GPU Computing Coursework: CUDA-Accelerated Prioritized Experience Replay (PER) Sum-Tree
Theory, Implementation, Tests, and Expected Results

Aditya Pandey

February 9, 2026

Contents

1 Overview

This report documents a CUDA-accelerated implementation of a **sum-tree** (flat segment tree) to support **Prioritized Experience Replay (PER)** for reinforcement learning. The project provides:

- A CPU reference sum-tree (`cpu_tree.py`),
- A GPU sum-tree wrapper (`tree.py`) backed by a PyTorch C++/CUDA extension (`per_ext.cpp`, `per_kernels.cu`),
- A baseline GPU sampler using `cumsum + searchsorted` (`baseline_cumsum.py`),
- A minimal GPU PER replay buffer (`replay.py`),
- Benchmarks (`bench_per.py`),
- Unit tests for updates, sampling distribution, and replay (`test_update.py`, `test_sample.py`, `test_per_replay.py`),
- An example CartPole training script using PER (`train_cartpole.py`).

The design goal is to make priority **updates** and **sampling** efficient on GPU without recomputing a full prefix-sum CDF every time.

2 Reference Paper

This implementation is based on the PER idea introduced in:

- T. Schaul et al., *Prioritized Experience Replay*, arXiv:1511.05952, 2015. DOI: 10.48550/arXiv.1511.05952.

3 Theory

3.1 Prioritized Experience Replay (PER)

In PER, transitions are sampled non-uniformly, typically proportional to a priority value derived from TD-error:

$$p_i \leftarrow (|\delta_i| + \epsilon)^\alpha$$

and the sampling probability becomes

$$P(i) = \frac{p_i}{\sum_k p_k}.$$

The exponent $\alpha \in [0, 1]$ controls prioritization strength; $\epsilon > 0$ prevents zero probability.

3.2 Importance Sampling (IS) Weights

PER biases the training distribution, so learning is corrected using importance sampling weights:

$$w_i = (N \cdot P(i))^{-\beta},$$

where N is the current buffer size and β anneals from β_0 toward 1. Typically weights are normalized:

$$\tilde{w}_i = \frac{w_i}{\max_j w_j} \leq 1.$$

3.3 Why a Sum-Tree / Segment Tree

A baseline approach samples via a CDF:

$$\text{cdf}[j] = \sum_{k \le j} p_k,$$

then binary-search for a uniform $u \in [0, \sum p)$. On GPU, `torch.cumsum` + `torch.searchsorted` is easy but recomputing `cdf` is $\Theta(N)$.

A sum-tree stores hierarchical partial sums so that:

- **update** of one leaf is $\mathcal{O}(\log L)$,
- **sample** of one index is $\mathcal{O}(\log L)$,

where L is the next power-of-two above capacity.

Sampling: draw $u \sim \mathcal{U}[0, \text{total})$ and descend:

$$\begin{aligned} &\text{if } u \le s_{\text{left}} : \text{go left} \\ &\quad \text{else } : u \leftarrow u - s_{\text{left}},\ \text{go right} \end{aligned}$$

until reaching a leaf.

4 Build and Runtime Requirements

4.1 Prerequisites

- Python + PyTorch with CUDA support
- A CUDA-capable GPU and CUDA toolkit compatible with the PyTorch build
- A C++ compiler toolchain suitable for PyTorch extensions

4.2 How the extension is compiled

Two entry points exist:

- `per/_ext.py` compiles/loads at import-time using `torch.utils.cpp_extension.load`.
- `build_ext.py` compiles the extension as a standalone smoke test.

5 Implementation Structure (Files)

File	Responsibility
`cpu_tree.py`	CPU reference flat sum-tree (NumPy): update + sample + total.
`tree.py`	GPU sum-tree wrapper around the extension: update, sample, priority queries.
`baseline_cumsum.py`	Baseline GPU sampler using `cumsum+searchsorted`.
`_ext.py`	Loads/compiles CUDA extension at import time.
`per_ext.cpp`	PyBind11 bindings for CUDA functions.
`per_kernels.cu`	CUDA kernels: update, coalesced update, sample.
`replay.py`	Minimal PER replay buffer on GPU using `GpuSumTree`.
`bench_per.py`	Benchmarks comparing update/sample latency across implementations.
`train_cartpole.py`	Example training script on CartPole using `GPUReplayPER`.
`test_update.py`	Tests sum-tree update invariants (root sum, internal sums).

`test_sample.py`	Tests sampling distribution matches priorities (approx).
`test_per_replay.py`	Tests replay sampling shapes and priority update behavior.

6 Core Data Structure: Flat Sum-Tree Layout

Both CPU and GPU implementations use the same flat representation:

- Let capacity be C (not necessarily power-of-two).
- Let $L = 2^{\lceil \log_2 C \rceil}$.
- Tree array has size $2L$ (index 0 unused, root at index 1).
- Leaves are stored at indices $[L, L + C)$.

