

Contents

1 Chapter 6 — Advanced Lab: From CPU Loops to GPU Batches	1
1.1 1. Why This Lab Exists	2
1.2 2. The Starting Point — Canonical CPU Implementation	2
1.2.1 2.1. Warm-Up: Run the CPU Batch Once	3
1.3 3. Why the CPU Version Becomes Painful	4
1.4 4. GPU Mental Model: From Loops to Batches	5
1.4.1 4.1. CPU vs GPU in One Sentence	5
1.4.2 4.2. Arrays with a Passport: Tensors and Devices	5
1.4.3 4.3. From For-Loops to Batches	6
1.5 5. A Guided Tour of the GPU Implementation	6
1.5.1 5.1. Building the GPU World	6
1.5.2 5.2. Sampling Users and Queries in Batches	7
1.5.3 5.3. Computing Relevance on the GPU	7
1.5.4 5.4. Simulating Sessions in Parallel	8
1.5.5 5.5. Safety Guard and Reward Parity	8
1.5.6 5.6. Interactive Bandit Policies on GPU Batches	9
1.6 6. Running the GPU Compute Arc	9
1.6.1 6.1. Basic Run on CPU (No GPU Required)	10
1.6.2 6.2. Switching to GPU	11
1.6.3 6.3. Generated Artifacts and Plots	11
1.7 7. GPU Batch Runner: Full Scenario Grid	12
1.7.1 7.1. Running the GPU Batch Runner	12
1.7.2 7.2. When Does GPU Actually Help?	13
1.8 8. Best Practices for Safe CPU to GPU Migration	13
1.8.1 8.1. Always Start on CPU	13
1.8.2 8.2. Keep Bandit Logic on the CPU (for Now)	13
1.8.3 8.3. Avoid Excessive <code>.cpu()</code> / <code>.numpy()</code> Conversions	14
1.8.4 8.4. Seed Management: Align NumPy and Torch	14
1.8.5 8.5. Trust Parity Logs and Guardrails	14
1.9 9. Hands-On Tasks	15
1.9.1 Task 9.1 — CPU vs GPU Time Comparison	15
1.9.2 Task 9.2 — Batch Size Trade-Off	16
1.9.3 Task 9.3 — Feature Mode Sanity Check	16
1.9.4 Task 9.4 — Extend a Diagnostic	17
1.10 10. Reflection: When to Reach for the GPU	17

1 Chapter 6 — Advanced Lab: From CPU Loops to GPU Batches

Author: Vlad Prytula

Time budget: 2–3 hours (spread over a few sessions if needed)

Prerequisites: Comfortable with NumPy and basic Python; no prior PyTorch or GPU experience assumed.

1.1 1. Why This Lab Exists

In the core Chapter 6 narrative, you met the **canonical implementation** of template bandits:

- Static templates vs LinUCB vs Thompson Sampling
- Simple vs rich context features
- Honest failure with impoverished features, recovery with richer signals

All of that was implemented in a **clean but sequential** NumPy style:

- `scripts/ch06/template_bandits_demo.py`
- `scripts/ch06/run_bandit_matrix.py`

These scripts are perfect for:

- Reading end-to-end logic
- Instrumenting with print statements
- Connecting equations to code line-by-line

They are **not** perfect when you want to:

- Run many seeds or large grids of scenarios
- Go from 20k episodes to 200k or 2M
- Iterate quickly on feature variants or hyperparameters

At some point, the canonical implementation becomes what you feel as a **compute bottleneck**: the code is mathematically right, but your iteration speed collapses.

This lab is your guided path from:

- “I can read and run the Chapter 6 CPU code”
to
- “I can confidently use a GPU-accelerated version, understand what changed, and know when it is safe to trust it.”

We will walk from **zero GPU knowledge** to:

- Understanding the mental model of GPU batches
- Reading the main GPU implementation
- Running GPU-accelerated experiments
- Applying best practices to avoid subtle bugs

You do not need to be a “PyTorch person” to finish this lab. Think of the GPU as a slightly stricter NumPy: arrays live on a device, and you try very hard not to move them around unnecessarily.

1.2 2. The Starting Point — Canonical CPU Implementation

Before touching GPUs, make sure you are comfortable with the **canonical CPU path**. We will treat it as ground truth.

