TETRIS: Memory-efficient Serverless Inference through Tensor Sharing

(W2) Paper Reading

Nan Lin

Shanghai Jiao Tong University

2024-07-05

Outline

Basic Info

- Authors:
  - Jie Li, Laiping Zhao, Yanan Yang (Tianjin University, TANKLAB)
  - Kunlin Zhan (58.com)
  - Keqiu Li (Tianjin University, TANKLAB)
- Published: *2022 USENIX Annual Technical Conference*

Outline

Revision: Serverless computing

- **Cloud computing**
- **Serverless computing**: Hiding infrastructures and platforms from customers.
- Key concepts:
  - **Function**: Basic element
  - **FaaS**: **Function** as a service
  - **BaaS**: **Backend** as a service

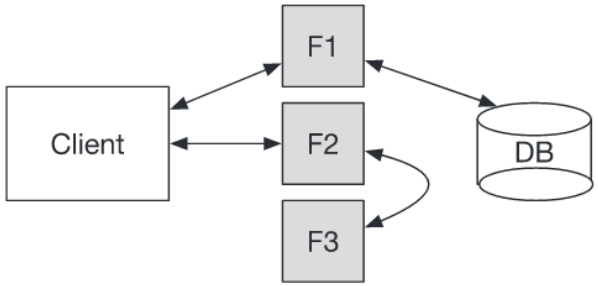
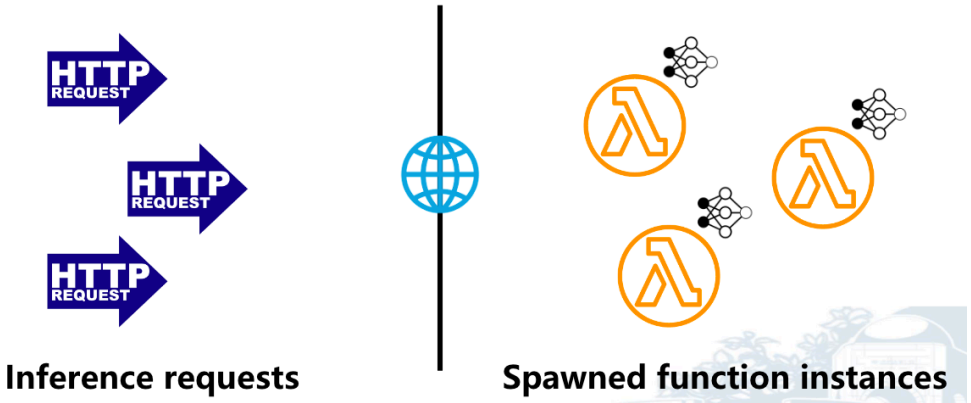


Fig. 1. An example of a serverless application.

Revision: Serverless inference

- **Inference**: Deep Learning (DL)
- **Serverless inference**: Deploying DL inference service atop a serverless platform.
- Benefits:
  - Easy to use
  - Cost effective
  - Fast autoscaling



Outline

Structure of this part:

- Drawbacks of serverless inference
- Observations
- Conclusion

Drawbacks of serverless inference

Current serverless platform - AWS Lambda

- Cannot support large models (limit 10 GB $\ll$ MT-NLG language model 2 TB)
- *one-to-one mapping policy*
- Caching: Functions are *short-lived* while memory is not released due to preservation or caching.

Reduce the memory footprint of DL inference service!