Invariants:

$$\text{tree}[i] = \text{tree}[2i] + \text{tree}[2i + 1] \quad \text{for all internal } i,$$

and

$$\text{tree}[1] = \sum_{k=0}^{C-1} p_k.$$

7 CPU Reference: `cpu_tree.py`

7.1 Intent

`CpuSumTree` is a correctness reference. Its behavior should match the GPU version, making it useful for validation and debugging.

7.2 Functions (line-by-line intent)

- `__init__`: compute L, allocate tree array.
- `update(idx, new_p)`:
  - for each update: compute delta at leaf,
  - apply delta upwards to the root.
- `sample(u)`: descend from root to leaf using the left-sum rule.
- `total`: return root sum.

7.3 Full source

Listing 1: cpu_tree.py

```
import numpy as np

class CpuSumTree:
    """
    Flat sum-tree on CPU (numpy).
    - capacity = N (not forced to power of 2)
    - internal nodes stored in tree[1..2*L-1], leaves at [L..L+N-1]
    """
    def __init__(self, capacity: int):
        self.capacity = int(capacity)
        self.L = 1 << (self.capacity - 1).bit_length()
        self.tree = np.zeros(2 * self.L, dtype=np.float32)

```

```
    def update(self, idx: np.ndarray, new_p: np.ndarray):
        # idx: [B] int64, new_p: [B] float32
        for i, p in zip(idx, new_p):
            pos = self.L + int(i)
            delta = float(p) - float(self.tree[pos])
            self.tree[pos] = float(p)

            pos >>= 1
            while pos >= 1:
                self.tree[pos] += delta
                pos >>= 1

    def sample(self, u: np.ndarray) -> np.ndarray:
        # u: [B] float32 in [0, tree[1])
        out = np.empty(u.shape[0], dtype=np.int64)
        for j, x0 in enumerate(u):
            x = float(x0)
            node = 1
            while node < self.L:
                left = node << 1
                s_left = float(self.tree[left])
                if x <= s_left:
                    node = left
                else:
                    x -= s_left
                    node = left + 1
            leaf = node - self.L
            if leaf >= self.capacity:
                leaf = self.capacity - 1
            out[j] = leaf
        return out

    @property
    def total(self) -> float:
        return float(self.tree[1])
```

8 GPU Sum-Tree Wrapper: `tree.py`

8.1 Intent

`GpuSumTree` stores the tree as a CUDA `float32` tensor and delegates update/sample to the extension.

8.2 Key operations

- `update`: calls `per_ext.update(tree, idx, new_p, L)`.
- `sample`: draws stratified uniforms and calls `per_ext.sample`.
- `leaf_priorities`: reads leaf values directly from `tree`.
- `update_coalesced`: alternative update kernel for lower contention.

8.3 Full source

Listing 2: tree.py

```
import torch
from ._ext import per_ext
```

```
3
4  class GpuSumTree:
5      """
6      Flat segment tree:
7        - L = next power of two >= capacity
8        - tree is size 2*L, float32 on CUDA
9        - root at index 1
10       - leaves at [L : L+capacity)
11     """
12     def __init__(self, capacity: int, device: str = "cuda"):
13         assert capacity > 0
14         self.capacity = int(capacity)
15         self.L = 1 << (self.capacity - 1).bit_length()
16         self.device = torch.device(device)
17         self.tree = torch.zeros(2 * self.L, device=self.device, dtype=
               torch.float32)
18
19     @property
20     def total(self) -> torch.Tensor:
21         return self.tree[1]
22
23     def leaf_values(self) -> torch.Tensor:
24         return self.tree[self.L : self.L + self.capacity]
25
26     def update(self, idx: torch.Tensor, new_p: torch.Tensor):
27         """
28         idx: int64 CUDA tensor of shape [n] in [0, capacity)
29         new_p: float32 CUDA tensor of shape [n]
30         """
31         per_ext.update(self.tree, idx, new_p, self.L)
32
33     def sample(self, batch_size: int) -> torch.Tensor:
34         # total priority mass on GPU
35         total = self.tree[1].item()
36         if total <= 0.0:
37             raise RuntimeError("Cannot sample: total priority is 0")
38
39         # stratified sampling: u_i in [i*seg, (i+1)*seg)
40         seg = total / batch_size
41         u = (torch.arange(batch_size, device=self.tree.device, dtype=torch
               .float32) +
42              torch.rand(batch_size, device=self.tree.device, dtype=torch.
                   float32)) * seg
43
44         return per_ext.sample(self.tree, u, self.L, self.capacity)
45
46     def leaf_priorities(self, idx: torch.Tensor) -> torch.Tensor:
47         """
48         idx: int64 CUDA tensor [B] in [0, capacity)
49         returns: float32 CUDA tensor [B] of leaf priorities
50         """
51         return self.tree[self.L + idx]
52
53     def total_priority(self) -> torch.Tensor:
54         """Return total priority mass (CUDA scalar tensor)."""
55         return self.tree[1]
56
57     def update_coalesced(self, idx, new_p):
58         per_ext.update_coalesced(self.tree, idx, new_p, self.L)
```

9 Baseline Sampler: `baseline_cumsum.py`

9.1 Intent

This baseline recomputes a full CDF via `torch.cumsum` and uses `torch.searchsorted`. It is correct but typically slower for frequent updates due to $\Theta(N)$ prefix sum recomputation.

