FPGA Acceleration of Deep Reinforcement Learning using On-Chip Replay Management[*]

Yuan Meng[†]
ymeng643@usc.edu
University of Southern California
Los Angeles, United States

Chi Zhang[†]
zhan527@usc.edu
University of Southern California
Los Angeles, United States

Viktor Prasanna
prasanna@usc.edu
University of Southern California
Los Angeles, United States

ABSTRACT

A major bottleneck in parallelizing deep reinforcement learning (DRL) is in the high latency to perform various operations used to update the Prioritized Replay Buffer on CPU. The low arithmetic intensity of these operations leads to severe under-utilization of the SIMT computation power of GPUs. In this work, we propose a high-throughput on-chip accelerator for Prioritized Replay Buffer and learner that efficient allocates computation and memory resources to saturate the FPGA computation power. Our design features hardware pipelining on FPGA such that the latency of replay operations is completely hidden. Our experimental results show that the performance of the key operations in managing Prioritized Replay Buffer including sampling and priority insertions are improved by factor of $21\times \sim 40\times$ compared with the state-of-the-art implementations on CPU and GPU. In addition, our system design leads to up to $4.3\times$ improvement in overall throughput compared with the state-of-the-art CPU-GPU implementations.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

KEYWORDS

prioritized replay buffer, deep reinforcement learning, FPGA

ACM Reference Format:
Yuan Meng, Chi Zhang, and Viktor Prasanna. 2022. FPGA Acceleration of Deep Reinforcement Learning using On-Chip Replay Management. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3528416.3530227

1 INTRODUCTION

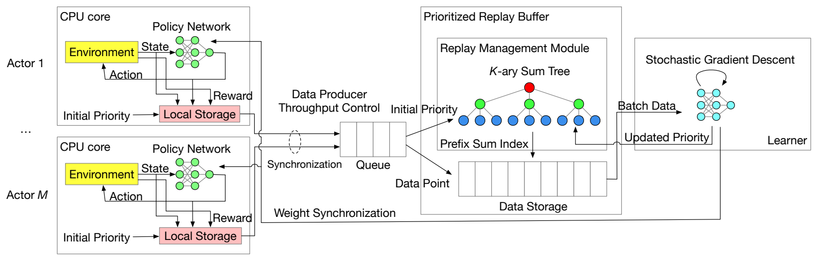
Reinforcement Learning (RL) is widely used in many application areas including self-driving cars, robotics, surveillance, etc. [2, 36, 38]. In RL, an agent iteratively interacts with an environment to improve its policy such that the expected accumulated reward along the trajectory is maximized. Existing distributed RL frameworks [12, 16, 33] employ a general architecture consisting of parallel actors, a centralized learner and a Prioritized Replay Buffer [31] as shown in Figure 1a. Parallel actors concurrently perform data collection from the environment and data insertion into the Prioritized Replay Buffer. The centralized learner samples data from the Prioritized Replay Buffer and performs stochastic gradient descent (SGD) [30]. The new priorities after learning are updated in the Prioritized Replay Buffer. The key operations of Prioritized Replay Buffer include sampling and priority update. The priority of each data point is proportional to the loss function, and the sampling distribution is proportional to the priority. The priority of each data point in the Replay Buffer is stored in a K-ary Sum Tree data structure [41] that can perform sampling and update in $O(\log N)$ time, where N is the number of total data points in the Replay Buffer.

The latency of sampling/priority update is high on CPU due to the sequential execution of different data points in a batch. Although it can be parallelized using multi-threading, it is undesirable as it may slowdown the execution of actors. On GPU, sampling different data points in a batch can be performed in parallel as it is a read-only operation. However, both priority update and sampling have low arithmetic intensity. This makes GPU-based acceleration memory bound and fails to saturate the computation power of GPUs.

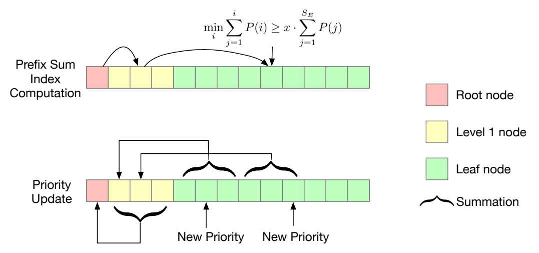
To tackle these drawbacks, we propose a generic accelerator on CPU-FPGA heterogeneous platform to implement the architecture in Figure 1a with a specialized on-chip Replay Management Module (RMM) on FPGA. FPGAs have emerged as a promising platform for accelerating both computation and memory intensive AI applications [4, 5, 18]. By efficiently allocating compute and memory resources for Prioritized Replay Buffer on FPGA, the allocated resources are fully utilized despite the low arithmetic intensity of sampling and priority update. In our design, the latency of replay operations is completely hidden via hardware pipelining. The remaining resources on FPGA are used to design a high-throughput learner such that the computation power of FPGA is fully saturated to achieve superior performance. Specifically, our key contributions are:

- We map actors onto CPU for fast data collection and the centralized learner onto FPGA for fast gradient computation. In particular, we map the RMM on FPGA to increase system throughput and develop a data-structure hardware co-optimization to further improve the performance.
- We develop a generic accelerator template in High Level Synthesis (HLS) for a wide range of RL algorithms featuring a novel Replay Management Module (RMM) that enables parallel insertion, parallel sampling and parallel priority updates of the K-ary Sum Tree data structure [41]. We optimize the performance of the RMM using:

[*]This work is supported by NSF under grant No. CNS-2009057.
[†]Both authors contributed equally to the paper.



(a) Framework overview


$\min_i \sum_{j=1}^{i} P(i) \geq x \cdot \sum_{j=1}^{S_E} P(j)$
Prefix Sum Index Computation
Priority Update
New Priority
New Priority
🟥 Root node
🟨 Level 1 node
🟩 Leaf node
Summation

(b) Illustration of sampling and priority update in Prioritized Replay Buffer

Figure 1: A generic view of parallel reinforcement learning

- Specialized variable-precision fixed point data format for storing priority values in the RMM;
- Partitioning of the K-ary Sum Tree that enables conflict-free parallel data accesses;
- Pipelined replay operations that allow concurrent access to multiple memory banks storing the K-ary Sum Tree.

