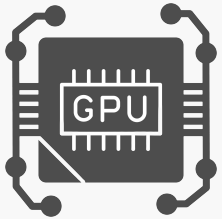
Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference

M. Yu (CUHK-Shenzhen & HKUST)
A. Wang, D. Nie, H. Yang, Y. Ding (Alibaba Group)
D. Chen, H. Yu, X. Luo, Z. Li, W. Wang (HKUST)
R. Chen (Nokia Bell Labs)

USENIX ATC' JULY 2025

01 Introduction

Introduction



Growing Interest in ML Inference on Serverless Platforms with GPU support:

- Ease of programming, maintenance, autoscaling, and pay-as-you-go billing

Current Limitation

Today's platforms lack efficient GPU support. (AWS Lambda, Alibaba Function Compute)

- Run an ML model in a container (or a microVM), early binding of functions to GPUs
  - Wasted GPU resources during idle.
  - Low utilization since 85% of functions run ≤ 1 request/min.

Four Desirable properties of serverless inference platform

① Pay-per-GPU-use billing, ② High GPU utilization via sharing, ③ Meet latency SLOs

④ Platform should achieve all three properties without requiring detailed knowledge about inference models

- Existing systems or research achieve some, but none satisfy all four together.

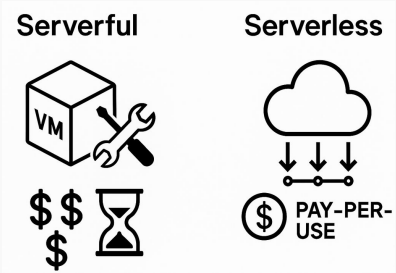
Serverless Inference

Serverful Approach (e.g., AWS SageMaker)

- Users configure VMs, GPUs, CPUs manually.
- Must manage resource provisioning & scaling.
- Pay for idle resources → high cost.

Serverless Inference (e.g., Alibaba Cloud)

- Users publish models as functions.
- Cloud handles provisioning, autoscaling, scheduling, fault tolerance.
- Pay-per-use → no idle cost.



Workload Characteristics

- Requests are dynamic and bursty.
- Serverless scales quickly to meet demand.
- Billing at fine granularity (e.g., 1 ms).

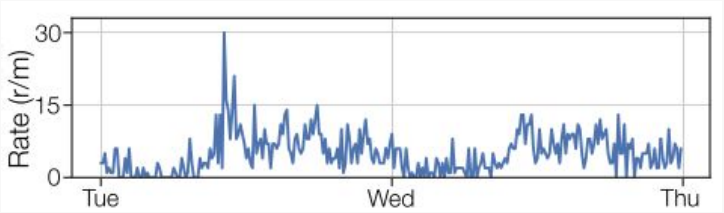


Figure: A two-day request trace of a typical GPU inference function in Alibaba Cloud

GPU Support in Serverless Platforms

Existing solutions and their inefficiency

- Deploy inference models as long-running containers where each container, when created, is bound to a specific GPU (i.e., early binding).
- Most functions are low-frequency (≤1 req/min).
- With one model per GPU, utilization is very low at realistic request rates
- Costly for both cloud users and providers

Model	Cold-start	Mem. footprint
ResNet-152 [17]	8	1.6 GB
Bert-qa [26]	11	2.4 GB
Stable Diffusion [18]	25	5.1 GB
Llama3-8B [11]	48	13 GB
Qwen-14B [16]	57	20.1 GB
Llama2-13B [10]	61	24.5 GB

Table: Cold start time(s) and memory footprint

Cost-Saving Strategy

- Reclaim GPUs when functions are inactive and provide to another function → avoid paying for idle GPUs (serverless)
- But it leads to frequent cold starts when functions are invoked again.
- Cold start includes: Function container setup, ML framework startup, GPU runtime creation, Model initialization

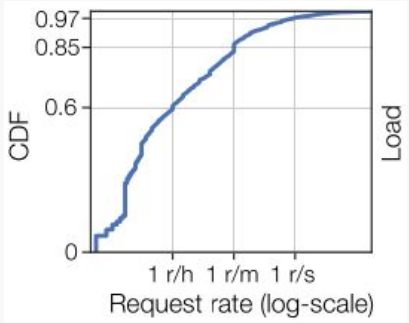


Figure: CDF of average function request rates from a one week production trace

02 Key Insight and Challenges

Late Binding & Model Swapping

- An efficient serverless inference platform should use late binding, where GPUs act as a shared pool and idle models stay in host memory until needed.
- This allows models to be swapped into any available GPU on demand, reducing costs and enabling deployment without major system changes.

