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*

https://www.usenix.org/conference/atc24/presentation/wu-hao

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

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*

Abstract

The dynamic workload and latency sensitivity of DNN inference drive a trend toward exploiting serverless computing for scalable DNN inference serving. Usually, GPUs are spatially partitioned to serve multiple co-located functions. However, existing serverless inference systems isolate functions in separate monolithic GPU runtimes (e.g., CUDA context), which is too heavy for short-lived and fine-grained functions, leading to a high startup latency, a large memory footprint, and expensive inter-function communication. In this paper, we present StreamBox, a new lightweight GPU sandbox for serverless inference workflow. StreamBox unleashes the potential of streams and efficiently realizes them for serverless inference by implementing fine-grain and auto-scaling memory management, allowing transparent and efficient intra-GPU communication across functions, and enabling PCIe bandwidth sharing among concurrent streams. Our evaluations over real-world workloads show that StreamBox reduces the GPU memory footprint by up to 82% and improves throughput by 6.7X compared to state-of-the-art serverless inference systems.

1 Introduction

Deep Neural Network (DNN) inference has been widely adopted in today's intelligent applications, such as autonomous driving [2, 17], virtual reality [32, 43], image recognition [16, 40]. Usually, inference services are implemented as a workflow consisting of multiple DNN models. As shown in Fig. 1, the traffic monitoring application [36] consists of an object detection model and two recognition models for faces and cars. The demand for parallel computing in DNN models is continuously growing, and therefore inference computations are gradually moving to GPUs [7, 38, 46].

Serverless inference (i.e., deploying inference workflows on serverless computing) is gaining increasing popularity for proven high elasticity, cost efficiency (i.e., pay-as-you-go), and transparent deployment [2, 37, 42]. In serverless inference, each DNN model is encapsulated as a "*function*", which can automatically scale up or down according to the change in number of requests. Each function (i.e., container) running on GPU requires a GPU runtime (e.g., CUDA context),

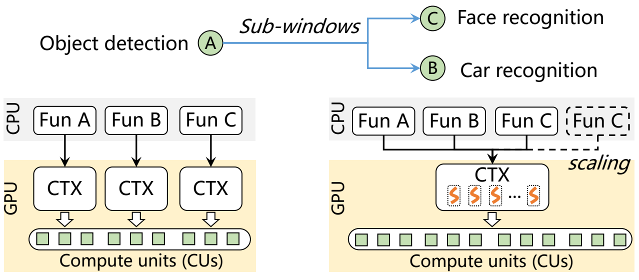


Figure 1: The deployment of a *Traffic* serverless inference workflow. Left: One GPU runtime per function (state-of-the-art). Right: One GPU runtime per inference workflow. CTX denotes CUDA context.

which encapsulates all hardware resources and libraries associated with the program. Since DNN inference does not need the entire GPU, state-of-the-art serverless inference systems, such as INFless [42], Astraea [47], and LLAMA [36], co-locate multiple functions on a single GPU (each function is associated with a separate GPU runtime), due to the Nvidia *Multi-Process Service* (MPS [27]) that facilitates spatial GPU sharing among GPU runtimes (left graph in Fig. 1). Unfortunately, running separate monolithic GPU runtimes is too heavy for short-lived and fine-grained functions. Our analysis (Section 3.1) reveals that isolating functions with monolithic GPU runtimes has several deficiencies including an excessive memory footprint (more than 90% data redundancy in GPU memory), unacceptable cold start overhead (over 5s), and redundant data transfers in communication.

This paper presents StreamBox, a new lightweight GPU sandbox for serverless inference. The key idea is to enable functions within an inference workflow (i.e., functions with different DNN models or functions scaling out) to share a GPU runtime instead of being isolated in redundant GPU runtimes, as shown in Fig. 1 (right); and to realize that by using GPU streams. GPU streams are commonly provided by modern GPU libraries (e.g., CUDA [42] and ROCm [10]) to enable concurrent kernel executions within a GPU runtime and to share address space, similar to threads in a process. They have been widely used to increase the parallelism of kernels and improve GPU utilization [15, 22, 26].

Leveraging GPU streams to achieve very low latency (10-100ms [41]) and high concurrency for serverless inference poses several challenges. C1: *Efficient memory allocation and sharing.* When sharing a GPU memory address between functions, coarse-grained memory allocation leads to significant overhead (over 30ms) and idle memory. Furthermore, existing memory management mechanisms (cudaMemPool and PyTorch) build a static memory pool without the philosophy of auto-scaling, leading to increased user cost under serverless pay-as-you-go billing models. C2: *Inter-stream communication.* The transparent function deployment in serverless prevents the user program from choosing an appropriate communication method according to the location of the function. Moreover, intra-GPU communication in a shared address space still suffers from redundant data transfers. C3: *High-performance parallelism.* Streams utilize the PCIe link between GPU and CPU exclusively. The first arriving stream monopolizes the bandwidth until all its data transfers are completed. This hurts the parallelism of functions.

StreamBox addresses these challenges as follows. First, we design an *auto-scaling memory pool* combined with *fine-grained memory management*. We allocate and recycle memory at layer granularity to achieve high memory efficiency. We also utilize offline profiling of DNN inference and build memory usage models to make the memory pool resilient to the exact memory usage of concurrent functions. Second, we provide a *unified communication framework* that transparently switches the communication methods between functions according to the distribution of functions in the GPU cluster. We store intermediate data in GPU memory to accelerate communication and also propose an *elastic communication store* to avoid memory competition between running functions. Finally, we achieve *fine-grained PCIe bandwidth sharing* among streams on top of the closed-source GPU driver. We partition the data of different functions into right-sized data blocks. We also design pinned memory buffer and preemptive transfer; and re-build synchronization to achieve high-performance transfers of these data blocks.

We implement StreamBox in OpenWhisk by extending Apache TVM on Nvidia V100 GPU. We evaluate the efficiency and effectiveness of StreamBox with diverse inference workflows and models using Azure cloud traces [37]. Compared to state-of-the-art, our experimental results show that StreamBox reduces GPU memory footprint and startup latency by up to 82% and 98% (startup latency less than 5ms), respectively, and improves the throughput by 6.7X. Furthermore, compared to Stream-only that simply runs functions with streams, StreamBox reduces memory footprint and startup latency by up to 61% and 49%, respectively, and improves throughput by 1.46X.

In summary, we make the following contributions.

- We provide an in-depth understanding of the deficiencies of state-of-the-art serverless inference systems caused by the monolithic GPU runtime.

Table 1: State-of-the-art GPU serverless inference systems

Systems	GPU split	Runtime	Startup	Footprint	Commu
INFless [42]	MPS	Redundant	5-8s	GBs	Func-IPC
Llama [36]	MPS	Redundant	5-8s	GBs	Func-IPC
Astraea [47]	MPS	Redundant	5-8s	GBs	CUDA-IPC
StreamBox	***Stream***	***Shared***	***5ms***	***MBs***	***Intra-GPU***

- We unleash the potential of streams for new lightweight GPU runtimes for serverless inference workflow through the design of StreamBox. To the best of our knowledge, this is the first study considering GPU streams for serverless inference workflow.
- We present a series of optimizations in terms of I/O, memory management, and communication, incorporating the characteristics of DNN inference (e.g., layered structure) and the requirements of short-lived and auto-scaling serverless functions.
- An easy-to-port implementation on Nvidia GPUs and extensive experiments that clearly demonstrate the advantage and efficacy of StreamBox over the state-of-the-art. We have open-sourced StreamBox at: https://github.com/CGCL-codes/streambox.git.

2 Background

2.1 DNN Inference Workflow

DNN models comprise multiple versatile layers (i.e., convolutional, pooling, and fully connected layers) that are executed in sequence. Each layer includes a set of complex computations such as matrix multiplication, which makes DNN inference better suited for hardware accelerators (i.e., GPUs) [7, 38, 41]. Inference services are usually structured as workflows including multiple DNN models (Fig. 11). For instance, the *traffic* workflow [38] starts with an object detection model, and sends objects (i.e., sub-windows) about people or cars to face and car recognition model for further identification, respectively. The number of functions for recognition dynamically scales with the number of objects detected by the upstream functions. Note that as models of a single workflow are developed and deployed under the same ML framework (e.g., PyTorch) for consistent performance, their running environments are usually identical.

