FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks

Jonghyun Bae$^1$ Jongsung Lee$^{1,2}$ Yunho Jin$^1$ Sam Son$^1$ Shine Kim$^{1,2}$ Hakbeom Jang$^2$ Tae Jun Ham$^1$ Jae W. Lee$^1$

$^1$Seoul National University $^2$Samsung Electronics
Rise of DNNs

- DNNs are the key enabler of today’s AI application

Object detection and classification [1]

Speech-to-text [2]

Rise of DNNs

- DNNs are the key enabler of today’s AI application
- Two types of DNN workloads: Training >> Inference
  - 3x the computation: forward propagation, backward propagation, and weight update

Rise of DNNs

- DNNs are the key enabler of today’s AI application
- Two types of DNN workloads: Training >> Inference
  - 3x the computation: forward propagation, backward propagation, and weight update

Rise of DNNs

- DNNs are the key enabler of today’s AI application
- Two types of DNN workloads: Training >> Inference
  - 3x the computation: forward propagation, backward propagation, and weight update

Data Reuse in DNN Training

- Data reuse pattern from forward propagation to backward propagation
  - Requiring input activation ($X$), and output error ($dY$) to calculate input gradient map ($dX$), weight gradient ($dW$), and finally weight ($W$)

Simplified data reuse pattern in a layer
Data Reuse in DNN Training

- Data reuse pattern from forward propagation to backward propagation
  - Requiring input activation ($X$), and output error ($dY$) to calculate input gradient map ($dX$), weight gradient ($dW$), and finally weight ($W$)

```
<table>
<thead>
<tr>
<th>Activation (X)</th>
<th>Weight (W)</th>
<th>Weight grad. (dW)</th>
<th>Error (dX)</th>
<th>Error (dY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward pass</td>
<td>Activation (Y)</td>
<td>Weight update</td>
<td>Backward pass</td>
<td></td>
</tr>
</tbody>
</table>
```

Simplified data reuse pattern in a layer
Data Reuse in DNN Training

- Data reuse pattern from forward propagation to backward propagation
  - Requiring input activation ($X$), and output error ($dY$) to calculate input gradient map ($dX$), weight gradient ($dW$), and finally weight ($W$)

Simplified data reuse pattern in a layer
Memory Capacity Wall in DNN Training

- DRAM footprint increases with (1) **deeper neural nets** (for accuracy) and (2) **larger batch size** (for training throughput)

![GPU memory usage for DNN training](image)

- **Baseline GPU mem**
- **Input + Intermediate result**
- **Weight**
- **Temporary buf.**

<table>
<thead>
<tr>
<th>Model</th>
<th>Batch Size</th>
<th>Memory Usage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-1922</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>DenseNet-1001</td>
<td>2x</td>
<td>2x</td>
</tr>
<tr>
<td>BERT-XLarge</td>
<td>8x</td>
<td>8x</td>
</tr>
<tr>
<td>HBMP</td>
<td>1x</td>
<td>2x</td>
</tr>
<tr>
<td></td>
<td>2x</td>
<td>8x</td>
</tr>
</tbody>
</table>

**Legend:**
- □ Baseline GPU mem
- □ Input + Intermediate result
- □ Weight
- □ Temporary buf.
Overcoming GPU Memory Capacity Wall

- Previous approach: Buffering-on-memory
  - Host DRAM BW contention by BW-intensive task on CPU (e.g., data augmentation)

![Diagram showing DRAM bandwidth contention and buffering-on-memory](image)

<table>
<thead>
<tr>
<th>Model</th>
<th>50%</th>
<th>70%</th>
<th>90%</th>
<th>50%</th>
<th>70%</th>
<th>90%</th>
<th>50%</th>
<th>70%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-1922</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DenseNet-1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERT-XLarge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Normalized throughput of buffering-on-memory with bandwidth-intensive tasks on CPU
Overcoming GPU Memory Capacity Wall

