AntMan: Dynamic Scaling on GPU Clusters for Deep Learning

Authors: Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou,
Zhi Li, Yihui Feng, Wei Lin, Yangqing Jia
Alibaba Group

Presenters: Shi Pu, Jianbin Zhang

Deep Learning Training

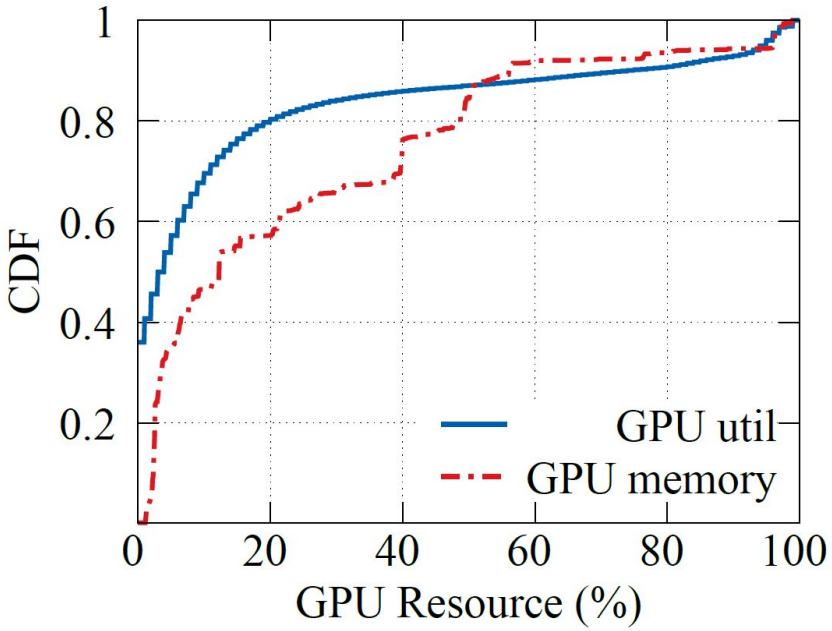
Deep learning

- Spans across multiple fields
  - Computer vision
  - Language understanding
  - Speech recognition
  - Advertisement
  - …
- Adopts data parallelism in multiple GPUs to speed up training
- Large companies use multi-tenant clusters to improve hardware

Characteristics of Production DL Clusters

Low utilization of in-use GPUs

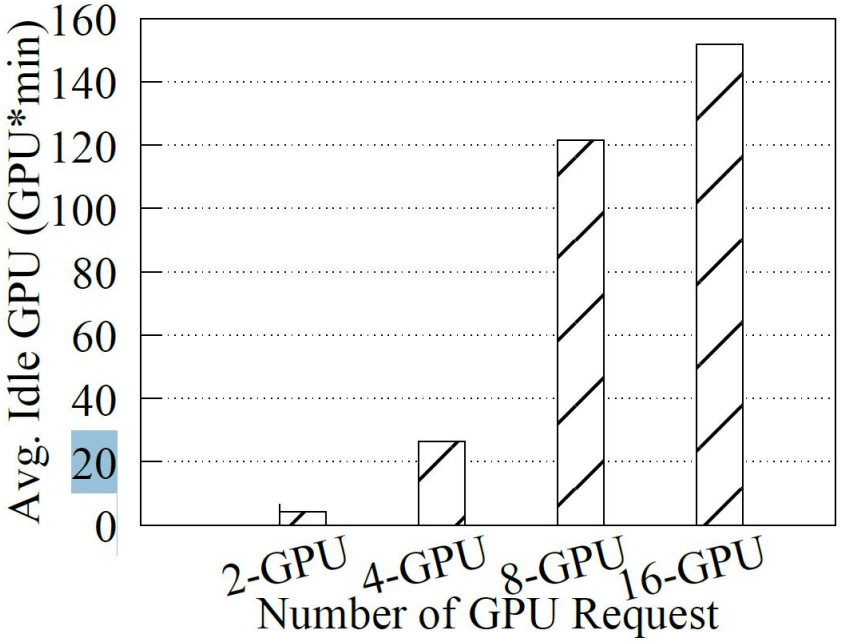
- 20% of the GPUs are consuming more than half of the GPU memory
- 10% of the GPUs achieve higher than 80% utilization



Characteristics of Production DL Clusters (cont.)

Idle waiting for gang-schedule

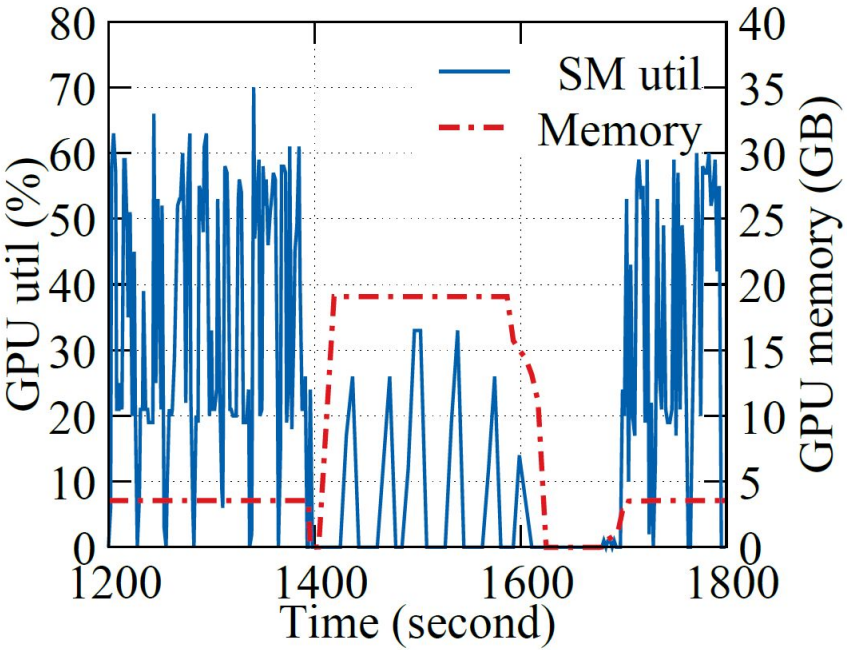
- A job will not start until all required GPUs are simultaneously available
- Wasted GPU cycles



Characteristics of Production DL Clusters (cont.)

Dynamic resource demand

- Training pipeline contains several different phases
- Each phase has different resource requirements
- Hardware is underutilized because jobs request resources according to their peak usage

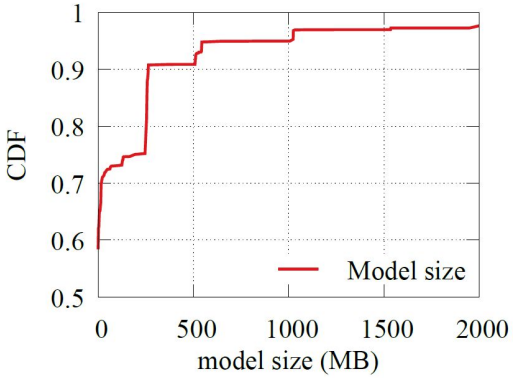


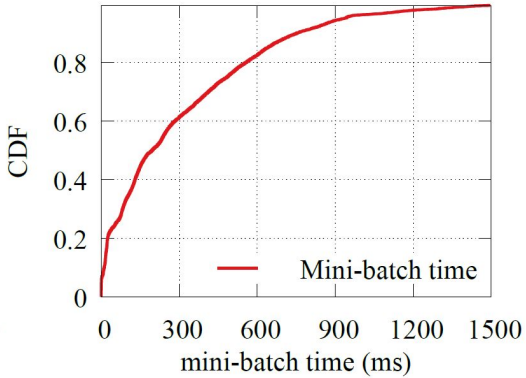
Solution: Resource Sharing!

Challenges of GPU resource sharing:

- Achieve performance isolation for important jobs (resource-guarantee jobs)
  - Ensure Job performance as same as dedicated GPU execution
- Prevent potential failure from GPU memory contention
  - Dynamic resource demand happens

Opportunities in DL Uniqueness





Small models

- 90% of DL models occupy only 500 MB

Short mini-batch

- 80% of tasks consume a minibatch with 600 ms

Similar mini-batch

- Allows performance profiling

AntMan: Dynamic Scaling on GPU Clusters

- Improves GPU utilization
  - While providing performance guarantee on resource-guarantee jobs
- Utilizes spare GPU resources to co-execute multiple jobs
  - Dispatches opportunistic jobs to best-effort utilize GPU resources

Design

AntMan co-designs:

- DL framework
  - Dynamic memory management
  - Dynamic computation management
- Cluster scheduler
  - Global scheduler
  - Local coordinator

Dynamic Scaling Mechanism

Challenge: Executing jobs

- Satisfying minimal requirements
- Preventing GPU memory usage outbreak failures
- Adapting to the fluctuation in computation unit usage

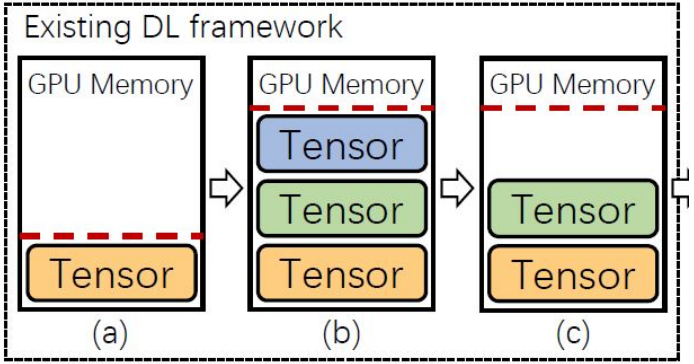
Dynamic scaling mechanism:

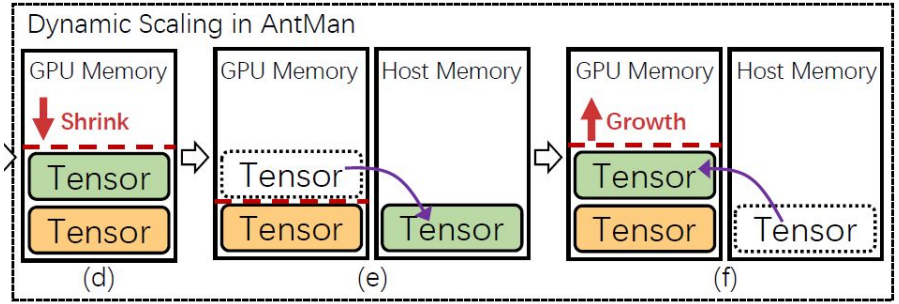
- Fine-grained dynamic control
- GPU memory
- Computation unit

Memory Management

Universal Memory

- Use host memory to prevent failure
- Dynamically adjust cached memory
- Thanks to unique pattern of DL
  - Avoids tensor copy
  - Negligible performance overhead

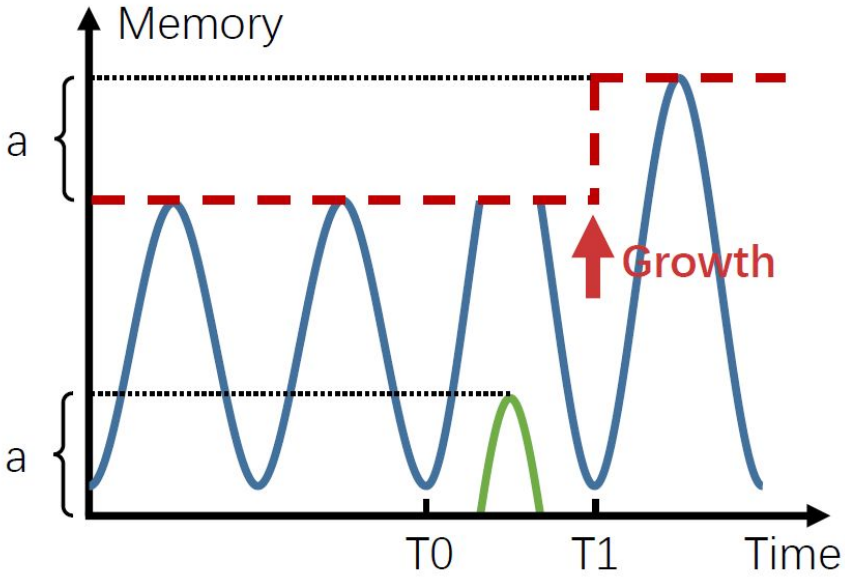




Memory Management Example

To handle burst demand:

- T0: Memory requirement increases
- T1: AntMan raises GPU memory's upper limit according to the usage of the host memory

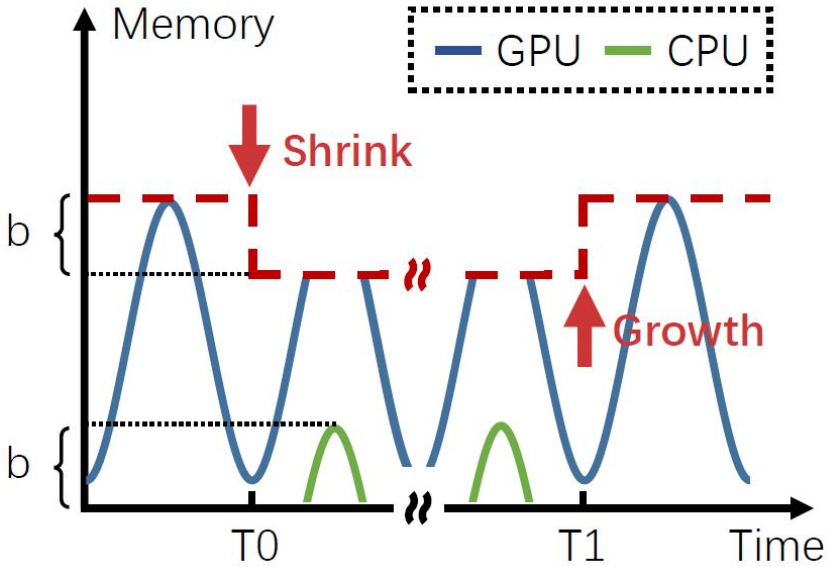


(a) Scaling for memory burst up.

Memory Management Example (cont.)

To secure memory for other jobs:

- T0: Memory requirement decreases
- T1: Memory requirement grows back after other jobs are finished

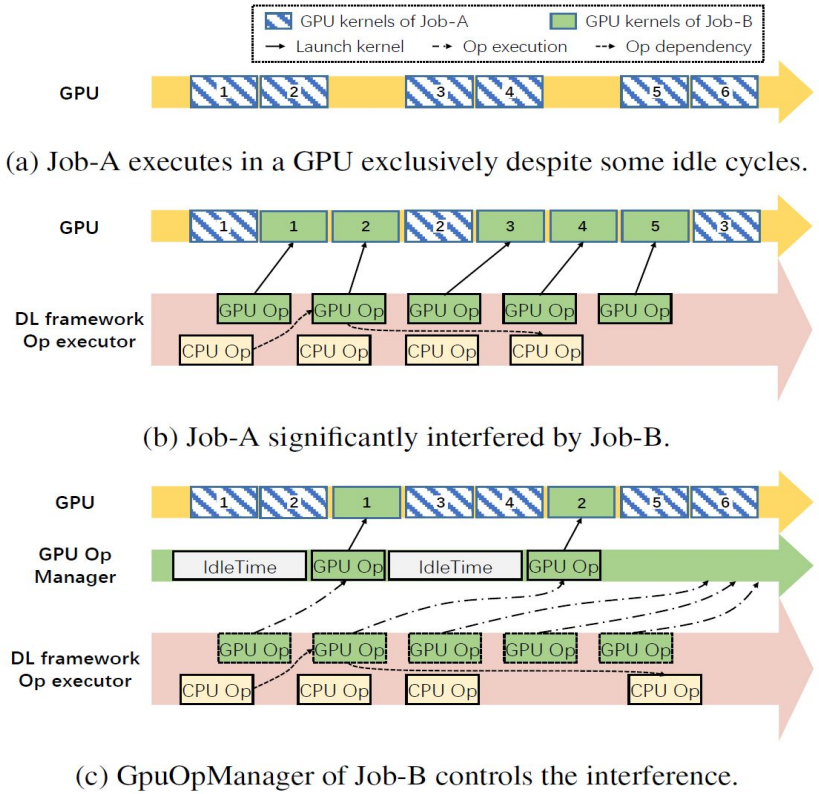


(b) Scaling to secure memory.

Computation Management

GpuOpManager

- Controls launching frequency
- Profiles GPU execution time
- Distributes idle time slots

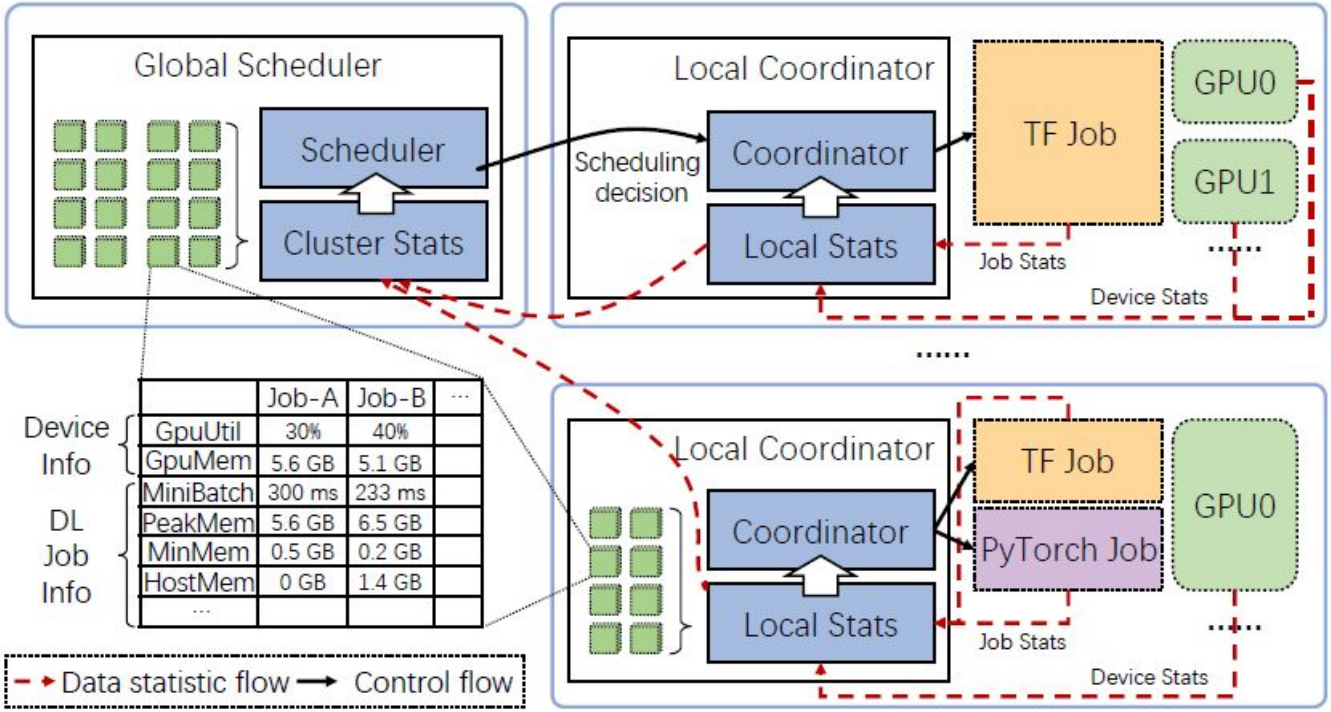


(a) Job-A executes in a GPU exclusively despite some idle cycles.

(b) Job-A significantly interfered by Job-B.

(c) GpuOpManager of Job-B controls the interference.

Collaborative Scheduler



Scheduling Policy

Goals:

- Primary: Multi-tenant fairness
- Secondary: Improve cluster efficiency
  - Opportunities jobs to steal idle cycles in GPUs

Scheduling Policy (cont.)

Global scheduler

- Resource-guaranteed jobs are given sufficient GPU resources
- Execute long queueing delay jobs as opportunistic jobs
- Allocate opportunistic jobs on the freest candidates

Algorithm 1 scheduleJob(in job, out nodes)

```
nodes0 ← findNodes(job.gpu, constraints ← job.topo)
nodes1 ← findNodes(job.gpu, constraints ← M)
nodes2 ← minLoadNodes(nodes1, job.gpu)
if job.isResourceGuarantee:
    if numGPUs(nodes0) >= job.gpu:
        return nodes0
    else:
        reserve(nodes0)
else:
    return nodes2
```

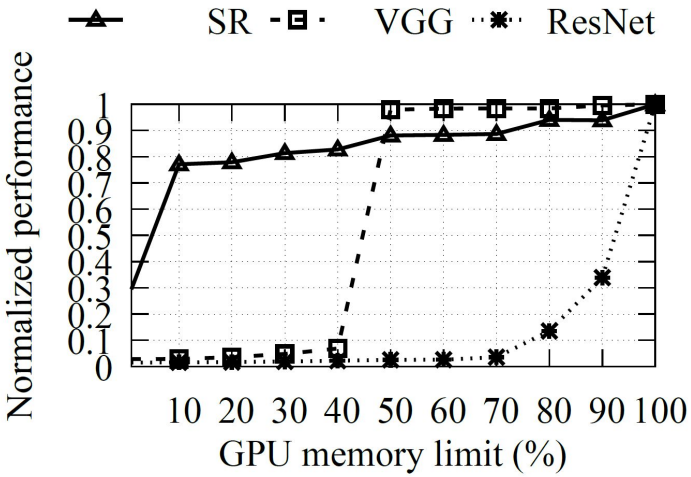
Scheduling Policy (cont.)

Local Coordinator

- Limit the usage of GPU resources of opportunistic jobs
- Optimizes aggregated job performance when there exists multiple opportunistic jobs

Job Upgrade

- Upgrade opportunistic jobs when resources permit



Implementation

- TensorFlow and PyTorch frameworks
  - Modification mainly in memory allocator, GPU executor and interfaces
- Prototype in Kubernetes
  - 2000 LOC in Python
- Implementation in Fuxi (Alibaba's internal cluster scheduler)
  - 10,000 LOC in C++
  - Including failover support and testing

Evaluation

Dynamic GPU memory scaling

- Preempt killed Job-A.
- FIFO introduces 17.1 minutes queuing delay.
- Pack failed Job-B due to insufficient memory.
- UMem used host memory when needed.
- AntMan showed great results.

	Model	Arrival	GpuMem	BS	Quota
Job-A	GCN	0 min	3.5 GB	1400	No
Job-B	ResNet	26 min	30.0 GB	360	Yes

Table 1: Setup and information of two jobs.

	Preempt	FIFO	Pack	UMem	AntMan
Job-A	Failed	43.0	43.1	43.4	43.9
Job-B	91.1	108.2	Failed	541.6	91.8

Table 2: Job status and JCT (min) of two jobs executing in different configurations.

Cluster Experiment

- Job queueing delay reduces by 2x on average
- Almost 99% jobs suffer zero job interference

	Avg.	90% tile	95% tile
Dec. 2019	1132	1978	5960
Apr. 2020	550	124	489

Table 4: One-week queuing delay statistic in seconds.

