Effective Candidate Invariant Generation Using GPGPUs and Optimisations

Ben Lloyd-Roberts*, Filippos Pantekis†, Phillip James‡, Liam O'Reilly§ and Michael Edwards¶
Department of Computer Science, Swansea University
Swansea, Wales, United Kingdom
Email: *ben.lloyd-roberts@swansea.ac.uk, †filippos.pantekis@swansea.ac.uk,
‡p.d.james@swansea.ac.uk, §l.p.oreilly@swansea.ac.uk, ¶michael.edwards@swansea.ac.uk

***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 inclusion of unreachable states where safety conditions are violated by 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, remove these false positives. However, such invariants are difficult to deduce automatically. This paper advances existing work on using reinforcement learning (RL) and correlation measures to generate candidate invariants. This entails mapping programs to an RL environment, incentivising agents to explore state spaces and analysing observations for invariant patterns. We observe certain complications surrounding the computation of these correlation coefficients when processing large industrial programs. This paper presents our approach using General Purpose Graphics Processing Units (GPGPUs) to overcome these challenges. We detail the steps taken to optimise our GPU kernel and present results on tested devices and inputs. We achieve runtime performance orders of magnitude higher than multi- and single-threaded CPU-side implementations, with speedups ranging from $\approx 21\times$ to $\approx 318\times$ when generating correlation coefficients for producing candidate invariants.**

I. Introduction

Formal verification of railway control systems is a field with substantial academic attention [1]–[4]. Contemporary research has focused on developing techniques that verify the safety of large interlocking systems controlling railways and their signalling systems. Successful approaches for scaling have included abstraction techniques [5], decomposition techniques [6] and induction-based model checking [7], [8]. It is often the case that approaches involving inductive verification, providing good performance when successful [7], [8], can exhibit so-called false positive counterexamples. A transition from a safe state to an unsafe state is a safety violation and highlighted as a counterexample. However, when the source state is unreachable this becomes a false positive counterexample [9]. Invariants are properties that reduce the state space to an over-approximation of reachable states that exclude such problematic transitions. However, automatically deducing invariants for a given state space remains non-trivial and computationally expensive [10].

Ben Lloyd-Roberts is funded by the EPSRC Centre for Doctoral Training in Enhancing Human Interactions and Collaborations with Data and Intelligence Driven Systems (EP/S021892/1) and Siemens Mobility UK.

Previously, it has been shown that it is possible to use reinforcement learning (RL) and simple measures of correlation to generate candidate invariants[1] for reducing state spaces when performing inductive verification [11], [12]. The control systems under verification, composed of arbitrary Ladder Logic programs, undergo a first formal mapping of program-based state spaces to a reinforcement learning environment. Thereafter, reinforcement learning agents can be incentivised to explore large regions of state spaces, before using agent observations to analyse correlations across states which emerge throughout the exploration. Although reinforcement learning drives state space exploration, its evaluation is beyond the scope of this work. In this paper, we summarise this approach via a simple example and introduce the challenges encountered at scale, namely the computation of correlation coefficients. We present our approach to overcome such challenges using General Purpose Graphics Processing Units (GPGPUs). We do so by mapping the correlation coefficient calculation step in a manner favourable to GPGPU computation and establishing a range of optimisations to the kernel to maximise throughput.

II. Formal Verification and Invariants

The failure of critical systems can result in catastrophic consequences. It is imperative these systems undergo rigorous efforts, before their deployment, to ensure they adhere to strict safety criteria. Conventionally, verification processes involve modelling the different states a system can assume during its operation and systematically checking if a formally encoded safety property is respected by all reachable system states.

A. Modelling Ladder Logic

Many industrial critical systems, PLCs [13] 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 1 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

[1]These properties are approximated based on an observed subset of reachable states. They must be checked using a proof system before being considered 'true' invariants.

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 modelled, and used to derive useful invariants to help inductive verification.

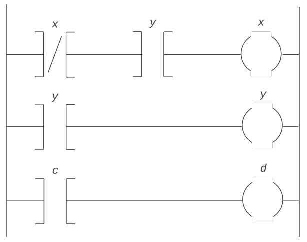

Fig. 1. Rudimentary Ladder Logic program for controlling P_1.

Both Kanso et al., [9] and James et al., [5], [7], [14], demonstrated the systematic translation of Ladder Logic to propositional formulae, representing Ladder Logic program verification as SAT problems. Figure 2 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 Labelled Transition System (LTS) composed of a set of states and connecting transitions. The former describes unique configurations a system may assume during its operation, while the latter describes 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 verification of safety.

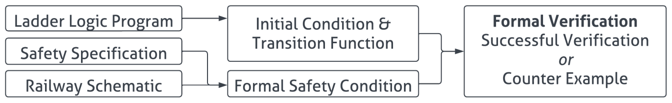

Fig. 2. Pipeline for formally verifying interlockings against a set of safety criteria.

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 [14], we model ψ_{P_1} as a Ladder Logic Labelled 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 3 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 3 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 II-B.

B. Invariant Finding

James et al., [14] 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 [15]. 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 [16]. However, this abstraction may introduce states which 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 [10], 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 domains [17], [18]. However, for complex programs, this becomes computationally expensive.

