Heterogeneous Computing for Constraint-based Causal Structure Learning

Tobias Maltenberger
tobias.maltenberger@student.hpi.de
Hasso Plattner Institute

Christian Warmuth
christian.warmuth@student.hpi.de
Hasso Plattner Institute

ABSTRACT

Learning cause-effect relationships is a central goal in many research areas. Various techniques and algorithms exist to learn causal structures solely based on observational data since interventions are often infeasible for real-world applications. The PC algorithm is a constraint-based causal structure learning algorithm to learn the underlying causal structure of observed variables by performing conditional independence tests (CI-tests). Motivated by the steady advances in hardware technology, purely CPU- or GPU-based parallel implementations of the PC algorithm such as pcalg and cuPC do not harness the computational power of all processing units available in a heterogeneous setting.

In this paper, we study heterogeneous CPU-GPU computing for constraint-based causal structure learning. We evaluate two state-of-the-art implementations of the PC algorithm (i.e., pcalg and cuPC) and derive potential performance improvements by utilizing the available heterogeneous setup. We propose a general approach for heterogeneous constraint-based causal structure learning using static workload-partitioning. The core idea is to perform the CI-tests simultaneously on both CPU and GPU and merge the results afterwards. We provide a reference implementation based on pcalg and cuPC for our heterogeneous approach. Our experimental evaluation shows that for large problem sizes the additional overhead of the heterogeneous setting becomes negligible and we achieve comparable performance to modern PC algorithm implementations while utilizing the available resources of both CPU and GPU.

CCS CONCEPTS

• **Computing methodologies** → **Causal reasoning and diagnostics**.

KEYWORDS

Causal Structure Learning, Causal Discovery, Causality, PC Algorithm, Heterogeneous Computing, Parallel Programming, CUDA

1 INTRODUCTION

Learning causal structures between variables is of primary interest in empirical sciences. This enables understanding of causal relationships between observed variables and predicting the impact of different actions in a complex system without time-intensive, costly, and sometimes infeasible interventions [27, 34]. In causal structure learning, we try to learn causal relations from data. Causal graphical models represent causal relationships in the form of directed acyclic graphs (DAGs) in which a node represents a random variable. A directed edge between two nodes in the graph exists if and only if there is a direct causal relationship. Causal structure learning is the task of learning all DAGs coherently with the observational data at hand. Constraint-based approaches perform several conditional independence (CI) tests to find the DAG's underlying undirected graph (also called skeleton) in a process known as adjacency search. After that, we orient the undirected edges by the recurring application of deterministic rules. Sprites et al. [34] propose the PC algorithm, which runs in polynomial time with respect to the number of vertices in a bounded degree graph. The original PC algorithm is thereby order-dependent, which prohibits parallel execution as results depend on the ordering of performed CI-tests. PC-stable is an order-independent version of the classic PC algorithm [5], which enables parallel execution.

Parallelized order-independent adaptations of the original PC algorithm are proposed with ParallelPC [16] which is implemented in the pcalg package [13] and in the bnlearn package [32] for R as well as cuPC [38] on the GPU. With numbers of computing cores in the low two-digit range, CPUs are optimized for efficient sequential code execution (for branch prediction or caching). Graphical processing units (GPUs) are optimized for highly-parallel arithmetic operations on hundreds or thousands of cores [8].

The performance of existing PC implementations running in parallel exclusively on one processing unit (i.e., CPU or GPU) is bound by the power of the respective processing unit and algorithmic optimizations [13, 38]. However, when running on one processing unit for a descent problem size, it already fully utilizes the PU while leaving the other processing unit idle. Motivated by long runtimes, even for parallelized approaches and idle PUs during processing, we analyze bottlenecks of CPU- or GPU-only strategies for constraint-based causal structure learning with pcalg and cuPC. We then propose a novel heterogeneous approach for constraint-based causal structure learning harnessing both CPU and GPU for a heterogeneous computing setup. Furthermore, we implement this heterogeneous approach based on cuPC and pcalg and experimentally evaluate the approach with focus on the implications of a static workload-partitioning-strategy.

The rest of the paper is structured as follows: Section 2 lays the theoretical foundation on causal graphical models and the PC algorithm. Section 3 covers two prominent CPU- and GPU-based order-independent implementations of PC algorithm: pcalg and cuPC. Moreover, we discuss related work for hybrid CPU-GPU algorithms and workload-partitioning-strategies in Section 4. Section 5 motivates, proposes, and elaborates on the implementation of our heterogeneous approach for constraint-based causal structure learning. We conduct our experiments to evaluate the novel

approach's performance and suitability, which is the topic of Section 6. Section 7 summarizes the contributions to the field of causal structure learning and outlines future work.

2 THEORETICAL FOUNDATION

This section presents the theoretical foundation on causal graphical models, causal structure learning, and the PC algorithm needed for the following elaborations.

2.1 Causal Graphical Model

Let $G = (V, E)$ be a graph consisting of a finite set of n vertices V and a set of edges $E \subseteq V \times V$. An edge $(V_i, V_j) \in E$ is called directed (i.e, $V_i \rightarrow V_j$) if $(V_i, V_j) \in E$ and $(V_j, V_i) \notin E$. If $(V_j, V_i) \in E$ and $(V_i, V_j) \in E$, then the edge is called undirected (i.e., $V_i - V_j$). If an edge exists between two vertices, the vertices are adjacent.

In causal structure learning, a directed edge $V_i \rightarrow V_j$ implies that V_i is a direct cause of V_j [34]. If all edges are directed and no cycles are contained in the graph G, the graph is called directed acyclic graph (DAG). The skeleton of a graph G contains all nodes and edges of graph G but replaces them with undirected edges. We call a triple (V_i, V_j, V_k) unshielded if V_i and V_j as well as V_j and V_k are adjacent in G but V_i and V_k are not adjacent. A v-structure describes an unshielded triple (V_i, V_j, V_k) if they are oriented as such $V_i \rightarrow V_j \leftarrow V_k$. The notation $V_i \perp\!\!\!\perp V_j \mid S$ implies that V_i is independent of V_j given $S \subseteq V \setminus \{V_i, V_j\}$. We refer to S as a separation set for the edge (V_i, V_j) if $V_i \perp\!\!\!\perp V_j \mid S$.

The Markov equivalence class of DAGs are DAGs with the same skeleton and v-structures. This Markov equivalence class can be uniquely described by a completed partially directed acyclic graph (CPDAG) with both directed and undirected edges [26]. From here on, we call the different factors in a causal model "variables". In the context of graphical models, we refer to them as "nodes".

Causal structure learning aims to find the CPDAG that best describes the causal relationships for the m samples of n random variables. For causal structure learning, we make following assumptions where P denotes the joint probability distribution of V [34]:

Causal Markov Condition: G and P satisfy the causal Markov condition if and only if a node of G is independent of all its non-descendants given all its parents.

Causal Faithfulness Condition: G and P fulfill the causal faithfulness condition if and only if conditional independence is only implied by the causal Markov condition.

Causal Sufficiency Condition: Causal sufficiency holds if all common causes of the observed values are included in the data.

Causal structure learning can be divided into three paradigms: Score-based methods for causal structure learning evaluate all possible DAGs based on some defined score function as the likelihood. Score-based methods may become an NP-hard problem [6] and are therefore not suitable for high dimensional datasets. Examples of score-based algorithms are GES [3] and Chow-Liu [4]. Constraint-based approaches perform several conditional independence tests (CI-tests) to find the underlying undirected skeleton of the DAG in a process, also known as adjacency search. Hybrid algorithms combine score-based approaches with either variables selection or constraint-based approaches. Variable selection or constraint-based methods lead to smaller search spaces for the score-based approach [24]. In this paper, we only focus on constraint-based causal structure learning algorithms.

