Accelerating Constraint-Based Causal Discovery by Shifting Speed Bottleneck

Ce Guo
c.guo@imperial.ac.uk
Imperial College London
London, United Kingdom

Wayne Luk
w.luk@imperial.ac.uk
Imperial College London
London, United Kingdom

ABSTRACT

Causal discovery is a technique to find the causal relationship between variables using data. This technique has many applications in data mining and knowledge discovery. However, the high data dimensionality results in a significant computational efficiency problem. A common speed bottleneck in conventional causal discovery methods is the execution of conditional independence (CI) tests. This paper proposes, analyzes, and evaluates a novel acceleration strategy for causal discovery, which has low communication costs and can effectively exploit FPGA on-chip memory and parallelism. First, we propose an algorithmic method to shift the speed bottleneck from CI test execution to CI test generation. Second, we design a hardware accelerator for CI test generation on FPGAs. Third, we evaluate the proposed approach by comparing the accuracy-speed trade-off against four state-of-the-art accelerated causal discovery tools on CPUs and GPUs. Our accelerated implementation running on an Intel Arria 10 GX FPGA shows a superior accuracy-speed trade-off in 12 causal discovery problems. The implementation achieves up to 8.8 times speedup over the cuPC software running on an NVIDIA GeForce RTX 2080 Ti GPU. It also achieves up to 155.7 times speedup over the stable.fast software running on an Intel Xeon Silver 4110 octa-core CPU. To the best of our knowledge, the proposed approach is the first FPGA-based acceleration approach for constraint-based causal discovery.

ACM Reference Format:
Ce Guo and Wayne Luk. 2022. Accelerating Constraint-Based Causal Discovery by Shifting Speed Bottleneck. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22), February 27–March 1, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3490422.3502363

1 INTRODUCTION

Causal discovery [1, 2], also known as casual structural learning [3], is a technique to find the causal relationship between random variables. A causal discovery algorithm uses data sampled from random variables to produce a graph representation for the causal relationships between these variables.

Causal discovery has many applications in data mining and knowledge discovery. For example, causal discovery enables a data-driven approach to finding genetic causes of diseases, including cancers [4, 5]. The conventional way to study gene expression is to conduct controlled experiments. Specifically, to investigate whether a target gene has an expected effect, one needs to randomly divide the samples into two groups, knock down the target gene in one group, and compare the expected effect in both groups. However, the high experiment cost often makes large-scale controlled experiments impossible. Causal discovery provides an alternative way to explore gene expression since causal discovery algorithms can extract promising causal relationships directly from a data set of multiple genes and effects. It is only necessary to run small-scale controlled experiments to verify the discovered relationships. This data-driven approach can significantly improve productivity and cut down the cost of gene expression analysis.

The long computation time of causal discovery limits its applications. For instance, an experiment [6] shows that a widely used software tool for causal discovery, pcalg [7], takes around 73 hours to produce a causal graph for a gene expression data set [8] with 1643 variables on an Intel Xeon CPU with eight cores running at 2.5 GHz.

In this study, we propose an FPGA-accelerated approach for constraint-based causal discovery. An experimental implementation of the proposed approach can finish causal discovery for up to 5000 variables within one minute. The acceleration of constraint-based causal discovery is a considerable challenge for FPGAs. In particular, the execution of conditional independence (CI) tests, which is a common speed bottleneck in conventional causal discovery methods, appears intractable for FPGA-based acceleration. A critical insight of this study is to shift the speed bottleneck from the execution of CI tests to their generation process. The shifting of the bottleneck facilitates the design of accelerated hardware. In summary, the main contributions of this paper include the following:

(1) An acceleration strategy that shifts the speed bottleneck from CI test execution to a novel FPGA-friendly CI test generation process (section 3).
(2) An FPGA-accelerated causal discovery design and its implementation based on the proposed strategy (section 4).
(3) An empirical study comparing the proposed approach against state-of-the-art accelerated causal discovery tools on CPUs and GPUs regarding speed and accuracy (section 5).

This study is the first to accelerate constraint-based causal discovery using FPGAs to the best of our knowledge. An open-source implementation of the FPGA-accelerated design is available at https://github.com/ceguo/cdcsf.

FPGA '22, February 27–March 1, 2022, Virtual Event, USA.

ACM ISBN 978-1-4503-9149-8/22/02…$15.00
https://doi.org/10.1145/3490422.3502363

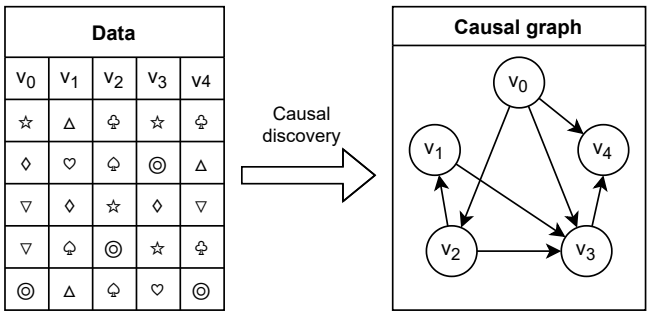


Figure 1: Problem setting of causal discovery

The organization of the remainder of this paper is as follows. Section 2 discusses the background and related work, including the causal discovery problem, conventional solutions, and acceleration methods; section 3 presents the proposed FPGA acceleration approach for constraint-based causal discovery from an algorithmic perspective; section 4 describes the design and implementation for the proposed approach using FPGAs; section 5 shows an empirical evaluation by comparing our FPGA-accelerated implementation against state-of-the-art causal discovery tools running on CPUs and GPUs.

2 BACKGROUND AND RELATED WORK

We briefly introduce the causal discovery problem and its solutions in this section. Section 2.1 explains the causal discovery problem and the Peter-Clark (PC) algorithm; section 2.2 gives an overview of three accelerated causal discovery tools on multi-core CPUs and GPUs.

2.1 Causal discovery and the Peter-Clark algorithm

A general problem setting for causal discovery is shown in Figure 1. A data set contains n ($n > 0$) samples taken from a set of random variables V. Each sample is a vector of $|V|$ elements, giving the values of the variables. The causal discovery algorithm discovers the causal relationships from data and encodes the relationships into a graph $G = (V, E)$. The node set V is the set of variables; the edge set E is the set of relations between variables. An edge from node v_A to v_B corresponds to the causal relationship that v_A is a cause of v_B.

Two dominating approaches for causal discovery are constraint-based and score-based methods [9]. We focus on constraint-based causal discovery methods in this study. A basic constraint-based causal discovery algorithm is the Peter-Clark (PC) algorithm. The algorithm works in two steps. The first step is causal skeleton discovery, which extracts an undirected graph from data. The second step is to determine the directions of the edges to build a completed partially directed acyclic graph (CPDAG). The first step is the most computationally demanding part of the algorithm [10].

There are a few variants of the PC algorithm which differ in the skeleton discovery step. The most widely-used variant is PC-stable [11]. Given a set of variables V, the skeleton discovery procedure of PC-stable is shown in Algorithm 1. The function $\perp(v_A, v_B, C)$ in

Algorithm 1: Skeleton discovery in PC-stable algorithm

```
0  E ← {(v_A, v_B) : v_A ≠ v_B and v_A ∈ V and v_B ∈ V}
1  l ← 0
2  f_stop ← false
3  while l < l_max and f_stop = false do
4      for (v_A, v_B) ∈ V × V do
5          f_stop ← true
6          R ← ∅
7          for C ⊆ (neighbor(v_A) − {v_B}) and |C| = l do
8              if ⊥(v_A, v_B, C) then
9                  R ← R ∪ (v_A, v_B)
10                 f_stop ← false
11     E ← E − R
12     l ← l + 1
13 return E
```

the algorithm (Line 8) is a conditional independence (CI) test. In this function, $v_A \in V$ and $v_B \in V$ are two random variables; $C = \{v_{c_0}, v_{c_1}, ..., v_{c_{L-1}}\}$ is a set of $L \geq 0$ random variables. The function determines whether v_A and v_B are conditionally independent given C. In other words, the function returns 'true' if and only if, given any value combination of C, (i) the probability distribution of v_A is the same for all values of v_B, and (ii) the probability distribution of v_B is the same for all values of v_A. The random variable set C is called the *conditioning set* of the CI test since it serves as the shared condition in these conditional probability distributions.

