathbf{V}$, significance level α, CI test $CI(\dots)$, observational data D
Output: Estimated skeleton $\mathcal{C}$, matrix of separation sets Sep

1: Start with fully connected skeleton $\mathcal{C}$ given $\mathbf{V}$ and $l \leftarrow -1$
2: **repeat**
3: $l \leftarrow l + 1$
4: **for all** Vertices V_i in $\mathcal{C}$ **do**
5: Let $a(V_i) \leftarrow adj(\mathcal{C}, V_i)$;
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 separation set $S^{i,j} \subseteq a(V_i) \setminus \{V_j\}$ with $|S^{i,j}| = l$.
11: Let $\mathrm{p}(V_i, V_j | S^{i,j}) \leftarrow CI(V_i, V_j, S^{i,j}, D)$
12: **if** $\mathrm{p}(V_i, V_j | S^{i,j}) \geq \alpha$ **then**
13: Delete edge $V_i - V_j$ from $\mathcal{C}$;
14: Set $Sep_{i,j} \leftarrow S^{i,j}$;
15: **end if**
16: **until** edge $V_i - V_j$ is deleted in $\mathcal{C}$
17: or all $S^{i,j} \subseteq a(V_i) \setminus \{V_j\}$ with $|S^{i,j}| = l$ were chosen
18: **until** all adjacent vertices V_i, and V_j in $\mathcal{C}$ such that
19: $|a(V_i) \setminus \{V_j\}| \geq l$ were considered
20: **until** each adjacent pair (V_i, V_j) in $\mathcal{C}$ satisfies $|a(V_i) \setminus \{V_j\}| \geq l$
21: **return** $\mathcal{C}$, Sep

observational data D. After execution, the algorithm outputs the estimated skeleton $\mathcal{C}$ of $\mathcal{G}$, and a matrix of the corresponding separation sets Sep.

In brief, the algorithm applies CI tests to every pair of variables (V_i, V_j) in $\mathcal{C}$ given all possible separation sets $S^{i,j}$, with the size of the current level l, constructed from adjacent variables of V_i without V_j (see Algorithm 1 lines 2–20). We denote the list of all possible separation sets for a pair of variables (V_i, V_j) in the specific level l with $\mathbf{S}^{i,j,l}$. If the CI test determines independence for the pair of variables (V_i, V_j) given a separation set $S^{i,j}$, the corresponding edge $V_i - V_j$ is removed from the skeleton $\mathcal{C}$ and the separation set $S^{i,j}$ stored (see lines 12–15). Thus, the adjacency search of the PC-stable, as well as, the original PC algorithm, only conducts CI tests for pairs of variables (V_i, V_j) given separation sets $S^{i,j}$ with a size of $l = 0$ up to the maximum size of the adjacency sets of the variables in the underlying true DAG of the CGM, i.e., up to $\max_{i=1,\dots,N}\{|adj(\mathcal{G}, V_i)|\} - 1$. Hence, under the assumption of a sparse underlying true DAG, the algorithm's complexity becomes polynomial [109].

In detail, the algorithm starts with a fully connected skeleton $\mathcal{C}$ (see line 1 of Algorithm 1). For every level $l = 0, \dots, \max_{i=1,\dots,N}\{|adj(\mathcal{G}, V_i)|\} - 1$ the adjacency sets $a(V_i) = adj(\mathcal{C}, V_i)$ are computed for each variable V_i with respect to the current skeleton $\mathcal{C}$. The adjacency sets are stored in a separate data structure to ensure the order-independence of the PC-stable [36] (see lines 4–6).

Next, each pair of variables (V_i, V_j) adjacent in $\mathcal{C}$, and for which there exist enough adjacent variables to construct a separation set $S^{i,j}$ of size l, i.e., $|a(V_i) \setminus \{V_j\}| \geq l$ is considered for independence testing. Subsequently, for the pairs of variables (V_i, V_j), fulfilling the above criteria, CI tests are conducted

repeatedly with a changing separation set $S^{i,j}$. The CI tests are conducted until all possible separation sets $S^{i,j}$ drawn from $\mathbf{S}^{i,j,l}$ in the current level l have been considered, or the pair of variables was determined independent. The pair of variables is determined independent in the case that the p-value $\mathrm{p}(V_i, V_j | S^{i,j})$ calculated within the CI test is larger or equal to the given significance level α (see line 12 of Algorithm 1). In this case, the corresponding edge is deleted from the skeleton $\mathcal{C}$, and the separation set $S^{i,j}$ is stored in the matrix of separation sets at the corresponding position $Sep_{i,j}$ (see lines 13–14). Note, for $l = 0$, no separation set, i.e., $S^{i,j} = \emptyset$, is considered in the CI test, and an indicating flag is stored accordingly. Once, all pairs of adjacent variables (V_i, V_j) in the current version of the skeleton $\mathcal{C}$ have been considered, the algorithm increases l by one.

This procedure continues until l reaches the maximum size of the adjacency sets of the variables in the underlying true DAG of the CGM, i.e., $l = \max_{i=1,\ldots,N}\{|adj(\mathcal{G}, V_i)|\} - 1$. Then, the adjacency search returns the estimated skeleton $\mathcal{C}$ and a matrix of separation sets Sep, which are the basis for the second phase of the PC algorithm, respectively the PC-stable, the edge orientation. For detail, on the edge orientation, we refer the reader to Spirtes et al. [251] or others [36, 109].

2.3.2 Parallel Extensions of the PC Algorithm

Several extensions to the order-independent variant of the PC algorithm address parallel execution on multi-core CPUs [123, 146, 169, 224, 234]. In general, parallel execution requires synchronization and communication between parallel execution units [61], which introduces overhead that limits speedup. Thus, minimizing this overhead is a goal when designing parallel algorithms. In the case of parallel extensions of the PC algorithm, this is achieved by defining a parallel execution strategy, which executes each pair of variables (V_i, V_j), respectively each edge, in parallel, on an individual execution unit, e.g., a CPU core [146, 234].

For example, the parallelPC algorithm [123] executes the loop iterating the adjacent pairs of variables (see lines 7 – 18 of Algorithm 1) in parallel in the following way. First, the pairs of variables are split into batches equal to the number of parallel execution units. Next, these batches are distributed to the parallel execution units. Then, each execution unit processes its batch, executing lines 7 – 18 of Algorithm 1. Upon completion, the results from each batch are merged in a synchronization step before the subsequent level is executed.

Beyond the parallel execution of the adjacency search, the framework for parallel constraint-based structure learning [234] proposes processing the less time-consuming orientation of unshielded triples to v-structures in parallel.

2.4 Conditional Independence Testing

Conditional independence testing is at the core of methods for constraint-based CSL [251]. Considering the PC algorithm's adjacency search (see Algorithm 1), CI tests contribute most of the computation time [123]. The suitable CI test for given data D is directly determined by the underlying data distribution [42] and plugged into the PC algorithm. Common choices of CI tests are Fisher's z-test [59] for data distributed according to the Gaussian distribution model or Pearson's χ^2 test [190] for data following the discrete distribution model. Note

that in practice, data is oftentimes transformed to fit one of the two distribution models [150]. Yet, there exists a wide range of CI tests for mixed data [100, 157, 266, 286], non-linear settings [200, 291] or other distributions [87, 195, 236, 238, 288], or using non-parametric approaches [48, 67, 96, 214, 237, 260, 289].

2.4.1 Gaussian Distribution Model

In the Gaussian case, we assume that the set of variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ is multivariate normal distributed. In this case, standard statistical theory, e.g., cf. [46], defines a pair of variables (V_i, V_j) with $V_i, V_j \in \mathbf{V}$ and $i \neq j$ as independent if and only if the correlation coefficient $\rho(V_i, V_j)$ is equal to zero. Whereas, conditional independence for a pair of variables (V_i, V_j) given a conditioning set $S^{i,j} \subset \mathbf{V} \setminus \{V_i, V_j\}$ holds if and only if the partial correlation coefficient $\rho(V_i, V_j | S^{i,j})$ is equal to zero [121, 277]. Thus, testing a pair of variables for independence requires to calculate either $\rho(V_i, V_j)$ if no conditioning set is given, or $\rho(V_i, V_j | S^{i,j})$ if a conditioning set is considered.

Based on observational data D with n data samples, the sample correlation $\hat{\rho}(V_i, V_j)$ for the pair of variables (V_i, V_j), with $i, j = 1, \ldots, N$ and $i \neq j$, is defined by

$$\hat{\rho}(V_i, V_j) = \frac{\sum_{q=1}^{n} \left(V_i^{(q)} - \overline{V_i}\right)\left(V_j^{(q)} - \overline{V_j}\right)}{\sqrt{\sum_{t=1}^{n} \left(V_i^{(t)} - \overline{V_i}\right)^2} \sqrt{\sum_{u=1}^{n} \left(V_j^{(u)} - \overline{V_j}\right)^2}} \tag{2.1}$$

with $\overline{V_i}$, with $i = 1, \ldots, N$, defined by

$$\overline{V_i} = \sum_{q=1}^{n} V_i^{(q)},$$

where $V_i^{(q)}$ denotes the q-th entry, i.e., data sample, of the corresponding variable [129]. The partial correlation coefficient $\rho(V_i, V_j | S^{i,j})$ is efficiently estimated using the inverse of the corresponding correlation matrix $Cor(V_i, V_j, S^{i,j})$ that is described as follows. The correlation matrix $Cor(V_i, V_j, S^{i,j})$ has a dimension of $(2 + |S^{i,j}|) \times (2 + |S^{i,j}|)$. An entry of the correlation matrix $Cor(V_i, V_j, S^{i,j})_{a,b}$ contains the sample partial correlation coefficients $\hat{\rho}(V_a, V_b)$ for all $V_a, V_b \in S^{i,j} \cup \{V_i, V_j\}$ [277]. Accordingly, the sample partial correlation coefficient is defined by

$$\hat{\rho}(V_i, V_j | S^{i,j}) = \frac{-r_{i,j}}{\sqrt{r_{i,i} r_{j,j}}}, \tag{2.2}$$

with $r_{i,j} = Cor(V_i, V_j, S^{i,j})^{-1}_{i,j}$. In case the determinant of the correlation matrix is equal to zero, the pseudo-inverse can be computed using the Moore-Penrose generalized matrix inverse [192].

Standard statistical hypothesis testing theory is applied to test whether the correlation coefficient or partial correlation coefficient is equal to zero or not [130]. In the following, we assume that $\hat{\rho}(V_i, V_j | S^{i,j}) = \hat{\rho}(V_i, V_j)$ for $S^{i,j} = \emptyset$ and refer to the partial correlation coefficient only. According to Fisher's z-test [59], the Fisher z-transformation is applied to the sample partial correlation coefficient to test the significance of the difference between two partial correlation coefficients:

$$Z(V_i, V_j|S^{i,j}) = \frac{1}{2}\log\left(\frac{1+\hat{\rho}(V_i, V_j|S^{i,j})}{1-\hat{\rho}(V_i, V_j|S^{i,j})}\right). \tag{2.3}$$

Based on the z-transformation, the p-value is calculated using

$$\mathrm{p}(V_i, V_j|S^{i,j}) = 2\left(1-\Phi\left(\sqrt{n-|S^{i,j}|-3}\left|Z(V_i, V_j|S^{i,j})\right|\right)\right), \tag{2.4}$$

where $\Phi(\cdot)$ denotes the cumulative distribution function of a standard normal distribution. Thus, the null-hypothesis $\hat{\rho}(V_i, V_j|S^{i,j}) = 0$ is rejected against the two-sided alternative $\hat{\rho}(V_i, V_j|S^{i,j}) \neq 0$ if given the significance level α for the p-value it holds that $\mathrm{p}(V_i, V_j|S^{i,j}) < \alpha$.

2.4.2 Discrete Distribution Model

For discrete data, we assume that the set of variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ has the corresponding discrete domains $\{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$. Hence, when examining the hypothesis on conditional independence for a pair of variables (V_i, V_j) with $V_i, V_j \in \mathbf{V}$ given conditioning set $S^{i,j}$, we take the corresponding discrete domains $\mathcal{V}_i, \mathcal{V}_j$, and $\mathcal{S}^{i,j}$ into account. Independence $V_i \perp\!\!\!\perp V_j|\ S^{i,j}$ holds if for all realization values or vectors of values v_i, v_j, and $s^{i,j}$ with $(v_i, v_j, s^{i,j}) \in \mathcal{V}_i \times \mathcal{V}_j \times \mathcal{S}^{i,j}$ holds that $P(V_i = v_i, V_j = v_j|S^{i,j} = s^{i,j}) = P(V_i = v_i|S^{i,j} = s^{i,j}) \cdot P(V_j = v_j|S^{i,j} = s^{i,j})$. Within the context of the discrete distribution model, the properties of marginals over contingency tables are a common choice in statistical hypothesis testing for conditional independence [2, 153].

For example, consider the well-known Pearson's χ^2 test [190]. The CI test builds upon examining an overall discrepancy between the expected frequency and the actual frequency. In this context, the expected frequency is given by

$$E_{v_i v_j s^{i,j}} = \frac{f_{v_i + s^{i,j}} \cdot f_{+ v_j s^{i,j}}}{f_{++ s^{i,j}}}, \tag{2.5}$$

where $V_i = v_i$, $V_j = v_j$, and $S^{i,j} = s^{i,j}$, and the actual frequency is $f_{v_i v_j s^{i,j}}$. Note that $f_{v_i v_j s^{i,j}}$ represents the corresponding entry in the contingency table. Further, let the marginal with respect to V_i be denoted by,

$$f_{+ v_j s^{i,j}} = \sum_{\mathcal{V}_i} f_{v_i v_j s^{i,j}}, \tag{2.6}$$

the marginal with respect to V_j be denoted by

$$f_{v_i + s^{i,j}} = \sum_{\mathcal{V}_j} f_{v_i v_j s^{i,j}}, \tag{2.7}$$

and the marginal with respect to V_i, V_j be denoted by,

$$f_{++ s^{i,j}} = \sum_{\mathcal{V}_i} \sum_{\mathcal{V}_j} f_{v_i v_j s^{i,j}}, \tag{2.8}$$

such that f_{+++} is equal to the total data sample size n. Moreover, let the size of the domains of $\mathcal{V}_i$, $\mathcal{V}_j$, and $\mathcal{S}^{i,j}$ be denoted by $|\mathcal{V}_i|$, $|\mathcal{V}_j|$, and $|\mathcal{S}^{i,j}|$, respectively. Accordingly, the test statistic of Pearson's χ^2 test is:

$$\chi^2\left(V_i, V_j | S^{i,j}\right) = \sum_{V_i \widehat{=} \mathcal{V}_i, V_j \widehat{=} \mathcal{V}_j, S^{i,j} \widehat{=} \mathcal{S}^{i,j}} \frac{\left(N_{v_i v_j s^{i,j}} - E_{v_i v_j s^{i,j}}\right)^2}{E_{v_i v_j s^{i,j}}}, \tag{2.9}$$

which equals zero whenever $E_{v_i v_j s^{i,j}}$ equals zero.

Under the null hypothesis of conditional independence, i.e., $V_i \perp\!\!\!\perp V_j \mid S^{i,j}$, the test statistic $\chi^2\left(V_i, V_j | S^{i,j}\right)$ is asymptotically distributed as χ^2_{df} with the degrees of freedom df defined by

$$df = (|\mathcal{V}_i| - 1)(|\mathcal{V}_j| - 1) \cdot |\mathcal{S}^{i,j}|. \tag{2.10}$$

Consequently, the p-value is calculated by

$$\mathrm{p}(V_i, V_j | S^{i,j}) = 1 - \mathrm{F}(\widehat{\chi}^2), \tag{2.11}$$

with F defined as the cumulative distribution function of χ^2_{df} and $\widehat{\chi}^2$ defined as the calculated statistic derived from the marginals over the observed frequencies in the corresponding contingency table.

Thus, the null-hypothesis $\hat{\rho}(V_i, V_j | S^{i,j}) = 0$ is rejected against the two-sided alternative $\hat{\rho}(V_i, V_j | S^{i,j}) \neq 0$ if, given the significance level α, for the p-value it holds that $\mathrm{p}(V_i, V_j | S^{i,j}) < \alpha$.

Apart from Pearson's χ^2 test [190], other CI tests also build upon examining the difference between the marginals over contingency tables [2, 153]. For example, a permutation test [268] or a test based on stochastic complexity [156].

2.4.3 Mixed Discrete-Continuous Data and Data with Non-Linear Relationships

In the case of mixed discrete-continuous data, we assume that the set of variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ follows the idea of the mixed additive noise model [98]. Thus, $\mathbf{V}$ contains continuous variables V^{Con} and discrete variables V^{Dis} and it holds that $V^{Con}, V^{Dis} \subset \mathbf{V}$, $V^{Con} \cup V^{Dis} = \mathbf{V}$ and $V^{Con} \cap V^{Con} = \emptyset$. Furthermore, we assume that the functional relationships between continuous variables can be linear and non-linear.

While there exists a range of CI tests that cover either non-linear continuous [48, 67, 214, 260, 289, 291] or linear mixed discrete-continuous [266] data, only few recent works cover mixed discrete-continuous data with non-linear relationships [97, 100, 157, 286]. The CI tests employ kernel-based techniques, following the reproducing kernel Hilbert spaces [66], e.g., [48, 260, 289, 291], likelihood-ratio tests, e.g., [266] or information theory [67, 97, 100, 157, 214, 286].

For mixed discrete-continuous data or data with non-linear relationships, this thesis focuses on information-theoretic CI tests, particularly [97, 100, 214]. These CI tests consider information-theoretic measures, e.g., the Conditional Mutual Information (CMI) that characterizes the information flow between systems [94], to test for independence between a pair of variables $(V_i, V_j | S^{i,j})$ [14]. The estimation of the CMI from data D depends on the underlying data distribution [71]. CMI estimators have been developed for the continuous case [63, 116, 271] and extended to the mixed discrete-continuous case [71, 161, 210]. However, CMI estimation lacks theory on finite sample behavior, and therefore for application in CI tests, the null distribution must be determined, for example using permutation-based schemes [48, 97, 100, 214].

Algorithm 2 CI test using a local nearest-neighbor permutation scheme [214]
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 data samples n, estimator function $estimator_{knn}(\dots)$
Output: p-value p, test statistic cmi

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

In this case, CI tests based on information-theoretic measures combine an appropriate CMI estimator with a permutation scheme to determine the null distribution. For example, the information-theoretic CI test for time series with non-linear multivariate normal distributed data `CMIknn` [214] builds upon the CMI estimator by Frenzel and Pompe [63]. The information-theoretic CI tests for mixed discrete-continuous data with non-linear relationships [97, 100] build upon the CMI estimators by Gao et al. [71] or Mesner and Shalizi [161]. The three CI tests utilize a local nearest-neighbor permutation schema. Furthermore, the employed CMI estimators also build upon nearest-neighbor approaches in all three cases.

Algorithm 2 outlines the CI test based on a local nearest-neighbor permutation scheme for data with non-linear relationships by Runge [214]. The algorithm starts by computing for each data sample a with $a \in \{1, \dots, n\}$ 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 data sample to all other data samples (see Algorithm 2 lines 1–3).

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 data samples (see lines 5–10). Next, the data samples are iterated in the previously determined order (see lines 11–18). For each data sample, one of its nearest neighbors is drawn from the list and placed at the data sample's position in the local permutation $\hat{D}$. The drawing mechanism chooses the nearest neighbor of the current data sample that a previous data sample has not drawn unless it is the k^{th}_{perm}-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 $estimator_{knn}(\ldots)$ (see Algorithm 2 line 21). In the case of the two CI tests for mixed discrete-continuous data with non-linear relationships [97, 100], the estimator function is exchanged with the CMI estimator by Gao et al. [71] or Mesner and Shalizi [161]. The calculated CMI value is stored for each permutation in a list $\hat{cmi}$.

After the CMI values for all permutations are estimated, the CMI value *cmi* is computed from the original non-permuted data samples (see 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 the value of *cmi* as the test statistic.

2.5 Graphics Processing Units

In recent years, the advancement in GPU hardware has fueled the adoption of GPUs as an accelerator in many domains beyond its classical purpose for graphics and video processing [17, 56, 127, 220]. Given its original purpose, the GPU has a hardware layout (see Section 2.5.1) tailored for parallel and throughput-oriented processing [113]. Efficient execution of code on a GPU requires reflecting these unique hardware characteristics. Therefore, programming frameworks [4, 170, 257], such as `CUDA` [170] targeting NVIDIA GPUs (see Section 2.5.2), have been designed to support development tailored for the device-specific execution model, i.e., Single Instruction Multiple Threads (SIMT) [136].

2.5.1 GPU Hardware in Heterogeneous Systems

Figure 2.1 (see p. 25) shows a simplified heterogeneous system architecture with a host system and multiple graphics cards, referred to as devices. The illustrated graphics cards depict the Ampere architecture of NVIDIA's A100 GPU [31].

The host system consists of one or more CPUs and provides a large amount of DRAM, often referred to as main memory or host memory. Nowadays, each CPU has multiple cores [92]. While the CPU is connected to DRAM via the memory bus, CPUs are connected with each other through a separate interconnect, e.g., Intel Ultra Path Interconnect (UPI) [166] or X-Bus 4B [32] (see Figure 2.1 Interconnect I). Within the heterogeneous system, the host is connected to one or more graphics cards via a dedicated interconnect, e.g., PCI-E [18],

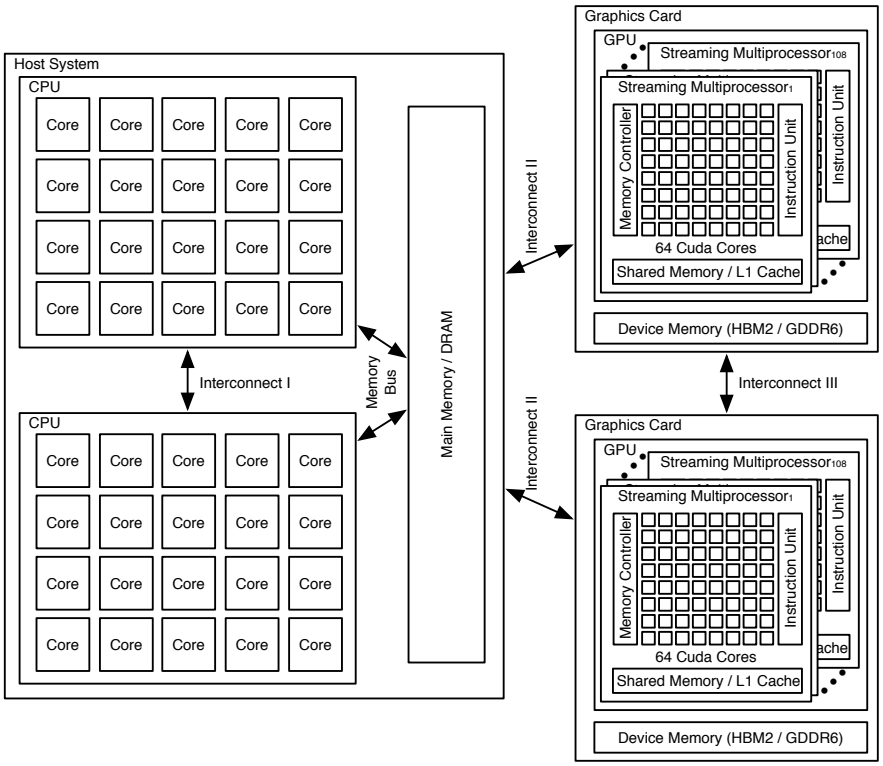


Fig. 2.1: Exemplary system architecture of a heterogeneous hardware setup with a *Host System* (left) containing multiple CPUs and multiple *Graphics Cards* (right) as so-called *devices* with a GPU each. Throughout this thesis, we use the term GPU synonymous with the term graphics card.

NVLink [60], or Infinity Fabric [131] (see Figure 2.1 Interconnect II). The interconnect is used for all data transfer between the host system and any device. If the system contains multiple graphics cards, the graphics cards can have separate interconnects for dedicated inter-graphics card communication (see Figure 2.1 Interconnect III). In this case, not all data transfer necessarily goes through the host system. Note different interconnect technologies can be used for any of the mentioned interconnects.

Each graphics card has a GPU and dedicated device memory, e.g., based on High Bandwidth Memory (HBM) version 2 or GDDR6, referred to as global memory. The global memory is accessible by all processing units within the GPU. The GPU contains multiple Streaming Multiprocessors (SMs), e.g., 108 SMs in the A100. Each SM consists of multiple `CUDA` cores, e.g., 64 for the A100, has a dedicated instruction unit, a memory controller, and contains a separate memory. The memory within the SM is fast to access by the `CUDA` cores, yet, it is limited in size, with a capacity between 64kB to 192kB. This memory is used jointly as L1 cache and shared memory. In contrast to the L1 cache, shared memory allows for explicit data allocation by the developer.

In the architectural diagram (see Figure 2.1), the CUDA cores represent the actual processing units that execute operations of scheduled threads. For simplicity, further hardware details such as underlying integer or floating-point processing units, other functional units, and registers per CUDA core are omitted. While we illustrate the difference between the graphics card containing both device memory and the GPU (see Figure 2.1), we use the term GPU synonymous with the term graphics card in the remainder of the thesis.

The depicted heterogeneous hardware setup contains multiple memories in different locations. From a GPU perspective, access to the host system's DRAM has the highest latency and low throughput due to explicit data transfer to the device via Interconnect II. Yet, DRAM provides the largest capacity within the system. The device memory has the largest capacity within the device, e.g., up to 80 GB, and offers faster access to data than the host system's DRAM. Other memories on the device are considerably smaller in capacity but have higher latency and throughput. For example, the L2 cache, the L1 cache, shared memory, or registers. The L2 cache, which is not depicted in Figure 2.1, is visible for the entire device and has the second-largest capacity on the device. L1 cache and shared memory share the same physical memory, which is smaller than the L2 cache, and they exist per SM. Hence, the L1 cache and shared memory are only visible to its corresponding SM. The smallest units to keep data are the registers per CUDA core, which provide the fastest access to data.

Execution on a GPU follows the SIMT execution model. Meaning that multiple CUDA cores, commonly 32, process the same scheduled instruction in lockstep. As a GPU is a throughput-oriented device [113], each CUDA core executes the instruction on different data loaded to the CUDA core's register with a previous load instruction.

2.5.2 CUDA Programming Framework

Specific frameworks and languages are developed for simplified development in heterogeneous environments with GPUs. The CUDA programming framework and the CUDA C language are developed for development and code execution on NVIDIA GPUs [170]. CUDA follows the general GPU programming pattern that combines host code, executed by the host system's CPU, specific instructions for memory management of the device's global memory, and GPU kernel code [220].

Host code contains any computation executed on the CPU, memory management instructions and also the GPU kernel launch operation. Instructions for memory management are instructions for explicit data allocations on the GPU, explicit data transfer instructions between host and device, or instructions for Unified Memory (UM) [217]. Note, data explicitly allocated on and transferred to one GPU is only accessible on that particular device. In contrast, data allocated using UM can be accessed by CPU and any GPU, as data is migrated on a page-level granularity to the accessing device transparently by the Memory Management Unit (MMU) [217]. Further, explicit data allocations on a GPU are limited by the device's memory size. In contrast, using UM larger amounts of memory can be allocated. But, the amount of data present at the same time remains limited by the device memory's capacity, requiring the MMU to handle data evictions.

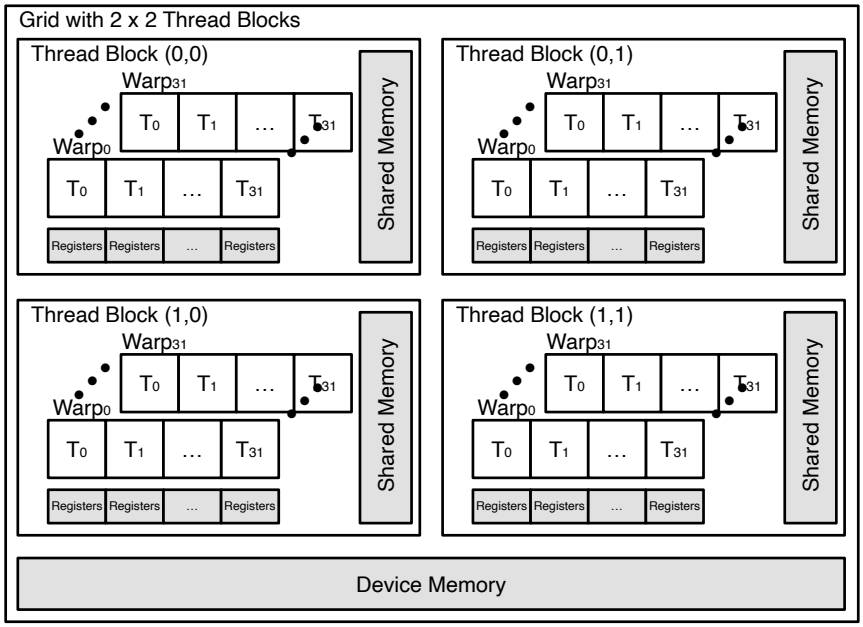


Fig. 2.2: Organization of GPU threads (T) within a GPU kernel in the `CUDA` framework. Each GPU kernel is launched with a grid of thread blocks summarizing GPU threads within warps. Note, some hardware memory (marked in grey) is mapped to illustrate the visibility of memory locations within threads and thread blocks. Device memory is available to all GPU threads, whereas shared memory is only visible to threads within the same thread block. Data in registers is thread-local.

GPU kernels contain the operations executed on the GPU. Upon GPU kernel launch parameters are set, e.g., to specify how the parallel execution is organized, or to define the amount of available shared memory used.

The organization of the parallel execution of a GPU kernel is depicted in Figure 2.2. A GPU kernel is launched with a number of thread blocks TB specified by a three-dimensional grid. In the presented example (see Figure 2.2), a GPU kernel is launched with a 2×2-sized grid. Each thread block can contain up to 1 024 threads T, also organized in a three-dimensional manner. Internally the threads are mapped to warps with 32 threads each. The number of threads per thread block T_{TB} defines the thread block size TBS. Hence, the total number of threads for a GPU kernel T_{kernel} is calculated by

$$T_{kernel} = TB \times T_{TB}. \tag{2.12}$$

When programming, each thread block TB is identified through a three-dimensional id (bx, by, bz) within the grid. Similarly, each thread T is identified within its thread block through a three-dimensional id (tx, ty, tz). These ids map tasks and data to the executing thread blocks and threads.

During GPU kernel execution, threads, warps, and thread blocks are mapped onto the GPU hardware in the following way. Each thread block and its threads are mapped onto one SM. This mapping ensures that the shared memory of the SM is efficiently accessible by all threads within the same thread block. The device memory acts as a global memory accessible by all threads across all thread blocks. Further, each thread has its own register space to store thread-local variables (see Figure 2.2, p. 27). If threads require more private memory, e.g., for local arrays or due to register spilling [162], each thread receives a local share within the device memory. Note, this can impact the performance due to the higher latency when accessing device memory [160].

The threads are executed following the SIMT execution model. This execution model leads to the organization of 32 threads in dedicated warps. Each warp is scheduled on the SM according to its thread block, and the warp's threads process the same instruction. In the case of branching *if-statements*, the warp will execute both conditional branches. Within each branch, the threads that do not follow this branch are disabled [92].

2.6 Summary

In this chapter, we defined the concepts of Causal Graphical Models (CGMs) and Causal Structure Learning (CSL). We detailed the PC-stable algorithm [35], the order-independent variant of the well-known PC algorithm [249], which is the basis for parallel variants of the PC algorithm. PC-stable is a constraint-based method for CSL that builds upon the application of CI tests to learn the CGM. Accordingly, we introduced three different CI tests that cover data with different distributional assumptions and characteristics. This thesis targets to accelerate the PC-stable algorithm in the context of the three CI tests through parallel execution on GPUs. Hence, we described the unique features of GPU hardware that need to be reflected when designing algorithms for execution on a GPU. Further, we introduced the CUDA programming framework [170] with its abstraction to define and map tasks for parallel execution to the GPU.

3

Related Work

This chapter provides an overview of related literature on the parallel execution of algorithms for constraint-based CSL in Section 3.1. Further, this chapter discusses work on GPU processing beyond the memory capacity of a single GPU relevant to our research question (RQ2) in Section 3.2.

In Section 3.1, we review the existing work on algorithms for the parallel execution of constraint-based CSL (see Section 3.1.1). In particular, we distinguish between algorithms that address parallel execution of the PC algorithm on multi-core CPUs (see Section 3.1.2) and algorithms that employ GPUs as an accelerator to achieve fast runtimes (see Section 3.1.3). Further, we consider parallel algorithms for constraint-based CSL that use FPGAs, scale to distributed systems, or employ partitioning schemes (see Section 3.1.4). Note that at the beginning of this thesis, to the best of our knowledge, no published work on co-processor acceleration for constraint-based CSL, e.g., using FPGAs or GPUs existed. Given the focus of this thesis, we do not cover works on parallel execution for techniques to CSL other than constraint-based methods, such as parallel score-based approaches [128, 202], a parallel GPU-accelerated variant of the LiNGAM algorithm [241], called ParaLiNGAM [239], or parallel local-to-global constraint-based CSL algorithms [172, 254].

In Section 3.2, we discuss approaches for out-of-core GPU computing (see Section 3.2.1) and for executing algorithms on multiple GPUs (see Section 3.2.2). Note we use the term *out-of-core* to refer to operations that exceed the GPU's global memory and spill data to the system's DRAM. The out-of-core approaches are relevant to scale CSL to arbitrary-sized datasets independent of the available amount of GPU global memory. To achieve fast runtimes by utilizing more than one GPU, approaches to execution on multiple GPUs are important.

3.1 Parallel Constraint-Based CSL

Research on parallel execution of constraint-based CSL algorithms is a fairly young field, with few publications [106, 123, 126, 146, 147, 224, 234, 287, 290]. The majority of research in constraint-based causal discovery focuses on qualitative improvements regarding the accuracy, stability, or the relaxation of assumptions [19, 22, 35, 39, 87, 201, 203, 207, 246, 251, 258, 266] or its applicability in specific domains [82, 101, 124, 134, 148, 151, 215, 216]. Often, these extensions build upon the same core algorithms also considered for parallel execution,

strengthening the importance of the parallel constraint-based CSL algorithms. Further, parallel execution of algorithms for constraint-based CSL is a valuable approach to address the algorithms' long runtimes, which hinder their application in practice [123]. In the following, we provide an overview of the existing parallel constraint-based CSL algorithms and discuss each algorithm in detail.

3.1.1 Overview of Existing Parallel Constraint-Based CSL Algorithms

algorithm	execution environment	parallelized part	task granularity	task distribution
vertical parallel PC [146, 147]	multi-core CPU	adjacency search	edge, block of variables	static, dynamic
horizontal parallel PC [147]	multi-core CPU, distributed system	adjacency search	subset of data samples	static, dynamic
bnlearn [234]	multi-core CPU	Markov blanket discovery, adjacency search, v-structures	variable, edge	static
parallel-PC [123, 126]	multi-core CPU	adjacency search	edge	static
load-balanced parallel PC [224]	multi-core CPU	adjacency search	edge	dynamic
Fast-BNS [106]	multi-core CPU	adjacency search	CI tests	dynamic
cupc-E [287]	single GPU	adjacency search	CI tests	static
cupc-S [287]	single GPU	adjacency search	CI tests	static
reversed-order PC [290]	single GPU	adjacency search	CI tests	static
FPGA-CDCSF [78]	FPGA	conditioning set filtering	sub-score	static
MrPC [169]	distributed system	adjacency search	edge	dynamic
PCA & PCB [44]	distributed system	adjacency search	edge, data sample	static, dynamic
PEF [77]	multi-core CPU, distributed system	adjacency search	cluster of variables	static
pPC [95]	multi-core CPU, distributed system	adjacency search	cluster of variables	static

Table 3.1: Overview of existing parallel constraint-based CSL algorithms.

Table 3.1 provides an overview of existing parallel constraint-based CSL algorithms. We organize the algorithms into three groups, i.e., multi-core CPU-based

algorithms, GPU-based algorithms, and further algorithms (top to bottom), separated by horizontal lines, and characterize the algorithms along four dimensions relevant to the context of parallel execution.

Groups of Parallel Constraint-Based CSL Algorithms

We group the algorithms for parallel constraint-based CSL according to their primary targeted execution environment. In particular, we consider algorithms that target multi-core CPUs, algorithms for execution on the GPU, and further algorithms that target other co-processors, are designed for distributed systems, or apply partition-based techniques.

Multi-Core CPU-Based Algorithms

The first group, shown at the top in Table 3.1, contains the majority of existing algorithms for parallel constraint-based CSL. These algorithms target multi-core CPUs for parallel execution [106, 123, 126, 146, 147, 224, 234]. We describe each algorithm in detail in Section 3.1.2.

GPU-Based Algorithms

The second group, shown in the middle of Table 3.1, refers to algorithms that apply GPU acceleration to constraint-based CSL [287, 290]. Besides the approaches presented in this thesis, to the best of our knowledge, only three GPU-based constraint-based CSL algorithms exist. We detail the algorithms in Section 3.1.3.

Further Algorithms

The third group, shown at the bottom of Table 3.1, refers to further approaches for parallel execution of constraint-based CSL. These algorithms use other co-processors for acceleration, are designed for distributed systems, or apply partition-based techniques, which are also suited for distributed systems. Each approach is explained in Section 3.1.4.

Dimensions of Parallel Constraint-Based CSL Algorithms

We characterize each algorithm according to the following dimensions; the targeted *(i) execution environment*, the *(ii) parallelized part* of the algorithm, the *(iii) task granularity*, and the *(iv) task distribution* strategy.

(i) Execution Environment

Each algorithm is designed to take advantage of the parallel computing capabilities of a distinct execution environment. In particular, we distinguish algorithms that leverage a system with a *multi-core CPU*, algorithms that target execution on a *single GPU*, algorithms that utilize an *FPGA*, or algorithms that perform computations in parallel in a *distributed system*.

(ii) Parallelized Part

The majority of algorithms execute the *adjacency search* (see Algorithm 1 in Section 2.3.1) in parallel. Additional parts considered for parallel execution are the *Markov blanket discovery*, the calculation of *v-structures*, or *conditioning set filtering*. The *Markov blanket discovery* is an optional step performed prior to the adjacency search to limit the search space [234]. This step is often performed in local-to-global constraint-based CSL approaches [172, 254]. The calculation of *v-structures* is a common step that is performed on the estimated skeleton and executed prior to the application of rules for edge orientation. For detail see [36, 109, 251]. *Conditioning set filtering* is a technique applied prior to the adjacency search to obtain scored conditioning sets for each edge and perform CI tests only with high-scored conditioning sets [78].

(iii) Task Granularity

The granularity of a task for parallel execution generally impacts the degree of parallelism and the demand for communication. In the context of algorithms for parallel constraint-based CSL, coarse-grained tasks, such as a *block of variables*, an individual *edge*, or a *cluster of variables*, require little communication but offer only a limited degree of parallelism. In contrast, fine-grained tasks, e.g., a *subset of data samples*, multiple *CI tests*, or a *sub-score*, expose a high degree of parallelism at increased communication cost. Note that the fine granular task *sub-score* refers to the score calculation during the conditioning set filtering.

(iv) Task Distribution

In the context of constraint-based CSL, tasks for parallel execution are either distributed to the Processing Units (PUs) using a *static* mapping or a *dynamic* distribution strategy. *Static* task distribution maps tasks to the PUs before execution [158]. For example, a naive *static* mapping distributes the same number of tasks to each PU. Such *static* mapping introduces low overhead and is well suited under the assumption of equal-sized tasks. However, in the case of non-uniformly-sized tasks, such a *static* task distribution suffers from load imbalance. *Dynamic* task distribution strategies address these load imbalances at runtime, incurring computational overhead. For example, a simple *dynamic* distribution strategy places tasks on idle PUs at runtime [15].

3.1.2 Multi-Core CPU-Based Parallel Variants of the PC Algorithm

According to the dimensions presented in Table 3.1, all constraint-based CSL algorithms designed for multi-core CPUs consider the parallelization of the adjacency search. Only `bnlearn` [234] addresses the Markov blanket discovery and computation of the v-structures, also. Most of the approaches consider an edge as a task for parallel execution [123, 126, 146, 147, 224, 234]. Only one algorithm [146, 147] considers a more coarse-grained task for parallel execution, i.e., a block of variables. Two approaches [106, 147] explore more fine-grained tasks for parallel execution using CI tests or groups of data samples. The impact of selected task granularities on the runtime of constraint-based CSL is explored in the context of `Fast-BNS` [106]. The tasks for parallel execution are statically distributed to the CPU cores in the majority of algorithms [123, 126, 146, 147, 234].

Only two approaches use dynamic task distribution [106, 224]. To the best of our knowledge, all parallel algorithms are evaluated under the assumption of datasets following either the Gaussian distribution model or consider discrete data, thus applying common CI tests, such as the well-known Pearsons χ^2 test [190] or Fisher's z-test [59]. Next, we introduce and discuss each algorithm.

Vertical Parallel PC [146, 147]

The `vertical parallel PC` [146, 147] is designed for discrete data and applies three steps during the parallel adjacency search. At first, the algorithm performs tests for marginal independence, i.e., with separation set $S = \emptyset$, using Balanced Incomplete Block (BIB) designs [145]. The BIB designs are used to define the tasks for parallel execution as blocks of variables. The blocks of variables get assigned to the processing CPU threads in a static manner. Within each block of variables, the processing CPU thread computes the contingency table once over all variables. Based on this contingency table, the processing CPU thread performs all CI tests, with $S = \emptyset$, for all pairs of variables constructed from its assigned block of variables. Next, the algorithm performs promising CI tests based on a heuristic before all remaining higher-order CI tests are executed. For detail on the heuristic, we refer the reader to the original work [146]. Concerning parallel execution, in these two steps, the task for parallel execution is defined as an edge. The edges are distributed to the processing threads dynamically. Thus, synchronization of the processing threads is required when picking the edge to process to ensure that one edge is only processed by one CPU thread.

Discussion

The optimizations applied by the `vertical parallel PC` algorithm assume the application of a CI test that builds upon computing marginals over the contingency table. In particular, the reuse of the contingency table computed for a block of variables is a promising optimization to save memory access and computation. Yet, computing the contingency table for a block of variables increases its memory footprint, which makes the adoption of this optimization in the context of a GPU-accelerated variant of the PC algorithm impractical. Further, the use of BIB designs for distributing tasks to PUs introduces a computational challenge, as finding a BIB design is NP-complete [38]. The authors of the `vertical parallel PC` algorithm address this computational challenge by pre-calculating a set of BIB designs, which they use at runtime [147]. Nevertheless, this set of pre-calculated BIB designs may not suffice to cover all possible dataset sizes.

Horizontal Parallel PC [147]

The `horizontal parallel PC` algorithm [147] is a variation of the `vertical parallel PC` algorithm [147] and is designed to also run in a distributed system. The `horizontal parallel PC` algorithm performs the same three steps as described for the `vertical parallel PC` algorithm. In contrast to `vertical parallel PC`, the task for parallel execution is defined as a subset of data samples. A master process sends the subsets of data samples to worker processes that compute their share of the contingency table. Next, the master process collects and joins the contingency tables from the worker processes and performs

the CI tests based on the contingency table. Note if the data is incomplete, BIB designs and the optimization based on BIB designs cannot be used. In that case, only a single CI test is performed per contingency table.

Discussion

The authors [147] argue that the `horizontal parallel PC` algorithm is suited for CSL when the number of data samples n is large, e.g., $n \approx 500\,000$. In these settings, it is assumed that the additional time to communicate contingency tables between the worker processes and the master during CI test calculation remains low compared to the overall runtime. In the context of GPU-acceleration, we see the potential to adopt this fine-grained task definition, even in cases with a smaller number of data samples $n \approx 1\,000$. In particular, we see the potential to benefit from coalesced memory access [41, 173] when GPU threads process consecutively stored data samples to compute contingency tables. Further, the GPU's shared memory can keep introduced communication locally and mitigate the impact on the runtime.

Bnlearn [234]

The `bnlearn` library [234], written in the `R` language [198] with `C` extensions, implements a variety of CSL algorithms. The library uses the `parallel` library in `R` to realize parallel implementations of constraint-based CSL algorithms, such as the PC algorithm. The author of the `bnlearn` library [234] proposes to parallelize the Markov blanket discovery, which is applied in local-to-global constraint-based CSL algorithms to reduce the search space in the subsequent adjacency search. Further, he proposes to parallelize the adjacency search and the computation of the v-structures. In the context of the Markov blanket discovery, the task for parallel execution is defined as a variable. Otherwise, for the adjacency search and the v-structure detection, pairs of adjacent variables, i.e., edges, are used as tasks for parallel execution. The `bnlearn` library uses a master process to statically distribute the tasks in all three parallelized parts to a user-defined number of worker processes.

In an experimental evaluation on discrete datasets, the authors find that the number of CI tests performed by each worker varies, as it depends on the topology of the underlying true Direct Acyclic Graph (DAG) of the CGM [234]. Furthermore, they find that communication and synchronization costs impede optimal scaling with the number of workers.

Discussion

The `bnlearn` library proposes a general framework for parallel constraint-based CSL. The parallel execution of the Markov blanket discovery is relevant for local-to-global constraint-based CSL algorithms, e.g., see [172, 254]. Also, we find that in comparison to the adjacency search, the v-structure computation has little impact on the runtime of constraint-based CSL algorithms, which is why we do not consider it for GPU acceleration in our work. In our work, we focus on the PC algorithm, which is a global constraint-based CSL algorithm that builds upon an adjacency search. For parallel execution of the adjacency search, `bnlearn` defines edges as tasks for parallel execution. With a static distribution

of edges to workers, load imbalance is introduced, but the authors of the `bnlearn` library argue that dynamic task distribution could benefit in specific settings, e.g., for gene regulatory datasets [234]. In the context of GPU acceleration, static task distribution is well suited to the device's Single Instruction Multiple Threads (SIMT) execution model. Thus, GPU-accelerated constraint-based CSL algorithms must cope with the mentioned load imbalance. To address the load imbalance in our work, we propose more fine-grained tasks, such as groups of CI tests that are related to the same edge. Further, if such groups of CI tests are mapped to GPU threads within the same warp, local communication via shared memory is ensured, which keeps communication costs low.

Parallel-PC [123, 126]

The `parallel-PC` algorithm [123] is a parallel variant of the PC algorithm, which is implemented in the R-package `ParallelPC` [126]. The `parallel-PC` algorithm utilizes CPU cores to process the CI tests within the adjacency search in parallel. Within each level l of the adjacency search, the `parallel-PC` algorithm statically distributes the CI tests for the remaining edges to the CPU cores. Thereby, the algorithm ensures that CI tests related to the same edge get assigned to the same CPU core. Further, this mapping ensures that once one CI test signals independence for a given edge, the remaining CI tests for that edge can be skipped without communication between processing CPU cores. At the end of each level l the processing cores synchronize and communicate the deleted edges. In addition to the parallelization approach, the `parallel-PC` algorithm provides a memory-efficient option. The memory-efficient option batches processing of the edges to ensure that the available DRAM of the system is not exceeded and does not become a bottleneck. In the case of running the memory-efficient option, the batches are processed sequentially, and the edges within each batch are processed in parallel.

The authors of the `parallel-PC` algorithm [123, 126] evaluate their parallel adjacency search for a series of gene expression data, assuming the Gaussian distribution model. The `parallel-PC` algorithm reduces the runtime for a dataset from the DREAM 5 challenge with 1 643 variables [152] from more than 24 hours running single-threaded to less than 6 hours running on 8 CPU cores. Further, the parallel adjacency search also achieves speedup when applied to other constraint-based CSL methods [126], e.g., `FCI` and `RFCI` [36], causal inference methods such as `IDA` [144] and `Joint-IDA` [168], and `PC-Simple` [19]. Thus, the `parallel-PC` algorithm highlights the broad applicability of a fast adjacency search.

Discussion

The parallel execution of the adjacency search in the `parallel-PC` algorithm works similarly to the one implemented in the `bnlearn` library. Thus, the `parallel-PC` algorithm also suffers from load imbalance, and the same implications, as discussed above, apply to our GPU-accelerated variants of the PC algorithm. In addition, the memory-efficient option implemented by the `parallel-PC` algorithm provides a mechanism that we build upon in our block-based out-of-core GPU-accelerated approach to handle datasets that exceed the GPU's memory capacity.

Load-Balanced Parallel PC [224]

In the context of parallel execution on multi-core CPUs, we propose the `load-balanced parallel PC` [224], which is designed to address the load imbalance of the adjacency search of the `parallel-PC` algorithm and of the `bnlearn` library. `Load-balanced parallel PC` implements a dynamic task scheduling mechanism using a task producer and multiple workers consuming the tasks. The tasks are defined for each level and consist of one edge, respectively performing all CI tests of that edge. In each level l, the task producer fills a central queue with tasks for each of the remaining edges. Next, each worker takes one task and processes the corresponding edge by performing all necessary CI tests. Once a task is finished, the worker pulls the next task from the central queue until the queue is empty and the current level l is processed. Note that if an edge is found as independent, the worker removes the corresponding edge from a global data structure accessible by all workers. Based on the task definition, we argue that dynamic task scheduling is only required for levels $l \geq 1$. In level $l = 0$, the task's size is uniform, as each task comprises exactly one CI test.

In an experimental evaluation using gene expression data that are assumed to follow the Gaussian distribution model, we find that speedups of factors of up to 2.4× can be achieved using `load-balanced parallel PC` over a parallel adjacency search with static task distribution.

Discussion

Our work confirms the assumptions from previous work on parallel execution of the PC algorithm [123, 234] that addressing load imbalance can speed up the execution. However, we only showed this effect for high-dimensional datasets following the Gaussian distribution model. Furthermore, the speedup gained over approaches using static task distribution is only small and highly depends on the structure of the underlying true DAG of the CGM. We assume that in the context of our GPU-accelerated constraint-based CSL algorithms, the communication and synchronization costs of dynamic task distribution exceed the achievable performance benefits. Therefore, we do not consider dynamic task distribution following a task producer and worker model for our proposed GPU-accelerated algorithms.

Fast-BNS [106]

The `Fast-BNS` [106] algorithm focuses on the parallel execution of the adjacency search within the PC algorithm under the assumption of discrete data. For the adjacency search in level $l = 0$, the algorithm considers each edge as a task for parallel execution and distributes the edges equally to the processing CPU threads in a static manner. For level $l \geq 1$, `Fast-BNS` defines the task for parallel execution as a fixed number of CI tests related to one edge. Note that the authors experimentally determine that a suitable number of CI tests is between 6 to 8 [106]. The distribution of tasks is dynamic, using a central queue. Thus, each edge gets placed into the central queue. Next, the processing CPU threads take one edge at a time and perform a fixed number of CI tests. After execution, the edge is added back to the queue in case the edge is not found independent, and further CI tests can be performed. The processing CPU threads continue

performing CI tests for the subsequent edges until all tasks in the current level l have been processed. In addition to the dynamic CI test-level parallel execution, the `Fast-BNS` algorithm proposes three additional optimizations. First, the algorithm considers the undirected edges, i.e., $V_i - V_j$ with $i, j = 1, \ldots, N$ and $i \neq j$, for performing CI tests. Thus, in case an edge $V_i - V_j$ was determined as independent based on the adjacency of V_i, the algorithm does not perform CI tests based on the adjacency of V_j for the same edge. Second, the `Fast-BNS` algorithm assumes that all data is stored in column-major order, e.g., the data samples related to one variable V_i are stored consecutively in memory. Thus, `Fast-BNS` reduces the number of cache misses over variants that assume data is stored in row-major order. Note that the algorithm transposes the input data matrix to achieve column-major order. Lastly, the `Fast-BNS` algorithm computes the separation sets for the fixed number of CI tests performed by one processing CPU thread on the fly to keep the memory footprint low.

In an experimental evaluation, the authors of `Fast-BNS` find that their proposed approach is 4.8 to 24.5 times faster than existing parallel solutions [106]. Further, they find that their CI test-level parallel approach outperforms data-sample-level and edge-level parallel approaches.

Discussion

The first optimization applied by the `Fast-BNS` algorithm, i.e., considering undirected edges $V_i - V_j$, is often implicitly performed but not explicitly mentioned by existing algorithms, e.g., see the PC-stable implementation of the `pcalg` package [111]. In our GPU-accelerated CSL algorithms, we apply the same optimization. Also, in our GPU-accelerated CSL algorithm addressing the discrete data, we assume that observational data is in column-major order [81] (see Section 4.3.1). Lastly, in our GPU-accelerated CSL algorithms, we also apply the third optimization to compute the separation set for each CI test on the fly, as avoiding a large memory footprint is crucial in the context of GPUs, given the limited size of the GPU global memory. The same optimization is also implemented in two other GPU-accelerated algorithms for CSL [287]. Similarly to `Fast-BNS`, we also use fine-grained tasks for parallel execution in our GPU-accelerated CSL algorithms. In the context of the Gaussian distribution model, we apply CI test-level parallelism. Whereas, in the context of the discrete distribution model, we argue that more fine-grained tasks based on data samples are better suited for GPU acceleration. In particular, if GPU threads within the same warp jointly process data samples of a CI test, we can achieve efficient coalesced memory accesses.

3.1.3 GPU-Based Variants of the PC Algorithm

As presented in Table 3.1, existing GPU-based approaches are designed for execution on a single GPU. The approaches found in related literature [287, 290] focus on GPU acceleration of the PC algorithm's adjacency search and consider CI tests as tasks for parallel execution. The task distribution is static, mapping each task to one GPU thread, e.g., using the abstraction via `CUDA` threads. The approaches are evaluated under the assumption of datasets following the Gaussian distribution model and apply Fisher's z-test [59]. While all three GPU-accelerated algorithms from related literature are categorized in the same way

according to Table 3.1, each algorithm implements a different optimization for performing the CI tests. We describe and discuss each algorithm in more detail in the following.

Cupc-E [287]

The `cupc-E` algorithm [287] performs the adjacency search of the PC algorithm on a single GPU. The algorithm processes all levels l sequentially, launching a separate GPU kernel for each level. Note that `cupc-E` provides unique implementations for the GPU kernel in each level, up to level $l = 14$. In the GPU kernel for level $l = 0$, each GPU thread is responsible for performing the CI test for one edge. The mapping of tasks for parallel execution to GPU threads occurs statically, building upon the `CUDA` threading model [170]. In subsequent levels $l \geq 1$, the `cupc-E` algorithm defines the task for parallel execution as multiple CI tests of an edge. The GPU kernels for each level $l \geq 1$ are launched so that the GPU threads process multiple CI tests of multiple edges in parallel, achieving two degrees of parallelism [287]. While processing higher levels $l \geq 1$, the `cupc-E` algorithm applies the following four optimizations.

- (i) The GPU threads within each thread block use shared memory to store frequently accessed elements from the adjacency sets of the processed edges.
- (ii) The `cupc-E` algorithm monitors the edge removals and stops GPU threads that perform CI tests on edges removed by other GPU threads.
- (iii) Each GPU thread computes the separation set required for performing its CI test on the fly to avoid up-front computation of all possible separation sets, which requires additional memory.
- (iv) The `cupc-E` algorithm adds a compacting step before launching any GPU kernel in levels $l \geq 1$.

The compacting step (iv) ensures that the information on adjacent variables is stored consecutively in memory to achieve coalesced memory access and improve the cache performance [287]. Thus, instead of using one-hot encoding in an adjacency matrix to signal that variables are adjacent, the compacted version uses adjacency lists storing indices to the adjacent variables.

In an experimental evaluation using gene expression datasets, `cupc-E` outperforms a serial CPU-based algorithm on average by factors of 525×.

Discussion

The `cupc-E` algorithm is similar to our proposed GPU-accelerated CSL algorithm for data following the Gaussian distribution model (see Section 4.2). In detail, our algorithm also applies two degrees of parallelism and performs multiple CI tests of multiple edges in parallel. Further, we adapted the on-the-fly calculation of the separation sets in higher levels $l \geq 2$ (iii) from `cupc-E`. In contrast, to `cupc-E`, our approach maps CI tests of the same edge to GPU threads within the same thread block and uses the shared memory to communicate the CI test's decisions locally. Thus, we reduce global communication. Furthermore, our approach does not include a compacting step, as it adds computational overhead and does not reduce the required amount of GPU global memory.

Cupc-S [287]

The `cupc-S` algorithm is an extension of the `cupc-E` proposed by the same authors [287]. `cupc-S` accelerates the adjacency search of the PC algorithm using a GPU. The algorithm applies a similar parallel execution strategy as `cupc-E` resulting in two degrees of parallelism and incorporates the same optimization (i-iv). The key difference in `cupc-S` is an optimization to share and reuse one computed pseudo-inverse based on a separation set S in multiple CI tests. In detail, each GPU thread computes a separation set S^i for a given variable V_i, with $i = 1, \ldots, N$, first. Next, the GPU thread computes the pseudo-inverse based on S^i and performs multiple CI tests for V_i and multiple neighbors of V_i using the pseudo-inverse based on S^i. Thus, `cupc-S` still processes multiple CI tests of multiple edges in parallel.

In the experimental evaluation, the applied optimization in `cupc-S` results in an average speedup by a factor of $1\,296\times$ over a serial CPU-based algorithm and more than a factor of $2\times$ speedup over `cupc-E` [287].

Discussion

The proposed optimization in `cupc-S` is tailored to data following the Gaussian distribution model and the application of Fisher's z-test [59]. Yet, we find that sharing and reusing intermediate results is an efficient way to reduce the overall computational demand of the PC algorithm. We adopt the idea of reusing computations based on one separation set S for multiple CI tests in our GPU-accelerated information-theoretic CSL algorithm (see Section 4.4). Furthermore, we find that sharing the pseudo-inverse computation is a promising optimization to be included in our proposed GPU-accelerated CSL algorithm targeting data following the Gaussian distribution model in future work.

Reversed-Order PC [290]

The `reversed-order PC` algorithm parallelizes the adjacency search and uses tensor operations to perform Fisher's z-test [59]. The tensor operations can be executed on a GPU. In contrast to the PC algorithm, the `reversed-order PC` algorithm performs CI tests level-wise in a decreasing order concerning the size of separation sets. Thus, the algorithm starts by performing CI tests with separation sets of size $l = N - 2$, where N is the number of variables and decreases l by one within each level. The authors argue that their reversed-order approach reduces the total number of CI tests under the assumption of dense DAGs of CGMs. Further, the authors base their idea on the observation that a separation set that contains redundant nodes does not sacrifice the quality of the results [290]. Concerning the parallel execution, the `reversed-order PC` applies two degrees of parallelism. On a higher abstraction level, the `reversed-order PC` considers a CI test as a task for parallel execution and batches multiple CI tests together. Assuming that data follows the Gaussian distribution model and the application of Fisher's z-test [59] tensors are used to estimate the partial correlation coefficient. The tensor-based computations are parallelized, too.

In a reference implementation, the authors use the `PyTorch` library [185] to apply the tensor operations based on a batch of CI tests. The authors of the `reversed-order PC` algorithm provide a runtime experiment on a CGM with a

dense DAG that contains 95 nodes, in which the GPU-based `reversed-order PC` algorithm is 825 times faster than the original PC algorithm.