- We develop a generic throughput-oriented learner module on FPGA that exploits both neural network model parallelism and data parallelism.
- For widely used RL algorithms including DQN [23] and DDPG [21], our experimental results demonstrate that: 1) our on-chip RMM achieves up to $21\times \sim 40\times$ speedup on key operations compared with the CPU and GPU baselines; and 2) the throughput of the overall system is improved by $4.3\times$.

2 PARALLEL REINFORCEMENT LEARNING

2.1 WorkFlow Overview

We show a generic view of existing parallel RL workflow [8, 12, 16] in Figure 1a. It consists of 4 key components:

2.1.1 Actors. Each actor contains an instance of the environment, a policy network represented as a neural network and a local storage. The environment outputs the current state s. The policy network computes the action a given the current state s via neural network inference. The action a is actuated in the environment to obtain the next state s' and the reward r. The policy network computes the current loss P as the initial priority using (s, a, s', r). Each actor contains a local storage to temporarily store the data points consisting of tuple (s, a, s', r, P) collected by the actor. When the local storage is full, all the data points are popped out and inserted into a shared Queue. The data points in the shared Queue are inserted into the global Prioritized Replay Buffer [31].

2.1.2 Prioritized Replay Buffer. Prioritized Replay Buffer [31] has been proposed to sample data points with probability proportional to the current loss to speed up training. It consists of a Data Storage and a Replay Management Module (RMM). **Data Storage** is used to store data points produced by the actors. During training, batches of data points of size $\mathcal{B}$ are popped out from the Queue and inserted into next available locations in the Data Storage. FIFO replacement policy is used when the Data Storage is full. **RMM** manages the priority P_i associated with the i-th data point in the Data Storage. Internally, it is implemented as a K-ary Sum Tree [41]. In a K-ary Sum Tree, each node has K child nodes and the value of each node is the sum of values of its child nodes. The i-th leaf node stores the actual priority value P_i. A K-ary Sum Tree provides efficient prefix sum index computation for sampling to perform training and priority update after each training iteration.

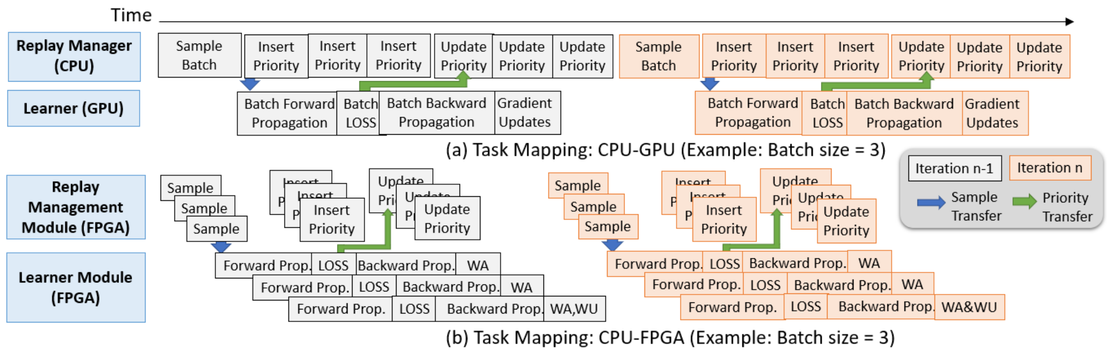
Key operations. **Sampling** from the Prioritized Replay Buffer decides which samples (indices) are used for training the neural network. For each sample, a data point x_i is selected according to a priority distribution $\Pr(i) = P(i)/\sum_i P(i), i \in [0, S_E)$, where S_E is the total number of data points in the Prioritized Replay Buffer. To do so, we first sample $x \sim U(0, 1)$. Then, we use the cumulative density function ($cdf = \sum_{j=1}^{i} \Pr(j), i \in [0, S_E)$) to derive the sample index $i = cdf^{-1}(x)$. This is equivalent to finding the minimum index i, such that the prefix sum of the probability up to i is greater than or equal to x, the target prefix sum value:

$$\min_i \sum_{j=1}^{i} P(i) \geq x \cdot \sum_{j=1}^{S_E} P(j) \qquad (1)$$

Such index i is known as **prefix sum index**. To find index i, we traverse from the root node to the leaf node level by level as shown in Figure 1b. During the traversal of each level, we need to read the prefix sum of priority values from all the child nodes. The time complexity of finding prefix sum index is $O(K \log_K N)$, where N is the number of elements in the replay buffer. **Priority Update** requires updating the current priorities using newly computed priorities. This operation is performed after each training iteration. To update the priority, we update the node values from the leaf node to the root node as shown in Figure 1b. The time complexity of priority update is $O(\log_K N)$. **Data Insertion** is performed when new data points are popped out from the Queue. Data Insertion includes inserting the actual data points into the Data Storage and updating the priorities from zero to the initial values.

Data Parallelism. Note that sampling and priority update are always performed on a batch of data with size $\mathcal{B}$. Since computing the prefix sum index is read-only, sampling different data points inside a batch can be fully parallelized. Performing priority update on a batch of data can be reduced to classic parallel sum reduction problem as shown in Figure 1b.

Arithmetic Intensity. The ratio of number of operations performed to the amount of data accessed is known as the algorithm's



(a) Task Mapping: CPU-GPU (Example: Batch size = 3)

(b) Task Mapping: CPU-FPGA (Example: Batch size = 3)

Figure 2: (a) Existing High-Level CPU-GPU Mapping; and (b) Our Proposed FPGA-based mapping

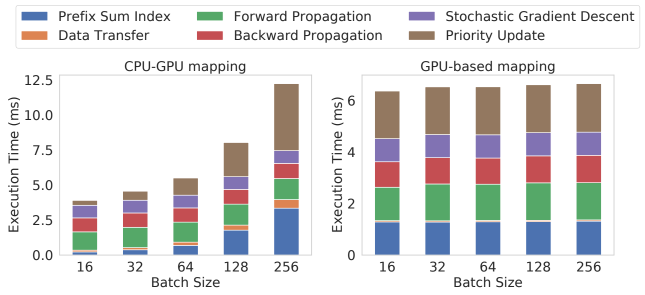


Figure 3: Execution time breakdown of a training iteration. Data Transfer refers to sending data points from the Data Storage to the learner.

arithmetic intensity [26]. The arithmetic intensity of sampling is 1 FLOPS/word as each data read from the memory is only used once during the tree traversal. The arithmetic intensity of priority update is 0.5 FLOPS/word because each data is used once after each read and write.