Interference	0%	0~1%	1~2%	2~3%	3~4%
# of jobs	9895	26	30	20	29

Table 5: Interference analysis on mini-batch time for 10K production jobs

Other Benchmarks

- Efficient memory shrinkage and growth
- Dynamic GPU computation unit scaling
- Prototype trace experiment
- Omitted for brevity

Conclusion

- Implementation deployed on DL infrastructure at Alibaba
- Provides dynamic scaling primitives for DL jobs
- GPU utilization maximized for opportunistic jobs while avoiding job interference
- 42% improvement in GPU memory utilization
- 34% improvement in GPU computation resources utilization

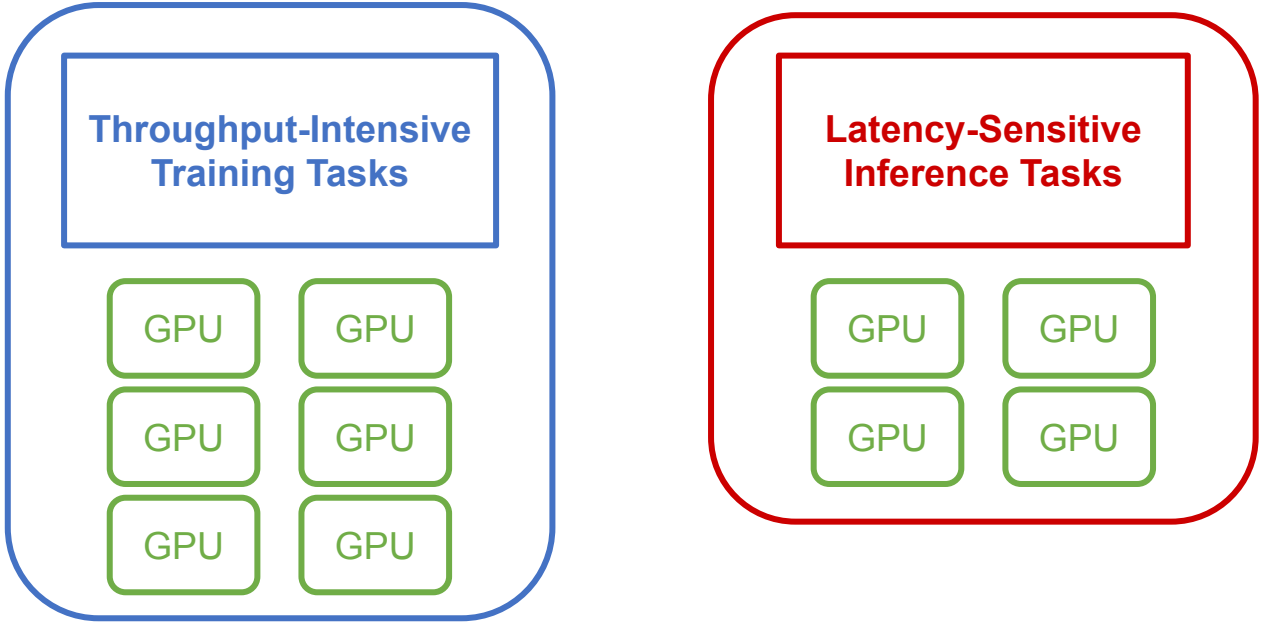
PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications

Authors: Zhihao Bai, Zhen Zhang, Yibo Zhu, Xin Jin

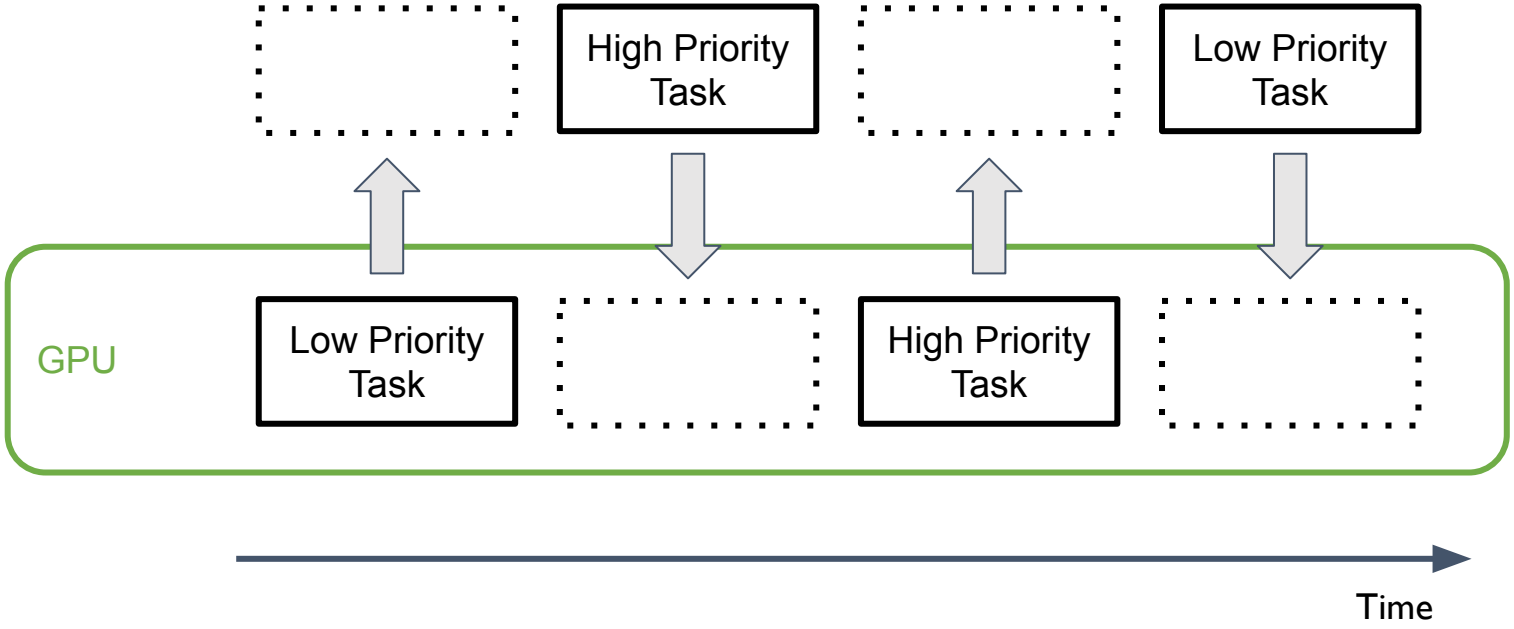
John Hopkins University ByteDance Inc.

Presenters: Wenqi Zhu, Jianbin Zhang

Training and Inference Tasks



What if we can time-share GPU tasks?



GPU Task Switching Overhead

- Four components
  - Task cleaning
  - Task initialization
  - Memory allocation
  - Model transmission
- SLA violated

Instance Type	g4dn.2xlarge	p3.2xlarge
GPU Type	NVIDIA T4	NVIDIA V100
Task Cleaning	155 ms	165 ms
Task Initialization	5530 ms	7290 ms
Memory Allocation	10 ms	13 ms
Model Transmission	91 ms	81 ms
Total Overhead	5787 ms	7551 ms
Inference Time	105 ms	32 ms

GPU Task Switching Problems

- Training BERT to Inference ResNet on Nvidia T4: 6 seconds
- GPU memory is limited
- DL models are large and getting larger
- GPU memory isolation

Solutions

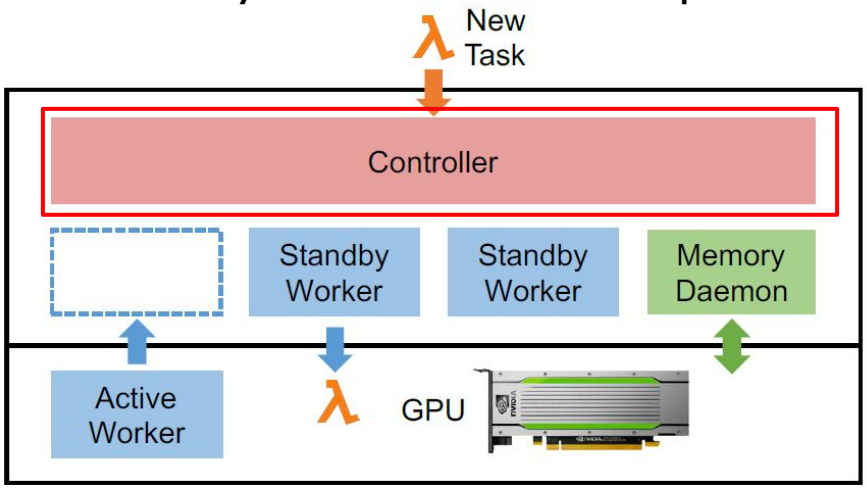
- Current:
  - Nvidia MPS: high overhead due to contention between tasks
  - Salus: allows fast job switching only if all models are preloaded in GPU memory
- Proposed:
  - PipeSwitch!

PipeSwitch: Fast Pipelined Context Switch

- Enable GPU-efficient multiplexing of multiple DL applications with fine-grained time-sharing.
- Achieve millisecond-scale task switching overhead to satisfy SLOs requirements.

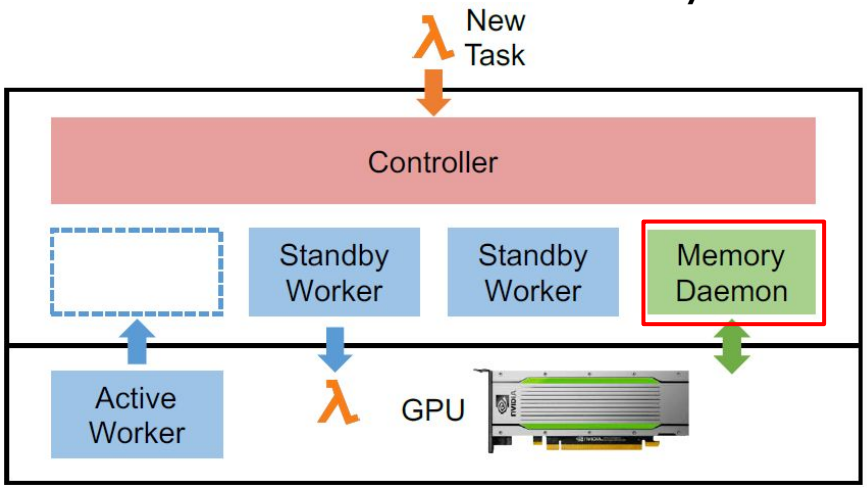
PipeSwitch: Overall Architecture

- Controller
  - Central Component.
  - Receives task from clients.
  - Controls the memory daemon and worker processes.



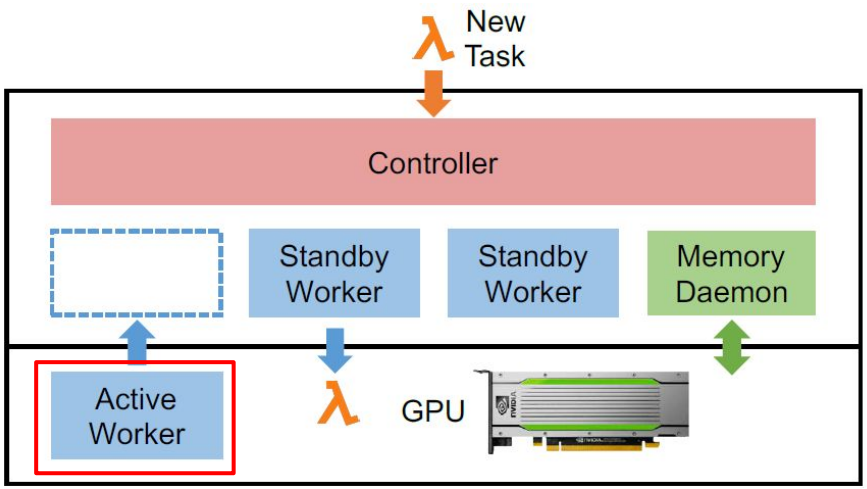
PipeSwitch: Overall Architecture

- Memory Daemon
  - A customized memory manager between GPU driver and worker process.
  - Allocates the GPU memory for worker.
  - Transfers the model from CPU host memory to GPU memory.



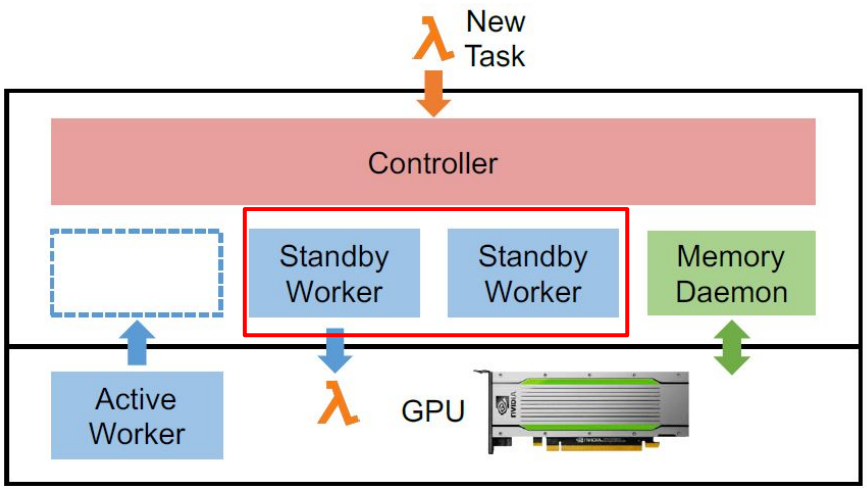
PipeSwitch: Overall Architecture

- Active Worker
  - The worker process that is currently executing the task.



