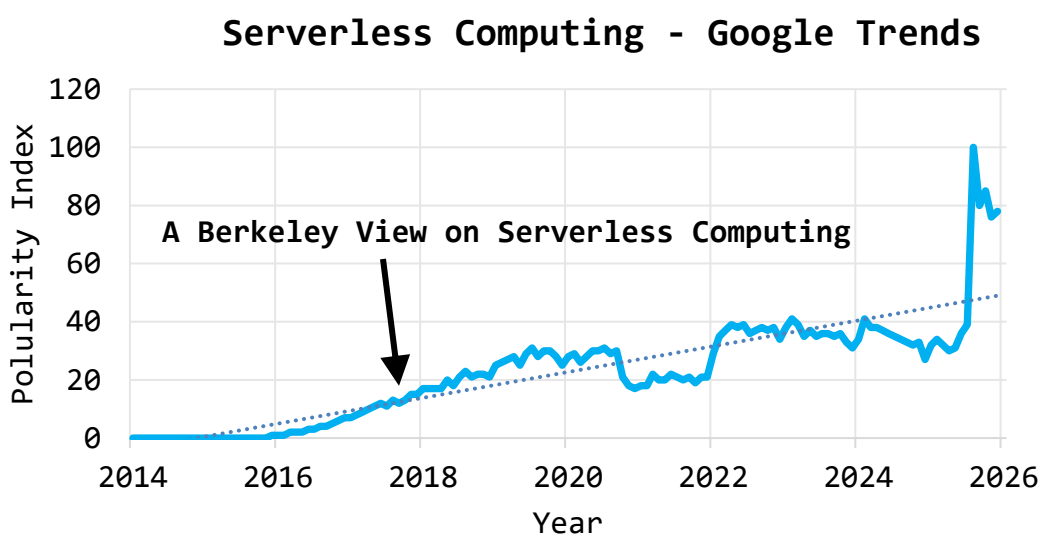
gShare: Efficient GPU Sharing with Aggressive Scheduling in Multi-tenant FaaS platform

Yanan Yang[1] Zhengxiong Jiang[2] Meiqi Zhu[2]
Hongqiang Xu[2] Yujun Wang[2] Liang Li[1]
Jiansong Zhang[1] Jie Wu[1]
China Telecom Cloud Computing Research Institute[1]
China Telecom Cloud Technology Co. Ltd.[2]

Serverless computing is acceleratedly arising

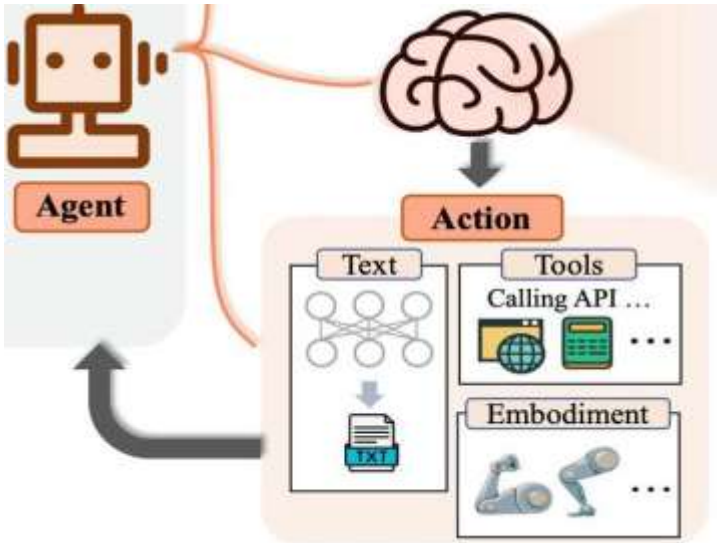
- The first commercial platform AWS Lambda in 2014
- More and more cloud services are using FaaS (Function-as-a-service) paradigm



We live in the age of

Artificial Intelligence

- Smart city/transport, LLMs, AI agent, etc.



ML serving/inference:
a **critical part** of our daily life.

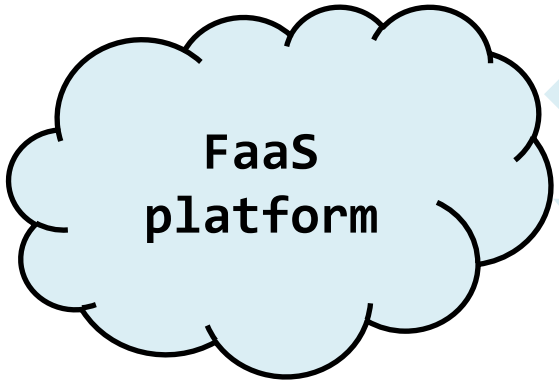
A Common Case:

Serving ML model with FaaS Functions

- Resource management free
- Auto-scaling in/out
- Pay-as-you-go

upload code

service



Cloud Infra.

