FUTUREWEI Technologies

USENIX ATC'22 Report

July 11-13, 2022

Norbert Egi
Infrastructure Software Lab

USENIX Annual Technical Conference (ATC) 2022

- July 11-13, 2022, Carlsbad, CA
- 394 submissions, 64 accepted papers (16.5% acceptance rate)
- Most popular submission topics (some papers were):
  - Distributed Systems (26%)
  - Storage (24%)
  - Machine Learning (21%)
  - Operating Systems (15%)
  - Networking (14%)
  - Databases (13%)
  - Security (13%)

Silver Sponsors

PagerDuty

VMWARE UNIVERSITY RESEARCH FUND

General Sponsors

Google

TWO SIGMA

Bronze Sponsors

Alibaba Group 阿里巴巴集团

Meta

Open Access Sponsor

NetApp

FUTUREWEI Technologies

ATC'22 Technical Sessions

TRACK 1

- Storage 1
- Distributed Systems 1
- Operating Systems 1
- Networking 1
- Security
- Distributes Systems 2
- Deployed Systems 1
- Storage 2
- Compilers and PL
- Storage 3
- NICs
- Deployed Systems 2

TRACK 2

- Containers
- Machine Learning 1
- Disaggregated Systems
- Finding Bugs
- Machine Learning 2
- Operating Systems 2
- Machine Learning 3
- Networking 2

ATC'22 Best Paper Awards

- **Riker: Always-Correct and Fast Incremental Builds from Simple Specifications,** Charlie Curtsinger, Grinnell College; Daniel W. Barowy, *Williams College*
    - Riker is a forward build system (to improve correctness over traditional build tools by discovering dependencies automatically) that guarantees fast, correct builds. Riker automatically discovers fast incremental rebuild opportunities. Riker models the entire POSIX filesystem—not just files, but directories, pipes, and so on. This model guarantees that every dependency is checked on every build so every output is correct.

- **Co-opting Linux Processes for High-Performance Network Simulation,** Rob Jansen, U.S. Naval Research Laboratory; Jim Newsome, Tor Project; Ryan Wails, *Georgetown University, U.S. Naval Research Laboratory*
    - This paper presents the design and implementation of Phantom, a novel tool for conducting distributed system experiments. In Phantom, a discrete-event network simulator directly executes unmodified applications as Linux processes and innovatively synthesizes efficient process control, system call interposition, and data transfer methods to co-opt the processes into the simulation environment. Our evaluation demonstrates that Phantom is up to 2.2× faster than Shadow, up to 3.4× faster than NS-3, and up to 43× faster than gRaIL in large P2P benchmarks while offering performance comparable to Shadow in large Tor network simulations.

FUTUREWEI Technologies

Selected Papers of Interest

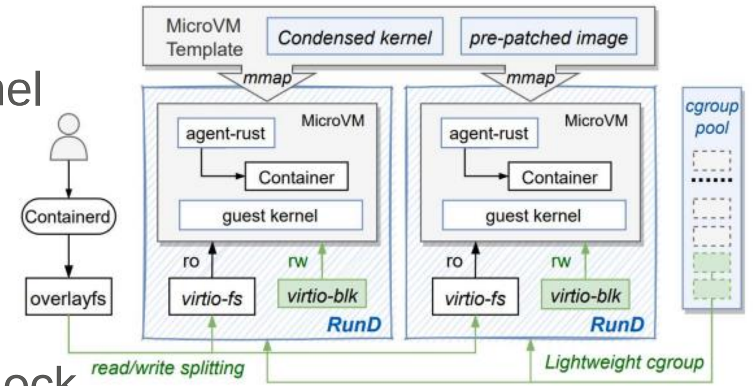
- RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing
- KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing
- Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing
- Direct Access, High-Performance Memory Disaggregation with DirectCXL
- Tetris: Memory-efficient Serverless Inference through Tensor Sharing
- Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training
- Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory
- Whale: Efficient Giant Model Training over Heterogeneous GPUs
- Cachew: Machine Learning Input Data Processing as a Service
- SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing

Challenge: No viable solution for *Strong-Isolation* + *Low-Overhead* multi-tenant container environments with (i) high-concurrency container *startup* and (ii) high-density container *deployment*;

RunD's 3 main optimizations:

- ***Read-write splitting rootfs***
  - User-provided images are read-only for OS
  - User-generated data does not need to be persisted
- ***Condensed kernel and pre-patched image***
  - Condense the guest kernel to build serverless-customized kernel
  - Generate a pre-patched kernel image for template startup
- **Lightweight cgroups with cgroup pool**
  - Aggregates all subsys into one single dedicated one
  - "cgroup rename", as a special case, does not need any global lock
  - Pre-create and maintain lightweight cgroups in a pool



- *Step 1: containerd -> RunD runtime*
- *Step 2: runc-container rootfs (ro and rw) -> VMM.*
- *Step 3: MicroVM template -> sandbox.*
- *Step 4: lightweight cgroup -> attached to sandbox.*

$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ *Guest-to-Host optimizations*

KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing

Challenge: creating RDMA connections is slow, i.e. user must create RCQP (control-plane reliable-connected queue-pair), which can take a very long time (10s ms instead is us) – mostly an issue for elastic apps (not for traditional applications, such as, RDMA-enabled databases, filesystems, scientific applications)

Key reasons:

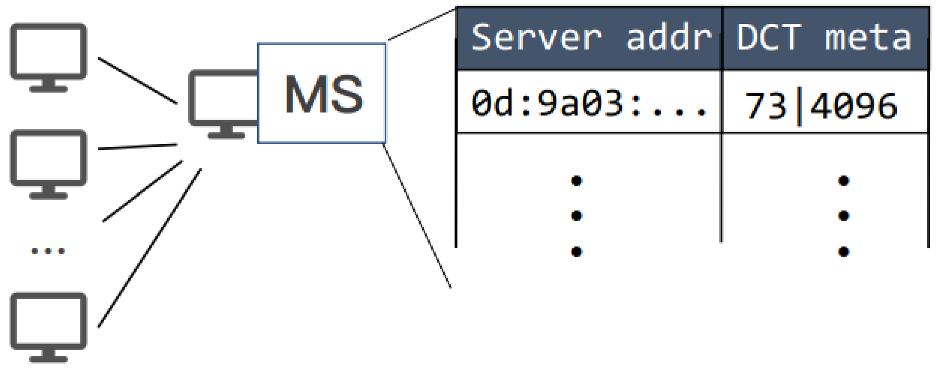
1. Loading the driver context at the user-space
2. Creating and configuring the hardware queues

Solution: Cache RCQPs in a connection pool

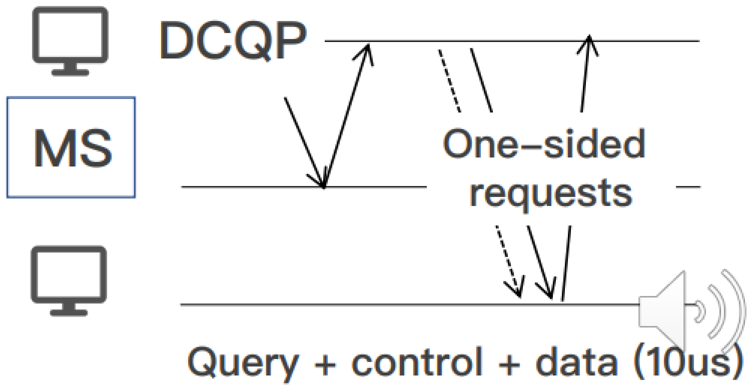
- The QPs in the pool can be **reused** by future applications with no connection cost
- **Challenge #1**: User-space QPs cannot be shared (not designed for sharing among applications)
- **Solution #1**:
  - Share QPs in a kernel-space QP pool (thus, different applications can share the same QP)
  - Kernel-space RDMA driver also support full-fledged RDMA
  - This also prevents user-space driver loading (i.e. kernel can pre-load all the driver context during boot time)

KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing

- **Challenge #2**: Massive QPs cached in the pool (RCQPs are 1-to-1 mapping, i.e. M x N QPs cached)
- **Solution #2**:
  - Dynamically Connected Transport (DCT): less-used (but widely supported) advanced RDMA transport
  - DCQP can connect to multiple nodes w/o user connection (hw can piggyback the connection extremely fast)
  - Retrofit DCT as the shared connection → MetaServer
    - Dedicate few nodes in the cluster to store the DCT metadata
    - A separate architecture allows metadata to be queried with one-sided RDMA
    - Each machine maintains QPs connected to nearby MetaServers
- **Challenge #3**: Shared QPs can be corrupted by various reasons
- **Solution #3**: Add additional checks to prevent QP corruptions
- **Github**: https://github.com/SJTU-IPADS/krcore-artifacts

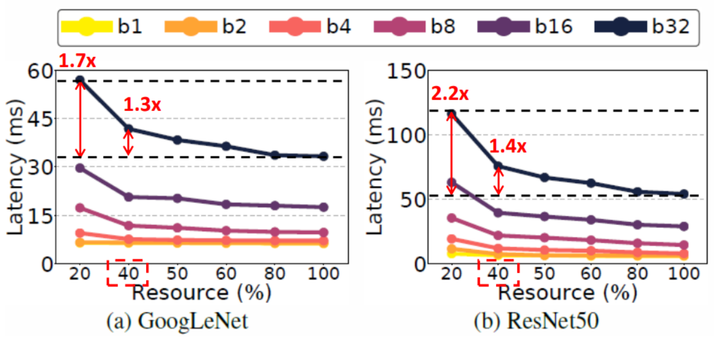


Server addr	DCT meta
0d:9a03:...	73|4096
⋮	⋮



Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

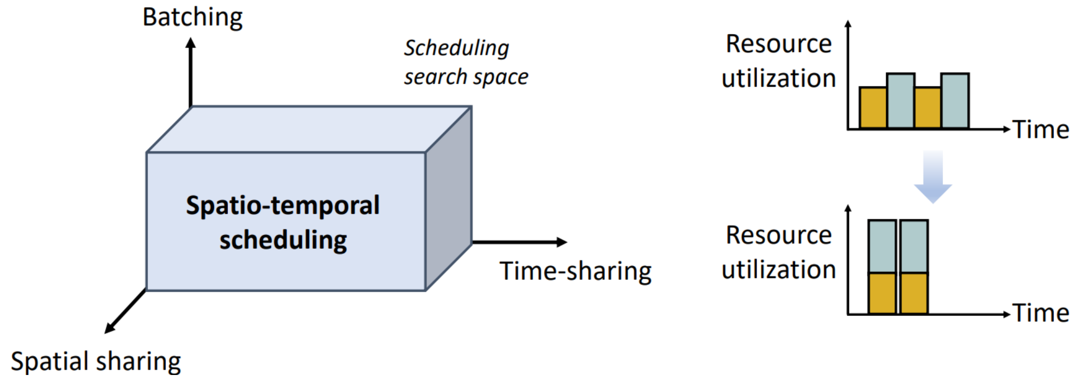
Challenge: Current approaches of GPU resource sharing (i.e. batching and time-sharing) underutilize GPUs and as such do not meet the stringent requirements of ML inference (i.e. (1) SLOs and (2) multiple heterogeneous ML models)