PipeSwitch: Overall Architecture

- Standby Worker
  - The worker process that is idle/task initializing/task cleaning.
  - A server has one or more standby workers.



How to reduce context switching overhead?

Model Transmission → Pipelined Model Transmission

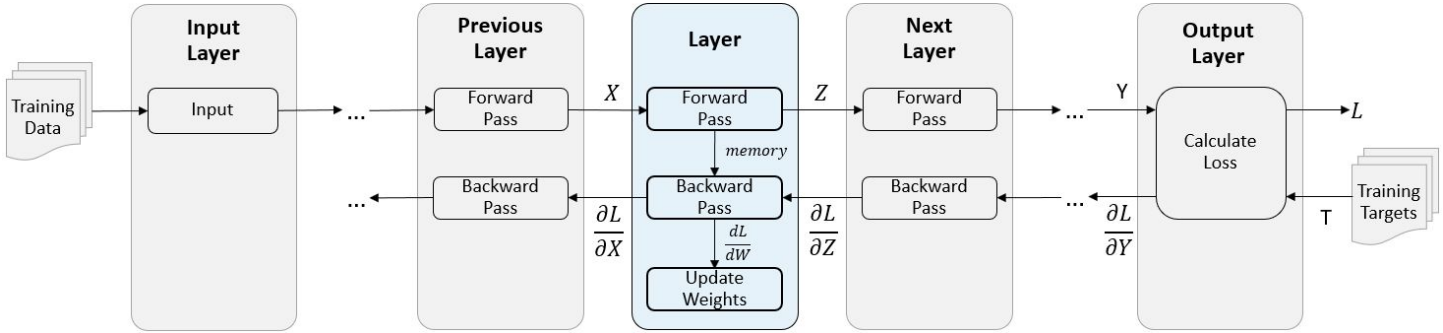
Memory Allocation

Task Initialization

Task Cleaning

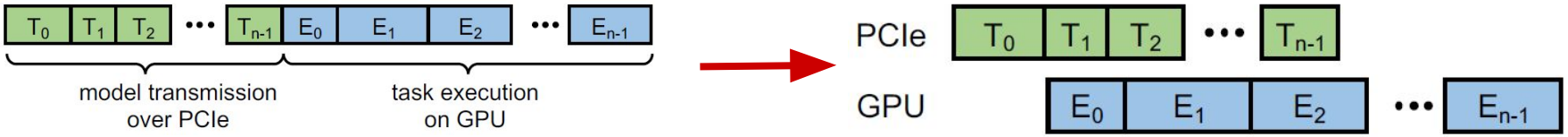
Pipelined Model Transmission: Basic Idea

- Insight: DL workloads have well-defined layered structures.
  - Computation takes place layer by layer.
  - To compute current layer, we only need
    - Output from earlier layers
    - Parameters of the current layer



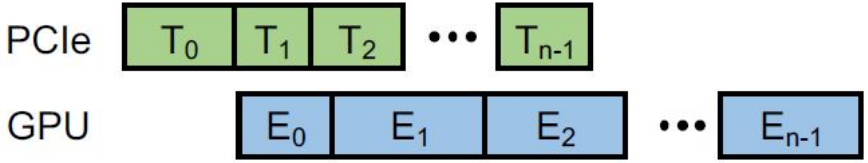
Pipelined Model Transmission: Basic Idea

- Model is transmitted via PCIe from CPU to GPU.
  - Overhead is bounded by PCIe bandwidth!
- Leverage layered structure of DL model,
  - We can pipeline model transmission and task execution!
  - Model transmission overhead can be hidden.



Per-Layer vs Per-Group Granularity

- Basic way for pipelining: on *per-layer* granularity
- Introduce large pipelining overhead!
  - A PCIe call is invoked per layer.
  - Synchronization overhead between transmission and computation.
- Group multiple layers, perform pipelining on *per-group* granularity.
  - Still need to pay pipelining overhead, but once a group!



Per-Group Granularity

- Trade-off between pipelining efficiency and pipelining overhead.
- Small groups
  - Per-layer in the extreme case
  - Huge pipelining overhead
- Big groups
  - Entire model in one group in the extreme case
  - Low pipelining efficiency
- PipeSwitch's solution to find **optimal** grouping strategy **efficiently**
  - **Optimal Model-Aware Grouping** algorithm

Formulate the optimal grouping question

n = number of total layers in a model

F(B, i):

B = already formed groups, from layer 0 to i-1

i = the first layer that have not formed a group

return total time of the optimal grouping strategy from layer i to n-1, given B.

We have the following recursive formula:

$$F(\{\},0) = \min_i \boxed{F(\{group(0,i)\},i+1)}$$

F({group(0,i)}, i+1) can be recursively computed using this formula.

To find the optimal grouping strategy for the entire model,

we divide all possible combinations into n cases based on how the first group is formed.

Pruning opportunity 1

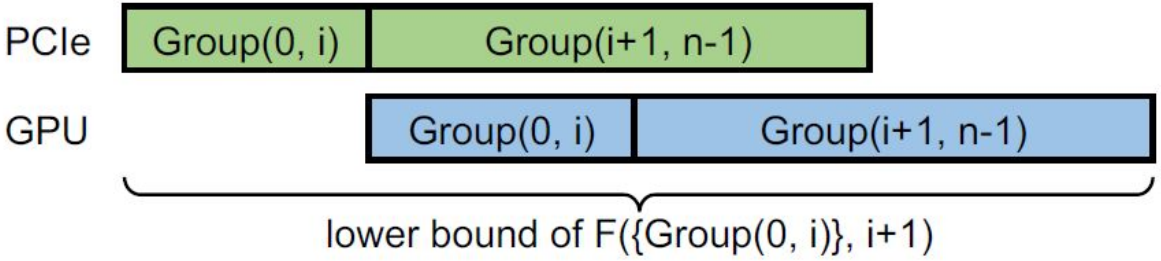
$$F(\{\},0) = \min_i F(\{group(0,i)\}, i+1)$$

Considering a best scenario:

- Given formed Group(0,i), all the remaining layers are combined in one group.
- Computation of Group(i+1, n-1) can happen right after the computation of Group(0,i) finishes.

The total time of case i in this scenario is considered as lower bound.

If the lower bound of case i is already larger than current optimal time, we can prune this case i.



Pruning opportunity 1: Compute Lower-bound

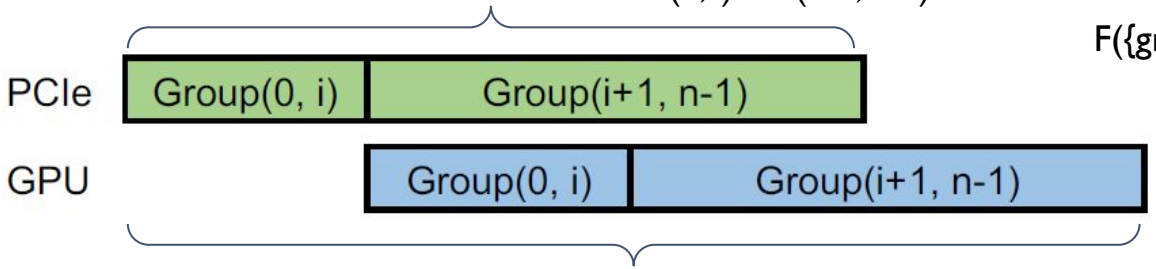
T(i,j) = Transmission times for a group from layer i to j, including overhead of invoking multiple PCIes.

Calculated based on the size of layer i to j and PCIe bandwidth.

E(i,j) = Execution times for a group from layer i to j.

Profiled on the GPU.

Lower bound of transmission time of case i: T(0, i) + T(i+1, n-1)



Lower bound of execution time of case i: T(0, i) + E(0, i) + E(i+1, n-1)

F({group(0,i)}, i+1) ≥

min(Transmission time *lower bound*,

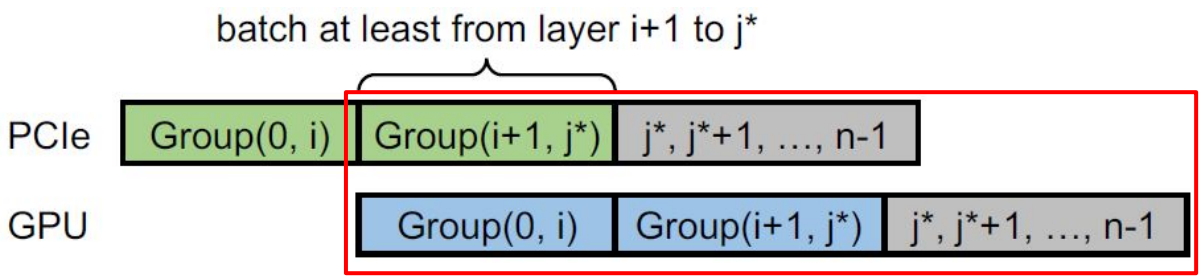
Execution time *lower bound*)

Pruning opportunity 2

$$F(\{\},0) = \min_i F(\{group(0,i)\}, i+1)$$

We can safely pack multiple layers in a group without affecting pipeline efficiency.

But which layers?



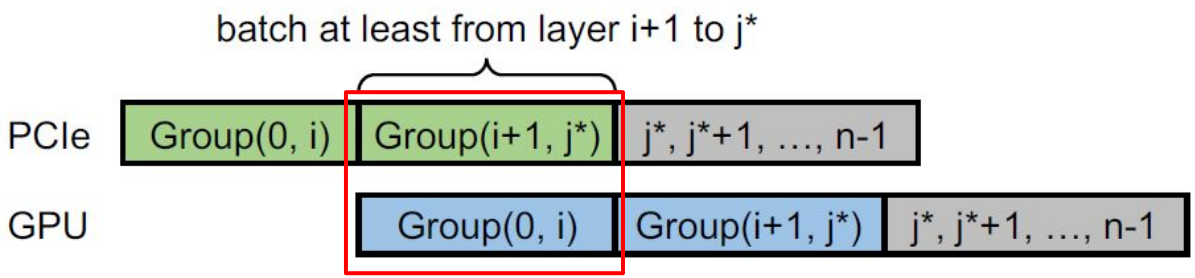
Group(0,i) already fixed.
We are applying the recursion equation to find the second group [i+1, j].
We want to pick a best j in range (i+1, n-1)

Pruning opportunity 2

$$F(\{\},0) = \min_i F(\{group(0,i)\}, i+1)$$

We can safely pack multiple layers in a group without affecting pipeline efficiency.

But which layers?



Find a max j* in [i+1, n-1].

s.t. Transmission overhead of Group(i+1, j*) is hidden behind computation of Group(0,i)

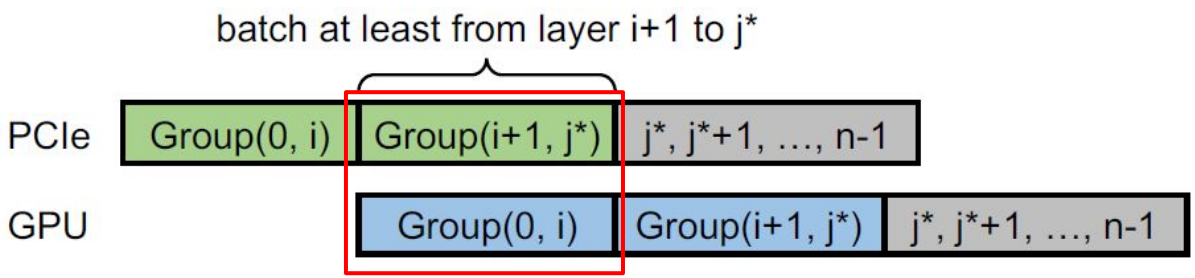
Which means T(i+1, j*) ≤ E(0,i)

Pruning opportunity 2

$$F(\{\},0) = \min_i F(\{group(0,i)\}, i+1)$$

We can safely pack multiple layers in a group without affecting pipeline efficiency.