The key scripts live under:

- `scripts/ch06/template_bandits_demo.py`

- `scripts/ch06/run_bandit_matrix.py`

The core experiment is implemented by:

- `run_template_bandits_experiment` in
`scripts/ch06/template_bandits_demo.py`

It:

1. Generates a simulator world from `SimulatorConfig`
2. Evaluates each template as a static policy
3. Runs LinUCB and Thompson Sampling over `n_bandit` episodes
4. Logs aggregate metrics (Reward, GMV, CM2, Orders) and per-segment results

The batch runner:

- `scripts/ch06/run_bandit_matrix.py`

wraps this core experiment into a small **scenario grid**:

1. Simple features vs rich features vs rich_est
2. Optional rich-regularization modes
3. A few world and bandit seeds

It then:

- Runs scenarios sequentially or in parallel threads
- Captures stdout into a JSON artifact
- Writes results under `docs/book/drafts/ch06/data/`

1.2.1 2.1. Warm-Up: Run the CPU Batch Once

From your repository root, run:

```
python scripts/ch06/run_bandit_matrix.py \
--n-static 1000 \
--n-bandit 20000 \
--max-workers 4
```

Output (abridged, representative):

```
Planned scenarios:
[1/5] simple_baseline      features=simple      world_seed=20250701  bandit_seed=20250801  pri...
...
[1/5] Running scenario 'simple_baseline'...
...
[OK] Completed 'simple_baseline'
...
```

Saved batch results to `docs/book/drafts/ch06/data/bandit_matrix_20250701T120000Z.json`

You do not need to inspect every line yet. The takeaway:

- The CPU pipeline is working.

- You know how to launch the canonical experiment.
 - You have a **reference JSON artifact** that later GPU runs must agree with (within noise).
-

1.3 3. Why the CPU Version Becomes Painful

The CPU implementation is written the way we teach algorithms:

- Clear loops
- Direct calls to `sample_user`, `sample_query`, `compute_reward`
- Easy to print intermediate values

But numerically, it behaves like this:

- Each episode is a **separate Python loop iteration**
- Each episode walks through:
 - Sample user
 - Sample query
 - Compute base scores
 - Apply each template
 - Simulate user response
 - Update bandit

Imagine a single cashier scanning items one by one, printing receipts, and answering questions. Every action requires the cashier's attention.

Now imagine you want to:

- Increase episode count from 20k to 200k
- Sweep over many seeds and feature modes
- Compare multiple hyperparameter settings

The cashier analogy turns into a bottleneck:

- Even if each episode is cheap, **Python loop overhead + per-episode simulator calls add up.**
- CPU vectorization helps in places (NumPy inside each episode), but the outer loop remains serial.

What you want instead is an **assembly line**:

- The GPU carries out the same simple operation (e.g., dot products, sampling, elementwise activations) on many items in parallel.
- You pay some setup cost (batching, moving data to GPU), but then thousands of episodes are processed in one sweep.

This is exactly what the GPU implementation under:

- `scripts/ch06/optimization_gpu/`

is designed to do.

1.4 4. GPU Mental Model: From Loops to Batches

If you are new to GPUs, the mental model can feel intimidating. Let's strip it down to something concrete.

1.4.1 4.1. CPU vs GPU in One Sentence

- **CPU:** a few powerful cores; great at complex control flow; mediocre at doing the same tiny operation millions of times.
- **GPU:** thousands of lightweight cores; terrible at complex branching; excellent at doing the same tiny operation on large arrays.

In Chapter 6, almost all of our heavy work is:

- Matrix multiplications, dot products, softmax and top-k, random draws, simple elementwise arithmetic

These map perfectly to the GPU.

1.4.2 4.2. Arrays with a Passport: Tensors and Devices

The only new concepts you need from PyTorch are:

- `torch.Tensor`: like a NumPy array, but can live on CPU or GPU.
- `device`: where the tensor lives ("cpu", "cuda", "mps", ...).

In `scripts/ch06/optimization_gpu/template_bandits_gpu.py`, this is wrapped by:

- `_as_device(device)`
`(scripts/ch06/optimization_gpu/template_bandits_gpu.py:132-164)`

which chooses:

- CUDA if available
- Otherwise Apple MPS if available
- Otherwise CPU

You can think of it as:

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

but with a few extra safety checks.

Once a tensor is on a device, you want to:

- Keep all subsequent operations on the same device
- Avoid moving data back-and-forth (`.cpu().numpy()`) inside tight loops

The GPU code in this repo is written so that:

- Inputs are moved to the GPU once per batch.
- All simulation steps stay in GPU memory.
- Only aggregated results are brought back to NumPy at the edges.

