# GPU-Accelerated Atari Emulation for Reinforcement Learning

Steven Dalton, Iuri Frosio, Michael Garland
NVIDIA, USA

## Abstract

We designed and implemented a CUDA port of the Atari Learning Environment (ALE), a system for developing and evaluating deep reinforcement algorithms using Atari games. Our CUDA Learning Environment (CuLE) overcomes many limitations of existing CPU-based Atari emulators and scales naturally to multi-GPU systems. It leverages the parallelization capability of GPUs to run thousands of Atari games simultaneously; by rendering frames directly on the GPU, CuLE avoids the bottleneck arising from the limited CPU-GPU communication bandwidth. As a result, CuLE is able to generate between 40M and 190M frames per hour using a single GPU, a finding that could be previously achieved only through a cluster of CPUs. We demonstrate the advantages of CuLE by effectively training agents with traditional deep reinforcement learning algorithms and measuring the utilization and throughput of the GPU. Our analysis further highlights the differences in the data generation pattern for emulators running on CPUs or GPUs. CuLE is available at `https://github.com/NVLabs/cule`.

## 1 Introduction

Initially triggered by the success of Deep Q-Networks (DQN [17]), research in Deep Reinforcement Learning (DRL) has continually grown in popularity in the last years [13, 16, 17]. Beyond opening the door to the development of new intelligent agents that effectively interact with a complex environment, DRL soon proved to be a challenging computational problem requiring the redesign of algorithms and systems to achieve peak performance on modern architectures.

In this work we address the computational problems surrounding DRL by mostly focusing our attention on the inefficiencies along the *inference path* in Figure 1. We show that the performance bottlenecks primarily stem from several limitations inherit in all systems utilizing CPUs to generate data: the inability of CPUs to run a large set of environments simultaneously; the limited bandwidth between the CPU and GPU; and the consequent underutilization of the GPU. To mitigate these limitations we implemented CuLE (CUDA Learning Environment), a DRL library containing a CUDA enabled Atari 2600 emulator. Although the tasks exposed through Atari 2600 games are relatively simple, they emerged as an excellent benchmark for DRL [4, 14], and still represent a challenging set for the development of new DRL methods. Our CUDA optimized emulator renders frames directly on the GPU thereby avoiding off-chip communication while achieving a high level of utilization by processing thousands of environments in parallel—something that was so far achievable only through distributed systems.

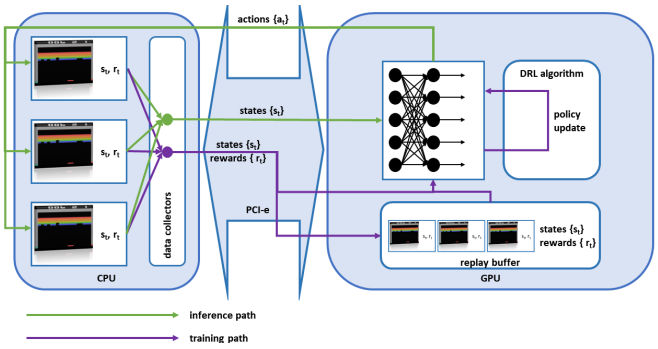



Figure 1: In a typical DRL system, environments run on CPUs, whereas GPUs execute DNN operations. The limited CPU-GPU communication bandwidth and small set of CPU environments prevent full GPU utilization.

Fig. 1 illustrates the interaction between the data generation and training components of a typical DRL system. The data generation system simulates one or more environments. It typically resides on the CPU and produces observable states and rewards at time $t$, denoted by $\{s_t\}$ and $\{r_t\}$, as the output of advancing the environment from $t-1$. Next, the data is migrated to the GPU for processing by a Deep Neural Network (DNN) to select the next action, $\{a_t\}$, followed by a copy of $\{a_t\}$ back to the CPU, and its execution. This sequence of operations defines the *inference path*, which is devoted to the generation of training data. Periodically, states and rewards are also sent to the GPU (*training path* in Fig. 1) to update the DNN, according to the training rule defined by the DRL algorithm. A computationally efficient DRL system should balance the data generation and training processes, while simultaneously minimizing the communication overhead along the *inference path* in order to consume, along the *training path*, as many data per second as possible [1, 2]. This is often a non-trivial problem and consequently many reference DRL implementations do not utilize the full computational potential of modern systems [24].

The choice of the training algorithm has a significant impact on the computational aspects of DRL. On-policy al-

gorithms, like A3C [16], GA3C [1, 2], A2C [19], or PPO [23], generally show good convergence properties, but they only use training data generated with the current policy—the rate of generation and consumption of the data are forced to be the same, and the *training engine* remains idle while data is produced along the inference path, and vice versa. On the other hand, off-policy algorithms (like DQN or Rainbow [17, 10]) have worse convergence properties but they can store stale data in a replay buffer (see Fig. 1) and use them for continuous training along the *training path*; this leads to better utilization of the GPU, and an easier implementation on distributed systems [7, 11, 24]. We characterize the tension between these two classes of algorithms, in terms germane to the high-performance computing community, as *latency-oriented* versus *throughput-oriented* methods. The strict dependency between the data generation and processing for on-policy methods means they naturally benefit by taking many low latency steps in the environment in order to process many steps sequentially. In contrast, off-policy methods are able to reuse data several times and therefore benefit from more diverse payloads of data that are processed in a bulk throughput-oriented fashion. The limitations surrounding CPU oriented environments therefore represent a barrier to the advancement of the field, as large experiment times and the well documented instability of many DRL algorithms limit the exploration of new algorithms and implicitly skew results towards low latency approaches. Note that the on- vs off-policy characterization is not mutually exclusive as many DRL methods, such as PPO[23] and IMPALA[7], incorporate ideas from both classes to trade-off between sample efficiency and performance.

In our experiments we show that CuLE generates more Frames Per Second[1] (FPS) (up to 190K for the fastest Atari game and a set of 4096 environments) compared to its CPU counterpart (which generates approximately 30K FPS), while running on a single GPU, and show the extension of CuLE to multi-GPUs. In the DRL context, FPS is a particularly relevant metric as the convergence of the training algorithm is typically related to the number of consumed frames. Some DRL algorithms, like PPO [23], are designed to be sample efficient, i.e. to reach convergence using a small amount of training data; this approach moves part of the computational load from the data generation engine to the training process, but it does not solve the computational issues that are typical of DRL and generally lead to complex algorithms that are difficult to analyze theoretically. We study the relation between FPS and the computational complexity of the training procedure for well known procedures like PPO [23], and A2C [19], and show that the adoption of CuLE is beneficial in all the aforementioned cases.

Despite the higher throughput, the FPS per environment generated by CuLE is lower when compared to the same metric measured for other data generation engines, such as OpenAI Gym [5]. In other words, the CuLE environments are larger in number, but in a vanilla implementation they explore the temporal dimension of the simulated games less efficiently, in a *throughput-oriented* manner. We analyze as well the effect of code divergence on the FPS metric, show that these two peculiarities of CuLE do not significantly affect the convergence of well-established DRL algorithms, such as PPO [23] and A2C+V-trave [7].

Our contribution can be summarized as follow:

- we identify common computational bottlenecks in several DRL implementations, that prevent the effective utilization of high throughput compute units, and effective scaling of the algorithms to distributed systems;
- we demonstrate that GPU-based environment emulation is a valid alternative to the traditional CPU-based approach, as it leads to an overall improved utilization of the computational resources;
- we identify the main advantages and limitations of the proposed GPU-based emulation approach, and highlight new research questions opened by it;
- and we introduce CuLE, a freely available DRL companion library for the GPU emulation of Atari games.

The scope of our paper is therefore dual: beyond proposing CuLE as an effective tool to develop and test DRL algorithms, we study the computational aspects of DRL, and hope that the insights provided by the analysis of the peculiarities and limitations of CuLE may be of value in the development of more efficient DRL systems.