Layers [i+1, j*] can be packed in a group directly. We only need to search for j ≥ j*.



Find a max j* in [i+1, n-1].

s.t. Transmission overhead of Group(i+1, j*) is hidden behind computation of Group(0,i)

Which means T(i+1, j*) ≤ E(0,i)

Optimal Model-Aware Grouping

Algorithm 1 Optimal Model-Aware Grouping

```
function FINDOPTGROUPING(B, x)
    opt_groups ← ∅, opt_groups.time ← ∞
    // find first group from layer i to j*
    j* ← x
    for layer i from x to n − 1 do
        if T(x, i) ≤ B.delay then
            j* ← i
        else
            break
    // recursively find the optimal grouping
    for layer i from j* to n − 1 do
        if opt_groups ≠ ∅ then
            // compute lower bound
            trans_time ← T(x, i) + T(i + 1, n − 1)
            exec_time ← max(T(x, i), B.delay)
                        + E(x, i) + E(i + 1, n − 1)
            lower_bound ← min(trans_time, exec_time)
            if lower_bound > opt_groups.time then
                continue
        // recursively find rest groups
        first_group ← Group(x, i)
        rest_groups ← FindOptGrouping(
                        B + first_group, i + 1)
        cur_groups ← first_group + rest_groups
        if cur_groups.time < opt_groups.time then
            opt_groups ← cur_groups
    return opt_groups
```

- Given n layers
  - Worst case time complexity: O(2^n)
  - But pruning techniques work really well in practice
- The algorithm finds the optimal grouping strategy that minimizes the total time for the pipeline.
  - Proved with induction.
- The algorithm is applicable to the general case.
  - Work for DAGs model like ResNet

How to reduce context switching overhead?

Model Transmission → Pipelined Model Transmission

Memory Allocation

Task Initialization

Task Cleaning

How to reduce context switching overhead?

Model Transmission → Pipelined Model Transmission

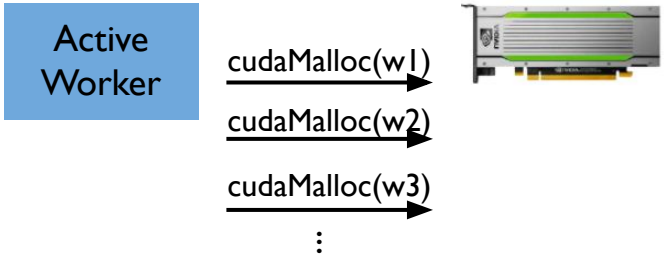
Memory Allocation → Unified Memory Management

Task Initialization

Task Cleaning

Unified Memory Management: Basic Idea

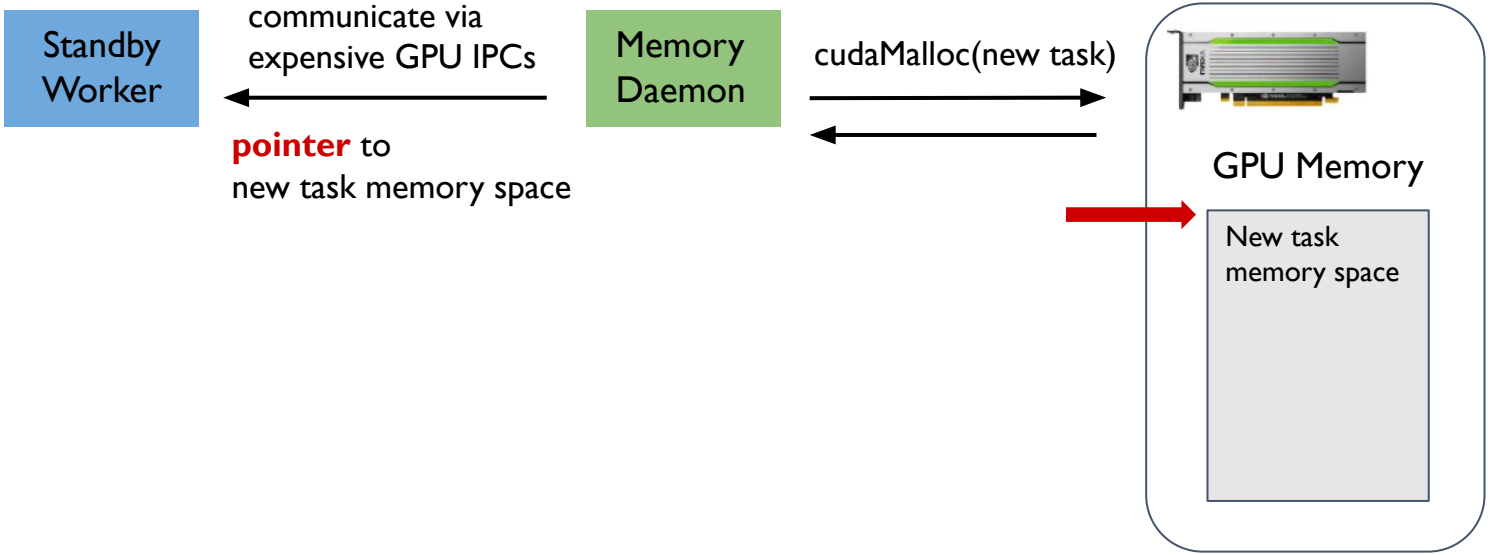
- Worker calls *malloc* function (e.g., *cudaMalloc* for NVIDIA GPUs) to request GPU memory
  - Many function calls introduce large overhead!
- Native GPU memory management is designed for general-purpose apps.
  - Very heavy-weight!



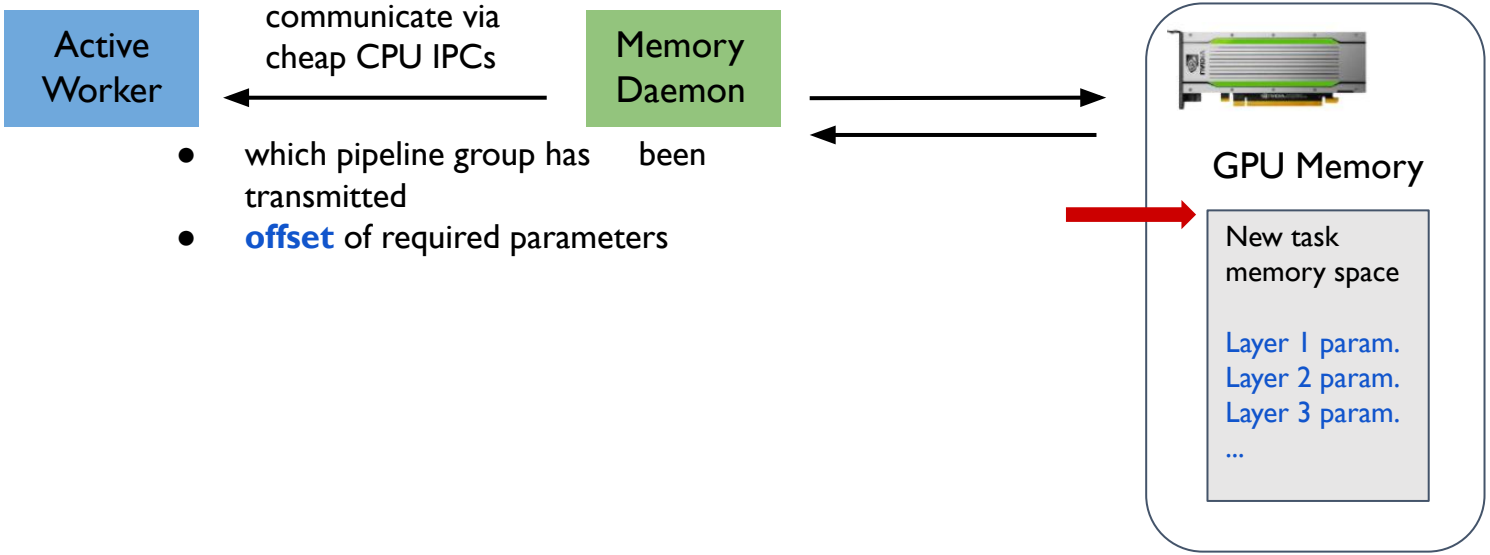
Unified Memory Management: Basic Idea

- Insight 1: For DNN models in GPU memory
  - the amount of memory it needs is fixed.
- Insight 2: For intermediate results in GPU memory
  - they change in a simple and deterministic pattern.
- Leverage DL characteristics, PipeSwitch customizes a **memory daemon**
  - to request a large amount of GPU memory at the beginning.
  - to handle memory allocation/release using light-weight stack-like mechanism.

At initialization stage



At execution stage



- which pipeline group has been transmitted
- **offset** of required parameters

How to reduce context switching overhead?

Model Transmission → Pipelined Model Transmission

Memory Allocation → Unified Memory Management

Task Initialization

Task Cleaning

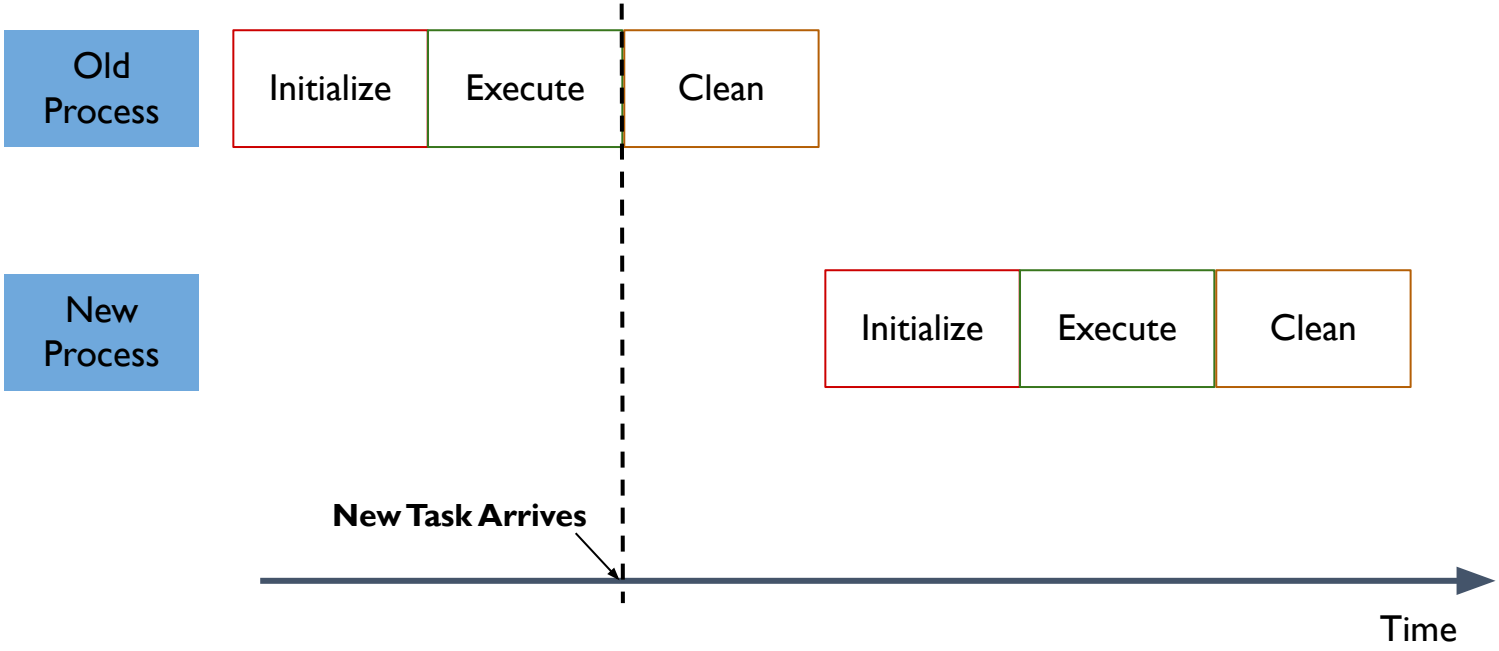
How to reduce context switching overhead?

