Nixie

Efficient, Transparent Temporal Multiplexing for Consumer GPUs

Yechen Xu, Yifei Wang, Nathanael Ren, Yiran Chen, Danyang Zhuo

Duke
UNIVERSITY

Local ML Is Exploding

Users now run several independent ML apps on one desktop-class GPU

LLM chat
assistants

Coding agents
autonomous loops

Image gen
diffusion

Video gen
generative

llama.cpp

opencode

Comfy

FAST VIDEO

SGL

OpenClaw

OneTrainer

Ostris

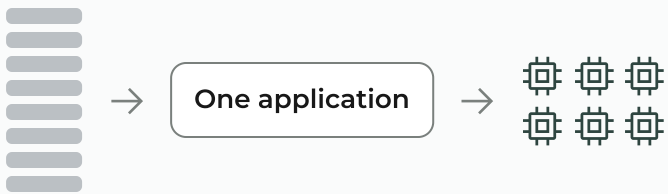
Why local?

- Privacy
- Cost
- Offline
- Control

Workloads: Consumer≠ Datacenter

Datacenter-serving assumptions cannot be applied to local ML workloads.

Datacenter serving

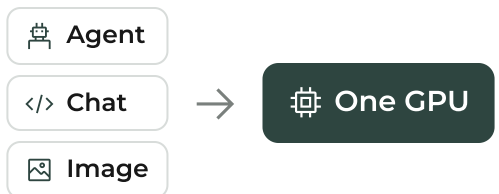


Managed service
Owned codebase & modified freely

Batched requests for one model
Optimized for throughput

Plentiful GPU & host memory
Flexible for load-balancing/offloading

Consumer local ML



Apps from many upstreams
Cannot modify each individually → transparency required

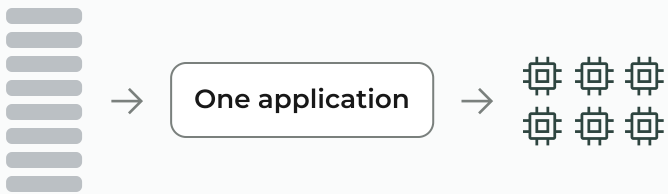
Single user, various models
Cannot efficiently batch requests

One GPU & Limited memory
More challenging for offloading

Workloads: Consumer≠ Datacenter

Datacenter-serving assumptions cannot be applied to local ML workloads.

Datacenter serving

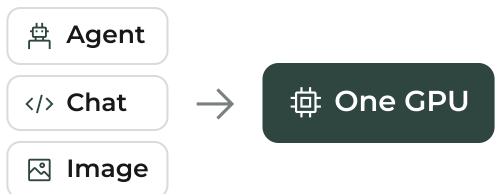


Managed service
Owned codebase & modified freely

Batched requests for one model
Optimized for throughput

Plentiful GPU & host memory
Flexible for load-balancing/offloading

Consumer local ML



Apps from many upstreams
Cannot modify each individually → transparency required

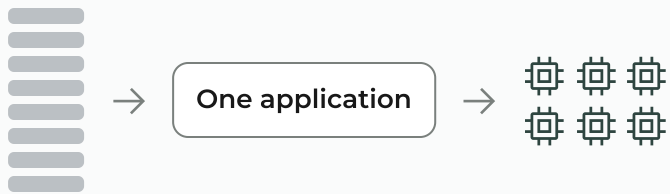
Single user, various models
Cannot efficiently batch requests

One GPU & Limited memory
More challenging for offloading

Workloads: Consumer≠ Datacenter

Datacenter-serving assumptions cannot be applied to local ML workloads.

Datacenter serving

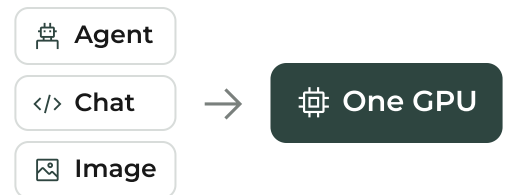


Managed service
Owned codebase & modified freely

Batched requests for one model
Optimized for throughput

Plentiful GPU & host memory
Flexible for load-balancing/offloading

Consumer local ML



Apps from many upstreams
Cannot modify each individually → transparency required

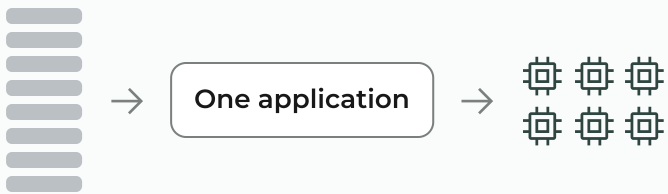
Single user, various models
Cannot efficiently batch requests

One GPU & Limited memory
More challenging for offloading

Workloads: Consumer≠ Datacenter

Datacenter-serving assumptions cannot be applied to local ML workloads.

Datacenter serving

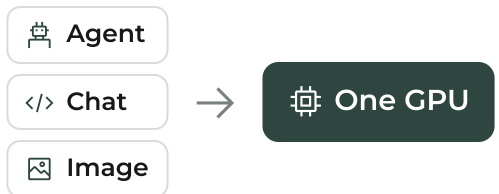


Managed service
Owned codebase & modified freely

Batched requests for one model
Optimized for throughput

Plentiful GPU & host memory
Flexible for load-balancing/offloading

Consumer local ML



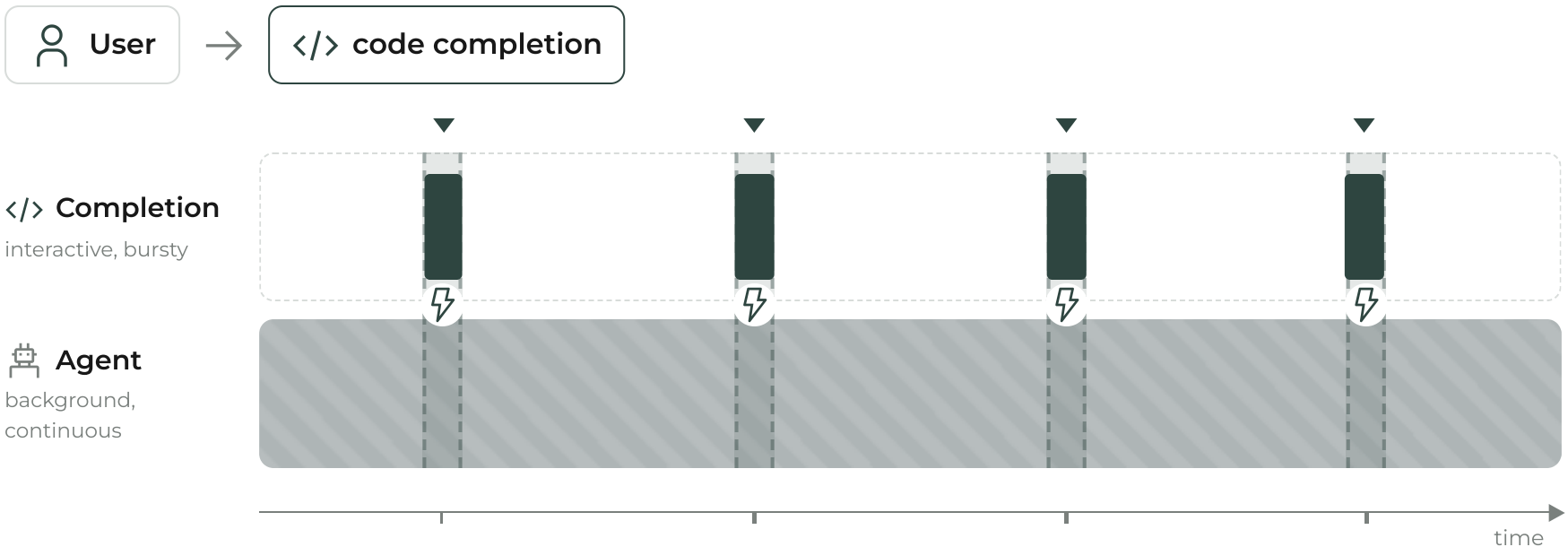
Apps from many upstreams
Cannot modify each individually → transparency required

Single user, various models
Cannot efficiently batch requests

One GPU & Limited memory
More challenging for offloading

Example: Agent + Code Completion

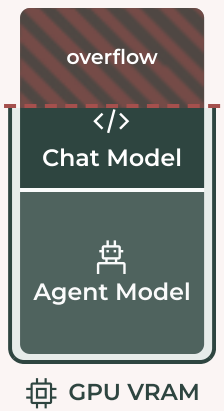
One background agent, repeated interactive requests – one GPU



