QTAccel: A Generic FPGA based Design for Q-Table based Reinforcement Learning Accelerators

Yuan Meng*, Sanmukh Kuppannagari*, Rachit Rajat*, Ajitesh Srivastava*, Rajgopal Kannan†, Viktor Prasanna*
*Department of Electrical and Computer Engineering, University of Southern California
†US Army Research Lab-West
Email: *{ymeng643, kuppanna, rrajat, ajiteshs, prasanna}@usc.edu, †rajgopak@usc.edu

***Abstract*—Q-Table based Reinforcement Learning (QRL) is a class of widely used algorithms in AI that work by successively improving the estimates of Q-values – quality of state-action pairs, stored in a table. They significantly outperform Neural Network based techniques when the state space is tractable. Fast learning for AI applications in several domains (such as robotics), with tractable 'mid-sized' Q-tables, still necessitates performing a large number of rapid updates. State-of-the-art FPGA implementations of QRL do not scale along with the Q-Table state space, thus they are not efficient for such applications. In this work, we develop a novel FPGA based design of QRL and SARSA (State Action Reward State Action), scalable to large state spaces and thereby facilitating a large class of AI applications. Our architecture provides higher throughput while using significantly fewer on-chip resources, is capable of supporting a variety of action selection policies that covers Q-Learning and variations of bandit algorithms, and can be easily extended for multi-agent Q learning. Our pipelined implementation fully handles the dependencies between consecutive updates allowing it to process one sample every clock cycle. We evaluate our architecture for Q-Learning and SARSA algorithms and show that our designs achieve a high throughput of up to 180 million samples per second.**

Acknowledgement: This work was supported by the U.S. National Science Foundation under grants CNS-1643351 and OAC-1911229.

I. Introduction

Reinforcement Learning (RL) is a Machine Learning technique that governs the interactions of a goal-directed agent interacting with an uncertain environment [1]. More formally, a RL agent senses the environment to determine the current state and chooses state dependent action that in the long run will maximize the cumulative rewards. Reinforcement Learning has found widespread success in a plethora of applications including robotics, games (Go, Atari, etc.), computer vision, healthcare and several others [2].

Q-Table based Reinforcement Learning (QRL) algorithms are classic algorithms for learning agent behavior [3]. QRL works by successively improving the agent's evaluation of the "quality" of taking an action in a state – Q value for the state-action pair. The Q table stores the values for all possible state-action pairs.

Extensive research has been performed on accelerating Deep Neural Network based Q learning algorithms (also known as Deep Reinforcement Learning (DRL) [4]–[6]). DRL gained attention due to its ability to tractably learn over very large state spaces (greater than tens of millions). However, this has led to a lack of research in accelerating classic QRL. QRL acceleration merits attention as it can significantly outperform DRL for medium sized state spaces (hundreds of thousands to a few million state-action pairs that can fit on the on-chip memory of the target platform). Specifically, (i) QRL provides theoretical guarantee with respect to convergence to optimality, and (ii) the update step for QRL, unlike DRL, does not require the complexity of using neural networks [7, Ch. 7, p. 207-251] which require backpropagation. Hence, for applications such as robotics, accelerating QRL is expected to result in better performance.

FPGAs have emerged as a platform of choice for applications requiring fine-grained parallelism and energy-efficiency [8]. This makes them a suitable platform for accelerating QRL which are sequential in nature (Section III-B). Accelerators can achieve high throughput by using deep pipelined architectures to exploit parallelism within each update step. State-of-the-art FPGA devices [9], [10] provide abundant user-controllable on-chip memory resources (up to 500 Mb) allowing support for medium sized Q tables.

In this work, we significantly improve upon the state-of-the-art FPGA implementation for Q-Learning [11] by developing an architecture scalable to large state spaces. We highly optimize the number of required multipliers to a small constant as opposed to being proportional to the size of the state space in [11]. Furthermore, we generalize the architecture to support arbitrary action selection policies [12] and develop the first known FPGA implementation of SARSA algorithm [1]. An abstract-only version of this paper was published in the proceedings of ACM FPGA 2020 [13]. Our specific contributions are as follows:

- We develop QTAccel: a generic pipelined FPGA architecture for QRL. Our pipelined architecture handles all dependencies between consecutive updates and processes one sample in every clock cycle.
- We show the generality of our architecture by implementing two Q Table based algorithms which differ in action selection policies: Q-Learning (greedy action selection) and SARSA ($\epsilon-$greedy action selection [14]).
- Our design increases the limit on the on-chip Q-Table size compared to the design in [11] by more than $1000\times$ for a similar sized device by reducing the number of required multipliers to a small constant. This also enables launching parallel pipelines to solve multi-agent Q learning problems.

- We discuss how our architecture can be customized for Multi-armed Bandit (MAB) [15] which are critical to next generation 5G wireless networks [16]. Energy-efficiency is a significant requirement for such problems [16]. Thus, QTAccel provides a pathway for energy-efficient high-throughput FPGA implementations for the same.
- Using experimental evaluations we show that our implementations achieve a high throughput of 180 million samples/s.