Given invariants are properties that persist across system states, one reasonable approach is to observe as many unique reachable states as possible to analyse 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

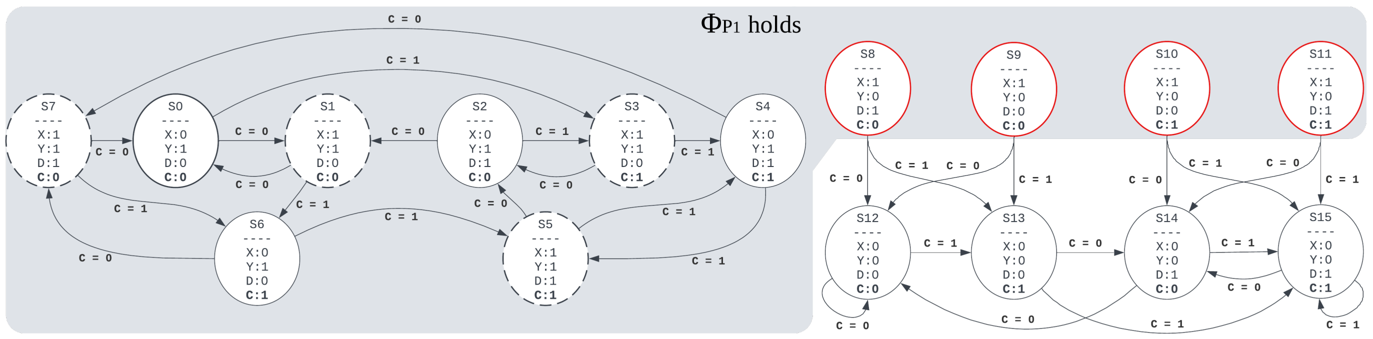

Fig. 3. Depiction of the Ladder Logic Labelled Transition System $LTS(\psi_{P_1})$. The grey region indicates the states where the safety property Φ_{P_1} holds. Nodes with a dashed outline represent the initial states from Val_0. The red highlighted states are problematic for induction-based verification as they are unreachable safe states that transition to unsafe states.

confronted with the combinatorial explosion of large state spaces [19]. For example, a particular interlocking program we consider, denoted P_2, consists of 6452 variables, 2222 of which are program inputs. This produces 2^{2222} potential transitions from any one of 2^{6452} possible states. Thus we turn to probabilistic, gradient-based methods that are optimised to approximate a function prioritising state discovery.

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 3 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, $s_0, ..., s_{11}$, 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 $s_{12}, ..., s_{15}$, 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.

III. Reinforcement Learning and Exploration

Reinforcement learning (RL) is a machine learning paradigm designed to optimise 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. In the following section, we introduce the general RL framework, and how its components are adapted for invariant finding.

A. Background

Formally, an RL environment is represented as a Markov Decision Process (MDP) [20]. A finite discounted MDP M is a five-tuple $(S, A, P_a(s, s'), R_a(s, s'), \gamma)$, where, 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 $P_a(s, s') = P(s_{t+1} = s'|s_t, a_t)$ describes 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 [21] and deterministic systems [22], 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 [23]. We discuss our approach to reward shaping for invariant finding in Section III-B. Finally, $\gamma \in [0, 1)$ is a discount scalar applied to rewards considered over future time steps. Experiences are accumulated according to one of two simulation paradigms: episodic or continuous. The distinction between both 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 summarises its experience accumulated over a finite number of time steps, denoted $\tau = (s_1, a_1, r_1, s_2, a_2, r_2, ..., s_h, a_h, r_h)$. 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 entails approximating a value function which 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 optimising a behaviour policy, $\pi(a|s)$, mapping states to the actions most likely to maximise cumulative rewards. Optimal value functions and optimal policies are reciprocal in that converging to one will converge to the other [24]. Optimisation 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 [25] and gradient-based optimisation [26]. So-called actor-critic [27] 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 Optimisation (PPO) [28].

B. Ladder Logic Exploration

Representing the invariant finding problem for inductive verification 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 [11] provide the foundation for this work. Given $LTS(\psi_{P_1})$, the observations space of the corresponding Ladder Logic MDP, $M(\psi_{P_1})$, has the form $S = \text{Val}_C \cup \text{Val}_I$. 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 $A = \text{Val}_I$ consists of program inputs given 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 principle difference in this context is that states are computed explicitly, ensuring state observations do not include the problematic states described in Section II-B. Such a framework allows us to progressively optimise state space coverage, record unique state observations from agent trajectories, and suggest candidate invariants using simple measures of binary correlation [29]. Aiming to maximise state space coverage we implement a reward scheme which 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 [30] and initial state sampling using the least-visited states from past experience [31].

Given our learning objective, where the number of state-action pairs is minimal, such as $M(\psi_{P_1})$, dynamic graph traversal [32] makes parametric deep reinforcement learning (DRL) redundant. However, when modelling a large Ladder Logic MDP, such as $M(\psi_{P_2})$, one must contend with the combinatorial explosion of state-action pairs. Algorithms relying on exhaustive search may then lack the heuristics or flexibility to prioritise state novelty. We hypothesise using approximate learning techniques to navigate large state spaces may devise better search heuristics [33]. Invariants suggested throughout exploration can then be validated via inductive checks. To that end, both rates of state discovery and time to compute invariants should be as performant as possible.