## 2 Related Work

**RL background and speedup** In RL, an agent interacts with an environment and collects experiences to optimize a decision-making policy with the objective of maximizing the expected sum of discounted rewards, $\mathbb{E}[R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}]$, where $\gamma$ is the discount factor (e.g. 0.99), and $r_t$ is the reward at time $t$. The value of a state, $V(s_t) = \mathbb{E}[R_t|s_t]$, is the expected discounted return from that state under a given policy, $\pi$, used to select an action, $a_t = \pi(s_t)$. The Q-value, $Q(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]$, is the expected return after executing the action $a_t$ in state $s_t$. In DRL, $\pi(s_t)$, $V(s_t)$, or $Q(s_t, a_t)$ are computed by one or more DNNs.

Efforts to accelerate DRL algorithms have been underway for several years. Two approaches, not necessarily orthogonal to each other, are common: the development of new algorithms with improved sample efficiency, or system-level optimizations aimed at increasing the rate of generation and consumption of the training data. CuLE belongs to the second family, therefore we will focus on system-level optimizations for DRL. Our aim is not to be exhaustive,

[1] Raw frames are reported here and in the rest of the paper, unless otherwise specified. These are the frames that are actually emulated, but only 25% are rendered and used for training. Training frames are obtained dividing the raw frames by 4—see also [7].

but rather to highlight the computational limitations that are common to different DRL algorithms, in connection with their implementations, and in particular for those classes of algorithms that we use for our experiments. Following [24], we divide DRL algorithms into policy gradient and Q-value learning methods.

**Policy gradient algorithms** The goal of policy gradient methods is to directly learn the relation between an observed state, $s_t$, and the corresponding optimal action, $a_t$. A typical example of this class of algorithm is A3C [16], an asynchronous, on-policy, actor-critic method, that can be accelerated (4× or more) by a hybrid CPU-GPU implementation, GA3C [1, 2]. A3C uses one DNN, stored on a parameter server, to compute the policy and value functions; multiple CPU-based agents make copies of the DNN to interact with the environment in parallel and generate training experience to improve the policy. $N$-step bootstrapping is used to reduce the variance of the critic, $V(s_t;\theta)$, and train the DNN through asynchronous gradient descent, i.e., the agents send updates to the server after every $t_{max} = 5$ actions, whereas new weights are propagated to the agents after each update. A3C takes four days to learn a single Atari game (Table 1) with 16 agents on a 16 core CPU, while GA3C reaches convergence in approximately one day. Fig. 1 represents the flow of computation in the GA3C system. States and rewards are collected from multiple CPU environments in large queues, to increase the GPU occupancy along both the *inference* and *training paths*, but the limited PCI-e bandwidth prevents achieving full utilization of the CPU and GPU; consequently, scaling to multiple GPUs cannot provide any additional speed-up. PAAC [6] and A2C [19] achieve higher efficiency by removing queues and resorting to a synchronized system, but the GPU remains severely underutilized.

Several implementations of policy gradient algorithms on distributed systems have been proposed in the DRL literature. For example, IMPALA [7], an on-policy DRL system for scaling A2C to hundreds of CPU cores, runs hundreds of environments, or learners, on distributed CPUs while training is desynchronized to hide latency. As a consequence, the algorithm becomes off-policy, and V-trace is introduced to correct for off-policy training data and to stabilize the training procedure. Acceleration on a DGX-1 has also been demonstrated for on-policy, policy gradient methods (like A2C and PPO [23], a policy gradient algorithm with a modified cost function for improved sample efficiency), using large batch sizes to increase the GPU occupancy, and asynchronous distributed models that hide latency, but require periodic updates to remain synchronized [24].

Generally speaking, policy gradient methods can be sped up in different ways. In the on-policy case, the fact that data generation and consumption cannot occur concurrently, leads to unavoidable idle time on CPU or GPU. The off-policy case offers the possibility to hide this latency and scale to distributed systems; this allows increasing the number of environments and consequently the convergence stability, but the limited CPU-GPU communication bandwidth often remains an obstacle to the full utilization of the computational resources. Furthermore, the efficiency generally scales sub-linearly on distributed systems, and though Atari games can be solved in a few hours or even minutes, the cost of such systems makes them prohibitive for many researchers.

**Q-Value algorithms** In contrast with policy gradient methods, that directly search for the policy that maximizes the expected reward, Q-value methods indirectly construct a policy by first learning to estimate value of being in a particular state and selecting a specific action, $Q(s_t, a_t;\theta)$. Given a function that approximates the Q-value, a policy that exploits this knowledge may be induced by greedily selecting the action that maximizes the current Q-value estimate, $a_t^* = \text{argmax}_a\, Q(s_t, a;\theta)$. To avoid over-fitting during training an exploratory $\epsilon$-greedy policy can be used, to select a uniform random action (instead of $a_t^*$) with probability $\epsilon$.

DQN [17] optimizes the Q-value estimator by gradient descent on $\mathbb{E}[(y_t - Q(s_t, a_t;\theta))^2]$, where $y_t = r_t + \gamma \max_a Q(s_{t+1}, a;\theta^-)$ is the data-estimated Q-value. The stability of DQN is improved by using a separate "target" network $\theta^-$, periodically copied from $\theta$, to estimate the values of states during training and selecting experiences from a large cache, known as a replay buffer. The vanilla DQN algorithm has been improved and accelerated in several ways—the most relevant to mention for the scope of our discussion is Rainbow [10], that combines into a single method five different DQN enhancements: DoubleDQN [27], Dueling Networks [28], Prioritized Replay [22], n-step learning [21], and NoisyNets [8]. Gorila [18] and Ape-X DQN [11] achieve a significant (although sub-linear) speedup of DQN using hundreds of CPUs as samplers or learners, with central server for network updates. Compared to vanilla DQN, a distributed, prioritized replay buffer can support faster learning while using hundreds of CPUs for simulation and a single GPU for training - nonetheless, proper scaling of the batch size and learning rate are required, and scaling DQN or Rainbow to large distributed systems without careful hyper-parameter tuning may be problematic [11].

Q-value algorithms, and off-policy methods in general, can be mapped (and have been, see Table 1) more easily to distributed systems. Thanks to their off-policy nature a replay buffer is easily incorporated to store and reuse past experiences (see Fig. 1), which leads to higher sample efficiency (i.e., an overall smaller number of experiences is needed to reach convergence, as experiences can be used multiple times during training); furthermore, since the data generation and training engines can work in parallel, transmission latency can be hidden and a higher GPU utilization achieved. Nonetheless, implementation on large scale systems incurs a communication overhead and thus generally achieving sub-linear scaling on large distributed systems.

| Algorithm | Time | Frames | FPS | Resources | Off-Policy | Notes |
|---|---|---|---|---|---|---|
| Ape-X DQN [11] | 5 days | 22,800M | 50K | 376 cores, 1 GPU | Y | — |
| Rainbow [10] | 10 days | 200M | — | 1 GPU | Y | — |
| Distributional (C51) [3] | 10 days | 200M | — | 1 GPU | Y | — |
| A3C [16] | 4 days | — | 2K | 16 cores | N | — |
| GA3C [1, 2] | 1 day | — | 8K | 16 cores, 1 GPU | N | — |
| Prioritized Dueling [28] | 9.5 days | 200M | — | 1 GPU | Y | — |
| DQN [17] | 9.5 days | 200M | — | 1 GPU | Y | — |
| Gorila DQN [18] | 4 days | — | — | > 100 cores | Y | — |
| Unreal [12] | — | 250M | — | 16 cores | Y | — |
| Stooke (A2C / DQN) [24] | hours | 200M | 35K | 40 CPUs, 8 GPUs (DGX-1) | N/Y | — |
| IMPALA (A2C + V-Trace) [7] | mins/hours | 200M | 250K | 100-200 cores, 1 GPU | Y | — |
| CuLE (Random policy) | — | — | 40K-196K | 1 GPU | — | System I, no training, 4096 environments |
| CuLE (Inference path) | — | — | 40K-170K | 1 GPU | — | System I, no training, 4096 environments |
| CuLE (PPO) | hours | <200M | 20K | 1 GPU | Y | System II, 1024 environments |
| CuLE (A2C + V-trace) | 1 hour | 200M | 50K | 1 GPU | Y | System I, 1200 environments |
| CuLE (A2C + V-trace) | mins | 200M | 180K | 4 GPUs | Y | System III, 1200×4 environments |

