Characterizing Deep-Learning I/O Workloads in TensorFlow

Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Pawel Herman, Erwin Laure
KTH Royal Institute of Technology
Sweden

Luis Santos
Instituto Superior Técnico, Portugal

Sai Narasimhamurthy
Seagate Systems UK, UK

EPiGRAM HS

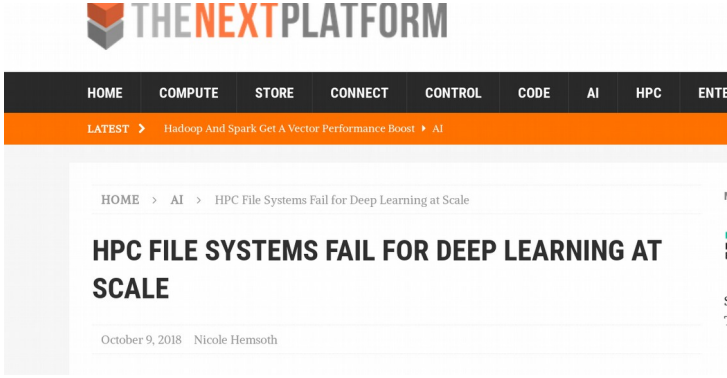
PDSW-DISC 2018

Outline

- Motivation
- Introduction to TensorFlow’s input pipeline
- Contributions
- Performance Evaluation
- Conclusion

Motivation

- Deep-Learning workloads are increasingly common on HPC systems
  - Taking advantage of high performance system for training
  - Traditional applications adopting deep-learning methods
- Deep-Learning I/O Workloads features very different characteristics comparing to traditional HPC applications
  - Small individual read/write vs collective read/write
  - Favors individual I/O
- Characterize I/O pattern being the first step for implementing improvements

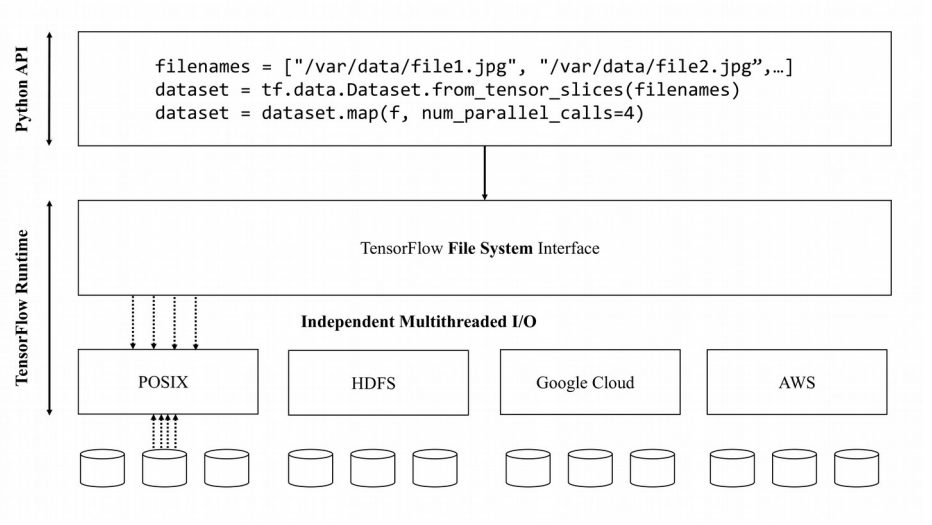


Typical HPC I/O vs Deep-Learning I/O

- HPC
  - Larger files (limited)
  - Collective I/O
    - Processes sharing the same files
  - Repetitive tasks
    - Same data input
    - e.g. iterative solvers
  - Regular write
    - Saving intermediate states / time steps
- Deep-Learning
  - Smaller files (many)
  - Individual I/O
    - Files individually loaded and used by processes
  - Repetitive tasks
    - Different data input
    - e.g. different sample batches
  - Model saved at the end of training
    - Checkpoints made regularly

TensorFlow Data Pipeline

- Dedicated input pipeline to prepare training samples for computation
  - Dataset API
- Extensive support to different I/O systems
  - POSIX
  - Hadoop Distributed File System
  - Google Cloud Storage
  - Amazon S3
- Consumer producer model
  - Network consumes training samples/batches for computation and optimization
  - Data pipeline produces samples/batches that are ready for consumption
  - ***Embarrassingly parallel problem***
    - File only used by one particular worker during training
    - Data read from file are not shared (no collective I/O needed)