9.2 Full source

Listing 3: baseline_cumsum.py

```
import torch

class GpuCumsumSampler:
    """
    Baseline GPU sampler using cumsum + searchsorted.
    Priorities stored as a flat CUDA tensor.
    """
    def __init__(self, capacity, device="cuda"):
        self.capacity = capacity
        self.device = device
        self.priorities = torch.zeros(capacity, device=device, dtype=torch
            .float32)

    @torch.no_grad()
    def update(self, idx, new_p):
        self.priorities[idx] = new_p

    @torch.no_grad()
    def sample(self, batch_size):
        cdf = torch.cumsum(self.priorities, dim=0)
        total = cdf[-1]

        seg = total / batch_size
        u = (torch.arange(batch_size, device=self.device) +
             torch.rand(batch_size, device=self.device)) * seg

        idx = torch.searchsorted(cdf, u, right=False)
        idx.clamp_(0, self.capacity - 1)
        return idx
```

10 Extension Loader: `_ext.py`

10.1 Intent

Uses PyTorch's `load()` to compile and load a C++/CUDA extension at import-time. It points to:

- `ext/per_ext.cpp`
- `ext/per_kernels.cu`

10.2 Full source

Listing 4: _ext.py

```
from pathlib import Path
from torch.utils.cpp_extension import load

_THIS_DIR = Path(__file__).resolve().parent
```

```
_ROOT = _THIS_DIR.parent

# Unique name avoids collisions if you have multiple builds
_EXT_NAME = "per_ext"

per_ext = load(
    name=_EXT_NAME,
    sources=[
        str(_ROOT / "ext" / "per_ext.cpp"),
        str(_ROOT / "ext" / "per_kernels.cu"),
    ],
    extra_cuda_cflags=["-O3", "--use_fast_math"],
    extra_cflags=["-O3"],
    verbose=True,
)
```

11 Standalone Builder: `build_ext.py`

11.1 Intent

Compiles the extension without importing the package, and performs a minimal CUDA availability check.

11.2 Full source

Listing 5: build_ext.py

```
from torch.utils.cpp_extension import load
import torch

per_ext = load(
    name="per_ext",
    sources=["ext/per_ext.cpp", "ext/per_kernels.cu"],
    extra_cuda_cflags=["--use_fast_math"],
    extra_cflags=["-O3"],
    verbose=True,
)

if __name__ == "__main__":
    print("Built per_ext OK")
    # quick smoke test
    assert torch.cuda.is_available()
    print("CUDA available:", torch.cuda.is_available())
```

12 C++ Binding Layer: `per_ext.cpp`

12.1 Intent

Registers the CUDA entry points into a Python module via PyBind11:

- `update(tree, idx, new_p, L)`
- `update_coalesced(tree, idx, new_p, L)`
- `sample(tree, u, L, capacity)`

12.2 Full source

Listing 6: per_ext.cpp

```cpp
#include <torch/extension.h>

// CUDA forward decls
void update_cuda(torch::Tensor tree,
                 torch::Tensor idx,
                 torch::Tensor new_p,
                 int64_t L);

void update_coalesced_cuda(torch::Tensor tree,
                           torch::Tensor idx,
                           torch::Tensor new_p,
                           int64_t L);

torch::Tensor sample_cuda(torch::Tensor tree,
                          torch::Tensor u,
                          int64_t L,
                          int64_t capacity);

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("update", &update_cuda, "Sum-tree update (CUDA)");
    m.def("update_coalesced", &update_coalesced_cuda, "Sum-tree update (
        warp-coalesced) (CUDA)");
    m.def("sample", &sample_cuda, "Sum-tree sample (CUDA)");
}
```

13 CUDA Kernels: `per_kernels.cu`

13.1 Design intent and correctness

Update kernel. Each thread updates one leaf:

1. Locate leaf `pos = L + idx[i]`.
2. Use `atomicExch` to set leaf to the new value and retrieve the old value.
3. Compute `delta = new - old`.
4. Propagate `delta` upward using `atomicAdd` on parents until reaching the root.

`atomicExch` makes duplicate indices in the same update batch safe.

Coalesced update kernel. Attempts to reduce atomic contention by grouping updates within a warp when threads share the same parent (uses warp intrinsics).

Sample kernel. Each thread performs a tree descent using its own uniform variate `u[i]`. The final leaf index is clamped to `capacity-1` if it lands in the padded region.