Table 1: Approximate, average training times, number of raw frames to reach convergence, FPS, computational resources, compared to CuLE. Data taken from [11]; approximate FPS are taken from the corresponding papers, when available. We also indicate the on/off policy nature of each algorithm.

| System | Intel CPU | NVIDIA GPU |
|---|---|---|
| I | 12-core Core i7-5930K @3.50GHz | Titan V |
| II | 6-core Core i7-8086K @5GHz | Tesla V100 |
| III | 20-core Core E5-2698 v4 @2.20GHz × 2 | Tesla V100 × 8, NVLink |

Table 2: Systems used for experiments.

**Other Data Generation Engines** Frameworks to generate training experience already exist, like the well-known OpenAI Gym, built on top of the C++ Stella emulator for Atari games [5]. Other frameworks are targeted to specific problems like navigation [15], aim at creating large benchmark for DRL [25], or accelerate simulation on the CPU with optimized C++ multi-threaded implementations, while also providing large, GPU-friendly batches for inference and training [26]. To the best of our knowledge, however, none of these attempts directly address the problem of optimizing the *inference path*, i.e., trying to increase the number of environments while avoiding the bandwidth bottleneck, by providing a direct implementation of the environments on the GPU.

# 3 CuLE

Atari 2600 games are widely used to develop and evaluate DRL algorithms, as they strike the right balance between complexity, diversity and ease of simulation [4, 14]. Their adoption in the DRL community has been driven by the development of the Arcade Learning Environment (ALE), a framework that allows developing AI agents for Atari 2600 games, built on top of the Atari emulator Stella [4]. Despite the number and variety of games developed for the Atari 2600, the hardware is in fact relatively simple. It has a 1.19Mhz CPU and can be emulated much faster than real-time on modern hardware. The cartridge ROM (typically 2–4kB) holds the game code, while the console RAM itself holds 128 bytes. The game screen is $160 \times 210$ pixels wide, with a 128-colour palette; 18 actions can be input to the game via a digital joystick. Beyond Atari emulation, ALE provides a game-handling interface which transforms each game into a standard reinforcement learning problem by offering the accumulated scores and the observed status as an observable output of the system. ALE was later integrated into the OpenAI Gym project as well [5], but since it was written in C++ and designed to run on a CPU, it suffers from the previously mentioned drawbacks (poor scalability of the total number of environments, and CPU-GPU limited communication bandwidth), that we overcome with CuLE.

We implement CuLE in the CUDA parallel programming model, where a sequential host program executes parallel programs (known as kernels) on a GPU. Each thread runs the same scalar sequential program that, in the case of CuLE, is an Atari emulator similar to ALE in spirit. Therefore, we adopt a 1-to-1 mapping between threads and emulators (and therefore games)—in other words, each thread emulates the execution of one instance of an Atari game. Although this is not the most computationally efficient way to run an Atari emulator on a GPU, it makes the implementation relatively straight-forward and has the additional advantage that the same emulator code can be executed on the CPU for debugging and benchmarking.

We make several unique changes to the behavior of the emulator to support mapping to the CUDA architecture. In a trivial implementation a single thread may first update the game state and then update the pixels in the framebuffer for rendering the output frame. However, the contrasting nature of these two tasks, the first dominated

by reading from the RAM/ROM and writing tens of bytes to RAM, while the second writes hundreds of pixels to the framebuffer, poses a serious issue in terms of performance, such as divergence and the number of registers required per thread. We mitigate these issues by decomposing the execution into a first state update phase and a second frame rendering phase. During state update, a first CUDA kernel loads all the data related to the processor, ROM, and RAM, and executes all instructions sequentially to update the game state data with the exception of the framebuffer. Instead of writing updates to the framebuffer, each thread writes in a global buffer the updates it would have made to Television Interface Adaptor (TIA). The TIA is a secondary processor embedded in the Atari emulator whose aim is to translate these updates into frames on the display—we emulate it through a second CUDA kernel, that reads from the cached buffer and generate the frames. Since the requirements in terms of registers per thread and the chance of having divergent code are different for the state update and TIA kernels, we use two different kernels for better efficiency.

We also update the reset strategy to better fit our execution model. In the traditional emulator, the system is reinitialized by executing 64 startup frames at the end of each episode. Furthermore, wrapper interfaces for RL, such as ALE, randomly execute an additional number of frames (up to 30) to introduce randomness into initial states. The combination of reset and random initial steps add a large variability in the generated states, since 1 emulator could require up to 94 updates to produce the next frame. Resetting is not an issue when 1 emulator is mapped to a single CPU core, but it could result in massive divergence between thousands of emulators executing in SIMD fashion on a GPU. To address this issue we generate and store a cache of initial states. At the end of an episode, our emulator randomly selects one of the cached states as a seed and copies it into the terminal emulator state. By default we generate 30 seed states but the user can change this parameter at construction.

A peculiarity of CuLE is determined by the fact that, in contrast with the latency optimized nature of CPUs, GPU kernels thrive on throughput, but execute programs more slowly in a resource constrained environment. Therefore our emulators execute more slowly (less frames per second per environment) then their CPU counterparts, but may be executed in a bulk parallel fashion using thousands of threads, thus achieving an overall higher FPS, when considering the aggregate frames produced by all environments in parallel. This data generation pattern is peculiar to CuLE and possibly to the case of other environments simulated on a GPU with a similar technique.

The complexity of the CuLE engine is hidden from the end user by a Python interface which is mostly compatible with (and indeed generalizes) the OpenAI Gym syntax, the only difference being the possibility to call the *step()* function (to advance environments by one step) in a parallel fashion. The Python interface also offers the possibility to seed the environments in parallel with a state from a different CuLE system or OpenAI Gym, which may be effective to increase the diversity of the collected experiences in the case of the massive number of environments offered by CuLE, and to decorrelate the status of the environments, especially at the beginning of the training procedure, when policy gradient method tend to be easily trapped into local minima [24].

Sample code including vanilla DQN, A2C, PPO, and A2C+V-Trace is provided with the CuLE at `https://github.com/NVLABs/cule`. The same code supports running on multiple GPUs, for which we make use of the multi-GPU facilities provided by the NCCL backend of the PyTorch framework, launching one process per GPU and updating gradients in a distributed manner using the NVIDIA's NCCL library for multi-GPU communication.

Despite the fact that many of the choices made for the implementation of CuLE were informed by ease of debugging (like associating one state update kernel to one environment) or need for flexibility (like emulating the Atari console instead of directly writing CUDA code for each Atari game), the computational advantage provided by CuLE over traditional CPU emulation remains impressive. Specific CUDA implementations of any environment may be even more computationally efficient, although less flexible. This remains an interesting future direction of research.