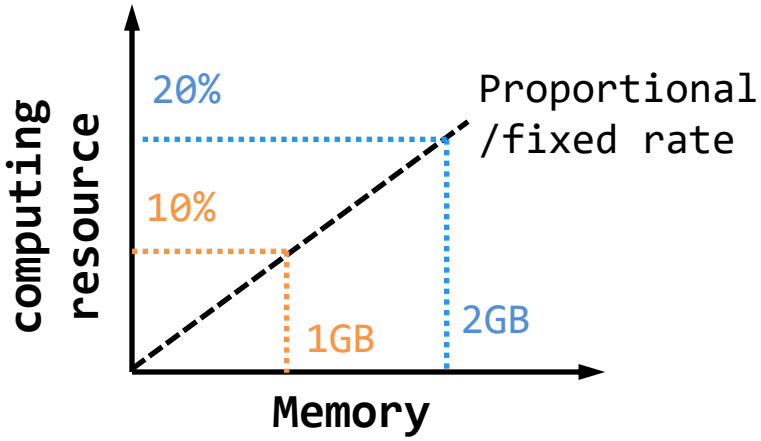
Middleware

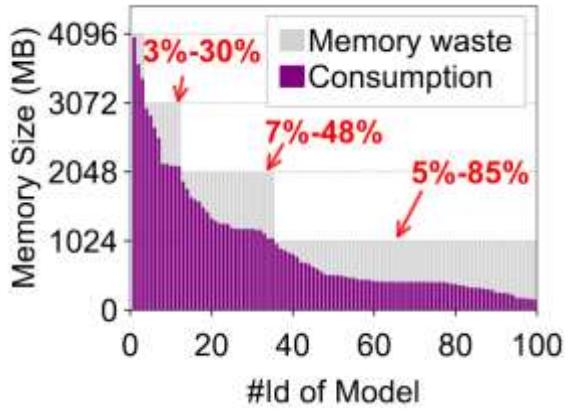
shield the cloud details

GPU has been supported in current FaaS platforms:
Azure Container Apps, Alibaba Cloud Functions, etc.

The Problem

- Coarse-grained GPU functions lacks efficiency
- Resource over-provisioning leads to high user cost

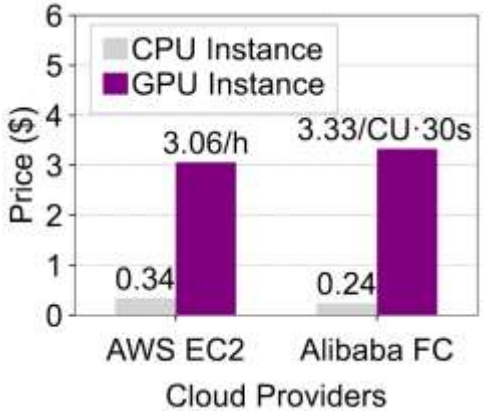




Small models still take a considerable proportion

The Problem

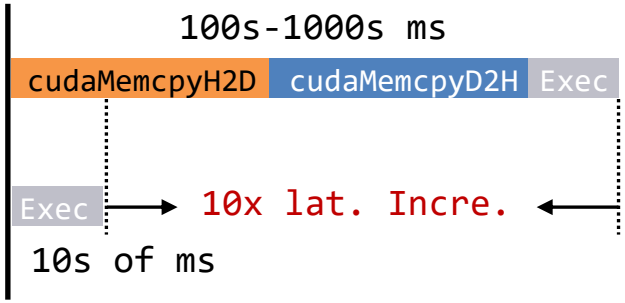
- Coupled GPU cannot be flexibly reclamated
- GPU function idling exacerbates resource waste

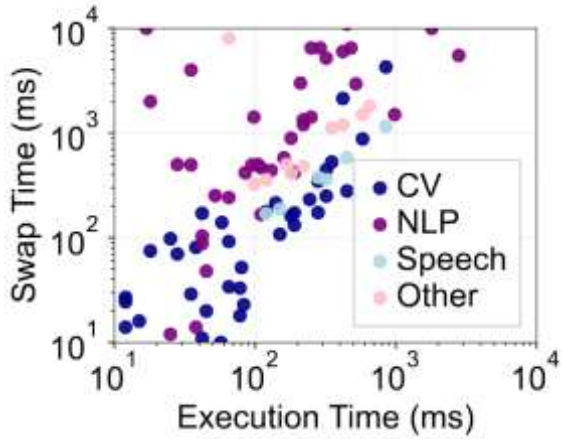



```
# GPU are much more expensive than CPUs
# 10× higher keepalive cost v.s. CPU functions
```