II. Related Work

Extensive research has been performed on accelerating DRL. Parallelization techniques for DRL implementations targeting cloud as well as single machines with multi-core and GPUs have been developed [4], [17]–[19]. FPGA accelerators have also been developed for DRL [6], [12], [20]. Such implementations enable tractable learning of agents in extremely large state spaces of size greater than tens of millions. However, for medium sized state spaces ranging in several hundreds to a few millions, significantly better performance can be expected from QRL due to the simplicity of the update step compared to the backpropagation updated step of DRL.

Limited research has been done on accelerating QRL. Authors in [11] develop an accelerator for Q Learning. The limitation of their design is that the on-chip resource required is proportional to the number of state-action pairs. This limits the scalability of the design. A parallel publication [21] describes an optimized architecture that saves significant resource compared to [11], which makes use of a comparator tree that utilizes LUTs proportional to the state-action size. In [22], the authors develop a FPGA implementation for SARSA customized to the task of dynamic power management. However, the design works with just one state space and four actions. Thus, technically it is a stateless multi-arm band (MAB) [16] implementation as opposed to a generic SARSA implementation.

III. Background

A. Reinforcement Learning (RL)

Reinforcement Learning (RL) is an area of machine learning concerned with how an agent in an environment takes actions so as to maximize the reward [1]. A RL problem involves an *agent*, which is the learner and decision maker, taking some action in an *environment* and observe its new *state* and the *reward* for taking the action. The *environment* and *agent* interact in discrete time steps. The *agent* receives some representation of the *environment's state* $S_t \in S$ at each time step t. S is one of the possible *states* for the *environment*. The *agent* takes an action A_t where $A_t \in A(S_t)$ is one of the possible *actions* in *state* S_t. The *agent* then receives a *reward* R_{t+1} for taking the *action* A_t in state S_t. The RL algorithm then calculates the *quality function* for the state-action pair. This process is run over multiple episodes of training each lasting for several time steps until some convergence criteria. The *quality function* provides information regarding the optimal action to take in each state.

B. Q-Table Based Algorithms

Q-Learning [1], [23] and SARSA (State-Action-Reward-State-Action) [1], [24] are classical reinforcement learning algorithms which use Q-tables for learning. Q-table stores the "quality" of each state-action pair.

Q-Learning is a model-free and off-policy reinforcement learning algorithm. This means that the learning is based on trial-and-error and that the training is not done based on the current policy but uses some "off-policy" [1] approach like greedy or random selection. The algorithm includes a Q-value (quality value) for each state-action pair. The update formula for Q-Learning is:

$$\begin{aligned} &Q_{t+1}(S_t, A_t) = \\ &Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{1}$$

Here $Q(S_t, A_t)$ is the Q-value for the current state-action pair. R_{t+1} is the reward for taking action A_t on state S_t. $\max_a Q(S_{t+1}, a)$ is the Q-value of the next state-action pair which gives the maximum reward. α is the learning rate which is simply a measure of how much to take the newer value as compared to the old value. γ is the decay rate, which is used to make sure that future rewards are given less preference than current reward. The algorithm involves starting from any one random state S_t and choosing an action A_t from state S using some policy (can be random selection, greedy or ϵ-greedy [14]). After taking the action from the state the algorithm observes the reward R_{t+1} and the next state S_{t+1}. It then updates the Q-value using the formula given above. Finally, the next state S_{t+1} becomes the current state and the algorithm continues till the final/goal stage is reached. The algorithm is repeated multiple times for convergence. **SARSA** [1], [24] (State-Action-Reward-State-Action) is a model-free and on-policy reinforcement learning algorithm. In an on-policy learning process the training of agent is based on a specific policy and hence promotes "exploration" instead of "exploitation" [25]. The update formula for SARSA is [1]:

$$\begin{aligned} &Q_{t+1}(S_t, A_t) = \\ &Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \end{aligned} \tag{2}$$

Various action selection policies have been studied in the literature such as:

- $\epsilon-$Greedy [14] Action policy: The action with highest Q-value is chosen greedily with probability of $1-\epsilon$, and other actions are chosen with a probability $\frac{\epsilon}{|A|}$ where ϵ determines the exploration/exploitation nature of the agent
- Boltzmann Action policy [26]: An action is chosen with a probability proportional to $exp(Q(s,a)/T)$

In our discussion, we focus on using $\epsilon-$greedy.

C. Q-Table vs DQN based Q-Learning

Q-Learning involves updating Q-values for each state-action pair. In traditional table based approach all the Q-values for all the state-action pair need to be stored in a table. This requires huge tables when the number of state-action pairs is in the

order of billions or more. To remove this requirement, some approximation function [7, Ch. 7, p. 207-251], [27] can be used instead to calculate the Q-value for the required state-action pair. The most common approximation functions are neural networks like multi layer perceptron. For large state space using approximation function is desirable. But for small and medium sized state space (a few million) approximation methods pose several challenges. Firstly, approximation functions do not guarantee convergence. Secondly, they lead to complex architectures. Multilayer perceptron based Q-learning architecture can require 15 clocks per Q-value calculation using fixed point operations or as high as 600+ using floating point operations [12]. This is extremely slow compared to table based approach where an efficient pipelined architecture can perform a Q-value calculation every clock cycle (Section VI-D). This makes table based Q-learning quite effective for edge application like robotics, where medium sized state space is required.