1.4.3 4.3. From For-Loops to Batches

Conceptually:

- CPU version:
 - Loop over `episode_idx` from 1 to `n_episodes`
 - Within each loop, sample a single user, single query, single ranking outcome
- GPU version:
 - Choose a batch size B (e.g. 1024)
 - Sample B users and B queries **at once**
 - Compute relevance and template boosts for all B episodes simultaneously
 - Simulate user behavior for all B episodes on the GPU

The bandit algorithms themselves (LinUCB and Thompson Sampling) remain **sequential in episode index**:

- We still update the posterior after each selected action.
- We still reveal only the chosen action's reward to the policy.

The GPU's job is not to change the algorithm; it is to **amortize simulator work** across many episodes.

1.5 5. A Guided Tour of the GPU Implementation

The GPU implementation lives primarily in:

- `scripts/ch06/optimization_gpu/template_bandits_gpu.py`

and is orchestrated by:

- `scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py`
- `scripts/ch06/optimization_gpu/run_bandit_matrix_gpu.py`

We will follow the same structure as the CPU code:

1. Build the world and templates
2. Precompute GPU-friendly representations
3. Simulate static templates
4. Run bandit policies interactively

1.5.1 5.1. Building the GPU World

The CPU version asks the simulator for:

- Catalog, users, queries, behavior, reward config

The GPU version does the same, but then **packs everything into tensors** in a `GPUWorld` data-class:

- See `_prepare_world` in
`scripts/ch06/optimization_gpu/template_bandits_gpu.py:260-319`

Key pieces:

- `product_prices`, `product_cm2`, `product_discount`, `product_is_pl`
- `product_embeddings`, `normalized_embeddings`
- `template_boosts` (boost values per template x product)
- `lexical_matrix` and `pos_bias` for relevance
- `segment_params` (price and PL distributions per segment)

You can think of `GPUWorld` as:

- “All the catalog and behavior data, but now as dense tensors on a device.”

Analogy:

- The CPU version keeps shelf labels, prices, and product attributes in many small Python objects.
- The GPU version moves them onto a big **warehouse whiteboard** that all workers can see in one glance.

Once `GPUWorld` is constructed, all costly operations (matrix multiplications, top-k, sampling) happen inside these tensors.

!!! note “Code – Simulator (GPU world construction)” - KG IDs: CH-6, DOC-ch06-template_bandits_demo
- Files: - `scripts/ch06/template_bandits_demo.py`:300-420 (CPU static templates and bandit runs) - `scripts/ch06/optimization_gpu/template_bandits_gpu.py`:260-319 (GPU `GPUWorld` construction)

1.5.2 5.2. Sampling Users and Queries in Batches

The sampler helpers in the GPU implementation correspond closely to the CPU’s per-episode sampling:

- `_sample_segments` chooses user segments according to `cfg.users.segment_mix`
- `_sample_theta` samples continuous user preferences for price and PL, plus a categorical preference vector
- `_sample_theta_emb` samples user embedding noise for semantic relevance

These live in:

- `scripts/ch06/optimization_gpu/template_bandits_gpu.py`:320-409

The important difference:

- All of them work on **whole batches** (`batch_size` episodes) at a time.
- They return tensors of shape (`batch_size`, ...) instead of scalars.

You can imagine `batch_size=1024` as:

- Drawing 1024 users
- Drawing 1024 queries
- Generating 1024 episodes worth of latent preferences

in one go, using vectorized math and GPU random generators.

1.5.3 5.3. Computing Relevance on the GPU

The base relevance scores (before templates) are computed by:

- `_compute_base_scores` in
`scripts/ch06/optimization_gpu/template_bandits_gpu.py:620-639`

This mirrors the Chapter 5 relevance model:

- Cosine similarity between query and product embeddings
- Lexical overlap via a precomputed matrix
- Gaussian noise

All operations are **tensor operations** on the chosen device:

- `normalized_queries @ world.normalized_embeddings.T`
- `world.lexical_matrix[query_intent_idx]`
- `torch.randn(...)` with a `torch.Generator`

The output is a tensor:

- Shape: `(batch_size, num_products)`
- Semantics: base score for each episode x product pair

1.5.4 5.4. Simulating Sessions in Parallel

The heart of the GPU acceleration is:

- `_simulate_sessions` in
`scripts/ch06/optimization_gpu/template_bandits_gpu.py:640-736`

This function:

1. Adds template boosts to base scores for every template and episode at once.
2. Applies `top_k` to get rankings for each template and episode.
3. Steps through positions `pos = 0, ..., top_k-1` in a vectorized loop:
 - Computes utilities from price sensitivity, PL preferences, category affinity, and semantic match.
 - Samples clicks and purchases via logistic models and behavioral noise.
 - Updates satisfaction and purchase counts.
4. Aggregates reward components (GMV, CM2, Orders, clicks) into a `TemplateBatchMetrics` tensor bundle.

Conceptually, you are running:

- “all templates x all episodes”

simultaneously, but with careful use of tensor shapes so the GPU can execute it efficiently.

1.5.5 5.5. Safety Guard and Reward Parity

A crucial design principle in this project is:

- “If the theorem doesn’t compile, it’s not ready.”
 Here: “If the GPU rewards don’t match the CPU rewards, we do not trust the speedup.”

The GPU implementation therefore includes an explicit **safety guard**:

- `_validate_reward_sample` in
`scripts/ch06/optimization_gpu/template_bandits_gpu.py:792-821`

On each batch, it:

1. Picks a random episode + template from the tensor metrics.
2. Reconstructs a concrete (`ranking`, `clicks`, `buys`) trace.
3. Calls the canonical `reward.compute_reward` from Chapter 5.
4. Asserts that the scalar reward matches the tensor reward (within tight tolerance).
5. Ensures the Chapter 5 click-weight guard remains enforced.

If there is any mismatch, the GPU code raises an error instead of silently drifting.

!!! note “Code – Reward Safety” - KG IDs: EQ-5.7, CH-6 - Files: - `scripts/ch06/optimization_gpu/template_batched.py`:1-40 (GPU reward validation) - `scripts/ch06/optimization_gpu/PARITY_FINDINGS.md`:1-40 (parity log, reward safety guard)

This is the GPU analogue of a **unit test baked into the simulation loop**.

1.5.6 5.6. Interactive Bandit Policies on GPU Batches

The last piece is to connect the batched simulator to the bandit policies:

- `_run_policy_interactive` in
`scripts/ch06/optimization_gpu/template_bandits_gpu.py`:1120-1184

The control flow is:

1. While `produced < n_episodes`:
 - Simulate a batch of episodes and extract features + per-template metrics.
 - For each episode in the batch (loop in Python):
 - Convert features to NumPy for the policy (LinUCB or TS).
 - Select an action (template index).
 - Record the chosen reward and update the policy.
2. Aggregate per-episode and per-segment results.
3. Collect diagnostics (template selection counts and frequencies).

Notice the hybrid:

- **Outer logic**: still sequential per episode (to respect bandit semantics).
- **Inner simulator**: fully batched on the GPU.

This preserves the correctness of the learning algorithm while amortizing the heavy work.

!!! note “Code – Agent (GPU bandits)” - KG IDs: `MOD-zoosim.policies.lin_ucb`, `MOD-zoosim.policies.thompson_sampling` - Files: - `scripts/ch06/template_bandits_demo.py`:620-1120 (CPU bandit loop) - `scripts/ch06/optimization_gpu/template_bandits_gpu.py`:1120-1184 (GPU interactive loop)

1.6 6. Running the GPU Compute Arc

Now that you have a conceptual map, let’s run the GPU-accelerated version of the core Chapter 6 compute arc:

- Simple features (failure)
- Rich features (recovery)

The orchestrator is:

- `scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py`

1.6.1 6.1. Basic Run on CPU (No GPU Required)

Even if you do not have a GPU, you can run the GPU implementation in **CPU mode**. This is the safest first step.

Run:

```
python scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py \
--n-static 2000 \
--n-bandit 20000 \
--device cpu \
--batch-size 1024
```

Output (abridged, representative):

CHAPTER 6 --- GPU COMPUTE ARC: SIMPLE -> RICH FEATURES

Static episodes: 2,000
Bandit episodes: 20,000
Base seed: 20250601
Batch size: 1024
Device: cpu

Experiment 1: Simple features (failure mode) --- features=simple, ...

RESULTS (Simple Features):

Static (best): GMV = 123.45
LinUCB: GMV = 88.90 (-28.0%)
TS: GMV = 95.12 (-23.0%)

Experiment 2: Rich features (oracle) --- features=rich, ...