13.2 Full source

Listing 7: per_kernels.cu

```cpp
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <ATen/cuda/CUDAContext.h>
```

```
5
6  static inline void check_cuda(torch::Tensor t, const char* name) {
7      TORCH_CHECK(t.is_cuda(), name, " must be a CUDA tensor");
8      TORCH_CHECK(t.is_contiguous(), name, " must be contiguous");
9  }
10
11 // GPU UPDATE (naive)
12 __global__ void update_kernel(
13     float* tree,              // size: 2*L (index 0 unused, root at 1)
14     const int64_t* idx,        // leaf indices in [0, capacity)
15     const float* new_p,        // new priorities
16     int64_t n,
17     int64_t L
18 ) {
19     int i = blockIdx.x * blockDim.x + threadIdx.x;
20     if (i >= n) return;
21
22     int64_t pos = L + idx[i];
23
24     // atomic exchange at leaf to handle duplicates safely
25     float old = atomicExch(&tree[pos], new_p[i]);
26     float delta = new_p[i] - old;
27
28     // propagate delta to parents
29     pos >>= 1;
30     while (pos >= 1) {
31         atomicAdd(&tree[pos], delta);
32         pos >>= 1;
33     }
34 }
35
36 void update_cuda(torch::Tensor tree, torch::Tensor idx, torch::Tensor
       new_p, int64_t L) {
37     check_cuda(tree, "tree");
38     check_cuda(idx, "idx");
39     check_cuda(new_p, "new_p");
40
41     const auto n = idx.numel();
42     const int threads = 256;
43     const int blocks = (n + threads - 1) / threads;
44
45     update_kernel<<<blocks, threads, 0, at::cuda::getDefaultCUDAStream()
           >>>(
46         tree.data_ptr<float>(),
47         idx.data_ptr<int64_t>(),
48         new_p.data_ptr<float>(),
49         n,
50         L
51     );
52 }
53
54 // helper: warp-level shuffle reduction on delta values (optional)
55 static __device__ __forceinline__ float warp_reduce_sum(float v) {
56     for (int offset = 16; offset > 0; offset >>= 1) {
57         v += __shfl_down_sync(0xffffffff, v, offset);
58     }
59     return v;
60 }
61
62 // GPU UPDATE (warp-coalesced atomics)
```

```
__global__ void update_kernel_coalesced(
    float* tree, const int64_t* idx, const float* new_p, int64_t n,
        int64_t L
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;

    int64_t pos = L + idx[i];
    float old = atomicExch(&tree[pos], new_p[i]);
    float delta = new_p[i] - old;

    // propagate delta to parents (warp-coalesced atomics)
    pos >>= 1;
    while (pos >= 1) {
        // group threads by parent index to reduce atomics
        int64_t parent = pos;
        // find lanes with the same parent
        unsigned mask = __match_any_sync(0xffffffff, parent);
        float v = delta;
        // full warp reduction, then only leader does atomicAdd
        float all = warp_reduce_sum(v);
        int leader = __ffs(mask) - 1;
        if ((threadIdx.x & 31) == leader) {
            atomicAdd(&tree[parent], all);
        }
        pos >>= 1;
    }
}

void update_coalesced_cuda(torch::Tensor tree, torch::Tensor idx, torch::
    Tensor new_p, int64_t L) {
    check_cuda(tree, "tree");
    check_cuda(idx, "idx");
    check_cuda(new_p, "new_p");

    const auto n = idx.numel();
    const int threads = 256;
    const int blocks = (n + threads - 1) / threads;

    update_kernel_coalesced<<<blocks, threads, 0, at::cuda::
        getDefaultCUDAStream()>>>(
        tree.data_ptr<float>(), idx.data_ptr<int64_t>(), new_p.data_ptr<
            float>(), n, L);
}

// GPU SAMPLE
__global__ void sample_kernel(
    const float* tree, const float* u, int64_t* out, int64_t B, int64_t L,
        int64_t capacity
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= B) return;

    float x = u[i];
    int64_t node = 1;
    while (node < L) {
        int64_t left = node << 1;
        float s_left = tree[left];
        if (x <= s_left) {
            node = left;
```

```
        } else {
            x -= s_left;
            node = left + 1;
        }
    }
    int64_t leaf = node - L;
    if (leaf >= capacity) leaf = capacity - 1;
    out[i] = leaf;
}

torch::Tensor sample_cuda(torch::Tensor tree, torch::Tensor u, int64_t L,
    int64_t capacity) {
    check_cuda(tree, "tree");
    check_cuda(u, "u");

    const auto B = u.numel();
    auto out = torch::empty({B}, torch::TensorOptions().device(u.device())
        .dtype(torch::kInt64));

    const int threads = 256;
    const int blocks = (B + threads - 1) / threads;
    sample_kernel<<<blocks, threads, 0, at::cuda::getDefaultCUDAStream()
        >>>(
        tree.data_ptr<float>(), u.data_ptr<float>(), out.data_ptr<int64_t
            >(), B, L, capacity);
    return out;
}
```

13.3 Expected results from kernels

- After any sequence of updates, `tree[1]` equals the sum of leaf values (within float tolerance).
- Sampling approximates the intended discrete distribution proportional to leaf priorities.
- Coalesced updates should have similar correctness; performance depends on contention patterns.

14 Minimal GPU PER Replay Buffer: `replay.py`

14.1 Intent

`GPUReplayPER` stores transitions in GPU tensors and uses `GpuSumTree` for proportional sampling and importance weights.