2.2 Serverless Inference on GPU Servers

Serverless computing has received significant attention in recent years. By breaking an application into small functions that can be executed and scaled automatically, serverless computing promises applications with high elasticity, cost efficiency, and easy deployment [24, 30, 37]. Serverless computing is especially appealing for DNN inferences because

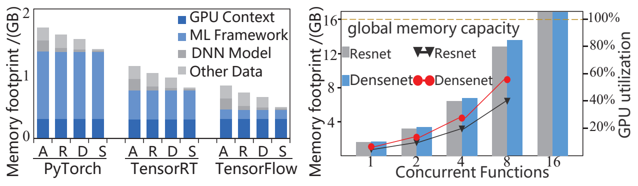


Figure 2: Left: Breakdown of the GPU memory footprint of functions, where A, R, D, and S denote the models selected from the inference workflows (Fig. 11): AlexNet, ResNet, DenseNet, and SSD. Right: GPU utilization and memory usage of different function concurrency.

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

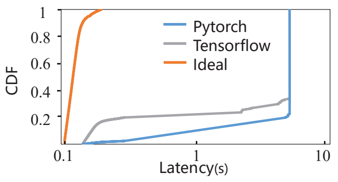


Figure 3: Left: Breakdown of function cold start latency. Right: CDF for end-to-end latency in *Traffic* workflow with warming up function. Ideal refers to sharing the GPU runtime.

requests of inference are bursty and dynamic [8,42,46]. Many serverless inference systems have emerged (Table 1).

Monolithic GPU runtime. When deploying a serverless system on a GPU, each function (i.e., container [13]) starts by creating a GPU runtime (e.g., CUDA context), which encapsulates all the hardware resources and libraries related to the program. Concurrent GPU runtimes co-located on a GPU are scheduled in time-slice (i.e., temporal sharing) by default.

Recent studies suggest that DNN inference can better benefit from spatial sharing on GPU to improve utilization [12,47], because DNN inference typically does not need the entire GPU, given the varied parallelism [15, 47] and the small batch sizes [8, 46]. Nvidia GPUs utilize *Multi-Process Service* (MPS) to enable concurrent execution of GPU runtimes on GPUs. With MPS, users can specify the percentage of compute resources (GPU%) available to each container over its lifetime. Most serverless inference systems [5,36,42,47] adapt MPS to spatially partition the GPU among functions (Table 1).

3 Motivation

In this part, we illustrate the limitations of the monolithic GPU runtime, which is currently adopted in serverless inference systems; and motivate the use of streams as new lightweight GPU sandboxes. We then discuss the main challenges to efficiently realize streams for serverless inference workflows.

3.1 Limitations of Monolithic GPU Runtime

To show the deficiencies of monolithic GPU runtimes, we use INFless [42], a state-of-the-art serverless inference system based on OpenFaas and Nvidia MPS. We consider popular ML frameworks including PyTorch and TensorFlow; and the latest inference engine TensorRT [28]. The models are taken from Torch Hub [35], and run on Nvidia V100 GPUs.

Observation 1: *High redundancy and excessive memory footprint. Isolating functions with separate GPU runtimes causes over 90% data redundancy in GPU memory.*

Fig. 2 (left) shows the GPU memory footprint of functions with different models and ML frameworks. We can see that the overall memory footprint of an inference function (ResNet) can be as high as 1.5GB, where the GPU runtime (i.e., context and libraries from ML framework) occupies up to 95%. This is much larger than the memory required for the inference itself (i.e., DNN model and temporary results). Moreover, in the inference workflow, GPU runtimes for functions with different models are the same (in blue), resulting in extreme data redundancy when multiple functions (with separate GPU runtimes) share a GPU. This, in turn, results in low deployment density. For example, a V100 GPU (memory capacity is 16GB) can accommodate at most 8 concurrent functions. This not only leads to waste in the expensive GPU memory but also causes low utilization of computing resources, with more than 60% idle resources as shown in Fig. 2 (right).

Observation 2: *Unacceptable cold start overhead. The cold start latency of GPU runtime is over 5s, and commonly used warm-up methods are ineffective on GPUs.*

The cold start of an inference function can be divided into: *container startup, GPU runtime initialization (i.e., the initialization of the context and ML framework), and the loading of DNN model and input data* (Fig. 3 (left)). The major latency comes from the initialization of GPU runtime (up to 5s). This is unacceptable for latency-sensitive DNN inference. Existing studies focus on optimizing the model loading [5,23], and use warm-up methods [12,14,42] (i.e., keeping functions alive for a while after completion) to reduce cold starts. Unfortunately, warm-up methods on GPUs are not practical. First, it is ineffective on GPUs since warming-up functions use a lot of GPU memory (**observation 1**), especially when there are functions with diverse models and various GPU% allocation. Second, resource reallocation in MPS requires restarting GPU runtimes. This has been also observed in other studies [7,10,12]. As shown in Fig. 3 (right), the warm-up method on GPU still suffers from noticeable cold start overhead.

Observation 3: *Inefficient communication. Communication between functions suffers from redundant data copys between CPUs and GPUs due to the isolation between GPU runtimes.*

Taking a traffic workflow as an example, when the batch size is 16, function F1 (i.e., object detection) transfers 125MB of data to function F2 (i.e., face recognition). Under the existing communication method (Fig. 4 (left)) in serverless inference systems, function F1 first copies data from GPU to CPU memory, then transfers the data to F2 through external storage, and F2 copies the data back to GPU. This results in a long data transfer path. On the other hand, under the Nvidia GPU CUDA-IPC communication method, which transfers a data handle instead of raw data, F1 first gets a data handle using `cudaIpcGetMemHandle()`, and passes the handle to F2 using standard IPC mechanisms. Then F2 maps the data into its own address space using `cudaIpcOpenMemHandle()`. However, operating the handle introduces additional overhead and requires CPU-side assistance. Ideally, since F1 and F2 reside on the same GPU, F2 should be able to access the data of F1 directly (Fig. 4 (right)). Hence, the ideal solution can significantly reduce the communication latency.

3.2 Potential of Streams

As shown above, monolithic GPU runtimes are impractical and inefficient for serverless inference workflow. This motivates us to find a new lightweight GPU sandbox for functions. We argue that functions of the same inference workflow do not need strong isolation, especially identical functions that scale up when the workload increases. Hence, using streams as sandboxes for inference workflow is an ideal choice. Modern GPU libraries (e.g., CUDA [42] and ROCm [10]) commonly provide streams to execute kernels concurrently and share the address space in a GPU runtime. On the other hand, streams can partition the GPU through software-only methods, such as Elastic Kernel [9, 31, 49] instead of hardware-support GPU partitioning methods (MPS or MIG). These methods dynamically map kernels to GPU compute units without runtime restarts, simplifying resource allocation for functions.

Opportunity. Streams offer four benefits. (1) *Fast startup and small memory footprint*: there is no need to initialize a GPU runtime for each function, which avoids the startup latency and eliminates data redundancy; (2) *Fast resource reconfiguration*: spatially partitioning GPU resources across streams can be realized through software-only solutions (Elastic Kernels [31]). Consequently, users can reconfigure resources without restarting the GPU runtime; (3) *Effective resource sharing*: streams share one address space, thus we can easily share memory among functions and thus improve GPU memory utilization; (4) *Efficient communication*: Sharing address space among streams also facilitates zero-copy data passing between functions in a GPU.

Earlier adoption of streams to share GPU resources. Previously, streams were mainly used at kernel-level to increase the parallelism of kernels (i.e., deploying as many kernels as possible in a single GPU). REEF uses streams to co-run best-effort tasks with real-time tasks [15]. KRISP explores kernel-wise resource allocation [10]. Other studies have investigated streams for DNN inferences. Remmer [26] uses streams to parallelize operators without data dependencies in DNN inference. Nvidia inference system Triton [29] and Pipeswitch [5] utilize streams to overlap model loading and kernel computation. Tacker [48] uses streams to exploit parallelism between Tensor Cores and CUDA Cores in Nvidia GPUs. However, to the best of our knowledge, this is the first study considering streams for serverless inference.

