StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow

Hao Wu, Yue Yu, and Junxiao Deng, *Huazhong University of Science and Technology;* Shadi Ibrahim, *Inria;* Song Wu and Hao Fan, *Huazhong University of Science and Technology and Jinyinhu Laboratory;* Ziyue Cheng, *Huazhong University of Science and Technology;* Hai Jin, *Huazhong University of Science and Technology and Jinyinhu Laboratory*

Presenter: Mausam Basnet, University of North Texas

Date: 2024-10-03

Introduction

Serverless computing is gaining popularity for deploying scalable DNN inference workflows.

- Requests for DNN inference are brusty and dynamic.
- They provide high elasticity, cost efficiency (i.e., pay-as-you-go), and straightforward deployment.

- In serverless inference workflow, each DNN model is encapsulated as a "function"
- Each function running on GPU requires a GPU runtime (e.g., CUDA context)

Default Time Slicing

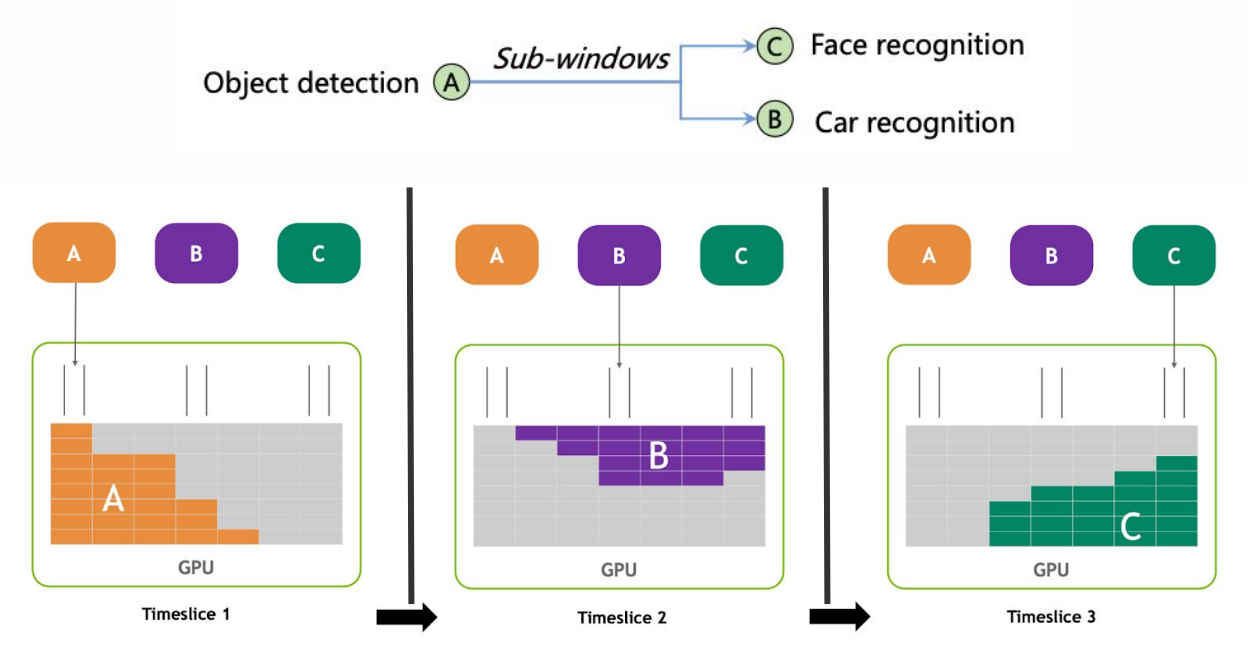

Fig: One GPU runtime per inference workflow

Current State-of-the art

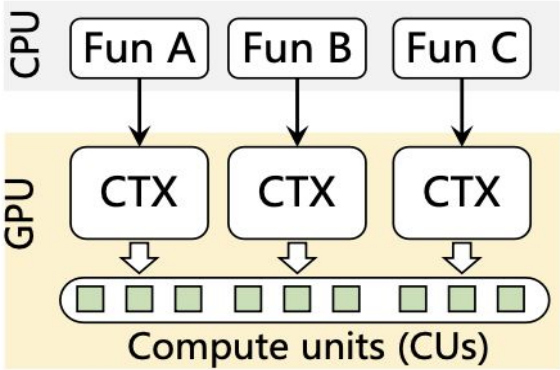

Fig: One GPU runtime per function

- Use Nvidia Multi-Process Service (MPS)
- Spatial Sharing of GPU
- Each function starts by creating a GPU runtime (e.g., CUDA context)
- However, challenges arise in efficiently managing GPU resources