Discussion

While the authors of the `reversed-order PC` algorithm show speedup in the case of a CGM with a dense DAG, the evaluation is limited to CGMs with DAGs that contain up to $N = 100$ variables. Thus, it remains open to examining the performance in high-dimensional settings. Further, note that at the time of writing, the published work on `reversed-order PC` [290] is not peer-reviewed. The idea of the `reversed-order PC` algorithm results in performing a high number of CI tests with large separation sets. This can become a significant challenge for data that follows other distributions, e.g., discrete data, where the memory demand for a CI test increases with the size of the separation set. Thus, transferring the idea of the `reversed-order PC` algorithm becomes impractical in these settings. Furthermore, the optimization using tensors is tailored to Fisher's z-test [59]. Nevertheless, we believe that mapping other CI tests to tensor operations could be beneficial, as tensor operations are highly optimized for performance. In the context of our GPU-accelerated algorithms, we find that the batching of tensors operations could be a relevant improvement to one of our early ideas for processing levels $l \geq 2$ using `CUDA-X` libraries for data following the Gaussian distribution model (see Section 4.2.5).

3.1.4 Further Approaches for Parallel Constraint-Based CSL

Apart from algorithms targeting GPUs and multi-core CPUs, we consider six additional approaches (see Table 3.1). The algorithm `FPGA-CDCSF` [78] employs an FPGA to achieve speedup. `MrPC` [169], `PCA` and `PCB` [44] are extensions of the PC algorithm that implement MapReduce-based approaches to target computation in distributed systems. The algorithms `pPC` [95] and `PEF` [77] build upon divide-and-conquer strategies suited for execution in distributed systems.

While the FPGA-based approach filters conditioning sets in parallel, the other five algorithms parallelize the adjacency search. The FPGA-based approach and `PCA` consider fine granular tasks, i.e., the computation of a sub-score or processing data samples, as the tasks for parallel execution. In contrast, the other algorithms assume a coarse-grained task, e.g., using an edge or a cluster of variables. The algorithms `MrPC` [169], `PCA` and `PCB` [44] distribute their tasks dynamically, whereas the other approaches use a static task distribution. Next, we elaborate on and discuss each algorithm individually.

FPGA-Based Constraint-Based CSL: FPGA-CDCSF [78]

The algorithm `FPGA-CDCSF` [78] shifts the computationally expensive step of performing all CI tests within the adjacency search by introducing a method for Conditioning Set Filtering (CSF). The CSF method precedes the CI test execution with the goal of scoring and ranking all conditioning sets. Only conditioning sets with high scores are considered for CI tests in the subsequent steps. Thus, the algorithm `FPGA-CDCSF` [78] reduces the number of performed CI tests. Yet, the authors of `FPGA-CDCSF` argue that the introduced CSF method is computationally demanding, too. Therefore, they offload the score computation to the

FPGA. FPGA-CDCSF uses an FPGA design with multiple sub-score evaluation modules to compute sub-scores for each conditioning set in parallel. We assume that the fine-grained parallel tasks of computing sub-scores are distributed to the sub-score evaluation modules in a static manner. Apart from the score computation, any other calculation is performed on the CPU, e.g., performing the CI tests. The FPGA-CDCSF algorithm implements a residual-based CI test [288], which they optimize by caching and reusing calculated residuals.

In an experimental evaluation, the authors of FPGA-CDCSF [78] find that their algorithm is superior to CPU-based algorithms [111, 126, 234] and the GPU-based algorithm cupc-E [287] concerning runtime and quality of learned CGMs measured by the Structural Hamming Distance (SHD) [269].

Discussion

The authors of the FPGA-CDCSF algorithm argue that the applied score computation, which solves small least-square problems, is better suited for execution on FPGA than on GPU. Foremost, the authors argue that GPUs lack load balancing methods for fine-grained parallelism and may not be efficient for solving low-dimensional least-square problems. Based on their judgment, we did not consider integrating CSF into our proposed GPU-accelerated variants of the PC algorithm. Furthermore, the FPGA-CDCSF algorithm focuses on applying a residual-based CI test. In contrast, our proposed GPU-accelerated algorithms for constraint-based CSL focus on two CI tests commonly used in practice under the assumption that data follows the Gaussian distribution model (see Section 2.4.1) or data is discrete (see Section 2.4.2). Additionally, we consider GPU acceleration for an information-theoretic CI test for mixed discrete-continuous data and data with non-linear relationships (see Section 2.4.3).

Distributed Constraint-Based CSL: MrPC [169], PCA & PCB [44]

The MrPC algorithm is an adaptation of the PC algorithm to perform the adjacency search in a distributed system. MrPC builds upon the well-known MapReduce framework [45], and within each level l, considers each edge as a task for parallel execution. Within each level, a driver distributes all edges to all executors, i.e., PUs, within the distributed systems, first. Note that each executor requires all edges to compute the correct adjacency sets in the subsequent step. Next, the executors perform all required CI tests for all edges. To avoid redundancy, we assume that each executor communicates the edge it is currently processing. Once an edge is processed, the executor communicates the result back to the driver. Upon receiving a result, the driver updates a global data structure storing the estimated skeleton. After completion of the final level l, the estimated skeleton is returned by the driver.

In an experimental evaluation, the authors of MrPC [169] find that the speedup over a sequential implementation varies between factors of 2.75× to 7.35×. Higher speedup is achieved on larger datasets concerning the number of variables N and the number of data samples n. The authors argue that for smaller datasets, the overhead of the MapReduce framework [45] implemented in Spark [285] limits the speedup.

These results are in line with previous work on MapReduce-based variants of the PC algorithm under the assumption of discrete data, namely PCA and

PCB [44]. PCB, which considers edges as tasks for parallel execution, outperforms a PC algorithm implementation running on a single system only for a larger number of variables with $N \geq 500$ [44]. In the case of PCA, which uses fine granular tasks based on data samples, the overhead of the MapReduce framework is higher. Hence, PCA is slower than a PC algorithm running on a single system.

Discussion

In our work on GPU-accelerated algorithms for constraint-based CSL, we do not consider distributed systems. Thus, in the context of a single heterogeneous system, the proposed approach by MrPC is of limited applicability. Yet, in future work, we could envision extending our GPU-accelerated algorithms to operate in distributed GPU-based systems. In that case, the MapReduce-based approach sets a basis for distributing parallel tasks. To achieve high utilization of the GPUs in each executor of the distributed system, we must redefine the task for parallel execution as a block of edges instead of a single edge. It is subject to future research to determine a suitable size for the block of edges, depending on the performed CI test.

Partition-Based Algorithms: PEF [77] & pPC [95]

The partition-based algorithms for CSL, PEF [77], and pPC [95] apply a divide-and-conquer approach to large-sized input with thousands of variables. The algorithms split the input into $\mathcal{K}$ partitions, apply CSL on these partitions to learn subgraphs, and eventually merge the subgraphs. The partition-based algorithms are not primarily designed for parallel execution, yet, given their nature, Gu & Zhou [77] propose to run the chosen CSL algorithm for each of the $\mathcal{K}$ partitions in parallel on a multi-core CPU system.

Huang & Zhou [95] argue that the benefit of parallel execution for partition-based algorithms is limited by the number of partitions $\mathcal{K}$. They propose a partitioned PC algorithm, called pPC, which allows benefiting from parallel execution of existing parallel variants of PC algorithm. pPC partitions the variables into $\mathcal{K}$ clusters by computing the Mutual Information (MI) and reuses the MI to perform the CI tests in level $l = 0$. Following the adjacency search of the PC algorithm, a skeleton graph is computed for each cluster. In a subsequent cluster screening step, the pPC algorithm merges the $\mathcal{K}$-estimated skeleton graphs into a single estimated skeleton.

Discussion

The partition-based algorithms define tasks for parallel execution as clusters of variables. It is suggested that the number of clusters $\mathcal{K}$ remains small [88], which results in very coarse-grained tasks. These coarse-grained tasks are suited for distributed learning [77] but improper for GPU acceleration. However, suppose the clusters contain sufficient variables. In that case, applying a GPU-accelerated adjacency search to discover the skeleton graph or determine a subgraph could be applied on each cluster to achieve speedup. In the context of multi-GPU-based CSL, the partitioning scheme could be adapted to determine a cluster of variables to be processed on each GPU. Yet, in our proposed multi-GPU-based constraint-based CSL algorithm, we do not apply the partitioning approach, as it does not guarantee the exact same results as the original PC stable algorithm.

3.2 GPU Acceleration Beyond a Single GPU's Memory Capacity

With the rising interest in GPGPU [220], a variety of algorithms have been designed to leverage the GPU's computing power [11, 57, 105]. Using the GPU to accelerate computation introduced new issues, for example, the transfer bottleneck [76] or the restricted amount of GPU global memory [292]. In the basic execution model, GPU-accelerated algorithms require all input data to be stored in GPU global memory before execution and GPU global memory to be reserved for the results [220]. The basic execution model limits the GPU-accelerated algorithms to process data that fits entirely in GPU global memory. To overcome this restriction out-of-core, also referred to as out-of-memory, GPU solutions have been devised.

We discuss several out-of-core solutions in Section 3.2.1. Beyond out-of-core GPU solutions, extending GPU-accelerated algorithms to operate on multiple GPUs also helps to overcome the memory capacity limits of a single GPU. Furthermore, execution on multiple GPUs provides more compute resources to achieve higher speedup compared to a single GPU. We detail multi-GPU solutions in Section 3.2.2.

Note, in the following, we focus on related work that influenced or inspired, is relevant to, or could be used to extend our work on scaling the GPU-accelerated adjacency search of PC-stable beyond a single GPU. Therefore, we neither claim that the following list of considered work is complete nor most up-to-date.

3.2.1 Out-of-Core GPU Computing

Out-of-core GPU-accelerated algorithms are designed to cope with large datasets that exceed the GPU's memory capacity. Generally, we can categorize out-of-core approaches for GPU-accelerated algorithms into two main groups, application-specific approaches, and application-agnostic approaches.

Application-specific approaches utilize application characteristics to split and stream the workload using explicit data transfers. For example see [108, 163, 280]. In contrast, application-agnostic approaches provide frameworks to hide explicit data management, particularly data movement between the host system's DRAM and the GPU global memory, e.g., see [69, 70, 112, 133, 217, 292].

Nowadays, GPU vendors provide technology, such as Unified Memory (UM) [217] or NVLink 2.0 [30], which allow for application-agnostic approaches. For example, UM enables demand paging and page eviction for efficient, transparent data migration. As another example, NVLink 2.0 enables the GPU to directly access DRAM, using the hardware-based address translation services and cache-coherence.

However, these technologies introduce new performance bottlenecks [69, 120, 133, 284]. Therefore, new application-specific [139, 140] optimizations are proposed. We discuss selected approaches from the two groups, including optimizations based on the concepts of UM and NVLink 2.0's coherence, in the following.

Application-Specific Approaches to Out-of-Core GPU Computing

Application-specific approaches to out-of-core GPU computing commonly split data that exceeds the GPU's memory capacity into smaller-sized consumable chunks [280] that are processed minimizing communication between tasks [108].

In the case of a GPU-accelerated out-of-core general matrix multiplication [280], the matrix is split into blocks with a size that saturates the PCI-E bandwidth and allows storing multiple blocks in GPU global memory at the same time. `CUDA` streams [137] are used to overlap data transfer and GPU kernel execution to optimize performance. Furthermore, the blocks of the matrix multiplication are processed in an order that enables to use results of one block as input to the subsequent block to reduce data transfer.

Kabir et al. [108] propose an algorithm for Singular Value Decomposition (SVD) that operates in out-of-memory situations. The authors design the algorithm independent of the particular PU so that it can be applied in the context of GPU-based SVD but also for CPU-based SVD with problem sizes that exceed DRAM. The proposed SVD algorithm minimizes global communication between different memory layers, e.g., between the system's DRAM and the GPU's global memory, by caching tiles of the matrix in each algorithm's step. Furthermore, the proposed SVD algorithm overlaps the remaining communication with computation [108], e.g., using `CUDA` streams [137].

Both approaches above explicitly manage the data allocation on the GPU, the data transfer to and from the GPU, and the orchestration of GPU kernel launches to overlap data transfer and kernel execution [108, 280].

Another application-specific approach, called `EMOGI`, targets graph-traversal algorithms in GPUs [163]. `EMOGI` builds upon the concept of zero-copy memory [218]. Zero-copy memory is resident on the system's DRAM but allows for direct access by GPU threads. Thus, performance depends highly on efficient memory access that reduces the number of data transfers via PCI-E. `EMOGI` employs aligned and coalesced memory access to minimize the number of PCI-E-based data transfers. Therefore, the `EMOGI` approach adapts the graph traversal algorithms so that GPU threads of one warp work on the same edgelist.

Recent application-specific approaches build upon vendor-specific application-agnostic concepts to overcome the GPU's memory capacity limits. In this context, the application-specific approaches leverage application characteristics to optimize performance. In the context of database query processing, Lutz et al. [139] build upon the coherence feature of NVLink 2.0 to implement a no-partitioning hash join that scales to arbitrarily join build sizes. The no-partitioning hash join uses a hybrid hash table that allocates memory on the GPU and spills to the CPU's DRAM closest to the GPU if the GPU's memory capacity is exceeded. Further, the hybrid hash table is stored in a contiguous array in virtual memory that abstracts the physical location of the GPU and CPU memory pages. Thus, the runtime performance of the hybrid hash table gracefully degrades when increasing the hash table's size [139].

The `Triton join` [140] is another out-of-core GPU-based join in the context of database query processing that also builds upon the coherence feature of the fast NVLink 2.0 interconnect. The algorithm implements a novel GPU-partitioned join strategy that caches a working set in GPU global memory to reduce data transfer. The memory pages of the cached working set are mapped into a single contiguous array in virtual memory, which also maps all memory pages spilled to the CPU's DRAM. In the contiguous array in virtual memory, the GPU and CPU pages are interleaved with the goal of keeping the fast interconnect busy at all times.

Implications Concerning our Proposed GPU-Accelerated CSL Algorithms for Scaling Beyond a Single GPU's Memory Capacity

In Section 5.2, we propose a GPU-accelerated adjacency search for PC-stable that scales beyond a single GPU's memory capacity. Our block-based algorithm employs explicit data management and handles all data allocations and transfers, similar to the out-of-core general matrix multiplication [280] or the out-of-memory SVD algorithm [108]. Inspired by these works, our algorithm splits the input dataset into smaller blocks with a size that saturates the interconnect, and our algorithm keeps multiple blocks in GPU global memory simultaneously. Furthermore, we redesign the original PC-stable algorithm to allow for caching of blocks to reduce data transfer. Lastly, we incorporate pipeline parallelism to overlap data transfer and GPU kernel execution, which aligns with ideas from the out-of-memory SVD algorithm [108] or the `Triton join` [140].

In future work, we see the potential for performance improvement of our UM-based algorithm (see Section 5.1) by utilizing fast interconnects such as NVLink 2.0 with cache coherence following the ideas of Lutz et al. [139] and the `Triton join` [140]. In particular, we could also map the GPU and CPU pages of small blocks of data in an interleaved manner into a single contiguous array in virtual memory to achieve high interconnect utilization. Following the performance gains achieved by the `EMOGI` algorithm [163], we should revisit our proposed algorithm to ensure that the GPU threads within the same warp perform coalesced memory access whenever possible.

Application-Agnostic Approaches to Out-of-Core GPU Computing

Application-agnostic approaches provide general frameworks to realize out-of-core GPU computing. The frameworks handle data allocation and data transfer transparently and optimize the data allocation and data transfer strategies for the dominant memory access patterns. Further, modern GPUs provide a dedicated page migration engine integrated into the MMU, which allows the implementation of out-of-core GPU-accelerated algorithms using the concept of UM [217]. However, UM and on-demand page migration introduce overhead. This overhead makes page faults a performance bottleneck for out-of-core scenarios when GPU global memory is oversubscribed [284]. Several solutions are proposed to address this new performance bottleneck [69, 70, 112, 133, 155, 284, 292].

One of the first solutions to address performance overhead by on-demand page migration proposes incorporating a compute unit agnostic page fault handling mechanism that enables the PU to continue computation under a page fault [292]. Additionally, the solution incorporates a simple software-based prefetcher that batches transfers for on-demand pages every $20\mu s$ and fills the batch with speculative prefetched pages [292]. In an experimental evaluation, the proposed solution efficiently addresses the overhead of paging GPU global memory in settings where the data fit into the GPU global memory. However, the overhead can still impact an application's runtime in out-of-core settings.

The `ETC` framework [133] addresses the performance loss in out-of-core settings with three different techniques. First, the framework uses proactive eviction, which creates space for on-demand page migration. Second, `ETC` employs memory-aware throttling to address thrashing costs by reducing parallelism in

case of high numbers of page faults. Third, ETC utilizes linearly compressed pages to increase the effective memory capacity of a GPU. The authors of the ETC framework [133] find that the ideal combination of the three techniques depends on the application. Therefore, ETC uses memory coalescing statistics to classify applications and apply the appropriate techniques.

Ganguly et al. [69] propose a locality-aware pre-eviction policy to address shortcomings of commonly used pre-eviction policies that struggle with performance slowdown due to page faults in out-of-core situations. In particular, they design a tree-based neighborhood eviction strategy that incorporates well with the hardware prefetching mechanism incurring no overhead. Depending on the page size, the locality-aware pre-eviction strategy achieves a speedup of up to 93% compared to commonly applied, least recently used replacement strategies. The authors further observe that pre-eviction and prefetching benefit applications with regular access patterns but can be counter-productive for applications with irregular access patterns [69]. Based on these observations, a programmer-agnostic runtime is proposed that automatically uses the most suitable access technique, switching between remote zero-copy access and on-demand page migration [70]. Remote zero-copy is used for applications with an irregular access pattern, whereas on-demand page migration upon page faults is used for applications with a regular access pattern. To decide on the most suitable memory access technique, consequently, memory allocation strategy, the programmer-agnostic runtime tracks hardware access counters that determine the access pattern and access frequency. Compared to state-of-the-art techniques, the proposed programmer-agnostic runtime achieves performance improvements for applications with irregular access patterns of 22% to 78% [70].

Another approach to address the high costs of GPU page faults for applications with irregular access patterns handles page faults in large-sized batches [112]. While page faults are commonly processed in small-sized batches already, the authors employ a thread oversubscription technique increasing the thread concurrency to achieve larger batch sizes. The thread oversubscription builds upon an extension of the virtual thread [283] to overcome scheduling limits and address costly context switching. Additionally, the proposed approach takes page eviction off the critical path using bidirectional transfers within the DMA engine to overlap page eviction with on-demand page migration. In an evaluation, the authors show that using the two proposed optimizations results in an average speedup of factor 2× over state-of-the-art solutions [112].

Based on a quantitative evaluation, Yu et al. [284] find that page eviction policies are often inappropriate for resolving the performance bottleneck caused by page faults. Based on their observations, the authors find optimization opportunities to improve a least recently used policy. In the case of applications with regular access patterns, proactive eviction and adaptive prefetching are suited to address the performance bottleneck. In contrast, the authors find that memory-aware throttling and capacity compression work well for applications with irregular access patterns. Thus, the observations by Yu et al. [284] are in line with previous results [69] and support the idea of the ETC framework [133] to utilize a combination of techniques suited to the application.

Another approach is proposed with the DRAGON framework [155]. The DRAGON framework extends the GPU addressable memory space to Non-Volatile Memory (NVM) devices to process extremely large datasets, which even exceed the DRAM capacity of the host system. The framework utilizes Linux's page-

caching mechanism and read-ahead operations. In an experimental evaluation, the DRAGON framework outperforms UM-based executions by factors of up to 2.3× in out-of-core settings, where data fits into the host's DRAM.

Implications Concerning our Proposed GPU-Accelerated CSL Algorithms for Scaling Beyond a Single GPU's Memory Capacity

Section 5.1 proposes a GPU-accelerated adjacency search for PC-stable that scales beyond a single GPU's memory capacity by utilizing UM. While the algorithm's implementation has little overhead from a programmer's perspective, it encounters performance issues due to page faults and on-demand page migration in out-of-core situations. In this thesis, we use the UM-based algorithm as a naive baseline that is compared to our block-based algorithm with explicit memory management. Thus, we did not optimize our UM-based algorithm beyond the use of common best practices [219].

However, in future work, we can further optimize our UM-based algorithm using the frameworks described above to address performance issues caused by page faults and on-demand page migration. In particular, our algorithm's memory access pattern highly depends on the CI test characteristics. For example, the Pearson χ^2 CI test [190], which processes data samples sequentially, exhibits regular memory access patterns. In contrast, Fisher's z-test [59] exhibits more irregular memory access, when retrieving entries from the correlation data structure. Therefore, following the observations by Yu et al. [284], we could employ a proactive eviction strategy [69, 133] for CI tests that exhibit regular memory access. In cases of CI tests where irregular accesses dominate, we could use remote zero-copy [70], large-size page fault batches [112], or memory-aware throttling in combination with capacity compression [133]. Based on the experimental evaluations from the related literature [69, 70, 112, 133, 284], we think integrating any of these optimizations results in, at most, two times faster runtimes.

3.2.2 Multi-GPU Computing

Algorithms are designed to run on multiple GPUs for two reasons. First, the algorithms leverage the additional computing power to process a given workload faster. Or second, the algorithms take advantage of the additional memory capacity available on multiple GPUs to overcome the memory capacity limit of a single GPU. Algorithms designed to run on multiple GPUs must deal with similar problems as out-of-core GPU algorithms (see Section 3.2.1). Additionally, algorithms running on multiple GPUs must deal with load balancing and inter-GPU communication. These two issues add to the communication challenge of out-of-core GPU algorithms, i.e., the data transfer between the host system's DRAM and GPU and dealing with resulting page faults and on-demand page migration. Also, the algorithms must operate correctly in a multi-GPU setting.

The Navier-Stokes solver is an early example of a multi-GPU-aware algorithm [263]. The authors propose to divide the data into subdomains distributed to the individual GPUs. Each GPU performs computations on its assigned subdomain, including operations on so-called ghost cells. Ghost cells are used to communicate overlapping boundaries that have been computed in the subdomains of the other GPUs. The algorithm handles all data transfer and GPU kernel execution explicitly. The algorithm assigns one CPU thread per GPU to

have separate `CUDA` contexts and uses a Posix barrier to synchronize between the CPU threads, and thus implicitly synchronizing between the GPUs. Splitting the data into subdomains, which have little to no overlap, and relying on explicit data management, allows for easy scaling of multi-GPU algorithms beyond one system to clusters of GPU servers, too [34].

Due to the increasing adoption of multi-GPU systems and the demand to develop multi-GPU algorithms, frameworks have been designed to ease development [12, 20]. The `MAPS-Multi` multi-GPU framework [12] provides automatic GPU kernel partitioning and memory allocation based upon a classification of the application's input and output memory access patterns. Furthermore, `MAPS-Multi` takes care of GPU kernel scheduling and boundary exchange between GPUs. The `AMGE` framework [20] provides similar capabilities as `MAPS-Multi` but exploits GPU features for peer-to-peer memory access between GPUs. Furthermore, the `AMGE` framework partitions the GPU kernels at thread block boundaries and uses memory access pattern information to minimize the number of remote memory accesses.

With the introduction of UM [217] and its associated on-demand page migration mechanism, GPU vendors provide easy-to-use functionality for implementing multi-GPU algorithms. Similar to out-of-core settings, using UM, with its on-demand page migration mechanism, causes new performance bottlenecks. In particular, an imbalance in the page distribution across GPUs and the inability to move pages between GPUs majorly impact the performance [8]. The `Griffin` hardware-software solution [8] addresses these issues by introducing a dedicated runtime that migrates pages based on locality and uses a delayed first-touch migration policy to achieve even page distribution across multiple GPUs. Further, the advent of modern interconnects, such as NVLink, allow for faster inter-GPU and host-to-GPU communication. Li et al. [132] study the impact of the GPU topology, focusing on different interconnect technologies. The authors find that depending on the topology, the choice of GPUs used in a multi-GPU environment has a measurable impact on communication efficiency and an application's overall performance.

Even though application-agnostic frameworks for eased development in multi-GPU environments, such as UM [217] or `Griffin` [8], exist, optimizations tailored for the specific application yield higher performance. Lutz et al. [139] extend their proposed hash join to operate on multiple GPUs. Therefore, the algorithm distributes the hybrid hash table across the GPUs by interleaving the pages. Thus, they reduce the communication between the host system and the GPUs while leveraging the fast bi-directional inter-GPU interconnect for communication. Similarly, Ran et al. [213] tailor multiple database join operations to multi-GPU systems. The authors leverage unique algorithm characteristics for each join operation to minimize communication when data exchange is needed. For example, the authors propose to exchange blocks between GPUs using a ring exchange plan for their block-based nested-loop join.

Discussion

This thesis proposes two different algorithms for a multi-GPU-accelerated adjacency search of the PC-stable (see Chapter 5). The block-based algorithm builds upon explicit management of memory allocations and data transfer, similar to the multi-GPU Navier-Stokes solver [263]. We also use one CPU thread

per GPU, which handles the communication and GPU kernel orchestration for the assigned GPU. Concerning load balance, we find that the task granularity used by the `AMGE` framework [20] may be too fine-granular. However, we define relatively small-sized tasks using the same granularity used for our out-of-core GPU algorithm that targets a single GPU.

Our proposed UM-based algorithm serves as an easy-to-implement baseline for comparison. Therefore, we only apply well-known best practices [219]. In particular, we advise the page migration engine to move pages to specific GPUs based on locality information, following the idea of `Griffin` [8]. Therefore, we tailor our algorithm for the adjacency search so that within each level l the same subpart of data is processed by the same GPU. Note that we are aware that such a strategy considering data locality can cause load imbalance. Further, in our experimental evaluation, we select different combinations of available GPUs for execution and confirm the observation of Li et al. [132] that the selected combination of GPUs has a measurable impact on an algorithm's runtime. However, tailoring our algorithm to automatically select the most suitable combination of GPUs based on the interconnect topology is left for future work.

3.3 Summary

In this chapter, we discussed existing literature on the parallel execution of constraint-based CSL. We found that most parallel CSL algorithms target multi-core CPUs as execution devices. Therefore the acceleration potential by parallel execution on the CPU is largely exhausted. Existing GPU-based CSL algorithms, which to the best of our knowledge, did not exist at the start of this thesis, are tailored to CI tests for data that follows the Gaussian distribution model. Accordingly, we have a GPU baseline to compare our GPU-accelerated algorithm for data following the Gaussian distribution model and contribute novel GPU-accelerated algorithms for CI tests targeting data with other distribution assumptions or characteristics. Furthermore, we provided an overview of methods for out-of-core GPU computing and execution on multiple GPUs and discussed the implications for our GPU-accelerated algorithms. We identified that the performance of application-agnostic solutions, such as UM, depends upon the dominating memory access pattern.

4

GPU-Accelerated CSL on a Single GPU

This chapter presents our approaches for GPU-accelerated CSL, focusing on the well-known PC algorithm [249], particularly the order-independent variant PC-stable [35] (see Algorithm 1, p. 18). The PC-stable algorithm learns the Completed Partially Directed Acyclic Graph (CPDAG) describing the equivalence class of the Direct Acyclic Graph (DAG) $\mathcal{G}$ of a CGM for a given dataset in two steps (see Section 2.3.1). In the first step, the algorithm performs an adjacency search to discover the skeleton $\mathcal{C}$ of the CPDAG. In the second step, the algorithm performs a rule-based edge orientation on the remaining edges in $\mathcal{C}$. In the adjacency search, i.e., the first step of the algorithm, numerous CI tests are performed. In the worst case, the number of CI tests is exponential to the number of variables in the dataset. Hence, the adjacency search dominates the overall runtime of the algorithm [169]. This first step yields potential for performance improvement through parallel execution, e.g., on multi-core CPUs [123, 224, 234]. We argue that further acceleration is possible through parallel execution on GPUs. Thus, this thesis' subject is developing and implementing an efficient parallel adjacency search leveraging the parallel processing power of GPUs to improve the runtime of constraint-based CSL (see (RQ1)).

Therefore, in Section 4.1, we deduce parallel execution strategies for a GPU-accelerated adjacency search within the PC-stable algorithm with varying task granularity. The deduction follows Foster's methodology for parallel algorithm design [61], and the developed parallel execution strategies reflect GPU-specific hardware characteristics. Based upon the derived parallel execution strategies, we introduce three algorithms for a GPU-accelerated adjacency search within the PC-stable. Each algorithm is designed for CI tests following specific data characteristics. Section 4.2 describes the first algorithm that is designed for data following the Gaussian distribution model. Section 4.3 explains the second algorithm that targets discrete data. The third algorithm presented in Section 4.4 focuses on mixed discrete-continuous data and data with non-linear relationships. *Parts of this chapter have been published in three research papers [81, 83, 226].*

4.1 Execution Strategies for a GPU-Accelerated Adjacency Search in PC-Stable

An algorithm that targets the GPU needs to harness the massively parallel computing capabilities of the device [220]. Therefore, an adequate parallel execution

strategy is required. Generally, parallel execution strategies are derived using frameworks for designing parallel programs, such as Foster's Design Methodology [61], a well-known guide for developing parallel programs. Foster's Design Methodology is a four-step process that consists of the steps of *Partitioning*, *Communication*, *Agglomeration*, and *Mapping* described as follows: (1) The *Partitioning* step divides the problem into small-sized tasks for parallel processing. (2) The *Communication* step determines the communication pattern of the small-sized tasks and detects whether these tasks communicate locally or globally. (3) The *Agglomeration* step groups small tasks into larger tasks to improve performance, considering communication costs and hardware characteristics. (4) The *Mapping* step assigns the agglomerated tasks to physical processing units. This step is essential in load balancing and task scheduling in shared-nothing systems. However, in a uniprocessor or shared memory system, the operating system's task scheduling is often sufficient [61].

The partitioning step and the communication step consider algorithmic aspects only. Thus, the PC-stable algorithm and aspects of the data distribution models and corresponding CI tests (see Section 2.4) are relevant in these steps. GPU-specific hardware characteristics and resulting constraints (see Section 2.5) are considered in the last two steps. In the following sections, we describe the step-by-step application of Foster's Design Methodology to derive suitable parallel execution strategies within PC-stable. For reference within the steps, we first define tasks within PC-stable at different levels of granularity.

4.1.1 Definition of Tasks Within PC-Stable

Within the PC-stable, results from each level l strongly depend on each other. Pairs of variables (V_i, V_j), with $i, j = 1, \ldots, N$ and $i \neq j$, determined as independent in one level are removed from the adjacency $adj(\mathcal{C}, V_i)$. Thus, they are irrelevant in subsequent levels. Therefore the PC-stable processes the levels l with $l = 0, \ldots, \max_{i=1,\ldots,N}\{|adj(\mathcal{G}, V_i)|\} - 1$ sequentially and defines the tasks in the context of a level. Note that the same applies to the original PC algorithm [249]. The PC-stable is a candidate for functional decomposition. In contrast to domain decomposition, where a given problem is divided based on the processed data, functional decomposition separates the given problem based on the computation performed [61]. According to a functional decomposition, four possible tasks within PC-stable are identified and defined as follows:

(I) A task is defined as the adjacency of a variable $adj(\mathcal{C}^l, V_i)$ given the skeleton $\mathcal{C}^l$ within the current level l.
(II) A task is defined as the individual pair of variables (V_i, V_j), synonymously denoted by the edge $E^{i,j}$, with $i < j$.
(III) A task is defined as the CI test for the edge $E^{i,j}$ given a distinct separation set $S_o^{i,j}$ with $o = 1, \ldots, \binom{|adj(\mathcal{C}^l, V_i)|-1}{l}$.
(IV) A task is defined as the individual data sample for a distinct CI test, i.e., $D_m(V_i, V_j, S^{i,j})$ indexed with $m = 1, \ldots, n$, where n is the total number of data samples available in the given data D.

To get an intuition of these four tasks, we illustrate each task definition in one of the Figures 4.1 – 4.4 (see pp. 53–54). Each figure shows the same set of computations represented by rectangles. Note that a single rectangle refers to the

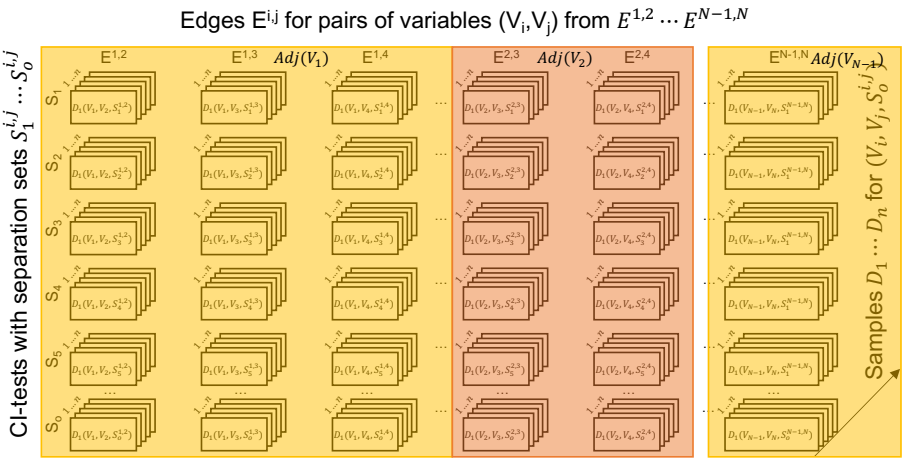


Fig. 4.1: Illustration of task (I) defined as adjacencies $adj(\mathcal{C}^l, V_i)$. Each colored set of columns refers to the task of processing one adjacency.

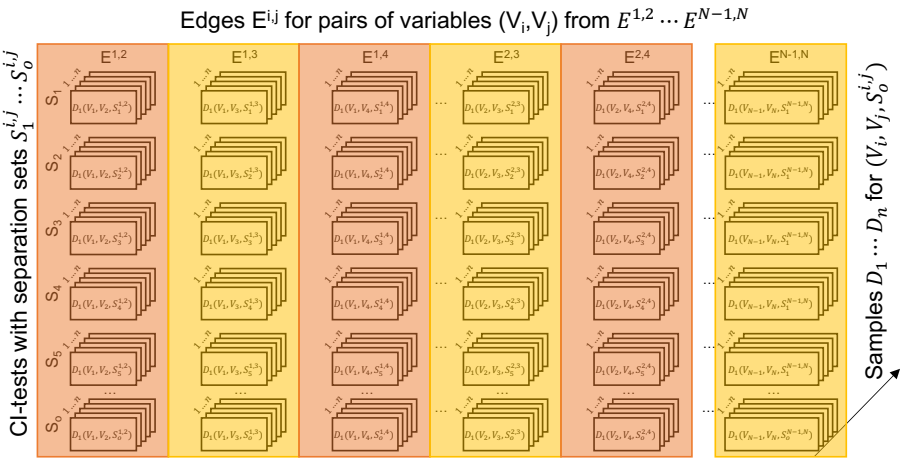


Fig. 4.2: Illustration of task (II) defined as edges $E^{i,j}$. Each colored column of stacked rectangles refers to the task of processing one edge.

processing of one data sample. In each figure, colors show how the computations are grouped to form the defined tasks (I) – (IV).

A task defined as the adjacency of a variable $adj(\mathcal{C}^l, V_i)$ (I), shown in Figure 4.1, presents the most coarse-grained task. The task consists of multiple edges $E^{i,j}$, related CI tests, and all required data samples. In contrast, a task defined as the edge $E^{i,j}$ (II), shown in Figure 4.2, is more fine-grained. The task contains only those CI tests and respective data samples, related to the edge $E^{i,j}$. Figure 4.3 (see p. 54) illustrates tasks defined as one CI test of an edge (III). In this case, each task contains only the required data samples for the particular CI test. A task that is defined as a data sample of a distinct CI test (I), as shown in Figure 4.4 (see p. 54), is the most fine-grained task.

Comparing the figures, it becomes visible that depending on the task definition, the task size ts and the total number of tasks tn vary. Both parameters are relevant for the design of an effective parallel execution strategy. For the

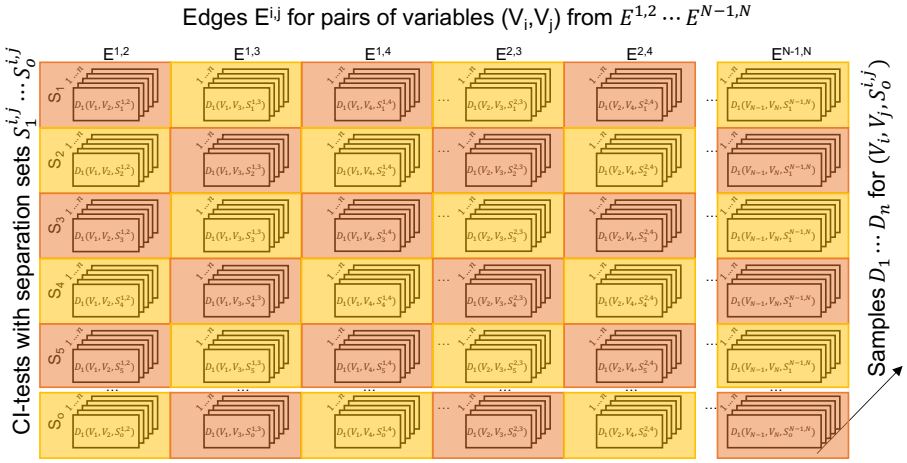


Fig. 4.3: Illustration of task (III) defined as CI tests. Each colored stack of rectangles corresponds to the task of processing one CI test.

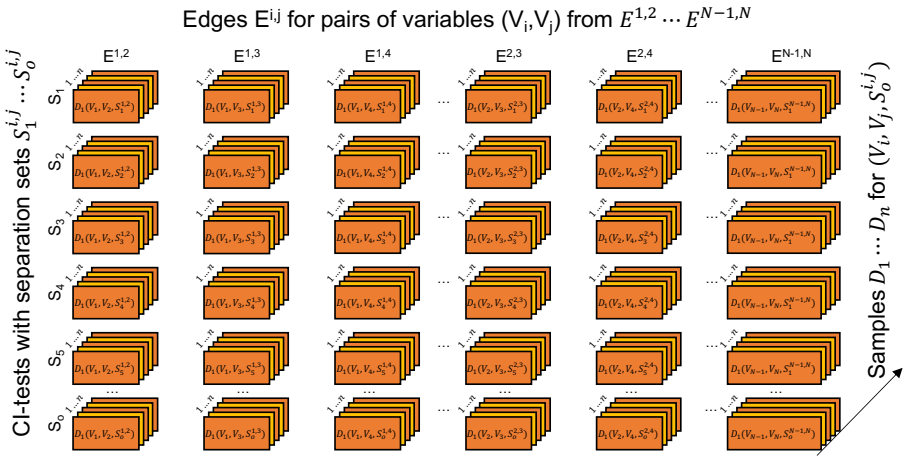


Fig. 4.4: Illustration of task (IV) defined as data samples D_m. Each colored rectangle refers to the task of processing a single data sample.

PC-stable algorithm, we know that the current level l and the assumption on the underlying true DAG of the CGM impact both parameters for any of the task definitions. In particular, we have three different cases: A *fully connected CGM for levels* $l \geq 1$, a *sparse CGM for levels* $l \geq 1$ and *any CGM in level* $l = 0$. In the following, we consider each case separately and determine the two parameters ts and tn for each of the four task definitions.

A) Fully Connected CGM for Levels $l \geq 1$

First, we present the case of a fully connected CGM in any level $l \geq 1$, as shown in Figure 4.5 (see p. 55). The assumption of a fully connected CGM presents the computational worst-case, as all CI tests for all edges must be performed. In this case, the task size ts and the total number of tasks tn are computed for each task definition in the following ways:

(I) If the task is defined as an adjacency $adj(\mathcal{C}^l, V_i)$, then:

$$\begin{aligned} ts &= (N-1) \times \binom{N-2}{l} \times n \\ tn &= N. \end{aligned} \tag{4.1}$$

(II) If the task is defined as an edge $E^{i,j}$, then:

$$\begin{aligned} ts &= \binom{N-2}{l} \times n \\ tn &= N \times (N-1). \end{aligned} \tag{4.2}$$

(III) If the task is defined as a CI test, then:

$$\begin{aligned} ts &= n \\ tn &= N \times (N-1) \times \binom{N-2}{l}. \end{aligned} \tag{4.3}$$

(IV) If the task is defined as an individual data sample D_m, then:

$$\begin{aligned} ts &= 1 \\ tn &= (N-1) \times \binom{N-2}{l} \times n. \end{aligned} \tag{4.4}$$

Considering the four different task definitions, we find an increasing amount of tasks for parallel execution from task definition (I) to task definition (IV). At the same time, the size of the tasks decreases from task definition (I) to task definition (IV). We consider task definition (IV), i.e., processing one data sample D_m, as an atomic task. Thus, we set the size $ts = 1$.

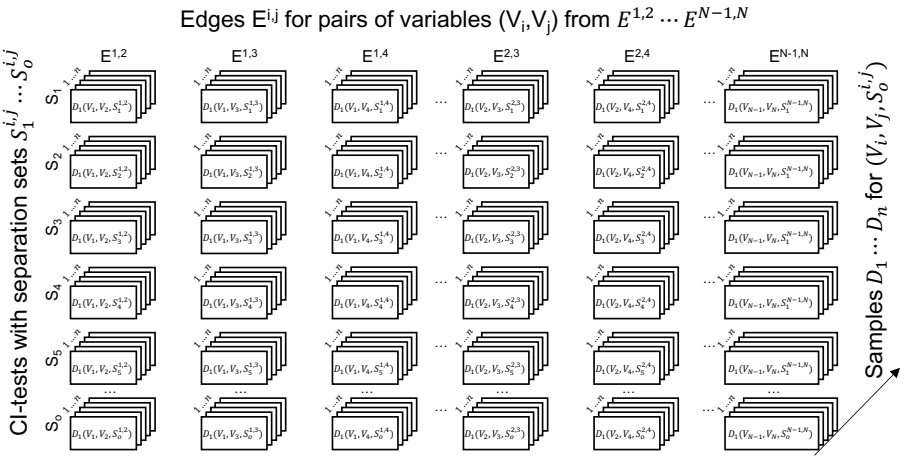


Fig. 4.5: Illustration of tasks within the adjacency search of the PC-stable algorithm for parallel execution assuming a fully connected CGM for any level $l \geq 1$. Depicted are undirected edges $E^{i,j}$ (x-dimension), CI tests (y-dimension), and data samples D (z-dimension).

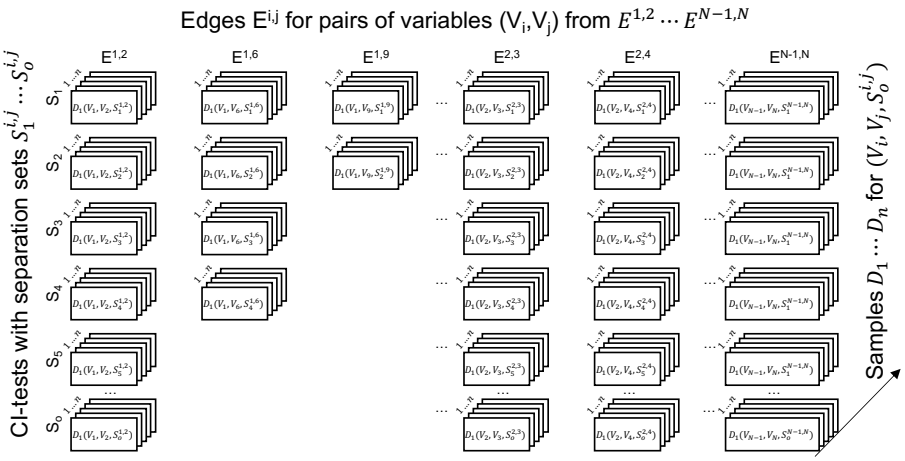


Fig. 4.6: Illustration of tasks within the adjacency search of the PC-stable algorithm for parallel execution assuming a sparse connected CGM for any level $l \geq 1$. Depicted are undirected edges $E^{i,j}$ (x-dimension), CI tests (y-dimension), and data samples D (z-dimension).

B) Sparse CGM for Levels $l \geq 1$

Second, we consider the case of a sparse CGM in any level $l \geq 1$, as shown in Figure 4.6. Here, the values for the task size ts and the total number of tasks tn are relative to the structure of the skeleton $\mathcal{C}^l$ within the current level l and the underlying true DAG of the CGM. The structure of the skeleton $\mathcal{C}^l$ determines the number of remaining edges. The underlying true DAG of the CGM influences the number of conducted CI tests per edge. Once one CI test determines the pair of variables for a given edge as independent, all remaining CI tests for that edge can be skipped. Hence, depending on the task definition, either exact values are computed or upper bounds are estimated for the task size ts and the total number of tasks tn as follows:

(I) If the task is defined as an adjacency $adj(\mathcal{C}^l, V_i)$, then:

$$\begin{aligned} ts &\leq (N-1) \times |\mathbf{S}^{i,j,l}| \times n \\ tn &= N, \end{aligned} \tag{4.5}$$

where $|\mathbf{S}^{i,j,l}|$ denotes the list of all possible separation sets for a pair of variables (V_i, V_j) in level l and is calculated as:

$$|\mathbf{S}^{i,j,l}| = \binom{|adj(\mathcal{C}^l, V_i)| - 1}{l}. \tag{4.6}$$

(II) If the task is defined as an edge $E^{i,j}$, then:

$$\begin{aligned} ts &\leq |\mathbf{S}^{i,j,l}| \times n \\ tn &= N \times (N-1). \end{aligned} \tag{4.7}$$

(III) If the task is defined as a CI test, then:

$$\begin{aligned} ts &= n \\ tn &\leq N \times (N-1) \times |\mathbf{S}^{i,j,l}|. \end{aligned} \tag{4.8}$$

(IV) If the task is defined as an individual data sample D_m, then

$$\begin{aligned} ts &= 1 \\ tn &= N \times (N-1) \times |\mathbf{S}^{i,j,l}| \times n. \end{aligned} \tag{4.9}$$

Similar to a fully connected CGM (see A)), the number of tasks tn increases from task definition (I) to (IV), while the size of the tasks ts decreases. However, in contrast to a fully connected CGM, we cannot always compute an exact value for tn and ts. In some cases, it is only possible to estimate an upper bound.

C) Any CGM in $l = 0$

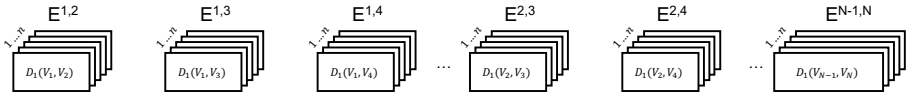


Fig. 4.7: Illustration of tasks within the adjacency search of the PC-stable algorithm for parallel execution for any CGM in level $l = 0$. Depicted are undirected edges $E^{i,j}$ (x-dimension) and data samples D (z-dimension). We do not illustrate the CI tests explicitly, as the task based on a CI test is equivalent to the task based on an edge.

Third, we explain the special case for level $l = 0$ that applies to any CGM, as shown in Figure 4.7. Level $l = 0$ represents a special case, as only one CI test per edge $E^{i,j}$ needs to be performed. Thus, the task definitions based on an edge $E^{i,j}$ (II) and based on one CI test (III) are equivalent. Further, under the assumption that no background knowledge is provided, the skeleton $\mathcal{C}^0$ is fully connected. Therefore, the task size ts and the total number of tasks tn for each task definition are computed as follows:

(I) If the task is defined as an adjacency $adj(\mathcal{C}^l, V_i)$, then:

$$\begin{aligned} ts &= (N-1) \times n \\ tn &= N. \end{aligned} \tag{4.10}$$

(II & III) If the task is defined as an edge $E^{i,j}$ or as a CI test, then:

$$\begin{aligned} ts &= n \\ tn &= N \times (N-1). \end{aligned} \tag{4.11}$$

(IV) If the task is defined as an individual data sample D_m, then

$$\begin{aligned} ts &= 1 \\ tn &= N \times (N-1) \times n. \end{aligned} \tag{4.12}$$

Again, the number of tasks increases from task definition (I) to task definition (IV), while the size of the tasks decreases. Similar to a fully connected CGM exact values of tn and ts can be computed. Thus, determining a parallel execution strategy for this case should be straightforward.

4.1.2 Application of Foster's Methodology: Step (1) Partitioning

The functional decomposition (see Section 4.1.1) results in four definitions of tasks (I) – (IV). According to Foster's Design Methodology [61], the partitioning step favors the smallest possible task. For the PC-stable, regardless of the underlying true DAG of the CGM and the current level l, this task is defined as the individual data sample for a distinct CI test, i.e., $D_m(V_i, V_j, S^{i,j})$.

Based on a data sample $D_m(V_i, V_j, S^{i,j})$, the task's computations contribute to processing one CI test. For its computations, the task accesses the associated memory locations of the data sample for V_i, V_j, and $S^{i,j}$. Further, the task's actual computations concerning D_m depend on the chosen CI test. Thus, in the case of the Gaussian distribution model (see Section 2.4.1), each task computes part of the correlation coefficient. In the case of discrete data (see Section 2.4.2), each task increments an entry of the contingency table, marginals, and total. For mixed discrete-continuous data and data with non-linear relationships (see Section 2.4.3), a task computes a partial value of the Conditional Mutual Information (CMI) estimate.

The total number of tasks tn following task definition (IV) is computed using (4.4), (4.9), or (4.12). The choice of computation depends on the structure of the underlying true DAG of the CGM and the current level l. Thus, the total number of tasks tn ranges from $N \times (N-1) \times n$ for level $l = 0$ to $N \times (N-1) \times \binom{N-2}{l} \times n$ for a fully connected CGM in any level $l \geq 1$. Therefore, this task definition exposes a massive amount of parallelism in any setting.

4.1.3 Application of Foster's Methodology: Step (2) Communication

According to the partition design, which identifies D_m as the smallest task, three communications are identified within the PC-stable. The first communication occurs while performing a CI test. The second communication is needed to communicate CI test results' while processing an edge $E^{i,j}$. And the third identified communication happens during the update of the skeleton $\mathcal{C}$. Note that the computations based on D_m are independent of any other task and require no additional communication.

Performing a CI Test

Generally, the p-value is computed and returned when a CI test is performed. To calculate the p-value, all tasks related to a single CI test must share their local result computed based on D_m. This communication involves n tasks and can be realized in different ways. For example, all tasks can broadcast their local result leading to $n \times n$ communications. Further, at the cost of synchronization, the tasks' local result can be stored in a central memory location. Yet another approach is to employ a parallel summation algorithm, which allows for concurrent communication. Regardless of the approach, this communication step yields one p-value for each CI test.

Processing an Edge $E^{i,j}$

The results of all CI tests related to an edge $E^{i,j}$ need to be communicated to decide on the independence of $E^{i,j}$. Therefore, this communication involves up to $\binom{N-2}{l} \times n$ tasks. Ideally, the results of each CI test have been aggregated over

the associated tasks upfront. Such an aggregation step could reduce the number of tasks involved with the communication to $\binom{N-2}{l}$. Note that the number of communications depends on the number of separation sets $|\mathbf{S}^{i,j,l}|$ for $E^{i,j}$ in l. Again different ways exist to realize communication. Given that the adjacency search of the PC-stable provides an early termination criterion (see line 16 of Algorithm 1 in Section 2.3.1), we suggest a centralized communication. An early termination occurs if $E^{i,j}$ is determined as independent based on one CI test's p-value. In that case, all remaining communications and computations for that edge become obsolete.

Updating the Estimated Skeleton $\mathcal{C}$

Another step in PC-stable's adjacency search updates the skeleton $\mathcal{C}$. Therefore the independence decision for each edge $E^{i,j}$ is communicated globally. The independence decision of each edge $E^{i,j}$ is an aggregated result across all CI tests related to that edge $E^{i,j}$. Hence, updating the skeleton involves communication between up to $N \times (N-1) \times \binom{N-2}{l} \times n$ tasks. Aggregation of the result per edge $E^{i,j}$ can limit the number of tasks required for communication to $N \times (N-1)$. Further, the independence decision of an edge is stored at distinct locations within the skeleton $\mathcal{C}$. Therefore, communications can occur concurrently.

4.1.4 Application of Foster's Methodology: Step (3) Agglomeration

The partitioning determined the individual data sample for a distinct CI test (IV), i.e., $D_m(V_i, V_j, S^{i,j})$ as the smallest possible task that provides ample parallelism. Building upon that task definition, the communication step identified three relevant communications within a single level l of the adjacency search in PC-stable. The agglomeration step aims to agglomerate tasks to design an efficient algorithm for particular hardware, in the present case, a GPU. Therefore, the definition of the agglomerated task should aim to reduce communication while retaining enough opportunity for parallel execution. At the same time, the agglomeration must take into account GPU-specific hardware characteristics, e.g., the size of shared memory, the grouping of GPU threads into thread blocks, and their execution in warps. In the following, we, therefore, discuss how the agglomeration of tasks addresses the identified communications.

Addressing Communication While Performing a CI Test

The communication required while performing one CI test cannot occur entirely concurrently. Hence, we suggest agglomerating the n tasks that perform one CI test to reduce the needed communication. Therefore, the computations over r data samples, i.e., r tasks, are grouped to form an agglomerated task. Note that $1 < r \leq n$ holds. Thus, the number of tasks that need to communicate while performing one CI test is reduced by a factor of $\frac{1}{r}$. The number of necessary communications is reduced respectively. Unless we set $r = n$, the choice of r should reflect the GPU hardware characteristics, as discussed below.

If $r = n$, the communication is removed. The agglomerated task is a unique CI test, as depicted in Figure 4.3 (see p. 54). According to (4.8), this task definition results in up to $N \times (N-1) \times |\mathbf{S}^{i,j,l}|$ unique tasks, which provides ample parallelism. If $r < n$, the number of agglomerated tasks ta related to one CI test is impacted by the chosen value of r as follows:

$$ta = \left\lceil \frac{n}{r} \right\rceil. \tag{4.13}$$

To reflect the GPU hardware characteristics, a value of ta should satisfy:

$$ws \leq |ta| \leq T_{TB_{max}} \quad \text{with} \quad ta \bmod ws \equiv 0, \tag{4.14}$$

where ws is the warp size of the GPU, and $T_{TB_{max}}$ refers to the GPU's maximum number of threads per thread block. Thus, the choice of r should lead to a number of agglomerated tasks ta that is a multiple of the warp size. Having a multiple of the warp size ensures that all GPU threads within a warp operate on the same CI test's computations. Furthermore, this improves data locality and enables coalesced memory access [41, 173]. Additionally, the choice of r should result in a number of agglomerated tasks ta that does not exceed $T_{TB_{max}}$. Not exceeding $T_{TB_{max}}$ ensures that GPU threads can process all agglomerated tasks related to the same CI test within the same thread block. Thereby the GPU threads can use shared memory for communication.

Addressing Communication During Edge Processing

Generally, concurrent communication of the CI test results for one edge $E^{i,j}$ is possible. Yet, once the edge $E^{i,j}$ was found independent based on one CI test, performing the remaining CI tests for the edge $E^{i,j}$ and consequently communicating their results becomes unnecessary overhead. To address this overhead, we suggest agglomerating q tasks belonging to the computation of CI tests of an edge $E^{i,j}$. An appropriate value of q is chosen as follows:

$$n \leq q \leq |\mathbf{S}^{i,j,l}| \times n \quad \text{with} \quad q \bmod n \equiv 0. \tag{4.15}$$

Note, q is chosen as a multiple of n to assure that all tasks belonging to one CI test are agglomerated to the same task. In the following, we discuss the implications of the choice of the value of q to the agglomerated task.

In the case of $q = |\mathbf{S}^{i,j,l}| \times n$, the agglomeration combines all tasks required to process the entire edge $E^{i,j}$, as shown in Figure 4.2 (see p. 53). Thus, all communication of CI test results is removed, enabling early termination and removing unnecessary computations. At the same time, combining all tasks concerning the CI tests for an edge $E^{i,j}$ can result in highly imbalanced tasks. The imbalance results from the current skeleton $\mathcal{C}^l$ and the influence of the underlying true DAG of the CGM on the number of CI tests per edge (see Figure 4.6, p. 56). Furthermore, this agglomeration results in $N \times (N-1)$ aggregated tasks, which provides enough parallelism on multi-core CPUs [123, 224, 234]. However, on a GPU, a small number of tasks can limit exploitation of the parallel hardware, e.g., if $N \leq 100$.

Therefore, for execution on the GPU, a smaller value of q, which exposes more parallel tasks, is better suited. In this case, the chosen value of q must compensate for the communication overhead of smaller aggregated tasks with their improved load balance over large aggregated tasks.

Further Considerations

The communication during the update of the skeleton $\mathcal{C}^l$ can occur concurrently. Therefore, an agglomeration addressing this communication is not necessary.

Furthermore, a more coarse-grained task definition, according to the adjacency $adj(\mathcal{C}^l, V_i)$ (I) (see Figure 4.1, p. 53), exposes only up to N aggregated tasks. Yet, for efficient execution on a GPU, thousands to millions of tasks are generally encouraged [220]. Thus, considering an agglomeration up to processing an entire adjacency $adj(\mathcal{C}^l, V_i)$ seems unsuited.

Note that the computational demand of processing a single data sample D_m varies depending on the applied CI test. Therefore, the agglomeration step does not yield a single most-suitable task definition for parallel execution of the PC-stable on a GPU across all existing CI tests. Rather, the previously derived task definitions form a set of possible abstractions that can be applied depending on the characteristics of the CI test. In the remainder of this thesis, the following two task definitions are particularly relevant: A task is defined as r data samples D_m within a CI test or a task is defined as q CI tests of an edge $E^{i,j}$.

4.1.5 Application of Foster's Methodology: Step (4) Mapping

The mapping step assigns the tasks resulting from the agglomeration to the physical processing unit. Note this step is often skipped in a uniprocessor or shared memory system. In the case of a GPU, the actual assignment of tasks to Streaming Multiprocessors (SMs) is done via a dedicated hardware thread block scheduler. Yet, it is worth considering mapping tasks to the abstractions within the `CUDA` framework, i.e., to GPU threads and thread blocks, while considering warps and the Single Instruction Multiple Threads (SIMT) execution model. Such a mapping ensures that associated tasks are mapped to GPU threads within the same thread block enabling local communication via shared memory. Further, tasks associated with co-located data in GPU global memory can be mapped to GPU threads within the same warp to ensure coalesced memory access. Based on the two agglomerated tasks from the previous step, we define three mappings of tasks to the GPU's processing units (M1-M3). The first mapping (M1) is applied to tasks consisting of r data samples from a CI test. The second mapping (M2) considers tasks concerning multiple CI tests of an edge $E^{i,j}$. The third mapping (M3) is a combination of the mappings (M1) and (M2). The following paragraphs detail each mapping (M1-M3).

(M1) Tasks Consisting of r Data Samples From a CI Test

In the case of a task that processes r data samples from one CI test, the mapping as shown in Figure 4.8 (see p. 62) is applied. A task is mapped to one GPU thread. Each task gets assigned r data samples in a stride of the warp size, and GPU threads within a warp process consecutive tasks related to the same CI test. Thereby, the mapping ensures coalesced memory access within a warp. Further, all tasks associated with one CI test are mapped to the same thread block. Hence, these tasks can utilize shared memory for fast local communication, e.g., when computing the p-value. Suppose the number of tasks associated with one CI test is larger than $T_{TB_{max}}$. In that case, each GPU thread in the same thread block gets assigned multiple tasks to avoid global communication, e.g., via GPU global memory. Assigning multiple tasks to one GPU thread has a similar effect as increasing the value of r. The number of required thread blocks TB within a grid for this mapping equals the number of CI tests, i.e., $TB = N \times (N-1) \times |\mathbf{S}^{i,j,l}|$.