RESULTS (Rich Features):

Static (best): GMV = 123.45 (shared world)
LinUCB: GMV = 126.89 (+2.8%)
TS: GMV = 157.00 (+27.2%)

IMPROVEMENT (Rich vs Simple):

LinUCB: +38.0 GMV (+42.8%)
TS: +61.9 GMV (+65.0%)

You should recognize:

- The failure with `features=simple` (bandits underperform static templates).
- The recovery with `features=rich` (Chapter 6's +3% / +27% GMV story).

Behind the scenes, all the heavy lifting is already happening on batched tensors via PyTorch. You are just running it on CPU.

1.6.2 6.2. Switching to GPU

If you have a GPU available, the lab is now a one-flag change:

```
python scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py \
--n-static 2000 \
--n-bandit 20000 \
--device cuda \
--batch-size 1024
```

or simply:

```
python scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py \
--n-static 2000 \
--n-bandit 20000 \
--device auto \
--batch-size 1024
```

Internally:

- `device="auto"` selects "cuda" if `torch.cuda.is_available()` is `True`, otherwise "`mps`" or "`cpu`".
- All tensors are created directly on that device.

What you should check:

- The **GMV numbers** match the CPU `device=cpu` run within minor stochastic variation.
- The wall-clock runtime for:
 - `n_bandit = 20_000` is similar or slightly faster.
 - Larger `n_bandit` (e.g. 100k, 200k) becomes noticeably faster on GPU.

1.6.3 6.3. Generated Artifacts and Plots

The script saves:

- JSON summaries:
 - `docs/book/drafts/ch06/data/template_bandits_simple_gpu_summary.json`
 - `docs/book/drafts/ch06/data/template_bandits_rich_gpu_summary.json`
- Figures:
 - Segment GMV comparison
 - Template selection frequencies for simple and rich features

The last part of the script dynamically imports:

- `scripts/ch06/plot_results.py`

to reuse the CPU plotting logic.

!!! note “Code – Env (compute arc CLI)” - KG IDs: DOC-ch06-compute-arc - Files: - `scripts/ch06/ch06_compute_arc.py:1-120` (CPU compute arc) - `scripts/ch06/optimization_gpu/ch06_com` (GPU compute arc)

1.7 7. GPU Batch Runner: Full Scenario Grid

The compute arc focuses on **two experiments** (simple vs rich) on a single world. When you want to sweep multiple scenarios, use:

- `scripts/ch06/optimization_gpu/run_bandit_matrix_gpu.py`

This is the GPU counterpart of:

- `scripts/ch06/run_bandit_matrix.py`

1.7.1 7.1. Running the GPU Batch Runner

Example:

```
python scripts/ch06/optimization_gpu/run_bandit_matrix_gpu.py \
--n-static 1000 \
--n-bandit 20000 \
--batch-size 1024 \
--device auto \
--max-workers 1 \
--show-volume
```

Output (abridged, representative):

CHAPTER 6 --- GPU BATCH RUNNER

```
[1/5] Running scenario 'simple_baseline' on auto...
Scenario Summary --- simple_baseline
```

```
-----
Static templates (per-episode averages):
...
```

```
Summary (per-episode averages):
Policy          Reward      GMV      CM2    Orders  DeltaGMV vs static
Static-...       123.45    123.45    80.00   0.80    +0.00%
LinUCB          100.01    88.90    70.00   0.70    -28.00%
ThompsonSampling 110.11   95.12    72.00   0.72    -23.00%
```

```
...
```

```
Saved batch results to docs/book/drafts/ch06/data/bandit_matrix_gpu_20250701T121000Z.json
```

Differences from the CPU batch runner:

- Additional flags:
 - `--batch-size` controls the simulation batch size on GPU.
 - `--device` chooses "cpu", "cuda", "mps", or "auto".
 - `--show-volume` adds Orders columns to tables.
- Default `--max-workers` is 1 (conservative), because:
 - Running multiple GPU-heavy jobs in parallel can cause memory pressure.
 - If you do increase it, keep an eye on GPU memory usage.

1.7.2 7.2. When Does GPU Actually Help?

