Concurrency and Computation: Practice and Experience

WILEY

SPECIAL ISSUE PAPER OPEN ACCESS

Scaling Invariant Generation Using State Space Embeddings and GPU Streaming

Ben Lloyd-Roberts | Filippos Pantekis | Phillip James | Liam O'Reilly | Michael Edwards

Department of Computer Science, Swansea University, Swansea, UK

Correspondence: Ben Lloyd-Roberts (ben.lloyd-roberts@swansea.ac.uk)

Received: 20 May 2025 | **Revised:** 26 August 2025 | **Accepted:** 4 October 2025

Funding: This work was supported by the EPSRC Center for Doctoral Training in Enhancing Human Interactions and Collaborations with Data and Intelligence Driven Systems (EP/S021892/1).

Keywords: distributed systems | formal verification | massively parallel | reinforcement learning

ABSTRACT

The formal verification of railway control systems can ensure the safety of complex scheme plans through techniques such as induction-based model checking. While inductive verification performs well in complex settings, it often produces false positives due to its consideration of transitions from unreachable safe states to unsafe states. Invariants that reduce the state space to an over-approximation of reachable states, excluding transitions from safe to unsafe states, can help remove these false positives. However, such invariants are difficult to deduce automatically. In previous work, we have demonstrated that reinforcement learning (RL) and descriptive statistics can be used to generate candidate invariants, trivially within small programs, and can be realistically scaled to industrial settings using hardware-accelerated implementations. This paper extends our hybrid approach that uses General Purpose Graphics Processing Units (GPGPUs) to overcome these challenges while scaling effortlessly horizontally and in a distributed manner. We detail the implementation of a three-kernel pipeline responsible for consuming streamed data and computing correlation coefficients. Finally, we present a breadth of time samples that highlight the performance and scalability of this approach.

1 | Introduction

Formal verification of railway control systems is a field with substantial academic attention [1–12]. Research within the domain has focused on developing techniques that verify the safety of large interlocking systems controlling railways and their signaling systems. Successful approaches for scaling have included abstraction techniques [8], decomposition techniques [9, 10] and undertaking induction-based model checking [3, 4, 7]. It is often the case that approaches involving inductive verification (IV), providing good performance when successful [3, 7], can exhibit so-called false positive counterexamples when attempting to verify a given safety property.

Within such settings, a transition from a safe state to an unsafe state is a safety violation and often highlighted as a counterexample to a safety property that one would like to show holds for the system under consideration. However, the source state can sometimes be unreachable (due to over-approximation), meaning this becomes a false positive counterexample [3]. Invariants are properties that reduce the state space to exclude such problematic transitions. However, automatically deducing invariants for a given state space remains non-trivial and computationally expensive [13].

Previously, it has been shown that it is possible to use reinforcement learning (RL) and simple measures of correlation

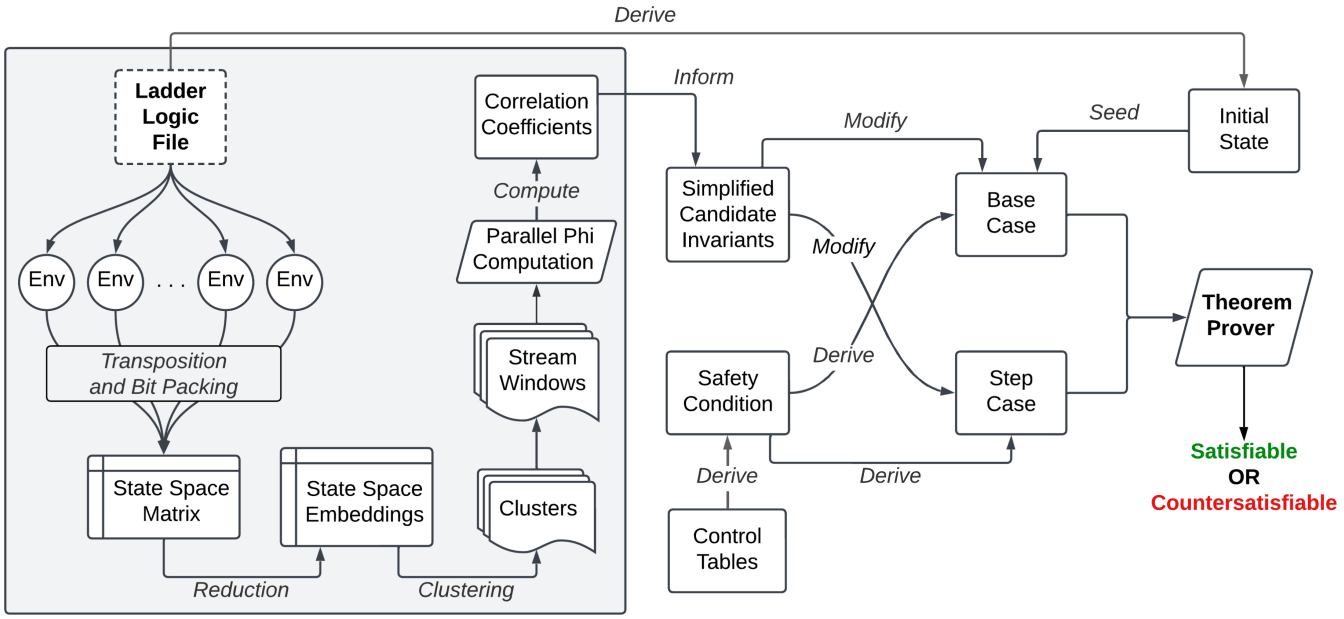

FIGURE 1 | Overview of parallelized invariant mining framework. Ladder logic programs are modeled as reinforcement learning environments where unique state observations are aggregated to form a state matrix. Throughout exploration, snapshots of the produced state matrix are reduced and clustered using unsupervised learning to reveal latent patterns between program variables composing the reachable state space. Variables within these clusters are sampled and passed to the GPU, where massively parallel correlation analysis is performed. Resultant coefficients are then translated into candidate invariants and incorporated in the verification process to help prove safety conditions. The grayed area denotes the processes discussed in this paper.

to generate candidate invariants for reducing state spaces when performing inductive verification [14, 15]. Figure 1 provides an overview of the framework we have designed and continue to refine for such an approach. First, the control systems under verification, composed of Ladder Logic programs, undergo a formal mapping of program-based state spaces to a reinforcement learning environment. Thereafter, reinforcement learning agents can be incentivized to explore large regions of state spaces, before using agent observations to analyze correlations across states that emerge throughout the exploration. Due to memory constraints, it is unlikely that the complete set of reachable states can be observed for large, complex systems. Consequently, candidate invariants are generated from a subset of reachable states and must be verified using a theorem prover before being considered "true" invariants. Once an invariant has been proposed and verified, it can be incorporated into the safety property we are trying to prove.

Recently, we demonstrated how specialized hardware can accelerate correlation coefficient calculations during the invariant generation phase [16]. This paper extends that work by illustrating how phases of the proposed framework support parallel execution and enable scalability, predominantly bounded by available hardware resources. This revised approach represents a novel framework facilitating a multi-phase pipeline with several stages, lending itself to parallelism and concurrent operation to produce large datasets of invariants for a given ladder logic program.

Due to the seemingly disparate fields composing this framework, each major section of this paper introduces the relevant background before presenting the contributions. Note, the greyed section of Figure 1 highlights the primary phases of our invariant mining framework discussed in this paper, as the systematic proving of generated invariants and their use in proving safety properties in the railway domain are beyond the scope of this work.

Section 2 provides an overview of Ladder Logic formal verification, related work, and the invariant finding problem for inductive verification. Section 3 introduces reinforcement learning and explains how state-of-the-art model-free algorithms enable Ladder Logic programs to be modeled as RL environments, supporting parallel exploration and training within a unified framework. Section 4 revisits our previously proposed method for invariant generation [16], and extends it with new heuristics to improve invariant expressiveness and address the combinatorial complexity of computing statistical relationships in large observation matrices. Section 5 presents our use of dimensionality reduction and unsupervised learning to structure program state spaces in a way that allows for statistical partitioning of the data sent to the GPU for correlation computation. Finally, Section 6 describes the implementation of scalable, GPU-based correlation computation in detail.

2 | Formal Verification and Invariants

The failure of critical systems can have catastrophic consequences. It is imperative that these systems undergo rigorous efforts to check their correctness according to formal specifications, before their deployment, to ensure they adhere to strict safety criteria. Conventionally, verification processes involve modeling the different states a system can assume during its operation and systematically checking if formally encoded safety properties are respected by all reachable system states.

2.1 | Modeling Ladder Logic

Many industrial critical systems, PLCs [17] in particular, are programmed using Ladder Logic, a Boolean language stemming from the control of electrical relays. These programs, or Ladders, are constructed using Boolean variables and logical connectives. Figure 2 illustrates a rudimentary example, henceforth referred to as P_1, consisting of four variables, or latches: x, y, c and d. A latch represents a Boolean contact, denoted by a pair of vertical bars, or a coil, denoted by parentheses. Contacts are marked as "normally closed" by a diagonal line, or "normally open" where the line is omitted, representing negated and non-negated Booleans, respectively. Logical connectives between contacts form expressions, or rungs, which are evaluated and assigned to coils. Horizontal and vertical connections between contacts represent logical conjunction and disjunction, respectively. Thus, P_1 comprises three horizontal connections and no vertical connections. While P_1 does not capture the complexity of industrial Ladder Logic, it does possess the requisite properties to illustrate how such programs can be systematically modeled and used to derive useful invariants to help inductive verification.

Both Kanso et al. [3] and James et al. [4, 6, 8], demonstrated the systematic translation of Ladder Logic to propositional formulae, representing Ladder Logic program verification as a SAT problem. Figure 3 provides a high-level illustration of how this can be applied to a railway interlocking. A Ladder Logic program is translated into a model comprising initial configurations and a mathematical transition function. Formally, the model in this case is a Labeled Transition System (LTS) composed of a set of states and connecting transitions. States in this context describe unique configurations a system may assume during its operation, while transitions describe how changes occur during operation. Concurrently, a generic safety specification is translated into a concrete propositional safety condition. This property is bespoke for a given railway plan and forms the basis for the verification of safety.

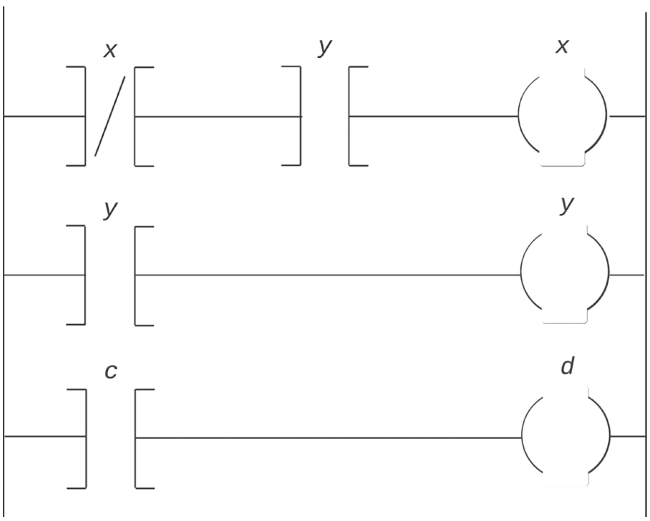

FIGURE 2 | Rudimentary Ladder Logic program for controlling P_1. Ladders represent Boolean programs composed of rungs, contacts, and coils. Ladders are executed from top to bottom, with the right-most entity representing the output, and the remaining entities representing latches used to compute the output.

Following definitions of a Ladder Logic formula, as outlined by James et al. we construct a propositional representation of P_1 in terms of conjunct equivalence relations between coils and rungs, denoted ψ_{P_1}:

$$((x' \leftrightarrow (\neg x \wedge y)) \wedge (y' \leftrightarrow y) \wedge (d' \leftrightarrow c))$$

Following definitions outlined in Reference [6], we model ψ_{P_1} as a Ladder Logic Labeled Transition System, $LTS(\psi_{P_1})$. This produces a set of states composed of unique valuations over program coils, denoted $\text{Val}_C = \{x, y, d\}$, and inputs, denoted $\text{Val}_I = \{c\}$. Transitions between these states are determined by the semantics of the underlying program P_1. Figure 4 illustrates the complete state space of $LTS(\psi_{P_1})$ as a directed graph. Here, nodes represent states and edges represent transitions from one state to another. Nodes are enumerated $s_0, \ldots, s_n$ and edges are annotated with elements of Val_I, that is, the value of a program input used to induce state transitions. We differentiate between coils and program inputs within state nodes by highlighting the latter in boldface. Note that the set of initial states Val_0 should form part of a system's functional specification. For our example, valid initial states are those where $x \wedge y$ holds. Thus, $\text{Val}_0 = \{s_1, s_3, s_5, s_7\}$. These nodes are identified in Figure 4 by a dashed outline. The grey region sectioning the state space represents the subset of states satisfying a safety property, Φ_{P_1}, introduced in Section 2.2.

2.2 | Invariant Finding

James et al. [6] explored two differing approaches to model checking: Bounded model checking (BMC) and inductive verification (IV). The former unrolls the transition function from initial states up to a given k-bound [18]. Every unrolled state is then checked to see if it upholds the given safety property. This results in one of two outcomes. A positive result indicates the property is true for all system states up to the given bound. Conversely, a counterexample trace highlights a run of the system from an initial state up to a state where the property is violated. The issue with this approach is that the bound needed to ensure that all reachable states are evaluated is unknown. Additionally, the resources needed to explore up to that bound grow exponentially as the bound increases, making a complete check often infeasible for complex systems. Inductive verification leverages two checks, a base-case and a step-case, to ensure the safety property is satisfied by initial states and subsequent states following a set of transitions, respectively. This approach has been shown to be relatively cheap in terms of resources needed, by reasoning about a system algebraically, to show that certain properties hold [19]. However, this abstraction may introduce states that produce false-positive counterexamples. These indicate a safety condition is violated by some state, but fail to consider whether the states involved are indeed reachable from the initial states of the system.

