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

19th USENIX Conference on File and Storage Technologies (21')

Jonghyun Bae; Jongsung Lee; Yunho Jin; Sam Son; Shine Kim;
Hakbeom Jang; Tae Jun Ham and Jae W. Lee

Paper Notes

By JeongHa Lee

Abstract

Deep neural networks (DNNs) are widely used in various AI application domains such as computer vision, natural language processing, autonomous driving, and bioinformatics. As DNNs continue to get wider and deeper to improve accuracy, the limited DRAM capacity of a training platform like GPU often becomes the limiting factor on the size of DNNs and batch size—called *memory capacity wall*. Since increasing the batch size is a popular technique to improve hardware utilization, this can yield a suboptimal training throughput. Recent proposals address this problem by offloading some of the intermediate data (e.g., feature maps) to the host memory. However, they fail to provide robust performance as the training process on a GPU contends with applications running on a CPU for memory bandwidth and capacity. Thus, we propose FlashNeuron, the *first* DNN training system using an NVMe SSD as a backing store. To fully utilize the limited SSD write bandwidth, FlashNeuron introduces an offloading scheduler, which selectively offloads a set of intermediate data to the SSD in a compressed format without increasing DNN evaluation time. FlashNeuron causes minimal interference to CPU processes as the GPU and the SSD directly communicate for data transfers. Our evaluation of FlashNeuron with four state-of-the-art DNNs shows that FlashNeuron can increase the batch size by a factor of 12.4x to 14.0x over the maximum allowable batch size on NVIDIA Tesla V100 GPU with 16GB DRAM. By employing a larger batch size, FlashNeuron also improves the training throughput by up to 37.8% (with an average of 30.3%) over the baseline using GPU memory only, while minimally disturbing applications running on CPU.

Introduction

- Unlike inference, the training algorithm **reuses the intermediate results** (e.g., feature maps) produced by a forward propagation during the backward propagation

  ➔ requiring a lot of memory space

  - **The use of multiple GPUs** : can achieve near-linear improvements in throughput.
    - ↔ comes with the linear increase in the GPU cost
  - Utilize the **host CPU memory as a backing store (most popular!!) ➔ buffering-on-memory**
    - ↔ training process on the GPU contends with applications running on the CPU for memory bandwidth and capacity (e.g., data augmentation tasks)

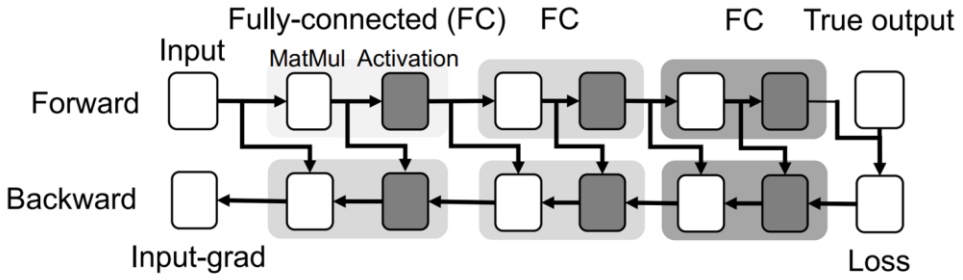


Figure 1: DNN training iteration and data reuse pattern.

- **FlashNeuron**, the first DNN training system using a high-performance SSD as a backing store. **➔ buffering-on-SSD**
  - Kernel side: offloading scheduler
  - Host side: lightweight user-level I/O stack (utilizing GPU-Direct technology)

Overcoming GPU Memory Capacity Wall

- The GPU training throughput of a **buffering-on-memory** system while the CPU is continuously running a multi-threaded data augmentation task.
  ➔ The throughput is noticeably degraded.

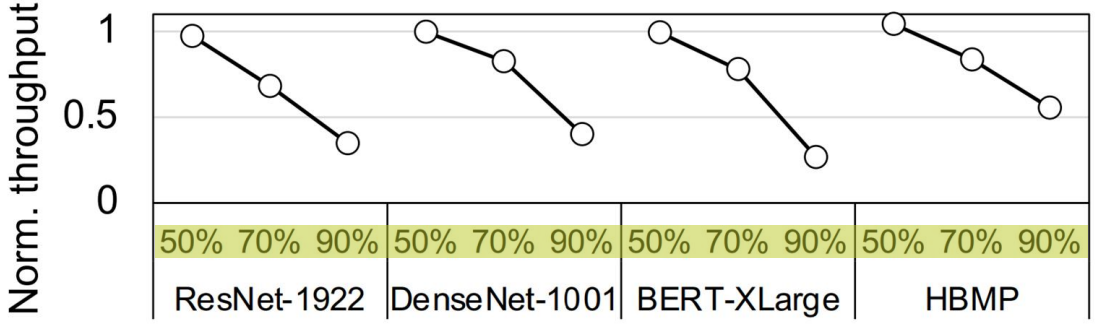


Figure 4: Normalized throughput of buffering-on-memory system (vDNN [55]-like) when the host CPU is running a data augmentation with varying degrees of contention.