For small workloads (e.g. 2k episodes), a GPU may not be faster:

- There is overhead for:
 - Creating GPU tensors
 - Transferring any input data
 - Initializing CUDA context

The GPU shines when:

- `n_bandit` is large (e.g. 50k–500k episodes)
- `batch_size` is tuned to fill the GPU reasonably well

Analogy:

- Calling the GPU for 100 episodes is like renting a 100-seat bus to transport 3 people. The bus moves fast, but boarding takes time and you are not using its capacity.
 - Calling the GPU for 100k episodes with `batch_size=4096` is like running a shuttle service at full capacity. You amortize the boarding cost and benefit from the bus's speed.
-

1.8 8. Best Practices for Safe CPU to GPU Migration

This lab is not just about “making it faster.” It is about making sure the GPU version is:

- Correct (matches the canonical implementation)
- Reproducible (same seeds, same answers)
- Understandable (you can debug it when needed)

Here are the core practices I want you to internalize.

1.8.1 8.1. Always Start on CPU

Even when your goal is GPU acceleration:

1. Run the GPU code with `--device cpu`.
2. Compare outputs against the canonical CPU scripts:
 - `template_bandits_demo.py`
 - `run_bandit_matrix.py`
3. Only when you see parity in metrics do you switch `--device` to `cuda` or `auto`.

This avoids a common failure mode:

- You change both the **algorithm** and the **hardware** at once, and when things diverge, you have no idea which change is responsible.

1.8.2 8.2. Keep Bandit Logic on the CPU (for Now)

Notice that:

- The GPU implementation still runs LinUCB and Thompson Sampling in Python/NumPy.
- Only the heavy simulation pieces live on the GPU.

This is intentional:

- The bandit algorithms are not the bottleneck; the simulator is.
- Keeping policies on CPU makes them easier to inspect, instrument, and test.

In more advanced projects, you might move some policy computation to the GPU (e.g. neural networks). For this chapter, we keep that complexity out of the picture.

1.8.3 8.3. Avoid Excessive `.cpu()` / `.numpy()` Conversions

As a rule of thumb:

- Inside the simulator:
 - Stay in `torch.Tensor` on the chosen device.
- At the edges:
 - Convert to NumPy only when passing data into bandit logic or returning final summaries.

Excessive back-and-forth conversion has two risks:

1. Performance: each conversion can involve synchronization and memory movement.
2. Bugs: it is easy to accidentally detach from the correct batch, mix up shapes, or forget to move a tensor back.

The existing GPU code is written to minimize these conversions; use it as a template for your own extensions.

1.8.4 8.4. Seed Management: Align NumPy and Torch

Look at how seeds are handled in:

- CPU implementation:
 - `scripts/ch06/template_bandits_demo.py:560-620`
- GPU implementation:
 - `scripts/ch06/optimization_gpu/template_bandits_gpu.py:1240-1274`

Both use the same pattern:

- Static templates: `base_seed`
- LinUCB: `base_seed + 1`
- Thompson Sampling: `base_seed + 2`

On GPU, this is done with:

- `torch.Generator(device=device_obj).manual_seed(base_seed + k)`
- `np.random.default_rng(base_seed + k)`

This alignment is critical:

- It allows you to run CPU and GPU versions with the **same base seeds** and expect comparable trajectories.
- Differences then reflect only the small numerical differences (e.g. float32 vs float64), not entirely different random histories.

1.8.5 8.5. Trust Parity Logs and Guardrails

The file:

- `scripts/ch06/optimization_gpu/PARITY_FINDINGS.md` documents historical parity mismatches and their fixes:

- Reward safety guard alignment with Chapter 5
- Seed alignment between CPU and GPU

Treat this as:

- A living changelog connecting theory to implementation.
- A reminder that performance optimizations must not silently change semantics.

If you ever extend the GPU implementation:

- Add new findings there.
 - Run small CPU vs GPU comparisons to ensure you have not regressed.
-

1.9 9. Hands-On Tasks

This section turns the narrative into concrete steps. Each task builds confidence that you can **use** and **extend** the GPU implementation without being a PyTorch expert.

1.9.1 Task 9.1 — CPU vs GPU Time Comparison

Goal:

- Measure how runtime scales with `n_bandit` on CPU vs GPU.