Model Transmission → Pipelined Model Transmission

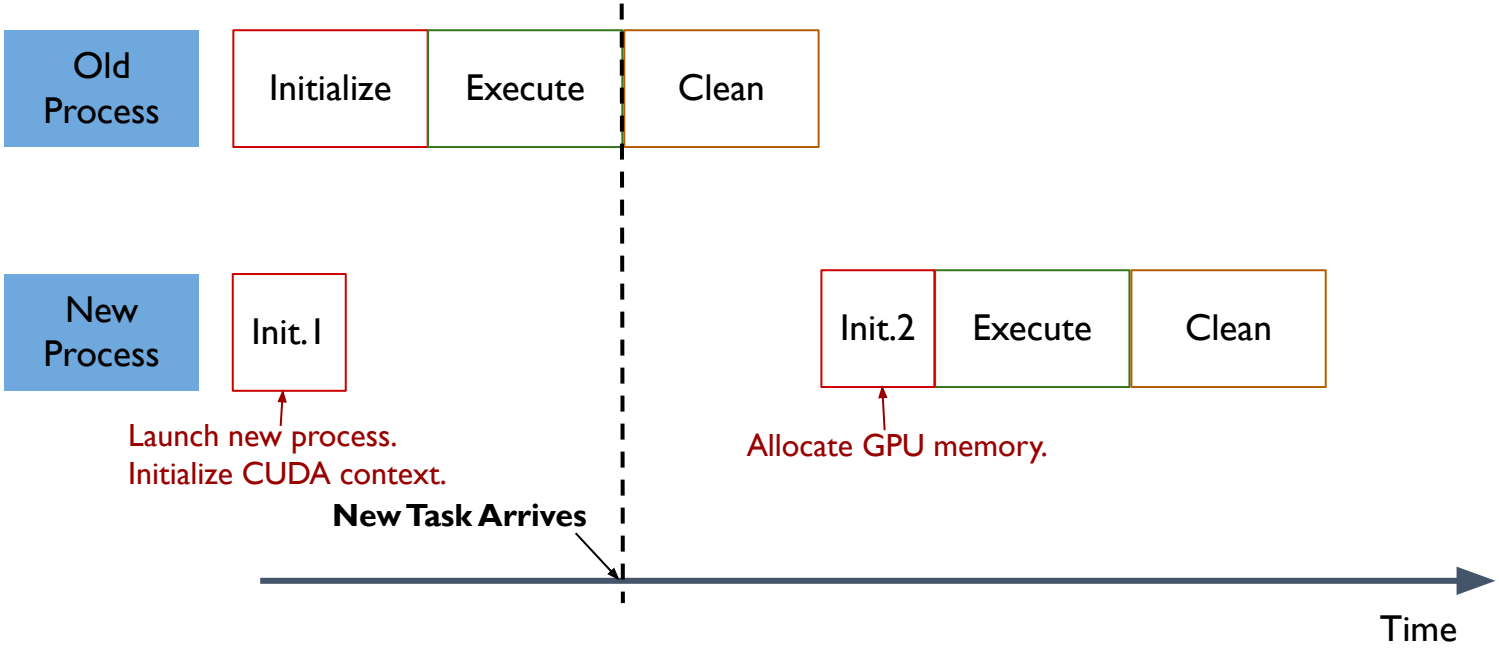
Memory Allocation → Unified Memory Management

Task Initialization
Task Cleaning
} → Active-Standby Worker Switching

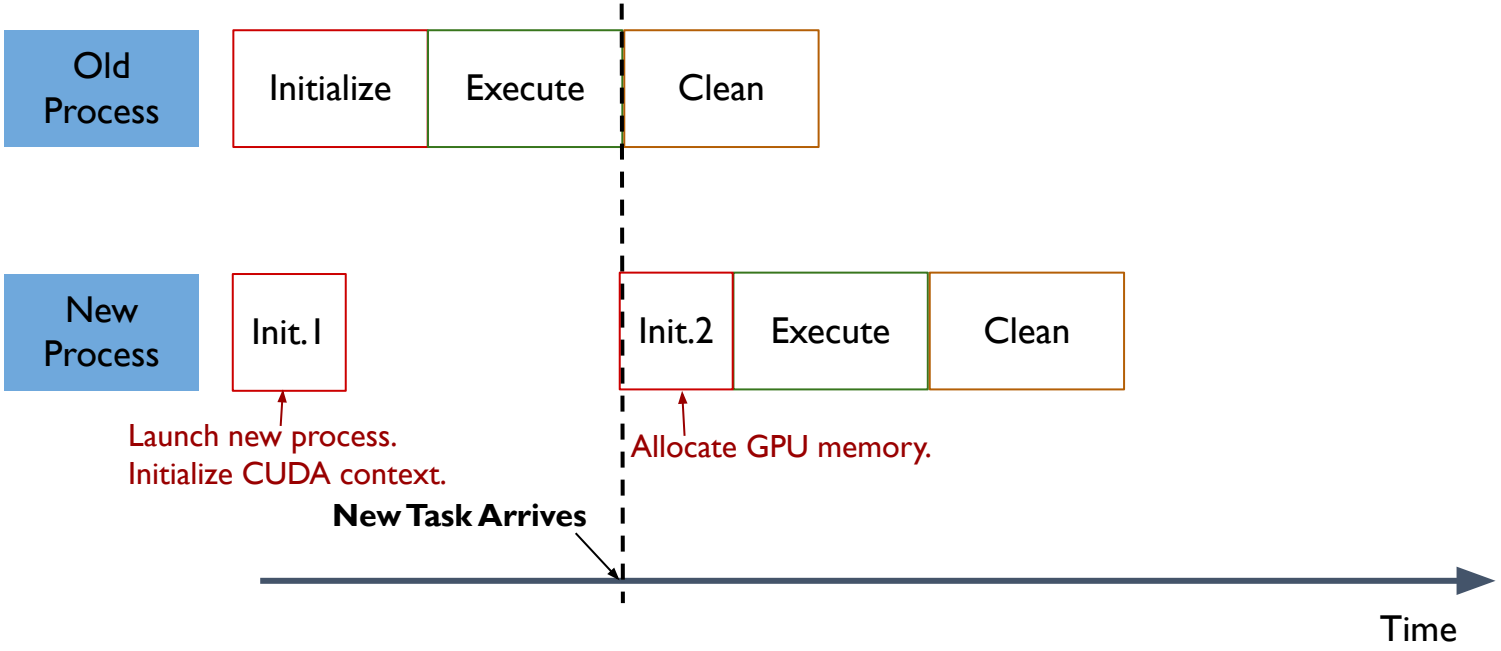
Sequential Initialization and Cleaning



Active-Standby Worker Switching



Active-Standby Worker Switching

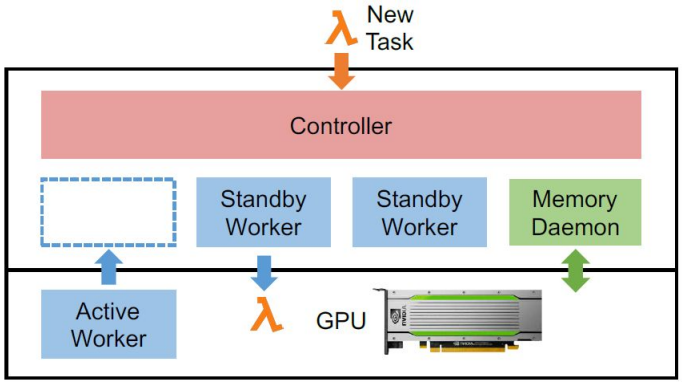


How is a task executed?

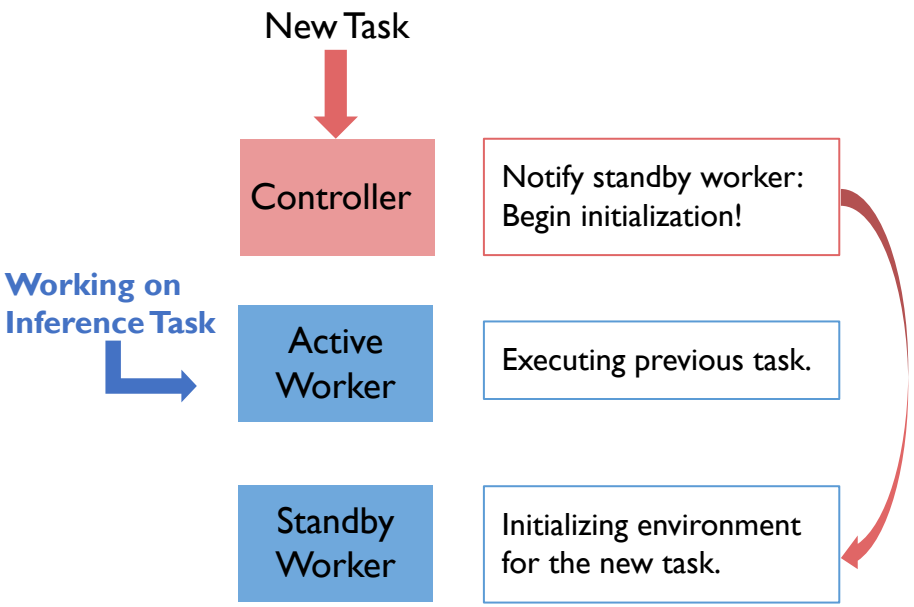
Model Transmission → Pipelined Model Transmission

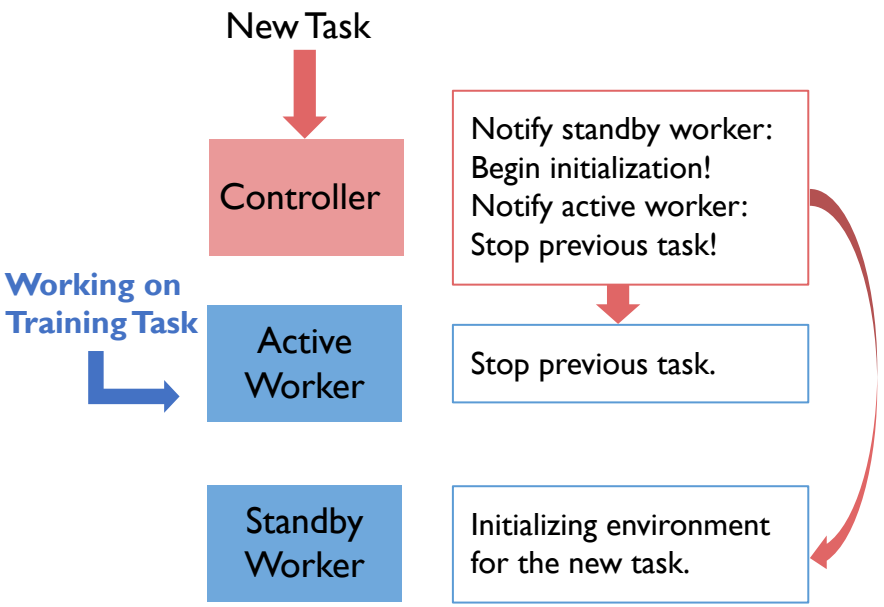
Memory Allocation → Unified Memory Management