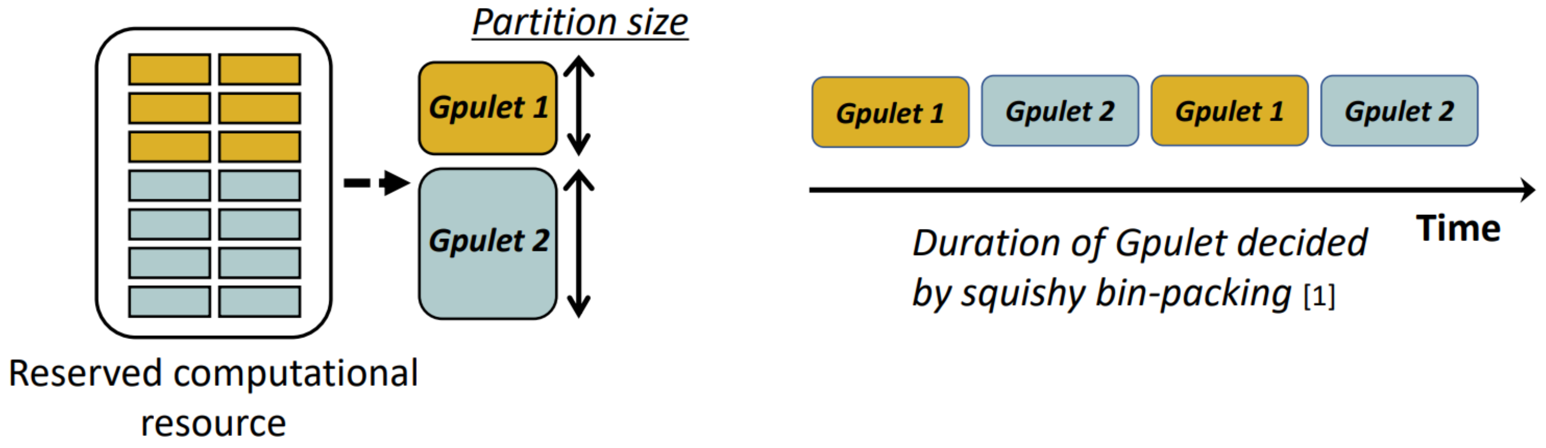
Diminishing return **beyond 40%**

Little improvement in **smaller batch** sizes

Solution: Spatio-temporal scheduling: schedule tasks with batching, time-sharing, **and spatial sharing**



Approach: Need an abstraction of spatial/temporal resource → **Gpulet**: *A share of spatial/temporal partition of GPU resource*

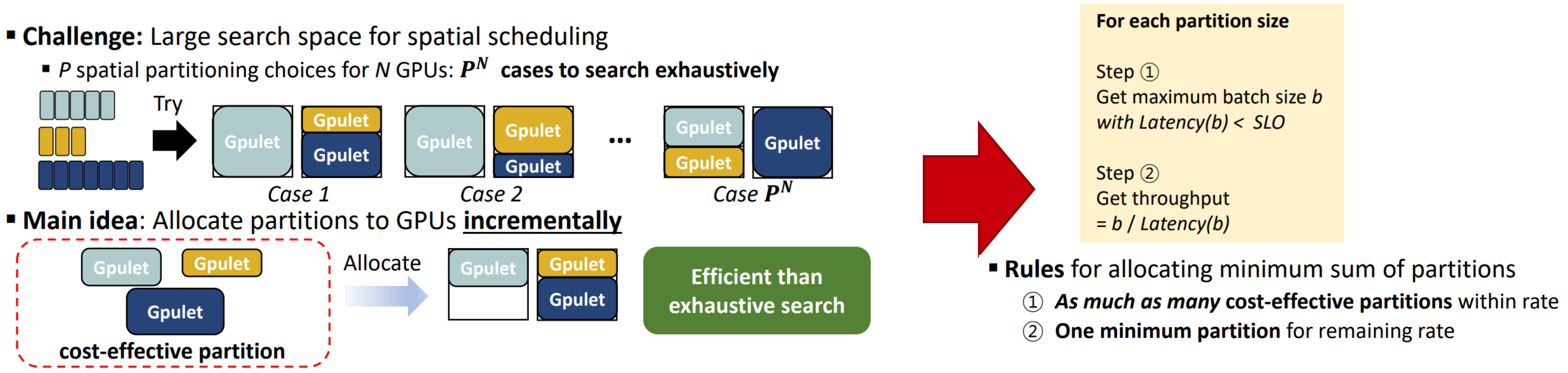

Partition size
Gpulet 1
Gpulet 2
Reserved computational resource
Gpulet 1
Gpulet 2
Gpulet 1
Gpulet 2
Time
Duration of Gpulet decided by squishy bin-packing [1]

Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

Design Overview and Objectives of Gpulet Scheduler

- (A) Cost-effective scheduling
  - Careful resource selection to (1) maximize performance and (2) minimize resource usage
- (B) Dynamic reorganization
  - Achieve scalability and minimize overhead of reorganization
- (C) Interference prediction
  - Predict interference when more than two models share the same resources

(A)



- **Challenge:** Large search space for spatial scheduling
  - *P* spatial partitioning choices for *N* GPUs: P^N **cases to search exhaustively**
- **Main idea**: Allocate partitions to GPUs **incrementally**

For each partition size

Step ①
Get maximum batch size *b* *with Latency(b) < SLO*

Step ②
Get throughput
= b / Latency(b)

- **Rules** for allocating minimum sum of partitions
  - ① ***As much as many* cost-effective partitions** within rate
  - ② **One minimum partition** for remaining rate

(B) **Challenge:** Large overhead (i.e. loading kernels, warming up models) exists for preparing a new Gpulet

Solution: Hide overhead by shadowing in the background

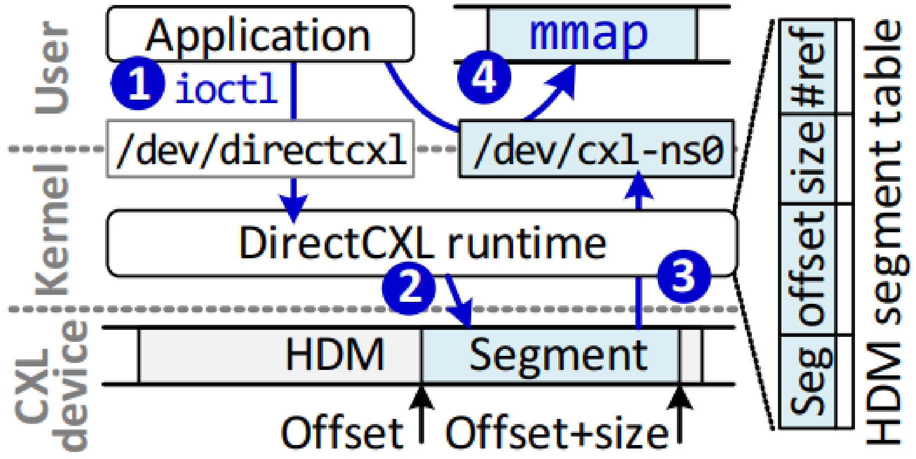
FUTUREWEI Technologies

Direct Access, High-Performance Memory Disaggregation with DirectCXL

Challenge: RDMA-based remote memory capabilities such as caching pages and network-based data exchanges are not suited for memory disaggregation as they significantly deteriorate the performance. DirectCXL on the other hand directly connects remote memory resources to the host's computing complex and allows users to access them through sheer load/store instructions. However, there is no software runtime that would support efficient operation.

Solution: Build a software stack that enables applications to use disaggregated memory through cxl-namespaces.

With DirectCXL, once a virtual hierarchy is established between a host and CXL device(s), applications running on the host can directly access the CXL device by referring to HDM's memory space. However, the software runtime/driver to be built has to manage the underlying CXL devices and ***expose their HDM in the application's memory space***. In this design, the DirectCXL runtime ***simply splits the address space of HDM into multiple segments, called cxl-namespace***. The runtime then allows the applications to access each CXL-namespace as memory-mapped files (mmap).



DirectCXL software runtime

Tetris: Memori-efficient Serverless Inference through Tensor Sharing

Challenge: Current serverless inference platforms are highly memory-inefficient, mainly due to high number of instances being launched (one per request load), and also the number of parameters are huge.

Root cause:

- Large amount of data redundancy in memory, caused by the multi-instance nature of serverless platforms (e.g. AWS Lambda creates a new instance for each request if there are no hot instances). As such, these instances are sandboxed and if the same model parameters are loaded in different instances they have to replicated, causing high memory redundancy.
- Early-provisioning and long keep-alive of instances to reduce cold starts (e.g. 15-60 mins in AWS Lambda)

SOTA approaches:

- Runtime sharing: processing multiple requests within a single instance (it reduces memory redundancy by decreasing the number of launched instances)
  - Batching
    - Grouping and processing requests in batch
  - Multi-threading
    - Processing requests concurrently

Tetris: Memory-efficient Serverless Inference through Tensor Sharing

Challenge: Even with runtime sharing, parameterized tensors are still redundant (which comprise the bulk of data in the memory during inference, e.g. VGG19 – 93%).