The PC-stable algorithm starts from a fully connected graph and traverses through the edges multiple times. When the algorithm visits an edge (v_A, v_B), it generates a collection of conditioning sets $\mathfrak{C} = \{C_0, C_1, ...C_{N_\mathfrak{C}-1}\}$ for CI tests. Then, the algorithm decides whether to remove (v_A, v_B) by running CI tests using the conditioning sets in $\mathfrak{C}$. If v_A and v_B are conditionally independent given any set $C \in \mathfrak{C}$, the algorithm puts the edge (v_A, v_B) which would be deleted into the set R. Once the algorithm finishes testing all $C \in \mathfrak{C}$, it removes all edges in R from the graph.

2.2 Hardware acceleration

There have been studies on the acceleration of constraint-based causal discovery methods targeting CPUs and GPUs. This subsection covers three accelerated tools: ParallelPC, cuPC and gpuPC.

ParallelPC [10, 12] is the first implementation for parallel causal discovery to the best of our knowledge. The core idea is to execute CI tests in parallel. In the experiments, ParallelPC is not only faster but more accurate than pcalg. However, a major problem in this implementation is that the code is written in R, which does not support efficient execution.

cuPC [6] is the first approach that accelerates causal discovery using GPUs. The parallel algorithm is a lossless transformation of PC-stable. The cuPC accelerator works with two implementation-level optimizations. First, if the CI test for a pair of variables returns 'true', cuPC cancels all remaining tests immediately. Second, the CI tests sharing the same conditioning set take place in the same local thread, reducing the thread management cost. The paper claims that in a challenging data set, cuPC reduces the execution time

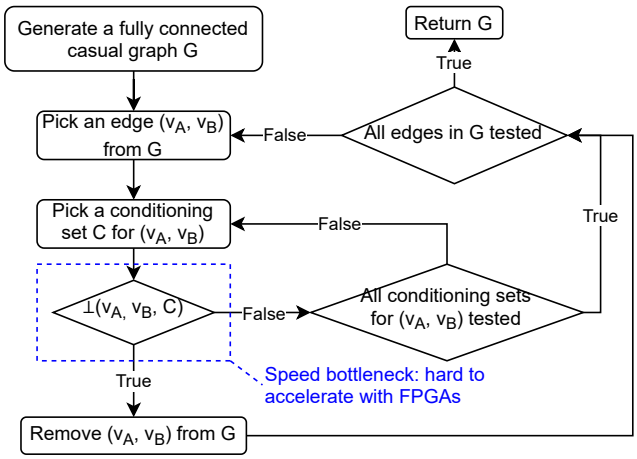


Figure 2: Conventional causal discovery workflow

from 11 hours with ParallelPC to about 4 seconds with an NVIDIA GTX 1080 GPU, achieving around 10000 times speedup.

gpuPC [13] is a GPU-based accelerator for discrete data. For CI tests, discrete distributions need auxiliary data structures, e.g., contingency tables. Such data structures occupy so much memory that the limited capacity of the GPU memory becomes a major challenge. Unlike cuPC, the optimization for gpuPC is designed for discrete data to facilitate parallel computation of marginal probabilities using contingency tables.

3 ACCELERATION STRATEGY

This section proposes an FPGA acceleration strategy for causal discovery. Section 3.1 discusses the motivation with an analysis of challenges and an algorithmic framework to address the challenges; section 3.2 proposes the conditioning set filtering (CSF) procedure that consolidates the algorithmic framework to facilitate FPGA acceleration; section 3.3 analyzes FPGA-related properties of CSF.

3.1 Motivation

Existing acceleration approaches for constraint-based causal discovery follows the workflow shown in Figure 2. This workflow is a generalized version of the PC-stable algorithm in Algorithm 1. The speed bottleneck in this workflow is still CI test execution. However, unlike PC-stable, the workflow does not limit the temporal order of edge traversing or conditioning set generation. As a result, an accelerated design can generate and execute multiple CI tests in parallel to address the speed bottleneck.

Although the conventional acceleration strategy is well studied with CPUs and GPUs, developing effective acceleration strategies for FPGAs is challenging. In particular, we face the following three major challenges:

(1) Challenge of discovery accuracy. We aim to build an FPGA-accelerated causal discovery design without compromising accuracy. However, it may not be wise to follow the acceleration strategy for CPUs and GPUs for accuracy. Specifically, conventional acceleration strategies usually employ low-complexity CI test algorithms to control design complexity, improve execution speed and facilitate optimizations. However, a low-complexity CI test algorithm can produce suboptimal causal graphs since it can incur a high error in edge removal decisions. For instance, the cuPC toolbox [6] only supports a Gaussian CI test algorithm based on Pearson's correlation coefficient. Although easy to implement, the Gaussian CI test produces less reliable decisions than state-of-the-art CI tests such as permutation-based and model-powered tests [14–17].

(2) Challenge of FPGA-unfriendly operations. The numeric operations in CI tests usually lack efficient FPGA implementations. The problem is significant even for simple CI test algorithms. For instance, a common statistic in CI test algorithms is partial correlation [18]. Accelerated causal discovery tools, such as cuPC [6] and bnlearn [19], calculate this statistic using an efficient routine based on matrix inversion. Although the routine has low time complexity, fast and numerically stable matrix inversion is challenging for FPGAs [20, 21].

(3) Challenge of limited generality in parallelization strategies. Since existing strategies depend highly on the nature of the specific CI tests, it is difficult to apply the strategy that works for one CI test algorithm to a different algorithm. For instance, a critical optimization in cuPC is to share a pseudo-inverse matrix among all CI tests using the same conditioning set. However, other CI tests, such as the conditional mutual information test [22], do not involve this pseudo-inverse operation. Therefore, the optimization in cuPC is no longer applicable.

The three challenges persist as long as we deal with the speed bottleneck by accelerating CI test execution. This study aims to address the speed bottleneck without accelerating CI test execution. Our key insight is to shift the speed bottleneck from CI test execution to another calculation to facilitate FPGA acceleration. Specifically, instead of making the CI test *execution* faster using FPGAs, we create an FPGA-friendly CI test *generation* procedure that reduces the number of CI tests so that the execution of CI tests no longer dominates the execution time. The newly introduced calculation is still computationally expensive, but it can be accelerated on FPGAs. On the other hand, the execution of the reduced set of CI tests can finish within a reasonable time on CPUs.

In the conventional causal discovery workflow in Figure 2, the generation and execution procedures for CI tests are tightly coupled. For instance, the PC-stable algorithm starts to execute a CI test immediately when the conditioning set and the edge become available. As a result, the generation and execution procedures should take place on the same hardware platform to minimize latency [6, 23]. However, we hope to run the two procedures on different hardware platforms. Therefore, we propose an algorithmic framework in Algorithm 2 to decouple the two procedures. In this algorithmic framework, `GenerateCITests()` is a CI test generation procedure that facilitates FPGA acceleration. On the other hand, the execution of the generated CI tests, Q, takes place on CPUs. The newly inserted CI test generation procedure in Algorithm 2 should cut down

Algorithm 2: Generic algorithm shifting speed bottleneck

$E \leftarrow \{(v_A, v_B) : v_A \neq v_B \text{ and } v_A \in V \text{ and } v_B \in V\}$
$l \leftarrow 0$
while *not converge* **do**
  for $(v_A, v_B) \in V \times V$ **do**
    $R \leftarrow \emptyset$
    $Q \leftarrow$ `GenerateCITests()` ▷ FPGA-accelerated
    for $(v_A, v_B, c) \in Q$ **do**
      $I \leftarrow \perp(v_A, v_B, c)$ ▷ CPU
      if I *is true* **then**
        $R \leftarrow R \cup (v_A, v_B)$ ▷ CPU
    $E \leftarrow E - R$
return E

