PyTorch Model Performance Analysis and Optimization – Part 2

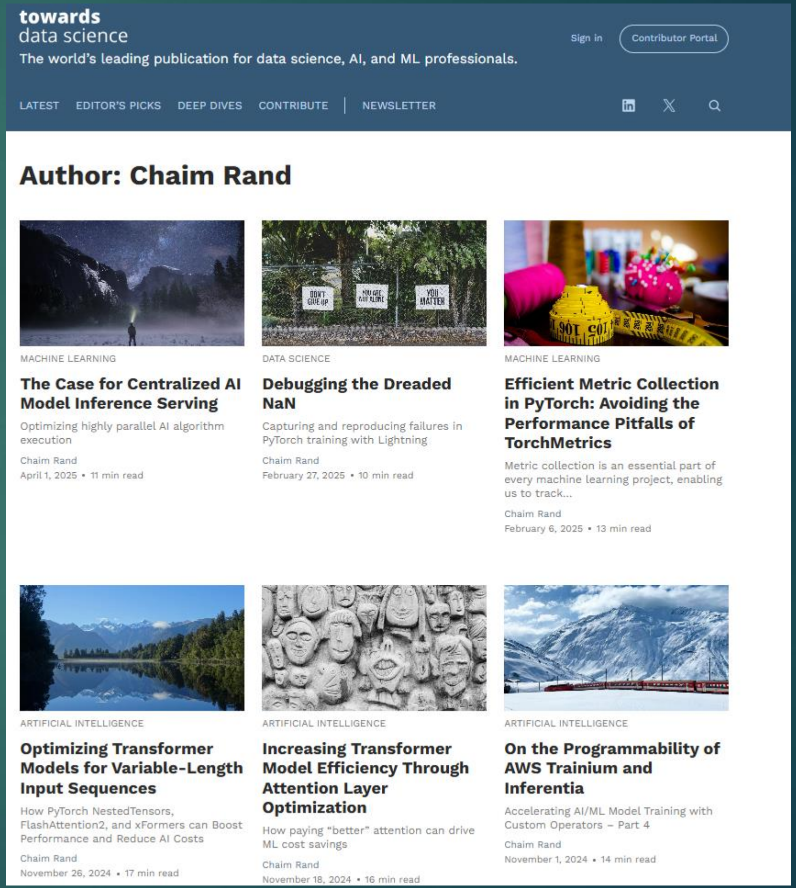
Bottlenecks on the Data Input Pipeline

Chaim Rand and Yitzhak Levy

May 2025

Chaim Rand

- AI/ML/CV Algorithm Developer
- Areas of interest
  - Cloud Based AI/ML
  - AI/ML Model Performance Optimization
- Blogging Hobbyist
  - https://towardsdatascience.com/author/chaimrand/
  - https://chaimrand.medium.com/
- Thanks to Yitzhak Levy for help with preparation



Agenda

- Brief Recap
- Bottlenecks on the Data Input Pipeline
  - Common Causes
- Discovery Through Data Caching
- Tips, Tricks, and Techniques (TTTs)

