TETRIS: Memory-efficient Serverless Inference through Tensor Sharing

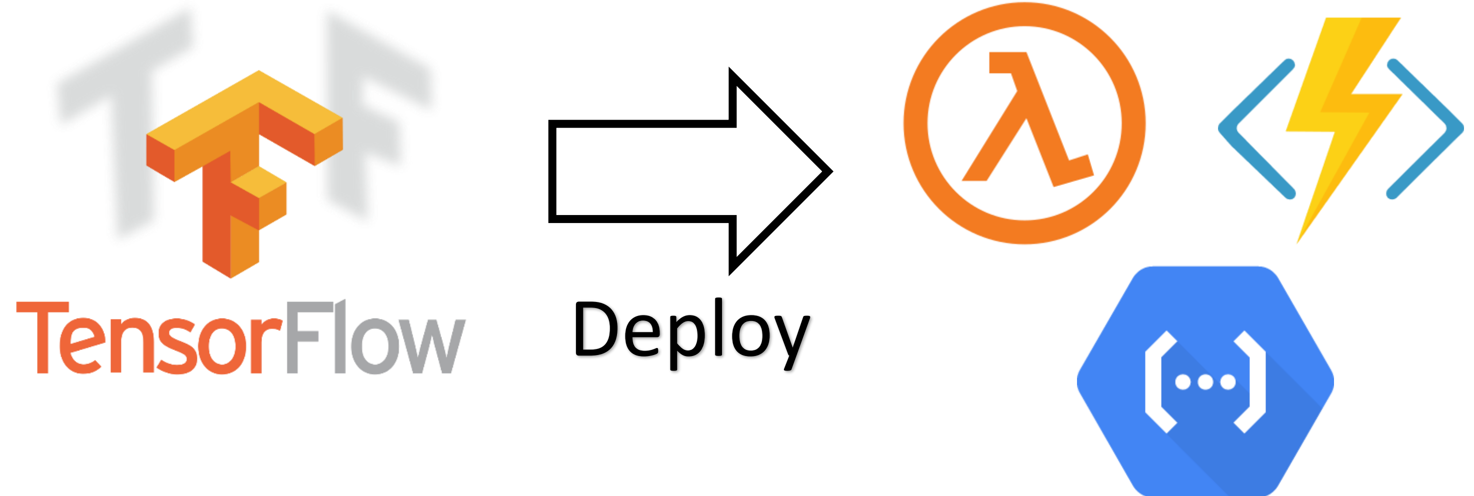
Jie Li[1], Laiping Zhao[1], Yanan Yang[1], Kunlin Zhan[2], Keqiu Li[1]

[1]Tianjin University, [2]58.com

Serverless Inference

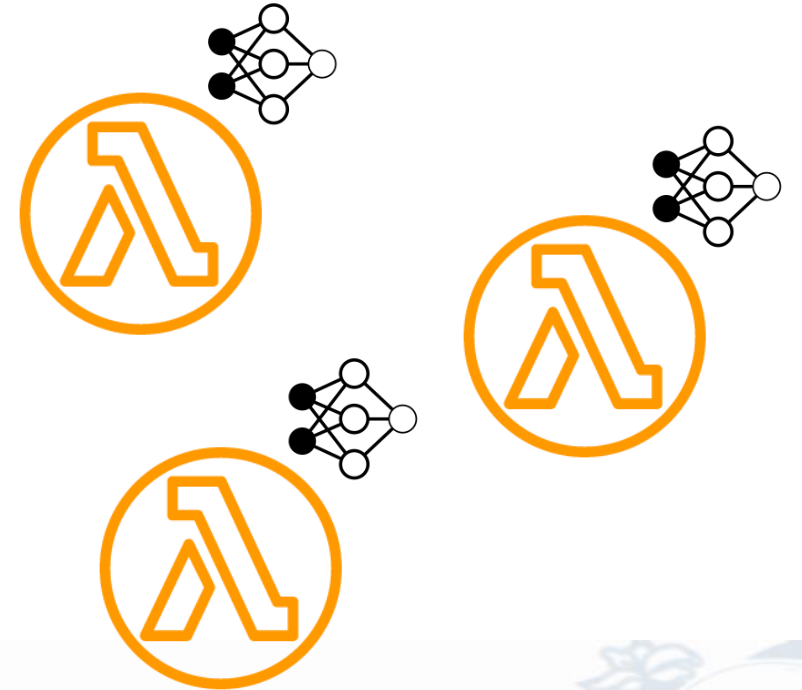
Benefits of Serverless Inference:

- Easy to use
- Cost effective
- Fast autoscaling



However, the current serverless inference platforms are highly memory inefficient!

Serverless Inference

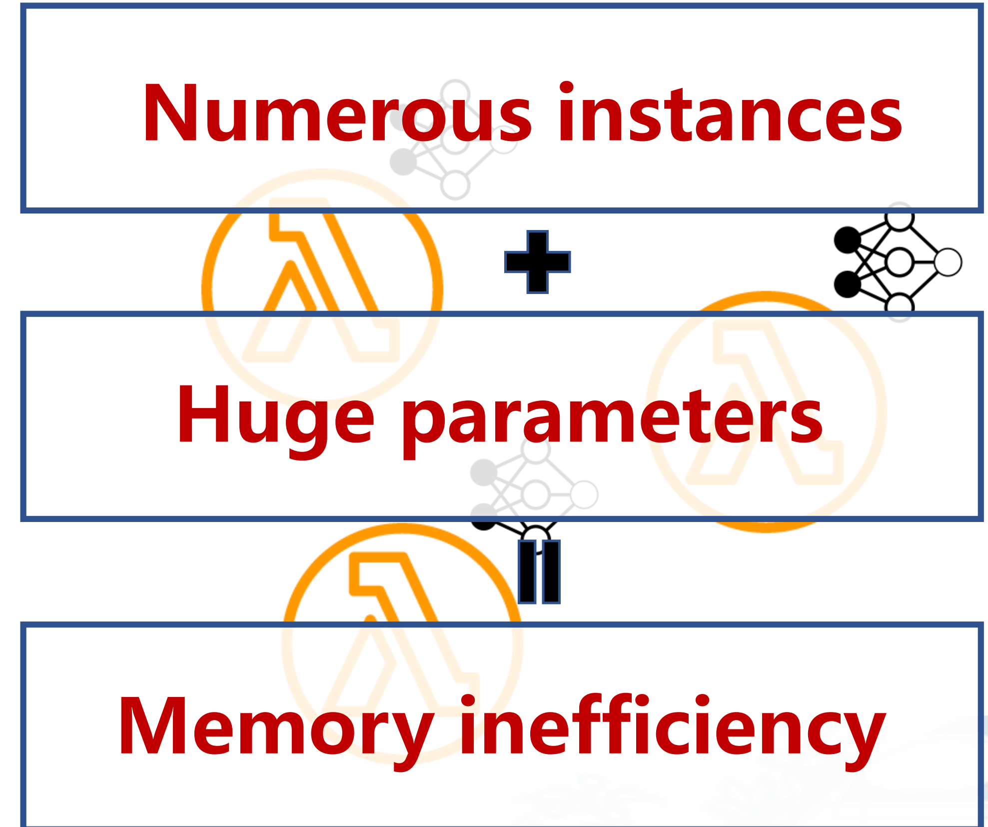


Inference requests

Spawned function instances

Serverless Inference

Inference requests



Spawned function instances

Serverless Inference

Drawback of Serverless Inference:

- Memory Inefficiency
  - High memory redundancy

The problem to be solved in this work

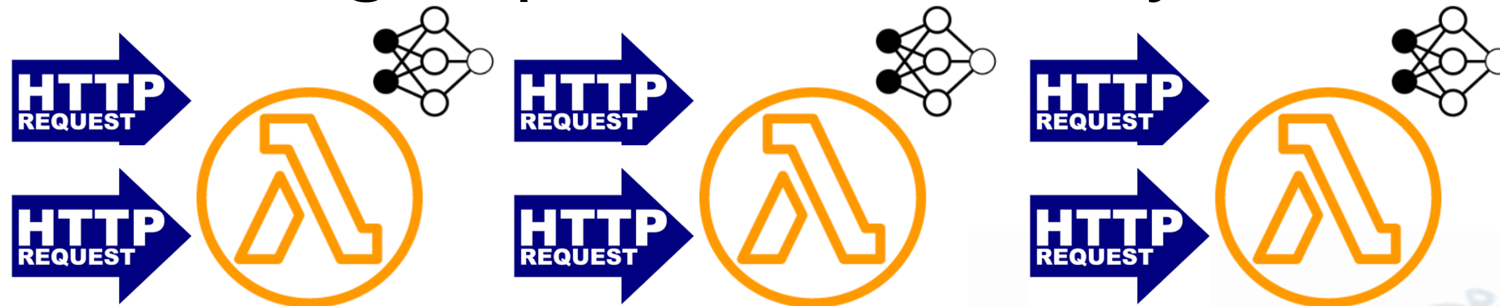
Causes:

- Multiple function instances
  - *One-to-one mapping policy* in AWS Lambda
- Early instance provisions
- Long keep-alive periods
  - 15-60 minutes in AWS Lambda

Existing approaches

Runtime Sharing:

- Processing multiple requests within a single instance
  - Batching
    - Grouping and processing requests in batch
  - Multi-threading
    - Processing requests concurrently

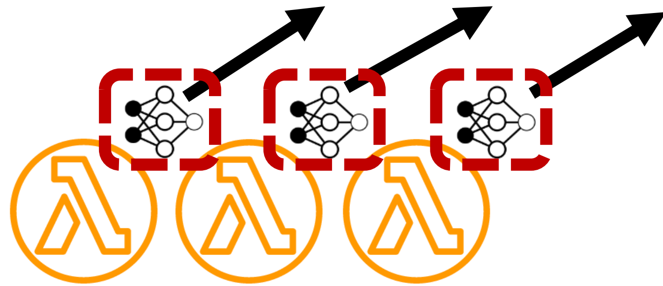


The runtime sharing methods reduced memory redundancy by decreasing the number of launched instances

Our contributions

Key observation:

- **Tensor redundancy**



Parameterized tensors

The parameterized tensors are loaded into memory repeatedly across function instances

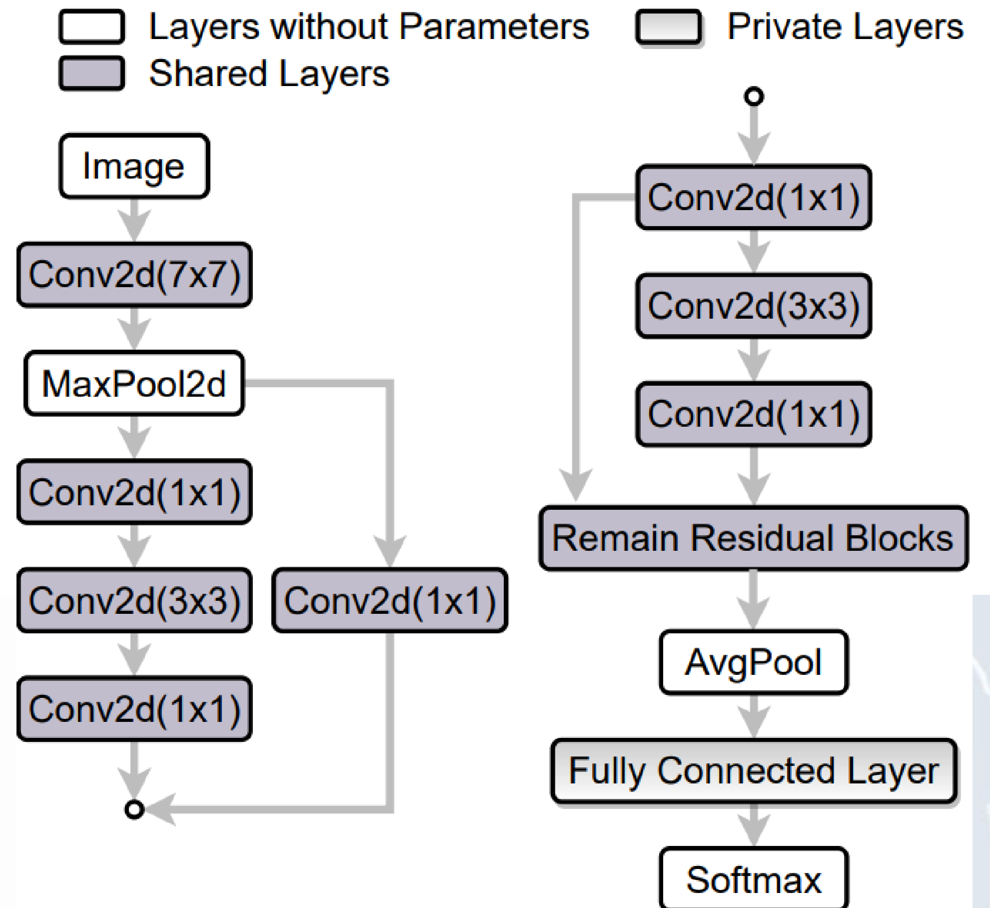
Our contributions

Key observation:

- **Tensor redundancy**

Tensor redundancy exists across distinct functions:

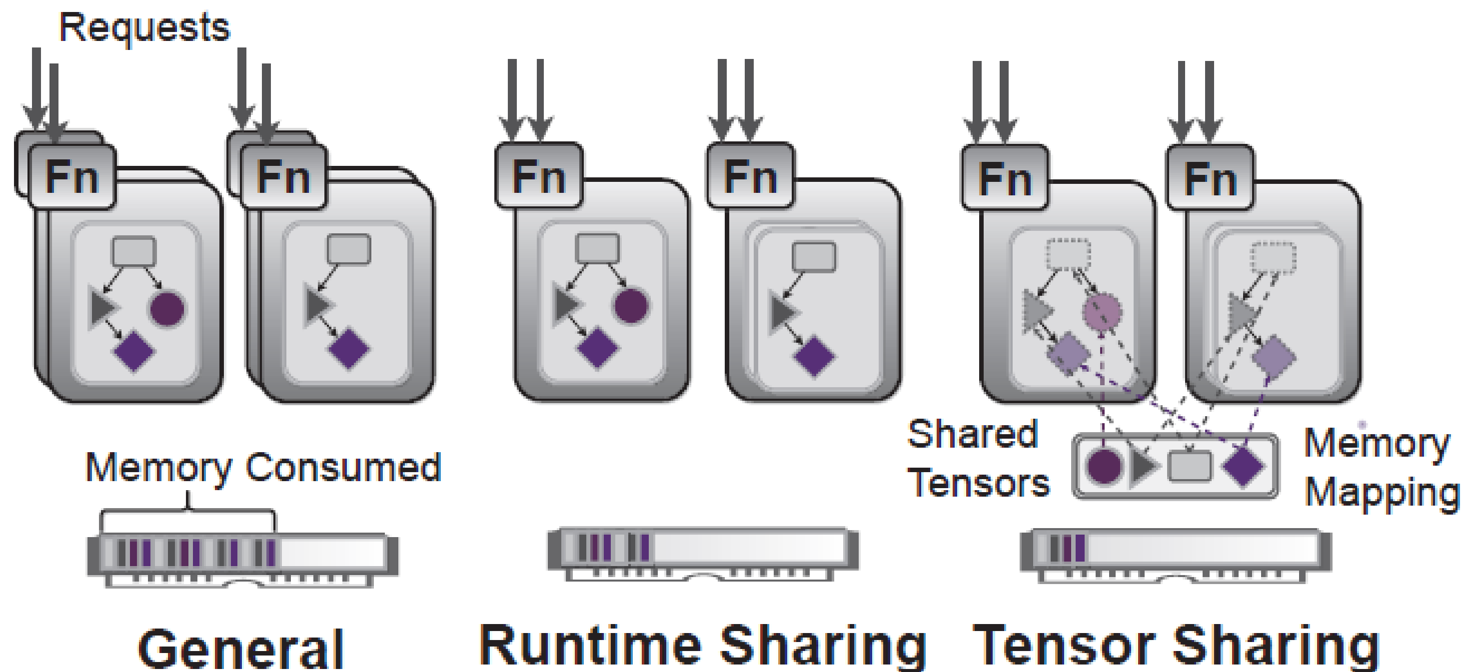
- The same model used in distinct model pipelines
- Different downstream models retrained from the same pre-trained parameters



Our contributions

Summarize:

- An observation of the **tensor redundancy** problem
- An lightweight and user-space solution that eliminates the tensor redundancy through **tensor sharing**



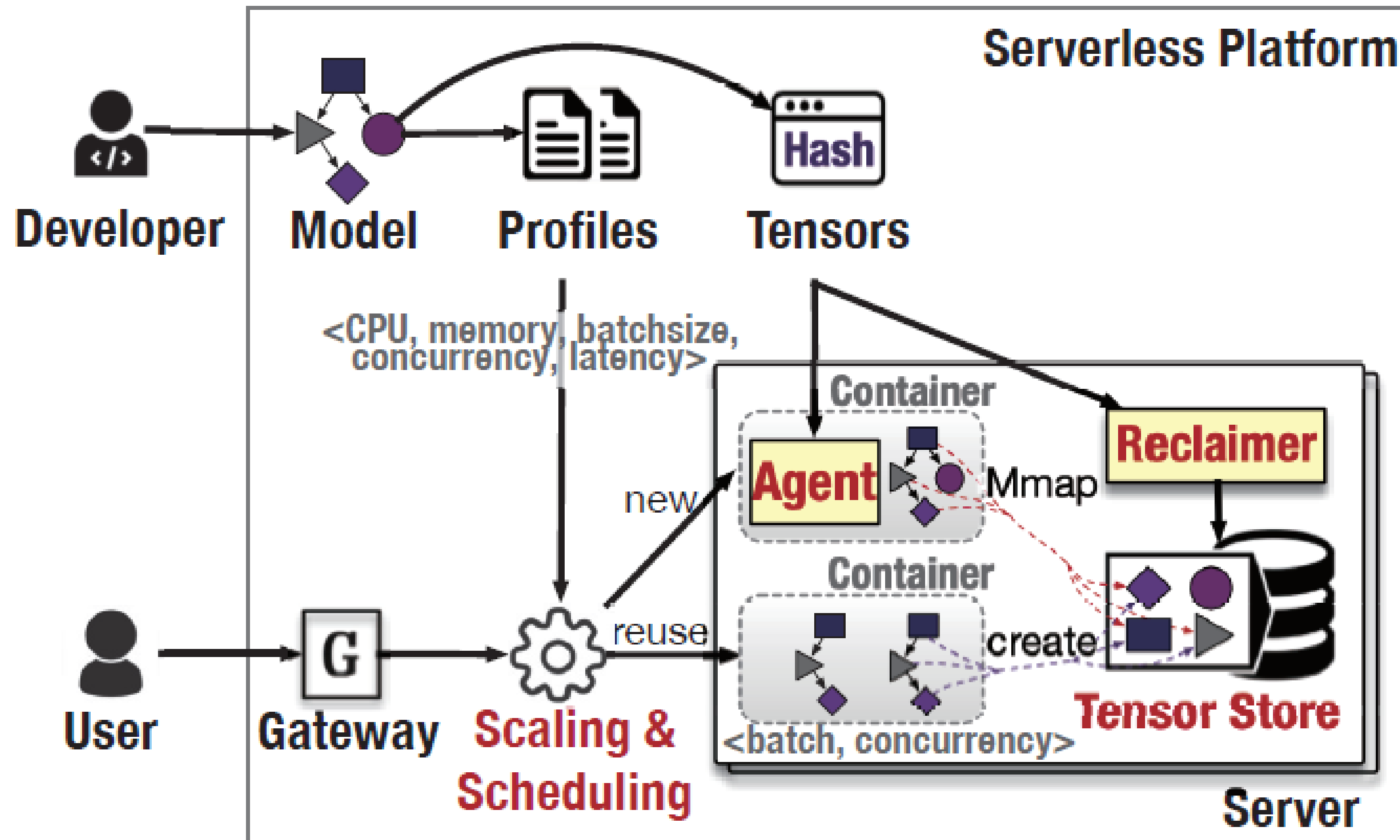
Design of TETRIS

Overview:

- TETRIS improves memory efficiency can be improved through a combined optimization of **runtime sharing** and **tensor sharing**

Design of TETRIS

Overview:



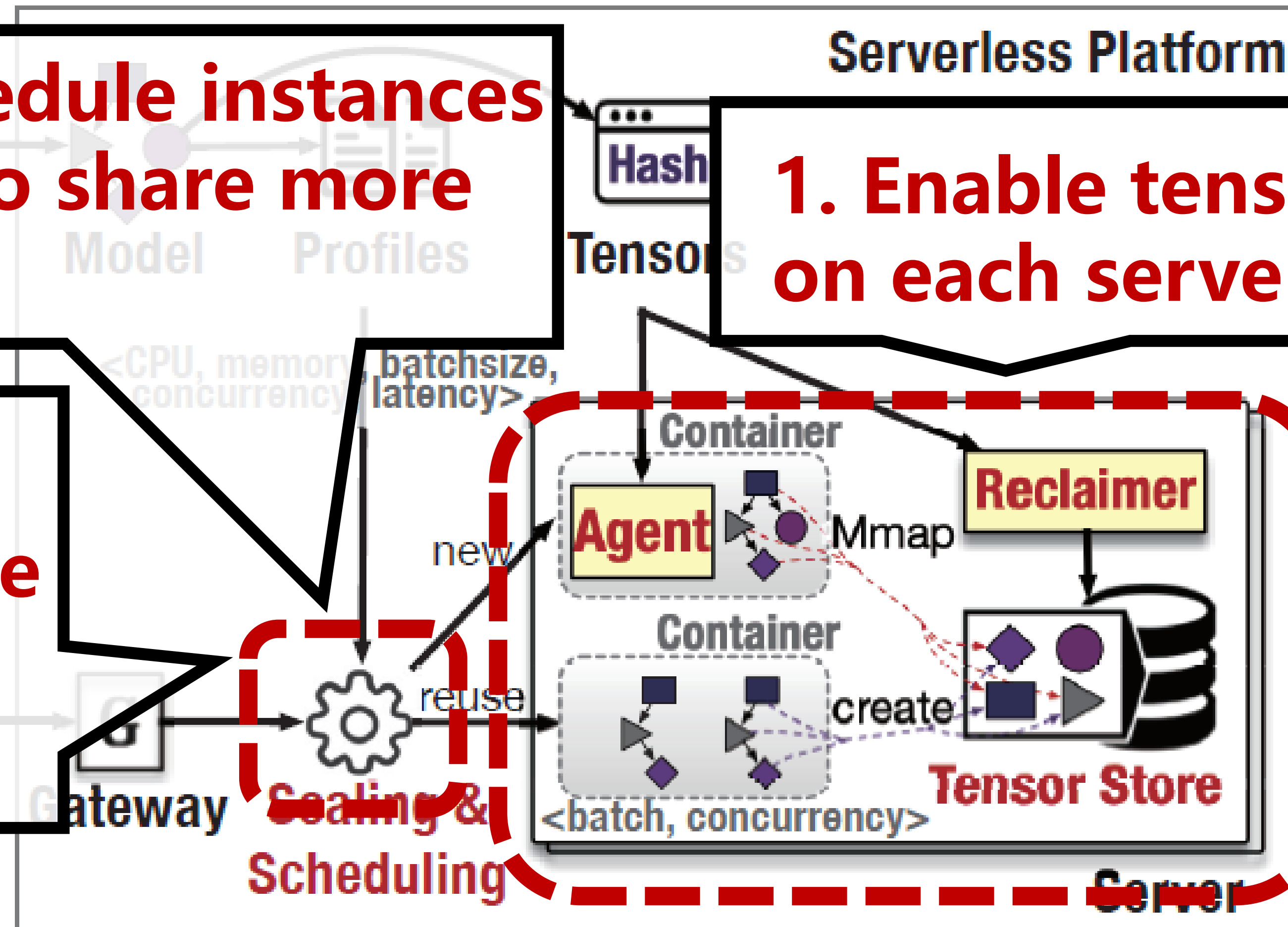
Design of TETRIS

Overview:

2. Carefully schedule instances across servers to share more tensors

1. Enable tensor sharing on each server

3. Scaling fewer instances to serve requests under SLO constrains



Design of TETRIS

How to share tensors of function instances on the same server?

- First, make a shared memory region across function instances **(The Shared Tensor Store)**

  (implemented by mounting a shared tmpfs to each container)
- Second, take over the model loading process of function instances and put tensors into the shared region **(The Agent)**
- Third, make tensors in the shared region to be reclaimed correctly **(The Reclaimer)**

Design of TETRIS

How to share tensors of function instances on the same server?

- How does the Agent load tensors?
  - <u>Create a new memory region if the tensor has never been loaded</u>
  - <u>Mmapping existing memory region if the tensor has already been loaded</u>