Their observations (5 in total)

Observation 1: Time consumption

- The function startup time is significantly longer than the inference computation time.
- Merely considering the startup, the majority of time is spent on loading model parameters. (populates the variables from a serialized model file to the computational graph)

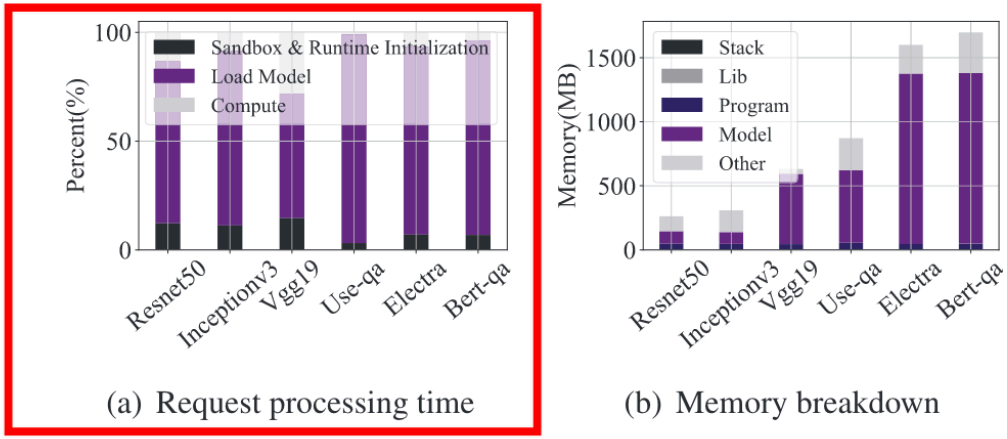
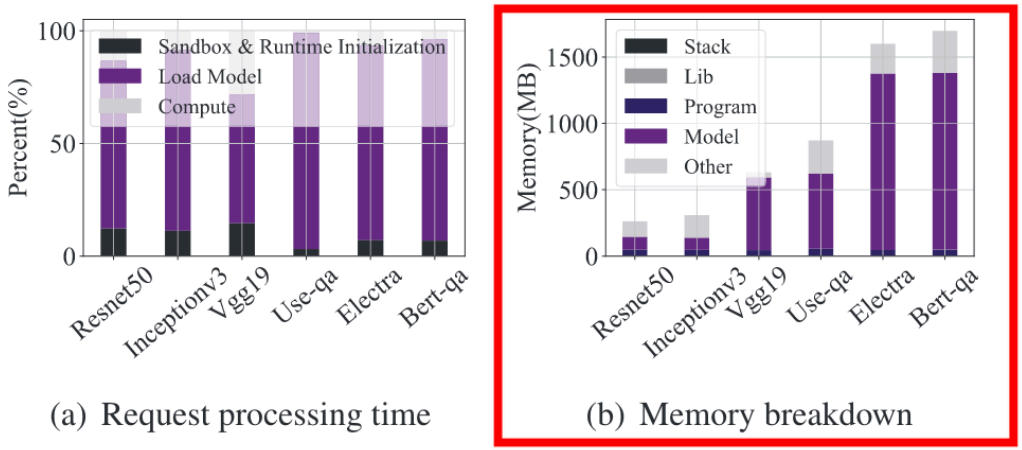


Figure 2: Request processing time and memory breakdown for various inference models.

Their observations (5 in total)

Observation 2: Memory consumption

- The model parameters occupy a major portion of all memory consumption
- Size of text processing model > others (e.g. image, audio, video processing model)



(a) Request processing time (b) Memory breakdown

Figure 2: Request processing time and memory breakdown for various inference models.

Their observations (5 in total)