Algorithm 3: Conditioning set filtering (CSF)

function `GenerateCITests()`
  $\{C_0, C_1, \ldots, C_{S-1}\} \leftarrow$ `GenerateShortlist()` ▷ CPU
  for $i \leftarrow 0$ **to** $S - 1$ **do**
    $k_i \leftarrow$ `ComputeScore`(C_i) ▷ FPGA
  $\theta \leftarrow$ `Quantile`$(\{k_0, k_1, \ldots, k_{S-1}\}, \alpha)$ ▷ CPU
  $\mathfrak{C} \leftarrow \{C_i : k_i > \theta\}$ ▷ CPU
  return `SetOfCITests`$(\mathfrak{C})$

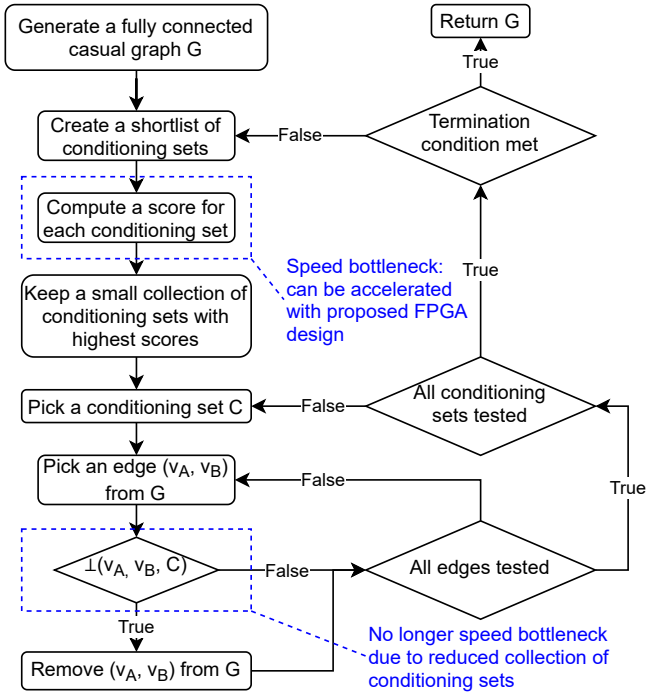


Figure 3: Proposed workflow for FPGA acceleration

the number of CI tests such that CI test execution is no longer computationally demanding. However, the CI test generation procedure itself may dominate the execution time.

3.2 Conditioning set filtering

Algorithm 2 only serves as a framework rather than a concrete algorithm. Therefore, we need to design an FPGA-friendly CI test generation procedure. This section proposes conditioning set filtering (CSF), an FPGA-friendly CI test generation procedure.

Algorithm 3 summarizes the CSF procedure, which involves four calculations to generate a set of CI tests. First, the CPU generates a shortlist of conditioning sets $\{C_0, C_1, \ldots, C_{S-1}\}$. Second, the FPGA computes a score k_i for each conditioning set in the shortlist and returns a list of scores $\{k_0, k_1, \ldots, k_{S-1}\}$. Third, the CPU obtains the scores from the FPGA and computes a threshold θ by taking the α-quantile of the scores. Fourth, the CPU takes a list of conditioning sets with higher scores than θ. At this stage, the CPU obtains a set of CI tests based on the high-score conditioning sets. The numeric parameter α should be a positive real number smaller than one. It should take a value close to one, such as 0.95 or 0.99, to filter out most conditioning sets.

The most critical part of CSF is the function `ComputeScore`(C_i) since it decides which conditioning sets are filtered out. Also, the evaluation of the score function can dominate the execution time of CSF. In this study, we propose the following score function:

$$\rho(C) = \sum_{v \in V} \rho_C(v) \tag{1}$$

where

$$\rho_C(v) = \text{kurt}(X_C \times \mathbf{w} - \mathbf{x}_v) \cdot \deg_G(v) \tag{2}$$

$$\text{kurt}(x) = \frac{E[(x - \bar{x})^4]}{(E[(x - \bar{x})^2])^2} \tag{3}$$

$$\mathbf{w} = \arg\min_{\mathbf{w}} \frac{1}{2} ||X_C \times \mathbf{w} - \mathbf{x}_v||_2 \tag{4}$$

where $\deg_G(v)$ the degree of node v in graph G; $\mathbf{x}_v$ is the data vector of variable v; X_C is the matrix containing the data vectors of all variables in the conditioning set C. In other words, the score $\rho(C)$ for a conditioning set C is the sum of sub-scores $\rho_C(v)$ for all $v \in V$. The sub-score $\rho_C(v)$ is the product of (i) the degree of the node and (ii) the kurtosis of the residual vector obtained from a least-squares model that predicts $\mathbf{x}_v$ using X_C. We use kurtosis because it indicates the number of independent components in a mixture of signals [24, 25].

By substituting Algorithm 3 into Algorithm 2, we obtain the causal discovery workflow in Figure 3. CI test execution is no longer the speed bottleneck in this workflow since CSF can control the number of CI tests by filtering out low-score conditioning sets. However, the evaluation of the score function in Equation 1 during CI test generation becomes the new bottleneck. Therefore, we propose to accelerate the evaluation of the score function using FPGAs. The proposed approach addresses the three challenges in section 3.1 altogether. The fundamental problem behind all three challenges is that FPGAs cannot execute accurate CI tests efficiently. The CSF approach addresses this fundamental problem by shifting the bottleneck from CI test execution to CI test generation. Therefore, it is no longer necessary to accelerate CI test execution using FPGAs.

3.3 Analysis of FPGA acceleration

We derive a data traffic model for the communication between the CPU and the FPGA for the evaluation of Equation 1. In each score evaluation process, the data transmitted from the CPU to the FPGA are S conditioning sets. Each conditioning set contains $|C_i|$ variable indices. The corresponding data size is:

$$D_{\text{CPU}\to\text{FPGA}} = S \cdot |C|_{\max} \cdot \omega_{\text{index}} \tag{5}$$

where $|C|_{\max}$ is the maximum value of $|C_i|$; ω_{index} is the number of bits for each variable index. On the other hand, the data transmitted from the FPGA to the CPU are S scores. The corresponding data size is:

$$D_{\text{FPGA}\to\text{CPU}} = S \cdot \omega_{\text{score}} \tag{6}$$

where ω_{score} is the number of bits for a score defined in Equation 1. Therefore, the total data traffic between CPU and FPGA is

$$\begin{aligned} D_{\text{total}} &= D_{\text{CPU}\to\text{FPGA}} + D_{\text{FPGA}\to\text{CPU}} \\ &= S \cdot (|C|_{\max} \cdot \omega_{\text{index}} + \omega_{\text{score}}) \end{aligned} \tag{7}$$

Besides data traffic, we can estimate the execution time by modeling the amount of computation on the FPGA. The execution time of the least-squares solver is algorithm-dependent. However, an efficient solver should be able to calculate the exact solution of the least-squares problem with around $n^2 \cdot |C|_{\max}$ FLOPs where n is the number of samples. On the other hand, the calculation of the kurtosis should take no more than $4 \cdot n$ FLOPs. As a result, the total number of FLOPs for all the S scores is

$$T_{\text{total}} = S \cdot |V| \cdot n \cdot (n \cdot |C|_{\max} + 4) \tag{8}$$

Considering the above models, we can observe that CSF has the following three FPGA-friendly properties:

(1) The CPU-FPGA communication time is small compared to the evaluation time of the score function. The ratio between the total number of FLOPs and the data traffic $\frac{T_{\text{total}}}{D_{\text{total}}}$ tends to be large in real-world causal discovery problems. For example, the most computationally demanding problem in [26] has the following settings: $|C|_{\max} = 3$, $n = 2000$, and $|V| = 441$. Taking the most compact representation for node indices, we have $\omega_{\text{index}} = \lceil \log_2 441 \rceil = 9$. Also, assuming that we use single-precision floating point numbers for the scores, we have $\omega_{\text{score}} = 32$. The corresponding ratio $\frac{T_{\text{total}}}{D_{\text{total}}} \approx 3\times10^7$. The transmission of one bit of data can lead to 30 million FLOPs. As a result, the communication between the CPU and the FPGA is unlikely to bottleneck the overall speed.