Challenge: Models Compete for Memory

Each model fits alone – together they overflow, on the GPU *and* in host DRAM

GPU capacity pressure



- ✓ Each model fits **alone**
- ✗ Stacked, they **spill past capacity**

dashed = GPU capacity limit

Host DRAM pressure

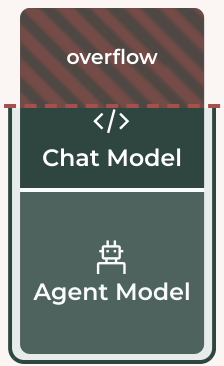


Shared with other apps -- and the pinned spill **can't be reclaimed**

Challenge: Models Compete for Memory

Each model fits alone – together they overflow, on the GPU *and* in host DRAM

GPU capacity pressure



- ✓ Each model fits **alone**
- ✗ Stacked, they **spill past capacity**

dashed = GPU capacity limit

Host DRAM pressure

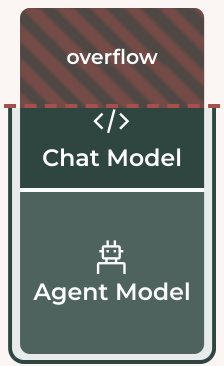


Shared with other apps -- and the pinned spill **can't be reclaimed**

Challenge: Models Compete for Memory

Each model fits alone – together they overflow, on the GPU *and* in host DRAM

GPU capacity pressure



- ✓ Each model fits **alone**
- ✗ Stacked, they **spill past capacity**

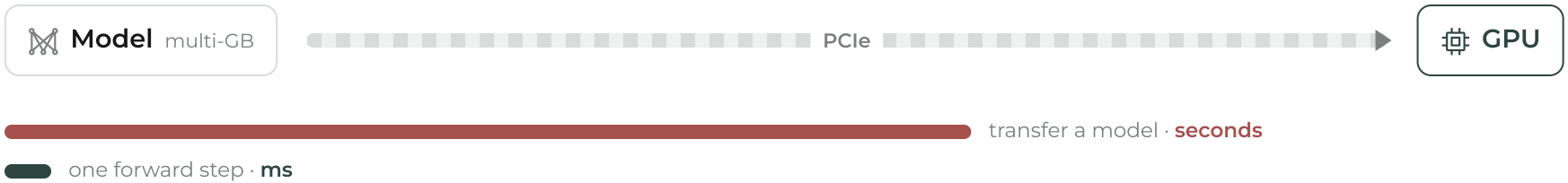
dashed = GPU capacity limit

Host DRAM pressure



Shared with other apps -- and the pinned spill **can't be reclaimed**

Challenge: Efficient App-Level Switching



Restart-based

App (warm)
- weights
- caches
- network states

kill

App (cold)
- ~~rebuilt~~
- ~~from~~
- ~~zero~~

Full app state lost on every switch

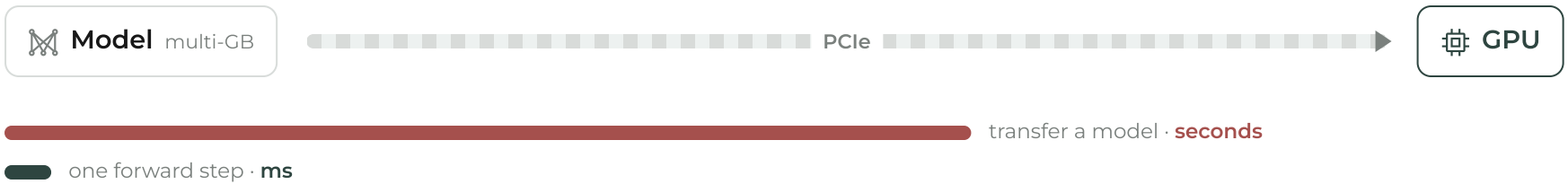
Uncoordinated · fault-based UVM

NVIDIA Unified Memory (UVM): transparent GPU↔host paging, moved **reactively on faults**



No coordination -- each swap **» its run window → thrashing**

Challenge: Efficient App-Level Switching



Restart-based

App (warm)
- weights
- caches
- network states

kill

App (cold)
- ~~rebuilt~~
- ~~from~~
- ~~zero~~

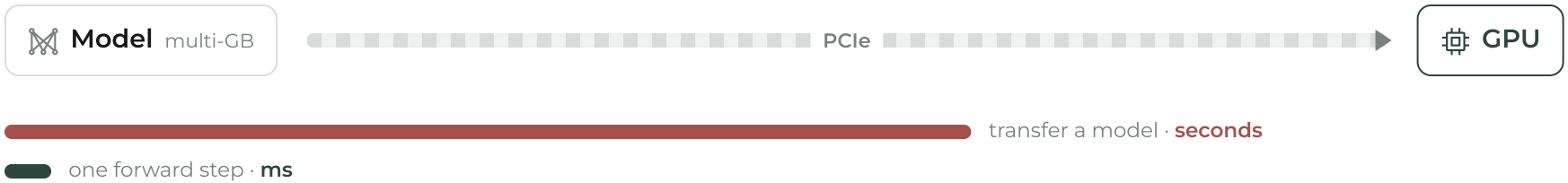
Full app state lost on every switch

Uncoordinated · fault-based UVM

NVIDIA Unified Memory (UVM): transparent GPU↔host paging, moved **reactively on faults**

No coordination -- each swap » **its run window → thrashing**

Challenge: Efficient App-Level Switching



Restart-based

App (warm)
weights
caches
network states

kill

App (cold)
~~rebuilt~~
~~from~~
~~zero~~

Full app state lost on every switch

Uncoordinated · fault-based UVM

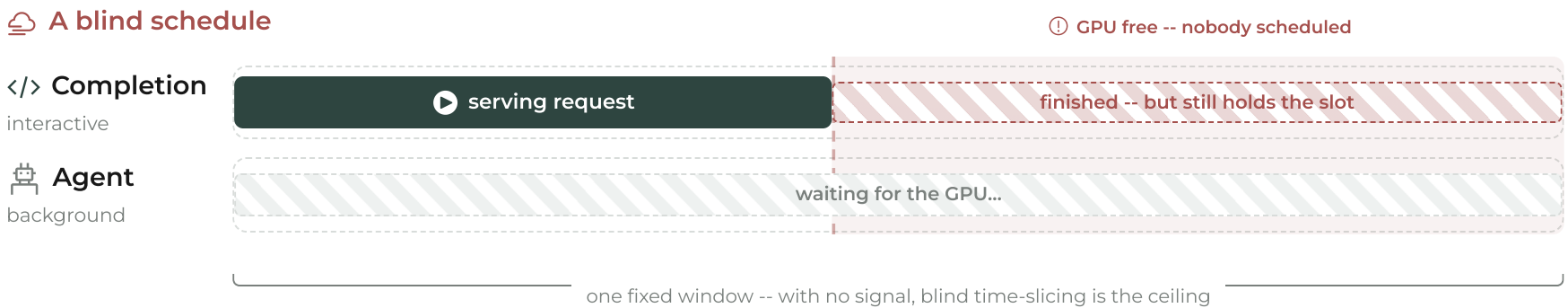
NVIDIA Unified Memory (UVM): transparent GPU↔host paging, moved **reactively on faults**



No coordination -- each swap **» its run window → thrashing**

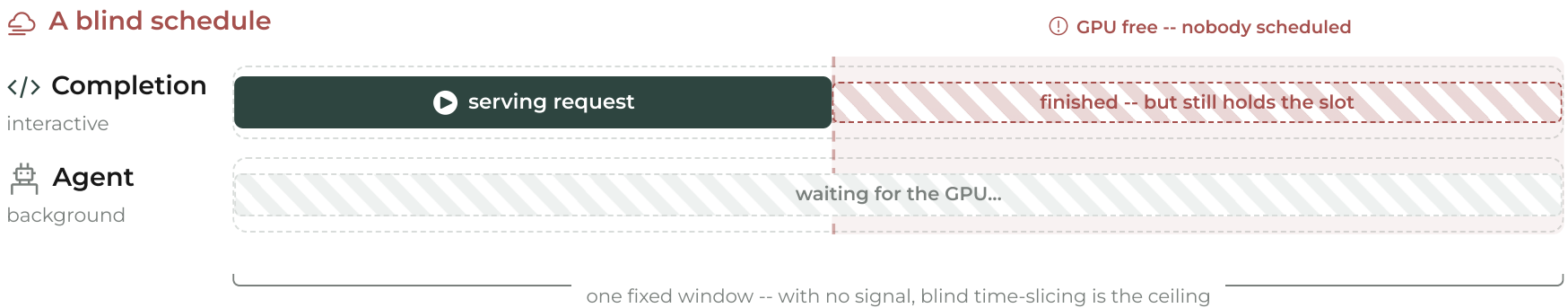
Challenge: Scheduling Without Hints

Scheduling relies on GPU application behavior, but more difficult to observe



Challenge: Scheduling Without Hints

Scheduling relies on GPU application behavior, but more difficult to observe



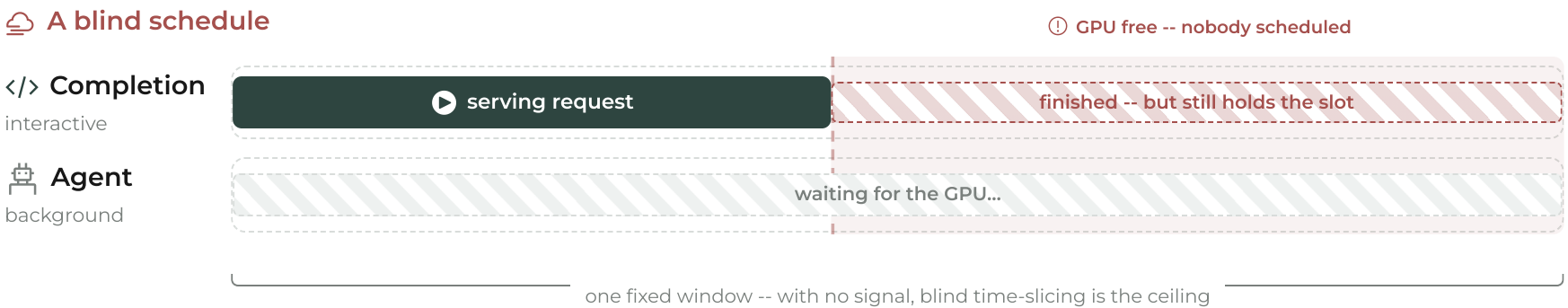
Goal **Schedule by responsiveness to cut idle** Promote interactive work, amortize background -- but this needs a signal.