Inefficiency of Existing Methods

- Inflexible and heavy model swapping
- SLO-Oblivious scheduling decisions





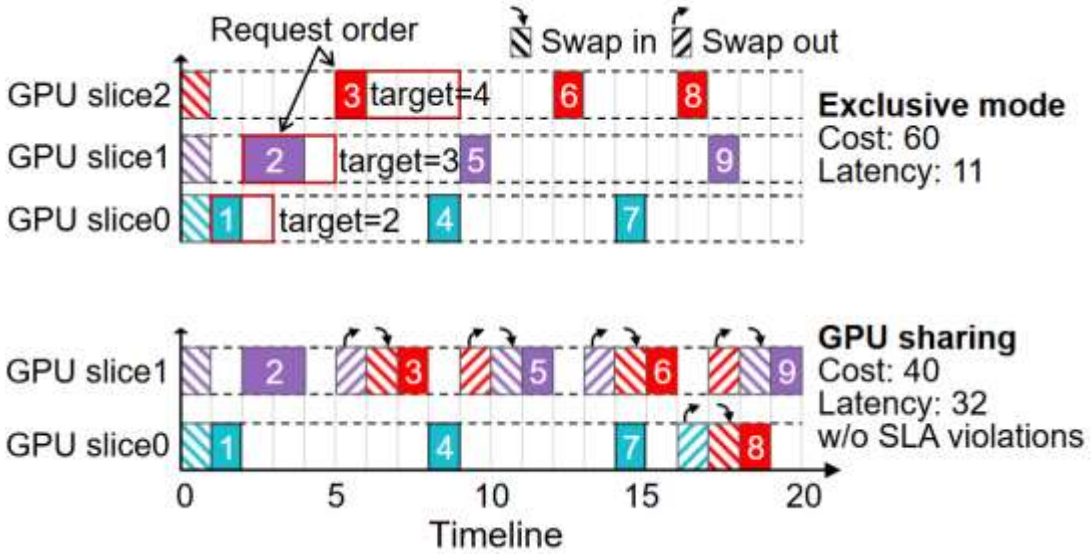
Model swapping may experience long latency
Simple heuristic scheduling results in either low GPU utilization or SLO violations

Motivations

Goal: Cost-efficient GPU-enabled Muti-tenant FaaS platform

- Fine-grained GPU allocation (time-slicing with vGPU)
- Resource decoupling & Flexible sharing
- Efficient GPU overselling

← *principle*



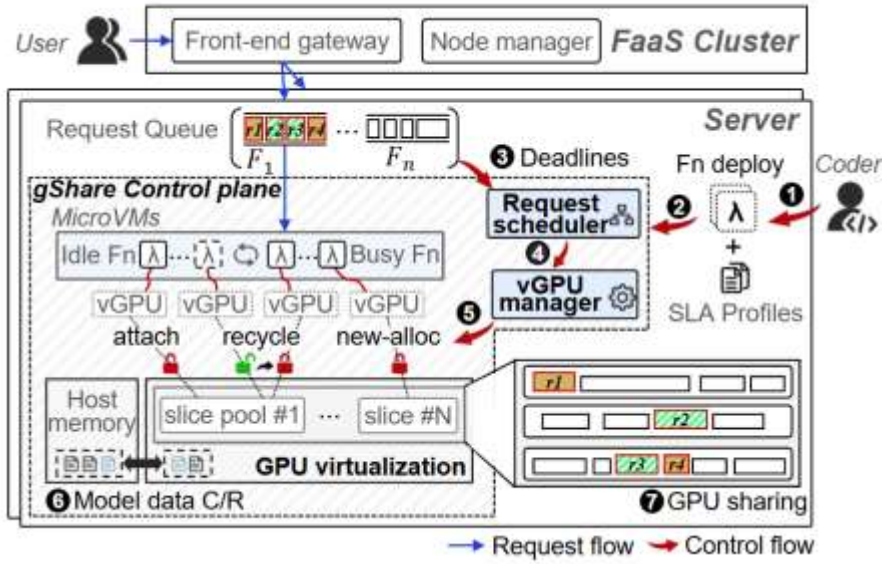
Low efficiency

↓

High efficiency

Design of gShare

- Designed for Muti-tenant FaaS platform
- GPU sharing with GPU virtualization technology
- Discounted billing model for over-selling



Domain-specific GPU function management policy

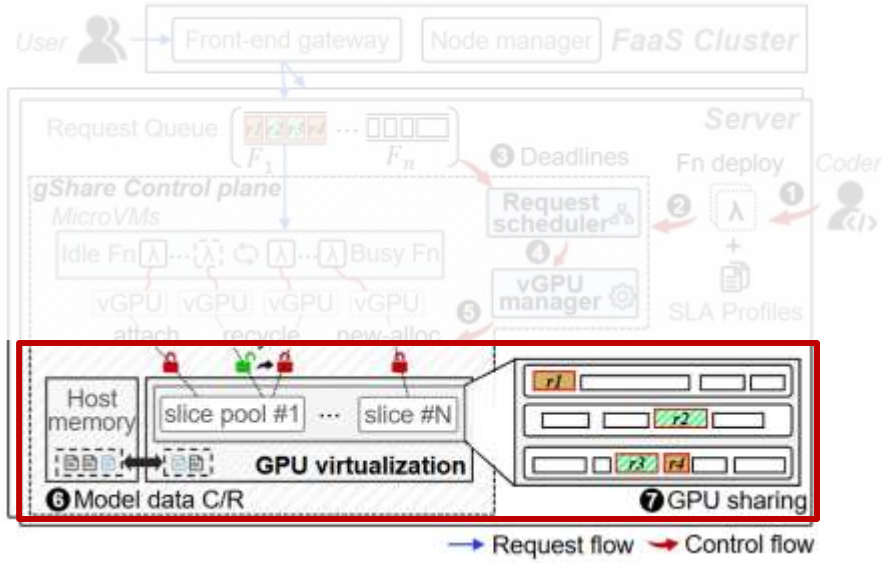
#1 Fine-grained GPU allocation (e.g., minimum 1%vGPU)

#2 Decoupled and flexible GPU recycling with vGPU hot-plugging and fast model swapping