(2) CSF allows efficient exploitation of on-chip memory resources on FPGAs. Specifically, the space complexity of the least-squares solver can be as low as $O(|C|_{\max})$. As a result, the least-squares solver can spare a lot of on-chip memory resources to store data. Practically, we should use the on-chip memory to store the entire data set to maximize computational efficiency. In other words, the data scalability depends on the capacity of the on-chip memory. Fortunately, modern FPGAs often have sufficient on-chip memory to hold real-world data sets. For instance, the largest data set in [10] is only 10332 Kb with the 16-bit unsigned integer representation. In contrast, an Intel Arria 10 GX 1150 FPGA has 67244 Kb on-chip memory; a Xilinx Virtex UltraScale XCVU 440 FPGA has 88600 Kb on-chip memory.

(3) The evaluation of the score involves a large number of low-dimensional least-squares problems. Efficient solutions to these problems require the exploitation of fine-grained parallelism. Specifically, the dimensionality of a least-squares problem in CSF is equal to the size of the corresponding conditioning set, which is usually small. For instance, in the experiments with real-world data in [12], the size of the largest conditioning set is only $L = 5$. FPGAs can effectively exploit the fine-grained parallelism in these low-dimensional least-squares problems with arithmetic operations customized for low dimensionality [27].

Although we design CSF to facilitate FPGA acceleration, it is also possible to implement CSF on other processors, including multi-core CPUs and GPUs. However, CSF may not benefit these processors for two reasons:

(1) Since the convergence rate of least-squares problems is data-dependent, the evaluation of the sub-score function for different nodes may require different execution times. Therefore, an efficient design requires a load balancing mechanism for function evaluation. On the FPGA platform, once a function evaluation block finishes the evaluation of the sub-score for a node, it can start processing another node with little thread management cost. However, since the time to evaluate a single sub-score is short, a similar load balancing mechanism on other processors, especially GPUs, can lead to a significant thread management cost [28].

(2) CPUs and GPUs may not be efficient for the low-dimensional least-squares problems in CSF. Unlike FPGAs, hardware facilities on CPUs and GPUs that support least-square solvers, such as dot product instructions, cannot be customized to fit low-dimensional vectors. On the contrary, these facilities on mainstream CPUs and GPUs are usually optimized for high-dimensional vectors [29, 30] to improve computational efficiency for applications such as deep learning and numerical simulation. However, they are not beneficial to CSF due to the low dimensionality of the least-squares problems. There have been studies on parallel least-squares solvers optimized for low-dimensional problems, e.g., CAQR for multi-core CPUs [31] and the DFO for GPUs [32]. However, the best-supported dimensionality of these solvers is still too high for CSL. For example, the highest dimensionality for CSF across all real-word data sets in [12] is only $L = 5$. However, in an evaluation of CAQR [31], the speedup over the serial implementation decreases as L goes down from 10^5 to 1. When $L = 5$, CAQR is slower than the serial implementation regardless of the number of cores. Also, in the evaluation of DFO for GPUs in [32], the only evaluated dimensionality is $L = 500$, which is too large for CSF. Moreover, DFO is unlikely to perform well when $L = 5$ because the parallel QR decomposition algorithms supported by DFO are similar to the one used in CAQR.

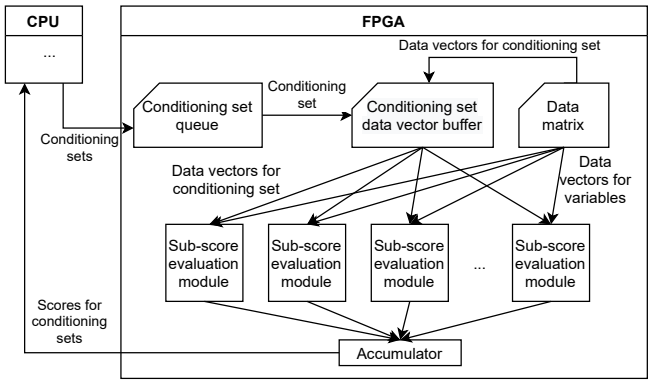


Figure 4: Block diagram for proposed FPGA design

Algorithm 4: Sub-score evaluation

0 $\mathbf{w} \leftarrow \mathbf{0}_l$; $\kappa^{\text{old}} \leftarrow +\infty$; $\kappa^{\text{new}} \leftarrow 0$
1 **while** $|\kappa^{\text{old}} - \kappa^{\text{new}}| > \epsilon$ **do**
2 $\quad \kappa^{\text{old}} \leftarrow \kappa^{\text{new}}$; $\psi_r \leftarrow \psi_o \leftarrow \psi_{o^2} \leftarrow \psi_{o^4} \leftarrow 0$
3 $\quad$ **for** $i \in [0..(n-1)]$ **do**
4 $\quad\quad r \leftarrow y_i - \mathbf{x}_i \cdot \mathbf{w}$
5 $\quad\quad \mathbf{g} \leftarrow r \cdot \mathbf{x}_i$
6 $\quad\quad \alpha \leftarrow \frac{r^2}{||\mathbf{g}||_2^2 + \epsilon}$
7 $\quad\quad \mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \mathbf{g}$
8 $\quad\quad \psi_r \leftarrow \psi_r + r$
9 $\quad\quad o \leftarrow r - \bar{r}$
10 $\quad\quad \psi_{o^2} \leftarrow \psi_{o^2} + o^2$; $\psi_{o^4} \leftarrow \psi_{o^4} + o^4$
11 $\quad \bar{r} \leftarrow \frac{\psi_r}{n}$
12 $\quad \kappa^{\text{new}} \leftarrow \frac{n \cdot \psi_{o^4}}{\psi_{o^2}^2 + \epsilon}$
13 **return** $\kappa^{\text{new}} \cdot \deg(y)$

4 DESIGN AND IMPLEMENTATION

This section presents a hardware design on an FPGA platform to accelerate the conditioning set filtering (CSF) procedure discussed in section 3. Section 4.1 presents the hardware design; section 4.2 describes our experimental implementation for the design.

4.1 Hardware design

Figure 4 shows a block diagram for the proposed FPGA design. The FPGA design includes a queue to buffer the conditioning sets received from the CPU. At the beginning of a score evaluation process, the queue releases a conditioning set. The conditioning set data buffer fetches the data vectors for the conditioning set. Next, the sub-score evaluation modules evaluate the sub-scores in parallel using Equation 2. The accumulator sums up the sub-scores according to Equation 1. When the accumulator finishes accumulating all the sub-scores for the conditioning set, it sends the final score back to the CPU.

The most critical component in the FPGA design is the sub-score evaluation module since this module deals with the speed bottleneck in CSF. Since this module involves non-trivial numeric operations and control logic, we present its functionality as a piece of pseudo-code in Algorithm 4. The constant ϵ in the pseudo-code is a small positive number to avoid divide-by-zero errors. The while-loop in the algorithm is a first-order optimization routine that solves the least-squares problem in Equation 4 with a stochastic Polyak step size [33]. The variables ψ_{o^2} and ψ_{o^4} collect statistical information from the residual r to update the kurtosis κ^{new} after the for-loop.

A design with more sub-score evaluation modules can compute more sub-scores in parallel. Therefore, an effective way to improve the speed is to deploy more sub-score evaluation modules within the constraints of resources and timing. For instance, we manage to deploy 56 sub-score evaluation modules in our experimental implementation on an Intel Arria 10 GX FPGA. Moreover, each sub-score evaluation module includes the following two non-obvious optimizations.