2.1.3 Learner. Following [8, 12, 16], we use a centralized learner to perform policy updates due to its stability in terms of convergence. At each step, the learner i) samples a batch of indices via the RMM; ii) accesses the actual data points in the Data Storage using the sampled indices; iii) performs forward propagation to compute the loss; iv) performs backward propagation to compute the gradients; v) updates the weights of the neural network using the gradients via stochastic gradient descent [30]; iv) updates the priorities of the sampled batch data using the loss via the Replay Management Module.

2.2 Performance Metric

The execution of actors and the centralized learner are independent. The throughput of data collection is fixed given the total number of actors. Thus, the convergence rate of parallel RL is fully determined by the throughput of the learner. Following existing work [8, 12, 16], we measure the performance using **training throughput**, which is defined as the number of gradient steps performed by the learner per second (GPS).

2.3 Mapping Methodology Overview

In this section, we discuss potential mapping methodologies and motivate the implementation of RMM and training on FPGA. The training throughput is affected by the speed of **prefix sum index computation, data transfer from Data Storage to device memory, neural network training and priority update**. Potential mapping methodologies are shown in Table 1. Following [8, 12, 16], each actor is mapped onto a CPU core. This is because most environment simulators can only run on CPU. The learner is always mapped onto the accelerator (GPU or FPGA) for fast neural network updates.

Table 1: Potential Mapping Methodologies

Mapping	Actors	Learner	RMM	Data Storage
CPU-GPU	1 actor/CPU core	GPU	CPU	CPU DRAM
GPU-based	1 actor/CPU core	GPU	GPU	GPU DRAM
FPGA-based	1 actor/CPU core	FPGA	FPGA	FPGA DRAM

2.3.1 CPU-GPU and GPU-based Mapping. We show a high-level execution timeline of CPU-GPU and GPU-based mapping in Figure 2. Note that RMM is implemented as a binary Sum Tree following existing open-source RL frameworks [20]. In CPU-GPU mapping, the RMM is implemented on CPU. The Data Storage is in host DDR memory. During a training iteration of the learner, a batch of data is sampled from the Prioritized Replay Buffer based on the priority distribution. The data is transferred from the host DDR memory to the GPU memory. Then, the GPU computes the forward propagation to obtain the loss, which is the new priority. It then sends the new priority to RMM for priority update. In the meantime, the GPU performs the backward propagation and performs stochastic gradient descent [30] to update the weights of the neural network.

Limitations of CPU-GPU mapping. We show the execution time breakdown of CPU-GPU mapping in Figure 3. Specifically, we profile the performance of DQN [23] in Ape-X [12] framework using open source RLlib [20] on LunarLander environment [1]. The policy network is a 3-layer MLP with hidden layer size 256. The RMM is executed using a single thread. The limitations of CPU-GPU mapping are: i) **data transfer time**: As the batch size increases, the data transfer time is comparable with forward propagation time, which slows down the learner. ii) **limited cores for thread-level data parallelism**: As shown in Section 2.1.2, data parallelism can be utilized to accelerate sampling and priority update. However, existing parallel RL frameworks allocate up to hundreds of actors, each running on a CPU core [12]. This makes allocation of additional threads for Replay Management Module undesirable as it may slowdown the data collection and the overall system.

Limitations of GPU-based mapping. We show the execution time breakdown of GPU-based mapping in Figure 3 with the same configuration as in CPU-GPU mapping. Note that the data transfer time is almost negligible as RMM stores all the data on GPU. We observe that the breakdown of different batch sizes are almost identical. It indicates that the SIMT of GPUs can fully parallelize the execution of different data points inside a batch. However, the low arithmetic intensity of sampling and priority update makes these operations memory bound and the GPU threads idle most of the time.

2.3.2 FPGA-based Mapping.

Memory requirement. A typical Prioritized Replay Buffer contains $10K$ to $1M$ data points [23]. The number of nodes in a K-ary Sum Tree is at most 2 times the Buffer size $K = 2$. Each node contains a 32-bit floating point (4 byte) priority value. Thus, it takes 0.08 MB to 8 MB to store the K-ary Sum Tree. It is smaller than the available on-chip memory of most state-of-the-art data center FPGAs [13, 39]. Therefore, we implement the RMM using on-chip memory to achieve high performance. For tasks using image-based state space like Atari Games [1], it consumes around 7 GB memory with 1 million 84×84 gray-scale images. Thus, the complete data can be stored in FPGA DDR memory [13, 39]. This avoids the data transfer between the host and the device during replay sampling.

High-level methodology. By efficiently allocating computation and memory resources for Prioritized Replay Buffer on FPGA, the allocated computation resources can be fully utilized despite the low arithmetic intensity of sampling and priority update. In addition, the latency of sampling, priority update and training can be completely hidden via hardware pipelining. The remaining resources on FPGA are used to design a high-throughput learner such that the computation power of FPGA is fully utilized to achieve superior performance.

Applicability. Note that the advantages of FPGA-based mapping is only applicable when the system bottleneck is the RMM. If the training becomes the bottleneck (e.g., the total time to perform forward propagation, backward propagation and stochastic gradient descent can fully hide the RMM operations as shown in Fig. 2), the performance of GPU-based mapping will be superior to FPGA-based mapping due to higher GPU device performance (clock frequency, larger number of floating-point compute cores, etc). However, the neural network used to solve tasks with low-dimensional state space is 3-layer perceptron (MLP) in existing RL implementations as shown in [21, 23]. In this case, the computation time of RMM operations including prefix sum index computation and priority update dominates that of MLP training.

3 ACCELERATOR DESIGN

We employ parallel actors on the CPU to collect data points from the environment and pipelined learners on the FPGA to compute the gradients and update the neural network weights. The overall system architecture is shown Fig. 4. The data points collected by the actors are sent to and stored in the FPGA DRAM, and their indices and priorities are managed using RMM on the FPGA on-chip SRAM. All activations and gradients are stored on-chip. Neural network weights are stored in the Device memory and streamed on-chip as needed. Our accelerator consists of two top-level building blocks: 1. a K-ary Sum Tree based RMM for accelerating Replay operations and 2. a Learner Module that supports high-throughput training.