IV. Invariant Proposal

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 goal of constructing new candidate invariants guide the determination of these associations.

A. 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, minimising errors in approximation and computational overhead.

Covariance indicates the direction of a linear relationship between two random variables but does not normalise the magnitude of this association. In contrast, ϕ-correlation coefficients, an adaptation of Pearson's correlation for dichotomous data [29], are tailored for binary pairs, making them favourable 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.

B. ϕ-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:

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

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 [34]. 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 normalised, 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 presents us with a systematic means of inferring invariance between arbitrary 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. Table I 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.

Consider the complete inverse relation between $\neg x$ and $x \vee \neg y$ in Table I. 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 $s_8 \dots s_{15}$ when verifying the safety property Φ_{P_1} we introduced in Section II-B.

	$\neg x$	$x \wedge y$	$x \vee \neg y$	$y \wedge \neg x$	$\neg x \vee \neg y$
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

TABLE I
PARTIAL CORRELATION MATRIX COMPUTED FROM $M(\psi_{P_1})$. DUPLICATE COEFFICIENTS AND COEFFICIENTS DESCRIBING SELF-CORRELATION HAVE BEEN SET TO 0.

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 and validated by an inductive check 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 an 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 increase. Thus, we turn to specialist hardware and optimisation to scale this approach.

In the next section, we introduce how this problem can be represented and addressed using General Purpose Graphics Processing Units (GPGPUs). To illustrate how one can practically compute millions of ϕ-coefficients given large RL environments, we use an industrial Ladder Logic program, P_2, implementing a Siemens Mobility interlocking. Throughout a single training run, our RL agent produced four matrices of observation data, containing 1, 2, 10 and 30 million unique rows, or states, respectively. Each matrix has 6452 columns, representing the variables composing each state.

V. GPU ACCELERATION

Graphics Processing Units (GPUs), as we know them today, generally feature greater arithmetic performance than their counterparts (CPUs) primarily due to the computational model they employ and their special nature of hardware [35]. This is no surprise considering the nature of graphics tasks these devices are tailored for. Recently, the computational capabilities of GPUs have been exposed through programming interfaces that broaden their area of application beyond the graphics domain, in what is known as a General Purpose GPU (GPGPU). In this work, the two terms are used interchangeably.

GPUs have successfully been used to accelerate a broad range of tasks including medical imaging [36], machine learning [37], cryptography [38] and, more broadly, hard combinatorial problems [39]–[42]. This work emphasises the need for careful planning and optimisation in the development of performant and efficient GPU applications, further highlighting the intricacy of development on these devices.

A. Background

Our work is based on NVIDIA GPUs as they offer comprehensive tooling for development and optimisation. Before exploring our implementation choices, we must first lay some fundamentals on the principle of operation, hardware, and programming considerations of these devices.

GPUs are massively parallel coprocessors, the strength of which stems from thousands of simultaneously executing threads. Threads on the GPU are significantly lighter incarnations than what is commonly found on the CPU-side. GPU threads are not intended to act as independent units and are grouped into blocks. Blocks are logical structures of threads which can be mono-, bi-, or tri-dimensional and can comprise up to 1024 threads at present. 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 than blocks. The geometry of the grid and block structures is used to differentiate computation, as each thread has knowledge of its coordinates in its respective block, as well as the coordinates of the block in the grid. For tasks exhibiting spatial locality, this geometry may offer a better mapping to the input data, yet this geometry presents no inherent performance penalty for applications that do not utilise it. In these cases, a simple mono-dimensional grid of mono-dimensional blocks is typically used to simplify index calculations.

GPUs are manycore processors and like others, classify as Single Instruction Multiple Data (SIMD) [43] vector processors. NVIDIA places their GPUs in a variant of SIMD, called SIMT where T stands for Threads. The difference is subtle but important as, unlike SIMD processors, threads in the GPU can, in principle, execute independently, even within their blocks. In practical terms, the physical GPU comprises a number of Streaming Multiprocessors (SMs), each of which can have a number of resident blocks. Blocks remain resident in their respective SMs throughout their lifetime, which ends when all their threads have exited. For the purposes of execution, each block is partitioned into groups of 32 adjacent threads, called warps. Hardware warp schedulers exist in each SM which are responsible for selecting one of the warps assigned to them and issuing one of its instructions. Threads in a warp cannot execute independently and must instead execute the same instruction in lockstep. When threads in the same warp follow divergent execution paths, the throughput of the warp reduces, leading to a deterioration in performance. As such, it is crucial to keep warp divergence to a minimum and where unavoidable, facilitate re-convergence as early as possible.