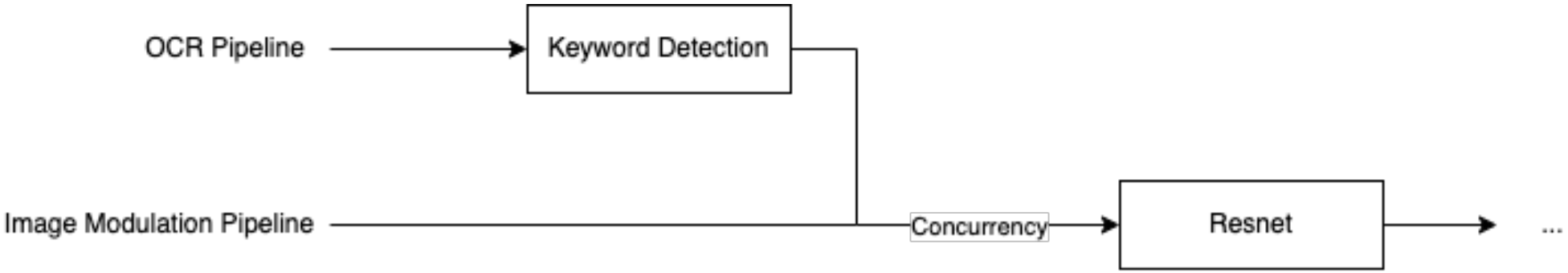
Observation 3: Runtime redundancy

- Multilaunched instances lead to duplicated runtime memory consumption.
- *Solution*: Pack multiple requests by **batching** and **concurrent execution**.
- **Batching**: Increase computing load
- **Concurrent execution**: Increase resource contention among threads

Their observations (5 in total)

Observation 4: Tensor redundancy (extensive replication of parameters)

- Online web services have strict latency requirements (< 100 ms), avoid excessive network communications
- Model reused directly: different demands (contexts) use highly similar models.



Their observations (5 in total)

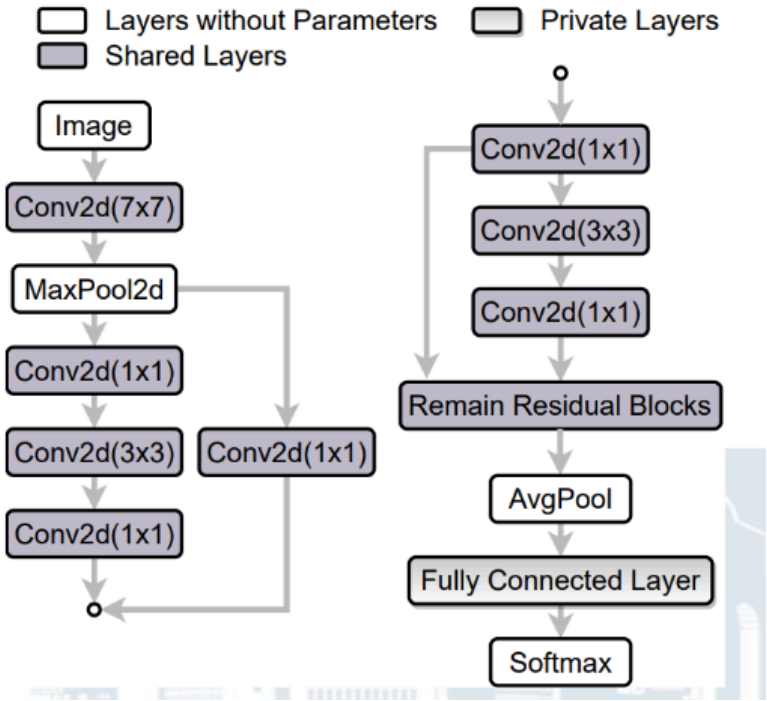
Transfer learning: Pretrain with massive datasets, in order to be reused in related tasks.

Benefits:

- Enable rapid development
- Avoid overfitting, improves the accuracy

Their observations (5 in total)

Problem: Which layer(s) should be reused?



Their observations (5 in total)

Problem: Which layer(s) should be reused?

Table 1: Tensor redundancy at *58.com*.

Application Domains	Image	Text	Audio
Representative Models	Resnet50 EfficientNet MobileNet	Bert Roberta Albert GPT	VGGish
Applications	Advertising Housing Secondhand Trading	Reading Extraction Text Summary Text Classification	Audio Classification
Redundant Part	Bottom Layers	Embeddings Middle Layers	Embeddings
Redundancy Ratio	>70%	>31%	>90%

Their observations (5 in total)

Observation 5: Cache redundancy

- Existence of cache exacerbate the problem of runtime & tensor redundancy.
- Sources of cache:
  - Function kept alive after the execution has finished
  - Inherent in DL frameworks (to accelerate computing)

Conclusion

Conclusion:

- Memory-intensive when startup and computing (**Observation 1** and **Observation 2**)
- Thus, we need to reduce the memory footprint
- Redundancy exists mainly in three parts:
  - Runtime redundancy (**Observation 3**)
  - Tensor redundancy (**Observation 4**)
  - Cache redundancy (**Observation 5**)

Outline

The advantage of their work

TETRIS ...