Advantages

- Cost-Efficient : Eliminates GPU memory usage during idle time → true pay-per-GPU-use.
- Better Consolidation : Large host memory allows many low-frequency functions to share limited GPU memory.
- Low Latency : Efficient model swapping is better/faster than cold starts → helps meet SLOs.
- Model-Agnostic : Requires no model-specific knowledge; GPU pool manages memory seamlessly.

Challenges

Implementing GPU pooling & late binding in serverless platforms introduces three key challenges that must be addressed for efficiency and SLO compliance.

C1: Efficient GPU Pooling & Model Swapping

- Functions must synchronize with a remote GPU pool, unlike local GPU execution.
- This adds communication overhead, making it difficult to keep inference latency low.

C2: GPU Memory Management

- Late binding requires the system to monitor and manage GPU memory automatically.
- Must work without detailed knowledge of each model's structure → needs a unified memory management layer across the pool.

C3: SLO Compliance & Resource Efficiency

- Platform must design scheduling and model placement algorithms to use late binding effectively.
- Must balance meeting latency SLOs while ensuring high GPU utilization and cost efficiency.

03 Torpor System Design

Architecture Overview

Torpor has two major components: Cluster manager & Worker nodes

Cluster Manager:

- Request router → decides which worker node should handle an incoming request.
- Node manager → keeps track of workers' status (available GPUs, load, etc.).

Worker nodes:

- Host function instances and execute inference requests.
- Has router to forward requests to the right GPU executor.
- Use a GPU client to redirect CUDA API calls to the assigned executor.

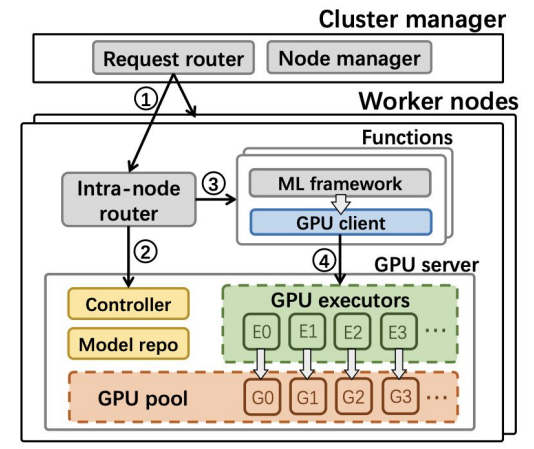


Figure: Architecture overview of Torpor

1. Request arrives at Torpor cluster → routed to worker node hosting the target function.

2. Router in worker node synchronizes with GPU server → queries for an available executor.

3. Router forwards request to function instance with the assigned executor ID.

4. Function instance processes the request → CUDA API calls asynchronously redirected to GPU executor.

5. Result returned to user after execution completes.

Architecture overview

GPU Remoting

Idea: Function instances do not directly own GPUs; they access GPUs through a GPU server.

- A GPU client in each function instance intercepts CUDA API calls.
- Calls are redirected asynchronously to the assigned GPU executor on the server.
- Only perform synchronization after final output.
- Optimization: Batch consecutive CUDA calls to reduce overhead further.

Model Swapping

- Swap models on demand at request level.
- Use pinned memory pool for faster host↔GPU transfer.
- Support cross-GPU swapping via NVLink.
- Pipeline Execution: Overlap model transfer and computation.
- Eviction: Keep a copy in host memory & invalidate GPU memory region.

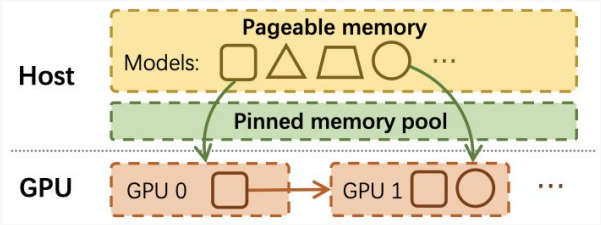


Figure: example of model swapping

Memory Management

Memory Management Requirements in Torpor

- **Late Binding:** Needs a share the pool of GPUs among functions ->functions always expect identical addresses across GPUs.
- **Model Swapping:** Frequent (de)allocations with native cudaMalloc add high overhead and latency.

Memory Address Management

- ML frameworks (e.g., PyTorch) store parameters in blocks.
- Torpor tracks blocks address → fast translation to physical addresses.
- Avoids heavy metadata maintenance → low overhead mapping.

