PEARL: Enabling Portable, Productive, and High-Performance Deep Reinforcement Learning using Heterogeneous Platforms

Yuan Meng
ymeng643@usc.edu
University of Southern California
United States

Michael Kinsner
michael.kinsner@intel.com
Intel Corporation
United States

Deshanand Singh
deshanand.singh@intel.com
Intel Corporation
United States

Mahesh Iyer
mahesh.iyer@intel.com
Intel Corporation
United States

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

ABSTRACT

Deep Reinforcement Learning (DRL) is vital in various AI applications. DRL algorithms comprise diverse compute kernels, which may not be simultaneously optimized using a homogeneous architecture. However, even with available heterogeneous architectures, optimizing DRL performance remains a challenge due to the complexity of hardware and programming models employed in modern data centers. To address this, we introduce PEARL, a toolkit for composing parallel DRL systems on heterogeneous platforms consisting of general-purpose processors (CPUs) and accelerators (GPUs, FPGAs). Our innovations include: 1. A general training protocol agnostic of the underlying hardware, enabling portable implementations across various platforms. 2. Incorporation of DRL-specific optimizations on runtime scheduling and resource allocation, facilitating parallelized training and enhancing the overall system performance. 3. Automatic optimization of DRL task-to-device assignments through throughput estimation. 4. High-level API for productive development using the toolkit. We showcase our toolkit through experimentation with two widely used DRL algorithms, DQN and DDPG, on two diverse heterogeneous platforms. The generated implementations outperform state-of-the-art libraries for CPU-GPU platforms by up to 2.2× throughput improvements, and 2.4× higher performance portability across platforms.

CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*; *Reinforcement learning*.

KEYWORDS

Heterogeneous Computing, Deep Reinforcement Learning

ACM Reference Format:

Yuan Meng, Michael Kinsner, Deshanand Singh, Mahesh Iyer, and Viktor Prasanna. 2024. PEARL: Enabling Portable, Productive, and High-Performance Deep Reinforcement Learning using Heterogeneous Platforms. In *21st ACM International Conference on Computing Frontiers (CF '24), May 7–9, 2024, Ischia, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3649153.3649193

CF '24, May 7–9, 2024, Ischia, Italy

ACM ISBN 979-8-4007-0597-7/24/05.
https://doi.org/10.1145/3649153.3649193

1 INTRODUCTION

Deep Reinforcement Learning (DRL) is extensively applied in various domains, including robotics, surveillance, etc. [7, 25]. Most DRL algorithms involve three collaborative compute kernels: policy execution, training, and dataset management. In policy execution, parallel Actors gather data through inference on the policy, interact with the environment, and deposit the data into a Prioritized Replay Buffer for dataset storage. In training, a centralized Learner samples data from the Prioritized Replay Buffer to update the policy model. The dataset management within the Prioritized Replay Buffer is facilitated by a sum tree data structure storing data priorities [27].

DRL training is highly time-consuming. Due to the distinct compute kernels in DRL that may not be efficiently optimized using a homogeneous architecture, there has been a growing trend in using heterogeneous architectures to accelerate DRL algorithms [9, 14, 16]. However, even with access to heterogeneous resources, DRL application developers still face several challenges: (a). *Sub-optimal performance*: DRL's distinct components require careful placement and scheduling onto heterogeneous devices based on both computational and hardware characteristics. Sub-optimal placement and scheduling can lead to under-utilization of heterogeneous resources, resulting in missed opportunities for performance improvement. (b). *Lack of portability across platforms*: The optimal DRL primitive-to-hardware assignments can change based on varying algorithms and platforms. Consistently achieving high-performance implementations requires portable solutions that can map and distribute DRL onto various devices, but existing frameworks lack such flexibility. (c). *Low development productivity*: The growing diversity of heterogeneous resources in data centers [1, 24, 26] have increased the need for hardware optimizations and bridging between different programming models. This significantly increases the required learning effort and programming time for application developers.

In this work, we address the above challenges by proposing PEARL, a toolkit that enhances the performance, productivity, and portability [20] of DRL system development on heterogeneous platforms. PEARL provides DRL application developers with tools and familiar

interfaces for running DRL using heterogeneous platforms, while abstracting away the low-level hardware intricacies. Specifically, it takes a Python program from the user with functionalities similar to existing RL ecosystems and frameworks (e.g., PyTorch [19], RLlib [14]). Its main novelties compared to existing RL frameworks are unique intermediate abstraction layers below the Python interface. They define the runtime scheduling and automatic design space exploration to enable effective utilization of heterogeneous resources. Additionally, they integrate fine-grained acceleration of individual primitives. These are realized by a Host Runtime Coordinator, a System Composer, and a Parameterized Library of Accelerated Primitives.

Our key contributions are:

- We propose a general DRL heterogeneous training protocol that is agnostic of the types of underlying accelerators, thus portable to different heterogeneous platforms.
- We propose a dynamic resource management mechanism, which fine-tunes the training workload assigned to the CPUs and the accelerators during runtime.
- We develop a parameterized library that contains accelerated DRL primitives on various architectures (CPU, GPU, and FPGA). We offer a Python-based User API to enable productive DRL application development on heterogeneous platforms.
- We develop a novel System Composer for identifying optimal device assignments and accelerator configurations, ensuring high performance of the DRL implementation.
- We assess our toolkit using representative DRL algorithms on various benchmarks and platforms. Compared with existing DRL frameworks, our implementations lead to up to 2.2× speedup, and 2.4× higher performance portability. Our implementations are achieved with just dozens of lines of code, demonstrating high development productivity.

2 BACKGROUND

2.1 Deep Reinforcement Learning

A generalized DRL training process comprises four primitives: Actors, Learner, Replay Manager (RM), and Experience (Exp) Memory. These primitives work and interact as follows:

Actors: Each Actor maintains a Deep Neural Network (DNN) policy network, inferring an action based on an input environment state. Each Actor operates on an instance of the environment simulator, applying the inferred action. The environment responds and generates a tuple {state, action, new state, reward}, constituting an experience (i.e., a data point) for training. Multiple copies of the Actor repeat this process to collect experiences, which populate a training dataset called the Replay Buffer.