- provides a complete solution for the memory bottleneck problem.
- is highly efficient and supports to guarantee the Service Level Objectives (SLOs).
- is a prototype system built on (open-source) OpenFaaS and TensorFlow Serving.

Overview of TETRIS

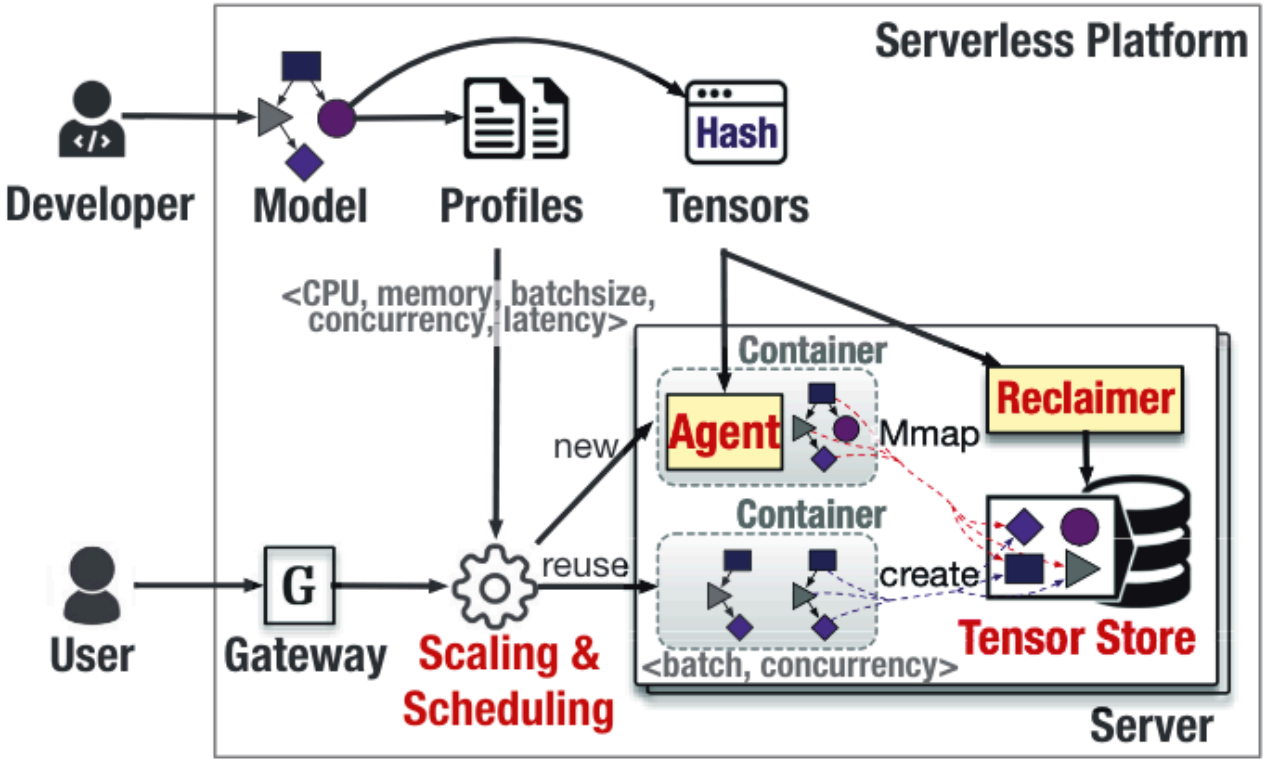


Figure 6: An overview of TETRIS.