RECAP - Motivation

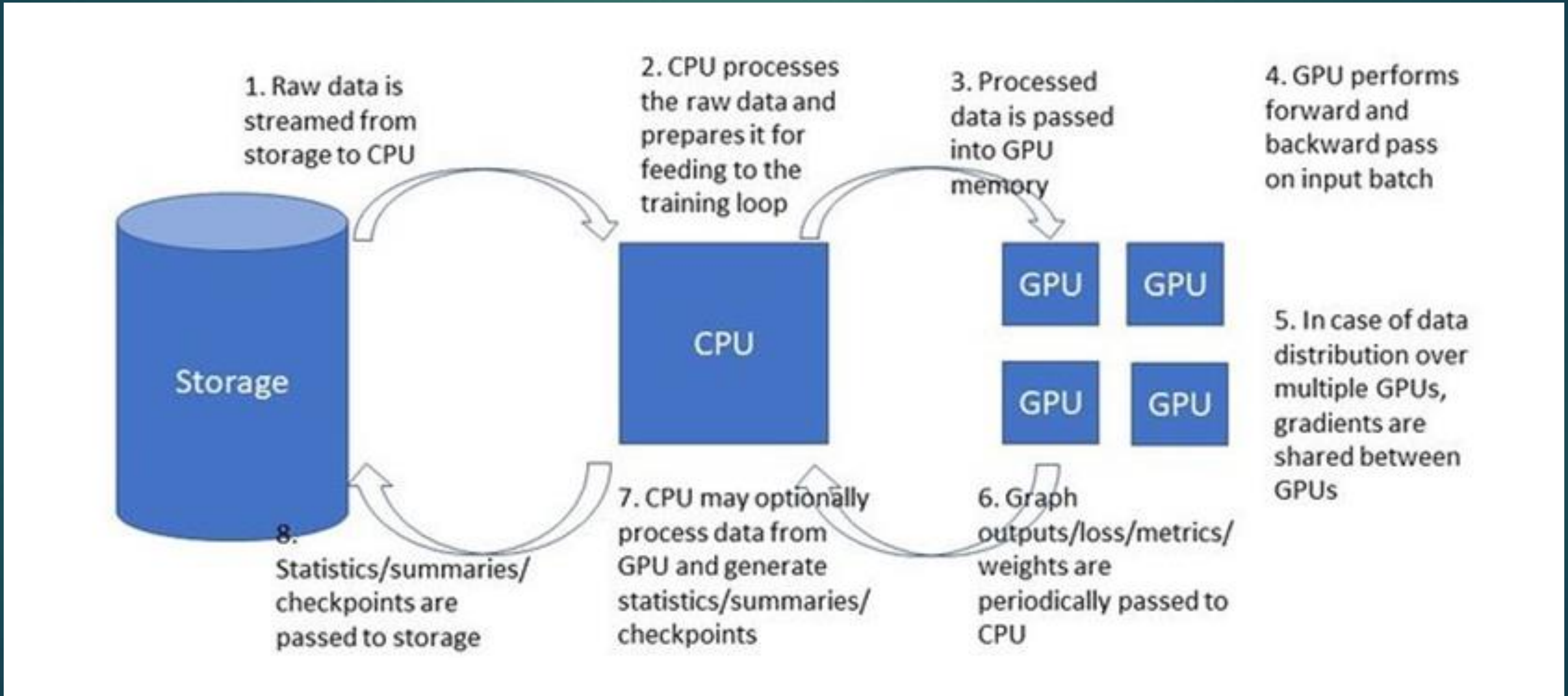
- AI models are resource intensive and expensive to train/run
  - E.g. Amazon EC2 P5 Instance (8 H100) is ~$100 per hour
- ML workloads are prone to performance bottlenecks
- Simple optimization techniques can deliver significant acceleration and cost savings

Key Messages:

→ **AI/ML developers must take responsibility for the runtime performance of their workloads**

→ **You don't need to be a CUDA expert to see results**

RECAP - Training Pipeline



RECAP - Optimization Methodology

- Objective - Maximize throughput (samples per second)
- Use performance profilers to measure resource utilization and identify bottlenecks
- → **Integrate into model development lifecycle**