#3 Profiling-based SLO-aware GPU shceduling & sharing

Design of gShare

- Designed for Muti-tenant FaaS platform
- GPU sharing with GPU virtualization technology
- Discounted billing model for over-selling



Domain-specific GPU function management policy

#1 Fine-grained GPU allocation (e.g., minimum 1%vGPU)

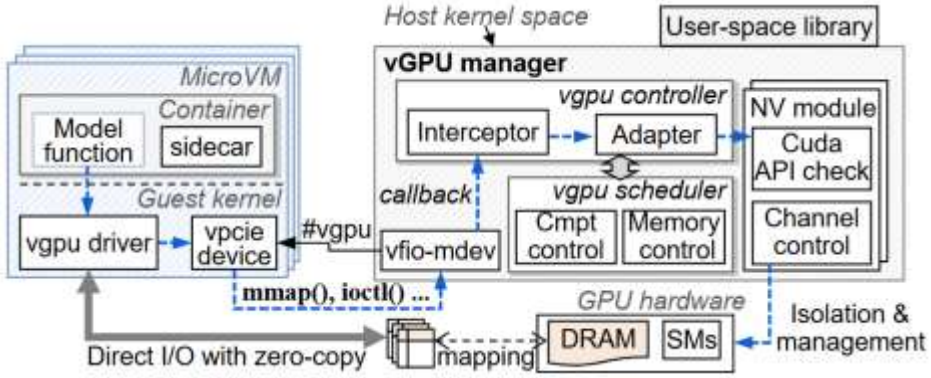
#2 Decoupled and flexible GPU recycling with vGPU hot-plugging and fast model swapping

#3 Profiling-based SLO-aware GPU shceduling & sharing

Challenge #1

- MPS can not be used in multi-tenant VM env.
- GPU virtualization is rarely studied before
- Nivida’s roprietary solution lacks flexibility

Our Solution:



Better compatibility than user-space API interception

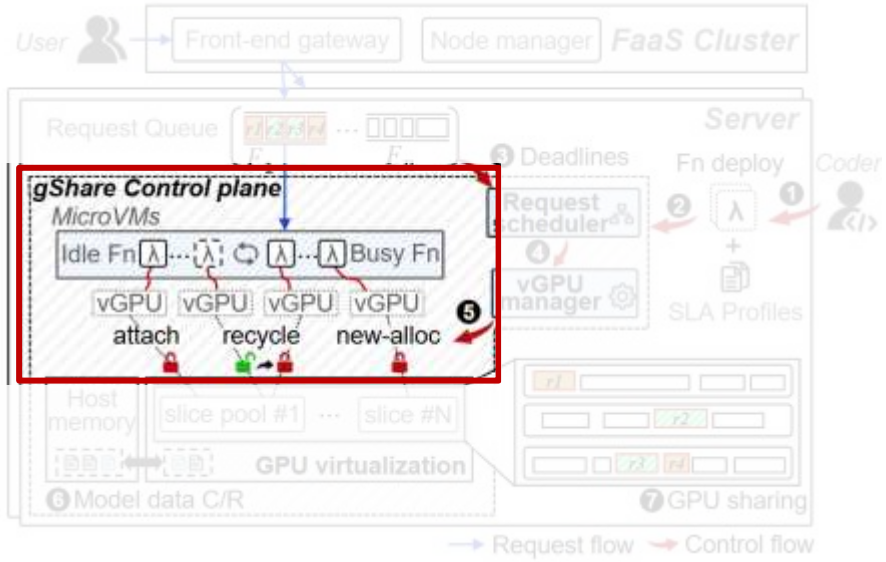
Natively direct I/O access reduces overhead

Arbitrary device memory and time slicing allocation

A complete vGPU implementation based on driver interception

Design of gShare

- Designed for Muti-tenant FaaS platform
- GPU sharing with GPU virtualization technology
- Discounted billing model for over-selling



Domain-specific GPU function management policy

#1 Fine-grained GPU allocation (e.g., minimum 1%vGPU)

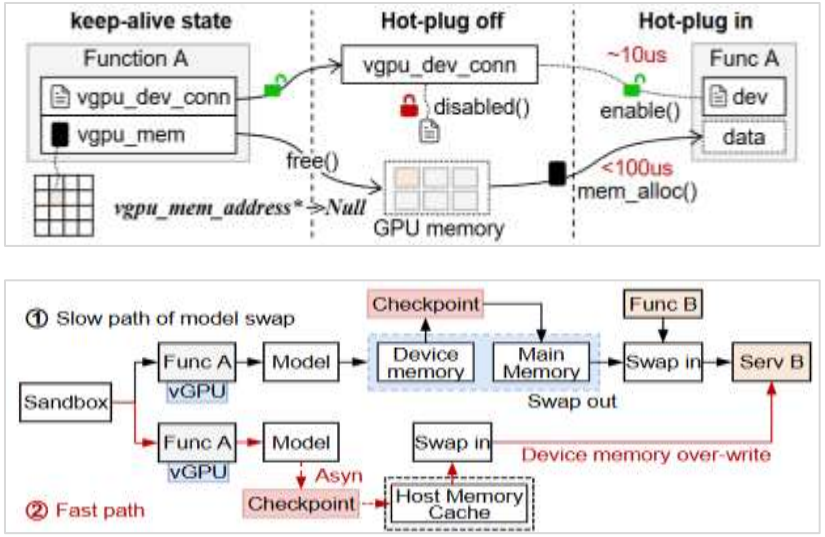
#2 Decoupled and flexible GPU recycling with vGPU hot-plugging and fast model swapping