Basic Structures

- **Tensor store**: shared memory region across function instances
- **Agent**:
  - Take over the model loading process of function instances
  - Put tensors into the shared region
- **Reclaimer**: ensure that the shared tensors are reclaimed correctly

Scaling & Scheduling

1. **Profiling**: measures the inference latency of each model.
- (c, m, b, p, l): allocated_cpus, memory_config, max_batch_size, num_threads, inference_latency
- computes and store the hashes of tensors

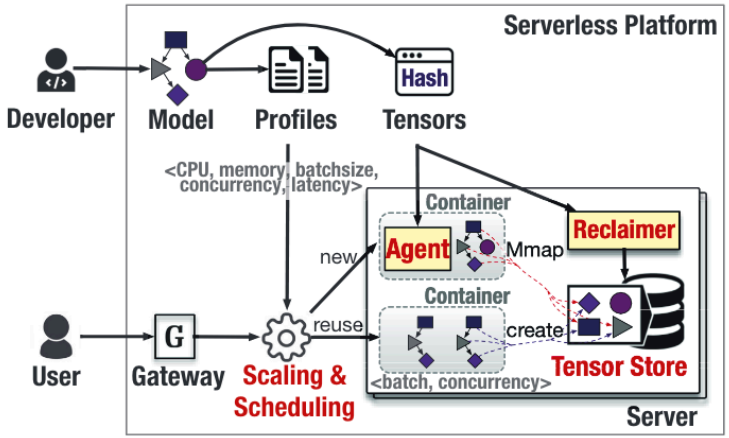


Figure 6: An overview of TETRIS.

Scaling & Scheduling

2. **Scaling**: monitor real-time RPS, judge whether current instances are sufficient to satisfy the requests.

- minimize memory while satisfing latency SLO
- Reminder: (c, m, b, p, l), n configurations for each x_n instances, numerated $[1, ..., n]$ ($\forall i \in \mathbb{N}, x_n \in \mathbb{N}$)
- R is the *Request Per Second (RPS)*

$$\min \quad \sum_{i=1}^{n} m_i x_i \quad \text{(allocated memory)}$$

$$l_i \leq t_{\text{slo}} \quad \forall i, \forall x_i \geq 1, \forall b_i \geq 1 \quad \text{(SLO requirement)}$$

$$\sum_{i=1}^{n} x_i b_i p_i / l_i \geq R \quad \forall i \quad \text{(RPS requirement)}$$

Scaling & Scheduling

2. **Scaling**: monitor real-time RPS, judge whether current instances are sufficient to satisfy the requests.

Problem: NP-complete solution

Alternative solution:

- Descending order of throughput
- Select config with higher normalized throughput

Scaling & Scheduling

3. **Scheduling**: Dispatches the instances with maximum *tensor similarity* θ

- First filter out servers that do not satisfy the requirements
- Select the optimized value of θ_{ij} with instance i, server j

$$\theta_{ij} = \frac{\text{Mem}\left(T_i \cap T_{\text{store}}^j\right)}{\text{Mem}\left(T_i\right)}$$

1. First-time sharing of tensors

First-time sharing of tensors through **agent**. Agent parses the computational graph and read the hash values of the tensors. Check if the **tensor store** has already stored it:

- If so, using syscall or Mmap
- Create a new memory region

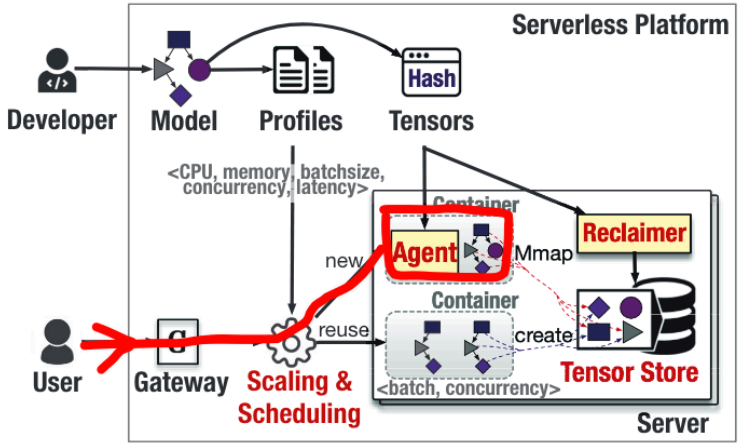


Figure 6: An overview of TETRIS.