Several memory regions are available on the GPU which reside either on- or off-chip. Generally, off-chip memory regions are larger but incur significant costs to interact with, relative to on-chip regions which are size-restricted and cheaper. Global memory is an off-chip memory type accessible by all threads in the grid. Global memory is the largest memory region and can be manipulated by the host system for data exchange between it and the device. Due to the high access costs, patterns of access exhibited by threads in their respective warp can have profound performance implications. Global memory transactions pass through both L2 and L1 caches. The former is an on-chip cache accessible by all SMs whereas the latter is local to each SM. Each SM houses a block of fast memory, typically a few kilobytes in size, which is split between the L1 cache and shared memory. The shared memory region can be manipulated by threads in the same block with changes visible to all, yet each resident block has access to different, non-overlapping shared memory regions.

Communication between threads beyond their enclosing block is discouraged as it results in strong coupling and incurs high memory latency costs. Interblock, and more so interwarp communication, is cheap and to a degree expected. Threads within a warp can exchange data between them, without using shared memory, by exchanging register contents. Such primitives are cheaper than shared memory operations and are particularly useful for collaborative reduction operations. The operations possible on each device vary depending on its Compute Capability (CC), which is a versioning system for hardware iterations.

B. GPU-Side Calculation of ϕ-Coefficients

Mapping tasks to the environment of GPUs presents certain technical challenges stemming primarily from the memory and computation model constraints they impose. Workloads such as the computation of ϕ-coefficients present an imbalance in compute and memory workload and demand careful optimisation to achieve a performant implementation.

To compute ϕ-coefficients between each distinct pair of columns of a matrix with width w and height h, we begin by first rotating the matrix by 90°, so that the columns become rows and vice-versa. We subsequently condense each row of the rotated matrix, which up to this point is represented as a sequence of bytes that each store a truth value, by bit-packing the truth values into a bit vector. This operation results in each row of the matrix consisting of $\lceil w \div 8 \rceil$ bytes. The rotated and condensed matrix is represented as a linear block of memory (i.e., a mono-dimensional array of bytes) produced by placing each row in sequence.

Prior to the GPU-side computation taking place, the host transfers a copy of the rotated and condensed matrix into the memory of the device. The linear representation is preserved on the device, yet it is not sufficient to align the linear block to memory boundaries alone, as individual rows within the structure will be addressed and accessed. As such, each row within the linear block of memory is padded during transfer to ensure alignment with memory boundaries.

In regards to computation on the GPU, we employ a single-shot kernel approach consisting of a mono-dimensional grid which in turn comprises $\sum_{k=1}^{h} k$ mono-dimensional blocks. Each block of this kernel selects a distinct pair of rows from the matrix, and carries out the computation of the ϕ-coefficient between the two, writing the result to a global memory location upon completion, designated to the block.

When a block in the kernel begins execution, all threads within it calculate a pair of positive integers $(i_a, i_b) \in \{(a, b) \mid 0 \leq a < h \wedge a < b < h\}$ in which, i_a and i_b are the indexes of the rows this block will process. This calculation is performed by mapping the unique index b of the current block in the grid, to each of the two integers, in a non-divergent manner. The calculation computing i_a is shown in Equation (1).

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

Following the calculation of i_a, i_b can be computed, using i_a as a bound, as shown in Equation (2).

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

After determining which rows their block is operating on, threads begin summarising the two, splitting the work evenly between them. The process of summarising is as follows. In a block comprising s many threads, each thread processes $\lceil (w \div 8) \div s \rceil$ many pairs of bytes, in equally many steps. On each step, each thread reads two corresponding bytes b_a and b_b from rows i_a and i_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 byte houses eight 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 byte. For each of the resulting bytes 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.

Due to the relatively high number of memory reads performed by each thread, it is integral to maximise memory throughput and cache utilisation. When multiple threads within a warp perform a global memory access, those are coalesced into the minimum number of memory transactions, each of which incurs an access cost in terms of time. Memory transactions are 32, 64, or 128 bytes wide and must begin at an address that is naturally aligned to their respective width. When threads within a warp access well-aligned, consecutive memory addresses (i.e., a linear access pattern), the accesses are grouped into the same transaction. In our chosen data representation (i.e., linear block of rows) each row is padded appropriately to be aligned to 128 bytes. During summarisation, on each step, threads within the warp each read a byte from linear addresses with all reads falling in the same 'window' of 32 bytes. This facilitates the use of a single memory transaction to fetch the full set of bytes processed by the threads of the warp.

When all threads in a warp have completed summarising their respective portions of the two rows, a collaborative effort is initiated to combine the values of each thread's own counters to a warp-level set of counters. More specifically, following this operation, the warp as a collective arrives to a set of four counts $W = \{w_{ab}, w_{anb}, w_{nab}, w_{nanb}\}$ such that $w_\beta \in W = c_\beta(t_0) + c_\beta(t_1) + \cdots + c_\beta(t_{30}) + c_\beta(t_{31})$, where $c_\beta(t_k)$ is the value of c_β computed by the k^{th} thread in the warp.