2.2 PC Algorithm

In a naive solution for constraint-based approach, one could perform all CI-tests in the form of $I(V_i, V_j \mid S)$ for any $S \subseteq V \setminus \{V_i, V_j\}$ to determine, if an edge exists between V_i and V_j (i.e., if V_i, V_j are adjacent in G) [34]. However, this naive approach is infeasible due to the exponentially growing number of conditional independence tests for large numbers of nodes n in the model [16].

A prominent constraint-based algorithm is the Inductive Causation (IC) algorithm proposed in [37], which is a conceptual algorithm. The conceptual algorithm IC is implemented in the PC algorithm, published by Sprites et al. [34]. The PC algorithm learns a Markov equivalence class of DAGs, which can be described by completed partially directed acyclic graphs (CPDAG). For high-dimensional settings, the PC algorithm is widely-used, as it is computationally feasible for sparse graphs containing up to thousands of variables. However, computational effort increases rapidly with the number of random variables n [10, 23].

The PC algorithm mainly consists of two steps. In the adjacency search, we search the graph containing only undirected edges, i.e., the skeleton, by performing CI-tests. In a second step, the remaining edges are directed by iteratively applying deterministic rules. In the following, we will focus on the adjacency search, as this entails the most computational costs [34] and existing implementations focus their improvements or parallelization on this aspect [13, 16, 38]. The adjacency search of the PC algorithm is organized in levels starting with level $l = 0$. The level indicates the size of the separation set S with which conditional independence is tested between two vertices (V_i, V_j). The level l is incremented once the edges are deleted after one level to reduce subsequent CI-tests for the following levels. In level $l = 0$, $S = \emptyset$, therefore we only test if $V_i \perp\!\!\!\perp V_j$.

In its original version, the PC algorithm [34] is order-dependent, i.e., the order in which variables are given affects the resulting skeleton. Here, we consider PC-stable, a variation of the original PC algorithm for which removing an edge in a level does not affect conditional independence tests on other edges in the same level [5]. The adjacency search of PC-stable algorithm can be summarised in pseudo code in Algorithm 1. A_G refers to the adjacency matrix which is used to encode the resulting undirected graph. If $A_G[i, j] = A_G[j, i] = 0$, there is no edge between vertex V_i and V_j. If $A_G[i, j] = A_G[j, i] = 1$, there exists an edge between V_i and V_j. The result of the adjacency search is the adjacency matrix which is symmetrical on the diagonal as we only output an undirected skeleton.

3 EXISTING CPU/GPU IMPLEMENTATIONS

In this section, we outline the characteristics of existing parallel CPU-/GPU-based PC-stable implementations. We focus on pcalg and cuPC later in this section as they are the basis for the implementation of our heterogeneous approach.

Algorithm 1: Adjacency Search of PC-stable [5]

1 **Input:** Vertex set V
2 **Output:** Skeleton G, separation set $SepSet$
3 G is fully connected, $l = 0$, $SepSet = \emptyset$
4 **repeat**
5 Copy G into G'
6 **for** any edge (V_i, V_j) **do**
7 **repeat**
8 Choose a new $S \subseteq adj(V_i, G') \setminus \{V_j\}$ such that $|S| = l$, where $adj(V_i, G')$ denotes the neighbors of V_j
9 Perform $I(V_i, V_j \mid S)$
10 **if** $V_i \perp\!\!\!\perp V_j \mid S$ **then**
11 Remove (V_i, V_j) from G by setting $A_G[i, j] = A_G[j, i] = 0$
12 Store S in $SepSet$
13 **end**
14 **until** (V_i, V_j) is removed or all sets S are considered
15 **end**
16 $l = l + 1$
17 **until** max degree $-1 \geq l$

A central limitation of the PC-algorithm for finding causal structures is the long runtime, which is, in the worst-case, exponential to the number of variables. Thus, optimizations and parallel implementations are proposed to harness modern hardware [13, 16, 20, 31, 38].

Le et al. [16] propose a parallel version of the order-independent extension PC-stable of the original PC algorithm which is implemented in the `ParallelPC` package in R [17]. In the `ParallelPC` implementation, tests for one edge are statically mapped to one CPU core to process within one level l in parallel. Furthermore, the `bnlearn` package in R [32] as well as the common package `pcalg` [13] provide a parallel implementation of PC-stable for multi-core CPUs with static workload distribution onto the execution units. In the following Section 3.1, we will focus on the popular `pcalg` implementation for a parallel PC-stable implementation for CPUs.

Schmidt et al. [31] propose a parallelized adjacency search making use of the parallel execution units of a GPU. They use static workload-partitioning; however, only focus on level $l = 0$ and level $l = 1$ and solely on Gaussian distributed data. cuPC [38], which will be elaborated on further in Section 3.2, also provides an optimized parallel implementation of PC-stable for GPU execution with multiple optimization strategies.

Madsen et al. [20] propose a parallel implementation for the PC-stable algorithm using the concept of Balanced Incomplete Blocks (BIB) [19, 35] for shared-memory computers. Madsen et al. focus on multinomial distributions and uses a ranking-strategy for higher-order CI-tests to perform the most promising CI-tests first.

3.1 pcalg: A CPU-Based Implementation

pcalg is a package for the R programming language [36], which can be used for causal structure learning and the causal effect estimation. The pcalg package contains multiple algorithms, including various implementations of the PC algorithm.

In the following, we focus on the PC-stable implementation in the pcalg package, which can be executed when calling *pc*(). When using `method="stable.fast"` as a parameter set by default, internally a C++ implementation of the PC-stable algorithm is called instead of a slower R-implementation [11].

For conditional independence calculation, the partial correlation matrix C is calculated, indicating correlation coefficients between variables. These partial correlations are Fisher's z-transformed, which is defined in Equation 1 and tested using a two-sided hypothesis test. In the setting of multivariate normal distribution, zero partial correlation is equivalent to conditional independence [12].

$$Z(\hat{p}(V_i, V_j \mid S)) = \left| \frac{1}{2} \times ln\left(\frac{1 + \hat{p}(V_i, V_j \mid S)}{1 - \hat{p}(V_i, V_j \mid S)} \right) \right| \tag{1}$$

If $Z(\hat{p}(V_i, V_j \mid S)) \leq \alpha$, we take $V_i \perp\!\!\!\perp V_j \mid S$ at significance level α. In level $l = 0$, this is simplified as only $Z(C[V_i, V_j])$ is compared with the threshold α. $C[V_i, V_j])$ denotes the (i, j)-th entry in the correlation matrix C over all n random variables.

pcalg uses openMP to execute the CI-tests in parallel. OpenMP, as an API for parallelized C++ programming [2], is used to perform the lines 6 - 15 of Algorithm 1 with `pragma omp parallel for` in parallel on all existing CPU cores.