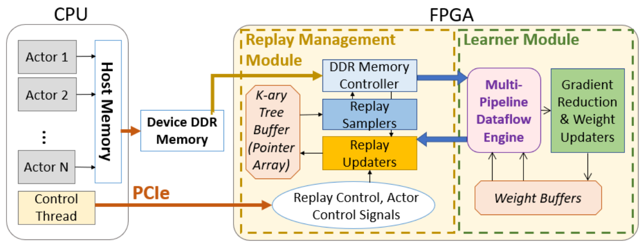


Figure 4: System Architecture

3.1 On-chip RMM

The primary objectives of our RMM design are: (1) providing sufficient effective memory bandwidth to alleviate the communication bottleneck in performing low-arithmetic replay operations. (2) overlapping the Sum Tree traversals of different data points in a batch as well as overlapping computation with data accesses using hardware pipelining. The basic hardware units for performing replay operations are the samplers and updaters. Note that an updater is responsible for both priority insertion and priority updates.

To achieve objective (1), we store the complete K-ary Sum Tree data structure using the on-chip SRAM on the FPGA. The nodes of the Sum Tree are ordered by tree level, and nodes in the same level are distributed to one or multiple SRAM banks. Each sampler/updater requires read/write access to one word every clock cycle, and each SRAM bank provides single-cycle access to any data element through a read/write port. To exploit the maximal effective SRAM bandwidth in performing the replay operations such that they are no longer memory-bound, we need to ensure that in each clock cycle, all the samplers and updaters can access the data without bank conflicts. Therefore, we propose a fined-grained scheduling to avoid any race condition in reading or modifying the same bank. Specifically, we allocate H samplers (updaters), where H is the tree height. Each sampler (updater) is only allowed to access a "critical region" of the Sum Tree at any time. A "critical region" is defined as a group of nodes in a certain level of the tree. Nodes on different levels are stored in separate SRAM banks. In any cycle, H samplers (updaters) can concurrently access different critical regions of the K-ary Sum Tree, ensuring no bank conflict and stall-free pipelining explained in the following.

To achieve objective (2), we apply pipelining to both sampling and update processes. The sampling process for each data point sequentially propagates through H levels and tracks the prefix sum, P, by accumulating the values from all K nodes (described in Sec. 2.1.2). We thus exploit pipeline parallelism between different data points in a batch. As shown in Fig. 5, the samplers are connected by FIFOs, each responsible for traversing up to K sibling nodes in the same level. As soon as a sampler $d (0 < d \leq H)$ complete processing data point a $(0 < a \leq \text{batch size})$, sampler $d + 1$ starts processing data point a and sampler d starts processing data point $a + 1$ in parallel. This way, the parallelism provided by all the samplers is fully utilized for batch sampling. The state machine logic for processing K sibling nodes in sampler d based on the prefix sum P is shown in Fig. 5. At the leaf level, the index where P reaches x' is the minimum index i whose prefix sum value $\geq$ the target prefix sum value $x \cdot \sum_{j=1}^{S_E} P(j)$ (RHS of Eq. 1), meaning that the data point at index i should be sampled. For the priority update operations, H

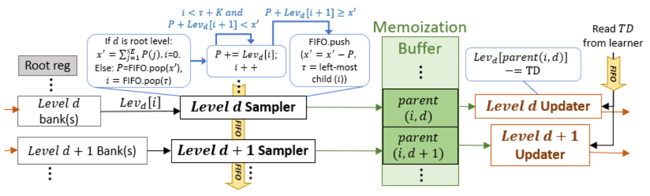


Figure 5: Replay Samplers and Updaters. τ ($\tau + K$) is the index of the left (right)-most child node in the current level.

updaters concurrently update the sum values at each level using the TD error obtained at the learner LOSS stage (The learner pipeline is introduced in Sec. 3.2). A FIFO is used as the communication channel between the learner pipeline and the updaters. The replay update process is overlapped with subsequent computations in the learner pipeline. To eliminate computation overhead of updaters in back-tracking the Sum Tree, we apply Memoization technique that dedicates a light-weight buffer (Fig. 5) to store the traversed path. This buffer only needs to store $H-1$ values for each data point, because the update of a priority value require updating the prefix sum values at its $H-1$ parents including the root.

HLS-generated floating point accumulator takes multiple cycles to compute [6], introducing loop-carried dependency in the prefix sum accumulation computation. This leads to pipeline stalls and prevents us from efficiently overlapping computation with data accesses as stated in objective (2). To workaround such inefficiency, we use fixed-point arithmetic that only requires single-cycle accumulation. We introduce a variable-precision fixed-point representation scheme specialized for storing the Sum Tree. We first identify the upper bound of the sum of the priorities. The upper bound is used to decide the range and integer bit-width of the root register. Each of the subsequent levels adopts integer bit-width of $W_b = W_b^{parent} - \log_2 K$ to avoid any overflow in calculating the sum of all K child values. This representation scheme does not affect the sampling result compared with floating point representation.

Overall, our RMM design enables single-cycle arithmetic operations with single-cycle data accesses, such that the replay operations can be completely hidden (overlapped) by the execution of Learner Module.

3.2 Learner Module

As the latency of replay operations are hidden by the training process of the learner, the learner becomes the bottleneck in FPGA-based mapping. Therefore, we carefully design the Learner Module with the goal of minimizing the execution time of each gradient step. SGD training algorithms consists of forward propagation (FW), loss computation (LOSS), backward propagation (BW), weight aggregation (WA) and weight update (WU) steps. The design principle of the Learner Module is to support both pipelining across different layers of the neural network and data parallelism (e.g., a batch of data points is split into smaller batches and processed concurrently). Based on this principle, we design a Multi-Pipeline Dataflow architecture composed of multiple learner pipelines and a module for WU weight gradient reduction as shown in Fig. 6. We also define the following two hardware parameters in the Learner Module:

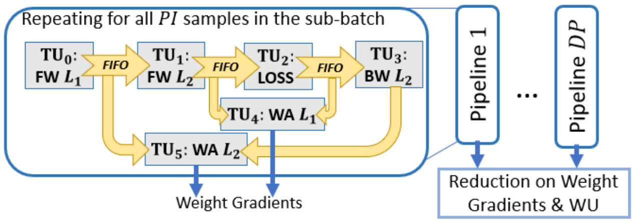


Figure 6: Learner Module Architecture: Multi-Pipeline Dataflow for L =3 MLP neural network. L_i denotes i^{th} Layer.