These problematic states can be eliminated by introducing sufficiently strong invariants [13] that exclude unreachable states from consideration, that is, properties which hold for at least all reachable system states. Derivation of such invariants can require extensive manual analysis of the software under verification by an experienced engineer. Automatic synthesis of invariants is a known problem in both academia and industry, with considerable diversity in proposed solutions and application

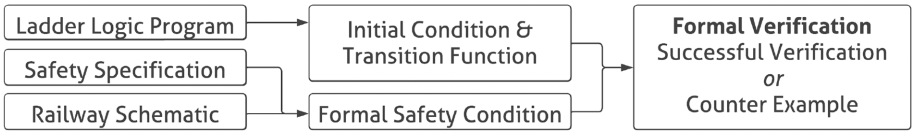

FIGURE 3 | Pipeline for formally verifying interlockings against a set of safety criteria.

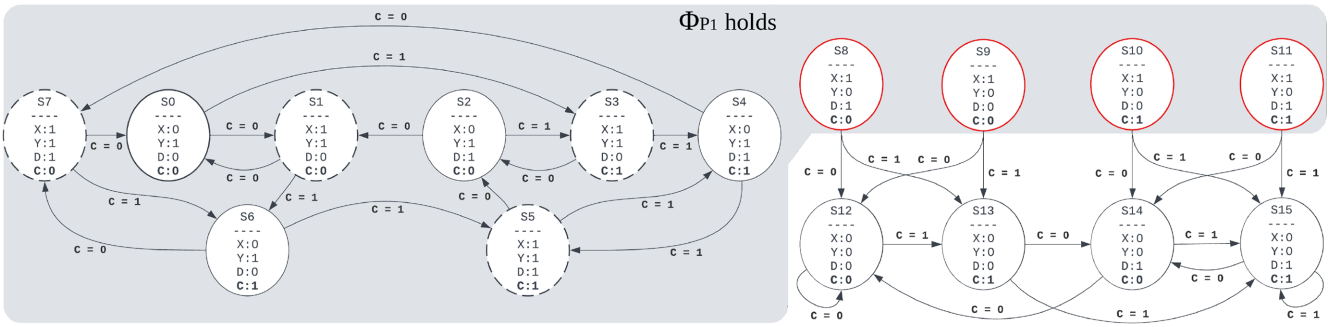

FIGURE 4 | Depiction of the Ladder Logic Labeled Transition System $LTS(\psi_{P_1})$. The gray region indicates the states where the safety property Φ_{P_1} holds (See Section 4.2). Nodes with a dashed outline represent the initial states from Val_0. The red-highlighted states are problematic for induction-based verification as they are unreachable safe states that transition to unsafe states.

domains [20, 21]. However, for complex programs, this becomes computationally expensive.

Given invariants are properties that persist across system states; one reasonable approach is to explicitly compute and observe as many unique reachable states as possible to analyze how the system configurations change. One could apply traditional systematic graph traversal algorithms to achieve state space coverage, but such approaches are known to perform sub-optimally when confronted with the combinatorial explosion of large state spaces [22].

For P_1, we treat $\Phi_{P_1} = x \vee y$ as an example safety condition that must be satisfied by our model and to be checked through inductive verification. Figure 4 depicts two disjoint sets of states for $LTS(\psi_{P_1})$. First, any initial states must satisfy Φ_{P_1}. The grey region indicates a subset of states, $s_0, \ldots, s_{11}$, where Φ_{P_1} holds. Applying a base check to the initial states, s_1, s_3, s_5, s_7, and a step case to the remaining reachable states, s_0, s_2, s_4, s_6, demonstrates they satisfy Φ_{P_1}. However, this approach also considers states $s_8, \ldots, s_{11}$ under the step-case, which transition to $s_{12}, \ldots, s_{15}$, violating Φ_{P_1} and producing false positive counterexamples.

Generating sufficiently strong invariants for the step-case can help avoid such transitions being checked during inductive verification, and avoid producing false positive counterexamples. Through subsequent sections of this work, we demonstrate how this problem can be automated by translating $LTS(\psi_{P_1})$ to an alternative model where reinforcement learning can be applied to help generate data necessary to propose candidate invariants.

3 | Reinforcement Learning and Exploration

Reinforcement learning (RL) is a machine learning paradigm designed to optimize sequential decision-making problems. Given a learning environment, RL algorithms use simulated experiences to build empirical models or software agents. These agents learn to map observations of their environment to the optimal sequence of actions over time, according to some reward scheme. In the following section, we introduce the general RL framework and how its components are adapted for invariant finding.

3.1 | Background

Formally, an RL environment is represented as a Markov Decision Process (MDP) [23]. A finite discounted MDP M is a five-tuple $(S, A, P_a(s, s'), R_a(s, s'), \gamma)$. Here, S is a finite set of states, known as the state space or observation space. It models environment configurations at discrete time steps t and provides context for agent decision-making. The action space A describes a set of actions an agent may perform from a given state s, at a particular time step. Transition probabilities $P_a(s, s') = P(s_{t+1} = s' | s_t, a_t)$ describe the likelihood of observing state s_{t+1} given action a_t is taken from state s_t. Note, this probability allows MDPs to model both stochastic [24] and deterministic systems [25], the latter applying to models of Ladder Logic programs. Reinforcing negative and positive associations between state-action pairs is the reward function, $R_a(s, s')$. This function informs an agent via a scalar signal r, where action a was taken from state s and produced state s'. Designing a reward function conducive to optimal decision-making is both task-dependent and non-trivial [26]. We discuss our approach to reward shaping for invariant finding in Section 3.2. Finally, $\gamma \in [0, 1)$ is a discount scalar applied to successive rewards considered over future time steps. Experiences are accumulated according to one of two simulation paradigms: Episodic or continuous. The distinction between the two approaches can be clarified in terms of an environment's reset logic. Episodic schemes, which this work adopts, train agents over a finite sequence of T_{Max} episodes, beginning from an initial state s_0 to some terminal state s_T. An agent's trajectory summarizes its experience accumulated over a finite number of

time steps, denoted $\tau = (s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_h, a_h, r_h)$. Here, h denotes the horizon; a time step beyond which rewards no longer influence an agent's prediction. Rewards observed from some time step t up to a terminal time step T are denoted $G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ and referred to as the return. RL training algorithms aim to leverage this return and experiences from an agent's previous trajectories to enable prediction within, and control over, the environment.

The first challenge of training RL agents entails approximating a value function that attributes a value to arbitrary environment states, indicating the quantitative benefit of occupying that state according to its reward potential. State values and state-action values, denoted $v(s)$ and $q(s, a)$ respectively, provide this estimate. Values are updated based on an empirical average, known as the expectation taken over observed returns, defined $\mathbb{E}[G_t | S_t = s, A_t = a]$. The second training objective of control refers to optimizing a behavior policy, $\pi(a|s)$, mapping states to the actions most likely to maximize cumulative rewards. Optimal value functions and optimal policies are reciprocal in that converging to one will converge to the other [27]. Optimization of both functions requires agent trajectories to accumulate experience of states, actions, and their ensuing rewards. In complex environments, these functions are approximated using deep neural networks [28] and gradient-based optimization [29]. So-called actor-critic [30] methods also approximate a parametric, typically state value, function $\hat{v}(s, \omega)$, learning action selection and value estimates independently. In this work, we employ a principle actor-critic algorithm, Proximal Policy Optimization (PPO) [31].

3.2 | Ladder Logic Exploration

Representing the invariant finding problem as a reinforcement learning task necessitates certain considerations. First, proposed invariants should be inferred from a large proportion of reachable system states. Second, these invariants must have a correct and formal representation when used by verification tools. Existing mappings from a Ladder Logic LTS to an MDP [14] provide the foundation for this work. Given $LTS(\psi_{P_1})$, the observation space of the corresponding Ladder Logic MDP, $M(\psi_{P_1})$, has the form $S = \mathrm{Val}_C \cup \mathrm{Val}_I$. Here, a unique valuation over the union of coils and inputs constitutes a distinct state, or observation, of $M(\psi_{P_1})$. The action space $A = \mathrm{Val}_I$ consists of program inputs, given that these values are determined externally and induce state transitions. For each state, $P_a(s, s') = P(s_{t+1} = s' | s_t, a_t)$, assigns probability 1 for transitions dictated by the semantics of the underlying Ladder Logic program P_1 and 0 for transitions that are not. As agents explore $M(\psi_{P_1})$, the environment unfolds as a set of reachable states S, that reflect those of $LTS(\psi_{P_1})$. The principal difference in this context is that states are computed explicitly, ensuring state observations do not include the problematic states described in Section 2.2. Such a framework allows us to progressively optimize state space coverage, record unique state observations from agent trajectories, and suggest candidate invariants using simple measures of binary correlation [32]. Aiming to maximize state space coverage, we implement a reward scheme that positively rewards novel observations over distinct episodes, deterring loop traversal with episode termination and negative reward signals. Additionally, we adopt count-based exploration reward strategies [33] and initial state sampling using the least-visited states from past experience [34].

3.2.1 | Environment Considerations

Given our objective of maximizing novel state discovery, deep reinforcement learning (DRL) is unnecessary for small state-action spaces such as $M(\psi_{P_1})$, where dynamic graph traversal [35] suffices. However, for larger Ladder Logic MDPs, the combinatorial explosion of states renders exhaustive search infeasible (see Section 3.2.2). In such settings, approximate learning methods can offer more effective exploration heuristics [36]. Candidate invariants discovered during exploration can then be verified through inductive checks. To that end, we focus on three core aspects: Novel state discovery, invariant synthesis, and verification via theorem proving.

Our simulation framework adopts an episodic RL paradigm with a novelty-driven reward scheme. If the agent revisits a previously seen state, the episode is terminated with a small negative reward and reset. This discourages loops while enabling reward-based performance differentiation across episodes. Inspired by prior work on count-based exploration [33], we incorporate an initial state sampling strategy [34], where states visited in previous episodes become candidates for new episode initializations, prioritizing less frequently visited states.

While value functions and policies drive agent behavior, the primary mechanism remains reward maximization. Thus, reward design must align with the problem domain and learning objective. In our case, this objective is the maximization of unique state observations. A simple reward function issuing positive signals for new states and negative ones for repeats is effective, though its success depends heavily on the structure of the environment. As state-action mappings become more complex, approximating the optimal policy becomes increasingly difficult, reinforcing the importance of well-shaped rewards and efficient exploration strategies.

3.2.2 | Industrial Ladder Logic

Our approach to state discovery has been validated on both synthetic Ladder Logic programs with known reachable state counts, for example, in P_1, and industrial ladder logic where such metrics are unknown. In the largest synthetic environment, agents achieved full coverage of up to 4.3 million states, despite a theoretical upper bound of 2^{50}.

In terms of industrial Ladder Logic programs, we consider four examples, denoted `IL-A`, `IL-B`, `IL-C`, and `IL-D`, each implementing part of a railway interlocking system. The complexity of each system is characterized by the number of binary variables in its observation and action spaces. `IL-A` comprises an observation space of 3565 binary variables and an action space of 1250 binary variables. This configuration reflects a moderately sized interlocking system. `IL-B` increases in complexity, with an observation space of 6452 and an action space of 2222 binary variables. The growth in dimensionality corresponds to a broader set of monitored components and control mechanisms. `IL-C`

expands further to 8944 observation variables and 2983 action variables. IL-D represents the most complex system, featuring 15,150 binary observations and 5008 binary actions.

Although industrial programs have even larger theoretical state spaces, practical limits such as memory and training time constrain exploration. Nevertheless, our approach reached 45 million unique states in an industrial setting, with no indication of stagnation. These results demonstrate the utility of heuristic guidance [37] and reward design [38] in avoiding redundant behavior and promoting continued exploration under uncertainty.

3.3 | Parallel Exploration

Training RL agents within this framework introduces two complementary requirements. Firstly, invariant mining critically depends on accumulating a sufficiently representative set of distinct state observations to ensure comprehensive exploration of the underlying state space [39]. Secondly, effective policy optimization via RL algorithms necessitates the collection of diverse experiences linked to cumulative rewards, ensuring statistically informative and stable policy updates [31, 40].

However, deploying a single agent within a solitary environment instance frequently fails to meet these requirements. Such a configuration results in slow exploration, sparse encounters with rare states, and high variance in policy gradient estimates, all of which impede convergence and generalization [30]. A practical solution to these limitations is parallelization, whereby multiple environment instances and separate workers sharing a policy are spawned by a parent process to perform rollouts independently and periodically synchronizing to aggregate collected experiences [41, 42]. Note that the RL framework's parent process is also capable of communicating with utility processes throughout the agent.

Parallelization involves running M identical environment instances concurrently, each generating trajectory segments of length T. At each synchronization interval, trajectories are aggregated, producing a collective batch of MT transitions for policy updates. Such a parallel architecture results in two primary benefits, each of which is explored in the following sections.