3.2 cuPC: A GPU-Based Implementation

Zarebavani et al. propose with cuPC, a GPU-based parallel algorithm that executes an order-independent version of the PC algorithm [38]. cuPC assumes multivariate normal distributed data with no missing observations. cuPC is is based on PC-stable, an order-independent implementation of PC [38].

cuPC is implemented in C++ for the CUDA [25] framework which is wrapped in R code. CUDA is an architecture for massively parallel execution for GPU programming introduced by NVIDIA [14]. CUDA differentiates between host (CPU) and device (GPU) in a heterogeneous setting. Data-parallel device functions are called kernels. When such a kernel is launched, the code is executed parallelly by multiple threads on the GPU organized in a grid. The concept of threads enables parallelization. Threads are part of a running process while running concurrently and sharing resources such as memory or compute power. A grid in the context of GPUs is organized into thread blocks of the same size [15]. The cuPC paper describes two different versions of the CUDA-based parallel PC Algorithm: cuPC-S and cuPC-E. We base our considerations on cuPC-S for which an implementation exists on GitHub[1]. cuPC-S applies various optimizations, as compacting the adjacency matrix before using it in the next level. cuPC-S also applies a local-sharing strategy to reduce redundant computations for CI-tests with common conditional set S. Furthermore, cuPC uses GPU shared memory to improve performance. cuPC-S also applies an early-stopping-strategy, when some thread already removes an edge, the subsequent CI-tests on other parallel executions are skipped.

In contrast to pcalg, the current implementation of cuPC is only implemented up until level $l = 14$ [38]. Therefore, for level $l > 14$ pcalg and cuPC are not comparable. Unlike the implementation of pcalg, cuPC does not take α to compare Fisher's z-transform of

[1] github.com/LIS-Laboratory/cuPC

the partial correlations (in line 9 in Algorithm 1). cuPC computes a threshold of τ per level as given in Equation 2. Therefore, results from pcalg and cuPC are not comparable in their original implementation as tests in particular levels might result in different edge removals for cuPC and pcalg.

$$\tau = \frac{\Phi^{-1}(1 - \frac{\alpha}{2})}{\sqrt{m - |S| - 3}} \tag{2}$$

Φ denotes the cumulative distribution function of a standard normal distribution. If $Z(\hat{p}(V_i, V_j \mid S)) \leq \tau$, we take $V_i \perp\!\!\!\perp V_j \mid S$, i.e. V_i is independent from V_j given the separation set S. In level $l = 0$, this is simplified as only $Z(C[V_i, V_j])$ is compared with the threshold τ.

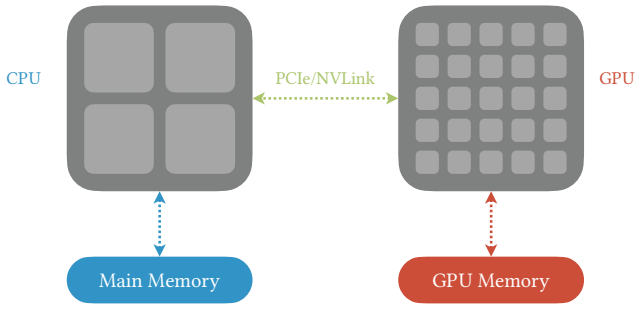


Figure 1: Heterogeneous computing with CPU and GPU

4 RELATED WORK

This chapter introduces the heterogeneous computing paradigm and workload-partitioning-strategies applicable to the scenario of causal structure learning in heterogeneous settings.

The heterogeneous computing paradigm aims to intelligently combine CPU with graphics processing units (GPUs) or field-programmable gate arrays (FPGAs). In all computing systems, a CPU is present as it is necessary to guarantee the correct functioning, while GPUs or FPGAs are optional. An FPGA is an integrated circuit in which logic blocks can be configured by the programmer to pursue specific and algorithm-tailored logical operations [22]. However, we leave FPGAs out of consideration in this context.

CPUs are highly optimized for task-parallelism and performance on sequential code as cache or branch prediction. With a number of cores in the low one- or two-digit range, CPUs are used to handle all computations and instructions on a computer. With current hardware advancements, CPUs are often supplemented by graphical processing units (GPUs) to speed up calculations. With hundreds or thousands of cores, GPUs are optimized for higher arithmetic intensive computations in parallel nature and often benefit CPU-only setups [8]. However, introducing GPUs as additional computing resources poses several challenges: workload must be partitioned, tasks need to be scheduled between the two PUs, and the results need to be synchronized afterward. This leads to organizational effort for cooperative problem solving and data transfer overhead between the two PUs as marked in blue in Figure 1. NVIDIA's NVLink technology and PCI Express (PCIe) are current technologies for transfer between CPU and GPU [1, 14, 18]. This leads to a transfer bottleneck as, for example, PCIe provides significantly lower bandwidth than the main memory.

To harness the strength of both PUs require optimizations and strategies different from traditional CPU- or GPU-centric approaches [21]. Heterogeneous computing aims to improve resource utilization by reducing idle times, decreasing energy consumption, and leveraging the structural strength of different PUs.

CPUs are often used as hosts for managing peripheral devices and task scheduling in a heterogeneous setting with both CPU and GPU. Therefore, the CPU sits almost idle and leaves potential untapped with regards to CPU-hardware-developments [21].

Processing on different PUs simultaneously and processing in parallel within PUs requires a mapping strategy of tasks to the different execution units. Workload-partitioning-strategies in a heterogeneous setting can be classified into static and dynamic. In a static partitioning-strategy, the mapping of tasks to the specific PUs is fixed before program execution and is indifferent to the hardware-specific capabilities of the PUs. A dynamic partitioning-strategy implies dynamic changes during runtime and according to hardware capabilities [21]. A static strategy introduces little overhead but is prone to load imbalance. In a static approach, edge removals are not necessarily evenly distributed across partitions, leading to load imbalance. A dynamic approach incurs more overhead but can handle a potential load imbalance.

In the context of constraint-based causal structure learning, we can partition tasks at vertex-level, at edge-level or at CI-test-level. On vertex-level, all CI-tests for all edges for one vertex are mapped to one PU. For edge-level, all CI-tests for one edge can be assigned to one PU. On CI-test-level, CI-tests for one edge can be assigned to different PUs. Schmidt et al. [29] propose an approach of dynamic workload-partitioning for multi-core systems on edge-level. In their dynamic approach, all edges per level are added to a queue that is processed in parallel by several workers. Each worker is mapped to a separate core in this multi-core setting.

5 HETEROGENEOUS PROPOSITION

In this section, we experimentally evaluate the bottlenecks of the two existing implementations pcalg and cuPC to determine which scenarios our heterogeneous approach should target. Moreover, we explain both the conceptual idea and a reference implementation of our novel approach for causal structure learning.

5.1 Potential for Optimization

To assess the optimization potential and to identify bottlenecks, we carefully analyze the adjacency search of the original cuPC implementation. Since cuPC is in general orders of magnitude faster than pcalg [38], we focus our attention on cuPC as our baseline.

We study the behavior of cuPC with the experimental setup described in in-depth later on in Section 6.1. Firstly, we modify the parameters during the data generation process to identify their impact on the algorithm's performance. We use the function `randomDAG()` of pcalg [11] to generate of synthetic data. We quantify the impact of the significance level α (which influences the threshold used by cuPC), the probability p (which influences the density of the underlying ground-truth graph), the number of random variables n as well as the number of data samples m.

This experiment yields that high probabilities p, significance levels α, and numbers of nodes n significantly increase the runtime of cuPC. We observe similar behavior for pcalg. The number of samples m, however, has a negligible impact for multivariate normal distributed data and is therefore not considered further.

Our next step is to analyze the number of CI-tests per level and their cost to assess which levels contribute to the long runtimes the most. For alpha $\alpha = 0.05$ and probability $p = 0.2$, we measure the number of CI-tests per level as shown in Figure 2.