Memory Block Allocation

- Reserves all GPU memory at bootstrap, manages blocks internally.
- Uses extended Buddy allocation to reduce fragmentation.
- Consolidates blocks per model

Isolation and Fault Handling in Torpor

Resource Isolation

- Container-level isolation for CPU & memory (like other FaaS).
- GPU server enforces one function per GPU at a time + memory region isolation.

GPU Runtime Modes

1. Runtime Sharing – one runtime for multiple models (trusted env., higher efficiency).
2. Runtime Isolation – dedicated runtime per model (untrusted env., stronger isolation).

   * Isolation mode switchable

Fault Handling

- Function failure: restarted automatically.
- Executor/Runtime failure: models migrated to another GPU executor via swapping.
- Runtime isolation: No cross-impact → other functions running on the same GPU are safe.
- Cluster-level resilience: metadata persisted, health checks & node replacement.

04

Torpor Policy Design

How Torpor meets the latency SLOs and delivers resource efficiency ?

Request Queueing

Goal: Maximize the number of SLO-compliant functions (e.g., 98% of requests finish within 100ms)

Approach: Prioritize functions more likely to meet their SLOs.

Key Challenges

1. How to quantify likelihood of SLO compliance?
2. How to order executions to improve compliance?

Solution: Required Request Count (RRC)

- Metric to measure the “effort” needed for a function to meet its SLO.
- $RRC=(pn-m)/(1-p)$

where n = total requests, m = requests meeting deadlines, p = SLO percentile (e.g., 98%).

Interpretation: Smaller RRC → higher chance of SLO attainment.

Execution Strategy

- Functions divided into high-priority (low RRC) and low-priority (high RRC).
- Dynamic threshold adapts to worker load → better global SLO compliance.

Scheduling and Model Swapping

Problem:

- Model swapping latency varies due to PCIe bandwidth contention.
- Larger models (e.g., BERT-QA) → heavier transfers → bigger slowdowns.

Interference-Aware Scheduling:

1. Model Categorization:
   a. Heavy models = bandwidth-intensive.
   b. Light models = smaller, less affected by contention.
2. Prefer GPU-to-GPU swapping via NVLink
   a. Faster than host-to-GPU.
   b. Reduces PCIe congestion.
3. When host-to-GPU is required:
   a. Avoid swapping multiple heavy models at the same time.
   b. Prioritize GPUs whose neighbors are idle or running light models.

Algorithm Flow:

- If model already on GPU → execute.
- Else if loaded on busy GPU → use GPU-to-GPU NVLink swap.
- Else → host-to-GPU swap, scheduled to minimize contention.

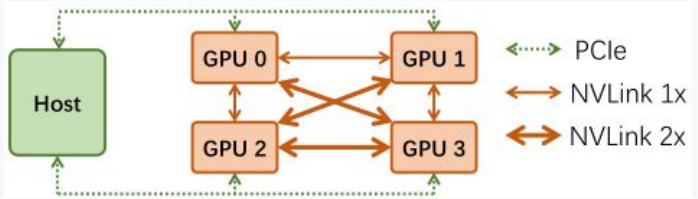


Figure: Topology of a 4-GPU worker node in Alibaba Cloud

Model Eviction Policy

Goal: reduce bandwidth contention + boost inference performance

Key idea: light models cost little to swap, heavy ones cost a lot

Insights

- Traditional eviction (e.g., LRU) only minimizes miss rate.
- Torpor focuses on performance impact of swapping different models.
- Light models → negligible swap overhead
- Heavy models → large PCIe/NVLink contention

Policy

Global GPU Memory Pooling

- Treats all GPU VRAM in a node as one pool
- Each model allowed up to 1 replica across GPUs
- Reduces redundant transfers, improves caching

Evict Light Models First

- Swapping light models = little/no performance hit
- Frees space quickly without hurting throughput

Fallback: LRU for Heavy Models

- If only heavy models remain → evict least-recently-used

05 Evaluation & Results

Implementation

Deployment: Implemented and deployed on Alibaba Cloud, a large-scale commercial serverless platform.

Codebase:

- GPU Server → ~4K lines of C++
- GPU Client → ~1.5K lines of C++

Integration:

- Intra-node router & cluster manager built atop existing Alibaba Cloud components.
- Late binding requires no intrusive changes to cluster management logic.

Function Template:

- Container image as a function template built on PyTorch.
- CUDA libraries (e.g., libcudart.so) replaced by GPU client → enables transparent GPU remoting.
- No modification to PyTorch needed.

Production Ready:

- Successfully deployed in real-world Alibaba Cloud environment.

Evaluation Setup

Platform & Cluster:

- Deployed on Alibaba Cloud with production-level specs.
- Cluster: up to 6 workers.
- Each worker: 48 vCPUs, 384 GB memory, 4× NVIDIA V100 (32 GB).