#3 Profiling-based SLO-aware GPU shceduling & sharing

Challenge #2

- GPU recycling and data recovery incur extra lantecy
- Proxy-based GPU manager design is not necessary
- Limited memory bandwidth between host and device

Our Solution:



vGPU Hot-Plugging and fast model swap

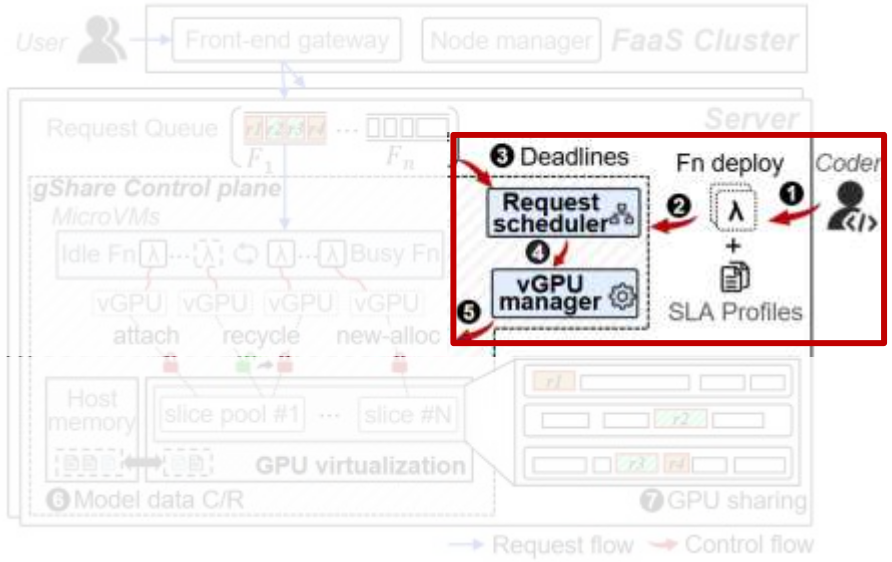
vGPU Hot-Plugging mechanism

Direct model data management inside function sidecar

Fast model sawp with checkpoint cache

Design of gShare

- Designed for Muti-tenant FaaS platform
- GPU sharing with GPU virtualization technology
- Discounted billing model for over-selling



Domain-specific GPU function management policy

#1 Fine-grained GPU allocation (e.g., minimum 1%vGPU)

#2 Decoupled and flexible GPU recycling with vGPU hot-plugging and fast model swapping

#3 Profiling-based SLO-aware GPU shceduling & sharing

Challenge #3

- Different user latency SLOs
- Diverse workload patterns and latency requirement
- The Optimal GPU allocation and request scheduling

Problem Formulation:

$$minimize: \sum_{j\in M} U_j \quad (1)$$

$$\sum_{j\in M,\delta\in t} a_{j,k}(\delta)\delta \geq \sum_{j\in M,\delta\in t} a_{j,k}(\delta)K_k, \quad \forall r_k \quad (2)$$

$$\sum_{j\in M,\delta\in t} a_{j,k}(\delta) = 1, \quad \forall r_k \quad (3)$$

$$\sum_{r_u,r_v\in R^t} s^j_{u,v} = \sum_{r_k,\delta\in t} a_{j,k}(\delta), \quad \forall j \quad (4)$$

$$s^j_{u,v} + s^j_{v,u} = 1, \quad \forall r_u, r_v, \forall j \quad (5)$$

$$\sum_{r_u\in R^t} s^j_{u,v} \leq 1, \quad \forall r_v, \forall j \quad (6)$$

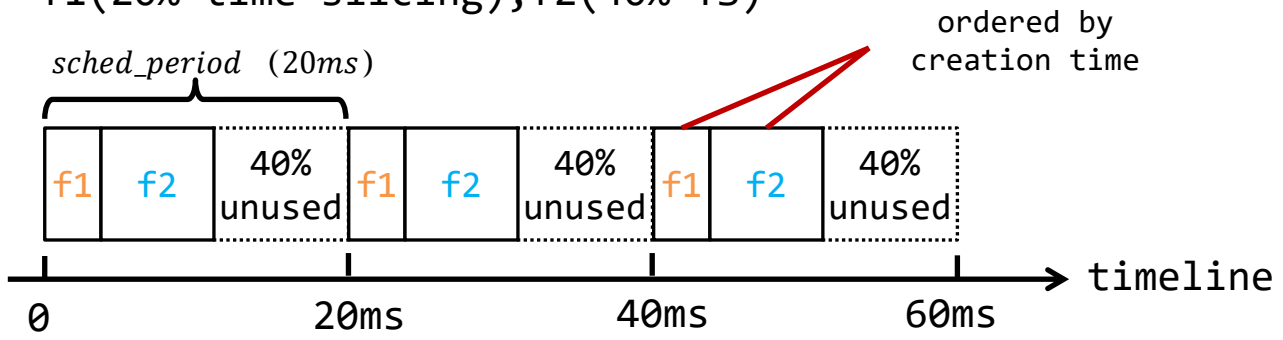
$$\sum_{r_v\in R^t} s^j_{u,v} \leq 1, \quad \forall r_u, \forall j \quad (7)$$

$$\sum_{\delta\in t} a_{j,v}(\delta)\delta \geq \sum_{\delta\in t} a_{j,u}(\delta)\delta + D_u - H(1 - s^j_{u,v}), \quad \forall r_u, r_v, \forall j \quad (8)$$

$$S_k\Delta t + I_k(1 - F_{k',k}) + D_k \leq \theta_k, \quad \forall r_k \quad (9)$$

$$0 \leq \sum_{j\in M,r_k} a_{j,k}(\delta) \leq M, \quad \forall \delta \in t \quad (10)$$

