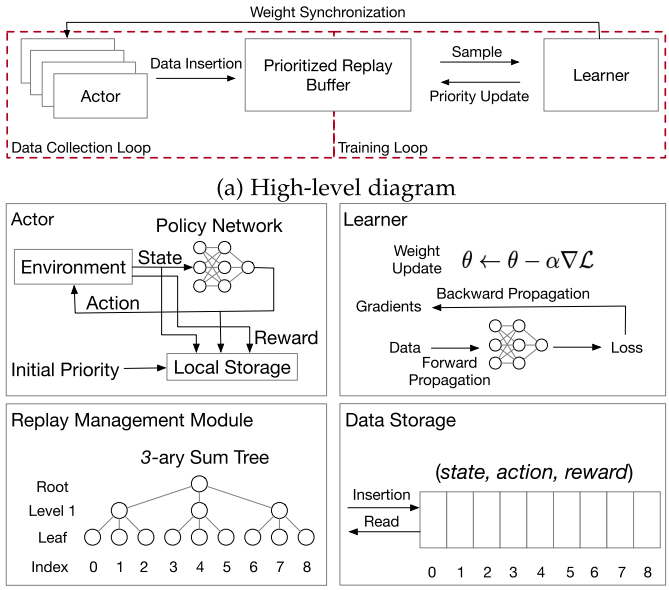
A Framework for Mapping DRL Algorithms With Prioritized Replay Buffer Onto Heterogeneous Platforms

Chi Zhang, Yuan Meng, and Viktor Prasanna, *Fellow, IEEE*

***Abstract*—Despite the recent success of Deep Reinforcement Learning (DRL) in self-driving cars, robotics and surveillance, training DRL agents takes tremendous amount of time and computation resources. In this article, we aim to accelerate DRL with Prioritized Replay Buffer due to its state-of-the-art performance on various benchmarks. The computation primitives of DRL with Prioritized Replay Buffer include environment emulation, neural network inference, sampling from Prioritized Replay Buffer, updating Prioritized Replay Buffer and neural network training. The speed of running these primitives varies for various DRL algorithms such as Deep Q Network and Deep Deterministic Policy Gradient. This makes a fixed mapping of DRL algorithms inefficient. In this work, we propose a framework for mapping DRL algorithms onto heterogeneous platforms consisting of a multi-core CPU, a GPU and a FPGA. First, we develop specific accelerators for each primitive on CPU, FPGA and GPU. Second, we relax the data dependency between priority update and sampling performed in the Prioritized Replay Buffer. By doing so, the latency caused by data transfer between GPU, FPGA and CPU can be completely hidden without sacrificing the rewards achieved by agents learned using the target DRL algorithms. Finally, given a DRL algorithm specification, our design space exploration automatically chooses the optimal mapping of various primitives based on an analytical performance model. On widely used benchmark environments, our experimental results demonstrate up to 997.3× improvement in training throughput compared with baseline mappings on the same heterogeneous platform. Compared with the state-of-the-art distributed Reinforcement Learning framework RLlib, we achieve 1.06× ∼ 1005× improvement in training throughput.**

***Index Terms*—Deep reinforcement learning, design space exploration, FPGA, GPU, heterogeneous platform, prioritized replay buffer.**



(b) Details of each component. The Prioritized Replay Buffer is further decomposed into Replay Management Module and Data Storage.

Fig. 1. Overview of existing parallel reinforcement learning frameworks [5].

I. Introduction

REINFORCEMENT Learning (RL) [1] is widely used in many application areas including self-driving cars, robotics, surveillance, etc. [2], [3], [4]. In RL, an agent iteratively interacts with an environment to improve its policy such that the expected accumulated reward along the trajectory is maximized. The policy is represented as a lookup table in class RL [1] while it is represented as a neural network in Deep Reinforcement Learning (DRL). Training DRL agents is extremely time consuming as it requires a large number of data by interacting with the environment and gradient updates to update the policy represented as neural networks to converge (e.g., AlphaGo [4]). The state-of-the-art parallel DRL frameworks [5], [6] employ a general architecture consisting of parallel actors, a centralized learner and a Prioritized Replay Buffer [7] as shown in Fig. 1(a). Parallel actors concurrently collect data from the environment and insert the data into the Prioritized Replay Buffer. The centralized learner samples data from the Prioritized Replay Buffer and performs stochastic gradient descent (SGD) [8] to update the policy. The new priorities after learning are updated by the Prioritized Replay Buffer. 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 Buffer is stored in a K-ary Sum Tree data structure [9] that can

Manuscript received 10 July 2022; revised 27 February 2023; accepted 18 March 2023. Date of publication 5 April 2023; date of current version 8 May 2023. This work was supported in part by NSF under Grant CNS-2009057 and OAC-2209563, and in part by equipment and donations from AMD Xilinx. Recommended for acceptance by M. D. Santambrogio. *(Chi Zhang and Yuan Meng contributed equally to this work.) (Corresponding author: Chi Zhang.)*

Chi Zhang is with the Department of Computer Science, University of Southern California, Los Angeles CA 90089 USA (e-mail: zhan527@usc.edu).

Yuan Meng and Viktor Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles CA 90089 USA (e-mail: ymeng643@usc.edu; prasanna@usc.edu).

Digital Object Identifier 10.1109/TPDS.2023.3264823

perform sampling and priority update in $O(\log N)$ time, where N is the total number of data points in the Replay Buffer. Heterogeneous platforms consisting of CPU, FPGA and GPU [10] are promising to accelerate DRL algorithms. This is because the speed of running the key DRL primitives significantly varies among different DRL algorithms. For instance, training neural network with small size on FPGA is faster than on CPU and on GPU, whereas training neural network with large size is faster on GPU than on CPU and on FPGA as shown in Section VII-C. Sampling with small batch size is faster on FPGA than on CPU and on GPU, whereas sampling with large batch size is faster on GPU than on CPU and on FPGA as shown in Section VII-C.

In this work, we propose a framework for mapping DRL algorithms onto heterogeneous platforms consisting of a multi-core CPU, a GPU and a FPGA. CPU is suitable for computation with complex control flows; GPU is suitable for computation with large data parallelism and FPGA is suitable for computation with abundant memory accesses and fine-grained data dependencies. We propose separate accelerators for each primitive on CPU, GPU and FPGA. We propose design space exploration that chooses the optimal mapping in a given DRL algorithm such that the overall training throughput is maximized. In addition, we relax the data dependency between priority update and sampling performed in the Prioritized Replay Buffer to hide the data transfer latency among the heterogeneous accelerators. Specifically, our key contributions are:

- On the CPU, we utilize OpenMP [11] to exploit data parallelism in sampling and priority update of Prioritized Replay Buffer. We use PyTorch [12] for neural network training.
- On the GPU, we develop a customized CUDA kernel for sampling and priority update based on parallel sum reduction. We use PyTorch with CuDNN [13] backend for neural network training.
- On the FPGA, we develop a generic throughput-oriented learner module that exploits both neural network model parallelism and data parallelism. We develop a generic accelerator template for the Replay Management Module (RMM) that exploits bank parallelism and is scalable to large batch size. Our proposed RMM supports parallel insertion, parallel sampling and parallel priority updates of the K-ary Sum Tree [9]. We optimize the performance of the on-chip data access using:
  - 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.
- To hide the latency caused by data transfer between heterogeneous accelerators used for training, we relax the data dependency between priority update and sampling in the Prioritized Replay Buffer. We empirically show that the performance discrepancy of the trained agents between DRL algorithms with and without data dependency is negligible on widely used benchmark environments.
- We propose a design space exploration and design automation workflow that optimally chooses the mapping of each component onto a CPU-GPU-FPGA heterogeneous system given an arbitrary DRL algorithm.
- For widely used DRL algorithms including DQN [14] and DDPG [15], the mapping generated by our framework demonstrate up to 997.3× speedup in training throughput compared with baseline mappings.

II. Background

A. WorkFlow Overview

We show a generic view of existing parallel RL workflow [5], [6] in Fig. 1(a). It contains a data collection loop and a training loop. The main component of the data collection loop is the actor and the main component of the training loop is the learner. The Prioritized Replay buffer is used to store data collected by the actor and to sample data for the learner. The details of each component are shown in Fig. 1(b).

1) Data Collection Loop: The data collection loop is inside each actor as shown in Fig. 1(b). 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 an 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 a reward r. The policy network computes the current loss P as the initial priority using $(s, a, s,' r)$. After observing the next state s', the policy network computes the next action a' and the data collection loop continues until the end of the training. 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 the Prioritized Replay Buffer [7]. The functionality of local storage is to reduce the frequency of adding data into the Prioritized Replay Buffer. This reduces the synchronization frequency when multiple actors add their data concurrently.

2) Training Loop: The training loop occurs between the learner and the Prioritized Replay Buffer. Following [5], [6], we use a centralized learner to perform policy updates. At each step, the learner i) samples a batch of indices via the Replay Management Module using the probability distribution proportional to the priorities; 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 (SGD) [8]; iv) sets the priorities of the sampled batch data to the new priorities after SGD update via the Replay Management Module.