14.2 Detailed behavior (function-by-function)

`PERBatch` dataclass. Holds sampled batch tensors: `obs, act, rew, next_obs, done, weights, indices`.

`__init__`. Allocates GPU storage for transitions and initializes:

- ring buffer pointers `pos` and `size` on CPU,
- a `GpuSumTree` of size `capacity`,
- `_max_priority` for initializing new transitions.

`add`. Writes one transition at `pos`, sets its tree priority to `_max_priority`, then advances the ring.

`sample`.

- Computes β by linear anneal from `beta0` to 1.0 over `beta_steps`.
- Samples indices proportional to priorities via `tree.sample`.
- Gathers tensors with advanced indexing `storage[idx]`.
- Computes normalized IS weights on GPU:

$$w = (N \cdot p)^{-\beta}, \quad w \leftarrow \frac{w}{\max w}.$$

`update_priorities`. Computes new leaf priorities:

$$p \leftarrow (|\mathrm{td}| + \epsilon)^{\alpha}$$

and updates the sum-tree. Tracks `_max_priority` for future insertions.

14.3 Full source

Listing 8: replay.py

```python
import torch
from dataclasses import dataclass
from .tree import GpuSumTree

@dataclass
class PERBatch:
    obs: torch.Tensor
    act: torch.Tensor
    rew: torch.Tensor
    next_obs: torch.Tensor
    done: torch.Tensor
    weights: torch.Tensor
    indices: torch.Tensor

class GPUReplayPER:
    """
    Minimal GPU PER replay:
      - stores transitions in CUDA tensors
      - priorities in a GPU sum-tree
      - proportional sampling + importance weights
    """
    def __init__(
        self,
        capacity: int,
        obs_shape,
        device: str = "cuda",
        alpha: float = 0.6,
        beta0: float = 0.4,
        eps: float = 1e-6,
        dtype_obs=torch.float32,
    ):
        self.device = torch.device(device)
        self.capacity = int(capacity)
        self.alpha = float(alpha)
        self.beta0 = float(beta0)
        self.eps = float(eps)

        # ring buffer state (keep on CPU for simplicity)
```

```
39          self.pos = 0
40          self.size = 0
41
42          # storage on GPU
43          self.obs = torch.empty((capacity, *obs_shape), device=self.device,
                dtype=dtype_obs)
44          self.next_obs = torch.empty((capacity, *obs_shape), device=self.
               device, dtype=dtype_obs)
45          self.act = torch.empty((capacity,), device=self.device, dtype=
               torch.int64)
46          self.rew = torch.empty((capacity,), device=self.device, dtype=
               torch.float32)
47          self.done = torch.empty((capacity,), device=self.device, dtype=
               torch.float32)
48
49          # priorities
50          self.tree = GpuSumTree(capacity, device=device)
51          self._max_priority = 1.0  # CPU scalar is fine
52
53      def __len__(self):
54          return self.size
55
56      @torch.no_grad()
57      def add(self, obs, act, rew, next_obs, done):
58          """
59          obs/next_obs can be numpy or torch; will be copied to GPU.
60          done should be bool or 0/1.
61          """
62          i = self.pos
63
64          # copy to GPU tensors
65          self.obs[i].copy_(torch.as_tensor(obs, device=self.device))
66          self.next_obs[i].copy_(torch.as_tensor(next_obs, device=self.
               device))
67          self.act[i] = int(act)
68          self.rew[i] = float(rew)
69          self.done[i] = float(done)
70
71          # set priority for this slot to current max (common PER trick)
72          p = torch.tensor([self._max_priority], device=self.device, dtype=
               torch.float32)
73          idx = torch.tensor([i], device=self.device, dtype=torch.int64)
74          self.tree.update(idx, p)
75
76          # advance ring
77          self.pos = (self.pos + 1) % self.capacity
78          self.size = min(self.size + 1, self.capacity)
79
80      @torch.no_grad()
81      def sample(self, batch_size: int, step: int, beta_steps: int = 200_000
            ) -> PERBatch:
82          """
83          beta anneals from beta0 to 1.0 over beta_steps.
84          """
85          if self.size == 0:
86              raise RuntimeError("Cannot sample from empty buffer")
87
88          # beta schedule
89          frac = min(1.0, step / float(beta_steps))
90          beta = self.beta0 + frac * (1.0 - self.beta0)
```

```
91
92          # sample indices proportional to priorities
93          idx = self.tree.sample(batch_size)  # int64 CUDA [B]
94
95          # gather transitions
96          obs = self.obs[idx]
97          next_obs = self.next_obs[idx]
98          act = self.act[idx]
99          rew = self.rew[idx]
100         done = self.done[idx]
101
102         # importance weights
103         total = self.tree.total_priority()  # CUDA scalar
104         prio = self.tree.leaf_priorities(idx).clamp_min(self.eps)
105         p = prio / total.clamp_min(self.eps)
106
107         N = float(self.size)
108         weights = (N * p).pow(-beta)
109         weights = weights / weights.max().clamp_min(self.eps)
110
111         return PERBatch(obs=obs, act=act, rew=rew, next_obs=next_obs, done
                =done, weights=weights, indices=idx)
112
113     @torch.no_grad()
114     def update_priorities(self, indices: torch.Tensor, td_error: torch.
            Tensor):
115         """
116         indices: int64 CUDA [B]
117         td_error: float tensor [B] on CUDA
118         priority = (|td| + eps)^alpha
119         """
120         p = (td_error.abs() + self.eps).pow(self.alpha).to(torch.float32)
121         self.tree.update(indices, p)
122
123         # track max priority on CPU to init new entries
124         max_p = float(p.max().item())
125         if max_p > self._max_priority:
126             self._max_priority = max_p
```