3.2.1 *Pipelining Factor PI.*

PI is the number of data points in the sub-batch processed by a learner pipeline. The learner pipeline for a L-layer neural network model consists of $n = 3 \times (L-1)$ stages: FW through $(L-1)$ layers of policy and value networks, computing LOSS, BW through $(L-2)$ layers, and WA for all $(L-1)$ weight tensors. Each of these stages is mapped to a unique Tensor Unit (i.e. systolic array of Multiply-Accumulate units), TU_i, $i \in [1, ..., n]$. To realize data streaming between stages, these modules (TUs) are connected by FIFOs, as depicted in Fig. 6. Note that as soon as the producer stage of a FIFO pushes data into the FIFO, the consumer stage pops data immediately in the next cycle and start executing. The gradients generated by each sample in the WA stage are accumulated into a scratchpad memory to be further aggregated with other pipelines.

3.2.2 *Data Parallel Factor DP.*

DP is the number of parallel learner pipelines in the Learner Module. It captures the data parallelism of the learner Module. Specifically, we make DP copies of the pipeline described in Sec. 3.2.1 to process sub-batches in parallel, as depicted in Fig. 6. After the intermediate weight gradients are obtained by all the DP pipelines, the WU stage modifies the weight tensors based on the reduction over all scratchpads. Note the WU task is serialized with the FW-BW-LOSS-WA stages of learner pipelines. This avoids read-after-write hazards by ensuring that weights are not modified during the FW and BW execution.

Note that the product of PI and DP equals the total batch size processed by the Learner Module. Conceptually, for a given batch size, higher Data Parallel Factor achieves higher throughput for FW-BW-WA stages, but causes longer overhead for reduction over all the pipelines. High Data Parallel and low Pipelining Factor also lead to low effective hardware utilization in each pipeline if PI is too small to saturate the concurrency provided by all n stages. The Data Parallel Factor and Pipelining Factor need to be carefully chosen for achieving the best performance under the constraints of a given FPGA device. We further describe the design space exploration process for searching the optimal accelerator parameters.

3.2.3 *Learner Design Space Exploration.*

The overall learner latency $T_{learner} = T_{pipeline} + T_{WU}$ for one RL gradient step is determined by PI, DP and compute resources allocated to the pipeline stages under the constraint of available resources. To minimize $T_{learner}$, we define a general methodology to derive the desired hardware parameters on any target device to minimize learner latency. Our DSE follows two steps:

a). Pipeline Local Optimization: The latency of a pipeline is characterized by $T_{pipeline} = (PI + n - 1) \times T_{max}$. As PI increases, n becomes negligible and the performance is bounded by the latency of the slowest pipleine stage, $T_{max} = max_{i=1}^{n} T_i$. The latency of a pipeline stage can be theoretically modeled as the number of operations processed divided by the total number of Multiply-Accumulation units (or DSP units) assigned to the TU. For given n and PI, to minimize $T_{pipeline}$, T_{max} needs to be minimized. This means that the latencies of all n stages need to be balanced. Therefore, we allocate compute resources (DSP units) to each TU proportional to the number of multiply-add operations in each stage. This ensures high pipeline resource utilization without expensive exhaustive search of all the combinations of resource allocations to stages. Specifically, we derive a ratio $r_1 ... r_n$ for n stages, normalized to the stage with minimum number of operations. The total number of DSPs allocated to TU_i, $i \in [1...n]$) is $r_i \times f$, where f is a factor that controls the total amount of DSPs allocated to a learner pipeline. f is derived in Step b). Global Optimization.

b). Global Optimization: The second part of the learner latency, T_{WU}, captures the latency of reduction over DP intermediate weight gradients and the latency of WU for all weight tensors: $T_{WU} = \sum_{j=1}^{L} (DP \times size(W_j))/(r_{WU} \times f)$. Note that $r_{WU} \times f$ is the total number of DSPs allocated to the WU stage. To exploit the maximum parallelism in WU, we let r_{WU} to be consistent with the number of SRAM banks for storing the weights.

Given a training batch size BS, we determine the optimal combination of PI and DP by searching for all possible combinations in BS steps and applying steps **a)** and **b)** described above to obtain the minimum $T_{learner}$ parameterized with f: $\underset{DP,PI}{\text{argmin}}(T_{pipeline} + T_{WU})$ Then, we increment f until one of the resource constraints (available number of DSPs, SRAM banks, Look-Up Tables) is reached.

3.2.4 *Learner: Algorithm-Specific Scheduling.*

Target Network. State-of-the-art Deep RL algorithms [9, 28] use target network(s) to stabilize training by using a fixed target throughout sequential episodes of training. We create parallel chains of Tensor Units to perform FW through both the value (or policy) network(s) and the target network(s) at the same time. We also develop a Target Synchronization Module (TSM) to average the target weights with the value (or policy) network weights [21]. The TSM is activated every fixed number of gradient steps as controlled by the main loop running on the host.

Actor-Critic. Actor-Critic algorithms (e.g., DDPG, SAC [9]) involve an actor network that produces an action and a critic network that approximates the value of a state-action pair. The actor and critic networks are trained interactively (The training process of the actor also involves the FW and BW of the critic). For actor-critic algorithms, we organize the TUs based on the task dependency graph for training both networks. Generally, the update of actor network is dependent on that of the critic [11]. We use a "Lagged Critic" mechanism to let the actor network use the obsolete critic weights from the previous training iteration (i.e., gradient step) when updating actor weights. This eliminates the intra-iteration read-after-write dependency between the critic network WU (write critic weights) and critic FW/BW in training the actor network (read critic weights). It lags the critic update (serving for the actor update) by only one iteration, which is a negligible cost compared to the millions of iterations required in total [11, 37].

4 EVALUATION

4.1 Experimental Setup

RL environments and algorithms: We choose widely used benchmarks CartPole and LunarLander [1] to evaluate the performance of our proposed methods. We evaluate our methods using two widely used RL algorithms DQN [23] and DDPG [21].