Data dependency: It is worth noticing that the priorities of the sampled batch data must be updated with the new priorities before the sampling of the next batch of indices. Otherwise, the next batch of indices will be sampled from the old probability distribution and it potentially causes convergence issues.

3) Prioritized Replay Buffer: Prioritized Replay Buffer [7] sits between the data collection loop and the training loop. It has been proposed to sample data with probability proportional to the current loss to speed up training [7]. It consists of a

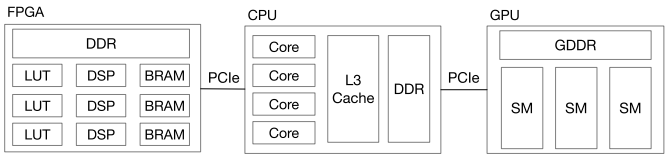


Fig. 2. Diagram of target heterogeneous platform.

Data Storage and a Replay Management Module (RMM). **Data Storage** is used to store data produced by the actors. At runtime, data collected from the actors is inserted into the next available location 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. Prioritized Replay Buffer supports **sampling** data to perform training and **priority update** after each training iteration. **Sampling** from the Prioritized Replay Buffer decides which samples (indices) are used for training the neural network. During sampling, a data point x_i is selected according to the probability distribution $\Pr(i) = P(i)/\sum_i P(i), i \in [0, S_E)$, where $P(i)$ is the priority of data point i and 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 $(cdf(i) = \sum_{j=1}^{i} \Pr(j), i \in [0, S_E))$ 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:

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

Such index i is known as **Prefix Sum Index**. **Priority Update** requires updating the current priorities using newly computed priorities. This operation is performed after each training iteration.

Internally, RMM is implemented as a **K-ary Sum Tree** [9] as shown in Fig. 1(b). 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. The worst case time complexity of sampling and priority update is $O(\log N)$, where N is the number of leaf nodes in the tree [9].

B. Target Heterogeneous Platform

We show a high level diagram of our target heterogeneous platform used to accelerate DRL algorithms in Fig. 2. It consists of a multi-core CPU, a GPU and a FPGA. Each core inside the CPU contains a L1 and L2 cache and all the cores share the same L3 cache and the DDR memory. The GPU consists of multiple streaming multiprocessors (SM) and a GDDR. Each SM consists of a shared memory and multiple streaming processors. The FPGA consists of a number of re-configurable compute units (DSP), arithmetic units (ALU or LUT), and large distributed on-chip SRAM.

C. Performance Metric and Challenges

The performance metric is **training throughput** defined as the number of gradient steps performed by the learner per second (GPS). Although the data collection throughput also affects the convergence speed of DRL algorithms, our framework allocates fixed resources for each actor (e.g. 1 CPU core/actor) so that the data collection throughput is in linear w.r.t the number of actors. By fixing the data collection throughput, the convergence speed is only affected by the training throughput [5], which is the primary objective to maximize in our framework.

III. Related Work

GORILA [16] proposes a parallel architecture of DQN [14] to play Atari games [17]. RLlib [18] proposes high level abstractions for distributed reinforcement learning built on top of the Ray library [18]. [19] proposes parallel reinforcement learning using MapReduce [20] framework with linear function approximation. [9] proposes K-ary Sum Tree data structure to improve the performance of the Replay operations. A few recent works have focused on hardware acceleration of DRL algorithms. A FPGA implementation of Asynchronous Advantage Actor-Critic (A3C) algorithm is presented in [21]. In [22] and [23], a hardware architecture is developed to accelerate Trust Region Policy Optimization (TRPO) [24]. In [25], a CPU-FPGA architecture is proposed to accelerate Deep Deterministic Policy Gradient (DDPG) [15], which combines Deep Q-Learning with policy optimization methods. [26] proposes an accelerator for PPO, which utilizes separate modules for actor-critic networks. The key advantages and disadvantages of these existing works are summarized in Table I. Prior works either focus on developing parallel paradigm for the DRL algorithms without hardware acceleration details ([18], [19]), or is limited to a specific DRL algorithm acceleration ([21], [22], [23], [26]). Moreover, none of the existing works efficiently support memory-bound primitives such as Prioritized Replay Buffer. Due to memory bottleneck, this creates a performance gap between the achieved throughput by these existing frameworks and the peak throughput provided by homogeneous platforms.

In our previous work [27], we proposed a mapping for DRL algorithms with Prioritized Replay Buffer onto a CPU-FPGA heterogeneous platform. The design achieves state-of-the-art training throughput when the Prioritized Replay Buffer and the learner fit onto the on-chip resources of the FPGA. However, it fails to handle cases where the on-chip resources are not enough for the Prioritized Replay Buffer and the learner. To achieve state-of-the-art training throughput in broader scenarios, in this paper, we propose a framework for mapping DRL algorithms with Prioritized Replay Buffer onto CPU-GPU-FPGA heterogeneous platforms to achieve superior training throughput for given DRL algorithms and their input parameters.

IV. Accelerator Design for Primitives

In this section, we discuss the accelerator design of various primitives including actor, neural network training, Prefix Sum

TABLE I
SUMMARY OF RELATED WORK

Framework	Target DRL Algorithm	Available Hardware Mapping (Learner-RMM-Data Storage)	Advantage	Disadvantage
GORILA [16]	DQN [14]/DDPG [15]	CPU-CPU-CPU/GPU-CPU-CPU	Asynchronous Update	Asynchronous SGD
ApeX [5]	DQN [14]/DDPG [15]	CPU-CPU-CPU/GPU-CPU-CPU	Achieves high rewards	Slow for small NN
RLlib [18]	Almost all	CPU-CPU-CPU/GPU-CPU-CPU	Supports a wide range of algorithms	Slow for small NN
[23]	TRPO [24]	FPGA-N/A-FPGA	Specialized hardware for fast DNN training	Supports only one algorithm
[25]	DDPG [15]	FPGA-FPGA-FPGA	Specialized hardware for fast DNN training	Supports only one algorithm, no optimization for replay

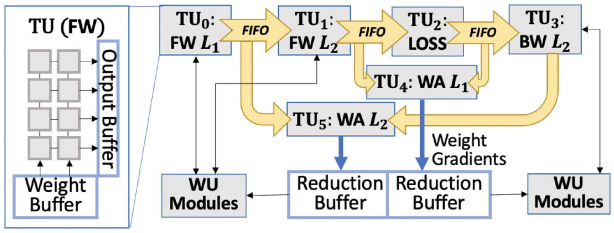


Fig. 3. Learner Module Architecture: An example pipeline for a 2-layer neural network. L_i means i^{th} Layer.

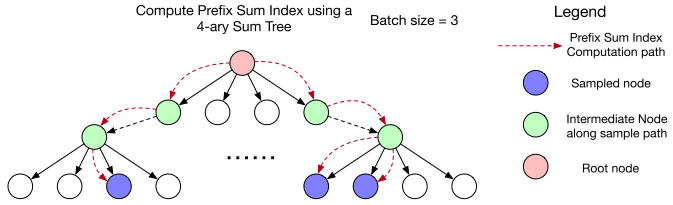


Fig. 4. Illustration of computing the Prefix Sum Index.

Index computation and priority update on multi-core CPU, GPU and FPGA.

A. Actor

Each actor is mapped onto a CPU core that performs environment emulation and neural network inference. Accelerating environment emulation is out of the scope of this paper. Our benchmark environments utilize existing open-source software in OpenAI gym [17]. Following [5], neural network inference is performed on a CPU core after each environment interaction.

B. Neural Network Training

1) Algorithm Description: We consider Stochastic Gradient Descent (SGD) [8] training algorithms composed of forward propagation (FW), loss computation (LOSS), backward propagation (BW), weight aggregation (WA) and weight update (WU) steps.

2) Training on CPU and GPU: We use PyTorch [12] to train neural networks on CPU and GPU. On CPU, PyTorch utilizes OpenMP [11] to exploit intra-operation parallelism. On GPU, PyTorch utilizes CuDNN backend to exploit massive SIMT of GPU.

3) Training on FPGA: We carefully design the Learner Module with the goal of minimizing the execution time of each gradient step. 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 a learner pipeline as shown in Fig. 3.

The learner pipeline for a L-layer neural network model consists of $n = 3 \times L$ stages: FW through L layers of policy and value networks, computing LOSS, BW through $(L-1)$ layers, and WA for all the L weight tensors. Each of these stages is mapped to a unique Tensor Unit, TU. Each TU is a systolic array of Multiply-Accumulate elements. We express all the FW, BW and WA as general matrix multiplication (For CNN, we apply the im2col algorithm). A TU for FW (BW) exploit parallelism both among different output neurons (access different weights in parallel). A TU for WA takes the activations generated by FW and activation gradients generated by BW, and outputs the weight gradients for accumulation. It exploits parallelism along the neurons in two adjacent layers. The WU modules update the weight buffers in FW (BW) TUs after the accumulation of weight gradients from all the samples in a batch is completed. To realize data streaming between stages in a pipeline, these modules (TUs) are connected by FIFOs.