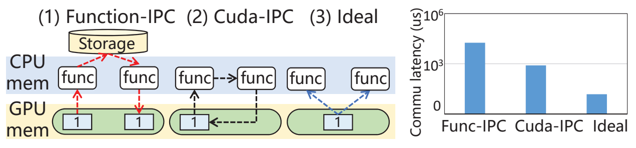


Figure 4: Left: Existing communication methods in serverless inference systems. Right: The latency of each communication method.

3.3 Challenges of Stream-based Sandbox

Despite potential performance and memory gains, using streams for serverless functions faces several challenges.

C1. How to allocate memory for auto-scaling functions efficiently? When sharing a GPU memory address between functions, coarse-grained memory management results in latency and unnecessary GPU memory reservation. On the one hand, the total memory of DNN inference is allocated in advance by default. This coarse-grained memory allocation not only leads to significant overhead (over 30ms) but also reserves a substantial amount of unnecessary memory. On the other hand, existing memory management mechanisms (cudaMemPool and PyTorch) build a static memory pool to cache freed memory. Without the philosophy of auto-scaling, this can also lead to excessive memory usage and increase users' costs under pay-as-you-go billing models.

C2. How to achieve efficient inter-function communication under the transparent deployment of serverless? When functions are scheduled on the same GPU, zero-copy data transfer can be achieved through shared GPU runtime. However, the placement (distribution) of functions depends on the workload and the scheduling, and leads to different communication patterns between functions (i.e., intra-GPU, inter-GPU, and inter-node communication). It is transparent to users, making it difficult to set an appropriate communication method (Fig. 7). Moreover, to enable intra-GPU communication through shared address space of streams, intermediate data needs to reside in GPU memory and consumes memory. Therefore, the intermediate data requires efficient management to avoid memory competition with running functions.

C3. How to alleviate the problem of IO blocking to achieve high-performance function parallelism? Streams use PCIe bandwidth exclusively since there is only one IO engine per GPU runtime in the GPU driver. So the first arriving stream monopolizes the bandwidth until all data transfers are completed, and other streams' executions are delayed due to this serial data transfer (Fig. 10 (left)). Note that the serial transfer between streams is acceptable for traditional monolithic tasks, where parallel streams are used to accelerate the processing

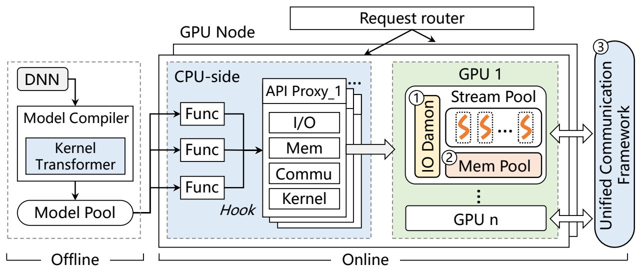


Figure 5: Architecture of StreamBox

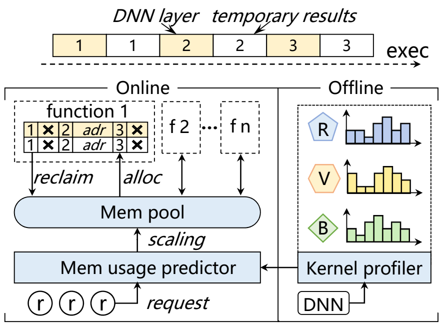


Figure 6: Fine-grained memory management in StreamBox

of the same data that are transferred at one time.

4 StreamBox Overview

The goal of StreamBox is to enable streams as efficient GPU sandboxes for serverless inference workflow:

- *Auto-scaling memory pool* that provides fine-grained GPU memory management and resilient scaling based on the exact memory usage of auto-scaling functions.
- *User-transparent communication framework* that affords unified communication APIs for developers to compose function workflow and leverages an elastic communication store to achieve intra-GPU communication while avoiding memory competition with running functions.
- *Fine-grained PCIe bandwidth sharing* that enables efficient data transfer of concurrent functions.

Fig.5 shows the architecture of StreamBox. StreamBox transforms the source code of DNN model according to the Elastic Kernel [31] to enable GPU partition on each kernel (Section 8). The transformed model code is stored in the *model pool* for reuse. Arriving requests are then routed to a GPU node where a function instance is launched to execute the model code. StreamBox hooks GPU APIs (i.e., memory allocation, communication, I/O transfer, and kernel) from functions, and forwards them to an API Proxy responsible for each workflow. APIs of memory allocation are managed by *Auto-scaling Memory Pool* (Section 5); APIs of communication (provided by us) are managed by *Unified Communication Framework* (Section 6); APIs of I/O transfer are scheduled by *IO Daemon* (Section 7); APIs of launching kernel are directly passed to the GPU driver. Note that we employ existing workflow-aware function scheduling [25, 47] (i.e., placing functions of a workflow on the same GPU node whenever possible) to optimize inter-function communication.

StreamBox in shared environments. StreamBox is ideal for functions from the same user and a trusted domain: It supports that a workflow typically has mutual trust and thus functions in a workflow do not require a strong isolation (this assumption is consistent with other serverless systems [1, 19, 21]). However, StreamBox can be used in multi-tenant scenarios. Specifically, it allows to start multiple GPU runtimes (i.e., multiple API proxy processes for each workflow or user) for isolating untrusted code or users if necessary. Note that a single GPU is typically shared by up to 3 workflows due to GPU capacity limitations. We also evaluate the performance of StreamBox in multi-tenant scenarios (Section 9.4).

5 Auto-scaling Memory Pool

To reduce memory allocation overhead and unnecessary GPU memory reservation, StreamBox builds an auto-scaling memory pool to improve memory efficiency, which enables fine-grained sharing of GPU memory among functions through layer-level memory allocation and recycling. In addition, to avoid charging users for idle GPU memory (pay-as-you-go), StreamBox utilizes offline profiling of DNN inference and builds memory usage models to make the memory pool resilient to exact memory usage of functions. Existing memory pool methods (e.g., cudaMemPool and PyTorch) can only statically cache freed memory, and lack the philosophy of auto-scaling.

5.1 Fine-grained Memory Management

Fig. 6 shows the memory management when running a DNN inference under StreamBox. We leverage that the computation of DNN models are performed sequentially (layer by layer) and that DNNs adopt static pattern to enable on-demand memory allocation and recycling for layers.

Lazy allocation. StreamBox hooks the memory allocation calls (i.e., `cudaMemAlloc()`) from each function, and registers the variable name in a *mapping table*. The mapping table records (and keeps) the mapping between variables and physical addresses, where the address of a variable is kept empty until it is accessed. For example, as shown in Fig. 6, as the computation of layer 3 has not started yet, its addresses in the mapping table are empty. When a variable is accessed for the first time (e.g., the variable is a parameter in a kernel), a physical address is allocated from the GPU memory pool. Streambox does not introduce additional address mapping; it gets valid addresses from a pre-allocated memory pool, and allocates them only when variables are accessed. This lazy allocation reduces memory occupation by idle variables.

Eager recycling. For efficient memory management, a layer and its results can be freed when the computation of the next one starts. However, unlike temporary data – that can be directly freed (with no harm) – functions can share a DNN model (multiple requests for the same DNN model may arrive), therefore direct memory recycling will result in frequent model loading. In StreamBox, while temporary results are recycled in time, we propose to cache layers in the memory pool as long as they are accessed. Specifically, given that DNNs exhibit static patterns, we propose to pre-run inference task *offline* to obtain how many times each variable will be accessed (expected frequency). We instrument the *Mapping table* to record the access count (frequency) for each variable during a sequence of kernel executions. Accordingly, when the access count of a variable reaches the expected frequency, the variable can be marked as reclaimable. A background thread periodically queries the mapping table and returns reclaimable addresses back to the memory pool. As a result, eager recycling can reduce memory occupation by useless variables.