Replay Buffer: Unlike pre-labeled datasets in supervised learning, the Replay Buffer in DRL is continuously filled by online interactions of Actors with the environment, and its data points are dynamically changing as the policy evolves. In state-of-the-art DRL, the Prioritized Replay Buffer is widely used for managing data with probabilities proportional to the current policy loss to enhance training quality [11, 23]. It incorporates a Replay Manager (RM) associating a priority (i.e., probability of being sampled) with each experience in the Experience (Exp) Memory. During data sampling, a data point (i.e., experience) x_i is selected based on the probability distribution $\Pr(i) = P(i)/\sum_i P(i), i \in [0, \text{replay buffer size})$, where $P(i)$ represents the priority of data point i. This selection is achieved by identifying the minimum index i for which the prefix sum of probabilities up to i is greater than or equal to x, where x is a uniformly generated random target prefix sum value between 0 and the total priority sum [23]:

$$\min_i \sum_{j=1}^{i} P(j) \geq x, x \sim U[0, \sum_{j=0}^{\text{replay buffer size}} P(j)] \quad (1)$$

To enable rapid sampling and scalable update operations for large Exp Memory, priorities are managed using a sum tree data structure [23, 28]. Replay sampling and replay update operations on an n-ary sum tree are defined in [27].

Learner: In each training iteration, a batch of indices are sampled via the RM to obtain experiences by reading from the Exp Memory. Then, the Learner performs training using stochastic gradient descent (SGD, [22]) on the policy network. During the computation of the loss function in SGD, an updated priority is produced and written back to the Replay Buffer via the RM. Policy network parameters are updated and sent to the Actors to ensure that experience collection employs the latest policy.

DRL Workload Characterization: The characteristics of Deep Reinforcement Learning (DRL) primitives exhibit variations not only among themselves but also across different learning functions, policy models, hyper parameters, etc. Consequently, relying on a fixed architectural solution proves inadequate for optimizing hardware utilization and achieving high-throughput DRL across the diverse spectrum of algorithms and applications. As examples, In Figure 1, we illustrate throughput performance of key compute primitives (replay sampling, replay update, and learner) for two algorithms (DQN [18], DDPG [15]) and policy models (MLP, CNN), on the roofline models for a CPU, GPU, and FPGA. In this example,

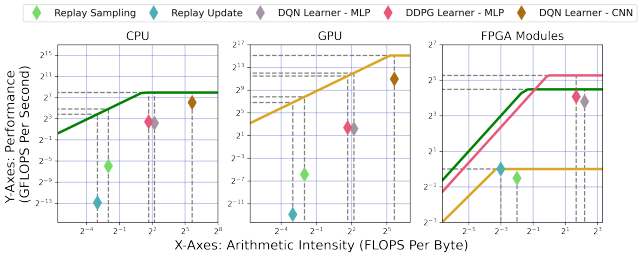


Figure 1: DRL Primitives Workload Analysis

primitives such as small MLP policies (commonly used in classical control and robotics benchmarks [10]) and replay operations exhibit low arithmetic intensities and high-latency memory accesses, making them memory-bound and challenging to optimize on multi-core or data-parallel architectures (CPU and/or GPU). The performance of these primitives can benefit from a near-memory fashion design using spatial architecture (FPGA). Learner functions with higher arithmetic intensity and data reuse, such as CNN policies used in vision-based applications [18], justify the data parallel resources provided by GPUs. Still, the characteristics of DRL performance may vary due to significant differences among replay and learner configurations based on applications, as well as diverse rooflines resulting from device bandwidth and compute capabilities.

2.2 Target Platforms

Today's data centers comprise highly heterogeneous machines combining a variety of processors, accelerators, and memory [1, 3, 4]. Based on the DRL workload characterization in Section 2.1, we justify that there is a compelling need for dynamic mapping of DRL algorithms using such a heterogeneous platform to consistently achieve high performance. Our toolkit is motivated by this need, addressing the optimization challenges and emphasizing performance, portability, and productivity in the design automation for DRL application users. PEARL is designed to adapt to a wide range of heterogeneous computing platforms with interconnected CPUs and accelerators like GPUs and FPGAs. Developing applications on such platforms typically demands expertise in designing hardware and bridging between different programming models, which requires a learning curve that hinders the productivity of application developers. PEARL's strength lies in its ability to support high-performance DRL across diverse heterogeneous hardware, while abstracting away complex hardware details.

2.3 Related Work

A number of works have implemented DRL on parallel and distributed systems. RLlib introduces high-level abstractions for distributed reinforcement learning, built on top of the Ray library [14]. Other works, such as [12, 27], implement parallel DRL algorithms by employing multiple parallel Actor threads and a centralized Learner thread, utilizing deep learning libraries like Tensorflow and LibTorch. These works leverage CPU and GPU data parallel resources for training, but do not efficiently optimize memory-bound primitives (such as small model training and replay operations) on specialized hardware. In recent years, some research works have focused on hardware acceleration for DRL algorithms. For instance, [9] and [16] present FPGA implementations for specific algorithms, the Asynchronous Advantage Actor-Critic (A3C) and the Proximal Policy Optimization (PPO). [17, 28] introduced an FPGA-based accelerator design for the Replay Buffer and mapped several DRL algorithms onto an FPGA-based heterogeneous platform. However, they only target a specific heterogeneous device setup and lack performance portability across different heterogeneous platforms; Moreover, these work map each primitive onto a single device, in the case of the Learner being the bottleneck, they lack the flexibility to improve its runtime performance using different devices. Our work bridges these gaps by developing a generalized protocol that makes the development of DRL portable to different heterogeneous platforms, accompanied by runtime heterogeneous resource management to fully saturate the heterogeneous compute power.

3 RUNTIME SYSTEM & TRAINING PROTOCOL

3.1 System Design

The implementation generated by PEARL is based on a parallel DRL system managed by a Host Runtime Thread. Figure 2 shows the setup of such a system. Multiple Actor threads generate new data points (experiences) and periodically synchronize weights from the Learner. They send the experiences to the Host Runtime Thread through Data Collection Queues (DCQs). The Host Runtime Thread interacts with the RM through an RM Request Queue (RRQ), where

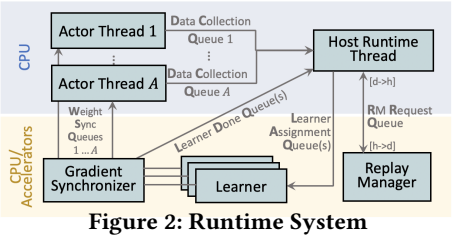


Figure 2: Runtime System

the host initiates sampling (or update) requests and receives outputs of sampled indices (or updated priorities). Parallel Learner modules can be implemented using both CPU threads and an accelerator, and they are initiated by the Learner Assignment Queues (LAQ). Their outputs are aggregated by a Gradient Synchronizer to produce the final weight gradients.