Let B denotes the batch size and DP (Data Parallel factor) denotes the number of pipelines described in Fig. 3. A total number of DP such pipelines are allocated to exploit data parallelism in a batch. Each pipeline processes a sub-batch of $\frac{B}{DP}$ data points, and a reduction buffer is used to average the weight gradients obtained in each pipeline. Conceptually, for a given batch size, higher DP achieves higher throughput for FW-BW-WA stages, but causes larger time or area overhead for reduction over all the pipelines. High DP can also lead to low effective hardware utilization in each pipeline if the resulting sub-batch size is too small to saturate the concurrency provided by all the stages. The Data Parallel Factor (DP) needs to be carefully chosen to achieve the best performance under the constraints of a given FPGA device. The design space exploration process for searching the optimal DP is described in Section VI-A2.

C. Prefix Sum Index Computation

1) Algorithm Description: We illustrate an example of finding Prefix Sum Index in Fig. 4 and Algorithm 1. The inputs to the algorithm are the current node values v and the batch size B. The outputs are B sampled indices using the probability distribution according to the node values. First, we compute the Prefix Sum as rand()×v[root], where rand() samples a random number uniformly from [0,1]. In order to find the Prefix Sum Index, we traverse from the root node to the leaf node level by level as shown in Fig. 4. During the traversal of each level, we

Algorithm 1: Prefix Sum Index Computation on CPU.

```
1: Input: node priority values v, batch size B
2: root = getRoot();
3: #pragma omp parallel for
4: for i = 0; i < B; i++ do
5:    prefixSum[i] = rand()×v[root] {Sample prefix sum}
6:    node = root; {start from root node}
7:    currentVal = prefixSum[i];
8:    while !isLeaf(node) do
9:       for childNode in getChildNodes(node) do
10:         cumulativeSum = currentVal - v[childNode];
11:         if cumulativeSum ≤ 0 then
12:            break;
13:         currentVal = cumulativeSum;
14:      node = childNode;
```

find the child node such that the cumulatively sum of the values stored in the child nodes are greater or equal to the target prefix sum value and continue to expand on that child node until it is the leaf node, which is the node to sample. 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 and K is the number of child nodes of each parent in the tree.

2) Acceleration on CPU: Prefix Sum Index Computation is a read-only operation. Thus, the Prefix Sum Index of different data inside a batch can be computed fully in parallel. As shown in Algorithm 1, it is realized via OpenMP [11] directives on CPU by exploiting thread-level data parallelism via shared memory.

3) Acceleration on GPU: Accelerating batch Prefix Sum Index computation on GPU is similar to CPU via thread-level data parallelism. Instead of using a OpenMP [11] directives to decorate the for loop, we compute the index for data parallelism on GPU inside a batch using CUDA thread and block index:

$$\text{i} = \text{threadIdx.x} + (\text{blockIdx.x} \times \text{blockDim.x}) \quad (2)$$

Due to kernel launch overhead, the advantage of accelerating Prefix Sum Index computation on GPU is only observable when the batch size B is large. For small batch sizes, it is even slower than running on CPU using a single thread.

4) Acceleration on FPGA: The primary objectives of the FPGA-based RMM design are: (1) providing sufficient 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, and overlapping computation with data accesses using hardware pipelining. To achieve objective (1), we store the complete K-ary Sum Tree data structure using the on-chip SRAM on the FPGA. A typical Prioritized Replay Buffer contains 1 million data points [14]. The number of nodes in a K-ary Sum Tree is at most 2 million when $K = 2$, which can fit the available on-chip memory of most state-of-the-art FPGAs [28], [29]. The nodes of the Sum Tree are ordered by the tree level. Each SRAM bank provide single-cycle access to any data element through a read/write port. Nodes on different tree levels are stored in separate SRAM banks. In any FPGA cycle, all the sampling and update stages can concurrently access different tree levels

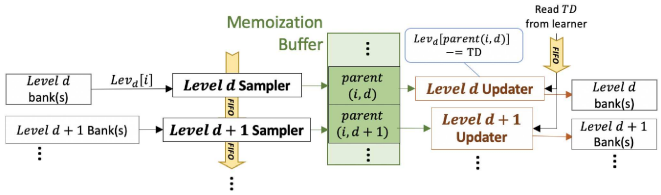


Fig. 5. Replay Samplers and Updaters.

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. Suppose H is the height of the tree. A sampling pipeline sequentially process prefix-sum on H tree levels using H pipeline stages. As shown in Fig. 5, the samplers are connected by FIFOs, each responsible for traversing up to K sibling nodes in the same level. The parallelism provided by all the samplers is fully utilized for concurrently processing different samples at different tree levels. At each sampler, a register is used to track the accumulated priority while traversing the child nodes. The target child node index and remaining priority values are passed onto the next sampler when the threshold shown in line 11 of Algorithm 1 is reached. At the leaf level, the index where the accumulated priority sum reaches the target prefixSum (line 5 of Algorithm 1) is the minimum index i satisfying (1)); Thus, the data point at index i should be sampled.

We further exploit data parallelism among different banks by partitioning a sum tree into S sub-trees rooted at the nodes in the first $\log_K^S$ tree levels, and assign a pipeline to each sub-tree. The shared priority prefix sums managed by the first $\log_K^S$ tree levels are broadcast to all the pipelines. The rest of the tree nodes are local to each pipeline and stored in separate banks, such that data points in a batch can be processed concurrently by different pipelines if their target prefix-sum fall into different sub-trees. We refer to the number of RMM sampling(update) pipelines, S, as the bank parallel factor.

HLS (High level synthesis)-generated [30] floating point accumulator takes multiple cycles to compute [31], 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 register for storing the tree root. 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 its K child values. This representation scheme does not affect the sampling outcome compared with using floating point representation.

D. Priority Update

1) Algorithm Description: The inputs of one priority update are a batch of indices, a batch of new priorities associated with

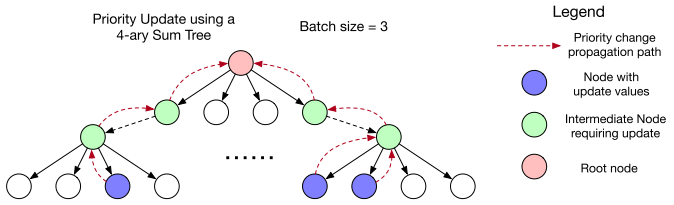


Fig. 6. Illustration of updating the K-ary Sum Tree.

Algorithm 2: Priority Update on CPU.

1: **Input:** a batch of indexes idx, a batch of new priorities p, node values v.
2: `#pragma omp parallel for`
3: **for** $i = 0; i < B;$ i++ **do**
4: node = getNodeFromIndex($idx[i]$); {get node}
5: $\Delta[i]$ = getValue(node) - $p[i]$; {priority change}
6: **while** node != getRoot() **do**
7: `#pragma omp atomic`
8: v[node] += $\Delta[i]$;
9: node = getParent(node);

each index and the node values. The output is the node values with updated priorities. An example with batch size equal to 3 is shown in Fig. 6. To update the priority of each data, we update the node values from the leaf node to the root node as illustrated by the propagation path in Fig. 6. We compute the change of priority values at the leaf node and iteratively update the intermediate nodes. The time complexity of priority update of each data is $O(\log_K N)$, where N is the number of elements in the replay buffer and K is the number of child nodes of each parent in the tree.

2) Acceleration on CPU: We utilize thread-level data parallelism via OpenMP directives for acceleration on CPU as shown in Algorithm 2. As opposed to Prefix Sum Index computation, priority updates of different data in a batch share the same underlying data structure. Thus, it is important to make sure no data race happens during the parallel update. To do so, we use OpenMP atomic directive as shown in line 7 in Algorithm 2.

Remarks: It is worth noticing that the data parallelism of priority updates reduces as the computation level approaches the root and it is fully serialized at the root level. The performance of priority update on CPU is hindered due to poor cache performance. This is because when a thread updates the shared memory in its cache, the other threads on different cores have to invalidate their cache and load the data from main memory. This significantly impairs the performance of thread-level parallelism on CPU.

3) Acceleration on GPU: The thread id of data parallelism on GPU is computed in (2). We use `atomicAdd` to perform atomic updates on GPU. The performance of GPU-based acceleration is also limited by the expensive, frequent and unavoidable atomic operations when updating the priority values at the root level of the tree.

4) Acceleration on FPGA: For the FPGA-accelerated priority update operations, in each pipeline, H updaters concurrently update the sum values at each level using the loss obtained at the actors or the learners. 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 $H-1$ node indices for each data point, as its $H-1$ parents during sampling are tracked, where H is the number of updaters.

On CPU and GPU, low arithmetic intensity of the priority update operation limits their achievable performance by external memory bandwidth. Also, updating across levels requires non-streaming (high latency) accesses to discontinuous external memory locations (∼140 cycles for CPU DDR4 [32], 80∼150 cycles for GPU GDDR6 [33]). We exploit the on-chip SRAM on FPGA to circumvent these limitations.