In terms of implementation, threads within the same warp can benefit from warp shuffling operations via register exchange that allows for performant reduction operations between them. To compute each of the four w_β counts, we use the `__reduce_add_sync` intrinsic, available on devices of CC $\geq$ 8.0. On devices where $5.0 \leq$ CC $<$ 8.0, the behaviour of this intrinsic is emulated using manual warp down-shuffling paired with addition operations. Following either variant of the operation, the first thread in the warp has access to each count $w_\beta \in W$. The first thread writes each count to respective shared memory locations, designated to the warp. Once all warps in the block have finished, a reduction operation like the aforementioned is performed to sum the counts written into shared memory by each warp, into four counts $b_{ab}, b_{anb}, b_{nab}, b_{nanb}$ which form the final block-wide result of the summarisation operation.

The final step is the calculation of the ϕ-coefficient based on the summarised counts, which is performed by the 0^{th} thread of the block. The computation is floating point intensive, specifically for double-precision operations, yet it forms an insignificant portion of a block's workload, therefore we saw little benefit in parallelising it across the block. Where reasonable, the `fma` intrinsic is used, to compiler efforts in Fused Multiply and Add (FMA) contraction for these operations. Once the calculation is completed, the resulting 64-bit floating point result is stored in a designated (to the block) location in global memory, and the block exists.

While the first thread of a block is performing the ϕ-coefficient calculation, the whole block remains resident on the SM occupying resources. To minimise the under-utilisation of SM resources, we opted for the smallest size of block possible, that facilitates complete SM utilisation. More specifically, given the maximum number of threads T and blocks B the SM of the target device can accommodate, we chose blocks of size $S = \lfloor ((T \div B + 31) \div 32) \rfloor \times 32$. In the case of the Ampere microarchitecture (compute capability 8.X) for instance, where either a total of 1536 threads or 24 blocks are allowed per SM, the block size chosen is 64 threads.

C. Kernel Optimisations

Our kernel is memory-bound, meaning that the time each thread spends interacting with memory is disproportionate to the time spent computing. This is natural considering the relatively simple computations involved and focused initially our efforts on the optimisation of memory resource utilisation.

For reasons discussed in Section V-A, the way threads within a warp access global memory can significantly impact performance. The rationale behind the rotation of the input matrix by $90°$ is that threads within each warp will need to access consecutive values along what was, before the rotation, the columns of the matrix. Without rotation, threads would have to traverse the columns of the matrix, effectively jumping between non-adjacent addresses. In the worst case, each thread of a warp would require a separate memory transaction to fetch its respective portion of each row, on every step.

Besides rotating the matrix, increasing the fetch size of each thread from one to four bytes offers a two-fold benefit. First, a single, maximally sized transaction window is used as each of the 32 threads reads four consecutive bytes. Secondly, the 32-bit value fetched by each thread can be processed in the same number of steps as its 8-bit counterpart, resulting in better computational resource utilisation. GPUs are 32-bit devices and instructions involved in the computation, such as the PTX[2]

[2]PTX is an intermediate assembly-like language in the chain of compilation.

instruction `popc.b32` which operates on a 32-bit binary type, may not achieve maximum throughput when used with smaller types. In certain cases, using a smaller type than expected by the instruction results in a type conversion which itself consumes additional clock cycles. It is worth highlighting that small savings such as these accumulate due to their frequency of repetition in threads, across all parallel units. Besides, GPU hardware from CC ≥ 7.0 onwards can only compute the result of 16 `popc.b32`'s per clock cycle per SM. This instruction is rather costly and a dominant part of our compute workload.

This kernel necessarily relies heavily on memory resources, particularly on global memory. The access pattern exhibited by the threads of each warp is linear, well aligned, and maximises transaction window utilisation achieving approximately 97% memory throughput. While the access pattern of each row is regular, the order in which rows are accessed is not. This presents difficulty in caching efficiently, especially on the L1 level. More specifically, threads within each block use the block's index in the grid to compute a pair of indices (i_a, i_b), in the manner described in Section V-B, which is the same between them. These indices correspond to two rows on the matrix. It is interesting to note that adjacent blocks are more likely than not to compute the same i_a and different i_b. It would be beneficial for the reads of row i_a to be cached in L1 in the event two or more blocks resident on the same SM operate on that row. Since the specific scheduling algorithm employed for blocks is not published, it is not possible to reliably structure the grid in a way that makes good use of the L1 cache. Nevertheless, we opted for reads to row i_a to be cached in both L1 and L2 and disabled caching for i_b. The intuition here is to maximise the available portion(s) of i_a both on the SM level and also in the on-chip global cache, whilst accepting that i_b will have to be fetched each time. This proves effective, with an overall 45% L2 cache hit rate, especially for larger inputs where it is likely that many blocks across SMs or even in the same SM will be operating on the same i_a and different i_b between them.

Despite the kernel's heavy reliance on memory, improvements in compute performance can still offer a reduction in overall runtime. The compute portion of this kernel is dominated by the summarisation of rows, which we sought to optimise. The computation of e_{ab}, e_{anb}, e_{nab}, and e_{nanb}, described in Section V-B, involves bitwise operations such as negation and the AND operation. In practice, the result of each expression is put through a further bitwise AND with a bit mask which is intended to discount padding bits. This is necessary as, a row with n many bits where n is not a multiple of 32 would have been padded to account for the aforementioned 4-byte reads. In our examination of the PTX output produced by the compiler (NVCC version `12.3.107`), we observed the transformation of bitwise operations from the high-level C code into the equivalent instructions in PTX without further optimisation. For instance, part of the compiler's output for the expression $a\&{\sim}b\&mask$ between 32-bit binary types is shown in Listing 1.