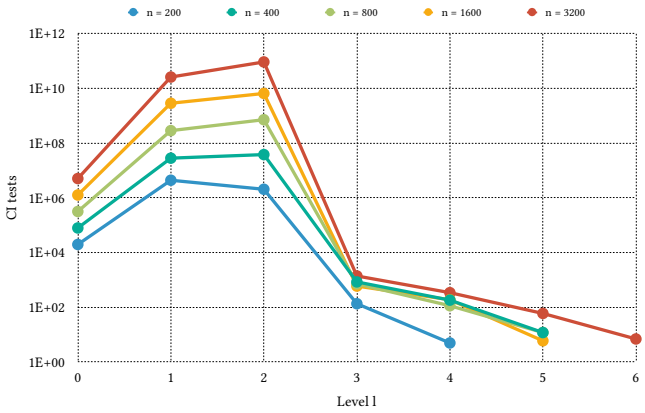


Figure 2: Number of CI-tests per level l. Parameters: m = 6400, p = 0.2, α = 0.05

The results in Figure 2 indicate that the majority of CI-tests are performed in level $l = 1$ and level $l = 2$. In another experiment, we study the computational runtime per level. This experiment shows that $l = 1$ together with level $l = 2$ take up $> 90\%$ of the computational runtime. Therefore, we analyze this behavior further and measure the GPU utilization during the execution of cuPC per level. Gathering the GPU metric `sm_utilization` with `nvprof` indicates allows us to quantify the utilization of the GPU's streaming multiprocessors during the execution. Since cuPC has a different kernel-function for each level, we obtain a detailed view of the utilization progression. Especially in the two levels $l = 1$ and $l = 2$, the algorithm fully utilizes the GPU as shown in Table 1.

Table 1: GPU utilization per level l for varying nodes n in percentage. Bold numbers indicate a GPU utilization > 90%. Parameters: m = 6400, p = 0.2, α = 0.05

l	n = 200	n = 400	n = 800	n = 1600	n = 3200
0	16.75%	36.34%	74.33%	90.87%	**97.46%**
1	**90.52%**	**90.45%**	**91.55%**	**94.45%**	**98.92%**
2	45.96%	51.69%	36.91%	35.83%	**100.00%**
3	22.16%	29.07%	34.89%	35.52%	49.49%
4	2.92%	12.80%	16.04%	14.79%	28.94%
5		3.54%	4.73%	5.73%	13.49%
6					9.35%

Using the two commands `top` and `mpstat`, we observe that the CPU is idling most of the time during the GPU-accelerated execution of CI-tests by cuPC. Given that the GPU is already fully utilized for $n >= 3200$ nodes harnessing the computational power of the otherwise idling CPU is not only a reasonable improvement but the core motivation for our heterogeneous approach.

5.2 Conceptual Idea

Our heterogeneous proposition for constraint-based causal structure learning is to execute a subset of the CI-tests for each level l on both CPU and GPU simultaneously. There are several ways of partitioning the CI-tests for a given level into two subsets as mentioned in Section 4. Firstly, we can perform all CI-tests for one edge on one PU. Secondly, we can perform all CI-tests for all edges of a vertice n on one PU. Thirdly, we can split CI-tests for one edge across different PUs. However, as Zarebavani et al. [38] outline, parallelizing the different CI-tests for an edge on the GPU yields a significant overhead and leads to redundant tests. Therefore, distributing work using such a fine-grained partitioning strategy across different PUs only exacerbates this performance issue.

Consequently, in our approach, we partition CI-tests based on their corresponding vertice n. Therefore, all CI-tests for all edges of a given vertice n are computed on one PU. Our heterogeneous algorithm targets especially scenarios with a large number of nodes (e.g., $n >= 1600$). Therefore, the granularity of partitioning all CI-tests on vertex-level is sufficient.

Our heterogeneous approach is based on static workload-partitioning without prior knowledge about the relative computational power of the different PUs or the transfer times between GPU and CPU. We define the partitioning factor pf as follows:

$$pf = \frac{\text{number of vertices processed by the CPU}}{\text{number of all vertices } n} \tag{3}$$

We perform the CI-tests for the edges that belong to the first $n * pf$ vertices in the adjacency matrix on CPU. Similarly, we compute the conditional independence tests for the last $n * (1 - pf)$ vertices of the graph's adjacency matrix on the GPU. Our heterogeneous approach operates in three distinct phases:

(1) **Partition the CI-tests on vertex-level**
(2) **Execute the CI-tests on both PUs simultaneously**
(3) **Merge the partially updated adjacency matrices**

Figure 3 describes our heterogeneous approach pictorially. In step (1), we partition the CI-tests on vertex-level based on the full adjacency matrix A. Thereafter, in phase (2), both CPU and GPU perform the CI-tests simultaneously for their assigned vertices in the submatrices B and C. On the PUs, all edges for which CI-tests must be performed are determined, and CI-tests are performed in parallel. Therefore, the heterogeneous approach partitions on vertex-level, but all CI-tests are performed in parallel on the respective PUs. In step (3), we join the updated submatrices B and C into the full adjacency matrix D after both PUs have finished.

In addition to the fundamental assumptions underlying causal structure learning outlined in Section 2.1, we assume a reasonably large problem (e.g., with $n >= 1600$ nodes) to ensure that the GPU would be fully utilized when performing the CI-tests exclusively. However, we consider only problem sizes that fit into the high-bandwidth GPU memory. If the problem size exceeds the GPU

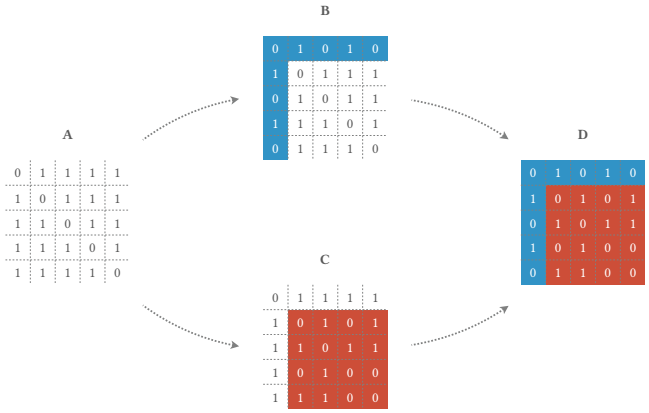


Figure 3: Three-phase algorithm execution of our heterogeneous approach based on static workload-partitioning

memory, an out-of-core implementation, as proposed by Schmidt et al. [30], becomes necessary as otherwise, the execution failed.

5.3 Implementation

We base our heterogeneous approach's reference implementation[2] on cuPC (Section 3.1) and pcalg (Section 3.2). Therefore, we extract both the adjacency search and the independence test C++ components of pcalg[3], remove their R package bindings, and embed the code fragments into the CUDA source code of cuPC[4]. Our primary focus is the feasibility of performing CI-tests as part of the adjacency search in a heterogeneous setting.

Initially, our heterogeneous approach for conducting the adjacency search on a graph G given its correlation matrix C has to initialize the CPU and GPU data structures, respectively.

On the CPU side, we initialize an internal graph structure based on the Boost Graph Library [33], a well-established C++ library providing both data structures and frequently used algorithms for graphs. Moreover, we allocate memory for the separation sets S and supplementary data structures used for information for the final edge direction that we do not consider further.