TABLE II
PROCESSING UNIT FOR ALL THE DRL COMPONENTS

Processing Unit	Input Queue Data	Processing Kernel	Output Queue Data
Actor	DNN Weights	Environment Step & DNN Inference	Samples
Learner	Samples	DNN Training	DNN Weights
RMM	TD Error from DNN Inference or Training	Priority Update or Priority Sampling	Indices
Data Storage	Samples or Indices	Data Insertion or Data Sampling	Samples

V. OVERALL SYSTEM DESIGN

Given the optimized accelerators for each primitive, the objective of the overall system design is to i) relax the data dependency as noted in Section II-A2 to maximize the training throughput; ii) design efficient data transfer mechanism between heterogeneous platforms such that the data transfer latency is completely hidden.

A. Data-Dependency Relaxed Training Loop

As mentioned in Section II-A2, sampling next batch cannot be performed until the new priorities of the previous batch are updated. In this case, sampling, training and priority update are executed sequentially. If this data dependency in the training loop can be removed, sampling, training and priority update can be executed in parallel and it will significantly improve the training throughput. However, it is important that the DRL performance degradation caused by removing the data dependency is negligible in terms of rewards, which is verified by the experimental results in Section VII-B

B. Data Transfer System Design

We view each key component in a DRL system as a Processing Unit composed of an input queue, a processing kernel, and an output queue. The detailed Processing Unit compositions of each key component in a DRL is shown in Table II. Each Processing Unit is hosted by a CPU thread, while the processing kernel can be mapped to an accelerator (GPU or FPGA) in a heterogeneous system. The goal of our data transfer system is to enable fast and atomic data transfers both within a Processing Unit (e.g.,

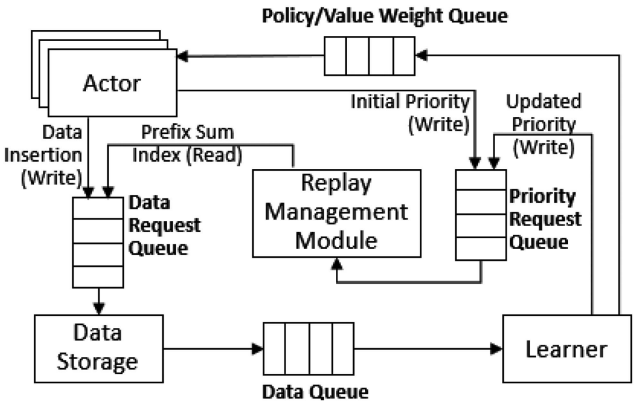


Fig. 7. Interactions among various Processing Units.

communication between the host CPU thread and the processing kernel on an accelerator), and among different Processing Units.

The interactions between Processing Units is shown in Fig. 7. The synchronization inside the Learner and the Prioritized Replay Buffer are managed by the accelerators they are mapped to. Let N be the number of Actor threads running on CPU. We allocate 3 more CPU threads, one for the Learner, one for the RMM and one for the Data Storage. These CPU threads are responsible for coordinating data transfer between the Actors, the Learner and the Data Storage. Specifically, First-in-First-out (FIFO) queues are used to realize concurrent and asynchronous communication among the host threads. Each queue has a producer end (tail) and a consumer end (head). For queues with multiple producers (e.g., the Data Storage Request Queue is shared by all the Actor threads and the RMM thread), a mutex is used to ensure atomic access to the tail. When two components connected by a queue are both mapped to the host CPU, the data communication between them is simply implemented using a shared queue described as above. When two components connected by a queue are mapped to the same accelerator device (e.g. both RMM and Learner are mapped to an FPGA), the queue is implemented using the on-chip resources of the accelerator. When two components connected by a queue are mapped to a CPU and an accelerator, the shared queue only stores the pointers of the data. The actual data transfer from the host to the device and from the device to the host is performed inside each Processing Unit.

VI. Design Space Exploration

The objective of design space exploration (DSE) is to maximize the system training throughput in terms of the number of gradient steps performed per second by mapping each component depicted in Fig. 7 onto appropriate devices. This section describes our DSE workflow through three steps (Fig. 8):

1) *Profiling of the Performance of the Primitives.* Given a DRL algorithm, it outputs the performance of each primitive under various mapping assumptions using an analytical performance model (on FPGA) or by profiling using real hardware (on CPU and GPU). The results populate a Primitive-Platform Performance Table. It also determines the required hardware parameters on the FPGA device.
2) *Mapping of Primitives:* It constructs the complete mapping space by taking inputs from the performance table, prunes the mapping space and outputs the optimal primitive mapping that achieves the highest training throughput.
3) *Template Instantiation:* It takes the primitive mapping results and hardware parameters as inputs, and populate a host template using a primitive code base to generate an end-to-end implementation.

A. Profiling the Performance of the Primitives

The given parameters of DRL algorithms include: i) RMM: the depth D, the number of child nodes of a parent node F and the total number of data points of the Prioritized Replay Buffer; ii) learner: the batch size B used in training; iii) learner: the architecture of the neural network. Both the RMM and the learner can be mapped onto either CPU, GPU and FPGA.

1) Profiling of Primitive on CPU and GPU: On CPU and GPU, we profile the execution time of a single batched neural network training step $T_{train}^{CPU(GPU)}$, batched Prefix Sum Index computation $T_{prefix}^{CPU(GPU)}$, and batched priority update $T_{priority}^{CPU(GPU)}$. Profiling on the GPU are performed under the assumption that only one of the RMM or the learner is accelerated by the GPU. It indicates that all the available GPU resources can be allocated to exploit its maximum data parallelism. This means that the GPU profiling results represent a Processing Kernel's peak performance that can be obtained on the GPU, and the actual performance may be lower if two Processing Kernels sequentially share the same GPU resource (i.e., when both learner and RMM operations are mapped to the GPU).

2) Architecture Exploration on FPGA: On FPGA, we develop cycle-accurate performance models for the RMM and the learner based on their input parameters. Using these models, we obtain the optimal hardware configurations that maximizes the training throughput and use the results to guide the rest of the DSE.

In modern multi-die FPGAs, an FPGA chip is built by a manufacturing process that combines multiple Super-Logic Regions (SLRs) components (i.e., dies) mounted on a passive Silicon Interposer [28]. 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 for the RMM and the rest of them for the learner. This helps in ensuring fast routing and higher clock frequency of the design.

The execution time of Prefix Sum Index computation with batch size B can be computed as $\frac{B}{S} \times F$ FPGA cycles, and the execution time of priority update with batch size B can be computed as $\frac{B}{S}$ FPGA cycles, where S is the bank parallel factor as mentioned in Section IV-C4. We set the bank parallel factor S to be proportional to the batch size for scalable acceleration. To fully utilize the parallelism provided by H stages, if $B \leq H$, we

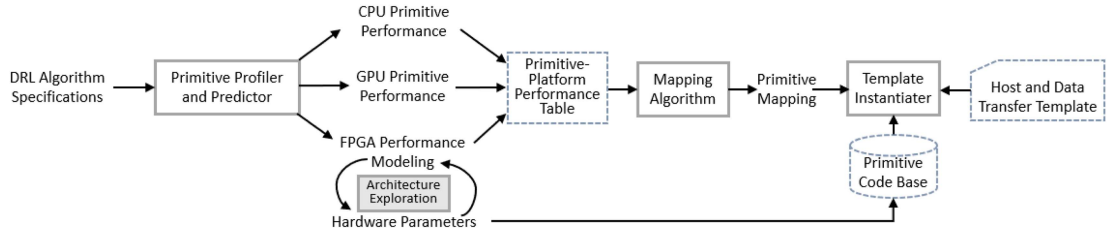


Fig. 8. Design space exploration and automation workflow.

use only a single pipeline ($S = 1$). As the batch size B increases, we increase the S until reaching the resource bound in an SLR.

The overall learner latency $T_{learner} = T_{pipeline} + T_{WU}$ of a single gradient step is determined by DP (number of learner pipelines), $PI = \frac{B}{DP}$ (the sub-batch size processed by each pipeline), and computation resources allocated to the pipeline stages. To minimize $T_{learner}$, our architecture exploration methodology follows two steps:

>a). Pipeline Local Optimization: The latency of a pipeline is computed as $T_{pipeline} = (PI + n - 1) \times T_{max}$. The performance is bounded by the latency of the slowest pipeline stage, $T_{max} = max_{i=1}^{n} T_i$. 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 computation resources (DSP units) to each Tensor Unit TU proportional to the number of multiply-add operations in each DNN layer propagation. Specifically, we derive a ratio $r_1 \ldots r_n$ for n stages, normalized to the stage with minimum number of operations. The total number of DSPs allocated to i-th Tensor Unit $TU_i, i \in [1 \ldots n]$) 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).