IV. QTAccel: A Generic Architecture for QRL

In this section, we present QTAccel: our generic pipelined architecture for QRL.

A. Device Model

We implement our architecture design on a FPGA. The starting state, learning rate and discount factor are user inputs whose values are stored in registers. Q values associated with each state-action pair and rewards observed by the learning agent are stored in on-chip memory. The transition to next state from current state-action pair is implemented as combinational logic and LUTs. The action selector used to generate random actions is implemented using linear feedback shift registers (LFSR).

B. Architecture Details

A QRL algorithm executes the following steps until convergence: (i) Start from any random state S_t. (ii) Select a state dependent action A_t based on the behavior policy. (iii) Determine the next state S_{t+1}. (iv) Read the Q-value and the reward for the current state-action pair $Q(S_t, A_t)$ and R_{t+1}. (v) For S_{t+1} select an action A_{t+1} based on the update policy. (vi) Read the Q-value for the new state-action pair $Q(S_{t+1}, A_{t+1})$ from the Q table. (vii) Compute the updated Q-value - $Q_{t+1}(S_t, A_t)$ for the original state action pair S_t, A_t. (viii) Select S_{t+1} as the current state for the next iteration and write the new Q-value back into the table $Q_{t+1}(S_t, A_t)$.

To accelerate QRL algorithm we use the following resources: (i) 2 $|S| * |A|$ sized tables implemented in internal memory (BRAM), to store the Q values and reward values for all state-action pairs. Another equally sized table is needed to store the probability distribution of all state-action pairs for RL algorithms which rely on stochastic distribution for action selection, hence in that case 3 $|S| * |A|$ sized tables would be required. (ii) A behavior policy based action generation module for selecting the action for the current state. (iii) An update policy based action selection module which chooses the action for the next state S_{t+1}. (iv) A transition function, which takes the inputs (S_t, A_t) and returns the next state S_{t+1}.

We propose a generic 4 stage pipelined architecture for QRL accelerators. We start with empty Q-table and a reward table. Figure 1 shows the complete pipelined architecture. The following are the operations in the 4 stages of pipeline:

1) **First stage:** If this is the first iteration of the episode the start/current state S_t is selected randomly, otherwise it is the next state S_{t+1} calculated in the previous iteration. An action A_t is chosen for the current state based on the behavior policy (e.g. random selection for Q-learning or $\epsilon-$greedy for SARSA), using random number generator. We also provide a transition function module which takes as input the current state S_t and an action A_t, and outputs the new state S_{t+1} based on the state-action pair. The transition function module acts as a black box and the correlations between its states and actions are application-specific. For example, in a grid based robotics application, states are usually represented as the co-ordinates and the actions are usually directions of movement, and transition function outputs the new co-ordinates. In this stage we also read the Q-value $Q(S_t, A_t)$ and the reward value R_{t+1} for the current state-action pair. γ and $1 - \alpha$, are calculated to be used in later stages.
2) **Second stage:** An action A_{t+1} is chosen for the next state S_{t+1} based on the update policy (e.g. greedy policy for Q-learning, $\epsilon-$greedy policy for SARSA or probability distribution based policy for generic table based approach). Using this state-action pair (S_{t+1}, A_{t+1}) the Q-value $Q(S_{t+1}, A_{t+1})$ for this state-action pair is read from the memory.
3) **Third stage:** This is the main computation stage of the pipeline. It calculates the followings:
   - $R_{t+1} * \alpha-$ The product of learning rate and the reward function
   - $(1-\alpha) * Q(S_t, A_t)-$ The product of $1-\alpha$ calculated in the first stage of pipeline and the current Q value for the current state-action pair (S_t, A_t)
   - $\alpha * \gamma * Q(S_{t+1}, A_{t+1})-$ The product of the product of learning rate and discount factor $\alpha * \gamma$ which we calculated in the first stage of pipeline and the Q-value for the next state-action pair calculated in the first and second stage of the pipeline.

   Then using an adder to sum up all three values, giving the new updated Q value $Q_{t+1}(S_t, A_t)$ for the current state-action pair.

$$Q_{t+1}(S_t, A_t) = (1-\alpha) * Q(S_t, A_t) + \alpha * R_{t+1} + \alpha * \gamma * Q(S_{t+1}, A_{t+1}) \quad (3)$$

4) **Fourth stage:** In this stage we write the new/updated Q value for the current state-action pair back into the Q-table and if necessary in the Q_{max} table.