15 Benchmarks: `bench_per.py`

15.1 Intent

Measures update and sample latency for:

- `GpuSumTree` (extension),
- baseline `GpuCumsumSampler`,
- CPU reference `CpuSumTree`.

15.2 Expected benchmark trends

- With frequent updates, sum-tree should outperform `cumsum` baseline by avoiding full prefix-sum recomputation.
- Sampling performance depends on capacity and batch size, typically favoring GPU for large workloads.

15.3 Full source

Listing 9: bench_per.py

```
import time
import torch
import numpy as np

from per.tree import GpuSumTree
from per.baseline_cumsum import GpuCumsumSampler
from per.cpu_tree import CpuSumTree


def time_cuda(fn, iters=100):
    # warmup
    for _ in range(5):
        fn()
    torch.cuda.synchronize()

    t0 = time.time()
    for _ in range(iters):
        fn()
    torch.cuda.synchronize()
    return (time.time() - t0) / iters


def time_cpu(fn, iters=200):
    for _ in range(10):
        fn()
    t0 = time.time()
    for _ in range(iters):
        fn()
    return (time.time() - t0) / iters


def benchmark(capacity, batch_size, iters=200):
    print(f"\n=== capacity={capacity} batch={batch_size} ===")
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # Data
    idx_gpu = torch.randint(0, capacity, (batch_size,), device=device,
       dtype=torch.int64)
    p_gpu = torch.rand(batch_size, device=device, dtype=torch.float32)

    # GPU sum-tree
    if device == "cuda":
        t = GpuSumTree(capacity)

        def upd_tree():
            t.update(idx_gpu, p_gpu)

        def samp_tree():
            _ = t.sample(batch_size)

        t_upd = time_cuda(upd_tree, iters=iters)
        # Need non-zero total before sample timing
        t.update(torch.arange(capacity, device="cuda", dtype=torch.int64),
                 torch.ones(capacity, device="cuda", dtype=torch.float32))
        t_samp = time_cuda(samp_tree, iters=iters)

```

```
        print(f"GpuSumTree update: {t_upd*1e6:.2f} us")
        print(f"GpuSumTree sample: {t_samp*1e6:.2f} us")

        # Baseline cumsum sampler
        b = GpuCumsumSampler(capacity)
        b.update(idx_gpu, p_gpu)

        def upd_base():
            b.update(idx_gpu, p_gpu)

        def samp_base():
            _ = b.sample(batch_size)

        t_upd2 = time_cuda(upd_base, iters=iters)
        # make priorities non-zero
        b.priorities[:] = 1.0
        t_samp2 = time_cuda(samp_base, iters=iters)

        print(f"Baseline update:   {t_upd2*1e6:.2f} us")
        print(f"Baseline sample:   {t_samp2*1e6:.2f} us")

    # CPU sum-tree
    c = CpuSumTree(capacity)
    idx_cpu = np.random.randint(0, capacity, size=(batch_size,), dtype=np.
        int64)
    p_cpu = np.random.rand(batch_size).astype(np.float32)

    def upd_cpu():
        c.update(idx_cpu, p_cpu)

    def samp_cpu():
        total = c.total
        if total <= 0:
            return
        u = (np.arange(batch_size, dtype=np.float32) + np.random.rand(
            batch_size).astype(np.float32)) * (total / batch_size)
        _ = c.sample(u)

    t_upd_cpu = time_cpu(upd_cpu, iters=iters)
    # make non-zero
    c.update(np.arange(capacity, dtype=np.int64), np.ones(capacity, dtype=
        np.float32))
    t_samp_cpu = time_cpu(samp_cpu, iters=iters)

    print(f"CpuSumTree update: {t_upd_cpu*1e6:.2f} us")
    print(f"CpuSumTree sample: {t_samp_cpu*1e6:.2f} us")


def main():
    caps = [2**12, 2**14, 2**16, 100_000]
    batches = [256, 1024, 4096]
    for cap in caps:
        for b in batches:
            benchmark(cap, b, iters=200)


if __name__ == "__main__":
    main()
```

16 Example Training Script: `train_cartpole.py`

16.1 Intent

Demonstrates how `GPUReplayPER` can be used in a DQN-like loop for `CartPole-v1`, including:

- epsilon-greedy action selection,
- replay insertions,
- PER sampling with IS weights,
- TD-error-based priority updates,
- periodic target network updates.