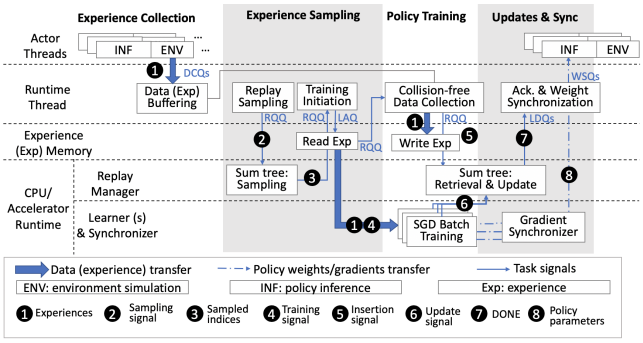


Figure 3: DRL Heterogeneous Training Protocol

3.2 DRL Heterogeneous Training Protocol

To perform training on a given heterogeneous system, we propose a general DRL heterogeneous training protocol (Figure 3). The training protocol can be ported to various heterogeneous devices since the interactions among processors and accelerators are defined at the application layer (i.e., DRL logical components), and are not bound to a specific type of accelerator. We show the essential data exchange and handshake signals between modular components as ❶-❽ in Figure 3. We provide a runtime code template that manages the thread pools and the accelerators, allowing the "plug and play" of heterogeneous devices for DRL primitives. It is a Python program executed on the Host Runtime Thread, which utilizes a loop whose iterations follow this protocol.

3.2.1 Replay-Collision-Free Scheduling. Our protocol features a novel scheduling optimization to encourage concurrency while maintaining algorithm correctness. We adopt a strategy of deferring the immediate insertion of experiences into the Replay Buffer when experiences are received from Actor threads. We maintain a data collection buffer to cache experiences generated by the Actors, and only insert them when the buffer is full. Upon experience insertion, we schedule the batched insertion operations after the sampling process concludes. This optimization has two advantages. Firstly, this approach permits us to compare the insertion index against the sampled indices, hence effectively mitigating the potential contamination of data when the Learner and Actors concurrently modify the same indices of the Replay memory. We refer to this procedure as "collision-free data collection" shown in Figure 3. Secondly, by

sequencing data insertion after the sampling phase, we align its execution concurrently with the training process. This hides the time overheads of the priority retrieval and update operations initiated by experience insertion in the training pipeline.

3.3 Runtime System Optimizations

Algorithm 1 Dynamic Resource Management

Input: Actor thread-pool size A, training batch size BS, sub-batch size trained on CPU b (initially, $b = 0$); runtime-profiled execution time of all Actor threads T_{Actor} in one iteration, a Learner (accelerator) gradient step $T_{Learner}$ in one iteration, the Host gradient synchronization $T_{sync-host}$ and CPU training time (initially, $T_{train-CPU} = T_{sync-host} = 0$)
sorted = sort(decreasing, $\frac{T_{Actor}}{A}$, $\frac{T_{Learner}}{BS}$, $T_{train-CPU}$, $T_{sync-host}$)
if sorted[0]==$\frac{T_{Learner}}{BS}$ and sorted[0]>2 × $\frac{T_{Actor}}{A}$ **then**
  freed = $pool_{Actors}$.size()/2; $pool_{Actors}$.size() − = freed;
  activate $pool_{train-CPU}$; $pool_{train-CPU}$.size() = freed;
  $pool_{train-CPU}$.submit(train(b + +), sync-host())
  learner.submit(train($BS-=b$))
else if sorted[0]==$T_{train-CPU}$ **then**
  $pool_{train-CPU}$.submit(train(b − −), sync-host())
  learner.submit(train($BS-=b$))
else if sorted[0]==$T_{sync-host}$ **then**
  $pool_{train-CPU}$.size() − −;
  $pool_{Actors}$.size() ++;

PEARL's runtime system design integrates a few optimizations that increase the effective utilization of heterogeneous resources and hide communication overheads.

3.3.1 Dynamic Heterogeneous Resource Allocation. To efficiently map DRL onto a heterogeneous platform, we first utilize the predicted result from our performance model (Section 5.2) to initially determine the mapping of each primitive onto a single accelerator at compile time. Even when optimally mapped to the most suitable accelerator, the Learner can remain the system's bottleneck. Meanwhile, if the Actors' data generation rate is significantly higher than the Learner's data consumption rate, the sample efficiency of DRL [6] can be negatively affected due to squandering of experiences information. To further fine tune Learner acceleration using heterogeneous hardware and avoid severe Actors-Learner load unbalancing, we develop a mechanism that supports dynamic re-allocation of CPU threads to process a sub-batch training. This mechanism is iteratively executed within the host runtime thread, as outlined in Algorithm 1. In instances where the amortized Learner latency dominates compared to the Actors by a large factor, it activates a pool of threads for training on CPU ($pool_{train-CPU}$) and re-assign Actor threads into CPU-training threads (which functions as a parallel sub-module of the Learner). Accordingly, the runtime thread also performs gradient synchronization to aggregate the gradients from the learner accelerator and the CPU-training threads. This logic is only activated if CPU threads are involved in training, and overhead from waiting for intermediate gradients is profiled and recorded in the variable storing host synchronization time $T_{sync-host}$. When the host gradient synchronization time emerges as a bottleneck, the number of CPU training threads is reduced to alleviate its overhead. This optimization strategy helps fully exploit the heterogeneity offered by both processors and accelerators, facilitating parallelized policy training and ensuring workload balance.

3.3.2 Communication Overhead Reduction. Our scheduling allows concurrent execution of the Actor threads (data collection) and the sampling → policy training (Learner) → experience update (RM) process. We also overlap Learner computation with replay operations. This is achieved by host-device (or on-chip) streaming communication queues between the RM and the Learner, so that training using each data point starts asynchronously as soon as the Learner receives them (rather than waiting for the full batched sampling). Additionally, we use double buffering to alleviate the weight transfer overheads between the processor and Learner accelerator. Two buffers with sizes of the complete policy weights is allocated in the host memory (shared by Actors threads and runtime thread). In each iteration i, the CPU threads read from buffer $i\%2$ while the Learner writes into buffer $1 - i\%2$.

4 PARAMETERIZED LIBRARY OF PRIMITIVES

4.1 Replay Manager (RM)

The RM performs three replay operations on a sum tree, where leaf nodes store the priorities for all experiences, and a parent node stores the sum of priorities of its children: (1) Priority sampling: Based on Equation 1, sampled indices are obtained by traversing the tree performing prefix sum from root to leaf. The computations are explained in [27]. (2) Priority retrieval: Given the indices of the experiences, it outputs the priorities stored at the corresponding leaf nodes. (3) Priority update: the inputs are the indices of experiences and the changes to their priorities Δ; It applies the changes Δ to the priorities (and sums of priorities) stored in parent nodes in all the tree levels. Note that Insertion of priorities is realized with priority retrievals followed by priority updates.