StreamBox Approach

- Functions share a GPU runtime instead of being isolated in redundant GPU runtimes

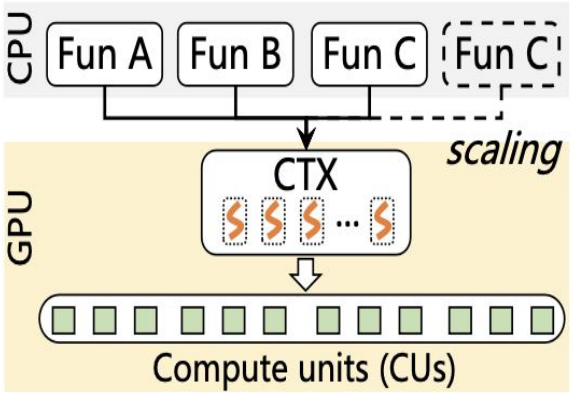

Fig: One GPU runtime per inference workflow

Limitation

Limitation 1

High redundancy and excessive memory footprint.

Limitation 2

Unacceptable cold start overhead

Limitation 3

Inefficient communication

High redundancy and excessive memory footprint

Memory Footprint: High memory consumption by GPU runtime (up to 95% of total memory footprint)

AlexNet (A), ResNet(R), DenseNet(D), and SSD(S)

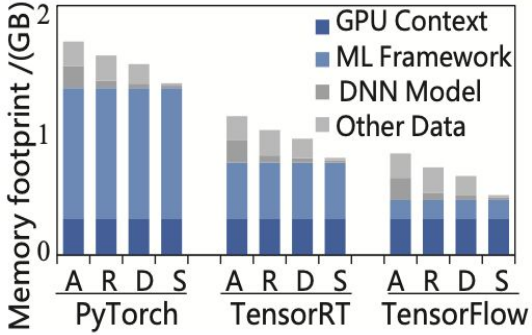

Fig: Breakdown of the GPU memory footprint of functions

Unacceptable cold start overhead

Container startup, GPU runtime initialization and the loading of DNN model and input data

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

Fig: Breakdown of function cold start latency

Inefficient communication

Example: In traffic workflows, data is redundantly copied between GPU and CPU before being transferred between functions, increasing latency.

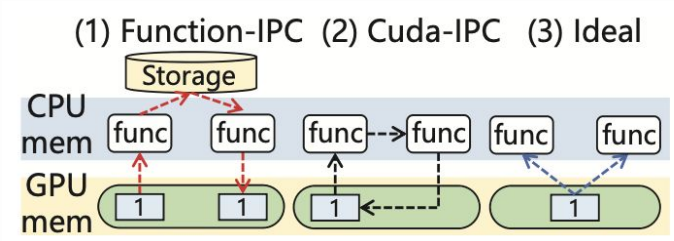

Fig: Existing communication methods in serverless inference systems

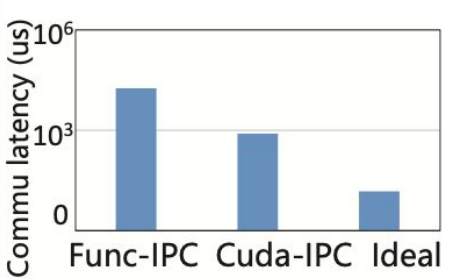

Fig: The latency of each communication method

Use of Streams and Challenges

Streams can execute kernels concurrently and share the address space in a GPU runtime.

- Modern GPU libraries (e.g., CUDA and ROCm) commonly provide streams.
- Streams can partition the GPU through software-only methods.
- Challenges in serverless functions
    - Allocate memory for auto-scaling functions efficiently
    - Achieve efficient inter-function communication
    - Alleviate the problem of IO blocking

Auto-scaling Memory Pool

Fine-grained Memory Management & Resilient Scaling

Lazy Allocation: Delays memory allocation until variables are accessed

Eager Recycling: Frees memory for layers once they are no longer needed

Memory Demand Profiling: Pre-runs models offline to determine precise memory needs

Elastic Memory Scaling: Dynamically adjusts memory pool based on real-time demand

