ReLAccS: A Multi-level Approach to Accelerator Design for Reinforcement Learning on FPGA-based Systems

Akhil Raj Baranwal, Salim Ullah, Siva Satyendra Sahoo, and Akash Kumar, *Senior Member, IEEE*

***Abstract*—Reinforcement learning, specifically Q-Learning, with human-like learning abilities to learn from experience without any a priori data, is being increasingly used in embedded systems in the field of control and navigation. However, finding the optimal policy in this approach can be highly compute-intensive, and a software-only implementation may not satisfy the application's timing constraints. To this end, we propose optimization methods at multiple levels of accelerator design for reinforcement learning. Specifically, at the architecture-level, we exploit the instruction-level parallelism and the spatial parallelism in FPGAs to improve the throughput over state-of-the-art designs by up to 34%. Further, we propose LUT-level optimizations to reduce the resource utilization and power dissipation of the accelerator. Finally, we propose algorithm-level approximation that can be used for acceleration of Q-Learning problems with more states and for reducing the peak power dissipation. We report up to 10x reduction in power dissipation with marginal degradation in quality of results.**

***Index Terms*—Reinforcement Learning, Embedded Systems, Cross-layer System Design, FPGA, High-level Synthesis**

I. Introduction

With the advent of deep learning techniques, there has been a steep rise in the variety of applications that can benefit from the rapid advances in machine learning. Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have gained popularity among applications that implement some form of computer vision and temporal dynamic behavior (e.g. Natural Language Processing) respectively. Similarly, there is an increase in the need for applying machine learning methods in applications that require expert human operators. Applications that implement one or more among – *optimization*, *control*, *monitoring*, and *maintenance* – and involve some form of decision-making can benefit from tools that exhibit *human-like* learning. Typical examples of such application areas include process planning, smart buildings, robotics, autonomous vehicles, inventory monitoring, etc.

Reinforcement Learning (RL)-based approaches provide the closest resemblance to how humans learn [1]. RL involves allowing an agent to learn from its interactions with the environment. RL methods combined with deep neural networks, called *Deep RL*, have been able to surpass human expertise in the field of video games and multiplayer contests [2], [3]. However, the high power dissipation and the latency of the associated neural network might be infeasible for resource and

A. R. Baranwal, S. Ullah, S. S. Sahoo and A.Kumar are with The Chair for Processor Design, TU Dresden, Germany (emails: akhil.baranwal@mailbox.tu-dresden.de; salim.ullah@tu-dresden.de; siva_satyendra.sahoo@tu-dresden.de; akash.kumar@tu-dresden.de)

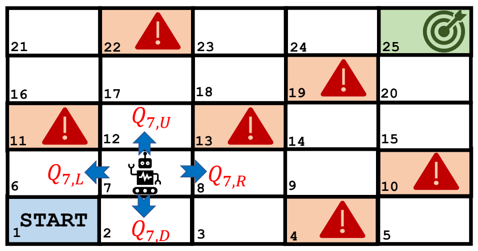
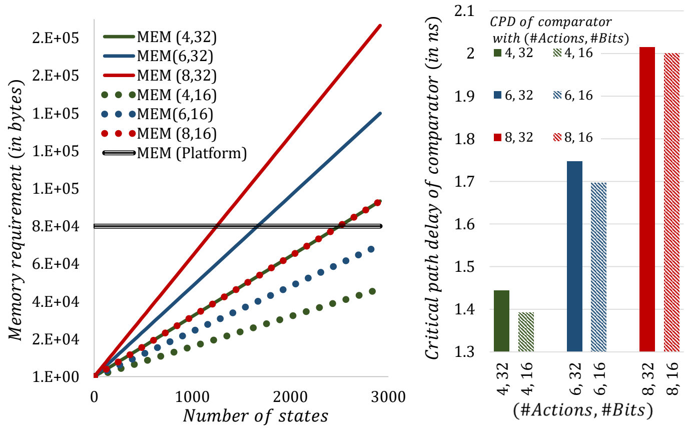


Fig. 1: Sample *grid-world*

performance constrained embedded systems. In this article, we limit our focus to traditional RL techniques. In such methods, the agent is allowed to take certain actions and the long-term effectiveness of each action is estimated from the returns obtained by the agent subsequent to that action till it reaches the goal. This way the agent can perform model-free learning, by allowing the machine to learn dynamically from experience rather than statically from a priori data. For instance, in the *grid-world* problem shown in Fig. 1, the autonomous agent, shown in the figure at the 7^{th} position on grid[1], is expected to be able to find the optimal path from the starting state to the goal state. In this problem, the position of the obstacles, shown in red in the figure, is not known a priori and the agent must be able to find the optimal path by continuously trying out different actions at each state. However, the agent must perform a large member of iterations/simulations to learn the *optimal policy* for the problem. The optimal policy search can be highly computation-intensive and a delayed search can lead to violation of timing constraints in reactive embedded systems.

The optimal policy search in the grid world problem would be to find the best action that should be taken at each state. For instance, as shown in Fig. 1, the agent, located at position 7 can decide to take one of four actions which are shown as arrows. So, each state in the grid is associated with four state-action pairs. One way of representing this policy is to assign a *Q-value* to each state-action pair. For instance, in the grid-world problem, the values $Q_{7,U}, Q_{7,D}, Q_{7,L}, Q_{7,R}$ represent the Q-value for taking the actions *Up*, *Down*, *Left* and *Right* respectively, from state 7. In the grid-world example, the Q-value $q_{s,a}$ of each state-action pair (s, a) indicates the average long-term benefit of taking action a from state s while aiming for the goal state. During the learning stage, the matrix

[1]In this sample problem, the agent's position on the grid at any specific instant is sufficient to define its current state



(a) Scaling of memory requirements (b) Scaling of delay

Fig. 2: Impact of problem complexity and design decisions on memory requirements and delay

Q, representing the Q-value of all state-action pairs in the problem, needs to be updated. This updating is based on the reward— a more short-term benefit—received by the agent from the environment for each of the pairs. Similar to Q, the rewards are represented by a matrix R. So, any computation of the learning process must have fast access to the Q and R matrices. This determines the memory requirements of the accelerator for RL. For instance, in the grid-world shown in Fig. 1, the two matrices, Q and R, contain 100 elements each—the product of the number of states (25) and the number of actions from each state (4). So the local memory would need to accommodate at least $2 \times s \times a$ matrix entries.

The plots shown in Fig. 2(a) show the rise in the estimated memory requirements for storing Q and R in local memory of an accelerator with increasing number of states. The trends are shown for $4, 6$ and 8 possible actions and with two types of precision—32-bits and 16-bits—used to represent the values in the matrix. The notation *MEM(a,b)* represents the memory requirements where there are a actions and the matrix entries are each expressed in 'b' bits. As expected, the memory requirement increases linearly with rising number of states. An essential computation step in the RL method is finding the maximum of Q-value for all possible actions from any arbitrary state. The hardware realization of the search for *maxima* involves using *comparators*, whose bit-widths are determined by the precision used for the Q-matrix values. The bar-charts, shown in Fig. 2(b), depict the critical path delay of such comparators. As seen in the figure, the reduction in the critical path delay with precision scaling is more prominent in the case of a lower number of actions.

The patterns shown in Fig. 2 provide an insight into some of the scopes and constraints in the design of an accelerator for RL. As shown in the plot, the maximum available memory can limit the problem size (in terms of state-action pairs) that can be solved by the accelerator. For instance, if the maximum memory in the hardware is limited to *MEM(Platform)* as shown by the horizontal line in Fig. 2(a), then, with a given number of actions, only problems with states fewer than the intersection points with the corresponding solid line can be computed. However, as shown by the dotted lines, representing the matrices' memory requirements, reduced precision can increase this threshold number of states. Further, as shown in the bar-chart in Fig. 2(a), precision scaling can reduce the propagation delay of operations in the critical path and result in improved clock frequency of operation. In addition, processing at lower precision ideally reduces the power dissipation of the accelerator. However, it must be noted that the quantization error introduced by such precision scaling can reduce the quality of result—in terms of the convergence of Q matrix values of the state-action pairs. In some cases of precision scaling, a larger number of trials can be performed to recover such degradation in results. In such cases, the throughput of the accelerator for RL should be improved to enable the system's response within a fixed time interval.

Most state-of-the-art works do not consider these aspects together—available resources on the hardware platform, hardware-level optimizations like precision scaling, microarchitecture-level optimizations for throughput etc. This can lead to bottlenecks in the use of RL in embedded systems that are resource-constrained and/or demand stricter timing requirements. To this end, we propose a multi-level optimization approach to the design of Field Programmable Gate Array (FPGA)-based accelerators for RL. While the proposed methods at each level are orthogonal to each other, they result in improving the overall performance in the design of accelerators for Q-Learning. The proposed techniques identify and exploit multiple opportunities for optimization in this specific problem – something that has been missing in existing approaches. The related contributions are listed below.

Contributions:

1) We propose novel architecture-level improvements in the design of the accelerator for RL. Specifically, we explore opportunities of Instruction Level Parallelism (ILP) and generate a pipelined micro-architecture by exploiting the spatial parallelism offered by FPGAs. Using these proposed improvements, we report up to 34% improvement in throughput compared to state-of-the-art designs.

2) At the hardware-level, we propose novel optimizations customized for the state-update process of the RL algorithm. Specifically, we implement constant arithmetic operations using lookup table (LUT)-level optimizations to provide hardware designs with varying performance metrics that can be used by the accelerator. For example, compared to Vivado constant multiplier IP-based implementation of the state-update equation, our LUTs-based low-level optimization can achieve up to 27% and 48% reductions in resource utilization and energy consumption, respectively. Our proposed designs can be implemented on any state-of-the-art FPGA having at least 6-input LUTs.

3) We propose a novel algorithm-level approximation method for overcoming the hardware limitations and memory bottlenecks to implement RL in resource-constrained embedded systems, which is extensively scalable, offering good trade-offs in speed-up, power, and resource utilization. The algorithm along with the proposed architecture can offer a speed-up of about $1192\times$ when using 275 mW and $153\times$

when using 29 mW.