Tensor redundancy exists across distinct functions:

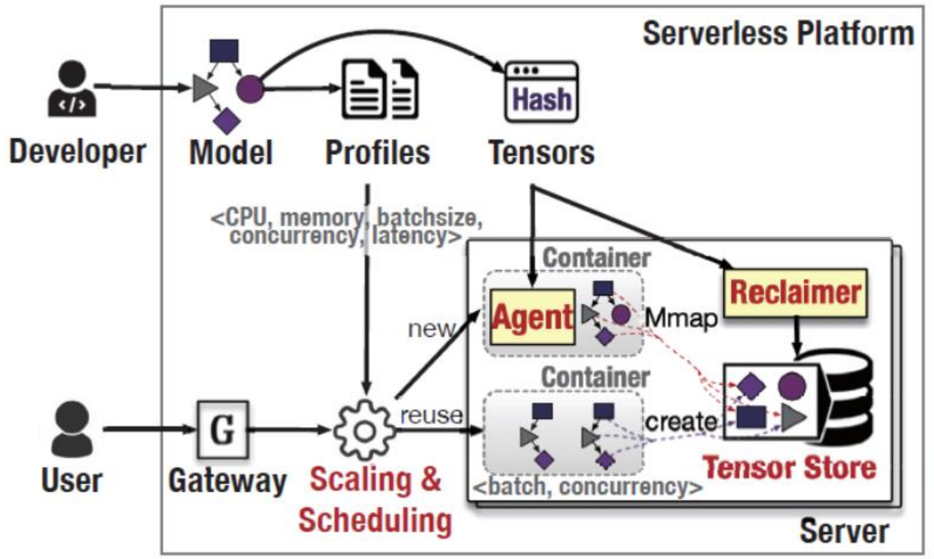
- The same model used in distinct model pipelines
- Different downstream models retrained from the same pre-trained parameters

Main contribution:

- Observation of the tensor redundancy problem
- A lightweight and user-space solution that eliminates the tensor redundancy through tensor sharing

System Overview:

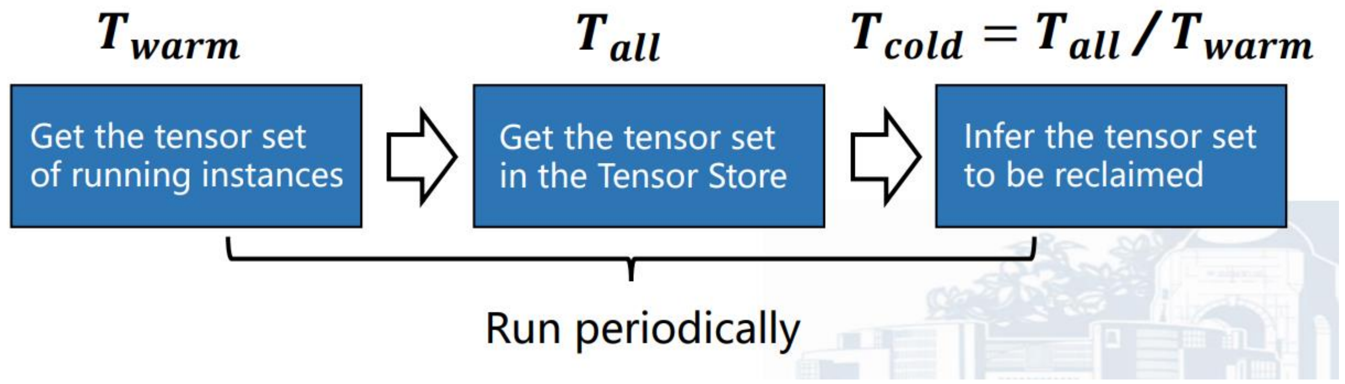
- TETRIS improves memory efficiency can be improved through a combined optimization of **runtime sharing** and **tensor sharing**
- It improves memory efficiency in 3 steps:
    1. By enabling tensor sharing on each server
    2. By carefully scheduling instances across servers to share more tensors
    3. Scaling fewer instances to serve requests under SLO constrains



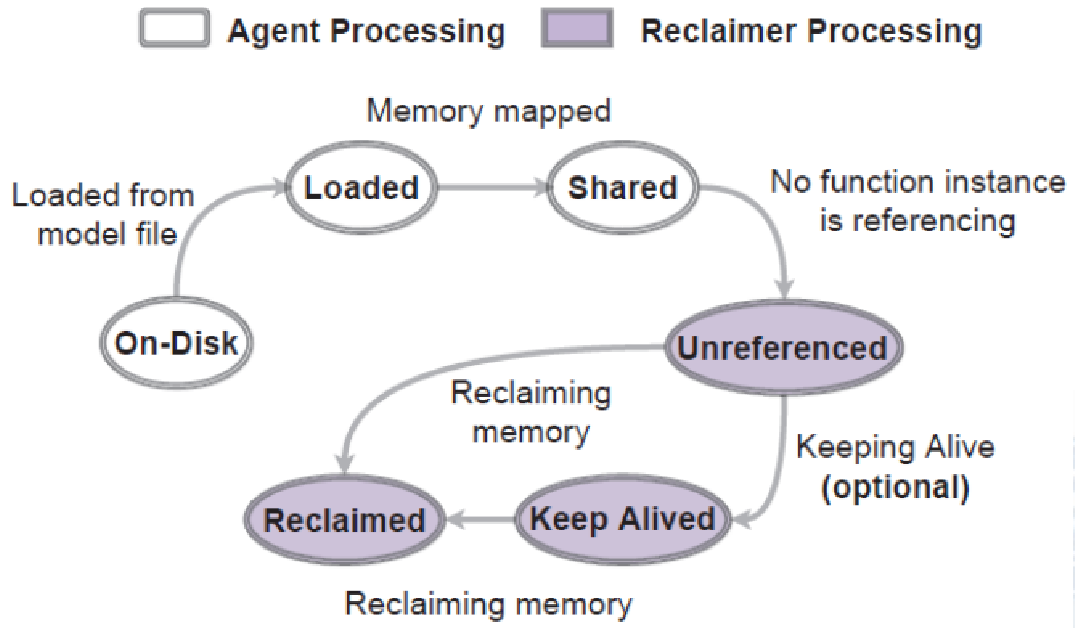
Tetris: Memory-efficient Serverless Inference through Tensor Sharing

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

- First, make a shared memory region across function instances **(The Shared Tensor Store)** (implemented by mounting a shared tmpfs to each container)
- Second, take over the model loading process of function instances and put tensors into the shared region **(The Agent)**
- Third, make tensors in the shared region to be reclaimed correctly **(The Reclaimer)**


T_{warm}
T_{all}
$T_{cold} = T_{all} / T_{warm}$
Get the tensor set of running instances
Get the tensor set in the Tensor Store
Infer the tensor set to be reclaimed
Run periodically

- The lifecycle of tensors



Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Challenge: Using lower precision for NN training is beneficial.

- Lower arithmetic complexity → Higher performance
- Less GPU memory footprint → Larger model training / Save memory bandwidth / Enable larger batch size

However, if not used correctly it can lead to adverse side effects:

- Numeric overflow/underflow
- Uncontrolled model accuracy loss

Improvement: Mixed-precision – mixture of FP16 & FP32 (FP32 handle numerically-dangerous computation)

TensorFlow / PyTorch categorize operations (ops) based on their numerical-safety levels

- Allowlist
  - Numerically-safe execution in FP16, is also performance-critical → Always FP16 (e.g. MatMul, Conv2D)
- Denylist
  - Numerically-dangerous in FP16 & their effects may also be observed in a downstream operation → Always FP32 (e.g. Exp, SoftMax)
- Inferlist
  - Numerically-safe in FP16, but may be made unsafe by and upstream denylist operation (e.g. BiasAdd)
- Clearlist
  - No numerically-significant effects (FP16 and FP32) (e.g. Max and Min)

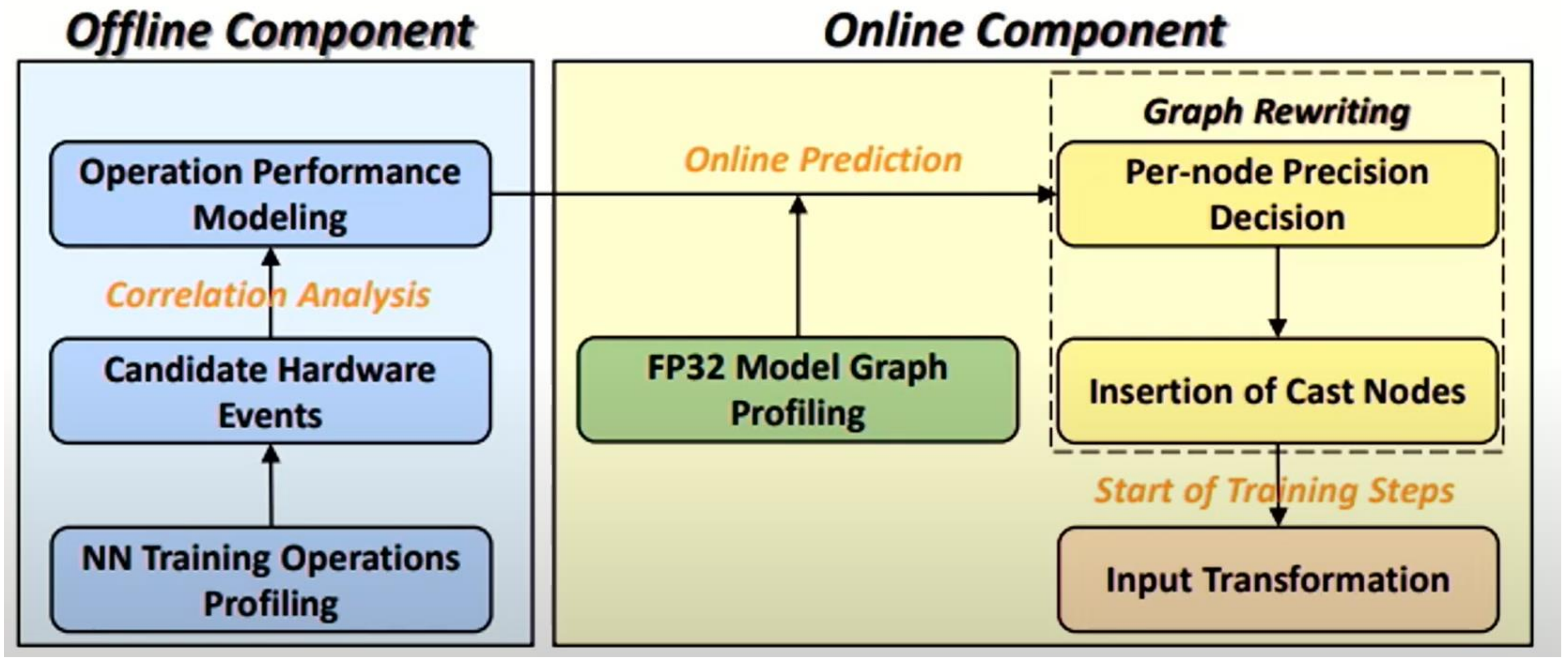
FUTUREWEI Technologies

Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Goals:

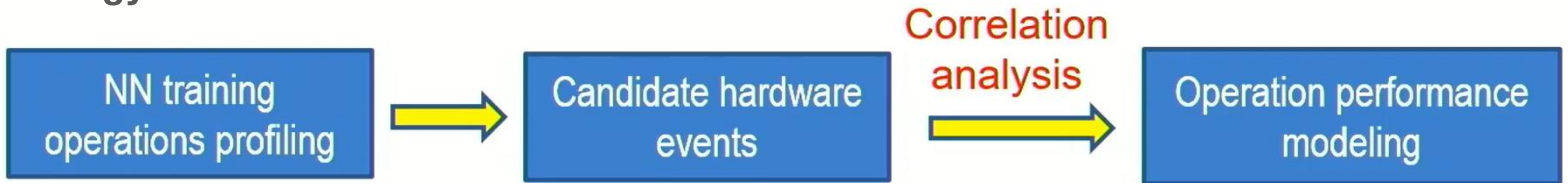
- Examine the casting cost assumptions
- Study the impact of tensor precision, tensor cores (TC), casting cost, and input data size on operation performance
- Provides insights on how to use mixed precision training

Architecture Overview:



Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Performance modelling: methodology


NN training operations profiling
Candidate hardware events
Correlation analysis
Operation performance modeling

- Predict casting cost and operators' performance in FP16
- Use lightweight regression-based modeling
- Operation-specific performance modeling

Performance modelling: casting cost

- Casting cost is related to input tensor size and CastOp instance initialization time

$$casting\_cost = r * tensor\_size + C_I$$

Performance modelling: operators in FP16

- Capture the performance correlation between FP32 and FP16

$$OP_{LP} = OP_{FP32} \cdot \left(\sum_{i=1}^{N} w_i \cdot PC_i\right) + \sigma$$

FUTUREWEI Technologies

Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

TODO:

- Decide data precision for each operation
- Decide which operations to be converted together to reduce number of cast operation nodes

GOALS:

- Minimizing the training time
- Minimizing the casting cost
- No adverse impact on the numerical safety (compared with traditional mixed-precision training)

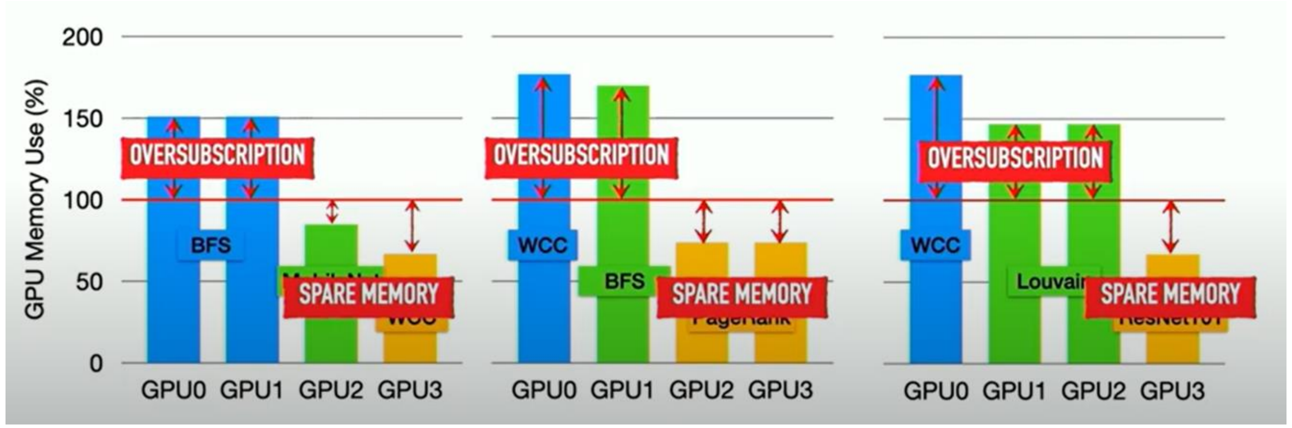
METHOD: Runtime graph rewriting: four graph traverses

- ***Traverse 1***
    - Are the nodes in *allowlist* (i.e. numerical-safe nodes)?
    - The performance benefit is larger than the casting cost?
- ***Traverse 2***
    - Check numerically-unsafe nodes → No hope for conversion
- ***Traverse 3***
    - Check if remaining nodes are in *inferlist* or *clearlist* → Determine their numerical-safety if converting
- ***Traverse 4***
    - For those numerically-safe nodes identified in the last traverse, the performance benefit is larger than the casting cost?

Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

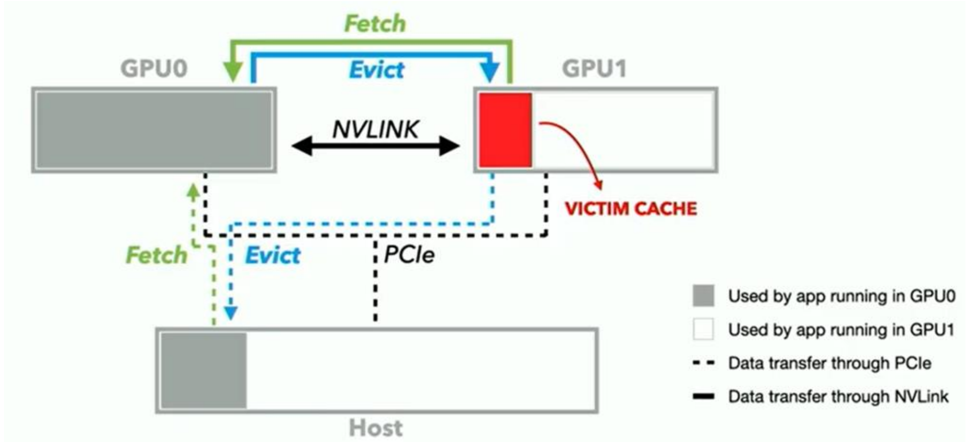
Challenge:

- Different applications can share multi-GPU servers to reduce infrastructure cost.
- As such, workloads are consolidated to improve resource utilization.
- However, memory space in such environments is not fully utilized. Some memory space stays idle.
- Each workload has highly varying memory demands
- GPU memory is not fully utilized in shared GPU clusters



Solution:

- Harvest neighbor GPU memory, by spilling a fraction of oversubscription to undersubscribed GPUs
- NVLink is well suited, it's 3x faster than PCIe (both BW and Latency)
- Also, Unified Virtual Memory (UVM) is widely supported in both Nvidia and AMD devices


Fetch
Evict
GPU0
GPU1
NVLINK
VICTIM CACHE
Fetch
Evict
PCIe
Host
▨ Used by app running in GPU0
□ Used by app running in GPU1
- - Data transfer through PCIe
— Data transfer through NVLink

Architecture: HUVM: Hierarchical Unified Virtual Memory

- New data-path with NVLink to harvest the spare memory of neighbor GPU
- Unutilized remote memory region is dedicated to be a "victim cache"
- They can evict pages to the victim cache via NVLink and fetch pages back vice versa

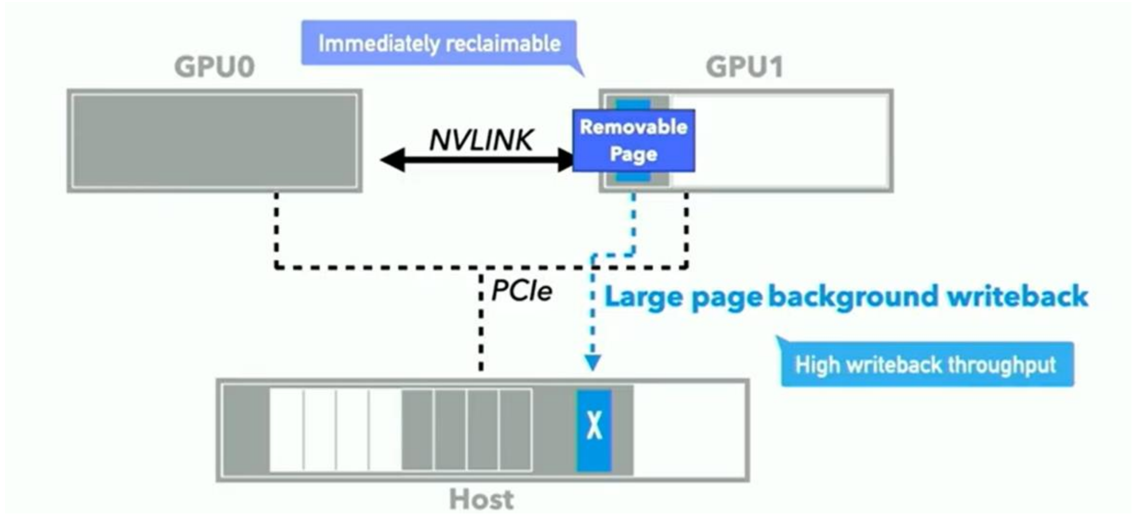
Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

Goals of HUVM: Reducing performance cost of memory oversubscription

- **Effective Harvesting**
  - Harvest small and temporarily available spare memory neighbor GPU
  - Reduce eviction/fetch latency with spare memory
- **Minimal Interference**
  - Minimize performance impact of workloads running in neighbor GPU
- **Framework-agnostic**
  - No modification of applications or frameworks