TensorFlow I/O Pipeline Features

```
def parse_function(filename, label):
    image_string = tf.read_file(filename)
    image = tf.image.decode_png(image_string, channels=3)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize_images(image, [224, 224])
    return image, label
```

- DL training needs small individual I/O
  - Solution
    - *tf.dataset.map()*
      - Executes a mapped capture function, containing I/O and transformation operations
      - *num_parallel_calls* controls how many executions at the same time
      - A number of threads that is equal to *num_parallel_calls* is spawn to execute the capture function
    - *tf.dataset.interleave()*
      - Similar to map(), but expends one entry into many items to downstream operation
      - e.g. one TFRecord → many samples, one folder → samples in folder
  - Similar to how parallel I/O in MPI-IO maximizes bandwidth between workers and storage targets, but on a thread level

TensorFlow I/O Pipeline Features

- DL training on GPUs requires large number of samples continuously to fill pipeline
  - Training pipeline (consumer) consumes batches from I/O pipeline (producer)
  - On powerful platforms speed of I/O pipeline might not catch up training pipeline
  - When training pipeline triggers I/O pipeline it needs to stay idle and wait for data
  - Both pipeline are executed on different devices, presents possible parallelism

TensorFlow I/O Pipeline Features

- DL training on GPUs requires large number of samples continuously to fill pipeline
  - Solution
    - Prefetch
      - *dataset.prefetch(1)*
      - Executes input pipeline in advance → data ready for consumption as soon as computation pipeline is ready
      - Stores a number of ready for training batches in a host memory buffer
      - As soon as number of batches in buffer goes below threshold triggers I/O pipeline again
      - Exploit parallelism by utilizing CPU and GPU at the same time
    - Prefetch directly to GPU
      - *tf.contrib.data.prefetch_to_device('/gpu:0')*
      - New feature in recent release
      - Must be the last transformation applied in the pipeline
      - Further avoid copying delay between host and GPU memory by prefetching to buffer on GPU memory

Checkpoint

- Save parameters between execution to disk
  - *tf.train.Saver()*
  - Three files generated
    - Metadata: Description of the computation graph
    - Index: Describes Tensors of a graph
    - Data file: Actual data stored in variables
  - Cleanup old checkpoints: only keep the latest copies

Checkpoint

- Checkpoint I/O traffic (and I/O from movement of training data) can be bursty
  - Each checkpoint can take several hundreds of Megabytes
  - TensorFlow checkpoint saver currently does not ensure data flushed to disk and does not support Async checkpont
  - Burst-buffer
    - Usually a persistent while fast storage medium
    - Commonly implemented with Non-volatile memory
    - Acts as an intermediary between mediums with different speed and size tradeoff
    - Absorbs bursty traffic to avoid delay in application execution
    - e.g. DataWarp by Cray and IME by DNN

Checkpoint

- Checkpoint I/O traffic (and I/O from movement of training data) can be bursty
  - Solution
    - Use a burst-buffer to absorb traffic
    - On Linux calls *syncfs()* to force OS to write files to disk
    - Issue a copy command as a sub-process
      - This time let OS and file system decide which to perform disk write
      - Ensure one copy is saved

Contributions

1) Show that Threading is an effective way of increasing bandwidth utilization
   - Through a STREAM like benchmark
2) Prefetch is key to high performance and efficient use of devices on machine
   - Through AlexNet miniapp
3) Burst buffer is essential for maintaining high performance pipeline
   - Quick checkpointing without delaying next training iteration
   - Data staging on burst buffer for fast ingestion *(not covered by this work)*