Toolchains: We develop a parameterized FPGA kernel template using High-Level Synthesis (HLS) for quick customization and easy integration with domain-specific frameworks (e.g., Pytorch [29]). We follow the VITIS hardware development flow [17] for bitstream generation. OpenCL is used to implement the data transfer between the host and the FPGA.

Hyper-parameters: Following the default hyper-parameters of existing RL algorithms [21, 23], the neural network models for approximating the Q function and the policy are 3-layer MLP with 64 hidden units. The size of the Prioritized Replay Buffer is 1 million. We set $K = 64$ in the K-ary Sum Tree.

Hardware: Our experiments are conducted on Intel(R) Xeon(R) Gold 5120 CPU, a GTX 3090 GPU and a Xilinx Alveo U200 accelerator board [40].

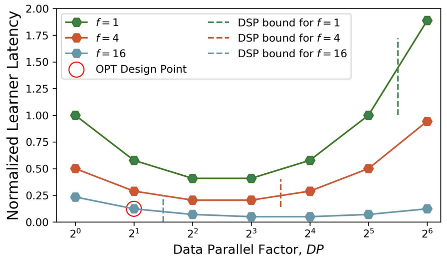


Figure 7: Illustration of design space exploration (DSE). The learner latency is normalized to the value at $DP = 1, f = 1$. The OPT Design Point yields the minimum latency within the DSE bound.

4.2 Optimal Accelerator Configuration

We identify the optimal accelerator configuration for a given RL algorithm, batch size and target device constraints following the procedure of design space exploration (DSE) as described in Sec. 3.2.3. In Fig. 7, we show an example of identifying the optimal DP and f for accelerating DQN with batch size 64. We search all combinations of DP, PI ($DP \times PI$= batch size) and increase f until finding the design point that yields the lowest learner latency for one complete gradient step within the hardware resource constraints. We summarize the final hardware resource utilization obtained using the DSE for both the algorithms in Table 2. On-chip BRAM and URAM are both included as the SRAM resources on Xilinx FPGAs [39]. In modern multi-die FPGAs, an FPGA chip is built by a manufacturing process that combines multiple Super-Logic Regions

Table 2: Accelerator Configuration and Resource Utilization

Algorithm	Hardware parameters and resource utilization			
DQN (Percentages are derived wrt 2 SLRs on the device)	Pipeline Factor (*PI*)	Data Parallel Factor (*DP*)	Max. *TU* Parallelism ($r \times f$)	# SLR Constraint (RMM, learner)
	BS/2	2	32	(1,1)
	SRAM	**REG**	**LUT**	**DSP**
	6.6 MB (47%)	459 K (43%)	346 K (67%)	1600 (45%)
DDPG (Percentages are derived wrt 3 SLRs on the device)	Pipeline Factor (*PI*)	Data Parallel Factor (*DP*)	Max. *TU* Parallelism ($r \times f$)	# SLR Constraint (Replay, learner)
	BS/2	2	32	(1,2)
	SRAM	**REG**	**LUT**	**DSP**
	9.2 MB (28%)	1035 K (58%)	708 K (81%)	2560 (43%)

(SLRs) components (i.e., dies) mounted on a passive Silicon Interposer [39]. During place and route of our design, cross-SLR routing results in long wires that reduces operating clock frequency. To better port our design to modern FPGAs composed of multiple SLRs, we limit the resource constraint to 1 SLR in our DSE for DQN [23] learners module, and 2 SLRs for DDPG [21] learners module, This helps in ensuring fast routing and higher clock frequency of the design.

4.3 Evaluation of RMM and Learner Module

4.3.1 Replay Management Module. In order to show the superiority of our proposed Replay Management Module (RMM) on FPGA, we compare against two baselines discussed in Section 4.1. Specifically, we compare the time for sampling, insertion and priority update of the implementations with increasing batch size. The results are shown in Fig. 8. Sampling is often the bottleneck since it cannot be hidden during the training process. Even with ~ 7× slower operating frequency, our RMM achieves up to 7.4× and 21× lower latency on batch sampling compared with optimized K-ary Sum Tree implementations using CPU and GPU, respectively. Up to ~ 40× (~ 125×) speedup is observed for insertion (update). The performance improvement are mainly from (1) on-chip Replay Management enables single-cycle data accesses to all the priority values. (2) RMM on FPGA ensures parallel accesses to multiple levels of the Sum Tree at any time during the training due to fine-grained pipeline scheduling of Replay accesses without memory bank conflicts. Note that the insertions take longer than replay updates on FPGA-accelerated RMM. This is due to the inevitable PCIe latency that occurs when transferring data from host memory to device memory for insertion into the replay Data Storage. However, it does not add extra overheads to the overall system throughput (*GPS*) because the latency is completely hidden by the learners training process.

4.3.2 Learners Module. We evaluate the speed up of learners acceleration using the execution time per gradient step for various batch sizes. We profile the training execution time per gradient step of an optimized Pthreads implementation. The baseline implementations are evaluated on the GPU. Fig. 9 shows the training execution time per gradient step of DQN [23] and DDPG [21] on the same benchmark environment CartPole [24]. GPU threads can concurrently process independent samples in a batch in forward and