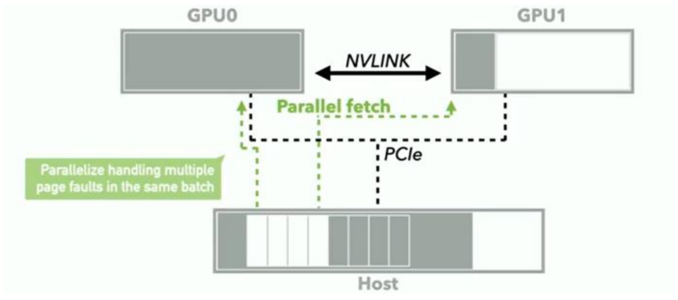
Solution: memHarvester: memory manager for HUVM – Centralized coordinator for data-path in HUVM leveraging both PCIe and NVLink

1. **Hiding eviction latency**
   - Pre-eviction to reduce eviction latency (reduces the # of on-demand evictions)
   - Background writeback operation: data is copied to host memory in the background in the case remote GPU needs more of its own memory and can't host the other GPU's pages anymore

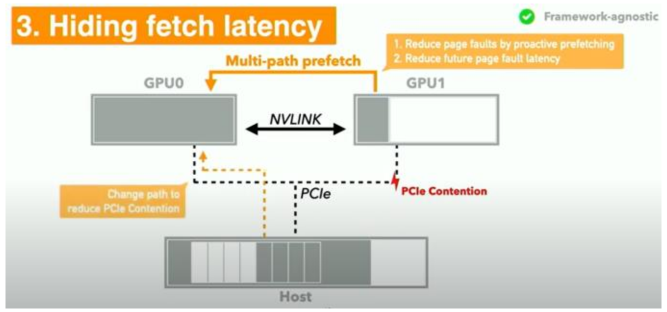


Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

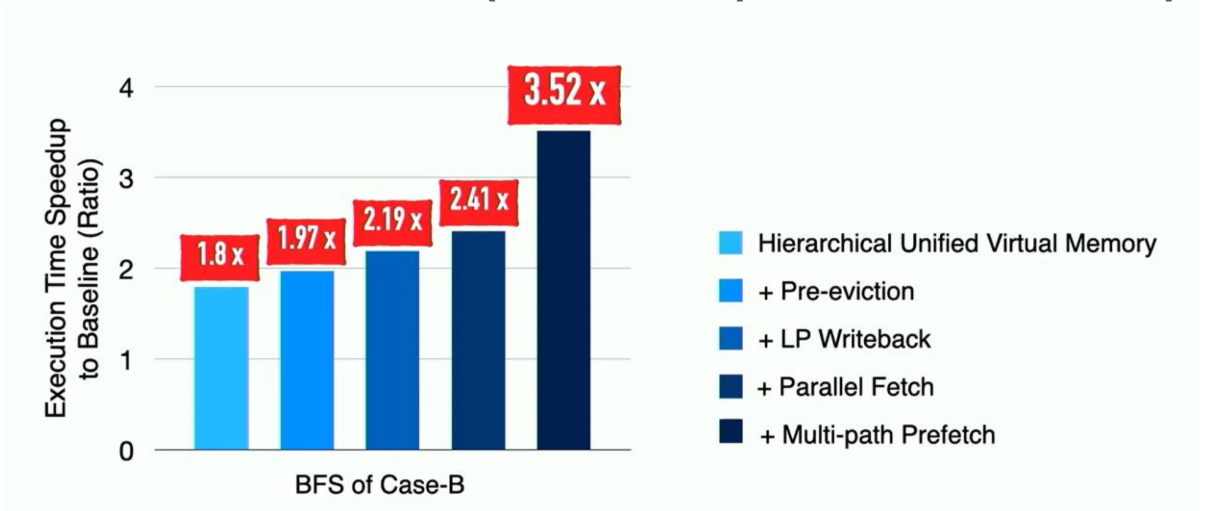
2. **Reducing fetch latency**
   - Parallel fetching: leveraging the parallel data-path with both PCIe & NVLink. As GPU applications are highly parallel, page-faults occur in batch, as such the driver should handle all page requests as well before resuming the application



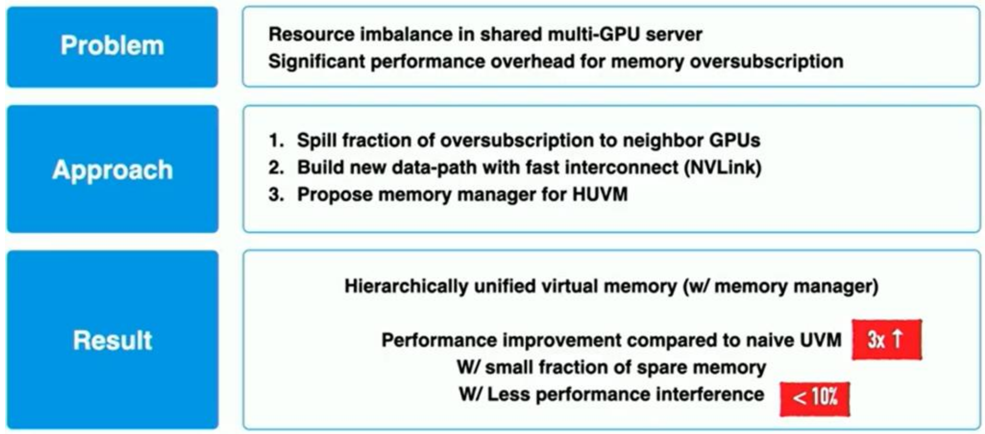
3. **Hiding fetch latency**
   - memHarvester pre-fetched with multi-path fetching leveraging the new data-path with both PCIe and NVLink to enhance parallelism
   - This in return leads to (1) reduced page faults by proactive prefetching, and (2) reduce future page fault latency


3. Hiding fetch latency
Framework-agnostic
1. Reduce page faults by proactive prefetching
2. Reduce future page fault latency
Multi-path prefetch
GPU0
GPU1
NVLINK
Change path to reduce PCIe Contention
PCIe
PCIe Contention
Host

<u>**Performance evaluation (breakdown per individual technique**</u>)



<u>CONCLUSION</u>

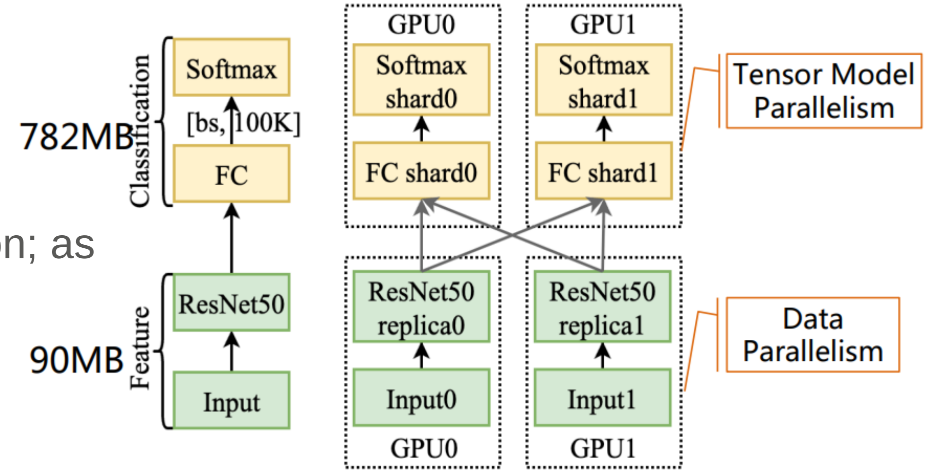


Problem	Resource imbalance in shared multi-GPU server Significant performance overhead for memory oversubscription
Approach	1. Spill fraction of oversubscription to neighbor GPUs 2. Build new data-path with fast interconnect (NVLink) 3. Propose memory manager for HUVM
Result	Hierarchically unified virtual memory (w/ memory manager) Performance improvement compared to naive UVM 3x ↑ W/ small fraction of spare memory W/ Less performance interference < 10%

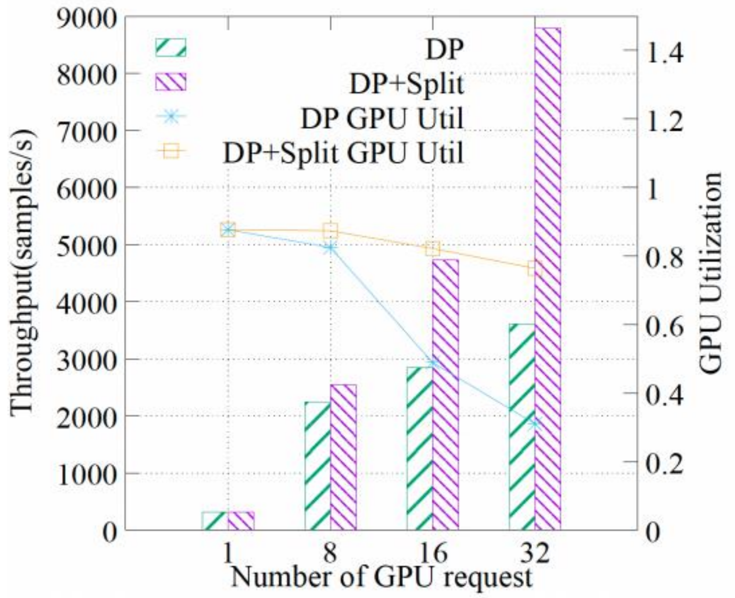
Whale: Efficient Giant Model Training over Heterogeneous GPUs

Challenge:

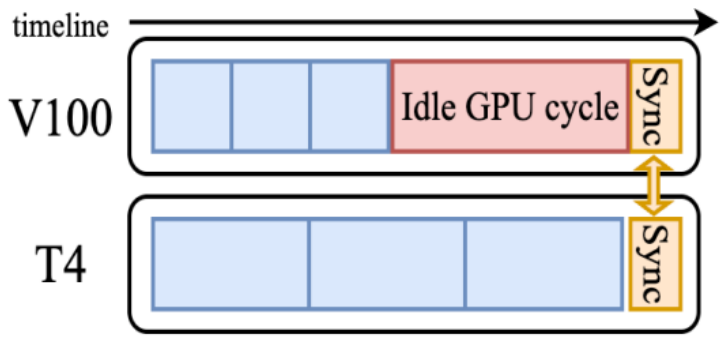
- Model sizes have grown so drastically that training systems have hit both (1) a bandwidth wall, due to high communication overhead for data parallel execution; as well as (2) a GPU memory limit.
- Pure pipeline parallelism does not scale well with more GPUs
- Nested data-parallelism combined with pipeline-parallelism yields better performance
- Applying different strategies to different model parts produces the best performance
- Heterogeneity in GPU Clusters pose additional efficiency challenges
    - Different GPU types: different computing/memory/network capacity
    - Imbalance in computing time → low utilization
    - Gap between model development and the hardware environment