2. Releasing the tensors

Each tensor is labeled with a *reference number*. (referenced by other instances?)

- When the agent add a new mapping to an existing tensor, the *reference number* is added by 1
- When an instance is released after completion, the *reference number* is decreased by 1
- The tensor memory is released when the *reference number* is 0

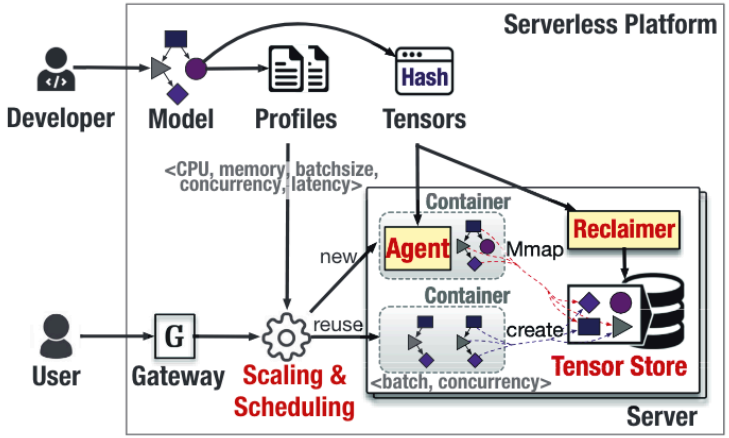


Figure 6: An overview of TETRIS.

Summary: Lifecycle of a tensor

Loading a tensor

```
1  Status LoadTensor(Tensor& tensor,TensorReader& reader) {
2      // Get tensor hash value.
3      std::string tensor_hash = GetHash(reader,tensor);
4      // Get or create tensor lock in
5      // Shared Tensor Store atomically.
6      TensorLock lock = CreateOrGetTensorLock(tensor_hash);
7      // Obtain ownership of a tensor lock.
8      lock.Lock();
9      // Check if the tensor in Shared
10     // Tensor Store already exists.
11     if(!TensorExists(tensor_hash)) {
12         // Allocate the tensor memory in
13         // Shared Tensor Store and load
14         // the model parameters.
15         CreateTensor(reader,tensor,tensor_hash);
16     } else {
17         // Mapping already existing tensor
18         // memory from the Shared Tensor Store.
19         MmapTensor(tensor,tensor_hash);
20     }
21     // Release the lock.
22     lock.Unlock();
23     return Status::OK();
24 }
```

Listing 1: Simplified code snippet for loading tensors.