3.3.1 | Improved State Coverage

Independent rollouts initiated with distinct random seeds can expedite the exploration process across diverse regions of the state space. Formally, defining the state trajectory for worker $m \in 1, \ldots, M$ as

$$T^{(m)} = \{s_0^{(m)}, s_1^{(m)}, \ldots, s_T^{(m)}\}, \quad s_{t+1}^{(m)} \sim P(\cdot|s_t^{(m)}, a_t^{(m)}) \tag{1}$$

where P refers to the environment's transition function. After k synchronization steps, the set of unique states encountered across all workers can be defined as:

$$U_k = \bigcup_{m=1}^{M} \bigcup_{t=0}^{T-1} T_{k,t}^{(m)}, \tag{2}$$

where state coverage is quantified by $C_k = |U_k|$. Assuming that each rollout is independent, the expected number of unique states grows approximately as

$$\mathbb{E}[C_k] \approx |S|\left(1 - (1 - p_T)^{kM}\right), \tag{3}$$

where p_T is the probability that a particular state is visited in a single rollout of length T, and $|S|$ is the size of the state space. This growth is quasi-linear with the number of workers M. This rapid state accumulation is crucial for robust statistical analyses throughout training. Practically, given our state space comprises binary vectors, bit packing and hashing each state $\alpha : S \to 1, \ldots, L$ maintains computational tractability.

3.3.2 | Learning Efficiency via Parallel On-Policy Rollouts

Parallel rollouts also improve the stability and efficiency of policy updates. In PPO, policy parameters θ are updated by maximizing a surrogate objective function that constrains the deviation between successive policies:

$$\hat{L}(\theta) = \frac{1}{MT} \sum_{(s,a,\hat{A}) \in B} \min\left(r_\theta(s,a)\,\hat{A}, \text{clip}(r_\theta(s,a), 1-\varepsilon, 1+\varepsilon)\,\hat{A}\right), \tag{4}$$

where the following probability ratio quantifies how much the updated policy deviates from the previous policy:

$$r_\theta(s,a) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}. \tag{5}$$

and $\hat{A}$ defines the estimate of the advantage function. The estimated gradient of this objective function, $g(\theta)$, necessary for updating policy network parameters, is estimated from a batch B of experiences, in this case collected by multiple parallel workers, and can be expressed as:

$$g_B(\theta) = \frac{1}{MT} \sum_{(s,a,\hat{A}) \in B} \nabla_\theta \log \pi_\theta(a|s)\,\tilde{A}(s,a). \tag{6}$$

Here, $\tilde{A}(s,a)$ represents the clipped advantage function, guiding policy updates by quantifying the benefit of taking an action a from a state s. Crucially, when trajectories generated by parallel workers are independent, the variance of this gradient estimator diminishes proportionally to the size of the batch:

$$\text{Var}[g_B(\theta)] \approx \frac{1}{MT}\sigma^2, \tag{7}$$

where σ^2 denotes the intrinsic variance of the gradient estimate from a single trajectory. This reduction in variance is particularly valuable, as lower gradient variance yields more stable and informative updates, directly enhancing the rate and reliability of convergence.

However, employing parallel rollouts necessitates careful consideration of several practical aspects. Notably, the synchronization interval, the length of trajectory segments each worker collects before updating the policy, must be carefully balanced. If it is too small, frequent communication incurs substantial overhead; if too large, policy updates risk becoming outdated. Empirically,

choosing synchronization intervals in the range [1024, 2048] has been shown to strike an effective balance between communication overhead and gradient updates. Additionally, overlapping rollout collection with the learning updates helps to maximize computational throughput, ensuring efficient utilization of resources and minimizing idle time.

3.4 | State Matrix Representation and Sampling

The outcome of the reinforcement learning process is a growing state matrix $M_S \in \{0,1\}^{n \times t}$, where each column represents a unique state observed during exploration, and each row corresponds to one of the n Boolean variables defined by the underlying Ladder Logic program. The height n of the matrix is fixed, determined by the program's variable set, $|\text{Val}_c \cup \text{Val}_I|$, while the width t increases as new environment states are discovered and aggregated across parallel agents.

This matrix is continuously updated as the agents explore, with each new column $s^{(i)} \in \{0,1\}^n$ appended to M_S when a novel state is encountered. Because the matrix grows monotonically in width, subsequent phases of the framework, such as embedding, clustering, and correlation computation, can operate on snapshots of M_S, taken at regular intervals $t_0, t_1, \ldots$ communicated by the RL framework's parent process. These snapshots enable invariant mining to proceed concurrently with exploration, facilitating a pipelined architecture in which state discovery and structural analysis are decoupled but synchronized through periodic sampling.

4 | Candidate Invariant Generation

Invariants, and more broadly propositions, define a relationship, or dependency, between their constituent variables. Although methods exist to measure the mathematical relationships across a variety of observed data, the reliability of these statistics often hinges on the quantity and diversity of those observations. By accumulating a diverse dataset during the exploration of an RL environment, we can gather sufficient state observations to compute associations between program variables. We describe the process by which our data and the goal of constructing new candidate invariants guide the determination of these associations.

4.1 | Variable Correlation

Statistical measures of association can uncover complex relationships between data, offering a distinct advantage over simple observation. For instance, correlation coefficients and covariance are fundamental statistics used to measure relationships between two variables. Each measure has specific applications and assumptions for different types of data. The motivation for selecting a particular statistic for invariant finding lies in its ease of interpretation, especially when translating numerical values into propositional formulae. Additionally, the chosen statistic should be reliable and efficient, minimizing errors in approximation and computational overhead.

Covariance indicates the direction of a linear relationship between two random variables, but does not normalize the magnitude of this association. In contrast, ϕ-correlation coefficients, an adaptation of Pearson's correlation for dichotomous data [32], are tailored for binary pairs, making them favorable for Boolean values. Moreover, ϕ-coefficients can be computed using counts of empirical observations, without relying on weighted probabilities to calculate, thus reducing the need for approximation. As we will explore in the following section, ϕ-coefficients also support the construction of multi-variate properties with little additional computational cost.

4.2 | Correlation Coefficients

We introduce the use of ϕ-correlation coefficients as a basic statistical measure of association between binary variables. Let a and b be two arbitrary Boolean variables. We compute a correlation coefficient:

$$\phi = \frac{n_{ab}\, n_{nanb} - n_{anb}\, n_{nab}}{\sqrt{n_{a\bullet}\, n_{na\bullet}\, n_{\bullet b}\, n_{\bullet nb}}} \tag{8}$$

where n_{ab} refers to the count of observations $(a \wedge b)$, n_{nanb} the count of $(\neg a \wedge \neg b)$, n_{nab} the count of $(\neg a \wedge b)$ and n_{anb} the count of $(a \wedge \neg b)$. Variables forming the radicand refer to total counts, where $n_{a\bullet}$ is the sum of counts where a holds, $n_{na\bullet}$ is the sum of counts where $\neg a$ holds, $n_{\bullet b}$ is the sum of counts where b holds, and $n_{\bullet nb}$ is the sum of counts where $\neg b$ holds. Conventionally, these values are represented via contingency tables [43]. Note, for propositional logic, these coefficients rely on truth values and, thus, can be computed for arbitrary atoms, as with a and b, or propositions composed of arbitrary atoms, such as $(\neg a \Rightarrow b)$ and $(c \wedge \neg a \Rightarrow d)$. The truth values for each respective multi-variate expression can easily be determined with the existing valuations over their constituent atoms. The limit on generating complex propositions then relies on the resources required to determine their truth value.

Coefficients between two binary pairs, therefore, depend on their observed variability with respect to one another. Such values are normalized, falling within the closed interval $[-1, 1]$. Binary pairs resulting in a coefficient of 1 share a complete positive correlation, that is, they are synchronously true or false. Conversely, coefficients of -1 represent a complete inverse relation between the binary pair, indicating they always represent the negation of each other. Extrema of ϕ thus present us with a systematic means of inferring invariance between arbitrary variable pairs.

To increase the likelihood of proposing useful invariants, we extend the computation of ϕ-coefficients to include not only the atoms composing P_1, that is, x, y, d and c, but their negation, $\neg x$, conjunction, $x \wedge y$, and disjunction $x \vee y$. Determining ϕ for all unique combinations of binary pairs produces $\frac{N(N-1)}{2}$ values, where N represents the number of variables or propositions composing those binary pairs. The complete expansion process is explained in greater detail in Section 4.3. Table 1 shows a subset of the coefficients calculated from observations of $M(\psi_{P_1})$, concerning variables x and y. Rows and columns failing to contain ϕ extrema have also been excluded, leaving propositions worth considering for invariant generation.

TABLE 1 | Partial correlation matrix computed from $M(\psi_{P_1})$. Duplicate coefficients and coefficients describing self-correlation have been set to 0.

	$\neg x$	$x \wedge y$	$x \vee \neg y$	$y \wedge \neg x$	$\neg x \vee \neg y$
x	−1.0	1.0	1.0	−1.0	−1.0
$\neg x$	0.0	−1.0	−1.0	1.0	1.0
$x \wedge y$	0.0	0.0	1.0	−1.0	−1.0
$y \wedge \neg x$	0.0	0.0	0.0	0.0	1.0
$x \vee \neg y$	0.0	0.0	0.0	−1.0	−1.0

Consider the complete inverse relation between $\neg x$ and $x \vee \neg y$ in Table 1. As a proposition, this can be expressed as the negation of their conjunction, that is, $\neg(\neg x \wedge (x \vee \neg y))$, given they never share the same truth value. Note that expressions of this form can be further simplified using De Morgan's Law and logical identities, making the aforementioned invariant equivalent to $x \vee (\neg x \wedge y)$. Complete positive correlations, as exhibited between $y \wedge \neg x$ and $\neg x \vee \neg y$, can be expressed as equivalence relations, that is, $(y \wedge \neg x) \leftrightarrow (\neg x \vee \neg y)$ given they will always share the same truth value. Both this expression and its simplified form of y serve as a sufficiently robust invariant to exclude states $s_8 \ldots s_{15}$ when verifying the safety property Φ_{P_1} we introduced in Section 2.2.

Estimates of ϕ are computed based on the variability of any two variables across the reachable state space, meaning all states must be observed to compute the true correlation. Consequently, ϕ converges over time, warranting a distinction between three types of invariants as described by coefficients. Extrema of −1 or 1 in the event of complete state space coverage may be considered "true invariants". We define "candidate invariants" as those generated from ϕ extrema, irrespective of state coverage, but have yet to be verified via an inductive check. "Possible invariants" concern ϕ values produced by undefined operations. This occurs when one of the total counts sums to 0, causing a division-by-zero error and producing a `NaN`. In such cases, the binary pair may indeed lead to a candidate invariant, but not without further state space exploration. These invariants allow for the iterative generation of properties as exploration yields more state observations. This does, however, present a number of computational challenges as the number of variables, propositions, and observations increases. As indicated in Reference [16], we turn to specialist hardware and optimization to scale this approach.

4.3 | Expansion of State Observations

As previously noted, invariants derived from single atomic propositions, that is, those involving only a single pair of Boolean variables, are typically insufficient to constrain the model to a degree that eliminates spurious counterexamples during inductive verification. To increase the expressive capacity of candidate invariants, the set of base state observations is expanded using logical connectives to construct more complex, multivariate propositions. While this expansion increases the likelihood of discovering robust invariants, it also substantially enlarges the hypothesis space, thereby increasing the number of features N over which statistical correlations must be computed. This, in turn, imposes additional computational and memory requirements. Specifically, the number of ϕ-correlation coefficients produced grows quadratically with N, which is lower-bounded by the number of Boolean variables comprising each environment state.

Let $M_S \in \{0,1\}^{m \times n}$ denote the initial Boolean matrix representing m observed environment states over n Boolean variables. The expansion procedure comprises two stages, both of which preserve the width n of the matrix while increasing its height. In the first expansion stage, M is vertically duplicated by appending the element-wise logical negation of each row, yielding a matrix $M'_S \in \{0,1\}^{2m \times n}$. Formally, if $M_S = [r_1; \ldots; r_m]$, then

$$M'_S = [r_1, r_2, \ldots r_m, \overline{r_1}, \overline{r_2}, \ldots, \overline{r_m}]$$

where $\overline{r_i}$ denotes the bitwise negation of row r_i.

In the second expansion stage, all unique unordered pairs of rows in M' are used to generate new rows via logical conjunction (AND) and disjunction (OR). Since M' contains $2m$ rows, the total number of unique unordered pairs is given by the binomial coefficient:

$$\binom{2m}{2} = \frac{2m(2m-1)}{2} \tag{9}$$

Each such pair produces two new rows, one for the conjunction and one for the disjunction, resulting in an additional $2 \cdot \binom{2m}{2} = 2m(2m-1)$ rows.

The total number of rows in the final matrix M''_S is thus:

$$2m + 2m(2m-1) = 2m(2m-1+1) = 4m^2 \tag{10}$$

Hence, the full expansion transforms the original matrix $M_S \in \{0,1\}^{m \times n}$ into an augmented matrix $M''_S \in \{0,1\}^{4m^2 \times n}$ that includes all original rows, their negations, and all pairwise logical conjunctions and disjunctions. This expansion increases the potential for expressing complex invariants, but incurs a corresponding computational cost due to the enlarged feature set, both in terms of retaining the expanded data and the ensuing impact on ϕ calculations, as demonstrated by Figure 5.

In the next section, we introduce a number of optimization strategies to mitigate these costs, including the application of dimensionality reduction techniques, clustering of semantically similar rows, and a streaming architecture that incrementally processes sub-matrices on the GPU. These approaches collectively reduce memory footprint and computational overhead while preserving the statistical fidelity required for invariant discovery.