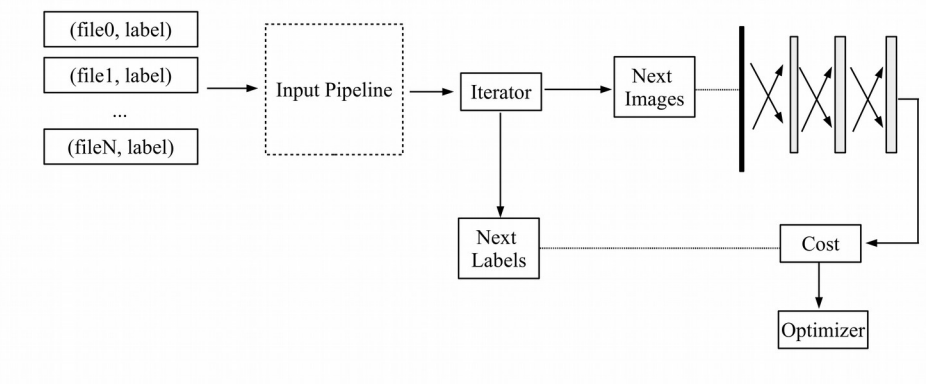
STREAM Benchmark

1) Read a list of file paths and labels
2) Shuffle list
3) Apply capturing function for processing
    1) Individual file I/O
    2) Decode image
    3) Resize
4) Batch
5) Attach iterator
6) Iterator continuously invoked
7) Create a stream of inflow

- Compute images per second
- Compute MB/s

AlexNet Mini-app

- Input preprocessing of images
  - File I/O
    - Read a list of files and labels
    - tf.read()
  - Image decoding
    - tf.image.decode_png()
    - The function also decodes JPEG files
  - Image resize to size 244x244
    - tf.image.resize_images()
- Apply batching, prefetch and attach iterator
- Invoke optimize, draw batch, update



AlexNet Mini-app with Checkpoint

- Extends AlexNet mini-app with checkpointing
  - Snapshots taken every defined number of iterations
    - Calls tf.train.Saver() to create checkpoint files, use syncfs() to ensure checkpoint is flushed to disk where files are stored
      - File systems such as ext4 saves files in memory and writes data to disk when operating system see fit
  - Evaluate performance when checkpointing to different storage devices
- Proof of concept burst buffer

  1) Perform checkpoint routines and use NVMe as storage with Intel Optane
     - Save snapshots
     - Sync to disk
  2) Issue copy command to copy newly created file to slow storage in background
  3) Checkpoint safely stored in NVMe storage while being swap to permanent storage in background
     - Training continues

Evaluation

- Blackdog
  - Eight core Intel Xeon E5-2609v2
  - NVIDIA Quadro K4000
  - 72 GB DRAM
  - 4TB HDD (non-RAID)
  - 250 GB SSD
  - 480GB NVMe
  - Ubuntu Server 16.04
    - Gcc 7.3.0
    - CUDA 9.2
    - TensorFlow 1.10

- Tegner
  - Intel E5-2690 v3 Haswell
  - NVIDIA K80
  - 512 GB RAM
  - Lustre parallel file system
  - CentOS 7.4
    - Gcc 6.2.0
    - CUDA 9.1
    - TensorFlow 1.10

