StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow

(W5) Paper Reading

Nan Lin

Shanghai Jiao Tong University

2024-07-26

StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow

Hao Wu[†], Yue Yu[†], Junxiao Deng[†], Shadi Ibrahim[§], Song Wu[†‡], Hao Fan[†‡], Ziyue Cheng[†], Hai Jin[†‡]
[†]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China*
[‡]*Jinyinhu Laboratory, China*
[§]*Inria, Univ. Rennes, CNRS, IRISA, France*

- Published on **2024 USENIX**
- Click the link to open: https://www.usenix.org/conference/atc24/presentation/wu-hao

Outline

Outline

Motivation

Their work

Performance

Observations: Monolithic GPU Runtime

Monolithic GPU Runtime:

- DNN inference does not need the entire GPU
- State-of-the-art colocate multiple function on a single GPU, with each function is associated with a separate GPU runtime.

which brings about multiple problems.

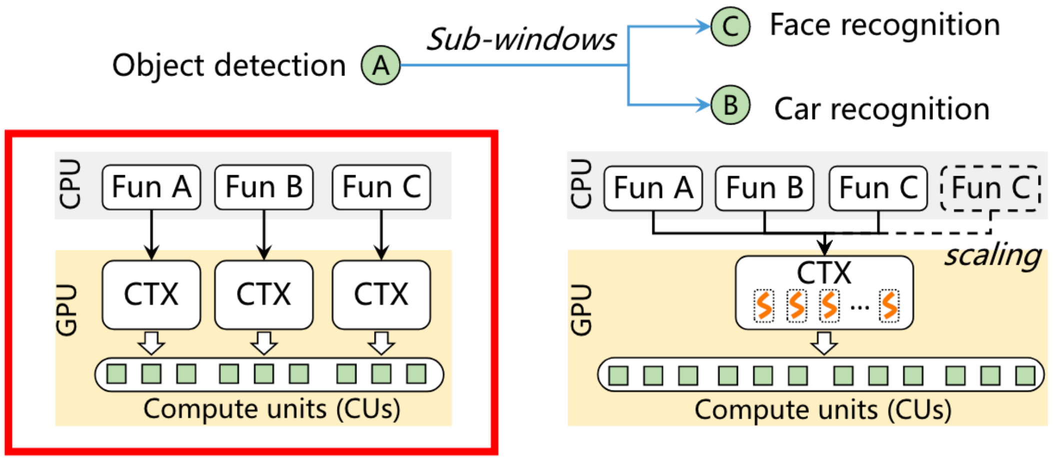


Figure 1: The deployment of a *Traffic* serverless inference workflow. Left: One GPU runtime per function (state-of-the-art). Right: One GPU runtime per inference workflow. CTX denotes CUDA context.

Observations: Monolithic GPU Runtime

Problems:

- *High redundancy and excessive memory footprint*
    - GPU runtime occupies up to 95%
    - Low deployment density when multiple functions sharing one GPU
- Unacceptable runtime of cold start overhead

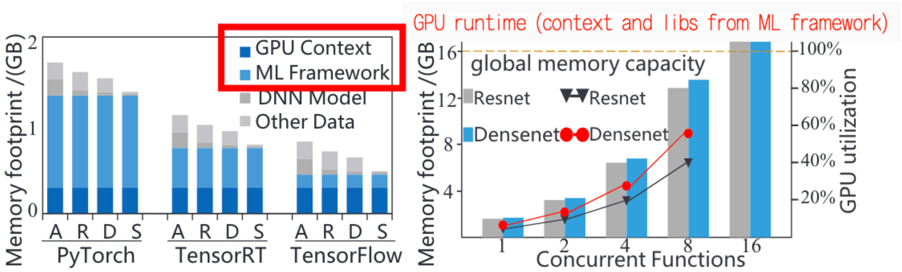


Figure 2: Left: Breakdown of the GPU memory footprint of functions, where A, R, D, and S denote the models selected from the inference workflows (Fig. 11): AlexNet, ResNet, DenseNet, and SSD. Right: GPU utilization and memory usage of different function concurrency.