The rest of the paper is organized as follows. In Section II, we provide the relevant background and a brief overview of related works. The system model used for the evaluation of the proposed methods is presented in Section III. In Section IV, the proposed optimization methods at different levels are explained in detail. In Section V, we discuss the results from the experimental evaluation of the proposed methods and conclude the article in Section VI with a discussion on the scope for related future research.

II. Background and Related Works

A. Reinforcement Learning

Recent advancements in machine learning have tried to integrate deep learning into the more traditional RL-based techniques. However, traditional RL algorithms are a low-cost alternative to more advanced DeepRL methods. In addition to having a lower resource requirement overhead, RL implementations usually result in lower power dissipation. Further, unlike the decision-making with DeepRL techniques, that requires an inference across some neural network, the decision-making with RL usually involves looking-up in a table of values to determine the best action. This makes RL an attractive option for applications where the decision-making process should not be costlier than the application itself. For example, Sahoo, et al. [4] report using RL for dynamic adaptation to varying operating environments of an embedded System-on-Chip. Similarly, Chatterjee, et al. [5] have shown the usability of RL in light-weight algorithms on small IoT nodes. In this context, we have limited the scope of this article to applications that find RL as a suitable algorithm for decision-making. A comparison of the implementation of DeepRL and RL for any specific application is beyond the scope of this paper.

Q-Learning is one of the more widely used RL algorithms for handling dynamic unknown environments in a model-free, stochastic way without requiring adaptations within the agent. It aims at finding an optimal policy for a finite Markov Decision Process (MDP), which implies that the states and possible actions for each state are known and has several use-cases in literature. In [6], the authors use RL to distribute computation load onto edge devices. Similarly, in [7], the authors proposed non-linear control strategies in a continuous action domain to optimise adaptability and response time of the system. In [8], the authors proposed RL techniques to manage thermal optimisation for increasing the lifetime of multicore systems. In healthcare, the demands for quick and reliable medical diagnosis and the necessity to assisting medical personnel for better decision-making signal the expected use of advanced big data analysis and machine learning techniques [9].

In several decision making scenarios the agent has to work with limited visibility. For instance, in the scenario presented in [10] with the RoboCup context, or the navigation required by space-exploration rover [11], it can be noted that the position of the robot heavily influences the actions it can take and therefore, it is not necessary for the robot to process all decisions at once. With the flexibility of a policy-approximating algorithm like Q-Learning, an agent can virtually *reprogram* itself as it sequentially progresses through an environment with or without a known trend. This opens up opportunities for automation, control, and decision making to complex environments that can be *learned* by an agent gradually, much like the characteristics of a human, but with the precision and memory of electronics.

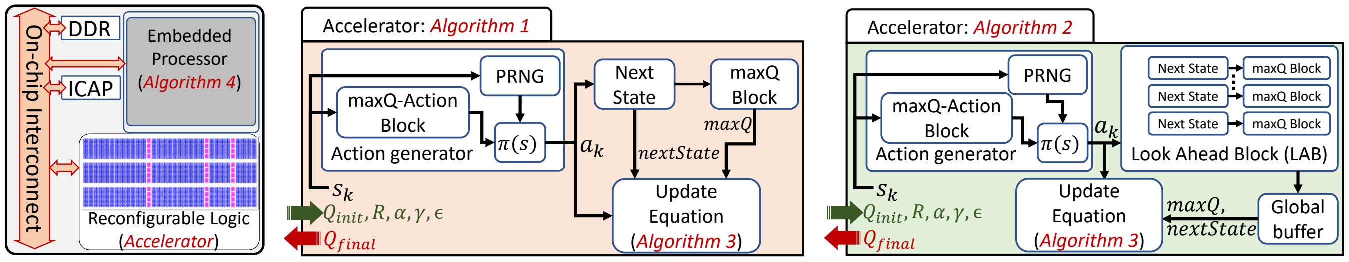
Algorithm 1 Q-Learning: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$

```
Require: $\mathcal{S}$, $\mathcal{A}$, $R$, $T$, $\alpha$, $\gamma$, $\pi$
    States $\mathcal{S} = \{1, \dots, N\}$;
    Actions $\mathcal{A} = \{1, \dots, Z\}$;
    Reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$;
    Transition mapping function $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{T}$;
    Learning rate $\alpha \in [0, 1]$;
    Discounting factor $\gamma \in [0, 1]$;
    Action choosing strategy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{T}$
 1: Initialize $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ arbitrarily
 2: Start with arbitrary state $s \leftarrow s_0 \in \mathcal{S}$
 3: loop
 4:   while $k <$ maxIterations do
 5:     Select action $a$ from state $s$
 6:     $a \leftarrow \pi(s)$
 7:     $s' \leftarrow T(s, a)$            {Receive the new state}
 8:     $r \leftarrow R(s, a)$               {Receive the reward}
 9:
        $Q_{(s,a)} = (1-\alpha) \times Q_{(s,a)} + \alpha \times (R_{(s,a)} + \gamma \times Q_{(s+1,a)_{\max}})$   (1)
                                          {Update the Q-value}
10:     $s \leftarrow s'$          {Update state for next iteration}
11:   end while
12: end loop
```

Approximate implementations of the Q-Learning algorithm [12] have been reported to converge, as the focus is not the exact values, but sufficient distinction to guarantee convergence. Generic applications of Q-Learning implement a exploration-exploitation dilemma because of the huge number of states required to process a complex environment. For example, the implementation of Q-Learning in Atari by learning from pixel data uses a total of 10^{67970} states [11], [13]. Since available hardware resources are limited when dealing with such a large number of states, it is imperative that a co-design approach between software and hardware is established to efficiently accelerate the RL task. To this end, we propose Reinforcement Learning Accelerator on FPGA based Systems (ReLAccS), a multi-level optimization platform for RL applications that aims to be a configurable hardware implementation that is driven dynamically with high-level languages to bridge the gap between the slow, yet flexible software side and the fast, yet deterministic hardware side.

B. Related Works

Applications that implement RL have to suffice with a finite, discrete state-action space, and potentially face issues with the scaling of state-action pairs. The primary cause of this inability to scale can be attributed to the excessive time of convergence with larger state-action pairs. For instance, in using RL for



(a) Platform Model (b) Implementation of Algorithm 1 (c) Implementation of Algorithm 2

Fig. 3: Hardware platform and accelerator model

navigation in robots, Wicaksono [14] indicates the limitation caused by the substantial training time when using a large Q-table. Such high convergence times may create real-time constraint violations in some applications. This provides the primary motivation for the acceleration of the Q-Learning algorithm. The low-cost RL accelerators provide an alternative to more resource-hungry DeepRL techniques and provide the scope for faster convergence for an increasing number of state-action pairs. The need for acceleration of RL is also emphasized by more recent works such as [15], where Camelo, et al. show considerable speed-ups compared to traditional RL implementations for both single- and multi-agent systems.

Most software-based accelerations of Q-Learning exploit the parallelism generated with having independent runs on the same data to achieve optimality quicker. For instance, the *Gorila* architecture proposed in [16] creates parallel actors and learners on a distributed neural network to reduce the total wall time required to process a given Q-Learning scenario. Similarly, in [17], the authors propose an algorithm-agnostic framework for achieving a speed-up of $8\times$ using their *PAAC* architecture. However, accelerating a specific application on a GPU simply exploits the amount of *sparsity* associated with the particular application, as shown by [18]. Moreover, it is possible for the global rewards to change based on the actions that an agent takes in an environment, which is a major roadblock to the parallelization.

A limited number of hardware-based accelerators for Q-Learning have been proposed which focus on generating a parallel architecture specifically designed for RL. Da Silva et al. [19] propose a novel FPGA-based architecture to accelerate the Q-Learning task through massive parallelisation of resources. S. Spanò, et al [12] propose modifications on this architecture to focus on targets with low-power and limited resource applications while maintaining high throughput. They implement approximated multipliers and a tree of binary comparators as a trade-off in area and speed. It must be noted that the methods presented in this article are orthogonal to those proposed in [12] and can be integrated into our proposed design flow. To the best of our knowledge, the designs in [12], [19] report the highest throughput metrics for Q-Learning accelerations implemented in FPGAs.

The hardware architecture proposed in [19] divides the Q-matrix in multiple *lanes*. Although this can achieve a very high throughput, as reported, it must be noted that only one lane is active during one iteration. Therefore, although their design has some design attributes of a Single Instruction Multiple Data (SIMD) architecture, it is limited in capacity by the RL algorithm itself, which is a rather sequential algorithm. We consider this observation a huge inspiration for our proposed design, wherein we focus our efforts on implementing Instruction Level Parallelism (ILP) and pipelining the design. Further, [19] implements its rewards as registers, and not as elements in Block RAMs (BRAMs), which makes the design rather static in terms of re-programmability, especially in scenarios where environmental parameters demand dynamic volatile rewards [20] or the rewards are being constantly learned [21]. To the best of our knowledge, no architecture presently claims to be extensively scalable to an abnormally high number of states, like a typical scenario of playing Atari [13], primarily due to resource constrains and a lack of efficient data management directives.

To summarize the related works, all software-based acceleration techniques focus on improving the performance by using generic methods such as improving the caching etc. Similarly, all the state-of-the-art works that use hardware optimization, use generic methods such as having parallel datapaths, reducing the critical path delay by using hardware approximation etc. However, the inherently sequential nature of the Q-Learning algorithm poses limitations to the level of improvements that can be achieved with such generic methods. To address these limitations, we identify and exploit the optimization opportunities specific to Q-Learning. The proposed methods result in improvements that span across multiple levels and we report considerable performance gains over the state-of-the-art designs.