First, instead of using the up-to-date sample mean to calculate the kurtosis, we use an out-of-date version $\bar{r}$ based on the residuals in the previous for-loop iteration, as shown in Line 11. Without this optimization, it is necessary to store the residuals for all the n data points $[r_0, r_1, \ldots, r_{n-1}]$ in the on-chip memory to estimate their central moment. Also, we need to design a separate hardware block to evaluate this central moment in each iteration of the while-loop. The proposed optimization avoids such resource usage for residual storage and central moment evaluation. The price for this optimization is that the while-loop can occasionally take one more iteration than the unoptimized design. Specifically, the while-loop should terminate when the kurtosis based on the up-to-date sample mean is approximately equal to the kurtosis in the previous iteration. However, the out-of-date sample mean can enlarge the difference between the two kurtosis values, delaying the termination of the while-loop. Fortunately, if the while-loop fails to terminate in an iteration due to the out-of-date sample mean, the up-to-date sample mean must be available at the beginning of the next iteration. Therefore, the maximum delay is only one iteration.

Second, the conventional exit condition of the while-loop is based on the convergence status of the weight vector $\mathbf{w}$ [34]. However, we set the exit condition based on the kurtosis, as shown in Line 1. This setting can potentially cut down the number of iterations for the least-squares solver. In particular, when the least-squares solver stops due to the convergence of the weight vector $\mathbf{w}$, the kurtosis must be ready. However, the weight vector may still need updates when the kurtosis is ready. Since we are only interested in the kurtosis, we set the while-loop exit condition directly based on the kurtosis to finish the loop as early as possible.

4.2 Experimental implementation

We use the C programming language to describe both the hardware and the software. We describe the hardware in section 4.1 following the OpenCL 1.2 standard which is supported by the Intel FPGA SDK for OpenCL toolchain. We write the software following the C99 standard to ensure compiler compatibility.

Besides the hardware design on FPGAs, the proposed approach has three major routines running on CPUs: `Shortlist`(S) produces a shortlist of S conditioning sets for the FPGA to evaluate; `Quantile`(K, α) returns the α-quantile of the score array K obtained from the FPGA; $\perp(v_A, v_B, C_i)$ tests whether two variables

Table 1: Benchmarks used in experiments

Name	Type	# Nodes	# Edges
Hailfinder	Real-world	56	66
Hepar2	Real-world	70	123
Win95pts	Real-world	76	112
Pathfinder	Real-world	109	195
Andes	Real-world	223	338
Diabetes	Real-world	413	602
Pigs	Real-world	441	592
Link	Real-world	724	1125
Munin	Real-world	1041	1397
Random5kf2	Synthetic	5000	5048
Random5kf3	Synthetic	5000	7492
Random5kf4	Synthetic	5000	10017

v_A and v_B are conditionally independent given the conditioning set C_i.

In each iteration, we generate a connected dominating set from the graph as the conditioning set shortlist and use a sorting-based approach to calculate the quantiles. Specifically, we sort the list of scores $\{k_0, k_1, \ldots, k_{S-1}\}$ in ascending order using Quicksort and return the $\lfloor S \cdot \alpha \rfloor$-th element in the sorted list. The time complexity is identical to the Quicksort algorithm, which is $O(S \log S)$ on average and $O(S^2)$ in the worst case. All computation takes place on a single CPU thread.

We are aware that there are fast parallel algorithms to calculate quantiles on CPUs [35–37] and GPUs [38]. However, these algorithms do not demonstrate a speed advantage because the array size S for real-world data is small. For example, the experiments in [38] show that the Quicksort-based quantile algorithm on one CPU core is three times as fast as an optimized GPU-accelerated algorithm when $S < 16000$. In contrast, the largest possible S across all the data sets in [6] is only 5361.

We use the residual-based conditional independence test (RCIT) [39] in our implementation. We cache the residuals during the execution of CI tests using the thread-local memory. With this optimization, if the residuals for a variable in the current CI test have been computed in a previous CI test, the current test can directly retrieve the residuals from the thread-local memory without recomputation.

In addition to the above settings, we design an implementation-level optimization to run additional edge removal on the host CPU when the FPGA is busy. Specifically, once the FPGA starts to calculate scores for the shortlist, the CPU starts to execute CI tests to remove edges using the conditioning sets whose scores are below θ in the previous shortlist. Once the FPGA finishes the calculation, the CPU immediately stops working on the previous shortlist and starts to compute the quantile of the current scores.

5 EVALUATION

This section presents an empirical evaluation of the proposed approach using the experimental implementation described in section 4.2. Section 5.1 gives the experiment settings, including software tools, hardware environments and data sets; section 5.2 presents experiments on the trade-off between accuracy and speed; section 5.3 describes experiments comparing the speed of causal discovery tools with the same accuracy.

5.1 Experiment settings

Causal graphs used in experiments are shown in Table 1. These graphs include all benchmarks classified as large or massive graphs in [19]. We sample a data set from each causal graph following the settings in [40]. We do not use the gene expression data in [6] because the underlying causal structure is unknown. Since there are rarely large causal graphs with ground truths, we include three synthesized networks in this empirical study to discover the capability of different causal discovery methods on large causal graphs.

Table 2 shows the causal discovery tools we compare in the experiments. We compare these tools since they involve concerns and optimization for speed. Note that we do not test two types of causal discovery tools for a fair comparison. First, we do not test discrete-variable-only systems the discretization of continuous variables can lead to a big information loss. Second, we do not test score-based methods. The execution time of score-based methods is hardly comparable with the constraint-based methods in general. For example, the FPGA implementation of a score-based method [41] takes 224 seconds to discover the 'alarm' network on a Xilinx Alveo U200 acceleration card. However, the constraint-based toolkit stable.fast takes only 0.9 seconds.

Table 3 presents hardware platforms used in this study. The settings related to code compilation are as follows. We use the Intel FPGA SDK for OpenCL 20.1 to compile the hardware kernel for CSF. The resource usage of the implementation is shown in Table 4. We use GCC 8.2.1 to compile C and C++ code for bnlearn, stable.fast, and the host code of CSF. The optimization flags for bnlearn and stable.fast follow their default settings; the optimization flag for CSF is '-O3'. We use the NVIDIA CUDA compiler 11.1 to compile the GPU code for cuPC with the '-O3' optimization flag following the compilation guide in [6]. In all experiments, we set $\alpha = 0.95$ to filter out 95% of conditioning sets in each shortlist.

In addition to the causal discovery tools in Table 2, we develop CPU and GPU implementations for our CSF-based approach. Specifically, we build the CPU software in the C programming language using a single CPU thread. The CPU software is over 100 times slower than the FPGA version. Since the software is mainly for model-checking, its optimization was not as thorough as other tools in Table 2. Therefore, we do not include detailed results for a fair comparison. A multi-threaded implementation with appropriate load-balancing is beyond the scope of this paper. Assuming that the throughput for score evaluation can scale linearly with the number of physical cores, we estimate that the FPGA implementation can achieve 14-16 times speedup over the multi-threaded implementation running on the octa-core CPU in Table 3. In our GPU implementation, we use each thread to compute the scores for a subset of nodes for the same conditioning set. This implementation is 6—9 times slower than the FPGA implementation in the experiments. Our opinion on the relatively suboptimal GPU speed is that the proposed approach does not benefit GPUs, as we discuss in section 3.3. However, we cannot rule out that some new and non-obvious optimizations can improve GPU performance. As a

Table 2: Compared causal discovery tools

Tool	Algorithm	Hardware	Programming Language	Reference
bnlearn	Parallel PC-Stable	Multi-core CPU	C++ with R interface	[19]
cuPC	GPU-oriented PC-Stable	GPU	CUDA C with R interface	[6]
stable.fast	Classic PC-Stable	Multi-core CPU	C++ with R interface	[7]
ParallelPC	Parallel PC-Stable	Multi-core CPU	R	[10, 12]
Proposed	Conditioning Set Filtering	FPGA, Multi-core CPU	C and OpenCL	This paper

Table 3: Hardware platforms

Platform	CPU	GPU	FPGA
Model	Intel Xeon Silver 4110	NVIDIA GeForce RTX 2080 Ti	Intel Arria 10 GX 10AX115S2F45I1SG
Lithography	14nm	12nm	20nm
Base frequency	2.1GHz	1545 MHz	240MHz
Cores	8 physical / 16 logical cores	4352 CUDA cores	N/A
Memory	192GB DDR4	11GB GDDR6	16GB DDR4 via PCIe 2.0x4

Table 4: Resource usage

Resource	ALUT	RAM	FF	DSP
Total	854400	2713	1708800	1518
Used	495233	2421	520650	921
Used(%)	58%	89%	30%	61%

result, we omit detailed results of this GPU implementation since they provide insufficient evidence to support our opinion.