4.1.1 RM on CPU and GPU. The computations in replay operations can be viewed as a sequence of operations traversing all levels of the sum tree from the root to a leaf. Our RM implementations on CPU and GPU are parameterized with the tree depth, fanout, BS, and W, where BS is the batch size of the replay operation requests, and W is the number of workers (degree of parallelism) allocated. Each worker is responsible for sampling or updating $\frac{BS}{W}$ priorities. All workers share concurrent accesses to the sum tree. We use mutex to ensure the correctness of parallel priority updates that potentially collide on the same node.

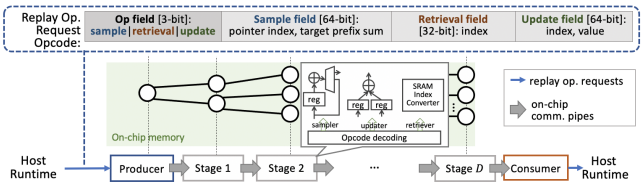


Figure 4: FPGA - Replay Manager Hardware Module

4.1.2 RM on FPGA. We develop an accelerator template (parameterized with the tree depth and fanout) that can be re-configured to support a range of fanout and tree sizes. We adopt a design of multiple pipeline stages processing a stream of operation requests as shown in Figure 4. Each pipeline stage is a hardware module responsible for operating on a certain tree level and exclusively stores all the nodes on that level. Different replay operation requests in a batch are concurrently processed by different pipeline stages.

The request fed into the accelerator has a unified operation code as shown in the top of Figure 4. The requests are decoded at each pipeline stage, and the corresponding operations are executed in an online manner. We apply the memoization technique in the updaters by using a dedicated register to store the sampled indices at each tree level so that the replay update does not need to backtrace through the tree, re-computing these indices.

4.2 Learner

The Learner takes a batch of experiences, and performs SGD constituting forward propagation (FP), loss function (LOSS), backward propagation (BP), and gradient aggregation (GA).

4.2.1 Learner on CPU and GPU. We use PyTorch [2, 19] to implement DNN training on CPUs and GPUs. On the GPU, PyTorch utilizes CuDNN [19] or Xe Matrix Extensions [2] backend to exploit SIMD parallelism. We also support using multiple streams, each stream independently processes the FP, LOSS, BP, and GA on a sub-batch of experiences. Compared to bulk processing a full batch of data, this helps overlap the data transfer and computation time between sub-batches of data. The GPU-based Learner code is parameterized to specify the number of streams.

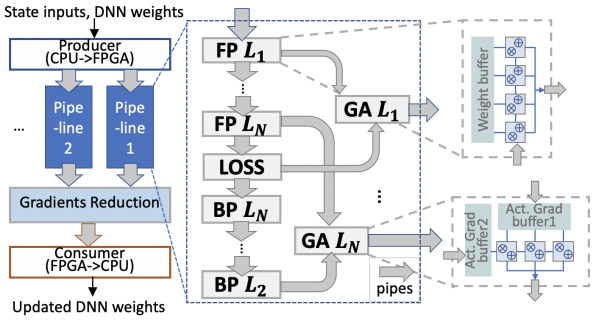


Figure 5: FPGA - Learner Hardware Module

4.2.2 Learner on FPGA. On FPGA, we design a Learner Module that supports both pipeline parallelism across different neural network layers and data parallelism among sub-batches of data. As an example, we show the design for an N-layer MLP in Figure 5. Each pipeline stage uses buffers to store intermediate activations, and uses an array of multiplier-accumulator units to compute matrix-vector multiplication for a given input. The number of multiplier-accumulator units allocated to each layer is controlled by a unique unroll factor UF, which will be tuned to ensure load balancing for best performance (Section 5.1). To realize data streaming between modules, they are connected by on-chip FIFO pipes. To support data parallelism, we make DP copies of such pipelines. Each pipeline generates the gradients for a sub-batch of experiences, which are accumulated before sending them back to the host.

5 SYSTEM COMPOSER

Given the user-specified Replay Manager (RM) and Learner metadata in the Optimizer Construction Program as inputs, the goals of the system composer are to (A) determine the best-performing accelerator configuration within each device for all the primitives, and (B) determine an optimal primitive-to-device assignment that maximizes system performance.

5.1 Accelerator Setup and Performance Estimation

To realize goal (A), we customize the parameterized accelerators described in Section 4 to suit the user-input RM and Learner specifications. Based on the customized accelerators, we obtain the expected latency of executing each primitive in one DRL iteration on each of the available devices, and store these latency numbers in a Primitive Device Assignment Matrix for further analysis of system performance in goal (B).

The Primitive Device Assignment Matrix is a 3×N table. The N rows denote the N available devices; each column refers to either one primitive or a combination of both primitives to be assigned to one device. Each entry T_y^x in the table denotes the latency of performing one iteration of a given primitive (or a combination of 2 primitives) x on device y (For RM, the latency includes times of the sampling, updates and insertions). We explain how the table entries are populated based on accelerator setups as follows:

Primitive Setup on a CPU/GPU : For the primitives that can be mapped to the CPU, i.e., RM and Actors, we allocate their number of threads initially based on the ratio of their single-iteration latency for processing/producing one experience in order to match their throughput. Note that based on this setup, if RM ends up being mapped to an accelerator that provide faster RM processing, the Actors will be initially set to occupy all available threads, and will be further dynamically adjusted based on the runtime data processing speed of Actors and Learner (Section 3.3.1). For the RM on a GPU, the degree of parallelism is set to BS. The sum tree is stored in the GPU global memory. For the Learner on a GPU, we search for the best-performing number of streams in the range $[1, BS]$ by recording their per-SGD-step latencies.

Accelerator Configuration on an FPGA: The RM and the Learner can both be mapped to the same FPGA device only if the total buffer size required by the RM and Learner modules is smaller than the total amount of SRAM resources. This is to avoid efficiency losses in accesses to off-chip memory. For the RM, the number of Autorun kernels in the pipeline is configured to match the tree depth, and the buffer sizes are configured based on their corresponding tree levels. For the Learner, the number of pipelines DP is set to the largest value within resource capacity. The amount of compute resources allocated to each pipeline stage, UF, is tuned such that all pipeline stages are load balanced (for the maximal effective hardware utilization):

$$T_{stage} = \frac{\#MAC^{\text{FP L}_1}}{UF^{\text{FP L}_1}} = \ldots = \frac{\#MAC^{\text{GA L}_N}}{UF^{\text{FP L}_N}}; \text{where } \sum UF \leq \frac{\#\text{DSPs}}{DP}. \quad (2)$$

We obtain the latency of accelerators on FPGA through performance modeling:

$$T_{RM}^{\text{sampling}} = 2 \times F \times (BS + D) + T_{comm}^{(i \to FPGA)} \quad (3)$$

$$T_{RM}^{\text{update or insert}} = 2 \times (BS + D) + T_{comm}^{(i \to FPGA)} \quad (4)$$

$$T_{Learner} = T_{stage} \times (BS + 3 \times (\#\text{layers} - 1)) \quad (5)$$

In equations 3-5, the pipeline latencies are calculated by multiplying single pipeline stage latency by the batch size BS and pipeline fill/drain overhead D (D equals the sum tree depth in RM and # layer propagation's in Learner, respectivaly). T_{comm} refers to the communication time of taking inputs from device i executing other

primitives. They are filled in Algorithm 2 - Equation 6 depending on whether the communication is within the same device (e.g., through DDR) or across different devices (e.g., through PCIe).