E.g., f1, f2 on GPU_0
f1(20% time slicing),f2(40% TS)



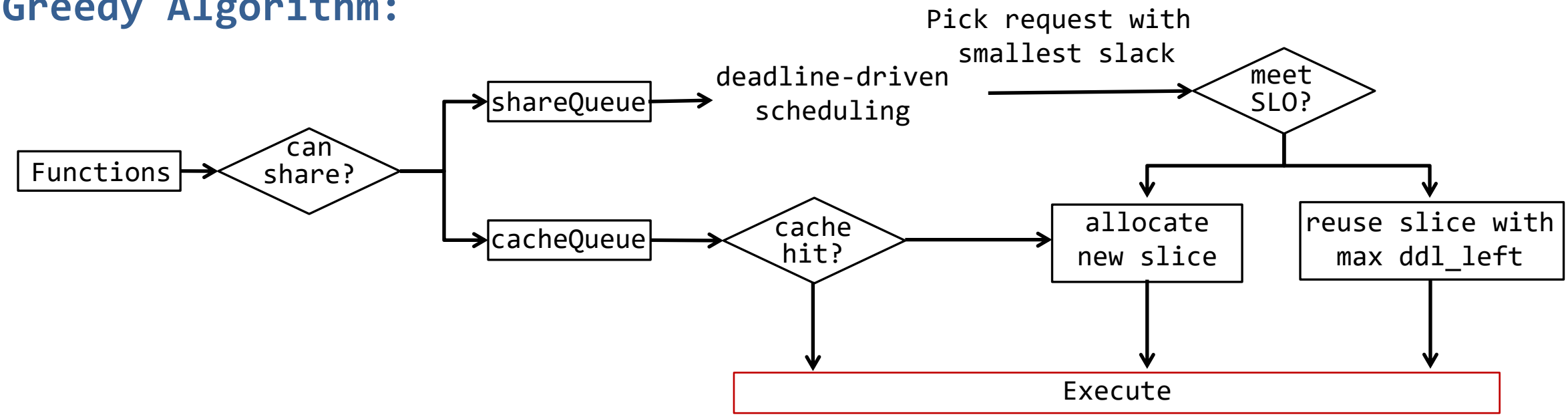
$$slice = func\_mem/dev\_cap * sched\_period$$

(1) Given a request queue and time slot of each model function
(2) Decide which GPU, which slot to execute request
(3) When f1 is idle, f3 can replace it to reduce cost
or creating f4(<60%) on GPU_0 after f2

Our Solution

- A hybrid scheduling policy (cacheQueue+slackQueue)
- Deadline-driven GPU allocation & request scheduling

Greedy Algorithm:



Implementation

- Kata-containers + NVIDIA GPU
- 60,000 lines of C code on vGPU design
- Rewritten a total of 57 ioctl interfaces (17 NV_ESC APIs and 40 UVM_APIs)
- Adaptation for AMD ROCm are in progress

Interfaces	Parameters
create_vgpu()	vgpu_config, vgpu_mem_address*, tenant_id
delete_vgpu()	vgpu_mem_address*
get_vgpu_status()	vgpu_mem_address*
set_vgpu_config()	vgpu_mem_address*, vgpu_config

A subset of the vGPU user-space library

Evaluation

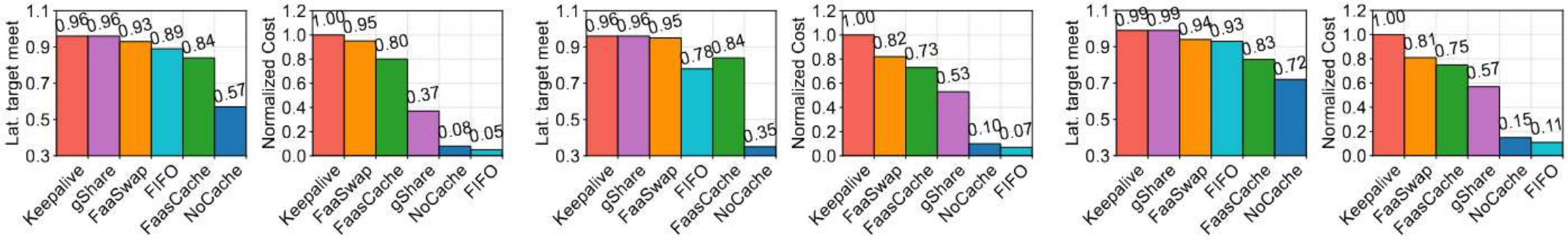
- A testbed comprising 20 physical servers
- NVIDIA A100 GPUs (9 GPU vGPU types ranging from 128MB-40 GB)
- MLPerf following 3 traces from three production clusters
- Latency target is defined as 1.5× of p90 execution time

Compared systems:

- Keeaplive policy (15-minute timeout setting in AWS Lambda)
- FaasCache[ASPLOS'21] (priority-based swap policy)
- FaaSwap[] (state-of-the-art method in GPU sharing)
- NoCache (aggressive scheduling strategy)
- FIFO (a homogeneous version of gShare)

Evaluation

Q-1: Can gShare achieve better cost-efficiency under different workloads?