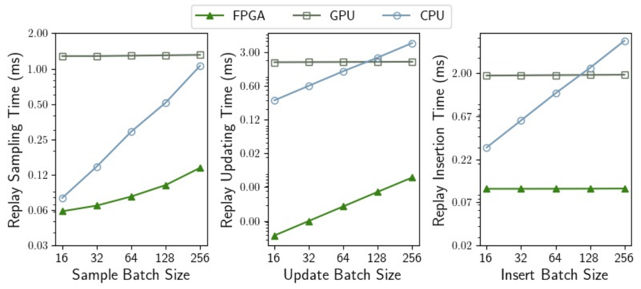


Figure 8: Scalability comparison of Replay implementations

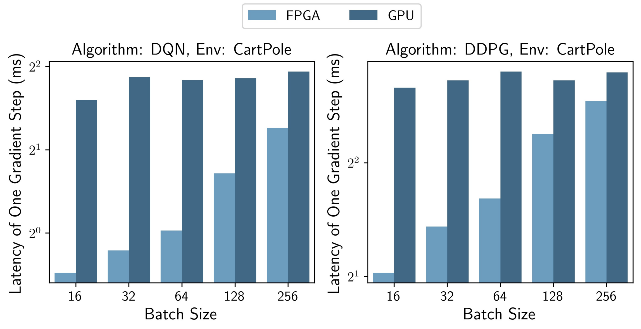


Figure 9: Comparison of learners performance

backward propagation and re-use the weights. The performance of GPU learner depends on the high data-reuse and high arithmetic intensity of large batch sizes. The high memory access latency with low data re-use in smaller batch size training severely hinder GPU performance as shown in Fig. 9. The external memory fetch dominates the execution while the amount of arithmetic operations cannot saturate all the 10K Cuda Cores. While GPU scales better to larger batch sizes, existing work has shown that RL algorithms trained using batch size larger than 256 are prone to converge to local optimums [37]. The FPGA Learner Module takes advantage of both batched data streaming in a pipelined manner and data parallelism across batches. We partition the weight buffers and the scratchpad memory (Sec. 3.2) to match the parallelism factor f in each layer. This ensures that we provide enough bandwidth and low-latency (single-cycle) memory access to match the rate of data consumption by the computing units. Overall, we observe consistent speedup ranging from 1.5× to 4.3× for all the batch sizes.

4.3.3 System Bottleneck Analysis. We plot the roofline models [27] of all the platforms, and the actual achieved performance under various mappings involved in each algorithm in Fig. 11. For CPU and GPU plots, the slope of the roof-line is the available DDR bandwidth and the horizontal line shows the peak performance of the device. For the FPGA plot, three roof-lines are plotted for different modules according to their effective bandwidth and theoretical peak performance. The first takeaway from Fig. 11 is that on CPU and GPU, low arithmetic intensity of replay operations and small MLP training time bounds their achievable performance by memory bandwidth. It suggests that simply increasing parallelism does not result in higher throughput. Note that the peak performance on the roofline assumes 100% hardware utilization and streaming memory access.

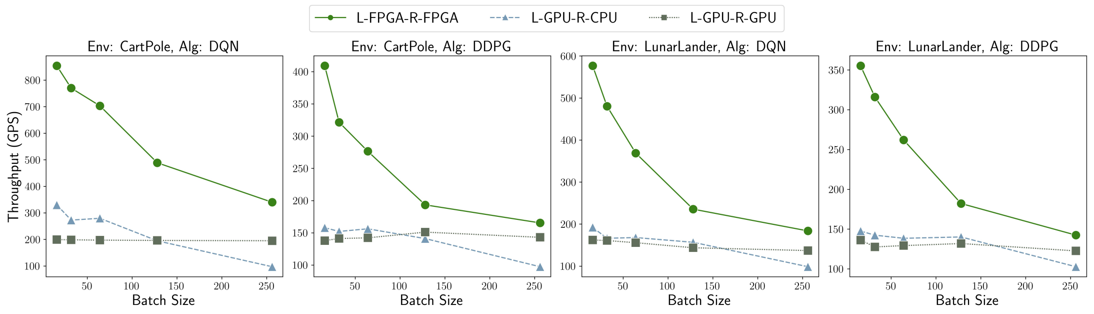


Figure 10: Comparisons of the overall system performance measured using *GPS*. The legend shows the mapping methodology, where L represents the learner and the R represents the Replay Buffer. For example, L-FPGA-R-CPU indicates a design that maps the learner onto the FPGA and the RMM onto the CPU.

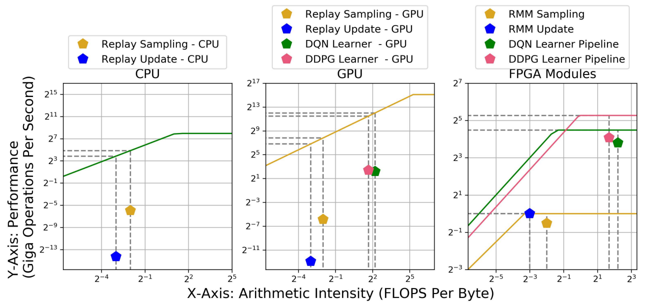


Figure 11: Roofline plots for all devices. The batch size is 256 in all the curves.

The gap between the actual achieved performance point with the roofline is due to (1) the actual implementation does not use up all threads/cores on CPU and GPU, and (2) tree traversal across levels requires non-streaming (high latency) accesses to discontinuous external memory locations (~140 cycles for CPU DDR4 [14], 80~150 cycles for GPU GDDR6 [15]). The second takeaway is that by allocating on-chip SRAM banks that provide sufficient effective bandwidth to the RMM and the Learner Module, our FPGA-based design prevents the replay operations and the learner from being bounded by the memory bandwidth. Our design also achieves high effective resource utilization observed from the low gap between the achieved performance point with the roofline. By removing the bottlenecks on CPU and GPU, our on-chip RMM design and Learner Module outperforms CPU-GPU and GPU-based implementations.

4.4 Evaluation of Overall System Throughput

Fig. 10 shows the comparison of the overall system throughout measured in *GPS*. We compare against two state-of-the-art implementations: [20] that uses binary Sum Tree for Replay management, and [41] that uses K-ary Sum Tree for Replay management. We fix the implementation of Replay management on CPU and map the learners onto CPU, GPU and FPGA. Our method achieves $1.3 \sim 3.7\times$ speedup among all the evaluated algorithms and RL environments compared with CPU-GPU mapping baselines. Our method achieves up to $1.2 \sim 4.3\times$ throughput improvement compared with the GPU baselines. In addition, we fix the learner implementations on the FPGA and compare the performance of a CPU-based Replay management versus our proposed RMM. The result shows that the GPS improves up to 263% just from mapping the replay onto FPGA using our RMM design.

5 RELATED WORK

Existing works improve the training throughput by simply increasing the number of actors and learners. GORILA [25] proposes a parallel architecture of DQN [23] to play Atari games [1]. RLlib [20] proposes high level abstractions for distributed reinforcement learning built on top of the Ray library [20]. [19] proposes parallel reinforcement learning using MapReduce [7] framework with linear function approximation. [41] proposes K-ary Sum Tree data structure to improve the performance of the Replay operations only on CPU. A few recent works have focused on hardware acceleration of RL algorithms. A FPGA implementation of Asynchronous Advantage Actor-Critic (A3C) algorithm is presented in [3]. In [34] and [35], a hardware architecture is developed to accelerate Trust Region Policy Optimization (TRPO) [32]. In [10], a CPU-FPGA architecture is proposed to accelerate Deep Deterministic Policy Gradient (DDPG) [21], which combines Deep Q-Learning with policy optimization methods. [22] proposes an accelerator for PPO, which utilizes separate modules for actor-critic networks.