5 | Data Reduction, Clustering and Window Construction

In this section, we outline a pipeline for identifying structurally related subsets of variables within high-dimensional Ladder Logic program state spaces. The goal is to reduce the computational burden of pairwise correlation analysis by embedding variables into a lower-dimensional space, where clustering can be used to guide selective ϕ-coefficient computation. We begin by introducing the embedding process, followed by visualizations of the clustered structures and a discussion of how these clusters

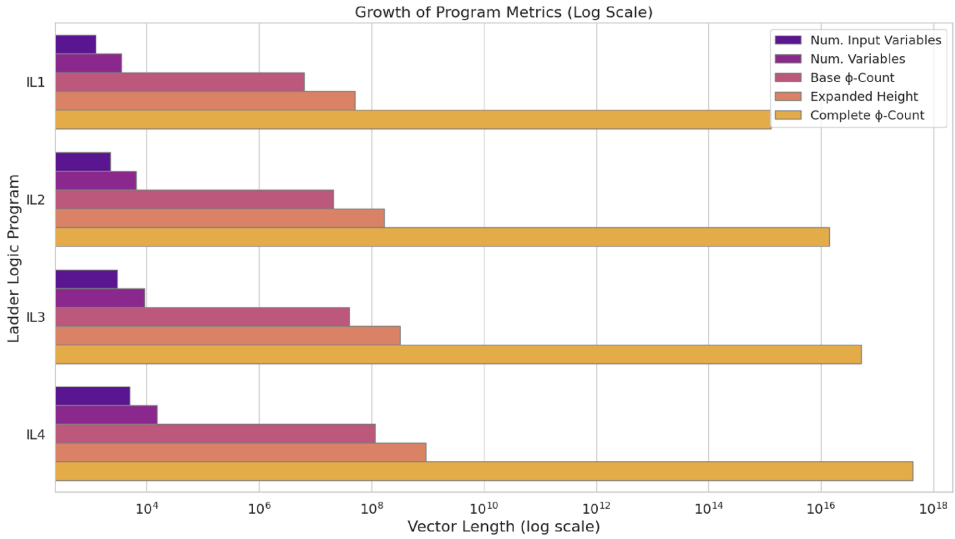

FIGURE 5 | Matrix rows required to represent state data before invariant generation can take place. "Num. Input Variables" refers to the number of values our RL agents can influence to trigger a state change. "Num. Variables" refer to the number of boolean values composing a single environment state. Expanded height describes the number of rows produced during the expansion phase we introduced in Section 4.3. The "Base" ϕ-count represents the number of correlation coefficients needed for a base state matrix without expansion. The "Complete" ϕ-count refers to the number of correlation coefficients needed to describe all unique row pairs within the expanded state matrix.

are subsequently used to define input windows for GPU-based processing.

5.1 | Generating State Embeddings

Given the prohibitive cost of exhaustively computing ϕ-correlation coefficients over the fully expanded state matrix, we seek efficient strategies for prioritizing variable pairs likely to exhibit statistical dependency. As discussed in Section 4.2, the total number of pairwise comparisons grows quadratically with the number of features N, many of which may be unrelated. Rather than allocating computational effort uniformly across the coefficient space, we aim to identify structured subsets of variables whose internal relationships justify more focused correlation analysis.

To this end, we employ dimensionality reduction as a precursor to unsupervised clustering, allowing us to project high-dimensional Boolean feature vectors into a low-dimensional space where local similarity between rows can be visually and algorithmically assessed. Among the many available techniques, t-distributed Stochastic Neighbor Embedding (t-SNE) [44] offers a compelling approach for this application. t-SNE preserves local structure in the data by mapping features such that pairwise similarities in high-dimensional space are approximated by corresponding similarities in a lower-dimensional embedding. This property makes t-SNE particularly suitable for identifying latent groupings among variables, even when their separability is not linearly defined.

Figures 6 and 7 show the result of applying t-SNE to two state matrices generated across separate exploration phases of the `IL-C` Ladder Logic program. State space samples were generated from a training run using PPO and a unique random seed.

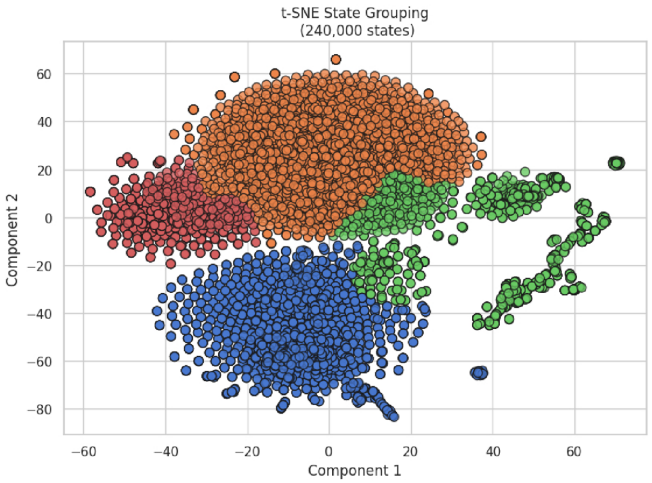

FIGURE 6 | t-SNE projection of 240,000 states observed from Ladder Logic program `IL-C`, grouped using agglomerative clustering.

Each cluster defines a candidate subset for focused ϕ-coefficient computation.

The groupings are colored based on applying hierarchical agglomerative clustering [45] to the learned embeddings. While the clustering outcome here is not used for downstream modeling or point classification, it enables us to partition variables into subsets on which ϕ-correlation computation can be concentrated, where local neighborhoods have been preserved. By leveraging this form of clustering on t-SNE embeddings, we can reduce the search space for pairwise correlation significantly, without relying on arbitrary sampling or manual feature selection. It is worth noting that t-SNE tends to distort global structure for the sake of visualization, thus inter-cluster distances should not be interpreted literally.

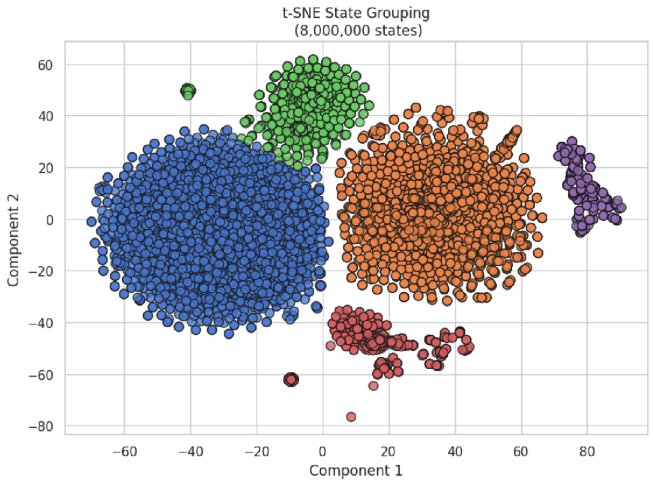

FIGURE 7 | t-SNE projection of 8,000,000 states observed from Ladder Logic program `IL-C`, grouped using agglomerative clustering.

5.2 | Window Generating for GPU Streaming

Once the state space embeddings have been generated and partitioned, the resultant variable clusters serve as a guide for structuring the proposal of invariants. Specifically, these clusters define candidate subsets of variables over which ϕ-correlation coefficients will be computed using GPU-accelerated matrix operations. Moreover, each data point within clusters represents a disjoint state observation vector, implying that each cluster can, in principle, be processed independently, enabling parallel processing across multiple GPUs.

The fundamental challenge at this stage is to construct observation windows that are sufficiently expressive and respect the limited global memory of the GPU. While the base state matrix may grow indefinitely in width due to ongoing reinforcement learning exploration, the GPU kernels must operate on bounded subsets, or slices, of this data (See Section 6.3). The cluster-derived variable groups thus form the starting point for constructing manageable sub-matrices to be expanded, as shown in Section 4.3, and streamed to the GPU for processing in discrete windows.

Each observation window is defined over a rectangular sub-matrix $W \in \{0,1\}^{h \times w}$. The window height h is not fixed a priori but is dynamically determined based on the size of the input cluster and the available GPU memory. The primary constraint on window construction is the number of ϕ-coefficients to be computed given the input data, which in this case scales quadratically with the number of rows in the expanded sub-matrix:

$$|\Phi| = \binom{h}{2} = \frac{h(h-1)}{2} \tag{11}$$

We begin with the smallest cluster obtained from the state space embedding, selecting the subset of variables it defines. Window construction proceeds in stages, beginning with the set of variables assigned to a given embedding cluster. Once a cluster is selected, the constituent points serve as a base data matrix of seed rows for the expansion phase. The window width w can be determined thereafter, representing how many state observations will be carried over to the GPU for each window's payload.

Before proceeding with full expansion, a feasibility check is performed to determine whether a complete application of the expansion algorithm, comprising negation, conjunction, and disjunction steps, would exceed the maximum window height permitted by the GPU's available global memory. If the full expansion is feasible, the entire process is carried out, producing a matrix of height $h = 4m^2$, as previously described in Section 4.3.

If, however, the cluster size renders a full expansion infeasible by exceeding the memory threshold, a truncated expansion is performed. In this case, the negation step is applied in full, doubling the seed rows to 2 m. The remaining allocated row space permits r additional rows calculated as the difference between the maximum allowable height and the post-negation height, then $\frac{r}{2}$ conjunction and $\frac{r}{2}$ disjunction rows are generated from randomly sampled or priority-ranked row pairs.

This adaptive approach ensures that expansion remains within memory limits while functioning as a heuristic designed to increase the representational capacity of the matrix. By systematically generating compound features, we capture latent relationships across variables. It also maintains the ability to process larger clusters partially, without discarding valuable structural information entirely.

By grounding window construction in the clustered structure of the data, identified through unsupervised learning, we prioritize variables with shared statistical behavior, thereby focusing computational effort on regions of the state space more likely to produce meaningful invariant structures. In the next section, we provide implementation-level details on how these windows are processed on the GPU to facilitate scalable ϕ computation.

6 | GPU Acceleration of Coefficient Calculation

Modern graphics processors expose specialized hardware for use in the acceleration of mathematically intensive tasks. We present one more contribution to the ever-growing list of non-graphics applications of GPUs, and specifically the acceleration of ϕ-correlation coefficient calculation. The work presented in this section is an improved form of our earlier work [16] that utilizes streaming to a GPU, as part of a heterogeneous system.

6.1 | Background

Graphics processors came to exist from the need for performing more complex graphics calculations at greater speeds [46]. In recent years, Graphics Processing Units (GPUs) have successfully been used in applications stretching far beyond the domain of graphics [47–52]. Some tasks, particularly those tangential to common graphics routines, naturally lend themselves to acceleration using GPUs regardless of which domain they fall under. This, in turn, shapes the evolution of GPUs into versatile co-processors that now include features that are less graphics-oriented and geared towards general computation. It must nevertheless be remembered that these devices are not

general-purpose processors [46] and are currently not suitable for all tasks. Generally, the aim being satisfied by using GPUs for a given task is performance. The specialized nature of these processors, combined with their high arithmetic throughput in certain operations, makes GPU acceleration lucrative for applications where timely completion is important.

The application case being presented in this, and our earlier work [16], is an example of an embarrassingly parallelisable task that features heavy reliance on arithmetic operations over large data sets and must be completed in a timely manner. Before exploring the intricacies of our use case, however, we must briefly cover some fundamental background on the model of computation employed by modern GPUs and some details of the hardware. For this work, we are focusing on NVIDIA GPUs and specifically those of the Ampere microarchitecture [53], although the approach applies to a range of GPU architectures.

GPUs are massively parallel processors, typically comprising thousands of cores and capable of managing billions of threads. Threads are treated as the smallest unit of execution and incur no context-switching penalties during execution. Computation in GPUs is encoded in the form of a kernel that contains code that each thread must execute. Code is written in high-level languages such as CUDA-C++ and is supported by CUDA libraries and toolkit. The conversion from high-level CUDA code to binary (SASS) code involves an intermediate translation of the high-level code into assembly (PTX) code. PTX instructions are well documented [54] and their direct use through the high-level code is facilitated, with special importance for low-level optimization. To the contrary, the conversion from PTX to SASS, the SASS ISA, and the performance of machine instructions on different architectures are not documented officially [55], yet the plethora of detailed literature contributions offer a remedy [56–60].

Threads are grouped into blocks, which are logical structures that can be mono-, bi-, or tri-dimensional and can comprise up to 1024 threads. Blocks themselves are in turn grouped into a grid which can also be mono-, bi-, or tri-dimensional and is far less limited in size. The geometry of the grid and block structures is relevant to tasks that exhibit spatial locality; however, tasks without such spatial properties are unaffected. Threads can differentiate themselves and the data they each operate on using their coordinates in their surrounding block and the coordinates of that block in the surrounding grid. Each block gets further partitioned into groups of 32 threads called warps. Threads in a warp are expected to operate in unison with one another, ideally executing all their instructions in lockstep. Where divergence occurs in the execution paths of threads within the same warp, this generally results in a performance penalty until threads re-converge.

GPU processors consist of tens of identical Streaming Multiprocessors (SMs). Each SM typically houses different types of processing units, such as 32-bit integer processing units. SMs are themselves partitioned into four identical partitions, each of which has its own processing units, warp scheduler, and register file. During execution, blocks from the grid are assigned to an SM where the warp schedulers manage execution time allocation between the pool of warps assigned to them, in conjunction with the available resources in their SM partition. The over-subscription of SM resources by multiple warps facilitates latency hiding, whereby a warp may be stalled awaiting for some operation to complete, while others continue to execute unaffected.