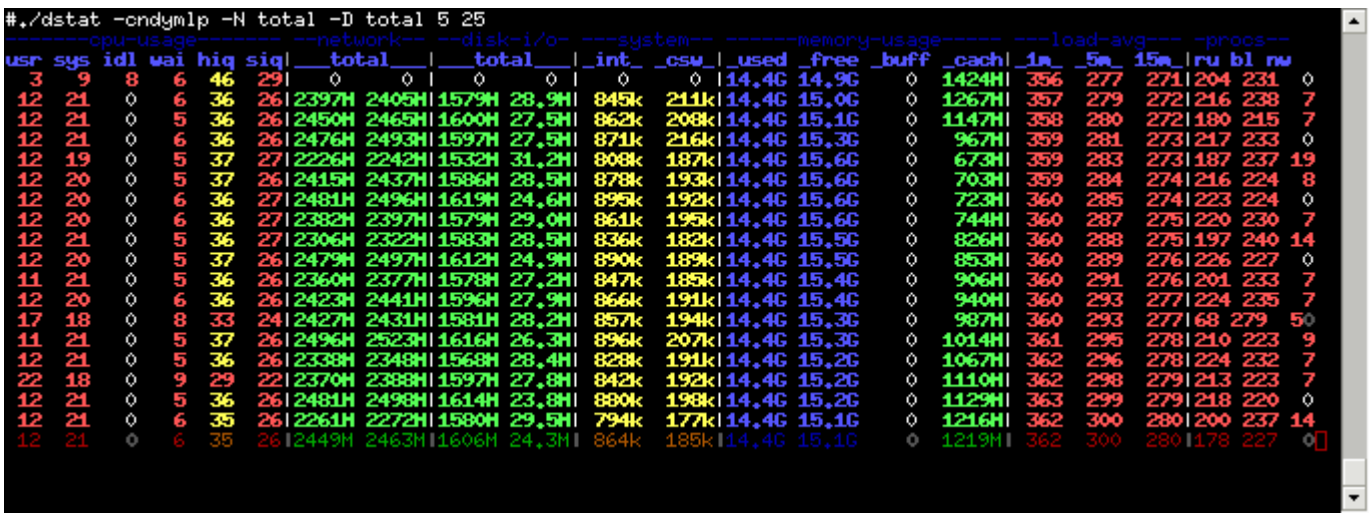
Storage Devices

- Hard Disk Drive (HDD)
  - 4 TB (non RAID)
  - IOR Read 163 MB/s, Write 133.14 MB/s
- Solid State Drive (SSD)
  - Samsung 850 EVO 250 GB
  - IOR Read 280.55 MB/s, Write 195.05 MB/s
- Intel Optane (Opt.)
  - Intel Optane 900p 480GB on PCI-E
  - IOR Read 1603.06 MB/s, 511.78 MB/s
- Lustre
  - Parallel file system used by Tegner
  - IOR Read 1968.618 MB/s, 991.914 MB/s

- Operating system often cashes recent files
  - Passes POSIX FADV DONTNEED to posix_advice() for files
  - # echo 1 > /proc/sys/vm/drop caches
    - Only possible on Blackdog where we have root permission
  - Only reads new file during a test, never read previous accessed files

Evaluation

- Monitor system I/O activities with *dstat*
  - A system resources monitoring tool which produces different statistics
  - Sampled every second
  - Able to track different disk activity