- Previous approach: Buffering-on-memory
  - Host DRAM BW contention by BW-intensive task on CPU (e.g., data augmentation)

- New solution: Buffering-on-SSD
Overcoming GPU Memory Capacity Wall

- **Previous approach: Buffering-on-memory**
  - Host DRAM BW contention by BW-intensive task on CPU (e.g., data augmentation)

- **New solution: Buffering-on-SSD**
  - With peer-to-peer communication, no host DRAM bandwidth or CPU cycles consumed
Our Proposal: FlashNeuron

- **Key idea**: DNN training using a high-performance SSD as a backing store
  - **Offloading scheduler**: Identify a set of tensors to offload and generates an offloading schedule
  - **Memory manager**: Manage offloading/prefetching and tensor allocation/deallocation
  - **Lightweight user-level I/O stack**: Customized stack for p2p communication

- **Key results**
  - Batch size: \(12.4x\) to \(14.0x\) over the maximum allowable batch size on 16GB HBM
  - Training throughput improvement: Up to \(37.8\%\) (\(30.3\%\) on average) over the baseline
  - Cost efficiency: \(35.3x\) higher cost efficiency assuming the same capacity of DRAM and SSD
System Overview

Training model structure

DNN Training Framework

Operation Core

Memory Manager

Offloading Scheduler

Peer-to-peer Direct Storage Access

User-space NVMe Driver
System Overview

DNN Training Framework

Operation Core

Memory Manager

Peer-to-peer Direct Storage Access

User-space NVMe Driver

Profiling result

Offloading Scheduler

Scheduling result

Training model structure
System Overview

Training model structure

DNN Training Framework

Operation Core

Memory Manager

Tensor offload/prefetch

Memory (de-)allocation

Result of offload/prefetch

Offload/prefetch using tensor index

Peer-to-peer Direct Storage Access

User-space NVMe Driver

Offloading Scheduler

System Overview

Training model structure

DNN Training Framework

Operation Core

Memory Manager

Tensor offload/prefetch

Memory (de-)allocation

Result of offload/prefetch

Offload/prefetch using tensor index

Peer-to-peer Direct Storage Access

User-space NVMe Driver

Offloading Scheduler
**System Overview**

![Diagram of the system overview](diagram.png)

- **Training model structure**
- **DNN Training Framework**
  - **Operation Core**
- **Memory Manager**
- **Peer-to-peer Direct Storage Access**
  - **User-space NVMe Driver**
  - **Offloading Scheduler**
- **I/O cmd.**
- **Completion**

- **Offload/prefetch using tensor index**
- **Result of offload/prefetch**
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

- Compression-friendly:
  - : Uncompressible
  - : Low
  - : High

- Tensor Size:
  - 6MB
  - 2MB
  - 4MB
  - 4MB
  - 4MB
  - 2MB
  - 2MB
  - \(= 24 \text{MB}\)

- Spillover: 16MB
- GPU memory: 8MB
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

Compression-friendly:

- Uncompressible
- Low
- High

Tensor Size:

- 6MB
- 2MB
- 4MB
- 4MB
- 4MB
- 2MB
- 2MB

Spillover: 10MB

GPU memory: 8MB

Offloading tensor A

= 18MB
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

Compressed-friendly:  

| Compression-friendy | Uncompressible | Low | High |

Tensor Size:  

| A | B | C | D | E | F | G |
| 6MB | 2MB | 4MB | 4MB | 4MB | 2MB | 2MB |

Spillover: 8MB

GPU memory: 8MB

Offloading tensors A and B

4MB + 4MB + 4MB + 2MB + 2MB = 16MB
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

Tensor Size:
- A: 6MB
- B: 2MB
- C: 4MB

Spillover: 4MB
GPU memory: 8MB

Compression-friendly:
- Uncompressible
- Low
- High

Offloading:
- Tensor A
- Tensor B
- Tensor C

4MB + 4MB + 2MB + 2MB = 12MB
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