In terms of memory, there are several memory types on the GPU, each with its own visibility, access cost, and special considerations. Of most importance to this work are global memory, shared memory, and caches. Global memory is the largest memory type that resides off-chip and is the backbone for communication of data between the GPU (device) and host system. Access to global memory can be made from all threads, but incurs a substantial cost that is noted in literature [57] to be in the region of 290 clock cycles for the Ampere microarchitecture. On-chip L2 cache is available, which is accessible by all SMs, with global memory requests passing through it by default. L1 cache resides within each SM and only applies to global memory interactions performed by threads of blocks that reside in the SM. L1 and shared memory share the same physical memory space in each SM, with the carveout between them being configurable globally, prior to a kernel launch. Shared memory is inexpensive to access w.r.t. global memory, and is accessible by all threads of a block in an SM. Cross-block thread collaboration can be achieved by exchanging data through shared memory, yet it is also common for each thread to use shared memory as transient storage, independent of other threads.

When threads within a warp perform a global memory operation simultaneously, a set of memory transactions is performed. Global memory transactions are 32-, 64-, or 128-byte windows of memory contents in transit. The pattern in which threads access global memory, along with the size of the data and distance between them, dictates how many memory transactions are required. The fewer memory transactions required, the lower the latency of the interaction. A warp of threads accessing contiguous memory locations will trigger the most optimal set of memory transactions, which may mean that all data fits into a single memory transaction. A good access pattern typically results in better cache utilization and, by extension, hit rate. Beyond memory transaction minimization, threads within the same warp benefit from additional features, such as the ability to exchange register contents directly, that further aid their collaboration.

This cursory overview of GPU fundamentals serves as an illustration of the complexity of these systems. With carefully designed and optimized solutions, the capabilities of this hardware can be exploited to deliver great speedups.

6.2 | Single-Shot Phi Coefficient Calculation

In our earlier work [16], we presented an optimized, single-shot kernel for the computation of ϕ-coefficients between every distinct pair of columns of an input matrix storing unique state observations. To aid memory coalescing, the input matrix was initially rotated by 90°, turning columns into rows and vice-versa, before being packed. The packing step condensed the bytes of each row, each of which represents a truth value, into bit-packed words, effectively using each bit as a truth value instead. In our revised approach, the data structure responsible for storing aggregated state observations houses variables across rows

and appends new states as columns, removing the need for unpacking, transposition, and repacking.

Each thread in the kernel selected a distinct, relative to other threads, pair of rows from the transposed matrix and carried out the process of computing the ϕ correlation coefficient between them. The initial data transformation steps were undertaken to permit the threads of each warp to access chunks of their assigned rows in as few memory transactions as possible and in a coalesced manner, whilst also allowing each thread to process, in the manner described in Section 6.3.3, the 32 truth values in the same number of clock cycles as processing a single truth value.

This single-shot kernel is effective at performing this computation, but suffers from a number of limitations. More specifically, for the kernel to operate on the rows, the matrix has to fit entirely in GPU memory, limiting the maximum dimensions of the matrix. Additionally, the computation of ϕ-coefficients is necessarily performed by an individual thread, whilst the rest of its block remains resident but idle. Both limitations are addressed by our streaming approach, described in subsequent sections.

6.3 | Streaming Phi Coefficient Calculation

As described earlier, new columns are introduced to the matrix as new observations are made. To overcome the matrix size limitations imposed by the GPU's memory size in our earlier work, we employ a streaming approach whereby the matrix is streamed, in slices, to the GPU for the correlation coefficients to be computed. A slice, in this instance, is a sub-matrix that has the same height as the original matrix and a variable width. As new observations are added to the matrix, they are also being passed to the GPU side, in slices, for consideration in the ϕ-coefficient calculation.

The GPU computation is supervised by a process responsible for communicating with the RL framework. This process also manages the state of the GPU, on-device memory allocations, and kernel execution flow. In terms of kernels, three kernels are involved in the computation: The State Update, Phi Computation, and Post-Processing kernels.

6.3.1 | Supervisory Process Design

We designed the ϕ-coefficient calculation system as part of a discrete supervisory process that communicates with the RL framework via TCP. This design choice is predominantly motivated by scalability, as detailed in Section 6.4.

The supervisory process is responsible for initializing resources and memory allocations on the GPU(s) involved in ϕ computation, maintaining state persistence for these devices, and finally de-initializing the devices once the computation is finished. This process is also tasked with workload distribution since new slices of the state matrix may arrive at any point and must be consumed and handled as they arrive.

When a state matrix slice is sent to the supervisory process, which is diagrammatically depicted in Figure 8, the process allocates and initializes sufficient memory on the device before streaming received data into it. Following the arrival and transfer of the full slice, a State Update kernel is launched by this process to consume the received slice, updating the internal GPU state in the process, and discarding the slice thereafter. The process transitions onto a mid-stream state in which the height of subsequent slices is expected to match that of the first slice that was received, which will be treated as a continuation of the stream. In this state, new slices can continue to arrive until a termination request is received.

Once a termination request is received, the process transitions into a conclusion state in which a two-kernel pipeline is launched. The first kernel converts the on-device accumulated data into per-row-pair ϕ-coefficients, and is followed by the second kernel, which post-processes the data on the device to be in a format suitable for transmission. Subsequently, the computed ϕ-coefficients are returned via the TCP connection, and the process transitions back to its initial state. Return to the initial state involves the release of all device handles, memory allocations, and stored state in both device and host.

In anticipation of large matrix slices and/or memory-constrained GPU host systems, we opted to pipe received data from the network stack directly to the device, and vice versa, without fully staging them in a host-side buffer first. In practice, this is done using a small[1] intermediary transfer buffer on the host system, where data is read into before being transferred to the GPU or network connection, respectively. Despite repeated transfers to and from the GPU resulting in accumulated transfer overhead costs, we observed that the TCP transfer rate, even on local connections, was sufficient to hide these latencies. We probe no further into network and PCI-bus latencies as we are not in a position to adequately control factors affecting them, such as networking hardware. We do, however, anticipate their effect to be insignificant w.r.t. the computation at hand in the general case.

6.3.2 | Memory Management

As described in Section 6.3.1, state data is preserved in both host and device memory throughout the processing of streamed state matrix slices and until the eventual conclusion of the computation. State data held by the host is relatively small in size and not a priority in terms of resource management. To the contrary, on-device memory space is limited and a determining factor for the streamed data size limits.

All kernels involved in this computation interact primarily with global memory. During processing of the slices in the stream, the slice being processed, as well as the per-row-pair bit counts, need to co-exist in global memory. To that effect, when the first slice of height h and width w is being sent on the stream, the supervisory process allocates a contiguous block of $4 \times 8 \times \frac{h \times (h-1)}{2}$ bytes to house four 64-bit counters, and a region to house the incoming slice. For reasons outlined in Section 6.3.3, the byte length of each row in device memory must be divisible by 4. We therefore compute $w' = \lceil \frac{w}{32} \rceil \times 4$ before computing the byte size of the memory block needed as $(w' + p(w')) \times h$. Here, a block of memory is being allocated with the intention of storing data treated and traversed as a two-dimensional structure. To facilitate coalesced row access by the threads, additional padding is added on each row.

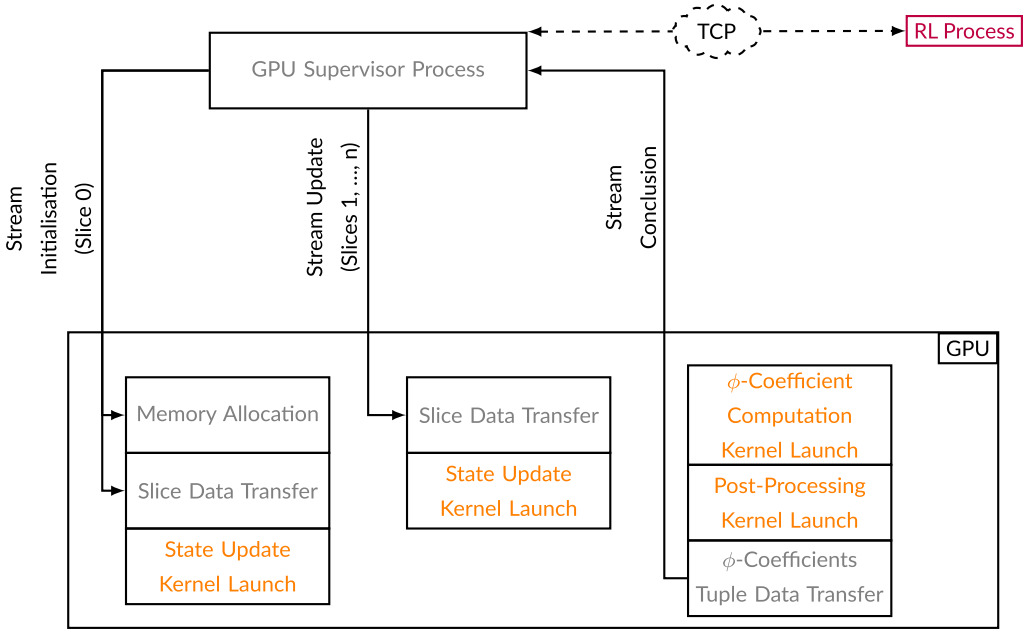

FIGURE 8 | Block diagram of interactions between the Supervisory Process and the GPU on command from the RL Process.

In the aforementioned size calculation, the function $p : \mathbb{N} \to \mathbb{N}$ maps the length of a row to a device-dependent constant padding amount. It is worth noting that the data found in any padding added is explicitly disregarded by the relevant kernels.

Both device memory allocations remain active throughout the processing of all state matrix slices in the stream. Since slices in the stream are only bounded in height, some may be wider/narrower than their predecessor. The memory block holding the slice is re-allocated only if the slice being received is greater in size than its predecessor. Naturally, the per-row padding applicable will vary based on the width of incoming slices and is computed dynamically on each occasion. To facilitate the padding of rows, it is necessary to hold a small intermediary buffer in host memory that is sufficiently large to accommodate a single row of a slice. Rows are implicitly padded during transfer from the host buffer to a specific offset in device memory, which includes padding w.r.t. the previously transferred row.

When processing of slices is concluded and the correlation coefficients are requested, the memory structure is altered by releasing the allocation of the slice buffer and replacing it with an allocation of a memory block of size $8 \times \frac{h \times (h-1)}{2}$, which is used to house the per-row-pair correlation coefficients. This is a transient buffer until post-processing takes place and the data is returned.

6.3.3 | State Update Kernel

The State Update kernel is responsible for consuming a slice of the state matrix and updating the device state accordingly. This kernel is launched once a slice is fully received into device memory by the supervisory process.

In this kernel, blocks of threads act as units that are each responsible for summarizing a distinct, w.r.t. other blocks, pair of rows (r_a, r_b). The rows corresponding to each block are determined by the index b of the block within the one-dimensional grid. More specifically, in a state matrix with h rows, the index i_a of row r_a is calculated using Equation (12) followed by index i_b of row r_b that is calculated using Equation (13).

$$i_a = \left\lfloor \frac{2 \times h - 1 - \sqrt{(2 \times h - 1)^2 - 8 \times b}}{2} \right\rfloor \tag{12}$$

$$i_b = b - \left(i_a \times h - \frac{i_a(i_a + 1)}{2} \right) + i_a + 1 \tag{13}$$

Conceptually, row index pairs are derived by traversing a triangular integer array consisting of h rows, where the t^{th} row is comprised of the integers $[t + 1, t + 2, \ldots, h]$. Pairing every element of this structure with its corresponding row index results in a set of distinct integer pairs that each act as an index for a row from the state matrix.

Once all s threads of a block have determined which row pair (r_a, r_b) their block is responsible for, they begin summarizing the rows collaboratively as a sliding window of width s over the row pair. For rows of width w, each thread performs $\lceil (w \div 32) \div s \rceil$ steps. On each step, each thread reads two corresponding 32-bit words b_a and b_b from rows r_a and r_b respectively, and computes the bitwise expressions $e_{ab} = b_a \& b_b$, $e_{anb} = b_a \& \sim b_b$, $e_{nab} = \sim b_a \& b_b$, and $e_{nanb} = \sim b_a \& \sim b_b$. As each word houses 32 truth values (i.e., bits) from each row, the aforementioned bitwise expressions effectively perform the respective logic operation between each pair of corresponding truth values in the word. For each of the resulting words e_{ab}, e_{anb}, e_{nab}, and e_{nanb}, the thread counts the number of set bits and accumulates them in the thread-local counters c_{ab}, c_{anb}, c_{nab}, and c_{nanb} respectively. It is worth noting that GPU registers are naturally 32-bit wide and bitwise and bit-count operations between 32-bit words are inexpensive [57, 58]. This motivated our choice to represent rows as bit vectors that are subsequently processed in units of 32 bits.

Padding bits added to rows to permit reading in 32-bit words must be discarded by threads so as not to form part of their respective counts. More specifically, for $0 \leq p < 32$ many padding bits, each thread in the block computes a bit mask $m = 0xFFFFFFFF << ((i+1) \times 32 - min((i+1) \times 32, w - p))$ as it is about to process the i^{th} 32-bit word. $m \neq 0xFFFFFFFF$ will only ever hold if $i = \lfloor \frac{(w-p)}{32} \rfloor$ and $p > 0$. In other words, the value of m has all bits set for all threads, except the thread processing the last 32-bit word that has been padded, in which case, the p least significant bits will be unset. Threads perform a bitwise AND between m and each of e_{ab}, e_{anb}, e_{nab}, and e_{nanb} before counting the set bits in each and updating the respective counts.