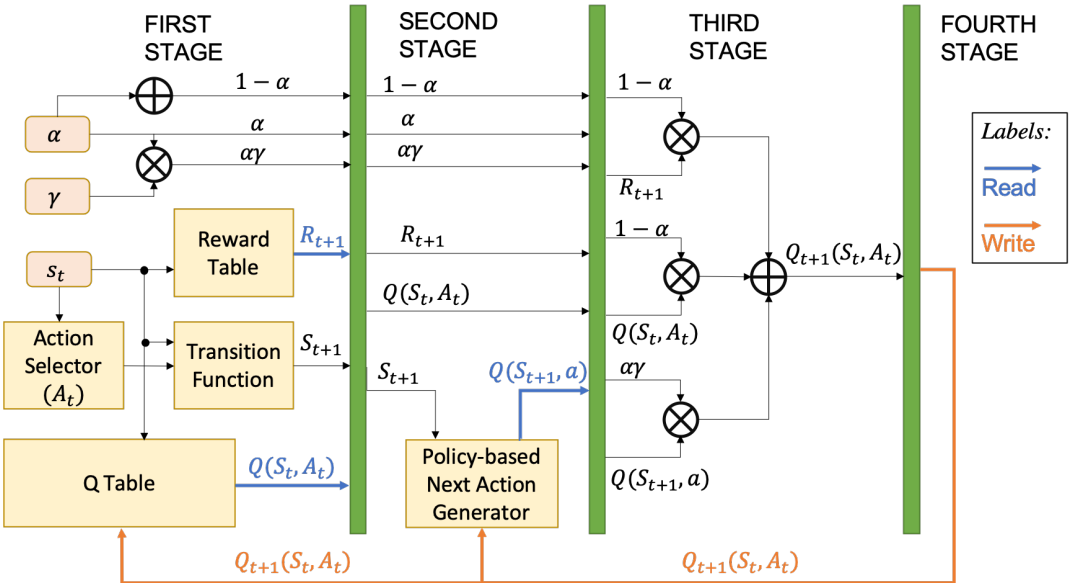

Fig. 1: Pipelined architecture for QRL

V. Reinforcement Learning Accelerators

We customize QTAccel to implement two QRL algorithms - Q Learning and SARSA on FPGA. We perform several optimizations to improve the performance of the algorithms. Moreover, we discuss how any general Q table based algorithm can be implemented using QTAccel.

A. Q Learning

We propose the following optimization for Q learning on top of our generalized architecture. We note that in Q learning the action for the next state S_{t+1} is calculated based on greedy policy i.e. we choose the action with highest Q-value. Hence, instead of accessing all the entries of Q-table corresponding to the next state and finding the action with maximum Q-value, we use an array - Q_{max} of size equal to the number of states which stores the maximum Q-value for all the states. Thus, the action is selected by a single access to Q_{max}. In the fourth stage, while writing back the new Q-value into the Q-table, an update is made to the Q_{max} if the new Q-value is higher than the current value in the Q_{max} array for the state.

B. SARSA

In this work, we implement SARSA with $\epsilon-$greedy policy selection. Under this policy, the maximum Q-value is read with probability $1-\epsilon$ and all other Q-values for this state are read with probability $\frac{\epsilon}{|A|}$, where $|A|$ is the number of actions. Similar to the Q-Learning algorithm, we use an array Q_{max} to store the maximum Q-values for each state. A random number generator is used to sample the action using the probability distribution described above. To simulate $\epsilon-$greedy approach, we use a random number generator to generate a N bit random number. If the number is between 1 and $(1-\epsilon)*2^N$ then we read the maximum Q-value else, any Q-value for this particular state can be selected with equal probability. As we know the range beforehand, we can use the random number to directly index one of the Q-values. Since SARSA is on-policy where the behavior policy is the same as the update policy, the sampled action which is available at the beginning of 3rd stage will be forwarded to the 1st stage as the next-step action.

VI. Experimental Evaluation

A. Experimental Setup

We implement the reinforcement learning accelerators designed in this work using Xilinx UltraScale+ FPGA (xcvu13p). We perform place-and-route simulations using Vivado Design Suite 2019.1. A Q-learning python implementation is also developed running on 2.3 GHz Intel Core i5 CPU as the baseline for throughput comparison. We use the grid world application to evaluate the performance of our accelerator.

In the grid world application, the environment is a grid of cells and the agent is the robot which starts at one of the cells in one of the cells and its aim is to reach a goal cell while avoiding obstacles (unreachable cells) and walls. Under this setting, the states represent the cells and the actions represent the moves of the robot. The agent randomly selects a start state and traverses the grid by choosing actions, collecting rewards and updating the Q values. Figure 2 is an example which shows a grid with 16 cells with starting and goal cells labeled. Four actions are available - left, up, right, and down. Every state-action pair in this environment is assigned with a reward value. Reaching the goal state yields maximum rewards while hitting a wall yields negative rewards.