Tensors are identified by hash values

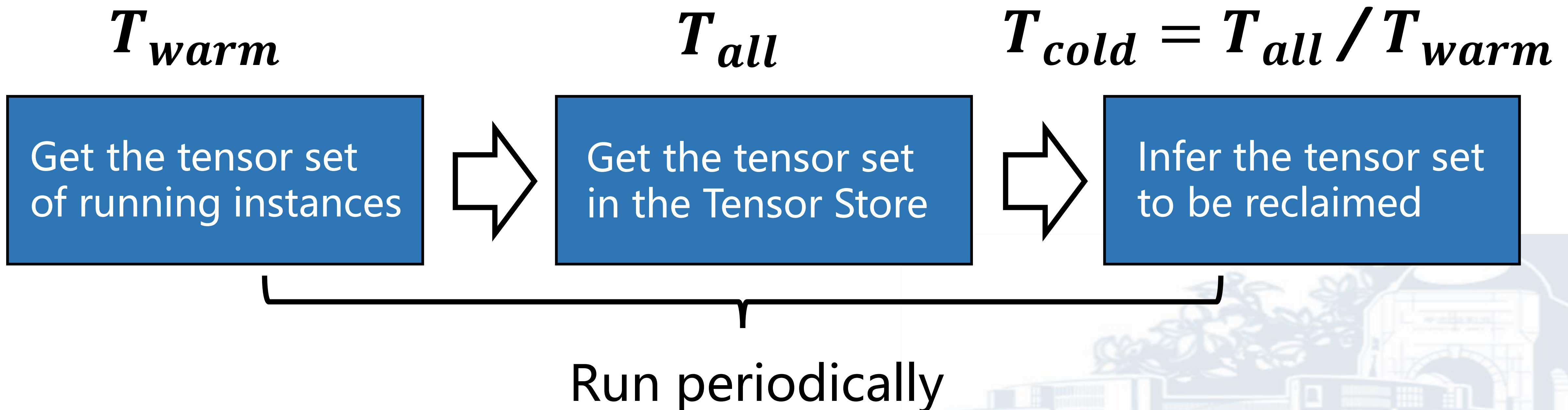
```
Status LoadTensor(Tensor& tensor,TensorReader& reader) {
    // Get tensor hash value.
    std::string tensor_hash = GetHash(reader,tensor);
    // Get or create tensor lock in
    // Shared Tensor Store atomically.
    TensorLock lock = CreateOrGetTensorLock(tensor_hash);
    // Obtain ownership of a tensor lock.
    lock.Lock();
    // Check if the tensor in Shared
    // Tensor Store already exists.
    if(!TensorExists(tensor_hash)) {
        // Allocate the tensor memory in
        // Shared Tensor Store and load
        // the model parameters.
        CreateTensor(reader,tensor,tensor_hash);
    } else {
        // Mapping already existing tensor
        // memory from the Shared Tensor Store.
        MmapTensor(tensor,tensor_hash);

    // Release the lock.
    lock.Unlock();
    return Status::OK();
}
```

Design of TETRIS

How to share tensors of function instances on the same server?

- How does the Reclaimer detect and reclaim unreferenced tensors?

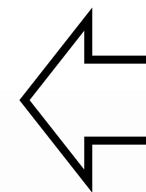


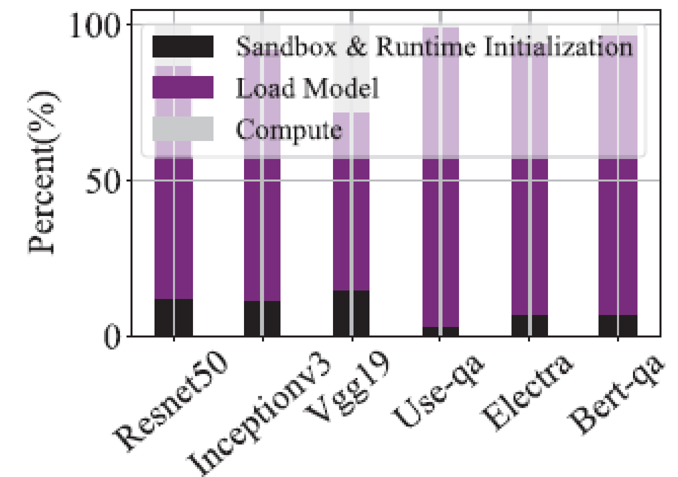
Design of TETRIS

How to share tensors of function instances on the same server?

- How does the Reclaimer detect and reclaim unreferenced tensors?
  - Unreferenced tensors can be **kept alive** to accelerate function instance startups

The loading of massive model parameters dominates the startup process of function instances



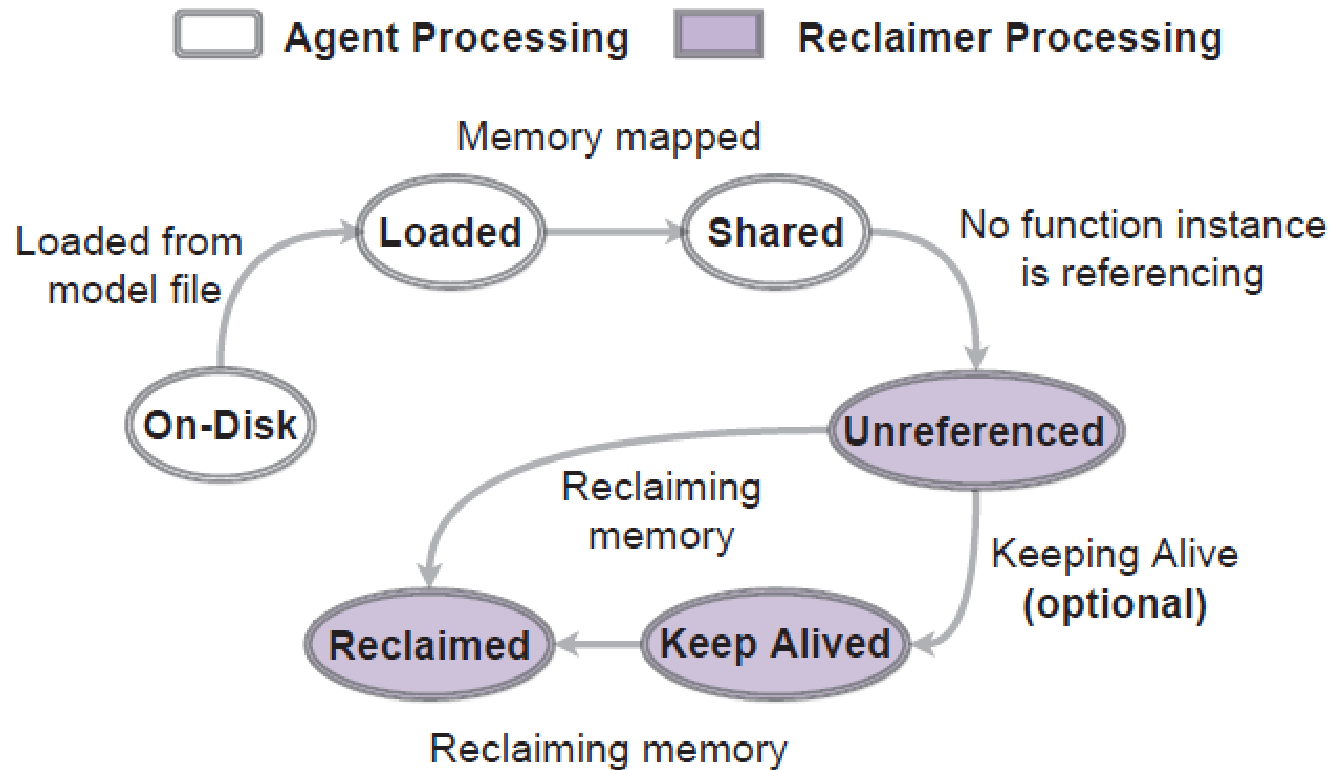


(a) Request processing time

Design of TETRIS

How to share tensors of function instances on the same server?