5.2 Resilient Scaling

To make the memory pool resilient to the exact memory usage of functions in the shared runtime, StreamBox employs elastic scaling to the memory pool. This scaling is achieved through the estimation of memory requirements based on workload analysis and offline profiling.

Memory demand profiling. As described in Section 5.1, DNN inference accesses a proportion of the data over time and not all at once. Thus, considering the total accessed data when scaling the memory pool will result in significant memory waste. Instead, we pre-run each model offline to record its exact memory usage during the execution and use this data to allocate the "exact" resources needed in later execution. We measure and record the memory usage for each kernel in each DNN model with different batch sizes. Note that the memory demand is the actual memory required by the inference task after using the *lazy allocation* and *eager recycling* optimizations.

Real-time memory pool scaling. On the one hand, DNNs exhibit frequent changes in memory demands (see Fig. 15(b)), thus frequent memory pool scaling may incur high overhead. On the other hand, concurrent functions may compete for GPU resources (e.g., memory bandwidth) and therefore their execution times can vary [47], making offline memory demand prediction inaccurate. Consequently, in StreamBox, we periodically adjust the memory pool size at a fixed interval (Algorithm 1). The time interval $T_{interval}$ is the overhead of allocating and recycling 200MB of memory. That is $T_{interval} = T_{alloc} + T_{free}$. We estimate the maximum memory usage in future intervals based on offline profiling. Since the

Algorithm 1 Real-time memory pool scaling

```
Input: offline profiling of memory usage and runtime per kernel
Output: A new memory pool size $M_{resize}$
 1: while $there\ are\ functions\ running$ do
 2:     $M_c, M_{next}, m_{next} \leftarrow 0$;
 3:     for $func \leftarrow concurrent\_functions$ do
 4:         $k_c \leftarrow$ concurrent_kernel($func$);
 5:         $m_c \leftarrow$ memory_usage($k_c$);
 6:         for $k_n \leftarrow$ kernels_in_next_interval($func, T_{interval}$) do
 7:             $m_{next} \leftarrow Max(m_{next}$, memory_usage($k$));
 8:         end for
 9:         $M_c \leftarrow M_c + m_c$, $M_{next} \leftarrow M_{next} + m_{next}$;
10:     end for
11:     $M_{resize} = Memory\_pool\_size() - Max(M_c, M_{next})$
12:     sleep($T_{interval}/2$);
13: end while
```

actual execution time of kernels will only be longer than the time profiled offline, we can ensure that the memory pool size meets the actual demands. We resize the memory pool (line 11) based on the difference between the maximum memory usage in the next time interval and the current memory pool size (negative value means scaling up). This periodic approach not only senses the memory demands of functions accurately, but also avoids frequent memory allocation and recycling.

For memory allocation and recycling of the GPU memory pool, we utilize asynchronous CUDA API (i.e., `cuMemAllocAsync()`, and `cuMemFreeAsync()`) to avoid affecting streams. Additionally, we leverage existing CUDA memory pool mechanisms [42] to mitigate issues such as memory fragmentation.

6 User-transparent Communication Framework

StreamBox provides a transparent communication interface for developers to compose function workflow. The *unified communication framework* adaptively switches the communication methods between functions according to the distribution of functions in the GPU cluster. Furthermore, StreamBox employs an *elastic communication store* that can efficiently utilize the idle GPU memory to store intermediate data.

6.1 Unified Communication Framework

There are three communication methods according to the distribution of functions in GPU cluster: 1) Functions running in the same GPU can share addresses directly, enabling the fastest intermediate data transfer. 2) Functions on different GPUs can utilize the *Point-to-Point* (P2P) mechanism through high-speed NVLink to transfer data. 3) For functions on different nodes, *Remote Procedure Call* (RPC) is used.

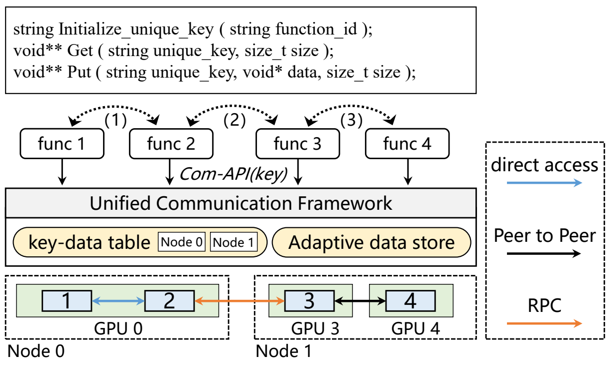
Easy-to-use communication API. As shown in Fig. 7, we provide communication APIs that conform to the external storage paradigm [3] in serverless systems. Intermediate data are transferred using PUT and GET APIs. Developers need to allocate a globally unique index to each intermediate data, which is then passed to subsequent functions. The PUT API records the index and the physical address of the data in a *mapping table* for each node in the CPU memory. When a function uses GET API to access data with an index, the corresponding communication mechanism is chosen based on the locations of functions. There are two types of mapping tables: global and per-node. The search starts with the table of the node where the function is located. If the key is not hit, then it will search the global table, which will be updated periodically (every second by default).

Optimizing intra-GPU communication. We focus on the communication between functions within the same GPU. We maintain a shared *communication store* in GPU to 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, without having to get and open the data handler as in the CUDA IPC approach. Furthermore, considering the elastic scaling of serverless workflows [42, 47], where multiple functions may simultaneously request and access the communication store, we design a producer-consumer tool to avoid competition and ensure correctness in accessing the communication store.

6.2 Elastic Communication Store

Storing data on the GPUs can speed up data transfers but consumes GPU memory. In addition, when the idle GPU memory is insufficient, intermediate data should be moved out to avoid interfering with running functions. StreamBox implements an elastic communication store to manage intermediate data residing in GPUs. Specifically, to maximize the capacity of communication store, we monitor the memory pressure on GPU in real-time to fully utilize idle memory. When the idle memory of the current GPU is insufficient, we adaptively move data between GPUs to avoid expensive data copies between GPU and CPU.

Memory pressure awareness. We scale the communication



```
string Initialize_unique_key ( string function_id );
void** Get ( string unique_key, size_t size );
void** Put ( string unique_key, void* data, size_t size );
```

Figure 7: Unified communication framework in StreamBox

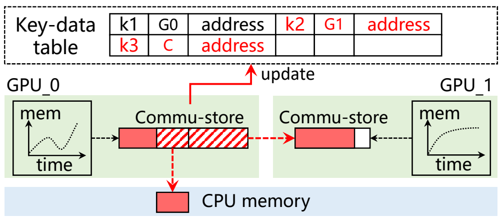


Figure 8: An example of adaptive inter-GPU movement

store based on the idle memory on each GPU. We utilize the model described in Section 5.2 to predict the GPU memory usage of running functions within a future time window (the time window is set to 20ms, which is equivalent to the transfer time of 200MB on PCIe). This allows us to host intermediate data without affecting running functions. When the memory pressure increases, the communication store triggers an early move of intermediate data. Furthermore, we use only a part of idle memory in each GPU as a communication store, so that we can handle bursty requests. Note that due to our proposed *lazy allocation*, the memory required by bursty requests is greatly reduced.

Adaptive inter-GPU movement. When the idle GPU memory of the current GPU is insufficient, we first try to move the intermediate data to neighboring GPUs (Fig. 8). Because GPUs are connected with high-speed interconnects like NVLink [18] (up to 300GB/s), which is much faster than the PCIe (12GB/s) link with the CPU. If all the GPUs in the node are insufficient, then we move the data to CPU memory. Inter-GPU data movement policies are critical to communication performance and system robustness. So we propose the following rules. **Rule 1:** *Proactively move the data being accessed remotely.* When a function uses GET API to access data that is located on a remote GPU, we prioritize moving the data to the GPU where the function is running. **Rule 2:** *Try to be even.* Intermediate data are stored on the current GPU by default. If a GPU memory is insufficient and data needs to be moved to another GPU, we try to balance GPU memory pressure and prioritize (select) GPUs with lower memory pressure. **Rule 3:** *Prioritize larger data.* If the memory pressure on the current GPU increases, we should quickly recycle enough memory for running functions. Therefore, we prioritize moving large data out. We plan to integrate function scheduling to further explore intermediate data management in the future.