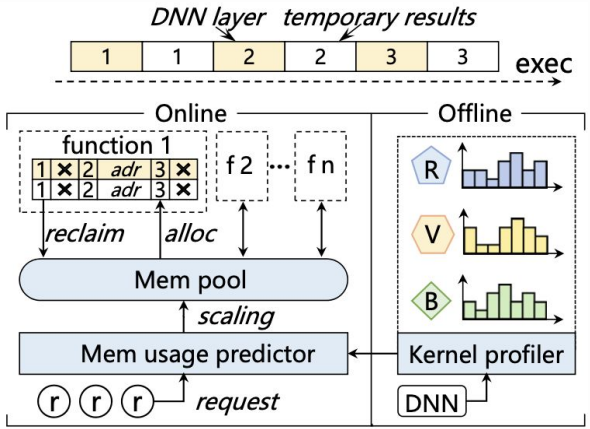

Fig: Fine-grained memory management in StreamBox

User-transparent Communication Framework

Unified Communication Framework & Elastic Communication Store

Three Communication Methods

Easy-to-Use APIs: PUT/GET for seamless data transfer with unique indexing.

Shared Store: Direct data access, bypassing CUDA IPC.

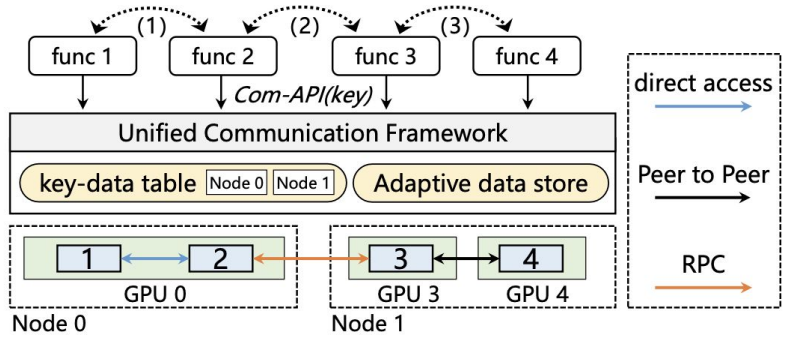

Fig: Unified communication framework in StreamBox

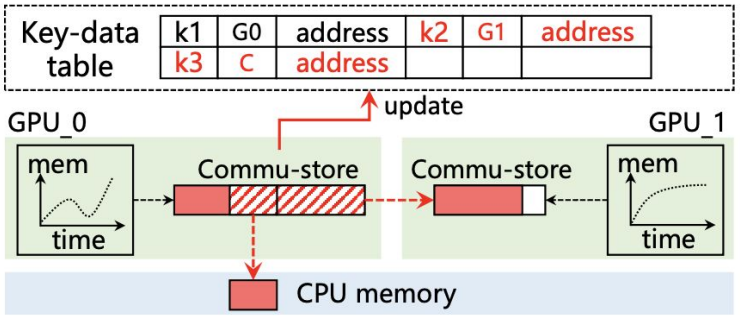

Fig: An example of adaptive inter-GPU movement

Fine-grained PCIe Bandwidth Sharing

Partitions the data of functions into smaller data blocks to overcome monopoly of a Stream

Fine-Grained Scheduling: Hooks transfer calls, divides data into 2MB blocks, and uses a global queue with preemption for efficient sharing.

Optimal Block Size: 2MB blocks maximize transfer bandwidth, avoiding excessive overhead and bandwidth waste.

Preemptive Batching: Groups six blocks per transfer batch to accommodate new requests without blocking.