5.2 Heterogeneous System Composition

Algorithm 2 Heterogeneous System Composition Algorithm

1: **Input:** Primitive Device Assignment Matrix M,
2: `# Step 1: Primitive Placement`
3: D[RM], D[Learner] = $argmax_{i,j}\{\frac{\text{Iteration Batch Size}}{T_{\text{itr}}}\}$
4: where i, j denotes available devices for RM and Learner in M,

$$T_{\text{itr}} = \underbrace{T_{RM}^{\text{sampling}}(i)}_{\uparrow T_{\text{comm}}^{(i\to j)}} + \max\Big(\underbrace{T_{RM}^{\text{insert}}(i)}_{\uparrow T_{\text{comm}}^{(cpu\to i)}}, \underbrace{T_{RM}^{\text{update}}(i)}_{\uparrow T_{\text{comm}}^{(j\to i)}} + T_{\text{Learner}}(j)\Big) \quad (6)$$

5: **Output** D[Learner], D[RM]
6: `# Step 2: Memory Component Placement`
7: Initialize D[Exp Memory]; min_traffic$\leftarrow \infty$
8: $C_{\text{Learner}} \leftarrow BS \times (E+1)$; $C_{\text{Actor}} \leftarrow N_{Actor} \times E$; $C_{\text{RM}} \leftarrow BS$
9: **for** i in [Learner, Actors, RM] **do**
10: Total data traffic = $\sum^{i' \in \{\text{Learner,Actors,RM}\}} \frac{C_{i'}}{\text{bandwidth}(D[i], D[i'])}$
11: **if** Total data traffic < min_traffic **then**
12: min_traffic $\leftarrow$ Total data traffic; D[Exp Memory] $\leftarrow D[i]$;
13: **Output** D[Exp Memory]

Based on a completed Primitive Device Assignment Matrix, we develop a Heterogeneous System Composition Algorithm (Algorithm 2). It first determines the best device assignment of the primitives to maximize achievable compute throughput, then places the memory component (Exp Memory) to minimize the total data traffic.

In Step 1 (lines 2-5, Algorithm 2), the training throughput can be estimated using the processed batch size in each iteration, BS, and the iteration execution time, T_{itr}. T_{itr} is defined in Equation 6. The critical path in an iteration is the priority sampling followed by SGD training and priority update, while the other replay operations overlap with the training process. The required costs of communication with other compute modules are encapsulated in each component of Equation 6 corresponding to the candidate devices i, j for RM and Learner, where i, j are permutated to include all the device assignment choices. When $i = j$, the latencies are sampled from the third column of the Compute-Performance Table. The complexity of Step 1 is $O(N^2)$, given N available devices on the heterogeneous platform. In Step 2 (lines 7-13, Algorithm 2), we decide on the device assignment of the Exp Memory. The data traffic wrt the Exp Memory during each iteration includes BS words of sampling indices from the D_{Learner}, $BS \times E$ sampled experiences to the D_{Learner} (where E is the size of each experience for the given benchmark), and $N_{Actor} \times E$ inserted experiences from the Actors. These communication costs are denoted as C in Algorithm 2. We place Exp Memory on the device that minimizes the total data traffic based on available bandwidths between devices (e.g., PCIe) and within each device (e.g., DDR). The complexity of Step 2 is $O(1)$, as the number of primitives is constant.

6 EVALUATION

6.1 Experiment Setup

To show the portability of our toolkit to different platforms, we conduct our experiments on two heterogeneous platforms. The first platform, $Server_{CG}$, has a Host CPU and an integrated GPU that shares the same die. The second platform, $Server_{CGF}$, consists of a Host CPU connected to a GPU and an FPGA, both through PCIe with 16 GB/s bandwidth. The specifications of these platforms are summarized in Table 1. For FPGA bitstream generation, we follow the oneAPI development flow [13]. We select three widely-used RL benchmarking environments: CartPole, MountainCar, and Pong, in the OpenAI Gym software simulation environment [5]. We demonstrate our toolkit using two representative DRL algorithms widely applied in various applications, DQN [18] and DDPG [15]. The algorithm, size of the states and actions, and policy model for solving each benchmark are shown in Table 2. We evaluate the training throughput as the number of Experiences processed Per Second ($EPS = \frac{\text{Training batch size}}{T_{itr}}$, where T_{itr} is the execution time of one training iteration defined in Equation 6).

Table 1: Specification of Heterogeneous Platforms

Platform	$Server_{CG}$		$Server_{CGF}$		
Device	**CPU Intel Core i9-11900KB**	**GPU Intel UHD Graphics Xe**	**CPU Intel Xeon Gold 6326**	**GPU Nvidia Geforce 3090**	**FPGA Intel DE10-Agilex**
Processs	10 nm	10 nm	10 nm	8 nm	10 nm
Hardware Parallelism	2 sockets, 16 cores	32 Unified Pipelines	2 sockets, 64 cores	10496 CUDA Cores	4510 DSPs
External Memory	32 GB	32 GB	256 GB, DDR4	24 GB, HBM	32 GB, DDR4
Frequency	3.3 GHz	1.6 GHz	2.9 GHz	1.7 GHz	400 MHz

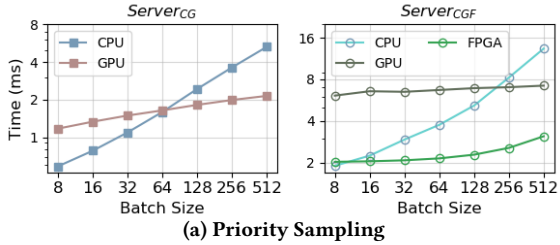
Table 2: Benchmarking Environments and Algorithms

Environment	Algorithm	State Dim.	Action Dim.	DNN Policy
CartPole	DQN	4	1	3-layer MLP, hidden size 64
MountainCar	DDPG	8	4	4-layer MLP, hidden sizes 256,128
Pong	DQN	84×84	6	CNN in [18]

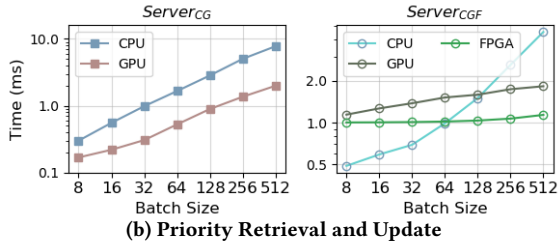
6.2 Performance of Accelerated Primitives

Since EPS is bounded by latencies of the primitives in each iteration, we first show the device assignment tradeoffs for each primitive.