- The amount of **memory bandwidth consumed by CPU task**
  ➔By controlling the number of data augmentation threads
  (50%: 21GB/s, 70%: 29GB/s, 90%: 36GB/s)

- The proposed solution buffers inputs and intermediate data (tensors) to SSDs.
  ➔ **buffering-on-SSD**
  - Specifically, This study leverage **direct peer-to-peer communication between the GPU and NVMe SSD** devices so that data buffering does not consume either host CPU cycles or memory bandwidth.

FlashNeuron Design

FlashNeuron is a library that can be integrated into popular DNN execution frameworks.

- **Offloading scheduler**
  - Identifies a set of **tensors to offload**
  - Generates an **offloading schedule**
- **Memory manager**
  - **Orchestrates data transfers** between the GPU memory and the SSDs using p2p direct storage access.
- **Peer-to-peer direct storage access**
  - **Communication between the GPU and NVMe SSD devices** without consuming either host CPU cycles or memory bandwidth

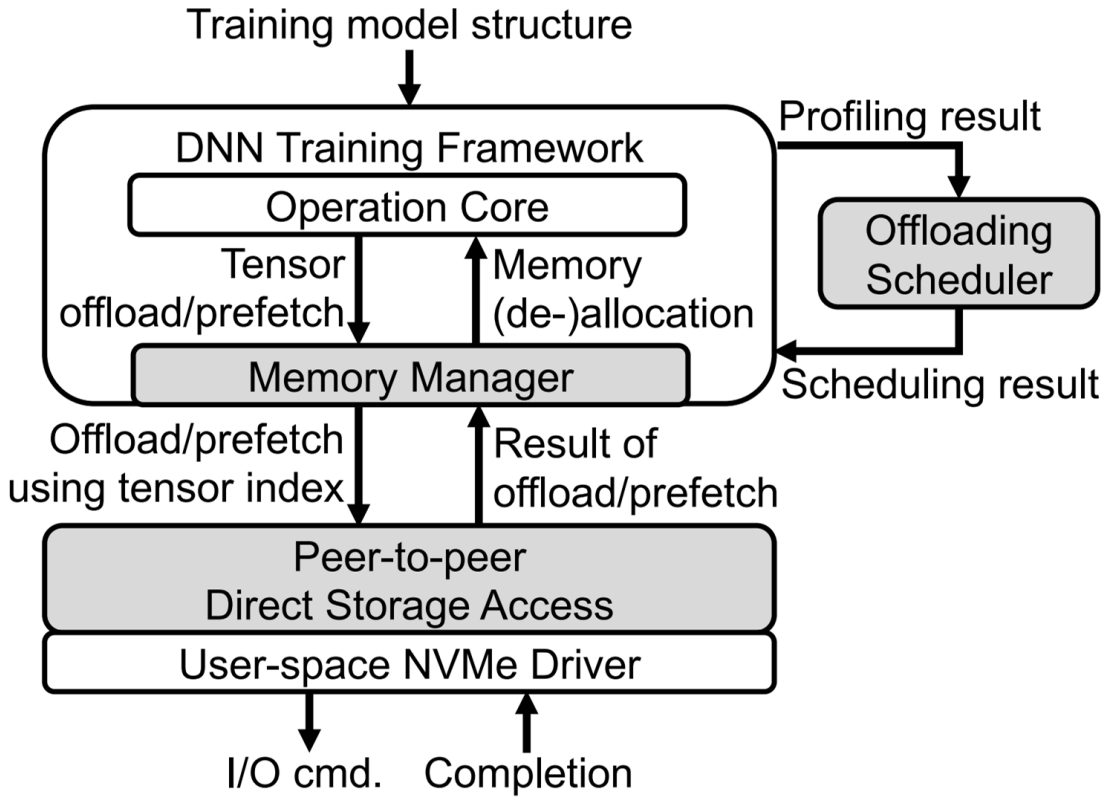


Figure 5: System overview of FlashNeuron.