Summary: Lifecycle of a tensor

Lifecycle of a tensor

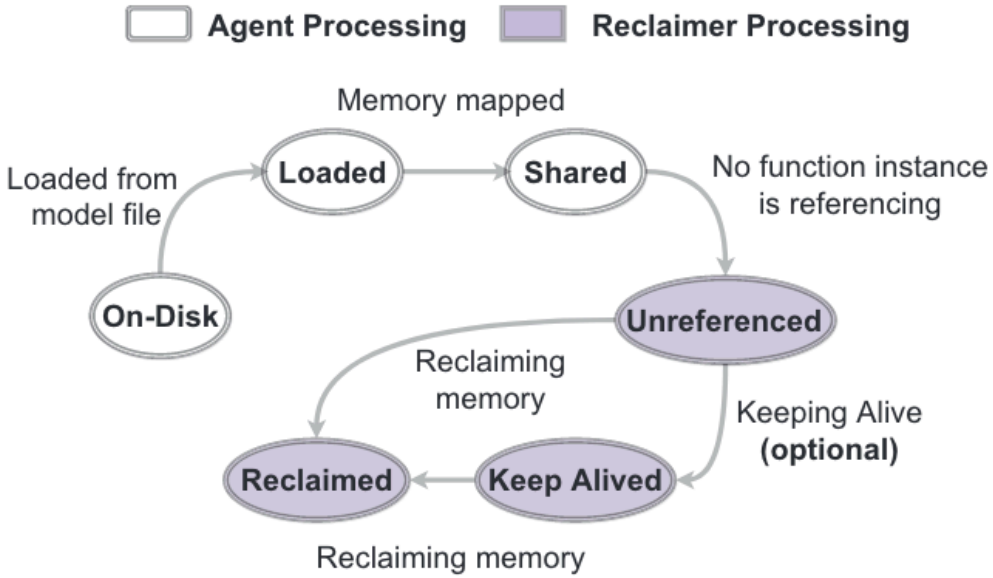
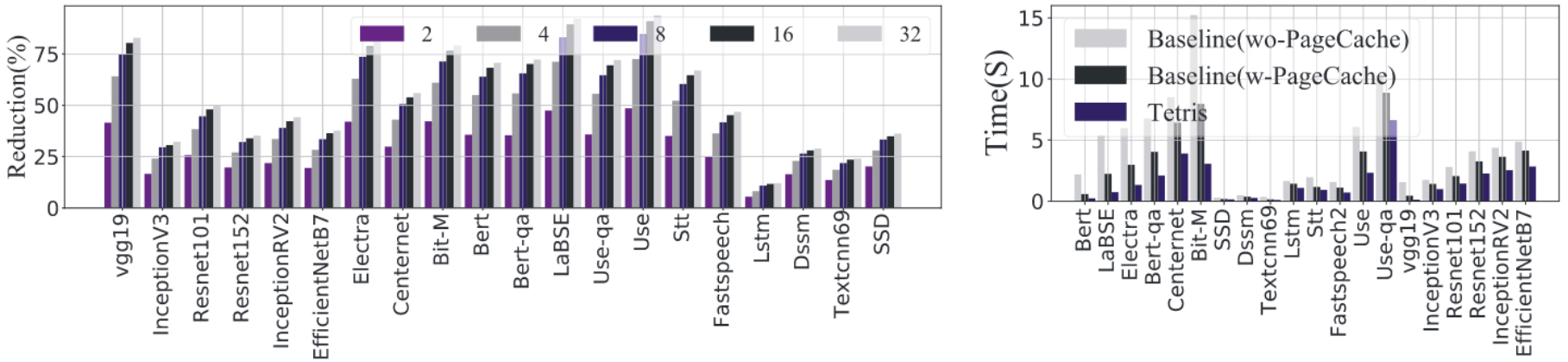


Figure 7: Overview of the tensor lifecycle in TETRIS.

Outline

Memory saving

- Memory reduction rate
- Acceleration of function startup



(a) Memory reduction under different # of instances

(b) Intra-model acceleration

Figure 9: (a) Memory reduction rate under tensor sharing over the number of instances from the same function. (b) Accelerating the startup using tensors from existing instance of the same function.