In Figures 6, we present the total execution latencies for batched Replay Manager (RM) operations. They are plotted across a range of commonly used training batch sizes (a significant DRL hyper-parameter affecting DRL iteration time). For PCIe-connected GPU and FPGA on $Server_{CGF}$, all the latencies of primitives in Figure 6 include the data transfer (PCIe) time. Note that the latencies



(a) Priority Sampling



(b) Priority Retrieval and Update

Figure 6: Replay Manager

for priority retrieval and update are combined since these operations are typically performed together during priority insertion and update processes. Our observations reveal superior scalability of GPU- and FPGA-accelerated replay operations compared to the multi-threaded CPU implementation. The RM operations are memory-bound. While GPU data parallel compute resources exhibit good scalability, they are underutilized due to high-latency global memory accesses that cannot be hidden by the computations. The FPGA accelerator processes the sum tree operations in a near-memory manner, storing the data structure on-chip, thus delivering the highest scalability.

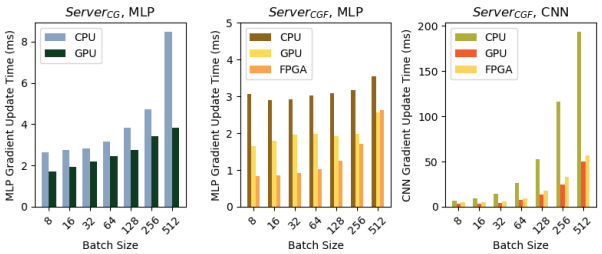
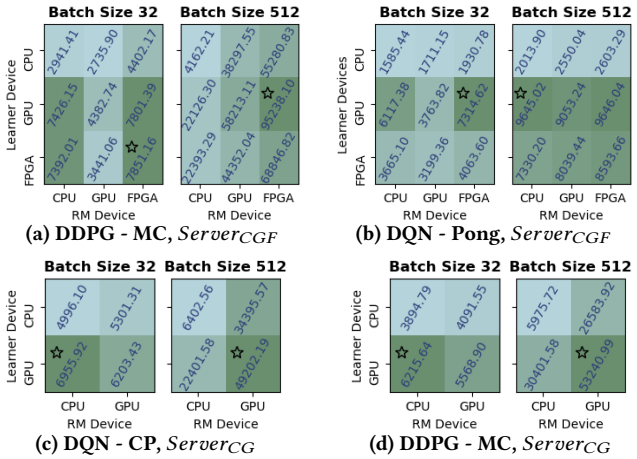


Figure 7: Learner

In Figure 7, we show the Learner execution times for one gradient update iteration. Batched layer propagations exhibit a higher arithmetic intensity compared to replay operations. Consequently, the advantages of utilizing data parallel architectures (GPUs) lead to consistently lower gradient update latency compared to CPU. The FPGA accelerator design surpasses GPU performance when arithmetic intensity is low. This is particularly evident when dealing with smaller neural network sizes and batch sizes. As the batch size increases, the execution time of training primitives on GPU begins to outperform that on FPGA. This shift is due to hidden memory overhead at larger batch computations and a higher clock frequency on the GPU.

6.3 System Composition



(a) DDPG - MC, Server$_{CGF}$ (b) DQN - Pong, Server$_{CGF}$

(c) DQN - CP, Server$_{CG}$ (d) DDPG - MC, Server$_{CG}$

Figure 8: System Composition.

In Figures 8, we show the achieved throughput *EPS* for all device assignment choices, as well as the compositions returned by the PEARL toolkit, on both heterogeneous platforms. In all the subfigures, the color gradients in the grids are proportional to the magnitudes of the achieved throughput on their corresponding device assignment. The stars denote the optimal mappings returned by our System Composer. We observe that the choice of device for the primitive with the highest latency significantly influences variations in throughput. Specifically, for small-batch computations (i.e., grid plots with batch size 32), the color gradient changes most drastically along the horizontal axis, because replay operations result in significant overheads as Learner computations are small; On the other hand, for large-batch computations (i.e., grids with batch size 512), the color gradient changes most drastically along the vertical axis, as the Learner dominates each training iteration and replay operation overheads are hidden. Note that when multiple device assignment choices lead to the same throughput, our toolkit selects the one with the lowest total data traffic (e.g., Figure 8b).

6.4 Comparison with Existing DRL Libraries

We compare PEARL-generated optimal implementations with two state-of-the-art DRL frameworks, RLlib [14] and OpenAI Stable Baselines 3 (SB3) [21], on Server$_{CGF}$. The performance of RLlib and SB3 are obtained using the optimal settings required by each of them (i.e., using GPU for training). The detailed performance across different benchmarks are shown in Table 3.

System Throughput. The additional flexibility of supporting FPGA accelerators along with our runtime optimizations enable PEARL to achieve up to 1.9×, 2.2× and 1.4× improvements in *EPS* for the three benchmarks. Even using the same set of hardware (CPU-GPU), our novel scheduling and resource allocation leads to 21% to 55% higher *EPS*. We also evaluate the effect of our runtime dynamic heterogeneous resource allocation. In our experiments, the cases where CPU actor threads are re-allocated for collaborative training are labeled with * in Table 3. These are the scenarios where the Learner requires large-batched data or a large model for training, and this re-allocation leads to a 15% to 35% improvement in *EPS*. Another study focused on mapping DRL onto FPGA-based heterogeneous platforms [28], and evaluated using the CartPole benchmark. Due to the different hardware and optimal device assignments, *EPS* is not directly comparable. Nonetheless, we compare the effective heterogeneous resource utilization (achieved throughput given the peak throughput of all the processors and accelerators in the platform). For CartPole DQN batch-32 training, PEARL achieves 7.9K *EPS* using a CPU-FPGA with a total peak performance of 0.46 TFLOPS; [28] achieved an amortized throughput of 7.1K *EPS* using a CPU-FPGA with 0.72 TFLOPS. Despite having 36% lower available device performance, our result shows a 11% higher *EPS*.