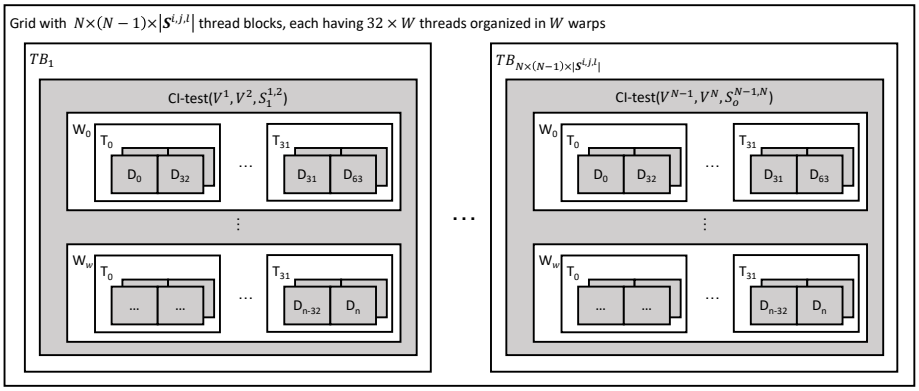


Fig. 4.8: Mapping of tasks that contain r data samples of D (small grey boxes) related to one CI test (large grey box) concerning the abstraction of execution units within `CUDA` (white boxes). TB denotes thread blocks. In each thread block W represents the warps and T the GPU threads.

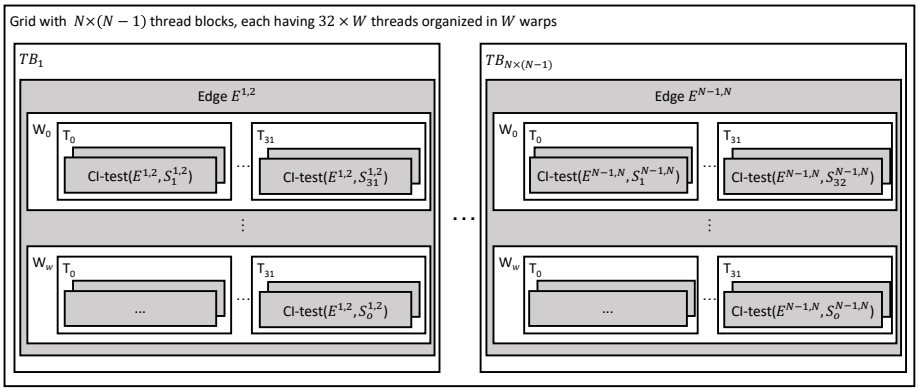


Fig. 4.9: Mapping of tasks that contain multiple CI tests (small grey boxes) related to one edge $E^{i,j}$ (large grey box) concerning the abstractions of execution units within `CUDA` (white boxes). TB denotes thread blocks. In each thread block W represents the warps and T the GPU threads.

(M2) Tasks Consisting of Multiple CI Tests of an Edge $E^{i,j}$

If tasks process multiple CI tests of a given edge $E^{i,j}$, the mapping shown in Figure 4.9 is applied. Each task is mapped to one GPU thread, which becomes responsible for processing all data samples of the associated CI tests. All tasks concerning the same edge $E^{i,j}$ are mapped to the same thread block. Thus, the CI tests' independence decisions can be communicated locally via shared memory, which also allows realizing the early termination criterion.

Suppose the number of tasks associated with one edge $E^{i,j}$ is larger than $T_{TB_{max}}$. In that case, each GPU thread in the same thread block gets assigned multiple tasks to keep the communication of CI test results locally. Again, increasing the value of q, i.e., assigning more CI test to the same task, has a similar effect. The number of required thread blocks TB within a grid for this mapping equals the number of edges, i.e., $TB = N \times (N-1)$.

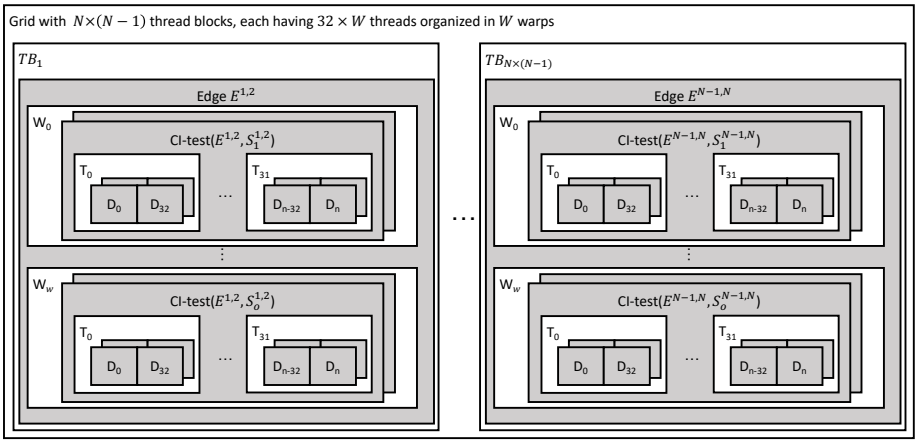


Fig. 4.10: Fused tasks that contain r data samples (smallest grey boxes) from multiple CI tests (mid-sized grey boxes) related to one edge $E^{i,j}$ (large grey box) mapped to the abstractions of execution units within `CUDA` (white boxes). TB denotes thread blocks. In each thread block W represents the warps and T the GPU threads.

(M3) Fused Mapping of r Data Samples of Multiple CI Tests of One Edge $E^{i,j}$

The mapping of r data samples from a CI test (see (M1)) leverages data locality to enable coalesced memory access. Yet, it does not address the global communication of CI test results and the early termination criterion. The mapping of multiple CI tests of an edge $E^{i,j}$ (see (M2)) uses local communication to exchange CI test results but lacks any improved data access mechanism. Therefore, we consider a combination of the two mappings (M1) and (M2). Fusing the two mappings (M1, M2) creates the potential to benefit from coalesced memory access and local communication when exchanging results from CI tests of an edge $E^{i,j}$. Figure 4.10 presents the fused mapping.

The tasks processing r data samples of multiple CI tests concerning the same edge $E^{i,j}$ are mapped to GPU threads within the same thread block. In case the number of tasks exceeds the number of GPU threads in that thread block, multiple tasks are grouped to each of the GPU threads. According to mapping (M1), the r data samples are assigned to each task in a warp-sized stride to enable coalesced memory access. Further, by assigning all tasks concerning the same edge $E^{i,j}$ to the same thread block, CI test results can be communicated locally within shared memory, as mapping (M2) suggested. Therefore, the number of required thread blocks TB within a grid is the same as for mapping (M2), i.e., $TB = N \times (N-1)$.

The derived tasks, identified communication patterns, agglomerated tasks, and deduced mappings are the building blocks for the efficient parallel execution strategies of the PC-stable. In the subsequent sections, we apply these building blocks to develop efficient GPU-accelerated algorithms for multiple CI tests with different characteristics.

4.2 GPU-Accelerated Adjacency Search in PC-Stable for the Gaussian Distribution Model

This section details our algorithm for a GPU-accelerated adjacency search in PC-stable under the assumption that data follows the Gaussian distribution model (see Section 2.4.1). First, a runtime analysis of the original CPU-based PC-stable algorithm on several real-world gene expression datasets reveals a large potential for performance improvements in levels $l = 0, 1$ (see Section 4.2.1). Section 4.2.2 outlines our proposed GPU-accelerated algorithm that addresses the potential for performance improvement. The outline of the algorithm mainly focuses on data transfer and GPU kernel orchestration. The GPU kernel implementations for level $l = 0$ and level $l = 1$, which constitute the performance improvement, are presented in Sections 4.2.3 and 4.2.4. Finally, Section 4.2.5 introduces two variants to process levels $l \geq 2$. One variant leverages functionality from `CUDA-X`[1] libraries, whereas the second variant follows the ideas of Zarebavani et al. [287] but differs in the use of GPU shared memory. *Parts of this section have been published in a research paper [226].*

4.2.1 Determining Compute Intensive Parts of the Adjacency Search of PC-Stable - A Runtime Analysis on Selected Gene Expression Datasets

In this section, we analyze the runtime of the PC-stable algorithm to determine the most compute-intensive parts that can benefit from GPU acceleration. First, we consider the computational demand from a theoretical point of view. Next, we introduce a selection of gene expression datasets we use for experimental runtime analysis. Finally, we assess the results from the experimental runtime analysis to reveal the potential for performance improvement, which we can address with our GPU-accelerated adjacency search in PC-stable considering the Gaussian distribution model.

Theoretical Consideration Concerning the Computational Demand

The adjacency search of the PC-stable algorithm [35] learns the skeleton $\mathcal{C}$ of the CPDAG of the CGM in an iterative level-wise process, starting with level $l = 0$ (see Section 2.3). Within each level l, all remaining edges, $E^{i,j}$ with $i, j = 1, \ldots, N$, are tested for conditional independence. Therefore, CI tests are performed with separation sets of size l, which are constructed from the adjacent variables of V_i. Thus, the number of CI tests per level increases up to a certain level l, which depends on the underlying true DAG of the CGM. Therefore, the worst-case computational complexity of the PC-stable algorithm is exponential to the number of variables N [251].

Similarly, the computational complexity of an individual CI test appropriate for data following the Gaussian distribution model, i.e., Fisher's z-test [59] (see Section 2.4.1), increases with a higher level l. In the case of Fisher's z-test, the computation of the pseudo-inverse constitutes the majority of the computational demand. Matrix inversion using the well-known Gauss-Jordan algorithm

[1] `https://developer.nvidia.com/gpu-accelerated-libraries`

has a computational complexity concerning the dimension of a matrix *dim* as follows:

$$\mathcal{O}\left(dim^3\right) \text{ [240]}, \tag{4.16}$$

which in the case of Fisher's z-test leads to a complexity of

$$\mathcal{O}\left(\left|\{V_i, V_j\} \cup S^{i,j}\right|^3\right). \tag{4.17}$$

Following these theoretical considerations, the most compute-intensive parts of the adjacency search are in very high levels l, which require computing a large number of CI tests with a high computational complexity each.

In contrast, it is well known that the statistical power of CI tests suffers from the curse of dimensionality, i.e., with larger-sized separation sets [142]. In theory, assuming $n \to \infty$, the statistical power of CI tests is not an issue. Yet, in real-world settings, such as genetics, low data sample sizes are common and restrict the statistical power of the CI test with larger-sized separation sets. Furthermore, under the assumption of sparse CGMs, often the case in real-world settings [150], the overall complexity of the PC-stable algorithm is polynomial to the number of variables [109]. In fact, the number of CI tests in any level l is bounded by the size of the adjacency sets of each variable. Similarly, the maximum reachable level ml is bounded by the size of the adjacency sets. Once no separation set of size l can be constructed from any remaining adjacency set, the maximum level is reached. Thus, very high levels l are often not considered in real-world settings, and the number of CI tests might be reasonably small.

Experiments Concerning the Computational Demand

In the following, we conduct runtime measurements of the PC algorithm's adjacency search using real-world gene expression datasets. The measurements aim to understand the implications of the prior mentioned characteristics on the number of conducted CI tests and the runtime within each level l. These results are then used to determine computationally intensive levels l best suited for parallel execution on the GPU.

Selected Gene Expression Datasets

We consider a selection of real-world gene expression datasets [125, 143, 152], which have been used in literature to evaluate parallel variants of the PC algorithm [123, 226, 287]. Table 4.1 summarizes the characteristics of the selected

dataset	N - number of variables	n - number of data samples
`NCI-60` [125]	1 190	47
`MCC` [125]	1 380	88
`BR51` [125]	1 592	50
`S.aureus` [152]	2 810	160
`S.cerevisiae` [143]	5 361	63

Table 4.1: Characteristics of selected real-world gene expression datasets used in research on the inference of regulatory relationships. The first three datasets contain microRNA and mRNA information, whereas the last two datasets contain transcription factors.

gene expression datasets. The datasets have a high dimensionality with N in the range from 1 190 to 5 361. However, the number of observed data samples n is low and ranges from 47 to 160.

Runtime Measurements on Selected Gene Expression Datasets

For the runtime analysis, we execute the single-threaded implementation of the PC-stable, available in the `pcalg` package [111]. We follow existing literature and set the significance level $\alpha = 0.01$ [109]. As the parameter α impacts the CI decision, changes of the parameter will result in differences in the number of conducted CI tests, respectively runtimes. Note that larger values of α tend to result in a higher density in the learned skeleton $\mathcal{C}$. The results of the measurements are shown in Tables 4.2 and 4.3.

dataset	total number of CI tests	percentage of CI tests in level $l = 0$	level $l = 1$	levels $l \geq 2$
`NCI-60`	4 017 475	17.61%	82.16%	0.23%
`MCC`	21 950 296	4.34%	93.00%	2.66%
`BR51`	29 696 242	4.26%	95.61%	0.13%
`S.aureus`	170 430 911	2.32%	95.48%	2.20%
`S.cerevisiae`	69 321 855	20.73%	78.76%	0.51%

Table 4.2: Number of performed CI tests in the adjacency search of PC-stable, and their distribution to individual levels, for selected gene expression datasets, setting the tuning parameter $\alpha = 0.01$.

Table 4.2 depicts the percentages of CI tests conducted within each level compared to the total number of CI tests. The results indicate that most CI tests are performed in level $l = 1$ across all datasets. The second-largest number of CI tests is conducted in level $l = 0$. At most, 2.66% of all CI tests are carried out in higher levels $l \geq 2$, e.g., in the case of the `MCC` dataset. For all datasets, the adjacency search terminated after level $l = 4$, which we account for an insufficient statistical power in any higher level given the low data sample sizes of the gene expression datasets. For the dataset with the largest number of variables, `S.cerevisiae`, 14 367 480 CI tests are performed in level $l = 0$, 54 598 391 CI tests are computed in level $l = 1$, and the remaining levels $l \geq 2$ account for 355 984 CI tests.

Given the increasing computational complexity of CI tests in higher levels, only considering the number of conducted CI tests is insufficient to conclude on the levels l that are particularly interesting for acceleration. Therefore, we present the distribution of the total runtimes concerning individual levels in Table 4.3 (see p. 67). The impact of the CI tests in level $l = 0$ on the overall runtime of the adjacency search is negligible for all datasets. Looking at the `S.cerevisiae` dataset, level $l = 0$ accounts for only 0.09% of the runtime, although 20.73% of all CI tests are performed in this level. Level $l = 1$ dominates the overall runtime for all investigated gene expression datasets. The levels $l \geq 2$ have a minor influence on the total runtime. However, it is noteworthy that the impact on the total runtime in percent is greater than the percentage share in performed CI tests.

dataset	total runtime	percentage of runtime in		
		level $l = 0$	level $l = 1$	levels $l \geq 2$
NCI-60	8.06 s	0.27%	98.96%	0.77%
MCC	59.39 s	0.05%	94.57%	5.38%
BR51	67.57 s	0.05%	99.57%	0.38%
S.aureus	1 037.23 s	0.01%	98.03%	1.96%
S.cerevisiae	479.59 s	0.09%	99.28%	0.63%

Table 4.3: Runtimes of the adjacency search of PC-stable in seconds, and its distribution to individual levels, for selected gene expression datasets, setting the tuning parameter $\alpha = 0.01$

Discussion

From a theoretical point of view, we find that the higher levels l expose more CI tests, thus more parallel tasks. In contrast, the runtime measurements reveal that in real-world settings, the adjacency search in levels $l = 0, 1$ can benefit the most from parallel execution on a GPU. However, it remains open to investigate other settings, for example, when datasets contain more data samples n or the CGMs are dense. In these cases, we think that higher levels $l \geq 2$ also benefit from GPU acceleration. Thus, we do not generally disregard these levels for GPU acceleration.

4.2.2 Outline of the GPU-Accelerated Adjacency Search

Based on the observations from the runtime assessment on real-world gene expression datasets, we develop GPU kernels to accelerate computations in levels $l = 0, 1$. But, we also present GPU-accelerated variants for higher levels $l \geq 2$. We develop an algorithm for a parallel adjacency search that leverages these GPU kernels and GPU-accelerated variants for runtime performance. We tailor the algorithm and the GPU kernels to the Gaussian distribution model. Algorithm 3 (see p. 68) outlines our proposed algorithm and shows the necessary operations, data structures, and interactions between the host system and the GPU. Details on the implemented GPU kernels and the GPU-accelerated variants are discussed separately in the subsequent Sections 4.2.3, 4.2.4, and 4.2.5.

Input

Algorithm 3 (see p. 68) receives as input the vertex set $\mathbf{V} = \{V_1, \ldots, V_N\}$ representing the dataset's N variables (see Section 2.1) and the correlation matrix $sCor$ of size $N \times N$. Further, the algorithm takes the parameters α as the significance level, ml as the maximum level, and n as the number of data samples as input. The correlation matrix $sCor$ contains the sample correlation $\hat{\rho}(V_i, V_j)$ (see (2.1), p. 20) for all pairs of variables (V_i, V_j) with $i, j = 1, \ldots, N$ and $i \neq j$. The significance level α serves as a threshold for the CI tests. The maximum level ml restricts the levels l considered during the adjacency search.

Suppose there is no restriction based on ml. In this case, the adjacency search could reach a level $l = N - 2$ under the assumption of a fully connected CGM. Implicitly the value of ml also sets the maximum size of any separation

Algorithm 3 GPU-based adjacency search for the Gaussian distribution model
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$, significance level α, maximum level ml, number of data samples n
Output: Estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep

```
1: l ← 0
2: Let C and Ĉ be |V| × |V| matrices with all entries set to 1
3: Let Sep be an |V| × |V| × ml matrix with all entries set to −1
4: transferToGPU(V, sCor, C, Ĉ, Sep)
5: launchGPUKernel(level0, {V, sCor, C, Sep, α, n})
6: update(ml)
7: while ml > l do
8:    l ← l + 1
9:    copyTo(Ĉ, C)
10:   if l = 1 then
11:      launchGPUKernel(level1, {V, sCor, C, Ĉ, Sep, α, n})
12:   else
13:      launchGPUVariant(levelL, {V, sCor, C, Ĉ, Sep, α, n, l})
14:   end if
15:   update(ml)
16: end while
17: transferFromGPU(C, Sep)
18: return C, Sep
```

set, which is needed to pre-allocate GPU global memory for the data structure holding all separation sets Sep. To avoid out-of-GPU-memory situations, we recommend a value of ml that is calculated as follows:

$$ml = \left\lfloor \frac{GPU_{mem} - 3 \times N \times N \times \text{SIZEOF}(dt)}{N \times N \times \text{SIZEOF}(dt)} \right\rfloor, \tag{4.18}$$

where GPU_{mem} is the capacity of the GPU's global memory and SIZEOF(dt) returns the bytes required for the data type dt.

Output

After completion, the adjacency search returns two matrices. The first matrix of dimension $N \times N$ stores the estimated skeleton $\mathcal{C}$. The second matrix Sep of dimension $N \times N \times ml$ contains all separation sets $S^{i,j}$ corresponding to the pairs of variables (V_i, V_j) estimated to be independent.

Description of the Procedure

The algorithm starts with the initialization of required data structures and parameters. Thus, it sets the level to $l = 0$. Further, it initializes three data matrices. The first two matrices of dimension $N \times N$ store the estimated skeleton $\mathcal{C}$ and the copy of the skeleton $\hat{\mathcal{C}}$, which is necessary to ensure order independence within the PC-stable. All entries within these two matrices are set to 1. The third matrix Sep of dimension $N \times N \times ml$ stores the separation sets $S^{i,j}$ for independent pairs of variables (V_i, V_j). All entries of this matrix are set to -1.

Next, the algorithm transfers all data structures required within the GPU kernels to the GPU global memory (see line 4). Note that the data structures remain in GPU global memory until the operations in all levels have finished.

After these preparation steps, the actual calculations are performed. Therefore, the GPU kernel for level $l = 0$ is launched. The GPU kernel launch occurs synchronously, meaning that the CPU thread blocks until the GPU kernel is processed. The synchronous kernel launch ensures that level $l = 0$ has been finished before the next level is processed. Likewise, all subsequent kernel launches are also synchronous, if not otherwise stated.

The GPU kernel for level $l = 0$ is provided the vertex set $\mathbf{V}$, the correlation matrix $sCor$, the matrices for the estimated skeleton $\mathcal{C}$ and the separation sets Sep, and the significance level α. For a detailed description of the operations within the GPU kernel for level $l = 0$ and the applied mapping of tasks to the GPU execution units (see Section 4.2.3, p. 70).

Once the GPU kernel for level $l = 0$ is finished, the UPDATE(...) function checks if the maximum level ml needs adjustment. Therefore, the size of the most extensive adjacency set of any variable V_i in the current skeleton $\mathcal{C}^l$ is computed, which we denote with $\max_{i=1,\dots,N}\{|adj(\mathcal{C}^l, V_i)|\}$. The maximum level ml is updated as follows:

$$ml = \begin{cases} \max_{i=1,\dots,N}\{|adj(\mathcal{C}^l, V_i)|\} - 1, & \text{if } \max_{i=1,\dots,N}\{|adj(\mathcal{C}^l, V_i)|\} - 1 < ml \\ ml, & \text{otherwise.} \end{cases} \tag{4.19}$$

This UPDATE(...) function is implemented as a separate GPU kernel, and only the parameter ml is transferred.

After updating ml, subsequent levels are processed in a loop (see lines 7 – 16 of Algorithm 3), as long as the condition $ml > l$ holds. Within this loop, the algorithm first increments the level l by one. Next, the COPYTO(...) function is called, which copies the content from the data structure of the current version of $\mathcal{C}$ to the placeholder data structure $\hat{\mathcal{C}}$ to ensure order independence. Suppose $l = 1$, the GPU kernel for level $l = 1$ is launched with the following parameters; a vertex set $\mathbf{V}$, the correlation matrix $sCor$, the data structures for the estimated skeleton $\mathcal{C}$, the copy of the estimated skeleton $\hat{\mathcal{C}}$, separation sets Sep, and the significance level α. The performed operations and the applied mapping of the GPU kernel for level $l = 1$ are described in Section 4.2.4. Otherwise, if level $l \geq 2$, the GPU variants for any higher level are executed, as described in Section 4.2.5. Depending on the variant, either a separate procedure with multiple CPU and GPU functions is called, or a GPU kernel is directly launched. Once all levels l up to ml are processed, the estimated skeleton $\mathcal{C}$ and the corresponding separation sets Sep are copied from the GPU to the host system's DRAM (see line 17 of Algorithm 3). Afterward, the algorithm returns these two data structures.

Additional Notes

For simplification, we left out calculating, maintaining, and returning additional data structures, such as the list of maximum p-values $pMax$. Further note that at the time of writing this thesis, a further optimized version of a GPU-based adjacency search for data following the Gaussian distribution model has been published [287], as discussed in Section 3.1.3. While both versions follow the same general outline, the optimized version [287] adds an adjacency compacting step after each level, e.g., added between lines 15–16.

Algorithm 4 GPU Kernel for level $l = 0$ for the Gaussian distribution model
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$, estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep, significance level α, number of data samples n
# of blocks: $N \times \lceil \frac{N}{\delta} \rceil$
# of threads per block: δ

```
1: row ← bx, col ← by × tx, N ← |V|
2: if row < col & col ≤ N then
3:   if C[row][col] = 1 then
4:     z ← √(n − 3) × 0.5×LOG1P(2×sCor[row][col] / 1−sCor[row][col])
5:     p ← 2 × (1−NORMCDF(z))
6:     if p ≥ α then
7:       C[row][col] ← 0
8:       C[col][row] ← 0
9:       Sep[row][col][0] ← 0
10:    end if
11:  end if
12: end if
```

4.2.3 GPU Kernel for Level 0

In the following, we describe the GPU kernel for level $l = 0$ for data that follows the Gaussian distribution model. First, we present the chosen parallel execution strategy, according to Section 4.1. Further, we describe the input, the output, the GPU kernel launch parameters and detail the operations executed within the GPU kernel.

Execution Strategy: Mapping of Tasks to Execution Units

In level $l = 0$, the separation set for each pair of variables (V_i, V_j) is the empty set, i.e., $S^{i,j} = \emptyset$ with $i, j = 1, \ldots, N$ and $i \neq j$. Thus, only a single CI test is conducted per edge $E^{i,j}$, and the problem becomes embarrassingly parallel [234]. Note that we stick to the term CI test for simplicity, even though the term direct independence test is more precise in the case of an empty separation set. Further, in the Gaussian distribution model, each CI test processes the sample correlation values instead of entire observations. Therefore, the mapping (M2) is applied, and each task processes the sample correlation values for one CI test. Since each task effectively corresponds to processing a single edge $E^{i,j}$, no communication of CI test decisions is required. Thus, only the concurrent communication to update the skeleton $\mathcal{C}$ remains. According to the mapping (M2), each task is mapped to an individual GPU thread with the following constraint. GPU threads within the same warp process edges that are stored consecutively in each data structure. Thus, coalesced memory access of the GPU threads in the same warp is achieved [41, 173]. Following these considerations, we implement the GPU kernel for level $l = 0$ as outlined in Algorithm 4.

Input and Output

The GPU kernel receives the vertex set $\mathbf{V}$, the correlation matrix $sCor$, the estimated skeleton matrix $\mathcal{C}$ with dimension $N \times N$, the separation sets matrix Sep of dimension $N \times N \times ml$, the significance level α, and the number of data

samples n as input. The GPU kernel has no explicit output but manipulates the data structures $\mathcal{C}$ and Sep during execution. Note that no copy of the skeleton $\mathcal{C}$ is required in level $l = 0$, as the level is, per definition, order-independent.

GPU Kernel Launch Parameters

The GPU kernel is launched with $N \times \lceil \frac{N}{\delta} \rceil$ thread blocks and δ threads per block. An appropriate number of thread blocks is chosen as follows:

$$ws \leq \delta \leq T_{TB_{max}} \quad \text{and} \quad \delta \bmod ws \equiv 0, \tag{4.20}$$

where ws denotes the warp size of the GPU, and $T_{TB_{max}}$ represents the maximum number of threads per thread block supported by the GPU. Choosing δ according to (4.20) ensures that the number of threads per block follows the hardware constraints. The maximum number of supported threads per block is not exceeded, and enough GPU threads are used to utilize entire warps. The problem of processing the CI tests in level $l = 0$ is embarrassingly parallel and further, GPU threads within the same thread block do not use shared memory. Therefore, there is little difference in selecting small or large values of δ.

Description of the Operations Within the GPU Kernel

After the GPU kernel launch, each GPU thread computes the indices *row* and *col* based on its internal thread and block ids (see line 1 of Algorithm 4, p. 70). These indices refer to positions within the skeleton matrix $\mathcal{C}$. The algorithm only considers elements in the upper triangular of the skeleton matrix, as the CI test is direction-independent. This simple optimization reduces the number of performed CI tests by half. Therefore, in line 2, GPU threads accessing the lower triangular are disregarded. At the same time, an out-of-bounds check is performed. The check for the existence of an edge (see line 3 of Algorithm 4) is only needed if an extension of the PC algorithm that allows for prior knowledge in the form of exclusion lists is applied [159]. Suppose all checks are passed. In this case, the p-value is calculated according to the mathematical model introduced in Section 2.4.1 (see lines 4–5 of Algorithm 4). In the GPU kernel implementation, we use the functions LOG1P(...) and NORMCDF(...) provided by the `CUDA` Math API [177].

If the p-value is greater than or equal to the significance level α, the corresponding edge $E^{row,col}$ is deleted in $\mathcal{C}$, i.e., the fields $\mathcal{C}[row][col]$ and $\mathcal{C}[col][row]$ in the skeleton matrix are set to 0. Note, we set the fields in both upper and lower triangular for technical reasons and use in subsequent levels. In contrast, the entry in the separation sets matrix Sep is only marked in the upper triangular matrix (see line 9). This step concludes the operations of one GPU thread. Once all GPU threads have performed their computation, the GPU kernel terminates.

4.2.4 GPU Kernel for Level 1

In the following, we describe the GPU kernel for level $l = 1$ in the case that data follows the Gaussian distribution model. We first describe the chosen parallel execution strategy, according to Section 4.1. Afterward, we describe the input, the output, the GPU kernel launch parameters and elucidate the operations executed within the GPU kernel.

Execution Strategy: Mapping of Tasks to Execution Units

In level $l = 1$, the parallel execution strategy must handle a varying number of CI tests per edge $E^{i,j}$ with $i, j = 1, \ldots, N$ and $i \neq j$ and should allow for early termination when processing multiple CI tests per edge $E^{i,j}$. In level $l = 1$, each separation set for any pair of variables (V_i, V_j) has a size of 1. Thus, the set of separation set candidates $\mathbf{S}^{i,j,1}$ contains the adjacency sets $adj(\mathcal{C}^l, V_i)$ and $adj(\mathcal{C}^l, V_j)$ based on the current version of the skeleton $\mathcal{C}^l$. As a result, the number of CI tests for any edge $E^{i,j}$ varies between 0 and $N - 2$. Depending on the structure of the underlying true DAG of the CGM, it is unpredictable if CI tests based on all separation sets of an edge $E^{i,j}$ need to be performed before the edge is found independent.

Based on these requirements, we apply mapping (M2), and each task processes the sample correlations related to multiple CI tests. The resulting number of tasks per edge $E^{i,j}$, denoted by u, should reflect GPU hardware characteristics, such as the warp size ws or the maximum number of threads per thread block $T_{TB_{max}}$. Therefore, we specify u as follows:

$$ws \leq u \leq T_{TB_{max}} \quad \text{and} \quad u \bmod ws \equiv 0. \tag{4.21}$$

Setting u according to (4.21) ensures that all tasks related to the same edge $E^{i,j}$ can be mapped to GPU threads within the same thread block. Thus, the CI test results are communicated locally via shared memory. The shared memory is also used to realize the early termination. Further, the number of tasks per $E^{i,j}$ ensures that GPU threads working on the edge $E^{i,j}$ can be grouped in entire warps. Note that choosing a large value for u, e.g., $u = T_{TB_{max}}$, results in using higher amounts of shared memory, potentially limiting the number of thread blocks executed on the same SM. Also, more CI tests are performed before the early termination criterion is checked, potentially performing unnecessary CI tests. To avoid unnecessary CI tests, we suggest choosing smaller values of u, e.g., $u = 32, 64$ or 128. Building upon these considerations, we outline the GPU kernel for level $l = 1$ in Algorithm 5 (see p. 73).

Input and Output

The GPU kernel takes the vertex set $\mathbf{V}$, the correlation matrix $sCor$, the estimated skeleton matrix $\mathcal{C}$, and the copy of the estimated skeleton matrix $\hat{\mathcal{C}}$ with dimension $N \times N$ each, and the separation sets matrix Sep of dimension $N \times N \times ml$, the significance level α, and the number of data samples n as input. Like the kernel for level $l = 0$, the kernel for level $l = 1$ has no explicit output but manipulates the data structures $\mathcal{C}$ and Sep during execution. Note that using the data structure $\hat{\mathcal{C}}$ ensures order independence according to PC-stable [35].

GPU Kernel Launch Parameters

The GPU kernel is launched with $N \times N$ thread blocks and δ threads per block. We set $\delta = u$ following (4.21). Thereby, we ensure that all GPU threads working on tasks stemming from the same edge $E^{i,j}$ belong to the same thread block.

Algorithm 5 GPU kernel for level $l = 1$ for the Gaussian distribution model
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$, estimated skeleton matrix $\mathcal{C}$, copy of estimated skeleton matrix $\hat{\mathcal{C}}$, separation sets matrix Sep, significance level α, number of data samples n
# of blocks: $N \times N$
# of threads per block: δ

```
1: row ← bx, col ← by, N ← |V|
2: if row < col & C[row][col] ≠ 0 then
3:   Initialize list pVals of length δ in shared memory
4:   for a ← tx; a < N; a ← a + δ do          ▷ Iterate separation sets of each task
5:     pVals[tx] ← −1
6:     if row ≠ a & col ≠ a & (Ĉ[row][a] ≠ 0 or Ĉ[col][a] ≠ 0) then
7:       r ← (sCor[row][col] − sCor[row][a] × sCor[col][a]) / sqrt((1.0 − sCor[col][a]^2) × (1.0 − sCor[row][a]^2))
8:       z ← sqrt(n − 1 − 3) × 0.5 × LOG1P(2 × r / (1 − r))
9:       pvals[tx] ← 2 × (1 − NORMCDF(z))
10:    end if
11:    SYNCTHREADS()
12:    if tx = 0 then
13:      for b ← 0; b < δ; b ← b + 1 do
14:        if pVals[b] ≥ α then
15:          C[row][col] ← 0
16:          C[col][row] ← 0
17:          Sep[row][col][0] ← a + b
18:          break
19:        end if
20:      end for
21:    end if
22:    SYNCTHREADS()
23:    if C[row][col] = 0 then
24:      break                                  ▷ Early return
25:    end if
26:  end for
27: end if
```

Description of the Operations Within the GPU Kernel

At first, each GPU thread sets indices *row* and *col* according to its block ids (see line 1 of Algorithm 5). These indices identify the edge $E^{row,col}$ that a GPU thread processes. Under the same assumption as kernel $l = 0$, the algorithm considers only the upper triangular of the estimated skeleton matrix. The fulfillment of this criterion is checked in line 2. At the same time, the GPU thread checks if its corresponding edge $E^{row,col}$ exists. Suppose both checks evaluate true. Then the GPU thread starts processing its assigned task. Therefore, as a next step, the list $pVals$ of size δ is initialized in shared memory. This list is used for local communication of the computed p-values between GPU threads within the same thread block, i.e., GPU threads processing the same edge $E^{row,col}$.

All tasks belonging to one edge $E^{row,col}$ are processed in parallel by δ GPU threads. The GPU threads iterate the separation sets, i.e., CI tests, belonging to their task (see lines 4–27). Within each iteration, each GPU thread considers one separation set indexed by a (see line 4). Next, each GPU thread sets the

entry in $pVals$ at the position of its thread index to -1 and checks the validity of the current separation set at index a. A separation set is valid if the edge indices row and col differ from the separation set index a, i.e., $V_{row} \neq V_a, V_{col} \neq V_a$, and if there exists an entry in the copy of the estimated skeleton matrix for either one of the edges $E^{row,a}$ or $E^{col,a}$. Suppose that the separation set is valid. Then, the p-value is computed based on Fisher's z-transform of the inverted three × three correlation matrix, constructed from the sample correlation entries $sCor[row][col], sCor[row][a], sCor[col][a]$. The computed p-value is stored in the list $pVals$ at the position of the GPU thread's index (see lines 8–10). Note that the calculation follows the mathematical model introduced in Section 2.4.1. In our CUDA-based implementation, the GPU kernel for level $l = 1$ uses the functions LOG1P(...) and NORMCDF(...) provided by the `CUDA` Math API [177].

After synchronizing the GPU threads within the same thread block, one primary GPU thread, i.e., $tx = 0$, compares each p-value in $pVals$ against the significance level α. If one p-value is greater than or equal to α, the corresponding edge $E^{row,col}$ is deleted in $\mathcal{C}$, i.e., the fields $\mathcal{C}[row][col]$ and $\mathcal{C}[col][row]$ in the estimated skeleton matrix are set to 0. Again, we set the fields in both upper and lower triangular for technical reasons and use in subsequent levels. Further, the index of the separation set that determined edge $E^{row,col}$ as independent is stored in the upper triangular of the separation sets matrix Sep (see line 19).

Once the primary GPU thread finished iterating the list $pVals$, all GPU threads within the same thread block are synchronized (see line 24). Further, if the primary GPU thread marked the edge $E^{row,col}$ as independent, the GPU threads within a thread block return early from the outer loop (see lines 25–27). Otherwise, the GPU threads continue with the subsequent separation set belonging to their tasks. The GPU kernel terminates once all launched GPU threads have finished their computation.

4.2.5 GPU Acceleration for Levels 2 and Higher

In contrast to the first two levels, $l = 0, 1$, processing of levels $l \geq 2$ introduces two challenges, namely the computation of separation sets and the increased complexity of calculating a pseudo-inverse matrix for the p-value computation.

A separation set for an edge $E^{i,j}$ with $i, j = 1, \ldots, N$ and $i \neq j$ in any level $l \geq 2$ is constructed as a combination of variables of size l drawn from the adjacency set $adj(\mathcal{C}^l, V_i)$. Thus, a combination function COMBINATIONS(...) is needed, which computes the set of all possible separation sets $\mathbf{S}^{i,j,l}$ for an edge $E^{i,j}$. In contrast, in level $l = 0$, the separation set is empty; in level $l = 1$, the possible separation sets of size 1 are obtained from the adjacency set $adj(\mathcal{C}^l, V_i)$.

In levels $l = 0, 1$, the calculation of the matrix inverse for the p-value computation is based upon a two × two or a three × three matrix. Hence, implementing the matrix inverse is straightforward (see Algorithms 4 and 5). Yet, for levels $l \geq 2$, the matrix inverse for the p-value computation is based upon a matrix of dimension $(l+2) \times (l+2)$. Thus, calculating the pseudo-inverse matrix requires more computing steps, e.g., using the Moore-Penrose algorithm [165].

In the following, we present two variants for GPU-accelerated processing of levels $l \geq 2$. The first variant leverages functionality provided by existing `CUDA`-X libraries that build upon the `CUDA` framework. The second variant uses a customized GPU kernel and adopts functions from `cupc` [287] to compute the pseudo-inverse and the separation set candidates on the GPU.

Algorithm 6 CUDA-X library-based variant for levels $l \geq 2$
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$, estimated skeleton matrix $\mathcal{C}$, copy of estimated skeleton matrix $\hat{\mathcal{C}}$, separation sets matrix Sep, significance level α, number of data samples n, level l

1: **for** $row \leftarrow 0; row < |\mathbf{V}|; row \leftarrow row + 1$ **do**
2: **for** $col \leftarrow row; col < |\mathbf{V}|; col \leftarrow col + 1$ **do**
3: **if** $\hat{\mathcal{C}}[row][col] = 0$ **then**
4: continue
5: **end if**
6: $\mathbf{S}^{row,col,l} \leftarrow$ COMBINATIONS$(row, col, \hat{\mathcal{C}}, l)$
7: **for** $S^{row,col} \in \mathbf{S}^{row,col,l}$ **do**
8: $tCor \leftarrow$ CONSTRUCTAUXCORMATRIX$(sCor, row, col, S^{row,col})$
9: ALLOCATEONGPU$((\{U, VT, S^{-1}, iCor\}, (l+2)^2), (\{SV\}, l+2))$
10: CUSOLVERDNGESVD$(U, VT, SV, tCor)$
11: INVERSEDIAGONAL(S^{-1}, SV)
12: CUBLASSGEMM$(U, VT, S^{-1}, iCor)$
13: TRANSFERFROMGPU$(iCor)$
14: $r \leftarrow \frac{-1 \times iCor[1]}{\sqrt{(iCor[0] \times iCor[l+2+1])}}$
15: $z \leftarrow \sqrt{(n-1-3)} \times 0.5 \times$ LOG1P$(\frac{2 \times r}{1-r})$
16: p $\leftarrow 2 \times (1 -$ NORMCDF$(z))$
17: **if** p $\geq \alpha$ **then**
18: $\mathcal{C}[row][col] \leftarrow 0$
19: $\mathcal{C}[col][row] \leftarrow 0$
20: **for** $a \leftarrow 0; a < l; a \leftarrow a + 1$ **do**
21: $Sep[row][col][a] \leftarrow S^{row,col}[a]$
22: **end for**
23: break
24: **end if**
25: **end for**
26: **end for**
27: **end for**

CUDA-X Library-Based Variant

In the following, we describe an algorithm for processing levels $l \geq 2$ in the case of the Gaussian distribution model. The algorithm utilizes functions from CUDA-X libraries to realize the required matrix pseudo-inverse calculation. To the best of our knowledge, CUDA-X libraries do not ship a function for matrix pseudo-inverse computation. But the CUDA-X libraries cuBLAS [176] and cuSOLVER [179] provide functions for singular value decomposition [119] and matrix multiplication, which are building blocks to compute the pseudo-inverse matrix. The pseudo-inverse is needed to compute the p-value. Therefore the CUDA-X library functions need to be called in the context of each CI test. As a result, we assume sequential processing of the CI tests and cannot apply any of the previously introduced mappings from Section 4.1.

Next, we describe the input, the output, and the operations performed by our proposed algorithm, which we outline in Algorithm 6. Afterward, we discuss the poor GPU utilization as a key limitation of the algorithm and present two strategies to achieve a higher GPU utilization for better performance.

Input and Output

The algorithm receives the vertex set $\mathbf{V}$, the correlation matrix $sCor$, the estimated skeleton matrix $\mathcal{C}$, the copy of the estimated skeleton matrix $\hat{\mathcal{C}}$, the separation sets matrix Sep, the significance level α, the number of data samples n, and the current level l as input. The algorithm has no explicit output. During execution, the data structures storing the estimated skeleton matrix $\mathcal{C}$ and the separation sets matrix Sep are directly manipulated.

Description of the Operations of the `CUDA`-X Library-Based Algorithm

The described algorithm is executed for each level $l \geq 2$ individually. At the start, a CPU thread iterates over the upper triangular of the estimated skeleton matrix copy $\hat{\mathcal{C}}$ using the row and column indices row, col to access elements. Then, it skips all deleted edges (see lines 3–5 of Algorithm 6, p. 75).

For the existing edges (see lines 6–23), the algorithm first computes the set of all possible separation sets $\mathbf{S}^{row,col,l}$ of size l using the function COMBINATIONS$(row, col, \hat{\mathcal{C}}, l)$ (see line 6). The function COMBINATIONS$(\dots)$ extracts the adjacency sets $adj(\mathcal{C}^l, V_{row})$ and $adj(\mathcal{C}^l, V_{col})$ from $\hat{\mathcal{C}}$ to compute all possible separation sets of size l, which are returned.

Next, for each separation set $S^{row,col} \in \mathbf{S}^{row,col,l}$, the p-value is calculated (see lines 8–16). Therefore, an auxiliary correlation matrix $tCor$ of dimension $(l+2) \times (l+2)$ is constructed. It is filled with values from the sample correlation matrix $sCor$ at positions according to combinations of the variables V_{row}, V_{col} and $S^{row,col}$ (see line 8). Additional auxiliary matrices $U, VT, S^{-1}, iCor$ of dimension $(l+2) \times (l+2)$ and a vector SV of size $l+2$ are allocated in GPU global memory (see line 9). The matrices U, VT, and the vector SV are needed for singular value decomposition. The matrix $iCor$ will store the inverse of the auxiliary correlation matrix. After these steps, the function CUSOLVERDNGESVD$(\dots)$ from the library `cuSOLVER` is called to perform singular value decomposition. Next, the diagonal matrix S^{-1} is computed based on SV as follows:

$$S(i,j)^{-1} = \begin{cases} \frac{1}{SV(i)}, & \text{if } i = j; \\ 0, & \text{otherwise, with i,j = 0, \dots, l+1.} \end{cases} . \tag{4.22}$$

To conclude the inverse calculation, the function CUBLASSGEMM$(\dots)$ from the library `cuBLAS` performs matrix multiplication. The inverted correlation matrix $iCor$ is copied from the GPU to the host system's DRAM. Lastly, to compute the p-value, the same operations used in the GPU kernel for level $l = 1$ are performed on the CPU (see lines 14–16). Only the procedures in lines 8–13 are performed on the GPU or interact with it.

If p $\geq \alpha$ evaluates true, the edge is removed from the skeleton matrix $\mathcal{C}$, the corresponding separation set $S^{row,col}$ is stored in Sep (see lines 17 – 22) and the algorithm proceeds with the subsequent edge. Otherwise, the algorithm processes the next separation set.

Limitation of the Algorithm

The outlined algorithm makes use of existing functionality to compute the p-value. Yet, the utilization of functions from the `CUDA`-X library restricts the use of the GPU to the distinct function calls in lines 10–12 of Algorithm 6. Further,

the GPU operations are performed on small-sized matrices with dimensions starting at four × four in level $l = 2$. Thus, few GPU threads are launched, resulting in high underutilization of the GPU compute resources.

To the best of our knowledge, functions from `cuBLAS` or `cuSOLVER` can only be called from the CPU and are not available in GPU device code. Therefore, the algorithm only conducts a single CI test at a time. Thus, effectively processing all edges and CI tests of edges sequentially. The following two strategies are considered to overcome this limitation and achieve higher GPU utilization.

Strategies to Achieve a Higher GPU Utilization

The first strategy to achieve higher GPU utilization employs `CUDA` streams [137] to perform multiple CI tests in parallel. `CUDA` streams allow the concurrent execution of multiple operations on the GPU. Thus, we let multiple CPU threads process separate edges simultaneously. These CPU threads launch the GPU operations in separate `CUDA` streams to achieve parallel execution.

The second strategy to achieve higher GPU utilization builds upon the idea of fusing the computations for multiple CI tests into the same GPU operations. Therefore, the auxiliary data matrix $tCor$ is enlarged to store the values of the sample correlation matrix, referring to multiple CI tests along its diagonal. For example, consider fusing two 4×4 auxiliary correlation matrices $tCor_1, tCor_2$. The resulting fused auxiliary correlation matrix $tCor_{fused}$ of dimension 8×8 stores $tCor_1$ in the upper left block and $tCor_2$ in the lower right block. Hence, this strategy trades-off GPU compute resource utilization with an increased memory demand of $tCor_{fused}$ and $iCor_{fused}$.

Both strategies address higher GPU utilization. Yet, they also come with strong restrictions. For the `CUDA` stream-based strategy, the number of possible `CUDA` streams, limited to several dozens [137], becomes a restriction. Further, this strategy performs many steps on the CPU. The matrix fusion strategy is limited by its memory demand. Also, it is not guaranteed that the utilized `CUDA`-X libraries efficiently handle many fused CI tests, as the data structure's size increases quadratically to the number of fused CI tests. Therefore, we consider a GPU kernel-based variant, too.

GPU Kernel-Based Variant

In the following, we describe an algorithm for processing levels $l \geq 2$ that uses a customized GPU kernel in the case of the Gaussian distribution model. We first discuss the chosen parallel execution strategy according to Section 4.1. Afterward, we describe the input, the output, and the GPU kernel launch parameters. Lastly, we elucidate the operations performed within the GPU kernel and highlight differences in related work [287].

Execution Strategy: Mapping of Tasks to Execution Units

The GPU kernel for levels $l \geq 2$ presents a generalized case of the GPU kernel for level $l = 1$. Thus, in levels $l \geq 2$, we apply the same task definition and task mapping as for the GPU kernel for level $l = 1$ (see Section 4.2.4). Note that the range of the number of CI tests per edge $E^{i,j}$ increases with higher levels l, up to a limit. Therefore, the choice of an appropriate value for the parameter u according to (4.21) is vital to achieving a balanced load. In Algorithm 7 (see p. 78), we outline the GPU kernel algorithm for any level $l \geq 2$.

Algorithm 7 GPU kernel for levels $l \geq 2$ for the Gaussian Distribution Model
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$, estimated skeleton matrix $\mathcal{C}$, copy of estimated skeleton matrix $\hat{\mathcal{C}}$, separation sets matrix Sep, significance level α, number of data samples n, level l
# of blocks: $N \times N$
# of threads per block: δ

1: $row \leftarrow bx, col \leftarrow by, N \leftarrow |\mathbf{V}|$
2: **if** $row < col \,\&\, \mathcal{C}[row][col] \neq 0$ **then**
3: Initialize list $pVals$ of length δ in shared memory
4: Initialize lists $w, r1$ of length $(l+2)$ in thread-local memory
5: Initialize matrices $tCor, iCor, v, r2$ of size $(l+2) \times (l+2)$ in thread-local memory
6: $finished \leftarrow false$
7: **for** $y \leftarrow 0; y < 2; y \leftarrow y+1$ **do**
8: **for** $a \leftarrow tx; a < \binom{|adj(\hat{\mathcal{C}}, V_{row})|-1}{l}; a \leftarrow a+\delta$ **do**
9: $pVals[tx] \leftarrow -1$
10: $S^{row,col} \leftarrow$PARALLEL_COMB$(|adj(\hat{\mathcal{C}}, V_{row})|-1, l, a)$
11: SETAUXCORMATRIX$(tCor, sCor, row, col, S^{row,col})$
12: PSEUDOINVERSE$(iCor, tCor, v, r1, w, r2)$
13: $r \leftarrow \frac{-1 \times iCor[1]}{\sqrt{(iCor[0] \times iCor[l+2+1])}}$
14: $z \leftarrow \sqrt{(n-1-3)} \times 0.5 \times$LOG1P$(\frac{2 \times r}{1-r})$
15: $pvals[tx] \leftarrow 2 \times (1 -$NORMCDF$(z))$
16: SYNCTHREADS()
17: **if** $tx = 0$ **then**
18: **for** $b \leftarrow 0; b < \delta; b \leftarrow b+1$ **do**
19: **if** $pVals[b] \geq \alpha$ **then**
20: $\mathcal{C}[row][col] \leftarrow 0$
21: $\mathcal{C}[col][row] \leftarrow 0$
22: $S^{row,col} \leftarrow$PARALLEL_COMB$(|adj(\hat{\mathcal{C}}, V_{row})|-1, l, a+b)$
23: **for** $c \leftarrow 0; c < l; c \leftarrow c+1$ **do**
24: $Sep[row][col][c] \leftarrow S^{row,col}[c]$
25: **end for**
26: break
27: **end if**
28: **end for**
29: **end if**
30: SYNCTHREADS()
31: **if** $\mathcal{C}[row][col] = 0$ **then**
32: $finished \leftarrow true$
33: $break$ ▷ Early return
34: **end if**
35: **end for**
36: **if** $finished \neq true$ **then**
37: $row \leftarrow by; col \leftarrow bx;$
38: **else**
39: break
40: **end if**
41: **end for**
42: **end if**

Input and Output

The GPU kernel receives the same input parameters as the GPU kernel for level $l = 1$ (see Algorithm 5) with the addition of the current level l. Like the GPU

kernel for level $l = 1$, the GPU kernel for any level $l \geq 2$ has no output but directly changes entries in the data matrices $\mathcal{C}$ and Sep in GPU global memory.

GPU Kernel Launch Parameters

The GPU kernel is launched with $N \times N$ thread blocks and δ threads per block. Like the GPU kernel for level $l = 1$, we set $\delta = u$ following (4.21).

Description of the Operations Within the GPU Kernel

Once the GPU kernel is launched, each GPU thread sets indices row and col according to its block ids (see line 1 of Algorithm 7, p. 78). These indices identify the edge $E^{row,col}$ that all GPU threads within the same thread block process. In line 2, the algorithm checks if the edge $E^{row,col}$ is in the upper triangular of the estimated skeleton matrix and whether the edge is still present. If both checks hold, the GPU threads start processing the edge $E^{row,col}$.

At first, several auxiliary data structures are initialized. These data structures include a list for the p-values $pVals$ of length δ allocated in shared memory and auxiliary lists and matrices allocated in GPU thread-local memory needed for the pseudo-inverse calculation. In the `CUDA`-based implementation of the GPU kernel, these data structures are initialized as fixed-size arrays at compile time, using templated GPU kernels. Using fixed-size arrays avoids explicit memory allocations via MALLOC(...), which presents a performance bottleneck. As a final preparation step, we set the flag $finished = false$, indicating that the edge $E^{row,col}$ was not yet processed. The flag indicates early termination at a later stage of the algorithm (see line 39).

The following outer loop (see lines 7 – 41) is added for technical reasons. It is used to process the separation sets constructed from the neighborhoods of the variables V_{row}, V_{col} separately. The algorithm starts with the neighborhood of V_{row}. Next, the tasks belonging to the edge $E^{row,col}$ are processed in parallel by δ GPU threads. Therefore the GPU threads iterate all possible separation sets belonging to their task concerning the neighborhood of V_{row} first (see lines 8–40). Within each iteration, each GPU thread considers one separation set indexed by a. Based on the index a, the current level l, and the size of the neighborhood of V_{row}, each GPU thread determines its corresponding separation set using the parallel combination function, PARALLEL_COMB(...). Note this function is adapted based on related work [287].

Next, the function SETAUXCORMATRIX(...) sets the entries of an auxiliary correlation matrix $tCor$. The values for $tCor$ are extracted from the sample correlation matrix $sCor$ at positions related to the variables V_{row}, V_{col}, and the GPU thread's current separation set $S^{row,col}$. The auxiliary correlation matrix is inverted using an implementation of the Moore-Penrose algorithm [165] within the function PSEUDOINVERSE(...) (see line 12). Afterward, the p-value is computed using the inverted correlation matrix $tCor$ according to the equations defined in Section 2.4.1. Each GPU thread stores its computed p-value in the list $pVals$ at the position corresponding to the GPU thread's thread id.

After synchronizing the GPU threads within the same thread block, one primary GPU thread, i.e., $tx = 0$, compares each p-value in $pVals$ against the significance level α. If one p-value is greater than or equal to α, the according entries in the estimated skeleton matrix $\mathcal{C}$ are set to zero, i.e., $\mathcal{C}[row][col] = 0$, $\mathcal{C}[col][row] = 0$. Further, the corresponding separation set $S^{row,col}$ is computed

and stored in the separation set matrix *Sep*. Once all entries of $pVals$ are processed, the GPU threads within the same thread block are synchronized.

If the primary GPU thread removed the edge $E^{row,col}$, the GPU threads set the flag for early termination $finished = true$ and exit early from both loops. Otherwise, if all possible separation sets constructed from the neighborhood of V_{row} have been processed, then the indices *row* and *col* are exchanged (see line 37). Hence, in the second iteration of the outer loop, all separation sets computed from the neighborhood of V_{col} are processed. Once all GPU threads have terminated early or finished the second iteration of the outer loop, the GPU kernel terminates.

Additional Notes

Generally, the GPU kernel-based variant for higher levels $l \geq 2$ follows similar ideas as the `cuPC-E` algorithm proposed by Zarabavani et al. [287]. In contrast to our proposed GPU kernel, `cuPC-E` operates on a compacted version of the adjacency structure (see Section 3.1.3 for more detail). Further, the `cuPC-E` algorithm uses shared memory to improve access to the current row of the compacted adjacency structure. In contrast, our proposed algorithm uses shared memory for local communication of computed p-values. Thus, `cuPC-E` and its optimized version `cuPC-S` require atomic operations to communicate the calculated p-values, i.e., the results from CI tests, via GPU global memory. As our proposed GPU kernel operates on the uncondensed adjacency structure, we launch several threads blocks that do not perform p-value calculations. These thread blocks exit the algorithm at line 2, as their indices refer to edges removed in previous levels.

4.3 GPU-Accelerated Adjacency Search in PC-Stable for Discrete Data

This section details our algorithm for a GPU-accelerated adjacency search in PC-stable for discrete data (see Section 2.4.2). First, we outline the overall idea of our proposed algorithm in Section 4.3.1 and emphasize GPU memory management of auxiliary data structures. Afterward, we present the developed GPU kernels for level $l = 0$ (see Section 4.3.2) and for any level $l \geq 1$ (see Section 4.3.3). For each of the GPU kernels, we describe the execution strategy employed according to our definitions from Section 4.1. Further, we explain the operations performed by each GPU thread and mention implementation detail based on the `CUDA` framework [170]. *Parts of this section have been published in a research paper [81].*

4.3.1 Outline of the GPU-Accelerated Adjacency Search

Our proposed GPU-accelerated adjacency search for discrete data applies Pearson's χ^2 test [190] as a CI test. The CI test builds upon the computation of marginals over the contingency table, which requires auxiliary data structures. We discuss different strategies to handle these auxiliary data structures on the GPU. Afterward, we describe our proposed GPU-accelerated adjacency search algorithm in detail.

Handling the Memory Demand of Auxiliary Data Structures Needed for the Pearson χ^2 Test

The Pearson χ^2 test builds upon the computation of marginals over the contingency table to decide on conditional independence [190]. Thus, in the context of each performed CI test, a contingency table is constructed from the observational data D of the respective variables V_i,V_j, and $S^{i,j}$, with $V_i, V_j \in \mathbf{V}, i \neq j$ and $S^{i,j} \subset \mathbf{V} \setminus \{V_i, V_j\}$ (for detail see Section 2.4.2). The computed contingency table and the marginals are stored temporarily while performing the corresponding CI test. Therefore, auxiliary data structures are required. In the following, we present the memory demand of the auxiliary data structures with respect to relevant parameters. We discuss which memory in the GPU memory hierarchy to use for storing the auxiliary data structures and weigh different allocation strategies when using the GPU global memory.

Memory Demand of the Auxiliary Data Structures

The memory demand of the auxiliary data structures storing the contingency table and marginals during a CI test depends on the size of the domains $\mathcal{V}_i$, $\mathcal{V}_j$, and $\mathcal{S}^{i,j}$ of the corresponding variables V_i,V_j, and $S^{i,j}$. Each of the auxiliary data structures stores several integer values. The number of entries of the contingency table s_{ct} and marginals $\{s_{+v_j s^{i,j}}, s_{v_i + s^{i,j}}, s_{++s^{i,j}}\}$ are computed as follows:

$$\begin{aligned} s_{ct} &= |\mathcal{V}_i| \times |\mathcal{V}_j| \times |\mathcal{S}^{i,j}|, \\ s_{+v_j s^{i,j}} &= |\mathcal{V}_i| \times |\mathcal{S}^{i,j}|, \\ s_{v_i + s^{i,j}} &= |\mathcal{V}_j| \times |\mathcal{S}^{i,j}|, \\ s_{++s^{i,j}} &= |\mathcal{S}^{i,j}|. \end{aligned} \tag{4.23}$$

According to (4.23), the memory demand of one CI test increases drastically with the size of the separation set $S^{i,j}$, i.e., the level l in the adjacency search. Note that $|\mathcal{S}^{i,j}|$ is computed as the product from the discrete domains $\mathcal{V}_k$ for each variable $V_k \in S^{i,j}$. Moreover, the memory demand of a CI test increases polynomially with the size of the variables' domains. Table 4.4 (see p. 82) exemplifies these two dependencies showing the number of entries needed in the auxiliary data structures for increasing the variables' domains and level l, i.e., the size of the separation set $S^{i,j}$, for one CI test. Assuming 4-byte integers for each entry results in a memory demand, for the illustrated examples, that ranges from 32 B to 1 296 kB. Thus, depending on the level l and size of the variables' domains, the auxiliary data structures of one CI test fit in GPU thread-local memory, in shared memory, or require GPU global memory. Generally, the runtime performance of the adjacency search benefits from storing the auxiliary data structures closer to the actual processing units. However, these types of memory are often limited in capacity. We discuss the applicability of the three memory options for storing the auxiliary data structures below. In this context, we need to consider the execution of hundreds to thousands of CI tests in parallel, which increases the memory demand for storing the auxiliary data structures by a corresponding factor.

level l	0			2			4		
$\lvert\mathcal{V}_i\rvert = \lvert\mathcal{V}_j\rvert = \lvert\mathcal{S}^{i,j}\rvert$	2	4	8	2	4	8	2	4	8
number of entries	8	24	80	36	400	5 184	144	6 400	331 776

Table 4.4: Exemplary sizes of auxiliary data structures. The number of entries in the auxiliary data structures is computed based on level l and the size of the variables' domains. Showing the impact of increasing the level l and the size of the variables' domains $|\mathcal{V}_i|, |\mathcal{V}_j|, |\mathcal{S}^{i,j}|$.