7 Fine-grained PCIe Bandwidth Sharing

To enable PCIe bandwidth sharing among concurrent streams, StreamBox partitions the data of functions into smaller data blocks and uses an I/O daemon to globally schedule the transfer of these data blocks. To achieve efficient data transfers in StreamBox, we have to face four challenges. (1) ***Limited throughput***. Transfers at data block granularity incur additional system overhead and bandwidth waste; (2) ***High latency***. Transferring data between GPU and CPU relies on pinned memory [33], which incurs significant allocation overhead; (3) ***Long waiting time***. How to allow newly arrived requests to immediately claim their share of the PCIe bandwidth; (4) ***Invalid synchronization***. Since the original data has been divided into data blocks and transferred by IO daemon, the original synchronization in user program is invalid.

To address these challenges, we introduce a fine-grained IO scheduling, as shown in Fig. 9. We hook I/O transfer calls (i.e., `cudaMemcpyAsync()`) from functions and store the metadata (i.e., source address and destination address) in their *function queue*. Next, the data is divided into fixed size *blocks* and the metadata of these blocks are stored in a global *device queue*, in a round-robin manner. Newly arrived requests join the *device queue* through *Preemption Module*. Then, the IO daemon fetches data blocks according to the device queue and triggers the transfer of data blocks in turn.

7.1 Data Block

Reducing the size of the data blocks can facilitate fair sharing of PCIe bandwidth, but it can adversely impact performance and resource utilization. Using excessively small data blocks can lead to (1) high overhead when invoking transfer calls (i.e., when we divide the data into many blocks, the invocation of transfer calls for each block incurs extra overhead), and (2) waste of bandwidth. For example, in the case of V100, the peak PCIe bandwidth between CPU and GPU is 12GB/s, but the actual measured bandwidth is only 4GB/s when the data block size is set to 16KB.

Right-size of data blocks. As shown in Fig. 10 (right), we empirically find that only when the data block size exceeds 2MB, the transfer bandwidth approaches the peak bandwidth. Hence, we choose 2MB as the data block size in StreamBox. This is also the size commonly used for memory management in systems [20]. Note that we do not only perform data partitioning but also perform data fusion. For data that are smaller than block size (many small layers in DNNs that are less than 2MB), we combine multiple small layers into one data block.

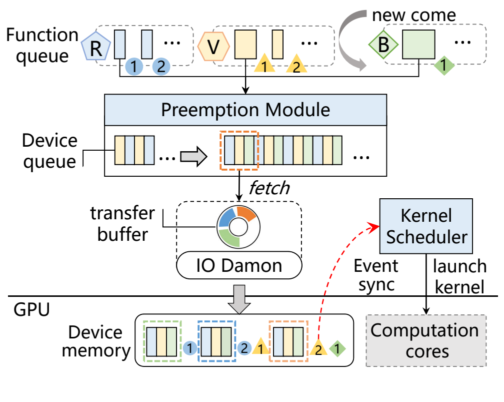


Figure 9: Fine-grained I/O scheduling in StreamBox

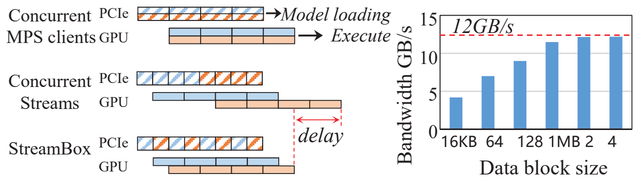


Figure 10: Left: a comparison of concurrent I/O scheduling for streams and MPS. Right: the impact of data block size on transfer bandwidth.

Preemptive transfer using batches. To avoid the extra overhead of invoking transfer calls, it is a common practice to trigger all transfer calls at once. Since the CPU program can invoke these calls asynchronously, the overhead of invoking the transfer calls can be hidden within data transfers. However, this approach will block transfer calls of newly arrived requests, as subsequent transfer calls can only begin to transfer after all the previously invoked transfers are completed. Therefore, we propose to invoke a batch of data block transfers at once, allowing newly arrived functions to be considered in the next batch of transfers. We set the batch size to six 2MB-data blocks in our experiments.

Shared pinned memory buffer. Streams transfer the data blocks to GPU memory through the pinned memory on the host. However, allocating pinned memory incurs significant overhead (200ms for 200MB). Therefore, we let functions share a buffer in the pinned memory, called *transfer buffer*.

7.2 Data Block Synchronization

Timely synchronization and quick launching of subsequent kernels are crucial for the performance. However, when we partition the data into blocks, the synchronizations in user code become ineffective, requiring us to re-build synchronization. For DNN inference with a fixed execution flow, data dependencies (i.e., layers dependencies) during computation (i.e., kernels) can be obtained through offline code analysis, as shown by the triangular symbols in Fig. 9. Therefore, after partitioning the data into blocks, we record the synchronization flags for each block. Only the last data block of each layer requires synchronization. As a result, a data block might contain multiple kernel synchronizations (i.e., multiple very small layers within a data block), or it might contain no kernel synchronization at all (i.e., a data block is only part of a larger layer). In more detail, after each data block transfer call is invoked, based on the synchronization flags of the data block, a background thread records an event (i.e., `cudaEventRecord()`) and invokes the `cudaStreamWaitEvent()` call. Then kernel scheduler continues to invoke subsequent kernels. Note that the event synchronization is asynchronously invoked and has almost no effect on data transfer and kernel invocation.

8 Implementation

StreamBox is implemented by extending Apache TVM and Nvidia CUDA with approximately 5,500 lines of C++ code.

Forwarding GPU APIs to streams. To enable functions running in containers to share one GPU runtime via streams, we leverage the widely embraced API forwarding method [10, 14] to start an *API Proxy* for each workflow. It is responsible for receiving GPU API calls (e.g., launching kernel) hooked from all functions in the workflow. Then the *API Proxy* dispatches them to streams. The API forwarding has negligible overhead as shown in Section 9.3.

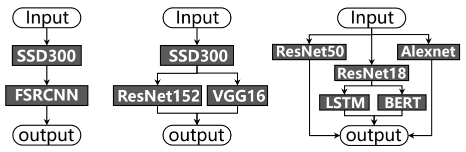
As for data (i.e., DNN model), we let functions and *API Proxy* mount a shared space. In addition, there are some CUDA APIs (e.g., `cudaDeviceSynchronize()`) that cause synchronization of all streams, so we need to replace such APIs with stream-friendly versions (e.g., `cudaStreamSynchronize()`) to prevent interference among streams.

Offline preparation. (1) *Kernel compiler*. The GPU partitioning for each kernel is implemented through a software-only solution. We extend Apache TVM to build a source-to-source compiler. It transforms all the kernels to PTB mode [31] and adds a parameter for each kernel to limit the number of GPU *Computation Units* (CUs) that can be used. The transformed kernels are compiled into customized operators in PyTorch through C++ extension [34]. (2) *Profiler*. We pre-run each DNN model under various batch sizes and input sizes to build memory usage models for our fine-grained memory management. The overhead of these offline preparations is discussed in Section 9.3.

GPU runtime management. In StreamBox, we employ the standard serverless keep-alive approach [4, 42]. It keeps the GPU runtime after a workflow request is completed and clears it if the warm-up duration exceeds the time limit (10 minutes). StreamBox supports warm-up of up to 3 GPU runtimes. Note that each GPU can handle up to 3 concurrent inference workflows, as shown in our experiments.

9 Evaluation

Setup. The experiments are conducted on a GPU server that consists of two Intel Xeon(R) Gold 5117 CPU (total 28 cores), 128GB of DRAM, and four Nvidia V100 GPUs (80 CUs and 16GB of memory). The software environment includes PyTorch-1.3.0, TensorFlow-2.12, and CUDA-10.1.