In terms of implementation, computing each word e_{ab}, e_{anb}, e_{nab}, and e_{nanb} in the aforementioned manner is simplified using hardware support for three-input per-bit logical operations in the form of the `lop3.b32` PTX instruction. For instance, given a mask m and two 32-bit words b_a and b_b from rows r_a and r_b respectively, e_{anb} is computed using `lop3.b32` e_{anb}, b_a, b_b, m, `0 × 20;`. Here, `0 × 20`, or in binary `0b00100000`, encodes the results column of the truth table of the expression $a\& \sim b\&c$ where the single set bit corresponds to the case where operands $a = c = 1$ and $b = 0$. This instruction results in a 32-bit word in which each bit is constructed by computing the aforementioned logic expression between the corresponding bits of the three input operands b_a, b_b, and m. Beyond simplicity, the `lop` instruction is computationally inexpensive [57], on occasion cheaper than computing the encoded logic formula expressly using discrete bitwise operations.

Naturally, handling of padding bits is only relevant to thread t_{s-1} of each block, as it will be the thread that operates on the final, potentially padded, word of each row. Thread divergence, when using a more intuitive conditional expression to compute the mask m, would therefore be of little significance. However, while examining the impact of such an expression, we observed the compiler flattening this branch using a pair of `setp` and `selp` PTX instructions. Based on data from literature [57, 58], the cost of this computation[2] is perhaps costlier than our non-divergent computation, which drove our choice of the non-divergent variant.

This kernel as a whole is designed in a manner to avoid branching during the row summarization steps of each thread. Threads in each warp compute in lockstep until they have exhausted their respective portions of the row pair. It is worth noting that, in cases where $\frac{w}{32}$ is not divisible by s, some threads may not participate in the last step of the computation. Threads that exit the computation sooner than their counterparts remain idle until they re-converge. We saw no benefit in addressing this divergence as its overall impact is minimal, even in the most degenerate case of only one thread participating in the final step (i.e., $\frac{w}{32}\ mod\ s = 1$).

On the thread level, each thread jumps s words forward in each processing step (i.e., coalesced access). This access pattern, though perhaps less intuitive, is chosen for minimizing the number of memory transactions required to fetch the respective words on each step. The 32 threads of each warp access a total of 128 bytes from each row, which are successive and in an appropriately aligned memory address to facilitate the minimum number of memory transactions possible.[3]

Once all threads in each warp have computed their respective counts in lockstep, they accumulate a warp-level sum of each of their four counts using a parallel reduction operation. These operations are natively supported among threads in the same warp and involve direct register exchange as opposed to memory interactions. As such, following the reduction operation, the first thread of each warp stores each of the sums in a corresponding, to the warp, section of shared memory. Once all warps have completed this operation, the first warp in the block performs a further reduction to accumulate a block-level sum of each of the four counts. Finally, the first four threads of the first warp each sum into successive memory locations in global memory, one of the four block-level counts before the block exits.

Whilst it is expected that no individual slice will be wider than the maximum value of a 32-bit unsigned integer to risk overflowing a counter in the kernel, the kernel itself is expected to be invoked numerous times in the processing of slices in the stream; hence, the per-block (i.e., per-row-pair) counts in global memory are stored as 64-bit values. This is a further benefit of our streaming approach, since processing each slice is done using exclusively 32-bit operations, while the final accumulation of collective counts is the only instance where costlier 64-bit operations are used.

6.3.4 | ϕ-Coefficient Computation Kernel

When the end of the stream is reached, the per-row-pair counts stored in device memory must be used to complete the computation of ϕ-coefficients as detailed in Section 4.2. To this end, a kernel is launched that maps every tuple of four counts into the result of the calculation. Threads in the kernel each correspond to a tuple of counts, which are the basis for computing the correlation coefficient between the rows. Threads read their corresponding counts e_{ab}, e_{anb}, e_{nab}, and e_{nanb} from global memory and proceed to calculate the result of Equation (14).

$$\phi = \frac{e_{ab} \times e_{nanb} - e_{nab} \times e_{anb}}{\sqrt{(e_{ab} + e_{nab}) \times (e_{anb} + e_{nab}) \times (e_{ab} + e_{nab}) \times (e_{nab} + e_{nanb})}} \tag{14}$$

Once all threads have completed their computation, they each write the result to the corresponding location in the results buffer. Whilst this operation is coalesced and completable with two memory transactions per warp, the same is not true of the initial read of tuples from global memory. Each thread reads four 64-bit values from global memory. It is therefore expected that the 1024 bytes of data fetched by each warp will, in current architectures, require several memory transactions. Moreover, global memory data movement instructions currently support a maximum of 16 bytes, which requires each thread to perform two memory fetches to receive their entire tuple. Despite tuples being coalesced in memory, the 16-byte constraint results in non-coalesced memory access. Fortunately, memory transactions triggered by the read of the initial 16 bytes from each thread return portions of memory, including the subsequently accessed 16 bytes of each thread. The excess data is cached in the SM-local L1 cache, which means that, in all likelihood, the subsequently accessed 16 bytes will result in a cache hit and no additional transaction will be triggered. In our tests of this kernel, this appears to be the case as the kernel achieves a near-optimal 46% L1 cache hit rate. It is worth noting

that the cache configuration used for this kernel is set to favor L1 cache space over shared memory, which is not utilized by this kernel.

In terms of compute utilization, the calculation at hand is heavy on double-precision operations, which cost significantly more than their single-precision counterparts. To put this into perspective, on devices of Compute Capability `8.6` and `8.9`, even simple operations between double precision operands, such as addition, have a throughput of 2 operations per clock cycle per SM [61] while their single-precision counterparts have a throughput of 128 per clock cycle per SM [61]. Devices of different Compute Capability versions feature different throughputs for single and double precision operations; however, throughput for the latter is consistently lower than the former.

In this instance, the combined effect of costly compute operations and high initial memory demand works favorably, as it is sufficient to hide the other, resulting in an even utilization of resources that approaches the theoretical limits of the devices profiled. As detailed in Section 6.3.7, with the optimizations already made, the wall-clock time impact of this kernel relative to the overarching computation is minimal, hence no further optimization efforts are required.

Finally, no data is transferred between this kernel and the host. Instead, the results buffer remains resident on device memory until the post-processing step is performed, as detailed in Section 6.3.5.

6.3.5 | Post-Processing Kernel

Following the computation of ϕ-coefficients by the kernel presented in Section 6.3.4, global memory on the device houses the per-row-pair counts alongside the per-row-pair ϕ-coefficients. The purpose of the post-processing kernel is to structure ϕ-coefficients in a manner that is suitable for transmission back to the RL framework.

There are as many threads in this kernel as row-pairs. As such, each thread corresponds to a ϕ-coefficient. Each thread identifies the indices i_a and i_b of the rows row_a and row_b that were involved in calculating the ϕ-coefficient to which it corresponds, in the manner described in Section 6.3.3.

Finally, threads write their corresponding tuple (i_a, i_b, ϕ) in global memory in a contiguous manner. The per-row-pair count buffer, which is still allocated at this point, is reused to house these tuples. Data in this buffer is no longer relevant and can thus be overwritten.

Like the ϕ-coefficient Computation kernel, this kernel is balanced in terms of compute and memory throughput. The runtime of this kernel is insignificant and its scope is limited, as it encodes a supplementary action to the ϕ-coefficient Computation. Despite this, we chose to keep the two kernels separate in an effort to reduce global memory operations whilst maintaining minimal global memory requirements.

Upon completion of this kernel, the global memory buffer previously used to house per-row-pair counts will contain the aforementioned tuples, ready for return to the RL framework via the host-side GPU supervisory process. Whilst the kernel and supervisory process do not currently filter data in any way, the decision to include row indexes in tuples was motivated in part by the option of implementing some form of filter in the future.

6.3.6 | Kernel Optimizations

Numerous opportunities for optimization presented themselves during the implementation of the kernel pipeline outlined in previous sections. From a design perspective, the three-kernel approach was driven by work-balancing and cache use factors. We believe that the level of decomposition of work is balanced and offers good performance without work degeneration. The nature of the task at hand is embarrassingly parallelisable, which reduces the need for thread-level decision making, that is, branching. Where a decision had to be made in the State Update kernel, we chose to manually flatten the branching code after observing a sub-optimal expression in the resulting PTX assembly, which carried over to the SASS code.

Beyond branching, analysis of emitted PTX showed an opportunity for fused multiply-and-add instructions on some occasions. For instance, in the ϕ-coefficient Computation kernel, the numerator $e_{ab} \times e_{nanb} - e_{nab} \times e_{anb}$ compiled[4] into the PTX code shown in Listing 1, where `%fd1`, `%fd6`, `%fd5`, `%fd2` are register variables corresponding to e_{ab}, e_{nanb}, e_{nab} and e_{anb} respectively. We observed that the instructions of this PTX code translated to their SASS equivalents without further optimization. We altered this part of the code, resulting in the PTX code shown in Listing 2, where a fused multiply-and-add instruction replaces the instructions for the multiplication of e_{ab} with e_{nanb} and subsequent subtraction of $e_{nab} \times e_{anb}$. In the PTX code shown, a `neg.f64` instruction is used to arithmetically negate the result of $e_{nab} \times e_{anb}$. In SASS, however, this negation forms part of the fused multiply-and-add instruction itself. Despite the absence in official documentation, arithmetic negation can form part of individual arithmetic instructions, as documented in literature [56, 57].

```
1 mul.f64       %fd16, %fd1, %fd6;
2 mul.f64       %fd17, %fd5, %fd2;
3 sub.f64       %fd18, %fd16, %fd17;
```

Listing 1 | Original PTX code for of $e_{ab} * e_{nanb} - e_{nab} * e_{anb}$.

```
1 mul.f64       %fd16, %fd5, %fd2;
2 neg.f64       %fd17, %fd16;
3 fma.rn.f64    %fd18, %fd1, %fd6, %fd17;
```

Listing 2 | Optimized equivalent of Listing 1.

6.3.7 | Performance of Kernels

In our earlier work [16], we demonstrated the applicability of our approach to a range of GPUs. For the evolution of that approach that is presented in this work, we focus on one GPU, the NVIDIA A100, and present a breadth of time samples gathered from a broad range of inputs. The NVIDIA A100 is an enterprise-level processor designed for compute tasks and was selected primarily

for its memory capacity, enabling us to collect time samples for larger inputs.

Time samples were gathered on a shared GPU compute cluster using one NVIDIA A100. To avoid potential influence by other jobs in the cluster, timing tasks held exclusive access to their assigned cluster nodes. Each job involved a simulation of a stream of ten equally wide matrix slices being sent to the supervisory process, followed by a request for the resulting ϕ-coefficients. The execution time of the state update kernel was measured for each matrix slice and averaged to produce a single measurement, while the ϕ-coefficient Computation and Post-Processing kernels, which execute in immediate succession, were timed as a unit.

To construct test inputs, a set of widths W = 1000, 4096, 8500, 131,072, 200,000, 262,144, 350,000, 524,288, 700,000, 1,048,576, 1,200,000, 1,600,000, 2,097,152, 2,500,000 and a set of heights H = 5000, 10,000, 15,000, 20,000, 25,000, 30,000 were chosen, to produce a set of width-height pairs $T = W \times H$. For each pair, a matrix slice was constructed and sent over the stream ten times, simulating a stream of ten equally sized matrix slices being processed and, consequently, triggering ten executions of the State Update kernel for each test. The matrix slices consisted of all true truth values; however, neither the composition of each slice nor the repetition of the same slice in the stream impacts kernel execution time.

Figure 9 presents a normalized surface plot of the average processing time per slice, in relation to its width and height. For scale, the highest execution time recorded was 109.2 s, for the width-height pair (30000, 2500000), while the lowest was 0.016 s for the input pair (1000, 5000). From the results shown, we observe that the height of the input slices has the greatest impact on time, which is in line with expectation due to the quadratic increase in row pairs. This can be more easily seen in Figure 10, where the time samples collected for different heights with a fixed width of 15,000 are plotted.

The change in width has a less notable impact on execution time as it primarily affects the residency duration of blocks. Changes in width lead to a linear increase in time as expected, which can be seen in Figure 11. Notably, the time difference observed when varying the width of the slice while keeping the height fixed is amplified by the number of row pairs being processed, which is in turn directly related to the height. This can be observed in Figure 9 in the form of a step between bars in the same height group. The step is steeper for larger heights than for the smaller ones.

While the utilization of compute and memory resources by the State Update kernel depends on the input, with smaller inputs

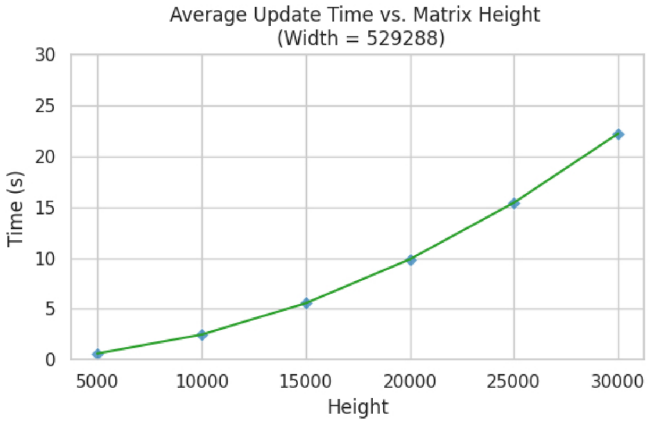

FIGURE 10 | State Update kernel time versus height when width is fixed to 529, 288.