Use of GPU Thread-Local Memory

In the case that GPU thread-local memory is mapped to SM registers, its size is limited. Thus, GPU thread-local memory is only relevant in edge cases. For example if level $l = 0$ and $|\mathcal{V}_i| = |\mathcal{V}_j| = |\mathcal{S}^{i,j}| = 2$ is considered. For cases requiring more memory, GPU thread-local memory regresses to GPU global memory, increasing overhead and making tracking of the utilized GPU global memory difficult. For our GPU-accelerated adjacency search, we refrain from using GPU thread-local memory to store the auxiliary data structures.

Use of Shared Memory

The amount of shared memory is limited to several KB per SM. If multiple thread blocks are concurrently scheduled on the same SM, the available shared memory is split between the thread blocks. Thus, it is realistic to store the auxiliary data structures in shared memory in most cases of level $l = 0$, which we consider in our algorithm. In higher levels l, the auxiliary data structures could be stored in shared memory for selected cases with low-sized domains. As these present few cases, we do not cover storing auxiliary data structures in shared memory in our GPU-accelerated adjacency search for levels $l \geq 1$.

Use of GPU Global Memory

GPU global memory has a capacity of dozens of GB, allowing it to store the auxiliary data structures while processing hundreds of CI tests in parallel in higher levels l with large-sized domains of variables. Therefore, GPU global memory is a compelling option for storing the auxiliary data structures during the GPU-accelerated adjacency search. Below, we discuss three options for allocating auxiliary data structures in GPU global memory. The three options must consider that the memory demand of CI tests increases with higher levels l and that the memory demand of individual CI tests also differs within a level.

Further, note that even the amount of available GPU global memory can be exceeded. In that case, the number of CI tests executed in parallel can be restricted to a certain degree. Yet, there exist cases in which GPU execution should no longer be considered. For example, if the memory demand of a single CI test's auxiliary data structures exceeds the available GPU global memory, or if the number of CI tests executed in parallel becomes so low, the GPU compute resources are underutilized.

Memory Allocation Within the GPU Kernel

As a first option, we discuss memory allocation of the auxiliary data structures within the GPU kernel. Each GPU thread allocates the required GPU global memory calling the function MALLOC(...) during GPU kernel execution. The GPU thread computes the exact size of the auxiliary data structures and allocates only the necessary amount of memory. Thus, this option poses high flexibility and allocates only the required memory. Yet, memory allocation from inside a GPU kernel degrades the overall performance of an executed GPU kernel [278]. Further, this option requires explicit error handling inside the GPU kernel in cases where the GPU global memory is exceeded upon a call of MALLOC(...) in any GPU thread. Despite its high flexibility, we do not consider the option to allocate the memory for the auxiliary data structure within the GPU kernel due to its impact on performance and overhead in error handling.

Allocation of Required GPU Memory Before GPU Kernel Launch

As a second option, we discuss allocating GPU global memory for the auxiliary data structures before launching the GPU kernel for a distinct level l. In particular, this option assumes that data for all possible CI tests in the current level l is allocated before the GPU kernel launch. When performing the CI tests, each GPU thread then computes the CI test's index to access a unique memory location. While this option avoids costly GPU global memory allocations from within the GPU kernel, it introduces multiple drawbacks. First, a computational effort is added to determine the memory demand for all CI tests in the current level l. This step requires building the set of separation set combinations $\mathbf{S}^{i,j,l}$ for all edges $E^{i,j}$, which accumulates runtime. Second, GPU global memory allocation for auxiliary data structures for all possible CI tests results in a high memory footprint. Note that GPU global memory is reserved even for those CI tests that are never performed due to early termination. Lastly, GPU global memory reallocation is needed after each level l, given that the required memory for each CI test increases with each subsequent level. The reallocation results in fragmented GPU global memory, which incurs overhead or causes memory allocation errors [253]. Due to these drawbacks, we avoid this option when designing our GPU-accelerated adjacency search.

Allocation of a Fixed-Size Memory Block Before GPU Kernel Launch

As a third option, we consider allocating a fixed-size GPU global memory block to store the auxiliary data structure before the first GPU kernel launch. The algorithm reuses the same GPU global memory block for performing CI tests across multiple levels to avoid memory fragmentation. Thus, the algorithm sets all values within the GPU global memory block to zero at the start of each level l. Note that GPU programming frameworks commonly support such functionality, e.g., see `CUDA`'s function CUDAMEMSET(...).

We use a conservative approximation to determine the size of the fixed-size GPU global memory block. First, we compute the required maximum amount of memory for the auxiliary data of a single CI test in the maximum level ml, denoted by mem_{ci}. To avoid overhead computing the exact memory demand for each CI test, we take mem_{ci}, which presents an upper bound on the memory demand for all CI tests. Using the value of mem_{ci}, we then calculate the memory

size of the fixed-size block that stores the auxiliary data structures. We denote this memory size by mem_{aux}.

The value of mem_{ci} is computed using (4.24), where $\max_{i=1,\ldots,N}\{|\mathcal{V}_i|\}$ denotes the largest domain within the set of variables, and $\text{SIZEOF}(dt)$ provides the size of the used data type in bytes. Note that the value of ml according to the underlying true DAG of the CGM is not known in advance. Hence, either a value for ml is provided as an input parameter, which restricts the search, or ml is assumed according to the worst case, i.e., a fully connected CGM, and set as $ml = N - 2$.

$$mem_{ci} = \text{SIZEOF}(dt) \times (\max_{i=1,\ldots,N}\{|\mathcal{V}_i|\})^{ml+2} \tag{4.24}$$

Afterward, the memory size of the fixed-size GPU global memory block for the auxiliary data structures mem_{aux} is calculated as follows:

$$mem_{aux} = \begin{cases} mem_{ci} \times N \times N \times \gamma & \text{if } mem_{ci} \times N \times N \times \gamma \leq mem_{free} \\ mem_{free} & \text{if } mem_{free} < mem_{ci} \times N \times N \times \gamma \\ & \text{and } mem_{ci} \leq mem_{free} \\ 0 & \text{otherwise.} \end{cases} \tag{4.25}$$

The parameter mem_{free} denotes the available GPU global memory after allocating all input and output data structures. The parameter γ is a factor that represents the number of CI tests per edge that are conducted in parallel.

The first case in (4.25) shows that enough GPU global memory is available to store the auxiliary data structures for γ CI tests for all edges. In the second case, the free GPU global memory is exceeded if γ CI tests for all edges are executed in parallel. However, the amount of available GPU global memory still allows processing at least one CI test. Thus, execution on GPU is still possible, but the degree of parallelism must be restricted. In the last case, i.e., $mem_{aux} = 0$, the amount of GPU global memory is insufficient to conduct a single CI test on the GPU, and the algorithm is aborted. In our GPU-accelerated adjacency search, described in the following, we employ this option to manage the GPU global memory for the auxiliary data structures.

Description of the GPU-Accelerated Adjacency Search

In Algorithm 8 (see p. 85), we outline our GPU-accelerated adjacency search tailored to discrete data. The algorithm builds upon the considerations mentioned above to manage GPU global memory for the auxiliary data structures. Algorithm 8 focuses on the necessary operations, data structures, and interactions between the host system and the GPU. The launched GPU kernels are detailed in the subsequent Sections 4.3.2 and 4.3.3.

Input

The algorithm receives as input the vertex set $\mathbf{V} = \{V_1, \ldots, V_N\}$ representing the dataset's N variables (see Section 2.1) and the observational data matrix D of dimension $N \times n$. Further, the algorithm gets the significance level α, the maximum level ml, and the number of data samples n as input. For technical

Algorithm 8 GPU-accelerated adjacency search for discrete data
Input: Vertex set $\mathbf{V}$, observational data matrix D, significance level α, maximum level ml, number of data samples n
Output: Estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep

1: Let dom be a vector of size $|\mathbf{V}|$
2: **for** $V_i \in \mathbf{V}$ **do**
3: $dom[V_i] \leftarrow |\mathcal{V}_i|$
4: **end for**
5: $mdom \leftarrow$ MAX(dom)
6: $mem_{free} \leftarrow$ ESTIMATEFREEGPUMEMORY($|\mathbf{V}|, n, ml$)
7: $mem_{ci} \leftarrow$ ESTIMATEAUXILIARYDATAMEMORYDEMAND($mdom, ml$)
8: **while** $mem_{ci} > mem_{free}$ & $ml \geq 0$ **do**
9: $ml \leftarrow ml - 1$
10: $mem_{free} \leftarrow$ ESTIMATEFREEGPUMEMORY($|\mathbf{V}|, n, ml$)
11: $mem_{ci} \leftarrow$ ESTIMATEAUXILIARYDATAMEMORYDEMAND($mdom, ml$)
12: **end while**
13: **if** $ml < 0$ **then**
14: Return with error: Not enough global memory available on the GPU
15: **end if**
16: ALLOCATEONGPU(($\{m_{aux}\}$,COMPUTEMEMBLOCKSIZE(mem_{ci}, mem_{free}, N)))
17: $l \leftarrow 0$
18: Let $\mathcal{C}$ and $\hat{\mathcal{C}}$ be $|\mathbf{V}| \times |\mathbf{V}|$ matrices with all entries set to 1
19: Let Sep be an $|\mathbf{V}| \times |\mathbf{V}| \times ml$ matrix with all entries set to -1
20: TRANSFERTOGPU($\mathbf{V}, D, \mathcal{C}, \hat{\mathcal{C}}, Sep$)
21: **if** HASENOUGHSHAREDMEMORY(...) **then**
22: LAUNCHGPUKERNEL($level0Shared, \{\mathbf{V}, D, \mathcal{C}, Sep, \alpha, n, dom, mdom\}$)
23: **else**
24: LAUNCHGPUKERNEL($level0, \{\mathbf{V}, D, \mathcal{C}, Sep, \alpha, n, dom, mdom, m_{aux}\}$)
25: **end if**
26: $update(ml)$
27: **while** $ml > l$ **do**
28: $l = l + 1$
29: COMPACT($\hat{\mathcal{C}}, \mathcal{C}$)
30: ADJUSTKERNELLAUNCHDIMENSIONS(l, mem_{aux})
31: LAUNCHGPUKERNEL($levelL, \{\mathbf{V}, D, \mathcal{C}, \hat{\mathcal{C}}, Sep, \alpha, n, dom, mdom, m_{aux}\}$)
32: UPDATE(ml)
33: **end while**
34: TRANSFERFROMGPU($\mathcal{C}, Sep$)
35: **return** $\mathcal{C}$, Sep

reasons, we assume that the observational data D has been pre-processed so that for each variable $V_i \in \mathbf{V}$, its corresponding domain $\mathcal{V}_i$ ranges from integer values $\{0, \ldots, |\mathcal{V}_i|\}$. Further, we assume that the observational data D is stored in a column-major format, where each column represents one variable and each row refers to one data sample. The significance level α serves as a threshold while performing the CI tests. The maximum level ml is an optional parameter that restricts the levels l considered during the adjacency search, i.e., the maximum size of any separation set.

Output

Upon successful completion, the adjacency search outputs two matrices. The first matrix of dimension $N \times N$ stores the estimated skeleton $\mathcal{C}$. The second matrix, denoted Sep, of dimension $N \times N \times ml$, contains the separation sets $S^{i,j}$ corresponding to the pairs of variables (V_i, V_j) estimated to be independent.

Description of the Procedure

The algorithm starts initializing necessary data structures, computing the memory demand of the auxiliary data structures, and setting parameters. First, it initializes a vector dom of size N, storing the sizes of the domains $|\mathcal{V}_i|$ of each variable $V_i \in \mathbf{V}$ (see lines 2–4 of Algorithm 8, p. 85). Note this operation is implemented in a dedicated GPU kernel, which counts the unique values of all variables $\mathbf{V}$ in parallel. Next, the size of the largest domain $mdom$ is determined. This value is required in the subsequent steps to compute the size of the fixed-size GPU global memory block for storing the auxiliary data structures. Therefore, the available GPU global memory mem_{free} is estimated in function ESTIMATEFREEGPUMEMORY(...), which considers the memory demand for the input and output data structures. The function ESTIMATEAUXILIARYDATAMEMORYDEMAND(...) computes the memory demand for one CI test mem_{ci} based on the values of $mdom$ and ml according to (4.24). The algorithm reduces the maximum level $ml = ml - 1$ and recalculates mem_{free} and mem_{ci} if the available GPU global memory mem_{free} is smaller than the memory required for one CI test mem_{ci}. This operation is repeated until either mem_{ci} is smaller than mem_{free} or the maximum level ml is smaller than zero. Suppose that the maximum level is smaller than zero. Then the algorithm terminates with an out of GPU memory error. Otherwise, the fixed-size GPU memory block m_{aux} for the auxiliary data structures is allocated on the GPU (see line 16). This operation concludes the preparation steps for the memory management of the auxiliary data structures.

Next, in lines 17–34, the proposed algorithm for discrete data follows similar steps as the GPU-accelerated algorithm for the Gaussian distribution model. First, the level l is set to 0. Then, the data matrices for the estimated skeleton $\mathcal{C}$ and its compacted version $\hat{\mathcal{C}}$ of dimension $N \times N$ and the matrix Sep of dimension $N \times N \times ml$ are allocated and initialized with values of 1 or -1. The algorithm transfers all data structures to the GPU required when executing the GPU kernels (see line 20).

Now, the algorithm starts processing level $l = 0$. There exist two variants for the GPU kernel for level $l = 0$. The first variant of the GPU kernel that utilizes shared memory to store the auxiliary data structures is used if the maximum domain $mdom$ is not large, i.e., the auxiliary data structures fit into shared memory. Otherwise, the second variant of the GPU kernel is called. This GPU kernel uses m_{aux} to store the auxiliary data structures. Hence, it does not benefit from faster access to shared memory but can process arbitrarily-sized maximum domains. For detail on the GPU kernel using shared memory, see Section 4.3.2.

After successfully processing level $l = 0$, the UPDATE(...) function checks if the maximum level ml can be reduced, according to (4.19). The subsequent levels are processed iterative, increasing the level l by one within each iteration, as long as the condition $ml > l$ holds (see lines 27 – 33). In this loop, the COMPACT(...)

procedure that builds upon related work [287] compacts the estimated skeleton matrix $\mathcal{C}$ and stores the result in $\hat{\mathcal{C}}$. In the compacted version, each row stores the indices to the remaining adjacent variables consecutive in memory. Next, the procedure ADJUSTKERNELLAUNCHDIMENSIONS(...) adjusts the dimensions of the GPU kernel launch parameter according to the current level l and the fixed-size GPU global memory block. The GPU kernel launch parameters are adjusted to cover the second case of (4.25), i.e., to restrict the number of CI tests performed in parallel if GPU global memory is exceeded otherwise. The GPU kernel for the current level l is launched with the adjusted launch parameters. For detail on the GPU kernel for levels $l \geq 1$, see Section 4.3.3. Once the GPU kernel has completed its computation, the UPDATE(...) function is called, and the algorithm proceeds with the subsequent level.

In the case that $ml \leq l$, the estimated skeleton matrix $\mathcal{C}$ and the separation sets matrix *Sep* are copied from the GPU to the host system's DRAM. Afterward, both data structures are returned, and the algorithm finishes.

`CUDA`-Based Implementation Detail

Our reference implementation of Algorithm 8 (see p. 85) targets NVIDIA GPUs. Therefore, the GPU kernels are implemented using `CUDA` [170]. The implementation utilizes the following `CUDA`-based procedures to manage GPU global memory. In the procedure ESTIMATEFREEGPUMEMORY(...), the total amount of GPU global memory is queried using CUDAMEMGETINFO(...). Data structures required on the GPU are allocated using CUDAMALLOC(...), and transfers are initiated via CUDAMEMCPY(...). If needed, CUDAMEMSET(...) is used to set all data structure entries to a given value, e.g., in lines 18–19.

The `CUDA`-based implementation of Algorithm 8 handles datasets, which exceed the GPU's memory capacity, by reducing the number of CI tests performed in parallel. As a result, the allocated GPU global memory for the auxiliary data structures is reduced to remain within the GPU's memory capacity limits.

4.3.2 GPU Kernel for Level 0

In the following, we describe the GPU kernel for level $l = 0$, assuming that the observational data D is discrete. In particular, we detail the variant of the GPU kernel for level $l = 0$ that uses shared memory to store the auxiliary data structures. Apart from keeping the auxiliary data structures in GPU global memory, the variant relying on GPU global memory behaves similarly to the one using shared memory. First, we present the chosen parallel execution strategy, according to Section 4.1. Further, we describe the input, the output, and the GPU kernel launch parameters. Lastly, we specify the operations performed within the GPU kernel.

Execution Strategy: Mapping of Tasks to Execution Units

In level $l = 0$, the separation set for any pair of variables (V_i, V_j) with $i, j = 1, \ldots, N$ and $i \neq j$ is empty, i.e., $S^{i,j} = \emptyset$, and only a single CI test is performed. Further, in the case of the Pearson χ^2 test [190], all observational data samples related to a pair of variables (V_i, V_j) need to be accessed. Thus, in contrast to existing GPU-accelerated adjacency searches for the Gaussian distribution model that apply the mapping (M2), e.g., see [226, 287], in the discrete

case for level $l = 0$, we use mapping (M1) (see p. 61). In this context, we define a task for parallel execution as processing r data samples of one CI test of an edge $E^{i,j}$. Following mapping (M1), each task is mapped to one GPU thread. All tasks belonging to the same CI test are grouped in the same thread block to enable local communication via shared memory while computing the auxiliary data structures of the CI test. Further, the r data samples are assigned to each task in a manner that fosters coalesced memory access of the GPU threads within the same warp [41, 173]. For example, having each GPU thread in the same warp access data samples in a stride of the warp size. Following these considerations, we implement the GPU kernel for level $l = 0$ using shared memory, assuming discrete data as outlined in Algorithm 9 (see p. 89).

Input and Output

The GPU kernel receives the vertex set $\mathbf{V}$, the observational data matrix D with dimension $N \times n$, the estimated skeleton matrix $\mathcal{C}$ with dimension $N \times N$, the separation sets matrix Sep of dimension $N \times N \times ml$, the significance level α, the number of data samples n, the vector of domain sizes dom, and the maximum domain $mdom$ as input. The GPU kernel has no explicit output, as the GPU kernel stores its results directly in the data structures $\mathcal{C}$ and Sep. The level $l = 0$ is, per definition, order-independent. Therefore, no copy of the estimated skeleton matrix $\hat{\mathcal{C}}$ is required.

GPU Kernel Launch Parameters

The GPU kernel is launched with $N \times N$ thread blocks and δ threads per block. The value of the parameter δ must take GPU-specific hardware characteristics into account. Therefore, we suggest setting δ according to the GPU's warp size ws, where ws is a positive integer that commonly equals 32, and the maximum number of supported GPU threads per thread block $T_{TB_{max}}$, as follows:

$$\begin{aligned} ws \leq \delta \leq T_{TB_{max}} \text{ and } \delta \bmod ws \equiv 0 \quad & \text{if } ws < n \\ \delta = ws \quad & \text{otherwise.} \end{aligned} \tag{4.26}$$

Following (4.26), ensures that the number of supported GPU threads per thread block is not exceeded. At the same time, the number of GPU threads is always set as a multiple of the GPU's warp size to fill the entire warps, if possible. Note that if the number of data samples n is not a multiple of the warp size ws, for example, in the case that $ws > n$, some GPU threads in one warp perform no operations. The parameter δ influences the number of data samples processed in parallel. Thus, large values of δ, e.g., $u = T_{TB_{max}}$, provide the most parallelization while using shared memory for local communication. However, the local communication, e.g., via atomic operations, becomes a potential performance bottleneck when choosing large values of δ. Hence, we suggest choosing smaller values of δ that retain a certain degree of parallelism, e.g., $\delta = \{64, 128\}$.

Description of the Operations Within the GPU Kernel

At the start of Algorithm 9 (see p. 89), each GPU thread sets its indices row and col that refer to the position of the edge $E^{row,col}$ to process within the estimated skeleton matrix $\mathcal{C}$. The indices are based on the thread block dimensions bx and

Algorithm 9 GPU kernel for level $l = 0$ for discrete data using shared memory
Input: Vertex set $\mathbf{V}$, observational data D, estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep, significance level α, number of data samples n, vector of domain sizes dom, maximum domain $mdom$
# of blocks: $N \times N$
# of threads per block: δ
Shared memory: $(mdom^2 + 2 \times mdom) \times \text{SIZEOF}(int)$

1: $row \leftarrow bx, col \leftarrow by$
2: **if** $row < col$ **then**
3: Initialize $f_{v_{row}v_{col}}$ of size $mdom^2$ in shared memory and set entries to 0
4: Initialize $f_{+v_{col}}$,$f_{v_{row}+}$ of size $mdom$ in shared memory and set entries to 0
5: **for** $a \leftarrow tx; a < n; a \leftarrow a + \delta$ **do**
6: ATOMICADD$(f_{v_{row}v_{col}}[D[V_{row}][a] \times dom[V_{row}] + D[V_{col}][a]], 1)$
7: **end for**
8: SYNCTHREADS()
9: **for** $a \leftarrow tx; a < dom[V_{row}] \times dom[V_{col}]; a \leftarrow a + \delta$ **do**
10: ATOMICADD$(f_{v_{row}+}[\frac{a}{dom[V_{col}]}], f_{v_{row}v_{col}}[a])$
11: ATOMICADD$(f_{+v_{col}}[a \bmod dom[V_{col}]], f_{v_{row}v_{col}}[a])$
12: **end for**
13: SYNCTHREADS()
14: **if** $tx = 0$ **then**
15: $\chi \leftarrow 0$
16: **for** $a \leftarrow 0; a < dom[V_{row}]; a \leftarrow a + 1$ **do**
17: **for** $b \leftarrow 0; b < dom[V_{col}]; b \leftarrow b + 1$ **do**
18: $exp \leftarrow \frac{f_{v_{row}+}[a] \times f_{+v_{col}}[b]}{n}$
19: **if** $exp \neq 0$ **then**
20: $o \leftarrow f_{v_{row}v_{col}}[a \times dom[V_{col}] + b]$
21: $\chi \leftarrow \chi + \frac{(o-exp)^2}{exp}$
22: **end if**
23: **end for**
24: **end for**
25: p $\leftarrow$PCHISQ$(\chi, (dom[V_{row}] - 1) \times (dom[V_{col}] - 1))$
26: **if** p $\geq \alpha$ **then**
27: $\mathcal{C}[row][col] \leftarrow 0$
28: $\mathcal{C}[col][row] \leftarrow 0$
29: $Sep[row][col][0] \leftarrow 0$
30: **end if**
31: **end if**
32: **end if**

by, see line 1. Thus, δ GPU threads jointly process the edge $E^{row,col}$. The algorithm considers elements in the upper triangular of the estimated skeleton matrix $\mathcal{C}$ only, as the performed CI test is direction-independent.

In the following, the auxiliary data structures for the contingency table and marginals are initialized in shared memory, and all entries are set to 0 (see lines 3–4 of Algorithm 9). Note that this step is skipped for the variant of the GPU kernel that uses a fixed-size GPU global memory block for the auxiliary data structures that is reserved before GPU kernel launch.

Afterward, the contingency table is computed jointly by the δ GPU threads within the same thread block. Therefore, the δ GPU threads iterate the n data samples in a stride of size δ (see lines 5–7). Each GPU thread increments one entry in the contingency table in each iteration using an ATOMICADD(...). The entry's location in the contingency table is computed based on the considered data sample's values concerning variables V_{row}, V_{col}.

The GPU threads within the same thread block are synchronized, before these GPU threads jointly calculate the marginals based on the contingency table (see lines 8–12). Again, the GPU threads apply an ATOMICADD(...) to avoid any conflicts while writing to the data structures of the marginals.

Upon completion of the marginal calculation, the GPU threads within the same thread block are synchronized and one main GPU thread, i.e., $tx = 0$, finalizes the computations of the CI test (see lines 14–31). Therefore, the main GPU thread computes the statistics derived from the marginals over the observed frequencies of the corresponding contingency table entry. Further, the main GPU thread computes the p-value using the function PCHISQ(...) [264] and compares the p-value with the significance level α. If $\mathrm{p} \geq \alpha$, the corresponding edge $E^{row,col}$ is deleted in $\mathcal{C}$, i.e., the fields $\mathcal{C}[row][col]$ and $\mathcal{C}[col][row]$ in the estimated skeleton matrix are set to 0. As a final step, the separation set is marked at the corresponding index in the upper triangular matrix (see line 29). The GPU kernel terminates once all GPU threads have finished.

4.3.3 GPU Kernel for Levels 1 and Higher

In the following, we describe the GPU kernel for any level $l \geq 1$, assuming that the observational data D is discrete. We first describe the applied parallel execution strategy according to Section 4.1. Next, we mention the input, the output, and the GPU kernel launch parameters. Finally, we elucidate the operations performed by each GPU thread during GPU kernel execution.

Execution Strategy: Mapping of Tasks to Execution Units

For the GPU kernel for any level $l \geq 1$, the execution strategy must handle a variable number of CI tests per edge $E^{i,j}$ with $i, j = 1, \ldots, N$ and $i \neq j$. The number of performed CI tests for any edge $E^{i,j}$ varies between 0 and $|\mathbf{S}^{i,j,l}|$, depending on the structure of the CGM's underlying true DAG and the adjacency set $adj(\mathcal{C}^l, V_i)$. Consequently, we require a mapping of tasks to the execution units that addresses communication while processing multiple CI tests per edge $E^{i,j}$. Further, each CI test for the discrete data accesses all n data samples of D for $V_i, V_j, S^{i,j}$. Therefore, the task mapping must reflect efficient memory access.

These two requirements are best addressed using mapping (M3). Hence, each task processes r data samples from u CI tests related to the same edge $E^{i,j}$. Thus, the total number of tasks t_{tot} of an edge $E^{i,j}$ is computed as follows:

$$t_{tot} = \left\lceil \frac{n}{r} \right\rceil \times \left\lceil \frac{|\mathbf{S}^{i,j,l}|}{u} \right\rceil. \tag{4.27}$$

According to mapping (M3), the total number of tasks t_{tot} is restricted by the maximum number of GPU threads per thread block $T_{TB_{max}}$ to ensure local communication of the CI test results for early termination. Further, the value of r is chosen to fulfill the following:

$$ws \leq \left\lceil \frac{n}{r} \right\rceil \leq T_{TB_{max}} \text{ and } \left\lceil \frac{n}{r} \right\rceil \bmod ws \equiv 0, \tag{4.28}$$

to facilitate efficient memory access by coalescing. According to (4.27) and (4.28), the choice of r and u is a balance between processing more data samples or CI tests in one task, respectively, in parallel by multiple tasks scheduled to GPU threads within the same thread block. Therefore, we suggest setting $r = \lceil \frac{n}{ws} \rceil$ or $\left\lceil \frac{n}{2 \times ws} \right\rceil$, which allows processing multiple data samples in the same task. At the same time, we suggest setting $u = \left\lceil \frac{|\mathbf{S}^{i,j,l}|}{2} \right\rceil$ or $\left\lceil \frac{|\mathbf{S}^{i,j,l}|}{4} \right\rceil$ to map multiple CI tests to the same task. Thus, a balance between processing data samples and CI tests in parallel in the same task is achieved. Based on these considerations, we outline the GPU kernel for levels $l \geq 1$ under the assumption that the observational data D is discrete in Algorithm 10 (see p. 92).

Input and Output

The GPU kernel gets the same input parameters as the GPU kernel for level $l = 0$ (see Section 4.3.2) with the addition of the compacted estimated skeleton matrix $\hat{\mathcal{C}}$, the fixed-size GPU global memory block m_{aux} for the auxiliary data structures, and the current level l. The compacted estimated skeleton matrix $\hat{\mathcal{C}}$ enables more efficient data access and ensures the order independence of PC-stable [35]. The GPU kernel manipulates the data structures $\mathcal{C}$ and Sep, which contain the results of the computation required in the subsequent levels.

GPU Kernel Launch Parameters

The GPU kernel is launched with $N \times \frac{N}{\beta}$ thread blocks and $\delta \times \gamma \times \beta$ threads per block. The value of δ is set to $\delta = \lceil \frac{n}{r} \rceil$ with a choice for the parameter r fulfilling the requirements defined in (4.28). Following (4.27), the value of γ is set to $\gamma = \left\lceil \frac{|\mathbf{S}^{i,j,l}|}{u} \right\rceil$ with an appropriate choice for the parameter u. Note that the parameter β represents an extension beyond the applied execution strategy and mapping (M3), which allows for additional parallelism by processing multiple edges within the same thread block [81]. In this work, we generally assume $\beta = 1$. Further, the GPU kernel reserves shared memory of size $\delta \times \gamma \times \beta \times \text{SIZEOF}(float)$ to store and share the local statistics computed by each GPU thread. To keep the amount of shared memory used by each thread block small, to allow for multiple thread blocks per SM, we suggest choosing smaller values for the parameters δ, γ, β, e.g., by setting $r = \lceil \frac{n}{ws} \rceil$ and $u = \left\lceil \frac{|\mathbf{S}^{i,j,l}|}{2} \right\rceil$.

Description of the Operations within the GPU Kernel

Upon GPU kernel launch, each GPU thread sets the index row corresponding to the block dimension bx. Next, the indices of variables in the adjacency set $adj(\mathcal{C}^l, V_{row})$ are loaded from the compacted estimated skeleton matrix $\hat{\mathcal{C}}$ into shared memory by GPU threads from the same thread block (see line 2 of Algorithm 10, p. 92). Using shared memory allows for fast access to adjacent variables, e.g., during the computation of the separation sets in lines 6–7.

In lines five and following, γ GPU threads jointly iterate all possible separation sets for the edge $E^{row,col}$, i.e., the pair of variables (V_{row}, V_{col}), in a stride of γ. Thus, $\delta \times \gamma$ GPU threads from the same thread block process γ CI tests

Algorithm 10 GPU kernel for arbitrary levels $l \geq 1$ for discrete data.
Input: Vertex set $\mathbf{V}$, observational data D, estimated skeleton matrix $\mathcal{C}$, compacted estimated skeleton matrix $\hat{\mathcal{C}}$, separation sets matrix Sep, significance level α, number of data samples n, level l, vector of domain sizes dom, maximum domain $mdom$, fixed-size GPU global memory block m_{aux} for auxiliary data structures
# of blocks: $N \times \frac{N}{\beta}$
# of threads per block: $\delta \times \gamma \times \beta$
Shared memory: $\delta \times \gamma \times \beta \times \text{SIZEOF}(float)$

1: $row \leftarrow bx$
2: Let $a_s(V_{row}) \leftarrow adj(\hat{\mathcal{C}}, V_{row})$ in shared memory
3: SYNCTHREADS()
4: $col \leftarrow a_s(V_{row})[by \times \beta + tz]$
5: **for** $c \leftarrow ty; c < \binom{|a_s(V_{row})|-1}{l}; c \leftarrow c + \gamma$ **do**
6: $Pos_{1..l} \leftarrow$PARALLEL_COMB$(|a_s(V_{row})| - 1, l, c)$
7: $S^{row,col} \leftarrow a_s(V_{row})[Pos_{1..l}]$
8: **if** $tx = 0$ **then**
9: In m_{aux} set $mdom^{l+2}$ entries of $f_{v_{row}v_{col}s^{row,col}}$, $mdom^{l+1}$ entries of $f_{+v_{col}s^{row,col}}$ and $f_{v_{row}+s^{row,col}}$, and $mdom^{l}$ entries of $f_{++s^{row,col}}$ to 0
10: **end if**
11: SYNCTHREADS()
12: COMPUTECONTINGENCYTABLE$(f_{v_{row}v_{col}s^{row,col}})$
13: SYNCTHREADS()
14: COMPUTEMARGINALS$(f_{+v_{col}s^{row,col}}, f_{v_{row}+s^{row,col}}, f_{++s^{row,col}})$
15: SYNCTHREADS()
16: COMPUTELOCALSTATISTIC(χ)
17: SYNCTHREADS()
18: **if** $tx = 0$ **then**
19: $\chi_0 \leftarrow 0$
20: **for** $g \leftarrow tz \times \gamma \times \delta + ty; g < tz \times \gamma \times \delta + ty \times \delta; g \leftarrow g + 1$ **do**
21: $\chi_0 \leftarrow \chi_0 + \chi[g]$
22: **end for**
23: p $\leftarrow$PCHISQ$(\chi_0, (dom[V_{row}] - 1) \times (dom[V_{col}] - 1) \times \prod_{V_z \in S^{row,col}} dom[V_z])$
24: **if** p $\geq \alpha$ **then**
25: **if** ACQUIREMUTEX(tz) **then**
26: $\mathcal{C}[row][col] \leftarrow 0$
27: $\mathcal{C}[col][row] \leftarrow 0$
28: **for** $d \leftarrow 0; d < l; d \leftarrow d + 1$ **do**
29: $Sep[row][col][d] \leftarrow S^{row,col}[d]$
30: **end for**
31: **end if**
32: **end if**
33: **end if**
34: SYNCTHREADS()
35: **if** $\mathcal{C}[row][col] = 0$ **then**
36: *break* ▷ Early return
37: **end if**
38: **end for**

in parallel. Within this loop, each GPU thread determines the separation set corresponding to the current iteration c (see lines 6–7). Next, for each of the

γ CI tests, one main GPU thread, i.e., $tx = 0$, sets all entries within the corresponding auxiliary data structure $f_{v_{row}v_{col}s^{row,col}}$,$f_{+v_{col}s^{row,col}}$, $f_{v_{row}+s^{row,col}}$ and $f_{++s^{row,col}}$ to 0.

Afterward, the δ GPU threads jointly compute the contingency table based on the observational data D in procedure computeContingencyTable(...) (see line 12). Once the contingency table is calculated and the GPU threads are synchronized, the δ GPU threads jointly compute the marginals over the contingency table in the procedure computeMarginals(...) (see line 14). Finally, the δ GPU threads determine a part of the statistics in the procedure computeLocalStatistic(...) (see line 16 of Algorithm 10, p. 92). Details of these three procedures are shown separately in Algorithm 11 (see p. 94).

In particular, in the procedure computeContingencyTable(...) (see lines 1–10 of Algorithm 11), the γ GPU threads iterate the n data samples of the observational data D. In each iteration, each GPU thread increments the location in the contingency table according to the values stored at the data sample's entries in D related to $S^{row,col}$, V_{row}, and V_{col}. Using an atomic operation for the increment avoids any write conflicts of multiple GPU threads.

In the procedure computeMarginals(...) (see lines 11–17 of Algorithm 11), the γ GPU threads jointly compute the marginals $f_{+v_{col}s^{row,col}}$, $f_{v_{row}+s^{row,col}}$ and $f_{++s^{row,col}}$. Therefore, the GPU threads iterate over the contingency table $f_{v_{row}v_{col}s^{row,col}}$ in a stride of size γ. Within each iteration, each GPU thread increments the marginals with the value from the current position of the contingency table. Again, atomic operations for the increment avoid any write conflicts if the γ GPU threads.

In the procedure computeLocalStatistic(...) (see lines 18–33 of Algorithm 11), each of the γ GPU threads computes one part of the statistics. At first, each GPU thread sets the value at its position in vector χ in shared memory to 0. Next, each GPU thread processes a share of the contingency table and calculates the expected frequency (see line 24) to compute the statistics (see line 27). Afterward, each GPU thread stores its computed statistics in the vector χ at the position corresponding to the GPU's thread id.

After the δ GPU threads have finished the computations of the procedure computeLocalStatistic(...), one main GPU thread for each of the γ CI tests, i.e., $tx = 0$, performs the final operations for the CI test. The main GPU thread computes the sum over the local statistics in lines 19–22 of Algorithm 10 (see p. 92) and stores the sum in χ_0. Next, the procedure pchisq(...) is called to calculate the p-value based upon χ_0 and the degrees of freedom df. Note df is calculated according to (2.10). If the p-value is larger than or equal to α, the entries in the estimated skeleton C corresponding to the edge $E^{row,col}$ are set to 0, and the separation set is stored in Sep (see lines 24–32 of Algorithm 10). Note that a lock is acquired in line 25 to avoid any write conflict between the γ main threads. If, the edge is marked as independent, the loop over the CI tests is terminated early (see lines 35 – 37 of Algorithm 10). Once all launched GPU threads have finished, the GPU kernel call terminates.

Additional Notes

The described GPU kernel launch parameters and operations refer to the standard case for the GPU kernel for arbitrary levels $l \geq 1$. Yet, if data is high-dimensional, the required memory for the auxiliary data structures can exceed

Algorithm 11 Procedures called within Algorithm 10.
The procedures assume access to all variables defined in Algorithm 10.
Input to functions refers to the data computed within the function added for better readability.

1: **procedure** COMPUTECONTINGENCYTABLE($f_{v_{row}v_{col}s^{row,col}}$)
2: **for** $g \leftarrow tx; g < n; g \leftarrow g + \delta$ **do**
3: $sum_S \leftarrow 0, lev_S \leftarrow 1$
4: **for all** $V_z \in S^{row,col}$ **do**
5: $sum_S \leftarrow sum_S + (D[V_z][g] \times lev_S)$
6: $lev_S \leftarrow lev_S \times dom[V_z]$
7: **end for**
8: ATOMICADD($f_{v_{row}v_{col}s^{row,col}}[sum_S \times dom[V_{row}] \times dom[V_{col}] + D[V_{row}][g] \times dom[V_{col}] + D[V_{col}][g]], 1$)
9: **end for**
10: **end procedure**

11: **procedure** COMPUTEMARGINALS($f_{+v_{col}s^{row,col}}$, $f_{v_{row}+s^{row,col}}$, $f_{++s^{row,col}}$)
12: **for** $g \leftarrow tx; g < dom[V_{row}] \times dom[V_{col}] \times \prod_{V_z \in S^{row,col}} dom[V_z]; g \leftarrow g + \delta$ **do**
13: ATOMICADD($f_{+v_{col}s^{row,col}}[\frac{g}{dom[V_{col}]}], f_{v_{row}v_{col}s^{row,col}}[g]$)
14: ATOMICADD($f_{v_{row}+s^{row,col}}[\frac{g}{dom[V_{row}] \times dom[V_{col}]} \times dom[V_{col}] + g \bmod dom[V_{col}]], f_{v_{row}v_{col}s^{row,col}}[g]$)
15: ATOMICADD($f_{++s^{row,col}}[\frac{g}{dom[V_{row}] \times dom[V_{col}]}], f_{v_{row}v_{col}s^{row,col}}[g]$)
16: **end for**
17: **end procedure**

18: **procedure** COMPUTELOCALSTATISTIC(χ)
19: $\chi[tz \times \gamma \times \delta + ty \times \delta + tx] = 0$
20: **for** $g \leftarrow tx; g < \prod_{V_z \in S^{row,col}} dom[V_z]; g \leftarrow g + \delta$ **do**
21: **for** $a \leftarrow 0; a < dom[V_{row}]; a \leftarrow a + 1$ **do**
22: **for** $b \leftarrow 0; b < dom[V_{col}]; b \leftarrow b + 1$ **do**
23: **if** $f_{v_{row}v_{col}s^{row,col}}[g] \neq 0$ **then**
24: $exp \leftarrow \frac{f_{+v_{col}s^{row,col}}[g \times dom[V_{row}]+a] \times f_{v_{row}+s^{row,col}}[g \times dom[V_{col}]+b]}{f_{++s^{row,col}}[u]}$
25: **if** $exp \neq 0$ **then**
26: $o \leftarrow f_{v_{row}v_{col}s^{row,col}}[(g \times dom[V_{row}] \times dom[V_{col}]) + (a \times dom[V_{col}]) + b]$
27: $\chi[tz \times \gamma \times \delta + ty \times \delta + tx] = \chi[tz \times \gamma \times \delta + ty \times \delta + tx] + \frac{(o-exp)^2}{exp}$
28: **end if**
29: **end if**
30: **end for**
31: **end for**
32: **end for**
33: **end procedure**

the reserved fixed-size GPU global memory block. In this case, the procedure ADJUSTKERNELLAUNCHDIMENSIONS(...) shown in Algorithm 8 determines a factor F that limits the parallel execution of the GPU kernel to constrain the memory demand. As a result, only $\frac{N}{F} \times \frac{N}{\beta}$ thread blocks are launched, and each thread block processes F edges sequentially. Hence, Algorithm 10 needs an additional loop over F elements, which covers all illustrated steps and sets the index row (see line 1) according to the current iteration.

4.4 GPU-Accelerated Adjacency Search in PC-Stable with an Information-Theoretic GPU-Based CI Test

This section details our GPU-based CI test, $\texttt{GPU}_{\texttt{CMIknn}}$, and an algorithm for a GPU-accelerated adjacency search in PC-stable that employs an extended version of $\texttt{GPU}_{\texttt{CMIknn}}$. The GPU-based CI test and the GPU-accelerated adjacency search are designed for information-theoretic CI tests that build upon Conditional Mutual Information (CMI) estimation using k-Nearest Neighbor (k-NN)-based approaches, e.g., [97, 100, 214].

In the following, we describe our two algorithms under the assumption of mixed discrete-continuous data with non-linear relationships (see Section 2.4.3) and the application of the CI test by Huegle et al. [100]. In our original publication of $\texttt{GPU}_{\texttt{CMIknn}}$ [83], we outlined the algorithms for continuous data with non-linear relationships, following the CI test developed by Runge [214].

First, we discuss options to implement the k-NN searches required for the CI test given GPU hardware constraints and the demand to conduct many different k-NN searches in parallel (see Section 4.4.1). In particular, we consider a K-D tree-based k-NN search and a brute-force-based k-NN search.

Based upon these considerations, we present $\texttt{GPU}_{\texttt{CMIknn}}$, our GPU-based information-theoretic CI test, in Section 4.4.2. In this part, we first focus on the overall idea of the algorithm and describe data transfer between host and GPU memory and the GPU kernel orchestration. Further, we detail the GPU kernels for computing the local permutations and the CMI estimates. Lastly, we mention the simplified implementation for the case of unconditioned independence tests, i.e., calculating the Mutual Information (MI) given an empty separation set $S^{i,j} = \emptyset$.

$\texttt{GPU}_{\texttt{CMIknn}}$ is a stand-alone CI test and can be plugged into the adjacency search of the standard PC-stable algorithm. Throughout this thesis, we denote this approach as $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$. Yet, to fully utilize the compute capabilities of the underlying GPU hardware, we propose a tailored algorithm for a GPU-accelerated adjacency search in PC-stable that employs an extended version of $\texttt{GPU}_{\texttt{CMIknn}}$. We call this approach $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ and detail the algorithm in Section 4.4.3. $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ applies two optimizations. First, the algorithm computes multiple CI tests in parallel to fully utilize the GPU parallel compute units. Second, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ partially reuses the computed local permutations when performing multiple CI tests. *Parts of this section have been published in a research paper [83].*

4.4.1 Approaches to Parallel k-NN Searches on GPU

In the context of information-theoretic CI tests, k-NN searches are used for density estimation [214]. The density $p(D_m)$ with $m = 1, \ldots, n$ for any data sample D_m is estimated by constructing a sphere around D_m that contains k nearest data samples [14]. Within CI tests, the k-NN search processes the entire set of n data samples, which results in costly computation for large datasets [14]. Two common approaches exist to realize k-NN searches, namely brute-force searches and tree-based searches [13, 183]. For brute-force and tree-based searches, efficient implementations exist on CPU [191] and GPU [72, 73, 204].

Brute-Force Search

The brute-force search is a naive solution that compares all data samples D_m with each other to compute the k nearest neighbors. While the brute-force search has a high computational complexity of $O(n^2)$, it requires no additional memory, except for storing the resulting k nearest neighbors for each data sample D_m.

Tree-Based Search

During a search, tree-based searches address the high computational demand by querying efficient tree-based structures, such as a k-d tree [13, 64] or a balltree [183]. Applying the k-d tree reduces the computational complexity of the k-NN search over all data samples n to $O(n \times log(n))$ on average. Yet, the tree-based search structure must be constructed first, which adds a one-time computational effort of $O(n \times log(n))$. Furthermore, the tree-based search structure either requires additional space in memory of size $O(n)$, or the tree-based search must manipulate the order of the n data samples in place. Note that the balltree is designed for high-dimensional data samples. In this setting, a k-d tree becomes inefficient due to the curse of dimensionality [64, 183]. In the context of CSL, settings suited for a balltree, e.g., due to extensive separation sets, are uncommon due to the low statistical power of CI tests in real-world settings with low sample sizes.

GPU-Based Brute-Force-Based k-NN Search

Garcia et al. [72] propose two GPU-accelerated implementations of the brute-force search that outperform CPU-based algorithms by up to two orders of magnitude. Their first implementation uses two separate `CUDA` kernels. The first `CUDA` kernel computes a distance matrix between all data samples. All distance computations are independent of each other. Therefore, the GPU threads compute one distance each in parallel. The second `CUDA` kernel sorts the distances. Within the sort kernel, the distances of a single data sample are sorted in parallel by one GPU thread each. The second implementation proposed by Garcia et al. [72] reformulates the brute-force k-NN search as matrix additions and multiplications to leverage highly optimized matrix operations available in the cuBLAS library [176]. A more recent GPU-accelerated implementation of the brute-force k-NN search is provided in RAPIDS [204], based on the cuML library [205]. The implementation follows the general idea of Garcia et al. [72]. Both works argue that brute-force search is well suited for execution on GPU.

GPU-Based K-D Tree-Based k-NN Search

Performing k-NN searches using standard k-d tree implementations on the GPU may result in poor performance due to branching and memory access inapt for GPU hardware. The buffer k-d tree addresses these shortcomings [73]. The buffer k-d tree consists of one top tree, leaf structures, and buffers for each leaf from the top tree. The top tree has a small height of, e.g., $h = 8$, and points to the leaf structures. The authors of the buffer k-d tree argue that the small height alleviates the high cost for execution on GPU during tree traversal [73]. When querying the buffer k-d tree, the buffers are filled with query indices that get processed once a threshold is reached. During buffer processing, the k-nearest

neighbors are determined for each query index in each buffer in parallel by one GPU thread each. The GPU threads perform a brute-force search within the leaf structure corresponding to the query index's buffer. Note that the initial construction of the k-d tree and orchestration tasks are performed on the CPU. The buffer k-d tree provides an efficient GPU-based approach to the k-NN search for vast amounts of queries [73].

Implications for a GPU-Accelerated Adjacency Search in PC-Stable With Information-Theoretic CI Tests

The k-NN search is one building block of the information-theoretic CI test (see Section 2.4.3). The number of k-NN searches for a CI test is computed as follows:

$$\#knn_{searches} = 2 + perm, \tag{4.29}$$

where $perm$ is the number of permutations of V_i. Each k-NN search is performed for all n data samples from the variables V_i, V_j and $S^{i,j}$ with $i, j = 1, \ldots, N$ and $i \neq j$, respectively, the $perm$ random permutations of V_i associated with a CI test. Within the adjacency search of PC-stable, thousands of CI tests are conducted. Remember that the number of CI tests increases polynomial with the number of variables N under the assumption of sparse graphs [109]. Thus, the efficient execution of the k-NN searches is crucial for fast runtimes. The k-NN search itself [72, 73] and the execution of multiple k-NN searches simultaneously provide ample opportunity for parallel execution on a GPU.

While the k-d tree-based k-NN search provides a better computational complexity over the brute-force approach, it introduces the following drawbacks. First, the number of queries to the k-d tree is limited to the number of data samples n. Thus, we cannot assume that the k-d tree construction time is amortized. Second, the k-d tree requires additional space to store at least $n \times d$ elements, where d is dimensionality, corresponding to the number of variables considered in a given CI test. The memory demand quickly exceeds the capacity of thread-local register space or shared memory. Thus, the k-d tree data structure must be stored in GPU global memory. At the same time, as the GPU's memory capacity is limited, the memory demand for the k-d tree data structures restricts the number of k-NN searches; consequently, CI tests, executed in parallel. Lastly, the construction of the k-d tree and the search within the k-d tree introduce branching of GPU threads as these operations are not well-suited for the SIMT execution model.

Considering these drawbacks, we apply the brute-force search to compute the k-nearest neighbors. Despite the higher computational complexity, we see the potential for faster runtimes due to the higher degree of parallelism for multiple reasons. First, the brute-force search is well-suited for the SIMT execution model of GPUs. Thus, the brute-force search avoids branching and ensures coalesced memory access [41, 173]. Second, the brute-force search requires no additional data structures in GPU global memory and avoids restricting the parallel execution. Furthermore, the brute-force search allows the application of a pipeline approach [68] while performing multiple CI tests in parallel. The pipeline approach aims to fuse operations applied to the same data to avoid storing large-sized intermediate results. Thus, for the brute-force search, we aim to compute the k-nearest neighbors for each data sample in a single pass. Hence, we only store the k-nearest neighbors, which may fit in GPU thread-local memory.

Algorithm 12 Outline of $\texttt{GPU}_{\texttt{CMIknn}}$
Input: Observational data D, variables V_i, V_j, separation set $S^{i,j}$, number of permutations $perm$, k-nearest neighbors within permutation k_{perm}, k-nearest neighbors within CMI estimation k_{CMI}, number of data samples n
Output: p-value p

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

4.4.2 $\texttt{GPU}_{\texttt{CMIknn}}$: A GPU-Accelerated Information-Theoretic CI Test

This section presents, $\texttt{GPU}_{\texttt{CMIknn}}$, our GPU-accelerated information-theoretic CI test designed to handle mixed discrete-continuous data and data with non-linear relationships (see Section 2.4.3). We first outline the algorithm of our proposed CI test with a focus on the CPU-based tasks, such as GPU kernel orchestration, data transfer, and final computations. Next, we present the GPU kernel for computing the local permutation and the GPU kernel for estimating the CMI. At the end of this section, we sketch the procedure for independence testing if the separation set is empty.

Outline of $\texttt{GPU}_{\texttt{CMIknn}}$

In Algorithm 12, we outline the procedure of our proposed CI test $\texttt{GPU}_{\texttt{CMIknn}}$. The procedure presented in this thesis is a GPU-based version of our CI test [100]. The same GPU acceleration concepts can be applied to other CMI estimation-based CI tests that build upon k-NN searches, e.g., see [97, 214]. In these cases, adaptations are needed for the operations in lines 6–13 and in the GPU kernel estimate$_{\text{CMIknn}}$, e.g., see our corresponding publication [83] using `CMIknn` [214].

Input and Output

$\texttt{GPU}_{\texttt{CMIknn}}$ takes as input the observational data D and indices of the variables V_i,V_j, and $S^{i,j}$, with $i, j = 1, \ldots, N$ and $i \neq j$, that point to the corresponding data samples within D. Further, $\texttt{GPU}_{\texttt{CMIknn}}$ requires the 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, and n, the number of data samples. Upon completion, the procedure returns the computed p-value.

Description of the Procedure of $\texttt{GPU}_{\texttt{CMIknn}}$

The algorithm starts with data preparation for 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 GPU global memory for the permutations of V_i, denoted $\hat{V}_i$, the intermediate CMI values, denoted $partial_{cmi}$, and an auxiliary data structure, denoted *used*, (see lines 1–2 of Algorithm 12). The auxiliary data structure *used* stores flags for each data sample that help avoid the reuse of that data sample during local permutation computation.

Next, the localPermutation GPU kernel is launched on the GPU to compute $\hat{V}_i$. We split the computation of the local permutations $\hat{V}_i$ and the CMI estimation into separate GPU kernels for two reasons. First and foremost, the CMI estimation builds upon the result of the local permutations $\hat{V}_i$, requiring global synchronization. Using separate synchronous GPU kernel launches realizes the global synchronization. Second, separate GPU kernel launches allow GPU kernel launch parameters appropriate to each task.

After the localPermutation GPU kernel is completed, the $\text{estimate}_{\text{CMIknn}}$ GPU kernel is launched, which calculates $perm + 1$ intermediate values for the CMI stored in $partial_{cmi}$. The $partial_{cmi}$ values are transferred from the GPU (see line 5) to finalize the CMI computation. Therefore, calculations align with the operations of the $\text{estimate}_{\text{CMIknn}}$ GPU kernel. In our case, we follow the approach of Huegle et al. [100], which builds upon a CMI estimator for mixed discrete-continuous data [161]. Thus, the calculation builds upon the following equation:

$$cmi = \frac{partial_{cmi}}{n}, \tag{4.30}$$

where $partial_{cmi}$ is computed as follows:

$$partial_{cmi} = \sum_{a=1}^{n} \digamma(C^a_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}) - \digamma(C^a_{\mathbb{V}_i\mathbb{S}^{i,j}}) - \digamma(C^a_{\mathbb{V}_j\mathbb{S}^{i,j}}) + \digamma(C^a_{\mathbb{S}^{i,j}}). \tag{4.31}$$

The $partial_{cmi}$ is based upon the counts C of points within the space $\mathbb{V}_i \otimes \mathbb{V}_j \otimes \mathbb{S}^{i,j}$ and the subspaces $\mathbb{V}_i \otimes \mathbb{S}^{i,j}$, $\mathbb{V}_j \otimes \mathbb{S}^{i,j}$ and $\mathbb{S}^{i,j}$, with $i, j = 1, \ldots, N$ and $i \neq j$, that are within the distance of the k-nearest neighbor taken from the joint space $\mathbb{V}_i \otimes \mathbb{V}_j \otimes \mathbb{S}^{i,j}$ (cf. Equations 20,21 in Mesner and Shalizi [161]).

In line 6 of Algorithm 12, the $base_{CMI}$ is computed based on the non-permuted case V_i,V_j,$S^{i,j}$. For each of the $perm$ permutations, the algorithm computes the CMI value. Furthermore, the algorithm checks if that CMI value is larger than or equal to the $base_{CMI}$ and accordingly increments a counter c.

Finally, the algorithm computes the p-value as the sum of the indicator function, i.e., the counter c, over the number of permutations (see line 13) and returns the result.

Local Permutation GPU Kernel

In the following, we describe the GPU kernel for computing the local permutations. The GPU kernel computes $perm$ local permutations of n data samples. First, we mention the input and output of the GPU kernel. Next, we define tasks for parallel execution and explain the GPU kernel launch parameters. Lastly, we describe the operations performed within the local permutation GPU kernel.

Algorithm 13 Local permutation GPU kernel in $\texttt{GPU}_{\texttt{CMIknn}}$
Input: Data samples $D[S^{i,j}]$ and $D[V_i]$, data structure for permutations $\hat{V}_i$, auxiliary data structure *used*, number of data samples n, number of permutations *perm*, k-nearest neighbors within permutation k_{perm}
# of blocks: $\left\lceil \frac{n}{\beta} \right\rceil$
# of threads per block: β
Shared memory: $\beta \times \text{DIMENSION}(S^{i,j}) \times \text{SIZEOF}(float)$

1: Initialize $sDist[k_{perm}]$ with BIG_FLOAT, $sPos[k_{perm}]$ with 0 in thread-local memory
2: Initialize S_{local} of size DIMENSION($S^{i,j}$) in thread-local memory
3: Set $S_{local} \leftarrow D[S^{i,j}][bx \times \beta + tx]$
4: **for all** $a \in \{0, \ldots, \left\lceil \frac{n}{\beta} \right\rceil - 1\}$ **do**
5: $S_{shared}[tx] \leftarrow D[S^{i,j}][a \times \beta + tx]$ 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: $dist \leftarrow$DISTMETRIC($S_{local}, S_{shared}[b]$)
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: $pos_{shuffled} \leftarrow$ CURAND() $\bmod(d + 1)$
22: SWAP($sPos[d], sPos[pos_{shuffled}]$)
23: **end for**
24: $u \leftarrow 0$
25: **while** ATOMICCAS($used[c \times n + sPos[u]], 0, 1) \neq 0$ and $u < k_{perm} - 1$ **do**
26: $u \leftarrow u + 1$
27: **end while**
28: $\hat{V}_i[c \times n + bx \times \beta + tx] \leftarrow D[V_i][sPos[u]]$
29: **end for**

Input and Output

The LOCALPERMUTATION GPU kernel (see Algorithm 13) takes the observational data D for the variable V_i, and the separation set $S^{i,j}$, the data structure to store the permutations $\hat{V}_i$ and the auxiliary data structure *used* as input. Additionally, the LOCALPERMUTATION GPU kernel receives the parameters for the number of data samples n, the number of permutations *perm*, and the number of k-nearest neighbors during local permutation k_{perm}. The GPU kernel stores V_i's computed *perm* local permutations in the data structure $\hat{V}_i$. The data structure remains on GPU for further processing.

Defining Tasks for Parallel Execution and GPU Kernel Launch Parameters

Based on Foster's methodology [61], we define a parallel task as the processing of one individual data sample. The task agglomerates the operations performed for the *perm* permutations based on one data sample. Each task is mapped to a unique GPU thread. The GPU threads are grouped in thread blocks at a warp-size granularity. Thus, the GPU kernel is launched with β GPU threads per thread block and $\left\lceil \frac{n}{\beta} \right\rceil$ thread blocks. The parameter β is chosen as a multiple of the warp size and does not exceed the maximum number of GPU threads per thread block, according to the GPU hardware characteristics. We encourage choosing a small value for β, i.e., $\beta = 32$, to allow for sufficient shared memory within one thread block. Further, the GPU kernel reserves shared memory of size $\beta \times \text{DIMENSION}(S^{i,j}) \times \text{SIZEOF}(float)$ bytes 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 bytes.

Description of the Operations Within the Local Permutation Kernel

Once the GPU kernel is launched, each GPU thread processes lines 1–29 of Algorithm 13 (see p. 100). Each GPU thread is responsible for processing one of the n data samples and computing the corresponding local permutations.

At first, two arrays of size k_{perm} named $sDist$ and $sPos$ are allocated and initialized in GPU thread-local memory. The array $sDist$ stores the distances and $sPos$ the positions of the k-nearest neighbors. Further, each GPU thread loads the values from $D[S^{i,j}]$ corresponding to its thread and block index into the GPU thread-local memory S_{local}.

Next, each GPU thread iterates the n data samples in a stride of size β. In each iteration, one stride of values from $D[S^{i,j}]$ is loaded into shared memory S_{shared} (see line 5 of Algorithm 13). After synchronizing the GPU threads in the same thread block, the stride is now stored in S_{shared}, and it is processed iteratively (see lines 7 – 16). If a value selected from the current stride corresponds to the data sample in S_{local}, the GPU thread skips this iteration.

Otherwise, the distance $dist$ is computed between S_{local} and $S_{shared}[b]$, i.e., the data sample from the current iteration within the current stride. A distance function DISTMETRIC(...) is applied to calculate the distance value $dist$. As a default, $\texttt{GPU}_{\texttt{CMIknn}}$ computes the Chebyshev distance [23]. The GPU thread updates the local arrays $sDist$ and $sPos$ with the computed distance $dist$. If 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 $sDist$. The GPU thread updates the array $sPos$, storing the positions of the corresponding data samples in $D[S^{i,j}]$, accordingly. 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 uses a random number generator, e.g., from NVIDIA's *cuRAND* library [178]. After initializing the random number generator, each GPU thread computes the permutations corresponding to its data sample (see lines 19–29). Thus, for each permutation, the following steps are executed.