5.2 Trade-off between accuracy and speed

The accuracy-speed trade-off for skeleton discovery and CPDAG discovery are respectively shown in Figure 5 and 6. Each sub-figure contains the accuracy and speed for one data set. The accuracy measure is the structural Hamming distance (SHD). A lower SHD means higher discovery accuracy. The speed measure is the total execution time in seconds. An implementation appears in a sub-figure if and only if it finishes execution in 7200 seconds. We avoid using the number of floating-point operations per second (FLOPs/s) as a speed measure. Due to the difference in algorithm and data, FLOPs/s does not decide accuracy or efficiency in causal discovery.

The primary conclusion from the results is that the proposed FPGA-accelerated implementation achieves a superior trade-off between accuracy and speed. Graphically, the superiority of the trade-off corresponds to the fact that the points representing CSF in all sub-figures are close to the origin of the coordinate system. Also, we have the following observations:

(1) The accuracy of CSF is high in both skeleton discovery and CPDAG discovery. Specifically, CSF achieves the best accuracy in 10 out of 12 data sets in skeleton discovery and 11 out of 12 data sets in CPDAG discovery. The key factor behind the high accuracy is as follows. The number of CI tests provided by the FPGA-accelerated generator is small. The small quantity of CI tests allows complex but accurate CI tests to finish within a reasonable time on CPUs. In contrast, to pursue high speed, the other four tools only simple but inaccurate CI tests.

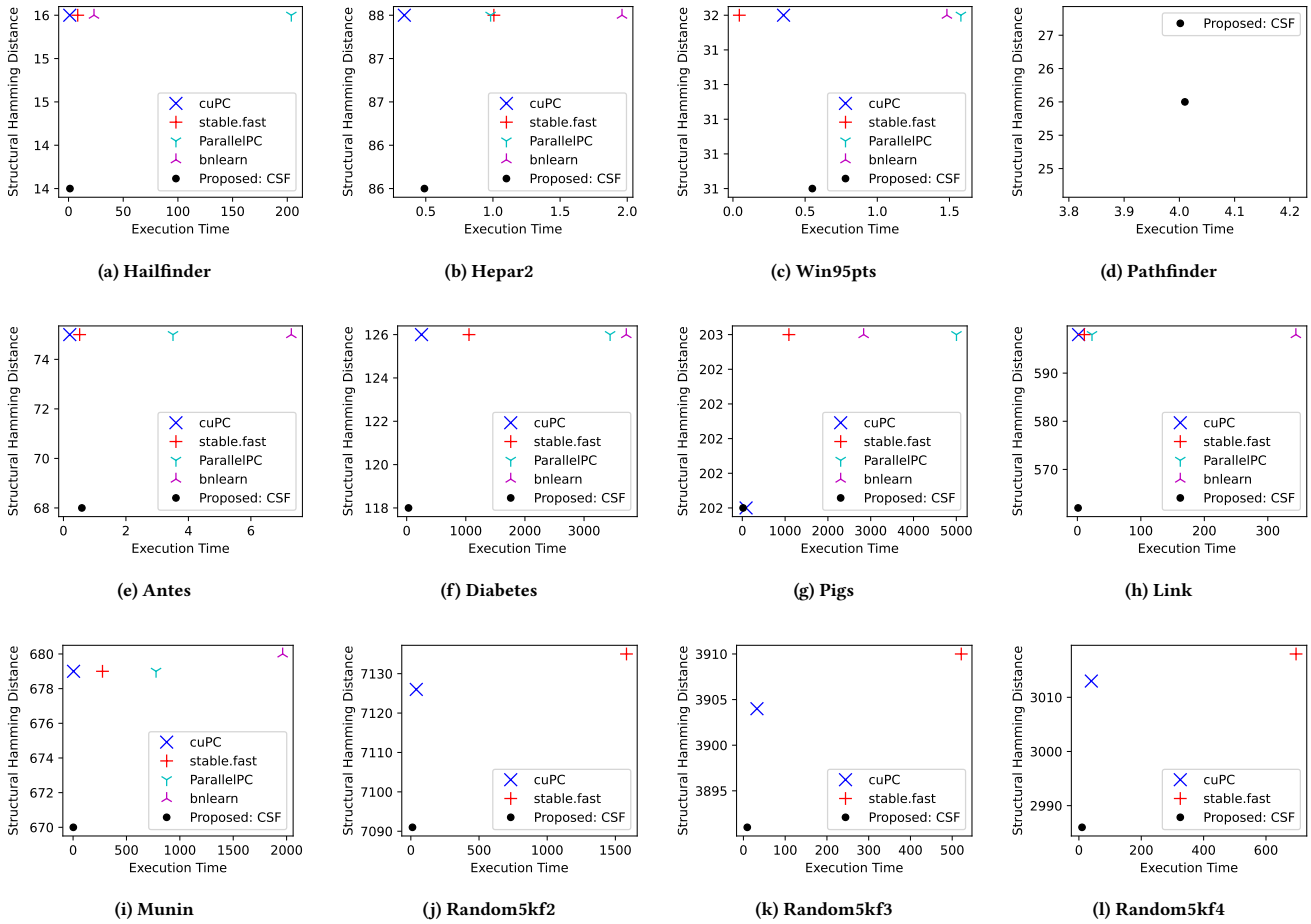
(2) The speed of CSF is outstanding for high-dimensional data sets. In both discovery tasks, CSF achieves the highest speed in 7 out of 12 data sets, namely Pathfinder, Diabetes, Pigs, Mulin, Random5kf2, Random5kf3, Random5kf4. In particular, CSF is the only approach that finishes within 7200 seconds for Pathfinder. Using Table 1 as a reference, we find that the 7 data sets are of high dimensionality. The reason behind the dimensionality–speed correlation is that a higher dimensionality results in more CI tests for the PC-stable algorithm. In this case, the overhead, including intra-platform communication and FPGA initialization, becomes negligible. In contrast, when the dimensionality and the number of CI tests are low, CSF's overhead dominates the execution time. In this case, executing all the CI tests in PC-stable can be less computationally expensive than the overhead-dominated CSF.

(3) When CSF does not demonstrate an advantage on speed for a data set, there is little need for acceleration since software tools are sufficiently fast. The 5 data sets on which CSF fails to achieve the highest speed are Hailfinder, Hapar2, Win95pts, Andes, and Link. The causal discovery problems with these 5 data sets are not as computationally demanding as the other data sets. In particular, the fastest CPU implementation, stable.fast, takes less than 5 seconds to finish. Practically, it is unnecessary to invest effort to further improve speed in such a situation.

In addition, we conduct an experiment to observe whether the bandwidth of the FPGA interface bottlenecks the speed. The PCIe 2.0x4 interface of the FPGA card provides a total bandwidth of 8 Gb/s. In the experiment, we run the interface in PCIe 1.0 mode to cut the bandwidth by half. However, we observe no changes in the speed throughout all experiments.

5.3 Speed comparison at fixed accuracy

We aim to provide a direct speed comparison between our CSF-based experimental implementation and other causal discovery tools. However, the experiments on the accuracy-speed trade-off do not support a fair speed comparison because different tools end



(a) Hailfinder (b) Hepar2 (c) Win95pts (d) Pathfinder

(e) Antes (f) Diabetes (g) Pigs (h) Link

(i) Munin (j) Random5kf2 (k) Random5kf3 (l) Random5kf4

Figure 5: Skeleton discovery: accuracy (structural Hamming distance, better lower) vs speed (execution time, better lower)

up with different accuracy. We believe that the comparison is fair only if we can fix a common accuracy level. Fortunately, we can terminate our experimental CSF implementation early to obtain a shorter execution time and a less accurate causal graph. In other words, we can downgrade the accuracy of CSF to the same level of reference tools for a fair speed comparison.