On the GPU side, we allocate linear memory for similar data structures using `cudaMalloc` and `cudaMallocManaged`, respectively [25]. `cudaMalloc` is a CUDA API call that allocates linear memory on the GPU that is not directly accessible to the CPU. Operating on GPU memory allocated with `cudaMalloc` from the CPU involves a call to `cudaMempy` for copying the GPU data into a temporary buffer residing in main memory. In our heterogeneous causal structure learning algorithm, we use it only for data structures that are exclusively used by the GPU (e.g., the separations sets S and the edge direction data). `cudaMallocManaged`, on the other hand, is a CUDA API call that allocates memory that will be transparently managed by the Unified Memory system. The Unified Memory system creates a shared memory address space that is accessible from both GPU and CPU. In addition to that, `cudaMallocManaged` minimizes the amount of data that needs to be copied back and forth between GPU and CPU by employing a technique called demand paging where only the modified memory areas are exchanged. Applied to our heterogeneous approach for causal structure learning, `cudaMallocManaged` is favorable for data structures that need synchronization between the CPU and the GPU, respectively. Therefore, we use it solely for the GPU's graph representation.

The next step of our heterogeneous adjacency search is to determine conditional independence between the graph's nodes across all levels ranging for $l = 0$ to $l = 14$. In alignment with the original cuPC implementation, our heterogeneous algorithm iterates over all levels and performs the three steps outlined in Section 5.2.

Firstly, we partition the CI-tests for the current level l based on the partitioning factor pf. The CPU's task is to perform the CI-tests for the first $n * pf$ nodes where n denotes the total number of nodes in the graph G. Since nodes are represented in the graph's adjacency matrix as rows and columns, the CPU operates on the first $n * pf$ rows and columns. The GPU's task is to perform the CI-tests on the last $n * (1 - pf)$ rows and columns, which forms a submatrix in the adjacency matrix A_G graphically shown in Figure 3.

Secondly, we perform the CI-tests simultaneously on both CPU and GPU. Therefore, we spawn a CPU thread that asynchronously calculates the CI-tests for the current level l on the CPU's share of the full adjacency matrix of G as determined in the first step. Moreover, we call cuPC's kernel functions to calculate the CI-tests for the current level l on the corresponding submatrix of G. Since both the asynchronously forked CPU thread and the GPU's kernel function for the level l return the control flow directly to the main thread, we have to synchronize the two. Therefore, we call `cudaDeviceSynchronize` from the CUDA API to wait until the kernel function terminates [25], and join the asynchronously spawned CPU thread with the main thread.

In level $l = 0$, the two PUs perform independence tests with an separation set $S = \emptyset$ (Section 2.2). Since the original implementation of pcalg operates on alpha α values for the independence criterion and cuPC uses a level-adjusted threshold τ, we modified the CPU components to use τ as well for comparability. Both CPU and GPU build up their initial graph structure in level $l = 0$ (i.e., fill their partial adjacency matrices of G with 1 in the (i, j)-th and (j, i)-th place if two nodes V_i and V_j are dependent). The CPU thread sequentially collects all edges in its share of G and then executes all edge tests for conditional independence in parallel using t threads managed by a framework for parallel programming called openMP [2]. t is a platform-dependent constant and typically resembles the number CPU cores. In a scenario where there are CI-tests for k edges to perform on the CPU side, each of the t threads computes CI-tests for k/t edges. This implies that although we partition the CI-tests on a vertex-level, we parallelize the execution on an edge-level to ensure high CPU utilization at all times. On the other side, the available GPU threads in the kernel function for level $l = 0$ map themselves onto entries in the GPU portion of G's adjacency matrix through their block-dimension, block-index, and thread-index.

In levels $l > 0$, both CPU and GPU conduct CI-tests given a separation set S (Section 2.2). However, instead of building up the initial graph structure by adding edges to it, for levels $l > 0$ the two

[2] github.com/christianwarmuth/heterogeneous_pc_algorithm
[3] github.com/cran/pcalg
[4] github.com/LIS-Laboratory/cuPC

PUs remove edges from the graph. Speaking in terms of the partial adjacency matrices of G, both CPU and GPU set the (i, j)-th and (j, i)-th place to 0 if the two nodes V_i and V_j are conditionally independent. Since the number of edges k for which CI-tests need to be performed in a given level l might be very small, we introduce an additional layer of parallelism to fully utilize the t threads. If there are $k < t$ edges, we parallelize the execution of CI-tests for single edges so that all t threads are fully utilized. We achieve this by integrating the parallelization capabilities of Armadillo [28], the scientific C++ library used for performing the CI-tests on the CPU side.

Thirdly, after both CPU and GPU have finished their calculations for a given level l and are synchronized in the main thread, we merge the partially updated adjacency matrices of G. Since the GPU's graph representation is allocated using `cudaMallocManaged` and is managed by the Unified Memory system, we can directly access both the GPU graph data structure and its CPU counterpart in the main thread. We copy the first $n * pf$ rows and columns from the CPU adjacency matrix into the GPU's graph, and transfer the submatrix spanning over $n * (1 - pf)$ rows and columns from the GPU into the CPU's adjacency matrix as depicted in Figure 3.

After performing all CI-tests across all 14 levels, the adjacency search is complete. We embed the reference implementation of our heterogeneous approach for the adjacency search into an R function, which executes the entire PC algorithm (Section 2.2). Therefore, we execute one additional task in the main thread. Similar to the level-wise merge phases, we copy both the partial adjacency matrices of G and the supplementary data structures (e.g., the separations sets S and additional information used for edge direction) from the CPU and the GPU, respectively, back to R. In the context of our approach, this is called the transfer phase.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our heterogeneous approach by running several experiments.

6.1 Experimental Setup

Our experiments are conducted on a bare-metal server with 2x Intel Xeon Gold 6148 CPUs @ 2.40 GHz with 20 cores, 2 threads per core, and 750 GB RAM @ 2666 MT/s attached to it. The server is equipped with 4x NVIDIA Tesla V100-SXM2 GPUs with 32 GB of high bandwidth memory. CPU and GPU are connected via PCIe Generation 3. Moreover, it runs on Ubuntu 18.04 LTS with GCC 7.5, OpenMP 4.5, and R 3.6.3. In addition to that, CUDA 10.1 with the driver version 430.50 as well as NVCC 9.1.85 are installed.

Regarding our profiling efforts, we use NVIDIA Visual Profiler (`nvprof`) 9.1.85, NVIDIA Nsight Systems (`nsys`) 2019.3.7.5, and the chronometry library `tictoc` [9] to measure the runtime of R function calls. On the C++ side, we use the system-wide monotonic clock `std::chrono::steady_clock` for high-precision time measurements. In our experiments, we use the R function `randomDAG` from the pcalg package [11] to generate synthetic data where the underlying DAG is known in advance.

We reduce measurement deviations by running each experiment 100 times and taking the arithmetic mean over all runtimes.

6.2 Performance Evaluation

We start our performance evaluation by analyzing the runtime for the memory allocation and data transfer phases for both CPU and GPU as shown in Figure 4. We see that the GPU memory allocation is significantly faster than its CPU counterpart. On the GPU side, we observe a steady increase between 0.001 and 0.003 seconds per 200 nodes. On the CPU side, we see a steady increase between 0.02 and 0.05 seconds per 200 nodes.

In regards to the data transfer phase, we measure the one-time task of copying the graph G, the separation sets S, and additional data for the edge direction back to R after processing all CI-tests on each level as discussed in Section 5.3. We show that the CPU data transfer is significantly faster than the GPU transfer. Between $n = 1600$ and $n = 1800$, it only shows a difference of 0.003 seconds, whereas between $n = 3000$ and $n = 3200$, we see an increase of 0.01 seconds. The GPU data transfer does also follow a linear growth pattern in its runtime for different numbers of nodes n, but the GPU transfer takes significantly longer than the transfer of the CPU.