III. System Model

A. Algorithm for basic Q-Learning

The different steps of Q-Learning are listed in Algorithm 1 along with the notation used in the rest of the article. As mentioned in Section I, the matrix Q stores the value function of each state-action pair and the matrix R stores the rewards associated with each state-action pair. A randomly initialized Q and an arbitrarily chosen state is used to start the learning process. At each iteration an action is chosen based on the policy π. Based on the chosen action, the next state is selected and the corresponding reward is used to update the Q matrix as shown in Eq. (1). We assume a deterministic environment

Algorithm 2 Q-Learning with Look-Ahead: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$

```
Require: $\mathcal{S}$, $\mathcal{A}$, $R$, $T$, $\alpha$, $\gamma$, $\pi$
 1: Initialize $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ arbitrarily
 2: Start with arbitrary state $s \leftarrow s_0 \in \mathcal{S}$
 3: LAB global buffer $\leftarrow Q_{(s+1,a)_{max}} \forall\ a$ for state $s$
 4: loop
 5:     while $k <$ maxIterations do
 6:         Select action $a$ from state $s$
 7:         $a \leftarrow \pi(s)$
 8:         $s' \leftarrow T(s,a)$                {Receive the new state}
 9:         $Q_{(s',a)_{max}} \leftarrow$ LAB global buffer
10:         LAB global buffer $\leftarrow Q_{(s+1,a)_{max}} \forall\ a$ for state $s'$
11:         $r \leftarrow R(s,a)$                 {Receive the reward}
12:         $Q_{(s,a)} = (1-\alpha) \times Q_{(s,a)} + \alpha \times (R_{(s,a)} + \gamma \times Q_{(s+1,a)_{max}})$
                                                  {Update the Q-value}
13:         $s \leftarrow s'$            {Update state for next iteration}
14:     end while
15: end loop
```

for our current work. Hence, the next state is a function of the current state and the chosen action only. The process of action selection and update of Q is repeated for $maxIterations$ times.

B. Accelerator

Fig. 3 shows the accelerator model used in this article. As shown in Fig. 3(a), we assume an FPGA-based System-on-Chip (SoC) as the hardware platform. It contains an embedded processor along with reconfigurable logic similar to the Zynq EPP [22]. The proposed accelerator architecture is implemented on the reconfigurable logic. Fig. 3 also shows the block diagram of two different accelerator designs that are implemented on the programmable logic. Fig. 3(b) shows the block diagram of the basic implementation of Algorithm 1. Similarly, Fig. 3(c) shows the block diagram of the proposed architecture (Algorithm 2), and is explained in detail in the next section. In both cases, the data structures for Q, R, α, γ and ϵ (used in the epsilon-greedy policy) are fetched from the main memory through streaming interfaces with the on-chip AXI interconnect [23]. Fig. 3 also shows how the hardware optimization (Algorithm 3) and the algorithm-level approximation (Algorithm 4) map into the overall system design.

IV. Multi-level Accelerator Design Approach for Reinforcement Learning

A. Accelerator Architecture

Hardware design demands rationalising trade-offs. In case of sufficient availability of resources, it is desirable to implement the maximum possible parallelism. As noted in [19], the most critical data path for Eq. (1) is finding the maximum Q-value. Hence, we implement the Q-Learning with Epsilon Greedy (QLEG) algorithm [1] and try to maximize the throughput with special focus on optimising the $maxQ$ calculation. A pipelined version of the QLEG algorithm is made by seeking instruction-level parallelism. The function graph for a single iteration (henceforth called *episode*) through

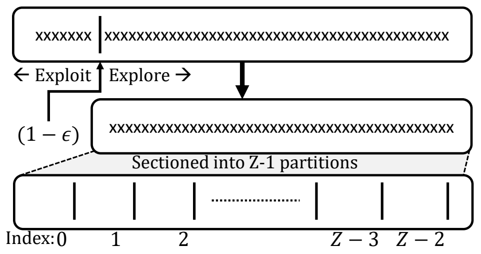

xxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
← Exploit
Explore →
$(1-\epsilon)$
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Sectioned into Z-1 partitions
Index: 0
1
2
$Z-3$
$Z-2$

Fig. 4: Evaluating the ϵ-greedy algorithm with one Pseudo Random Number Generator (PRNG) (the range of which is depicted by the top horizontal bar). ϵ is assumed to be 0.8 as an example. A total of Z unique comparators are used.

this pipelined implementation is shown in Fig. 5. Thereby, one iteration through the loop mentioned in Algorithm 1 takes 10 clock cycles to complete. We divide the functionality of the accelerator in discrete modules as shown in the block diagram in Fig. 3(c). The modules are further encapsulated by a control module that is responsible for handling the number of times the Q-values should be updated. The control module is also responsible for streaming the Q and reward matrices between the system's main memory and the accelerator's local memory. The proposed architecture is described in terms of the design blocks listed next.

1. Episode Iterator controls the number of iterations ($maxIterations$) made by the accelerator core and defines the data flow between two successive iterations.

2. maxQ-Action Block buffers the current Q-values for the present state and calculates the action corresponding to the maximum Q-value. The output is solely used by the *action-generator* module to implement the action choosing policy described by Eq. (2).

3. Action Generator is responsible for generating an action (a_k) according to the action choosing strategy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{T}$. Our proposed design uses just one random number to calculate both choices. A Linear Feedback Shift Register (LFSR) implemented as a PRNG is used along with comparators to determine this, as illustrated in Fig. 4. The LFSR is of 63 bits, as it is a small footprint design (2-degree polynomial tap) and at 200 MHz produces a sequence with a period of 1462+ years [24]. One comparator is used for the choice between exploitation/exploration, and $Z-1$ comparators are used for extracting the randomly chosen action. Here, $Z = |\mathcal{A}|$ represents the maximum number of possible actions from any arbitrary state. Thus, the sectioning of the entire range of the PRNG is done using a total of Z unique comparators. This means that the range of PRNG is divided first into two partitions, out of which the second is then divided into $Z-1$ partitions addressed using a partition index (i_p). Although i_p is obtained readily, it still has to be modified slightly before being mapped to a legitimate action-index (i_a). This can be best described by an intuitive example.

Suppose that $Z = 5$, $a_{max} = 2$, and that the PRNG generates $i_p = 2$. For a given state s, a_{max} is the action corresponding to the highest Q-value. This results in a total of ($Z-1 = 4$) partitions. For this scenario, the partitions should map to actions as shown below:

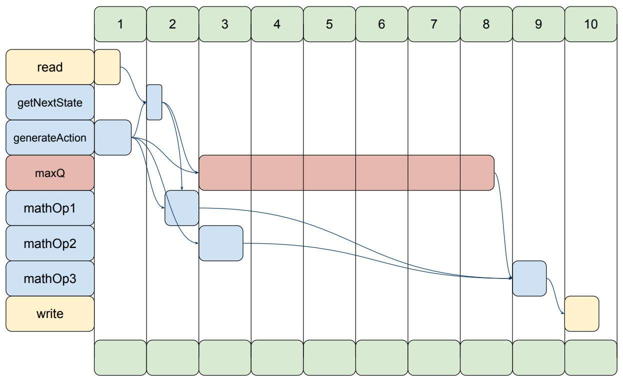


Fig. 5: Function graph for one episode, where $maxQ$-block proves to be the most critical datapath. Arrows represent data dependencies.

$\{i_p\}$	0	1	2	3
$\{i_a\}$	0	1	3	4

The skip in the value of i_a is because a_{max} should not be included from this random pick, and hence the mapping for $i_p \rightarrow i_a$ is simply as shown in Eq. (2).

$$i_a = \begin{cases} i_p & \text{if } i_p < a_{max} \\ i_p + 1 & \text{otherwise} \end{cases} \quad (2)$$

4. Look-Ahead Blocks After choosing the (s, a) pair, the algorithm needs to generate the $nextState$, which serves as a precursor to the $maxQ$ calculation. The calculation of the $nextState$ can be either from a heuristic or from a lookup based approach. Following the pipelined model described in Fig. 5, the lookup approach doesn't create any changes in the number of clock cycles required for an episode because of the following reasons:

1) the *maxQ* block is a sequential design and waits for a clock edge to begin the calculation
2) the *mathOp1* block is a combinational design and has enough available latency to be scheduled in the third clock cycle

Following the same pipeline mentioned in Fig. 5, the $maxQ$ operation uses the $nextState$ value to calculate the maximum Q-value based on Eq. (1). This operation also accounts for the most time consuming section, and as such has been given some privileges. The execution schedule in Fig. 5 is based on an implementation where we relax the $maxQ$-block's latency from the limitation of a single clock cycle. In other words, we force the $maxQ$-block to not be a purely combinational structure. Furthermore, we explicitly assume that all steps except the *maxQ* (in red) get completed within one clock cycle, and therefore, determine the clock frequency to satisfy these constraints. The algorithm makes many references to the Q-matrix during one iteration, most of which are requested by the $maxQ$ block, hence the Q-matrix is implemented using a two-port memory to reduce the latency further.

Since a typical scenario involves multiple episodes, often of the order of 10^5 or more, reducing the latency of one episode is a crucial key to achieving a significant drop in overall latency. One way of shortening this latency of the entire episode is to simply schedule the $maxQ$ operation as early as possible.

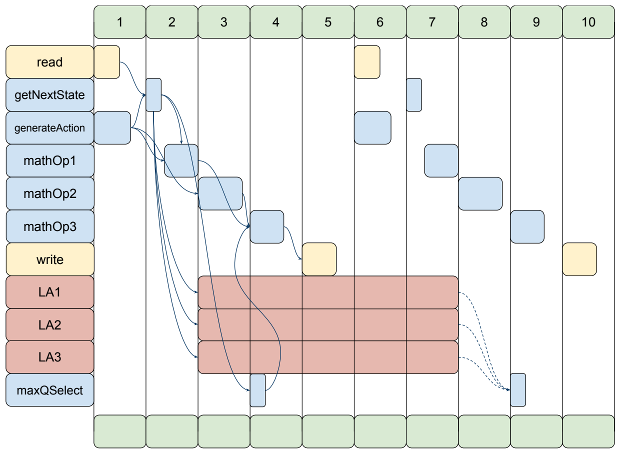


Fig. 6: Function graph for one episode with three Look-Ahead blocks assuming $Z = 3$. Solid arrows represent intra-loop data dependencies. Dashed arrows represent inter-loop dependencies. Dependencies for only one iteration are shown.