...but the app exposes no boundary to act on

1. **No yield / sleep / exit** Nothing marks when an app pauses or finishes.
2. **Async launches** Real GPU work stays out of view.

Challenge: Scheduling Without Hints

Scheduling relies on GPU application behavior, but more difficult to observe



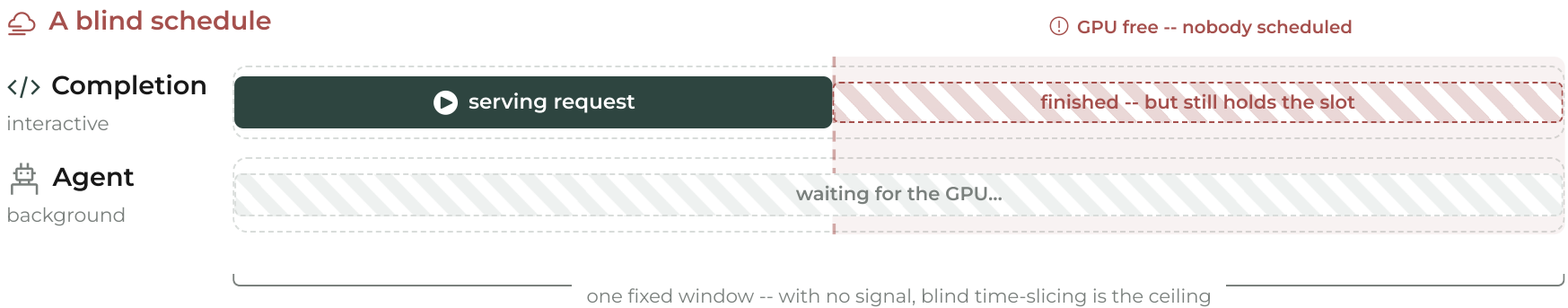
Goal **Schedule by responsiveness to cut idle** Promote interactive work, amortize background -- but this needs a signal.

...but the app exposes no boundary to act on

1. **No yield / sleep / exit** Nothing marks when an app pauses or finishes.
2. **Async launches** Real GPU work stays out of view.

Challenge: Scheduling Without Hints

Scheduling relies on GPU application behavior, but more difficult to observe



Goal **Schedule by responsiveness to cut idle** Promote interactive work, amortize background -- but this needs a signal.

...but the app exposes no boundary to act on

1. **No yield / sleep / exit** Nothing marks when an app pauses or finishes.
2. **Async launches** Real GPU work stays out of view.

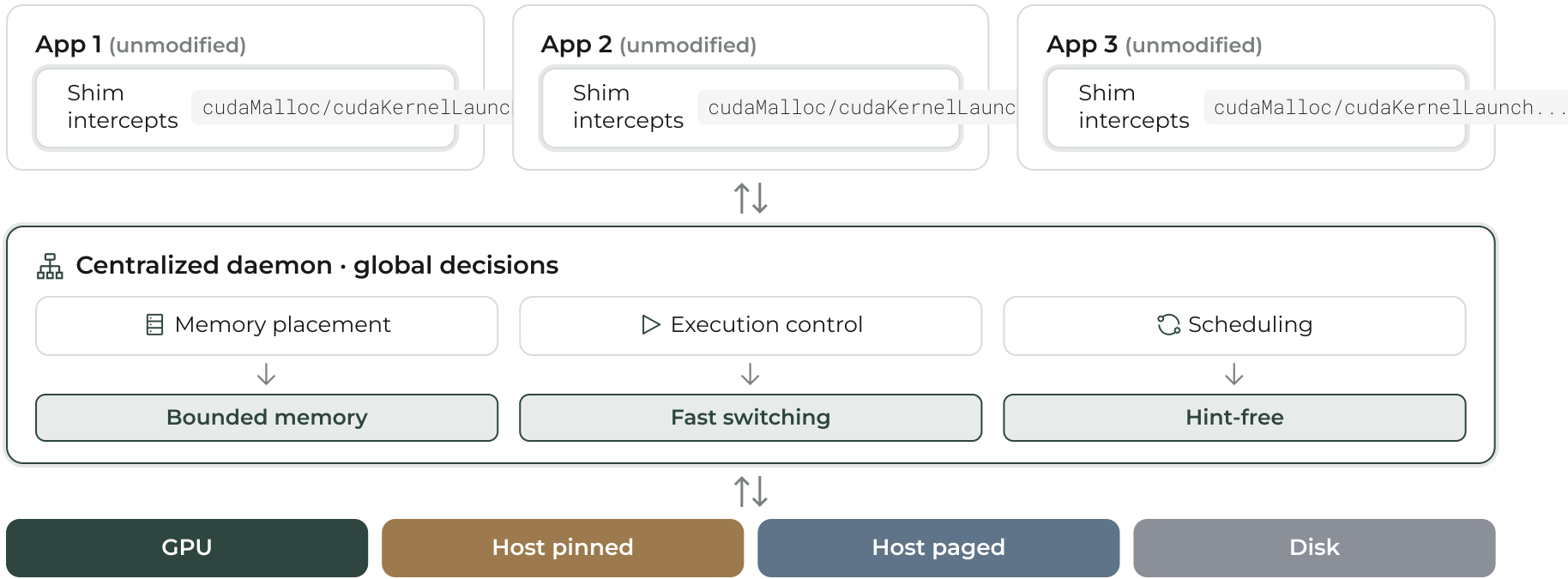
Challenge: Scheduling Without Hints

Scheduling relies on GPU application behavior, but more difficult to observe

CHALLENGE		REQUIREMENT
Memory models overflow GPU + host DRAM	→	**Bounded memory** across GPU + host tiers
Switching moving a model costs seconds	→	**Fast switching** no restart or reload
Scheduling no hints, no signals	→	**Hint-free scheduling** from GPU-visible behavior