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 B, we determine the optimal combination of PI and DP by searching for all possible combinations in B 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) Primitive-Platform Performance Table: The output of the Primitive Profiler is a 3×2 table. The 2 columns in the table refer to the RMM and the Learner. The 3 rows refer to the 3 devices. Each table entry stores the throughput of processing a batch of data, $TP_{primitive}^{platform}$, where $primitive$=RMM or Learner. Note that $TP_{primitive}^{platform} = B/T_{primitive}^{platform}$, where B is the batch size and $T_{primitive}^{platform}$ is the total execution time of the primitive on the platform (for RMM, this is the sum of sampling and update execution time in each DRL gradient step).

Algorithm 3: Mapping Algorithm.

Input: Primitive-Platform Performance Table $PTable$, where each table entry TP_i^j denotes the achieved throughput of primitive i on device j.
Output: Mapping assignment vector D, where each entry D_i denotes the optimal device for performing primitive i.
`# Step 1,2: Primitive Mappings`
for i in [Learner, RMM] **do**
  $D_i = argmax_j\{TP_i^j\}$
  if $D_{\text{Learner}} ==$ GPU and $TP_{\text{RMM}}^{\text{FPGA}} \geq TP_{\text{Learner}}^{\text{GPU}}$ **then**
    $PTable$.remove($TP_{\text{RMM}}^{\text{GPU}}$)
Output $D_{\text{Learner}}, D_{\text{RMM}}$
`# Step 3: Memory Component Mapping`
Initialize $D_{\text{Data Storage}}$; min _ traffic$\leftarrow \infty$
$C_{\text{RMM}} \leftarrow B$; $C_{\text{Learner}} \leftarrow B \times E$; $C_{\text{Actor}} \leftarrow N_{actor} \times E$
for i in [Learner, Actors, RMM] **do**
  Total data traffic $= \sum_{i' \neq i}^{i' \in \{\text{Learner,Actors,RMM}\}} C_{i'}$
  if Total data traffic $<$ min _ traffic **then**
    min _ traffic $\leftarrow$ Total data traffic; $D_{\text{Data Storage}} \leftarrow D_i$
Output $D_{\text{Data Storage}}$

B. Mapping Algorithm

Given a Primitive-Platform Performance Table, the Mapping Algorithm first determine the best mapping of the computation primitives to maximize the achievable peak training throughput. Then it places the memory component (Data Storage) to minimize the required data traffic in the system.

The achievable training throughput can be estimated as the minimum of the throughput among the learner, the priority update and Prefix Sum Index computation. Therefore, the objective is to search for a combination of platform assignment to the primitives, such that

$$\text{argmax}_{device1,2}\left\{\min\left(TP_{Learner}^{device1}, TP_{RMM}^{device2}\right)\right\} \quad (3)$$

We take a greedy approach to optimize the overall metric (gradient steps performed per second) following three steps as shown in Algorithm 3:

Step 1. We map the learner to the platform yielding the highest training throughput. We prove that $TP_{Learner}$ is the upper-bound of the achievable system throughput by enumeration method: (1) If for some RMM mapping platform, $TP_{RMM} >$

$TP_{Learner}$, according to (3), the achievable system throughput equals $TP_{Learner}$. (2) Else if $TP_{RMM} \leq TP_{Learner}$, the achievable system throughput is bound by RMM, thus lower than $TP_{Learner}$. While this maximizes the theoretical system throughput upper-bound, we still need to ensure that the system throughput is not bottlenecked by the RMM throughput. Step 2. We then map the RMM to the platform yielding highest replay operation throughput. Note that if the learner is mapped to GPU in the previous step, and achievable RMM throughput on a non-GPU devices is higher than the learner throughput (making learner the bottleneck in the system), we prune out the option of GPU for RMM (Algorithm 3 line 6-7). This is because letting RMM and learner share GPU resources will further reduce learner throughput from B/T^{GPU}_{train} to $B/(T^{GPU}_{train} + T^{GPU}_{RMM})$. On CPU and FPGA, this problem does not occur. Pre-defined hardware constraint for RMM (1 thread on CPU, 1 SLR on FPGA) ensures that RMM does not share hardware resource with the learner, thus cannot affect $TP_{Learner}$.

Step 3. Step 1 and Step 2 produce the D_{Learner} and D_{RMM} in the assignment vector. We decide the device assignment of the memory component, i.e., the Data Storage. The total data traffic to the Data Storage during each gradient step is B words of sampling indices from the D_{RMM}, $B \times E$ words of sampled data to the D_{Learner} (E is the size of each data point stored in the Data Storage), and $N_{actor} \times E$ words of inserted data from the actors running on CPU threads. These remote communication costs are denoted as C in Algorithm 3 line 11. We place the Data Storage on the device that yield the minimum total data traffic in remote communication with primitives on other heterogeneous devices (Algorithm 3 line 9-16). Overall, assuming there are m primitives to be deployed on n heterogeneous devices, the time complexity for the Mapping Algorithm is $O(m(m+n))$.

C. Template Instantiation

We develop a code base composed of (1) the complete CPU multi-thread host program template with the Inter-Processing Unit Data Transfer system, (2) the host program for interfacing a host CPU thread with a Processing Kernel on the accelerator (Intra-Processing Unit Data Transfer) under its various primitive mapping, and (3) the kernel programs on the accelerators under various mapping options (GPU and FPGA). As shown in Fig. 8, a Template Instantiater draws the device assignment result of primitive mapping from the Mapping Algorithm, and use it to obtain the parameterized code snippets (2) and (3) for each assigned accelerator from the code base. After compiling (3) to generate kernel executable (or bitstreams), the host code snippets in (2) are then filled into (1) the host program template, for an end-to-end complete implementation.

VII. Experiments

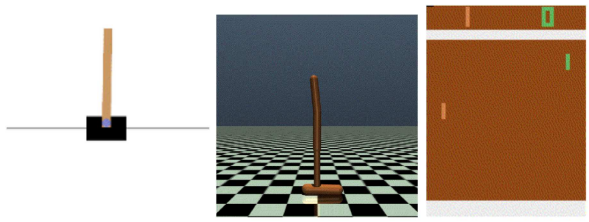
Our experiments aim to demonstrate i) the DRL performance in terms of rewards achieved by our framework is the same as the serial version of the corresponding DRL algorithm; ii) the execution time of each primitive varies w.r.t the input parameters including batch size, neural network architecture; ii) the superiority of the mapping generated by our framework compared with other mappings on various benchmark environments solved by different DRL algorithms.

TABLE III
Overview of Benchmark Environments, DRL Algorithms and Neural Network Architectures

Environment	Algo	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{A}\rvert$	NN Architecture	MACs
CartPole	DDPG	4	1	3-layer MLP hidden size 8	137
Hopper	DDPG	11	3	3-layer MLP hidden size 256	70.1K
Pong	DQN	84×84	6	ConvNet in [14]	18.8M

TABLE IV
Specifications of the Heterogeneous Devices

	CPU	GPU	FPGA
Process	14 nm	16 nm	16 nm
DDR/HBM Bandwidth	89 GB/s	550.0 GB/s	77 GB/s
Last-Level Cachce/ SRAM size	8 MB	3 MB	35 MB
Peak Performance (TOPS)	0.04 (FP32)	12.15 (FP32)	0.68 (INT32)



(a) CartPole (b) Hopper (c) Pong

Fig. 9. Benchmark software environments.

A. Experimental Setup

Hardware Platform and Toolchain: Our experiments are conducted using Intel(R) Xeon(R) Gold 5120 CPU@ 2.20 GHz with 56 cores, a Nvidia TITAN Xp GPU and a Xilinx Alveo U200 accelerator board. PCIe is used to connect the CPU with GPU, and CPU with FPGA, both with bandwidth 16 GB/s. 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 [12]). We follow the VITIS hardware development flow for bitstream generation. OpenCL is used to implement the data transfer between the host and the FPGA. Primitives on the CPU and GPU are developed using C++ and CUDA, respectively. The detailed specifications of heterogeneous devices are summarized in Table IV.

Benchmarking Software Environment: We select 3 benchmark environments including classic control task CartPole, MuJoCo task Hopper and Atari games Pong in the OpenAI gym software simulation environment [17]. The size of the observation space and the action space is shown in Table III.

- CartPole: As shown in Fig. 9(a), there is a pendulum placed upright on the cart. The objective is to balance the pole by pushing the cart to the left and right. The observation consists of the position and the velocity of the cart and pole.
- Hopper: As shown in Fig. 9(b), the hopper is a two-dimensional robot consisting of four main parts: i) the torso at the top; ii) the thigh in the middle; iii) the leg in the bottom; iv) and a single foot on which the entire body rests. The objective is to apply forces on the three hinges connecting the four body parts such that the hopper moves as fast as possible.
- Pong: As shown in Fig. 9(c), the objective of Pong is to control your paddle to bounce the ball below your opponent's paddle.

The DRL algorithms used to solve each environment and the neural network architectures are shown in Table III, where $|\mathcal{S}|$ denotes the dimension of state space and $|\mathcal{A}|$ denotes the dimension of action space.