FlashNeuron Design

Memory Manager

1. Tensor Allocation/Deallocation
   - **custom memory allocator**
     - **Reserve the whole GPU memory space initially** and manages memory allocation/deallocation.
       ➔ by tracking the lifetime of a tensor (reference counting mechanism)
   - **memory fragmentation problem** (in the GPU physical memory address space)
     - memory-resident (i.e., not offloaded) tensors from the lowest end of the memory address space
     - ephemeral (i.e., offloaded) tensors from the highest end of the memory address space.
2. Managing Offloading and Prefetching
3. Augmented Compressed-Sparse Row (CSR)* Compression and Decompression
   - The memory manager applies CSR compression **if the compression ratio** is **greater than one**.
   - The CSR compression is **only applied to output tensors of ReLU**.
4. Use of a Half-precision Floating Points (FP16) for Offloaded Tensors
   - During a forward path, the memory manager **converts the offloaded tensor from FP32 to FP16**.
   - During a backward propagation, the offloaded FP16 tensor is prefetched, **padded to FP32**, and reused.

Augmented Compressed-Sparse Row (CSR)

- Since a tensor is a multi-dimensional matrix, we **cast the tensor into a two-dimensional matrix** whose column has 128 entries.
- Then, we apply a **slightly different CSR format** where we replace a vector storing the column index of each element (often called JA vector) to a **set of bit-vectors** where **each bit vector represents a set of nonzero elements for a row**.

$$A_{IJ} = \begin{pmatrix} \underset{(0,0)}{10} & 0 & 0 & \underset{(0,3)}{12} & 0 \\ 0 & 0 & \underset{(1,2)}{11} & 0 & \underset{(1,4)}{13} \\ 0 & \underset{(2,1)}{16} & 0 & 0 & 0 \\ 0 & 0 & \underset{(3,2)}{11} & 0 & \underset{(3,4)}{13} \end{pmatrix}$$

$$\text{데이터}(A) = \begin{pmatrix} \underset{(0,0)}{10} & \underset{(0,3)}{12} & \underset{(1,2)}{11} & \underset{(1,4)}{13} & \underset{(2,1)}{16} & \underset{(3,2)}{11} & \underset{(3,4)}{13} \end{pmatrix}$$

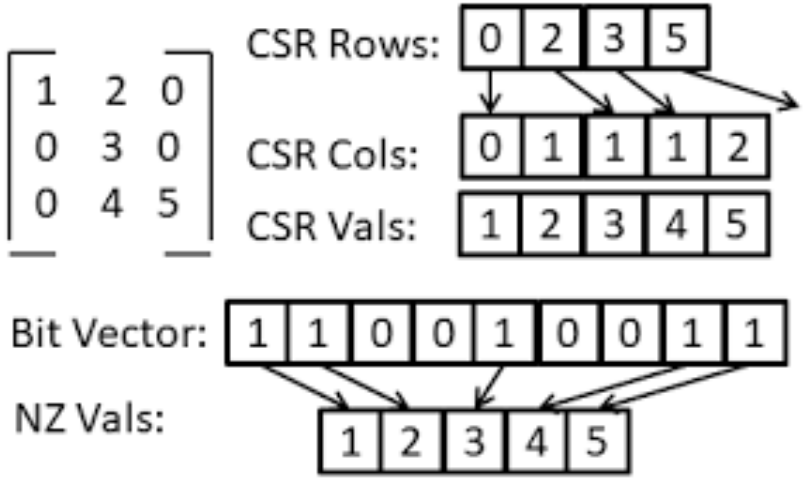
$$\text{열 인덱스 값}(JA) = \begin{pmatrix} \underset{(0}{0} & \underset{)}{3} & \underset{(1}{2} & \underset{)}{4} & \underset{(2)}{1} & \underset{(3}{2} & \underset{)}{4} \end{pmatrix}$$

$$\text{행 압축 정보}(IA) = \begin{pmatrix} \underset{(0)}{0} & \underset{(1)}{2} & \underset{(2)}{4} & \underset{(3)}{5} & \underset{(4)}{7} \end{pmatrix}$$

The row-compression metadata array (IA)

[the starting row index, the number of data items in the starting row, the cumulative data count up to the second row, … , … , the cumulative data count up to the last row]

https://ko.wikipedia.org/wiki/%EC%84%B1%EA%B8%B4_%ED%96%89%EB%A0%AC



https://dl.acm.org/doi/fullHtml/10.1145/3422575.3422777

- By doing so, the size of CSR format representation
  - ➔decreases by *8 bits (to represent the column index)* × *the number of nonzero elements in the matrix*
  - ➔increases by *1 bit* × *the total number of elements in the matrix*
- This representation is beneficial when more than one-eighth of all the elements are nonzero.