The comparison between CSF and a reference method is as follows. First, we measure the execution time T_{ref} and the corresponding structural Hamming distance (SHD) D_{ref} of the reference method. Next, we run CSF and take the execution time T_{CSF} when the SHD reduces to D_{ref} or less. Finally, we calculate the speedup by taking the ratio $\frac{T_{\text{CSF}}}{T_{\text{ref}}}$.

We focus on the CPDAG discovery task in this experiment. Regarding data, we only use the data sets that meet the following two conditions. The first condition is that CSF should achieve the highest accuracy with the data set. This condition guarantees that CSF can reach the accuracy of the reference implementation. The second condition is that at least one non-FPGA tool can finish within 7200 seconds. Six data sets meet both conditions: Diabetes, Pigs, Mulin, Random5kf2, Random5kf3, Random5kf4. Regarding causal discovery tools, we focus on cuPC and stable.fast since they can finish execution for any of the six data sets within 7200 seconds.

Table 5: Execution time at fixed accuracy level

Reference	Network	D_{ref}	T_{ref} (s)	T_{CSF} (s)	Speedup
cuPC	Diabetes	126	251.10	28.68	8.8
cuPC	Pigs	227	88.51	18.87	4.7
cuPC	Mulin	1018	5.16	1.80	2.9
cuPC	Random5kf2	8132	61.22	12.02	5.1
cuPC	Random5kf3	4568	49.83	9.51	5.2
cuPC	Random5kf4	3454	58.16	10.08	5.8
stable.fast	Diabetes	126	1054.17	28.68	36.8
stable.fast	Pigs	504	1100.72	9.51	115.7
stable.fast	Mulin	874	277.31	2.29	121.1
stable.fast	Random5kf2	8381	1670.96	10.73	155.7
stable.fast	Random5kf3	5039	602.94	7.75	77.8
stable.fast	Random5kf4	3982	778.35	9.62	80.9

The results are shown in Table 5. In all tests, the proposed FPGA implementation can reach the accuracy of cuPC and stable.fast

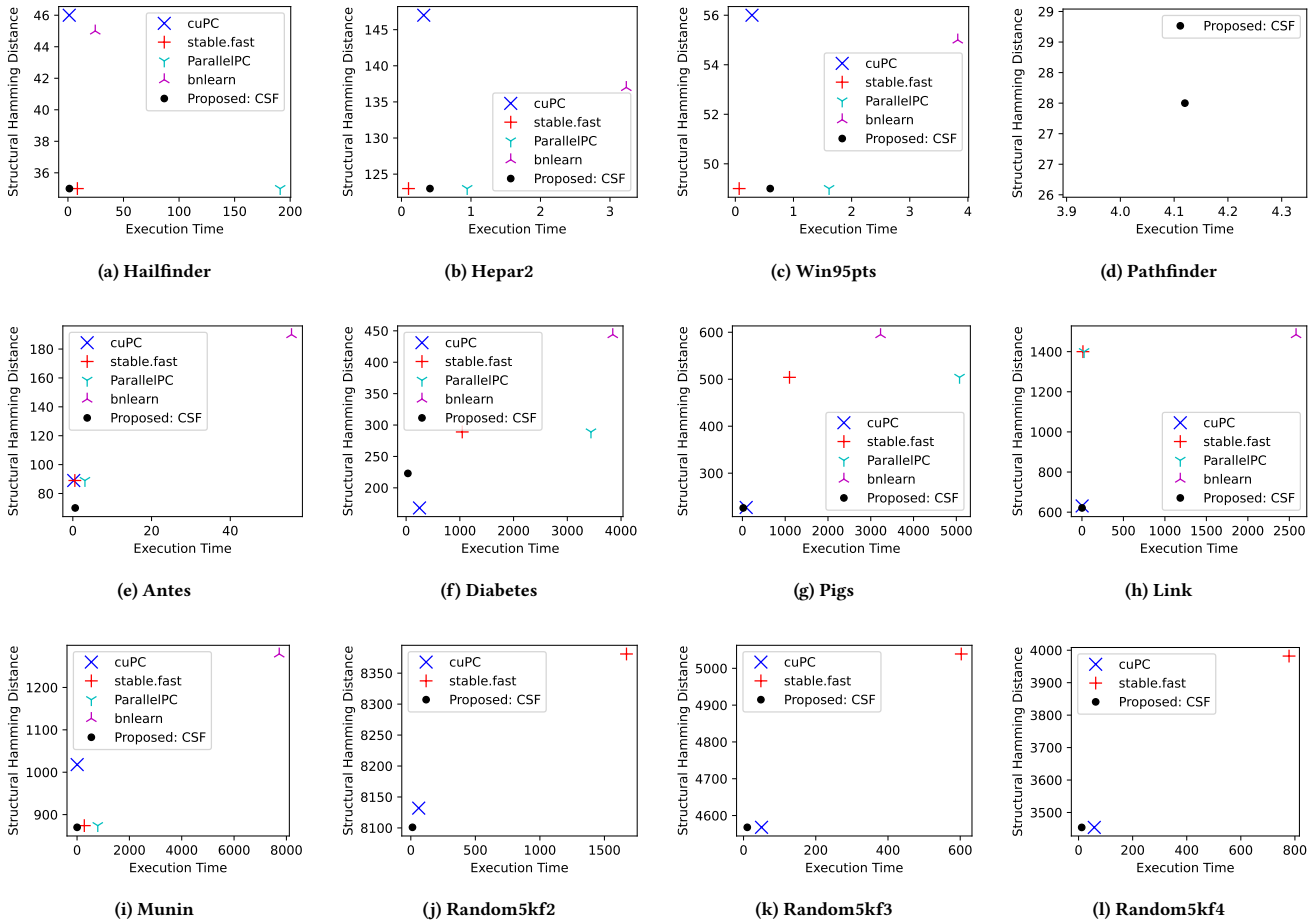


Figure 6: CPDAG discovery: accuracy (structural Hamming distance, better lower) vs speed (execution time, better lower)

within 30 seconds. The CSF-based implementation achieves 2.9–8.8 times speedup against cuPC and 36.8–155.7 times speedup against stable.fast.

6 CONCLUSIONS AND FUTURE WORK

Causal discovery is a critical but computationally demanding procedure in data mining and knowledge discovery. A common speed bottleneck for constraint-based causal discovery is the execution of CI tests. Conventional acceleration strategies speed up the execution of CI tests by running them in parallel. However, it is difficult to accelerate this calculation using FPGAs.

Instead of accelerating the execution of CI tests using FPGAs, we propose an acceleration strategy that shifts the speed bottleneck from the FPGA-unfriendly execution procedure of CI tests to the FPGA-friendly generation procedure of the tests. The proposed approach allows the execution of CI tests to finish within a reasonable time on CPUs. On the other hand, we develop an FPGA-accelerated design to remove the new speed bottleneck, CI test generation. An OpenCL implementation on an Intel Arria GX FPGA demonstrates a superior trade-off between accuracy and speed in 12 causal discovery problems. The implementation achieves up to 8.8 times speedup over the cuPC software running on an NVIDIA RTX 2080 Ti GPU. It also shows up to 155.7 times speedup over the stable.fast software running on an octa-core Intel Xeon Silver 4110 CPU.

Directions of future work for enhancing the proposed approach include: (i) design-level and implementation-level hardware optimizations, (ii) bottleneck shifting techniques for other computationally demanding problems, (iii) cloud-based designs with multiple CPUs, GPUs and FPGAs, (iv) tools that automate the proposed approach, and (v) applications for biomedical science and agent-based modeling.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the shepherd for their valuable comments and suggestions. The support of UK EPSRC (grant number EP/V028251/1, EP/L016796/1 and EP/N031768/1) and Intel is gratefully acknowledged.