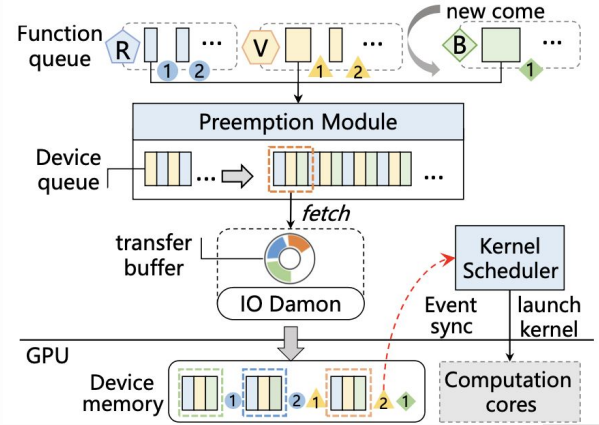

Fig: Fine-grained I/O scheduling in StreamBox

StreamBox Architecture

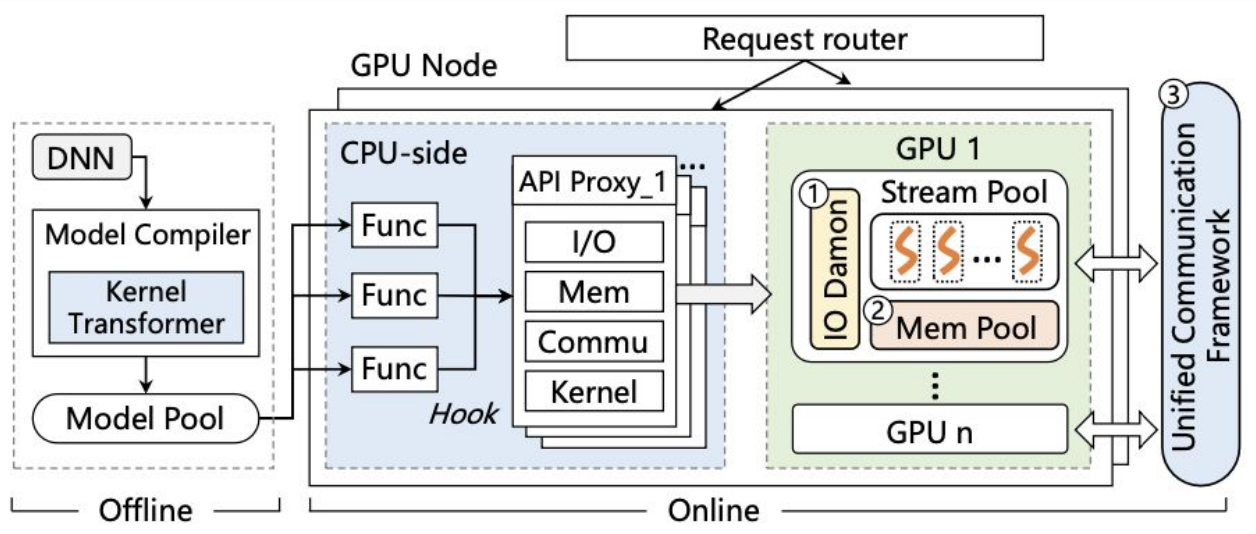

Fig: StreamBox Architecture

Evaluation

Experimental Setup

Hardware: 28-core CPU, 128GB DRAM, 4x Nvidia V100 GPUs (16GB each).

Software: PyTorch-1.3.0, TensorFlow-2.12, CUDA-10.1.

Workflows:

- Image Processing: Face recognition, image enhancement.
- Traffic: Object detection, vehicle/face ID, license plate extraction.
- Social Media: Vision and language models for post translation and classification.

Comparison:
StreamBox vs. INFless, Astraea, Stream-only

Performance of StreamBox : Memory Footprint

Up to 82% Reduction in GPU memory usage

Compared to Others:

- INFless & Astraea: Cuts memory usage by 79%-82%, outperforming these systems by avoiding redundant GPU runtimes.

- Stream-only: Achieves a 71% reduction due to precise memory management.

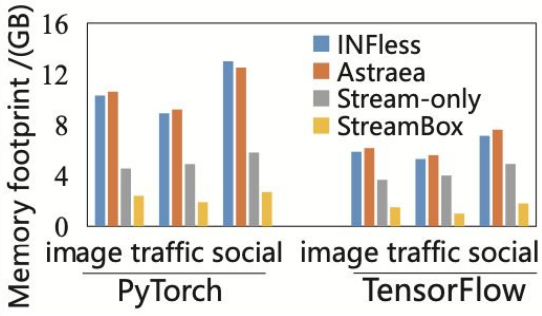

Fig: Memory usage of different inference workflows and ML frameworks

Performance of StreamBox : Throughput

StreamBox improves system throughput by 5.3X-6.7X

Compared to Others:

- INFless & Astraea: Achieves 6.7X and 5.3X higher throughput, respectively
- Stream-only: Improves throughput by 1.46X due to better memory management and communication efficiency.