FlashNeuron Design

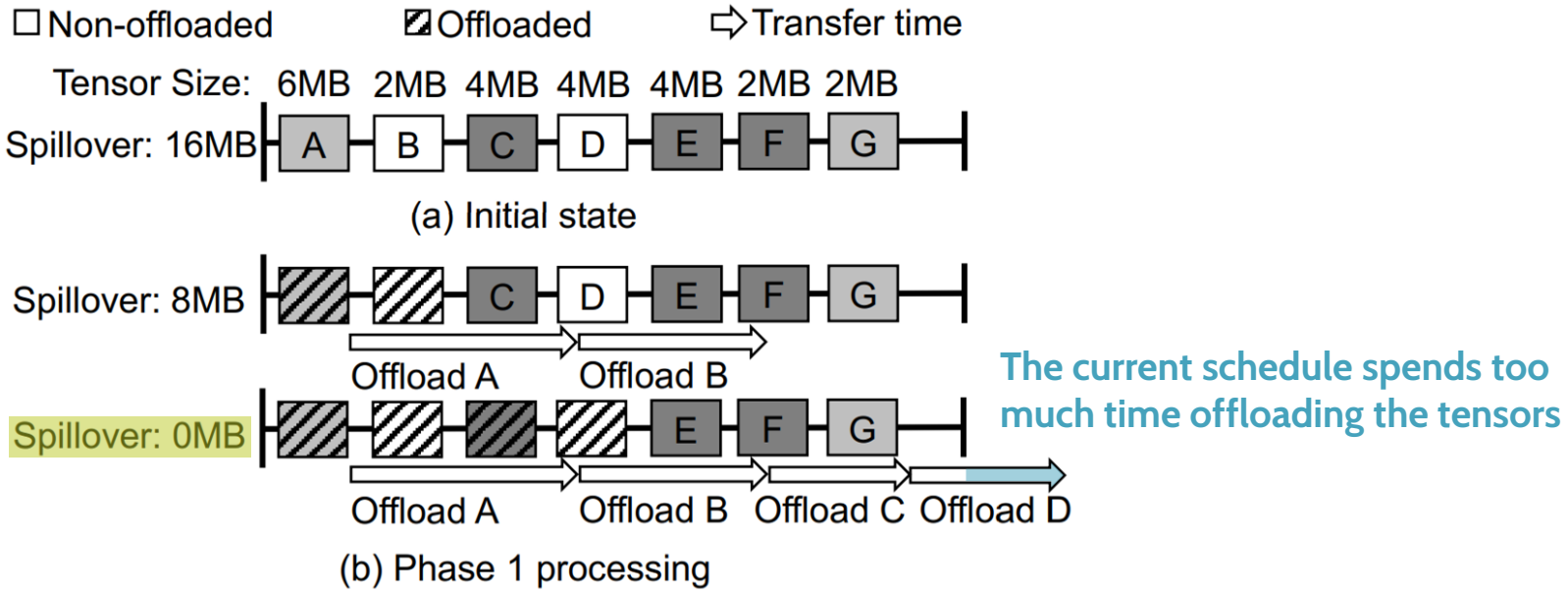
Offloading Scheduler

- Profiling Iteration
  - At profiling iteration, all the tensors that are buffered during the forward pass are offloaded to the SSD ➔ the system can run with a large target batch size.
  - The information collected during this iteration is passed to the offloading scheduler.
    i) the size of each buffered tensor
    ii) the time it takes to offload each buffered tensor
    iii) the expected compression ratio of a tensor using CSR format and half-precision floating-point conversion
    iv) the execution time for the forward pass and the backward pass (excluding tensor offloading time)
    v) the total size of the other memory-resident objects (e.g., weights, temporary workspace)