- **Profile**

  identify bottlenecks in the pipeline and under-utilized resources

- **Optimize**

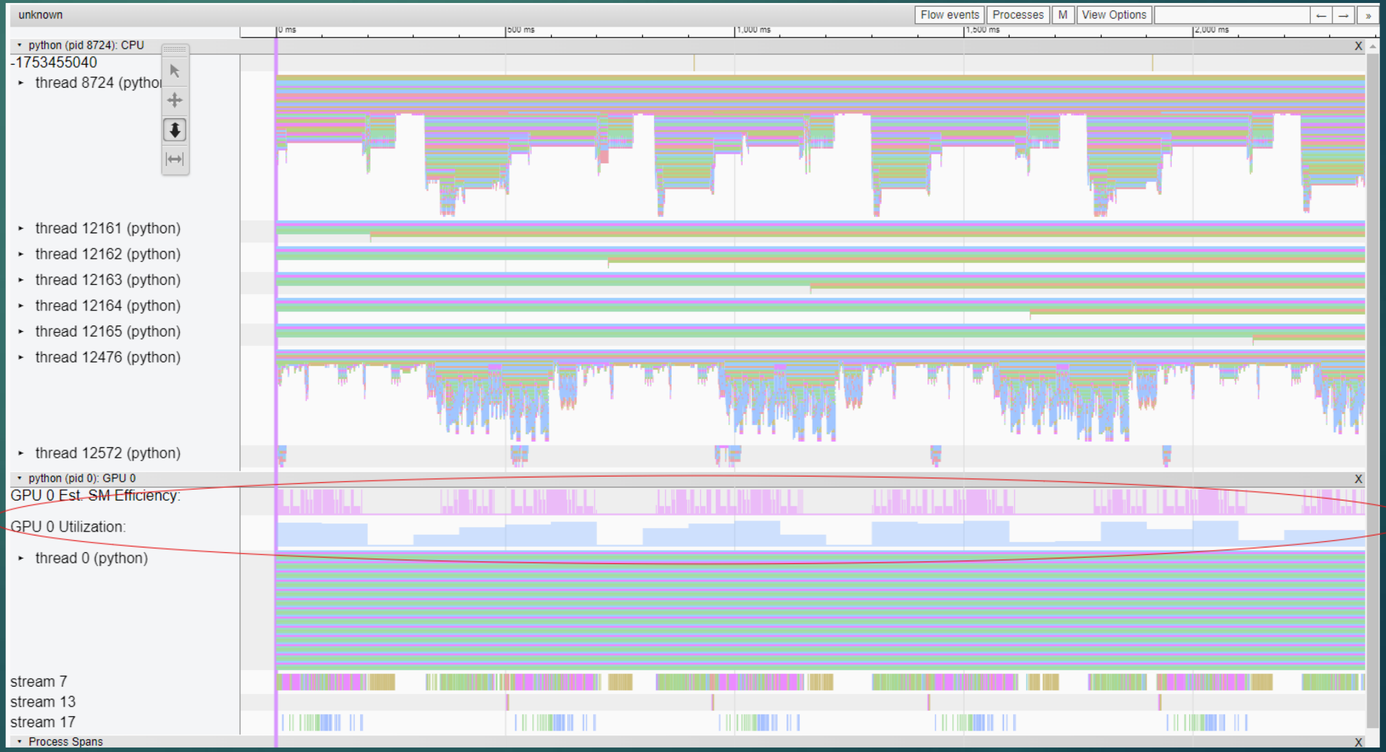
  address bottlenecks and increase resource utilization

- **Repeat**

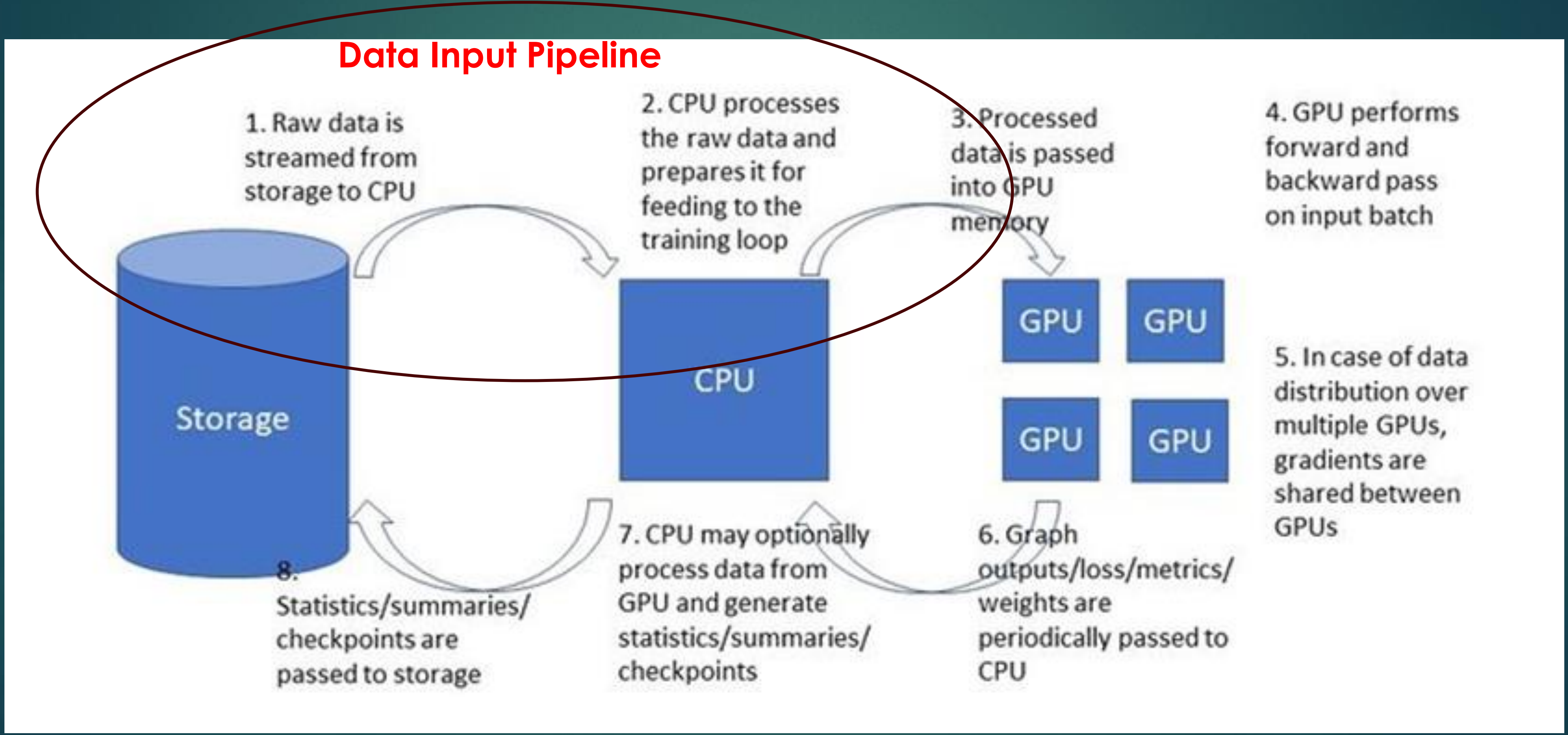
  until satisfied with the throughput and resource utilization