B. Performance Metrics

We analyze the performance of our accelerator with the grid-world robotics problem described above with different state-action sizes. All evaluated sizes are listed in Table I

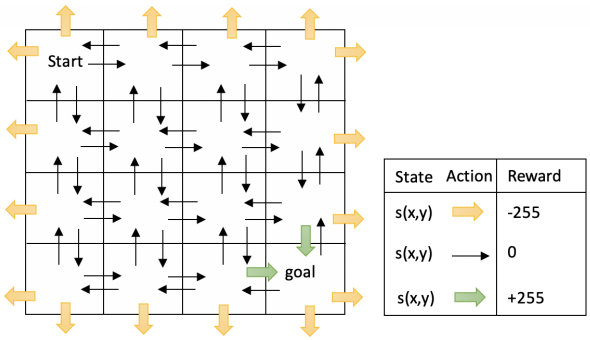

Fig. 2: Grid world example

Case	1	2	3	4	5	6	7
|S|	64	256	1024	4096	16384	65536	262144
|A|	4,8	4,8	4,8	4,8	4,8	4,8	4,8

TABLE I: Test Cases

|S| represents the maximum total number of states and |A| denotes the maximum total number of actions. The states are addressed as (x,y) coordinates. For example, when there are 256 total possible states, the address of the state is an 8-bit binary value where the most significant 4 bits represents the x-coordinate and the least significant 4 bits represent the y-coordinate. Actions are encoded as consecutive numbers. In the case of 4 actions, each action is addressed as a 2-bit binary value where 00 denotes going left, 01 denotes going up, 10 denotes going right and 11 denotes going down. When there are 8 actions addressed by 3-bits binary values, 000 denotes left, 001 denotes top-left, 010 denotes up, 011 denotes top-right, and so on in clockwise direction.

C. Resource Utilization

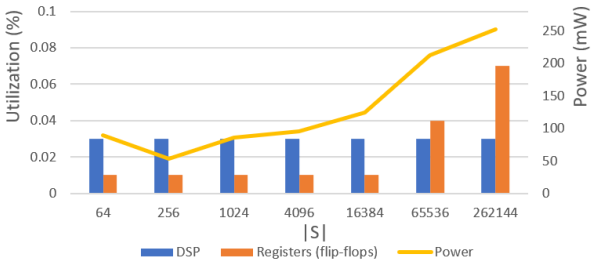

Fig. 3: Resource utilization for Q-learning

*1) **Q-Learning**:* Figures 3, 4 presents the resource utilization for various state sizes with 8 actions for Q-Learning. Our pipelined architecture efficiently uses 4 multipliers (each utilizing a single DSPs) in the design. As the problem size increases, the DSP usage stays the same. The logic/register utilization does not increase much either as the architecture is fixed for different state spaces as well. The overall logic/register utilization remains less than 0.1% for state-action pair size of 2 million. Block RAM utilization, shown in Figure 4, increases linearly with the state and action size. The bottleneck of our design is memory, as we need to store the whole

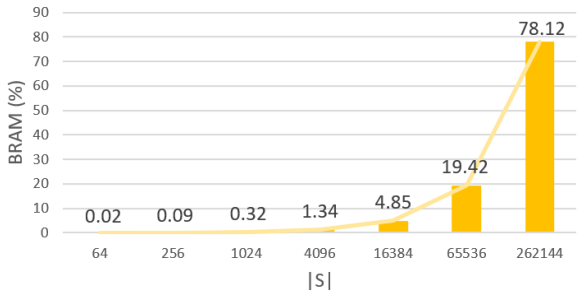

Fig. 4: BRAM utilization for both Q-learning and SARSA

reward table and Q-table in internal memory to reduce external communication. Hence, for storing larger state-action space, our design needs equally large BRAM resources to fulfill the memory requirements. For state-of-the-art FPGA devices we are able to support a state space of more than a million states-action pairs which is sufficient to support many robotics applications like space rovers.

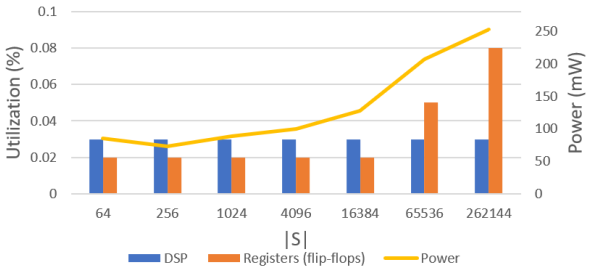

Fig. 5: Resource utilization for SARSA

*2) **SARSA**:* As mentioned in section V-B, the architecture for SARSA is very similar to Q-Learning. The main difference comes in stage 2 of the pipeline. Where instead of using greedy policy for finding the Q-value for the next stage S_{t+1}, $\epsilon-$greedy policy is used. To implement this we need a random number generator. Hence the logic utilization increases accordingly. A basic random number generator can be implemented as a linear feedback shift register and hence requires only a few registers for implementation. Hence our logic utilization (register) has increased accordingly. Using random number generator does not increase any DSPs or BRAMs utilization and hence those resources remain the same. Because of the increase in logic/register utilization the power utilization increases accordingly.

As evident from the results, we are able to support a state space of **262,144 states** and **8 actions** i.e. a state-action size of more than 2 million. This is equivalent to the grid size of **512X512** with 8 actions which is sufficient for typical robotics applications. Theoretically, a state-action pair size of 10 million can be supported using the available 360 Mb of on-chip UltraRAM. However, the synthesis tool times out for such large state spaces.