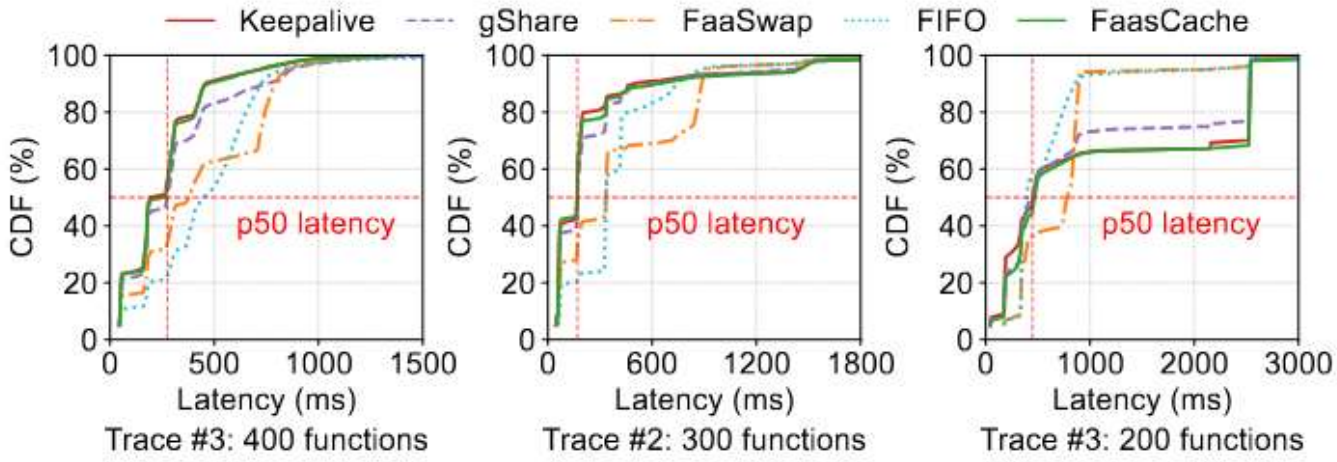


(a) Trace #1: 400 functions, QPS<5 (b) Trace #2: 300 functions ,QPS=9.6 (c) Trace #3: 200 functions, QPS>10

- gShare reduces GPU usage by 43%–63% while maintaining function performance comparable to baseline
- gShare delivers consistent performance improvements across diverse workload scenarios

Evaluation

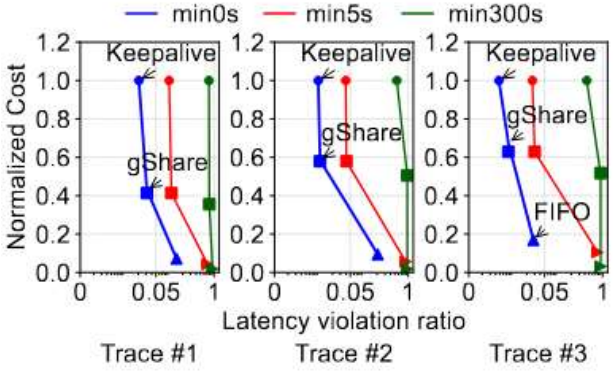
Q-2: How does gShare's GPU sharing mechanism affect request latency?

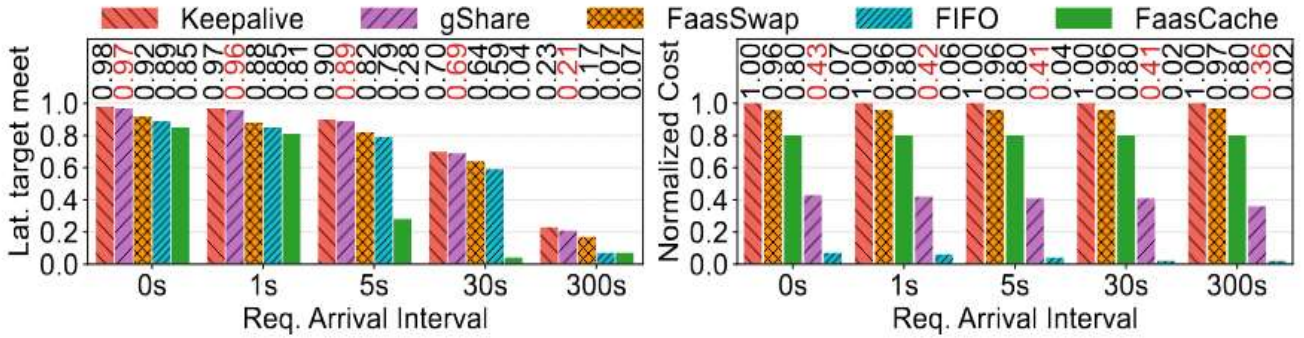


- The lazy scheduling mechanism in gShare may increase request waiting time, but only results in a 15% increase in p95 latency, with negligible degradation in p50 latency

Evaluation

Q-3: How does gShare perform under various function concurrency and latency constraints?