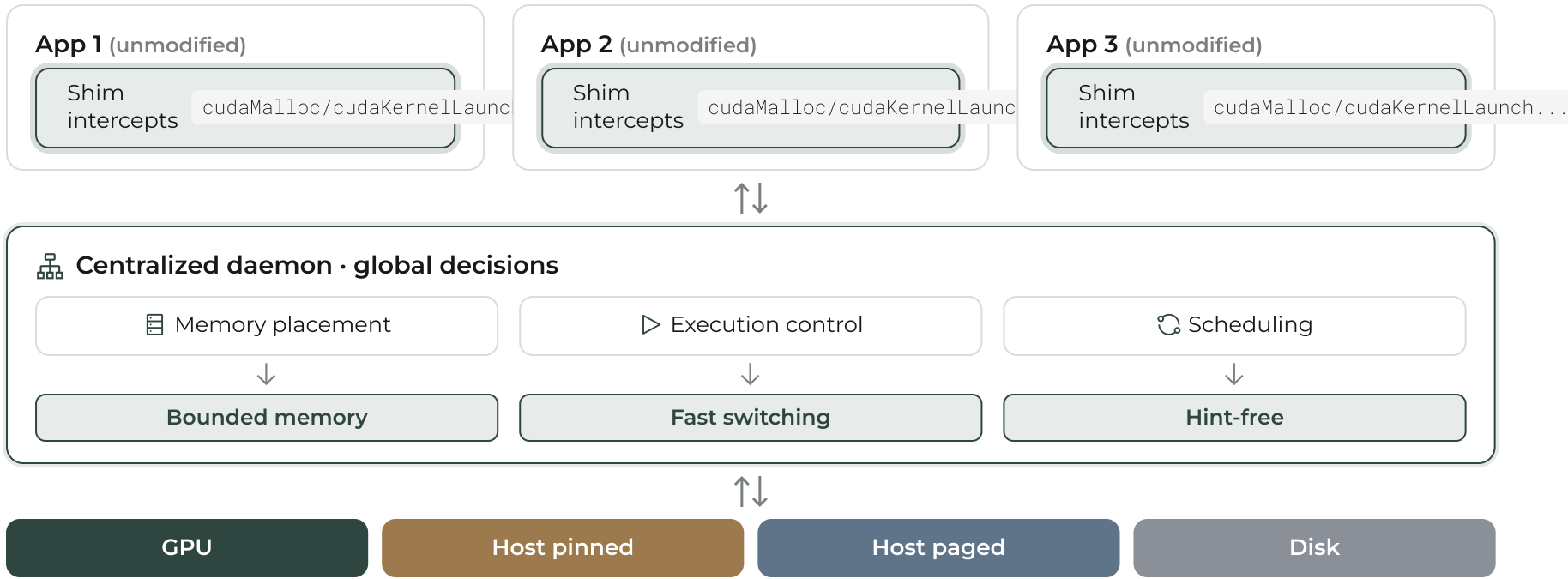
Nixie Overview

Two cooperating parts: a transparent shim in each app + a centralized daemon



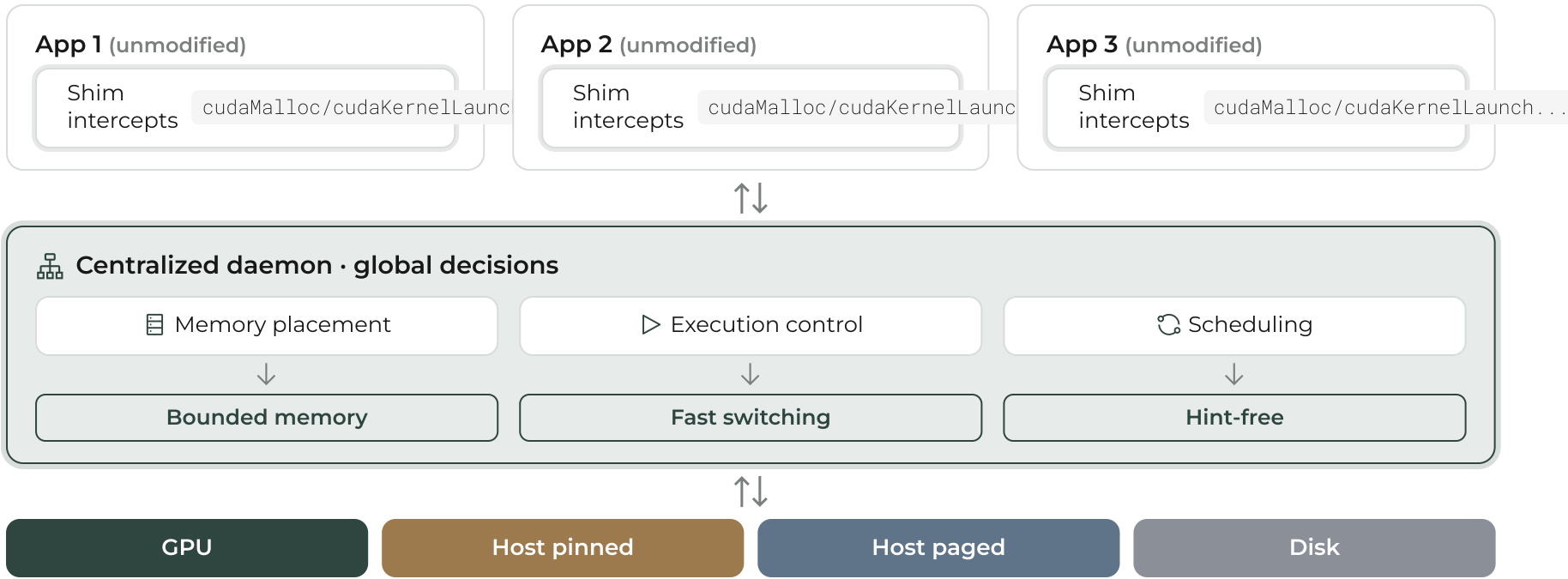
Nixie Overview

Two cooperating parts: a transparent shim in each app + a centralized daemon



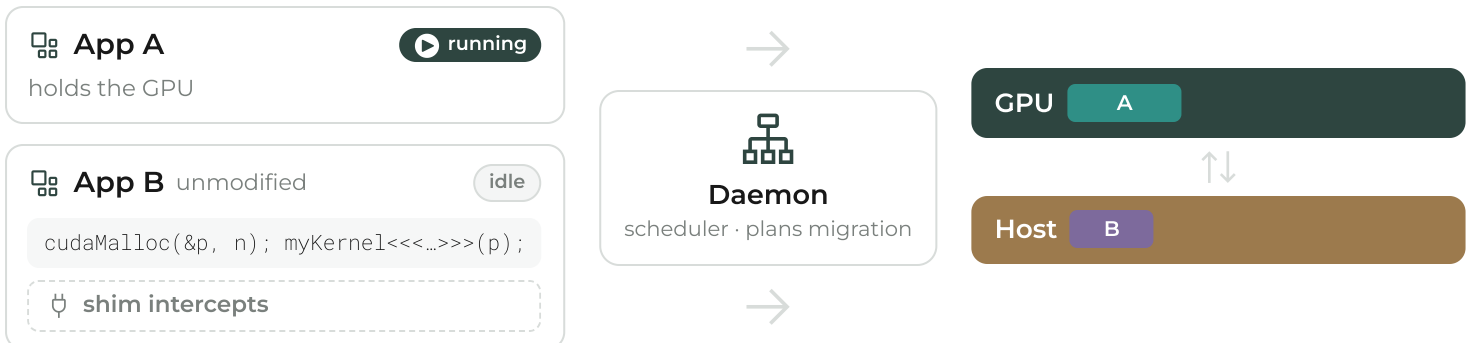
Nixie Overview

Two cooperating parts: a transparent shim in each app + a centralized daemon



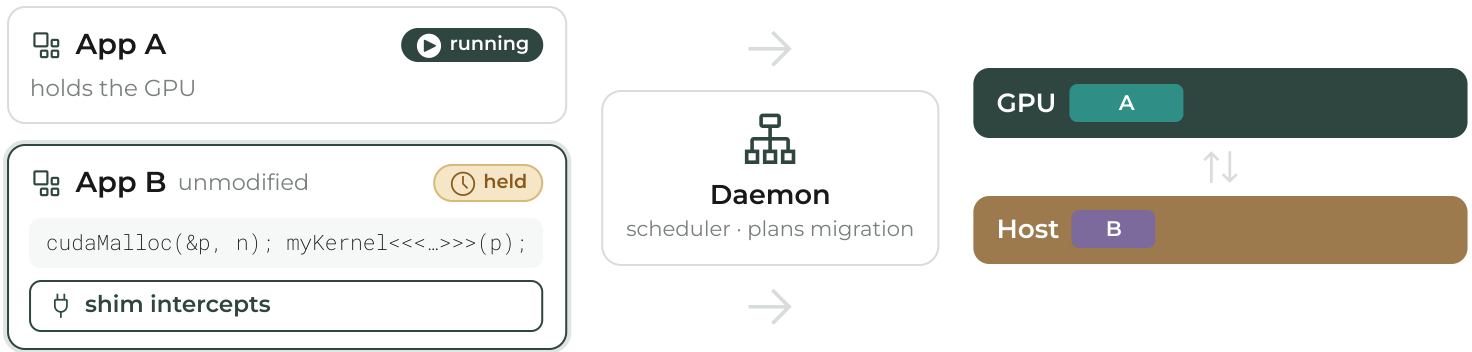
Co-management of Memory and Execution

Memory residency and permission to execute are coupled, kept transparent by remapping under stable addresses



Co-management of Memory and Execution

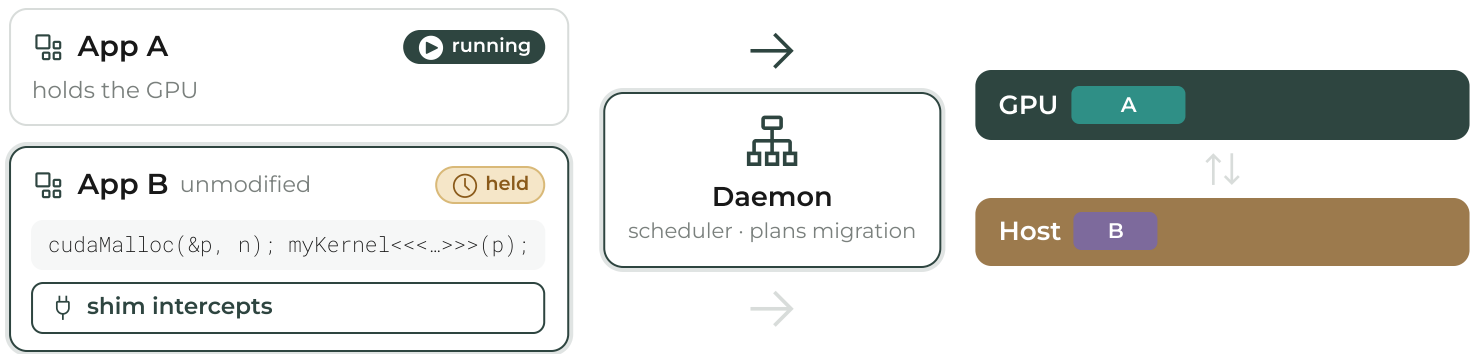
Memory residency and permission to execute are coupled, kept transparent by remapping under stable addresses



1. Unmodified **launch is intercepted** -- B's VA was reserved earlier at allocation

Co-management of Memory and Execution

Memory residency and permission to execute are coupled, kept transparent by remapping under stable addresses