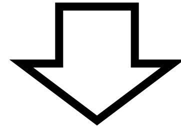
- The lifecycle of tensors



Design of TETRIS

How to share tensors of function instances across different servers?

- TETRIS does NOT support remote sharing
- TETRIS minimizes cluster memory consumption **through instance scheduling**



Greedy by the tensor similarity between instance $\boldsymbol{i}$ and server $\boldsymbol{j}$:

$$\Theta_{ij} = \frac{\boldsymbol{Mem}\left(\boldsymbol{T_i} \cap \boldsymbol{T^j_{store}}\right)}{\boldsymbol{Mem}(\boldsymbol{T_i})}$$

Design of TETRIS

How to share function instance runtimes under SLO constraints?

- Profile inference latency under various combinations of <CPU, memory, batch size, concurrency>
- Model the instance scaling process as an optimization problem

Different combinations of batch size and concurrency configurations lead to different memory efficiency

(a) DenseNet169

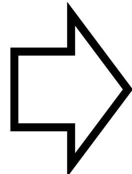
(b) Lstm

Design of TETRIS

How to share function instance runtimes under SLO constraints?

- Model the instance scaling process as an optimization problem

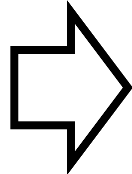
$$minimize: \quad \sum_{i=1} m_i x_i$$



Subject to minimize the memory consumption

$$l_i \le t_{slo}, \ \forall i \wedge x_i \ge 1 \wedge b_i = 1$$

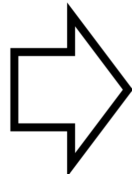
$$l_i \le t_{slo}/2, \ \forall i \wedge x_i \ge 1 \wedge b_i > 1$$



The SLO constrains

$$\sum_{i=1}^{n} x_i b_i p_i / l_i \ge R, \ \forall i$$

$$x_i \in \mathbb{N}$$



Ensure that the residual RPS can be fully processed by the newly spawned instances.

Design of TETRIS

How to share function instance runtimes under SLO constraints?

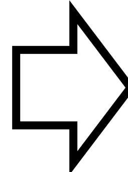
- Model the instance scaling process as an optimization problem

$$\text{minimize}: \sum_{i=1} m_i x_i$$

$$\sum_{i=1}^{n} x_i b_i p_i / l_i \geq R, \ \forall i$$

$$x_i \in \mathbb{N}$$

However, this problem is NP-Complete



A simple greedy solution:

- Greedily select configuration i with maximum $\frac{throughput_i}{memory_i}$ or $\frac{throughput_i}{memory_i + \alpha CPU_i}$

(To balance the CPU consumption)

Evaluation

DL Model	Size	Description	Download times
Bit-M [37]	3.5GB	Feature vector extraction	1.4k
[illegible]	1.8GB	Sentence Embedding	24.9K
Bert-qa [14]	1.3GB	Question Answering	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
Use [7]	980MB	Sentence Encoder	1.4M
Centernet [75]	731MB	Object Detection	12.8K
Use-qa [72]	568MB	Question Answering	16.3K
Use-large [7]	563MB	Sentence Encoder	1.1M
[illegible]	549MB	Image Classification	commercial
Bert [14]	392MB	Text Processing	197.5K
[illegible] [68]	255MB	Image Processing	2.2K
Resnet152 [26]	231MB	Image Processing	1.6K
InceptionResnetV2 [66]	214MB	Image Processing	6K
Stt [69]	176MB	Speech-To-Text	398
[illegible]	171MB	Image Processing	1.6K
Fastspeech2 [57]	119MB	Text-To-Speech	310
[illegible]	[illegible]	Image Processing	11.6K
SSD [43]	29MB	Object Detection	commercial
Dssm [29]	25MB	Text Processing	commercial
[illegible]	23MB	Text Processing	commercial
Textcnn69 [9]	11MB	Text Processing	commercial

