PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications

Zhihao Bai, Zhen Zhang, Yibo Zhu, Xin Jin
Deep learning powers intelligent applications in many domains
Training and inference

Training
High throughput

Inference
Low latency
GPUs clusters for DL workloads
Separate clusters for training and inference

Cluster for training: GPU, GPU, GPU, GPU

Cluster for inference: GPU, GPU, GPU, GPU
Utilization of GPU clusters is low

Today: separate clusters

<table>
<thead>
<tr>
<th>Time</th>
<th>Training</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daytime</td>
<td>25%</td>
<td>50%</td>
</tr>
<tr>
<td>Midnight</td>
<td>50%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Ideal: shared clusters

<table>
<thead>
<tr>
<th>Time</th>
<th>Training</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daytime</td>
<td>100%</td>
<td>75%</td>
</tr>
<tr>
<td>Midnight</td>
<td>75%</td>
<td>50%</td>
</tr>
</tbody>
</table>
Context switching overhead is high

New model

Old model
Context switching overhead is high

Latency: 6s
Drawbacks of existing solutions

- NVIDIA MPS
  - High overhead due to contention
- Salus[MLSys’20]
  - Requires all the models to be preloaded into the GPU memory

Latency: 6s
Goal: fast context switching

- Enable GPU-efficient **multiplexing** of multiple DL apps with **fine-grained time-sharing**
- Achieve **millisecond-scale** context switching latencies and high throughput

Latency: 6s
PipeSwitch overview: architecture
PipeSwitch overview: execution

• Stop the current task and prepare for the next task.
• Execute the task with pipelined model transmission.
• Clean the environment for the previous task.
Sources of context switching overhead

- Model transmission
- Memory allocation
- Task initialization
- Task cleaning
How to reduce the overhead?

Model transmission

Pipelined model transmission

Memory allocation

Task initialization

Task cleaning
DL models have layered structures
Sequential model transmission and execution

Transmit layer 0

\[ T_0, T_1, T_2, \ldots \]

Execute layer 0

\[ T_{n-1}, E_0, E_1, E_2, \ldots \]

model transmission over PCIe

task execution on GPU
Pipelined model transmission and execution
Pipelined model transmission and execution

Transmit layer 0

PCle: $T_0 \quad T_1 \quad T_2 \quad \cdots \quad T_{n-1}$

GPU: $E_0 \quad E_1 \quad E_2 \quad \cdots \quad E_{n-1}$
Pipelined model transmission and execution

Transmit layer 1

PCIe: T_0, T_1, T_2, ..., T_{n-1}

GPU: E_0, E_1, E_2, ..., E_{n-1}

Execute layer 0
Pipelined model transmission and execution

Transmit layer 2

Execute layer 1

PCle: $T_0$ | $T_1$ | $T_2$ | $\cdots$ | $T_{n-1}$

GPU: $E_0$ | $E_1$ | $E_2$ | $\cdots$ | $E_{n-1}$
Pipelined model transmission and execution

1. Multiple calls to PCIe;
2. Synchronize transmission and execution.
Pipelined model transmission and execution

PCle

Group (0, i)  Group (i+1, j)  ...  Group (k, n-1)

GPU

Group (0, i)  Group (i+1, j)  ...  Group (k, n-1)
Pipelined model transmission and execution

- Exponential time to find the optimal strategy
- Two heuristics for pruning
\[ \lambda = 0 \]

\[ \text{next } \lambda = 3 \]

---

Solution 탐색 시간을 줄이기 위해 tree pruning이 필요.

메우 많은 group의 경위수

Figure 3: Examples for two pruning techniques.
How to reduce the overhead?

- Model transmission
- Memory allocation
- Task initialization
- Task cleaning

Pipelined model transmission
Unified memory management
Unified memory management

Manage model parameters.
Allocate GPU memory.

Memory Daemon

Pointer
Offset

Workers

GPU memory
Unified memory management

Manage model parameters. Allocate GPU memory.

Memory Daemon

Pointer

Offset

Workers

DL의 data 유형
1) model  2) output

fixed

layer마다 size 고정!

\[
\begin{bmatrix}
M \\
N
\end{bmatrix}
\times
\begin{bmatrix}
K
\end{bmatrix}
\]

or
Unified memory management

Manage model parameters. Allocate GPU memory.