Portability. To show the performance portability of our toolkit, we adopt the portability metric for a framework to be consistent with that described in [20]:

$$\Phi(H) = \begin{cases} 0 & \text{if, } \exists i \in H, EPS_i = 0 \\ \dfrac{|H|}{\sum_{i \in H} \frac{1}{EPS_i}} & \text{otherwise} \end{cases} \quad (7)$$

where H can be either D or P: D denotes a set of device assignment choices using a single heterogeneous platform; P denotes a set of heterogeneous platforms; EPS_i is the achieved *EPS* using the i^{th} device assignment choice or platform in the set H. If the implementation cannot be portable to the i^{th} device assignment choice or platform in the set H, $EPS_i = 0$. The results are shown in the last two rows of Table 3. $\Phi(D)$ quantizes the ability to use different

Table 3: Comparison with Existing DRL Frameworks

Entry: Batch 32, 512	DQN-CartPole			DDPG-MountainCar			DQN-Pong		
	PEARL	RLlib	Stable Baselines3	PEARL	RLlib	Stable Baselines3	PEARL	RLlib	Stable Baselines3
EPS (Optimal)	7.93K, 95.2K	4.1K, 50.3K	4.6K, 56.1K	7.85K, 95.2K	3.6K, 48.5K	4.3K, 50.1K	7.3K, 9.6K*	5.6K, 6.2K	5.2K, 6.9K
EPS(*CPU-GPU*)	7.7K, 69K	4.1K, 50.3K	4.6K, 56.1K	7.4K, 58.2K*	3.6K, 48.5K	4.3K, 50.1K	6.1K, 9.6K*	5.6K, 6.2K	5.2K, 6.9K
$\Phi(D)$	4.9K, 21.4K	0, 0	0, 0	4.5K, 20.3K	0, 0	0, 0	2.9K, 4.6K	0, 0	0, 0
$\Phi(P)$	7.3K, 63.7K	3.7K, 49.8K	3.5K, 50.2K	6.9K, 68.2K	2.88K, 46.8K	3.3K, 44.5K	6.1K, 8.9K	5.4K, 6.8K	4.0K, 7.5K

heterogeneous resources given by a single platform. Other existing works that do not support accelerated RM or FPGA-based Learner are not portable to these device assignments ($\exists i \in D, EPS_i = 0$), thus having $\Phi(D) = 0$. In contrast, our work is portable to all assignment choices provided by $Server_{CGF}$. Our work enables the ability to utilize compute powers of a wider range of heterogeneous devices, thus achieving better device portability and higher performance. $\Phi(P)$ quantizes the ability to achieve performance across different platforms (i.e., both $Server_{CG}$ and $Server_{CGF}$), where EPS_i is the highest throughput achieved on the i^{th} platform. Our toolkit consistently achieves higher platform-throughput portability $\Phi(P)$ compared with the existing works.

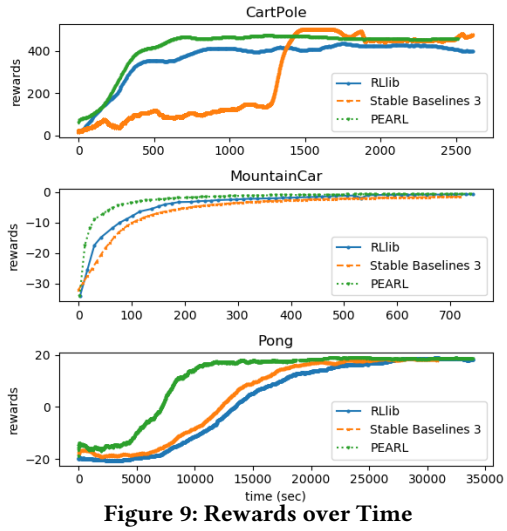


Figure 9: Rewards over Time

Algorithm Performance. Figure 9 plots the cumulative rewards collected by the agent policy over wall clock time. The curves are smoothed to show the sliding average rewards obtained in a window of 100 training iterations, and each curve is the mean of 5 runs of the algorithm-benchmark pair. For all the algorithms and benchmark applications, we consistently observe faster convergence, meaning our implementation improves throughput without significantly sacrificing algorithm performance in terms of reward and convergence rate.

6.5 User Productivity

For a quick assessment of programmability, we enlisted 5 graduate students familiar with RL but lacking expertise in heterogeneous hardware, aligning with PEARL's target user community, to implement two algorithms using PEARL. Table 4 quantifies the average development effort involved. Note that we exclude FPGA image compilation time in Table 4 (as consistent with established practice [8]), since it is an integral part of the oneAPI workflow, and is not a step directly specified by PEARL users. In addition to illustrating the effort required for developing a specific algorithm, we also present the Code Divergence (*CD*) to demonstrate productivity differences between development on the two distinct platforms. *CD* between platforms i and j is computed by $CD = 1 - \frac{|c_i \cap c_j|}{|c_i \cup c_j|}$ [20], where c represents the lines of user code. The *CD* value falls within the range [0,1]: a value of 0 indicates that a "single-source" code can be shared between both platforms, while a value of 1 implies that the user code is entirely different for the two platforms. In our case, *CD* is close to 0, as the only required changes when porting to different devices involve modifying the paths to input files.

Table 4: User Productivity

Algorithms	DQN	DDPG
User code	~75 lines	~110 lines
Development effort ◁	~12 minutes	~17 minutes
Productivity across platforms (*CD*)	~0.06	~0.04

◁ The compiling time for the FPGA image is excluded.

Overall, DRL application development through training in simulation is for tuning the best model and set of hyper-parameters before physical deployment. This requires repeated rounds of testing with different algorithms, hyper-parameters, and environmental scenarios to ensure the reliability of the agent. In state-of-the-art data centers, it is unrealistic for application users to hand-tune each round of testing. With PEARL, developers write only dozens of lines of code to generate the accelerated DRL implementation within minutes, significantly reducing the development effort and leading to more robust AI agents with faster development cycles.

7 CONCLUSION & FUTURE WORK

We presented PEARL, a toolkit for productive development of performance-portable DRL on heterogeneous platforms. Future directions include scaling the primitives across heterogeneous nodes, and developing general-purpose tools based on intermediate graph representations for mapping custom-defined training algorithms onto heterogeneous hardware.

ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation under grants CNS-2009057 and SPX-2333009, and the Intel Corporation. This work is also supported by the DEVCOM Army Research Lab under grant W911NF2220159.

REFERENCES

[1] 2021. Intel Heterogeneous DevCloud. https://devcloud.intel.com/oneapi/

[2] 2022. Intel Extension for PyTorch. https://github.com/intel/intel-extension-for-pytorch

[3] AMD. 2022. AMD Heterogeneous Accelerated Compute Clusters. https://www.amd-haccs.io/

[4] Lorena A Barba, Andreas Klockner, Prabhu Ramachandran, and Rollin Thomas. 2021. Scientific computing with Python on high-performance heterogeneous systems. *Computing in Science & Engineering* 23, 04 (2021), 5–7.