Listing 1. Compiler's PTX output for the bitwise expression $a\&{\sim}b\&mask$.

```
not.b32  %r4, %r2;       // r4 = ~b
and.b32  %r5, %r4, %r1;  // r5 = r4 & a
and.b32  %r6, %r5, %r3;  // r6 = r5 & mask
```

Here, an opportunity for improvement presents itself, by collapsing the multiple instructions emitted by the compiler into the `LOP3` instruction. This instruction is documented [44] as performing a logic operation between three operands using a look-up table. The logic operation is performed over the corresponding bits of the three input operands, storing the result in the corresponding bit of the output register. We manually substituted the bitwise expressions in the C code with the respective `LOP3` instruction, to enforce its use. In the example from Listing 1, the equivalent `LOP3` instruction used was `lop3.b32 %r4, %r1, %r2, %r3, 0x20;`. Here, `0x20`, or in binary `0b00100000`, encodes the results column of the truth table of the expression $a\&{\sim}b\&c$. This binary value is used to 'look up' the result (bit), dependent upon the truth values of the three operands. There is only one possible assignment to the variables that would make this expression hold, hence why only one bit is set in the corresponding results column. In our tests, the substitution of the four bitwise expressions with `LOP3` instructions yielded a consistent $3\% - 5\%$ improvement in runtime performance.

VI. Performance Evaluation of GPU Approach

To evaluate the performance of our kernel, we used a realistic input Ladder Logic program with 6452 variables. As such, we simulated the RL framework invoking the GPU kernel with a matrix of 6452 rows when 1, 2, 10, and 30 million observations (columns) had been made. For each input, we deployed the kernel and measured the time taken to produce the ϕ-coefficient values for each matrix. Ten runtime measurements were taken for each input on each GPU tested. Three systems were used housing an RTX 4090 at 2235MHz with 24GB GDDR6 memory, an RTX 3090ti at 1395MHz with 12GB GDDR6 memory, and eight A100 GPUs at 1410MHz with 40GB GDDR6 memory respectively. The latter system is a node in a GPU cluster [45], from which one GPU is utilised. To avoid interference from resource sharing, we ensured no other jobs could use this node.

To put GPU runtimes into perspective, we used our original C-based implementation for the CPU, running on a Ryzen 9 7950x, to collect runtime measurements. Our CPU implementation operates on the same principle as a block in the GPU kernel. More specifically, a sequentially incrementing integer $x \in [0, q)$ where $q = \sum_{k=1}^{h} k$ is the number of row pairs to be checked against, is used to base row index calculations as described in Section V-B. The corresponding rows are subsequently summarised before their ϕ-coefficient is computed. This approach is parallelised effortlessly across t many CPU threads, by splitting the range of integers into t sub-intervals and assigning one to each thread.

The runtime of GPUs for each input can be seen in Figure 4 alongside runtimes from the single- and multi-threaded CPU

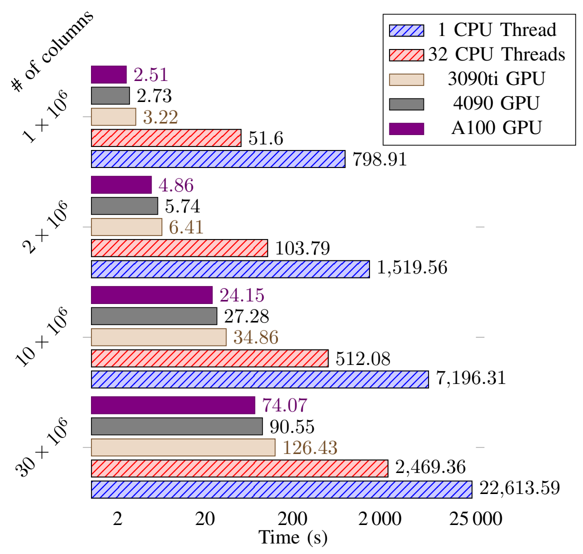

Fig. 4. GPU and CPU runtimes, in logarithmic scale, for a matrix of 6452 rows and a variable number of columns.

implementation. We note that for the 10×10^6 and 30×10^6 column inputs, a single time sample was taken from the single-threaded CPU implementation due to high time demands, whereas all others are the average of ten samples. For each GPU, time grows linearly w.r.t. the input problem. The A100 GPU, despite featuring an earlier hardware generation to the RTX 4090 and fewer cores, outperforms both others. This is to be expected given the significantly wider memory interface than its counterparts, which is crucial for a workload such as this. In terms of utilisation, this kernel achieves 97% memory, and 16% compute utilisation w.r.t. the respective theoretical limits. The low compute utilisation, in this instance, is an indication of more room for computation rather than poor resource management, as previously discussed. As the size of the input problem increases, any initialisation overheads form a smaller part of the kernel's execution time, hence resulting in a more accurate reflection of the kernel's compute time.