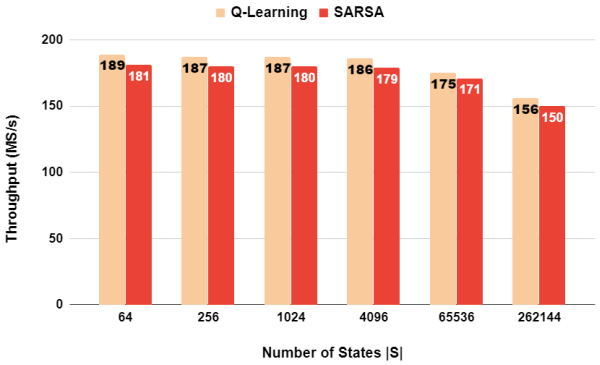

Fig. 6: Throughput for Q-Learning and SARSA

D. Throughput

Throughput (T_q) is measured by number (in Millions) of Q values/samples calculated per second (MS/s). Our pipelined architecture ensures that after the first iteration, the Q values are output every clock cycle. Figure 6 presents the throughput achieved by varying the state size $|S|$ and fixing the action size to 8 for both Q-Learning and SARSA. As evident from Figure 6, we achieve a consistent throughput of around 180 MS/s. The high throughput is sustained even on very large state space due to the efficiency of our architecture design in which the logic and DSP utilization does not increase significantly. Please note that the clock speed and hence the throughput starts decreasing for extremely large state space i.e. state-space of more than 100k. This is to be expected because for such large state space more than 50% of the BRAM would be fully utilized and this in itself puts a huge pressure on the FPGA device and hence degrades the clock speed.

E. Comparison with CPU implementation

	exp.	|S|=64	|S|=1024	|S|=16384	|S|=262144
CPU	|A|=4	105.5K	91.41K	74.17K	157.85K
FPGA	|A|=4	189M	187M	181M	156M
CPU	|A|=8	105.89K	88.7K	70.25K	15.2K
FPGA	|A|=8	189M	186M	179M	153M

* K: Thousands of Samples/s; M: Millions of Samples/s;

TABLE II: Throughput comparison with CPU

We run the Q learning algorithm on Intel Core i5 processor, and compare the achieved throughputs for different state-action sizes with FPGA implementation. For the CPU baseline, we run a python program in which the Q values are stored in a nested dictionary and are indexed by state coordinates tuples and actions. As evident from Table II, our design achieves a significantly higher throughput than CPU implementation. This is due to: (1) Q learning, essentially being a sequential algorithm, is executed in a sequential loop on CPU, not able to exploit much parallelism; (2) The limited cache size on processor (256KB L2 and 6MB L3) cannot hold all data in Q Table and rewards Table, the performance is therefore bounded by off-chip data accesses. On FPGA there is abundant on-chip memory which provides low access-latency to all the Q values and rewards.

F. Comparison with State of the Art

We compare our results with [11] which implements a Q-learning algorithm on FPGA. They conducted experiments on the Virtex 6 FPGA device. For fair comparison we also implemented our design on Virtex 7 FPGA device which has similar characteristics. Compared to [11] our resource utilization is very low for the same state-action size. For 132 state, 4 actions the design in [11] fully utilized the DSP and logic on the FPGA device. For the same state space with 8 action we only used 4 DSP (4 multipliers) and used ¡1% of logic. For the same state-action size, our throughput was more than 180 million samples per second (MS/s), which is more than $15\times$ higher than the throughput observed by [11].

The limitation of their design is the use of a finite state machine for each state-action pair. Thus, the number of multipliers required by their design is equal to the number of state-action pairs. However, as in any given iteration the Q value of only one state-action pair is updated, this leads to a lot of wasted computation by their design. Our pipelined architecture eliminates these wasted computations resulting in low resource utilization and high scalability. Moreover, the use of Q_{max} table further optimizes the amount of computation required. Figure 7 compares the number of DSPs used in our design and the state-of-the-art design [11] for same state-action pair size.

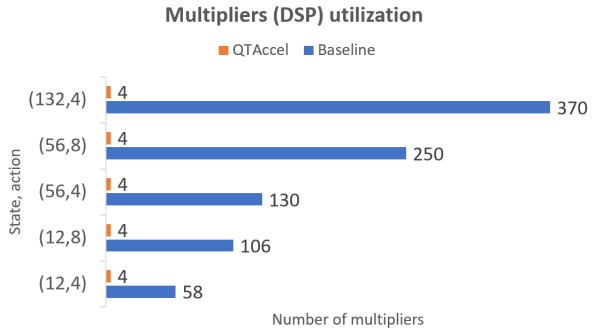

Fig. 7: Comparison of the number of DSPs used in baseline [11] and QTAccel.

Scalability - The number of DSPs available on the device becomes a bottleneck for scalability of the design in [11]. Our efficient pipelined design can support a state space of 131,072 (more than 1000X) compared with 132 supported by the design in [11] on a similar sized device while giving a throughput improvement of 15X. For state-of-the-art FPGA devices with on-chip memory of 450Mb, QTAccel can support a state-action pair size of more than 2 million. This makes our design well suited for a range of edge centric applications like robotics.

VII. Discussion

In this section we discuss how our pipelined design can be extended to support multi-agent training and generalized to solve other RL and Multi-Armed Bandit (MAB) problems.