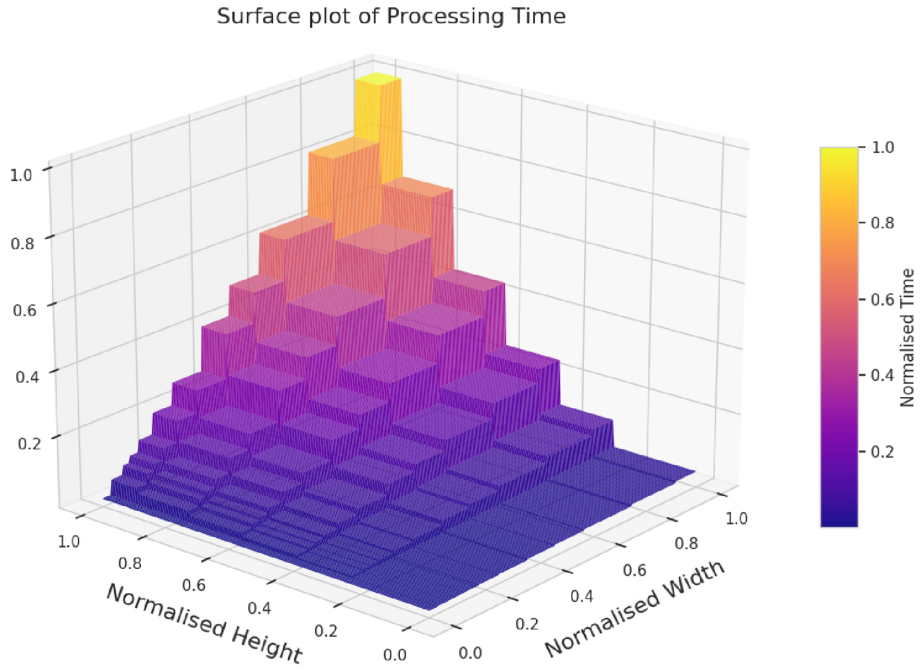

FIGURE 9 | Surface plot of normalized kernel execution times relative to the width and height of the slices in the stream.

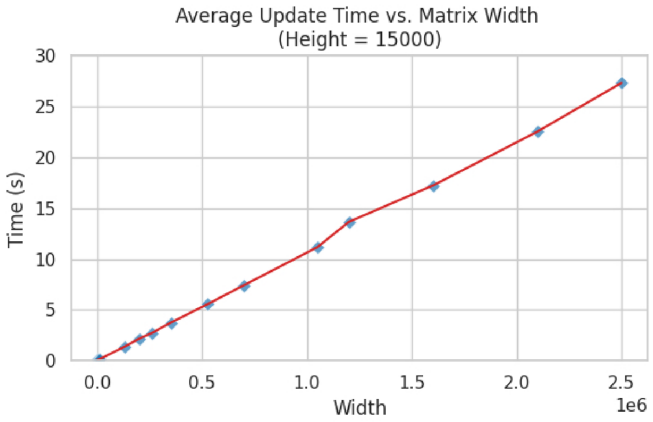

FIGURE 11 | State Update kernel time versus width when height is fixed to 15, 000.

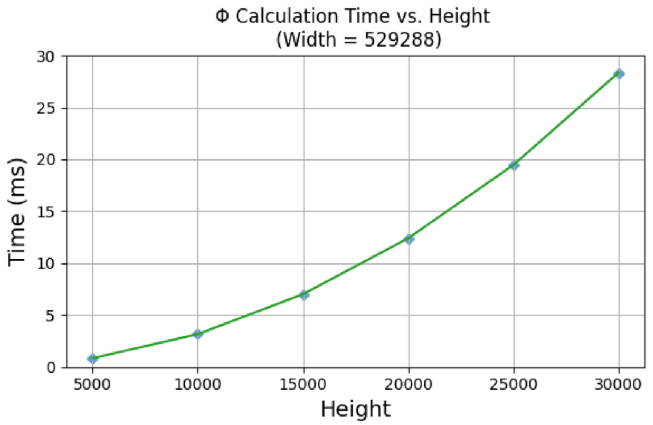

FIGURE 12 | Time needed for the computation of ϕ-coefficients and post-processing relative to state matrix height when width is fixed to 529, 288.

benefiting greatly from caching, the overall utilization of the two metrics remains balanced without notable spikes in execution time observed. The kernel's execution time offers further insight into the average per-row-pair processing time. In the case of the largest input, 449,985,000 row pairs were processed, of which each row comprised 312,500 bytes of data, in an average of ≈ 0.0242 ms per-row-pair. We speculate that the use of multiple GPUs in the manner described in Section 7.1.1 will, beyond increasing the maximum slice height that can be processed, reduce the per-row-pair processing time further.

Besides the State Update kernel, the ϕ-coefficient Computation and Post-Processing kernels were also profiled. Since both kernels are dispatched in succession, they were profiled as a unit. The two kernels form an insignificant portion of the computation and required between 1.12 and 28.41 ms to complete. As is to be expected, the total computation time of these kernels is quadratic w.r.t. the height of the slices that have been processed, as can be seen in Figure 12. The width of slices previously consumed does not impact the runtime of these kernels.

Overall, the results presented highlight the power of this approach in computing large numbers of correlation coefficients in a timely manner. Since our earlier work [16] required no device in-memory state to be updated and instead computed ϕ-coefficients following the summarization of each row pair, direct comparison to this work is difficult to perform accurately. Nevertheless, the runtime performance of this approach is equivalent if not better than our earlier work, and comes with no limitation[5] on the width of the matrix being processed. Moreover, the decoupling of ϕ-coefficient calculations and row-pair summarization results in the additional benefit of non-divergent execution of each kernel and thus better utilization of available resources.

To contextualize the time measurements presented here, we refer to the results of our earlier work [16], which include time measurements of our earlier approach on the CPU, obtained from a single-threaded implementation and a multi-threaded one that splits work evenly between 32 threads. While this and our previous approach are not directly comparable, their respective CPU-side implementations are. We elected not to simulate the streaming approach on the CPU, as this would incur further overheads, artificially slowing the CPU-side implementation further.

For a state matrix of (width, height) of (1000000, 6452), the single-threaded CPU implementation took 798.91 s (resp. 51.6 for the multi-threaded version). Processing a slice of equal size with streaming implementation, and measuring the time as described earlier on an A100, takes 1.993 s, while computing the ϕ-coefficients at the end of the stream takes 0.00136 s, giving a combined time of 1.99436 s. This yields a speedup of $\approx 400\times$ relative to the single-threaded (resp. $\approx 26\times$ to the multi-threaded) implementation. We observe similar speed-ups compared to the (2000000, 6452) input, where the single- and multi-threaded implementations took 1519.56 and 103.79 s, respectively, while the streaming approach took 3.98908 s to process the slice and 0.00140 s to compute ϕ-coefficients, totaling 3.99047 s. In turn, this shows a speedup of $\approx 381\times$ and $\approx 26\times$ relative to the single- and multi-threaded CPU implementations, respectively.

We highlight that an objective CPU-GPU comparison would require an intricate multi-modal analysis of implementations and the architecture of each device, which stretches beyond the scope of this work. We provide this comparison as a crude point of reference for the reader's understanding.

6.4 | Reflections on Scalability

The GPU-based ϕ-coefficient computation assistant described in previous sections is designed as an independent system that operates in tandem with the presented RL framework. Communication between the two is established via a network connection that permits the two systems to execute on different machines. Our main motivation behind this design is the ability to execute the GPU assistant in a GPU-compute environment, whilst simultaneously running the memory- and CPU-intensive RL exploration in a different environment tailored to their needs. This decentralized design permits effortless horizontal scaling of the GPU assistant, as multiple independent instances may execute in different environments while being controlled by the RL framework. Given that a single GPU can, in theory, consume an arbitrary number of windows extracted from a single embedding cluster,

scaling horizontally to a multi-instance framework means separate clusters can be streamed in parallel.

Vertical scaling of this approach would permit processing larger slices, importantly in relation to their height. Due to the nature of the application, and particularly for the State Update kernel, this type of scaling presents some difficulties. Rows of an incoming state matrix slice are each paired and processed with every subsequent row, and therefore, partitioning into smaller sub-partitions requires duplication of some of the data. The increase in supported slice dimensions, particularly in terms of height, achieved by using the memory capacity of multiple devices, is a benefit to this application. We are actively exploring an approach to achieve this, described in Section 7.1.1.

7 | Conclusions and Future Work

This paper has presented a scalable, learning-driven approach to candidate invariant inference for Ladder Logic programs, motivated by the challenges posed by inductive verification over large, safety-critical industrial systems. We began by outlining the formal verification problem and the role of invariants in reducing the state space during induction-based model checking. Reinforcement learning was introduced as a viable strategy for exploring Ladder Logic program state spaces, in particular, enabling parallel exploration and policy training in large state spaces.

Building on prior work, we extended our invariant mining framework with new heuristics to increase the expressiveness of candidate invariants while addressing the computational cost of ϕ-coefficient analysis. To mitigate this cost, we introduced a clustering pipeline grounded on dimensionality reduction, allowing structured subsets of variables to be processed efficiently. The final component of our system implemented a GPU-based correlation engine that streams expanded observation windows to device memory in a scalable and incremental manner. In addition, we revised our previous approach to support distributed work, enabling each key component of the invariant mining framework to communicate with the previous and ensuing segments of the pipeline. This resulted in an efficient and incremental approach to exploring Boolean-based state spaces and synthesizing invariants to assist with automated theorem proving.

7.1 | Future Directions

In addition to the contributions presented in previous sections, we continue to improve the scalability and robustness of our invariant mining framework in terms of state space exploration, learned variable embeddings, and the number of row-pairs considered during ϕ-coefficient computation. In this section, we briefly outline the future directions of this work.

7.1.1 | Multi-GPU Streaming

In earlier sections, we describe the design of the GPU ϕ-coefficient calculator, which is currently focused on using only one GPU per instance of the supervisory process. The State Update kernel currently constrains us, as its threads explore all unique row pairings and hence require the full slice to be available on device memory. As we expand this component of the RL framework, we intend to explore the feasibility of using four or six GPUs for processing matrix slices, whilst keeping all aggregated state resident on the memory of those devices.

To achieve multi-GPU utilization when processing a state matrix slice $S = (r_0, \ldots, r_{k-1})$ comprising k rows $r_0, \ldots, r_{k-1}$, we begin by constructing six partitions, $P_{1..6}$ of S, as per Equation (15). The computation currently performed by the State Update kernel on a single GPU could be performed across four or six GPUs using these partitions, in such a way that no GPU ever has more than $\frac{k}{2}$ rows resident in its memory.

$$
\begin{aligned}
m &= \frac{k}{2} & q &= \frac{k}{4} \\
P_1 &= (r_0, \ldots, r_{m-1}) & P_2 &= (r_m, \ldots, r_{k-1}) \\
P_3 &= (r_0, \ldots, r_{q-1}) & P_4 &= (r_q, \ldots, r_{m-1}) \\
P_5 &= (r_m, \ldots, r_{m+q-1}) & P_6 &= (r_{m+q}, \ldots, r_{k-1})
\end{aligned} \tag{15}
$$

More specifically, in the case of four GPUs, a kernel performing the procedure described in Section 6.3.3 would be launched on GPUs 1 and 2, which operate on partitions P_1 and P_2, respectively. GPUs 3 and 4 would initially be supplied with the pairs of partitions (P_3, P_5) and (P_4, P_5) respectively and perform an exhaustive summarization operation between all rows of one partition and the other, in the same manner as a Cartesian product. Subsequently, GPUs 3 and 4 would be supplied with the pairs of partitions (P_3, P_6) and (P_4, P_6), respectively, and repeat this operation. The pairings of rows each GPU makes in each aforementioned step can be visualized using the small example in Figure 13. Where six GPUs are used, the process remains identical except for the final step, which would instead be performed by GPUs 5 and 6. Specifically, GPUs 5 and 6 are tasked with performing the exhaustive summarization of (P_3, P_6) and (P_4, P_6), respectively.

Part of our focus in this investigation is to establish the circumstances under which a six-GPU approach offers a benefit in computation time over four GPUs. This is because, despite permitting all operations to be performed in parallel, a six-GPU approach requires further duplication of data from the host, since partitions P_3 and P_4 must be transferred to GPUs 3, 5, and 4, 6, respectively.

The remaining kernels are mostly unaffected by this change since all bit counts accumulated from successive executions of the State Update kernel would reference specific input rows wholly to permit the computation of ϕ-coefficients and, subsequently, the direct mapping of those ϕ-coefficient values to their respective row index pairs.