FlashNeuron Design

Offloading Scheduler

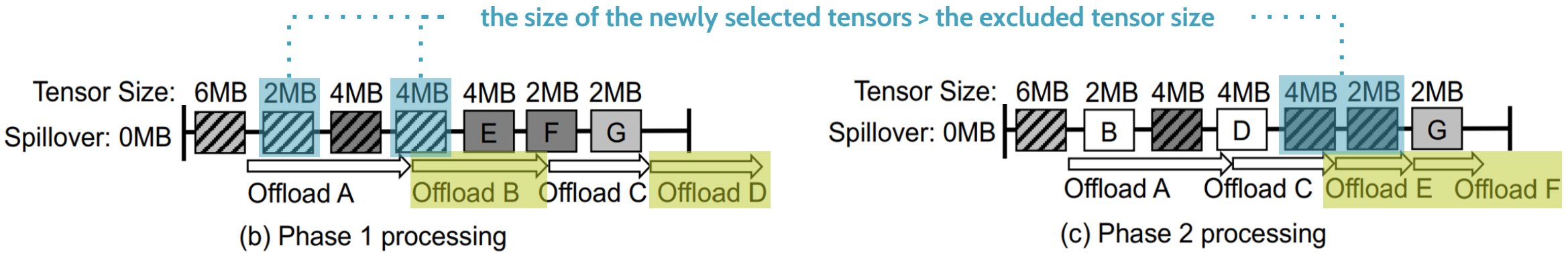
- Phase 1: Linear Tensor Selection
  - Iteratively select a certain number of tensors
    ➔ the total size of unselected tensors and memory-resident objects < the total GPU memory size
  - The scheduler checks **whether the total data transfer time is smaller than the total execution time** of all layers in the forward pass. ➔ If not, go to phase 2.



FlashNeuron Design

Offloading Scheduler

- Phase 2: Compression-aware Tensor Selection
  - Replaces the already selected tensors with **the highest compression-ratio tensors**
  - If **the total transfer time does not exceed the forward pass's total execution time**, the scheduler stops. ➔ Otherwise, it repeats this process.
    - **compression tensors have inversely proportional to the compression ratio** of the original offloading time
  - If the scheduler stops as it cannot find more compression-friendly tensors, the generated schedule is expected to incur some delay from tensor transfers.



FlashNeuron Design

Peer-to-Peer Direct Storage Access

- P2P-DSA
  - lightweight layer that **enables direct offloading/prefetching tensors** from GPUs to NVMe SSDs **without using the host DRAM buffer** to minimize host intervention.
    - **GDRCopy** is a fast GPU memory copy library based on NVIDIA GPUDirect
      - **GPUDirect** : a technology that exposes the GPU memory to be accessed directly by other PCIe peripherals. ➔ GPUDirect storage / GPUDirect RDMA
    - **Intel SPDK** exposes a block-level I/O interface directly to the user-space software.
  - To **maintain each tensor's metadata offloaded to SSDs**, P2P-DSA contains a metadata table consisting of a **long LBA value** and a **boolean value to check** the I/O completion.