Workloads:

- 8 popular ML models.
- All functions warmed up before testing.

Baselines:

- Native execution : default Alibaba Cloud serverless.
- INFless : state-of-the-art serverless inference system.

Model	Native	GPU remoting	Swap-PCIe	Swap-NVLink
DenseNet-169	30	25	27	26
DenseNet-201	36	28	30	30
Inception-v3	19	14	17	16
EfficientNet	17	12	13	13
ResNet-50	11	9	13	11
ResNet-101	20	14	22	16
ResNet-152	25	17	25	20
Bert-qa	42	43	144	45

Figure: Various models and their latencies (ms) with GPU remoting and model swapping

Metrics:

- SLO Attainment → % of functions meeting tail latency deadlines.
- GPU Load → proportion of time GPU is actively processing inference.
- Tail latency thresholds: CV models = 80ms, BERT-QA = 200ms (98th percentile).

Overhead of Torpor's Late Binding

Performance of Torpor's GPU remoting

This comparison shows inference latency of Torpor vs other GPU remoting techniques

- **GVirtuS**: High latency (138 ms / 68 ms).
- **Torpor w/o batch**: ↓79% (ResNet-152)
- **Torpor (full)**: ↓88% (ResNet-152) / 37% (Bert-qa)

Insight: Async redirection cuts sync overhead, and batching further reduces latency for call-heavy models.

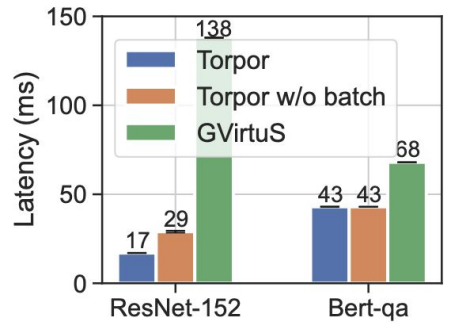


Figure: Inference latency with Torpor and other GPU remoting techniques

Performance breakdown of model swapping

This comparison shows how Torpor's model swapping designs improve inference latency.

- **Baseline**: Direct swap + inference (slowest).
- **Pinned**: Pinned memory pool → ~35% latency reduction.
- **Pipeline**: Overlaps swapping & execution → extra ~15% gain.
- **Group**: Groups parameters for efficient pipelining → up to 22% more improvement (notably ResNet-152).

Insight: Pinned memory, pipelining, and grouping progressively cut swapping overhead, with grouping especially effective for models with many small parameters.

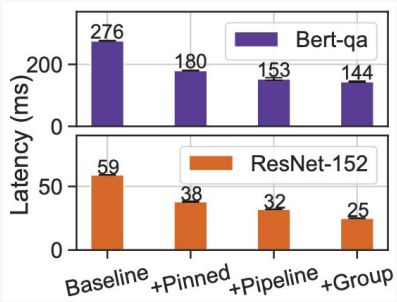


Figure: Performance breakdown of Torpor's model swapping via PCIe

Benefits of Torpor's Late Binding

GPU efficiency for low-frequency functions

This experiment evaluates GPU efficiency for low-frequency functions. Multiple ResNet-152 functions per GPU, request rate = 80 → 10 r/m.

- **Throughput**: Torpor sustains high throughput; >10× Native at 10 r/m
- **Latency**: Torpor sustains tail latency < 50 ms even with model swapping.

Insight: Late binding consolidates many low-frequency functions while preserving efficiency and SLOs.

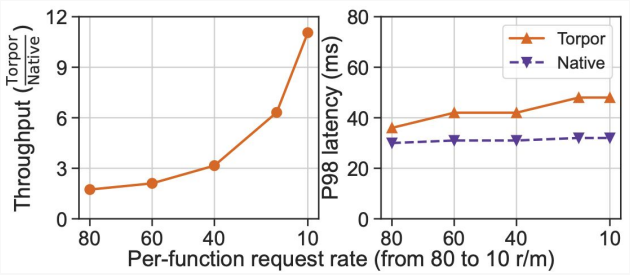


Figure: Performance of executing multiple ResNet-152 functions on a single GPU with Torpor's late binding (Torpor) and Native execution.

Cross-GPU load balancing for high-frequency functions.

Evaluates 40 high-frequency ResNet-152 functions on a 4-GPU worker (~200 r/m).

Load Distribution: Native: Some GPUs overloaded from request bursts & Torpor: On-demand model migration → balanced load across GPUs.

Tail Latency: Native: Severe queueing delays → multi-second tail latency & Torpor: Consistent tail latency ~35 ms on all GPUs.