A. Parallel Pipelines

Even for the largest state-action space that saturates on-chip memory, the DSP and Flip-Flop utilizations are fixed and small. Therefore, our design can be easily employed to train multiple agents on the same device. The design can be extended to operate in two modes:

State Sharing Learners: In this mode, we support applications where two agents perform a task sharing the same environment (i.e. same set of states, actions), for instance hunter game [28] or multi-agent box pushing with/without shared Q Table [29]. Our pipeline design can easily support 2 parallel pipelines without any change of configuration, as modern FPGAs support up to 2 concurrent accesses to the same block memory. Therefore for a given environment we can deploy two agents to explore the same environment and update the Q table, which effectively doubles the achievable throughput. In case of shared Q Table, when there are concurrent writes to the same state address, one pipeline arbitrarily overwrites the other. The effect on quality of training depend on the rate at which two agents collide onto the same state (If this rate is 100%, the throughput and convergence rate will be approximately the same as using one pipeline; however for widely-used behavior policy such as random action selection, collision is much less likely to happen and both the throughput and convergence rate should increase compared to those of single-pipeline implementation).

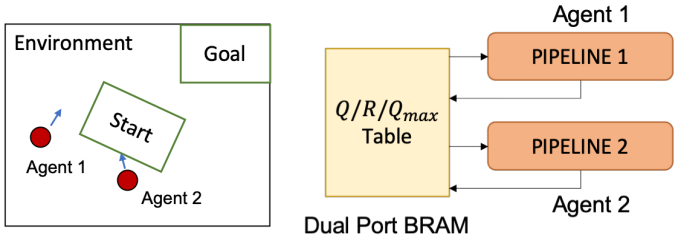

Fig. 8: 2 pipelines

Independent Learners: For problems where multiple independent agents need to be trained on separate environments (e.g. launching multiple rovers to explore the geomorphological features of a ground surface, each responsible for a subset of the entire state space), we can deploy N agents, each accessing a separate memory block which stores the Q values and rewards for states in its corresponding sub-environment. To avoid bank conflicts the Q and reward tables need to be stored in separate memory blocks. While N is upper bounded by available BRAM blocks on FPGA, we claim that this does not matter because the size of the state-action space of each sub-environment is generally larger than that of a single BRAM block.

B. Generalization to Other QRL and MAB

A policy in a RL algorithm is a probability distribution on the actions conditional on the current state. This can be represented as: an action a_i at state S_j is selected with probability

$$P(a_i|S_j) \propto f_t(S_j, a_i) \tag{4}$$

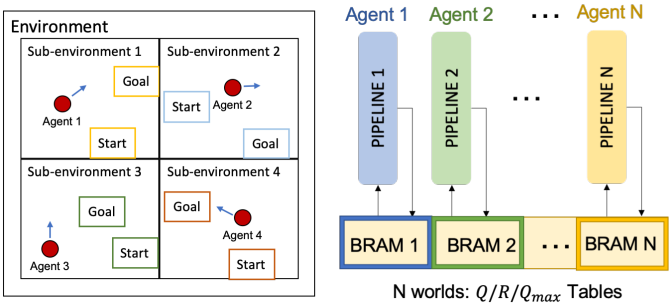

Fig. 9: N pipelines

for some temporal function f_t, that may be updated with every sample. To implement such probability distribution based policies, we use a table P which store the probability value for each state-action pair. In the second stage, the actions selection will evaluate the next action based on the probability distribution. To simulate the action selection using probability distribution we use a random number generator. Based on a random number generated in $[0, \sum_i f_t(S_j, a_i)]$, a binary search can provide the selected action in $\log n_j$ cycles, where n_j is the number of actions available at state S_j. In the final stage, the probability values need to be updated.

High-throughput, energy-efficient Multi-Arm Bandit (MAB) implementation is critical for next generation 5G wireless network applications such as distributed channel selection, opportunistic spectrum access, etc. [16]. To the best of our knowledge, no FPGA implementation exists for general MAB problems. In MAB, the agent chooses one out of $\mathcal{M}$ arms where each arm is associated with its own state S_m at time t and instantaneous reward $g_{m,t}$ which is obtained using some probability distribution (usually normal distribution [30]). The objective of the agent is to select arms in each time step to maximize the accumulated reward. As the actions and states are finite and discrete in MAB and policies such as $epsilon$-greedy are also used in MAB problems, we can adapt our design to accelerate MAB with only changes to the rewards table in the first stage. To sample rewards, uniform random numbers can be generated using linear feedback shift registers whose output can be summed up to obtain the normal distribution. [31]. While other methods exist to obtain normal distribution [32], [33], they require large number of clock cycles and are not efficient compared to our design where the pipeline is compact and provides high throughput.

Stateless Bandits are a variant of MAB which do not have states associated with rewards [16]. When implemented using QTAccel, the Q table will have just a single state and $\mathcal{M}$ actions - one action for each arm. The Q value for action m will be a function of the awards received for the arm m. For example, in EXP3 algorithm [34], the Q value of the action is an exponential function of the average reward. The probability distribution table will store the probabilities for the policies to be selected. For example, in EXP3 algorithm [34], the probability for action m is given as:

$$(1-\gamma)\frac{Q(m)}{\sum_m Q(m)} + \gamma\frac{1}{\mathcal{M}} \tag{5}$$

where $\gamma \in [0, 1]$ is a fixed constant. For **Stateful Bandits** [16], the state space can be represented by concatenation of the states of individual arms. Typically, the number of arms is very small ($\approx$5) [35], so the size of the resulting table will still be tractable.