Inference models

- 21 inference models collected from TF-Hub and 58.com

Model sizes

- 11MB to 3.5GB

Download times

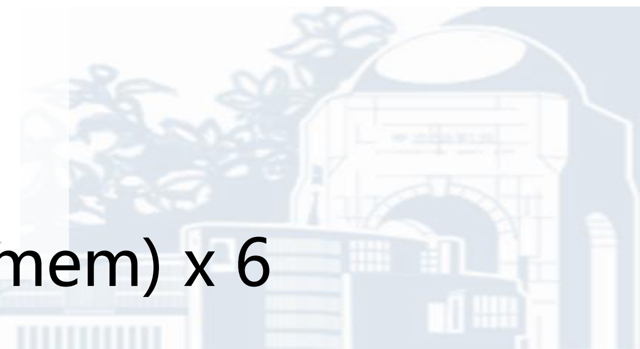
- 310 to 1.1M

Application domains

- Text, image, audio, etc

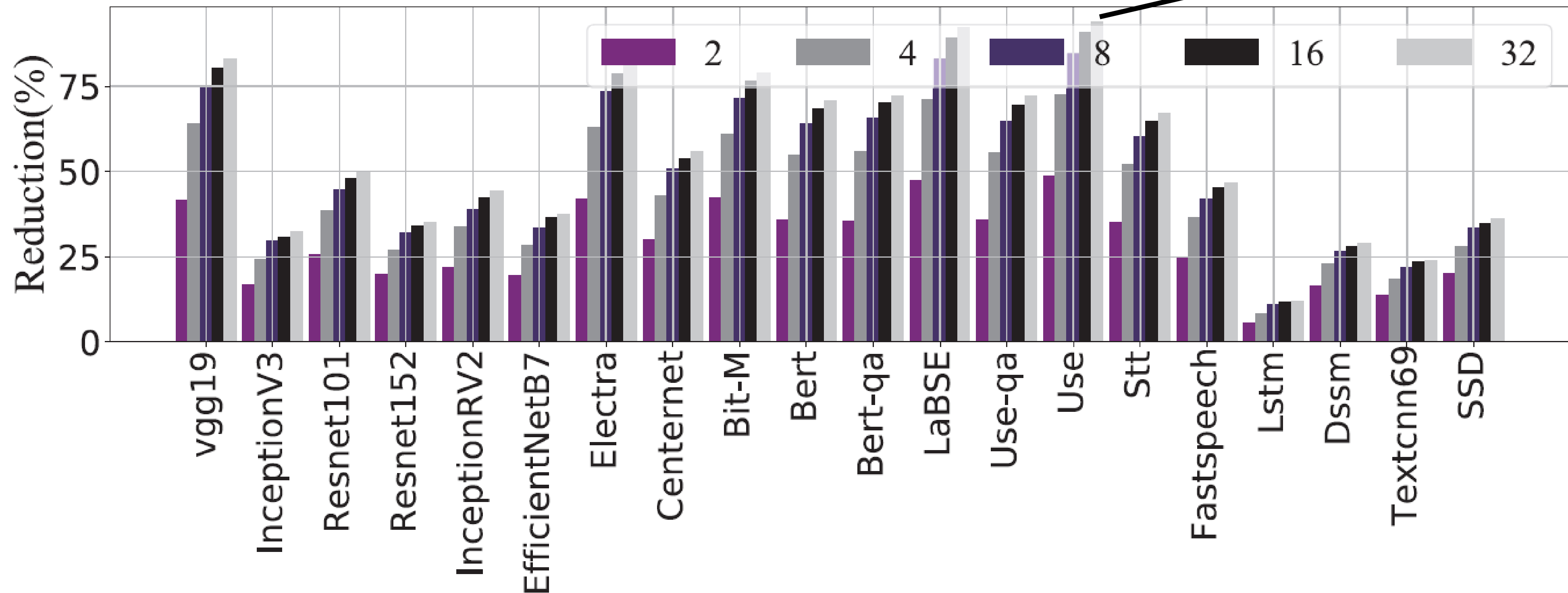
Testbed

- 8-server cluster
  - (80-vCPUs 256GB-mem) x 2, (32-vCPUs 128GB-mem) x 6



Evaluation

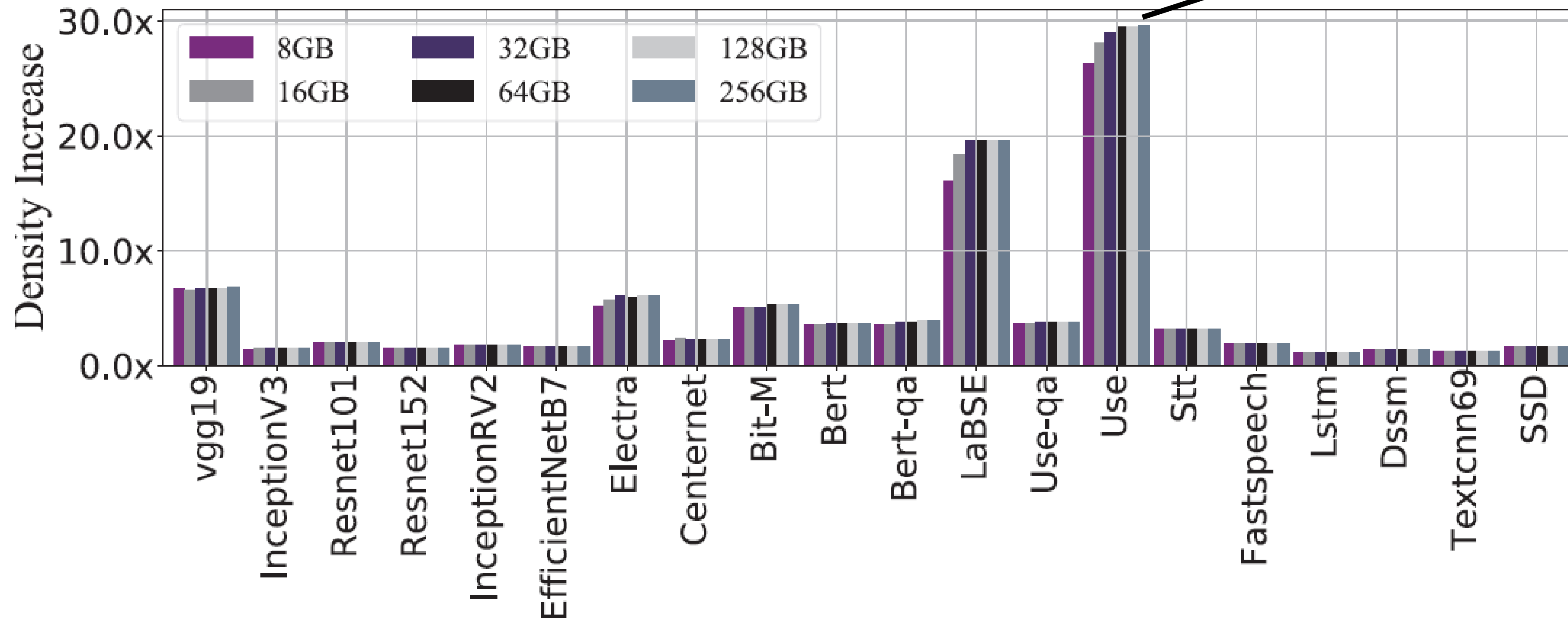
With tensor sharing, the memory consumption can be saved by up to **93%**



Memory reduction under different number of function instances

Evaluation

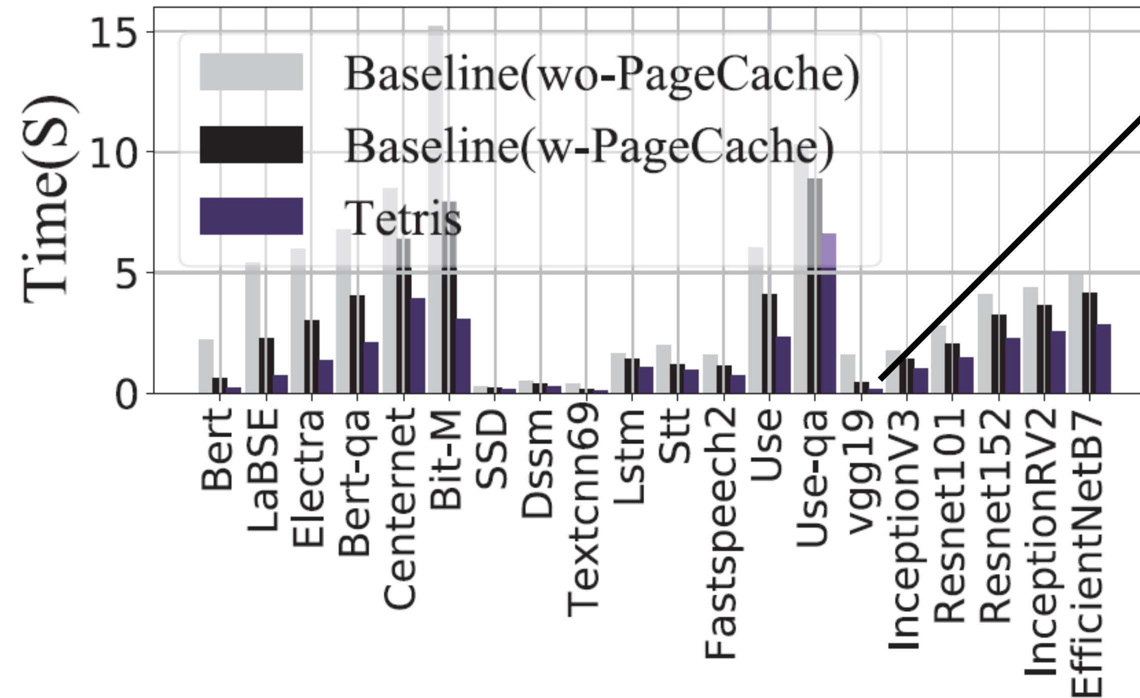
With tensor sharing, the function density can be improved by up to **30x**