REFERENCES

[1] P. Spirtes and K. Zhang, "Causal discovery and inference: concepts and recent methodological advances," in *Applied informatics*, vol. 3, pp. 1–28, SpringerOpen, 2016.

[2] C. Glymour, K. Zhang, and P. Spirtes, "Review of causal discovery methods based on graphical models," *Frontiers in genetics*, vol. 10, p. 524, 2019.

[3] M. Kalisch and P. Bühlmann, "Causal structure learning and inference: a selective review," *Quality Technology & Quantitative Management*, vol. 11, no. 1, pp. 3–21, 2014.

[4] T. D. Le, L. Liu, A. Tsykin, G. J. Goodall, B. Liu, B.-Y. Sun, and J. Li, "Inferring microRNA–mRNA causal regulatory relationships from expression data," *Bioinformatics*, vol. 29, no. 6, pp. 765–771, 2013.

[5] G. F. Cooper, I. Bahar, M. J. Becich, P. V. Benos, J. Berg, J. U. Espino, C. Glymour, R. C. Jacobson, M. Kienholz, A. V. Lee, *et al.*, "The center for causal discovery of biomedical knowledge from big data," *Journal of the American Medical Informatics Association*, vol. 22, no. 6, pp. 1132–1136, 2015.

[6] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, "cuPC: CUDA-based parallel PC algorithm for causal structure learning on gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 530–542, 2019.

[7] M. Kalisch, M. Mächler, D. Colombo, M. H. Maathuis, and P. Bühlmann, "Causal inference using graphical models with the R package pcalg," *Journal of statistical software*, vol. 47, no. 11, pp. 1–26, 2012.

[8] D. Marbach, J. C. Costello, R. Küffner, N. M. Vega, R. J. Prill, D. M. Camacho, K. R. Allison, M. Kellis, J. J. Collins, and G. Stolovitzky, "Wisdom of crowds for robust gene network inference," *Nature methods*, vol. 9, no. 8, pp. 796–804, 2012.

[9] M. Scutari, C. E. Graafland, and J. M. Gutiérrez, "Who learns better Bayesian network structures: Accuracy and speed of structure learning algorithms," *International Journal of Approximate Reasoning*, vol. 115, pp. 235–253, 2019.

[10] T. D. Le, T. Xu, L. Liu, H. Shu, T. Hoang, and J. Li, "ParallelPC: an R package for efficient causal exploration in genomic data," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 207–218, Springer, 2018.

[11] D. Colombo, M. H. Maathuis, *et al.*, "Order-independent constraint-based causal structure learning.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3741–3782, 2014.

[12] T. D. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, "A fast PC algorithm for high dimensional causal discovery with multi-core PCs," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 16, no. 5, pp. 1483–1495, 2016.

[13] C. Hagedorn and J. Huegle, "GPU-accelerated constraint-based causal structure learning for discrete data," in *SIAM International Conference on Data Mining (SDM)*, pp. 37–45, SIAM, 2021.

[14] M. J. Anderson and P. Legendre, "An empirical comparison of permutation methods for tests of partial regression coefficients in a linear model," *Journal of statistical computation and simulation*, vol. 62, no. 3, pp. 271–303, 1999.

[15] J. Runge, "Conditional independence testing based on a nearest-neighbor estimator of conditional mutual information," in *International Conference on Artificial Intelligence and Statistics*, pp. 938–947, PMLR, 2018.

[16] R. Sen, A. T. Suresh, K. Shanmugam, A. G. Dimakis, and S. Shakkettai, "Model-powered conditional independence test," in *International Conference on Neural Information Processing Systems*, pp. 2955–2965, 2017.

[17] R. Sen, K. Shanmugam, H. Asnani, A. Rahimzamani, and S. Kannan, "Mimic and classify: A meta-algorithm for conditional independence testing," *stat*, vol. 1050, p. 25, 2018.

[18] K. Baba, R. Shibata, and M. Sibuya, "Partial correlation and conditional correlation as measures of conditional independence," *Australian & New Zealand Journal of Statistics*, vol. 46, no. 4, pp. 657–664, 2004.

[19] M. Scutari, "Learning Bayesian networks with the bnlearn R package," *Journal of Statistical Software*, vol. 35, no. i03, 2010.

[20] M. Karkooti, J. R. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm," in *Asilomar Conference on Signals, Systems, and Computers*, 2005.

[21] A. Hadizadeh, M. Hashemi, M. Labbaf, and M. Parniani, "A matrix-inversion technique for FPGA-based real-time emt simulation of power converters," *IEEE Transactions on Industrial Electronics*, vol. 66, no. 2, pp. 1224–1234, 2018.

[22] G. A. Darbellay, "An estimator of the mutual information based on a criterion for conditional independence," *Computational Statistics & Data Analysis*, vol. 32, no. 1, pp. 1–17, 1999.

[23] A. Srivastava, S. P. Chockalingam, and S. Aluru, "A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2020.

[24] H. Li and T. Adali, "A class of complex ica algorithms based on the kurtosis cost function," *IEEE Transactions on Neural Networks*, vol. 19, no. 3, pp. 408–420, 2008.

[25] T. Kollo, "Multivariate skewness and kurtosis measures with an application in ica," *Journal of Multivariate Analysis*, vol. 99, no. 10, pp. 2328–2338, 2008.

[26] M. Rahman, A. Rasheed, M. Khan, M. A. Javidian, P. Jamshidi, M. Mamun-Or-Rashid, *et al.*, "Accelerating recursive partition-based causal structure learning," in *International Conference on Autonomous Agents and Multiagent Systems*, 2021.

[27] Y. Pang, S. Wang, Y. Peng, X. Peng, N. J. Fraser, and P. H. Leong, "A microcoded kernel recursive least squares processor using FPGA technology," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 1, pp. 1–22, 2016.

[28] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-GPU systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2010.

[29] N. Karunadasa and D. Ranasinghe, "Accelerating high performance applications with CUDA and MPI," in *2009 International Conference on Industrial and Information Systems (ICIIS)*, pp. 331–336, IEEE, 2009.

[30] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 1085–1097, IEEE, 2013.

[31] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[32] B. Sauk, N. Ploskas, and N. Sahinidis, "GPU parameter tuning for tall and skinny dense linear least squares problems," *Optimization Methods and Software*, vol. 35, no. 3, pp. 638–660, 2020.

[33] N. Loizou, S. Vaswani, I. H. Laradji, and S. Lacoste-Julien, "Stochastic Polyak step-size for SGD: An adaptive learning rate for fast convergence," in *International Conference on Artificial Intelligence and Statistics*, pp. 1306–1314, PMLR, 2021.

[34] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, pp. 421–436, Springer, 2012.

[35] M. B. Greenwald and S. Khanna, "Power-conserving computation of order-statistics over sensor networks," in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 275–285, 2004.

[36] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi, "Holistic aggregates in a networked world: Distributed tracking of approximate quantiles," in *ACM SIGMOD international conference on Management of data*, pp. 25–36, 2005.

[37] B. Haeupler, J. Mohapatra, and H.-H. Su, "Optimal gossip algorithms for exact and approximate quantile computations," in *ACM Symposium on Principles of Distributed Computing*, pp. 179–188, 2018.

[38] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *ACM SIGMOD international conference on Management of data*, pp. 611–622, 2005.

[39] H. Zhang, S. Zhou, and J. Guan, "Measuring conditional independence by independent residuals: Theoretical results and application in causal discovery," in *AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[40] H. Zhang, K. Zhang, S. Zhou, J. Guan, and J. Zhang, "Testing independence between linear combinations for causal discovery," in *AAAI Conference on Artificial Intelligence*, vol. 35, pp. 6538–6546, 2021.

[41] Y. Nitta and H. Takase, "An FPGA accelerator for Bayesian network structure learning with iterative use of processing elements," in *International Conference on Field-Programmable Technology*, pp. 29–34, IEEE, 2020.