Compression-friendly:
- Uncompressible
- Low
- High

Tensor Size:
- 6MB
- 2MB
- 4MB

Spillover: 0MB

GPU memory: 8MB

Offloading tensor A
Offloading tensor B
Offloading tensor C
Offloading tensor D

4MB + 2MB + 2MB = 8MB
Offloading Scheduler: Phase 1

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning

**Compression-friendly:**
- : Uncompressible
- : Low
- : High

**Tensor Size:**
- A: 6MB
- B: 2MB
- C: 4MB
- D: 4MB
- E: 4MB
- F: 2MB
- G: 2MB

**Spillover:** 0MB

**GPU memory:** 8MB

- Offloading tensor A
- Offloading tensor B
- Offloading tensor C
- Offloading tensor D
**Offloading Scheduler: Phase 2**

- Finding an optimal scheduler for a given target batch size
- **Phase 1**
  - Iteratively select a certain number of tensors from the beginning
- **Phase 2**
  - Replace the tensors as offloading candidates with more compression-friendly tensors

---

**Compression-friendly:**
- : Uncompressible
- : Low
- : High

**Tensor Size:**
- 6MB
- 2MB
- 4MB
- 4MB

**Spillover:** 0MB

**GPU memory:** 8MB

**Offloading:**
- Tensor A
- Tensor B
- Tensor C
- Tensor D
- Tensor E
- Tensor F
- Tensor G

**Offloading candidates:**
- Tensor A
- Tensor B
- Tensor C
- Tensor D
Offloading Scheduler: Phase 2

- Finding an optimal scheduler for a given target batch size
- **Phase 1**
  - Iteratively select a certain number of tensors from the beginning
- **Phase 2**
  - Replace the tensors as offloading candidates with more compression-friendly tensors

Compression-friendly: □: Uncompressible □: Low □: High

Tensor Size: 6MB 2MB 4MB 4MB 4MB 2MB 2MB

Spillover: 0MB

GPU memory: 8MB

- Offloading tensor A
- Offloading tensor B
- Offloading tensor C
- Offloading tensor E

A
B
C
D
E
F
G
Offloading Scheduler: Phase 2

- Finding an optimal scheduler for a given target batch size
- Phase 1
  - Iteratively select a certain number of tensors from the beginning
- Phase 2
  - Replace the tensors as offloading candidates with more compression-friendly tensors

Compression-friendly:

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Size</th>
<th>Spillover</th>
<th>GPU Memory</th>
<th>Offloading</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6MB</td>
<td>0MB</td>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>4MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>21MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>2MB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Peer-to-peer Direct Storage Access (P2P-DSA)

- Lightweight I/O stack to enable direct tensor offloading/prefetching
- Example walk-through

1. Index “1” transfer request
2. Set contiguous LBAs

<table>
<thead>
<tr>
<th>Metadata Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

P2P-DSA

Queue

PCIe Bus

GPU Base Address Register (BAR)

GPU

NVMe SSD

Index 0

0 1 ... 2048 ... 8192 ... Max. LBA
Peer-to-peer Direct Storage Access (P2P-DSA)

- Lightweight I/O stack to enable direct tensor offloading/prefetching
- Example walk-through

![Diagram of P2P-DSA]

1. LBA Allocator
2. Metadata Table
3. Create and push cmd.
4. Issue cmd.

**Metadata Table**

<table>
<thead>
<tr>
<th>Index</th>
<th>LBA</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>2048</td>
<td></td>
</tr>
</tbody>
</table>

**LBA Allocator**

- Peer-to-peer Direct Storage Access (P2P-DSA)
Peer-to-peer Direct Storage Access (P2P-DSA)

- Lightweight I/O stack to enable direct tensor offloading/prefetching
- Example walk-through