7.1.2 | Random State Discovery and Reinforcement Learning

In the reinforcement learning process, the framework could be enhanced by adopting a hybrid exploration strategy that dynamically switches between random action selection and policy-driven learning. Specifically, the agents would begin with a purely random policy, maximizing the rate of state discovery by avoiding the initial bias introduced by early-stage policy

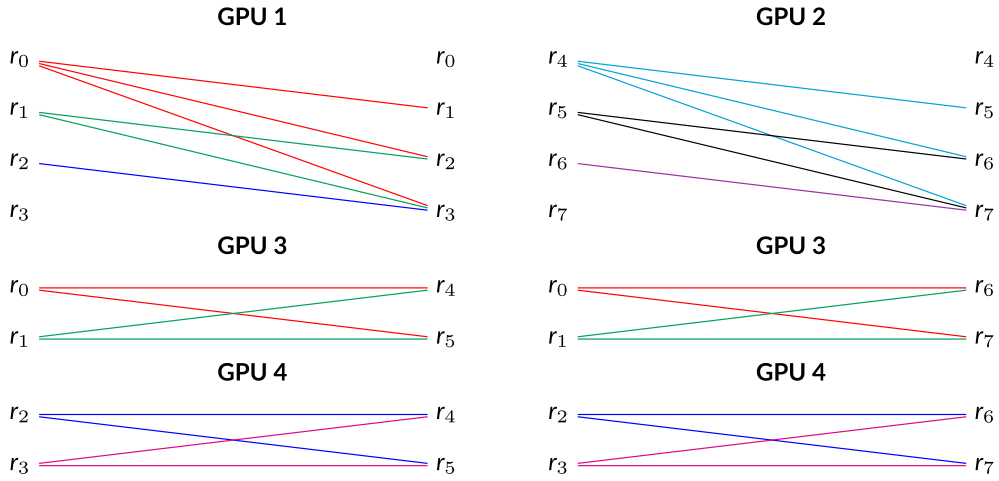

FIGURE 13 | Pairings of rows in each of four GPUs when consuming a matrix slice comprising $k = 8$ rows. Here, the partitions $P_1 = (r_0, \ldots, r_3)$, $P_2 = (r_4, \ldots, r_7)$, $P_3 = (r_0, r_1)$, $P_4 = (r_2, r_3)$, $P_5 = (r_4, r_5)$, and $P_6 = (r_6, r_7)$ are used. Each line color illustrates a fixed left-hand side for all unique pairings with the remaining right-hand side rows.

gradients and their computational overheads. During this phase, we monitor a moving window over the number of unique states discovered per a fixed number of environment steps. Once this discovery rate falls below a threshold, indicating diminishing returns from uninformed exploration, the system transitions to a policy optimization phase using PPO. This transition can be formalized by introducing a convergence criterion on the novelty function N_t, defined as the number of unique states discovered in the past t steps. When $\nabla N_t \approx 0$ over several intervals, policy training is enabled, and the reward function is reshaped to prioritize exploration in harder-to-reach regions of the state space. This segmentation allows the agent to exploit gradient-based optimization where it is most effective—namely, in structured regions where random exploration fails to reach informative states, and where exploration-exploitation trade-offs become more nuanced. Such a staged control mechanism could be implemented as a meta-controller or scheduler embedded in the training loop, selecting between exploration strategies based on online statistics of novelty and episode return.

7.1.3 | State Embeddings From Variational Autoencoders

In the embedding and clustering pipeline, replacing non-parametric methods such as t-SNE and UMAP with a parametric, generative deep learning model, specifically a variational autoencoder (VAE), offers a principled way to address scalability and structural fidelity in state space embeddings. Unlike t-SNE and UMAP, VAEs learn a continuous, low-dimensional latent representation through optimization of a reconstruction loss, forcing the model to preserve global patterns. This enables the model to generalize across unseen data and embed new states without retraining the entire mapping.

This property is particularly advantageous in reinforcement learning settings, where the number of observed environment states can grow into the millions. A trained VAE can continuously embed incoming state observations into a fixed-dimensional latent space, allowing clustering and statistical inference to be performed on a compressed and semantically meaningful representation. Crucially, VAEs preserve both global and local structure in the data manifold, mitigating the spatial distortions introduced by t-SNE and UMAP, which prioritize local neighborhood preservation at the expense of inter-cluster geometry.

As a result, variable groupings produced by clustering in the VAE latent space are expected to be more faithful to the underlying distribution of the state space. This improves the accuracy of downstream correlation analysis by aligning cluster boundaries with actual structural dependencies between variables. Moreover, because VAEs provide a generative model of the state space, latent representations may allow us to predict underexplored regions and synthesize representative inputs for invariant mining.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Endnotes

[1] With respect to incoming data, typically the size of a matrix row multiplied by a constant factor.

[2] Specifically considering the machine instructions produced for this PTX code in context.

[3] Where 128-byte transaction windows are possible, a single transaction will be sufficient to fulfil the request of all threads in the warp.

[4] Using NVCC 12.5 with flags `-code sm_80-arch compute_80` set.

[5] Beyond 64-bit type limitations imposed primarily by the State Update kernel's counters.

References

1. J. Groote, vS. Vlijmen, and J. Koorn, *The Safety Guaranteeing System at Station Hoorn-Kersenboogerd* (IEEE, 1995), 57–68.

2. A. Haxthausen, M. Bliguet, and A. Kjær, *Modelling and Verification of Relay Interlocking Systems* (Springer, 2008), 141–153.

3. K. Kanso, F. Moller, and A. Setzer, "Automated Verification of Signalling Principles in Railway Interlocking Systems," *Electronic Notes in Theoretical Computer Science* 250, no. 2 (2009): 19–31.

4. P. James and M. Roggenbach, "Automatically Verifying Railway Interlockings Using SAT-Based Model Checking," *Electronic Communications of the EASST* 35 (2011): 1–7.

5. A. Cimatti and A. Griggio, "Software Model Checking via IC3," in *Computer Aided Verification*, ed. P. Madhusudan and S. A. Seshia (Springer, 2012), 277–293.

6. P. James, A. Lawrence, and F. Moller, *Verification of Solid State Interlocking Programs* (Springer, 2013), 253–268.

7. A. Haxthausen, J. Peleska, and R. Pinger, *Applied Bounded Model Checking for Interlocking System Designs* (Springer, 2013), 205–220.

8. P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Techniques for Modelling and Verifying Railway Interlockings," *International Journal on Software Tools for Technology Transfer* 16, no. 6 (2014): 685–711.

9. C. Limbrée, Q. Cappart, C. Pecheur, and S. Tonetta, *Verification of Railway Interlocking-Compositional Approach With OCRA* (Springer, 2016), 134–149.

10. A. Fantechi, A. Haxthausen, and H. Macedo, *Compositional Verification of Interlocking Systems for Large Stations* (Springer, 2017), 236–252.

11. R. Cavada, A. Cimatti, S. Mover, M. Sessa, G. Cadavero, and G. Scaglione, *Analysis of Relay Interlocking Systems via SMT-Based Model Checking of Switched Multi-Domain Kirchhoff Networks* (IEEE, 2018), 1–9.

12. A. Amendola, A. Becchi, R. Cavada, et al., *NORMA: A Tool for the Analysis of Relay-Based Railway Interlocking Systems* (Springer, 2022), 125–142.

13. G. Cabodi, S. Nocco, and S. Quer, "Strengthening Model Checking Techniques With Inductive Invariants," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, no. 1 (2009): 154–158, https://doi.org/10.1109/TCAD.2008.2009147.

14. B. Lloyd-Roberts, P. James, and M. Edwards, *Mining Invariants From State Space Observations* (NWPT, 2022).

15. B. Lloyd-Roberts, P. James, M. Edwards, S. Robinson, and T. Werner, *Improving Railway Safety: Human-In-The-Loop Invariant Finding* (ACM, 2023), 1–8.

16. B. L. Roberts, F. Pantekis, P. James, L. O'Reilly, and M. Edwards, *Effective Candidate Invariant Generation Using GPGPUs and Optimisations* (IEEE, 2024), 77–86.

17. M. Tiegelkamp and K. H. John, *IEC 61131–3: Programming Industrial Automation Systems* (Springer, 2010).

18. E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design* 19 (2001): 7–34.

19. P. Bjesse and K. Claessen, *SAT-Based Verification Without State Space Traversal* (Springer, 2000), 409–426.

20. S. Bensalem, Y. Lakhnech, and H. Saidi, *Powerful Techniques for the Automatic Generation of Invariants* (Springer, 1996), 323–335.

21. A. R. Bradley, "SAT-Based Model Checking Without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, ed. R. Jhala and D. Schmidt (Springer, 2011), 70–87.

22. C. Savage, "Depth-First Search and the Vertex Cover Problem," *Information Processing Letters* 14, no. 5 (1982): 233–235, https://doi.org/10.1016/0020-0190(82)90022-9.

23. R. Sutton and A. Barto, *Reinforcement Learning: An Introduction* (MIT press, 2018).

24. M. Puterman, *Markov Decision Processes*, vol. 2 (Handbooks in operations research and management science, 1990), 331–434.

25. D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, *Deterministic Policy Gradient Algorithms* (PMLR, 2014), 387–395.

26. Y. Hu, W. Wang, H. Jia, et al., *Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping*, vol. 33 (Advances in Neural Information Processing Systems, 2020), 15931–15941.

27. F. Melo, S. Meyn, and I. Ribeiro, *An Analysis of Reinforcement Learning With Function Approximation* (ACM, 2008), 664–671.

28. V. Mnih, K. Kavukcuoglu, D. Silver, et al., "Playing Atari With Deep Reinforcement Learning," (2013), https://arxiv.org/abs/1312.5602.

29. S. Amari, "Backpropagation and Stochastic Gradient Descent Method," *Neurocomputing* 5, no. 4–5 (1993): 185–196.

30. V. Mnih, A. P. Badia, M. Mirza, et al., "Asynchronous Methods for Deep Reinforcement Learning," 2016 arXiv preprint arXiv:1602.01783.

31. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," (2017), https://arxiv.org/abs/1707.06347.

32. J. Ekström, "The Phi-Coefficient, the Tetrachoric Correlation Coefficient, and the Pearson-Yule Debate," *UCLA Department of Statistics* (2011), https://escholarship.org/uc/item/7qp4604r.

33. H. Tang, R. Houthooft, D. Foote, et al., "Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning," *Advances in Neural Information Processing Systems* 30 (2017).

34. C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén, "Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents," (2021), https://arxiv.org/abs/2103.13798.

35. S. Kwek, *On a Simple Depth-First Search Strategy for Exploring Unknown Graphs*, (Springer, 1997), 345–353.

36. H. Zhang and T. Yu, "AlphaZero," in *Deep Reinforcement Learning: Fundamentals, Research and Applications* (Springer, 2020), 391–415.

37. C. A. Cheng, A. Kolobov, and A. Swaminathan, "Heuristic-Guided Reinforcement Learning," *Advances in Neural Information Processing Systems* 34 (2021): 13550–13563.

38. A. Aubret, L. Matignon, and S. Hassas, "A Survey on Intrinsic Motivation in Reinforcement Learning," (2019), https://arxiv.org/abs/1908.06976.

39. M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, *Unifying Count-Based Exploration and Intrinsic Motivation* (ACM. Curran Associates Inc., 2016), 1479–1487.

40. J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," (2017), https://arxiv.org/abs/1502.05477.

41. A. V. Clemente, H. N. Castejòn, and A. Chandra, "Efficient Parallel Methods for Deep Reinforcement Learning," (2017), https://arxiv.org/abs/1705.04862.

42. D. Horgan, J. Quan, D. Budden, et al., "Distributed Prioritized Experience Replay," (2018), https://arxiv.org/abs/1803.00933.

43. B. Everitt, *The Analysis of Contingency Tables* (CRC Press, 1992).

44. V. d. L. Maaten and G. Hinton, "Visualizing Data Using t-SNE," *Journal of Machine Learning Research* 9, no. 11 (2008): 2579–2605.

45. D. Müllner, "Modern Hierarchical, Agglomerative Clustering Algorithms," (2011), https://arxiv.org/abs/1109.2378.

46. J. Peddie, *Introduction* (Springer International Publishing, 2022).

47. R. Ansorge, *Programming in Parallel With CUDA: A Practical Guide* (Cambridge University Press, 2022).

48. S. An and S. Seo, "Highly Efficient Implementation of Block Ciphers on Graphic Processing Units for Massively Large Data," *Applied Sciences* 10 (2020): 3711.

49. F. Pantekis, P. James, O. Kullmann, and L. O'Reilly, "Optimised Massively Parallel Solving of N-Queens on GPGPUs," *Concurrency and Computation: Practice and Experience* 36, no. 10 (2023): 1–19, https://doi.org/10.1002/cpe.8004.

50. M. Osama, A. Wijs, and A. Biere, "Certified SAT Solving With GPU Accelerated Inprocessing," *Formal Methods in System Design* 62 (2023): 79–118, https://doi.org/10.1007/s10703-023-00432-z.

51. B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, *Maximal Clique Enumeration With Data-Parallel Primitives* (IEEE, 2017), 16–25.

52. P. Zhang, E. Holk, J. Matty, et al., "Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms," in *IA^3'15. IEEE Computer Society* (Association for Computing Machinery, 2015).

53. NVIDIA Corporation, "NVIDIA Ampere GA102 GPU Architecture. Whitepaper, NVIDIA; 2788 San Tomas Expressway Santa Clara, CA. 95050," 2020.

54. NVIDIA Corporation, "Parallel Thread Execution ISA Version 8.8," (2025), https://web.archive.org/web/20250510202712/https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

55. NVIDIA Corporation, "CUDA Binary Utilities 12.5," (2025), https://web.archive.org/web/20250415174228/https://docs.nvidia.com/cuda/cuda-binary-utilities/.

56. A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, *Decoding CUDA Binary* (IEEE, 2019), 229–241.

57. H. Abdelkhalik, Y. Arafa, N. Santhi, and A. H. A. Badawy, *Demystifying the Nvidia Ampere Architecture Through Microbenchmarking and Instruction-Level Analysis* (IEEE, 2022), 1–8.

58. J. Zhe, M. Marco, S. Benjamin, and P. S. Daniele, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," (2018), https://arxiv.org/abs/1804.06826.

59. Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking," (2019), https://arxiv.org/abs/1903.07486.

60. I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, *Dissecting the CUDA Scheduling Hierarchy: a Performance and Predictability Perspective* (IEEE, 2020), 213–225.

61. NVIDIA Corporation, "CUDA C++ Programming Guide Release 12.9," (2025), https://web.archive.org/web/20250505133403/https://docs.nvidia.com/cuda/cuda-c-programming-guide/.