Overall, we observe consistently faster runtimes from the GPU kernel versus the CPU, on a range of GPUs. This is beneficial for our application as, besides significantly increasing our invariant finding capabilities, a hybrid approach such as this permits the CPU to continue execution after offloading work to the GPU. Additionally, scaling to utilise multiple GPUs, with little or no overhead, is possible as discussed in Section VIII.

Whilst the CPU we chose is similar in price and release date to the RTX 4090, we highlight that accurately comparing such different hardware is by no means trivial nor our objective. We provide this comparison to our initial CPU-side implementation as a general point of reference to the reader.

VII. Impact of Acceleration & Evaluation

The current approach to invariant generation for large programs focuses on associations between individual atoms rather than expressions. Although the latter has been successfully demonstrated on smaller examples, as highlighted in Section IV, its application to industrial Ladder Logic remains unexplored. However, with the computational acceleration provided by our GPU implementation, the determination of ϕ-coefficients for multi-variate propositions has become increasingly feasible. This advancement opens new avenues for analysing complex systems with greater efficiency.

Our invariant finding framework, while not guaranteeing complete state space coverage, ensures that a sufficient proportion is explored to intermittently propose invariants. Previously, the time required to collate observation data, compute ϕ, synthesise propositions, and resume exploration posed a significant bottleneck. Experimental results indicate that the reduction in lead time for ϕ computation enables greater property throughput. This improvement illustrates the practicality of our approach in real-world applications.

Furthermore, our method for computing ϕ-coefficients can be extended to consider other datasets concerning relationships between dichotomous data [46]. More broadly, our implementation may be adaptable to other correlation measures involving continuous values. This adaptability would increase applicability to broader big data analysis, correlation-based feature selection and dimensionality reduction [47].

VIII. Conclusions, Future Work and Scalability

Our current GPU kernel and work distribution are designed for one GPU to be used at a time. Whilst this delivers far superior runtime results relative to its CPU counterpart, we are exploring a scheme whereby multiple GPUs are used in the effort. We are investigating means by which the matrix rows can be distributed across GPUs in such a way that the same list of ϕ-coefficients can be calculated disjointedly, and without communication between devices.

Besides scaling across GPUs, we are investigating a streaming approach, where the GPU preserves the results of its summarisation for each row pair, so that when new columns are added to the matrix and the kernel is subsequently invoked, it suffices to summarise these new columns only, for the row pairs. As our kernel is designed to be run repeatedly as new observations are made by the RL framework, this approach will permit us to pick up where we left and only summarise the new columns added to the matrix, offering two-fold benefits. First, the memory of the GPU does not pose a hard limit anymore as only the relevant, non-summarised portion of the matrix needs to be stored. Secondly, the runtime of the GPU kernel depends solely on the size of the relevant portion of the matrix, without the need for re-computation of earlier sections. The global memory footprint of this approach depends on the number of rows in the matrix and will likely be $4 \times 8 \times (h \times (h-1) \div 2)$ bytes, as the four block-level counts b_{ab}, b_{anb}, b_{nab} and b_{nanb} will have to be preserved and represented as eight-byte counters to account for the continually increasing values. This is a rather small price to pay for the potential benefits of this approach, yet heuristics will have to be used to determine the points where launching this kernel is most beneficial.

References

[1] L. M. Barroca and J. A. McDermid, “Formal methods: Use and relevance for the development of safety-critical systems,” *The Computer Journal*, vol. 35, no. 6, pp. 579–599, 1992.

[2] J. F. Groote, S. F. van Vlijmen, and J. W. Koorn, “The safety guaranteeing system at station Hoorn-Kersenboogerd,” in *Proceedings of the Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*. IEEE, 1995, pp. 57–68.

[3] A. E. Haxthausen, M. L. Bliguet, and A. A. Kjær, “Modelling and verification of relay interlocking systems,” in *Monterey Workshop*. Springer, 2008, pp. 141–153.

[4] A. Ferrari and M. H. T. Beek, “Formal methods in railways: a systematic mapping study,” *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–37, 2022.

[5] 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*, vol. 16, no. 6, pp. 685–711, 2014.

[6] C. Limbrée, Q. Cappart, C. Pecheur, and S. Tonetta, “Verification of railway interlocking-compositional approach with ocra,” in *International Conference on Reliability, Safety, and Security of Railway Systems*. Springer, 2016, pp. 134–149.

[7] P. James and M. Roggenbach, “Automatically verifying railway interlockings using sat-based model checking,” *Electronic Communications of the EASST*, vol. 35, 2011.

[8] A. E. Haxthausen, J. Peleska, and R. Pinger, “Applied bounded model checking for interlocking system designs,” in *International Conference on Software Engineering and Formal Methods*. Springer, 2013, pp. 205–220.

[9] K. Kanso, F. Moller, and A. Setzer, “Automated verification of signalling principles in railway interlocking systems,” *Electronic Notes in Theoretical Computer Science*, vol. 250, no. 2, pp. 19–31, 2009.

[10] 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*, vol. 28, no. 1, pp. 154–158, 2009.