VIII. Conclusion

In this paper, we presented a pipelined FPGA architecture, QTAccel, for accelerating QRL. QTAccel achieves a high throughput of one sample per clock cycle, and the pipelined design makes efficient use of hardware resources and is able to scale to over one million state-action pairs on state-of-art FPGA. Experiments evaluations illustrate that our implementation outperforms the state-of-art Q learning accelerator in both throughput and resource utilization.

In future, we will customize our architecture to implement more variants of Multi-Armed Bandit problems. We will develop efficient pipelined implementation of probability based policy selection for such problems to ensure high-throughput architecture with limited stalls due to dependencies.

References

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

[2] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[3] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[4] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[5] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 499–513.

[6] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on fpga platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[7] W. M. van Otterlo M., *Reinforcement Learning State-of-the-Art*. Springer, Berlin, Heidelberg, 2012.

[8] S. R. Kuppannagari, R. Chen, A. Sanny, S. G. Singapura, G. P. C. Tran, S. Zhou, Y. Hu, S. P. Crago, and V. K. Prasanna, "Energy performance of fpgas on perfect suite kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[9] "Virtex-UltraScale+ FPGA Family," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[10] "Intel Stratix 10 MX FPGAs," https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html.

[11] L. M. Da Silva, M. F. Torquato, and M. A. Fernandes, "Parallel implementation of reinforcement learning q-learning technique for fpga," *IEEE Access*, vol. 7, pp. 2782–2798, 2018.

[12] P. R. Gankidi, "Fpga accelerator architecture for q-learning and its applications in space exploration rovers," Ph.D. dissertation, Arizona State University, 2016.

[13] R. Rajat, Y. Meng, S. Kuppannagari, A. Srivastava, V. Prasanna, and R. Kannan, "Qtaccel: A generic fpga based design for q-table based reinforcement learning accelerators," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 323–323.

[14] J. Langford and T. Zhang, "The epoch-greedy algorithm for multi-armed bandits with side information," in *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, Eds. Curran Associates, Inc., 2008, pp. 817–824. [Online]. Available: http://papers.nips.cc/paper/3178-the-epoch-greedy-algorithm-for-multi-armed-bandits-with-side-information.pdf

[15] M. N. Katehakis and A. F. Veinott Jr, "The multi-armed bandit problem: decomposition and computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987.

[16] S. Maghsudi and E. Hossain, "Multi-armed bandits with application to 5g small cells," *IEEE Wireless Communications*, vol. 23, no. 3, pp. 64–73, 2016.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[18] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3389–3396.

[19] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1705.04862*, 2017.

[20] P. R. Gankidi and J. Thangavelautham, "Fpga architecture for deep learning and its application to planetary robotics," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–9.

[21] 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. 186 340–186 351, 2019.

[22] V. L. Prabha and E. C. Monie, "Hardware architecture of reinforcement learning scheme for dynamic power management in embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, p. 065478, 2007.

[23] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[24] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," Tech. Rep., 1994.

[25] A. N. Burnetas and M. N. Katehakis, "Optimal adaptive policies for markov decision processes," *Math. Oper. Res.*, vol. 22, no. 1, pp. 222–255, Feb. 1997. [Online]. Available: http://dx.doi.org/10.1287/moor.22.1.222

[26] J. Johnson, J. Li, and Z. Chen, "Reinforcement learning: An introduction: R.s. sutton, a.g. barto, mit press, cambridge, ma 1998, 322 pp. isbn 0-262-19398-1." *Neurocomputing*, vol. 35, pp. 205–206, 01 2000.

[27] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds. San Francisco (CA): Morgan Kaufmann, 1995, pp. 30 – 37. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978155860377650013X

[28] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, 1993, pp. 330–337.

[29] M. Rahimi, S. Gibb, Y. Shen, and H. M. La, "A comparison of various approaches to reinforcement learning algorithms for multi-robot box pushing," in *International Conference on Engineering Research and Applications*. Springer, 2018, pp. 16–30.

[30] M. N. Katehakis and H. Robbins, "Sequential choice from several populations." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 92, no. 19, p. 8584, 1995.

[31] H. Forstén, *Generating normally distributed pseudorandom numbers on a FPGA*. [Online]. Available: https://hforsten.com/generating-normally-distributed-pseudorandom-numbers-on-a-fpga.html

[32] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 139–158.

[33] S. S. Roy, F. Vercauteren, and I. Verbauwhede, "High precision discrete gaussian sampling on fpgas," in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 383–401.

[34] S. Bubeck, N. Cesa-Bianchi *et al.*, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations and Trends® in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[35] N. Gulati and K. R. Dandekar, "Learning state selection for reconfigurable antennas: A multi-armed bandit approach," *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 3, pp. 1027–1038, 2013.