(a) Local Model (b) Distributed Model



ResNet50 #class=100K
DP vs Hybrid



Whale: Efficient Giant Model Training over Heterogeneous GPUs

Gaps and Opportunities:

- Lack of unified abstraction to support all of the parallel strategies and the hybrids
    - Unified abstraction for strategy expression
- Fully automatic parallel strategy has high cost for giant models
    - Incorporate user hints
- Inefficiency in utilizing heterogeneous GPUs
    - Parallel strategies should be used adaptively and dynamically
- Require significant model code refactoring
    - Minimize code change, switch among strategies easily

System overview:

- Two new high-level primitives for unified expression
- Transforms distributed models efficiently and automatically
- Hardware-aware load balancing algorithm
- Train the largest multi-modality model M6 with ten trillion models with only 4-lines of code change

Whale: Efficient Giant Model Training over Heterogeneous GPUs

Parallel Primitives:

- Parallel primitive is a Python context manager
- Operations defined under form one TaskGraph (TG)

```
import whale as wh
wh.init(wh.Config({
  "num_micro_batch": 8}))
with wh.replicate(1):
  model_stage1()
with wh.replicate(1):
  model_stage2()
```

Pipeline with 2 TaskGraphs

```
import whale as wh
wh.init()
with wh.replicate(total_gpu):
  features = ResNet50(inputs)
with wh.split(total_gpu):
  logits = FC(features)
  predictions = Softmax(logits)
```

Hybrid of replicate and split

replicate(device_count) annotates a TG to be replicated

- device_count: #devices for TG replicas

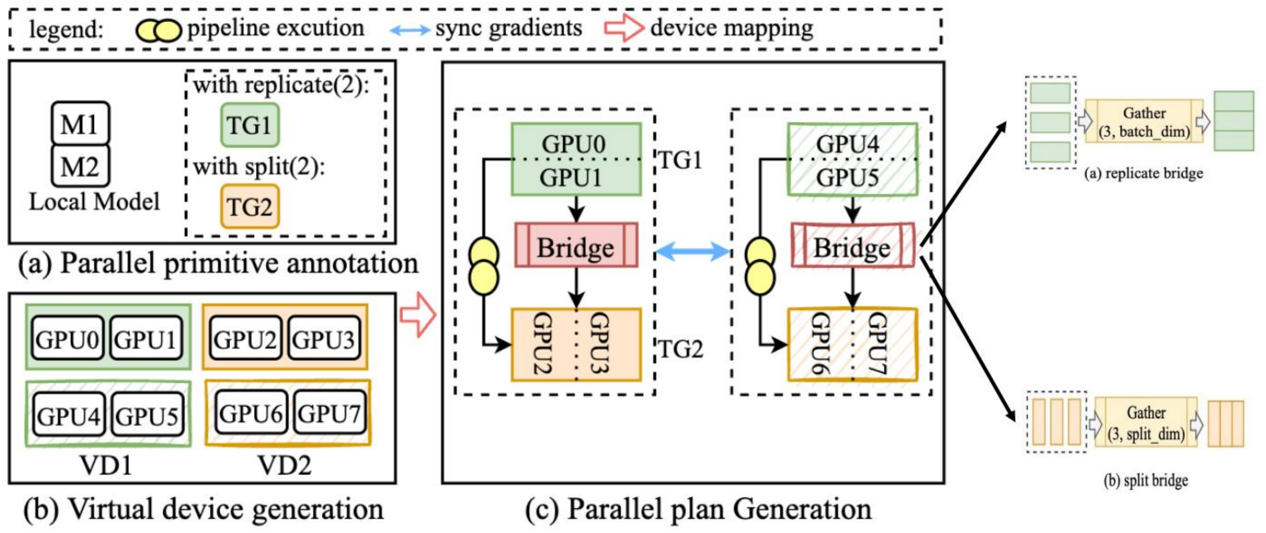
split(device_count) annotates a TG to apply intra-tensor sharding

- device_count: #devices for shared partitions

Hardware-aware Load Balancer

- Balance the computing load proportional to the device computing capacity, subject to memory constraints;
- Data parallelism: balance the FLOP by adjusting local batch while keeps the mini-batch unchanged;
- Tensor Model Parallelism: balance the FLOP of partitioned operations through uneven sharding;

Parallel Planner



Cachew: Machine Learning Input Data Processing as a Service

Problem:

- Data preprocessing and model training/tuning cannot be scaled independently → tough to fix bottlenecks (model training runs on fast GPUs, input pipeline runs on slow CPUs)

Input processing Challenges

A. Waiting for batches costs time and money (expensive GPUs are waiting for data)

B. Small set of pipelines account for most computation

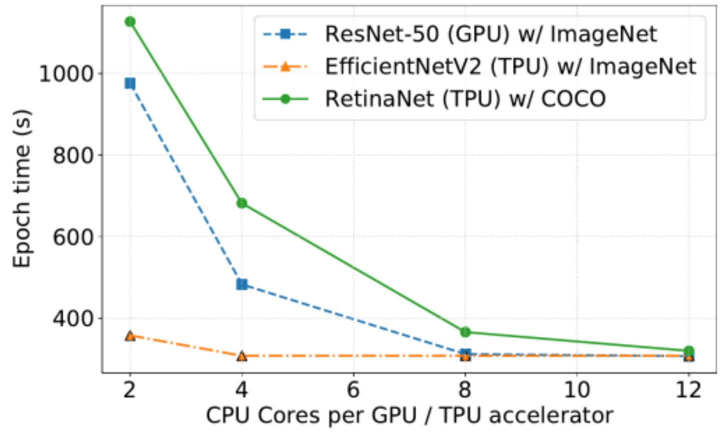
C. Preprocessing can consume more power than training

Opportunities

- A → **Scaling out**
- B / C → **Caching**

Current Landscape in ML Preprocessing

- Solutions for disaggregating input pipeline and model exist:
  - tf.data service from Google (open-sourced)
  - Data PreProcessing Service from Meta (closed-source)
- Fundamental mechanism is already there, **what is lacking is the functionality to perform these more automatically**



FUTUREWEI Technologies

Cachew: Machine Learning Input Data Processing as a Service

Caching:

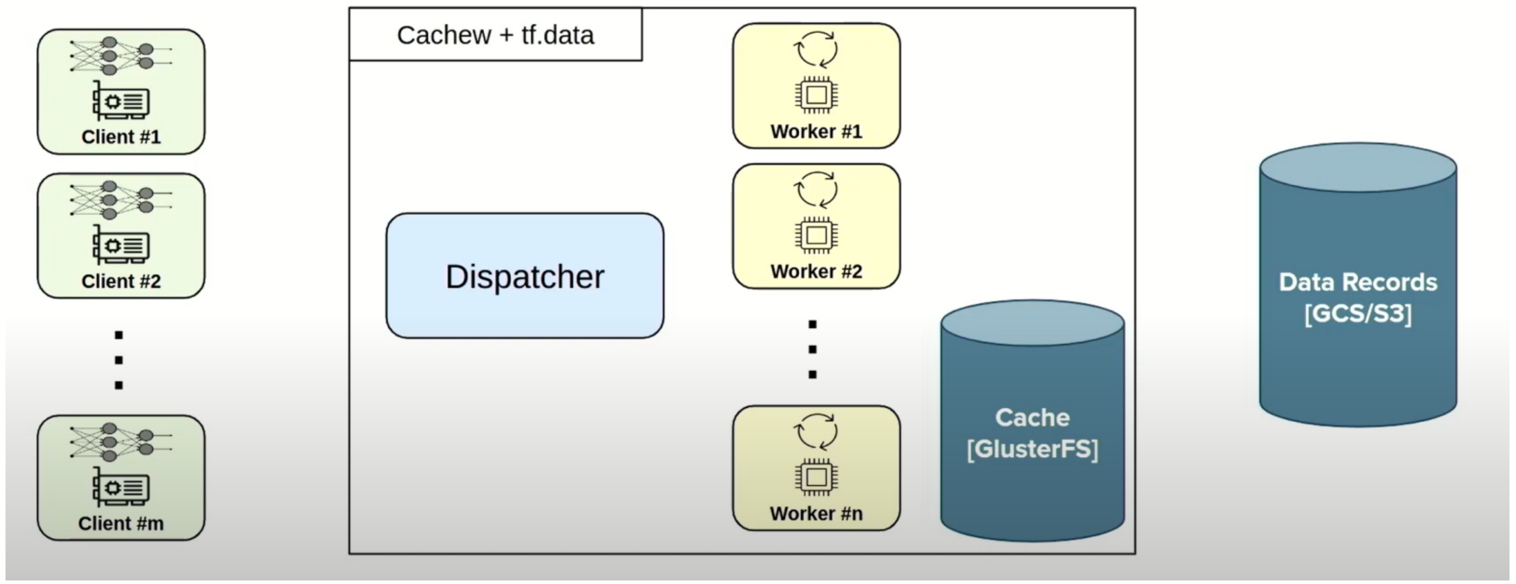
- Caching functionality already exist in many frameworks (it's not a research problem anymore)
- **Challenge:** when and where does caching work in the input pipeline; caching doesn't always make sense, i.e. reading from cache might be slower then recompute; even when it makes sense, it's not obvious where to utilize it in the input pipeline

As such, both scaling out and caching has to be done automatically → Contributions:

- **Autoscaling policy → How many resources** should be assigned to preprocessing?
- **Autocaching policy → When** and **Where** should data be cached?