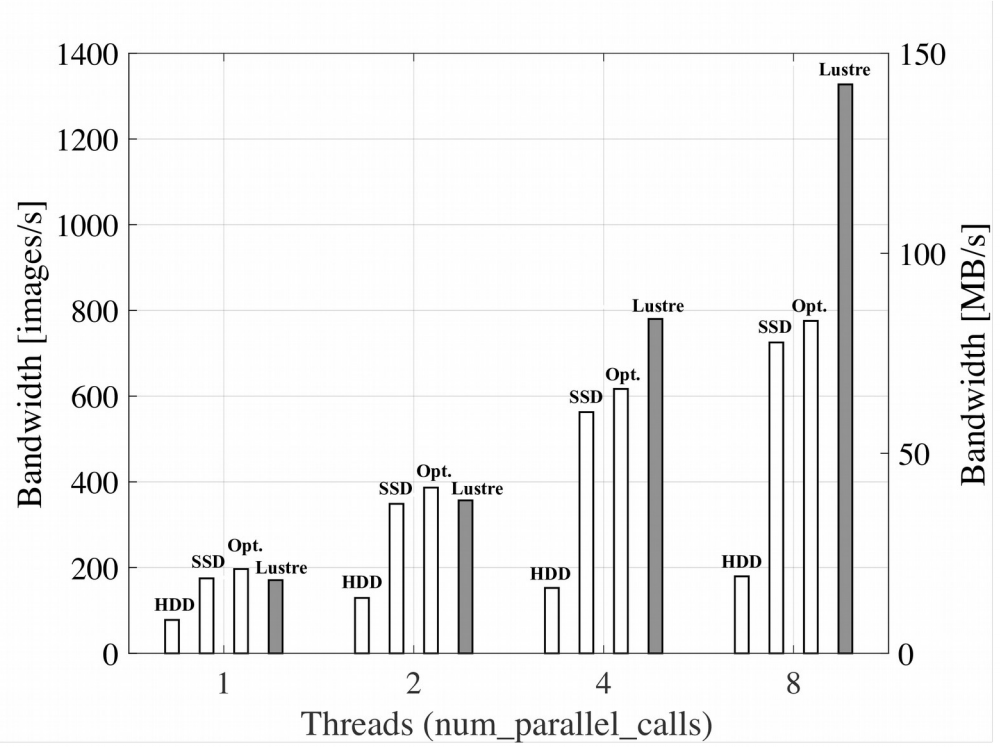
http://dag.wiee.rs/home-made/dstat/

Evaluation

- Micro-benchmark
  - Reads subset of ImageNet with 16,384 JPEG files with median size 112 KB
  - Mainly reports batch size 64
    - Iterator invoked 256 times per test to consume the whole dataset
  - Vary number of threads for individual I/O to one, two, four and eight
  - Tests reading performance when files are placed on:
    - HDD
    - SSD
    - Intel Optane
  - One warm-up run, repeat tests five times
    - Reports median bandwidth
    - MB/s
    - Images/s

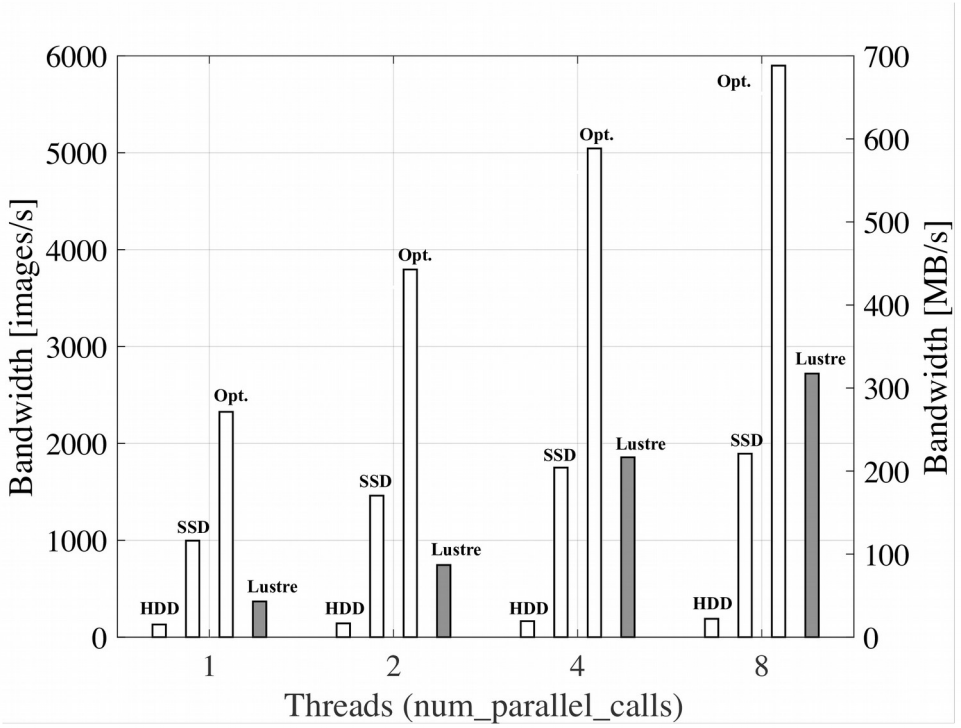
Evaluation

- Micro-benchmark
  - Double bandwidth when increases threads from one to two
  - Benefit for HDD diminishes when number of threads exceed four
    - 2.3x improvement with eight threads
  - Best bandwidth utilization by Lustre
    - True parallel read from different object storage targets
    - 7.8x improvement with eight threads
  - Poor bandwidth comparing to our IOR benchmark results