Task Initialization
Task Cleaning
} → Active-Standby Worker Switching

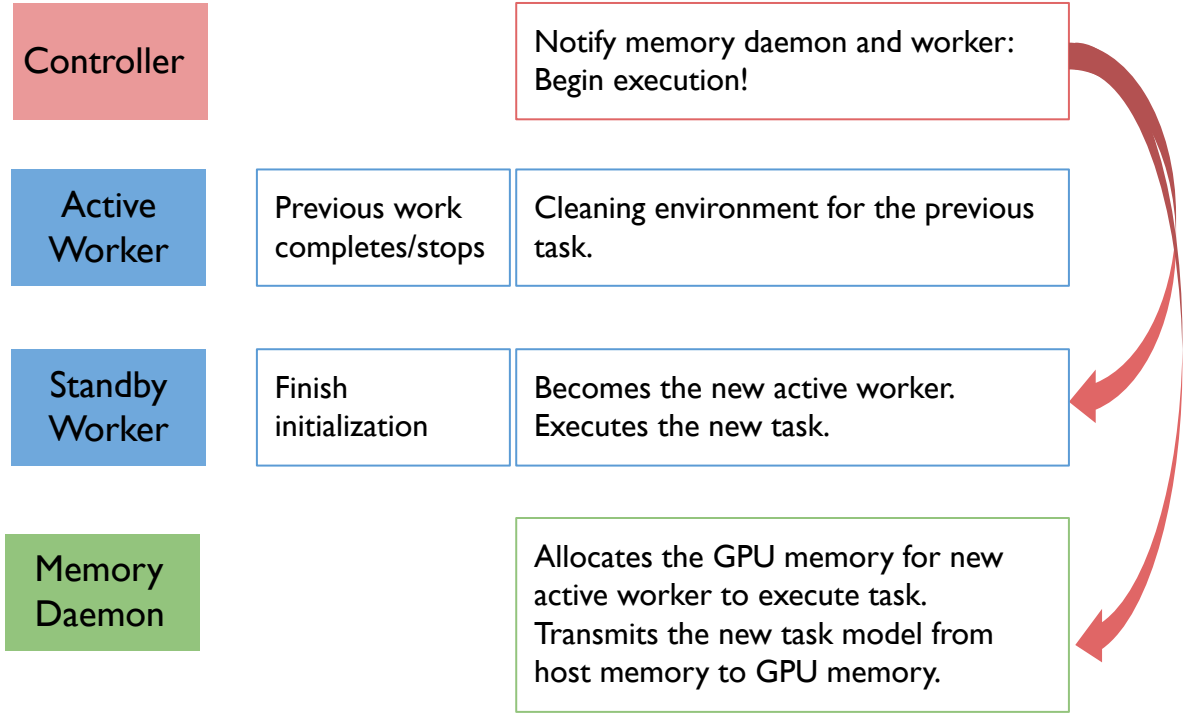


Controller starts a new task.





System executes new task.

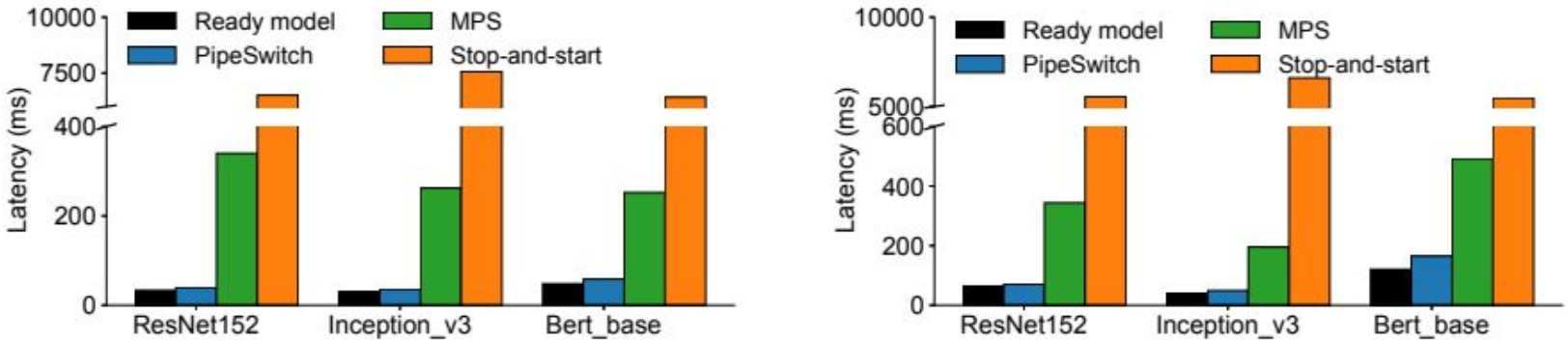


Implementation

- PyTorch framework plugins
  - Modification mainly in allocating GPU memory, sharing the GPU memory through CUDA IPC API, and getting the shared GPU memory
- Prototype in C++ and Python
  - 3600 LOC

Evaluation

End-to-End Experiments



(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).

(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Figure 5: Total latency experienced by the client for different mechanisms.

End-to-End Experiments (Cont'd)

	p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16)			g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8)		
	ResNet152	Inception_v3	Bert_base	ResNet152	Inception_v3	Bert_base
Stop-and-start	6475.40 ms	7536.07 ms	6371.32 ms	5486.74 ms	6558.76 ms	5355.95 ms
NVIDIA MPS	307.02 ms	232.25 ms	204.52 ms	259.20 ms	193.05 ms	338.25 ms
PipeSwitch	6.01 ms	5.40 ms	10.27 ms	5.57 ms	7.66 ms	34.56 ms

Table 3: Total overhead, i.e., the difference on total latency between different mechanisms and ready model.

Does Pruning Work? YES!

	ResNet152	Inception_v3	Bert_base
# of Layers	464	189	139
Algorithm 1	1.33 s	0.18 s	0.34 s
Only Pruning 1	2.09 s	0.30 s	0.88 s
Only Pruning 2	3.44 h	5.07 s	> 24 h
No Pruning	> 24 h	> 24 h	> 24 h

Table 5: Effectiveness of two pruning techniques.

Other Benchmarks

- Pipelined model transmission
- Unified memory management
- Active-standby worker switching
- Omitted for brevity

Conclusion

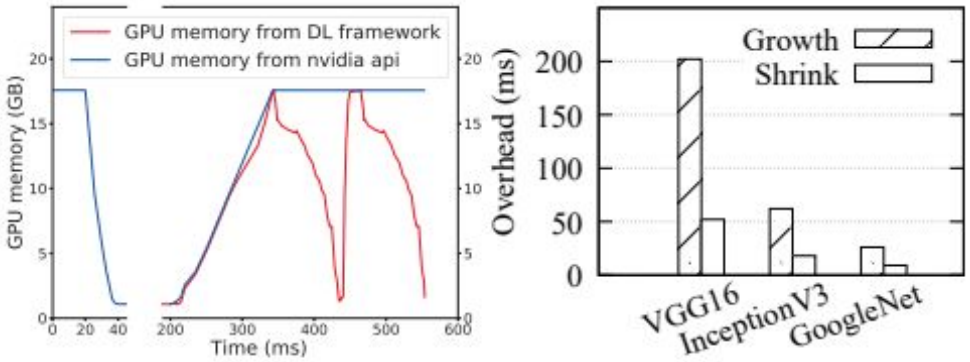
- Introduces pipelined context switching
- Total overhead of 5.4-34.6 ms
- GPU utilization near 100%

Backup Slides

AntMan

Efficient memory shrinkage and growth

- Memory shrinkage from 17.6 GB to 1.3 GB took 17 ms.
- Memory growth from 1.3 GB to 17.6 GB took 143 ms.
- Largest overhead is negligible 0.4%.



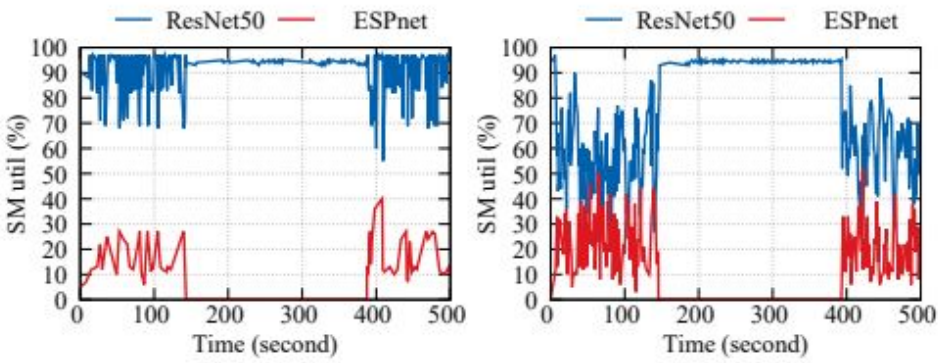
(a) A shrink-growth profiling on ResNet-50.

(b) Overhead of GPU memory scaling for typical models.

Figure 11: Efficiency of GPU memory scaling in AntMan.

Dynamic GPU computation unit scaling

- Naive packing is unfair to ESPnet because ResNet-50 launches more kernels.
- AntMan ensures the performance of ESPnet adaptively.



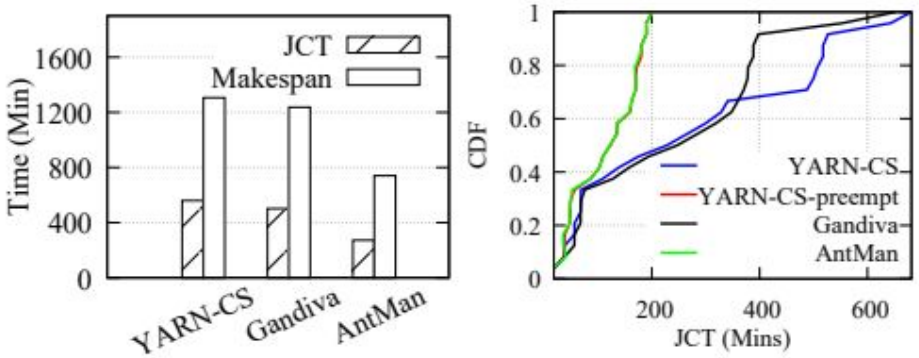
(a) Packing mode.

(b) Adaptive computation adjustment mode.

Figure 12: The SM utilization rates of packing mode in Gandiva [48] and an adaptive computation adjustment mode in AntMan for a 500s segment of execution of ESPnet and ResNet-50.

Trace Experiment

- AntMan improves
  - average JCT by 2.05x and 1.84x.
  - makespan by 1.76x and 1.67x.
- AntMan achieves similar JCT as YARN-CS-preempt with a much lower job preemption rates



(a) Comparison of YARN-CS, Gandiva, and AntMan.

(b) Job completion time of resource-guarantee jobs.

Figure 13: Trace experiment on 64 V100 GPUs.

PipeSwtich

Optimal Model-Aware Grouping

Algorithm 1 Optimal Model-Aware Grouping

```
function FindOptGrouping(B, x)
    opt_groups ← ∅, opt_groups.time ← ∞
    // find first group from layer i to j*
    j* ← x
    for layer i from x to n − 1 do
        if T(x, i) ≤ B.delay then
            j* ← i
        else
            break
    // recursively find the optimal grouping
    for layer i from j* to n − 1 do
        if opt_groups ≠ ∅ then
            // compute lower bound
            trans_time ← T(x, i) + T(i + 1, n − 1)
            exec_time ← max(T(x, i), B.delay)
                        + E(x, i) + E(i + 1, n − 1)
            lower_bound ← min(trans_time, exec_time)
            if lower_bound > opt_groups.time then
                continue
        // recursively find rest groups
        first_group ← Group(x, i)
        rest_groups ← FindOptGrouping(
                        B + first_group, i + 1)
        cur_groups ← first_group + rest_groups
        if cur_groups.time < opt_groups.time then
            opt_groups ← cur_groups
    return opt_groups
```

B = groups that have already formed from layer 0 to x-1.

x = the first layer that have not formed a group

Goal:

Recursively finds the optimal grouping strategy based on the equation.

