TETRIS: Memory-efficient Serverless Inference through Tensor Sharing

Jie Li¹, Laiping Zhao¹, Yanan Yang¹, Kunlin Zhan², Keqiu Li¹

¹Tianjin University, ²58.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

- High request load
- Numerous instances
- Huge parameters
- Memory inefficiency
Serverless Inference

Drawback of Serverless Inference:
• Memory Inefficiency
  • High memory redundancy

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

The problem to be solved in this work

Redundant
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:

1. Enable tensor sharing on each server

2. Carefully schedule instances across servers to share more tensors

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?
  • Create a new memory region if the tensor has never been loaded
  • Mmapping existing memory region if the tensor has already been loaded

Tensors are identified by hash values
Design of TETRIS

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

- How does the Reclaimer detect and reclaim unreferenced tensors?

\[ T_{\text{warm}} \]

Get the tensor set of running instances

\[ T_{\text{all}} \]

Get the tensor set in the Tensor Store

\[ T_{\text{cold}} = \frac{T_{\text{all}}}{T_{\text{warm}}} \]

Infer the tensor set to be reclaimed

Run periodically
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
Design of TETRIS

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

- The lifecycle of tensors

![Lifecycle Diagram]

Legend:
- Agent Processing
- Reclaimer Processing

States:
- Loaded
- Shared
- On-Disk
- Unreferenced
- Reclaimed
- Keep Alived

Actions:
- Memory mapped
- Loaded from model file
- No function instance is referencing
- Reclaiming memory
- Keeping Alive (optional)
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 $i$ and server $j$:

$$
\Theta_{ij} = \frac{\text{Mem}(T_i \cap T_{\text{store}}^j)}{\text{Mem}(T_i)}
$$
Design of TETRIS

How to share function instance runtimes under SLO constraints?
• Profile inference latency under various combinations of \(<\text{CPU}, \text{memory}, \text{batch size}, \text{concurrency}>\)
• Model the instance scaling process as an optimization problem

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

How to share function instance runtimes under SLO constraints?
- Model the instance scaling process as an optimization problem

\[ \text{minimize} : \sum_{i=1}^{n} m_i x_i \]
\[ l_i \leq t_{slo}, \quad \forall i \wedge x_i \geq 1 \wedge b_i = 1 \]
\[ l_i \leq t_{slo}/2, \quad \forall i \wedge x_i \geq 1 \wedge b_i > 1 \]
\[ \sum_{i=1}^{n} x_i b_i p_i / l_i \geq R, \quad \forall i \]
\[ x_i \in \mathbb{N} \]

- Subject to minimize the memory consumption
- The SLO constrains
- 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

However, this problem is NP-Complete

\[
\begin{align*}
\text{minimize:} & \quad \sum_{i=1}^{m} m_i x_i \\
\text{subject to:} & \quad l_i < t_{\text{slo}}, \forall i \land x_i > 1 \land b_i = 1 \\
& \quad \sum_{i=1}^{m} x_i b_i p_i / l_i \geq R, \forall i \\
& \quad x_i \in \mathbb{N}
\end{align*}
\]

A simple greedy solution:

- Greedily select configuration \( i \) with maximum \( \frac{\text{throughput}_i}{\text{memory}_i} \) or \( \frac{\text{throughput}_i}{\text{memory}_i + \alpha \text{CPU}_i} \)

(To balance the CPU consumption)
## Evaluation

### 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

<table>
<thead>
<tr>
<th>DL Model</th>
<th>Size</th>
<th>Description</th>
<th>Download times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit.M [37]</td>
<td>3.5GB</td>
<td>Feature vector extraction</td>
<td>1.4k</td>
</tr>
<tr>
<td></td>
<td>1.8GB</td>
<td>Sentence Embedding</td>
<td>24.9K</td>
</tr>
<tr>
<td>Bert-qa [14]</td>
<td>1.3GB</td>
<td>Question Answering</td>
<td>5K</td>
</tr>
<tr>
<td></td>
<td>1.2GB</td>
<td>Question Answering</td>
<td>5.4K</td>
</tr>
<tr>
<td>Use [7]</td>
<td>980MB</td>
<td>Sentence Encoder</td>
<td>1.4M</td>
</tr>
<tr>
<td>Centernet [75]</td>
<td>731MB</td>
<td>Object Detection</td>
<td>12.8K</td>
</tr>
<tr>
<td>Use-qa [72]</td>
<td>568MB</td>
<td>Question Answering</td>
<td>16.3K</td>
</tr>
<tr>
<td>Use-large [7]</td>
<td>563MB</td>
<td>Sentence Encoder</td>
<td>1.1M</td>
</tr>
<tr>
<td>ViT [97]</td>
<td>549MB</td>
<td>Image Classification</td>
<td>commercial</td>
</tr>
<tr>
<td>Bert [14]</td>
<td>392MB</td>
<td>Text Processing</td>
<td>197.5K</td>
</tr>
<tr>
<td>EfficientNetV2 [68]</td>
<td>255MB</td>
<td>Image Processing</td>
<td>2.2K</td>
</tr>
<tr>
<td>ResNet [52]</td>
<td>231MB</td>
<td>Image Processing</td>
<td>1.6K</td>
</tr>
<tr>
<td>EfficientNetV2 [66]</td>
<td>214MB</td>
<td>Image Processing</td>
<td>6K</td>
</tr>
<tr>
<td>Sit [6]</td>
<td>176MB</td>
<td>Speech-To-Text</td>
<td>398</td>
</tr>
<tr>
<td>Fastspeech2 [57]</td>
<td>171MB</td>
<td>Speech-To-Text</td>
<td>1.6K</td>
</tr>
<tr>
<td>EfficientNetV2 [67]</td>
<td>110MB</td>
<td>Text-To-Speech</td>
<td>310</td>
</tr>
<tr>
<td>SSD [43]</td>
<td>29MB</td>
<td>Object Detection</td>
<td>commercial</td>
</tr>
<tr>
<td>Dssm [29]</td>
<td>25MB</td>
<td>Text Processing</td>
<td>commercial</td>
</tr>
<tr>
<td>LT [17]</td>
<td>23MB</td>
<td>Text Processing</td>
<td>commercial</td>
</tr>
<tr>
<td>Textformer [25]</td>
<td>11MB</td>
<td>Text Processing</td>
<td>commercial</td>
</tr>
</tbody>
</table>
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%
Evaluation

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

More experimental settings
• 4 real-world applications
• 3 real-world workload traces (from Azure)
• Comparison systems:

<table>
<thead>
<tr>
<th>System</th>
<th>Runtime Sharing</th>
<th>Tensor Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetris</td>
<td>Combined</td>
<td>yes</td>
</tr>
<tr>
<td>Tetris-RO</td>
<td>Combined</td>
<td>no</td>
</tr>
<tr>
<td>INFless</td>
<td>Batching</td>
<td>no</td>
</tr>
<tr>
<td>Photons (modified)</td>
<td>Multi-threading</td>
<td>no</td>
</tr>
</tbody>
</table>
Evaluation

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