First, the positions within *sPos* are randomly shuffled (see lines 20–23). Next, positions from *sPos* are drawn until no other GPU thread has drawn the same data sample corresponding to the position before, or it is the last position in *sPos*. GPU threads use an atomic compare and swap operation ATOMICCAS(...) on the data structure *used* to ensure that no other GPU thread drew that position before (see line 25). Finally, the GPU thread uses the selected position from *sPos* to retrieve the value from $D[V_i]$, which is set to the GPU thread's corresponding position in the current permutation (see line 28).

In contrast to CPU-based information-theoretic CI tests [100, 214], we omit the explicit creation of a random permutation to process the n data samples. Instead, we process the n data samples over $\left\lceil \frac{n}{\beta} \right\rceil$ thread blocks in parallel and rely on a non-fixed execution order of thread blocks by the GPU thread block scheduler [92]. 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 subspace $\mathbb{S}^{i,j}$.

CMI Estimation GPU Kernel

In the following, we describe the GPU kernel for Conditional Mutual Information (CMI) estimation. The GPU kernel computes partial CMI values for each of the *perm* permutations and the base CMI from the original data samples. First, we report the input and output of the GPU kernel. Next, we define tasks for parallel execution and explain the GPU kernel launch parameters. Lastly, we describe the operations performed within the CMI estimation GPU kernel.

Input and Output

The ESTIMATE$_{\text{CMIKNN}}$ GPU kernel takes the observational data D for the variables V_i,V_j and the separation set $S^{i,j}$, the data structure for the permutations $\hat{V}_i$, and the data structure to store the partial CMI values $partial_{cmi}$ as input. Additionally, the ESTIMATE$_{\text{CMIKNN}}$ GPU kernel receives the parameters for the number of data samples n and the number of k-nearest neighbors within CMI estimation k_{CMI}. The GPU kernel stores the computed partial CMI values in the list $partial_{cmi}$ in GPU global memory upon termination. The list $partial_{cmi}$ must be transferred to the host system for further processing.

Defining Tasks for Parallel Execution and GPU Kernel Launch Parameters

Following Foster's methodology [61], we define a parallel task as processing one individual data sample in the context of a local permutation $\hat{V}_i$ or the original data $D[V_i]$. Each task is mapped to a unique GPU thread. If possible, GPU threads that process tasks related to the same local permutation or the original data are grouped in the same thread block at a warp-size granularity.

The GPU kernel is launched with γ threads per thread block and $(perm + 1) \times \left\lceil \frac{n}{\gamma} \right\rceil$ thread blocks. The parameter γ is chosen as a multiple of the warp size and does not exceed the maximum number of GPU threads per thread block, according to the GPU hardware characteristics. We encourage choosing a small value for γ, i.e., $\gamma = 32$, to allow for sufficient shared memory within one thread block. Further, the GPU kernel reserves shared memory of size $2 \times \gamma \times (\text{DIMENSION}(S^{i,j}) + 2) \times sizeof(float)$ bytes for each thread block.

Algorithm 14 CMI estimation GPU kernel in $\texttt{GPU}_{\texttt{CMIknn}}$
Input: Data samples $D[V_i]$, $D[V_j]$ and $D[S^{i,j}]$, data structure for permutations $\hat{V}_i$, data structure for partial cmi values $partial_{cmi}$, number of data samples n, k-nearest neighbors within CMI estimation k_{CMI}
# of blocks: $(perm + 1) \times \left\lceil \frac{n}{\gamma} \right\rceil$
# 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: $pos \leftarrow by \times \gamma + tx$
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** $a \in \{0, \ldots, \left\lceil \frac{n}{\gamma} \right\rceil - 1\}$ **do**
6: $pos_2 \leftarrow a \times \gamma + tx$
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: $dist \leftarrow$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 $C_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}, C_{\mathbb{V}_i\mathbb{S}^{i,j}}, C_{\mathbb{V}_j\mathbb{S}^{i,j}}, C_{\mathbb{S}^{i,j}} = 0$
17: **for all** $a \in \{0, \ldots, \left\lceil \frac{n}{\gamma} \right\rceil - 1\}$ **do**
18: $pos_2 \leftarrow a \times \gamma + tx$
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: **for all** $c \in \{\{V_i, V_j, S^{i,j}\}, \{V_i, S^{i,j}\}, \{V_j, S^{i,j}\}, \{S^{i,j}\}\}$ **do**
23: $dist \leftarrow$distMetric($D_{shared}[tx][c], D_{shared}[\gamma + b][c]$)
24: updateCounter($dist, sDist[k_{CMI}], C_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}, C_{\mathbb{V}_i\mathbb{S}^{i,j}}, C_{\mathbb{V}_j\mathbb{S}^{i,j}}, C_{\mathbb{S}^{i,j}}$)
25: **end for**
26: **end for**
27: **end for**
28: atomicAdd($partial_{cmi}[bx], F(C_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}) - F(C_{\mathbb{V}_i\mathbb{S}^{i,j}}) - F(C_{\mathbb{V}_j\mathbb{S}^{i,j}}) + F(C_{\mathbb{S}^{i,j}})$))

Description of the Operations Within the CMI Estimation Kernel

Once the GPU kernel is launched, each GPU thread processes lines 1–28 of Algorithm 14. Each GPU thread processes one of the n data samples related to one local permutation and adds its local result to the appropriate partial CMI value. Note that we include the calculation of the base CMI estimate from the non-permuted values $D[V_i]$ by launching one extra thread block, i.e., $bx = 0$.

At first, the array $sDist$ of size k_{CMI} is allocated and initialized in GPU thread-local memory. The array $sDist$ stores the k_{CMI}-nearest neighbors. Each GPU thread computes an index pos, which refers to the position of the GPU thread's data sample. Next, the GPU thread loads the values of its data sample into shared memory (see function loadIntoShared(...) at line 3 of Algorithm 14). Apart from the values $D[V_j][pos]$ and $D[S^{i,j}][pos]$, the function

loadIntoShared(...) either loads original value from $D[V_i][pos]$ if $bx = 0$, or the local permutation from $\hat{V_i}[bx \times n + pos]$ into shared memory $D_{shared}[tx]$.

Each GPU thread iterates all n data samples in a stride of size γ. In each iteration, values of one stride of data samples are loaded into shared memory $D_{shared}[\gamma + tx]$ using the function loadIntoShared(...). Internally, the function loadIntoShared(...) distinguishes loading values from $D[V_i]$ or its permutations $\hat{V_i}$ depending on the thread block id bx. After synchronizing the GPU threads within the same thread block, each GPU thread iterates the current stride's data samples stored in D_{shared}. In each iteration, the GPU thread computes the distance $dist$ between its data sample $D_{shared}[tx]$ and one data sample from the current stride $D_{shared}[\gamma + b]$, where $b \in \{0, \ldots, \gamma - 1\}$. The distance function distMetric(...) is applied to compute the distance $dist$, which defaults to the Chebyshev distance [23]. If the distance $dist$ is smaller than any value stored in $sDist$, then the distance $dist$ is inserted into $sDist$ at the position that keeps $sDist$ sorted. This operation removes the last element from $sDist$. After all n elements have been processed, $sDist$ contains the distances of the k_{CMI}-nearest neighbors in sorted order, and the entry in $sDist$ at position k_{CMI} refers to the k_{CMI}-nearest neighbor.

Next, the partial CMI values are computed. This step requires counting the number of points within the distance of the k_{CMI}-nearest neighbor, i.e., within $sDist[k_{CMI}]$, for the space $\mathbb{V}_i \otimes \mathbb{V}_j \otimes \mathbb{S}^{i,j}$ and the subspaces $\mathbb{V}_i \otimes \mathbb{S}^{i,j}$, $\mathbb{V}_j \otimes \mathbb{S}^{i,j}$ and $\mathbb{S}^{i,j}$. Consequently, the algorithm initializes the counters $C_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}, C_{\mathbb{V}_i\mathbb{S}^{i,j}}, C_{\mathbb{V}_j\mathbb{S}^{i,j}}$ and $C_{\mathbb{S}^{i,j}}$ for the spaces (see line 16). Again, each GPU thread iterates all n data samples in a stride of size γ, loading data samples into shared memory using the loadIntoShared(...) function.

The GPU thread computes the distance $dist$ in each required subspace for each element within the current stride. In the updateCounter(...) function, the GPU thread checks if $dist$ is within $sDist[k_{CMI}]$ and increments the respective counter if the check evaluates *true*. Finally, after n data samples are processed, each GPU thread computes its partial result based on its corresponding data sample, i.e., $F(C_{\mathbb{V}_i\mathbb{V}_j\mathbb{S}^{i,j}}) - F(C_{\mathbb{V}_i\mathbb{S}^{i,j}}) - F(C_{\mathbb{V}_j\mathbb{S}^{i,j}}) + F(C_{\mathbb{S}^{i,j}})$. Next, the GPU thread adds the partial result to the partial CMI value $partial_{cmi}[bx]$ corresponding to one permutation or the original CMI estimate, i.e., if $bx = 0$ (see line 28). This summation requires an atomic operation to synchronize between GPU threads from multiple thread blocks. Once all GPU threads have finished, the data structure $partial_{cmi}$ contains all partial CMI values for the $perm$ permutations and the non-permuted case.

The Case of Unconditioned Independence Testing

The version of $\texttt{GPU}_{\texttt{CMIknn}}$ presented in Algorithm 12 (see p. 98) assumes testing for statistical independence based on a non-empty separation set, i.e., $S^{i,j} \neq \emptyset$. For applying $\texttt{GPU}_{\texttt{CMIknn}}$ within the context of CSL, we need to consider unconditioned independence testing, i.e., $S^{i,j} = \emptyset$, too. To cover the unconditioned case, we adjust the algorithm of $\texttt{GPU}_{\texttt{CMIknn}}$ in the following ways. First, only observational data for the variables V_i and V_j is needed. Second, the local permutation calculation uses a permutation scheme based on shuffling equal-width bins [214]. Hence, the computational demand of the local permutation calculation reduces, and parallel execution on a GPU yields slower runtimes. Therefore, the local permutation is computed on the CPU. Thus, solely the computation

of the partial CMI values, $partial_{cmi}$, is accelerated on the GPU. Third, the ESTIMATE$_{\text{CMIKNN}}$ GPU kernel shown in Algorithm 14 (see p. 103) is adapted.

The ESTIMATE$_{\text{CMIKNN}}$ GPU kernel operates on the equal-width binned local permutations for the unconditioned case. Further, the ESTIMATE$_{\text{CMIKNN}}$ kernel computes the partial CMI values based upon the counts of points $C_{\mathbb{V}_i}$, and $C_{\mathbb{V}_j}$ within the distance of the k_{CMI}-nearest neighbor for the corresponding subspaces $\mathbb{V}_i$ and $\mathbb{V}_j$. For brevity, we omit a more detailed description of this specialized case of GPU$_{\texttt{CMIknn}}$ and refer the interested reader to our reference implementation available on GitHub[2].

4.4.3 A GPU-Accelerated Adjacency Search Using GPU$_{\texttt{CMIknn}}$

We propose two options to incorporate GPU$_{\texttt{CMIknn}}$ into the adjacency search of PC-stable. The first option is straightforward. PC-stable is designed independently of any CI test [35]. Thus, GPU$_{\texttt{CMIknn}}$ can be applied without any alternations. We refer to this option as GPU$_{\texttt{CMIknn}}$`-Single`. The second option introduces a tailored GPU-accelerated adjacency search for the PC-stable algorithm, which uses an extended version of GPU$_{\texttt{CMIknn}}$. The extended version of GPU$_{\texttt{CMIknn}}$ processes multiple CI tests in parallel to improve the GPU hardware utilization and speed-up. Furthermore, we optimize the computation by reusing computed local permutations. We refer to this option as GPU$_{\texttt{CMIknn}}$`-Parallel`.

In the following, we detail GPU$_{\texttt{CMIknn}}$`-Parallel`. Therefore, we first discuss the parallel execution strategy according to the definitions from Section 4.1. Second, we describe the adjacency search GPU$_{\texttt{CMIknn}}$`-Parallel` in depth, highlighting the applied adaptations to the original adjacency search of PC-stable [35].

Parallel Execution Strategy

GPU$_{\texttt{CMIknn}}$`-Parallel` aims to perform multiple CI tests using GPU$_{\texttt{CMIknn}}$ in parallel. For the previously defined GPU-accelerated adjacency searches, we applied one of the mappings defined in Section 4.1. Yet, in the case of GPU$_{\texttt{CMIknn}}$`-Parallel`, direct application of these mappings is not possible, as the performed CI test, GPU$_{\texttt{CMIknn}}$, uses two separate GPU kernels. Additionally, keeping the intermediate results, i.e., the local permutations, for all edges in GPU global memory simultaneously quickly exceeds the GPU's memory capacity.

Therefore, we propose a different parallel execution strategy, which builds upon processing the adjacency of each variable $adj(\mathcal{C}, V_i)$ with $i = 1, \ldots, N$ separately. Processing one adjacency at a time ensures that multiple CI tests are performed in parallel while the required amount of GPU global memory is constrained. Still, the algorithm can reuse computed local permutations in the context of one adjacency.

Thus, we propose to compute the local permutations for all separation sets $S^i \in \mathbf{S}^i$ based on $adj(\mathcal{C}, V_i)$ in a single GPU kernel. For this parallel version of the `localPermutation` GPU kernel, we define a parallel task as processing one observational data sample, i.e., $D_m(V_i, S^i)$, for $m = 1, \ldots, n$. As the GPU kernel processes all separation sets $\mathbf{S}^i$ in parallel, the GPU kernel handles $n \times |\mathbf{S}^i|$ of these tasks. Each task gets mapped to one GPU thread. The GPU threads are organized in multiple thread blocks for each separation set S^i. We do not

[2] https://github.com/ChristopherSchmidt89/gpucmiknn

apply task agglomeration but rely on global communication to check the *used* positions (see line 25 of Algorithm 13).

For the CMI calculation, we suggest using one GPU kernel to estimate the partial CMI values for all adjacent variables $V_j \in adj(\mathcal{C}, V_i)$ based on one separation set S^i at a time. For this parallel version of the ESTIMATE$_{\text{CMIKNN}}$ kernel, we define a parallel task as processing one observational data sample, i.e., $D_m(V_i, V_j, S^i)$, for the base CMI or $D_m(\hat{V}_i[p], V_j, S^i)$, for a permuted sample with $p \in \{1, \ldots, perm\}$. The GPU kernel computes the partial CMI values for all edges (V_i, V_j) for one separation set $S^i \in \mathbf{S}^i$ in parallel. Therefore, the GPU kernel processes a total of $n \times (perm + 1) \times adj(\mathcal{C}^l, V_i)$ tasks. Again, each task is mapped to exactly one GPU thread, organized into multiple thread blocks for any given edge (V_i, V_j). We do not agglomerate tasks to address global communication, which is needed to share the $parital_{cmi}$ values (see line 28 of Algorithm 14) between GPU threads from multiple thread blocks.

Algorithmic Outline of `GPU`$_{\texttt{CMIknn}}$`-Parallel`

In the following, we present the algorithmic outline of `GPU`$_{\texttt{CMIknn}}$`-Parallel` that builds upon the parallel execution strategy discussed above. First, we highlight differences in `GPU`$_{\texttt{CMIknn}}$`-Parallel` compared to existing GPU-accelerated and parallel CPU-based adjacency searches for PC-stable. Next, we describe the input and output of `GPU`$_{\texttt{CMIknn}}$`-Parallel` and elucidate the procedure in detail.

Differences to Existing Parallel Adjacency Searches Within PC-Stable

In contrast to existing GPU-based variants, which launch one GPU kernel per level [81, 226, 287], `GPU`$_{\texttt{CMIknn}}$`-Parallel` launches multiple GPU kernels for each variable V_i in any level. One GPU kernel computes the local permutations concerning the variable's adjacency $adj(\mathcal{C}, V_i)$. Depending on the size of the variable's adjacency, multiple GPU kernels are launched to calculate the CMI values.

In contrast to parallel CPU-based algorithms, which parallelize over edges [123, 224, 234], `GPU`$_{\texttt{CMIknn}}$`-Parallel` adopts a fine-grained nested parallel execution strategy. The fine-grained nested parallel execution strategy ensures that computed local permutations are reused to reduce the overall computational demand. Reusing the local permutations is achieved by defining tasks for parallel execution in the context of the adjacency of one variable. At the same time, mapping these tasks to GPU threads ensures that each GPU thread processes individual data samples over multiple CI tests in parallel.

Input and Output

Algorithm 15 (see p. 107) receives the standard input parameters of PC-stable, such as observational data D, the number of data samples n, the vertex set $\mathbf{V}$ representing the N variables $\mathbf{V} = \{V_1, \ldots, V_N\}$ (see Section 2.1), or the significance level α. Further, the algorithm takes input parameters specific to `GPU`$_{\texttt{CMIknn}}$, such as the number of permutations $perm$, the k-nearest neighbors within permutation k_{perm}, and the k-nearest neighbors within CMI estimation k_{CMI}. `GPU`$_{\texttt{CMIknn}}$`-Parallel` outputs the estimated skeleton matrix $\mathcal{C}$ and the corresponding separation sets matrix Sep.

Algorithm 15 $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`: GPU-based adjacency search of PC-stable
Input: Observational data D with n samples of N variables $\mathbf{V} = \{V_1, \ldots, V_N\}$, significance level α, number of permutations $perm$, k-nearest neighbors within permutation k_{perm}, k-nearest neighbors within CMI estimation k_{CMI}
Output: Estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep

1: Start with fully connected skeleton $\mathcal{C}$ and $l \leftarrow -1$
2: **repeat**
3: $l \leftarrow l + 1$
4: **if** $l = 0$ **then**
5: On GPU: Process all pairs of variables $(V_i, V_j) \in \mathcal{C}$ in parallel
6: **else**
7: **for all** variables V_i in $\mathcal{C}$ **do**
8: Let $a(V_i) \leftarrow adj(\mathcal{C}, V_i)$;
9: **end for**
10: **for all** variables V_i in $\mathcal{C}$ with $|a(V_i)| > l$ **do**
11: Compute all possible separation sets $\mathbf{S}^i$ from $a(V_i)$
12: Launch kernel localPermutationExt on GPU to compute $localPerm[S^i]$ for all $S^i \in \mathbf{S}^i$ with D, $perm$ and k_{perm}
13: **repeat**
14: Choose S^i from $\mathbf{S}^i$
15: Launch kernel estimate$_{\text{CMIknn}}$Ext on GPU to estimate CMI for all $V_j \in a(V_i) \setminus \{S^i\}$ with D, S^i, $perm$, k_{CMI} and $localPerm[S^i]$
16: **for all** $V_j \in a(V_i) \setminus \{S^i\}$ **do**
17: Compute p based on computed CMI values
18: **if** p $\geq \alpha$ **then**
19: Delete edge $V_i - V_j$ from $\mathcal{C}$
20: Store S^i in Sep
21: Remove V_j from $a(V_i)$
22: **end if**
23: **end for**
24: **until** all computed S^i were chosen or $|a(V_i)| = 0$
25: **end for**
26: **end if**
27: **until** each V_i in $\mathcal{C}$ satisfies $|a(V_i)| < l$
28: **return** $\mathcal{C}$, Sep

Description of the Procedure of $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`

The algorithm starts with a fully connected skeleton $\mathcal{C}$ in level $l = 0$. In level $l = 0$, the separation sets are empty, i.e., $S^{i,j} = \emptyset$, and the algorithm applies the version of $\texttt{GPU}_{\texttt{CMIknn}}$ developed for unconditioned independence testing. Thus, $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` processes all edges in parallel, according to existing GPU-accelerated algorithms (see lines 4–6 of Algorithm 15). We omit detail on level $l = 0$ for brevity, as the procedure is straightforward.

In levels $l \geq 1$, $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` performs the following steps. First, for each variable $V_i \in \mathcal{C}$, the algorithm obtains an adjacency set from the current skeleton $\mathcal{C}$ (see lines 7–9). Next, the algorithm iterates over all variables $V_i \in \mathcal{C}$ whose adjacency's size $|a(V_i)|$ is larger than the current level l (see lines 10–25).

The algorithm computes all possible separation sets $\mathbf{S}^i$ for V_i in each iteration. Next, the GPU kernel localPermutationExt calculates the local

permutations of V_i and all possible separation sets $\mathbf{S}^i$ in parallel. Upon termination of the GPU kernel localPermutationExt, the data structure $localPerm$ contains the local permutations. Note that the GPU kernel localPermutationExt is the extended version of the GPU kernel localPermutation (see Algorithm 13), which computes local permutations for multiple separation sets at once. Therefore, GPU kernel localPermutationExt is launched with additional thread blocks, setting δ in the second grid dimension. The value for δ is chosen according to the number of possible separation sets, i.e., $\delta = |\mathbf{S}^i|$.

In the subsequent step, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ iterates the possible separation sets $S^i \in \mathbf{S}^i$ (see lines 13–24 of Algorithm 15). In each iteration, the algorithm computes the partial CMI values for V_i and all $V_j \in a(V_i) \backslash \{S^i\}$ given the current separation set S^i. During this operation, the local permutations of V_i and S^i are reused to calculate the partial CMI values for V_i and all V_j. The corresponding partial CMI values get computed in the $\text{estimate}_{\text{CMIknn}}$Ext GPU kernel. The $\text{estimate}_{\text{CMIknn}}$Ext GPU kernel is an extension of the $\text{estimate}_{\text{CMIknn}}$ GPU kernel (see Algorithm 14) that estimates the partial CMI values for multiple edges at once. Therefore, the GPU kernel is launched with additional δ thread blocks in the third grid dimension. The value for δ is chosen according to the remaining adjacent variables $a(V_i)$, i.e., $\delta = |a(V_i) \setminus \{S^i\}|$.

Afterward, the algorithm computes the p-value using the corresponding partial CMI values for each $V_j \in a(V_i) \setminus \{S^i\}$. If $\mathrm{p} \geq \alpha$, the edge $E^{i,j}$ is removed from the current skeleton $\mathcal{C}$, the separation set S^i is stored in Sep at the position of $E^{i,j}$, and V_j is removed from $a(V_i)$ (see lines 18–22 of Algorithm 15). V_j is removed from $a(V_i)$ to avoid unnecessary computations in subsequent iterations. Once all possible $S^i \in \mathbf{S}^i$ have been chosen, or there is no adjacent variable left in $a(V_i)$, the inner loop is finished.

After all variables $V_i \in \mathcal{C}$ have been considered in the current level l, the subsequent level $l = l + 1$ is processed. All previous steps are repeated 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 $V_i \in \mathcal{C}$. At this point, the algorithm returns the current skeleton $\mathcal{C}$ and the corresponding separation sets stored in Sep.

Additional Notes

The outlined approach of $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ introduces two additional reasons for exceeding the GPU's memory capacity. In high-dimensional or dense CGMs, the number of possible separation sets $|\mathbf{S}^i|$ and the number of adjacent variables $|a(V_i)|$ can become too large to process them simultaneously, within the same GPU kernel. For both cases, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ provides a blocked version to avoid exceeding GPU global memory. If too many separation sets are processed at once, the blocked version splits the separation sets for a variable V_i into small-sized blocks. The small-sized blocks are processed in an additional loop, which performs all steps in lines 12–24 of Algorithm 15 for each block of separation sets. Similarly, to address computing partial CMI values for too many adjacent variables $V_j \in a(V_i)$ in parallel, the blocked version splits the adjacent variables into small-sized blocks. The blocks of adjacent variables are processed in an additional loop, which performs the steps shown in lines 15–23 in each iteration.

4.5 Summary

This chapter presented our three GPU-based variants of the PC algorithm's adjacency search designed for CI tests that are common in practice and cover a wide range of data characteristics. Therefore, we first deduced parallel execution strategies for a GPU-accelerated adjacency search within the PC-stable algorithm with varying task granularity, according to Foster's methodology for parallel algorithm design [61]. Using the deduced parallel execution strategies, we introduced one GPU-accelerated adjacency search for the case that data follows the Gaussian distribution model. We argued that, in this case, GPU acceleration is most beneficial in levels $l = 0, 1$ and developed GPU kernels for both levels. Additionally, we proposed two approaches to process higher levels $l \geq 2$. Further, we described a GPU-accelerated adjacency search for discrete data. In this context, we discussed different strategies to handle the required auxiliary data structures when units of GPU threads, i.e., warps, jointly compute marginals over contingency tables. We argued that allocating a fixed-size memory block in GPU global memory before the GPU kernel launch is a suitable solution. Finally, we proposed $\texttt{GPU}_{\texttt{CMIknn}}$, a GPU-accelerated CI test for mixed discrete-continuous data and data with non-linear relationships, and $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, a GPU-based adjacency search tailored to the use of $\texttt{GPU}_{\texttt{CMIknn}}$. $\texttt{GPU}_{\texttt{CMIknn}}$ requires the execution of k-NN searches during local permutation computation and CMI estimation. The implementation uses a pipelined parallel brute-force-based k-NN search that avoids intermediate results and leverages GPU thread-local memory. The $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ algorithm reuses the computed local permutations while executing multiple CI tests in parallel.

5

GPU-Based CSL Beyond a Single GPU's Memory Capacity

The GPU-accelerated adjacency search algorithms for data that follows the Gaussian distribution model, for discrete or mixed discrete-continuous data, and data with non-linear relationships proposed in Chapter 4 have a fundamental restriction. The algorithms terminate in error if high-dimensional datasets exceed the GPU's memory capacity. Such high-dimensional data has various reasons. Most notably, high-dimensional data occurs when models containing tens of thousands of variables or millions of observational data samples are considered. Also, intermediate results due to large domains in discrete variables, thousands of permutations, or considering many k-nearest neighbors lead to GPU memory capacity overruns.

Erroneous execution due to exceeding GPU memory is a common shortcoming in GPU-accelerated algorithms, which gave rise to both, application-agnostic [155, 292] and application-specific solutions [34, 108, 263, 280] (see Section 3.2). Possible application-specific solutions include extending algorithms to operate on multiple GPUs [34, 80, 263] or applying out-of-core approaches [108, 225, 280], e.g., splitting the problem into consumable-sized blocks, which are processed in a streaming fashion. Note that *out-of-core* commonly refers to approaches exceeding the system's DRAM. Within this thesis, we use the term *out-of-core* in the context of exceeding the GPU's global memory.

In this chapter, we introduce approaches to scale GPU-accelerated constraint-based CSL to arbitrarily large datasets (see (RQ2)). We focus on multi-GPU and out-of-core GPU execution as solutions to overcome the limitation of our proposed GPU-accelerated adjacency search algorithms when exceeding the memory capacity of a single GPU. In particular, we discuss two separate approaches for each solution.

The first approach builds upon the concept of Unified Memory (UM), which uses the Memory Management Unit (MMU) of modern NVIDIA GPUs to transparently migrate page-sized memory as required by the GPU kernels (see Section 5.1). Generally, the concept of UM simplifies memory management in heterogeneous GPU-based systems from a development perspective but introduces overhead resulting in up to 2.2× slower runtimes [120].

The second approach is based on explicit memory management. In our context, we define explicit memory management in the following way: The algorithm using explicit memory management is responsible for the data management to avoid exceeding the GPU's memory capacity while processing arbitrarily sized

datasets. The data management comprises splitting data into consumable-sized blocks, moving these blocks to and results from the GPU, and orchestrating the GPU kernel execution concerning the blocks of data. In our work, we propose a block-based approach that processes blocks of data that fit into GPU global memory in a stream-like manner (see Section 5.2). *Parts of this section have been published in two research papers [80, 225].*

5.1 Unified Memory (UM)-Based GPU-Accelerated Adjacency Search in PC-Stable

The concept of UM introduces a single memory space that is accessible by CPU and GPU [85]. Whenever any of the processing units access data, an integrated page migration engine within the MMU of the GPU ensures that the data is moved to the accessing device at a page-sized granularity. In contrast to direct memory access over the interconnect, the migration of pages ensures that data processing on the GPU benefits from the integrated caches and the bandwidth of the device's memory [218]. Within modern GPUs, dedicated hardware features provide page fault mechanisms and support large address spaces beyond the GPU's memory capacity, which extends the capabilities of UM [86]. Thus, in modern GPUs, UM provides demand paging and allows for GPU global memory oversubscription [217]. These capabilities enable processing datasets that exceed the GPU's memory capacity, without explicitly managing parts of the dataset as the active working set in GPU global memory. Building upon the capabilities of UM in modern GPUs, we propose an out-of-core and a multi-GPU approach to our GPU-accelerated adjacency search algorithms for the PC-stable.

5.1.1 A UM-Based Out-of-Core GPU Approach

Extending the existing GPU-accelerated adjacency search algorithms to an out-of-core-based processing model that overcomes the limitation of the GPU's memory capacity is straightforward. Under the assumption that the GPU hardware supports UM with demand paging and GPU global memory oversubscription, only the allocation of data structures is changed. In particular, all data structures required during the adjacency search are allocated within the virtual memory address space provided by UM. `CUDA`, for example, provides the function cudaMallocManaged(...), which allocates data structures in UM [170]. The data is transferred automatically through the page migration engine for data structures allocated in this manner. Thus, steps for explicit data transfer between the host system and the GPU in the adjacency search algorithms (see Algorithms 3, 8, 15) are omitted. Furthermore, in the case of discrete or mixed data, handling the auxiliary data structures (e.g., see lines 5–16 of Algorithm 8, p. 85) is simplified by allocating these data structures using UM.

5.1.2 A UM-Based Multi-GPU Approach

The UM-based multi-GPU approach builds upon the same extension to allocate all data structures within UM as described above. We make the following conceptual change to extend the single GPU adjacency search algorithms to operate on multiple GPUs.

For data that follows the Gaussian distribution model and discrete data, a separate GPU kernel is launched for each level l in the single GPU case. This GPU kernel processes all edges in the current skeleton $\mathcal{C}^l$ in parallel. In the multi-GPU case, the processing of the edges within one level is split equally among the GPUs available for processing. Thus, within each level l, one GPU kernel is launched on each GPU, processing a dedicated share of edges. Once the GPU kernels for the current level finished on each GPU, the next level starts.

In the case of the information-theoretic CI test, $\texttt{GPU}_{\texttt{CMIknn}}$, we consider two different execution strategies. For $\texttt{GPU}_{\texttt{CMIknn}}$`-Single`, each GPU processes multiple edges within each level l. For $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel`, each GPU operates on multiple adjacencies within each level l.

In all cases, we follow a static task distribution to the GPUs to ensure data locality [158], where tasks are either edges or adjacencies. Thus, each GPU operates on the same set of edges or adjacencies within each level, and migration of data between GPUs due to page faults is kept low. However, this static distribution of edges or adjacencies to GPUs may cause load imbalance. Techniques to address load imbalance, e.g., work stealing [15], are left for future work.

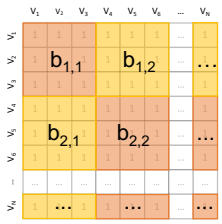
Besides the described conceptual change, the multi-GPU approach introduces two more minor adaptations. First, different GPU memory advices are used to guide the UM-based memory-managed data structures, following common best practices [219]. GPU memory advices are used, for example, to set preferred locations of data structures, to allow for prefetching of data, or to indicate that data is mostly read. In `CUDA`, GPU memory advices are set using cudaMemAdvise(...). Second, the multi-GPU approach launches a dedicated CPU thread for each GPU, which handles the GPU memory advices and GPU kernel launches per GPU.

5.2 Explicit Memory-Managed GPU-Accelerated Adjacency Search in PC-Stable

In high-dimensional settings, where input data exceeds the GPU's memory capacity, explicit memory management for a GPU-accelerated adjacency search splits the input data into consumable-sized blocks that fit into GPU global memory. Further, the GPU-accelerated adjacency search needs to orchestrate data transfer between the host system and GPU and GPU kernel execution concerning the data block present in GPU global memory. At the cost of this additional implementation effort, the explicit memory-managed approach is independent of modern GPU hardware features required for UM. Furthermore, any runtime overhead introduced by UM is avoided [120]. Based upon explicit memory management, we propose a block-based out-of-core GPU adjacency search for a single GPU and extend the approach to a multi-GPU setting. In the following, we describe both algorithms in detail.

5.2.1 A Block-Based Out-of-Core GPU Adjacency Search

The block-based out-of-core GPU adjacency search extends our aforementioned GPU-accelerated algorithms (see Section 4.2 and Section 4.3) to operate on high-dimensional data, exceeding a single GPU's memory capacity. The proposed block-based algorithm (see Algorithm 16, p. 115) splits the input data



(a) $\{b_{1,1}, b_{1,2}, b_{2,1}, b_{2,2}\}$ are example blocks of block size $bs = 3$ extracted from the estimated skeleton matrix $\mathcal{C}$.

```
struct Block{
        int* _skeleton;
        int* _skeletonHat;
        int* _Sep;
        int* _D;
        float* _sCor;
        int* _auxiliaryData;
        int* _mapping;
};
```

(b) `C` struct of a block, showing the included data structures.

Fig. 5.1: Illustration of data split into blocks (left) and an overview of the data structures included in each block (right).

into consumable-sized blocks, which are described next. Note that the GPU-accelerated adjacency search $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ (see Section 4.4.3) provides a similar block-based execution model.

Description of Blocks

In the context of the block-based out-of-core GPU adjacency search, a block is defined as an excerpt of the estimated skeleton matrix $\mathcal{C}$ with dimensions $bs \times bs$ (see Figure 5.1a). The block contains the corresponding data structures required for processing (see Figure 5.1b). The parameter bs denotes the block size, which is provided as an input to the algorithm. The choice of the value for bs follows two considerations. First, bs should be large enough for ample parallelism within each block to utilize the GPU hardware efficiently. Second, bs should be small enough to allow storing multiple blocks in GPU global memory. Multiple blocks are needed for constructing all possible separation sets and efficient execution through overlapping GPU kernel execution and data transfer. The data structures within each block are excerpts of the matrices storing the estimated skeleton $\mathcal{C}$, $\hat{\mathcal{C}}$, the matrix storing the correlations $sCor$ or observational data samples D, and the matrix storing the separation sets Sep and any required auxiliary data. Further, each block contains a mapping of its position within the original, non-blocked estimated skeleton data structures.

Input and Output

The block-based out-of-core GPU adjacency search receives the same input parameters as the single GPU Algorithms 3, 8 and produces the same output, namely the estimated skeleton matrix $\mathcal{C}$ and the separation set matrix Sep. Additionally, the block-based out-of-core GPU algorithm takes the block size bs, which determines the size of the blocks, as an input parameter.

Algorithm 16 Block-based out-of-core GPU adjacency search
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$ or observational data matrix D, significance level α, maximum level ml, number of data samples n, block size bs
Output: Estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep

```
1: l ← 0
2: Let C and Ĉ be |V| × |V| matrices with all entries set to 1
3: Let Sep be an |V| × |V| × ml matrix with all entries set to −1
4: blocks ←SPLIT(C, Ĉ, Sep, sCor, D, bs)
5: while ml > l do
6:   for all b in blocks do
7:     TRANSFERTOGPU(b)
8:     if l = 0 then
9:       LAUNCHGPUKERNEL(CITest0, {b, α, n})
10:    else
11:      sepsetblocks ← SEPSETCOMBINATION(b, l, blocks)
12:      for all s in sepsetblocks do
13:        TRANSFERTOGPU(s)
14:        LAUNCHGPUKERNEL(CITestL, {b, s, α, n})
15:      end for
16:    end if
17:    TRANSFERFROMGPU(b)
18:  end for
19:  l ← l + 1
20:  UPDATE(ml)
21: end while
22: MERGE(blocks)
23: return C, Sep
```

Description of the Procedure

At first, the algorithm initializes the required data structures (see lines 1–3 of Algorithm 16). Next, all blocks B of dimension $bs \times bs$ are extracted based on the input data. These blocks are stored in a list called *blocks*. Note that all blocks $b \in B$ are disjunct subsets of the input data. If the dimension of the input data N is not a multiple of bs, zero-padding is applied to ensure that all blocks $b \in B$ are of dimension $bs \times bs$.

After the data preparation, the algorithm processes all levels l sequentially as long as the condition $ml > l$ holds (see lines 5–21). All blocks b in *blocks* are iterated within each level, and each block b is processed. First, all data structures in b are transferred to the GPU (see line 8). If the algorithm operates on level $l = 0$, the GPU kernel to process the current block b is launched directly (see line 9). Depending on the underlying data distribution $CITest0$ either implements the CI test for the Gaussian distribution model or discrete data according to the algorithms mentioned above (see Algorithms 3 and 8).

Additional blocks to construct all possible separation sets are loaded to the GPU, for all other levels $l \geq 1$ Therefore, the list of *sepsetblocks* containing all combinations of sets of blocks of size l for the current block b is determined by calling the function SEPSETCOMBINATION(...) (see line 11 of Algorithm 16). Next, each set of blocks s in *sepsetblocks* is processed sequentially (see lines 12–15). The blocks within s are transferred to the GPU and a GPU kernel processing the CI tests for block b considering the set of separation set

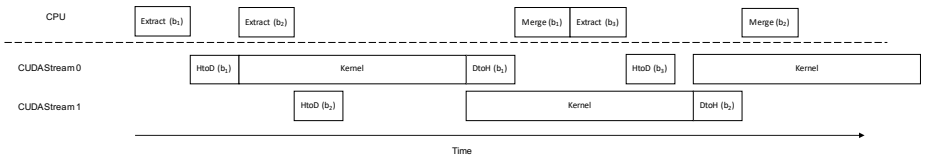


Fig. 5.2: Overlapping operations on CPU with data transfers (host to GPU, denoted HtoD, or GPU to host, denoted DtoH) and GPU kernel execution in two separate `CUDA` streams to reduce the overall runtime of the block-based out-of-core GPU adjacency search.

blocks s is launched. The GPU kernel implementation of $CITestL$ is set according to the CI test appropriate to the underlying data distribution (see Algorithms 3, 8).Once block b is processed in the current level l, the data of block b is transferred from the GPU to the host system (see line 17 of Algorithm 16).

If all blocks B have been processed, the level l is incremented, the maximum level ml is updated, and the algorithm continues with the subsequent level. In case $ml \leq l$, all possible CI tests are conducted, and the result data structures from all blocks $b \in B$ are merged (see line 22 of Algorithm 16). Afterward, the estimated skeleton $\mathcal{C}$ and the matrix containing the corresponding separation sets Sep are returned.

`CUDA`-Based Implementation

In a `CUDA`-based implementation of Algorithm 16, we apply the concept of `CUDA` streams to overlap data transfer and GPU kernel execution to reduce the overall runtime. In detail, we use two separate `CUDA` streams and one orchestrating CPU thread, as depicted in Figure 5.2. While the GPU kernel processing a given block b_1 operates in the first `CUDA` stream, the second `CUDA` stream is used to transfer data for the next block b_2. Once the operations in both `CUDA` streams are finished, the GPU kernel processing block b_2 is launched in the second `CUDA` stream, while the result of b_1 is transferred to the host system's DRAM, and the next block b_3 is transferred to the GPU in the first `CUDA` stream. Thus, the operations in the `CUDA` streams alternate between data transfer and GPU kernel execution. Only the first and last block's data transfer is not overlapped with GPU kernel execution. The CPU thread is responsible for extracting and storing the next block, initiating the data transfer, and asynchronous GPU kernel execution. Note, the same concept to overlap data transfer and kernel execution applies to the separation set blocks.

5.2.2 A Block-Based Multi-GPU Adjacency Search

For use in a multi-GPU system, the block-based out-of-core approach is extended. In particular, the block-based multi-GPU approach uses multiple CPU threads and a central task queue, as outlined in Algorithm 17 (see p. 117). The introduction of a central task queue enables load balancing across multiple GPUs. In this context, each block is defined as a separate task. Furthermore, for each GPU in use, exactly one CPU thread is started and responsible for communicating with that GPU. Having dedicated CPU threads for each GPU prevents the CPU from becoming the performance bottleneck.

Algorithm 17 Block-based multi-GPU adjacency search
Input: Vertex set $\mathbf{V}$, correlation matrix $sCor$ or observational data matrix D, significance level α, maximum level ml, number of data samples n, block size bs, GPU count g
Output: Estimated skeleton matrix $\mathcal{C}$, separation sets matrix Sep

```
 1: threadList ←createThreads(g)
 2: l ← 0
 3: Let C and Ĉ be |V| × |V| matrices with all entries set to 1
 4: Let Sep be an |V| × |V| × ml matrix with all entries set to −1
 5: blocks ←split(C, Ĉ, Sep, sCor, D, bs)
 6: while ml > l do
 7:   queue ←fillQueue(blocks)
 8:   for all t in threadList do
 9:     repeat
10:       b ←queue.pop()
11:       transferToGPU(b, GPU_t)
12:       if l = 0 then
13:         launchGPUKernel(CITest0, {b, α, n}, GPU_t)
14:       else
15:         sepsetblocks ←sepSetCombination(b, l, blocks)
16:         for all s in sepsetblocks do
17:           transferToGPU(s, GPU_t)
18:           launchGPUKernel(CITestL, {b, s, α, n}, GPU_t)
19:         end for
20:       end if
21:       transferFromGPU(b, GPU_t)
22:     until queue is empty
23:   end for
24:   l ← l + 1
25:   update(ml)
26: end while
27: merge(blocks)
28: return C, Sep
```

Input and Output

The number of GPUs used for processing is passed to the algorithm as an additional parameter, denoted by g. Otherwise, the algorithm receives the same input parameters as the block-based out-of-core GPU adjacency search (see **Input** of Algorithm 16). Further, the same output is returned.

Description of the Procedure

According to the parameter for the number of GPUs, g, multiple CPU threads are created and managed in a list (see line 1 of Algorithm 17, p. 117). In the following, the data preparation steps for the single-GPU case are performed (see lines 2–5). At the beginning of each level l, the central task queue *queue* is filled with all blocks to be processed in the multi-GPU case (see line 7). Then, each of the created CPU threads performs the following steps until all blocks in the central task queue have been processed (see lines 8–22).

First, the CPU thread t takes a block from the task queue. Then the data of the current block is transferred to the GPU GPU_t assigned to the CPU thread. In the case of level $l = 0$, the GPU kernel to process the block is started directly. Otherwise, the possible sets of separation set blocks are calculated and processed in a loop. In each iteration, the data of one separation set block is transferred to the corresponding GPU GPU_t before the GPU kernel is launched. Once all possible sets of separation set blocks have been processed, the results of the current block are transferred from GPU_t to the host system. The subsequent level is started when the central task queue *queue* contains no more elements. Once all levels have been processed, i.e., $ml \leq l$, the blocks are merged, and the algorithm returns a matrix containing the estimated skeleton $\mathcal{C}$, and a matrix of the corresponding separation sets Sep.

Note that the operations performed by each CPU thread are realized analogously to the single-GPU case (see Figure 5.2, p. 116). Thus, each CPU thread uses `CUDA` streams mapped to its assigned GPU to overlap data transfer and GPU kernel execution.

5.3 Summary

In this chapter, we presented two approaches to scaling GPU-accelerated constraint-based CSL to arbitrarily large datasets, i.e., datasets which exceed the GPU memory capacity. The first out-of-core approach builds upon the concept of UM. UM allows GPU global memory oversubscription and uses the MMU to migrate memory pages on demand to the requesting PU. This approach requires few adaptations but introduces overhead at runtime [120], despite the use of best practices [219]. The second out-of-core approach relies on explicit memory management. Therefore, the proposed algorithm splits the input dataset into consumable-sized blocks that fit into GPU global memory. Further, the algorithm orchestrates data transfer and GPU kernel execution to ensure that the required data resides in GPU global memory. This block-based out-of-core approach overlaps data transfer and GPU kernel execution to minimize overhead. We extended both approaches to operate on multiple GPUs to benefit from modern multi-GPU computing systems.

6

Evaluation

In this chapter, we present results from our experimental evaluation of the algorithms for GPU-accelerated CSL that we proposed in this thesis, as described in Chapters 4 and 5.

First, we describe the experimental setup in Section 6.1. Particularly, we detail the datasets with their associated CGMs and the hardware of heterogeneous systems used for the experiments. Additionally, we describe state-of-the-art implementations for constraint-based CSL and naive baseline implementations used for comparison in our evaluation.

Second, we present the results of our experiments for our GPU-accelerated algorithms for data with different distributions in Section 6.2, which address our first research question (RQ1). Concerning our second research question (RQ2), we cover experiment results for our GPU-based approaches that scale beyond a single GPU's memory capacity in Section 6.3. Our experiments cover our algorithms as introduced in Chapters 4 and 5, measure their runtimes to illustrate the possible performance gains over state-of-the-art CSL algorithms, and show the limitations of our approaches.

Finally, we discuss the results of our experimental evaluation in Section 6.4. Particularly, we argue for cases in which GPU acceleration for constraint-based CSL is well-suited and discuss settings in which GPU acceleration introduces too much overhead. We close this chapter with a summary in Section 6.5. *Parts of this chapter have been published in six research papers [16, 80, 81, 83, 225, 226].*

6.1 Experimental Setup

In this section, we describe the experimental setup used for our evaluation.

At first, we elucidate the datasets with their associated CGMs used to compare the runtimes of our proposed algorithms (see Section 6.1.1). We cover a range of real-world gene expression datasets and well-known benchmark Bayesian networks that provide an underlying CGM. We also describe synthetic data that can be used to examine settings beyond the abovementioned datasets.

Second, we detail the characteristics of different heterogeneous hardware systems used to execute our experiments (see Section 6.1.2). Our proposed algorithms have been evaluated at the time of publication on the system with the most recent hardware available to us. Therefore, we used three systems with different GPU generations and one additional multi-core CPU-only system.

Lastly, we elaborate on state-of-the-art implementations stemming from multiple well-known libraries used to compare to our proposed algorithms (see Section 6.1.3). If applicable, we also cover naive baseline implementations that we use for comparisons.

6.1.1 Description of the Datasets Used for Experimental Evaluation

In our experiments, we measure the runtimes of our proposed algorithms on multiple real-world datasets and well-known benchmark Bayesian networks. These datasets and networks are commonly used to compare approaches of CSL [78, 106, 123, 126, 169, 234, 287]. Accordingly, we obtain comparability to existing experimental evaluations. Furthermore, we show the applicability of our proposed algorithms in realistic settings. Additionally, we measure the runtime of our algorithms on synthetic data to illustrate settings that are not covered by the selection of real-world datasets. Using synthetic data allows us to investigate a broader range of cases to generalize our findings and pinpoint shortcomings. Furthermore, we rely on synthetic data in the case of mixed discrete-continuous data and continuous data with non-linear relationships, as, to the best of our knowledge, there are no suitable publicly available real-world datasets. Hereinafter, we first elucidate the real-world datasets and benchmark Bayesian networks. Second, we detail the synthetically generated data used for evaluating our proposed algorithm.

Real-World Datasets and Benchmark Bayesian Networks

Table 6.1 (see p. 121) summarizes the real-world gene expression datasets and the benchmark Bayesian networks used in the evaluation of this thesis. In particular, for each dataset, the table states, which distribution model the associated data is assumed to follow, the number of variables N, the number of data samples n, and the maximum size of the domain $\max_{i=1,\dots,N}\{|\mathcal{V}_i|\}$, with $\mathcal{V}_i$ representing the corresponding discrete domain of the variable V_i and $i = 1, \dots, N$. For all gene expression datasets, the data is assumed to follow the Gaussian distribution model. The datasets `NCI-60`, `MCC`, `BR51`, `DREAM5-INSILICO`, `S.AUREUS`, and `S.CEREVISIAE`, were downloaded from [123], and the dataset `TCGA` was downloaded from [193]. Concerning the number of variables, the datasets downloaded from [123] are high-dimensional, ranging from 1 190 variables for `NCI-60` to 5 361 variables for `S.CEREVISIAE`. Except for `DREAM5-INSILICO`, which has 850 data samples, the number of data samples of these gene expression datasets is small, ranging from 47 data samples for `NCI-60` up to 160 data samples for `S.AUREUS`.

In contrast, the `TCGA` dataset [193], which is not considered in any other work on CSL, has 55 572 variables and 3 189 data samples. We use `TCGA` as a representative of a high-dimensional dataset that exceeds a single GPU's memory capacity to evaluate our out-of-core GPU and multi-GPU algorithms.

The benchmark Bayesian networks, `ALARM`, `ANDES`, `LINK`, and `MUNIN`, downloaded from the bnlearn repository [233], assume a discrete distribution model. For each benchmark Bayesian network, the underlying CGM is provided. Accordingly, any number of data samples that follow the associated distribution can be generated. In our case, we choose to generate 10 000, 20 000, or 200 000 data samples to cover small-sized and large-sized datasets. Furthermore, for the

real-world gene expression dataset/benchmark Bayesian network	distribution model	N - number of variables	n - number of data samples	$\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\}$ - maximum size of the discrete domain
`NCI-60` [125]	Gaussian	1 190	47	-
`MCC` [125]	Gaussian	1 380	88	-
`BR51` [125]	Gaussian	1 592	50	-
`DREAM5-INSILICO` [152]	Gaussian	1 643	850	-
`S.AUREUS` [152]	Gaussian	2 810	160	-
`S.CEREVISIAE` [143]	Gaussian	5 361	63	-
`TCGA` [21, 193]	Gaussian	55 572	3 189	-
`ALARM` [10]	Discrete	37	10 000 / 200 000	4
`ANDES` [37]	Discrete	223	20 000	2
`LINK` [104]	Discrete	724	20 000	4
`MUNIN` [7]	Discrete	1 041	20 000	21

Table 6.1: Characteristics of gene expression datasets (top) commonly used to evaluate CSL algorithms and benchmark Bayesian networks (bottom) downloaded from the bnlearn repository [233]. For the gene expression datasets, the distribution model refers to the commonly assumed one, and the maximum size of the domain only applies to discrete data. To the best of our knowledge, there is no suitable publicly available real-world mixed discrete-continuous dataset.

discrete datasets, the table states the maximum size of the domain, which is 2 for `ANDES`, 4 for `ALARM` and `LINK`, and 21 for `MUNIN`. Note that the maximum size of the domain has implications on the memory demand for the contingency table and marginals constructed during the execution of a CI test. Concerning the number of variables, the four considered Bayesian networks cover small-dimensional datasets, e.g., `ALARM` and `ANDES`, with 37 and 223 variables, and high-dimensional datasets, e.g., `LINK` and `MUNIN`, with 724 and 1041 variables.

Synthetic Data

In the experimental evaluation of our proposed algorithms, we use synthetic data to investigate settings not covered by real-world datasets and benchmark Bayesian networks. To generate synthetic data, we employ two different options.

The first option builds upon data generation frameworks, e.g., as provided by the `pcalg` library [111] or the `MANM-CS` library [98]. These frameworks take parameters that characterize a CGM. Based on these parameters, the frameworks randomly generate CGMs and sample data accordingly. To the best of our knowledge, the data generation tool within the `pcalg` library [111] is restricted to generating data following a linear Gaussian distribution model. Therefore, we employ the `MANM-CS` library [98], which allows for generating CGMs with mixed discrete-continuous variables and non-linear relationships.

The second option we use to obtain synthetic data abstracts from any underlying CGM and causal mechanism and randomly generates numeric or discrete values. We employ this data generation approach to study the proposed algo-

rithms' behavior in isolated settings. In these settings, we are interested in the impact of a specific algorithm parameter. Such settings occur, for example, when we investigate the scalability concerning the number of CI tests of our implemented GPU kernel for a certain level l. A second example is when we measure the impact of increased maximum sizes of the domain $\max_{i=1,\dots,N}\{|\mathcal{V}_i|\}$.

In the following, we describe the synthetically generated data assuming the different distribution models considered in this thesis.

Gaussian Distribution Model

For the experimental evaluation of our proposed GPU-accelerated adjacency search for the Gaussian distribution model and its extension to scale beyond a single GPU's memory capacity, we generate synthetic data to understand the algorithms' scalability. Therefore, the synthetic datasets contain an increasing number of variables. Consequently, the number of CI tests that must be performed increases. Under the assumption that the utilized CI test, i.e., Fisher's z-test [59], works on a pre-computed square correlation matrix, the synthetic data comprises randomly generated correlation matrices. The data generation method sets the diagonal of the correlation matrix to 1. It also sets each entry in the upper matrix and its mirrored value in the lower triangular to a randomly drawn numeric value between 0 and 1.

To study our GPU-accelerated approaches to scale beyond a single GPU's memory capacity, we generate correlation matrices with an increasing number of variables ranging from $N = 5\,000$ to $N = 80\,000$ in steps of 5 000 or 10 000. Similarly, for our GPU-accelerated adjacency search on a single GPU, we generate random correlation matrices with dimensions of 10×10 up to $25\,000 \times 25\,000$ to investigate the behavior with an increasing number of performed CI tests. As a result, in level $l = 0$, between 45 to 312 487 500 CI tests are performed. Moreover, in level $l = 1$, considering the worst case of performing all possible CI tests, 360 to approximately $7.8 \cdot 10^{12}$ CI tests are conducted.

Discrete Distribution Model

In the case of our proposed GPU-accelerated adjacency search for discrete data, we consider five isolated settings. In particular, we want to study the impact of the number of variables N, the number of data samples n, the maximum size of the dimension $\max_{i=1,\dots,N}\{|\mathcal{V}_i|\}$, the number of CI tests conducted per edge, which we denote with CI_{edge}, and the impact of a decay factor df. The decay factor df describes the percentage of edges removed equally distributed from skeleton $\mathcal{C}$ within each level l of the adjacency search. We see df as a proxy for the density of underlying CGMs, but use a fixed deletion rate and strategy for better interpretability. The parameter CI_{edge} allows us to investigate the impact of load imbalance due to changing numbers of CI tests per edge [224]. To generate data according to these settings, we create datasets of dimension $N \times n$ and sample discrete values from $\{0, \dots, \max_{i=1,\dots,N}\{|\mathcal{V}_i|\}\}$. Generating data according to the parameters CI_{edge} and df is challenging. Therefore, we manually set the current skeleton $\mathcal{C}$ after each level l to guarantee to fulfil the parameter df, Furthermore, to ensure that parameter CI_{edge} is followed, we manipulate the function that determines the separation set for a given variable V_i to return a set of separation sets with a size equal to CI_{edge}.

Mixed Discrete-Continuous Data and Data with Non-Linear Relationships

Under the assumption of mixed discrete-continuous data or data with non-linear relationships, we utilize the `MANM-CS` library [98] to generate synthetic CGMs with associated data. We set the number of data samples n and the number of variables N according to the specification of each experiment. For each such parameter setting, we generate CGMs that contain a mix of discrete and continuous variables with selected ratios from $\{0.0, \ldots, 1.0\}$. The CGMs have an edge density randomly chosen from $\{0.1, \ldots, 0.5\}$. For edges between continuous variables, we randomly select functions from the following set: {*linear*, *quadratic*, *tanh*}. We use the library's defaults for the remaining data generation parameters of `MANM-CS`. Note that we use the same data generation method for experiments evaluating our CI test $\texttt{GPU}_{\texttt{CMIknn}}$ independent of its application within the PC-stable algorithm.

6.1.2 Heterogeneous Hardware Systems Used in Experiments

characteristics	`Galileo`	`Titan X`	`Delos`	`A40`
CPU	4× Intel® Xeon® E7-4850 v4	1× Intel® i7-6700K	2× Intel® Xeon® Gold 6148	1× AMD EPYC 7343
number of cores	16	4	20	16
size of DRAM	2 TB	64 GB	1.5 TB	96 GB
interconnect between CPU	QPI	-	UPI	-
interconnect between host & GPU	-	PCI-E 3.0	PCI-E 3.0	PCI-E 4.0
GPU	-	1× NVIDIA Geforce GTX Titan X	4× NVIDIA Tesla V100 SMX2	1× NVIDIA A40
size of GPU on-chip memory	-	12 GB GDDR5	32 GB HBM2	48 GB GDDR6
memory bandwidth on GPU	-	$336\frac{GB}{s}$	$900\frac{GB}{s}$	$696\frac{GB}{s}$
number of SMs	-	24	80	84
interconnect between GPUs	-	-	NVLink 2.0	-
operating system: version of Ubuntu	18.04	16.04	18.04	21.04
`CUDA` version	-	9.0	9.1, 11.2, 11.3	11.4

Table 6.2: Selected characteristics of the hardware systems used for our evaluation. The top parts refer to the hardware features of the CPUs and the GPUs. The bottom part mentions software specifications.

Table 6.2 (see p. 123) presents selected characteristics of the heterogeneous hardware systems used for our experimental evaluation. In particular, we report the number and type of CPUs together with the number of CPU cores. Further, we mention the size of the system's DRAM, the interconnect technology to connect CPUs, and the interconnect technology to link the host system with the GPUs, if applicable. We mention the number and type of GPUs, the size of the GPU on-chip memory with its bandwidth, and the number of Streaming Multiprocessors (SMs). We also state the interconnect technology used to connect multiple GPUs, if applicable. Lastly, we report the operating system and the `CUDA` version installed when the systems were used for experiments. All these features are listed for four different hardware systems. We use one multi-core CPU system with four CPUs called `Galileo`. Further, we run experiments on two heterogeneous systems equipped with one multi-core CPU and one GPU, named `Titan X` and `A40`. Lastly, we conduct experiments on a heterogeneous multi-GPU system with two multi-core CPUs and four GPUs called `Delos`.

Multi-Core CPU System: `Galileo`

`Galileo` is a CPU-only system with four Intel CPUs, each with 16 physical cores and hyperthreading. The CPUs are connected via Intel's QuickPath Interconnect (QPI). The system has a DRAM capacity of 2 TB. We use the `Galileo` system in one experiment measuring runtimes using a high-dimensional dataset that exceeds a single GPU's memory capacity. This high-dimensional dataset has a large memory footprint requiring a system with enough DRAM capacity.

Heterogeneous Single-GPU System: `Titan X`

`Titan X` is a single-GPU system with one four-core Intel CPU and 64 GB of DRAM. The host system has one attached GPU that is connected via PCI-E 3.0. The attached GPU is an NVIDIA Geforce GTX Titan X [262] with 24 SMs. The GPU has 12 GB of GDDR5 on-chip memory with a bandwidth of $336\frac{GB}{s}$. At the time we performed the experiments on `Titan X`, the operating system was Ubuntu version 16.4, and `CUDA` version 9.0 was installed.

Heterogeneous Multi-GPU System: `Delos`

`Delos` is a multi-GPU system with two 20-core Intel CPUs and 1.5 TB of DRAM. The host system has four attached GPUs that are connected via PCI-E 3.0 to the host system. The attached GPUs are NVIDIA Tesla V100 [174] with 80 SMs each. The GPUs have 32 GB of HBM2 on-chip memory with a bandwidth of $900\frac{GB}{s}$. The GPUs are connected via NVLink 2.0 [60] to each other.

Figure 6.1 (see p. 125) provides an overview of the `Delos` system's interconnect topology. The two CPUs are connected via Intel Ultra Path Interconnect (UPI). Each CPU is connected via a PCI-E Switch to two of the GPUs. The GPUs are connected in a ring topology via NVLink 2.0, either using two lanes, e.g., between GPUs $0-1$, $0-2$, $2-3$, or using a single lane, e.g., between GPUs $1-3$. There is no direct link between GPUs $0-3$ and $1-2$. In our experiments, we refer to these three groups of GPUs as *direct two lanes*, *direct one lane*, and *indirect*, respectively. The theoretical one-directional bandwidth is 25 GB/s for one NVLink 2.0 lane and 16 GB/s for PCI-E version 3. The `Delos` operating system is Ubuntu version 18.4. The `CUDA` versions were 9.1, 11.2 or 11.3 depending on the time the experiments were performed.