Observations: Monolithic GPU Runtime

Problems:

- High redundancy and excessive memory footprint
- *Unacceptable runtime of cold start overhead*
  - Initialization of GPU runtime (up to 5s)
  - Current warm-up methods not pratical

ResNet-152 (Pytorch/V100)	Latency
CPU sandbox initialization	20ms
CUDA context initialization	324ms
Libraries initialization	5530ms
Model transmission	29ms
Memory allocation/free	32ms
Total coldstart overhead	5874ms
Inference	31ms

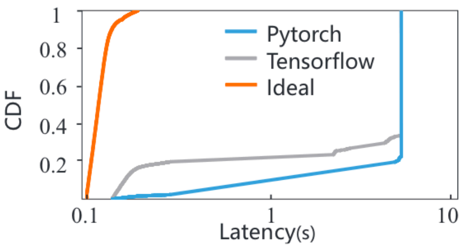
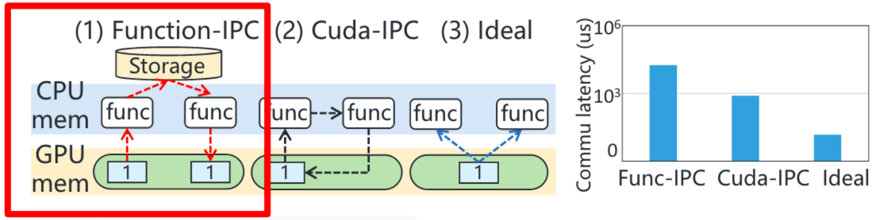


Figure 3: Left: Breakdown of function cold start latency. Right: CDF for end-to-end latency in *Traffic* workflow with warming up function. Ideal refers to sharing the GPU runtime.

Observations: Monolithic GPU Runtime

Problems:

- High redundancy and excessive memory footprint
- Unacceptable runtime of cold start overhead
- *Inefficient communication*
  - Long data transfer path: GPU → CPU → Storage → CPU → GPU
  - Isolation of functions cause excessive data copys



*IPC = Inter Process Comunication

Figure 4: Left: Existing communication methods in serverless inference systems. Right: The latency of each communication method.

Observations: Monolithic GPU Runtime

Conclusion: Limitations of **Monolithic GPU Runtime**

- High redundancy and excessive memory footprint
- Unacceptable runtime of cold start overhead
- Inefficient IPC