To do this, the architecture needs to *look ahead* and perform some calculations one iteration in advance, only one of which will be finally used. A look-ahead architecture is a massive block of Z parallel Look Ahead Block (LAB)s, with each LAB being an exact copy of a $maxQ$ operation. As described in Algorithm 2, the LABs buffer the $maxQ$ operation for all the next available states, only one of which is finally used. This leads to a function graph as shown in Fig. 6 and shows that the pipeline doesn't need to wait for the *getNextState* operation to schedule the calculation of $maxQ$ since all possible $maxQ$ values are being calculated irrespective of the action. Algorithm 2 differs from Algorithm 1 only in the sense of forwarding these $maxQ$ values, creating the extra steps of receiving $maxQ$ from the LABs and updating them with corresponding $nextState$ values to schedule further calculations. This approach increases the initiation interval for the first episode, but reduces the iteration interval to 5 cycles for subsequent episodes, generating a speed-up of two times when compared to the basic architecture shown in Fig. 5. The overheads of this design include:

1) A completely partitioned global buffer with a size of $Z \times (b_i + b_f)$ to solve the inter-loop dependency.
2) A local buffer for Z elements of the Q-matrix to remove multiple reads in the 6th clock cycle.

Henceforth, we refer to this micro-architecture as Epsilon Greedy with Look-Ahead (EGLA), which we employ as the core of ReLAccS.

B. Hardware Optimization

The lookup tables (LUTs) and the associated carry chains are the main computational units of any FPGA; therefore, an FPGA-based hardware accelerator should efficiently utilize these resources. The synthesis tools for FPGAs also primarily rely on these resources for the realization of different types of combinational and sequential circuits. In our proposed methodology, we have used the 6-input LUTs and the associated carry chains for a resource-efficient and energy-optimized implementation of the state-update equation described in Eq. (1).

As described in Section IV-A, we have used the parallelism offered by FPGAs to compute '$Q_{(\text{s+1,a})_{\max}}$' for the state-update equation efficiently. The state-update equation uses the '$Q_{(\text{s+1,a})_{\max}}$' value to compute the next state using three instances of the resource-intensive multiplication operation. The FPGA synthesis tools opt to use either DSP-blocks or the logic-based soft multipliers IPs for implementing these multipliers. However, the number of DSP-blocks is limited, and they have fixed locations on an FPGA. The limited stack of DSP-blocks may result in complete exhaustion of the DSP-blocks, and the fixed location of the DSP-blocks can result in degrading the overall performance of an accelerator [25], [26]. For example, the Q-Learning accelerator proposed in [19] consumes $\sim 94\%$ of the available LUTs on FPGA. Consequently, it is beneficial to have resource-efficient implementations of the state-update equation. As shown in Fig. 3, our proposed implementation of the state-update equation is equally utilized in both architectures of the hardware accelerator for reinforcement learning.

The total number of multiply operations in a single instance of the state-update equation can be reduced by analyzing the design time parameter 'α' in Eq. (1). Eq. (4) and Eq. (5) present two simplified variants of the state-update equation with only two multiplication operations in each. Depending upon the value of 'α', either of the equations can be used for computing the next state. Further, as the design-time parameters 'α' and 'γ' remain constant for a design, they can be used to implement Eq. (4) and Eq. (5) using *constant multipliers*. One of the inputs to a constant multiplier is a fixed constant number, and the other input is a variable. Compared to the generic multipliers, constant multipliers offer significant reductions in the overall resource utilization, critical path delay, and energy consumption.

$$For \quad \underline{Q_{(\text{s,a})} = a} \quad and \quad \underline{R_{(\text{s,a})} + \gamma \times Q_{(\text{s+1,a})_{\max}} = b} \tag{3}$$

$$Q_{(\text{s,a})} = \alpha(b - a) + a \tag{4}$$

$$Q_{(\text{s,a})} = (1 - \alpha)(a - b) + b \tag{5}$$

FPGA synthesis tools, such as Xilinx Vivado, provide both DSP-based and LUT-based constant multiplier implementations. However, in this work, we have utilized the 6-input LUTs and the associated carry chains to implement constant multipliers with better resources optimization than the multiplier IP provided by Vivado. Algorithm 3 presents our proposed technique for efficiently implementing constant multipliers. Our proposed method efficiently utilizes binary and ternary adders for computing the product. We configure the 6-input LUTs to implement these adders using the available fast carry chains in FPGAs. A ternary adder can add three operands simultaneously using the resources of a binary adder. Fig. 7 presents a Xilinx FPGA-based ternary adder implementation. The inputs to the LUTs are the bits of the shifted multiplicand. As described in Algorithm 3, to multiply an M-bit variable V with an N-bit constant C, we first identify the total number of $1's$, referred to as C_1, in the binary representation of C (line 1). To reduce the total number of shift and add operations

Algorithm 3 *Constant Multiplier using Shift Operation*

```
Require: An N-bit positive constant 'C' and an M-bit variable 'V'
 1: Identify the total number of 1's (C_1) and 0's (C_0) in the binary
    representation of C
 2: if C_0 < C_1 then
 3:     Compute $X$: $X = 2^N - C$
 4:     Identify the total number of 1's (X_1) in the binary represen-
        tation of X
 5:     if $X\_1 < \lceil \frac{C\_1}{3} \rceil$ then
 6:         Compute $C \times V = 2^N \times V - X \times V$
 7:     else
 8:         Compute $C \times V$ using C_1 times shift left and utilizing
            binary and ternary adders
 9:     end if
10: else
11:     Compute $C \times V$ using C_1 times shift left and utilizing binary
        and ternary adders
12: end if
```

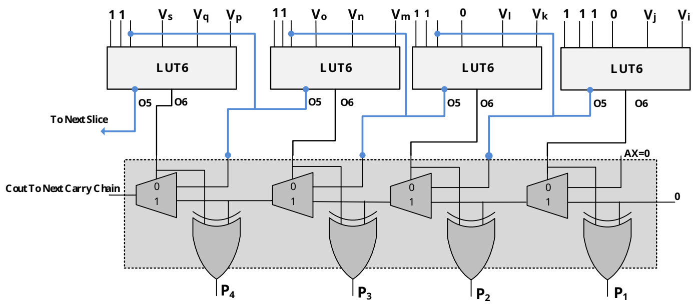


Fig. 7: Xilinx FPGA-based ternary adder

in the multiplication, we compare C_1 with $N - C\_1$ (line 2). For cases where $N - C\_1$ is smaller than C_1, we try to find the product C $\times$ V by shifting V left N times and then subtracting X-times shifted version of V from it (lines 3 – 6). For example, the 8-bit representation of constant 254 is '$0b11111110'$. The C_1 and C_0 are 7 and 1, respectively. In this case, we compute the parameter $X = 2^8 - 254 = 2$. Our proposed implementation initially computes $2^8 \times$ V (shift variable 'V' left by 8) and then subtracts $2 \times$ V (shift variable 'V' left by 1) from it. However, for all cases where C_1 is smaller than $N - C\_1$, we compute the product using the left shift and add operations (line 11). Similarly, for cases where the number of levels to compute product using ternary adders is smaller than the number of levels using subtraction (line 5), we compute product using left shift and add operations (line 8). For example, for constant $C = 249$ ('$0b11111001'$), parameter 'X' is 7('$0b00000111'$) and the number of 1's in the binary representation of 'X' ($X\_1 = 3$) is not less than the $\lceil \frac{C\_1}{3} \rceil$; therefore, the constant multiplier is instead implemented using binary *shift* and *add* operations. Moreover, the constant multipliers-based implementation of the state-update equation can considerably reduce the overall resource utilization when the constant parameters γ and α (or $1 - \alpha$) have values that are a direct power of 2. For such values of the constant parameters, the multiplication operations can be performed by a single shift operation.

C. Algorithm-level Approximation

A major limitation to implementing high-performance ac-

Algorithm 4 *ReLAccS*: Algorithm pseudo-code

```
Require: Q and R matrices
1: Generate sets $L_1, L_2$, and $M$ {Refer IV-C1}
2: Calculate set $V' = M - (L_1 \cap M)$
3: Calculate set $V$ {Refer Eqn 6 in IV-C2}
4: while $H \neq N$ do
5:     Prepare kernel data for iteration
6:     Run Accelerator according to Algorithm 1
7:     Fetch data from accelerator and update global $Q$-matrix
8: end while
```

celerators is the limited availability of resources. Different FPGA devices have varying amounts of LUTs, BRAMs, embedded DSPs, etc. This makes a design slightly specific to the device it was initially built upon. As explained in Section I, in an accelerator for RL, the resource utilization is a strong function of the product $N \times Z$, which is the number of total elements in the Q-matrix.

In [19], the authors show that only 132 states and 4 actions take up about 95% of the embedded multipliers and 94% of the LUTs when implemented on a Virtex-6 device. Typical use-cases often have demands for handling much larger Q-matrices. Complex autonomous robots, for example, need to make several decisions at once, out of which path planning is only a small subset. These decision scenarios are often modelled as MDPs, where each node represents a unique state and has as many child nodes as the actions available in that state. Given its inner representation of the world state, decision-making can simply translate to applying an existing policy, i.e., a function that maps every possible state to an action, but it can also involve simulations in order to predict the results of possible actions in the current situation [10].

Hence, we look at Q-Learning as a *graph*-problem, with each node and its child nodes representing a state (s_k) and the next possible (s_{k+1}) states respectively, with k being the k^{th} iteration in Algorithm 1. If we focus on the Bellman equation (Eq. (1)), we note that the updated value of $Q(s_k, a_k)$ is dependent only on values determined by the present (s_k, a_k) pair and the next possible $(s_{k+1}, a_{k+1_{max}})$ pairs. This means that for a given iteration, if the values $Q(s_k, a_k)$, $R(s_k, a_k)$, and $Q(s_{k+1}, a_{k+1_{max}})$ are available, the current value $Q(s_k, a_k)$ can be updated without any data hazards.