RECAP - PyTorch Profiler

- Relatively easy to use
- View results with TensorBoard (recently deprecated), Perfetto or Chrome



Data Input Pipeline



Bottlenecks on the Data Input Pipeline

CPU–GPU Division of Labor

<table>
<tr><th>Host</th><th>Device</th></tr>
<tr><td>

Load data samples (from disk/NFS/cloud storage)
Preprocesses data
Collate data into data batches
Copy data batches onto GPU
A bunch of other things:

GPU kernel loading
Logs metrics/training progress
Etc.

</td><td>

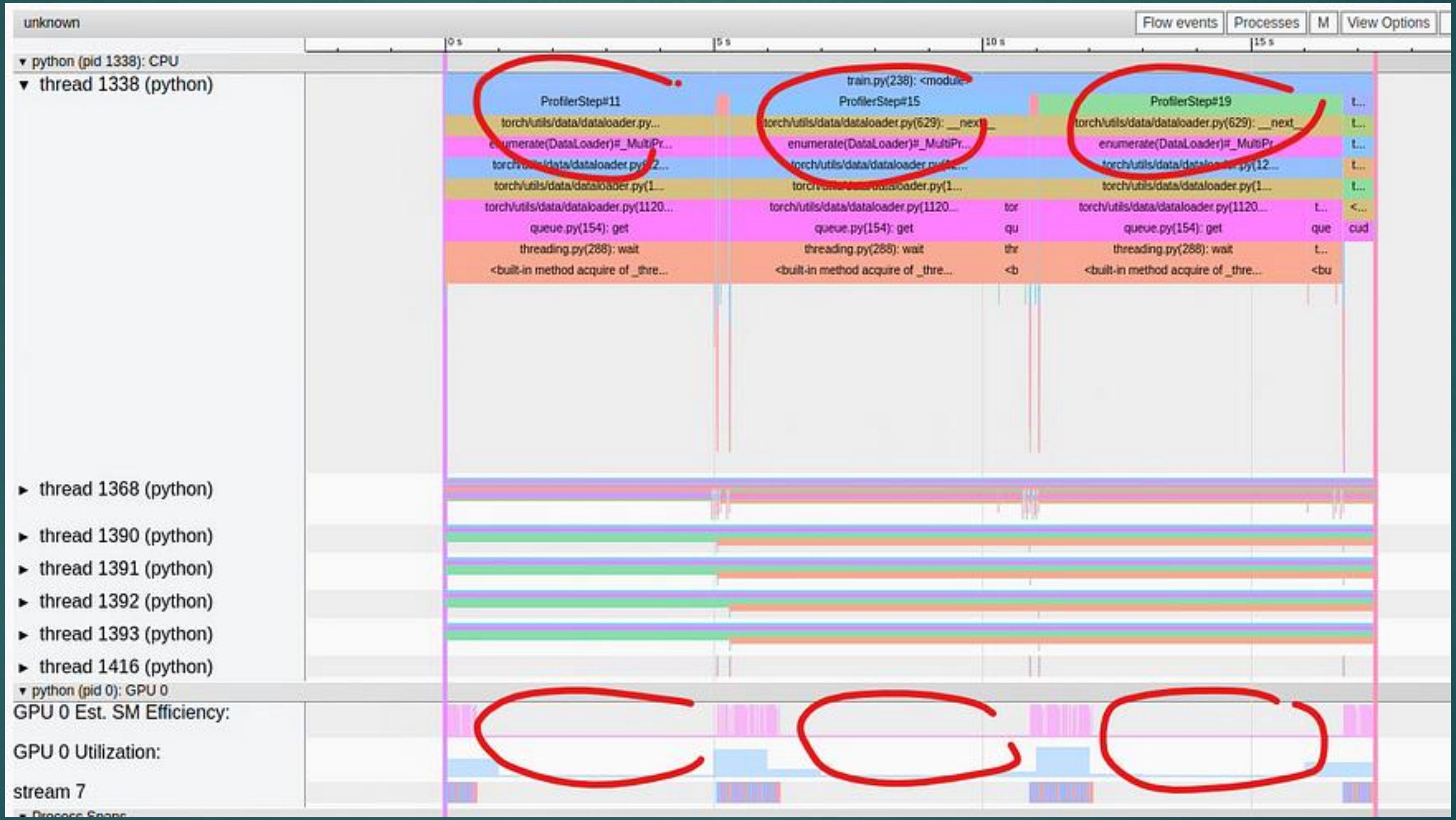
Run the model on data input from CPU (forward pass)
During training – backward propagation, update weights, gradient sharing/GPU2GPU communication