$$F(\{\},0) = \min_{i} F(\{group(0,i)\}, i+1)$$

Optimal Model-Aware Grouping

Algorithm 1 Optimal Model-Aware Grouping

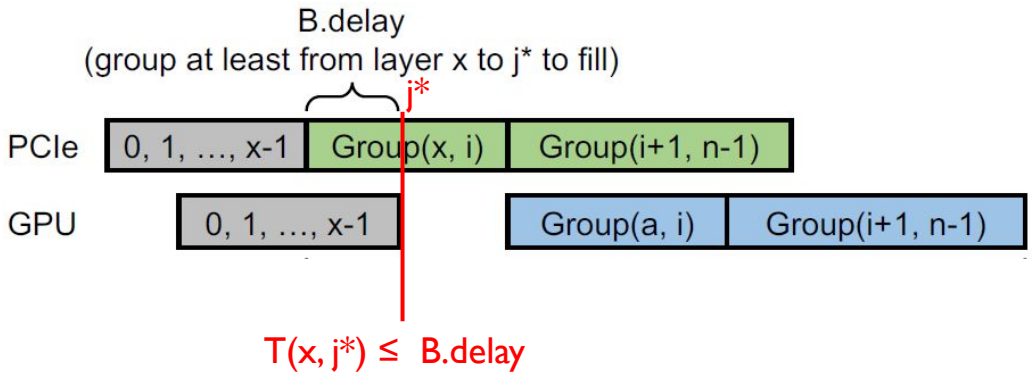
```
function FINDOPTGROUPING(B, x)
    opt_groups ← ∅, opt_groups.time ← ∞
    // find first group from layer i to j*
    j* ← x
    for layer i from x to n − 1 do
        if T(x,i) ≤ B.delay then
            j* ← i
        else
            break
    // recursively find the optimal grouping
    for layer i from j* to n − 1 do
        if opt_groups ≠ ∅ then
            // compute lower bound
            trans_time ← T(x,i) + T(i+1, n−1)
            exec_time ← max(T(x,i), B.delay)
                        + E(x,i) + E(i+1, n−1)
            lower_bound ← min(trans_time, exec_time)
            if lower_bound > opt_groups.time then
                continue
        // recursively find rest groups
        first_group ← Group(x,i)
        rest_groups ← FindOptGrouping(
                        B + first_group, i+1)
        cur_groups ← first_group + rest_groups
        if cur_groups.time < opt_groups.time then
            opt_groups ← cur_groups
    return opt_groups
```

B.delay = time to which new group can be formed

T(x, i) = transmission time of group [x,i]

j* = prune i ∈ [x, j*-1] from searching

Leverage pruning opportunity 2



Optimal Model-Aware Grouping

Algorithm 1 Optimal Model-Aware Grouping

```
function FindOptGrouping(B, x)
    opt_groups ← ∅, opt_groups.time ← ∞
    // find first group from layer i to j*
    j* ← x
    for layer i from x to n − 1 do
        if T(x,i) ≤ B.delay then
            j* ← i
        else
            break
    // recursively find the optimal grouping
    for layer i from j* to n − 1 do
        if opt_groups ≠ ∅ then
            // compute lower bound
            trans_time ← T(x,i) + T(i+1, n−1)
            exec_time ← max(T(x,i), B.delay)
                        + E(x,i) + E(i+1, n−1)
            lower_bound ← min(trans_time, exec_time)
            if lower_bound > opt_groups.time then
                continue
        // recursively find rest groups
        first_group ← Group(x,i)
        rest_groups ← FindOptGrouping(
                        B + first_group, i+1)
        cur_groups ← first_group + rest_groups
        if cur_groups.time < opt_groups.time then
            opt_groups ← cur_groups
    return opt_groups
```

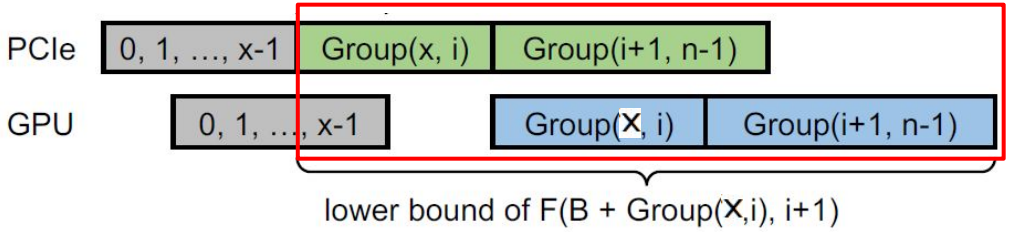
For case i

B.delay = time to which new group can be formed

T(x, i) = transmission time of group [x,i]

E(x, i) = execution time of group [x,i]

Leverage pruning opportunity I



For a chosen i ∈ [j*, n-1],

F({B+Group(x,i)} , i+1) ≥

min(Lower bound of transmission time of case i,

Lower bound of execution time of case i)

Optimal Model-Aware Grouping

Algorithm 1 Optimal Model-Aware Grouping

```
1:  function FINDOPTGROUPING(B, x)
2:      opt_groups ← ∅, opt_groups.time ← ∞
3:      // find first group from layer i to j*
4:      j* ← x
5:      for layer i from x to n − 1 do
6:          if T(x,i) ≤ B.delay then
7:              j* ← i
8:          else
9:              break
10:     // recursively find the optimal grouping
11:     for layer i from j* to n − 1 do
12:         if opt_groups ≠ ∅ then
13:             // compute lower bound
14:             trans_time ← T(x,i) + T(i+1, n−1)
15:             exec_time ← max(T(x,i), B.delay)
16:                         + E(x,i) + E(i+1, n−1)
17:             lower_bound ← min(trans_time, exec_time)
18:             if lower_bound > opt_groups.time then
19:                 continue
20:         // recursively find rest groups
21:         first_group ← Group(x,i)
22:         rest_groups ← FindOptGrouping(
23:                     B + first_group, i+1)
24:         cur_groups ← first_group + rest_groups
25:         if cur_groups.time < opt_groups.time then
26:             opt_groups ← cur_groups
27:     return opt_groups
```

Given:

group [x, i] is formed!

Line 21:

Given that group[x,i] is formed.

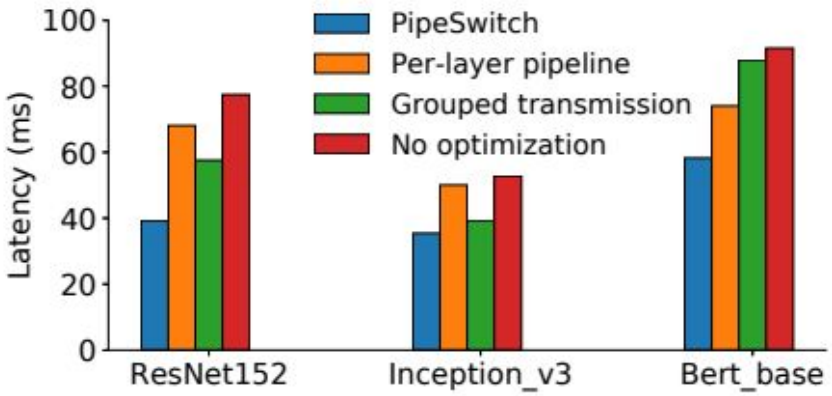
Line 22-23:

The function recursively applies itself to find the optimal groups in layers [i+1, n-1].

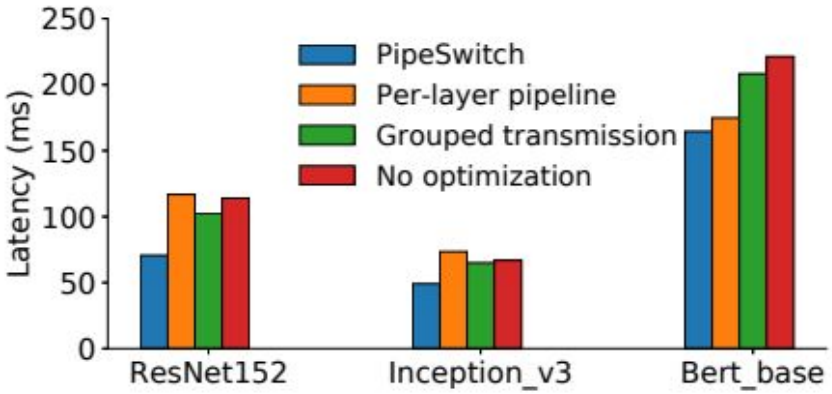
Line 24-26:

Update *opt_groups* if current strategy is better.

Pipelined Model Transmission

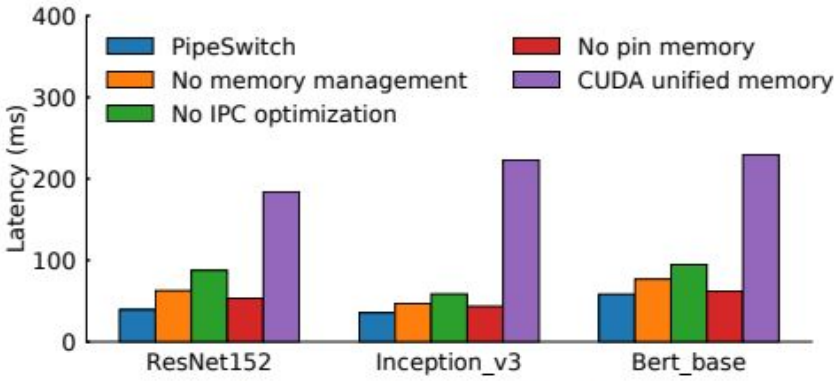


(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).

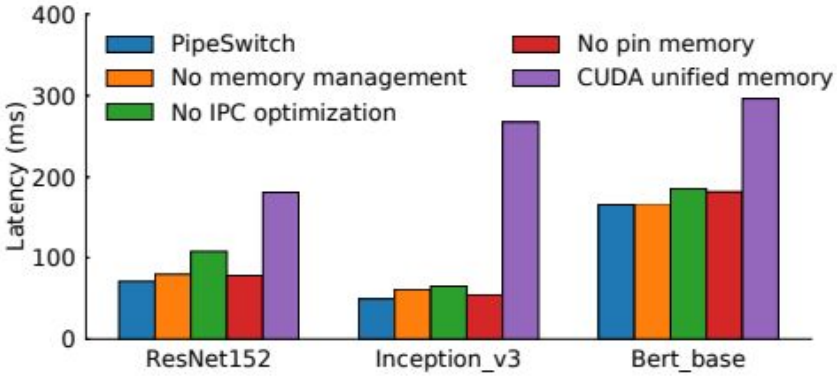


(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Unified Memory Management

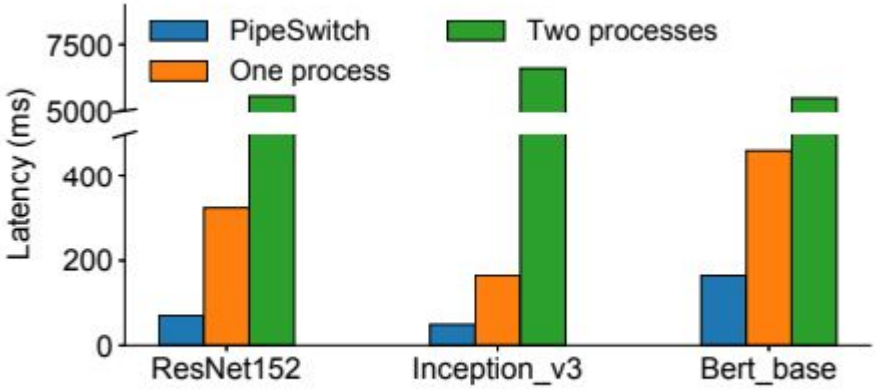


(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).

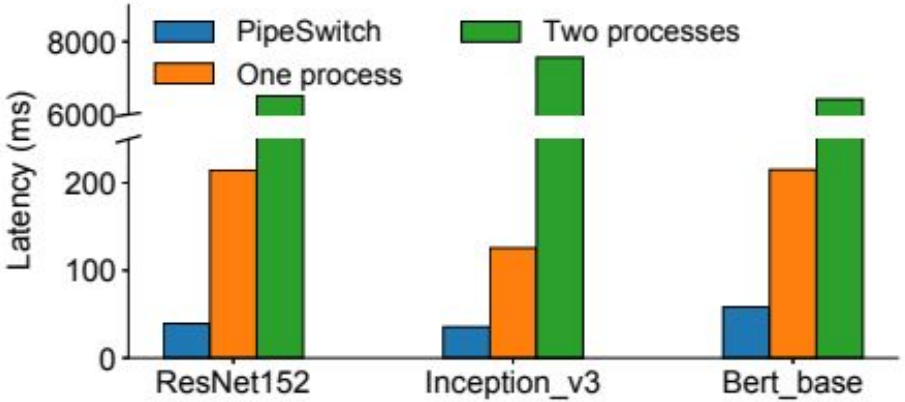


(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Active-Standby Worker Switching



(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).



(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).