[11] B. Lloyd-Roberts, P. James, and M. Edwards, “Mining invariants from state space observations,” in *Extended abstract at 33rd Nordic Workshop on Programming Theory, NWPT*, 2022.

[12] B. Lloyd-Roberts, P. James, M. Edwards, S. Robinson, and T. Werner, “Improving railway safety: Human-in-the-loop invariant finding,” in *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–8.

[13] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[14] P. James, A. Lawrence, F. Moller, M. Roggenbach, M. Seisenberger, A. Setzer, K. Kanso, and S. Chadwick, “Verification of solid state interlocking programs,” in *International Conference on Software Engineering and Formal Methods*. Springer, 2013, pp. 253–268.

[15] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal methods in system design*, vol. 19, pp. 7–34, 2001.

[16] P. Bjesse and K. Claessen, “Sat-based verification without state space traversal,” in *International Conference on Formal Methods in Computer-Aided Design*. Springer, 2000, pp. 409–426.

[17] S. Bensalem, Y. Lakhnech, and H. Saidi, “Powerful techniques for the automatic generation of invariants,” in *International Conference on Computer Aided Verification*. Springer, 1996, pp. 323–335.

[18] A. R. Bradley, “SAT-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[19] C. Savage, “Depth-first search and the vertex cover problem,” *Information Processing Letters*, vol. 14, no. 5, pp. 233–235, 1982.

[20] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[21] M. L. Puterman, “Markov decision processes,” *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[22] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” in *International conference on machine learning*. Pmlr, 2014, pp. 387–395.

[23] Y. Hu, W. Wang, H. Jia, Y. Wang, Y. Chen, J. Hao, F. Wu, and C. Fan, “Learning to utilize shaping rewards: A new approach of reward shaping,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 931–15 941, 2020.

[24] F. S. Melo, S. P. Meyn, and M. I. Ribeiro, “An analysis of reinforcement learning with function approximation,” in *Proceedings of the International Conference on Machine Learning*, 2008, pp. 664–671.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[26] S.-i. Amari, “Backpropagation and stochastic gradient descent method,” *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[27] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *arXiv preprint arXiv:1602.01783*, 2016.

[28] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[29] J. Ekström, “The phi-coefficient, the tetrachoric correlation coefficient, and the pearson-yule debate,” 2011.

[30] H. Tang, R. Houthooft, D. Foote, A. Stooke, O. Xi Chen, Y. Duan, J. Schulman, F. DeTurck, and P. Abbeel, “# exploration: A study of count-based exploration for deep reinforcement learning,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[31] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén, “Improving playtesting coverage via curiosity driven reinforcement learning agents,” *arXiv preprint arXiv:2103.13798*, 2021.

[32] S. Kwek, “On a simple depth-first search strategy for exploring unknown graphs.” in *WADS*, 1997, pp. 345–353.

[33] H. Zhang and T. Yu, “Alphazero,” *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pp. 391–415, 2020.

[34] B. S. Everitt, *The analysis of contingency tables*. CRC Press, 1992.

[35] NVIDIA, “Cuda c++ programming guide release 12.4,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 3 2024.

[36] R. Ansorge, Ed., *Programming in Parallel with CUDA: A Practical Guide*. Cambridge University Press, 2022.

[37] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Survey of machine learning accelerators,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–12.

[38] S. An and S. SEO, “Highly efficient implementation of block ciphers on graphic processing units for massively large data,” *Applied Sciences*, vol. 10, p. 3711, 05 2020.

[39] F. Pantekis and P. James, “Towards massively parallel gpu assisted sat,” in *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE Computer Society, 11 2022, pp. 120–126.

[40] 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*, 2023.

[41] M. Osama, A. Wijs, and A. Biere, “Certified sat solving with gpu accelerated inprocessing,” *Form. Method. Syst. Des.*, 8 2023.

[42] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, “Dynamic parallelism for simple and efficient gpu graph algorithms,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA^3 '15. New York, NY, USA: Association for Computing Machinery, 2015.

[43] M. Flynn, *Flynn's Taxonomy*. Springer US, 07 2011, pp. 689–697.

[44] NVIDIA, “Parallel thread execution isa version 8.4,” https://web.archive.org/web/20240324062329/https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 3 2024.

[45] A. Group, “Atos delivers BullSequana X410 supercomputer to Swansea University,” https://web.archive.org/web/20220703170610/https://atos.net/wp-content/uploads/2021/03/PR-Atos-delivers-BullSequana-X410-supercomputer-to-Swansea-University-final-GLOBAL.pdf, Atos Group, River Ouest, 80 quai Voltaire, 95877 Bezons cedex, Press Release, 2021.

[46] E. Kijsipongse, S. U-ruekolan, C. Ngamphiw, and S. Tongsima, “Efficient large pearson correlation matrix computing using hybrid mpi/cuda,” in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2011, pp. 237–241.

[47] S. Chormunge and S. Jena, “Correlation based feature selection with clustering for high dimensional data,” *Journal of Electrical Systems and Information Technology*, vol. 5, no. 3, pp. 542–549, 2018.