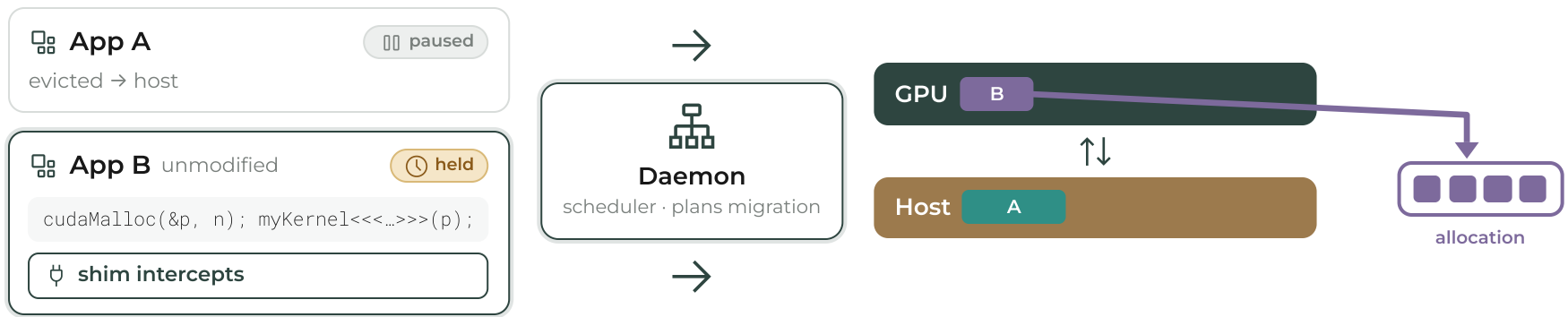


1. Unmodified **launch is intercepted** -- B's VA was reserved earlier at allocation

2. Shim **holds the launch** -- waits for the daemon **scheduler**'s approval & for memory to be resident

Co-management of Memory and Execution

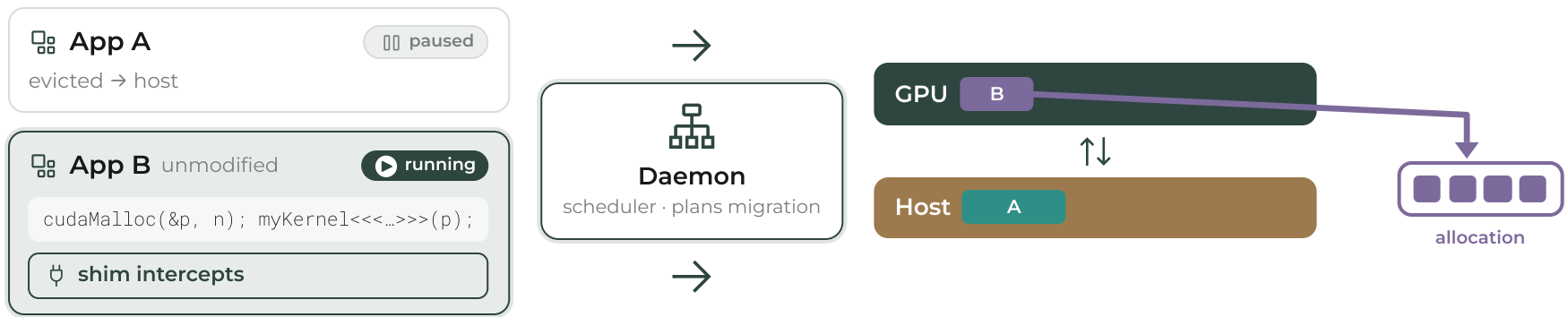
Memory residency and permission to execute are coupled, kept transparent by remapping under stable addresses



1. Unmodified **launch is intercepted** -- B's VA was reserved earlier at allocation
2. Shim **holds the launch** -- waits for the daemon **scheduler**'s approval & for memory to be resident
3. Daemon **coordinates the switch** -- A out, B in, remapped **block-by-block** to B's expected VAs

Co-management of Memory and Execution

Memory residency and permission to execute are coupled, kept transparent by remapping under stable addresses



1. Unmodified **launch is intercepted** -- B's VA was reserved earlier at allocation
2. Shim **holds the launch** -- waits for the daemon **scheduler**'s approval & for memory to be resident
3. Daemon **coordinates the switch** -- A out, B in, remapped **block-by-block** to B's expected VAs
4. B's launch proceeds -- **transparently**; neither app knew memory moved

Hierarchical Memory

One copy across tiers with bounded usage

App A active App B background OS / Browser

UVM · duplicated backing

GPU A A A A

Pinned A A A A B B B = copy of GPU + all of B

Host DRAM · tight

A copy | B copy | OS | Browser | !