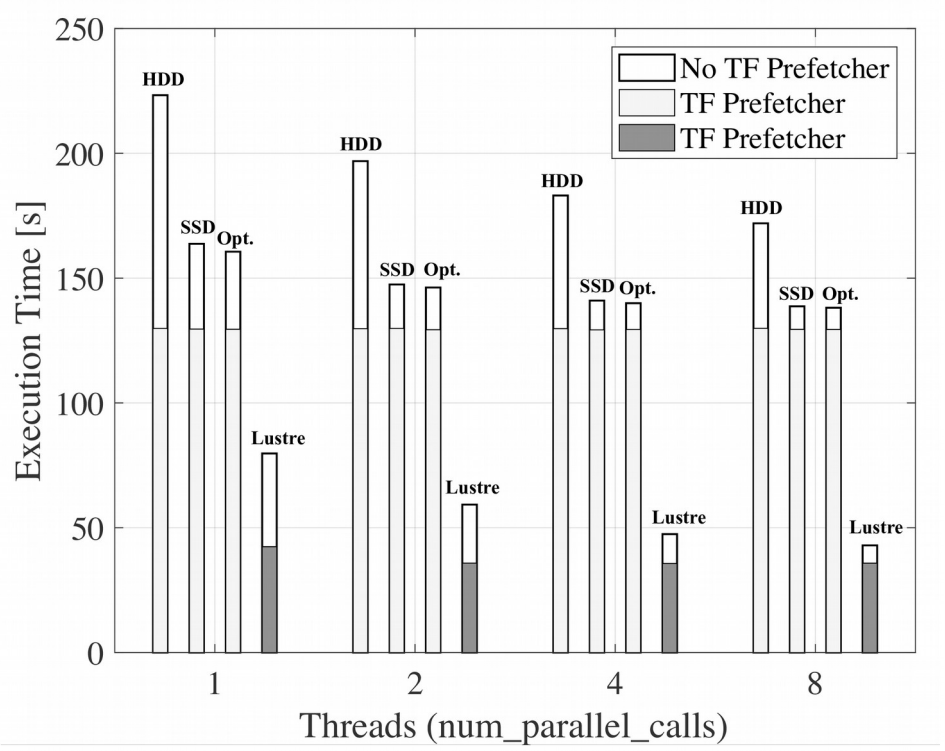
Evaluation

- Micro-benchmark
  - Empty input process except read
  - Optane achieves best bandwidth as expected



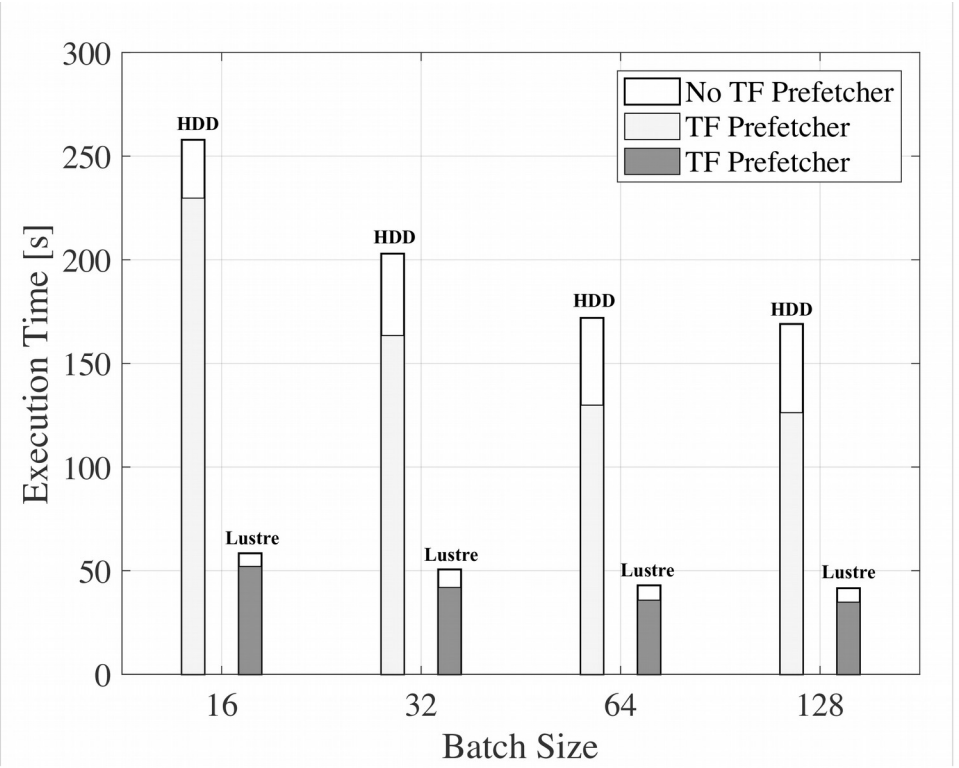
Evaluation

- AlexNet mini-app
  - Caltech 101 dataset
    - Median image size 12 KB
    - Executes 142 steps, batch size 64 and consume 9,088 images
    - One epoch per test
  - Varying number of threads
    - Effective on performance
    - Close to no effect to the SSD and Optane
  - Prefetching is very effective
    - Runtime becomes the same regardless of storage technology and number of threads used