Insight: Late binding enables dynamic model migration, preventing overloads and ensuring predictable latency.

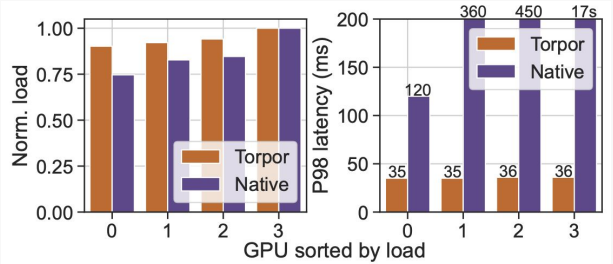


Figure: The normalized per-GPU load and the tail request latency with Torpor's late binding (Torpor) and Native

Torpor at A Node

Performance comparison

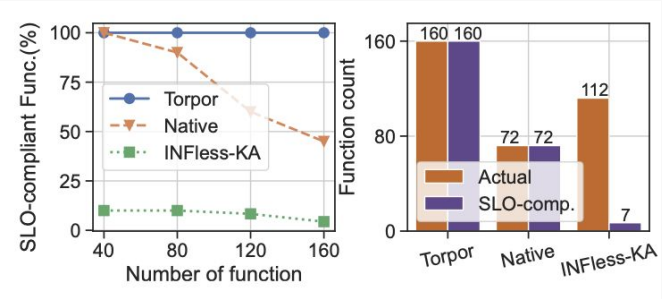


Figure: Performance comparison in terms of SLO compliance between Torpor, Native, and INFless-KA.

Efficiency of model heaviness

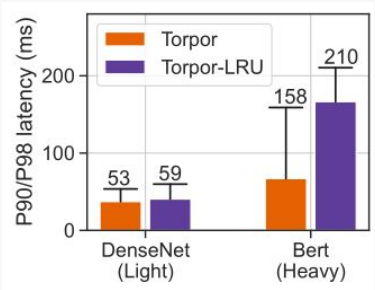


Figure: 90th (bars) and 98th (whiskers) tail latencies in Torpor and Torpor-LRU

Benefits of Torpor's policies

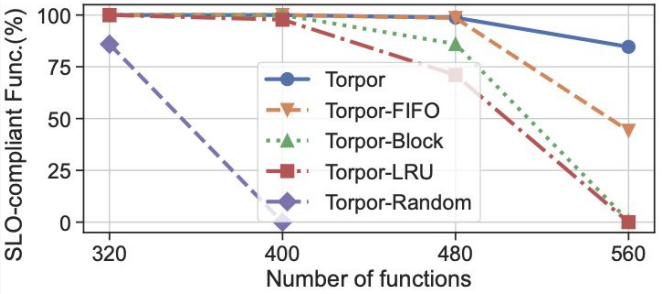


Figure: Ratio of SLO-compliant functions with the full Torpor and various different policies

Efficiency of memory allocation

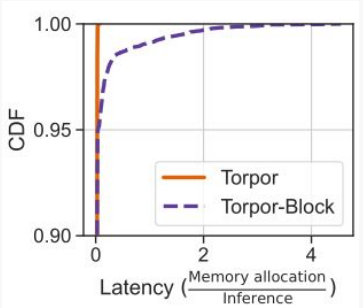


Figure: Latency CDF of memory allocation in Torpor and Torpor-Block.

05 Conclusion & Takeaways

Conclusion & Takeaways

Problem:

- Existing serverless platforms waste GPU resources due to early binding and suffer from cold start overheads

Solution (Torpor):

- Late binding with GPU pooling & model swapping.
- Asynchronous API redirection + efficient memory management.
- SLO-aware scheduling for performance & fairness.

Results:

- 10× throughput vs Native for low-frequency functions.
- Tail latency consistently <50 ms, even under swapping.
- Balanced GPU load, predictable performance.
- 70% user cost savings in Alibaba Cloud pilot deployment.

Key Takeaway:

- Torpor makes serverless inference GPU-efficient, cost-effective, and production-ready.

Limitations & Future Work

Workload Scope:

- Optimized for low-frequency, bursty functions; less benefit for stable workloads with high-frequency.

Single-tenant per GPU at a time

- Only one function executes at a time in one GPU. Can be extended to use gpu sharing techniques like MPS/MIG.

Heavy reliance on fast interconnects (NVLink)

- Not portable across hardware without NVLink.

No model-parallel or pipeline-parallel support

- Torpor doesn’t yet handle very large models that span multiple GPUs.
- That makes it unsuitable for today’s frontier LLMs (70B+ parameters)

Thanks!