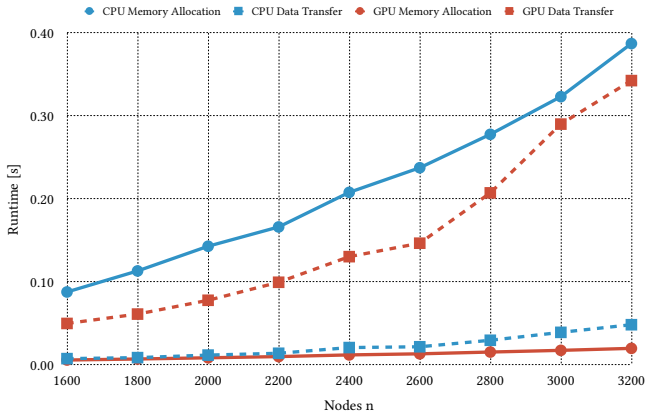


Figure 4: Memory management and data transfer runtimes per PU for varying nodes n. Parameters: $m = 6400$, $p = 0.2$, $\alpha = 0.05$, $pf = 0.05$

Even though the CPU works on significantly fewer rows and columns, respectively, it allocates memory for the full graph G as the partitioning factor pf could in theory be adjusted in-between levels. Therefore, the memory allocation runtimes for both CPU and GPU grow linearly with the number of nodes n. However, when it comes to data transfers, the CPU is clearly ahead in terms of performance. Given that considerably fewer entries in the adjacency matrix of G, separation sets S, and edge direction data points need to be copied back to R in the end, the runtimes differ wildly.

Next, we study the runtime of the merge phase for each level l as depicted in Figure 5. The merge phase constitutes the mutual data exchange between the partial adjacency matrices of the graph G between CPU and GPU. The merge phase takes about 0.54 seconds in level $l = 0$ and level $l = 2$. In level $l = 1$, however, it takes considerably longer with approximately 0.2 seconds. The runtime in all subsequent levels comes close to 0.02 seconds.

In each merge phase per level l, we have to synchronize all edge additions (for level $l = 0$) and deletions (for level $l > 0$) in

the adjacency matrix of G between CPU to GPU. While accesses into the internal graph representation of the CPU are quite cheap from the viewpoint of the main thread, reads and writes into the adjacency matrix of the GPU which is managed by the Unified Memory system are relatively expensive.

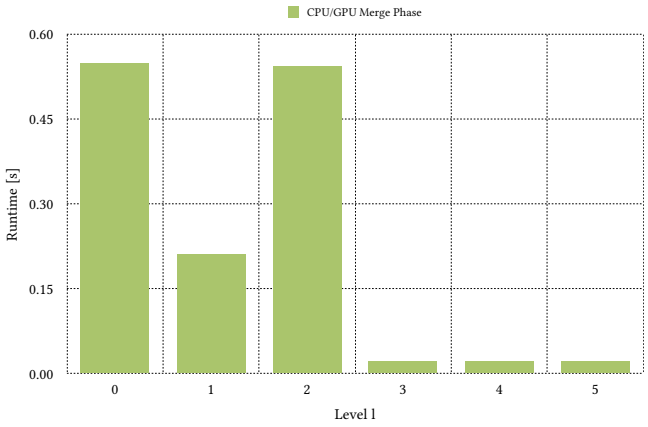


Figure 5: Runtimes for merge phase per level l. Parameters: $n = 2000$, $m = 6400$, $p = 0.2$, $\alpha = 0.05$, $pf = 0.05$

That is the reason why we only propagate differences between the partial adjacency matrices residing on the CPU and GPU, respectively. Keeping this in mind, it is not surprising that in level $l = 0$, the merge phase appears to be relatively expensive as numerous edges are added to the initial graph. Moreover, as we have seen in Section 5.1, level $l = 2$ has by far the most CI-tests which presumably result in many edge deletions. Therefore, the reason why for level $l = 1$, which entails many CI-tests as well, the merge phase is considerably cheaper than for level $l = 2$ is that fewer edges get removed from the graph G.

Using `cudaMallocManaged` for allocating linear memory on the GPU which is directly accessible to the CPU as well for synchronizing the partial adjacency matrices of G is not the only possible approach. It is also feasible to allocate linear GPU memory, which is not accessible to the CPU, using `cudaMalloc`, and calling `cudaMemcpy` to transfer this memory block into a temporary buffer in main memory to perform the merge phase. Although this entails actually two calls to `cudaMemcpy` as the updated adjacency matrix of G in the temporary buffer needs to be copied back to the GPU eventually, our benchmarks indicate, that for large n there is only a negligible difference between `cudaMallocManaged` and `cudaMalloc` with respect to performance. Since `cudaMallocManaged` does, however, provide a better programming experience, we use it for our reference implementation is discussed in Section 5.3.

Our next experiment addresses the question how the per-level CPU runtime behaves for different partitioning factors pf as shown in Figure 6. Increasing the partitioning factor pf yields higher per-level CPU runtimes for levels $l = 0$, $l = 1$, and $l = 2$. While it takes the CPU just 0.05 seconds to compute all CI-tests in level $l = 2$ for the partitioning factor $pf = 0.01$, it takes it 0.84 seconds for the partitioning factor $pf = 0.09$. In subsequent levels $l > 2$, the runtime reaches a plateau between 0.029 seconds and 0.044 seconds.

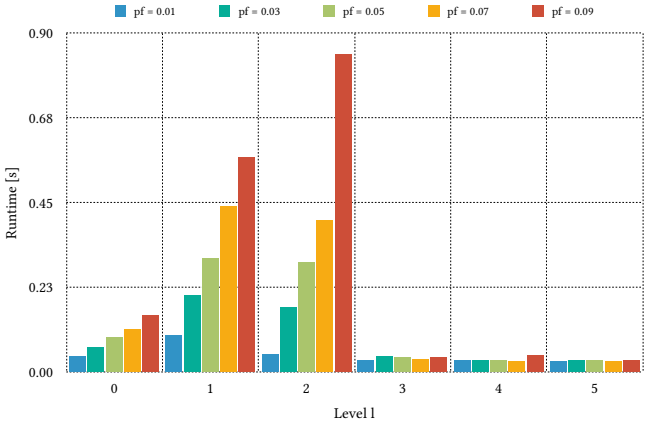


Figure 6: CPU runtimes per level l for varying partitioning factor pf. Parameters: $n = 2000$, $m = 6400$, $p = 0.2$, $\alpha = 0.05$

These observations coincide with our expectations based on the previous experiment (Figure 2) which analyzes the number of CI-tests per level l on the GPU only. Since the number of CI-tests grows heavily in levels $l = 0$ to $l = 2$ and stays constantly low for levels $l > 2$, the pure CPU runtime per level behaves accordingly and reflects this CI-test distribution.

Finally, we examine the CPU and GPU runtimes per level l for a fixed partitioning factor $pf = 0.09$ as depicted in Figure 7. We see that the CPU runtime is higher than the CPU runtime in all levels l for this experiment except for level $l = 2$, in which the GPU computations take significantly longer than the CPU computations (i.e., 3.54 seconds compared to 0.84 seconds).