# 4 Experiments

**Atari emulation** Here and in the following, we measure the FPS generated by CuLE and other Atari emulator engines under different load conditions of the DRL system. In the base case, that we refer to as *emulation only*, we select each action using a purely random policy. This represents an upper bound on the maximum achievable FPS. In the second case, referred to as *inference only*, we measure the FPS along the *inference path*—i.e., actions are selected accordingly to the output of a DNN before advancing each environment by one step. Data transfer between the CPU and the GPU occurs in this case when the emulators and the DNN run on different devices, while the GPU is alternatively used by CuLE for emulation and by the DNN for action selection when they run on the same device. This case is representative of the maximum throughput achievable by off-policy algorithms, when data generation and consumption can be totally decoupled and run on different devices. The last case (referred to as *training*) is the one in which the entire DRL system is at work—both inference and training occurs at the same time, possibly on the same GPU—in this case the GPU load further includes the DNN training step. This is representative of the case of on-policy algorithms, but the measured FPS strictly depends on the specific training algorithm and its implementation—we analyze in detail this complex

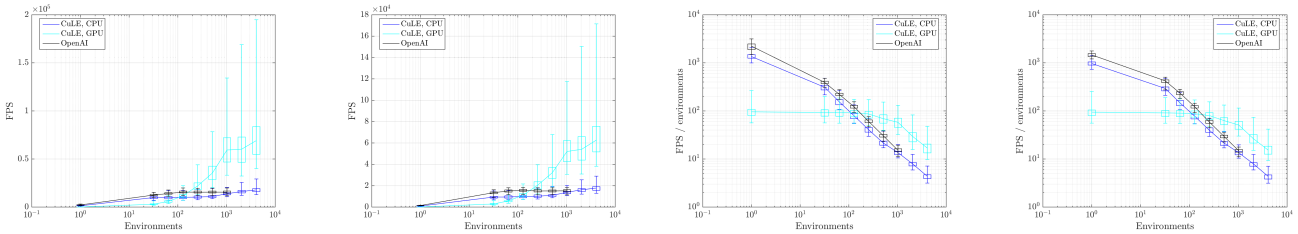
case in the last paragraph of this section, which also shows how to leverage CuLE at best to fully utilize one or more GPUs and reach convergence in a short time (see Table 3). Table 2 shows the systems used in our experiments.

We first compare (Fig. 2) emulation on the CPU and GPU by running the same Atari game through the same CuLE kernel on both devices, for a different number of environments and the entire set of Atari games, and measuring the FPS for the *emulation only* and *inference only* conditions. As a baseline, we also report the FPS measured for the widely used OpenAI Gym—in this case we had to limit the maximum number of environments because of Gym's relatively large memory footprint. When the number of environment is low ($< 128$), CPU emulation (performed either by OpenAI or by CuLE, CPU) is more efficient than its GPU counterpart—in this situation, the GPU computational power is not leveraged because of the low occupancy. For a larger number of emulated environments, on the other hand, the FPS generated by the GPU is significantly higher than that generated by the CPU (up to 3× for the median FPS generated by 4096 environment in the *inference only* case). Quite significantly, the median FPS generated by the OpenAI Gym engine reaches the maximum for 128 environments and starts decreasing for a higher number of environments in the *inference only* case, because of the additional cost associated with moving data between the CPU and the GPU and the delay introduced by the computation of the DNN forward pass—the ratio between the median FPS generated by CuLE on the GPU using 2048 environments and by OpenAI gym on the CPU with 128 environments is 3.62×.

**Factors affecting the FPS** Fig. 2 highlights that the FPS varies widely across different games: this is an effect of the game complexity, that affects both the CPU and GPU in the same way, and code divergence, which affects only the GPU. The GPU's SIMT (Single Instruction Multiple Thread) architecture serializes the execution of threads in the same warp that execute different code paths, thus decreasing total instruction throughput. The effect of game complexity on the execution speed is evident considering that Riverraid can run for *emulation only* at 29K FPS for CuLE, CPU, and 4096 environments, whereas Boxing reaches only 12K FPS in the same conditions—a 2.41× difference in terms of throughput. Since different games are also characterized by different branching factors, such difference is amplified by thread divergence on the GPU: Riverraid can run for *emulation only* at 190K FPS when emulated with CuLE, GPU and 4096 environments, while Boxing runs at 34K FPS with the same configuration—a 5.58× throughput ratio.

To investigate this aspect more in detail and better highlight the impact of thread divergence on throughput, we measure the FPS generated by CuLE, CPU and by CuLE, GPU in the *emulation only* case, for 512 environments and different Atari games (see Figs. 3- 4). Each environment is reset to the same initial state at the beginning, while the random action selection leads each environment to diverge from the others after a certain amount of steps, whose average values depends on the specific game. After some time, some environments reach the end of the episode and they are reset to the initial state to start a new one. Fig. 3 shows that, for CuLE, GPU, the FPS is maximum at the beginning of the episode, when all the environments are in a very similar state, therefore the chance that all the environment execute the same instruction in the kernel is high. After some frames, each environment evolves into a different state because of the randomness of the actions, and code divergence negatively impacts the FPS on the GPU, until it reaches its asymptotic value. After the first environments start resetting, however, this correlation is lost. The effect of code divergence on FPS can reach 30% in the worst case—this has to be compared with case of complete divergence within each thread and for each instruction, which would yield $1/32 = 3\%$ of the peak performances. Some minor oscillations of the FPS are also visible especially for games with a repetitive pattern, such as Breakout or Pong, where different environments can indeed be more or less correlated with a typical oscillation frequency. Running the same environments on the CPU (Fig. 4) does not show the performance peak at the beginning of the emulation, as divergence does not affect multithreading on a CPU—high frequency oscillations of the FPS are present in this case, but possibly due to interference with the operating system or again emulation of more or less computationally demanding states in the games.

Such behavior has interesting implications for DRL, when the simulator run on the GPU as in the case of CuLE—keeping the environment states aligned can increase the FPS, and thus generate more training frames per second, but this at the same time may be detrimental for the training procedure, as it would minimize the diversity of the experiences (and in fact the state of each environment is typically randomized at the beginning to maximize diversity). However, it is important to remember that divergence negatively affects the number of executed instruction per seconds only within a single warp; the fact that CuLE can manage a high number of environments, suggests that an optimal compromise can be reached by keeping environments within the same warp almost aligned, while at the same time maximizing the diversity of the environments in different warps. This is one the justifications for the introduction of the reset operation in the CuLE API, and suggests a future research direction aimed at finding the best compromise between computational and algorithmic optimizations in DRL, e.g. aimed at sampling interesting reset states that maximize the amount of information generated for effectively learning a policy.

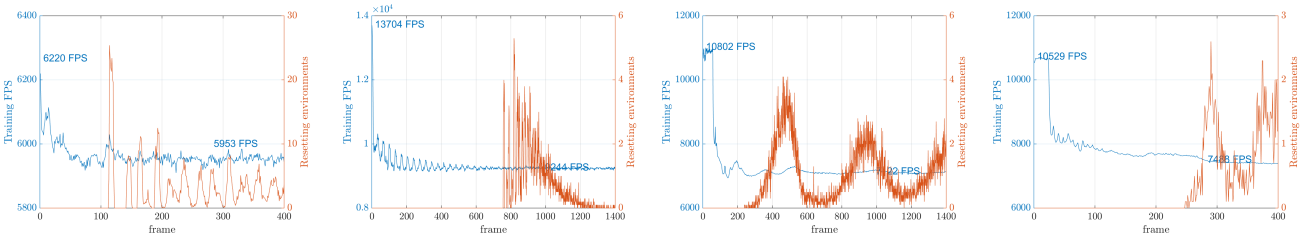



(a) FPS, *emulation only* (b) FPS, *inference only* (c) FPS per environment, *emulation only* (d) FPS per environment, *inference only*