(1) *Image Process* (2) *Traffic* (3) *Social Media*

Figure 11: Three inference workflows used in evaluation

DNN inference workflows. In this paper, we study three representative DNN inference workflows (Fig. 11). The *Image Processing* [47] workflow consists of face recognition followed by image enhancement. The *Traffic* [38] workflow uses an object detection model that can identify vehicles and people. It then performs subsequent analysis on all relevant images for vehicle and face identification, as well as license plate extraction. The *Social Media* [11] workflow combines computer vision models with language models to translate and classify posts based on text and linked images.

Workloads. These inference workflows are invoked by dynamic invocations that are simulated based on the production trace of Azure Function [4]. The trace contains 7-day request statistics with daily and weekly patterns. There are three typical types of request arrival patterns in the production trace including sporadic, periodic, and bursty.

Comparing targets. As shown in Table 2, we compare StreamBox with *INFless* [42], *Astraea* [47], and *Stream-only*. INFless is a state-of-the-art serverless inference system implemented based on OpenFaaS and uses CUDA MPS to partition GPU for functions. Astraea is a QoS-aware GPU management system for microservices. It proposes an auto-scaling GPU communication framework based on CUDA IPC. Stream-only is a naive version of running functions on streams without any optimizations. We enable warming up for these baselines, because the significant latency of cold start could cause all requests to time out. Furthermore, inspired by Tetris [23] and PipeSwitch [5], for all systems, we use pipelined model loading and enable model sharing between functions.

9.1 Overall Performance of StreamBox

As shown in Fig. 12–14, we first evaluate total GPU memory footprint, throughput, and end-to-end latency of all systems using real-world workload.

Less memory footprint: StreamBox reduces GPU memory usage by up to 82%. Fig. 12 (left) shows the change in GPU memory usage over time. We observe that StreamBox significantly reduces GPU memory usage, especially when the workload increases and multiple functions are running concurrently. Furthermore, since warm-up method is enabled, the functions (over 1.5GB) warmed up in INFless and Astraea saturate GPU memory (16GB). Fig. 12 (right) depicts the average memory usage (i.e., the overall memory usage divided by the execution time). We can see that Streambox can reduce

Table 2: Comparing targets in evaluation

Systems	**INFless**	**Astraea**	**Stream-only**	**StreamBox**
GPU runtime sharing	✗	✗	✓	✓
GPU partiton	MPS	MPS	PTB	PTB
Memory pool	✗	✗	Static	Auto-scaling
Elastic commu-store	✗	✗	✗	✓
communication	Func-IPC	CUDA-IPC	CUDA-IPC	***Direct access***
PCIe sharing	MPS	MPS	✗	✓

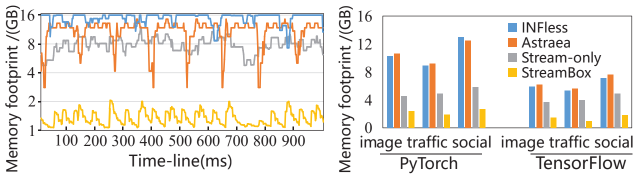


Figure 12: Left: Memory usage (log-scale) under real-world trace. Right: memory usage of different inference workflows and ML frameworks.

memory usage by up to 79% to 82% on average compared with INFless and Astraea, due to reduced redundant GPU runtimes. Astraea's memory footprint is larger than INFless due to the intermediate data cached in GPU memory.

StreamBox reduces memory footprint by 71% compared to Stream-only because of the fine-grained memory management. StreamBox allocates exactly the memory needed during DNN inference computation through *lazy allocation* and *eager recycling*. Although the memory usage varies under different ML frameworks, *StreamBox focuses on eliminating redundant GPU runtimes rather than shrinking GPU runtime*.

High throughput: StreamBox improves system throughput by 5.3X-6.7X. Fig. 13(a) and (b) show the average RPS achieved by Streambox and baselines across different ML frameworks and inference workflows. In Fig. 13(c), we further evaluate the throughput using the three types of production workloads. We observe that StreamBox achieves the highest system throughput, and improves the throughput by 6.7X and 5.3X on average compared with that of INFless and Astraea, respectively. Compared with Stream-only, StreamBox can improve the throughput by 1.46X.

As the warm-up method is ineffective on the GPU, existing serverless inference systems are still affected by cold starts, which causes a lot of request timeouts. This issue becomes more serious under *periodic* and *sporadic* workloads that incur more cold starts. PyTorch's throughput is lower than TensorFlow due to the larger cold start overhead and memory consumption, which further limits the deployment density (PyTorch's maximum function concurrency is 8, leaving over 20% of GPU cores idle and limiting warming up. In contrast, StreamBox supports 20 concurrent functions). StreamBox outperforms Stream-only due to more efficient memory management and communication. Additionally, its lower latency allows a larger batch size which further improves throughput.

Low latency: StreamBox can guarantee SLO and reduce end-to-end latency. Taking *Traffic* as an example, Fig. 14(a) and (b) plot the CDFs of the end-to-end latencies of StreamBox and baselines. We find that StreamBox achieves the lowest latency and SLO (200ms) violation. The frequent cold starts in existing serverless inference systems leads to a large number of requests waiting in queue until violating SLO. Using PyTorch shows higher SLO violation due to heavier GPU runtime, which limits the effectiveness of warming up method

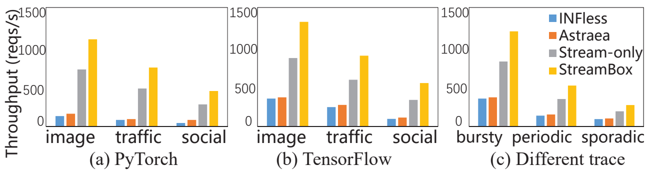


Figure 13: Comparison of throughput

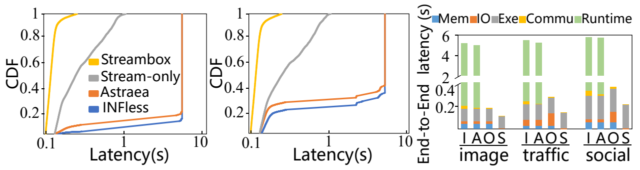


Figure 14: (a) The end-to-end latency (log scale) CDF of using PyTorch, (b) the latency CDF of TensorFlow, (c) the breakdown of end-to-end latency for INFless (I), Astraea (A), Stream-only (O), and StreamBox (S).

and concurrency of function. Furthermore, the left-most part on each curve in Fig. 14(a) and (b) shows the end-to-end latency when the function is warmed up in INFless and Astraea, StreamBox still outperforms all baselines.

Fig. 14(c) shows the breakdown of end-to-end latency. We can see that StreamBox reduces the end-to-end latency by up to 98% compared to INFless and Astraea, reducing startup latency from 5.8 s to 5 ms. Furthermore, apart from cold start optimization, StreamBox still outperforms stream-only by up to 49% on average, due to more efficient I/O (i.e., pipelined model loading), memory allocation, and communication (we will discuss these in Section 9.3). The latency of model loading is amplified in Stream-only due to serial I/O, and INFless has the highest communication latency, because it requires multiple data copies between CPU memory and GPU.

9.2 Optimizations in StreamBox

In this section, we focus on the contribution of each optimization to the performance of StreamBox, particularly compared with Stream-only.

Auto-scaling memory pool. StreamBox compresses the footprint of memory pool with fine-grained *Memory Management* (FM) and *Resilient Scaling* (RS). We perform an ablation study of these optimizations. Stream-only uses static memory pool (cudaMemoryPool). Fig. 15 (left) shows the memory usage and execution time, excluding the impact of GPU runtime. We find that StreamBox reduces memory usage by up to 91% compared to Stream-only due to the fine-grained memory management. StreamBox reduces footprint by 67% compared with stream+FM due to resilient memory pool. Furthermore, StreamBox reduces the execution time by up to 34% compared with Stream-only, because lazy allocation hides the memory allocation overhead. There is no significant perfor-

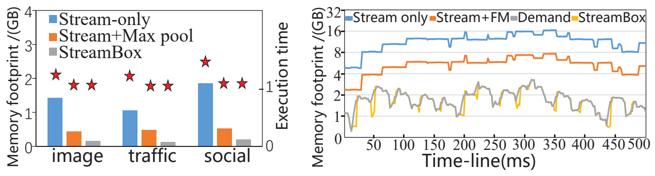


Figure 15: Left: comparison of GPU mem usage and execution time normalized based on StreamBox. Right: the mem usage (log-scale) under real-world trace.