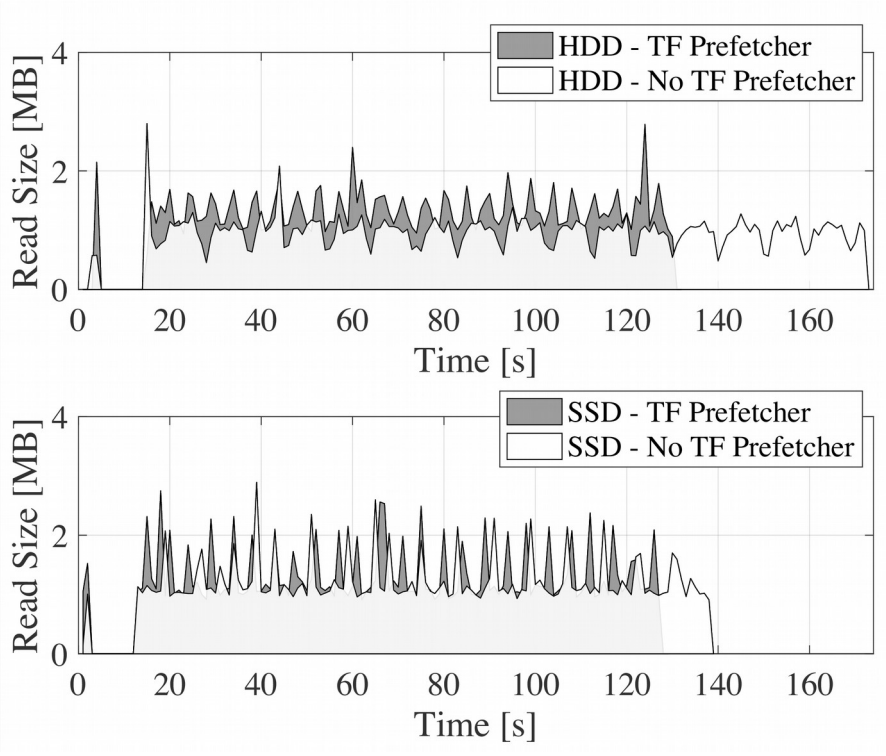
Evaluation

- AlexNet mini-app
  - Increased batch size enables better utilization



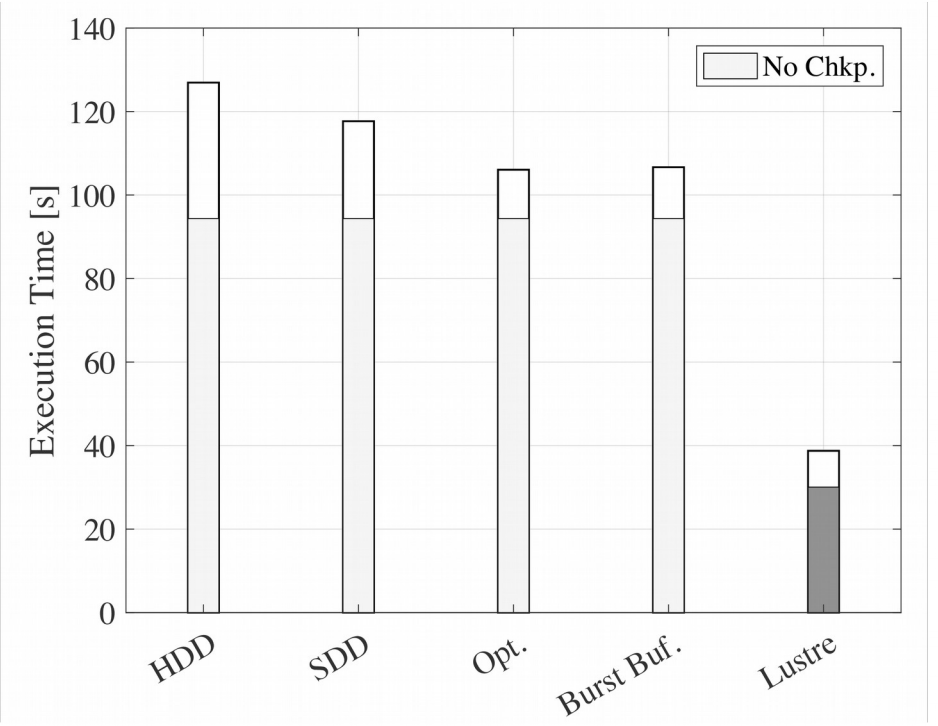
Evaluation

- AlexNet mini-app
  - Prefetching results in complete overlap of I/O and computation
  - I/O pipeline executed while computation of current batch is on going
  - Higher reading rate when prefetch enabled
  - Clear pattern of batch reading visible when prefetch not enabled
  - Initial idle period as fixed cost from initialization and shuffling of sample list



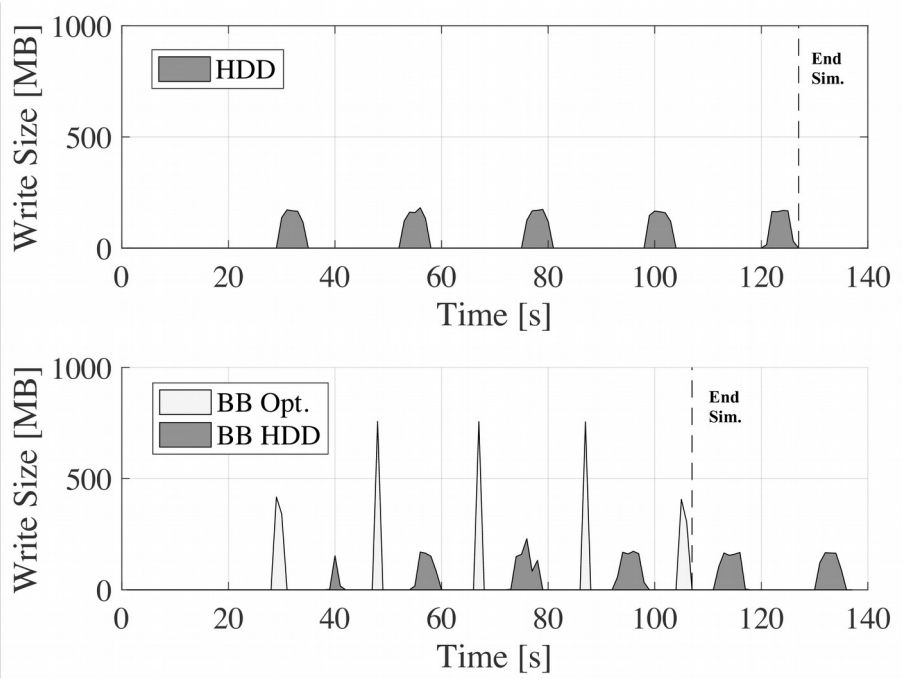
Evaluation

- Checkpoint and burst buffer
  - Execute 100 iterations, checkpoint every 20 iterations
    - Batch size 64
    - Training samples stored in SSD
    - Prefetch enabled
  - Each checkpoint contains ~600 MB
  - Slowest checkpoint to HDD
  - Lustre has best performance
    - Expected result
  - Checkpointing account to ~15% of execution time
  - Prototype burst buffer has similar performance comparing to only writing to Intel Optane
    - 2.6x improvement comparing to checkpointing directly to HDD



Evaluation

- Checkpoint and burst buffer
  - Five checkpoints made each test
  - Long duration when checkpoints are written and sync to HDD
  - Optane effectively absorbed all the burst from writing checkpoints
  - Files are moved to HDD in background for long term storage while training continues



Conclusion

- Performance of writing is a traditional I/O bottleneck
  - Not anymore with DL workload!
  - DL workload are small-read intensive, I/O system needs to optimize accordingly
- Traditional method of maximizing bandwidth by threading still applies
- Prefetching is key to pipeline performance optimization
  - Prefetching at different level of storage hierarchy will likely become a requirement
    - e.g. prefetching/staging of training samples in burst-buffer
- Using Burst-buffer is an effective way of absorbing or handling burst of I/O traffic

Funding for the work is received from the European Commission H2020 program
Grant Agreement No. 801039 (https://epigram-hs.eu/)