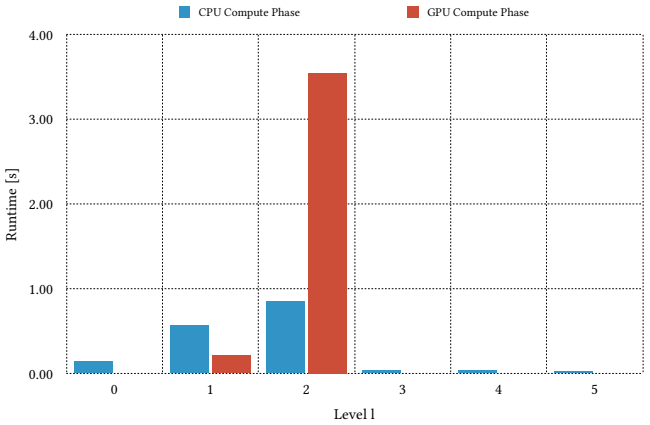


Figure 7: CPU and GPU runtimes for simultaneous CI-test computation per level l. Parameters: $n = 2000$, $m = 6400$, $p = 0.2$, $\alpha = 0.05$, $pf = 0.09$

In theory, the most favorable runtime distribution between CPU and GPU consists of identical runtimes for CPU and GPU across all levels. In this case, neither the CPU nor the GPU has to wait for the respective other PU in the main thread's synchronization step. In practice, this is not achievable with a single partitioning factor pf.

Level $l = 0$, for example, is well suited for GPU execution as performing a large number of independence tests without any intermediate synchronization steps in-between the tests is an embarrassingly parallel problem. Therefore, the partitioning factor $pf = 0.09$ is set too high for the CPU to keep up with the GPU. Similarly, the CPU runtime is higher than the GPU runtime in level $l = 1$ resulting in unnecessary wait times for the GPU.

However, in level $l = 2$, the GPU takes considerably more time to compute its share of the CI-tests than the CPU albeit the partitioning factor pf remains the same. Our heterogeneous approach performs static workload-partitioning with a constant partitioning factor pf across all levels. Therefore, for a given level l the difficulty of the CI-tests for a PU are not considered as well as skewed edge removals between the partitions, resulting in load imbalance. This not only slows down the individual PU's runtime but also the total execution time of our heterogeneous approach. Consequently, we conclude that the partitioning factor pf needs to be adjusted between levels.

Our final experiments sets our heterogeneous approach's runtime in context to cuPC, the state-of-the-art GPU implementation of the PC algorithm. Table 2 shows the absolute runtimes as well as the relative difference in performance of our heterogeneous algorithm and cuPC for varying numbers of nodes n.

Table 2: Total runtimes of our heterogeneous approach and cuPC for varying nodes n. Parameters: $m = 6400$, $p = 0.2$, $\alpha = 0.05$, $pf = 0.05$

n	Heterogeneous Approach [s]	cuPC [s]	Difference
1600	3.29	1.82	80.53%
1800	4.62	2.64	74.66%
2000	5.91	3.62	63.06%
2200	8.50	5.56	52.79%
2400	19.35	15.22	27.14%
2600	18.68	13.96	33.85%
2800	24.46	18.97	28.93%
3000	47.93	39.50	21.36%
3200	43.72	34.97	25.02%

Notably, we do not compare our proposed algorithm against pcalg, as cuPC is orders of magnitude faster than pcalg [38]. Considering a problem size of $n = 1600$, our heterogeneous approach takes roughly 80% longer for the adjacency search than cuPC, largely due to the additional merge phases. However, for a larger problem size of $n = 3200$, we see that the relative performance difference is only approx. 25%. The reason for this considerably lower difference is that the longer the execution time of the CI-tests across all levels is, the more the merge phase runtimes become negligible. We conclude that for large numbers of nodes $n >= 3200$, we achieve comparable performance between our heterogeneous approach and cuPC while using less GPU resources and utilizing the CPU resources.

6.3 Discussion

Our experimental evaluation of the proposed approach proves the feasibility of a heterogeneous algorithm for performing the adjacency search of causal structure learning. We also achieve comparable performance to the original implementation of cuPC.

Despite the sharp increase in memory allocation runtimes for the CPU, for large numbers of nodes n, the initialization of the data structures is not the dominating portion of the runtime. Nevertheless, the memory allocation times incur a non-negligible overhead which has to be accounted for in a heterogeneous.

Since we only merge the differences in the partial adjacency lists of G on the CPU and GPU side, respectively, our optimized merge phases do not incur a constant overhead as only the deleted edges from both PUs are merged after each level l. This yields excellent performance in levels with few CI-tests.

Increasing the partitioning factor pf yields higher CPU runtimes for each level which is reasonable as it closely resembles the pure CPU execution behavior of the pcalg implementation.

In addition to that, we see that our heterogeneous approach needs a dynamically adjusted partitioning factor pf for each level to fully utilize both CPU and GPU across all levels in the most efficient way possible. Determining the optimal partitioning factor pf is challenging, though. Not only the computational power of the participating PUs but also the characteristics of the problem (e.g., the size of the adjacency matrix G) and the cost of a CI-test for a given level l need to be taken into account. Moreover, skewed edge removals which can have significant performance implications should be considered. This, essentially, boils down to introducing a dynamic workload-partitioning strategy to further improve the performance of our heterogeneous approach.

Moreover, we currently use static block-sizes for the kernel functions in CUDA for the GPU computations across all levels in alignment with cuPC. Fang et al. [7], however, propose a dynamic block-size adjustment strategy which, in our heterogeneous setting, might further improve the performance of the GPU as idle times for underutilized kernel threads can be reduced.

7 CONCLUSION AND FUTURE WORK

State-of-the-art implementations for constraint-based causal structure learning algorithms like pcalg and cuPC harness the advancements in recent CPU and GPU hardware developments. However, they typically do not exploit the computational power of both PUs but let one of them idle. Yet, we experimentally demonstrated that with cuPC for a problem with number of nodes $n = 3200$ in the underlying DAG, a high-end GPU is fully utilized while the CPU is idle.

Therefore, we proposed a heterogeneous approach for constraint-based causal structure learning. Our approach makes use of a static workload-partitioning strategy on the adjacency matrix of G based on a constant partitioning factor pf. Moreover, it performs the corresponding CI-tests for each level l simultaneously on the CPU and GPU, respectively. After each level, the two PUs are synchronized to merge the partially updated adjacency matrices to resemble the full level calculation as if it was done by as single PU. Our reference implementation for the proposed approach is based on pcalg and cuPC.

However, the heterogeneous approach itself is fully independent of the specific CPU and GPU implementations.

In our experimental evaluation we showed that the additional memory allocation, the final data transfer, and the merge phases become less significant for larger numbers of nodes n as the computation time for the CI-tests increases sharply. In addition to that, we found that we cannot fully utilize the processing power of both PUs across all levels with a constant partitioning factor pf as we run into workload imbalances and different costs per CI-tests for different levels. Nevertheless, we demonstrated that the reference implementation of our heterogeneous approach based on pcalg and cuPC has comparable performance to cuPC while reducing the necessary GPU resources and utilizing otherwise idling CPU resources.

Future research efforts in this field should study the feasibility of a dynamic workload-partitioning-strategy which takes the cost of a CI-tests per level, the relative computational power of the PUs, their interconnect (e.g., PCIe or NVLink), and skewed edge removals into account. This can further improve the performance of our heterogeneous approach. Besides that, it might be worthwhile to analyze whether it is beneficial to calculate the CI-tests for certain levels exclusively on one PU during the adjacency search. This can be beneficial for both CI-tests with unique characteristics or in cases where there are extremely few CI-tests and the merge phase overhead is too high. Furthermore, dynamic block-size adjustments for the GPU kernel functions in CUDA might further reduce the overall runtime of our heterogeneous algorithm for constraint-based causal structure learning.