Steps:

1. Pick a fixed `n_static` (e.g. 2000).
2. Sweep `n_bandit` over {20_000, 50_000, 100_000}.
3. For each setting, run:

```
# CPU
time python scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py \
--n-static 2000 \
--n-bandit N \
--device cpu \
--batch-size 1024

# GPU (if available)
time python scripts/ch06/optimization_gpu/ch06_compute_arc_gpu.py \
--n-static 2000 \
--n-bandit N \
--device cuda \
--batch-size 1024
```

4. Record:
 - Wall-clock time
 - GMV results (to verify parity)

Output (example summary table you might build):

N	CPU time	GPU time	LinUCB GMV (CPU/GPU)	TS GMV (CPU/GPU)
20k	0m45s	0m40s	126.9 / 127.0	157.0 / 156.9
50k	2m00s	1m05s	127.1 / 127.2	156.8 / 156.8
100k	4m10s	1m50s	127.0 / 127.1	157.1 / 157.1

The exact numbers will differ, but you should see:

- Near-identical GMV values across CPU and GPU.
- Increasing relative speedup of GPU as N grows.

1.9.2 Task 9.2 — Batch Size Trade-Off

Goal:

- Understand how `batch_size` affects performance and memory usage.

Steps:

1. Fix `n_static=2000, n_bandit=50_000, device=cuda` (or `cpu` if no GPU).
2. Try:
 - `batch-size = 256`
 - `batch-size = 1024`
 - `batch-size = 4096`
3. Measure runtime and watch memory usage (on GPU, `nvidia-smi` can help).

Interpretation:

- Small batch sizes underutilize the GPU (more Python overhead, less parallelism).
- Very large batch sizes may:
 - Run out of memory
 - Increase variance in per-batch timing

You are looking for a **sweet spot** where:

- Runtime is near-minimal
- Memory usage is stable

1.9.3 Task 9.3 — Feature Mode Sanity Check

Goal:

- Verify that the “simple vs rich” narrative still holds under GPU acceleration.

Steps:

1. Run `ch06_compute_arc_gpu.py` with:
 - `--features` is controlled internally by the script (simple vs rich).
2. Compare:
 - Simple features GMV gap vs static.
 - Rich features GMV improvement vs static.
3. Compare CPU (`--device cpu`) vs GPU (`--device cuda`) runs.

Expected qualitative result:

- Simple features:
 - LinUCB and TS **underperform** static templates.
- Rich features:
 - TS achieves **+20–30% GMV** vs static in this simulator configuration.

If this narrative flips, treat it as a red flag and investigate:

- Configuration mismatches
- Seed misalignment
- Code changes that alter the reward definition

1.9.4 Task 9.4 — Extend a Diagnostic

Goal:

- Add a small, safe diagnostic to the GPU implementation and verify that it behaves as expected.

Ideas (pick one):

- Log the average fraction of PL products in the top-k under each template.
- Track the fraction of episodes where the bandit chooses the static best template.
- Count how often each user segment sees an improvement vs static.

Implementation sketch:

1. Identify where the relevant tensors live (`TemplateBatchMetrics`, segment indices, etc.).
2. Compute a simple summary statistic per batch.
3. Aggregate into a Python list or running average.
4. Print it at the end of the run (or add it to the JSON artifact).

The intention is not to redesign the algorithm, but to:

- Practice reading and modifying `template_bandits_gpu.py`.
- Build confidence that you can extend a GPU implementation safely.

1.10 10. Reflection: When to Reach for the GPU

After this lab, you should be able to answer:

1. **Why** did we move from the canonical CPU implementation to a GPU-accelerated one?
 - Not because CPUs are “bad,” but because experiments need to scale.
2. **What** changed in the GPU version?
 - Outer loops remain semantically the same.
 - Inner simulator steps are batched and moved to tensors on a device.
3. **How** do we know we did not break the math?
 - Reward parity checks.
 - Seed alignment.
 - Small-N CPU vs GPU comparisons.

The deeper lesson is not “use GPUs.” It is:

- **Separate correctness from performance.**

First, write a canonical implementation that you can reason about line-by-line.

Then, and only then, introduce acceleration—while constantly checking that you have not changed the underlying problem.

When you later move to:

- Chapter 7 (continuous actions with neural networks)
- Chapter 12 (differentiable ranking)
- Chapter 13 (offline RL with deep value functions)

this habit will protect you from many subtle, expensive bugs.

You now have your first serious experience with:

- Porting a nontrivial RL experiment from CPU loops to GPU batches.
- Maintaining theoretical guarantees and simulator semantics across implementations.

That is the level of care I expect from you as a practicing RL engineer.

Chapter 6 — Advanced Lab Draft — 2025