ReLAccS achieves the ability to scale itself using the inductive nature of the decision-making process. According to the Markov property, future states in a stochastic process do not depend on the previous states. This indirectly means that the present state is a sufficient statistic that gives us the same information for the future states as if we have the entire sequence of previously traversed states [27]. ReLAccS partitions the entire graph into smaller graphs with limited nodes (*subQ*), prepares relevant data for solving it as an independent MDP, and combines all partitions globally to achieve an analogous rendition of the Q-matrix that would have been achieved by solving the entire graph at once.

Hence, if the configurations of the sub-graph (*subQ*), the mapping of the all state-action pairs to their next states $Next(s_k, a_k) \rightarrow (s_{k+1})$, and a list of invalid states in the mapping are generated correctly, the application can update the entire graph in chunks. Henceforth, we abstract the accelerator, the configurations, the reward-functions, and any other related data required for one independent run of a sectioned MDP as a single entity, the *kernel*.

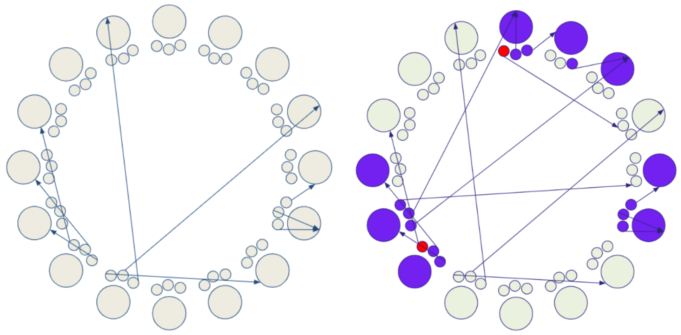

(a) Diversity in next-state mappings (b) Sectioning an arbitrary MDP

Fig. 8: For an MDP with all states having a fixed number of actions: (a) mappings can be ordered, sparse, or self-referenced. (b) The blue circles represent states loaded in a kernel. The corresponding blue dots represent valid actions and red dots indicate invalid actions.

The fact that the MDP is sectioned is abstracted from the underlying hardware. The sectioning of the MDP can be achieved within the hardware, but the clock-path-delay increases by a significant margin every time a new rule or a new mapping is implemented in hardware. This is where we leverage the multi-level coordination between the flexible processor subsystem and the deterministic programmable logic. We describe an arbitrary MDP in Fig. 8(a). Each larger circle represents a state s in the Q-matrix (a node in the MDP graph), and the smaller circles next to each state represent the possible actions available for the state (s). We can immediately note that the mappings for $(s_k, a_k) \rightarrow (s_{k+1}, a_{k+1})$ can either be ordered, sparse, or self-referenced. Examples for these three situations are listed below:

- *Ordered*: Sequentially deterministic systems. e.g.: A robot moving on the perimeter of a circle sectioned in N distinct positions.
- *Sparse*: Systems with variable parameters in the environment. e.g.: Typical navigation, control and automation problems, etc.
- *Self-referencing*: Any system with forbidden, or illegal states. e.g.: A robot trying to move into a boundary wall.

The $(s_k, a_k) \rightarrow (s_{k+1})$ mapping can be obtained through a lookup table. This lookup table is part of the data that is explicitly passed onto the kernel along with the custom $subQ$ and reward matrices. Since each $Next(s_n, a_z)$ element represents the next state it points to, the total utilization of this extra data is limited to $N \times Z \times b_s$, where b_s represents the number of bits allotted to represent a state. For an optimal implementation, $b_s = \log_2 N$. Following the sectioning of the MDP, it is possible to have many data hazards. We describe a simple data hazard in Fig. 9 and discuss the handling of such data hazards next.

1) Recognising Data Hazards

Let us assume that, at once, only H out of N states are loaded into the kernel. There can be multiple cases wherein

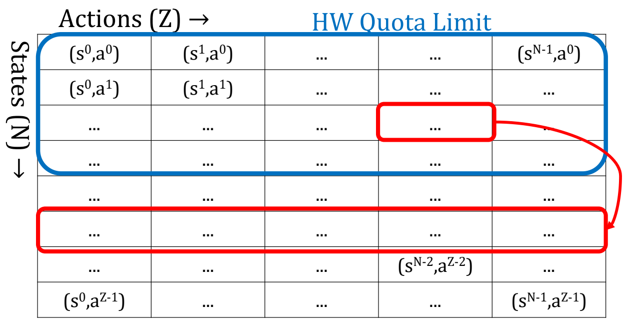


Fig. 9: Fault scenario for an arbitrary MDP. A (s_k, a_k) pair can be mapped to a state (s_{k+1}) that has not yet been loaded in the kernel. A kernel is defined by the hardware quota limit.

a (s_k, a_k) pair loaded in the kernel maps to a state (s_{k+1}) which is not yet loaded, as shown by the red arrow in Fig. 9. To find all such data hazards, three sets are populated:

1) $L_1 \rightarrow$ states to be loaded in kernel
2) $L_2 \rightarrow (s, a)$ pairs to be loaded in kernel
3) $M \rightarrow Next(p)\ \forall\ p \in L_2$

Next, we find the set V' of all invalid states, where $V' = M - (L_1 \cap M)$. The set V' describes all states that might experience a data hazard. Fig. 10 shows an example scenario, where the kernel can be loaded with the data for 5 states. In this scenario, any state-action pair mapping to the states 5 and 6 will result in data hazards. The figure shows the resulting values of L_1, L_2, M and V'.

2) *Removing Data Hazards*

After the matrices $Q, R, Next$ are ready, a new matrix V is formed, which has the dimensions $H \times Z$. Then, for every state in L_1, a valid action a_v is found. Intuitively, there must be at least one a_v for every state in L_1, because the selection of a state is justified only if a corresponding action is chosen. Then, *sequentially*, every element $v(s_n, a_z) \in V$ is chosen such that

$$v(s_n, a_z) = \begin{cases} a_z & \text{if } Next(s_n, a_z) \in L_1 \\ a_v & \text{if } Next(s_n, a_z) \notin L_1 \text{ and } z = 0 \\ v(s_n, a_{z-1}) & \text{otherwise} \end{cases} \tag{6}$$

The matrix V generated is passed to the kernel, and the EGLA micro-architecture employs it within the *actionGenerator* module. Here, the index z is used to show an arbitrary action. However, index v refers to its corresponding valid action. Eq. (6) thereby describes that if the next-state $Next(s_n, a_z)$ for a given (s, a) pair is contained in L_1, that specific action (z) will be stored in $v(s_n, a_z)$. If, however, the action a_z maps to an action which produces a next-state which is not contained in L_1, the last-known valid action a_v is substituted in its place. In the scenario shown in Fig. 10, the state-action pairs $(1, 0)$ and $(1, 1)$ are mapped to state 3 as that is the first valid mapping for state 1. Similarly, the state-action pair $(4, 2)$ is mapped to state 0 as it corresponds to the previous valid action for that state. These valid action assignments are made based on Eq. (6).

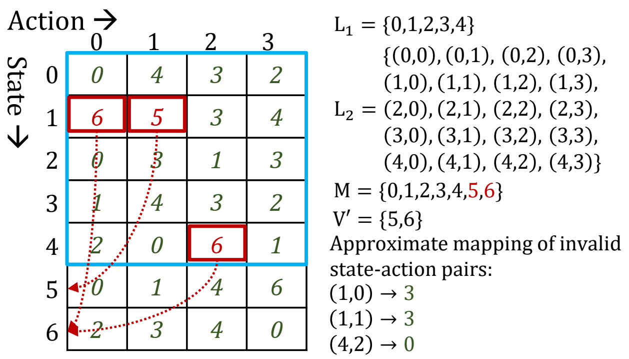

Action →
State →
$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 3 & 2 \\ 1 & 6 & 5 & 3 & 4 \\ 2 & 0 & 3 & 1 & 3 \\ 3 & 1 & 4 & 3 & 2 \\ 4 & 2 & 0 & 6 & 1 \\ 5 & 0 & 1 & 4 & 6 \\ 6 & 2 & 3 & 4 & 0 \end{array}$$
L_1 = {0,1,2,3,4}
L_2 = {(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3), (4,0), (4,1), (4,2), (4,3)}
M = {0,1,2,3,4,5,6}
V′ = {5,6}
Approximate mapping of invalid state-action pairs:
(1,0) → 3
(1,1) → 3
(4,2) → 0

Fig. 10: Sample example for recognizing and removing data hazards in the kernel.

3) *Extracting a Sub-Graph*

Generating a configured kernel has to be done by the embedded processor of the system, and we assume that the time for this configuration is masked by the time it takes for the previous kernel to be solved through the accelerator. At the outset, it looks like one process is scheduled on the software side, and one on the hardware side, as mentioned in the pseudo-code for ReLAccS in Algorithm 4. Since the ReLAccS algorithm is based on iterations of solving sections of the entire graph, it is imperative to choose the minimal number of unique kernels (U) that should be loaded. Without a fixed model of the system, it is difficult, and possibly computationally intensive to estimate U. Assuming that for a total of N states, only H of which can fit inside the kernel, there are ${}^{N}C_{H}$ possible combinations for selecting states. Since the mappings are heavily dependent on the application, it is onerous to generalise concrete heuristics for sectioning the graph in the most optimal way so as to achieve U kernels. Instead, we choose K kernels randomly, and make sure that the kernel loads every state available in the graph at least once in K kernel iterations. Many scenarios for RL applications like AI in robotics, navigation, and automation involve highly sparse MDPs [18], and we leverage this sparsity to justify the decision of random selection.