Outline

Overview

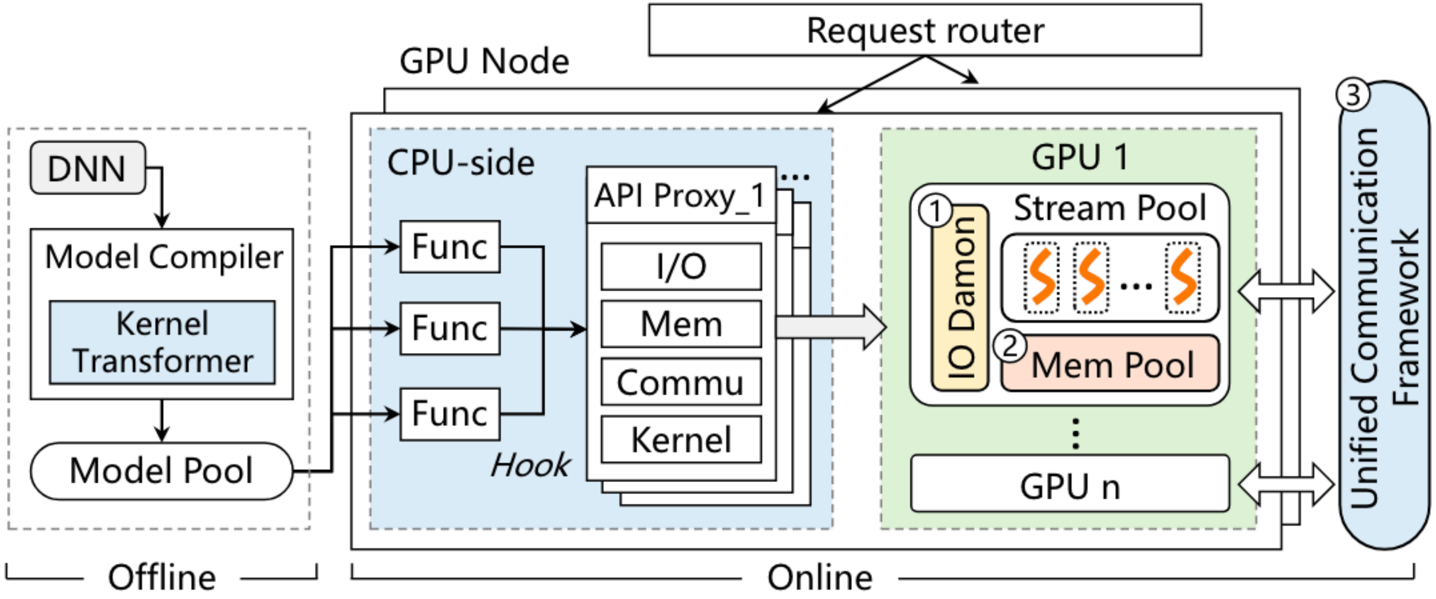


Figure 5: Architecture of StreamBox

Auto-scaling Memory Pool

Goal: Allocate exact memory usage of functions in the shared runtime

Principles:

- **Lazy allocation**
  - Variables allocated to physical memory when a variable is accessed for the first time
- **Eager recycling**
  - Cache layers in the memory pool as long as they are accessed
  - *Pre-run* inference tasks to infer how many times each variable is being accessed

Auto-scaling Memory Pool

Resilient scaling:

- **Offline pre-run**: Record exact memory usage during the execution
- **Real-time memory pool scaling**: Periodically adjust the memory pool size at a fixed interval.

$$T_{\text{interval}} = T_{\text{alloc}} + T_{\text{free}} \quad (200\ \text{MB})$$

- $T_{\text{offline}} < T_{\text{real-time}}$ ensured

Algorithm 1 Real-time memory pool scaling

Input: offline profiling of memory usage and runtime per kernel
Output: A new memory pool size M_{resize}

```
while there are functions running do
    M_c, M_next, m_next ← 0;
    for func ← concurrent_functions do
        k_c ← concurrent_kernel(func);
        m_c ← memory_usage(k_c);
        for k_n ← kernels_in_next_interval(func, T_interval) do
            m_next ← Max(m_next, memory_usage(k));
        end for
        M_c ← M_c + m_c, M_next ← M_next + m_next;
    end for
    M_resize = Memory_pool_size() − Max(M_c, M_next)
    sleep(T_interval/2);
end while
```

Unified Communication Framework

Three types of functions communications in GPU cluster:

- Functions deployed in the *same* GPU
- Functions deployed in *different* CPUs (*P2P* mechanism through *NVLink*)
- Functions deployed on different nodes, *Remote Procedure Call (RPC)* is used

Unified Communication Framework

Implementation of a shared *communication store* in GPU: cache intermediate data

- When a subsequent function on the same GPU wants to access the data, it can get the physical address directly from the communication store.

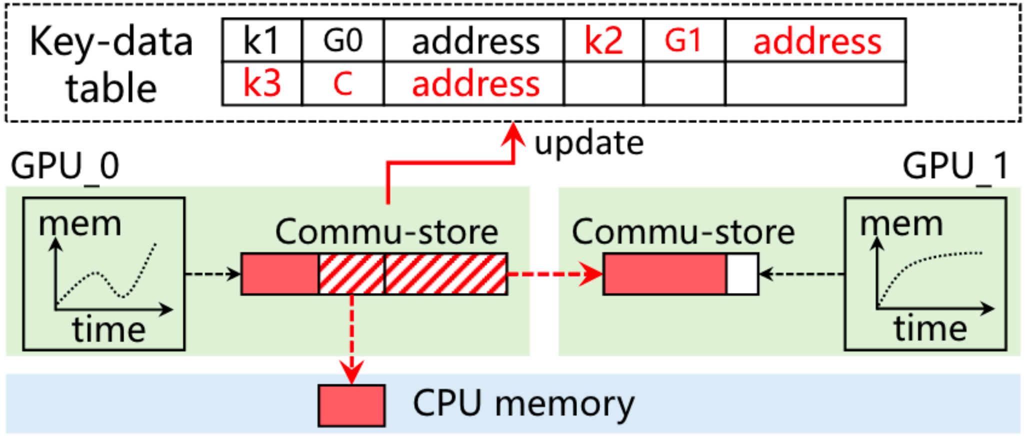


Figure 8: An example of adaptive inter-GPU movement

Unified Communication Framework

Characteristics:

- **Memory pressure awareness**
  - Prediction of GPU memory usage
  - Host intermediate data
  - Early move of intermediate data when the memory pressure increases
- **Adapative inter-GPU mouvement**
  - Neighboring GPUs $\rightarrow$ CPU: NVLink (300 GB/s) $\gg$ PCIe (12 GB/s)
  - Rules: Even between GPUs, Prioritize larger data, Prioritize moving

Outline

Memory Reduction

Reduces GPU memory usage by up to 82%.