Memory saving

- Function density

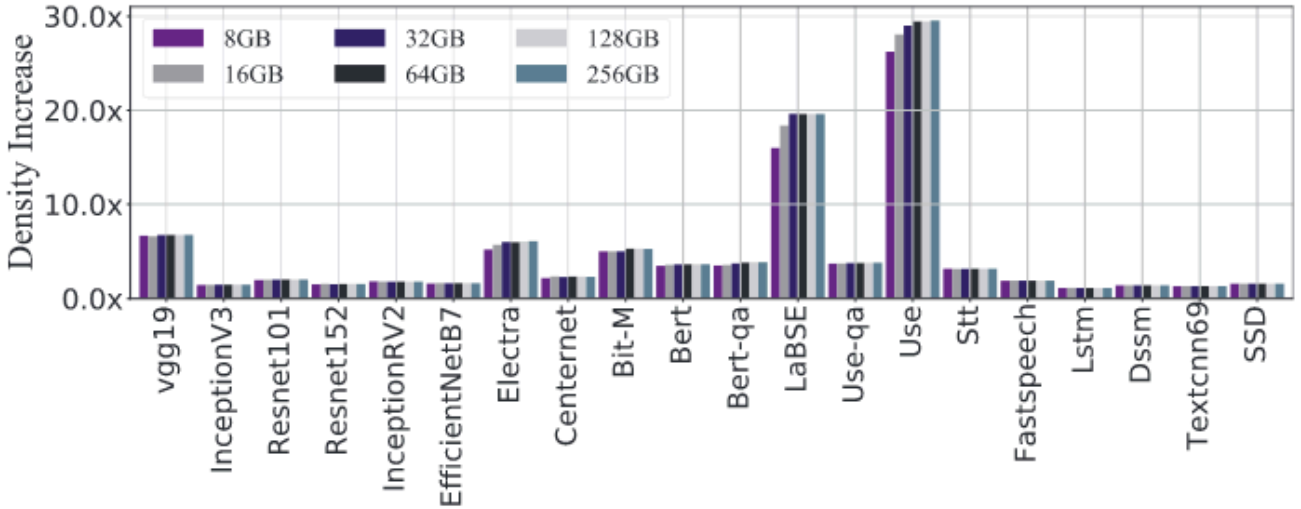


Figure 10: Function density improvement under various machine memory capacities.

Memory saving

- Reduce mean memory footprint by more than 86%

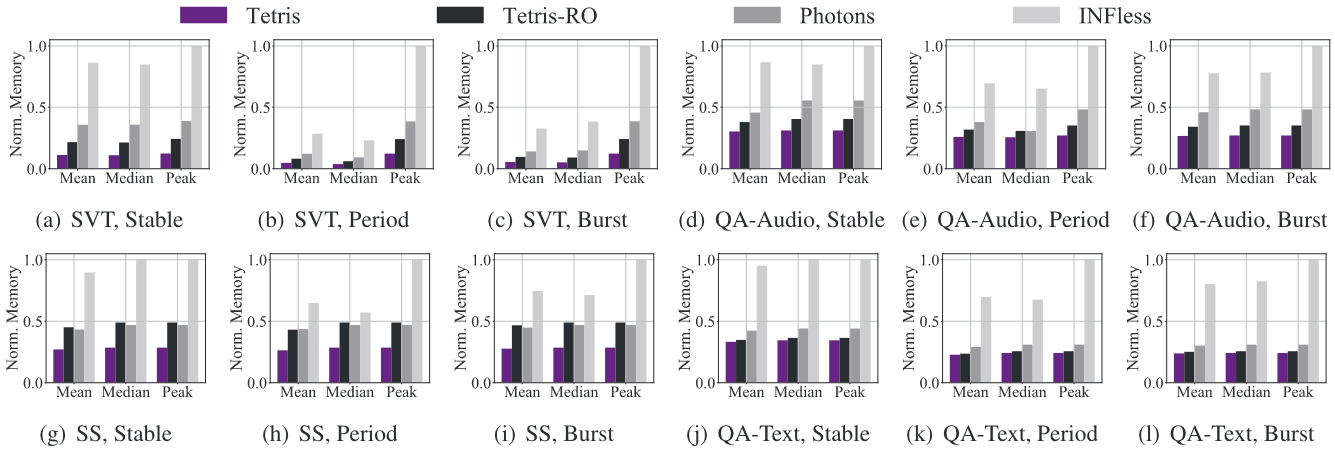


Figure 13: Normalized memory consumption by four applications under stable, period and bursty workloads.

Outline

Benefits of TETRIS:

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