For an arbitrary MDP, it is not necessary for all parent nodes to have the same number of child nodes. Easily, there can be two states in which there are different number of possible actions available. However, solving a MDP this way creates a loss of symmetry and therefore a stress on the hardware to implement specific heuristics for specific states. To avert this problem, we consider a trade-off in resources by forcing all states to have an equal number of actions (Z), resulting in all parent nodes having an equal number of child nodes. We mark these *extra* mappings as illegal states, which translate to few *extra* self-referencing mappings that have rewards similar to that of an illegal state, thereby training the agent to not choose the corresponding action.

4) *Handling Data Hazards (with respect to the kernel)*

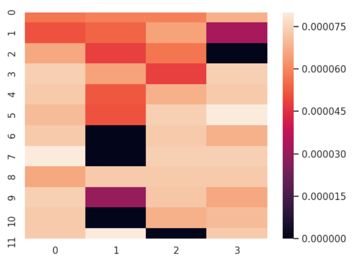
As compared to EGLA, the hardware architecture is slightly modified for ReLAccS. All actions are *double-checked* using matrix V, mentioned in Section IV-C2. Thus, the *actionGenerator* module creates an output that gets passed through matrix

V, where each generated action is validated for sanity. V is prepared in such a way that it re-maps illegal actions to legal ones, always assigning a suitable action for the algorithm to take (refer Eq. (6)). This means that updates aren't simply skipped if illegal actions are encountered and some value is actually updated in every iteration. Empirical results suggests that an extra lookup does not create a huge difference in the latency of the entire iteration and that even though the values of the final optimised Q-matrix are different, it does not impact the convergence.

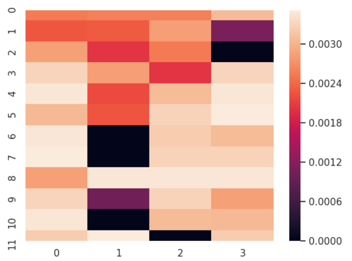
V. Experiments and Results

A. Experimental Setup

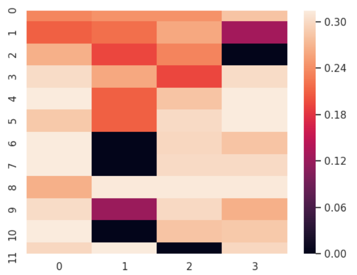
We have used C++ for the high-level implementation of all designs. For the FPGA-based implementation of the designs, we have utilized Verilog HDL, VHDL, and Xilinx Vivado Design Suite 18.3. For the calculation of the dynamic power of all implementations, Vivado Simulator and Power Analyzer tools have been utilized. All designs have been implemented on Xilinx Zynq UltraScale+ MPSoC (xczu3eg-sbva484-1-e device). We have compared our architectural implementations with the design of Da Silva et al. [19] and algorithmic implementations with a parallel implementation based on communication with cache as described in [28]. Further, the results of the proposed LUT-level hardware optimizations have been compared with various Vivado multiplier IP-based implementations. Each implementation of EGLA or ReLAccS is highly parameterised in terms of precision, number of states, actions, kernel-size, etc. The algorithm is converted to hardware-compatible data types and bit-widths depending on the configuration. For the testbench, the same algorithm is performed with double precision for all values and simulated for 10^8 iterations. The output from both the fixed-point hardware and the double precision software runs are compared for element-wise differences, which are stored as a heatmap. Henceforth, we refer to the output of the double precision implementation as the *golden output*. The notation $< b_i.b_f >$ is used all throughout to represent the number of bits for integers (b_i) and fractions (b_f) in the fixed point format. For most experiments, we model a robot moving in a 2D grid similar to that shown in Fig. 1. For the 2D grid problem, we vary N, the number of states, as a method of varying the problem complexity. This translates to varying the number of possible locations that the agent can reach in Fig. 1. For more generic problems, we change the $N \times Z$ value, where Z is the maximum number of possible actions from any arbitrary state, to change the complexity of the actual application. Since for the 2D grid model of a robot, the obstacles and goal states are clearly defined, we can dynamically generate a reward matrix and verify whether the Q-matrix is converging correctly or not. However, since the reward matrix is a heuristic function that can be implemented in the processor subsystem, the EGLA architecture can be synthesised for any Q-Learning application such as that in [4] involving arbitrary number of states and actions.



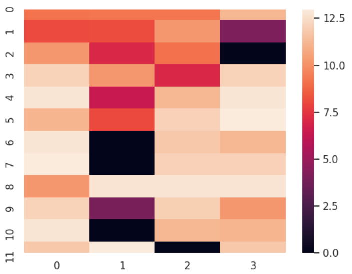
(a) $b_i = 10$ and $b_f = 24$



(b) $b_i = 10$ and $b_f = 18$



(c) $b_i = 10$ and $b_f = 12$



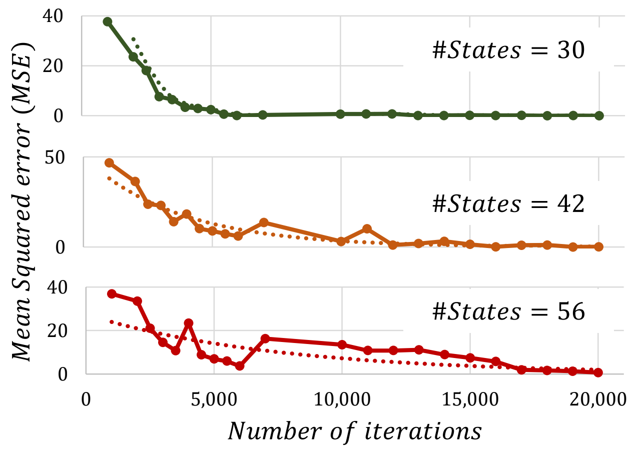
(d) $b_i = 10$ and $b_f = 6$

Fig. 11: Difference heatmaps for different fixed-point precision run for only 1000 iterations. Mean-Squared-Errors – (a) 7.25 (b) 6.84 (c) 9.21 (d) 26.15

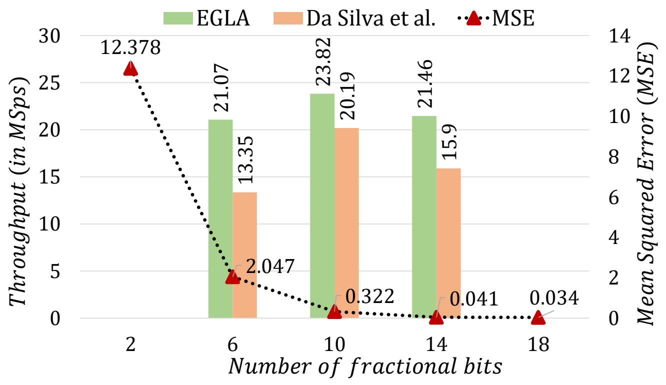
B. Results and Discussion

1) Architecture Design

1) *Effect of precision scaling:* To demonstrate the degradation of the Q-matrix with decreasing precision, the difference between the golden output and the hardware output are plotted as heatmaps in Fig. 11. All of these runs were done with only 10^3 episodes to highlight the degradation. As the available float-precision (b_f) decreases, the error increases. Still, the sparsity profile for these matrices are almost similar, showing the tolerance of the convergence given in-spite of the reducing precision. Even with the different error metrics, the final Q-matrix achieves similar distinctions between different actions for the same state, highlighting tolerance to approximations. If the iterations are increased, the error reduces and the average difference between the Q-matrix from hardware and golden outputs becomes a direct function of the precision of the fixed-point representation. Hence, for the same precision and number of iterations, an application with lower number of states is more probable to reach optimality. Alternatively, for the same precision and states, an application with higher number of iterations is more probable to reach optimality.
2) *Effect of increasing iterations:* For fixed values of states and precision, increasing the iterations increase the confidence for convergence of the Q-matrix. The mean-squared error is plotted against number of iterations in Fig. 12(a), which shows the error approaching zero for 30, 42, and 56 states, each implemented with 12 bits of precision. It can be seen that the average error between the Q-matrix for hardware and golden outputs gradually decreases and finally becomes a direct function of the precision of the fixed-point representation.
3) *Comparison with existing state of the art:* Fig. 12(b) depicts the throughputs for EGLA and [19] for different number of



(a) Variation of MSE with increasing number of iterations (along with trendlines)



(b) Variation of MSE with fractional bits (b_f)

Fig. 12: (a) Effect of increasing iterations on the Mean Squared Error (MSE). MSE decreases with increasing iterations. (b) Comparison of EGLA's throughput with Da Silva et al [19].

TABLE I

COMPARISON SCENARIOS FOR EGLA VS. DA SILVA ET AL'S DESIGN WITH DIFFERENT FRACTIONAL BITS OF PRECISION (b_f) AND THE NUMBER OF STATES (N)

States (N)	($b_i.b_f$)	Throughput (MSps)
132	10.6	13.35
30	10.10	20.19
56	10.14	15.9

fractional bits. We challenged EGLA with three of [19]'s fixed-point implementations for three different values of N as shown in TABLE I.

Marked in a dashed black line is the plot of mean-squared-error averaged for all elements against the number of fractional bits, each implemented for 56 states for 3×10^5 iterations. It can be seen that the error approaches zero as the precision is increased, and it can be inferred that there are strong relations between the number of states, the available precision, and the number of iterations. Using results from Fig. 12(a) and Fig. 12(b), we conclude that given the same wall-time, the pipelined EGLA micro-architecture consistently outperforms [19] as it is able to process much more iterations within the same time period, and is therefore more likely to converge.

We consider the implementation proposed by Da Silva et al. [19] to be the state-of-the-art in terms of optimised parallel architectures for Q-Learning and compare our architecture with the same parameters. Our accelerator achieves consistently better results in terms of maximum throughput measured in Mega Samples per second (MSps). It is important for us to highlight that this comparison is unfair in terms of resource utilization parameters because we implement our design on a xczu3eg-sbva484-1-e, while Da Silva et al. implement it on a Virtex-6 xc6vcx240t-1ff1156. Even though EGLA includes the ϵ-greedy action choosing strategy as an overhead, empirical results establish that for the same parameters, EGLA consistently performs better in terms of throughput even with this overhead while using much less resources.