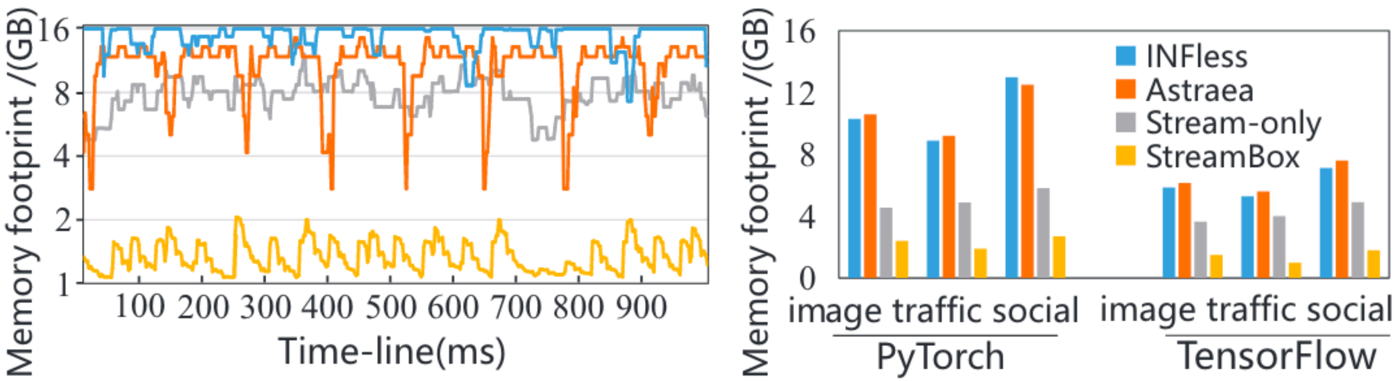


Figure 12: Left: Memory usage (log-scale) under real-world trace. Right: memory usage of different inference workflows and ML frameworks.

Improves Throughput

Improves system throughput by 5.3x-6.7x.

Figure 13: Comparison of throughput

Low Latency

Guarantee SLO and reduce end-to-end latency.

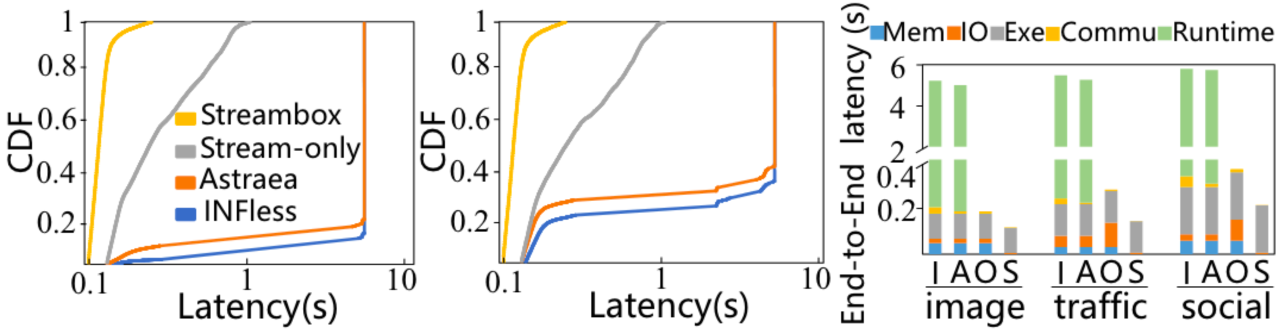


Figure 14: (a) The end-to-end latency (log scale) CDF of using PyTorch, (b) the latency CDF of TensorFlow, (c) the breakdown of end-to-end latency for INFless (I), Astraea (A), Stream-only (O), and StreamBox (S).