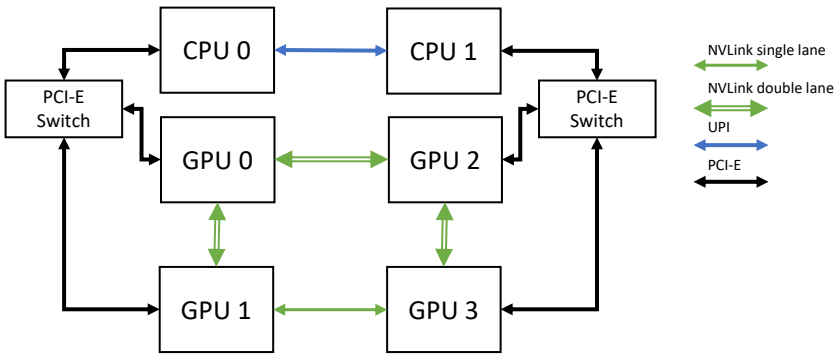


Fig. 6.1: Interconnect topology of the heterogeneous system `Delos` equipped with multi-core Intel CPUs and multiple NVIDIA GPUs. Blue connections refer to UPI, black connections refer to PCI-E 3.0, and green connections refer to NVLink 2.0, either single or double lanes.

Heterogeneous Single-GPU System: `A40`

`A40` is a single-GPU system with one 16-core AMD CPU and 96 GB of DRAM. The host system has one attached GPU that is connected via PCI-E 4.0. The attached GPU is an NVIDIA A40 [180] with 84 SMs. The GPU has 48 GB of GDDR6 on-chip memory with a bandwidth of $696 \frac{GB}{s}$. The operating system of `A40` is Ubuntu version 21.4 and `CUDA` version 11.4 was installed at the time experiments were performed.

6.1.3 Implementations from State-of-the-Art Libraries & Naive Baselines Used for Comparison

In the following, we describe the state-of-the-art libraries supporting constraint-based CSL, which are used in our experimental evaluation for comparison to our proposed GPU-accelerated algorithms. Furthermore, we detail naive baseline implementations and adaptations from existing libraries that we include for a more comprehensive experimental evaluation.

Overview of Implementations from State-of-the-Art Libraries

Table 6.3 (see p. 126) provides an overview of the state-of-the-art CSL libraries used in our evaluation. For each library, we report the function call used, the programming language, and the framework for parallel execution, as these attributes may impact the runtime performance. Furthermore, we state if the library employs GPU acceleration. We utilize four different libraries for comparison with our adjacency search algorithms. Under the assumption that the data follow the Gaussian distribution model, we compare with `pcalg` [111] and `cupc` [287]. If the observational data samples are discrete, we compare our algorithms to `bnlearn` [232] and `parallelPC` [126]. We choose the combination of libraries for comparison based on a previous runtime performance evaluation [99]. Although `bnlearn` and `parallelPC` support CI tests for the Gaussian distribution model, their runtime is slower than `pcalg`'s runtime. Similarly, `pcalg` can process discrete data samples, but the other two libraries are faster.

library	function called	programming language	framework for parallel execution	GPU acceleration
`pcalg` [111]	PC(...) with stable.fast	`R`, `C++`	`openMP` [40]	-
`parallelPC` [126]	PC_PARALLEL(...) with stable.fast	`R`	`parallel` [198]	-
`bnlearn` [232]	PC.STABLE(...)	`R`, `C`	`parallel` [198]	-
`cupc` [287]	CU_PC(...)	`CUDA`, `R`, `C`	`CUDA` [170]	yes
`tigramite` [214]	CMIKNN.RUN_TEST_RAW(...)	`python`	via `SciPy` [274]	-

Table 6.3: Overview of implementations from state-of-the-art libraries and study relevant characteristics. Libraries in the upper part are used for comparison with our adjacency search algorithms. `tigramite` is used for comparison with our GPU-accelerated information-theoretic CI test $\texttt{GPU}_{\texttt{CMIknn}}$.

Furthermore, we use the `CMIknn` CI test from the `tigramite` package [214] for comparison with our GPU-accelerated information-theoretic CI test $\texttt{GPU}_{\texttt{CMIknn}}$ and its application within the adjacency search in PC-stable.

pcalg

`pcalg` [111] is a well-known library that supports constraint-based CSL using the PC algorithm and its stable variant, PC-stable [35]. The library is written using the `R` framework [198] and supports efficient implementations of CI tests in `C++` using `Rcpp` [51]. For example, for data that follows the Gaussian distribution model, the implementation relies on `RcppArmadillo` [52] to leverage the efficient linear algebra implementation of the `Armadillo` library [221]. Further, the `C++` implementation uses `openMP` [40] for parallel execution of the adjacency search employing a static task distribution.

parallelPC

`parallelPC` [126] is a parallel variant of the PC-stable algorithm. The library is written entirely in `R` and extends the original `R`-based serial implementation of `pcalg`. The function PC_PARALLEL(...) in `parallelPC` uses the `R` library `parallel` for static distribution of tasks for parallel execution to the CPU cores.

bnlearn

`bnlearn` [232] is a library that supports score-based, constraint-based and hybrid methods for CSL. The library implements the PC-stable algorithm, which is executed by calling the function PC.STABLE(...). The library is called from the `R` environment but implements core functionality concerning CI tests in `C` for efficiency. To perform the CI tests on a system with multi-core CPUs the adjacency search is performed in parallel using the `R` library `parallel`. In this context, the tasks, i.e., edges, are statically distributed to the CPU cores.

cupc

To the best of our knowledge `cupc` [287] is the only GPU-accelerated library that supports constraint-based CSL. The library implements the PC-stable algorithm with a CI test for data that follows the Gaussian distribution model. The PC-stable algorithm is called from the `R` environment invoking the function CU_PC(...). Internally, the adjacency search is implemented in `C` and `CUDA` [170]. In particular, each level l of the adjacency search with the associated CI test is implemented in a separate `CUDA` kernel. Thus, for parallel execution on GPU hardware `cupc` builds upon the `CUDA` framework. Note that the code orchestrating the `CUDA` kernels is written in `C`.

tigramite

`tigramite` [214] is a `python` library for causal inference and causal discovery in time series data. The library implements the information-theoretic CI test `CMIknn` [214], which is called using the function CMIKNN.RUN_TEST_RAW(...). The CI test uses a KD-tree implementation from the `SciPy` library [274] to perform k-Nearest Neighbor (k-NN) searches. The `SciPy` library allows for parallel execution of the k-NN search queries on a multi-core CPU.

Naive Baseline Implementations

Implementations from the selected state-of-the-art libraries cover many cases targeted by our proposed algorithms. Nevertheless, in two experiments, we include naive baseline implementations. These implementations build upon the libraries mentioned above.

In particular, we add one GPU baseline implementation discrete data. This GPU baseline uses the same parallel execution strategy proposed in the `cupc-E` algorithm. Thus, each GPU thread processes an individual CI test. To realize this baseline, we adapt the `cupc-E` algorithm from the `cupc` library [287] and replace its CI test with the version of the Pearson χ^2 CI test implemented in our GPU-accelerated algorithm. In the experiments, we denote this implementation by `disc-cupc`.

In the second case, we add a `python`-based parallel adjacency search using the `CMIknn` [214] CI test that targets the CPU as an execution device. We use a `python`-based parallel adjacency search to have a single entry point for runtime comparison of the CPU-based `CMIknn` [214] CI test and our proposed GPU-accelerated algorithm. Thus, we incorporate our GPU-accelerated algorithm into the same `python`-based adjacency search. In our GPU-accelerated algorithn, we use a single CPU thread and realize the parallel execution within `CUDA`-based GPU kernels. The `CUDA`-based GPU kernels are integrated into the `python` code via `NumPy`'s C-API [84].

6.2 Experiments on GPU-Accelerated CSL Using a Single GPU

This section covers the experiments to address our first research question (RQ1). Thus, the measurements of our experimental evaluation examine the runtime performance of our GPU-accelerated CSL algorithms that execute on a single GPU, as described in Chapter 4. First, Section 6.2.1 provides measurement results for our proposed algorithms targeting the Gaussian distribution model (see Section 4.2). Next, Section 6.2.2 contains the experiment results concerning our introduced algorithm for discrete data (see Section 4.3). Section 6.2.3 describes the measurement results for our GPU-accelerated information-theoretic CI test, $\texttt{GPU}_{\texttt{CMIknn}}$ (see Section 4.4.2). Finally, Section 6.2.4 details the experiment results for $\texttt{GPU}_{\texttt{CMIknn}}$-Parallel, our proposed GPU-accelerated adjacency search building upon $\texttt{GPU}_{\texttt{CMIknn}}$ (see Section 4.4.3). *Parts of this section have been published in four research papers [16, 81, 83, 226].*

6.2.1 Experiments for a GPU-Accelerated Adjacency Search Assuming Data That Follows the Gaussian Distribution Model

This section describes the experiments for our proposed GPU-accelerated adjacency search algorithm, assuming that data follows the Gaussian distribution model. According to observations when performing the adjacency search of the PC algorithm on several real-world gene expression data, we argue that most CI tests are performed in levels $l = 0, 1$ (see Section 4.2.1 and [226]).

Therefore, our first measurements focus on these two levels only. In particular, we investigate the scalability of the corresponding GPU kernel implementations concerning an increasing number of CI tests using synthetic data. Further, we examine the runtimes on multiple real-world gene expression datasets, which we compare to a CPU baseline. Afterward, we present measurements on the proposed CUDA-X library-based variant to process levels $l \geq 2$. We focus on strategies to achieve high GPU utilization. Lastly, we elucidate measurements on the end-to-end performance on real-world gene expression datasets utilizing the GPU kernel-based variant for higher levels $l \geq 2$. In that experiment, we utilize the publicly available version of our GPU-accelerated algorithm implemented in our python package gpucsl[1] [16].

Experiment on Scalability using Synthetic Data for Levels $l = 0, 1$

In the first experiment, we examine the scalability of our CUDA-based implementations of the GPU kernels for levels $l = 0, 1$ with an increasing number of CI tests. With the experiment, we aim to determine if the selected parallelization strategies work for low- and high-dimensional datasets.

For the experiment, we generate synthetic data following the Gaussian distribution model as described in Section 6.1.1. We generate datasets with the number of variables N ranging from 10 to 25 000. We chose 25 000 as the upper limit, as datasets with a higher number of variables would exceed the GPU's memory capacity of the Titan X system (see Section 6.1.2 for detail) utilized in this experiment. Note that we assume a fully connected CGM; thus we can

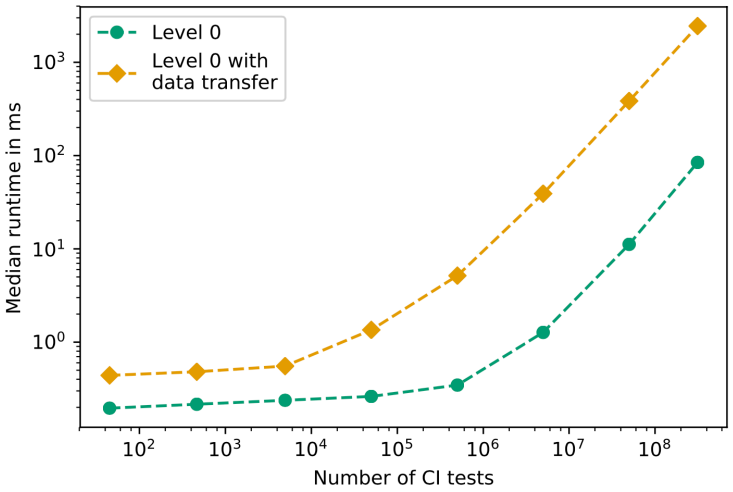
[1] https://github.com/hpi-epic/gpucsl

use the number of variables as a proxy for the number of CI tests that must be performed. We measure the GPU kernel execution time including and excluding data transfers. Hence, to measure the GPU kernel execution time, we take the current time prior to the GPU kernel launch and compute the difference to the time after a call of the function cudaDeviceSynchronize(...). For the measurements with data transfer, we include the time spent for data transfers, i.e., via a call of the function cudaMemcpy(...) but also add the times spent for GPU global memory allocation, i.e., via a call of the function cudaMalloc(...) and the freeing of GPU global memory by calling the function cudaFree(...). We use the `C++ chrono` library[2] for time measurements. The same setup is used for the measurements of both GPU kernels.

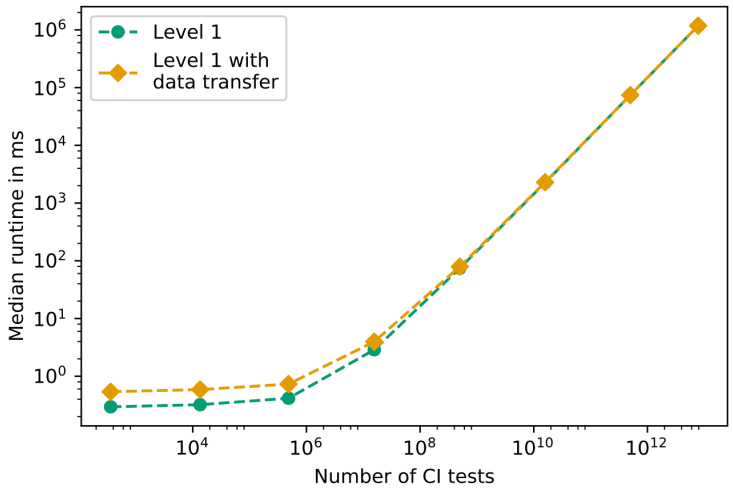
For the GPU kernel for level $l = 0$, a dataset with $N = 10$ results in performing 45 CI tests, and a dataset with $N = 25\,000$ requires 312 487 500 CI tests to be computed. In Figure 6.2a (see p. 130), we present the measured runtimes of the GPU kernel for level $l = 0$, excluding and including the data transfers between the host and GPU. In this isolated setting, we observe that the measured runtime, including data transfer, is up to an order of magnitude higher than the runtime of the GPU kernel without data transfer. The data transfers dominate the runtime, which poses a significant slowdown if only level $l = 0$ is processed. However, in the context of the adjacency search within PC-stable, the data structures transferred to GPU global memory can be reused by the GPU kernel for level $l = 1$ and in higher levels $l \geq 2$, too. In addition, we observe that our parallel `CUDA`-based implementation scales linearly concerning the number of CI tests for high-dimensional datasets. For low-dimensional datasets that require fewer than 100 000 CI tests, see the fourth data point in Figure 6.2a, we discover negligible differences in the runtimes. An in-depth investigation of the GPU kernel execution with the `nvprof` profiling tool [175] reveals that for the low-dimensional datasets, the SM utilization remains low. The corresponding performance metric `sm_efficiency` obtained from `nvprof` ranges from 15% for ten variables, see the first data point in Figure 6.2a, to 70% when slightly less than 100 000 CI tests are required. For the remaining high-dimensional datasets, which require more than 100 000 CI tests, the `sm_efficiency` reaches over 90%.

For the GPU kernel for level $l = 1$, the number of CI tests increases drastically. We assume the worst case, i.e., a fully connected CGM. Thus all edges are present in the current skeleton graph $\mathcal{C}$, and no edge is removed after any CI test. We construct this case by setting the significance level $\alpha = 1.0$. Hence, for a dataset with $N = 10$, the GPU kernel computes 360 CI tests, and for the dataset, with $N = 25\,000$, the GPU kernel performs approximately $7.8 \cdot 10^{12}$ CI tests. Figure 6.2b (see p. 130) presents the measured runtimes for the GPU kernel for level $l = 1$, again showing the pure GPU kernel execution and the runtime, including data transfer. Similar to the GPU kernel in level $l = 0$, we observe a linear increase in the runtime with an increase in the number of performed CI tests. In contrast to the GPU kernel in level $l = 0$, the data transfer overhead is small for low-dimensional datasets and neglectable for high-dimensional datasets. The `nvprof` profiling tool [175] reveals a low `sm_efficiency` for the first two data points and a value of close to 100% for any dataset with $N \geq 100$. In the case of datasets with $N \geq 100$, 485 100 CI tests are performed, which

[2] http://en.cppreference.com/w/cpp/header/chrono



(a) Median runtimes (10 runs) in milliseconds of the `CUDA`-based GPU kernel for level $l = 0$ with and without data transfer.



(b) Median runtimes (10 runs) in milliseconds of the `CUDA`-based GPU kernel for level $l = 1$ with and without data transfer.

Fig. 6.2: Runtimes in milliseconds of the `CUDA`-based implementations of the GPU kernels for levels $l = 0$ **(a)** and $l = 1$ **(b)** with an increasing number of CI tests. Note that both axes are in log scale.

saturates the available SMs. In contrast, lower-dimensional datasets expose an insufficient number of tasks for parallel execution to utilize all available SMs.

Considering both experiments, we find that the GPU of the `Titan X` system can process up to approximately 500 000 CI tests in parallel without significantly increasing the runtime. Once the GPU's SMs are fully utilized, the runtime increases linearly with an increasing number of CI tests.

Experiment on the Application to Real-World Gene Expression Data for Levels $l = 0, 1$

With the following experiment, we aim to investigate the performance gain when performing the CI tests in levels $l = 0, 1$ on the GPU over the execution on the CPU. Therefore, we measure the runtimes of the adjacency search up to level $l = 1$ on selected real-world gene expression datasets for our GPU-accelerated algorithm and a CPU-based implementation from a state-of-the-art library.

The gene expression datasets used in this experiment are `BR51`, `MCC`, `NCI-60`, `S.AUREUS`, and `S.CEREVISIAE` (see Section 6.1.1 for detail). For the CPU-based implementation, we choose `pcalg` [111] (version 2.5). The choice of `pcalg` has two reasons. First, `pcalg` is written using the `R` framework [198], a widely accepted environment for statistical computation providing a rich toolset and allowing integration of other languages to execute performance-critical parts of the code, for example, in `C++`. Second, `pcalg` leverages the language integration and implements the performance-critical adjacency search for data that follows the Gaussian distribution model in `C++`. Note that at the time of experiment execution, we made the following observation when running the adjacency search of `pcalg` in parallel on a multi-core CPU. In level $l = 0$, we found that running the adjacency search on multiple cores leads to faster runtimes than serial execution. In contrast, in level $l = 1$ and higher, we observed similar or slower runtimes when running the adjacency search on multiple cores compared to serial execution. We assume that synchronization overhead accounts for the slowdown. As a result of this observation, we use the parallel version of the

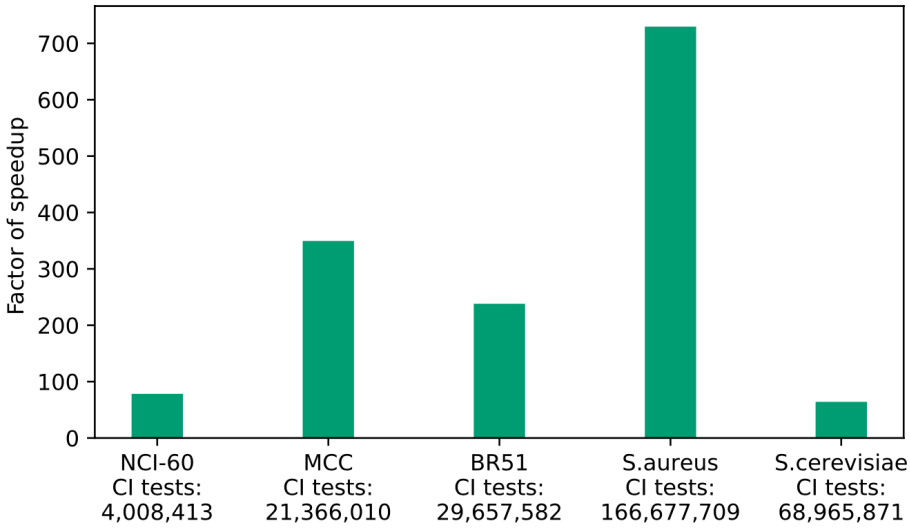


Fig. 6.3: Factors of speedup of the adjacency search up to level $l = 1$ when performing all CI tests on the GPU using the GPU kernels for levels $l = 0, 1$ compared to executing the adjacency search up to level $l = 1$ on the CPU using the implementation from `pcalg` (version 2.5). Note that we include data transfer between the host and GPU and that `pcalg` runs on four CPU cores in level $l = 0$ and single-threaded mode in level $l = 1$.

adjacency search provided by `pcalg` only in level $l = 0$. Note, we also considered `parallelPC` for comparison but experienced slower runtimes compared to `pcalg`, despite parallel execution in level $l = 1$, which we account for the more efficient `C++` implementation of `pcalg`. The runtime measurements are executed on the `Titan X` system (see Section 6.1.2 for detail) and include data transfers between the host and GPU. Further, we set the significance level to $\alpha = 0.01$.

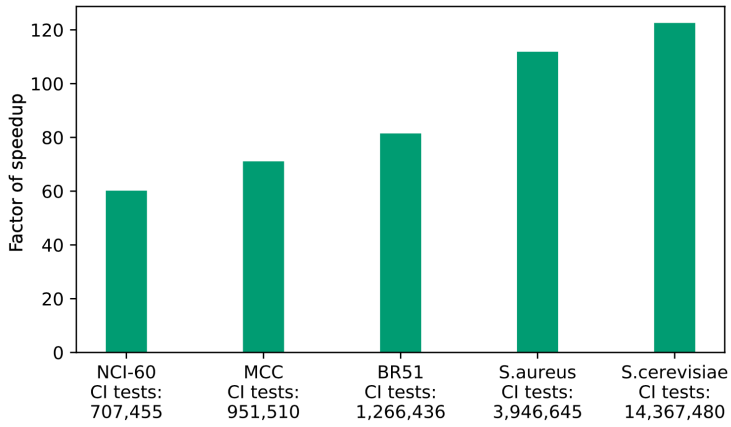
Figure 6.3 (see p.131) depicts the factors of speedup of our GPU-accelerated adjacency search over the CPU-based adjacency search of `pcalg` when executed up to level $l = 1$. Further, the figure presents the number of performed CI tests for both levels $l = 0, 1$. We obtain this number from the CPU-based algorithm, which stops iterating through the set of possible separation sets for an edge once one CI test signals the independence of that particular edge. Note that for the GPU-based adjacency search, the number of performed CI tests can be higher in level $l = 1$ as CI tests are performed in parallel by warps on the GPU.

We observe that our GPU-accelerated adjacency search outperforms the CPU-based algorithm for all datasets. The factors of speedup range from 64× for the `S.CEREVISIAE` dataset to 729× for the `S.AUREUS` dataset. It is noteworthy that a larger number of performed CI tests does not directly result in higher factors of speedup. For example, consider the two datasets, `MCC` and `BR51`, which have similar numbers of variables (`MCC`: $N = 1\,380$, `BR`: $N = 1\,592$). Roughly 8 million more CI tests are performed to process the dataset `BR51`, but a higher speedup is achieved for the dataset `MCC`. Even more CI tests are performed for the dataset `S.CEREVISIAE`, yet the factor of speedup achieved is smaller compared to any other dataset. In contrast, for the dataset `S.AUREUS`, the largest factor of speedup is achieved, and most CI tests are processed. Neither more variables nor more data samples are directly related to higher factors of speedup. To understand this matter further, we look at the speedup for each level individually in Figure 6.4 (see p. 133). Note that we exclude the data transfers between the host and GPU in both cases.

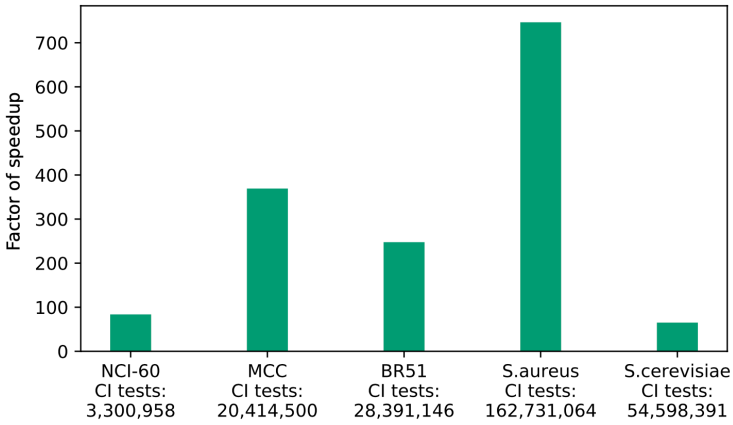
Figure 6.4a (see p. 133) focuses on the factors of speedup achieved in level $l = 0$. As expected, we observe that the number of conducted CI tests increases quadratically to the number of variables N. Note that in level $l = 0$, the number of performed CI tests is the same for the CPU-based and GPU-based adjacency searches. Furthermore, we observe that the factor of speedup increases as the number of performed CI tests increases. This observation supports our idea of GPU acceleration for constraint-based CSL for high-dimensional data. However, level $l = 0$ only accounts for a fraction of the overall runtime (see Table 4.3).

Figure 6.4b (see p. 133) shows the factors of speedup achieved in level $l = 1$, which accounts for most of the overall runtime (see Table 4.3). In level $l = 1$, we observe similar factors of speedup as presented in Figure 6.3, as the impact of level $l = 0$ and data transfer are small. Profiling the GPU execution with `nvprof` [175] did not reveal significant differences in the GPU kernel execution across the different datasets. Thus, we assume that the speedup factor achieved in level $l = 1$, in addition to the number of variables, respectively CI tests, is also influenced by the structure of the underlying true DAG of the CGM and the parallel execution strategy used in our GPU-accelerated algorithm.

In particular, the GPU kernel for level $l = 1$ uses GPU kernel launch parameters independent of the current version of the skeleton $\mathcal{C}$ obtained after the execution of level $l = 0$. As a result, the GPU kernel is launched with unnecessary thread blocks as the corresponding edges have already been removed in



(a) Factors of speedup for level $l = 0$.



(b) Factors of speedup for level $l = 1$.

Fig. 6.4: Factors of speedup per level for levels $l = 0$ **(a)** and $l = 1$ **(b)** in the case that all CI tests are performed on the GPU using the GPU kernel for the respective level compared to using the CPU-based implementation of `pcalg`. In level $l = 0$, `pcalg` uses four CPU cores, whereas, in level $l = 1$, `pcalg` runs serially. Note that the speedups do not account for data transfers between the host and GPU.

the previous level $l = 0$. Furthermore, our parallel execution strategy for the GPU kernel in level $l = 1$ can lead to performing unnecessary CI tests. For example, consider the case that the first CI test for an edge signals independence due to the underlying CGM. The CPU-based implementation, which processes CI tests of an edge sequentially, stops after this first CI test. In contrast, in our GPU-based implementation, the GPU threads of one thread block perform the CI tests of an edge in parallel. Hence, our algorithm performs at least as many CI tests as GPU threads are launched per thread block before edge removal.

We think that this overhead in our GPU-based implementation explains the lower factors of speedup for datasets with a higher number of CI tests and a

number of CI tests fused into one operation	1	2	8	64	512	4 096
`sm_efficiency`	5.6%	7.7%	16.4%	26.8%	77.3%	94.8%
`warp_efficiency`	32.7%	34%	39.4%	47.2%	88.9%	96.6%
`achieved_occupancy`	4.2%	4.3%	4.7%	13.6%	59%	84%

Table 6.4: `nvprof` metrics measured for the `CUDA`-X library-based variant for level $l = 2$ using varying numbers of fused CI tests. Generally, higher percentages are better.

larger number of variables, e.g., compare runtimes for `BR51` with `MCC`, or compare runtimes for `S.CEREVISIAE` to any other dataset. Nevertheless, in high-dimensional real-world datasets, our GPU-accelerated adjacency search with the GPU kernels for levels $l = 0, 1$ assuming that the data follows the Gaussian distribution model provides a large speedup compared to a commonly used CPU-based implementation.

Experiment on `CUDA`-X Library-Based Variant for Levels $l \geq 2$

With the following experiment, we aim to determine if our two proposed strategies to achieve a higher GPU utilization for our `CUDA`-X library-based variant for levels $l \geq 2$ result in runtimes that are comparable to runtimes of CPU-based algorithms. Synthetic data is used in the experiment, and the measurements are taken on the `Titan X` system (see Section 6.1.2 for detail).

We first consider the strategy to perform multiple CI tests concurrently in separate `CUDA` streams. For the measurements, the algorithm performed 4 096 CI tests with a separation set size of two, i.e., CI tests performed in level $l = 2$ of the adjacency search of PC-stable. Further, we varied the number of `CUDA` streams from 1 to 32, and the algorithm launched a dedicated CPU thread for each `CUDA` stream. In the case of one `CUDA` stream, the algorithm launches no additional CPU thread, but the current CPU thread handles the processing of the CI tests in the default `CUDA` stream. We observed an increase in the overall runtime when the number of `CUDA` streams increased. Using the `nvprof` profiler, we verified that the GPU kernels are performed concurrently but found that the runtime of the individual GPU kernels increased with more `CUDA` streams. Thus, the strategy building on `CUDA` streams for efficient execution of the `CUDA`-X library-based variant in levels $l \geq 2$ is not suited to achieve faster runtimes.

Second, we consider the strategy to fuse the computations for multiple CI tests into the same GPU operations. In Table 6.4, we present the profiling results using `nvprof` [175] for executions with different numbers of fused CI tests. We focus on the following metrics `sm_efficiency`, `warp_efficiency`, and `achieved_occupancy`[3], as they are indicators for the GPU utilization. Since performing a single CI test in levels $l \geq 2$ invokes multiple different GPU kernels, we report the metrics weighted according to the separate GPU kernel runtimes. Further note that a number of fused CI tests equal to one represents an execution without fusion. The profiling results show that the strategy achieves higher

[3] For detail see `https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference-7x`

number of CI tests fused into one operation	1	2	4	8	**16**	**32**	**64**	**128**	256	512	1 024	2 048	4 096
average runtime in s	11.5	9.6	7.7	7.2	**6.9**	**6.5**	**6.6**	**6.7**	9.9	22.6	74.3	291	1 149

Table 6.5: Average runtime of ten runs in seconds while processing 4 096 CI tests varying the number of CI tests fused into one operation. Fastest runtimes, i.e., below seven seconds, are highlighted in bold.

GPU utilization by fusing multiple CI tests. For example, the `sm_efficiency` reaches 77.3% when 512 CI tests are fused and processed within the same GPU kernel operations. For 4 096 fused CI tests, the SMs of the GPU are almost fully saturated. Generally, higher GPU utilization should also yield faster runtimes.

To verify if the higher GPU utilization yields faster runtimes, we measure the runtime of the fusion-based strategy when processing a total of 4 096 CI tests, assuming a separation set size of two, i.e., performing CI tests in level $l = 2$. The average runtimes of 10 runs are shown in Table 6.5. The fastest runtime is achieved by fusing 32 CI tests into a single operation. In that case, a speedup of a factor of 1.77× is achieved compared to performing the CI tests without fusion. Similar runtimes are achieved when fusing between 16 to 128 CI tests into one operation. Once 512 or more CI tests are fused into one operation, we observe an increase in runtime compared to sequentially processing the 4 096 CI tests. We observe that using a single GPU operation with 4 096 fused CI tests is approximately two orders of magnitude slower than processing the CI tests without fusion. We attribute this slowdown to a large increase in memory footprint. For example, for 32 fused CI tests in level $l = 2$, the algorithm creates the auxiliary data matrix $tCor_{fused}$ of size 128×128, requiring several kBs of GPU global memory. For 512 fused CI tests, the algorithm creates the auxiliary data matrix $tCor_{fused}$ of size $2\,048 \times 2\,048$, which already requires several MBs of GPU global memory. Lastly, in the case of 4 096 fused CI tests, the auxiliary data matrix requires hundreds of MBs of GPU global memory. As a result of a larger auxiliary data matrix, data transfer times are increased, and GPU caches cannot be used as efficiently as with small-sized matrices. Although the profiling metrics show the highest GPU utilization for large numbers of fused CI tests, a speedup is only achieved if fusing is limited to a few CI tests. Overall, fusing multiple CI tests achieves speedup below a factor of 2× compared to no fusing.

Despite this slight improvement in runtimes, the overall runtime of the `CUDA`-X library-based variant for levels $l \geq 2$ remains slower than the serial CPU-based implementation from `pcalg` (version 2.5) by more than two orders of magnitude. Thus, we conclude that using `CUDA`-X libraries, as proposed in Algorithm 6 (see Section 4.2.5), to perform many CI tests on the GPU is not suited to outperform CPU-based algorithms.

Experiment on End-to-End Performance of the PC Algorithm using Real-World Gene Expression Datasets

In this experiment, we focus on the end-to-end performance of the entire PC algorithm for datasets following the Gaussian distribution model. Therefore, we compare our GPU-accelerated adjacency search using the GPU-kernel-based variant for levels $l \geq 2$, as implemented in our `python`-based library `gpucsl` [16] to `pcalg` [111] (version 2.7.5) executed on the CPU, and the GPU-accelerated PC algorithm from `cupc` [287]. For each of the six datasets following the Gaussian distribution model, namely, `BR51`, `DREAM5-INSILICO`, `MCC`, `NCI-60`, `S.AUREUS`, and `S.CEREVISIAE` (see Section 6.1.1 for detail), we report the median end-to-end runtime from 10 runs and the runtimes for subparts of the algorithm. The subparts are the adjacency search, the edge orientation, the GPU kernel execution, and the GPU kernel compilation. Note that the time of the GPU kernel execution is included within the time spent for the adjacency search. Further, the GPU kernel compilation time is only measured for `gpucsl`. The `gpucsl` library builds upon `cupy` [182] to compile the GPU kernels for each level during runtime as required. This compilation time is included in the reported end-to-end time. In contrast, the GPU kernels in `cupc` are compiled during the compilation of the library and thus not reported. All experiment runs are performed on the `Delos` system (see Section 6.1.2 for detail) with `CUDA` version 11.2, and the significance level is set to $\alpha = 0.05$.

The measurement results are shown in Table 6.6 (see p. 137). In an end-to-end comparison, `gpucsl` has an average speedup of factor 9.5× over `pcalg` and outperforms `cupc` on average by a factor of 4.22×. For the `DREAM5-INSILICO` dataset, `cupc` is faster than `gpucsl` concerning the end-to-end performance by a factor of 2.94× and also faster in the GPU kernel execution by a factor of 4.54×. In the case of the `DREAM5-INSILICO` dataset, higher levels l are reached, and more CI tests are performed in these higher levels. Thus, `gpucsl` spends more time on the GPU kernel compilation. Furthermore, the `cupc-S` algorithm allows for sharing intermediate results while performing higher-order CI tests. Thus, it outperforms `gpucsl`. More detailed runtime analysis reveals that `gpucsl` spends most of its time on GPU kernel compilation. The adjacency search makes up between 7.65% to 33.3%, and from 4.2% to 14.9% is spent on the edge orientation. The actual GPU kernel execution dominates the adjacency search time for large datasets, e.g., `S.AUREUS`, `S.CEREVISIAE`, or `DREAM5-INSILICO`. To mitigate the GPU kernel compilation `gpucsl` offers an option to provide cached GPU kernels, which we did not consider in these measurements. In contrast, `cupc` has no on-the-fly GPU kernel compilation. Thus, the adjacency search constitutes 56.1% to 87.9% of the end-to-end runtime. The remaining 12.1% to 43.9% of the runtime is spent on the edge orientation. The actual GPU kernel execution makes up between 7.7% to 19.9% of the adjacency search. Thus, CPU-based data manipulation majorly impacts the end-to-end performance of `cupc`. As `cupc` shares the edge orientation implementation of `pcalg`, the higher end-to-end runtimes of `pcalg` stem from the adjacency search, despite being executed in parallel on 40 CPU cores.

dataset	library	end-to-end	adjacency search	edge orientation	GPU kernel execution	GPU kernel compilation
NCI-60	gpucsl	**1.83** s	**0.14** s	**0.09** s	**0.03** s	1.6 s
	cupc	2.51 s	2.05 s	0.46 s	0.37 s	N/A
	pcalg	2.75 s	2.3 s	0.45 s	N/A	N/A
MCC	gpucsl	**2.17** s	**0.17** s	**0.11** s	**0.05** s	1.89 s
	cupc	3.29 s	2.64 s	0.65 s	0.4 s	N/A
	pcalg	4.8 s	4.15 s	0.65 s	N/A	N/A
BR51	gpucsl	**1.91** s	**0.19** s	**0.08** s	**0.05** s	1.64 s
	cupc	6.51 s	5.72 s	0.79 s	0.44 s	N/A
	pcalg	6.23 s	5.45 s	0.78 s	N/A	N/A
S.AUREUS	gpucsl	**2.82** s	**0.47** s	**0.42** s	**0.26** s	1.93 s
	cupc	15.67 s	11.10 s	4.57 s	1.17 s	N/A
	pcalg	35.54 s	30.97 s	4.57 s	N/A	N/A
S.CEREVISIAE	gpucsl	**3.0** s	**1.0** s	**0.23** s	**0.51** s	1.77 s
	cupc	39.46 s	26.42 s	13.04 s	5.27 s	N/A
	pcalg	60.16 s	47.13 s	13.03 s	N/A	N/A
DREAM5-INSILICO	gpucsl	18.53 s	**3.36** s	**0.98** s	3.18 s	14.19 s
	cupc	**6.31** s	3.54 s	2.77 s	**0.70** s	N/A
	pcalg	316.60 s	313.84 s	2.76 s	N/A	N/A

Table 6.6: Runtimes in seconds measured on datasets following the Gaussian distribution model: End-to-end measurements show median wall clock times of 10 runs. Breakdowns of the measurements into adjacency search, edge orientation, and GPU kernel compilation are included. The GPU kernel execution time is included within the adjacency search time. For each dataset, the fastest runtimes are highlighted in bold. The CPU-based implementation from `pcalg` (version 2.7.5) is executed on 40 CPU cores. GPU-specific measures do not apply (N/A) to `pcalg`. Note that `gpucsl` is our proposed algorithm.

Summary

Our first measurements concerning GPU acceleration for constraint-based CSL under the assumption of data following the Gaussian distribution model focus on levels $l = 0, 1$ of the adjacency search within PC-stable. The results indicate that execution of the adjacency search is a promising option to reduce the long runtimes of a CPU-based adjacency search. Concerning higher levels $l \geq 2$, our experiments reveal that using available `CUDA-X` libraries, as suggested in Section 4.2.5, results in significantly slower runtimes than serial execution on the CPU. Thus, custom GPU kernels are required, as introduced in Section 4.2.5 and in `cupc` [287]. In an end-to-end evaluation on high-dimensional gene expression datasets, we find that the GPU-accelerated adjacency search, as implemented in our library `gpucsl`, is between 1.5 to 20 times faster than the CPU-based implementation found in `pcalg` [111] executed on 40 CPU cores. More speedup is possible by incorporating an optimization of `cupc-S` to share intermediate results, e.g., see the GPU kernel execution times for dataset `DREAM5-INSILICO` in Table 6.6. Further, we could include caching of compiled GPU kernels within `gpucsl` to eliminate the compilation overhead at runtime.

In conclusion, assuming that data follows the Gaussian distribution model, GPU acceleration is suited to address the long runtimes of the PC algorithm for high-dimensional datasets when executed on multi-core CPU systems.

6.2.2 Experiments for a GPU-Accelerated Adjacency Search Assuming Discrete Data

This section describes the experiments for our proposed GPU-accelerated adjacency search algorithm for discrete data. At first, we examine the scalability of our algorithm concerning selected characteristics of CGMs with discrete data. We compare our algorithm with a baseline implementation. Second, we present adjacency search runtime measurements on benchmark Bayesian networks. We compare our GPU-based algorithm to a GPU baseline and state-of-the-art parallel implementations for multi-core CPUs. Lastly, we compare the end-to-end runtime performance of our proposed GPU-accelerated algorithm to the end-to-end runtime performance of a state-of-the-art parallel CPU-based algorithm.

Experiment on Scalability using Synthetic Data

In this experiment, we compare the runtime of our proposed GPU-accelerated adjacency search for discrete data to the runtime of the GPU baseline implementation `disc-cupc` (see Section 6.1.3).

The two implementations differ in their parallel execution strategy. In particular, our implementation, which we call `gpuPC` in the following, uses GPU threads within the same warp to jointly compute the marginals over contingency tables. Thus, CI tests are processed on a warp granularity. In contrast, `disc-cupc` applies the parallel execution strategy of `cupc-E`, such that $\gamma \times \beta$ GPU threads within a thread block process γ CI tests for β edges. As a result of the different parallel execution strategies, the two GPU-based algorithms handle the GPU global memory for the auxiliary data structures differently. While `gpuPC` allocates GPU global memory for the auxiliary data structures as described in Section 4.3, `disc-cupc` lets each GPU thread allocate GPU global memory for the auxiliary data structures inside the GPU kernel.

For the measurement runs, we set the parameters of `disc-cupc` following the suggestions of the original `cupc-E` algorithm, i.e., $\gamma = 32$ and $\beta = 2$ [287]. The parameters for `gpuPC` are set to $\delta = 64$, $\gamma = 2$ and $\beta = 1$, which were determined through micro experiments. Further, we set the significance level $\alpha = 0.01$, following common practice [35] and perform the measurements on the `Delos` system (see Section 6.1.2 for detail) with `CUDA` version 9.1. The measurements are performed on synthetic data, and each run is executed at least 10 times.

We report the median factor of speedup of `gpuPC` over the baseline `disc-cupc` in Table 6.7 (see p. 139). In particular, we investigate the development of the factor of speedup while scaling dimensions relevant to the PC-stable, considering the characteristics of a CI test for discrete data and its execution on a GPU. Therefore, we examine the number of variables N, the number of data samples n, the maximum size of the discrete domain $\max_{i=1,\dots,N}\{|\mathcal{V}_i|\}$, with $\mathcal{V}_i$ representing the corresponding discrete domain of the variable V_i with $i = 1, \dots, N$, the number of CI tests per edge, which we denote with CI_{edge}, and a decay factor *df*. The decay factor *df* describes the percentage of edges removed

dimension	value of dimension				
	factor of speedup				
N - number of	1 000	2 000	4 000	8 000	
variables	4.2×	4.38×	4.1×	2.97×	
$\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\}$ - maximum	2	4	8	10	20
size of the discrete domain	3.6×	7.67×	7.5×	6.4×	∞
n - number of	10 000	50 000	100 000	500 000	1 000 000
data samples	8.4×	29.1×	37.7×	45.5×	45.6×
CI_{edge} - number of	1	4	16	32	128
CI tests per edge	9.4×	9.7×	6.25×	4.25×	4.36×
df - decay factor	0.9	0.75	0.5		
	8.2×	12.0×	∞		

Table 6.7: Factors of speedup of `gpuPC` (our) over `disc-cupc`, based on median runtimes of 10 runs. For each experiment all dimensions except the one of interest are fixed, with the following values $\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\} = 4$, $N = 1\,000$, $n = 10\,000$, $CI_{edge} = 32$, $df = 0.9$. Note that ∞ indicates that `disc-cupc` terminated in error as the algorithm ran out of memory.

equally distributed from skeleton $\mathcal{C}$ within each level l of the adjacency search. We consider CI_{edge} and df as a proxy for the impact of edge deletions from the skeleton $\mathcal{C}$ to ensure the comparability of experiments across different levels l. Furthermore, varying CI_{edge} allows for examing how the implementations cope with load imbalance due to a different number of CI tests per edge [224].

In our experiment, we make the following observations concerning the different dimensions. In the dimension of N, which corresponds to a different number of conducted CI tests, i.e., larger N imply more CI tests, the performance difference remains fairly constant. As the number of variables N doubles, the runtimes quadruple, following the adjacency search's polynomial complexity [109].

In the dimension of $\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\}$, we know that larger values lead to higher memory demand for the auxiliary data structures. Consequently, the measured runtimes increase with larger values of $\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\}$. As the increase in runtime is observed for `gpuPC` and `disc-cupc`, the speedup remains similar, except for $\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\} = 2$, where we see a small speedup. For $\max_{i=1,\dots,N}\{\lvert\mathcal{V}_i\rvert\} > 10$, `disc-cupc` fails to execute, as it exceeds the available GPU global memory while `gpuPC` continues to operate. However, the measured runtimes of `gpuPC` degrade; due to a limited degree of parallel execution and poorer caching behavior caused by massive auxiliary data structures.

In the dimension of n, the difference in the parallel execution strategies yields the most extensive performance gap between the two algorithms. `gpuPC` is up to 45 times faster than the `disc-cupc` for $n \geq 500\,000$, while it is only 8.4 times faster for $n = 10\,000$. The measured runtimes for `gpuPC` increase linear with n, and we observe that the slope of the linear function between the measured runtime and n is smaller for $n < 500\,000$ compared to $n \geq 500\,000$.

In the dimension of CI_{edge}, which reflects settings of load imbalance, the speedup is fairly constant for $CI_{edge} \geq 32$. For $CI_{edge} < 32$, the speedup is higher for smaller values of CI_{edge}. The different behavior concerning CI_{edge} results from the fine-granular execution strategy of `gpuPC`, which handles load imbalance in the discrete case better than the parallel execution strategy of `disc-cupc`, respectively, `cupc-E` [287]. In particular, in `disc-cupc`, at least 32 CI tests per edge are conducted in parallel due to the chosen value of γ and the GPU's warp size of 32, which explains the overhead for $CI_{edge} < 32$.

In the dimension of *df*, a small decay factor *df* indicates more remaining edges after each level, i.e., a dense underlying CGM. Hence, more CI tests are performed in each level. Also, higher levels may be reached with a small decay factor *df*, which leads to larger separation sets with higher memory demand for the auxiliary data structures. Therefore, `disc-cupc` fails for $df = 0.5$. Further, we observe that the performance gap increases with smaller *df*, as the parallel execution strategy in `gpuPC` handles larger numbers of CI tests and higher-order CI tests better than the parallel execution strategy in `disc-cupc`.

In summary, we observe that `gpuPC` is faster than the baseline `disc-cupc` in all considered dimensions relevant for the GPU-accelerated adjacency search in PC-stable under the assumption of discrete data. Particularly concerning the number of data samples n, `gpuPC` is over an order of magnitude faster. Furthermore, `gpuPC` scales for datasets with high memory demand for auxiliary data structures, while `disc-cupc` runs out of GPU global memory and terminates with an error. However, to handle datasets with a high memory demand for auxiliary data structures, e.g., with $\max_{i=1,\ldots,N}\{|\mathcal{V}_i|\} = 20$, `gpuPC` restricts the degree of parallelism, resulting in slower runtimes. Note that the mechanism to restrict the degree of parallelism is triggered when GPU global memory becomes scarce. Thus, the occurrence of this effect depends on dataset characteristics and the GPU hardware, i.e., the GPU's memory capacity.

Experiment on the Application to Benchmark Bayesian Networks

In the following experiment, we compare the runtimes of the adjacency search on four benchmark Bayesian networks, `ALARM` (with $n = 200\,000$), `ANDES`, `LINK`, and `MUNIN` (see Section 6.1.1 for detail). We compare our algorithm, `gpuPC`, to the GPU baseline `disc-cupc` and to two state-of-the-art CPU-based libraries that allow for parallel execution, `bnlearn`, and `parallelPC`.

For the two GPU-based implementations, we set the parameters as follows. For `disc-cupc`, we set $\gamma = 32$ and $\beta = 2$ following the suggestion of the original `cupc-E` algorithm [287]. For `gpuPC` we set $\delta = 64$, $\gamma = 2$ and $\beta = 1$. Further, we set the significance level $\alpha = 0.01$. The measurement runs are executed on the `Delos` system (see Section 6.1.2 for detail) with `CUDA` version 9.1. We set the number of CPU cores used by `bnlearn` and `parallelPC` to 40 corresponding to the number of available CPU cores. All measurements of the GPU-based algorithms include data transfer between the host and GPU. Further, all implementations are called from their `R` interface. Thus, measurements include times for data copies from `R` to `C` or `CUDA`, respectively.

We report median runtimes in seconds over 10 runs in Table 6.8 (see p. 141). Comparing the two CPU-based implementations, we observe a significant difference in runtimes. The implementation from `bnlearn` is faster by up to a factor of 400 compared to `parallelPC`. We assume that the difference in runtime results

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	110 740.50 s	273.79 s	97.45 s	14.99 s

Table 6.8: Median runtimes (over 10 runs) of the adjacency search in seconds for four discrete benchmark Bayesian networks. The adjacency searches in parallelPC and bnlearn run on 40 CPU cores. The adjacency searches of disc-cupc and gpuPC (our) are accelerated on the GPU.

from the efficient implementation in the C language in bnlearn and overhead in the implementation of parallelPC written entirely in R. Both GPU-accelerated adjacency searches are faster than the CPU-based adjacency searches. The GPU baseline disc-cupc is between 2.8 to 15.5 times faster than bnlearn. Our proposed GPU-accelerated adjacency search gpuPC has the fastest runtimes on all datasets and achieves speedup of factors of 18.3× to 62.1× over bnlearn and is up to 6.5 times faster than disc-cupc. Further, we do not find any difference in the observed pattern of the measured runtimes for small-dimensional, e.g., ALARM or ANDES, or high-dimensional data, e.g., LINK or MUNIN concerning the number of variables N. The same holds for datasets with small or large numbers of data samples n.

Experiment on End-to-End Performance of the PC Algorithm using Benchmark Bayesian Networks

In contrast to the previous experiments, which focus on the adjacency search, we now consider the end-to-end performance of the PC algorithm on discrete datasets. In this experiment, we use our GPU-accelerated adjacency search implementation from our python-based library gpucsl [16]. Note that the implementation uses the same GPU-kernel-based approach as used in gpuPC.

We use the Delos system (see Section 6.1.2 for detail) with CUDA version 11.2 to execute the measurement runs and the set significance level of the PC algorithm to $\alpha = 0.05$. We compare the end-to-end performance with the implementation from bnlearn in R executed in parallel on 40 CPU cores, according to the available CPU cores on the Delos system. Other implementations of the PC algorithm for discrete data, such as parallelPC or the python version of bnlearn, showed slower runtimes. Thus, they are not included in our comparison. We report the median end-to-end runtime from 10 runs and the runtime of subparts of the PC algorithm for three discrete datasets, namely, Alarm (with $n = 10\,000$), Link, and Munin (see Section 6.1.1 for detail). The subparts are the adjacency search, the edge orientation, the GPU kernel execution, and the GPU kernel compilation. Note that the time of the GPU kernel execution is included within the time spent for the adjacency search, and the GPU kernel compilation time is only measured for gpucsl. As gpucsl builds upon cupy [182] to compile the GPU kernel for each required level during runtime, we include the GPU kernel compilation time in the reported end-to-end time.

The measurement results are shown in Table 6.9 (see p. 142). Concerning the end-to-end runtime for the small-dimensional dataset Alarm, which has 37

dataset	library	end-to-end	adjacency search	edge orientation	GPU kernel execution	GPU kernel compilation
Alarm	gpucsl	2.59 s	**0.07** s	**0.01** s	0.01 s	2.51 s
	bnlearn	**2.51** s	1.74 s	0.77 s	N/A	N/A
Link	gpucsl	**9.16** s	**5.56** s	**0.09** s	4.83 s	3.51 s
	bnlearn	497.95 s	190.14 s	307.81 s	N/A	N/A
Munin	gpucsl	**114.68** s	**111.81** s	**0.09** s	111.06 s	2.78 s
	bnlearn	454.70 s	339.55 s	115.15 s	N/A	N/A

Table 6.9: Runtimes in seconds measured on discrete datasets: End-to-end measurements show median wall clock times of 10 runs. Breakdowns of the measurements into adjacency search, edge orientation, and GPU kernel compilation are included. The GPU kernel execution time is included within the adjacency search time. For each dataset, the fastest runtimes are highlighted in bold. The CPU-based implementation from `bnlearn` runs on 40 CPU cores. GPU-specific measures do not apply (N/A) to `bnlearn`. Note that `gpucsl` is our proposed algorithm.

variables, `bnlearn` is faster by approximately 3%. In contrast, for the two higher-dimensional datasets `Link` and `Munin`, `gpucsl` outperforms `bnlearn` by factors of 54.3× or 3.96×. For the `Alarm` dataset, `gpucsl` spends 96.9% of the runtime for the GPU kernel compilation, as the adjacency search and edge orientation require only 0.1 seconds combined. Otherwise, the GPU kernel compilation time accounts for 2.4% to 38.3% of the end-to-end runtime. For all datasets, the edge orientation constitutes less than 1% of the runtime of `gpucsl`. In contrast, for `bnlearn`, the edge orientation makes up 25.3% to 61.8% of the runtime. The `python`-based edge orientation is orders of magnitude faster than the R-based edge orientation of `bnlearn`. For the adjacency search, `gpucsl` is faster by factors ranging from 3.04× to 34.2× than `bnlearn`.

The GPU-accelerated implementation from our library `gpucsl` achieves speedup over parallel CPU-based implementations for high-dimensional datasets. For small dimensional datasets, `gpucsl`'s on-the-fly GPU kernel compilation becomes a performance bottleneck. The compiled GPU kernels could be cached to address long GPU kernel compilation times. Note that the measured runtimes of the adjacency search presented in Table 6.9 are higher than those presented in the previous experiment in Table 6.8 due to a larger significance level $\alpha = 0.05$ compared to $\alpha = 0.01$. As a result, the intermediate skeletons $\mathcal{C}^l$ in each level l remain denser, as fewer CI tests signal independence. Thus, more CI tests are performed in each level, leading to an increase in the runtime.

Summary

In our experimental evaluation of our proposed GPU-accelerated adjacency search for discrete data, we find that execution of the adjacency search on the GPU results in faster runtimes than execution on multi-core CPUs. For the adjacency search, we observe factors of speedup ranging from 3.04× to 62.1×. In end-to-end measurements of the PC-stable algorithm, using the GPU-accelerated adjacency search results in up to 54.3 times faster runtimes than using a parallel CPU-based adjacency search. Further, we show that the chosen parallel

execution strategy, i.e., using GPU threads to jointly compute the marginals over the contingency table for a CI test, outperforms existing parallel execution strategies, using each GPU thread to perform an entire CI test [287].

However, the impact of a change in the significance level α is larger on the runtime of the GPU-accelerated adjacency search than on the runtime of the CPU-based implementation from `bnlearn`. For example, for the dataset `MUNIN`, the measured runtime for the GPU-accelerated algorithm increases by a factor of 7.46 when $\alpha = 0.05$ versus $\alpha = 0.01$. In contrast, the runtime of the CPU-based implementation of `bnlearn` increases only by a factor of 1.24. We account for this effect to an increased density of the learned Completed Partially Directed Acyclic Graph (CPDAG) of the CGM due to the increase of α. An increased value of α potentially leads to more higher-order CI tests requiring larger amounts of GPU global memory for the auxiliary data structures. Auxiliary data structures with a higher footprint are less cache efficient, causing runtime performance degradation. Furthermore, in the last experiment for the dataset `MUNIN`, we restricted the maximum level reached in the adjacency search to $l = 3$. The restriction is needed, as our implementation of the GPU-accelerated adjacency search in `gpucsl` cannot automatically reduce the degree of parallelism if the auxiliary data structures exceed the GPU's memory capacity, as suggested in Paragraph "Additional Notes" of Section 4.3.3.

6.2.3 Experiments for an Information-Theoretic GPU-Based CI Test

This section describes the experiments for our proposed information-theoretic GPU-based CI test, $\texttt{GPU}_{\texttt{CMIknn}}$ (see Section 4.4.2). We focus our evaluation on continuous data with non-linear relationships as it allows for comparison to the CPU-based CI test `CMIknn` [214]. Therefore, we adjust the employed Conditional Mutual Information (CMI) estimator in $\texttt{GPU}_{\texttt{CMIknn}}$ and make adaptations accordingly (see "Outline of $\texttt{GPU}_{\texttt{CMIknn}}$" in Section 4.4.2 or our corresponding publication [83]). Thus, we utilize the same CMI estimator as `CMIknn`.

In the experiments, we particularly focus on the impact on the runtime when scaling one of several parameters relevant to the CI test. In detail, we consider the number of k_{CMI}-nearest neighbors, the number of data samples n, the number of permutations $perm$, and the size of the separation set $|S^{i,j}|$ with $i, j = 1, \ldots, N$ and $i \neq j$. We do not evaluate the runtime performance concerning changes in the number of k_{perm}, i.e., the number of k-NN during local permutation computation as k_{perm} does not impact runtime much [214]. If not stated differently, we chose the following default values for the parameters for the examined implementations of $\texttt{GPU}_{\texttt{CMIknn}}$ and `CMIknn`. We set $k_{perm} = 15$, which is slightly above the suggested range for `CMIknn` [214], and use $perm = 100$ to avoid excessive experiment runtimes. Note that the runtime increases linearly with the number of permutations $perm$. Also, we set $n = 1\,000$ and $|S^{i,j}| = 1$. Further, for $\texttt{GPU}_{\texttt{CMIknn}}$, we use the GPU kernel parameters $\beta = 32$ and $\gamma = 32$. Therefore, each GPU thread block is launched with a number of GPU threads filling an entire warp. At the same time, we keep the amount of shared memory required for each GPU thread block low. All measurements are performed on the `A40` system (see Section 6.1.2 for detail). When comparing the GPU-accelerated CI test $\texttt{GPU}_{\texttt{CMIknn}}$ to the CPU-based CI test `CMIknn`, the runtime measurements are influenced by two fundamental differences. First, the comparison focuses on executing on two different PUs with implementations tailored to the device,

i.e., running on a single CPU core in the case of `CMIknn` or performing the operations in parallel on a GPU. Second, the comparison considers two different k-NN estimation approaches. `CMIknn` uses a computationally efficient k-d tree search suited for execution on a CPU. In contrast, $\texttt{GPU}_{\texttt{CMIknn}}$ builds upon brute-force searches, which are computationally less efficient but better suited for the parallel execution model of the GPU.

Experiment on the Impact of the Value of k_{CMI} During CMI Estimation

According to Runge [214], the parameter for the k_{CMI}-nearest neighbors should be set to $k_{CMI} \approx \{0.1, \ldots, 0.2\} \times n$ to yield good statistical 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 if the data is placed in registers, which are highly restricted in size. Otherwise, performance degrades due to register spilling [162], as data structures within GPU thread-local memory are now placed within GPU global memory. Thus, we assume that the runtime performance of $\texttt{GPU}_{\texttt{CMIknn}}$ drops while k_{CMI} is increased.

method	k_{CMI} - number of nearest neighbors during CMI estimation									
	7	10	20	30	40	50	75	100	250	500
`CMIknn`	1.76 s	1.79 s	1.84 s	1.85 s	1.9 s	1.96 s	1.94 s	2.02 s	2.31 s	2.69 s
$\texttt{GPU}_{\texttt{CMIknn}}$	0.005 s	0.01 s	0.01 s	0.01 s	0.02 s	0.02 s	0.07 s	0.13 s	0.52 s	1.15 s

Table 6.10: Median runtimes in seconds over 20 CI tests, scaling k_{CMI} for CI tests with fixed parameters: $n = 1\,000$, $perm = 100$, $|S^{i,j}| = 1$.