For comparison, we choose $b_i = 10$ and $b_f = 14$ primarily because this is the highest precision [19] reports metrics for. We extend our comparisons for higher precision and show that EGLA achieves more throughput for $b_i = 10$ and $b_f = 18$ as compared to [19]'s $b_i = 10$ and $b_f = 14$. This directly means that EGLA guarantees higher probability of convergence and smaller error given the same wall time. A detailed analysis is shown in Table II, where the utilization of LUTs, registers, DSPs or Embedded Multipliers (EMs) and throughput are compared. Da Silva et al. (10.14) reaches 94% of LUT utilization with 142778 LUTs at $N \times Z = 528$, while EGLA (10.12) reaches just over 100% LUT utilization[2] with 70960 LUTs at $N \times Z = 2112$. At this point, 31.4% of these LUTs are being implemented as memory because no more BRAM slices are available. However, we implement a design successfully for (10.6) with a utilization of 67% LUTs and a 22.7% increase in throughput. An estimate of the clock frequency for the throughput values mentioned in TABLE II can be obtained by Eq. (7). This estimate is based on the extrapolated data obtained with a simulation for 10^8 iterations.

$$f_{clk}(\text{MHz}) = 11 \times \text{Throughput} \tag{7}$$

In another related work, Spanò et al. [12] report maximum clock frequencies for different implementations of their proposed methodology. However, they do not report the application's latency and initiation interval (both in terms of the number of clock cycles) for any number of iterations, which do not allow sufficient information for comparing the throughput of the application. Further, as demonstrated by our current work and that proposed by Da Silva et al. [19], the resource utilization and the throughput depend on the number and pattern of bits used to represent the data (Q-Matrix). However, Spanò et al. have not discussed their representation of fixed-point numbers while reporting their performance results. Nevertheless, for completeness of the results, we report the comparison in TABLE III with the top results quoted in [12]. For the simplest configuration ($N \times Z = 8 \times 4$ and 8-bit precision), we report a maximum clock frequency of 1020 MHz compared to 222 MHz reported in [12]. Similarly, for the most complex configuration ($N \times Z = 256 \times 16$ and 32-bit precision), we achieve a maximum clock frequency

[2]Since this design marks the upper limit of configurations that can be implemented, in this case the throughput is estimated based on the critical path delay reported by Vivado for the hardware synthesis.

TABLE II

COMPARISON OF THE PROPOSED ILP-BASED EGLA ARCHITECTURE WITH THE DESIGN PROPOSED BY DA SILVA, ET AL. [19].

$N \times Z$	Design	LUTs	Registers	DSP/EM	Throughput (in MSps)
48	[19] (10.6)	3387	1029	58	22.27
	EGLA (10.6)	1358	1330	2	**23.61**
120	[19] (10.14)	12379	3670	250	16.57
	EGLA (10.14)	3809	2067	3	**21.55**
	EGLA (10.18)	4681	2364	3	**17.65**
224	[19] (10.14)	23117	6792	490	15.90
	EGLA (10.14)	6216	3344	3	**21.46**
	EGLA (10.18)	7814	3872	3	**17.64**
528	[19] (10.14)	142778	16395	730	12.23
	EGLA (10.14)	14324	7117	3	**16.77**
	EGLA (10.18)	16395	8226	3	**15.79**
960	[19]	-	-	-	-
	EGLA (10.6)	17529	9168	3	17.02
	EGLA (10.12)	27451	17779	3	13.70
2112	[19]	-	-	-	-
	EGLA (10.6)	47805	29687	3	16.19
	EGLA (10.12)	70960	49763	3	13.19

TABLE III

COMPARISON OF EGLA WITH THE DESIGN PROPOSED BY SPANÒ, ET AL. [12]

$N \times Z$, Precision	Design	LUTs	LUTRAM	FF	CLK (MHz)	POWER (mW)
8×4,	[12]	193	32	154	222	37
8−bits	EGLA	80	0	195	**1020**	13
256×16,	[12]	4017	2560	1210	93	611
32−bits	EGLA	10790	2624	3412	**238**	477

of 238 MHz compared to 93 MHz reported in [12]. Further, as shown in the table, our implementations are more power-efficient than those presented in [12]. The power dissipation metrics have been obtained with the corresponding maximum clock frequencies shown in TABLE III. The higher resource utilization of EGLA is due to the additional resources required to implement the look-ahead logic.

2) Hardware Design

We have implemented our proposed technique of constant multipliers-based state-update equation for different values of α and γ. To show the efficacy of our implementation, we have compared our technique with the state-update equations using a generic variable multiplier, Vivado constant multiplier (Const. Mult), Vivado constant multiplier IP, and a state-of-the-art constant multiplier referred to as *FloPoCo* [29]. For the Vivado constant multiplier, the constant values of the parameters have been fixed in the HDL code of the multiplier. This design allows the synthesis tool to optimize the multiplier implementation with respect to the rest of the hardware. The *FloPoCo* design is also based on the utilization of *bit shifts* and *addition/subtraction* operation to implement a constant multiplier. However, as described previously in subsection IV-B, we use 6-input LUT-level optimizations to implement our proposed multiplier. Fig. 13 presents the total number of utilized LUT and power-delay product (PDP) of different implementations for three different random values of α and γ. These results have been normalized to the results obtained by implementing the state-update equation using a variable multiplier. The precisions of $Q_{(\text{s,a})}$, $R_{(\text{s,a})}$, and $Q_{(\text{s+1,a})_{\max}}$ are kept at 16-bit (10.6 Fixed-point format) for these experiments. Further, to represent the fractional values of α and γ in the range of $[0, 1]$, we have used a 1.10 fixed-point format (1 bit reserved for integer and 10 bits for denoting the precision). For example, $\alpha = \gamma = 0.875$ will be represented by constant $0.875 \times 2^{10} = 896$. All four designs ('Ours', 'Const. Mult', 'Vivado IP', and 'FloPoCo') offer reductions in the total utilized LUTs and energy consumption. Further, for all values of α and γ, our proposed constant multiplier implementation-based designs provide significant reductions in the utilized LUTs and PDP for the state-update equation. For example, compared to the Vivado IP-based implementation for $\alpha = \gamma = 704$, our proposed technique offers $\sim 27\%$ and $\sim 48\%$ reductions in the total utilized LUTs and PDP, respectively. Though the results are shown for 16-bit precision of variables and three different values of α and γ, we have observed similar reductions in the resources and energy consumption of the state-update equation for other values of α, γ, and precision of variables. The results also show a notable decrease in the LUTs utilization and energy consumption of the state-update equation for design parameters α and γ having values that are a direct power of 2.

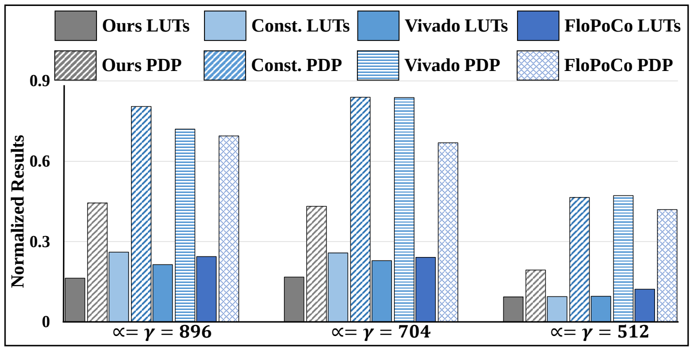


Fig. 13: Implementation results of constant multipliers-based state-update equation. Values are normalized to the corresponding results of state-update equation using variable multiplier (LUTs=729, PDP=0.256 nJ).

TABLE IV

SPEED-UPS OBTAINED USING DIFFERENT KERNEL SIZES WITH THE RELACCS ARCHITECTURE (NOTATION: RLX($N \times Z$)) WHEN COMPARED TO AN EXTRAPOLATED CACHE-LEVEL IMPLEMENTATION OF Q-LEARNING.

$N \times Z$	EGLA	RLX(64)	RLX(144)	RLX(256)	RLX(576)
528	1192	153	319.1	343.3	1192
840	977	32.9	141.7	228.8	241.46
2112	842	12.6	51.3	85.83	241.46
7200	–	8.97	36.22	65.2	160.97

3) *Algorithm Design*

To evaluate ReLAccS, we choose different values of $N \times Z$ and try to implement them with EGLA architecture and four different kernel-size configurations of ReLAccS, namely 64, 144, 256, and 576, each with a precision of (10.12). This provides us a measure of the resource overheads for supporting the ReLAccS platform, and also gives an estimate of the maximum speed-up that can be achieved along with trends of power and resource usage. We estimate the speed-up compared with the cache-level implementation of Q-Learning in [28]. There is sufficient reason to believe that it is difficult to store more than a limited amount of Q-values in the cache. But, even though [28] reports data only for a maximum of $N \times Z = 528$, we extrapolate this behaviour for higher values of $N \times Z$, *assuming* that there will be enough resources in a hypothetical system to store all the Q-values at once. We challenge ReLAccS against the fastest reported scenario of 4-processors with an estimated throughput of 17,229 samples per second, and report various implementations of the ReLAccS architecture in TABLE IV. The smallest implemented kernel ReLAccS(64) design shows a 153× speedup while consuming only 19.24 mW. The speed-up decreases as the kernel size is made smaller. This is obvious as decreasing the kernel size increases U, which consequently means more kernels need to be prepared to process the entire Q-table. The throughput for one kernel remains similar to the corresponding EGLA design. To understand the trade-off in resources and power utilization, ReLAccS architectures and EGLA(840) were compared mutually for $N \times Z = 840$ with a precision of (10.6). Fig. 14 shows the power, LUTs, registers, and BRAMs used in each design relative to a ReLAccS(64) design. The power metrics for all designs are reported at 100 MHz. It must be noted that most software-only implementations do not provide sufficient information for concrete comparisons. For example, the only metrics we can compare with are the number of iterations and the achieved throughput. Most software works do not mention that. However, for completeness, we compared the performance of ReLAccS with that presented in [15] by using the performance reported in [28] as the baseline. Camelo, et al. [15] report an average speed-up of 8.09× over [28]. Further, they achieve around 8.014× speed-up for a single-agent system. These speed-up values are much lower than almost all the values shown in TABLE IV.