System Architecture

- Builds on top of tf.data, because:
  - Disaggregation is already available
  - Open-source
  - Large-scale
  - Impactful

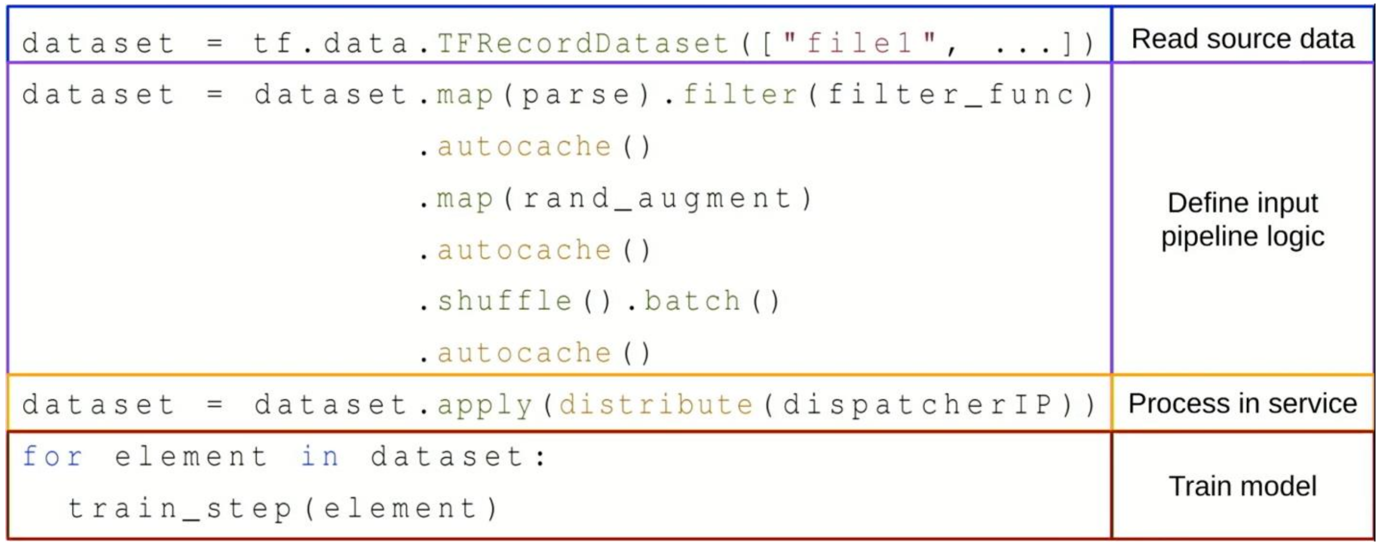


Cachew: Machine Learning Input Data Processing as a Service

Main features of Cachew:

- Multi-tenancy
- Disaggregation (inherited from tf.data)
- Autoscaling policy
- Autocaching policy: allows to decide if caching makes sense and if so where; also it can be used across jobs as well

Code Example

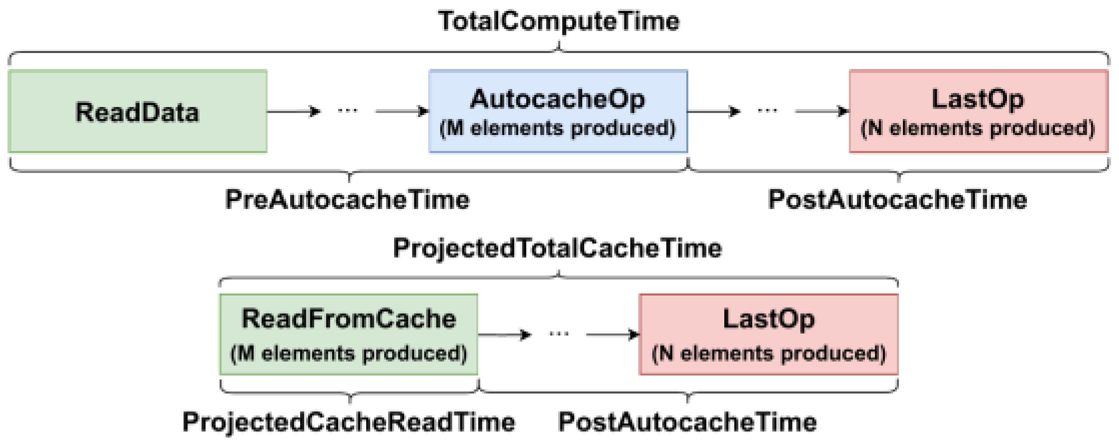


`dataset = tf.data.TFRecordDataset(["file1", ...])`	Read source data
`dataset = dataset.map(parse).filter(filter_func)`	Define input pipeline logic
`                  .autocache()`	
`                  .map(rand_augment)`	
`                  .autocache()`	
`                  .shuffle().batch()`	
`                  .autocache()`	
`dataset = dataset.apply(distribute(dispatcherIP))`	Process in service
`for element in dataset:`	Train model
`  train_step(element)`	

Cachew: Machine Learning Input Data Processing as a Service

Autocaching Policy

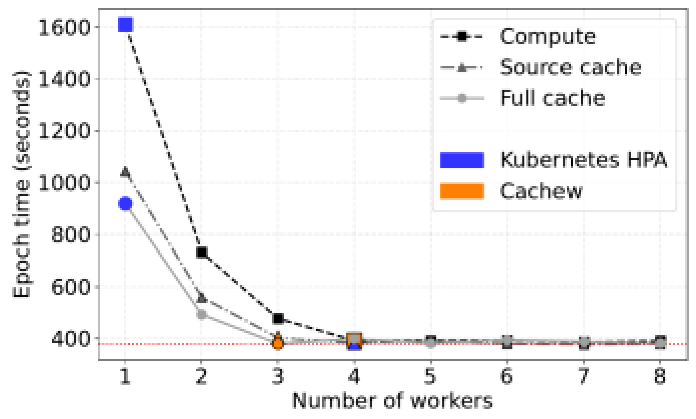
- Project throughput **for each autocache op** in input pipeline
- Chooses option with highest throughput



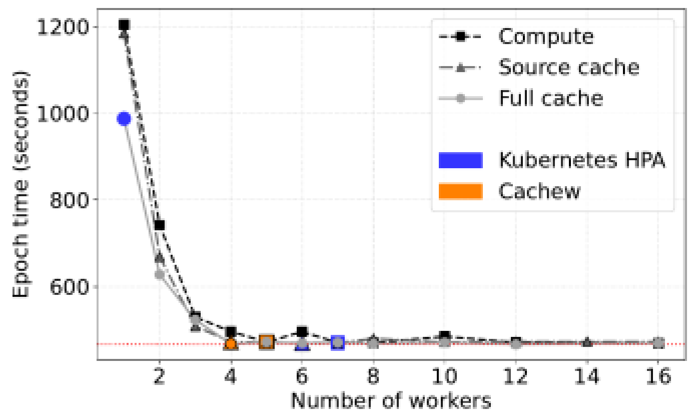
Autoscaling Policy

- Two steps are executed when processing a batch
- (1) Fetch batch from local buffer
- (2) Model training on batch

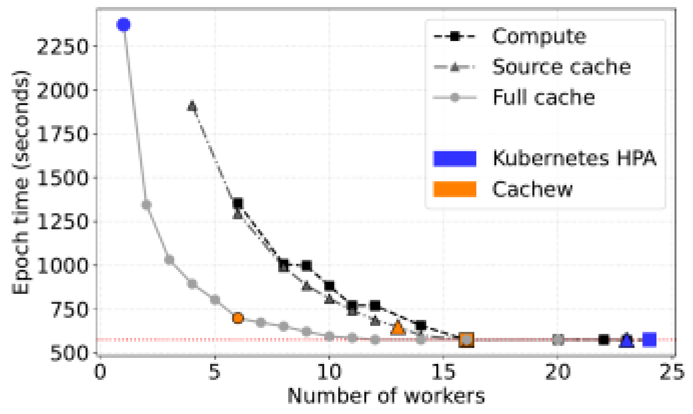
Autoscaling evaluation results



(a) ResNet50



(b) RetinaNet

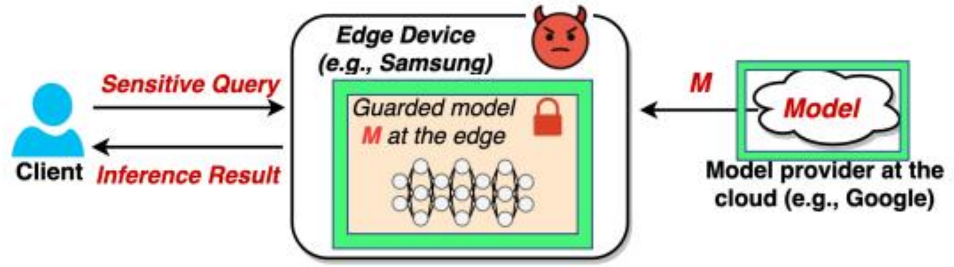


(c) SimCLR

SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

Background / Challenge:

- Edge-side inference requires low-latency, high-accuracy with confidentiality and integrity protection
- TEEs (Trusted Execution Environments) are promising to protect model confidentiality
- Edge-side TEEs are trusted, but edge-side GPUs are untrusted
    - CPU TEE does not support GPU, model providers cannot trust third-party GPUs
    - GPUs are essential however: Numerous edge devices have been integrated with GPU to accelerate edge intelligence



Requirements for secure Edge-side DNN Inference

- Ideal edge-side inference system should meet the following requirements
    - Performance
        - Low latency: utilize co-located GPU accelerator to speed up model inference
        - High accuracy: retain the same accuracy as the original model
    - Security
        - Model confidentiality: model parameters' plaintexts should be hidden
        - Inference integrity: any attacks (e.g. malicious modifications) on inference results should be detected

SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

Problem with prior work

- **TEE-shielding Approach:** Protects model confidentiality and inference integrity; Retains high-accuracy
  - **Limitations**: No GPU acceleration with extremely high inference latency (up to 36.1x) than insecure GPU inference
- **Partition-based Approach:** Low-latency with GPU acceleration
  - **Limitations**: Incur either confidentiality loss or accuracy loss; Integrity breaches on partitioned model

GOALS of SOTER

SOTER is a partition-based inference system that achieves all desired properties for edge-side DNN inference

- **Accelerate** heavy-weight computation with GPU and **retain high accuracy** as the original model
- **Protect model confidentiality** by hiding all parameters' plaintexts
- **Detect integrity breaches** (e.g., malicious modifications) on inference results

To achieve these goals, SOTER addresses two key questions:

(Q1) How can we utilize untrusted GPU for acceleration without sacrificing confidentiality or accuracy?