ACKNOWLEDGMENTS

We would like to thank Christopher Hagedorn and Johannes Huegle for the guidance and technical support as part of the seminar "Causal Inference" and beyond.

REFERENCES

[1] Jasmin Ajanovic. 2009. PCI Express 3.0 Overview. *Symposium on High Performance Chips* (2009), 1–143.
[2] OpenMP Architecture Review Board. 2018. *OpenMP Application Programming Interface.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, 08/29/2020.
[3] David Maxwell Chickering, David Heckerman, and Christopher Meek. 2004. Large-Sample Learning of Bayesian Networks Is NP-Hard. *Journal of Machine Learning Research* (2004), 1287–1330.
[4] C Chow and Cong Liu. 1968. Approximating Discrete Probability Distributions with Dependence Trees. *Transactions on Information Theory* (1968), 462–467.
[5] Diego Colombo and Marloes H Maathuis. 2014. Order-Independent Constraint-Based Causal Structure Learning. *Journal of Machine Learning Research* (2014), 3741–3782.
[6] Denver Dash and Marek J Druzdzel. 2013. A Hybrid Anytime Algorithm for the Constructiion of Causal Models from Sparse Data. *arXiv Preprint* (2013), 1–8.
[7] Juan Fang, Kuan Zhou, Chen Tan, and Hui Zhao. 2019. Dynamic Block Size Adjustment and Workload Balancing Strategy Based on CPU-GPU Heterogeneous Platform. *International Conference on Parallel and Distributed Processing* (2019), 999–1006.
[8] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. 2012. GPGPU Processing in CUDA Architecture. *arXiv Preprint* (2012), 1–16.
[9] Sergei Izrailev. 2016. *Package 'tictoc'.* https://cran.r-project.org/web/packages/tictoc/tictoc.pdf, 08/29/2020.
[10] Markus Kalisch and Peter Buehlmann. 2007. Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm. *Journal of Machine Learning Research* (2007), 613–636.
[11] M Kalisch, A Hauser, MH Maathuis, and Martin Maechler. 2020. *An Overview of the pcalg Package for R.* https://cran.r-project.org/web/packages/pcalg/vignettes/vignette2018.pdf, 08/29/2020.
[12] Markus Kalisch, Alain Hauser, Martin Maechler, and Diego Colombo. 2020. *Package 'pcalg'.* https://cran.r-project.org/web/packages/pcalg/pcalg.pdf, 08/29/2020.
[13] Markus Kalisch, Martin Maechler, Diego Colombo, Marloes H Maathuis, and Peter Buehlmann. 2012. Causal Inference Using Graphical Models with the R Package pcalg. *Journal of Statistical Software* (2012), 1–26.
[14] David Kirk. 2007. NVIDIA CUDA Software and GPU Parallel Computing Architecture. *International Symposium on Memory Management* (2007), 1–64.
[15] David B Kirk and W Hwu Wen-Mei. 2016. *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann.
[16] Thuc Le, Tao Hoang, Jiuyong Li, Lin Liu, Huawen Liu, and Shu Hu. 2016. A Fast PC Algorithm for High Dimensional Causal Discovery with Multi-Core PCs. *Transactions on Computational Biology and Bioinformatics* (2016), 1–13.
[17] Thuc Duy Le, Tao Hoang, Shu Hu, and Liang Zhang. 2015. *Package 'ParallelPC'.* https://cran.r-project.org/web/packages/ParallelPC/ParallelPC.pdf, 08/29/2020.
[18] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *Transactions on Parallel and Distributed Systems* (2019), 94–110.
[19] Anders L Madsen, Frank Jensen, Antonio Salmeron, Martin Karlsen, Helge Langseth, and Thomas D Nielsen. 2014. A New Method for Vertical Parallelisation of TAN Learning Based on Balanced Incomplete Block Designs. *European Workshop on Probabilistic Graphical Models* (2014), 302–217.
[20] Anders L Madsen, Frank Jensen, Antonio Salmeron, Helge Langseth, and Thomas D Nielsen. 2015. Parallelisation of the PC Algorithm. *Conference of the Spanish Association for Artificial Intelligence* (2015), 11–24.
[21] Sparsh Mittal and Jeffrey S Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *Computing Surveys* (2015), 1–35.
[22] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2008. Data Processing on FPGAs. *International Conference on Very Large Data Bases* (2008), 910–921.
[23] Radhakrishnan Nagarajan, Sujay Datta, Marco Scutari, Marjorie Beggs, Greg Nolen, and Charlotte Peterson. 2010. Functional Relationships between Genes Associated with Differentiation Potential of Aged Myogenic Progenitors. *Frontiers in Physiology* (2010), 1–160.
[24] Preetam Nandy, Alain Hauser, and Marloes H Maathuis. 2018. High-Dimensional Consistency in Score-Based and Hybrid Structure Learning. *The Annals of Statistics* (2018), 3151–3183.
[25] NVIDIA. 2019. *CUDA Runtime API.* https://docs.nvidia.com/pdf/CUDA_Runtime_API.pdf, 08/29/2020.
[26] Pekka Parviainen and Samuel Kaski. 2017. Learning Structures of Bayesian Networks for Variable Groups. *Journal of Approximate Reasoning* (2017), 110–127.
[27] Judea Pearl. 2009. *Causality.* Cambridge University Press.
[28] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: A Template-Based C++ Library for Linear Algebra. *Journal of Open Source Software* (2016).
[29] Christopher Schmidt, Johannes Huegle, Philipp Bode, and Matthias Uflacker. 2019. Load-Balanced Parallel Constraint-Based Causal Structure Learning on Multi-Core Systems for High-Dimensional Data. *International Conference on Knowledge Discovery and Data Mining* (2019), 59–77.
[30] Christopher Schmidt, Johannes Huegle, Siegfried Horschig, and Matthias Uflacker. 2019. Out-Of-Core GPU-Accelerated Causal Structure Learning. *International Conference on Algorithms and Architectures for Parallel Processing* (2019), 89–104.
[31] Christopher Schmidt, Johannes Huegle, and Matthias Uflacker. 2018. Order-Independent Constraint-Based Causal Structure Learning for Gaussian Distribution Models Using GPUs. *International Conference on Scientific and Statistical Database Management* (2018), 1–10.
[32] Marco Scutari. 2009. Learning Bayesian Networks with the bnlearn R Package. *arXiv Preprint* (2009), 1–22.
[33] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *The Boost Graph Library.* https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/table_of_contents.html, 08/29/2020.
[34] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. 2000. *Causation, Prediction, and Search.* MIT Press.
[35] Douglas R Stinson. 2004. *Combinatorial Designs: Constructions and Analysis.* Springer.
[36] The R Core Team. 2020. *R: A Language and Environment for Statistical Computing.* https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf, 08/29/2020.
[37] Thomas Verma and Judea Pearl. 1990. Equivalence and Synthesis of Causal Models. *Conference on Uncertainty in Artificial Intelligence* (1990), 255–270.
[38] Behrooz Zarebavani, Foad Jafarinejad, Matin Hashemi, and Saber Salehkaleybar. 2019. cuPC: CUDA-Based Parallel PC Algorithm for Causal Structure Learning on GPU. *Transactions on Parallel and Distributed Systems* (2019), 530–542.