Figure 2: FPS and FPS per environment on System I in Table 2, for OpenAI Gym [19], CuLE, CPU (i.e., the CuLE kernel running on a CPU), and CuLE, GPU, as a function of the number of environments, under two different load conditions: *emulation only* (i.e., when actions are selected accordingly to a purely random policy, no data transfer between the CPU and the GPU), and *inference only* (i.e., when actions are selected accordingly to the output of a policy DNN on the GPU). The boxplots indicate the minimum, $25^{th}$, $50^{th}$, $75^{th}$ percentiles and maximum values, measured on the entire set of 57 Atari games.

| Engine | OpenAI Gym | | | | CuLE, 1 GPU | | | CuLE, 4 GPUs | Game |
|---|---|---|---|---|---|---|---|---|---|
| Envs | 120 | 120 | 120 | 1200 | 1200 | 1200 | 1200 | 1200×4 | |
| Batches | 1 | 5 | 20 | 20 | 1 | 5 | 20 | 20×4 | |
| N-steps | 5 | 5 | 20 | 20 | 5 | 5 | 20 | 20 | |
| SPU | 5 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | |
| Training FPS | 3.6K | 3.0K | 2.7K | 3.8K | 12.3K | 11.9K | 11.3K | 44.9K | Pong |
| UPS | 5.9 | 24.9 | 22.3 | 3.2 | 2.0 | 9.9 | 9.4 | 9.3 | |
| Time [mins] | 11.2 | 14.5 | 5.0 | 6.9 | — | 4.7 | 2.9 | 1.9 | |
| Training frames (for average score: 18) | 2.4M | 2.6M | 0.8M | 1.6M | — | 3.4M | 2.0M | 5.2M | |
| Training FPS | 3.3K | 2.9K | 2.6K | 3.5K | 9.7K | 9.5K | 9.0K | 36.3K | Pacman |
| UPS | 5.6 | 23.8 | 21.4 | 2.9 | 1.6 | 7.9 | 7.5 | 7.6 | |
| Time [mins] | 33 | 42 | 35 | 41 | — | 19 | 20 | 4.6 | |
| Training frames (for average score: 1,500) | 6.6M | 7.2M | 5.4M | 8.7M | — | 11.1M | 10.8M | 10.0M | |

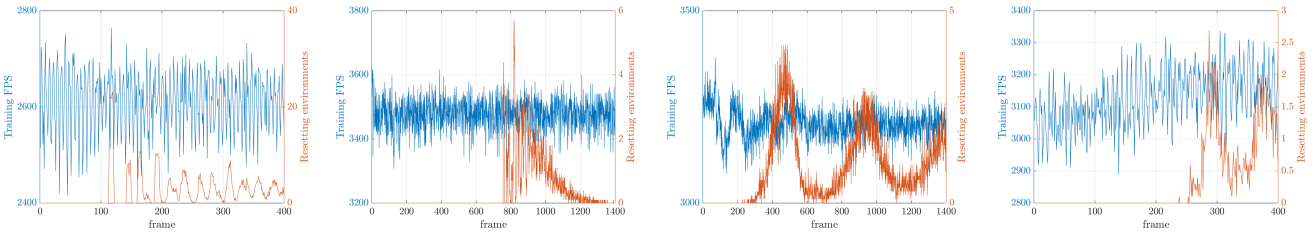
Table 3: Training FPS (the raw FPS are 4×), DNN's Update Per Second (UPS), convergence time (arbitrarily defined as the time, in minutes, to reach a score of 18 for Pong, and 1,500 for Ms-Pacman) and the corresponding number of training frames for A2C+V-trace and different configurations of the data generation engine, measured on System I in Table 2.




(a) Breakout, GPU (b) Pong, GPU (c) Ms Pacman, GPU (d) Space Invaders, GPU

Figure 3: Training FPS for different Atari games and 512 CuLE environments on the GPU, measured for System I in Table 2; actions are selected accordingly to a completely random policy; each panel also shows the number of resetting environments. CuLE generates more training FPS at the beginning, when all environments are in similar states and divergence within warps is minimized; once the environments start resetting, correlation is lost and FPS stabilizes. Minor oscillations in FPS can be observed, possibly associated to more or less computational demanding phases in the simulation of the environments—e.g., when a life is lost in the case or Breakout, or a goal is scored in the case of Pong.

**Performances during training** Figs. 5(a)-5(d) compares the FPS generated by different emulation engines on four specific Atari games, and different load conditions of the GPU (*emulation only*, *inference only*, and *training*), for 32, 512, and 2048 environments. In the *training* case, we execute the complete A2C loop, thus also including the DNN training step along the *training path* in Fig. 1. As already noticed in Fig. 2, the FPS decreases when moving from *emulation only* to *inference only*, because of the additional GPU time allocated for DNN inference. For CuLE, GPU and a high number of environments, we find an even more evident decrease of FPS for the *training* conditions,




(a) Breakout, CPU (b) Pong, CPU (c) Ms Pacman, CPU (d) Space Invaders, CPU

Figure 4: Training FPS for different Atari games and 512 CuLE environments on the CPU, measured for System I in Table 2; actions are selected accordingly to a completely random policy; each panel also shows the number of resetting environments. Oscillations in FPS are possibly associated to more or less computational demanding phases in the simulation of the environments—e.g., when a life is lost in the case or Breakout, or a goal is scored in the case of Pong.

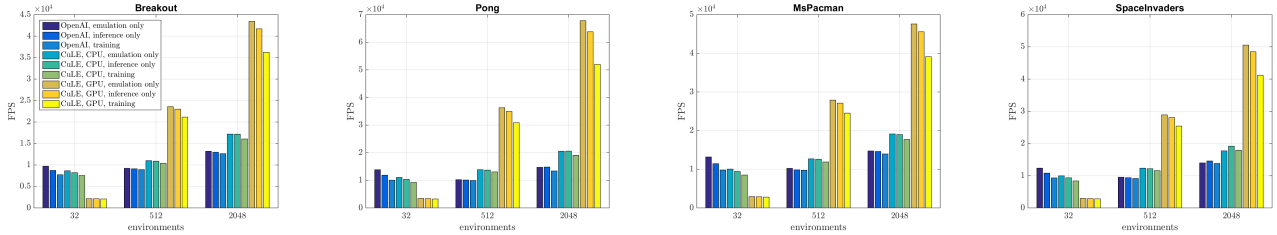



Figure 5: FPS generated by different emulation engines on System I in Table 2 for different Atari games, as a function of the number of environments, and different load conditions; for the *training* case, the main A2C [19] loop is run here.

which is on the other hand absent or less pronounced for OpenAI Gym and CuLE, CPU. This is expected in the case of System I in Table 2, which is equipped with a single GPU used both for computing the forward and the training passes of the DNN; as the number of environments grows, the batch size also grows and, beyond a certain point, more GPU time has to be allocated on the GPU for the training step (how to handle large batches with A2C and V-Trace is described in the last paragraph of this Section). In this case, the entire DRL procedure is bounded by the available computational resources. This is an important difference with respect to existing DRL implementations where either the CPU's computational power, for emulating environments, or the bandwidth limitations represent the main bottleneck and scaling to multiple GPUs is not convenient, as in the case of GA3C [1, 2], or sub-optimal, as in the case of some distributed systems as [7, 24]. By avoiding the bandwidth bottleneck, CuLE best leverages the computational power of the GPU, significantly reduces the idle time, and achieves a higher occupancy by running DNN and a large number of CuLE kernels on the same device. It is also important to notice that, when data generation and training can be decoupled, as in the case of DQN and many other off-policy algorithms, training can be easily moved to a different GPU and the *inference path* can be used at maximum speed, as in the inference case reported here. Table 1 shows that the average FPS generated by CuLE during *emulation only* and *inference only* on a single GPU is comparable to that achieved by large, and more costly, distributed systems. The same Table reports the FPS for CuLE during training with different DRL algorithms - in the case of PPO on a single GPU, our implementation suffers from an additional synchronization overhead (due to the on-policy nature of the algorithm that put the data generation and training engines in competition for the GPU computational resources) that does not affect others. Off-policy algorithms implemented on distributed systems are reported in the same Table.