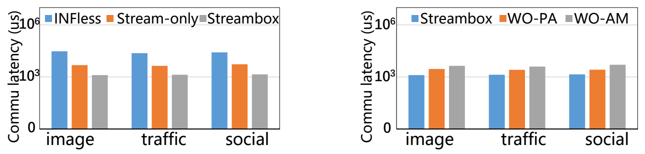


Figure 16: Left: comparison of communication latency. Right: the ablation study of StreamBox w/o and w/ optimizations.

mance difference between StreamBox and stream+FM, indicating that StreamBox reduces the memory pool size without affecting performance.

We further show the efficiency of our memory management under real-world workloads. Fig. 15 (right) shows that StreamBox can effectively reduce the memory usage compared to all baselines, and StreamBox's memory pool closely matches the actual memory demand (blue line). Due to the unpredictable bursty requests, sometimes the scaling of the pool cannot catch up with the surge of memory demand. However, the bursty requests only have overheads at the first few layers, and StreamBox's memory pool scales rapidly at layer granularity to accelerate subsequent layer allocation.

Efficient intra-GPU communication. Fig. 16 (left) shows the average communication overhead of different serverless inference systems. To increase the memory pressure, we follow the implementation in REEF [15] and run a video processing task to utilize the idle computation resource. We find that StreamBox reduces the communication overhead by 94% compared to INFless (Function-IPC), and by 73% compared to Stream-only. Function-IPC stores intermediate data in CPU memory. Stream-only lacks the management of communication buffer, leading to memory allocation overheads and data transfer between CPU and GPU when GPU memory is insufficient. StreamBox maximizes the communication buffer capacity through memory pressure aware techniques and employs adaptive inter-GPU data movement.

We perform an ablation study of memory *Pressure Awareness* (PA) and *Adaptive Movement* (AM). Fig. 16 (right) shows that PA and AM reduce the communication overhead by 55% and 72%, respectively. The reason is that PA and AM reduce the overhead of scaling communication store and prevent time-consuming data movement to CPU, respectively.

Fine-grained PCIe bandwidth sharing. To study the efficacy of I/O scheduling in StreamBox, we focus on *Traffic* and *Social media* because they involve concurrent model loading, such as concurrent car (ResNet) and face (VGG) recognition in *Traffic*. Fig. 17 (left) reports the latency of pipelined model loading for MPS (INFless), Stream-only, and StreamBox. Since we use the pipelined model loading method, model loading overhead overlaps with the computation, resulting in just a few milliseconds of latency. As expected, Stream-only significantly amplifies latency by 3.2X on average compared to INFless, because DNN models of concurrent functions can only be transmitted sequentially. StreamBox reduces latency by 91% compared to Stream-only because of efficient I/O sharing between streams. StreamBox reduces latency by 80% compared to MPS, because pinned memory buffer hides the overhead of allocating pinned memory.

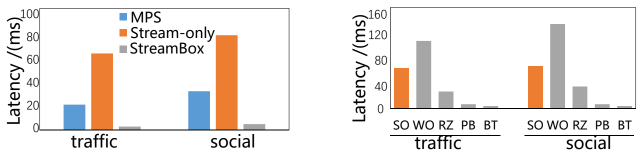


Figure 17: Left: a comparison of latency of pipelined model loading. Right: the ablation study of StreamBox w/o and w/ optimizations, where SO denotes Stream-only, and WO denotes StreamBox without optimizations.

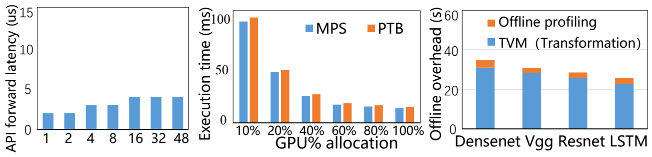


Figure 18: (a) The overhead of API forwarding, (b) the performance in GPU partition between MPS and PTB (i.e., elastic kernel) of ResNet, (c) the overhead of offline preparation.

We perform an ablation study of data block rightsize (RZ), pinned memory buffer (PB), and batch transfer (BT). Fig. 17 (right) shows that StreamBox without any optimizations is even worse than Stream-only because of wasted bandwidth caused by small data block size. Data block rightsize and pinned memory buffer bring major performance improvement, accounting for 85% of the total gain.

9.3 Overheads of StreamBox

Finally, we evaluate the overhead of API forwarding, elastic kernel, and offline preparation, and show that neither of them affects the performance of StreamBox in a significant way.

API forwarding. We need to hook GPU API calls from functions and forward them to corresponding streams. Functions and API Proxy reside on the same GPU node, thus they can communicate directly through Unix domain sockets, and we can warm up the sockets in advance. Fig. 18(a) shows that the

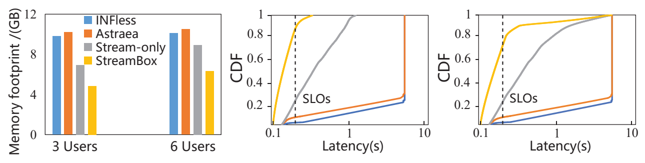


Figure 19: (a) Average memory usage under different number of concurrent users, (b) The latency (log scale) CDF of 3 users, (c) The latency CDF of 6 users.

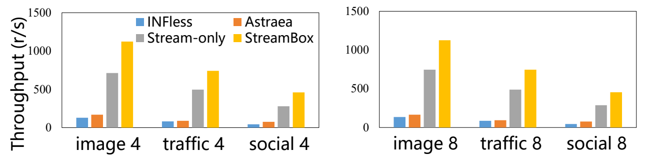


Figure 20: Left: the average throughput on cluster with 4 nodes. Right: the average throughput on cluster with 8 nodes.

overhead of API forwarding is only 1us-3us, and it remains within the range of 3us-5us as the concurrency of functions increases to 48 (the maximum concurrency on a single GPU in CUDA MPS). Furthermore, most of the GPU APIs are asynchronous, which means that the overhead of GPU API hooking and forwarding can be hidden within the execution.

Elastic kernel. Elastic kernel, also known as *Persistent Thread Block* (PTB) technology, is widely used in GPU partitioning for serverful applications [9,31,49] (i.e., multi-tasking on GPU server), kernel fusion [15,48], and compilation optimization [26]. It has been proven to improve GPU utilization without affecting performance (3% performance drop as shown in Fig. 18(b)).

Offline preparation. StreamBox relies on the offline profiling to get memory usage pattern under various batch sizes and input sizes. Furthermore, the source code needs to be transformed to PTB mode. As inference models are static and repeatedly invoked, the offline preparation only incurs an one-time cost and can be reused for later requests. Fig. 18(c) shows that the average time for offline profiling and transformation is 36s and 4s, respectively. Previous studies (e.g., Tetris [23] and REEF [15]) also mention that the overhead of offline preparation does not affect the performance of inference system, and that their overhead can reach 12 mins.

9.4 StreamBox at Scale

Multi-tenant performance. We take *Traffic* workflow as an example, and simulate 3 and 6 users using the workload from the production traces [4]. We let these users share a single GPU. Fig. 19(a) shows the average memory usage under different numbers of users. Although the memory usage under streambox slightly increased, it is still the lowest. As shown in Fig. 19(b) and (c), StreamBox experiences unavoidable cold starts when there are too many users on a GPU. Moreover, as the number of concurrent users exceeds the capacity of a single GPU (up to 3 users in our experimental setup), excessive requests are queued, resulting in SLO violations. Nevertheless, StreamBox's curve remains on the leftmost in the graph. In summary, StreamBox's effectiveness in reducing GPU runtime redundancy diminishes in a multi-tenant environment. However, it still mitigates data redundancy and cold starts as functions within a workflow share the same runtime.