Hyper-parameters: The number of actors are set to 16 for all environments according to existing work [5]. The number of child nodes per parent node K used in the Sum Tree implementation is set to 2 for simplicity. The sizes of the Data and Priority Request Queues are 1024. The size for Data Queue is 200 (10) for Pong (Hopper).

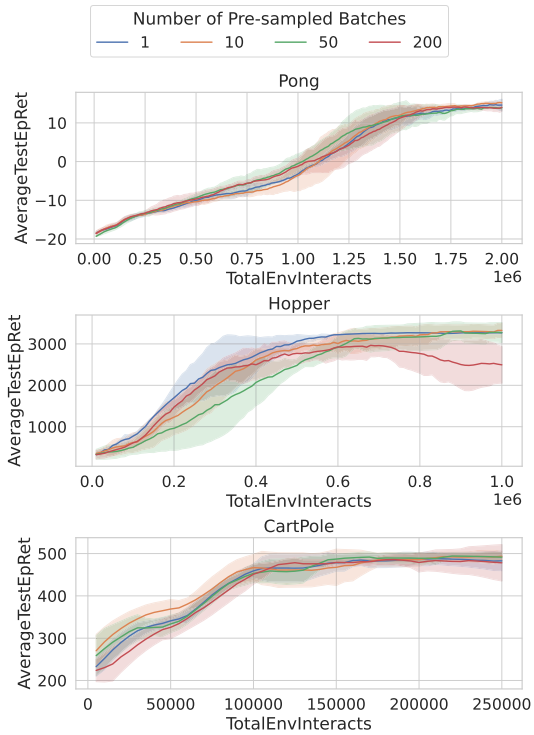


Fig. 10. Reward of RL agents trained in benchmark environments with various number of pre-sampled batches (D). The curve shows the mean, and the shaded area shows the standard deviation of 5 runs with different random seeds.

B. DRL Algorithm Performance

As discussed in Section V-A, the training throughput benefits from the data-dependency relaxed training loop, which removes the data dependency between sampling of the next batch data and the priority update of the previous batch data. The objective is to pre-sample batches of data such that the learner never waits. However, this may cause degradation of the reward performance. We denote the number of pre-sampled batches as D. D is the primary parameter that controls the degree of dependency relaxation. In order to empirically investigate the relationship between D and the achieved accumulated rewards, we train several agents with 5 different random seeds for each benchmark environment using the optimal mapping. We show the achieved accumulated rewards along the trajectory versus the total number of environmental interactions of each environment with various number of pre-sampled batches in Fig. 10. As D increases, it may impact the reward performance because the staleness of the priority distribution becomes more severe, as evident in the case when $D = 200$ for Hopper. Still, we observe that the performance degradation is consistently negligible with $D = 50$. In practice, we set $D = 50$ for all the three environments. By doing so, it improves the training throughput as the sampler does not have to wait for the updated priorities of the previous sampled batch while the rewards are not negatively impacted.

C. Primitive Acceleration Performance

We profile the performance of various primitives including neural network training in Fig. 11, Prefix Sum Index computation and priority update in Fig. 12.

Training primitive: The number of CPU threads used for training on CPU is 16. We observe that the training performance of FPGA dominates over CPU and GPU when the arithmetic intensity is low. (e.g., when the size of the neural network is small as in CartPole or when the batch size is less than 128 in Hopper and less than 16 in Pong). The training performance of GPU dominates over CPU and FPGA when the arithmetic intensity is high (e.g., when the batch size in Hopper is larger than 1024 and the batch size in Pong is larger than 16). This is because the kernel and memory overhead of GPU are not negligible when the arithmetic intensity is low. The high memory access latency with low data re-use in smaller batch size training makes the SIMT computation power of GPU severely unsaturated. The superiority of training performance on FPGA arises from our high throughput customized hardware design. However, the execution time of training primitive on GPU starts to outperform that on FPGA when the batch size increases due to negligible kernel launch overhead and higher clock frequency.

Prefix Sum Index computation: On CPU, we observe that the execution time of Prefix Sum Index computation decreases as the number of CPU threads increases. This is because Prefix Sum Index computation is a read-only operation that each computation inside a batch can be executed in parallel. On GPU, the execution time is almost the same when the batch size increases. This indicates that the computation of Prefix Sum Index is bound by the kernel launch overhead and fails to saturate the GPU's SIMT power. On FPGA, we observe a linear increase in the execution time as batch size increases above 1024 as the number of sampling and update pipelines reach the resource

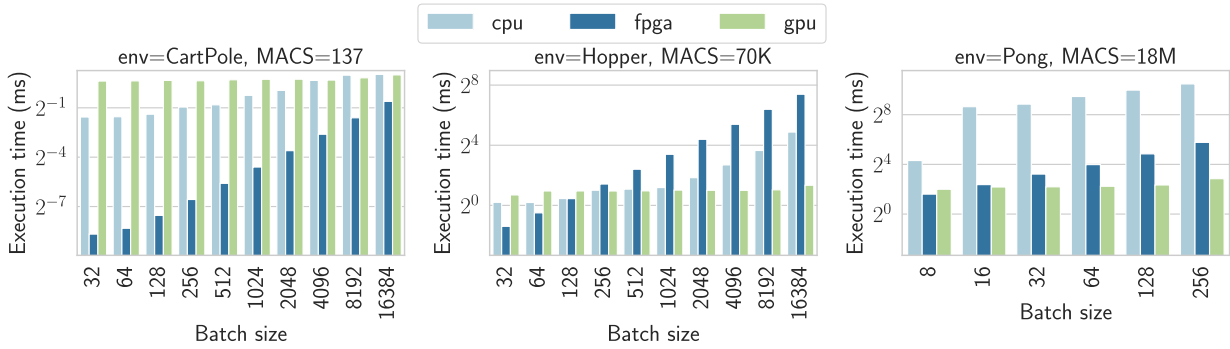


Fig. 11. Execution time (in milliseconds) of a single neural network training step in benchmark environments of various batch sizes on various hardware platforms.

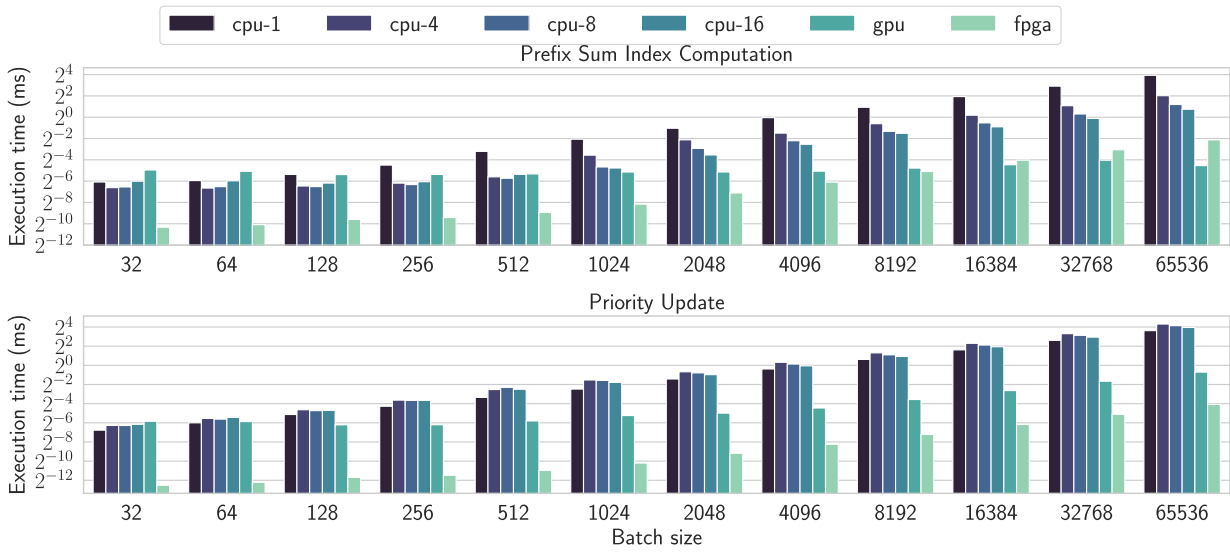


Fig. 12. Execution time (in milliseconds) of a single Prefix Sum Index computation and priority update of various batch sizes on various hardware platforms. The number after "cpu" denotes the number of threads using in OpenMP [11].

bound of an SLR. For batch sizes smaller than 1024, the increase in RMM operations latency is not as sever because we scale up bank parallelism with increasing batch sizes to make the performance more salable. For Prefix Sum Index computation, GPU is more salable to larger batch sizes since there are no data-dependency between obtaining samples within a batch and they can be executed in a SIMD manner. Compared to GPU, the FPGA Prefix Sum performance ranges from $41\times$ speedup to $4.6\times$ speed-down over all the batch sizes.