<table>
<thead>
<tr>
<th>Index</th>
<th>LBA</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>2048</td>
<td>True</td>
</tr>
</tbody>
</table>

Index 0

Transfer

Index 1

Max. LBA

0 1 ... 2048 ... 8192 ...
# Methodology

## System configurations

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon Gold 6244 CPU 8 cores @ 3.60GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100 16GB PCIe</td>
</tr>
<tr>
<td>Memory</td>
<td>Samsung DDR4-2666 64GB (32GB x 2)</td>
</tr>
<tr>
<td>Storage</td>
<td>Samsung PM1725b 8TB PCIe Gen3 8-lane x 2 (Seq. write: 3.3GB/s, seq. read: 6.3GB/s)</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu server 18.04.3 LTS</td>
</tr>
<tr>
<td>Python</td>
<td>Version 3.7.3</td>
</tr>
<tr>
<td>PyTorch</td>
<td>Version 1.2</td>
</tr>
</tbody>
</table>

## DNN models and datasets

<table>
<thead>
<tr>
<th>Network</th>
<th>Dataset</th>
<th># of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-1922</td>
<td>ImageNet</td>
<td>1922</td>
</tr>
<tr>
<td>DenseNet-1001</td>
<td>ImageNet</td>
<td>1001</td>
</tr>
<tr>
<td>BERT-XLarge</td>
<td>SQuAD 1.1</td>
<td>48 transformer blocks</td>
</tr>
<tr>
<td>HBMP</td>
<td>SciTail</td>
<td>24 hidden layers</td>
</tr>
</tbody>
</table>
Evaluation: Overall Results

- **12.4x to 14x** batch size increment compared to the baseline using GPU memory only
- **Up to 37.8% (30.3% on average)** training throughput improvement

<table>
<thead>
<tr>
<th>Baseline</th>
<th>P2P</th>
<th>P2P+CSR</th>
<th>P2P+FP16</th>
<th>Buffering-on-SSD (FlashNeuron)</th>
<th>Buffering-on-memory</th>
</tr>
</thead>
</table>

- **ResNet-1922**
  - Images/sec vs. Batch size
- **DenseNet-1001**
  - Images/sec vs. Batch size
- **BERT-XLarge**
  - Sequences/sec vs. Batch size
- **HBMP**
  - Sequences/sec vs. Batch size
Evaluation: Co-locating Bandwidth-Intensive Tasks on CPU

- **Throughput of DNN training on GPU**
  - Buffering-on-memory: **40.2%** throughput degradation when CPU utilizes 90% of the memory BW
  - FlashNeuron: **20.2%** throughput gain when CPU utilizes 90% of the memory BW

![Graph showing throughput with different scenarios](image)

<table>
<thead>
<tr>
<th>Model</th>
<th>Buffering-on-SSD (FlashNeuron)</th>
<th>Buffering-on-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-1922</td>
<td><img src="image" alt="Graph data" /></td>
<td><img src="image" alt="Graph data" /></td>
</tr>
<tr>
<td>DenseNet-1001</td>
<td><img src="image" alt="Graph data" /></td>
<td><img src="image" alt="Graph data" /></td>
</tr>
<tr>
<td>BERT-XLarge</td>
<td><img src="image" alt="Graph data" /></td>
<td><img src="image" alt="Graph data" /></td>
</tr>
<tr>
<td>HBMP</td>
<td><img src="image" alt="Graph data" /></td>
<td><img src="image" alt="Graph data" /></td>
</tr>
</tbody>
</table>
FlashNeuron enables large-batch training of very deep and wide neural networks

• Identify a bandwidth contention problem in recent buffering-on-memory proposal

• Introduce a novel offloading scheduler to fully utilize the scarce SSD write bandwidth

• Implement a lightweight user-space I/O stack customized for DNN training
Thank You!

Source code of FlashNeuron is available at https://github.com/SNU-ARC/flashneuron.git