For all the listed ReLAccS implementations, errors were calculated to get an estimate of the convergence. The relative averaged RMS error was considered to be a good metric here, as our focus is on achievement of optimality, and not the exact values themselves. We find this error by calculating the RMS error per element between the hardware output (simulated for 10^4 iterations) and the golden output (simulated for 10^8 iterations with double precision), averaged with the total number of elements present, relative to the mean of the golden output. For all implementations, this error was found to be about 5∼6%, which is a very satisfactory result considering only 6 fractional bits of precision.

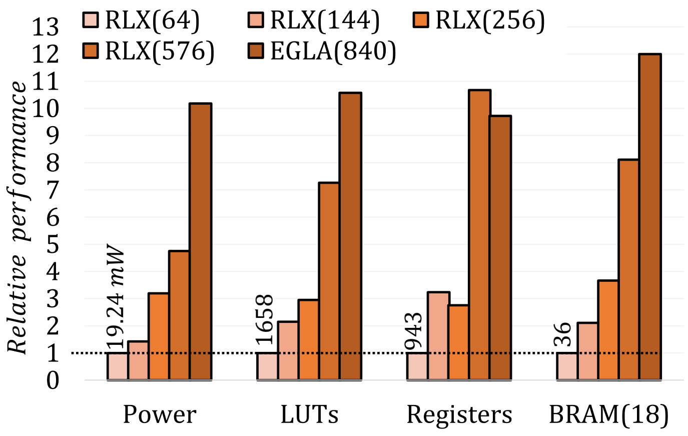


Fig. 14: Resource and Power comparisons for 6-bit implementations of ReLAccS architectures with different kernel sizes and the EGLA architecture relative for $N \times Z = 840$ and clock frequency of 100 MHz. All metrics are relative to that of RLX(64). Absolute metrics for RLX(64) are also shown.

VI. Conclusion

The accelerator designs for machine learning applications, with high computation and memory requirements, cannot be limited to one level of the computation stack. In this article, we propose a multi-level approach to designing accelerators for RL. The approach presented in this article results in considerable performance improvements and multiple implementation choices compared to state-of-the-art design approaches. The *multi-level* aspect stems from the proposed contributions at multiple abstraction levels—*algorithm*, *micro-architecture* and *FPGA-hardware*.

At the architecture level, the proposed look-ahead operation results in achieving up to 34% increase in the throughput over that reported in [19]. Similarly, the LUT-level optimizations and the resulting custom operations provide up to 27% and 48% reductions in resource utilization and energy consumption respectively, compared to Vivado's optimized designs. Finally, we presented algorithm-level approximation technique to reduce the resource utilization by up to 12× and the power dissipation by up to 10× compared to the high-throughput design, albeit with marginal degradation in the quality of results. Such approximations can enable the usage of RL in resource constrained embedded system.

The methods presented in this article are orthogonal to some related techniques and can be used to complement them. In such a scenario, given the constraints of an application, finding the optimal set of configuration for the application presents an interesting problem for related future research.

References

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

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[4] S. S. Sahoo, B. Veeravalli, and A. Kumar, “A hybrid agent-based design methodology for dynamic cross-layer reliability in heterogeneous embedded systems,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[5] B. Chatterjee, N. Cao, A. Raychowdhury, and S. Sen, “Context-aware intelligence in resource-constrained iot nodes: Opportunities and challenges,” *IEEE Design Test*, vol. 36, no. 2, pp. 7–40, 2019.

[6] J. J. Q. Yu, W. Yu, and J. Gu, “Online Vehicle Rou.ting With Neural Combinatorial Optimization and Deep Reinforcement Learning,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3806–3817, 2019.

[7] Z. Yan and Y. Xu, “Data-Driven Load Frequency Control for Stochastic Power Systems: A Deep Reinforcement Learning Method With Continuous Action Search,” *IEEE Transactions on Power Systems*, vol. 34, no. 2, pp. 1653–1656, 2018.

[8] A. Das, R. A. Shafik, G. V. Merrett, B. M. Al-Hashimi, A. Kumar, and B. Veeravalli, “Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems,” *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[9] K. Chui, W. Alhalabi, S. Pang, P. Pablos, R. Liu, and M. Zhao, “Disease diagnosis in smart healthcare: Innovation, technologies and applications,” *Sustainability*, vol. 9, p. 2309, Dec 2017.

[10] L. Hofer, *Decision-making algorithms for autonomous robots*. Theses, Université de Bordeaux, Nov. 2017.

[11] P. R. Gankidi, “FPGA Accelerator Architecture for Q-learning and its Applications in Space Exploration Rovers,” Master’s thesis, 2016.

[12] S. Spanò, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, “An efficient hardware implementation of reinforcement learning: The q-learning algorithm,” *IEEE Access*, vol. 7, pp. 186340–186351, 2019.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013.

[14] H. Wicaksono, “Q learning behavior on autonomous navigation of physical robot,” in *2011 8th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pp. 50–54, 2011.

[15] M. Camelo, J. Famaey, and S. Latré, “A scalable parallel q-learning algorithm for resource constrained decentralized computing environments,” in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pp. 27–35, 2016.

[16] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, “Massively parallel methods for deep reinforcement learning,” *CoRR*, vol. abs/1507.04296, 2015.

[17] A. Clemente, H. Castejón, and A. Chandra, “Efficient parallel methods for deep reinforcement learning,” 05 2017.

[18] A. Sapio, S. S. Bhattacharyya, and M. Wolf, “Efficient Solving of Markov Decision Processes on GPUs Using Parallelized Sparse Matrices,” *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, vol. 00, pp. 13–18, 2018.

[19] L. M. D. D. Silva, M. F. Torquato, and M. A. C. Fernandes, “Parallel Implementation of Reinforcement Learning Q-Learning Technique for FPGA,” *IEEE Access*, vol. 7, pp. 2782–2798, 2019.

[20] R. N. Anderson, A. Boulanger, W. B. Powell, and W. Scott, “Adaptive stochastic control for the smart grid,” *Proceedings of the IEEE*, vol. 99, no. 6, pp. 1098–1115, 2011.

[21] C. Daniel, O. Kroemer, M. Viering, J. Metz, and J. Peters, “Active reward learning with a novel acquisition function,” *Autonomous Robots (AuRo)*, January 2015.

[22] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig, “Xilinx Zynq-7000 EPP: An extensible processing platform family,” in *2011 IEEE Hot Chips 23 Symposium (HCS)*, pp. 1–24, IEEE, 2011.

[23] LogiCORE, IP, “AXI Interconnect (v2.1),” 2017.

[24] Xilinx, “Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators,” 1996.

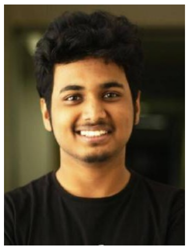
[25] S. Ullah, S. S. Murthy, and A. Kumar, “SMApproxlib: Library of FPGA-based approximate multipliers,” in *Proceedings of the 55th Annual Design Automation Conference*, DAC ’18, (New York, NY, USA), Association for Computing Machinery, 2018.

[26] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, “Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators,” in *Proceedings of the 55th Annual Design Automation Conference*, DAC ’18, (New York, NY, USA), Association for Computing Machinery, 2018.

[27] A. A. Mark1ov, *Theory of algorithms*. Moscow: Academy of Sciences of the USSR, 1954.

[28] A. M. Printista, M. L. Errecalde, and C. I. Montoya, “A parallel implementation of q-learning based on communication with cache,” *J. Comput. Sci. Technol., vol. 1, no. 6, p. 11*, 2002.

[29] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, “Optimal single constant multiplication using ternary adders,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 7, pp. 928–932, 2018.



Akhil Raj Baranwal received the bachelor’s (B.E.) degree from the Birla Institute of Technology and Science Pilani, Hyderabad, India, in 2020.

He joined the Chair for Processor Design at TU Dresden, Dresden, Germany as a Guest Researcher in January 2020, wherein he worked on exploiting FPGAs for deep reinforcement-learning based systems. His interests include Systems for ML and Accelerated Intelligence & Security.

Salim Ullah is a Ph.D. student at the Chair for Processor Design, Technische Universität Dresden. He has completed his BSc and MSc in Computer Systems Engineering from the University of Engineering and Technology Peshawar, Pakistan. His current research interests include the Design of Approximate Arithmetic Units, Approximate Caches, and Hardware Accelerators for Deep Neural Networks.

Siva Satyendra Sahoo is currently working as a Postdoctoral Researcher with the Chair for Processor Design at TU Dresden. He received his doctoral degree (Ph.D., 2015-2019) in the field of reliability in heterogeneous embedded systems from the National University of Singapore, Singapore. He completed his masters (M.Tech, 2010-2012) from the Indian Institute of Science, Bangalore in the specialization Electronics Design Technology. He has also worked with Intel India, Bangalore in the domain of Physical Design. His research interests include Embedded Systems, Machine Learning, Approximate Computing, Reconfigurable Computing, Reliability-aware Computing Systems, and System-level Design.

Akash Kumar (SM’13) received the joint Ph.D. degree in electrical engineering and embedded systems from the Eindhoven University of Technology, Eindhoven, The Netherlands, and the National University of Singapore (NUS), Singapore, in 2009. From 2009 to 2015, he was with NUS. He is currently a Professor with Technische Universität Dresden, Dresden, Germany, where he is directing the Chair for Processor Design. His current research interests include the Design, Analysis, and Resource Management of Low-Power and Fault-Tolerant Embedded Multiprocessor Systems.