Table 6.10 shows the median runtimes in seconds over the execution of 20 CI tests with $n = 1\,000$ data 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 k-d search trees to estimate the k-NN, 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 the k-NN, we see a runtime increase by a factor of 230×. Particularly for $k_{CMI} > 50$, the runtime performance of $\texttt{GPU}_{\texttt{CMIknn}}$ drops, which we account to register spilling. The observation confirms our assumption that the runtime performance of $\texttt{GPU}_{\texttt{CMIknn}}$ drops while k_{CMI} increases. Comparing the runtime of `CMIknn` and $\texttt{GPU}_{\texttt{CMIknn}}$, we find that for small values of k_{CMI}, e.g., up to $k_{CMI} = 50$, $\texttt{GPU}_{\texttt{CMIknn}}$ is up to a factor of 352 faster than `CMIknn` and remains faster by a factor of 2.3× even for $k_{CMI} = 500$. However, for these large values of k_{CMI}, one should note that $\texttt{GPU}_{\texttt{CMIknn}}$ operates in parallel while `CMIknn` runs on a single CPU core. As the parameter k_{CMI} significantly impacts the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$, we report measurements with several values of k_{CMI} in the following experiments.

Experiment on the Impact of the Number of Permutations *perm*

The number of permutations *perm* impacts the runtime of the local permutation computation and the CMI estimation. For both steps, the number of required compute operations increases linearly to the number of permutations *perm*. Within the $estimate_{CMIknn}$ GPU kernel, a larger number of permutations *perm* results in launching the GPU kernel with additional thread blocks. In contrast, the launch parameters for the *localPermutation* GPU kernel remain unaffected.

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

Table 6.11: Median runtimes in seconds over 20 CI tests, scaling *perm* for CI tests with fixed parameters: $n = 1\,000$, $|S^{i,j}| = 1$.

In Table 6.11, we report the median runtimes in seconds over the execution of 20 CI tests with $n = 1\,000$ data 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 observe an increase in the runtime from $perm = 50$ to $perm = 1\,000$, below a factor of 20×. For small values of k_{CMI}, i.e., $k_{CMI} = \{7, 20\}$, the runtime increases by a factor of up to 10×. For $k_{CMI} = 200$, the runtime increases by up to a factor of 18.25×. We assume that the slightly lower increase in runtime compared to `CMIknn` is due to better utilization of the parallel computing capabilities of the GPU, given that more GPU threads are launched during CMI estimation. However, for $k_{CMI} = 200$, the accesses to GPU global memory, due to register spilling, seem to dominate the performance. In that case, we observe a similar increase in runtime compared to `CMIknn`.

Experiment on the Impact of the Separation Set Size $|S^{i,j}|$

The size of the separation set $|S^{i,j}|$ directly increases the number of dimensions within the k-NN searches during local permutation computation and CMI estimation. Higher dimensions impact the runtime of the k-d tree approach and the brute-force approach. K-d trees generally suffer under the curse of dimensionality [13]. 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 6.12 (see p. 146), we present the median runtimes in seconds over the execution of 20 CI tests with $n = 1\,000$ data 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.

method	k_{CMI}	$\lvert S^{i,j}\rvert$ - separation set size 1	2	3	4	5
CMIknn	200	2.43 s	2.9 s	3.28 s	3.56 s	4.02 s
$\text{GPU}_{\text{CMIknn}}$	7	0.005 s	0.01 s	0.01 s	0.01 s	0.01 s
	20	0.01 s	0.01 s	0.01 s	0.01 s	0.01 s
	200	0.39 s	0.39 s	0.39 s	0.38 s	0.38 s

Table 6.12: Median runtimes in seconds over 20 CI tests, scaling $|S^{i,j}|$ for CI tests with fixed parameters: $n = 1\,000$, $perm = 100$.

For $\text{GPU}_{\text{CMIknn}}$, we observe that the runtime remains unaffected by the size of the separation set $|S^{i,j}|$ for all three chosen parameters of $k_{CMI} = \{7, 20, 200\}$. We find that loading the additional dimensions into the GPU thread-local or GPU shared memory does not add any measurable costs.

Experiment on the Impact of the Number of Data Samples n

The number of data samples n significantly impacts the runtime of k-NN-estimation approaches (see Section 4.4.1). Thus, we assume that the runtime of $\text{GPU}_{\text{CMIknn}}$ increases quadratic with an increasing number of data samples n, whereas the runtime of CMIknn increases approximately logarithmic concerning an increase of n. Furthermore, according to Runge [214], the parameter k_{CMI}, which has a significant impact on the runtime of $\text{GPU}_{\text{CMIknn}}$, should be chosen in correspondence to the number of data samples n. Therefore, we consider a choice of the value for k_{CMI} in correspondence to the number of data samples n in the following measurements, which we denote by $k_{CMI} = adaptive$. In this case, we set $k_{CMI} = 0.2 \times n$.

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

Table 6.13: Median runtimes in seconds over 20 CI tests, scaling n for CI tests with fixed parameters: $|S^{i,j}| = 1$, $perm = 100$. Note that $k_{CMI} = adaptive$ refers to a value dependent on the number of data samples n, i.e., $k_{CMI} = 0.2 \times n$.

In Table 6.13, we report the median runtimes in seconds over the execution of 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 data samples n from $n = 100$ to $n = 10\,000$. For CMIknn, we confirm that the runtime increases logarithmically with the number of data samples n independent of the chosen

value of k_{CMI}. Similarly, our measurements confirm a quadratic increase in runtime for $\texttt{GPU}_{\texttt{CMIknn}}$, when $k_{CMI} = \textit{adaptive}$. However, for small values of $k_{CMI} = \{7, 20\}$, we observe that the increase in runtime is less drastic. Again, we assume that for $k_{CMI} = \textit{adaptive}$, access to GPU global memory is the main bottleneck within the GPU kernel execution for large numbers of data samples n. 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. However, for a certain number of data samples n, the number of launched GPU threads exceeds the capabilities of the GPU hardware, and we observe the quadratic increase, e.g., for $n \geq 2\,500$. Comparing the $\texttt{GPU}_{\texttt{CMIknn}}$ to `CMIknn`, we find a large gain in runtime performance for data sample sizes of up to $n = 1\,000$ when performing the CI test on the GPU. We observe that the performance gain depends on the parameter k_{CMI} but is within the range of factors of 5.9× to 358×. For larger values of n, i.e., $n \geq 2\,500$, $\texttt{GPU}_{\texttt{CMIknn}}$ 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 $k_{CMI} = \textit{adaptive}$, `CMIknn` is faster by a factor of up to 6.5× than $\texttt{GPU}_{\texttt{CMIknn}}$ for large n.

Summary

In the experiments, we show the runtime performance of the GPU-accelerated information-theoretic CI test $\texttt{GPU}_{\texttt{CMIknn}}$. We find that the parameter k_{CMI} has the largest impact on the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$. For small values of k_{CMI}, e.g., $k_{CMI} = 7$, $\texttt{GPU}_{\texttt{CMIknn}}$ is up to a factor 352 faster than its single-threaded CPU-based counterpart. However, for large values of k_{CMI}, e.g., $k_{CMI} = 500$, $\texttt{GPU}_{\texttt{CMIknn}}$ only remains faster by a factor of 2.3× than the single-threaded CPU-based execution. Thus, if the CPU-based CI test runs in parallel on multiple CPU cores, we assume that $\texttt{GPU}_{\texttt{CMIknn}}$ has a slower runtime for large values of k_{CMI}. Other parameters, such as the number of permutations *perm*, the size of the separation set $|S^{i,j}|$, or the number of data samples n, with $n \leq 2500$, have little impact on $\texttt{GPU}_{\texttt{CMIknn}}$'s runtime. In conclusion, we find that employing our proposed GPU-accelerated CI test $\texttt{GPU}_{\texttt{CMIknn}}$ results in large speedups unless large values of k_{CMI} are required or millions of data samples n are processed.

6.2.4 Experiments for a GPU-Accelerated Adjacency Search with an Information-Theoretic GPU-Based CI Test

This section describes the experiments for our two proposed GPU-accelerated adjacency searches using the information-theoretic GPU-based CI test $\texttt{GPU}_{\texttt{CMIknn}}$. The first version $\texttt{GPU}_{\texttt{CMIknn}}$`-Single` refers to the case that the CI test $\texttt{GPU}_{\texttt{CMIknn}}$ is directly plugged into the adjacency search of PC-stable. The second version $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` refers to our adapted adjacency search as described in Section 4.4.2. We compare the two versions in the following experiments to a serial and a parallel CPU-based adjacency search.

In the first three experiments, we utilize the CI test `CMIknn` [214], thus focusing on continuous data with non-linear relationships and adjusting the GPU-based versions accordingly. In the last experiment, we consider the case of mixed discrete-continuous data with non-linear relationships and use the appropriate

CMI estimator, as described in Section 4.4.2 (see Algorithm 12). All measurements are performed on the `A40` system (see Section 6.1.2 for detail), and the parallel CPU-based adjacency searches run on 8 CPU cores.

Further, we set the following parameters as defaults. All CGMs are generated with the number of observations fixed to $n = 1\,000$. The adjacency search algorithms are launched with the number of k-NN during local permutation computation as $k_{perm} = 15$, the number of permutations set to $perm = 100$, and the significance level fixed to $\alpha = 0.01$. We set the launch parameters for the GPU kernels as follows $\beta = 32$ and $\gamma = 32$.

The first set of experiments compares the runtime of the algorithms when the number of variables N increases under the assumption of different values of k_{CMI}. The choice of parameter k_{CMI} impacts the quality of the learned CGM [214]. Thus, in the third experiment, we examine the influence of changing the value of k_{CMI} on the quality of the learned CGM using the Structural Hamming Distance (SHD) [269] as an evaluation metric. The SHD is a common measure to compare the quality of learned CGMs' CPDAG. In the last experiment, we consider the case of mixed discrete-continuous data with non-linear relationships and study the algorithms' runtime performance with different ratios of discrete variables in $\mathbf{V}$, denoted by dr, with $dr = \{0.0, \ldots, 1.0\}$. A value of $dr = 0$ corresponds to a CGM that contains only continuous variables, i.e., $\mathbf{V} = V^{Con}$ and consequently, $dr = 1$, refers to a CGM that has only discrete variables, i.e., $\mathbf{V} = V^{Dis}$. In line with the experiments from Section 6.2.3, we will also observe the influences of the different PUs and k-NN estimation approaches in all experiments.

Experiments on the Scalability with the Number of Variables N

In the following, we describe results from two experiments concerning the runtime when the number of variables N increases. First, we consider synthetic CGMs with up to $N = 50$ variables and densities between $\{0.1, \ldots, 0.5\}$. Second, we examine the runtime for high-dimensional sparse synthetic CGMs, with up to $N = 1\,000$ variables, and selected high-dimensional gene expression datasets.

Generally, the number of variables N directly impacts the number of CI tests to be performed, and thus larger values of N yield higher runtimes of the adjacency search of PC-stable. Remember that the computational complexity concerning N is exponential in the worst case and remains polynomial for sparse CGMs [109]. We assume that the difference in speedups relative to a serial execution on the CPU should remain fairly constant once the parallel computational units of a Processing Unit (PU) are saturated.

In Table 6.14 (see p. 149), we present the measurements based on synthetic generated CGMs with a density randomly drawn from $\{0.1, \ldots, 0.5\}$, increasing the number of variables N within the CGMs. We report the median speedup over 10 CGMs for $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, and `CPU-8` over the serial CPU-based adjacency search. `CPU-8` is the parallel CPU-based adjacency search running on 8 CPU cores.

The measurements show that the CPU-based parallel version, `CPU-8`, achieves a speedup of roughly a factor of 4× over the serial version, even though it utilizes 8 CPU cores. For $k_{CMI} = 200$, we observe slightly less speedup than for small values of k_{CMI}. This effect is explained given that a single CI test's runtime is higher for $k_{CMI} = 200$, which amplifies the

method	k_{CMI}	synthetic CGMs N - number of variables 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×	176.50×	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×	1 002.00×
$\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$	20	274.15×	265.17×	466.63×	522.87×	489.25×
	200	3.70×	3.48×	5.09×	4.68×	5.59×

Table 6.14: Factors of speedup over the serial CPU-based adjacency search, as the median of the runtimes over 10 different CGMs, with $n = 1\,000$, $perm = 100$, and $k_{perm} = 15$ for selected values of k_{CMI} when increasing the number of variables N. The significance level of the adjacency search is set to $\alpha = 0.01$. `CPU-8` refers to the parallel CPU-based adjacency search executed on 8 CPU cores.

impact of load imbalance in parallel execution of the PC algorithm's adjacency search [224]. For the GPU-based versions of the adjacency search, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$, and $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, we observe a high speedup for small values of $k_{CMI} = \{7, 20\}$. However, for $k_{CMI} = 200$, the achieved speedup is similar to that of the parallel CPU-based version `CPU-8`. Furthermore, we find that for smaller values of N, e.g., $N \leq 40$, the speedup of $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ over the serial CPU-based version increases as N increases. We account for the additional speedup to the implementation of $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, which allows for the parallel processing of multiple CI tests in the same GPU kernel. For larger N, i.e., $N \geq 50$, the SMs of the GPU are saturated, and no additional speedup is achieved.

Comparing $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$ with $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, we observe that for $k_{CMI} = \{7, 20\}$ $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ is between 1.44 to 2.93 times faster than $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$. In the case of $k_{CMI} = 200$, we observe similar speedups over the serial CPU-based adjacency search for both GPU-based versions.

Overall, we confirm the performance gain of $\texttt{GPU}_{\texttt{CMIknn}}$ in the context of the PC algorithm's adjacency search 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 6.2.3. In these settings, our proposed version $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ outperforms `CPU-8` by factors of up to 240. However, for large k_{CMI}, i.e., $k_{CMI} = 200$, `CPU-8` and both GPU-based versions of the PC algorithm's adjacency search have similar runtimes. Thus, if a fast runtime is a primary goal, we recommend choosing a small value for k_{CMI}. Although, this contradicts the recommendation for choosing k_{CMI} by Runge [214].

In Table 6.15 (see p. 150), we present the measured runtime in seconds for higher-dimensional sparse synthetic CGMs or selected gene expression datasets, which have purely continuous data. These measurements extend the previous results but focus on high-dimensional sparse CGMs. Due to the longer runtimes, we did not measure the serial CPU-based runtime and terminated

method	synthetic CGMs				NCI-60	MCC	BR51
	N - number of variables						
	100	250	500	1 000	1 190	1 380	1 592
CPU-8	1 399 s	5 425 s	17 586 s	62 967 s	82 287 s	DNF	DNF
$\text{GPU}_{\text{CMIknn}}$-Single	25.5 s	122 s	446 s	1 715 s	1 867 s	7 545 s	4 677 s
$\text{GPU}_{\text{CMIknn}}$-Parallel	14.4 s	74.7 s	281 s	1 088 s	764 s	1 653 s	1 680 s

Table 6.15: Runtimes measured in seconds for high-dimensional sparse synthetic CGMs and selected gene expression datasets used in previous work [226]. The sparse synthetic CGM are generated with $n = 1\,000$ and an edge density that results in DAGs with an average degree of approximately 1.5. The algorithms' parameters are set as follows: $perm = 100$, $k_{CMI} = 7$, $k_{perm} = 15$, $\beta = 32$, $\gamma = 32$, $\alpha = 0.01$. Note that experiment runs longer than 24 hours were terminated and marked with did not finish (DNF).

measurement runs that did not finish (DNF) within 24 hours. For the higher-dimensional sparse synthetic CGMs, we observe almost two orders of magnitude faster runtimes for $\text{GPU}_{\text{CMIknn}}$-Parallel compared to CPU-8. At the same time, $\text{GPU}_{\text{CMIknn}}$-Parallel is faster than $\text{GPU}_{\text{CMIknn}}$-Single by factors between 1.58× and 1.77×. With increasing numbers of variables N, we do not observe significant differences in the measured speedup, which aligns with the observation from the first experiment (see Table 6.14). However, the achieved speedup is less for the high-dimensional sparse CGMs than the achieved speedup for lower-dimensional denser CGMs. We account for the performance degradation to the higher load imbalance in sparse settings [224], which have a higher impact on the performance of the GPU-based versions.

Experiment on the Quality of the Learned CGM Concerning the Parameter k_{CMI}

The proposed GPU-accelerated adjacency search achieves a substantial speedup for small values of k_{CMI}, regardless of other parameters. However, according to Runge [214], the value of k_{CMI} should be chosen according to the number of data samples n, e.g., $k_{CMI} = n \times \{0.1, \ldots, 0.2\}$, to achieve high quality in the learned CGMs. Thus, in the following, we examine the impact of the parameter k_{CMI} on the Structural Hamming Distance (SHD) [269] when learning a CGM's CPDAG. The SHD is a measure that compares two graphs and counts edge insertions, deletions or flips required to transform one graph into the other. In our case, we compute the SHD from the learned CPDAG of the CGM and the underlying true DAG of the CGM. For the experiment, we randomly generate 50 CGMs with $N = 20, n = 1\,000$ and execute $\text{GPU}_{\text{CMIknn}}$-Parallel with $perm = 100$, $\alpha = 0.01$ and values for k_{CMI} from $\{7, \ldots, 200\}$. We set the computed SHD for $k_{CMI} = 7$ as a baseline, as it results in the highest measured speedup. Then, we calculate the difference between the baseline SHD and the SHD computed for the other values of k_{CMI}, denoted by Δ SHD. Hence, values of Δ SHD below 0 indicate a quality improvement of the learned CPDAG of the CGM compared to the base case $k_{CMI} = 7$.

In Figure 6.5 (see p. 151), we report the minimum, median and maximum Δ SHD from 50 randomly generated CGMs. We observe that the median Δ SHD

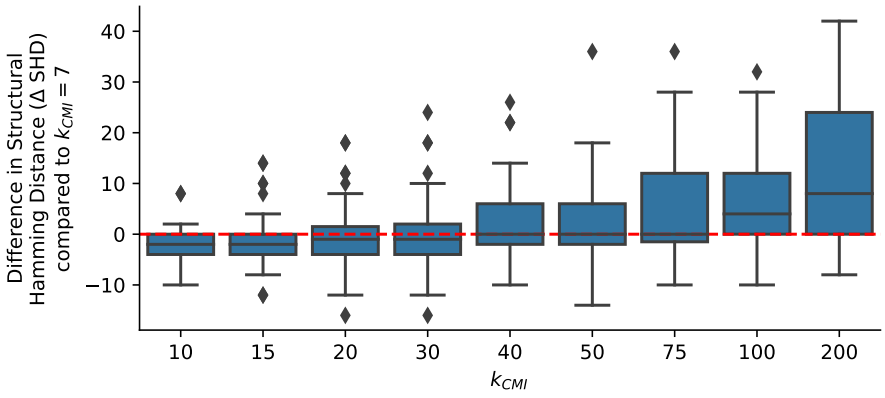


Fig. 6.5: Development of the SHD with increasing k_{CMI} as the difference regarding a baseline SHD with $k_{CMI} = 7$, computed for 50 randomly generated CGMs with $N = 20$, $n = 1\,000$. We execute $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ with parameters $perm = 100$ and $\alpha = 0.01$. Note that Δ SHD < 0 describes an improved quality of the learned CPDAG of the CGM compared to the baseline with $k_{CMI} = 7$.

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, we could conclude that small values of k_{CMI} are sufficient to learn the CGM's CPDAG, which favors the runtime improvement of our GPU-accelerated approach. However, 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's CPDAG based on the parameter k_{CMI}. Extended experiments on the impact of k_{CMI} on the quality of the learned CPDAG of the CGM are left for future work.

Experiment on the Scalability With Different Ratios of Discrete Variables Using the CMI Estimator by Mesner & Shalizi [161]

In the following experiment, we consider the case of mixed discrete-continuous data with non-linear relationships and study the adjacency search's runtime with different ratios of discrete variables in $\mathbf{V}$, denoted by dr, with $dr = \{0.0, \ldots, 1.0\}$. Thus, we consider the adjacency search with a CMI estimator appropriate for the data, as described in Section 4.4.2 (see Algorithm 12). Note that for the CPU-based adjacency search, we adapt the `CMIknn` [214] CI test accordingly and implement the CMI estimator by Mesner [161].

We assume that the number of discrete variables has little impact on the runtime of the GPU-based adjacency search, as the CI test $\texttt{GPU}_{\texttt{CMIknn}}$ utilizes a brute force-based k-NN search. In contrast, the CPU-based CI test relies on k-d trees for an efficient k-NN search. In the case of increasing the number of discrete variables, we assume that the performance of the k-d tree-based k-NN search degrades due to larger leaf sizes given an increased density of data.

In Table 6.16 (see p. 152), we report the median speedups over 10 CGMs for $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Single}$, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$, and `CPU-8` over the single-

method	dr - ratio of discrete variables					
	0.0	0.2	0.4	0.6	0.8	1.0
`CPU-8`	5.38×	5.13×	5.09×	5.6×	5.37×	5.35×
`GPU-Single`	1 039.12×	1 025.18×	1 215.02×	1 552.99×	1 663.28×	2 083.88×
`GPU-Parallel`	1 629.81×	1 595.59×	1 892.23×	2 565.19×	2 740.0×	3 508.89×

Table 6.16: Factors of speedup over serial a CPU-based adjacency search, as the median of the runtimes of 10 different CGMs, with $n = 1\,000$, $N = 20$, $perm = 100$ and $k_{perm} = 15$ when increasing the ratio of discrete variables dr from $dr = 0.0$ to $dr = 1.0$. The significance level of the adjacency search is set to $\alpha = 0.01$. `CPU-8` refers to the parallel CPU-based adjacency search executed on 8 CPU cores.

threaded CPU-based adjacency search with varying ratios of discrete variables $dr = \{0.0, \ldots, 1.0\}$. For the parallel CPU-based adjacency search `CPU-8`, we observe a similar speedup over the serial CPU-based version for all values of dr. This result is expected, as the implementation of the k-NN search is the same in both adjacency searches. In contrast, we observe that the speedup for both GPU-based adjacency searches increases for larger values of dr, i.e., more discrete variables within the CGM. In detail, the measured runtimes of the GPU-based adjacency searches remain steady, whereas, for the CPU-based adjacency searches relying on k-d trees, the runtimes deteriorate. As a result, the speedups of $\texttt{GPU}_{\texttt{CMIknn}}$`-Single` and $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` over the serial CPU version increase by a factor of 2× from $dr = 0.0$, i.e., purely continuous CGMs, to $dr = 1.0$, purely discrete CGMs. For $dr = 1.0$, $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` achieves a speedup of a factor of 3 508× over the serial CPU-based adjacency search and a speedup of a factor of 655× `CPU-8`.

Summary

In the aforementioned four experiments, we demonstrate the runtime performance of our GPU-accelerated adjacency search using the information-theoretic GPU-based CI test $\texttt{GPU}_{\texttt{CMIknn}}$. In particular, we find that the runtime of $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` mainly depends on the chosen value for the parameter k_{CMI}. For small values of k_{CMI}, e.g., $k_{CMI} = 7$, $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` outperforms a parallel CPU-based version running on 8 cores by up to a factor of 240×. However, for large values of k_{CMI}, e.g., $k_{CMI} = 200$, we observe that $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` has similar runtimes than the parallel CPU-based version. It is known that the chosen value of k_{CMI} impacts the quality of the CI test [214], consequently, the quality of a learned CGM's CPDAG. However, 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. Nevertheless, future work requires more research on k_{CMI}'s impact on the quality of the learned CGM's CPDAG.

Further, we observe that the number of variables N does not impact the achieved speedup. However, the density of the underlying CGM impacts the achieved speedup. Notably, for sparser high-dimensional CGMs, we observe lower factors of speedup, which we account for higher load imbalance that the GPU-accelerated adjacency search cannot handle efficiently. Lastly, concerning the ratio of discrete variables within the CGM, we observe higher speedups in the

runtime for CGMs with more discrete variables. For a purely discrete CGM, i.e., $dr = 1.0$, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ outperforms the CPU-based adjacency search executed on 8 CPU cores by a factor of up to 655×. This observation highlights the potential of $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ to speed up CSL for mixed discrete-continuous data but also data with non-linear relationships.

6.3 Experiments on GPU-Based CSL Beyond a Single GPU's Memory Capacity

This section describes the experiments that address our second research question (RQ2). Thus, the measurements of our experimental evaluation examine the scalability and performance of our proposed Unified Memory (UM)-based and explicit memory-managed approaches to execute our GPU-accelerated adjacency search algorithms in out-of-core and multi-GPU settings (see Chapter 5). We focus on executing the adjacency search under the assumption of data that follows the Gaussian distribution model, i.e., using the GPU kernels described in Section 4.2. Further, we restrict the execution of the algorithms to levels $l = 0, 1$, as higher levels $l \geq 2$ rely on a similar memory access pattern as level $l = 1$. In level $l = 1$, random memory access is dominant, whereas, in level $l = 0$, most memory access is sequential. *Parts of this section have been published in two research papers [80, 225].*

In the first set of experiments, we examine the runtime and scalability of our approaches using synthetic data with increasing numbers of variables N. Increasing the number of variables N corresponds to an increase in the required GPU global memory. Thus, we can consider datasets that exceed the GPU memory capacity of a single GPU or of multiple GPUs. Furthermore, we perform runtime measurements of our approaches using the `TCGA` dataset, which stems from research in integrative gene selection approaches [193]. With this measurement, we aim to demonstrate the applicability of our out-of-core and multi-GPU adjacency search algorithms in a realistic setting and compare the runtimes to parallel execution on a multi-core CPU. If not stated differently, all experiments are performed on the `Delos` system (see Section 6.1.2 for detail) with `CUDA` version 11.3, and we set the significance level $\alpha = 0.01$. Further, we set the GPU kernel launch parameter $\delta = 64$ and the block size $bs = 2\,048$. The choice of parameters was determined through micro experiments.

6.3.1 Experiments for Levels $l = 0, 1$ Using Synthetic Data

For the experiments using synthetic data, we first examine the impact of the interconnects in the `Delos` system on the runtime when performing our GPU-accelerated adjacency searches on multiple GPUs. We use the best-performing combination of GPUs in the subsequent experiments. In these experiments, we consider the scalability of our approaches when the number of variables N increases. First, we consider the runtime in levels $l = 0, 1$ separately. In that experiment, we aim to understand the implications of the dominant memory access pattern, i.e., sequential in level $l = 0$ and random in level $l = 1$, on the runtime of the UM-based approaches compared to the explicit memory managed, i.e., block-based approaches. Finally, we examine the combined runtimes when executing the adjacency search up to level $l = 1$.

Experiment on the Impact of the GPU Interconnects on the Runtime of the Multi-GPU Adjacency Search Algorithms

We perform the measurements on the `Delos` system, which is equipped with four GPUs. As detailed in Section 6.1.2, the `Delos` system connects the GPUs in a ring topology using different numbers of interconnect lanes to connect pairs of GPUs. To understand the impact of the number of lanes used by the interconnect, i.e., none, one, or two, we select one pair of GPUs from each group and measure the adjacency search's runtime for synthetic datasets with $N = 5\,000$ to $N = 45\,000$ variables.

For the UM-based multi-GPU adjacency search, non-surprisingly, we find that the fastest runtime is achieved if the pair of GPUs belong to the group *direct two lanes*, which has the highest bandwidth. The runtime is, on average, 12% faster than execution on GPUs from the *direct one lane* group and 345% faster than using GPUs from the *indirect* group. Note that these results align with observations found in related work [132]. As a result, for the subsequent experiments, we use combinations of GPUs that stem from the *direct two lanes* group. For example, when using two GPUs, we select GPU $0, 1$, or if three GPUs are used, we select GPU $0, 1, 2$ (see Figure 6.1 in Section 6.1.2).

We do not observe any measurable difference in the runtime when performing the same experiment using the block-based multi-GPU adjacency search algorithm. This result is expected as data transfer only occurs between the CPU and each GPU separately in the block-based multi-GPU adjacency search algorithm.

Experiment Measuring the Runtimes of the UM-Based and Block-Based GPU-Accelerated Adjacency Search Algorithms on Levels $l = 0, 1$ Separately

The GPU kernels for levels $l = 0$ and $l \geq 1$ have different memory access patterns. Due to empty separation sets $\mathbf{S}^0 = \emptyset$ in level $l = 0$, the calculation of the p-value requires only a single access to the correlation matrix Cor. Therefore, conducting multiple CI tests in parallel results in sequential and thus predictable memory access. In contrast, in levels $l \geq 1$ (we exemplary consider $l = 1$ in the following), the calculation of the p-value requires access to multiple locations within the correlation matrix Cor, according to the pair of variables (V_i, V_j) and the respective separation set $S^{i,j}$, with $i, j = 1, \ldots, N$ and $i \neq j$. Thus, processing multiple pairs of variables and corresponding CI tests in parallel causes random memory access. To understand the implications for the UM-based and block-based GPU-accelerated adjacency searches, we measure the runtimes on synthetic CGMs with different numbers of variables N. Also, we execute the algorithms on different numbers of GPUs g. In the case that $g = 1$, we perform the out-of-core GPU approaches, whereas we execute the multi-GPU approaches for cases with $g \geq 2$.

In Table 6.17 (see p. 155), we report the factors of speedup of the block-based GPU-accelerated adjacency searches over the UM-based GPU-accelerated adjacency searches for levels $l = 0, 1$ separately. For level $l = 0$, the UM-based algorithms are faster than the block-based algorithms in almost all cases. The following two effects explain this result. First, the UM-based algorithms leverage the page migration engine with prefetching mechanisms that are well-suited

l	g \ N	5 000	10 000	20 000	30 000	40 000	50 000	60 000	70 000	80 000
0	1	0.3×	0.3×	0.2×	0.3×	0.4×	0.4×	0.4×	0.4×	0.5×
	2	0.6×	0.4×	0.3×	0.3×	0.3×	0.2×	0.3×	0.4×	0.4×
	3	1.0×	0.9×	0.7×	0.6×	0.6×	0.6×	0.4×	0.7×	0.8×
	4	1.0×	1.1×	0.8×	0.7×	0.7×	0.6×	0.6×	0.6×	0.9×
1	1	0.6×	0.6×	0.6×	0.7×	0.7×	20.2×	43.8×	60.0×	73.8×
	2	0.8×	0.6×	0.9×	0.7×	0.7×	1.4×	25.0×	53.2×	75.7×
	3	2.6×	1.9×	1.8×	2.3×	2.6×	3.0×	15.4×	45.7×	70.8×
	4	1.8×	2.0×	1.6×	1.4×	1.6×	2.3×	9.9×	40.0×	66.7×

Table 6.17: Factors of speedup of the block-based GPU-accelerated adjacency search over the UM-based GPU-accelerated adjacency search for level $l = 0$ (top) and level $l = 1$ (bottom). The approaches are executed with different numbers of GPUs g and varying numbers of variables N.

to handle sequential memory access. Second, the block-based algorithms' performance suffers from the non-negligible overhead introduced through a central queue and the splitting of data into small-sized blocks.

For level $l = 1$, the UM-based algorithms are faster when executed on one or two GPUs for datasets with up to $N = 40\,000$ variables. The block-based multi-GPU algorithm is faster by factors of up to 2.6× for datasets with up to $N = 40\,000$ variables when three or four GPUs are utilized. For large datasets, with $N \geq 50\,000$, the block-based multi-GPU algorithm is faster by factors of up to 75.7×, and the block-based out-of-core GPU algorithm is faster by a factor of up to 73.8× compared to the UM-based approaches. We explain the observed speedup in level $l = 1$ as follows. First, for the UM-based multi-GPU approach, the GPU interconnect topology becomes a performance bottleneck when three or four GPUs are utilized due to remote data accesses and data migrations. Second, for the UM-based out-of-core approach, the data structures for large datasets with $N > 40\,000$ exceed the available GPU global memory. As a result, the runtime degrades due to a drastic increase in page faults. Similar effects are visible when running on more than one GPU for higher numbers of variables N. Note that the block-based approach is not facing these performance drops, as it is independent of the inter-GPU communication, the GPU interconnect topology and GPU page faults.

Runtimes of the UM-Based and Block-Based GPU-Accelerated Adjacency Search up to Level $l = 1$

In Figure 6.6 (see p. 156), we report the runtimes of the UM-based GPU-accelerated adjacency search and the block-based GPU-accelerated adjacency search executed on up to four GPUs g while processing CGMs with increasing numbers of variables N. Note that the out-of-core GPU approaches are executed in the case that $g = 1$, whereas for $g \geq 2$, the multi-GPU approaches are used. The vertical lines mark the number of variables for which the data structures exceed the GPU memory capacity on the `Delos` system of one to four GPUs from left to right.

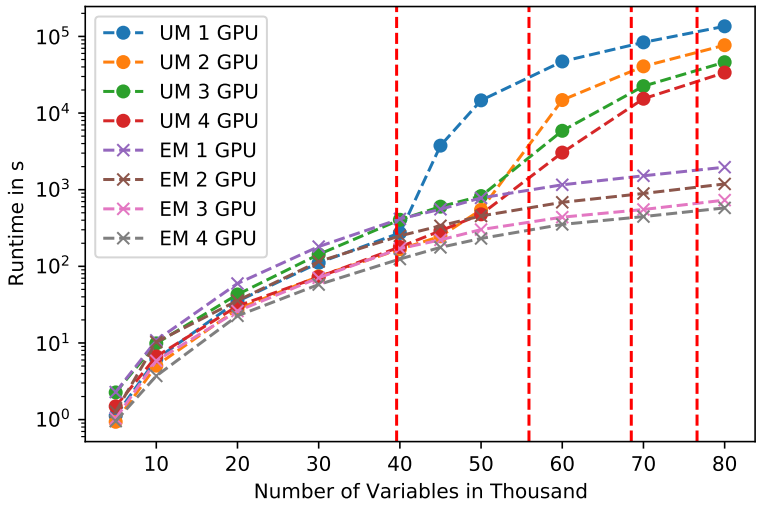


Fig. 6.6: Runtimes of the UM-based (UM) GPU-accelerated adjacency search and the block-based (EM) GPU-accelerated adjacency search up to level $l = 1$. The algorithms are executed on up to four GPUs and process CGMs with increasing numbers of variables N. The red dotted vertical lines mark the number of variables for which the CGM's data structures exceed the GPU's memory capacity of one to four GPUs (left to right).

Our measurements illustrate that the UM-based adjacency searches face a strong increase in runtime for specific numbers of variables. The observed behavior results from page faults occurring once the GPU global memory is exceeded by several GB, e.g., see the measurements for the UM-based approach executed on a single GPU (UM 1 GPU). When executed on multiple GPUs, drops in performance are also observed for the UM-based multi-GPU adjacency search. However, the performance drops are not as severe as in the single GPU case and occur for larger numbers of variables, e.g., $N \geq 60\,000$. When performing the UM-based adjacency search on three or four GPUs, we observe a drop in performance once the GPU memory capacity of the first two GPUs is exceeded. We assume this behavior results from costs for inter-GPU communication, page faults, and page migrations between the GPUs. For the block-based approach, we observe that the runtime scales quadratic to the number of variables N, as expected. For smaller numbers of variables, such that the GPU global memory suffices, e.g., $N < 40\,000$, the UM-based multi-GPU adjacency search is faster when running on two GPUs, whereas the block-based approach is faster when running on three or four GPUs. We also attribute this observation to inter-GPU communication cost, which degrades the runtime.

6.3.2 Experiment on the Application to Real-World Gene Expression Data up to Level $l = 1$

In the following, we compare the runtime of the UM-based and block-based GPU-accelerated adjacency searches with the parallel CPU-based implementation from `pcalg` [111] on the real-world gene expression dataset `TCGA` [193]. The

TCGA dataset contains 55 572 variables. Thus the corresponding data structures exceed a single GPU' global memory. We perform the GPU measurements on the Delos system and run the CPU-based algorithm on the Galileo system. Hence, the CPU-based adjacency search is executed on 32 cores in parallel.

device	CPU	GPU			
approach/library	pcalg	UM-based		block-based	
number of PUs	32 cores	$g = 1$	$g = 4$	$g = 1$	$g = 4$
block size				$bs = 2\,048$	$bs = 2\,048$
runtime in s	387 360	31 456	670	1 063	309

Table 6.18: Runtimes in seconds of the CPU-based adjacency search from pcalg [111] executed in parallel on 32 cores, the UM-based GPU-accelerated adjacency search, and the block-based GPU-accelerated adjacency. For the GPU-based adjacency searches, we set the number of GPUs to $g = \{1, 4\}$, e.g., either using the out-of-core GPU approaches or the multi-GPU approaches. We restrict the adjacency search to a maximum level of $l = 1$ to limit the experiment's runtime.

In Table 6.18, we report the runtime measurements. The CPU-based algorithm runs for over four days, whereas all GPU-based approaches finish in less than one hour. Using a single GPU leads to a speedup of a factor of 364× compared to execution on a multi-core CPU system. Furthermore, the block-based out-of-core GPU approach is faster by a factor of 29.6× compared to the UM-based out-of-core GPU approach. Extending the execution to four GPUs, we observe a speedup of up to a factor of 1 253× compared to execution on a multi-core CPU system. In that setting, the block-based multi-GPU approach is the fastest, with a runtime of 309 seconds, and is 2.17 times faster than the UM-based multi-GPU approach. Comparing the two UM-based approaches, we observe that the multi-GPU approach running on four GPUs is 46.6 times faster than the out-of-core GPU approach executed on a single GPU. Profiling with the nvprof profiler reveals that the UM-based out-of-core GPU algorithm faces many page faults due to the memory demand of the data structures for TCGA, which exceeds the single GPU's memory capacity.

6.3.3 Summary

Our approaches to scale the GPU-accelerated adjacency search beyond a single GPU's memory capacity can process high-dimensional real-world datasets that exceed a single GPU's memory capacity. In detail, leveraging the parallel computational power of four GPUs, we achieve a factor of speedup of 1 253× compared to a parallel CPU-based implementation running on 32 CPU cores. Also, we observe that our block-based approaches that rely on explicit memory management outperform UM-based approaches in settings where the data structures exceed the GPU's memory capacity. Particularly as our block-based approaches are not impacted by the GPU interconnect topology and do not require handling of page faults and page migrations between GPUs.

However, our experimental evaluation has two limitations. First, we restrict the execution of the adjacency search to a maximum level of $l = 1$. While this restriction suffices to examine the different behavior due to different memory access patterns, we cannot conclude on the runtime in higher levels $l \geq 2$. Particularly as the block-based approach requires additional optimizations in higher levels $l \geq 2$ to handle the number of possible combinations of the blocks required to allow for all possible separation sets. Second, we focus our evaluation on the assumption that all data follows the Gaussian distribution model. It is subject to future work to examine our approaches under the assumption of other data characteristics, e.g., considering discrete data.

6.4 Discussion

In this section, we discuss the evaluation results of our algorithms for GPU-accelerated adjacency searches in PC-stable. In most settings examined in our experiments, our proposed GPU-accelerated algorithms achieve faster runtimes than state-of-the-art parallel CPU-based implementations. However, we are aware that some settings are unfavorable for our GPU-accelerated algorithms, which we mention separately for each of our algorithms below.

GPU-Accelerated Adjacency Search for Data Following the Gaussian Distribution Model

For data that follows the Gaussian distribution model, we observe a speedup of up to a factor of 93.4×, comparing our GPU-based adjacency search to the parallel CPU-based variant from `pcalg` [111]. When executing the entire PC-stable, our GPU-based algorithm, as implemented in `gpucsl` [16], is up to 20 times faster than `pcalg`. The `cupc-S` algorithm [287] that incorporates optimizations specific to data following the Gaussian distribution model has a 4.54 times faster GPU kernel runtime in a selected setting than our GPU-based algorithm.

Despite the achieved speedup, our evaluation focused on high-dimensional datasets. Thus, we cannot generalize to data of any size. We assume that for small-dimensional datasets, the overhead in GPU processing, i.e., costs for data transfer or GPU kernel launches, outweigh the performance gains. This is particularly the case for datasets that expose insufficient parallelism to saturate the GPU's SMs.

GPU-Accelerated Adjacency Search for Discrete Data

For discrete data, our GPU-accelerated adjacency search achieves a speedup of up to a factor of 34.2× compared to the parallel CPU-based implementation from `bnlearn` [233]. Considering the execution of the entire PC-stable algorithm, our GPU-based algorithm, as implemented in `gpucsl` [16], is up to 54.3 times faster than `bnlearn`. Furthermore, our GPU-based adjacency search, which builds upon a parallel execution strategy tailored to the CI test for discrete data, outperforms a GPU-based variant that utilizes the parallel execution strategy from the `cupc-E` algorithm [287] by up to a factor of 6.5×.

However, in the end-to-end measurement, we observe that our GPU-based adjacency search has a longer runtime than `bnlearn` for datasets with few variables N and data samples n. Furthermore, we observe that the gap in runtime

between our GPU-based variant and the parallel CPU-based implementation decreases if the density of the CGM increases. Lastly, a large maximum size of the discrete domain $\max_{i=1,\ldots,N}\{|\mathcal{V}_i|\}$, with $\mathcal{V}_i$ representing the corresponding discrete domain of the variable V_i with $i = 1, \ldots, N$, can cause high demand for GPU global memory, which exceeds the GPU's memory capacity. In this case, our GPU-based algorithm restricts the degree of parallelism at the cost of increased runtimes, which in the worst case, results in slower runtimes than CPU-based execution.

GPU-Accelerated Adjacency Search with an Information-Theoretic GPU-Based CI Test

In the context of mixed discrete-continuous data or data with non-linear relationships, we examine the runtimes of our GPU-accelerated CI test $\texttt{GPU}_{\texttt{CMIknn}}$ and the corresponding GPU-based adjacency search $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$. For continuous data with non-linear relationships, $\texttt{GPU}_{\texttt{CMIknn}}$ is 352 times faster than the CPU-based CI test `CMIknn` [214]. Similarly, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ is faster by factors of up to 240× than a parallel CPU-based adjacency search using `CMIknn`. For mixed discrete-continuous data, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ outperforms a parallel CPU-based adjacency search using `CMIknn` by factors of up to 655×.

However, the achieved speedup requires that the parameter k_{CMI} during the k-NN search is chosen considerably small, i.e., $k_{CMI} < 50$. A small value of k_{CMI} impacts the quality of CI decisions, respectively, the learned CPDAG of the CGM. For larger values of k_{CMI}, the runtime performance between the parallel CPU-based adjacency search and $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ decreases. Eventually, $\texttt{GPU}_{\texttt{CMIknn}}\texttt{-Parallel}$ is slower than CPU-based execution. Furthermore, in our evaluation of the CI test's runtime, the CPU-based implementation runs on a single CPU core. A parallel CPU-based CI test implementation could reduce the gap in runtime.

GPU-Accelerated Adjacency Search Algorithms Beyond a Single GPU's Memory Capacity

Using synthetic data, we experimentally show that our out-of-core GPU-accelerated and multi-GPU-based adjacency searches scale to high-dimensional datasets that exceed the memory capacity of a single GPU. For a real-world gene expression dataset that exceeds the GPU's memory capacity, our block-based out-of-core GPU algorithm executed outperforms a parallel CPU-based implementation by a factor of 364×. On the same dataset, our block-based multi-GPU approach is 1 253 times faster than execution on a multi-core CPU. Further, our UM-based GPU-accelerated approaches are slower than the corresponding block-based approaches, using explicitly managed memory.

Despite these promising results, our evaluation only focuses on executing levels $l = 0, 1$. In higher levels $l \geq 2$, the block-based GPU-accelerated algorithms face a combinatorial challenge, as they must ensure that all possible separation set candidates can be constructed. We assume that this combinatorial challenge causes runtime overhead, making the UM-based GPU-accelerated algorithms or parallel CPU-based implementations favorable. Furthermore, our evaluation is based only on data that follows the Gaussian distribution model. Since our other GPU-accelerated adjacency search algorithms are tailored to CI tests for different data characteristics, we cannot conclude the direct transferability of the evaluation results.

6.5 Summary

This chapter presented our experimental evaluation of our proposed GPU-based CSL algorithms tailored to the GPU's Single Instruction Multiple Threads (SIMT) execution model. The results demonstrated that our GPU-based CSL algorithms outperform state-of-the-art parallel CPU-based implementations for selected CI tests. Thus, addressing our first research question (RQ1), appropriate parallel execution strategies and optimizations for the execution of constraint-based CSL on GPUs (see Sections 4.2, 4.3, and 4.4) yield improvements in the runtime. In detail, the experimental evaluation demonstrated that our GPU-based algorithms outperform state-of-the-art parallel CPU-based implementations by factors of up to 54.3× for discrete data, up to 93.4× for data assuming the Gaussian distribution model, up to 240× for continuous data with non-linear relationships, and up to 655× for mixed discrete-continuous data. Furthermore, we experimentally evaluated our proposed adjacency search approaches for out-of-core and multi-GPU settings. The results showed that our GPU-based approaches scale to arbitrarily large datasets, i.e., datasets that exceed the GPU's on-chip memory, while still achieving speedups over parallel CPU-based implementations. For example, on a large gene expression dataset that exceeds the GPU's memory capacity, we observed that our out-of-core approach is 364 times faster than a parallel CPU-based CSL algorithm. Thus, concerning our second research question (RQ2), our out-of-core and multi-GPU approaches allow scaling constraint-based CSL to arbitrarily large datasets, e.g., high-dimensional datasets, which exceed the GPU's on-chip memory.

In our experimental evaluation and throughout this thesis, we focused on applying our GPU-accelerated adjacency search algorithms in the context of PC-stable [35]. However, the adjacency search is a substantial part of other constraint-based methods, such as `FCI` [36], `RFCI` [36], `PC-Simple` [19], `pPC` [95], and more. In related work [126], it is shown that parallel execution of the adjacency search yields speedup for several of these methods, too. For these methods, we also see the potential that incorporating our GPU-accelerated adjacency search algorithms results in faster runtimes, which underlines the importance of our work and shows additional application areas.

7

Final Remarks

In this chapter, we first discuss the limitations of this thesis concerning the proposed GPU-accelerated CSL algorithms and their evaluation. Next, we describe opportunities for future work that address optimizations for additional speedup and extent the applicability to other CI tests. Finally, we conclude this thesis by answering our research questions through our contributions.

7.1 Limitations

This section discusses the limitations of this thesis, particularly concerning the evaluation and their implication on the generalizability of our contributions.

In our evaluation, we use a set of real-world gene expression datasets, benchmark Bayesian networks, as well as synthetic data. However, several of our experiments (see Sections 6.2.1, 6.2.2, or 6.3) are performed on a limited number of these datasets, particularly when comparing our GPU-accelerated algorithms to CPU-based algorithms. Thus, even though our measurements reveal that our GPU-accelerated algorithms achieve speedup over the execution on the CPU, we cannot claim that this observation generalizes to all possible datasets.

Further, we measure runtimes on different CPU and GPU hardware. However, for each experiment, we chose one specific hardware system for executing our GPU-accelerated algorithms and the CPU-based implementation if applicable. We did not examine the performance of our GPU-accelerated algorithms across systems with different GPU hardware, i.e., with different numbers of SMs or different sizes and technology of the GPU global memory. Thus, based on our current evaluation, it remains open to infer our algorithms' behavior on new GPU generations with changed hardware characteristics.

Additionally, in our experiments, we compare our GPU-based algorithms to parallel CPU-based implementations that execute on as many CPU cores as the hardware system provides. We select the best-performing implementation for the CPU-based implementation from a set of well-known libraries that implement the PC-stable algorithm. To the best of our knowledge, we chose the fastest existing implementation. Despite choosing implementations from well-known libraries, we cannot ensure that these implementations are highly optimized for execution on the CPU. Further, our choice to execute the algorithm on all available CPU cores does not necessarily yield the fastest runtime [123]. Hence, the

measured speedup of our GPU-accelerated algorithms over the CPU-based implementations is specific to the corresponding implementation. Thus, optimized CPU-based algorithms that yield faster runtimes than those selected for our evaluation may exist and can result in a smaller performance gap compared to our GPU-based algorithms.

Lastly, concerning our measurement setup, we cannot rule out an impact on the runtime measurements by operating system processes or other loads on the system. However, we aimed to reduce measurement inaccuracies through repetitive measurement execution while having exclusive access to the selected hardware system.

In this thesis, we focus on the PC algorithm [249], particularly the PC-stable [35], as it is a common choice for constraint-based CSL. However, we exclude a comparison to other approaches to CSL, such as the score-based method fGES [202] or a continuous optimization-based approach [293]. Thus, while our improved runtimes apply to the PC-stable, its variants and extensions, and further constraint-based methods building upon adjacency searches, other CSL methods may yield faster or higher-quality results in specific settings.

7.2 Future Work

As a result of the limitations mentioned in the previous section, we see an in-depth evaluation of our algorithms with respect to their applicability beyond the PC-stable algorithm as future work. Likewise, an in-depth evaluation of the influence of parameter k_{CMI} on the quality of CI decisions for our information-theoretic GPU-based approaches is necessary to ensure extensive applicability in real-world scenarios. Furthermore, an extended evaluation of our approaches to scale beyond a single GPU's memory capacity is needed to show its general utility. Particularly focusing on data with other characteristics than assuming that data follows the Gaussian distribution model.

Based on related work, we see further optimization potential of our algorithms. Specifically, the approach from `cupc-S` [287] to reuse the computed pseudo-inverse while using our parallel execution strategy needs to be explored. Further, examining how our Unified Memory (UM)-based approaches can benefit from current fast interconnects, frameworks to reduce page faults, or the introduction of load balancing mechanisms remains open.

Our work uses GPUs to accelerate the adjacency search of PC-stable in a heterogeneous system. We show that the causal structures can be learned in an acceptable time, even on high-dimensional datasets that exceed the memory capacity of a single GPU. However, based on heterogeneous systems, our approaches' scalability is limited by the number of available GPUs. Extending our work to distributed GPU-accelerated systems could become relevant when the computational demand increases due to increased dataset sizes or the application of more complex CI tests. Distributed CPU-based algorithms such as `MrPC` [169] or `PCB` [44] can serve as a basis.

Modern heterogeneous systems have other co-processors or at least multi-core CPUs in addition to GPUs. Our proposed GPU-based algorithms only require a few CPU threads for orchestration, e.g., launching the GPU kernels. Accordingly, other CPU cores remain unused. In order to use all available PUs in a system, idle CPU cores can be included in the adjacency search computation.

First investigations in the context of a master thesis supervised by the author of this thesis have shown a potential of several percent speedups compared to pure GPU-based execution. However, further investigations are necessary. In this context, the use of additional co-processors should also be examined.

This thesis considers parallel execution strategies for GPU-based adjacency searches concerning specific CI tests. However, many other CI tests [67, 96, 200, 236, 237, 238, 260, 288, 289] can also be plugged into the PC-stable algorithm. To avoid the development of tailored GPU-accelerated adjacency searches for each of these CI tests, we suggest developing a general framework for GPU acceleration of constraint-based CSL. As a first step, classifying existing CI tests according to their memory access patterns and computational steps helps to match them to the appropriate parallel execution strategy suggested in this thesis. For example, the k-NN-based CI tests [100, 214] share similar concepts, allowing GPU processing as described in Algorithm 12 (see Section 4.4).

7.3 Conclusion

Knowledge about the causal structures between variables of a system supports data-driven decision-making in many domains. Today, large amounts of observational data are collected from increasingly complex systems, which CSL methods process to discover the underlying causal structures. However, when processing observational data that are high-dimensional or contain complex non-linear relationships, CSL algorithms struggle with long runtimes of hours or days, which hinders their wide application in practice [123, 150]. Existing CSL algorithms have focused on the parallel execution on multi-core CPUs. We argue that this does not leverage the full potential of modern computing systems, which are often heterogeneous and equipped with co-processors, such as GPUs, to accelerate computations. GPUs typically provide several thousand computational cores for massively parallel data processing.

In this thesis, we examine how CSL algorithms can benefit from the parallel computational power of GPUs to achieve fast runtimes. Particularly, we focus on a widely accepted constraint-based CSL method, the PC algorithm, as it allows choosing a statistical CI test appropriate to the observational data's characteristics. We design parallel execution strategies for the PC algorithm that reflect GPU hardware characteristics and algorithmic details of selected CI tests. We choose three CI tests that are common in practice and cover a wide range of data characteristics. Consequently, we develop three GPU-accelerated variants of the PC algorithm that are tailored to these CI tests. Further, we propose GPU-based approaches to scale beyond a single GPU's memory capacity to allow the processing of arbitrarily large datasets. Our developed GPU-based algorithms allow us to answer our two research questions as follows:

- **RQ1:** *How can we improve the runtime of constraint-based CSL on a GPU* We argue that GPUs are well suited to accelerate constraint-based CSL, given their availability in modern heterogeneous computing systems and massively parallel data processing power. To develop GPU-accelerated constraint-based CSL algorithms, we apply a framework for designing parallel programs and derive parallel execution strategies that consider the GPU's

Single Instruction Multiple Threads (SIMT) execution model and favor local communication patterns that are realized using the GPU's shared memory. Further, we define tasks for parallel execution with different levels of granularity to reflect the algorithmic details of CI tests. For example, our algorithm that targets a CI test for discrete data defines processing of individual data samples as the task for parallel execution and uses units of GPU threads, i.e., warps, to jointly compute marginals over contingency tables. In an experimental evaluation, we compare our GPU-based algorithms to state-of-the-art parallel CPU-based implementations and observe factors of speedup of up to 54.3× for discrete data, up to 93.4× for data assuming the Gaussian distribution model, up to 240× for continuous data with nonlinear relationships, and up to 655× for mixed discrete-continuous data.

- **RQ2:** *How can we scale GPU-accelerated constraint-based CSL to arbitrarily large datasets?*
  We explore two techniques for extending our GPU-accelerated variants of the PC algorithm, tailored to specific CI tests, to process arbitrarily large datasets. First, we employ the concept of UM, which allows for GPU global memory oversubscription and uses the GPU's Memory Management Unit (MMU) for on-demand page migrations and page evictions in GPU global memory. Second, we implement a block-based approach that splits the input dataset into multiple small-sized blocks, which fit into GPU global memory. In the block-based approach, data transfers and GPU kernel execution are explicitly managed to avoid page faults that occur in the UM-based variant. Furthermore, we extend both approaches to operate on multiple GPUs for faster runtimes. Using a large gene expression dataset that exceeds the GPU's memory capacity, we observe that our block-based approach is 364 times faster than a parallel CPU-based CSL algorithm and 29.6 times faster than our UM-based approach.

Based on the answers to our research questions, we contribute to our research goal of *designing efficient execution strategies for constraint-based CSL algorithms that leverage the parallel processing power of GPUs to provide fast runtimes in case of high-dimensional data.* By accelerating the learning of causal structures from datasets that are high-dimensional or contain complex nonlinear relationships, we foster the adoption of CSL in practice.

A

Appendix

A.1 List of URLs

description	URL
Homepage of the top 500 supercomputer list	`https://www.top500.org/`
Homepage of NVIDIA's `CUDA`-X libraries	`https://developer.nvidia.com/gpu-accelerated-libraries`
Implementation of $\texttt{GPU}_{\texttt{CMIknn}}$ and $\texttt{GPU}_{\texttt{CMIknn}}$`-Parallel` on GitHub	`https://github.com/ChristopherSchmidt89/gpucmiknn`
Implementation of our GPU-accelerated python library `gpucsl` on GitHub	`https://github.com/hpi-epic/gpucsl`
Documentation of the `chrono` library	`http://en.cppreference.com/w/cpp/header/chrono`
Documentation of the `nvprof` profiling metrics	`https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference-7x`

Table A.1: List of URLs mentioned in footnotes of this thesis. Last Accessed: December 15, 2022.

A.2 List of Publications

Our main contributions[1] concerning GPU-accelerated constraint-based CSL have been published at international conferences and workshops, e.g., KDD, ICDM, SDM, and SSDBM:

- Hagedorn, C.; Lange, C.; Huegle, J.; Schlosser, R.: *GPU Acceleration for Information-theoretic Constraint-based Causal Discovery*. In *Proceedings of The KDD'22 Workshop on Causal Discovery*. PMLR, 2022, pp. 30–60.
- Braun, T.; Hurdelhey, B.; Meier, D.; Tsayun, P.; Hagedorn, C.; Huegle, J.; Schlosser, R.: *GPUCSL: GPU-Based Library for Causal Structure Learning*. In *2022 International Conference on Data Mining, ICDM 2022 – Workshops*. IEEE, 2022, pp. 1228–1231.
- Hagedorn, C.; Huegle, J.: *GPU-Accelerated Constraint-Based Causal Structure Learning for Discrete Data*. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 2021, pp. 37–45.
- Hagedorn, C.; Huegle, J.: *Constraint-Based Causal Structure Learning in Multi-GPU Environments*. In *Proceedings of the LWDA 2021 Workshops: FGWM, KDML, FGWI-BIA, and FGIR*. CEUR-WS.org, 2021, pp. 106–118.
- Schmidt, C.; Huegle, J.; Horschig, S.; Uflacker, M.: *Out-of-Core GPU-Accelerated Causal Structure Learning*. In *Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2020, pp. 89–104.
- Schmidt, C.; Huegle, J.; Uflacker, M.: *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)*. ACM, 2018, pp. 19:1–19:10.

Our complementary contributions[1] to hardware acceleration for CSL, benchmarking of CSL algorithms, and the application of CSL in the manufacturing domain have been published at international conferences, journals, workshops, and technical reports:

- Hagedorn, C.; Huegle, J.; Schlosser, R.: *Understanding Unforeseen Production Downtimes in Manufacturing Processes Using Log Data-Driven Causal Reasoning*. In *Journal of Intelligent Manufacturing* 33(7), 2022: pp. 2027–2043.
- Huegle, J.; Hagedorn, C.; Schlosser, R.: *A kNN-based Non-Parametric Conditional Independence Test for Mixed Data and Application in Causal Discovery*. In *ECML-PKDD 2023, accepted*. 2023.
- Huegle, J.; Hagedorn, C.; Perscheid, M.; Plattner, H.: *MPCSL – A Modular Pipeline for Causal Structure Learning*. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 2021, pp. 3068–3076.
- Huegle, J.; Hagedorn, C.; Böhme, L.; Pörschke, M.; Umland, J.; Schlosser, R.: *MANM-CS: Data Generation for Benchmarking Causal Structure Learning from Mixed Discrete-Continuous and Nonlinear Data*. In *WHY-21 @ NeurIPS*. WHY-21, 2021, pp. 1–15.

[1] Note that the author published several contributions under his birth name Schmidt.

- Huegle, J.; Hagedorn, C.; Uflacker, M.: *Unterstützte Fehlerbehebung durch kausales Strukturwissen in Überwachungssystemen der Automobilfertigung*. In *Software Engineering 2021 Satellite Events, Lecture Notes in Informatics (LNI)*. Gesellschaft für Informatik, 2021, pp. 1–2.
- Huegle, J.; Hagedorn, C.; Uflacker, M.: *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 (ICJAI)*. IJCAI, 2020, pp. 5246–5248.
- Schmidt, C.; Huegle, J.; Bode, P.; Uflacker, M.: *Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data*. In *Proceedings of The 2019 KDD Workshop on Causal Discovery*. PMLR, 2019, pp. 59–77.
- Schmidt, C.; Huegle, J.: *Towards a GPU-Accelerated Causal Inference*. In *HPI Future SOC Lab – Proceedings 2017*. Universitätsverlag Potsdam, 2020, pp. 187–194.