16.2 Expected outcome

On a correct setup (and with reasonable hyperparameters), training should learn a policy that achieves high CartPole return. Exact learning curves vary by random seed and system.

16.3 Full source

Listing 10: train_cartpole.py

```python
import time
import random
from dataclasses import dataclass

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

import gymnasium as gym

from per.replay import GPUReplayPER


@dataclass
class Config:
    env_id: str = "CartPole-v1"
    seed: int = 0

    total_steps: int = 300_000
    warmup_steps: int = 10_000

    capacity: int = 100_000
    batch_size: int = 256
    gamma: float = 0.99

    lr: float = 1e-3
    train_freq: int = 1
    target_update_freq: int = 1000

    # epsilon-greedy
    eps_start: float = 1.0
    eps_end: float = 0.05
    eps_decay_steps: int = 100_000

    # PER
```

```
    alpha: float = 0.6
    beta0: float = 0.4
    beta_anneal_steps: int = 200_000


class QNet(nn.Module):
    def __init__(self, obs_dim, act_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, act_dim),
        )

    def forward(self, x):
        return self.net(x)


def linear_schedule(step, start, end, duration):
    t = min(step / duration, 1.0)
    return start + t * (end - start)


def main(cfg: Config):
    random.seed(cfg.seed)
    np.random.seed(cfg.seed)
    torch.manual_seed(cfg.seed)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    env = gym.make(cfg.env_id)
    obs, _ = env.reset(seed=cfg.seed)
    obs_dim = obs.shape[0]
    act_dim = env.action_space.n

    q = QNet(obs_dim, act_dim).to(device)
    q_targ = QNet(obs_dim, act_dim).to(device)
    q_targ.load_state_dict(q.state_dict())

    opt = torch.optim.Adam(q.parameters(), lr=cfg.lr)

    buf = GPUReplayPER(
        capacity=cfg.capacity,
        obs_shape=(obs_dim,),
        alpha=cfg.alpha,
        beta0=cfg.beta0,
        beta_anneal_steps=cfg.beta_anneal_steps,
        device=device,
    )

    ep_ret = 0.0
    ep_len = 0
    t0 = time.time()

    for step in range(cfg.total_steps):
        eps = linear_schedule(step, cfg.eps_start, cfg.eps_end, cfg.
            eps_decay_steps)
        if random.random() < eps:
```

```
            act = env.action_space.sample()
        else:
            with torch.no_grad():
                o = torch.tensor(obs, dtype=torch.float32, device=device).unsqueeze(0)
                act = torch.argmax(q(o), dim=1).item()

        next_obs, rew, terminated, truncated, _ = env.step(act)
        done = terminated or truncated

        buf.add(
            torch.tensor(obs, dtype=torch.float32),
            act,
            float(rew),
            torch.tensor(next_obs, dtype=torch.float32),
            done,
        )

        obs = next_obs
        ep_ret += rew
        ep_len += 1

        if done:
            obs, _ = env.reset()
            ep_ret = 0.0
            ep_len = 0

        if step < cfg.warmup_steps:
            continue

        if step % cfg.train_freq == 0:
            batch = buf.sample(batch_size=cfg.batch_size, step=step)

            with torch.no_grad():
                q_next = q_targ(batch.next_obs).max(dim=1).values
                target = batch.rew + cfg.gamma * (1.0 - batch.done.float()) * q_next

            q_pred = q(batch.obs).gather(1, batch.act.long().unsqueeze(1)).squeeze(1)
            td = target - q_pred

            loss = (batch.weights * td.pow(2)).mean()

            opt.zero_grad()
            loss.backward()
            opt.step()

            buf.update_priorities(batch.indices, td.detach())

        if step % cfg.target_update_freq == 0:
            q_targ.load_state_dict(q.state_dict())

    env.close()


if __name__ == "__main__":
    main(Config())
```

17 Tests: Correctness and Expected Results

17.1 How to run

Typical command:

- `pytest -q`

These tests are designed primarily for CUDA; if CUDA is unavailable they return early.

17.2 `test_update.py`: Tree invariants

Expected results. All assertions pass:

- Root equals sum of leaves after random updates (within tolerance).
- Random internal nodes satisfy `tree[i] = tree[2i] + tree[2i+1]`.

Listing 11: test_update.py

```
import torch
from per.tree import GpuSumTree

def test_sum_invariant_after_random_updates():
    if not torch.cuda.is_available():
        return

    cap = 1024
    t = GpuSumTree(cap)

    # random updates (with duplicates)
    n = 5000
    idx = torch.randint(0, cap, (n,), device="cuda", dtype=torch.int64)
    new_p = torch.rand(n, device="cuda", dtype=torch.float32)

    t.update(idx, new_p)

    leaves_sum = t.leaf_values().sum()
    root = t.total

    assert torch.allclose(root, leaves_sum, rtol=1e-4, atol=1e-5), (root.
       item(), leaves_sum.item())

def test_internal_node_invariant_spotcheck():
    if not torch.cuda.is_available():
        return

    cap = 256
    t = GpuSumTree(cap)

    # set all leaves explicitly (unique update)
    idx = torch.arange(cap, device="cuda", dtype=torch.int64)
    new_p = torch.rand(cap, device="cuda", dtype=torch.float32)
    t.update(idx, new_p)

    # spot check random internal nodes
    # valid internal indices: [1, L-1]
    L = t.L
    internal = torch.randint(1, L, (50,), device="cuda", dtype=torch.int64
       ).tolist()