Function density improvement under various machine memory capacities

Evaluation

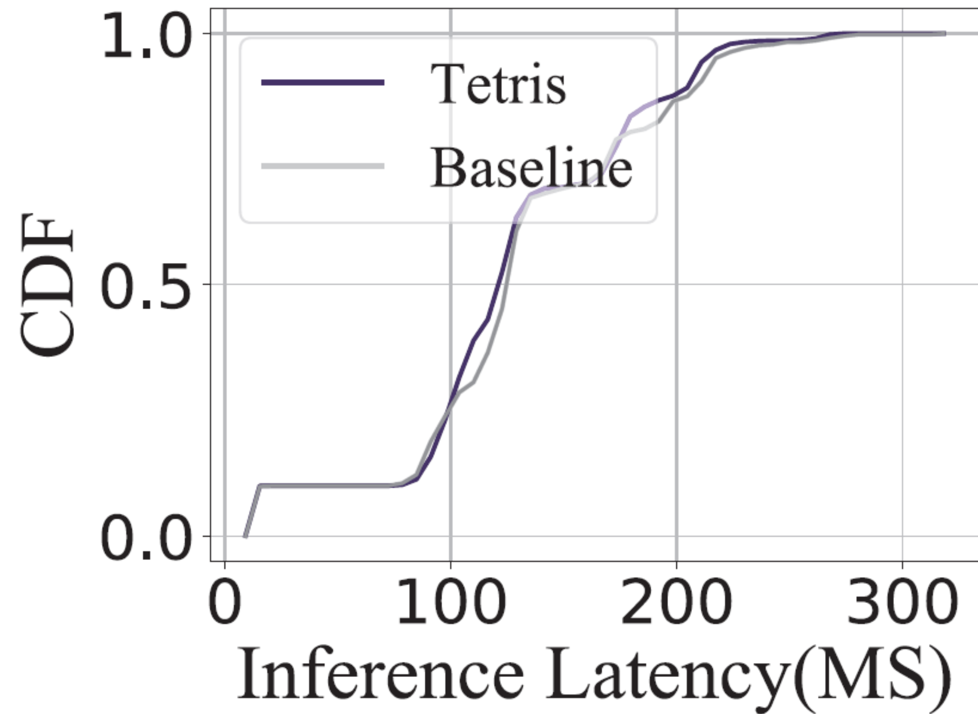
With tensor sharing, the function startup can be accelerated by up to **91.56%**



Startup time w/wo tensor sharing

Evaluation

The tensor sharing method does **NOT** introduce latency overhead



Inference time w/wo tensor sharing

Evaluation

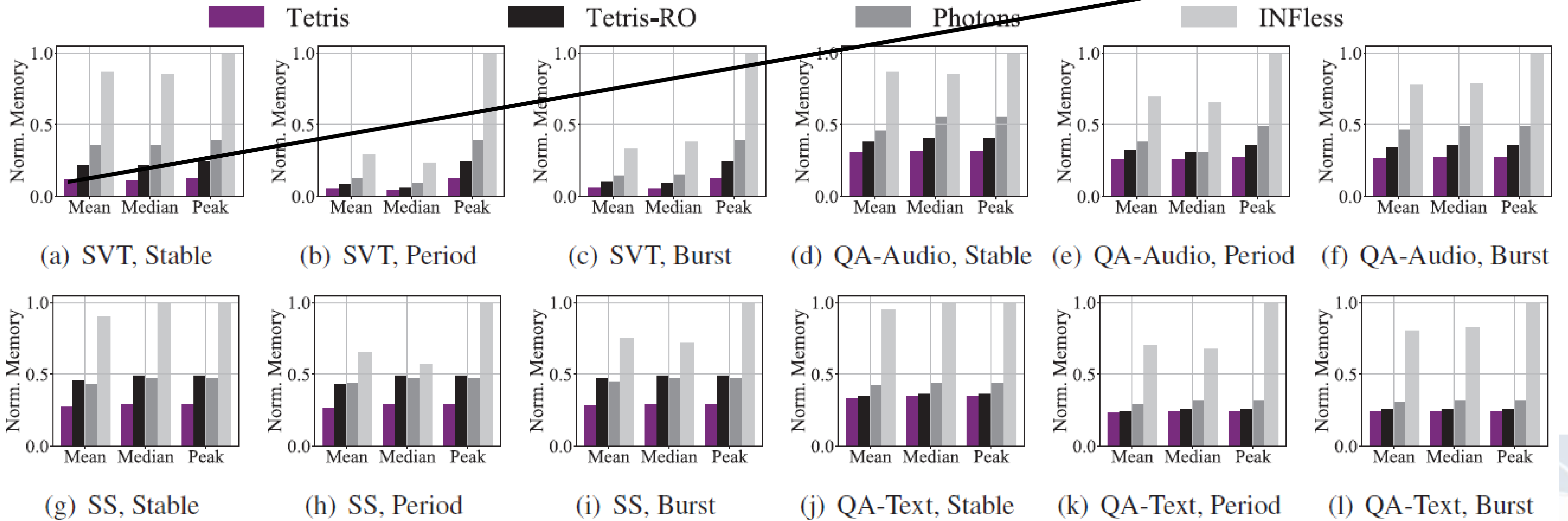
More experimental settings

- **4** real-world applications
- **3** real-world workload traces (from Azure)
- Comparison systems:

System	Runtime Sharing	Tensor Sharing
Tetris	**Combined**	**yes**
Tetris-RO	Combined	no
INFless	Batching	no
Photons (modified)	Multi-threading	no

Evaluation

Overall, TETRIS can reduce the mean memory footprint by more than **86%**



Normalized memory consumption by **four applications** under **stable**, **period** and **bursty** workloads

Conclusion

Benefits of TETRIS:

- Memory efficient
- No-harming performance
- Low overhead
  - Easy to implement
  - User transparent
  - No modification to ML models

Thank You！
Q & A