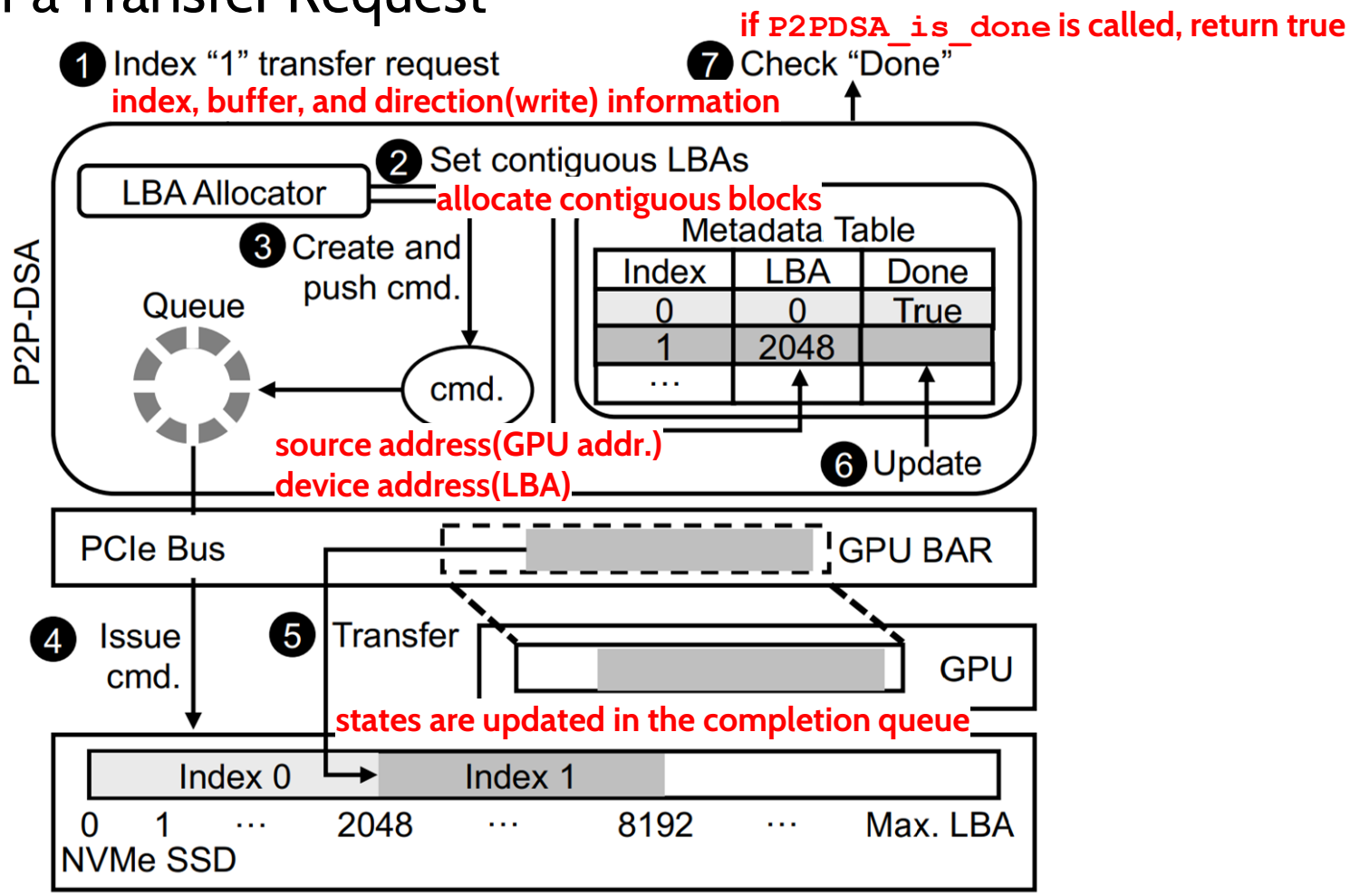
FlashNeuron Design

Peer-to-Peer Direct Storage Access

- Example Walk-through of a Transfer Request

- When `P2PDSA_issue` is called : ❶, ❷, ❸
- When `P2PDSA_update` is called : ❹, ❺
- When the `P2PDSA_update` is called once again : ❻, ❼

➔The reverse-path (prefetching data from the SSD to the GPU memory) is performed similarly.



Evaluation Methodology

- System configurations

Table 1: System configurations.

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

- Workloads

Table 2: Suite state-of-the-art DNN models and datasets used, major layer types and counts.

Network	**Dataset**	**# of layers**	**Structure**
ResNet-1922 [19]	ImageNet [12]	1922	(Conv1→BN1→ ReLU→Conv2→ BN2→ReLU)^{n}
DenseNet-1001 [22, 74]	ImageNet [12]	1001	(Conv1→BN1→ ReLU)$^{n-1}$→ Conv2→BN2→ ReLU
BERT-XLarge [13]	SQuAD 1.1 [52]	48 blocks	(Embd1→Embd2 →Embd3→FC1 →Attn→FC2 →LNorm)^{n}
HBMP [60]	SciTail [31]	24 hidden layers	FC^{m}→LSTM^{n}

The depths are increased from the original papers
➔ Use them as **proxies for future DNN models**

Performance Evaluation

- **Baseline** : uses GPU memory only ➔ **dotted line** : the best throughput in baseline
- **Arrow** : the maximum batch size for which the proposed offloading scheduler is able to find an effective schedule.
- **When the batch size is too large**, the limited bandwidth between the GPU and the SSD becomes the bottleneck
  - FlashNeuron (Memory) can utilize the nearly full PCIe write bandwidth (13.0GB/s)
  - FlashNeuron (SSD) is limited by the write throughput of the SSD device (3.0GB/s × 2)