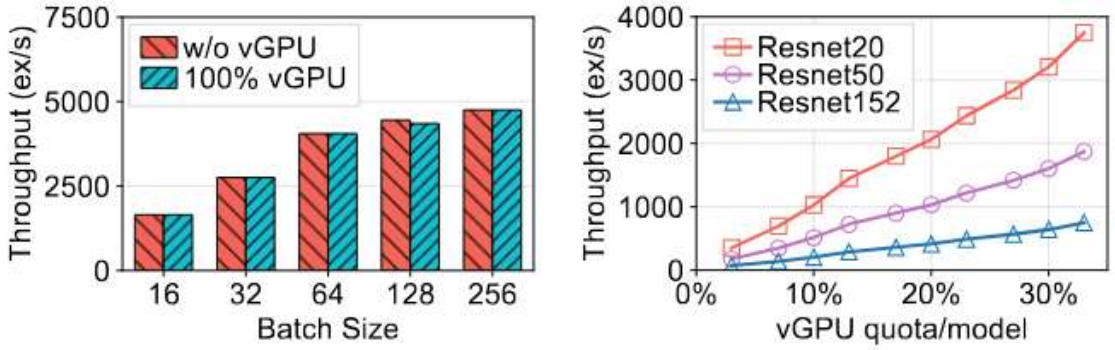




- gShare achieves Pareto optimality on cost-efficiency under different workload settings
- gShare performs best under sparse workloads with lenient latency targets

Evaluation

Q-4: What is the system overhead of gShare on vGPU allocation and reclamation?



(a) Performance loss of vGPUs (b) Resource isolation on vGPUs

- Share's vGPU incurs negligible performance degradation for microVM-based functions
- gShare ensures strong resource isolation across multiple tenant functions sharing the same GPU device

Evaluation

Q-5: how much economic benefit can gShare bring?

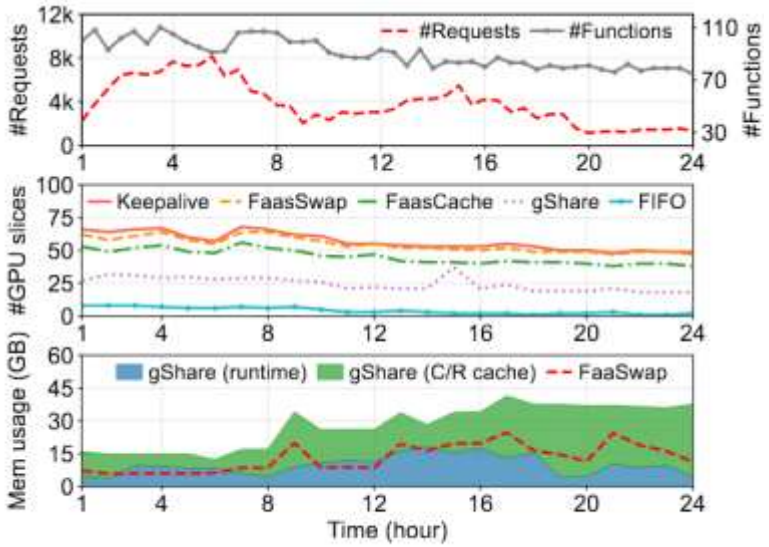


Figure 10. The cluster memory usage profiles.

Table 3. Price of GPU and memory in AWS and Aliyun.

	Types	GPU	$/Instance (per GPU)	Mem price
AWS	p4d.24xlarge	A100×8	$21.96/h ($2.74/h)	-/-
	c5.xlarge	-/-	-/-	$0.013/GB/h
Aliyun	gn8is-2x.8xlarge	L20×2	$4.53/h ($2.27/h)	-/-
	g7.xlarge	-/-	-/-	$0.0069/GB/h

- gShare achieves 75% of the optimal cost-efficiency but being 100× faster in practice
- gShare can help reduce GPU resource cost by hundreds of thousands of dollars every year in a typical FaaS cluster, benefiting both cloud provider and users

Conclusion

- We reveal the inefficiency of existing serverless GPU functions and **argue the decoupled GPU management** design
- **Small models are still important today!**
- We explore the optimum design space for GPU sharing and present its benefits

Table 4. Comparison of some representative related work.

Feature \ Method	MArk	Nexus	Clockwork	INFless	FaaSwap	gShare
GPU sharing	✗	✓	✗	✓	✓	✓
Fine-grained alloc.	✗	✗	✗	✓	✓	✓
vGPU support	✗	✗	✗	✗	✗	✓
SLO-aware scheduling	✓	✓	✓	✓	✗	✓
Model swap	✗	✗	✗	✗	✓	✓
FaaS scenario	✓	✗	✗	✓	✓	✓

Any Question can contact us

yangyn11@chinatelecom.cn

Thanks for listening!