Priority update: Different than Prefix Sum Index computation where different samples in a batch are independent, in Priority Update computation, multiple update requests poses write-after-write data access dependencies at the root. The execution time improvement of priority update is almost negligible or negatively affected when the number of CPU threads increases. This is due to the poor cache performance caused by memory access conflict as discussed in Section IV-D2. The execution time of priority update on GPU increases in linear as batch size increases due to the inevitable serial execution at root node and high memory access latency that cannot be hidden by computation. On the other hand, FPGA-based implementation features hardware pipelining and single-cycle on-chip data accesses to maximize the throughput of the serial execution. This leads to consistent superior performance of priority update of all the batch sizes ($11 \sim 98\times$ speedup compared to GPU) as shown in Fig. 12.

Overall, we observe that the optimal mapping of all the primitives vary as the input algorithm configurations change. This makes a fixed mapping of DRL algorithms in efficient and motivate the necessity to automatically generate the mapping based on the inputs.

D. System Mapping and Performance Analysis

In the bar plots of Fig. 13, we show the achieved system throughput for the three benchmarks under different batch sizes and mappings, and in the line plot we show the theoretical optimal training throughput. The theoretical optimal training throughput is the maximum throughput of the training primitive among various mappings of the learner. It is achieved when i) the learner is mapped onto the platform with lowest execution time of the neural network training primitive; ii) the mapping of RMM doesn't slowdown the training throughput; iii) the overhead

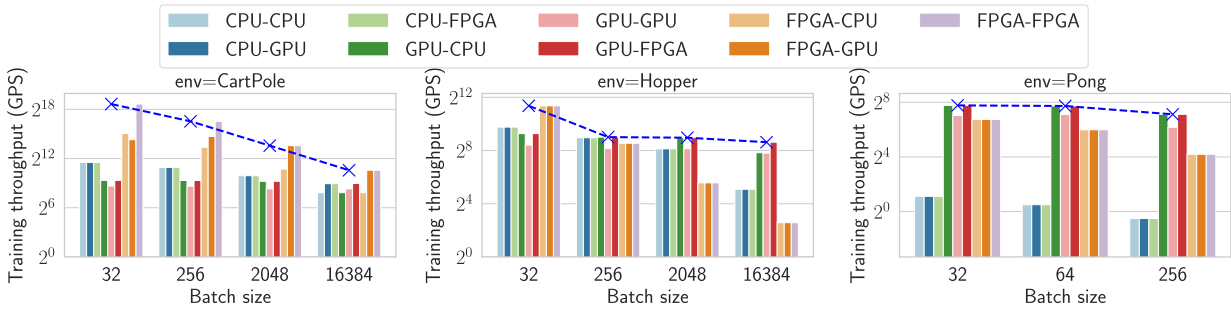


Fig. 13. The training throughput in gradient steps per second (GPS) of various batch sizes in various mappings. The mapping "X-Y" denotes that the learner is mapped onto "X" and Replay Management Module is mapped onto "Y". The line plot shows the theoretical optimal performance given environment and batch size.

TABLE V
SYSTEM DSE RESULT FOR VARIOUS CONFIGURATIONS AND THE RANGES OF RESULTING SPEEDUPS

	env=CartPole DNN MACS=137				env=Hopper DNN MACS=70K				env=Pong DNN MACS=18M		
Batch size	32	256	2048	16384	32	256	2048	16384	32	64	256
Optimal Mapping	F-F-F	F-F-F	F-F-F	F-F-F	F-F-F	G-C-G	G-C-G	G-F-G	G-C-G	G-C-G	G-C-C
2^{nd} Best Baseline Mapping & Our Speedup	F-C-F & 12×	F-G-F & 3.6×	F-C-F & 7.2×	G-F-G & 3.1×	F-C-F & 7.2×	G-C-G & 1.03×	G-G-G & 1.77×	G-C-G & 1.73×	G-G-G & 1.67×	G-G-G & 1.54×	G-G-G & 1.93×
Worst Baseline Mapping & Our Speedup	G-G-G & 997.3×	G-G-G & 240.5×	G-G-G & 316×	C-C-C & 6.68×	G-G-G & 7.71×	G-G-G & 1.81×	F-C-F & 10.5×	F-C-F & 66×	C-C-C & 100.9×	C-C-C & 149.8×	C-C-C & 197.6×

Note 1: The "Optimal" configuration refers to the optimal mapping returned by our proposed framework. The "2^{nd} Best" ("Worst") configuration is obtained by using the mapping that yields the highest (lowest) throughput OTHER THAN the optimal mapping.
Note 2: All the mappings are device specifications for Learner-RMM-Data Storage. F, C and G stands for FPGA, CPU, and GPU.

TABLE VI
DESIGN PARAMETERS AND RESOURCE ALLOCATION FROM FPGA ARCHITECTURE EXPLORATION

FPGA Hardware Parameters	Pipeline Factor (PI)	Data Parallel Factor (DP)	RMM Bank Parallelism (S)	# SLR Constraint (RMM, Learner)
	BS/2	2	1∼32	[1,2]
Resources	**SRAM**	**REG**	**LUT**	**DSP**
RMM (S=32)	4.5MB (12.8%)	263K (11%)	181K (15%)	1280 (18%)
DQN Learner	17.6MB (51%)	994K (42%)	615K (52%)	4315 (64%)
DDPG Learner	12.3MB (35%)	782K (33%)	721K (46%)	2557 (38%)

from thread-level synchronization of the data transfer queues is negligible to the system throughput. As shown in Fig. 13, the difference of the achieved performance using our system DSE to the theoretical optimum is within 5% as shown in the line plot in Fig. 13.

Accordingly with each configuration in Fig. 13, Table V shows their optimal mapping returned by the system DSE Mapping Algorithm, along with the ranges of speedups shown by the 2^{nd} best and worst baseline mappings. Table VI shows the hardware parameters returned by the Architecture Exploration when FPGA is used. For the CartPole benchmark, when both the DNN and batch sizes are small, both training and RMM operations are mapped on FPGA as it outperforms GPU. As the batch size increases (i.e., batch size $B > 2048$), the learner gradient step execution time becomes larger, such that the latency of RMM operations can be hidden using either GPU or FPGA. Our Mapping Algorithm chooses the mapping that minimizes the total number of devices and the amount of data communication, so it still maps both RMM and learner to the FPGA. In the Hopper benchmark, the same observation also applies to the medium-sized DNN when the batch size is small. As the batch size further increases, GPU outperforms FPGA on training due to its superior amount of parallel resources and higher frequency. As the learner is mapped to GPU for $B > 256$, although FPGA outperforms CPU for the RMM operations, the learner is the bottleneck and assigning RMM to either CPU or FPGA yields the same overall throughput. However, when the batch size reaches a threshold ($B = 16384$) where the CPU RMM operation latency can no longer be hidden by the learner, mapping the RMM to FPGA has obvious improvement over other baselines. For large DNN and batch size in the Pong game benchmark, the learner is consistently mapped to GPU due to its superior training performance. Even with slower RMM performance on CPU than FPGA, the learner remains the bottleneck of the system, so we map it to CPU for minimized number of devices and communication requirements.

We observe that all the three devices (CPU-GPU-FPGA) should be used for optimal system performance when the DNN is small and batch size is large (e.g., The Hopper benchmark using 3-layer MLP with batch size 16384). This is because large batch training favors GPU over CPU and FPGA, while large batch

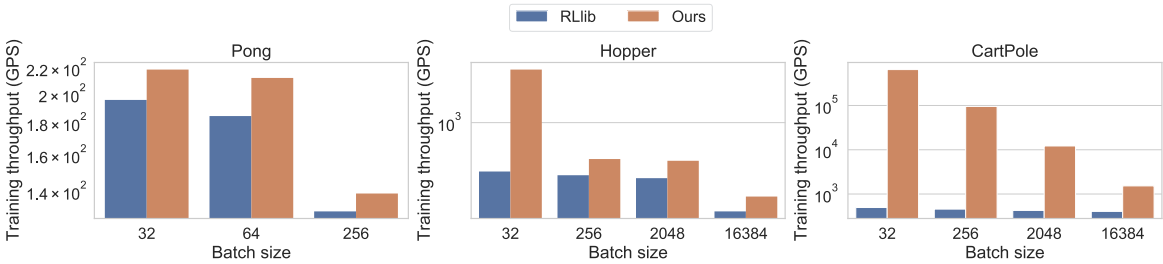


Fig. 14. The training throughput in gradient steps per second (GPS) of various batch sizes between the state-of-the-art distributed RL framework RLlib [18] and the optimal mapping by our proposed framework.

RMM operations can only be effectively accelerated without reducing system throughput on the FPGA (mapping RMM to CPU will shift the bottleneck from training to RMM operations, and mapping RMM to GPU will take over the resources for the learner, lowering the training throughput by about 30%). Compared with baseline mappings, our CPU-GPU-FPGA system achieves up to 11× higher throughput. In all the benchmarks and baselines, our framework achieves up to 997.3× speedup.