**Frames per second per environment** Fig. 2(c)-2(d) show the statistics for the FPS per environment generated by three different emulation engines on System I, as a function of the total number of emulated environments. For a limited number of environments, the CPU-based emulators generate frames at a much higher pace compared to CuLE GPU—this is reasonable as CPUs are optimized for low latency, and thus execute a higher number of instructions per second per thread. However, the FPS per environment decreases from the very beginning with the overall number of environments, since several environments have to share the same core. Since the GPU is designed to maximize throughput, it is characterized by a lower number of instructions per second per thread, which results in a smaller FPS for a small number of environments as already observed for CuLE GPU. Contrary the situation on the CPU, the FPS per environment remains practically constant on the GPU up to 256 environments (or, more generally speaking, until all the cores of the GPU are fully utilized), and starts decreasing after this point. As for the FPS, the trend per environment in *emulation only* and *inference only* cases are

very similar. The overall result is a higher throughput for the GPU, which generates more FPS compared to its CPU counterpart for a high number of environments. On the other hand, at the level of a single environment, frames are generated at a slower pace compared to that used for the development of traditional DRL algorithms and systems. In the context of DRL, this means that data generated by CuLE GPU span the environments direction efficiently, by providing larger statistics about the rewards that can be collected starting from a given state, and consequently lowering the variance of the value estimate, but collect samples in the temporal domain less efficiently, thus not helping much in reducing the bias in the estimate of the value. This opens new research questions about the development of new algorithms that can best leverage the large amount of data generated by CuLE GPU, as shown for instance in the last paragraph of this Section.

**Other limitations** Generating data directly on the GPU using a massive number of agents presents several challenges with respect to managing the relatively small amount of GPU DRAM. For example our implementation of A2C, using PyTorch [20], requires each environment to store 4 gray scale frames downsampled to 84x84 and some additional variables related to the simulator state. Although for 16K environments this translates into 1GB of memory, the primary issue is the combined memory pressure to store the DNN with 4M parameters and the meta-data during training. The large amount of training data generated by 16K environments easily exhausts the total amount of DRAM, when a single GPU is used. To mitigate this issue we constrain the training configuration to fewer than 5K environments, which is less than half the number of environments required to achieve half of peak FPS performance. We do not implement here any data compression scheme as in [11], although this may be part of future developments for CuLE and enable the use of a massively large amount of environments even on a single GPU. Training with multiple GPUs can also help mitigating the issue, especially in the case of decoupled inference and training which is typical of off-policy methods.

**Convergence curves** Given the fact that data generated by CuLE GPU spans the temporal and environment dimensions in very dissimilar ways from the widely adopted OpenAI Gym environment (see Fig. 2), we examine the convergence behavior of traditional DRL algorithms and CuLE GPU. Specifically, we evaluate DRL training with data generated by CuLE, GPU on a set of diverse games—Breakout, Ms Pacman, Pong, and Space Invaders—to demonstrate convergence. We imitate the evaluation procedures of [17, 27] on 57 Atari 2600 games from the arcade learning environment [4]: the testing scores are evaluated at regular intervals during training, by suspending learning and evaluating the latest agent on 10 or more episodes, that are truncated at 108K frames (or 30 minutes of simulated play), as in [27], if needed. Each training procedure is repeated at least five times with different DNN's weight initializations to collect sufficient statistics. For each game we train the DRL agent on System II in Table 2 with PPO [23], a policy-gradient method with improved sample efficiency, with hyper-parameters outlined in Table 4. Notice that in this case we do not optimize the hyperparameters for the large number of environments generated by CuLE, GPU. We also analyze with more details the case of A2C [19], an on-policy, gradient policy algorithm, in the next paragraph.

| Parameter | Value |
|---|---|
| Actors | 256 |
| Adam learning rate | 0.0005 |
| Adam $\epsilon$ | $1.5 \times 10^{-4}$ |
| Steps | 4 |
| Epochs (PPO) | 4 |
| Number of batches (PPO) | 4 |

Table 4: PPO hyperparameters.

Training is performed on a total of 50M training frames using 256 OpenAI Gym CPU environments and 256, 512, or 1,024 CuLE, GPU environments. The PPO testing scores as a function of the training time are illustrated in Fig. 6. Convergence is generally reached (or approached) in a time ranging from few minutes to few hours. In the case of Pong, Ms Pacman, and Space Invaders, the increase of the number of environments and FPS provided by CuLE leads to slightly faster convergence. However, the overall gain in terms of training time is not impressive in this case for two reasons: first of all, PPO is designed to be sample efficient, thus increasing the number of consumed frames provides only a marginal advantage; second and possibly more important, we do not optimize the algorithm hyperparameters to manage the large number of environments allowed by CuLE, which in the worst case (Breakout) leads to slower convergence for a larger number of environments. As expected, increasing the number of environments produces to more stable convergence: the standard deviations of the score is generally smaller when 1,024 environments are used. The bad performance of CuLE, GPU with 256 environments in Breakout and of OpenAI Gym with 256 environments in Space Invaders are in fact explained by at least one training run collapsing to a minimal score; we do not observe this undesirable behavior when the number of environments is large. In the next paragraph, we show in details how to fully leverage the high throughput generated by CuLE to achieve a significant speedup in terms of convergence time and effective scaling to multiple GPUs.

**A2C and V-trace** We first analyze in full details the use of CuLE with A2C on a single GPU. Beyond showing that faster convergence can be achieved with CuLE, this analysis further highlights the interconnections between the computational aspects and the convergence properties of the DRL algorithm, thus demonstrating how to use CuLE to gain further insights in the DRL field.

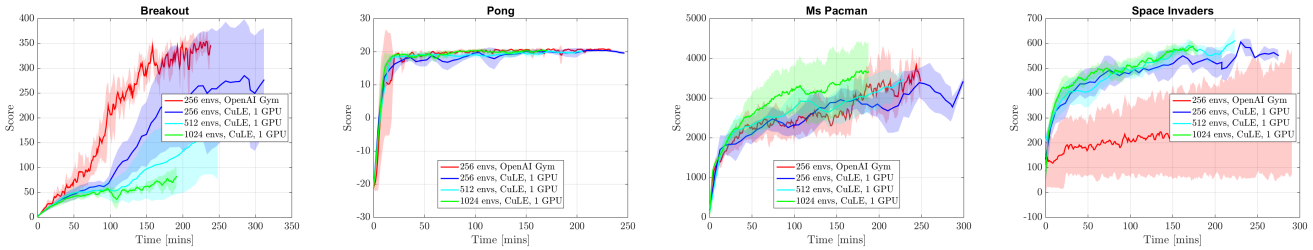
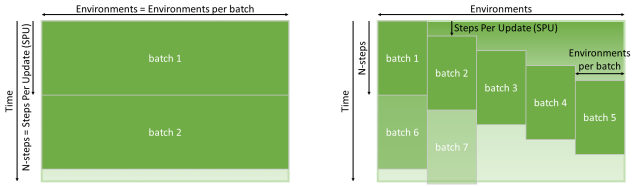


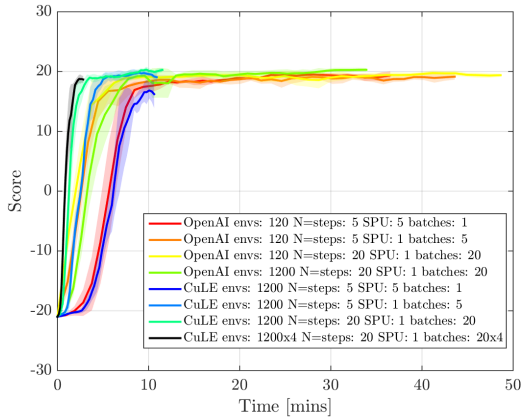

Figure 6: Average testing score (continuous lines) with standard deviation (shaded area) on four Atari games as a function of the training time, for PPO and system II in Table 2, using OpenAI and CuLE, GPU to emulate 50M training frames using a different number of environments. The same PPO hyperparameters in Table 4 have been used in all cases.