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

[6] Tianyue Cao. 2020. Study of sample efficiency improvements for reinforcement learning algorithms. In *2020 IEEE Integrated STEM Education Conference (ISEC)*. IEEE, 1–1.

[7] 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.

[8] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[9] 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.

[10] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*. PMLR, 1329–1338.

[11] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[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. 2022. Intel OneAPI. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[14] 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

[15] 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).

[16] 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.

[17] Yuan Meng, Chi Zhang, and Viktor Prasanna. 2022. FPGA acceleration of deep reinforcement learning using on-chip replay management. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 40–48.

[18] 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

[19] 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

[20] S John Pennycook, Jason D Sewall, Douglas W Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2021. Navigating performance, portability, and productivity. *Computing in Science & Engineering* 23, 5 (2021), 28–38.

[21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* 22, 268 (2021), 1–8. http://jmlr.org/papers/v22/20-1364.html

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

[23] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[24] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. 2012. Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 232–239.

[25] 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).

[26] Abdurrahman Yasar, Sivasankaran Rajamanickam, Jonathan W Berry, and Umit V Catalyurek. 2022. PGAbB: A Block-Based Graph Processing Framework for Heterogeneous Platforms. *arXiv preprint arXiv:2209.04541* (2022).

[27] Chi Zhang, Sanmukh Rao Kuppannagari, and Viktor K Prasanna. 2021. Parallel actors and learners: A framework for generating scalable RL implementations. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 1–10.

[28] Chi Zhang, Yuan Meng, and Viktor Prasanna. 2023. A Framework for Mapping DRL Algorithms With Prioritized Replay Buffer Onto Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* (2023).

A ARTIFACT APPENDIX

A.1 Abstract

We developed PEARL, a toolkit for system implementation of Deep Reinforcement Learning (DRL) on Heterogeneous platforms. From a user execution perspective, PEARL consists of two main parts: System Composer and Runtime Program. PEARL's System Composer produces the device assignments for DRL primitives (Learner and Replay Manager), given algorithm and device metadata. PEARL's Runtime Program takes a device-assignment configuration file as input and deploys the DRL training-in-simulation process on the heterogeneous platform.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Deep Q Network (DQN) and Deep Deterministic Policy Gradient (DDPG) for benchmarking PEARL.
- **Compilation:** Required: Python 3.8, Torch 2.0, CuDNN
  Optional: oneAPI and PyBind11 for compiling SYCL implementations.
- **Run-time environment:** Conda installed on a CPU with Linux OS.
- **Hardware:** A platform consisting of an Nvidia GPU connected to an Intel(R) Xeon(R) CPU via PCIe, or a CPU with integrated GPU (e.g., A core-i9 11900KB node on Intel DevCloud). Optional: an FPGA with oneAPI support connected to the CPU described above via PCIe.
- **Execution:** We have provided a set of steps to demonstrate the two parts of PEARL (e.g., for artifact evaluation), which takes approximately 10 minutes to complete.
- **Metrics:** Throughput (number of experiences or samples processed per second), rewards over time.
- **Output:** Terminal console outputs (e.g., printing messages indicating the program states), intermediate output files, and performance results.
- **Experiments:** Instructions are provided in PEARL's Github page under the "Example Usage" section (see Section A.5).
- **How much time is needed to prepare workflow (approximately)?:** Installing all the minimal dependencies through the Conda environment takes around 30 minutes.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access. We have made PEARL available at https://github.com/pgroupATusc/HeteroRL.

A.3.2 Hardware dependencies. Minimal Requirement for demonstration: A platform consisting of an Nvidia GPU connected to a multi-core CPU via PCIe or a CPU with integrated GPU (e.g., A core-i9 11900KB node on Intel DevCloud).

A.3.3 Software dependencies. The dependencies for using PEARL are explained at https://github.com/pgroupATusc/HeteroRL?tab=readme-ov-file#dependencies--installation. The minimal requirements include Conda, Python, Torch, CuDNN, and Gym.

A.4 Installation

We have provided the Conda environment file (`install_env.yml`) that contains metadata on the library dependencies for PEARL.

First, install PEARL using

`git clone` https://github.com/pgroupATusc/HeteroRL.git.

Then, install these library dependencies using

```
conda env create -f HeteroRL/install_env.yml
```

and activate the installed environment by

```
conda activate htroRLatari
```

A.5 Experiment workflow

We have provided a sequence of steps in the Example Usage section of the GitHub repository page https://github.com/pgroupATusc/HeteroRL?tab=readme-ov-file#example-usage. The Example Usage showcases three experiments: the first is for running the System Composer to generate an intermediate mapping (.json) file, the second and third are for executing the Runtime Program based on intermediate mapping files, which include two algorithm-benchmark pairs on different device mappings.

A.6 Evaluation and expected results

Main claims. Our paper presents PEARL, a toolkit for implementing DRL using heterogeneous platforms consisting of CPU, GPU, and/or FPGA devices. We claim that using PEARL, we are able to generate high *performance* implementation of DRL, *portable* to different device combinations or heterogeneous platforms while providing simple APIs to facilitate *productive* development.

Key results.

- *Portability* is demonstrated by the ability of the System Composer and Runtime Program to support various combinations of devices and interconnects. In the examples provided, we demonstrate the System Composer operating on a CPU-FPGA platform and the Runtime Program supporting two different mappings on a CPU-GPU platform. These can also be customized to support other device mappings and platforms (see Section A.7).
- *Productivity* is evident from the library interfaces (Python module imports for all primitive implementations in the libraries) and Python user APIs for specifying algorithms, device metadata, and deploying implementations. When adapting to different algorithms, the only code changes required are the Learner functions in the `Libs_Torch` directory and the configuration files (e.g., `alg_hp.json` and `Config.py`).
- *Performance* is quantified by the throughput metric, the number of experiences (samples) processed per second. As detailed in the Example Usage section of the GitHub page, these experiments demonstrate both throughput (as indicated in the console output) and reward convergence over training iterations (illustrated in output figures from Actor processes). These results corroborate our findings in Section 6.4.

A.7 Experiment customization

The experiments described in the Example Usage can be also extended to support other DRL algorithms and platforms.

Customize algorithms. To customize the algorithm hyper parameters, directly change the entries in `Config.py` and `alg_hp.json`. To implement new algorithms, additionally edit the `update_all_gradients` function, and import the corresponding modules in the Runtime Program.

Customize for platforms with configurable devices. To use SYCL implementations on integrated GPU and FPGA devices, make sure oneAPI and PyBind are installed as instructed in the "Dependencies & Installation" section of the Git page, and follow the step detailed in https://github.com/pgroupATusc/HeteroRL?tab=readme-ov-file#step-1-optional-compiling-py-sycl-libraries.