E. Comparison With State-of-The-Art

In the bar plots of Fig. 14, we show the training throughput in gradient steps per second (GPS) for various batch sizes between the state-of-the-art distributed RL framework RLlib [18] and the optimal mapping returned by our proposed framework for the benchmark environments described in Section VII-A. The mapping of Data Storage and RMM can only be on CPU in RLlib [18]. The mapping of Learner can be on CPU or GPU. We measure the training throughput of both mappings and report the higher value as the training throughput of RLlib in Fig. 14. For Pong, our framework improves the training throughput by $1.06\times \sim 1.14\times$. This is because the neural network architecture is a Convolutional Neural Network (CNN); the training time of the neural network dominates the overall training throughput. Both our framework and RLlib utilize Pytorch [12] backend to train neural networks. For Hopper, our framework improves the training throughput by $1.31\times \sim 6.39\times$. As the batch size increases, the speedup decreases. This is due to the neural network training dominates the overall training throughput. For CartPole, our framework improves RLlib up to $1005\times$. This is because in CartPole, all the components of our framework are mapped on the FPGA. This enables on-chip data storage and data transfer that significantly improves the training throughput. In general, we found that RLlib only optimizes large scale RL tasks while our framework optimizes RL tasks for all the scales.

Table VII shows the total power consumption of the complete system using our framework and the state-of-the-art implementation (RLlib). The power is the sum of the operating Thermal Design Power for the CPU and DDRs, the power reported by the Nvidia runtime profiler for the GPU, and the power reported using Vivado after place-and-route for the FPGA. The reported power consumptions for the benchmarks are evaluated at the largest batch size (16384 for CartPole and Hopper, 256 for Pong). Note that the mapping used by our framework and RLlib are different. For example, for the Hopper benchmark, our framework consumes slightly higher total power due to the additional FPGA component compared with RLlib mapping. The throughput improvement from our mapping is much larger than the increase in the power consumption. This leads to up to 3.8 times higher power efficiency (in terms of GPS per Watt).

TABLE VII
POWER AND POWER EFFICIENCY OF SELECTED BENCHMARKS

Benchmarks		Ours (Watts)	Ours (GPS/ Watt)	RLlib (Watts)	RLlib (GPS/ Watt)
Pong, DQN	Mapping (Learner-RMM-Data Storage)	G-C-C		G-C-C	
	Total System Power	565.3	0.246	564	0.23
Hopper, DDPG	Mapping (Learner-RMM-Data Storage)	G-F-G		G-G-C	
	Total System Power	237.2	1.598	228	0.87
CartPole, DDPG	Mapping (Learner-RMM-Data Storage)	F-F-F		C-C-C	
	Total System Power	217.5	7.012	220	1.841

VIII. Discussion & Conclusion

In a real-world scenario (e.g., robots, self-driving cars), the RL trial-and-errors require a large amount of training episodes, which can damage the physical agent devices such as the robots or cars if directed by unsafe policies [25]. Therefore, the typical workflow of applying Deep RL constitutes of two phases: the development phase (i.e., Training-in-Simulation) that uses a software simulator without actually deploying the agent in the field, and the deployment phase that executes the agent devices in a real-world environment. In the Training-in-Simulation (development) phase, the policy training is usually done by interfacing the policy model with simulation environments installed on a multi-core CPU. This process is extremely time-consuming as it involves a magnitude more than millions of sequential iterations, and takes days to months to complete [1]. On the other hand, the deployment is largely dominated by policy inference and only few iterations of on-line learning. Our work targets speeding up the Training-in-Simulation phase, and can largely reduce the production time before deployment.

In this work, we proposed a framework for optimizing DRL algorithms with Prioritized Replay Buffer to achieve the optimal training throughput based on the input specifications and

heterogeneous hardware configurations. We proposed separate accelerators for each primitives on multi-core CPU, FPGA and GPU. Then, we proposed data-dependency relaxed training loop to maximize the training throughput without affecting the DRL performance. Our experimental results verified our claims. Future work includes developing mapping framework for other DRL algorithms without Prioritized Replay Buffer.

ACKNOWLEDGMENTS

Seed funding from Ershaghi Center for Energy Transition is gratefully acknowledged.

REFERENCES

[1] L. P. Kaelbling et al., "Reinforcement learning: A survey," *J. Artf. Intell. Res.*, vol. 4, pp. 237–285, 1996.
[2] K. Chatzilygeroudis et al., "Black-box data-efficient policy search for robotics," in *Proc. IEEE RSJ Int. Conf. Intell. Robots Syst.*, 2017, pp. 51–58.
[3] O. Vinyals et al., "AlphaStar: Mastering the real-time strategy game starcraft II," *DeepMind Blog*, vol. 2, p. 20, 2019.
[4] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html
[5] D. Horgan et al., "Distributed prioritized experience replay," 2018, *arXiv:1803.00933*.
[6] L. Espeholt et al., "SEED RL: Scalable and efficient deep-Rl with accelerated central inference," 1910, *arXiv:1910.06591*.
[7] T. Schaul et al., "Prioritized experience replay," 2015, *arXiv:1511.05952*.
[8] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, vol. 22, pp. 400–407, 1951.
[9] C. Zhang et al., "Parallel actors and learners: A framework for generating scalable RL implementations," 2021, *arXiv:2110.01101*.
[10] Intel, "Oneapi for heterogeneous cloud," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html
[11] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf
[12] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035, [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[13] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[14] V. Mnih et al., "Playing atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.
[15] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2016, *arXiv:1509.02971*.
[16] A. Nair et al., "Massively parallel methods for deep reinforcement learning," 2015, *arXiv:1507.04296*.
[17] G. Brockman et al., "OpenAI gym," 2016, *arXiv:1606.01540*.
[18] E. Liang et al., "Ray RLlib: A composable and scalable reinforcement learning library," 2017, *arXiv:1712.09381*.
[19] Y. Li and D. Schuurmans, "MapReduce for parallel reinforcement learning," in *Recent Advances in Reinforcement Learning*, S. Sanner and M. Hutter Eds., Berlin, Germany: Springer, 2012, pp. 309–320.
[20] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.
[21] H. Cho et al., "FA3C: FPGA-accelerated deep reinforcement learning," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 499–513.
[22] S. Shao and W. Luk, "Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation," in *Proc. IEEE 27th Int. Conf. Field Programmable Log. Appl.*, 2017, pp. 1–6.
[23] S. Shao et al., "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *Proc. IEEE 29th Int. Conf. Application-Specific Syst. Architectures Processors*, 2018, pp. 1–8.
[24] J. Schulman et al., "Trust region policy optimization," 2015, *arXiv:1502.05477*.
[25] C. Guo et al., "Customisable control policy learning for robotics," in *Proc. IEEE 30th Int. Conf. Application-Specific Syst. Architectures Processors*, 2019, pp. 91–98.
[26] Y. Meng et al., "Accelerating proximal policy optimization on CPU-FPGA heterogeneous platforms," in *Proc. IEEE 28th Annu. Int. Symp. Field-Programmable Custom Comput. Machines*, 2020, pp. 19–27.
[27] Y. Meng et al., "FPGA acceleration of deep reinforcement learning using on-chip replay management," in *Proc. 19th ACM Int. Conf. Comput. Front.* New York, NY, USA: Association for Computing Machinery, 2022, pp. 40–48, doi: 10.1145/3528416.3530227.
[28] "Large FPGA methodology guide," 2012. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf
[29] "Intel stratix 10 MX FPGAS," [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html
[30] R. Nane et al., "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, pp. 1591–1604, Oct. 2016.
[31] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021.
[32] Intel, "Skylake specification," 2018. [Online]. Available: https://www.7-cpu.com/cpu/Skylake.html
[33] Intel, "GPU memory latency's impact, and updated test," 2021. [Online]. Available: https://chipsandcheese.com/2021/05/13/gpu-memory-latencys-impact-and-updated-test/

Chi Zhang received the bachelor of engineering degree from Southeast University, Nanjing, China, and the master of science degree in electrical engineering from the University of Southern California. He is currently working toward the PhD degree in computer science with the University of Southern California under the supervision of professor Viktor Prasanna. His primary research interests include parallel reinforcement learning.

Yuan Meng received the BS degree in electrical and computer engineering from Rensselaer Polytechnic Institute. She is currently working toward the PhD degree in computer engineering with the University of Southern California. She is recipient of Annenberg Fellowship with Ming Hsieh Department of Electrical and Computer Engineering. Her research interests include parallel computing, hardware acceleration, and machine learning.

Viktor K. Prasanna (Fellow, IEEE) received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from the Pennsylvania State University. He is Charles Lee Powell Chair in engineering with the Ming Hsieh Department of Electrical and Computer Engineering and professor of computer science with the University of Southern California. His research interests include parallel and distributed systems, reconfigurable computing, and applied ML. He serves as the director of the Center for Energy Informatics, USC. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and the IEEE International Conference on High Performance Computing (HiPC). He received 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, and the 2015 W. Wallace McDowell Award from the IEEE Computer Society for his contributions to reconfigurable computing.