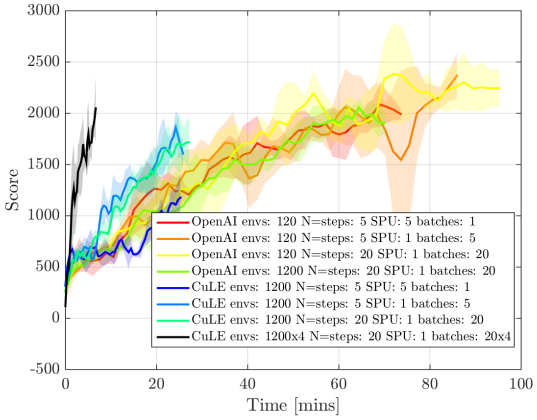

(a) Single batch strategy (b) Multi batch strategy

Figure 7: Single and multi batch batching strategies. Different strategies are defined by the number of batches, N-Steps and Steps Per Update (SPU) parameters. The on-policy single batch case is a special case of the more general, off-policy multi batch approach. The batching strategy affects both the computational and convergence aspects of the DRL algorithm, as shown in Fig. 8 and in Table 3.

As a baseline, we consider the vanilla A2C implementation, running 120 OpenAI Gym CPU environments; all the environments advance by N-steps=5 steps before sending their data to the GPU to update the DNN weights (Fig. 7(a)). This configuration takes on the average 11.2 minutes (and 2.4M frames) to reach a score of 18 for Pong and 33 minutes (and 6.6M frames) for a score of 1,500 and Ms-Pacman (Fig. 8, red line, and first column of Table 3). Using ten times more environments, CuLE GPU generate $3-4\times$ more FPS compared to OpenAI Gym, but the score as a function of the training time is lower (blue line, Fig. 8). Two factors contribute to this outcome: CuLE GPU generates less frames per second per environment compared to the baseline (see Fig. 2(a)-2(b)), therefore it also performs less Updates Per Second (UPS) to the DNN weights; furthermore, the CuLE GPU batch contains a larger number of environments, which leads to smaller variance in the estimate of the value function, but the temporal dimension is explored less efficiently in terms of time, which may be detrimental for learning.




(a) Pong (8M training frames total)




(b) Ms-Pacman (15M training frames total)

Figure 8: Average testing score (continuous lines) and standard deviation (shaded areas) on Pong and Ms-Pacman as a function of the training time, for A2C+V-trace and System I in table 2, and different batching strategies (see Table 3 for more statistics). The black line represents the case of 4 GPUs for System III in Table 2.

To better leverage CuLE, and somehow similar in spirit to the approach described for IMPALA [7], we employ a different batching strategy, illustrated in Fig. 7(b): all environments advance in parallel, but training data are read in batches to update the DNN weights every Steps Per Update (SPU) steps. It is important noticing that this strategy

requires off-policy correction, as only the most recent data in a batch are generated with the most recent policy. We use the V-trace off-policy correction firstly adopted by IMPALA [7]. This batching strategy significantly increases the DNN's Update Per Second (UPS, see second columns of OpenAI Gym and CuLE GPU in Table 3), at the cost of a slight decrease of FPS, due to the fact that the GPU has to dedicate more time to training. The net result is an increase of the overall training time when 120 OpenAI Gym CPU environments are used, as this configuration pays for the increased training and communication overhead, while the smaller batch size generates more noisy updates of the DNN weights. On the other hand, in the case of 1200 environments generated by CuLE GPU, the multi-batch strategy reduces the time to reach a score of 18 for Pong and 1,500 for Pacman respectively to 4.7 and 19 minutes. The number of frames required to reach the same score is indeed higher for CuLE GPU, which is therefore less sample efficient when compared to the baseline, but the higher FPS largely compensates for this. Extending the batch size in the temporal dimension (by setting N-steps=20) further increases the computational load of the GPU and thus reduces both the FPS and UPS, but it also reduces the bias in the estimate of the value function, making each DNN update step more effective, resulting in an overall decrease of the wall clock training time, the fastest convergence being achieved by CuLE GPU with 1,200 environments (and requiring less then 3 and 20 minutes to reach a score of 18 or 1,500 for Pong and Ms-Pacman respectively). Using OpenAI to run 1,200 environments and using the same batching strategy results in a similar sample efficiency, as expected, but in a longer training time because of the lower FPS generated by CPU emulation. It is worthy noticing that these same batching issues potentially affect CuLE combined with PPO (or other DRL algorithms) and therefore the training curves in Fig. 6. On the other hand, a multibatching strategy as the one illustrated here can significantly improve the convergence rate.

Finally, the black line in Fig. 8 represents the case of A2C+V-Trace coupled with CuLE and 4 GPUs, each simulating 1200 environments. This leads to an increase in FPS that grow almost linearly with the number of GPUs (the FPS generated and consumed by CuLE is in this case similar to that generated by IMPALA on a large CPU cluster) and a dramatic reduction of the convergence time, which goes down to 1.9 minutes to reach a score of 18 with Pong and 4.6 minutes to reach a score of 1,500 on Ms-Pacmann, as reported in the rightmost column of Table 3.

Overall, our analysis shows once more that, compared to OpenAI Gym, CuLE guarantees faster and more stable convergence; the batching strategy plays a significant role in this, as CuLE allows splitting a large set of environments into large batches, that guarantee a low variance in the estimate of the value, that are also long in the temporal direction, that guarantee small bias. This eventually leads to faster experimental turnaround time during the development and analysis of existing and novel DRL algorithms.

| Algorithm | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
|---|---|---|---|---|
| PPO | 9.38 | 4.45 | 2.12 | 1.1 |
| Rainbow DQN | 10.5 | 5.1 | 3.2 | 2.1 |

Table 5: Hours to complete 50M training frames (200M raw frames), by GPU count, measured on System III in Table 2, for two different DRL algorithms. The number of environments on each GPU is 2,048.

**Scaling to multiple GPUs** In the former paragraph we have shown almost linear scaling on multiple GPUs for A2C+V-trace. More generally speaking, by moving execution of the simulators onto the GPU CuLE can easily take advantage of multiple GPUs within a single system for different DRL algorithms. In Table 5 we report the time required to generate 50M updates (or training frames) of PPO and Rainbow DQN by running CuLE from 1 to 8 GPUs on a single system (notice that inter-GPU communication benefits from NVLinks for System III in Table 2). We use the PyTorch multiprocessing facilities to launch 1 process for each GPU and update gradients in a distributed manner using the NVIDIA NCCL multi-GPU communication backend. The data reported in this Table show efficient scaling of the training FPS with the number of GPUs. As in the case of A2C+V-trace, an efficient batching strategy is then required to fully leverage the high throughput generated by CuLE.

**Generalization for different systems** Table 6 reports the FPS measured for systems I and II in Table 2—the acceleration in terms of FPS provided by CuLE GPU is consistent across different systems, different algorithms, and larger in percentage when a large number of environments is used. Different DRL algorithms achieve different FPS depending on the complexity and frequency of the training step on the GPU (e.g., 26.5K for CuLE, GPU and DQN vs. 51K for CuLE, GPU and A2C on System I).