Cluster Performance. We enlarge the workload proportionally based on the number of GPU nodes. Fig. 20 shows the average throughput per GPU. We find that Streambox's throughput keeps stable when increasing the number of nodes and is still higher than that of INFless and Astraea.

Table 3: State-of-the-art serverless inference systems

Systems	Hardware	Runtime	Scheduling	Opimizations
Batch [2]	CPU	container	VCPU	batch size, cost
Tetris [23]	CPU	container	VCPU	model redundancy
FaaSwap [44]	GPU	Context	Temporal	context switch
INFless [42]	GPU/CPU	MPS	Spatial	batch size, warm-up
Llama [36]	GPU	MPS	Spatial	batch size, throughput
Astraea [47]	GPU	MPS	Spatial	batch size, GPU sharing
StreamBox	***GPU***	***Stream***	***Spatial***	***light GPU sandbox***

10 Related Works

Serverless inference. Existing serverless inference systems (Table 10) generally focus on GPU pooling [14], or optimizing batch size and computation resource allocation to improve throughput [2, 36, 42, 45, 47]. They typically address the cold start of heavy GPU runtime through pre-warming. StreamBox can be integrated into the above systems. Tetris [23] reduces the CPU-side memory footprint through eliminating DNN model redundancy and sharing container runtime, while StreamBox optimizes GPU-side redundant runtimes. FaaSwap [44] achieves fast switching among functions through model swapping and GPU remoting, while StreamBox focuses on spatial GPU sharing and supports concurrent functions within one GPU runtime using streams.

Workflow-friendly isolation in serverless. It is a common practice to weaken the CPU-side isolation between functions in a workflow for better performance [1, 19, 21, 39]. Faaslane [21] and Nightcore [19] utilize threads to run functions within a workflow, sharing the same address space for efficient data sharing. However, they do not address the redundancy in GPU runtimes.

Other GPU types and GPU virtualization. AMD GPUs also support streams, and the mapping of kernels to compute units can be achieved by AMD's CU Masking API [10] without the need for kernel recompilation. Secure isolation required by different workflows or users is provided by GPU virtualization (e.g., MIG), which is orthogonal to our work. Streambox leverages the widely embraced API forwarding method [10, 14] to facilitate GPU runtime sharing among functions within a workflow.

Fast task switch on GPU clusters. Many recent systems achieve fast initialization of DNN workloads on GPU clusters. REEF [15] enables fast preemption of real-time GPU task. PipeSwitch [5] enables a fast context switch for real-time inference by pipelined model loading. They focus on task initialization (model loading), which can be integrated into StreamBox, but they do not address GPU runtime cold start.
Memory management on GPU clusters. Some systems focus on memory management of DNN workload on GPU clusters, such as DeepUM [20], HUVM [6], and DeepPlan [38]. Nvidia and AMD also introduce the *Unified Virtual Memory* (UVM). However, they achieve GPU memory oversubscription and are coarse-grained for serverless. StreamBox provides fine-grained management for auto-scaling functions.

11 Conclusion

The monolithic GPU runtimes used in existing serverless inference systems are too heavy for short-lived functions, leading to a high cold start latency, a large memory footprint, and expensive communication. StreamBox uses streams to reduce the redundant GPU runtimes and efficiently share memory between functions. It introduces a series of stream enhancements based on the characteristics of serverless functions to achieve auto-scaling memory pooling, intra-GPU communication, and fine-grained PCIe sharing. Experimental results demonstrate the efficacy of StreamBox.

12 Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under grant 2022YFB4500704 and in part by the National Science Foundation of China under grants 62032008 and 62232011. Shadi Ibrahim and Hao Fan are the corresponding authors.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference*, pages 923–935, 2018.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 123–135, 2020.

[3] AWS S3. https://aws.amazon.com/s3/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[5] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: fast pipelined context switching for deep learning applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 499–514, 2020.

[6] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. Memory harvesting in multi-gpu systems with hierarchical unified virtual memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 625–638, 2022.

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing. In *Proceedings of the USENIX Annual Technical Conference*, pages 199–216, 2022.

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: an sla-aware batching system for cloud machine learning inference. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 493–506, 2021.

[9] Yujeong Choi and Minsoo Rhu. Prema: a predictive multi-task scheduling algorithm for preemptible neural processing units. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 220–233, 2020.

[10] Marcus Chow, Ali Jahanshahi, and Daniel Wong. Krisp: enabling kernel-wise right-sizing for spatial partitioned gpu inference servers. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 624–637, 2023.

[11] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, page 477–491, 2020.

[12] Aditya Dhakal, Sameer G. Kulkarni, and Kadangode Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the ACM Symposium on Cloud Computing*, page 492–506, 2020.

[13] Docker. https://www.docker.com/.

[14] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. DGSF: disaggregated gpus for serverless functions. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 739–750, 2022.

[15] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 354–369, 2022.

[16] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N. Sainath. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[17] Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil Dutt, and Jong-Chan Kim. R-tod: real-time object detector with minimized end-to-end delay for autonomous driving. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–204, 2020.

[18] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the ACM European Conference on Computer Systems*, page 249–265, 2023.

[19] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 152–166, 2021.

[20] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. Deepum: tensor migration and prefetching in unified memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 207–221, 2023.

[21] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: accelerating function-as-a-service workflows. In *Proceedings of the USENIX Annual Technical Conference*, pages 805–820, 2021.

[22] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: lightweight and parallel gpu task scheduling for deep learning. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 625–637, 2020.

[23] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: memory-efficient serverless inference through tensor sharing. In *Proceedings of the USENIX Annual Technical Conference*, pages 125–142, 2022.

[24] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: a technical primer for design architecture. *ACM Computing Surveys*, 54(10):1–34, 2022.

[25] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 782–796, 2022.

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 881–897, 2020.

[27] NVIDIA MPS. https://docs.nvidia.com/deploy/mps/.

[28] NVIDIA TensorRT. https://github.com/NVIDIA/TensorRT.

[29] NVIDIA Triton. https://developer.nvidia.com/triton-inference-server.

[30] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: rapid task provisioning with serverless-optimized containers. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, 2018.

[31] Sreepathi Pai, Matthew Thazhuthaveetil, and Ramaswamy Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, page 407–418, 2013.

[32] Reid Pinkham, Andrew Berkovich, and Zhengya Zhang. Near-sensor distributed dnn processing for augmented and virtual reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):663–676, 2021.

[33] Pinned Memory. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#pinned-memory.

[34] Pytorch CPP Extension. https://pytorch.org/tutorials/advanced/cpp_extension.html#.

[35] Pytorch Hub. https://pytorch.org/hub/.

[36] Francisco Romero, Mark Zhao, Neeraja Yadwadkar, and Christos Kozyrakis. Llama: a heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, page 1–17, 2021.

[37] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the USENIX Annual Technical Conference*, pages 205–218, 2020.

[38] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles*, page 322–337, 2019.

[39] Simon Shillaker and Peter Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Annual Technical Conferenc*, pages 419–433, 2020.

[40] Yuxuan Wang, Ryan Skerry, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, and Samy Bengio. Tacotron: a fully end-to-end text-to-speech synthesis model. *arXiv preprint arXiv:1703.10135*, 164, 2017.

[41] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 945–960, 2022.

[42] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 39–50, 2022.

[43] Juheon Yi and Youngki Lee. Heimdall: mobile gpu coordination platform for augmented reality applications. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[44] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2023.

[45] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proceedings of the USENIX Annual Technical Conference*, pages 1049–1062, 2019.

[46] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: serving DNNs in the wild. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 787–808, 2023.

[47] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. Astraea: towards qos-aware and resource-efficient multi-stage gpu services. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 570–582, 2022.

[48] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. Tacker: tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 800–813, 2022.

[49] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. Miriam: exploiting elastic kernels for real-time multi-dnn inference on edge gpu. *arXiv preprint arXiv:2307.04339*, 2023.