Memory Daemon

Pointer
Offset

Workers

GPU memory

DL의 데이터
1) model 2) output
↓
fixed

\[
\begin{align*}
M \times N &= K \\
M &\rightarrow \vec{\mathbf{f}} \rightarrow N
\end{align*}
\]

기존 GPU의 memory 관리
DL을 구현하기에는 too-heavy

⇒ allocation 간화하고 IPC optimization
How to reduce the overhead?

- Model transmission
- Memory allocation
- Task initialization
- Task cleaning

- Pipelined model transmission
- Unified memory management
- Active-standby worker switching
Active-standby worker switching

Time

Old Task
Init.  Execute  Clean

New Task
Init.  Execute  Clean

New Task Starts
Active-standby worker switching

- Old Task
  - Init.
  - Execute
  - Clean

- New Task
  - Init.
  - Execute
  - Clean

New Task Starts
Active-standby worker switching

Time

Old Task
- Init.
- Execute
- Clean

New Task
- Init. 1
- Launch the process.
  Create CUDA context.

New Task Starts

New Task
- Init. 2
- Execute
- Clean
- Allocate GPU memory.
Active-standby worker switching

Old Task
- Init.
- Execute
- Clean

New Task
- Init. 1
- Init. 2
- Execute
- Clean

New Task Starts

Pipe Switch 가 메모리 전체를 관리하기 때문에, Free 은 아니라 단계는 parallelize.
Implementation

- Testbed: AWS EC2
  - p3.2xlarge: PCIe 3.0x16, NVIDIA Tesla V100 GPU
  - g4dn.2xlarge: PCIe 3.0x8, NVIDIA Tesla T4 GPU

- Software
  - CUDA 10.1
  - PyTorch 1.3.0

- Models
  - ResNet-152
  - Inception-v3
  - BERT-base
Evaluation

• Can PipeSwitch satisfy SLOs?

• Can PipeSwitch provide high utilization?

• How well do the design choices of PipeSwitch work?
PipeSwitch satisfies SLOs

NVIDIA Tesla V100

NVIDIA Tesla T4

Latency (ms)

0 100 200 300 400 500 600 700 800 900 1000

ResNet152 Inception_v3 Bert_base

Latency (ms)

0 100 200 300 400 500 600 700 800 900 1000

ResNet152 Inception_v3 Bert_base
PipeSwitch satisfies SLOs

NVIDIA Tesla V100

- **Ready model**
- **PipeSwitch**
- **MPS**
- **Stop-and-start**

NVIDIA Tesla T4

- **Ready Model**
- **PipeSwitch**
- **MPS**
- **Stop-and-start**

33ms
PipeSwitch satisfies SLOs

NVIDIA Tesla V100

NVIDIA Tesla T4

Latency (ms)

39ms
PipeSwitch satisfies SLOs

NVIDIA Tesla V100

- Ready model
- PipeSwitch
- MPS
- Stop-and-start

NVIDIA Tesla T4

- Ready Model
- PipeSwitch
- MPS
- Stop-and-start

340ms
PipeSwitch satisfies SLOs

NVIDIA Tesla V100

Latency (ms)

NVIDIA Tesla T4

Latency (ms)
PipeSwitch satisfies SLOs

PipeSwitch achieves low context switching latency.
PipeSwitch provides high utilization.
PipeSwitch provide high utilization

![Graph showing throughput over scheduling cycles]

- **PipeSwitch**
- **Stop-and-start**
- **MPS**

Throughput (batches/sec)

- Upper bound

Scheduling cycles

- 1s
- 2s
- 5s
- 10s
- 30s
PipeSwitch provide high utilization

![Graph showing throughput over scheduling cycles] (Note: The graph shows the throughput in batches per second over different scheduling cycles. The graph includes bars for PipeSwitch, Stop-and-start, and MPS, with an upper bound indicated.)
PipeSwitch provide high utilization
PipeSwitch provides high utilization.

PipeSwitch achieves near 100% utilization.
Summary

• GPU clusters for DL applications suffer from low utilization
  • Limited share between training and inference workloads

• PipeSwitch introduces pipelined context switching
  • Enable GPU-efficient multiplexing of DL apps with fine-grained time-sharing
  • Achieve millisecond-scale context switching latencies and high throughput
Thank you!
zbai1@jhu.edu