squeezes OS / browser · can't reclaim

Nixie · one copy, multi-tier

GPU A A A A

Pinned B

Paged B

Disk B

Host DRAM · roomy

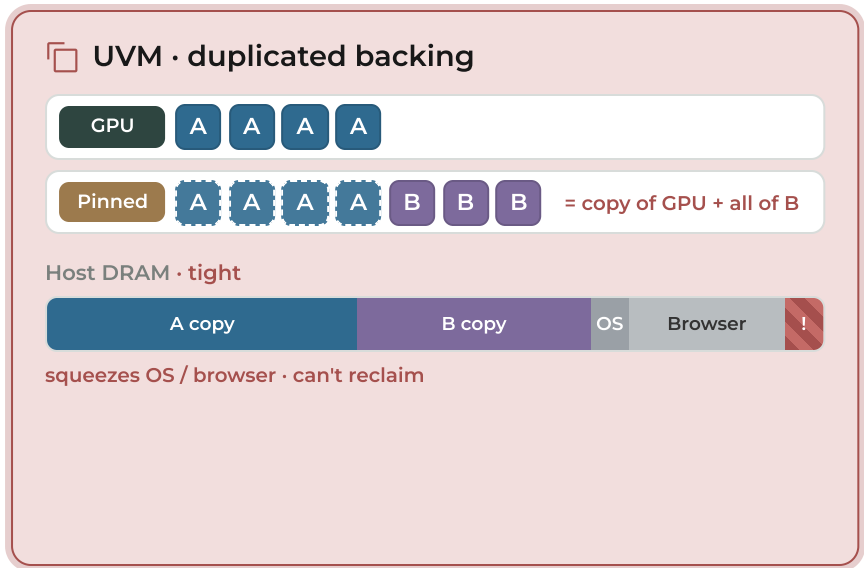
B | OS | Browser | free

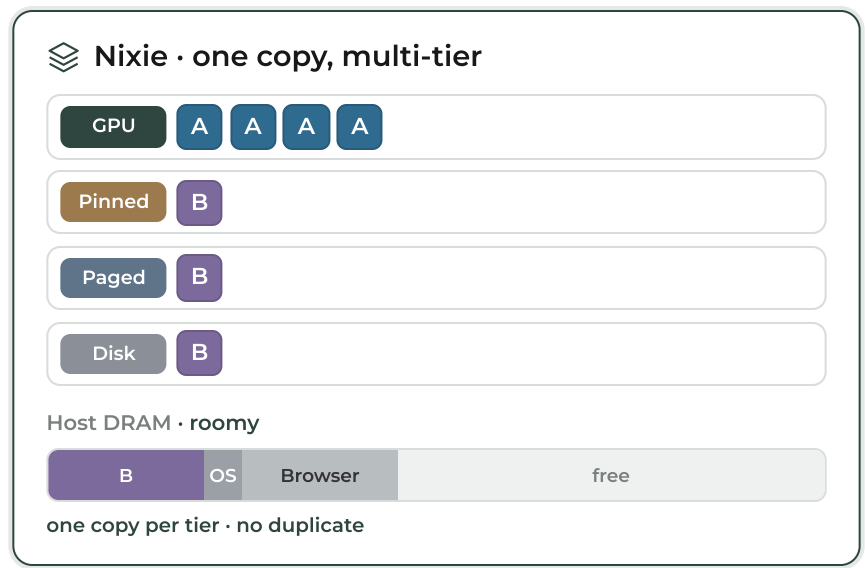
one copy per tier · no duplicate

Hierarchical Memory

One copy across tiers with bounded usage

App A active App B background OS / Browser

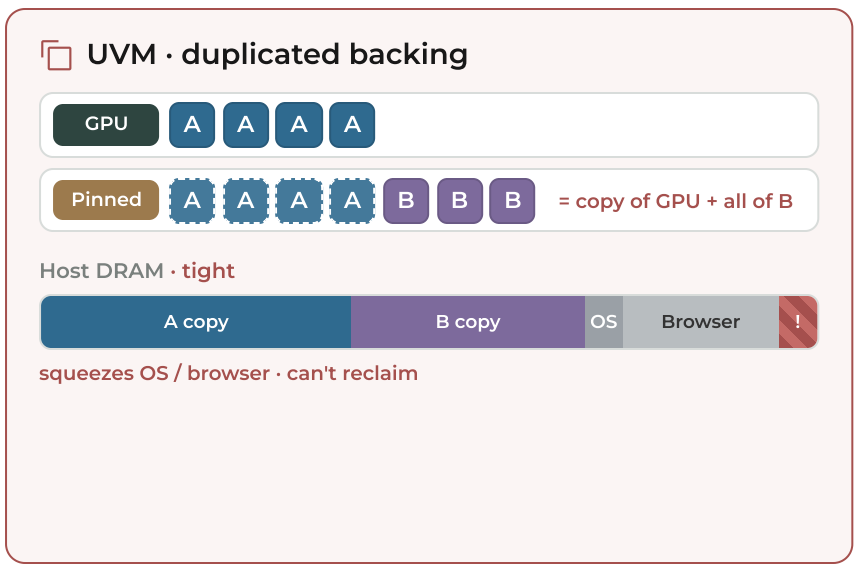


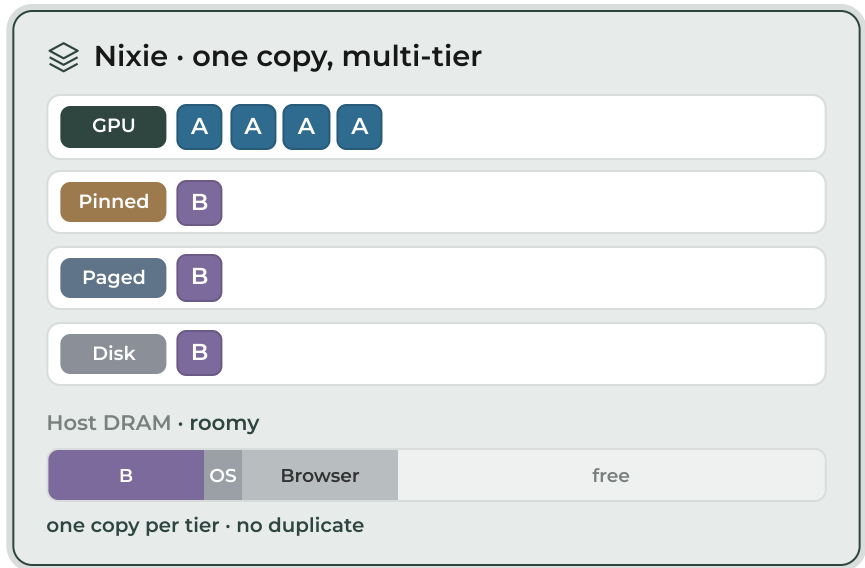


Hierarchical Memory

One copy across tiers with bounded usage

App A active App B background OS / Browser



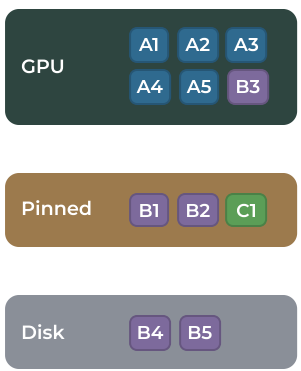


Migration w/ Global Planning & Pipelining

Proactive migration plan, not reactive fault handling

Plan the whole switch first

CURRENT STATE



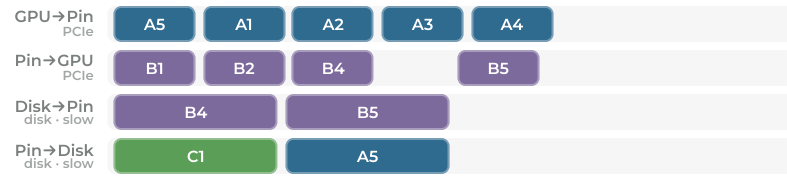
GENERATED PLAN

- A1 A2 A3 A4 — GPU → Pinned
- A5 — GPU → Pinned → Disk
- B1 B2 — Pinned → GPU
- B4 B5 — Disk → Pinned → GPU
- C1 — Pinned → Disk

① **Less movement** -- skip already-placed blocks (**B3**), evict only what's needed to admit B.

② **Chained moves first** -- start multi-hop transfers (disk→pin→GPU) early so their later hops finish off the critical path.

Nixie · pipelined, full-duplex



✓ Full-duplex PCIe (evict + fetch) overlaps the slow disk staging -- B4/B5 arrive just in time

UVM · fault-driven, half-duplex



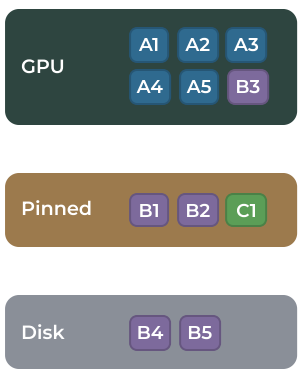
× Reactive page faults, half-duplex: evict & fetch never overlap; all of B kept pinned · **much longer migration**

Migration w/ Global Planning & Pipelining

Proactive migration plan, not reactive fault handling

Plan the whole switch first

CURRENT STATE



GENERATED PLAN

- A1 A2 A3 A4 — GPU → Pinned
- A5 — GPU → Pinned → Disk
- B1 B2 — Pinned → GPU
- B4 B5 — Disk → Pinned → GPU
- C1 — Pinned → Disk

① **Less movement** -- skip already-placed blocks (**B3**), evict only what's needed to admit B.

② **Chained moves first** -- start multi-hop transfers (disk→pin→GPU) early so their later hops finish off the critical path.

Nixie · pipelined, full-duplex

✓ Full-duplex PCIe (evict + fetch) overlaps the slow disk staging -- B4/B5 arrive just in time

UVM · fault-driven, half-duplex

× Reactive page faults, half-duplex: evict & fetch never overlap; all of B kept pinned · **much longer migration**

Migration w/ Global Planning & Pipelining

Proactive migration plan, not reactive fault handling

Plan the whole switch first

CURRENT STATE

GPU: A1 A2 A3 A4 A5 B3

Pinned: B1 B2 C1

Disk: B4 B5

GENERATED PLAN

A1 A2 A3 A4 — GPU → Pinned

A5 — GPU → Pinned → Disk

B1 B2 — Pinned → GPU

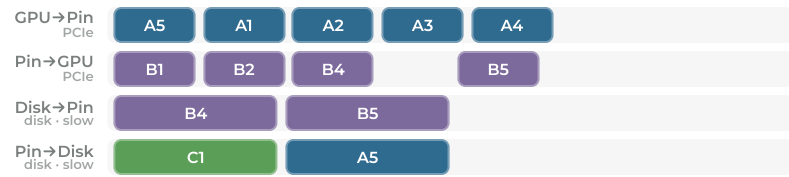
B4 B5 — Disk → Pinned → GPU

C1 — Pinned → Disk

① **Less movement** -- skip already-placed blocks (**B3**), evict only what's needed to admit B.

② **Chained moves first** -- start multi-hop transfers (disk→pin→GPU) early so their later hops finish off the critical path.

Nixie · pipelined, full-duplex



✓ Full-duplex PCIe (evict + fetch) overlaps the slow disk staging -- B4/B5 arrive just in time

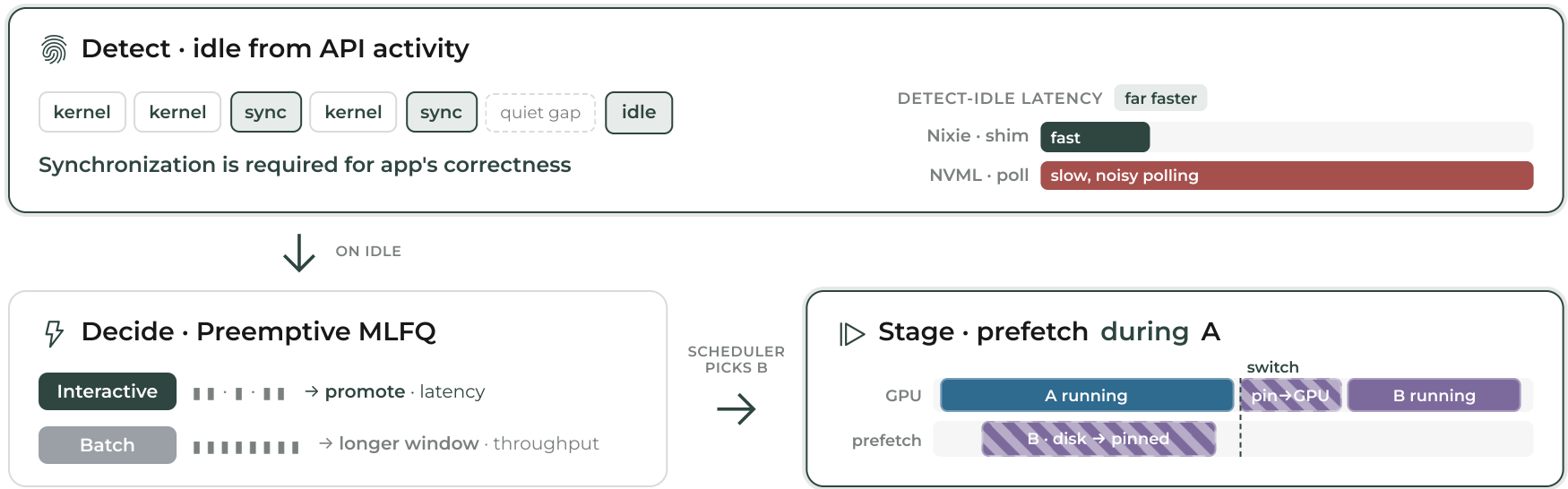
UVM · fault-driven, half-duplex



× Reactive page faults, half-duplex: evict & fetch never overlap; all of B kept pinned · **much longer migration**