Prior works on CPU and GPU focus on developing high level parallel framework for the RL algorithms without system level optimizations. Existing FPGA-based implementations only focus on specific RL algorithms, without a general framework for a wide range of RL algorithms or optimizing other components such as Replay Buffer Management. To address the low arithmetic intensity issue of acceleration on GPU, our work developed the first specialized on-chip RMM for general DRL, and uses hardware pipelining to completely hide the RMM latency.

6 CONCLUSION

In this work, we explored mapping of parallel RL onto heterogeneous platforms. Our profiling results suggested that the system bottleneck lies in the high latency to perform various operations used to update the Prioritized Replay Buffer. We proposed a Replay Management Module on FPGA such that the allocated computation resources are fully utilized despite the low arithmetic intensity of these operations. Our experiments show superior performance of both the Replay operations and the overall system compared with CPU-GPU and GPU-based mapping.

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

[2] Konstantinos Chatzilygeroudis, Roberto Rama, Rituraj Kaushik, Dorian Goepp, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2017. Black-box data-efficient policy search for robotics. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 51–58.

[3] Hyungmin Cho, Pyeongseok Oh, Jiyoung Park, Wookeun Jung, and Jaejin Lee. 2019. FA3C: FPGA-Accelerated Deep Reinforcement Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 499–513.

[4] Andrea Damiani, Giorgia Fiscaletti, Marco Bacis, Rolando Brondolin, and Marco D Santambrogio. 2022. BlastFunction: A Full-stack Framework Bringing FPGA Hardware Acceleration to Cloud-native Applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 2 (2022), 1–27.

[5] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. 2021. Utilizing cloud FPGAs towards the open neural network standard. *Sustainable Computing: Informatics and Systems* 30 (2021), 100520.

[6] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2020. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1014–1029.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[8] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2019. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. *CoRR* abs/1910.06591 (2019). arXiv:1910.06591 http://arxiv.org/abs/1910.06591

[9] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*. PMLR, 1407–1416.

[10] Ce Guo, Wayne Luk, Stanley Qing Shui Loh, Alexander Warren, and Joshua Levine. 2019. Customisable Control Policy Learning for Robotics. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 91–98.

[11] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[12] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. *CoRR* abs/1803.00933 (2018). arXiv:1803.00933 http://arxiv.org/abs/1803.00933

[13] Intel. 2017. Intel Stratix 10 MX FPGAs. https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html

[14] Intel. 2018. *SkyLake Specification*. https://www.7-cpu.com/cpu/Skylake.html

[15] Intel. 2021. *GPU Memory Latency's Impact, and Updated Test*. https://chipsandcheese.com/2021/05/13/gpu-memory-latencys-impact-and-updated-test/

[16] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2019. Recurrent Experience Replay in Distributed Reinforcement Learning.

[17] Vinod Kathail. 2020. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–174.

[18] Vasileios Leon, Kiamal Pekmestzi, and Dimitrios Soudris. 2021. Exploiting the Potential of Approximate Arithmetic in DSP & AI Hardware Accelerators. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 263–264.

[19] Yuxi Li and Dale Schuurmans. 2012. MapReduce for Parallel Reinforcement Learning. In *Recent Advances in Reinforcement Learning*, Scott Sanner and Marcus Hutter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–320.

[20] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. Ray RLLib: A Composable and Scalable Reinforcement Learning Library. *CoRR* abs/1712.09381 (2017). arXiv:1712.09381 http://arxiv.org/abs/1712.09381

[21] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. *CoRR* abs/1509.02971 (2016).

[22] Yuan Meng, Sanmukh Kuppannagari, and Viktor Prasanna. 2020. Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 19–27.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[24] Takao Moriyama, Giovanni De Magistris, Michiaki Tatsubori, Tu-Hoa Pham, Asim Munawar, and Ryuki Tachibana. 2018. Reinforcement Learning Testbed for Power-Consumption Optimization. *CoRR* abs/1808.10427 (2018). arXiv:1808.10427 http://arxiv.org/abs/1808.10427

[25] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *CoRR* abs/1507.04296 (2015). arXiv:1507.04296 http://arxiv.org/abs/1507.04296

[26] NVIDIA. 2022. Deep Learning Performance Documentation. https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.

[27] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. 2014. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 76–85.

[28] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. *Advances in neural information processing systems* 29 (2016), 4026–4034.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[30] H. Robbins and S. Monro. 1951. A stochastic approximation method. *Annals of Mathematical Statistics* 22 (1951), 400–407.

[31] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized Experience Replay. http://arxiv.org/abs/1511.05952 cite arxiv:1511.05952Comment: Published at ICLR 2016.

[32] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015. Trust Region Policy Optimization. *CoRR* abs/1502.05477 (2015). arXiv:1502.05477 http://arxiv.org/abs/1502.05477

[33] Lorenzo Servadei, Jin Hwa Lee, José A Arjona Medina, Michael Werner, Sepp Hochreiter, Wolfgang Ecker, and Robert Wille. 2022. Deep Reinforcement Learning for Optimization at Early Design Stages. *IEEE Design & Test* (2022).

[34] Shengjia Shao and Wayne Luk. 2017. Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–6.

[35] Shengjia Shao, Jason Tsai, Michal Mysior, Wayne Luk, Thomas Chau, Alexander Warren, and Ben Jeppesen. 2018. Towards hardware accelerated reinforcement learning for application-specific robotic control. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–8.

[36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[37] Adam Stooke and Pieter Abbeel. 2018. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811* (2018).

[38] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. 2019. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog* 2 (2019).

[39] xilinx. 2012. Large FPGA methodology guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf

[40] Xilinx. 2021. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

[41] Chi Zhang, Sanmukh Rao Kuppannagari, and Viktor K Prasanna. 2021. Parallel Actors and Learners: A Framework for Generating Scalable RL Implementations. *arXiv* (2021). arXiv:2110.01101 [cs.LG]