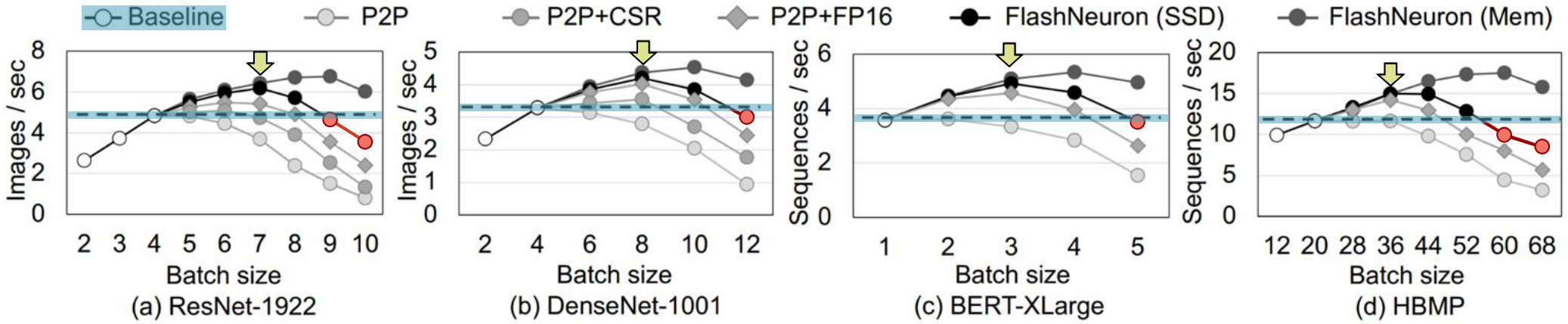
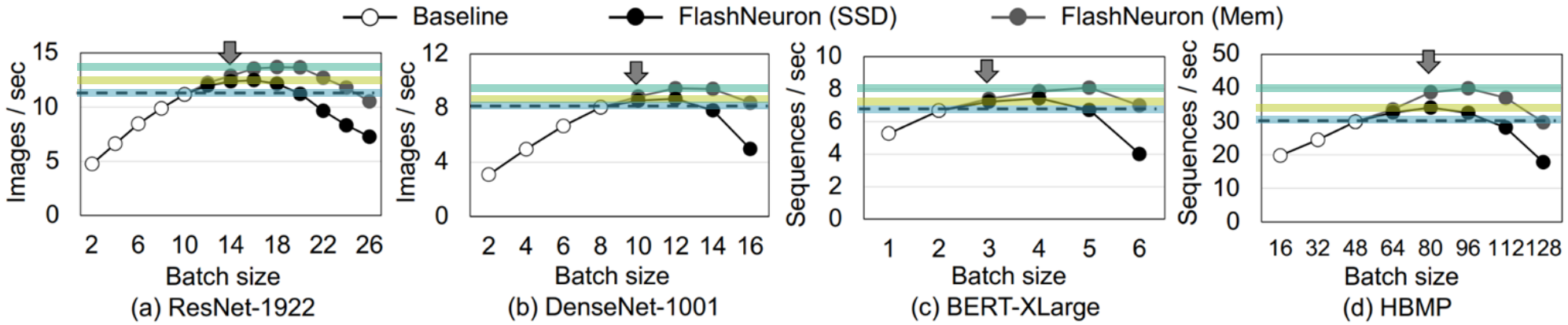


Figure 8: Throughput of FlashNeuron with varying batch sizes (P2P: Baseline with P2P, P2P+CSR: With P2P and CSR compression, P2P+FP16: With P2P and FP16 conversion). The arrow shows the maximum throughput of FlashNeuron (SSD).

➔ **The performance gap** between FlashNeuron (SSD) and FlashNeuron (Memory) can be closed by FlashNeuron (SSD) **employing additional SSDs to saturate the PCIe channel bandwidth**.

Performance Evaluation

- Half-precision Training



  - By employing larger batch sizes,
    - **FlashNeuron (SSD)** achieves 8.04% throughput improvement over baseline.
    - **FlashNeuron (Memory)** achieves 22.98% throughput improvement over baseline.

- Cost Efficiency
  - As of September 2020, DDR4 DRAM on the host CPU costs about $3.6/GB on average and NAND flash SSD about $0.102/GB. Assuming the same capacity, **FlashNeuron (SSD) achieves 35.3× higher costefficiency**.

Case Study: Co-locating Tasks on CPU

[Bandwidth-intensive tasks] data augmentation tasks on CPU while DNN training on GPU

- Throughput of DNN Training on GPU

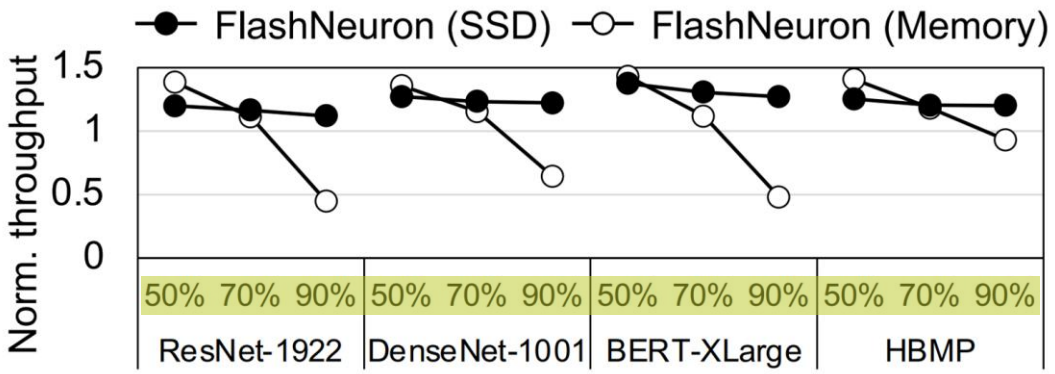


Figure 12: Normalized throughput of FlashNeuron (SSD) and FlashNeuron (Memory) when the host CPU is running a memory-intensive image transformation workload [23].