The same Table also reports the minimum and maximum GPU utilization measured while running each DRL algorithm. Off-policies algorithms, like DQN, are characterized by long GPU idle times occurring during CPU emulation, leading to a low GPU utilization. The GPU utilization increases when emulation is moved to the GPU, while GPU peak utilization is reached during the DNN training steps. A more efficient use of the computational resources may be achieved by running CuLE for *inference only* on a single GPU while using a second GPU for the DNN operations. GPU underutilization can be observed also for policy-gradient algorithms (A2C, PPO) when emulation is done on the GPU; peak utilization is higher for PPO, because of the major computational complexity of the training step of this algorithm. Nearly full GPU utilization is achieved only by CuLE, GPU for a large number of environments—in this case the entire system is bounded by

| Emulation engine | DRL algorithm | FPS [GPU utilization %] | | | |
|---|---|---|---|---|---|
| | | System I [256 envs] | System I [1024 envs] | System II [256 envs] | System II [1024 envs] |
| OpenAI | DQN | 6.4K [15-42%] | 8.4K [0-69%] | 10.8K [26-32%] | 21.2K [28-75%] |
| CuLE CPU | DQN | 7.2K [16-43%] | 8.6K [0-72%] | 6.8K [17-25%] | 20.8K [8-21%] |
| CuLE GPU | DQN | 14.4K [16-99%] | 25.6K [17-99%] | 11.2K [48-62%] | 33.2K [57-77%] |
| OpenAI | A2C | 12.8K [2-15%] | 15.2K [0-43%] | 24.4K [5-23%] | 30.4K [3-45%] |
| CuLE CPU | A2C | 10.4K [2-15%] | 14.2K [0-43%] | 12.8K [1-18%] | 25.6K [3-47%] |
| CuLE GPU | A2C | 19.6K [97-98%] | 51K [98-100%] | 23.2K [97-98%] | 48.0K [98-99%] |
| OpenAI | PPO | 12K [3-99%] | 10.6K [0-96%] | 16.0K [4-33%] | 19.2K [4-62%] |
| CuLE CPU | PPO | 10K [2-99%] | 10.2K [0-96%] | 9.2K [2-28%] | 18.4K [3-61%] |
| CuLE GPU | PPO | 14K [95-99%] | 36K [95-100%] | 14.4K [43-98%] | 28.0K [45-99%] |

Table 6: Average FPS and min/max GPU utilization during training for Pong with different algorithms and using different emulation engines on different systems (see Table 2); CuLE consistently leads to higher FPS and GPU utilization.

the GPU computational capability and scaling to multiple GPU is beneficial.

# 5 Discussion and Conclusion

The common allocation of the tasks in a DRL system dictates that environment simulation should run on CPUs, whereas GPUs should be dedicated to DNN operations [24]. Whereas most of the existing frameworks to generate training data in DRL [9, 15, 25, 26] follow this paradigm, bandwidth limitation and the limited capability to increase the number of emulated environments on CPUs constitute two limiting factors to effectively accelerate DRL algorithms, even when mapped to expensive distributed systems. By rendering frames directly on the GPU, CuLE seeks to overcome these limitations and offers the possibility to generate as many FPS as those generated by large, expensive CPU systems on a single GPU (see Table 1).

Given these premises, CuLE promises to be an effective instrument to develop and test DRL algorithms by significantly reducing the experiment turnaround. In the case of Q-value methods, CuLE offers a mechanism to generate training frames at high FPS on a single GPU, which is generally less costly when compared to a distributed system. But CuLE promises to be effective also for the acceleration of policy gradient methods, where the high number of agents can significantly reduce the variance in the estimate of the value function, especially when the batching scheme is taken into consideration and the algorithm can be slight off-policy, as in the case of A2C+V-Trace considered here.

Our analysis further highlights that CuLE generates training frames on a GPU with a atypical pattern that is distinct from data generated by CPU emulators: since GPUs are characterized by a number of instructions per second per thread which is lower compared to that of a CPU, CuLE does achieve on overall higher FPS, but the number of frames per second per environment is higher for the CPU. In other words, the experiences created by CuLE are generated by a large number of agents, but narrow in the time direction. This can be problematic for problems with sparse temporal rewards solved by a naive implementation of a DRL algorithm, but rather than considering this as a pure limitation of CuLE, we believe that this peculiarity opens the door to new interesting research questions. For instance, one could emulate a limited number of agents on the CPU, to effectively explore the temporal dimension of an RL problem, and identify interesting states with a mechanism similar to importance sampling [10, 28]. CuLE agents in a given warp may then be reset to the same interesting state, to avoid code divergence and best leverage the high FPS offered by CuLE to focus processing on states that contain valuable information for training, and thus in practice building a run-time replay buffer for on-policy algorithms. Our analysis of the A2C algorithm coupled with V-trace shows how to use CuLE as an effective instrument to investigate the complex interaction between the computational aspects and the convergence properties of a DRL algorithm. We also highlight that, although CuLE overcomes the limitations due to a limited number of available CPUs, it does hit a new obstacle, which is the limited amount of DRAM available on the GPU; studying new compression schemes, like the one proposed in [10], as well as training methods with smaller memory footprints may help extend the utility of CuLE to even larger environment counts. Since these are only two of the possible research directions for which CuLE is an effective investigation instrument, CuLE comes with a python interface that allows easy experimentation and is freely available to any research at `https://github.com/NVLABs/cule`.

# References


[1] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. GA3C: gpu-based A3C for deep reinforcement learning. *CoRR abs/1611.06256* (2016).

[2] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *ICLR* (2017).

[3] Bellemare, M. G., Dabney, W., and Munos, R. A distributional perspective on reinforcement learning. *CoRR abs/1707.06887* (2017).

[4] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (jun 2013), 253–279.

[5] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

[6] Clemente, A. V., Martínez, H. N. C., and Chandra, A. Efficient parallel methods for deep reinforcement learning. *CoRR abs/1705.04862* (2017).

[7] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR abs/1802.01561* (2018).

[8] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. Noisy networks for exploration. *CoRR abs/1706.10295* (2017).

[9] Greg, B., Vicki, C., Ludwig, P., Jonas, S., John, S., Jie, T., and Wojciech, Z. Openai gym.

[10] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *CoRR abs/1710.02298* (2017).

[11] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *CoRR abs/1803.00933* (2018).

[12] Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397* (2016).

[13] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *CoRR abs/1509.02971* (2015).

[14] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR abs/1709.06009* (2017).

[15] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., Kumaran, D., and Hadsell, R. Learning to navigate in complex environments. *CoRR abs/1611.03673* (2016).

[16] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning* (New York, New York, USA, 20–22 Jun 2016), M. F. Balcan and K. Q. Weinberger, Eds., vol. 48 of *Proceedings of Machine Learning Research*, PMLR, pp. 1928–1937.

[17] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (Feb. 2015), 529–533.

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

[19] OpenAI. Openai baselines: Acktr & a2c, 2017.

[20] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[21] Peng, J., and Williams, R. J. Incremental multi-step q-learning. *Machine Learning 22*, 1 (Mar 1996), 283–290.

[22] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *CoRR abs/1511.05952* (2015).

[23] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR abs/1707.06347* (2017).

[24] Stooke, A., and Abbeel, P. Accelerated methods for deep reinforcement learning. *CoRR abs/1803.02811* (2018).

[25] Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., de Las Casas, D., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T. P., and Riedmiller, M. A. Deepmind control suite. *CoRR abs/1801.00690* (2018).

[26] Tian, Y., Gong, Q., Shang, W., Wu, Y., and Zitnick, L. ELF: an extensive, lightweight and flexible research platform for real-time strategy games. *CoRR abs/1707.01067* (2017).

[27] van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. *CoRR abs/1509.06461* (2015).

[28] Wang, Z., de Freitas, N., and Lanctot, M. Dueling network architectures for deep reinforcement learning. *CoRR abs/1511.06581* (2015).