```

```
    for i in internal:
        left = 2 * i
        right = left + 1
        # right child might be out of bounds only if i >= L, but we
            sampled < L
        s = t.tree[left] + t.tree[right]
        assert torch.allclose(t.tree[i], s, rtol=1e-4, atol=1e-5), f"node
            {i} mismatch"
```

17.3 test_sample.py: Sampling distribution

Expected results. The L1 error between the empirical histogram and the target distribution stays below a forgiving threshold (e.g. < 0.10). This is a statistical test, so the tolerance is intentionally loose to avoid flakiness.

Listing 12: test_sample.py

```
import torch
from per.tree import GpuSumTree

def test_sampling_distribution_reasonable():
    if not torch.cuda.is_available():
        return

    cap = 128
    t = GpuSumTree(cap)

    idx = torch.arange(cap, device="cuda", dtype=torch.int64)
    p = (idx.float() + 1.0)  # priorities 1..cap
    t.update(idx, p)

    # sample many times
    B = 200_000
    samp = t.sample(B)

    hist = torch.bincount(samp, minlength=cap).float()
    hist /= hist.sum()

    target = p / p.sum()
    l1 = torch.abs(hist - target).sum().item()

    # forgiving threshold to avoid flakiness
    assert l1 < 0.10, f"L1 error too large: {l1}"
```

17.4 test_per_replay.py: Replay sampling + updates

Expected results.

- Sampled batch shapes match the requested batch size and observation dimensions.
- Indices lie within $[0, \text{capacity})$.
- IS weights are positive and normalized to ≤ 1.
- Priority updates run without error.

Listing 13: test_per_replay.py

```
import torch
from per.replay import GPUReplayPER
```

```
3
4  def test_per_replay_sample_shapes():
5      if not torch.cuda.is_available():
6          return
7
8      buf = GPUReplayPER(capacity=1024, obs_shape=(4,))
9      # fill some transitions
10     for i in range(200):
11         obs = torch.randn(4)
12         next_obs = torch.randn(4)
13         buf.add(obs, i % 2, 1.0, next_obs, False)
14
15     batch = buf.sample(batch_size=64, step=0)
16
17     assert batch.obs.shape == (64, 4)
18     assert batch.next_obs.shape == (64, 4)
19     assert batch.act.shape == (64,)
20     assert batch.rew.shape == (64,)
21     assert batch.done.shape == (64,)
22     assert batch.weights.shape == (64,)
23     assert batch.indices.shape == (64,)
24     assert batch.indices.min().item() >= 0
25     assert batch.indices.max().item() < 1024
26     assert batch.weights.min().item() > 0.0
27     assert batch.weights.max().item() <= 1.0 + 1e-6
28
29 def test_update_priorities_runs():
30     if not torch.cuda.is_available():
31         return
32
33     buf = GPUReplayPER(capacity=256, obs_shape=(4,))
34     for i in range(100):
35         buf.add(torch.zeros(4), 0, 0.0, torch.zeros(4), False)
36
37     batch = buf.sample(batch_size=32, step=10)
38     td = torch.randn(32, device="cuda")
39     buf.update_priorities(batch.indices, td)  # should not crash
```

18 Note on `per_ext.py` Filename

The repository includes a file named `per_ext.py` whose contents are C++ code (not Python). This is likely an artifact or alternate binding stub and should be renamed (e.g., `per_ext_alt.cpp`) to avoid confusion. It is *not* used by `_ext.py`.

Listing 14: per_ext.py (contains C++ code; consider renaming)

```
1  #include <torch/extension.h>
2
3  void update_cuda(torch::Tensor tree, torch::Tensor idx, torch::Tensor
       new_p, int64_t L);
4  void update_opt_cuda(torch::Tensor tree, torch::Tensor idx, torch::Tensor
       new_p, int64_t L);
5
6  PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
7      m.def("update", &update_cuda, "Sum-tree update (CUDA)");
8  m.def("update_opt", &update_opt_cuda, "Sum-tree update optimized with CUB
       (CUDA)");
9  // keep sample binding too if you have it
10 }
```

19 Conclusion

This coursework implements a GPU-friendly PER sum-tree with:

- correctness invariants validated by tests,
- GPU kernels designed for safe concurrent updates (via atomics),
- a replay buffer that computes IS weights and supports TD-error priority updates,
- a baseline method for performance comparison.

The design aligns with the original PER paper and demonstrates practical GPU programming considerations (atomics, warp-level primitives, memory layout, and integration via PyTorch extensions).

References

[1] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. *Prioritized Experience Replay.* arXiv:1511.05952, 2015. DOI: 10.48550/arXiv.1511.05952.