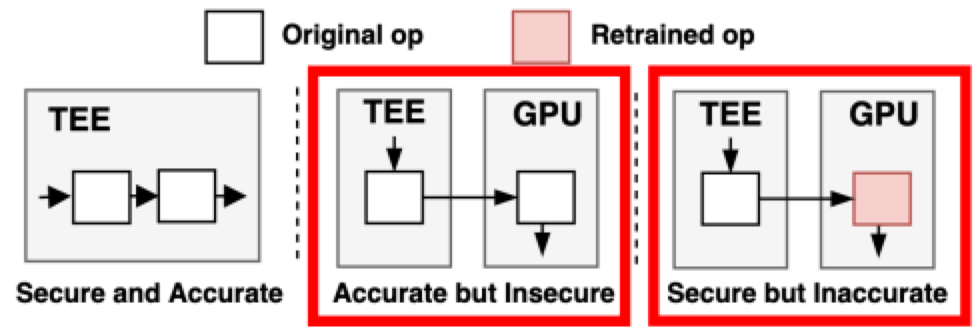
(Q2) How to efficiently detect integrity breaches outside the TEE?

FUTUREWEI Technologies

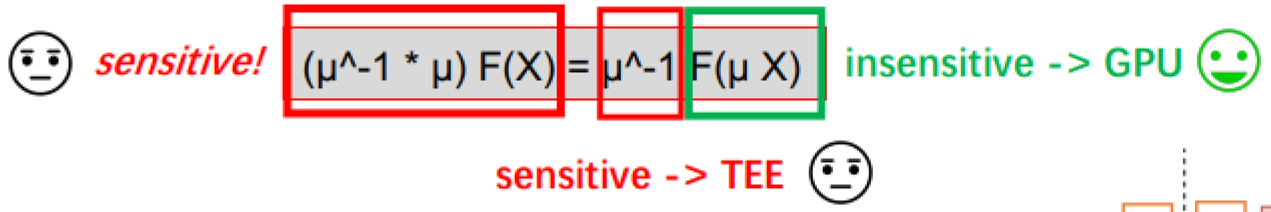
SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

(Q1) Bridging the Confidentiality-Accuracy GAP

➢ **Confidentiality-accuracy dilemma**

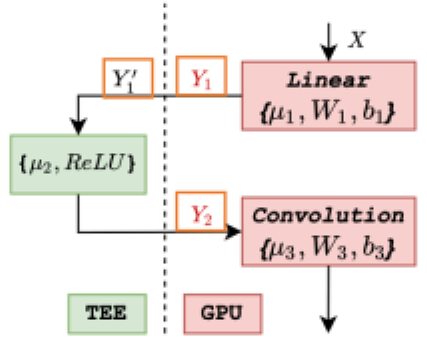


➢ **SOTER's key weapon: the general associativity property of common inference operators**



➢ **Major workflow**

- **Step 1:** Automatically profile an encrypted model in TEE
- **Step 2:** Morph a portion of associative operators' parameters with *hidden scalars*
- **Step 3:** Partition morphed operators to run on GPU
- **Step 4:** Execute operators in order, transmit IRs between kernels, restore execution results with hidden scalars in TEE



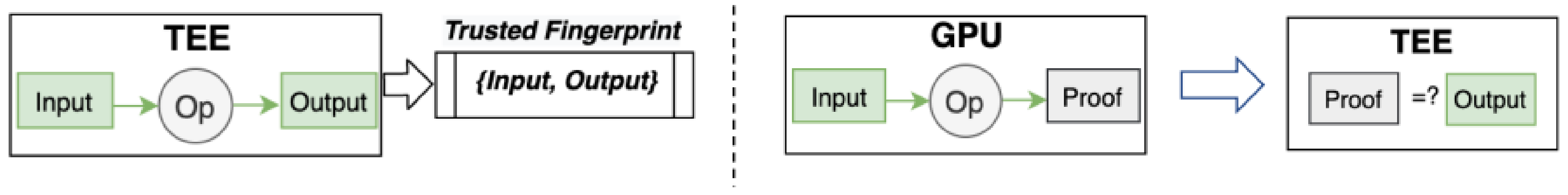
X
Y_1'
Y_1
Linear $\{\mu_1, W_1, b_1\}$
$\{\mu_2, ReLU\}$
Y_2
Convolution $\{\mu_3, W_3, b_3\}$
TEE
GPU

$$\boxed{Y_1} = \mu_1(W_1 * X + b_1)$$
$$\boxed{Y_1'} = Y_1 * \mu^{-1} = W_1 * X + b_1$$
$$\begin{aligned}\boxed{Y_2} &= \mu_2 * ReLU(Y_1') \\ &= 0 \Leftarrow if\ Y_1' \leq 0 \\ &= \mu_2 * (W_1 * X + b_1) \Leftarrow if\ Y_1' > 0\end{aligned}$$

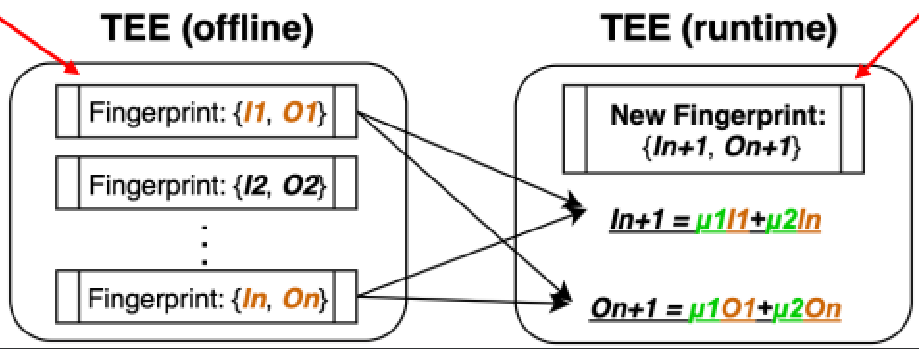
SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

(Q2) Detecting Integrity Breaches

- **Partition-based system *inevitably* open access to integrity breaches outside the TEE**
- **Detect integrity breaches: a straw man *Trusted Fingerprint* (TF) re-computing approach**



- **Key challenge: Obliviousness-timeliness dilemma**
  - If we use fixed TF, the adversary can easily observe and bypass the TF detection
  - If generate new TF as regular user input in CPU TEE, TFs become oblivious to observe, but TF generation (in CPU TEE) becomes the performance bottleneck, leading to slow detection
- **SOTER solves the challenge using the same associativity observation from confidentiality protection**
  - Associativity variant: If F(X1) = Y1; F(X2)=Y2; …; F(Xn)=Yn, then F(μ1X1+ μ2X2+…+ μnXn)= μ1Y1+ μ2Y2+…+ μnYn

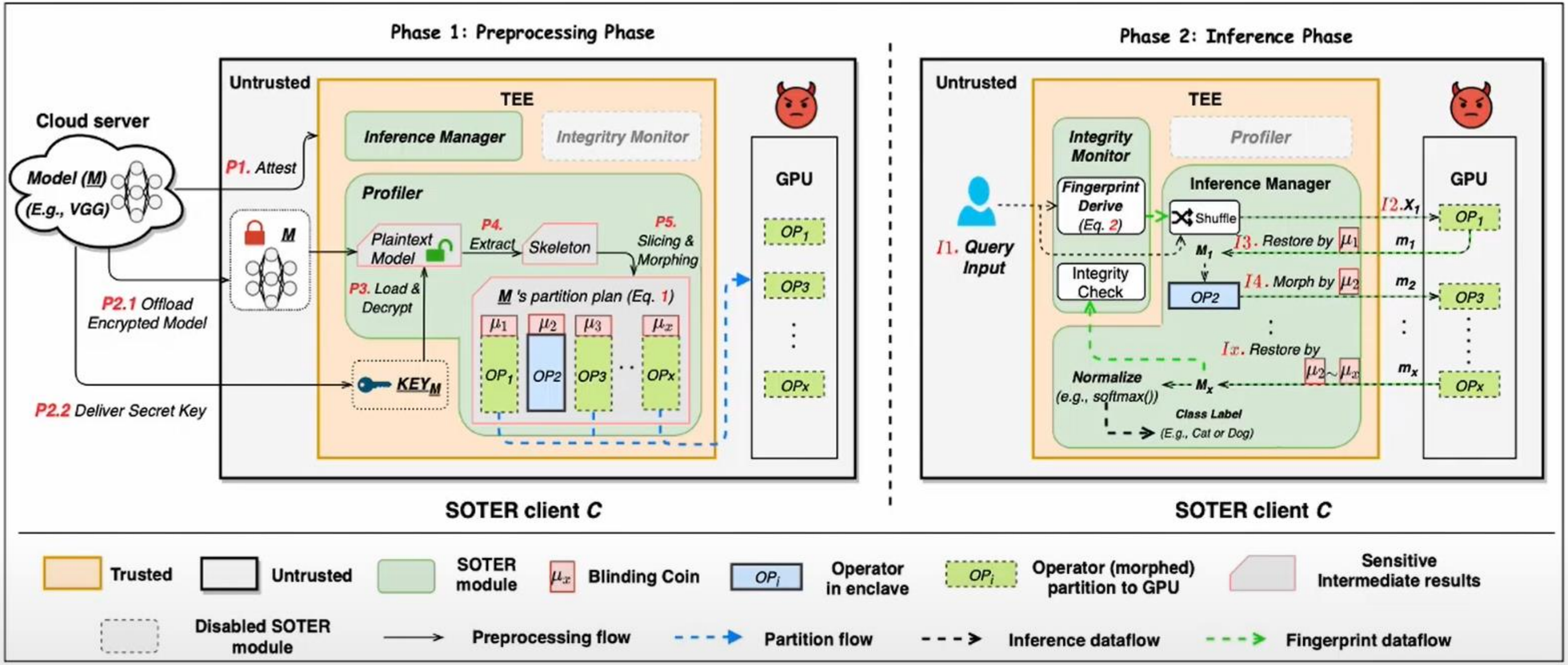


Step 1:
Prepare ***cornerstone TFs*** in the preprocessing phase

Step 2:
Generate new ***scalars*** and use the ***associativity variant*** to efficiently produce new TFs

SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

SOTER ARCHITECTURE



THANK YOU!!!

FUTUREWEI Technologies