- **The CPU DRAM bandwidth** consumed by the data augmentation task

- Execution Time of Data Augmentation on CPU

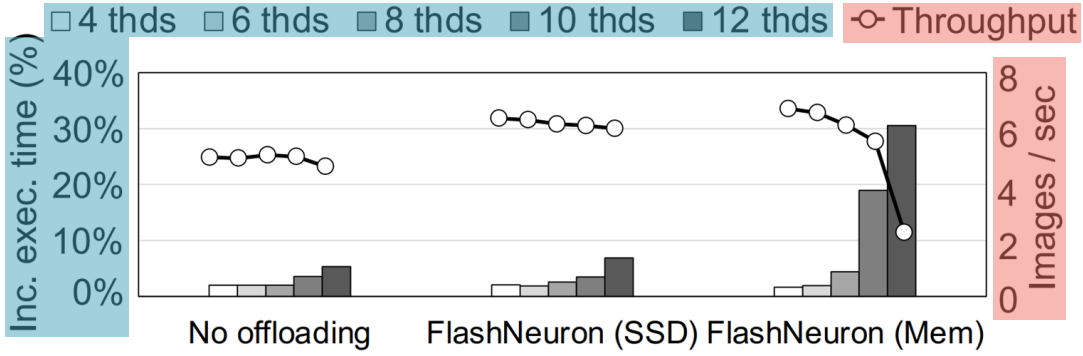


Figure 13: Increase in execution time of data augmentation tasks processing 256 2K (2048×1080) resolution images on CPU and training throughput of ResNet-1922 on GPU.

- **Bar graph** : augmentation task on CPU
  - baseline : data augmentation without GPU processes
- **Line graph** : DNN training throughput on GPU

Case Study: Co-locating Tasks on CPU

[Latency-critical tasks] DNN inference task

- Run a **BERT-as-service** on CPU: takes user-provided sentences as input and invokes BERT to return their embedding
- Concurrently running a **BERT** on GPU: training

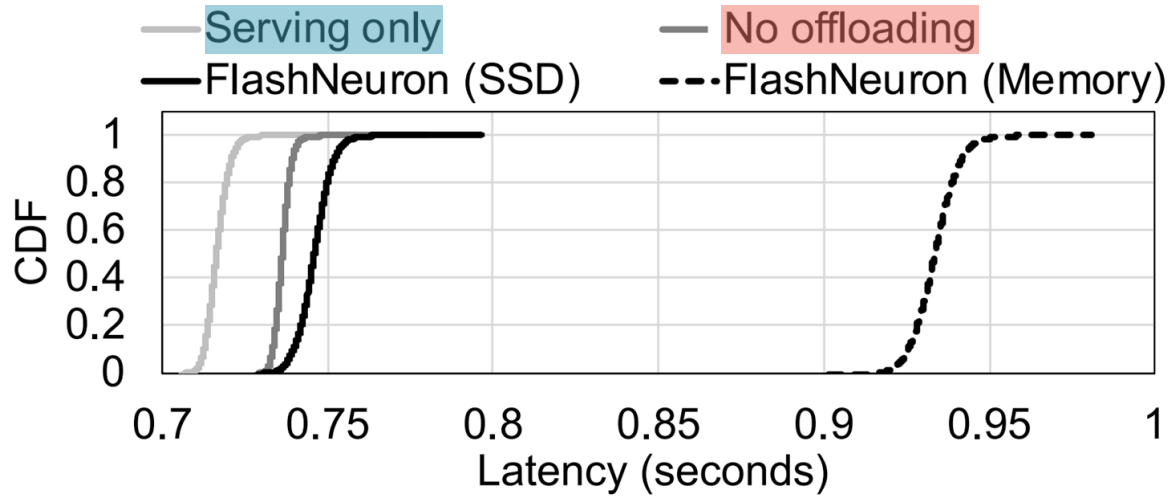


Figure 14: Query latency CDF of CPU inference across the various training scenario.

- **Serving only**: there is no process running on GPU
- **No offloading**: BERT is training but using GPU memory only

➔ **FlashNeuron(SSD)** incurs **less than 5%(*Serving only*) and 2%(*No offloading*) slowdown**.