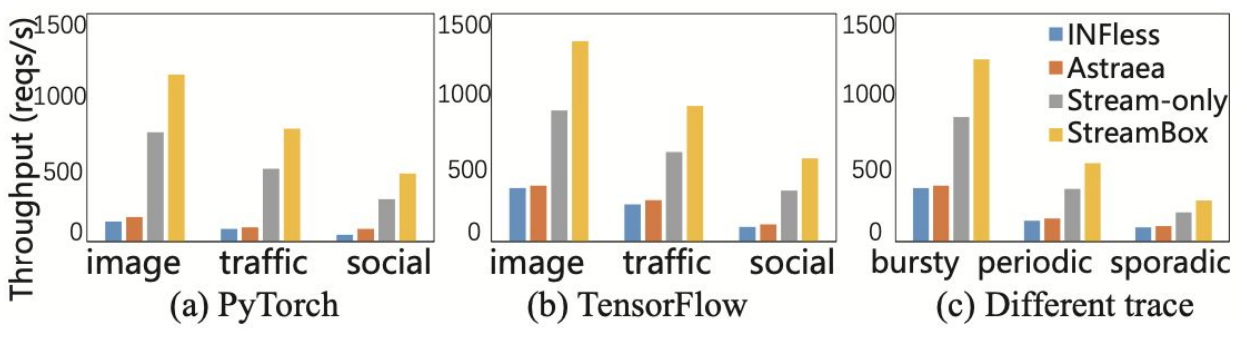

Fig: Comparison of throughput

Performance of StreamBox : Latency

Significant decrease in end-to-end latency

Compared to Others:

- INFless & Astraea: Reduces end-to-end latency by up to 98%. (startup time: 5.8 s to 5 ms)
- Stream-only: Lowers latency by up to 49%, thanks to optimized I/O, memory, and communication handling.

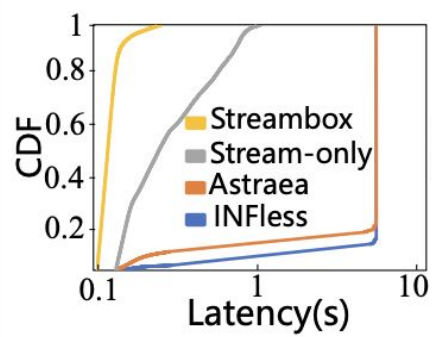

Fig: The end-to-end latency (log scale) CDF of using PyTorch

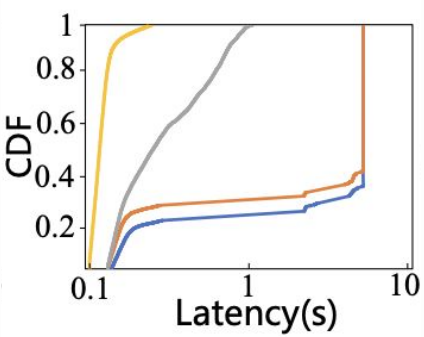

Fig: the latency CDF of TensorFlow

Optimizations in StreamBox

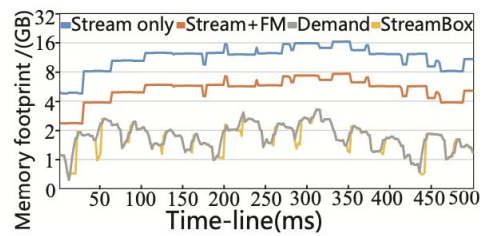

Fig: memory usage under real-world trace

Auto-scaling memory pool
Drops execution time by 34%
Cuts memory use by up to 91%
Closely tracks actual demand

Efficient intra-GPU communication
↓ by 94% (INFless)
↓ by 73% (Stream-only)

Fig: Comparison of communication latency

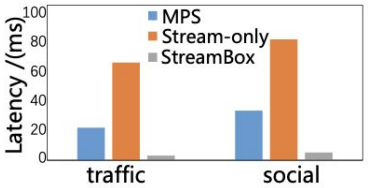

Fig: Latency of pipelined model loading

Fine-grained PCIe bandwidth sharing
↓ by 80% (MPS- INFless)
↓ by 91% (Stream-only)

Conclusion

- Monolithic GPU runtimes in serverless systems cause high cold start latency and large memory footprint.
- StreamBox reduces redundant GPU runtimes and enhances memory sharing.
- Introduces auto-scaling memory pooling for efficient resource management.
- Implements intra-GPU communication for improved function interaction.
- Demonstrated efficacy through experimental results.

Limitations and Potential Improvement

- **Scalability Challenges**: The approach may face difficulties in highly dynamic or large-scale scenarios.
- **Integration Complexity**: StreamBox enhancements may be difficult to integrate with existing serverless frameworks.

Thanks!

Do you have any questions?