Hint-Free Scheduling

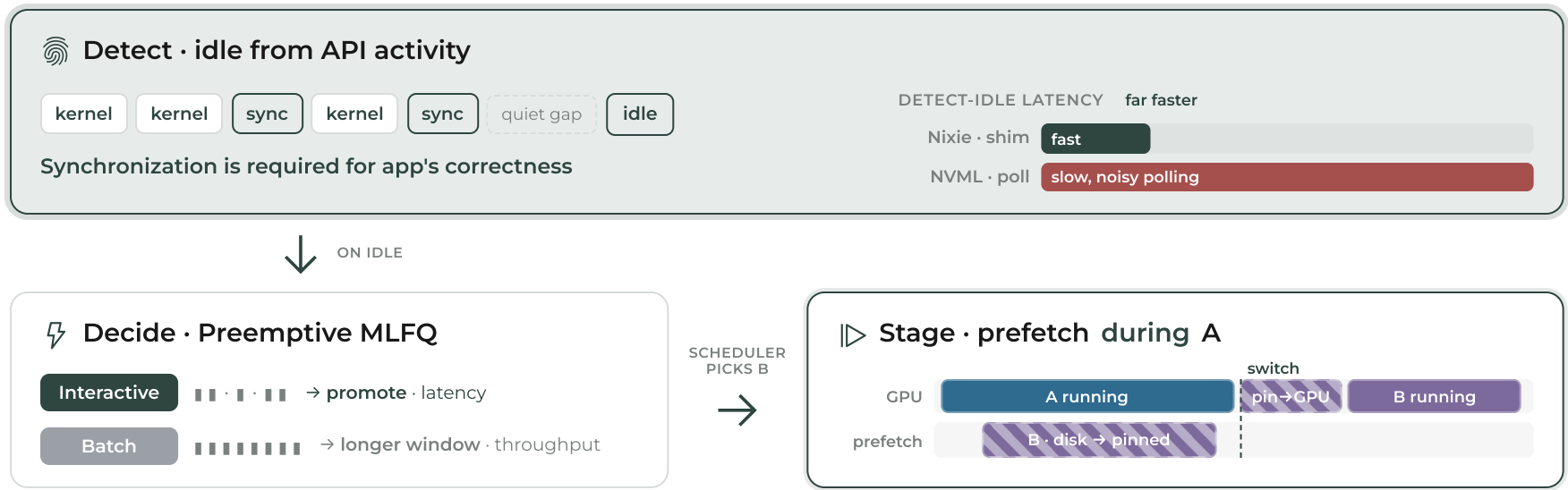
Infer request boundaries and priority from GPU activity without manual annotations



sync → idle → MLFQ → prefetch

Hint-Free Scheduling

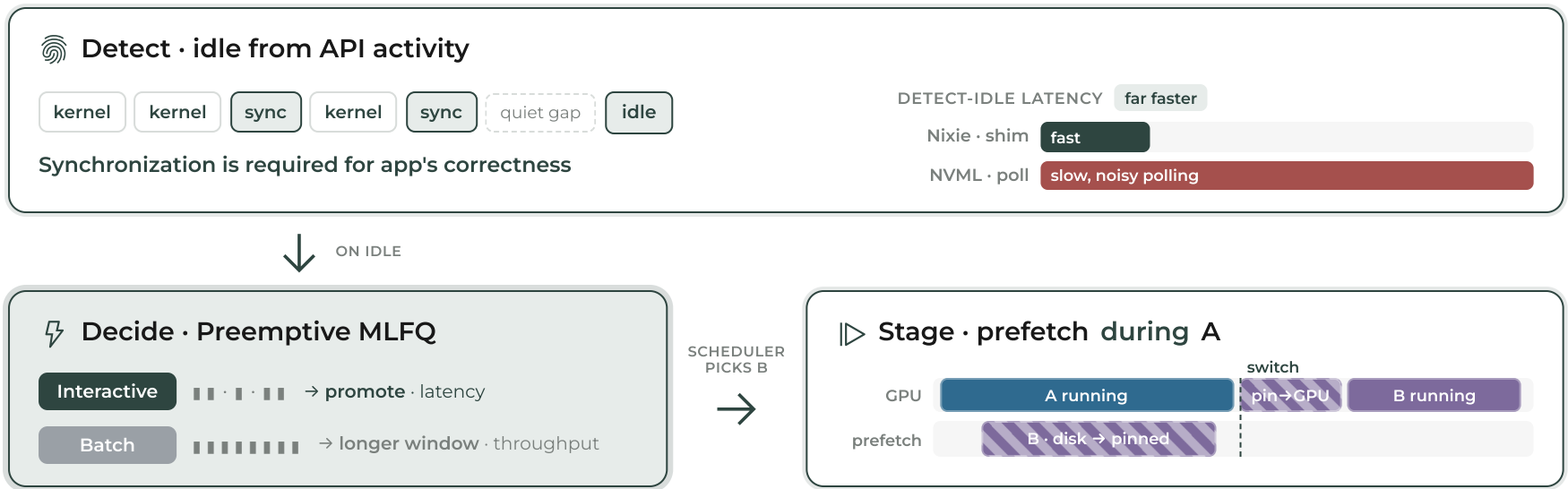
Infer request boundaries and priority from GPU activity without manual annotations



sync → idle → MLFQ → prefetch

Hint-Free Scheduling

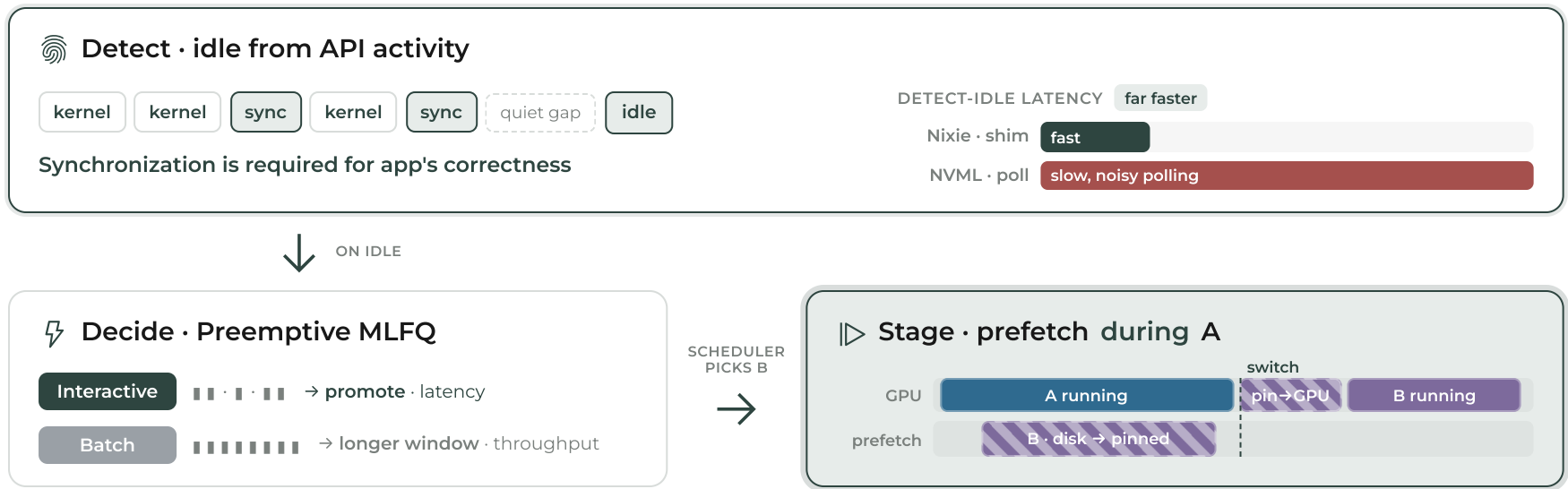
Infer request boundaries and priority from GPU activity without manual annotations



sync → idle → MLFQ → prefetch

Hint-Free Scheduling

Infer request boundaries and priority from GPU activity without manual annotations



sync → idle → MLFQ → prefetch

Evaluation

Speed

How fast is context switching -- latency & PCIe transfer throughput?

Memory

How much pinned host memory is needed for a target latency?

Real workloads

How does Nixie perform on real, unmodified applications?