Furthermore, our complementary contributions[2] that address hardware acceleration in other domains, such as database management systems or business applications, have been published at international conferences and workshops:

- Schmidt, C.; Uflacker, M.: *Workload-Driven Data Placement for GPU-Accelerated Database Management Systems*. In *BTW 2019 – Workshopband*. Gesellschaft für Informatik, 2019, pp. 91–94.
- Schmidt, C.; Dreseler, M.; Akin, B.; Roy, A.: *A Case for Hardware-Supported Sub-Cache Line Accesses*. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2018, pp. 1–3.
- Schwarz, C.; Schmidt, C.; Hopstock, M.; Sinzig, W.; Plattner, H.: *Efficient Calculation and Simulation of Product Cost Leveraging In-Memory Technology and Coprocessors*. In *The Sixth International Conference on Business Intelligence and Technology*. IARIA, 2016, pp. 12–18.

[2] Note that the author published several contributions under his birth name Schmidt.

A.3 Permission for Reuse of Published Material

A.3.1 Reuse of Material Published by ACM

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.
Contributing complete papers to any edited collection of reprints for which the author is not the editor, requires permission and usually a republication fee.
Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

A.3.2 Reuse of Material Published by IEEE

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Hasso Plattner Institute's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/right_link.html` to learn how to obtain a License from RightsLink.

A.3.3 Reuse of Material Published by SIAM

SIAM has sole use for distribution in all forms and media, such as microfilm and anthologies, except that the author(s) or, in the case of a "work made for hire" the employer will retain: The right to use all or part of the content of the paper in future works of the author(s), including the author's teaching, technical collaborations, conference presentations, lectures, other scholarly works and professional activities, or any other activity falling under the fair use provisions of the U.S. Copyright Act. If the copyright is granted to SIAM, then proper notice of SIAM's copyright should be provided.

A.3.4 Reuse of Material Published by Springer

Authors have the right to reuse their article's Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published article in their thesis according to current citation standards.

List of Figures

List of Tables

List of Abbreviations

BIB Balanced Incomplete Block
CGM Causal Graphical Model
CI Conditional Independence
CI test Conditional Independence test
CMI Conditional Mutual Information
CPDAG Completed Partially Directed Acyclic Graph
CPU Central Processing Unit
CSF Conditioning Set Filtering
CSL Causal Structure Learning
DAG Direct Acyclic Graph
DL Deep Learning
DRAM Dynamic Random Access Memory
FLOPS Floating Point Operations per Second
FPGA Field-Programmable Gate Array
GPGPU General-Purpose Graphics Processing Unit
GPU Graphics Processing Unit
GRN Gene Regulatory Network
HBM High Bandwidth Memory
HPC High-Performance Computing
IoT Internet of Things
k-NN k-Nearest Neighbor
MI Mutual Information
ML Machine Learning
MMU Memory Management Unit
NVM Non-Volatile Memory
PU Processing Unit
QPI QuickPath Interconnect
SHD Structural Hamming Distance
SIMT Single Instruction Multiple Threads
SM Streaming Multiprocessor
SVD Singular Value Decomposition
TPU Tensor Processing Unit
UM Unified Memory
UPI Intel Ultra Path Interconnect

References

[1] Agrawal, S. R.; Idicula, S.; Raghavan, A.; Vlachos, E.; Govindaraju, V.; Varadarajan, V.; Balkesen, C.; Giannikis, G.; Roth, C.; Agarwal, N.; Sedlar, E.: *A Many-Core Architecture for in-Memory Data Processing*. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 245–258

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

[3] Alonso-Barba, J. I.; de la Ossa, L.; Gámez, J. A.; Puerta, J. M.: *Scaling up the Greedy Equivalence Search Algorithm by Constraining the Search Space of Equivalence Classes*. In *International Journal of Approximate Reasoning* 54(4), 2013: pp. 429–451

[4] AMD Corporation: *ROCm, a New Era in Open GPU Computing*. `https://rocm.github.io/`, 2016. Accessed: June 15, 2021

[5] Anderson, R.; Mayr, E.: *Parallelism and Greedy Algorithms*. Technical report, Stanford University, 1984

[6] Andersson, S. A.; Madigan, D.; Perlman, M. D.: *A Characterization of Markov Equivalence Classes for Acyclic Digraphs*. In *Annals of Statistics* 25(2), 1997: pp. 505–541

[7] Andreassen, S.; Jensen, F.; Andersen, S.; Falck, B.; Kjærulff, U.; Woldbye, M.; Sørensen, A.; Rosenfalck, A.; Jensen, F.: *MUNIN: An Expert EMG Assistant*. In *Computer-Aided Electromyography and Expert Systems*, Pergamon Press. 1989, pp. 255–277

[8] Baruah, T.; Sun, Y.; Dinçer, A. T.; Mojumder, S. A.; Abellán, J. L.; Ukidave, Y.; Joshi, A.; Rubin, N.; Kim, J.; Kaeli, D.: *Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems*. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 596–609

[9] Bedford Taylor, M.: *The Evolution of Bitcoin Hardware*. In *Computer* 50(9), 2017: pp. 58–66

[10] Beinlich, I. A.; Suermondt, H. J.; Chavez, R. M.; Cooper, G. F.: *The ALARM Monitoring System: A Case Study with two Probabilistic Inference Techniques for Belief Networks*. In *AIME 89*. Springer, 1989, pp. 247–256

[11] Bell, N.; Garland, M.: *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. ACM, 2009, p. 11

[12] Ben-Nun, T.; Levy, E.; Barak, A.; Rubin, E.: *Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle*. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2015, pp. 1–12

[13] Bentley, J. L.: *Multidimensional Binary Search Trees Used for Associative Searching*. In *Communications of the ACM* 18(9), 1975: pp. 509–517

[14] Bishop, C. M.: *Pattern Recognition and Machine Learning*. Springer, 2006

[15] Blumofe, R. D.; Leiserson, C. E.: *Scheduling Multithreaded Computations by Work Stealing*. In *Journal of the ACM* 46(5), 1999: pp. 720–748

[16] Braun, T.; Hurdelhey, B.; Meier, D.; Tsayun, P.; Hagedorn, C.; Huegle, J.; Schlosser, R.: *GPUCSL: GPU-Based Library for Causal Structure Learning*. In *2022 International Conference on Data Mining, ICDM 2022 – Workshops*. IEEE, 2022, pp. 1228–1231

[17] Bress, S.: *The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS*. In *Datenbank-Spektrum* 14(3), 2014: pp. 199–209

[18] Budruk, R.; Anderson, D.; Shanley, T.: *PCI Express System Architecture*. Pearson Education, 2003

[19] Bühlmann, P.; Kalisch, M.; Maathuis, M. H.: *Variable Selection in High-Dimensional Linear Models: Partially Faithful Distributions and the PC-Simple Algorithm*. In *Biometrika* 97(2), 2010: pp. 261–278

[20] Cabezas, J.; Vilanova, L.; Gelado, I.; Jablin, T. B.; Navarro, N.; Hwu, W. W.: *Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes*. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*. ACM, 2015, pp. 3–13

[21] Cancer Genome Atlas Research Network; Weinstein, J. N.; Collisson, E. A.; Mills, G. B.; Shaw, K. R.; Ozenberger, B. A.; Ellrott, K.; Shmulevich, I.; Sander, C.; Stuart, J. M.: *The Cancer Genome Atlas Pan-Cancer analysis project*. In *Nature Genetics* 45(10), 2013: pp. 1113–1120

[22] Cano, A.; Gómez-Olmedo, M.; Moral, S.: *A Score Based Ranking of the Edges for the PC Algorithm*. In *Proceedings of the Fourth European Workshop on Probabilistic Graphical Models (PGM)*. 2008, pp. 41–48

[23] Cantrell, C. D.: *Modern Mathematical Methods for Physicists and Engineers*. Cambridge University Press, 2000

[24] Chen, Y.; Lin, Z.; Zhao, X.; Wang, G.; Gu, Y.: *Deep Learning-Based Classification of Hyperspectral Data*. In *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 7(6), 2014: pp. 2094–2107

[25] Chickering, D. M.: *Learning Bayesian Networks is NP-Complete*. In *Learning from Data: Artificial Intelligence and Statistics V*. Springer, 1996, pp. 121–130

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

[27] Chickering, D. M.: *Optimal Structure Identification with Greedy Search*. In *Journal of Machine Learning Research* 3(11), 2003: pp. 507–554

[28] Chickering, D. M.; Heckerman, D.; Meek, C.: *Large-Sample Learning of Bayesian Networks is NP-Hard*. In *Journal of Machine Learning Research* 5(10), 2004: pp. 1287–1330

[29] Chickering, D. M.; Meek, C.: *Selective Greedy Equivalence Search: Finding Optimal Bayesian Networks Using a Polynomial Number of Score Evaluations*. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2015, pp. 211–219

[30] Choquette, J.; Giroux, O.; Foley, D.: *Volta: Performance and Programmability*. In *IEEE Micro* 38(2), 2018: pp. 42–52

[31] Choquette, J.; Lee, E.; Krashinsky, R.; Balan, V.; Khailany, B.: *3.2 The A100 Datacenter GPU and Ampere Architecture*. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2021, pp. 48–50

[32] Chun, S.; Becker, W. D.; Casey, J.; Ostrander, S.; Dreps, D.; Hejase, J. A.; Nett, R. M.; Beaman, B.; Eagle, J. R.: *IBM POWER9 Package Technology and Design*. In *IBM Journal of Research and Development* 62(4/5), 2018: pp. 12:1–12:10

[33] Claassen, T.; Mooij, J. M.; Heskes, T.: *Learning Sparse Causal Models is Not NP-Hard*. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2013, pp. 172–181

[34] Coates, A.; Huval, B.; Wang, T.; Wu, D. J.; Ng, A. Y.; Catanzaro, B.: *Deep Learning with COTS HPC Systems*. In *Proceedings of the 30^{th} International Conference on International Conference on Machine Learning (ICML)*. JMLR.org, 2013, pp. 1337–1345

[35] Colombo, D.; Maathuis, M. H.: *Order-Independent Constraint-Based Causal Structure Learning*. In *Journal of Machine Learning Research* 15(116), 2014: pp. 3921–3962

[36] Colombo, D.; Maathuis, M. H.; Kalisch, M.; Richardson, T. S.: *Learning High-dimensional Directed Acyclic Graphs with Latent and Selection Variables*. In *The Annals of Statistics* 40(1), 2012: pp. 294–321

[37] Conati, C.; Gertner, A. S.; VanLehn, K.; Druzdzel, M. J.: *On-Line Student Modeling for Coached Problem Solving Using Bayesian Networks*. In *Proceedings of the Sixth International Conference on User Modeling*. Springer, 1997, pp. 231–242

[38] Corneil, D.; Mathon, R.: *Algorithmic Techniques for the Generation and Analysis of Strongly Regular Graphs and other Combinatorial Configurations**. In *Algorithmic Aspects of Combinatorics*, Annals of Discrete Mathematics 2, Elsevier. 1978, pp. 1–32

[39] Cui, R.; Groot, P.; Heskes, T.: *Copula PC Algorithm for Causal Discovery from Mixed Data (ECML PKDD)*. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 377–392

[40] Dagum, L.; Menon, R.: *OpenMP: An Industry-Standard API for Shared-Memory Programming*. In *IEEE Computational Science and Engineering* 5(1), 1998: pp. 46–55

[41] Davidson, J. W.; Jinturkar, S.: *Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses*. In *Proceedings of the*

ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI). ACM, 1994, pp. 186–195

[42] Dawid, A. P.: *Conditional Independence in Statistical Theory*. In *Journal of the Royal Statistical Society: Series B (Methodological)* 41(1), 1979: pp. 1–31

[43] de Campos, C. P.; Zeng, Z.; Ji, Q.: *Structure Learning of Bayesian Networks Using Constraints*. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*. ACM, 2009, pp. 113–120

[44] de Jongh, M.: *Algorithms for Constraint-Based Learning of Bayesian Network Structures with Large Numbers of Variables*. Dissertation, University of Pittsburgh, 2014

[45] Dean, J.; Ghemawat, S.: *MapReduce: Simplified Data Processing on Large Clusters*. In *Communications of the ACM* 51(1), 2008: pp. 107–113

[46] Dempster, A. P.: *Elements of Continuous Multivariate Analysis*. Addison-Wesley, 1969

[47] Dennard, R.; Gaensslen, F.; Yu, H.-N.; Rideout, V.; Bassous, E.; LeBlanc, A.: *Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions*. In *IEEE Journal of Solid-State Circuits* 9(5), 1974: pp. 256–268

[48] Doran, G.; Muandet, K.; Zhang, K.; Schölkopf, B.: *A Permutation-Based Kernel Conditional Independence Test*. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2014, pp. 132–141

[49] Drton, M.; Maathuis, M. H.: *Structure Learning in Graphical Modeling*. In *Annual Review of Statistics and Its Application* 4(1), 2017: pp. 365–393

[50] Economist, T.: *The World's Most Valuable Resource Is No Longer Oil, But Data.*. `https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data`, 2017. Accessed: June 29, 2021

[51] Eddelbuettel, D.; Francois, R.: *Rcpp: Seamless R and C++ Integration*. In *Journal of Statistical Software* 40(8), 2011: pp. 1–18

[52] Eddelbuettel, D.; Sanderson, C.: *RcppArmadillo: Accelerating R with High-performance C++ Linear Algebra*. In *Computational Statistics & Data Analysis* 71, 2014: pp. 1054–1063

[53] Elidan, G.; Gould, S.: *Learning Bounded Treewidth Bayesian Networks*. In *Journal of Machine Learning Research* 9(91), 2008: pp. 2699–2731

[54] Emmert-Streib, F.; Dehmer, M.; Haibe-Kains, B.: *Gene Regulatory Networks and Their Applications: Understanding Biological and Medical Problems in Terms of Networks*. In *Frontiers in Cell and Developmental Biology* 2(38), 2014: pp. 1–7

[55] Esmaeilzadeh, H.; Blem, E.; St. Amant, R.; Sankaralingam, K.; Burger, D.: *Dark Silicon and the End of Multicore Scaling*. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2011, pp. 365–376

[56] Fan, Z.; Qiu, F.; Kaufman, A.; Yoakum-Stover, S.: *GPU Cluster for High Performance Computing*. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*. IEEE, 2004, pp. 47–47

[57] Fang, W.; Lu, M.; Xiao, X.; He, B.; Luo, Q.: *Frequent Itemset Mining on Graphics Processors*. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2009, pp. 34–42

[58] Ferdman, M.; Hardavellas, N.; Ailamaki, A.; Falsafi, B.: *Toward Dark Silicon in Servers*. In *IEEE Micro* 31(04), 2011: pp. 6–15

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

[60] Foley, D.; Danskin, J.: *Ultra-performance Pascal GPU and NVLink Interconnect*. In *IEEE Micro* 37(2), 2017: pp. 7–17

[61] Foster, I. T.: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995

[62] Frawley, W. J.; Piatetsky-Shapiro, G.; Matheus, C. J.: *Knowledge Discovery in Databases: An Overview*. In *AI Magazine* 13(3), 1992: p. 57

[63] Frenzel, S.; Pompe, B.: *Partial Mutual Information for Coupling Analysis of Multivariate Time Series*. In *Physical Review Letters* 99(20), 2007: pp. 1–4

[64] Friedman, J. H.; Bentley, J. L.; Finkel, R. A.: *An Algorithm for Finding Best Matches in Logarithmic Expected Time*. In *ACM Transactions on Mathematical Software* 3(3), 1977: pp. 209–226

[65] Friedman, N.; Linial, M.; Nachman, I.; Pe'er, D.: *Using Bayesian Networks to Analyze Expression Data*. In *Journal of Computational Biology* 7(3-4), 2000: pp. 601–620

[66] Fukumizu, K.; Bach, F. R.; Jordan, M. I.: *Dimensionality Reduction for Supervised Learning with Reproducing Kernel Hilbert Spaces*. In *Journal of Machine Learning Research* 5, 2004: pp. 73–99

[67] Fukumizu, K.; Gretton, A.; Sun, X.; Schölkopf, B.: *Kernel Measures of Conditional Dependence*. In *Proceedings of the 20th International Conference on Neural Information Processing Systems (NIPS)*. Curran, 2007, pp. 489–496

[68] Funke, H.; Bress, S.; Noll, S.; Markl, V.; Teubner, J.: *Pipelined Query Processing in Coprocessor Environments*. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018, pp. 1603–1618

[69] Ganguly, D.; Zhang, Z.; Yang, J.; Melhem, R.: *Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory*. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 224–235

[70] Ganguly, D.; Zhang, Z.; Yang, J.; Melhem, R.: *Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription*. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 451–461

[71] Gao, W.; Kannan, S.; Oh, S.; Viswanath, P.: *Estimating Mutual Information for Discrete-Continuous Mixtures*. In *Advances in Neural Information Processing Systems* 30, 2017: pp. 1–12

[72] Garcia, V.; Debreuve, E.; Nielsen, F.; Barlaud, M.: *K-nearest Neighbor Search: Fast GPU-Based Implementations and Application to*

High-Dimensional Feature Matching. In *Proceedings of the International Conference on Image Processing, (ICIP)*. IEEE, 2010, pp. 3757–3760

[73] Gieseke, F.; Heinermann, J.; Oancea, C.; Igel, C.: *Buffer K-d Trees: Processing Massive Nearest Neighbor Queries on GPUs*. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML)*. 2014, pp. 172–180

[74] Glymour, C.; Zhang, K.; Spirtes, P.: *Review of Causal Discovery Methods Based on Graphical Models*. In *Frontiers in Genetics* 10(524), 2019: pp. 1–15

[75] Greenlaw, R.; Hoover, H. J.; Ruzzo, W. L.: *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995

[76] Gregg, C.; Hazelwood, K.: *Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer*. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2011, pp. 134–144

[77] Gu, J.; Zhou, Q.: *Learning Big Gaussian Bayesian Networks: Partition, Estimation and Fusion*. In *Journal of Machine Learning Research* 21(158), 2020: pp. 1–31

[78] Guo, C.; Luk, W.: *Accelerating Constraint-Based Causal Discovery by Shifting Speed Bottleneck*. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2022, pp. 169–179

[79] Guo, R.; Cheng, L.; Li, J.; Hahn, P. R.; Liu, H.: *A Survey of Learning Causality with Data: Problems and Methods*. In *ACM Computing Surveys* 53(4), 2020: pp. 1–37

[80] Hagedorn, C.; Huegle, J.: *Constraint-Based Causal Structure Learning in Multi-GPU Environments*. In *Proceedings of the LWDA 2021 Workshops: FGWM, KDML, FGWI-BIA, and FGIR*. CEUR-WS.org, 2021, pp. 106–118

[81] Hagedorn, C.; Huegle, J.: *GPU-Accelerated Constraint-Based Causal Structure Learning for Discrete Data*. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 2021, pp. 37–45

[82] Hagedorn, C.; Huegle, J.; Schlosser, R.: *Understanding Unforeseen Production Downtimes in Manufacturing Processes Using Log Data-Driven Causal Reasoning*. In *Journal of Intelligent Manufacturing* 33(7), 2022: pp. 2027–2043

[83] Hagedorn, C.; Lange, C.; Huegle, J.; Schlosser, R.: *GPU Acceleration for Information-theoretic Constraint-based Causal Discovery*. In *Proceedings of The KDD'22 Workshop on Causal Discovery*. PMLR, 2022, pp. 30–60

[84] Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; Fernández del Río, J.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; Oliphant, T. E.: *Array Programming with NumPy*. In *Nature* 585, 2020: pp. 357–362

[85] Harris, M.: *Unified Memory in CUDA 6*. `https://developer.nvidia.com/blog/unified-memory-in-cuda-6/`, 2013. Accessed: June 24, 2022

[86] Harris, M.: *Inside Pascal: NVIDIA's Newest Computing Platform*. https://developer.nvidia.com/blog/inside-pascal/, 2016. Accessed: June 24, 2022

[87] Harris, N.; Drton, M.: *PC Algorithm for Nonparanormal Graphical Models*. In *Journal of Machine Learning Research* 14(69), 2013: pp. 3365–3383

[88] Hartigan, J. A.: *Consistency of Single Linkage for High-Density Clusters*. In *Journal of the American Statistical Association* 76(374), 1981: pp. 388–394

[89] Hauser, A.; Bühlmann, P.: *Characterization and Greedy Learning of Interventional Markov Equivalence Classes of Directed Acyclic Graphs*. In *Journal of Machine Learning Research* 13(1), 2012: pp. 2409–2464

[90] Heckerman, D.; Geiger, D.; Chickering, D. M.: *Learning Bayesian Networks: The Combination of Knowledge and Statistical Data*. In *Machine Learning* 20(3), 1995: pp. 197–243

[91] Heinze-Deml, C.; Maathuis, M. H.; Meinshausen, N.: *Causal Structure Learning*. In *Annual Review of Statistics and Its Application* 5(1), 2018: pp. 371–391

[92] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann, 6. Edition, 2017

[93] Hennessy, J. L.; Patterson, D. A.: *A New Golden Age for Computer Architecture*. In *Communications of the ACM* 62(2), 2019: pp. 48–60

[94] Hlaváčková-Schindler, K.; Paluš, M.; Vejmelka, M.; Bhattacharya, J.: *Causality Detection Based on Information-Theoretic Approaches in Time Series Analysis*. In *Physics Reports* 441(1), 2007: pp. 1–46

[95] Huang, J.; Zhou, Q.: *Partitioned Hybrid Learning of Bayesian Network Structures*. In *Machine Learning* 111(5), 2022: pp. 1695–1738

[96] Huang, T.-M.: *Testing Conditional Independence using Maximal Nonlinear Conditional Correlation*. In *The Annals of Statistics* 38(4), 2010: pp. 2047–2091

[97] Huegle, J.: *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)*. IJCAI, 2021, pp. 4891–4892. Doctoral Consortium

[98] Huegle, J.; Hagedorn, C.; Böhme, L.; Pörschke, M.; Umland, J.; Schlosser, R.: *MANM-CS: Data Generation for Benchmarking Causal Structure Learning from Mixed Discrete-Continuous and Nonlinear Data*. In *WHY-21 @ NeurIPS*. WHY-21, 2021, pp. 1–15

[99] Huegle, J.; Hagedorn, C.; Perscheid, M.; Plattner, H.: *MPCSL – A Modular Pipeline for Causal Structure Learning*. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 2021, pp. 3068–3076

[100] Huegle, J.; Hagedorn, C.; Schlosser, R.: *A kNN-based Non-Parametric Conditional Independence Test for Mixed Data and Application in Causal Discovery*. In *ECML-PKDD 2023, accepted*. 2023

[101] Huegle, J.; Hagedorn, C.; Uflacker, M.: *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 (ICJAI). IJCAI, 2020, pp. 5246–5248

[102] Huegle, J.; Hagedorn, C.; Uflacker, M.: *Unterstützte Fehlerbehebung durch kausales Strukturwissen in Überwachungssystemen der Automobilfertigung*. In *Software Engineering 2021 Satellite Events, Lecture Notes in Informatics (LNI)*. Gesellschaft für Informatik, 2021, pp. 1–2

[103] Ismail Fawaz, H.; Forestier, G.; Weber, J.; Idoumghar, L.; Muller, P.-A.: *Deep Learning for Time Series Classification: A Review*. In *Data Mining and Knowledge Discovery* 33(4), 2019: pp. 917–963

[104] Jensen, C. S.; Kong, A.: *Blocking Gibbs Sampling for Linkage Analysis in Large Pedigrees with Many Loops*. In *The American Journal of Human Genetics* 65(3), 1999: pp. 885–901

[105] Jian, L.; Wang, C.; Liu, Y.; Liang, S.; Yi, W.; Shi, Y.: *Parallel Data Mining Techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)*. In *The Journal of Supercomputing* 64(3), 2013: pp. 942–967

[106] Jiang, J.; Wen, Z.; Mian, A.: *Fast Parallel Bayesian Network Structure Learning*. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 617–627

[107] Jouppi, N. P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; Boyle, R.; Cantin, P.-l.; Chao, C.; Clark, C.; Coriell, J.; Daley, M.; Dau, M.; Dean, J.; Gelb, B.; ...; Yoon, D. H.: *In-Datacenter Performance Analysis of a Tensor Processing Unit*. In *Proceedings of the 44^{th} Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 1–12

[108] Kabir, K.; Haidar, A.; Tomov, S.; Bouteiller, A.; Dongarra, J.: *A Framework for Out of Memory SVD Algorithms*. In *High Performance Computing (ISC)*. Springer, 2017, pp. 158–178

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

[110] Kalisch, M.; Bühlmann, P.: *Causal Structure Learning and Inference: A Selective Review*. In *Quality Technology & Quantitative Management* 11(1), 2014: pp. 3–21

[111] Kalisch, M.; Mächler, M.; Colombo, D.; Maathuis, M. H.; Bühlmann, P.: *Causal Inference Using Graphical Models with the R Package pcalg*. In *Journal of Statistical Software, Articles* 47(11), 2012: pp. 1–26

[112] Kim, H.; Sim, J.; Gera, P.; Hadidi, R.; Kim, H.: *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, pp. 1357–1370

[113] Kirk, D. B.; Hwu, W.-m. W.: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2. Edition, 2013

[114] Koivisto, M.; Sood, K.: *Exact Bayesian Structure Discovery in Bayesian Networks*. In *Journal of Machine Learning Resesarch* 5(5), 2004: pp. 549–573

[115] Koller, D.; Friedman, N.: *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009

[116] Kraskov, A.; Stögbauer, H.; Grassberger, P.: *Estimating Mutual Information*. In *Physical Review E* 69(066138), 2004: pp. 1–16

[117] Kühnert, C.; Beyerer, J.: *Data-Driven Methods for the Detection of Causal Structures in Process Technology*. In *Machines* 2(4), 2014: pp. 255–274

[118] Kummerfeld, E.; Ramsey, J.; Yang, R.; Spirtes, P.; Scheines, R.: *Causal Clustering for 2-Factor Measurement Models*. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. Springer, 2014, pp. 34–49

[119] Lahabar, S.; Narayanan, P. J.: *Singular Value Decomposition on GPU Using CUDA*. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2009, pp. 1–10

[120] Landaverde, R.; Tiansheng Zhang; Coskun, A. K.; Herbordt, M.: *An Investigation of Unified Memory Access Performance in CUDA*. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6

[121] Lauritzen, S. L.: *Graphical Models*, Volume 17. Clarendon Press Oxford, 1996

[122] Lauritzen, S. L.; Dawid, A. P.; Larsen, B. N.; Leimer, H.-G.: *Independence Properties of Directed Markov Fields*. In *Networks* 20(5), 1990: pp. 491–505

[123] Le, T. D.; Hoang, T.; Li, J.; Liu, L.; Liu, H.; Hu, S.: *A Fast PC Algorithm for High Dimensional Causal Discovery with Multi-Core PCs*. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16(5), 2019: pp. 1483–1495

[124] Le, T. D.; Liu, L.; Tsykin, A.; Goodall, G. J.; Liu, B.; Sun, B.-Y.; Li, J.: *Inferring MicroRNA–mRNA Causal Regulatory Relationships from Expression Data*. In *Bioinformatics* 29(6), 2013: pp. 765–771

[125] Le, T. D.; Liu, L.; Zhang, J.; Liu, B.; Li, J.: *From miRNA Regulation to miRNA-TF Co-regulation: Computational Approaches and Challenges*. In *Briefings in Bioinformatics* 16(3), 2015: pp. 475–496

[126] Le, T. D.; Xu, T.; Liu, L.; Shu, H.; Hoang, T.; Li, J.: *ParallelPC: An R Package for Efficient Causal Exploration in Genomic Data*. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. Springer, 2018, pp. 207–218

[127] LeCun, Y.; Bengio, Y.; Hinton, G.: *Deep Learning*. In *Nature* 521, 2015: pp. 436–444

[128] Lee, S.; Kim, S. B.: *Parallel Simulated Annealing with a Greedy Algorithm for Bayesian Network Structure Learning*. In *IEEE Transactions on Knowledge and Data Engineering* 32(6), 2019: pp. 1157–1166

[129] Lee Rodgers, J.; Nicewander, W. A.: *Thirteen Ways to Look at the Correlation Coefficient*. In *The American Statistician* 42(1), 1988: pp. 59–66

[130] Lehmann, E. L.; Romano, J. P.: *Testing Statistical Hypotheses*. Springer, 2006

[131] Lepak, K.; Talbot, G.; White, S.; Beck, N.; Naffziger, S.: *The Next Generation AMD Enterprise Server Product Architecture*. In *IEEE Hot Chips* 29, 2017

[132] Li, A.; Song, S. L.; Chen, J.; Li, J.; Liu, X.; Tallent, N. R.; Barker, K. J.: *Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect*. In *IEEE Transactions on Parallel and Distributed Systems* 31(1), 2020: pp. 94–110

[133] Li, C.; Ausavarungnirun, R.; Rossbach, C. J.; Zhang, Y.; Mutlu, O.; Guo, Y.; Yang, J.: *A Framework for Memory Oversubscription Management in Graphics Processing Units*. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 49–63

[134] Li, J.; Shi, J.: *Knowledge Discovery from Observational Data for Process Control using Causal Bayesian Networks*. In *Institute of Industrial Engineers Transactions* 39(6), 2007: pp. 681–690

[135] Li, J.; Wang, Z. J.: *Controlling the False Discovery Rate of the Association/Causality Structure Learned with the PC Algorithm*. In *Journal of Machine Learning Research* 10(17), 2009: pp. 475–514

[136] Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J.: *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. In *IEEE Micro* 28(2), 2008: pp. 39–55

[137] Luitjens, J.: *CUDA Streams: Best Practices and Common Pitfalls*. In *GPU Techonology Conference* 2015

[138] Lutz, C.; Bress, S.; Rabl, T.; Zeuch, S.; Markl, V.: *Efficient K-Means on GPUs*. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2018, pp. 1–3

[139] Lutz, C.; Bress, S.; Zeuch, S.; Rabl, T.; Markl, V.: *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects*. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2020, pp. 1633–1649

[140] Lutz, C.; Bress, S.; Zeuch, S.; Rabl, T.; Markl, V.: *Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects*. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. ACM, 2022, pp. 1017–1032

[141] Lv, Y.; Duan, Y.; Kang, W.; Li, Z.; Wang, F.-Y.: *Traffic Flow Prediction With Big Data: A Deep Learning Approach*. In *IEEE Transactions on Intelligent Transportation Systems* 16(2), 2015: pp. 865–873

[142] Maathuis, M.; Drton, M.; Lauritzen, S.; Wainwright, M.: *Handbook of Graphical Models*. CRC Press, 1. Edition, 2018

[143] Maathuis, M. H.; Colombo, D.; Kalisch, M.; Bühlmann, P.: *Predicting Causal Effects in Large-Scale Systems from Observational Data*. In *Nature Methods* 7(4), 2010: pp. 247–248

[144] Maathuis, M. H.; Kalisch, M.; Bühlmann, P.: *Estimating High-Dimensional Intervention Effects from Observational Data*. In *The Annals of Statistics* 37(6A), 2009: pp. 3133–3164

[145] Madsen, A. L.; Jensen, F.; Salmerón, A.; Karlsen, M.; Langseth, H.; Nielsen, T. D.: *A New Method for Vertical Parallelisation of TAN Learning Based on Balanced Incomplete Block Designs*. In *Probabilistic Graphical Models (PGM)*. Springer, 2014, pp. 302–317

[146] Madsen, A. L.; Jensen, F.; Salmerón, A.; Langseth, H.; Nielsen, T. D.: *Parallelisation of the PC Algorithm*. In *Advances in Artificial Intelligence (CAEPIA)*. Springer, 2015, pp. 14–24

[147] Madsen, A. L.; Jensen, F.; Salmerón, A.; Langseth, H.; Nielsen, T. D.: *A Parallel Algorithm for Bayesian Network Structure Learning from Large Data Sets*. In *Knowledge-Based Systems* 117, 2017: pp. 46–55

[148] Maier, M.; Taylor, B.; Oktay, H.; Jensen, D.: *Learning Causal Models of Relational Domains*. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2010, pp. 531–538

[149] Maldonado, G.; Greenland, S.: *Estimating Causal Effects*. In *International Journal of Epidemiology* 31(2), 2002: pp. 422–429

[150] Malinsky, D.; Danks, D.: *Causal Discovery Algorithms: A Practical Guide*. In *Philosophy Compass* 13(1), 2018: pp. 1–11

[151] Marazopoulou, K.; Ghosh, R.; Lade, P.; Jensen, D. D.: *Causal Discovery for Manufacturing Domains*. In *ArXiv* abs/1605.04056, 2016

[152] Marbach, D.; Costello, J.; Küffner, R.; Vega, N.; Prill, R.; Camacho, D.; Allison, K.; Kellis, M.; Collins, J.; Aderhold, A.; Stolovitzky, G.; et al.: *Wisdom of Crowds for Robust Gene Network Inference*. In *Nature Methods* 9(8), 2012: pp. 796–804

[153] Margaritis, D.: *Learning Bayesian Network Model Structure From Data*. Dissertation, School of Computer Science, Carnegie-Mellon University, 2003

[154] Margaritis, D.; Thrun, S.: *Bayesian Network Induction via Local Neighborhoods*. In *Advances in Neural Information Processing Systems*. MIT Press, 1999, pp. 505–511

[155] Markthub, P.; Belviranli, M. E.; Lee, S.; Vetter, J. S.; Matsuoka, S.: *DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access*. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 2018, pp. 1–13

[156] Marx, A.; Vreeken, J.: *Testing Conditional Independence on Discrete Data using Stochastic Complexity*. In *The 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2019, pp. 496–505

[157] Marx, A.; Yang, L.; van Leeuwen, M.: *Estimating Conditional Mutual Information for Discrete-Continuous Mixtures Using Multi-Dimensional Adaptive Histograms*. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. 2021, pp. 387–395

[158] Mattson, T.; Sanders, B.; Massingill, B.: *Patterns for Parallel Programming*. Addison-Wesley Professional, 1. Edition, 2004

[159] Meek, C.: *Causal Inference and Causal Explanation with Background Knowledge*. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 1995, pp. 403–410

[160] Mei, X.; Chu, X.: *Dissecting GPU Memory Hierarchy Through Microbenchmarking*. In *IEEE Transactions on Parallel and Distributed Systems* 28(1), 2017: pp. 72–86

[161] Mesner, O. C.; Shalizi, C. R.: *Conditional Mutual Information Estimation for Mixed, Discrete and Continuous Data*. In *IEEE Transactions on Information Theory* 67(1), 2021: pp. 464–484

[162] Micikevicius, P.: *Local Memory and Register Spilling*. `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`, 2011. Accessed: August 13, 2021

[163] Min, S. W.; Mailthody, V. S.; Qureshi, Z.; Xiong, J.; Ebrahimi, E.; Hwu, W.-m.: *EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs*. In *Proceedings of the VLDB Endowment* 14(2), 2020: pp. 114–127

[164] Mohri, M.; Rostamizadeh, A.; Talwalkar, A.: *Foundations of Machine Learning*. MIT Press, 2. Edition, 2018

[165] Moore, G. E.: *Cramming More Components onto Integrated Circuits*. In *Electronics* 38(8), 1965: pp. 114–117

[166] Mulnix, D. L.: *Intel® Xeon® Processor Scalable Family Technical Overview*. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html, 2019. Accessed: August 13, 2021

[167] Nandy, P.; Hauser, A.; Maathuis, M. H.: *High-Dimensional Consistency in Score-Based and Hybrid Structure Learning*. In *The Annals of Statistics* 46(6A), 2018: pp. 3151–3183

[168] Nandy, P.; Maathuis, M. H.; Richardson, T. S.: *Estimating the Effect of Joint Interventions from Observational Data in Sparse High-Dimensional Settings*. In *The Annals of Statistics* 45(2), 2017: pp. 647–674

[169] Nguyen, T.; Nguyen, D. T.; Le, T. D.; Venkatesh, S.: *MrPC: Causal Structure Learning in Distributed Systems (ICONIP)*. In *Neural Information Processing*. Springer, 2020, pp. 87–94

[170] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K.: *Scalable Parallel Programming with CUDA*. In *ACM SIGGRAPH 2008 Classes*. ACM, 2008, pp. 1–14

[171] Nielsen, J. D.; Kocka, T.; Peña, J. M.: *On Local Optima in Learning Bayesian Networks*. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 2003, pp. 435–442

[172] Nikolova, O.; Aluru, S.: *Parallel Discovery of Direct Causal Relations and Markov Boundaries with Applications to Gene Networks*. In *2011 International Conference on Parallel Processing (ICPP)*. IEEE, 2011, pp. 512–521

[173] Nordquist, B. S.; Lew, S. D.: *Apparatus, System, and Method for Coalescing Parallel Memory Requests*, 2009. Patent No. US7492368B1, Filed Jan. 24th, 2006, Issued Feb. 17th, 2009

[174] NVIDIA Corporation: *NVIDIA TESLA V100 GPU ACCELERATOR*. http://www.nvidia.com/content/PDF/Volta-Datasheet.pdf, 2017. Accessed: February 12, 2018

[175] NVIDIA Corporation: *Profiler User's Guide*. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf, 2018. Accessed: March 10, 2018

[176] NVIDIA Corporation: *cuBLAS Library*. https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf, 2022. Accessed: February 10, 2022

[177] NVIDIA Corporation: *CUDA Math API*. http://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf, 2022. Accessed: February 07, 2022

[178] NVIDIA Corporation: *cuRAND Library*. https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf, 2022. Accessed: June 06, 2022

[179] NVIDIA Corporation: *cuSOLVER Library*. https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf, 2022. Accessed: February 10, 2022

[180] NVIDIA Corporation: *NVIDIA A40 Powerful Data Center GPU For Visual Computing*. `https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf`, 2022. Accessed: September 22, 2022

[181] Ogarrio, J. M.; Spirtes, P.; Ramsey, J.: *A Hybrid Causal Search Algorithm for Latent Variable Models*. In *Conference on Probabilistic Graphical Models (PGM)*. PMLR, 2016, pp. 368–379

[182] Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C.: *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017

[183] Omohundro, S. M.: *Five Balltree Construction Algorithms*. Technical report, International Computer Science Institute Berkeley, 1989

[184] Parviainen, P.; Koivisto, M.: *Bayesian Structure Discovery in Bayesian Networks with Less Space*. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2010, pp. 589–596

[185] Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; Chintala, S.: *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In *Advances in Neural Information Processing Systems 32*, Curran. 2019, pp. 8024–8035

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

[187] Pearl, J.: *Causal Diagrams for Empirical Research*. In *Biometrika* 82(4), 1995: pp. 669–688

[188] Pearl, J.: *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2. Edition, 2009

[189] Pearl, J.; Verma, T.: *A Theory of Inferred Causation*. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Morgan Kaufmann, 1991, pp. 441–452

[190] Pearson, K. F.: *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*. In *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50(302), 1900: pp. 157–175

[191] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Édouard Duchesnay: *Scikit-learn: Machine Learning in Python*. In *Journal of Machine Learning Research* 12(85), 2011: pp. 2825–2830

[192] Penrose, R.: *A Generalized Inverse for Matrices*. In *Mathematical Proceedings of the Cambridge Philosophical Society*. Cambridge University Press, 1955, pp. 406–413

[193] Perscheid, C.; Grasnick, B.; Uflacker, M.: *Integrative Gene Selection on Gene Expression Data: Providing Biological Context to Traditional*

Approaches. In *Journal of Integrative Bioinformatics* 16(1), 2018: pp. 1–17

[194] Peters, J.; Janzing, D.; Schölkopf, B.: *Elements of Causal Inference – Foundations and Learning Algorithms*. MIT Press, 2017

[195] Peters, J.; Mooij, J. M.; Janzing, D.; Bernhard, S.: *Causal Discovery with Continuous Additive Noise Models*. In *Journal of Machine Learning Research* 15(58), 2014: pp. 2009–2053

[196] Petersen, M. L.; van der Laan, M. J.: *Causal Models and Learning from Data: Integrating Causal Modeling and Statistical Estimation*. In *Epidemiology* 25(3), 2014: pp. 418–426

[197] Putnam, A.; Caulfield, A. M.; Chung, E. S.; Chiou, D.; Constantinides, K.; Demme, J.; Esmaeilzadeh, H.; Fowers, J.; Gopal, G. P.; Gray, J.; Haselman, M.; Hauck, S.; Heil, S.; Hormati, A.; Kim, J.-Y.; Lanka, S.; Larus, J.; Peterson, E.; Pope, S.; Smith, A.; Thong, J.; Xiao, P. Y.; Burger, D.: *A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services*. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE, 2014, pp. 13–24

[198] R Core Team: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2020. `http://www.R-project.org/` Accessed: November 16, 2022

[199] Raghu, V. K.; Poon, A.; Benos, P. V.: *Evaluation of Causal Structure Learning Methods on Mixed Data Types*. In *Proceedings of 2018 ACM SIGKDD Workshop on Causal Disocvery*. PMLR, 2018, pp. 48–65

[200] Ramsey, J.: *A Scalable Conditional Independence Test for Nonlinear, Non-Gaussian Data*. In *ArXiv* abs/1401.5031, 2014

[201] Ramsey, J.: *Improving Accuracy and Scalability of the PC Algorithm by Maximizing P-value*. In *ArXiv* abs/1610.00378, 2016

[202] Ramsey, J.; Glymour, M.; Sanchez-Romero, R.; Glymour, C.: *A Million Variables and More: The Fast Greedy Equivalence Search Algorithm for Learning High-Dimensional Graphical Causal Models, with an Application to Functional Magnetic Resonance Images*. In *International Journal of Data Science and Analytics* 3(2), 2017: pp. 121–129

[203] Ramsey, J. D.; Zhang, J.; Spirtes, P.: *Adjacency-Faithfulness and Conservative Causal Inference*. In *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2006, pp. 401–408

[204] RAPIDS Development Team: *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. `https://rapids.ai` Accessed: November 16, 2022

[205] Raschka, S.; Patterson, J.; Nolet, C.: *Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence*. In *Information* 11(4), 2020: pp. 1–44

[206] Rau, A.; Jaffrézic, F.; Nuel, G.: *Joint Estimation of Causal Effects from Observational and Intervention Gene Expression Data*. In *BMC Systems Biology* 7(1), 2013: pp. 1–12

[207] Richardson, T.: *A Discovery Algorithm for Directed Cyclic Graphs*. In *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 1996, pp. 454–461

[208] Robinson, R. W.: *Counting Unlabeled Acyclic Digraphs*. In *Combinatorial Mathematics V*. Springer, 1977, pp. 28–43

[209] Roscher, R.; Bohn, B.; Duarte, M. F.; Garcke, J.: *Explainable Machine Learning for Scientific Insights and Discoveries*. In *IEEE Access* 8, 2020: pp. 42200–42216

[210] Ross, B. C.: *Mutual Information between Discrete and Continuous Data Sets*. In *PloS one* 9(2), 2014: pp. 1–5

[211] Rothenhäusler, D.; Heinze, C.; Peters, J.; Meinshausen, N.: *BACKSHIFT: Learning Causal Cyclic Graphs from Unknown Shift Interventions*. In *Advances in Neural Information Processing Systems* 28, 2016: pp. 1–9

[212] Rubin, D. B.: *Randomization Analysis of Experimental Data: The Fisher Randomization Test Comment*. In *Journal of the American Statistical Association* 75(371), 1980: pp. 591–593

[213] Rui, R.; Li, H.; Tu, Y.-C.: *Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment*. In *Proceedings of the VLDB Endowment* 14(4), 2020: pp. 708–720

[214] Runge, J.: *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 (AISTATS)*. PMLR, 2018, Volume 84, pp. 938–947

[215] Runge, J.: *Discovering Contemporaneous and Lagged Causal Relations in Autocorrelated Nonlinear Time Series Datasets*. In *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI)*. PMLR, 2020, Volume 124, pp. 1388–1397

[216] Runge, J.; Nowack, P.; Kretschmer, M.; Flaxman, S.; Sejdinovic, D.: *Detecting and Quantifying Causal Associations in Large Nonlinear Time Series Datasets*. In *Science Advances* 5(11), 2019: pp. 1–15

[217] Sakharnykh, N.: *Beyond GPU Memory Limits with Unified Memory on Pascal*. `https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/`, 2016. Accessed: August 13, 2021

[218] Sakharnykh, N.: *Maximizing Unified Memory Performance in CUDA*. `https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/`, 2017. Accessed: July 29, 2022

[219] Sakharnykh, N.: *Memory Management on Modern GPU Architectures*. `https://developer.nvidia.com/gtc/2019/video/s9727`, 2019. Accessed: June 04, 2021

[220] Sanders, J.; Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010

[221] Sanderson, C.; Curtin, R.: *Armadillo: A Template-Based C Library for Linear Algebra*. In *Journal of Open Source Software* 1(2), 2016: pp. 1–26

[222] Schmidt, C.; Dreseler, M.; Akin, B.; Roy, A.: *A Case for Hardware-Supported Sub-Cache Line Accesses*. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2018, pp. 1–3

[223] Schmidt, C.; Huegle, J.: *Towards a GPU-Accelerated Causal Inference*. In *HPI Future SOC Lab – Proceedings 2017*. Universitätsverlag Potsdam, 2020, pp. 187–194

[224] Schmidt, C.; Huegle, J.; Bode, P.; Uflacker, M.: *Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data*. In *Proceedings of The 2019 KDD Workshop on Causal Discovery*. PMLR, 2019, pp. 59–77

[225] Schmidt, C.; Huegle, J.; Horschig, S.; Uflacker, M.: *Out-of-Core GPU-Accelerated Causal Structure Learning*. In *Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2020, pp. 89–104

[226] Schmidt, C.; Huegle, J.; Uflacker, M.: *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)*. ACM, 2018, pp. 19:1–19:10

[227] Schmidt, C.; Uflacker, M.: *Workload-Driven Data Placement for GPU-Accelerated Database Management Systems*. In *BTW 2019 – Workshopband*. Gesellschaft für Informatik, 2019, pp. 91–94

[228] Schmidt, M.; Niculescu-Mizil, A.; Murphy, K.: *Learning Graphical Model Structure Using L1-Regularization Paths*. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2007, pp. 1278–1283

[229] Schulte, O.; Frigo, G.; Greiner, R.; Khosravi, H.: *The IMAP Hybrid Method for Learning Gaussian Bayes Nets*. In *Advances in Artificial Intelligence*. Springer, 2010, pp. 123–134

[230] Schwarz, C.; Schmidt, C.; Hopstock, M.; Sinzig, W.; Plattner, H.: *Efficient Calculation and Simulation of Product Cost Leveraging In-Memory Technology and Coprocessors*. In *The Sixth International Conference on Business Intelligence and Technology*. IARIA, 2016, pp. 12–18

[231] Schwarz, G.: *Estimating the Dimension of a Model*. In *The Annals of Statistics* 6(2), 1978: pp. 461–464

[232] Scutari, M.: *Learning Bayesian Networks with the bnlearn R Package*. In *Journal of Statistical Software* 35(3), 2010: pp. 1–22

[233] Scutari, M.: *Bayesian Network Repository*. `http://www.bnlearn.com/bnrepository`, 2012. Accessed: September 22, 2022

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

[235] Scutari, M.; Graafland, C. E.; Gutiérrez, J. M.: *Who Learns Better Bayesian Network Structures: Constraint-Based, Score-based or Hybrid Algorithms?*. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models (PGM)*. PMLR, 2018, pp. 416–427

[236] Sejdinovic, D.; Sriperumbudur, B.; Gretton, A.; Fukumizu, K.: *Equivalence of Distance-Based and RKHS-Based Statistics in Hypothesis Testing*. In *The Annals of Statistics* 41(5), 2013: pp. 2263–2291

[237] Sen, R.; Suresh, A. T.; Shanmugam, K.; Dimakis, A. G.; Shakkettai, S.: *Model-Powered Conditional Independence Test*. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*. Curran, 2017, pp. 2955–2965

[238] Shah, R. D.; Peters, J.: *The Hardness of Conditional Independence Testing and the Generalised Covariance Measure*. In *The Annals of Statistics* 48(3), 2020: pp. 1514–1538

[239] Shahbazinia, A.; Salehkaleybar, S.; Hashemi, M.: *ParaLiNGAM: Parallel Causal Structure Learning for Linear non-Gaussian Acyclic Models*. In *Journal of Parallel and Distributed Computing* 176, 2023: pp. 114–127

[240] Sharma, G.; Agarwala, A.; Bhattacharya, B.: *A Fast Parallel Gauss Jordan Algorithm for Matrix Inversion Using CUDA*. In *Computers & Structures* 128, 2013: pp. 31–37

[241] Shimizu, S.; Hoyer, P. O.; Hyvärinen, A.; Kerminen, A.: *A Linear Non-Gaussian Acyclic Model for Causal Discovery*. In *Journal of Machine Learning Research* 7, 2006: pp. 2003–2030

[242] Shpitser, I.; Evans, R. J.; Richardson, T. S.; Robins, J. M.: *Sparse Nested Markov Models with Log-Linear Parameters*. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2013, pp. 576–585

[243] Silander, T.; Myllymäki, P.: *A Simple Approach for Finding the Globally Optimal Bayesian Network Structure*. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2006, pp. 445–452

[244] Singh, K.; Gupta, G.; Tewari, V.; Shroff, G.: *Comparative Benchmarking of Causal Discovery Algorithms*. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data (CoDS-COMAD)*. ACM, 2018, pp. 46–56

[245] Singh, M.; Valtorta, M.: *Construction of Bayesian Network Structures from Data: A Brief Survey and an Efficient Algorithm*. In *International Journal of Approximate Reasoning* 12(2), 1995: pp. 111–131

[246] Sondhi, A.; Shojaie, A.: *The Reduced PC-Algorithm: Improved Causal Structure Learning in Large Random Networks*. In *Journal of Machine Learning Research* 20(164), 2019: pp. 1–31

[247] Spirtes, P.: *Introduction to Causal Inference*. In *Journal of Machine Learning Research* 11(54), 2010: pp. 1643–1662

[248] Spirtes, P.: *Calculation of Entailed Rank Constraints in Partially Non-Linear and Cyclic Models*. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2013, pp. 606–615

[249] Spirtes, P.; Glymour, C.: *An Algorithm for Fast Recovery of Sparse Causal Graphs*. In *Social Science Computer Review* 9(1), 1991: pp. 62–72

[250] Spirtes, P.; Glymour, C.; Scheines, R.: *From Probability to Causality*. In *Philosophical Studies* 64(1), 1991: pp. 1–36

[251] Spirtes, P.; Glymour, C.; Scheines, R.: *Causation, Prediction, and Search*. MIT Press, 2. Edition, 2000

[252] Spirtes, P.; Zhang, J.: *A Uniformly Consistent Estimator of Causal Effects under the k-Triangle-Faithfulness Assumption*. In *Statistical Science* 29(4), 2014: pp. 662–678

[253] Springer, M.; Masuhara, H.: *Massively Parallel GPU Memory Compaction*. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, 2019, pp. 14–26

[254] Srivastava, A.; Chockalingam, S. P.; Aluru, S.: *A Parallel Framework for Constraint-Based Bayesian Network Learning via Markov Blanket Discovery*. In *Proceedings of the International Conference for High*

Performance Computing, Networking, Storage and Analysis (SC). IEEE, 2020, pp. 1–15

[255] Stark, R.; Grzelak, M.; Hadfield, J.: *RNA Sequencing: The Teenage Years*. In *Nature Reviews Genetics* 20(11), 2019: pp. 631–656

[256] Stekhoven, D. J.; Moraes, I.; Sveinbjörnsson, G.; Hennig, L.; Maathuis, M. H.; Bühlmann, P.: *Causal Stability Ranking*. In *Bioinformatics* 28(21), 2012: pp. 2819–2823

[257] Stone, J. E.; Gohara, D.; Shi, G.: *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*. In *Computing in Science Engineering* 12(3), 2010: pp. 66–73

[258] Strobl, E. V.: *A Constraint-Based Algorithm for Causal Discovery with Cycles, Latent Variables and Selection Bias*. In *International Journal of Data Science and Analytics* 8(1), 2019: pp. 33–56

[259] Strobl, E. V.; Spirtes, P. L.; Visweswaran, S.: *Estimating and Controlling the False Discovery Rate of the PC Algorithm Using Edge-Specific P-Values*. In *ACM Transactions on Intelligent Systems and Technology* 10(5), 2019: pp. 1–37

[260] Strobl, E. V.; Zhang, K.; Visweswaran, S.: *Approximate Kernel-Based Conditional Independence Tests for Fast Non-Parametric Causal Discovery*. In *Journal of Causal Inference* 7(1), 2019: pp. 1–24

[261] Tao, F.; Qi, Q.; Liu, A.; Kusiak, A.: *Data-Driven Smart Manufacturing*. In *Journal of Manufacturing Systems* 48, 2018: pp. 157–169

[262] techpowerup: *NVIDIA GeForce GTX TITAN X*. `https://www.techpowerup.com/gpu-specs/geforce-gtx-titan-x.c2632`, 2016. Accessed: September 22, 2022

[263] Thibault, J.; Senocak, I.: *CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows*. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*. 2009, pp. 1–15

[264] Thornton, T.; McPeek, M. S.: *Case-Control Association Testing with Related Individuals: A More Powerful Quasi-Likelihood Score Test*. In *The American Journal of Human Genetics* 81(2), 2007: pp. 321–337

[265] Triantafillou, S.; Tsamardinos, I.: *Constraint-Based Causal Discovery from Multiple Interventions over Overlapping Variable Sets*. In *Journal of Machine Learning Research* 16(1), 2015: pp. 2147–2205

[266] Tsagris, M.; Borboudakis, G.; Lagani, V.; Tsamardinos, I.: *Constraint-Based Causal Discovery with Mixed Data*. In *International Journal of Data Science and Analytics* 6(1), 2018: pp. 19–30

[267] Tsamardinos, I.; Aliferis, C. F.; Statnikov, A. R.: *Algorithms for Large Scale Markov Blanket Discovery*. In *FLAIRS conference*. AAAI Press, 2003, pp. 376–380

[268] Tsamardinos, I.; Borboudakis, G.: *Permutation Testing Improves Bayesian Network Learning*. In *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part III (ECML PKDD)*. Springer, 2010, pp. 322–337

[269] Tsamardinos, I.; Brown, L. E.; Aliferis, C. F.: *The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm*. In *Machine Learning* 65(1), 2006: pp. 31–78

[270] van der Aalst, W.: *Data Science in Action*. In *Process Mining*, Springer. 2016, pp. 3–23

[271] Vejmelka, M.; Paluš, M.: *Inferring the Directionality of Coupling with Conditional Mutual Information*. In *Physical Review E* 77(2), 2008: pp. 1–12

[272] Verma, T.; Pearl, J.: *Causal Networks: Semantics and Expressiveness*. In *Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence (UAI)*. 1988, pp. 69–78

[273] Verma, T.; Pearl, J.: *Equivalence and Synthesis of Causal Models*. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence (UAI)*. Elsevier, 1990, pp. 255–270

[274] Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Millman, K. J.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C. J.; Polat, İ.; Feng, Y.; Moore, E. W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; SciPy 1.0 Contributors: *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*. In *Nature Methods* 17, 2020: pp. 261–272

[275] Vowels, M. J.; Camgoz, N. C.; Bowden, R.: *D'ya Like DAGs? A Survey on Structure Learning and Causal Discovery*. In *ACM Computing Surveys* 55(4), 2022: pp. 1–36

[276] Wang, Y.; Davidson, A.; Pan, Y.; Wu, Y.; Riffel, A.; Owens, J. D.: *Gunrock: A High-Performance Graph Processing Library on the GPU*. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2016, pp. 1–12

[277] Whittaker, J.: *Graphical Models in Applied Multivariate Statistics*. Wiley Publishing, 2009

[278] Widmer, S.; Wodniok, D.; Weber, N.; Goesele, M.: *Fast Dynamic Memory Allocator for Massively Parallel Architectures*. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*. ACM, 2013, pp. 120–126

[279] Winship, C.; Morgan, S. L.: *The Estimation of Causal Effects from Observational Data*. In *Annual Review of Sociology* 25(1), 1999: pp. 659–706

[280] Wu, J.; JáJá, J.: *Achieving Native GPU Performance for Out-of-Card Large Dense Matrix Multiplication*. In *Parallel Processing Letters* 26(2), 2016: pp. 1–17

[281] Wu, L.; Lottarini, A.; Paine, T. K.; Kim, M. A.; Ross, K. A.: *Q100: The Architecture and Design of a Database Processing Unit*. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 255–268

[282] Yin, X.; Hong, L.: *The Identification and Estimation of Direct and Indirect Effects in A/B Tests through Causal Mediation Analysis*. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 2019, pp. 2989–2999

[283] Yoon, M. K.; Kim, K.; Lee, S.; Ro, W. W.; Annavaram, M.: *Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Schedul-*

ing Limit. In *SIGARCH Computer Architecture News* 44(3), 2016: pp. 609–621

[284] Yu, Q.; Childers, B.; Huang, L.; Qian, C.; Wang, Z.: *A Quantitative Evaluation of Unified Memory in GPUs*. In *Journal of Supercomputing* 76(4), 2020: pp. 2958–2985

[285] Zaharia, M.; Xin, R. S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M. J.; et al.: *Apache Spark: A Unified Engine for Big Data Processing*. In *Communications of the ACM* 59(11), 2016: pp. 56–65

[286] Zan, L.; Meynaoui, A.; Assaad, C. K.; Devijver, E.; Gaussier, E.: *A Conditional Mutual Information Estimator for Mixed Data and an Associated Conditional Independence Test*. In *Entropy* 24(9), 2022: pp. 1–23

[287] Zarebavani, B.; Jafarinejad, F.; Hashemi, M.; Salehkaleybar, S.: *cuPC: CUDA-Based Parallel PC Algorithm for Causal Structure Learning on GPU*. In *IEEE Transactions on Parallel and Distributed Systems* 31(3), 2020: pp. 530–542

[288] Zhang, H.; Zhou, S.; Guan, J.; Huan, J. L.: *Measuring Conditional Independence by Independent Residuals for Causal Discovery*. In *ACM Transactions on Intelligent Systems Technology* 10(5), 2019: pp. 1–19

[289] Zhang, K.; Peters, J.; Janzing, D.; Schölkopf, B.: *Kernel-Based Conditional Independence Test and Application in Causal Discovery*. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2011, pp. 804–813

[290] Zhang, K.; Tian, C.; Zhang, K.; Johnson, T.; Jiang, X.: *A Fast PC Algorithm with Reversed-order Pruning and A Parallelization Strategy*. In *ArXiv* abs/2109.04626, 2021

[291] Zhang, Q.; Filippi, S.; Flaxman, S. R.; Sejdinovic, D.: *Feature-to-Feature Regression for a Two-Step Conditional Independence Test*. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, 2017, pp. 1–10

[292] Zheng, T.; Nellans, D. W.; Zulfiqar, A.; Stephenson, M.; Keckler, S. W.: *Towards High Performance Paged Memory for GPUs*. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 345–357

[293] Zheng, X.; Aragam, B.; Ravikumar, P.; Xing, E. P.: *DAGs with NO TEARS: Continuous Optimization for Structure Learning*. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*. Curran, 2018, pp. 9492–9503

Eigenständigkeitserklärung
Declaration of Authorship

Hiermit versichere ich an Eides statt, dass die vorliegende Arbeit bisher an keiner anderen Hochschule eingereicht worden ist sowie selbständig und ausschließlich mit den angegebenen Mitteln angefertigt worden ist. Die Stellen der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, sind durch Angaben und Quellen kenntlich gemacht.

Potsdam, 19. Dezember 2022

Christopher Hagedorn geb. Schmidt