Baselines

Ideal

- 2 GPUs

Memory only

- UVM

App-native

- Ollama

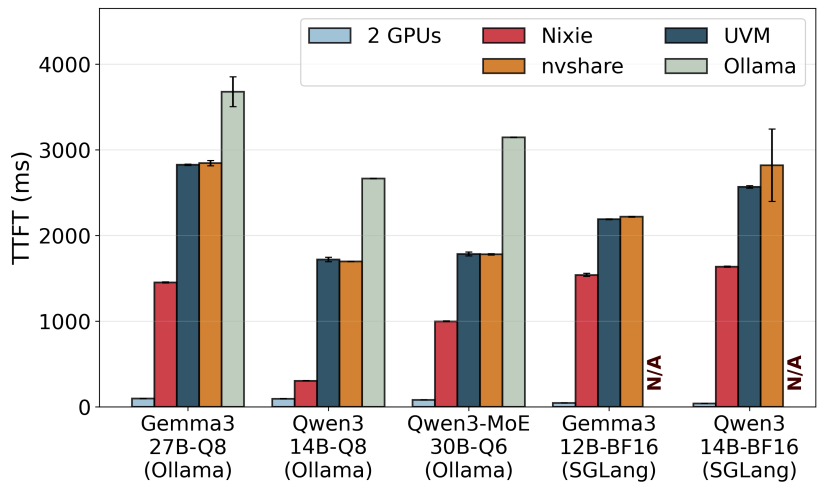
Prior systems

- nvshare
- TGS

Context-Switch Efficiency

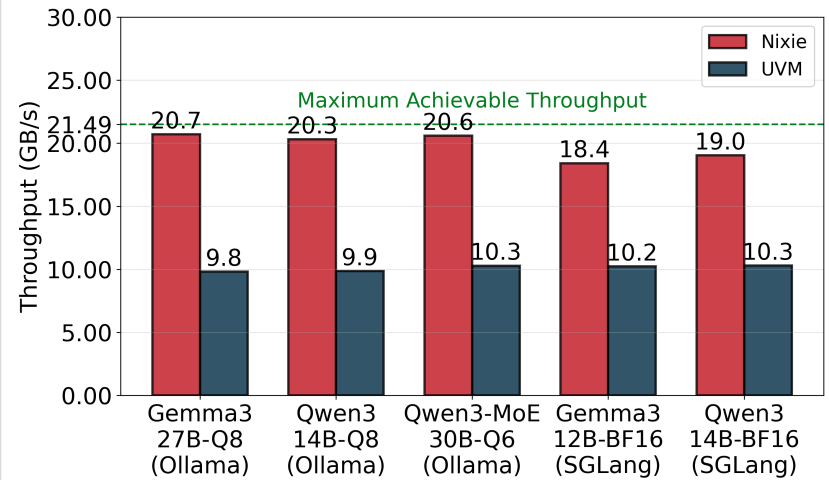
TTFT while rotating between two instances of the same app + model

Latency · TTFT (ms)



29–82% lower TTFT vs UVM, nvshare & Ollama

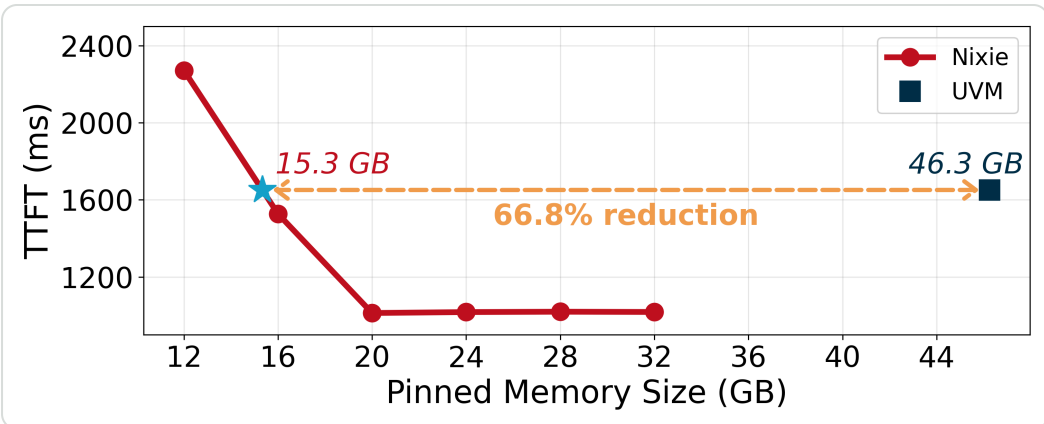
PCIe bandwidth (GB/s)



≈ 2× UVM bandwidth ~86–96% of 21.49 GB/s ceiling

Memory Efficiency

One curve, two ways to operate – trade pinned host memory against latency



Goal 1 · Lowest latency

At the ~20 GB knee, TTFT bottoms out at its minimum

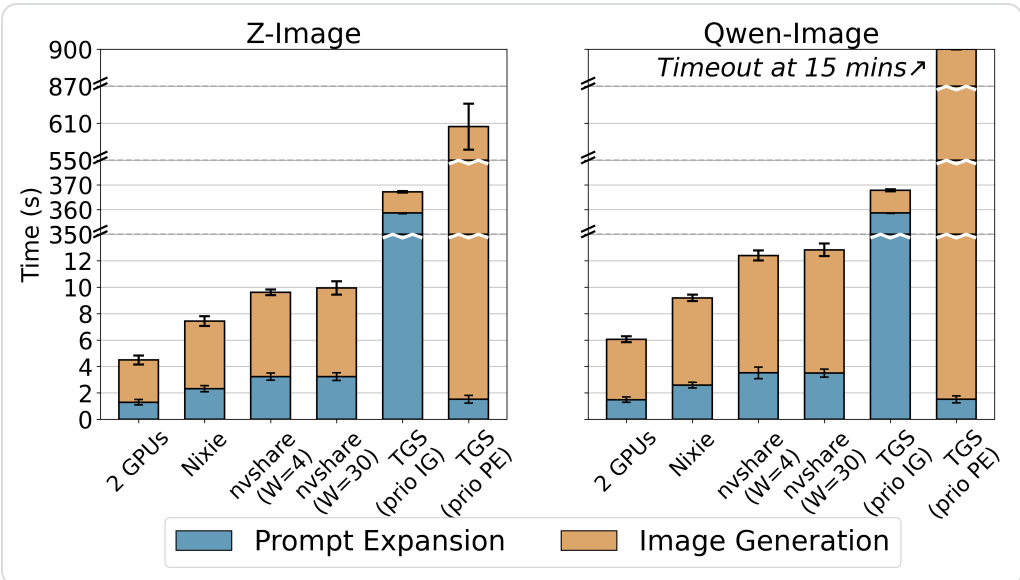
Goal 2 · Least memory

Match UVM's latency at just **15.3 GB** -- **66.8%** less pinned memory (vs 46.3 GB)

Pinned pages can't be swapped or compressed -- a smaller budget frees real DRAM.

Case Study: Prompt Extension

A real multi-app workflow: LLM + image generation



short prompt

↓

LLM expands

↓

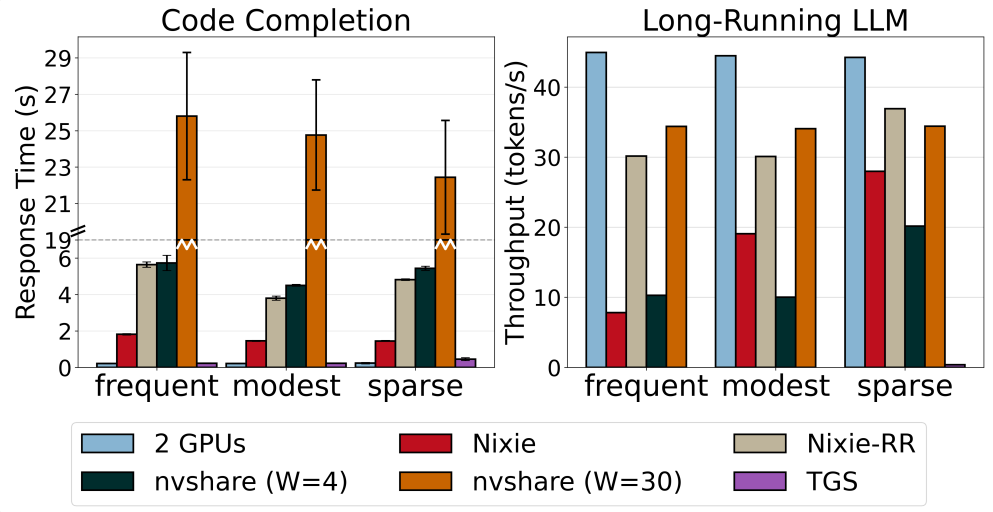
image model

1.3–1.4× faster than nvshare (4 s window)

Reaches **60–66%** of two dedicated GPUs

Case Study: Code Completion

Latency-sensitive completion alongside a background LLM job



Completion: **1.4–1.8 s**

nvshare-4s: 4.5–5.7 s · nvshare-30s: unusable

- **Nixie**: promotes the interactive job → keeps **+90%** background at the modest rate
- **Nixie-RR**: ablation with round-robin → **as slow as nvshare**
- **TGS**: fast, but **starves** the background job

Summary

Problem

Local ML creates independent, interactive GPU apps that overflow consumer GPU memory *and* host DRAM.

System

A transparent CUDA **shim** + centralized **daemon** co-manage execution, memory movement, and scheduling.

Performance

29.7–82.3% less context-switch overhead · up to **3.8×** better interactive latency on real apps.

Nixie -- transparent temporal multiplexing for consumer GPUs

OPEN SOURCE AT

https://github.com/XOR-op/nixie