</td></tr>
</table>

GPU Starvation

- GPU starvation occurs when the GPU is idle while it waits for the CPU to feed it input data
- Highly undesired due to high cost of GPU

GPU Starvation Profile Trace



Common Causes

- Raw data retrieval is slow
- Data pre-processing is compute-intensive (e.g., augmentations)
- Bottleneck on host to device copy

Profiling Strategies

- PyTorch Profiler
  - Complicated by multi-worker dataloader
  - See solving bottlenecks on the data input pipeline with pytorch profiler
- Other CPU profilers
- Caching on the data input pipeline

Identifying Bottlenecks Using Data Caching

Data Caching Strategy

- Step 1: Cache a data batch on the GPU and iterate over single batch to estimate upper bound of throughput
- Step 2: Cache a data batch on the CPU and compare throughput to previous result.
  - If lower, check for bottleneck on host to device data copy
- Step 3…: Cache at different points up the data input pipeline and Identify the stage where the throughput drops
  - Before/After collation
  - Before/After augmentations
  - After raw data loading
  - Etc.

Data Caching Examples

```
def cache_batch_on_device():
    # Object creation is same as the last function

    # Move the batch to the device,
    # iterate over simple range to estimate optimum training throughput of the model
    batch = next(iter(dataloader))
    batch = batch.to(device)
    # Optimization loop
    for _ in range(100):
        optimizer.zero_grad()
        output = model(batch)
        loss_value = loss(output, batch)
        loss_value.backward()
        optimizer.step()
        profiler.step()
```

```
def cache_before_moving_to_device():
    # Object creation is same as the last function

    # Keep the batch on the host
    # iterate over simple range to estimate optimum training throughput of the model
    host_batch = next(iter(dataloader))
    # Optimization loop
    for _ in range(100):
        batch = host_batch.to(device)
        optimizer.zero_grad()
        output = model(batch)
        loss_value = loss(output, batch)
        loss_value.backward()
        optimizer.step()
        profiler.step()
```

Optimization Tips, Tricks, and Techniques (TTTs)

Optimization Tips, Tricks, and Techniques

- Optimizing raw data retrieval
- Optimizing data processing
- Optimizing host to device data copy

TTTs – Optimizing Data Storage and Format

- Choose an instance type with appropriate network bandwidth
- Optimize data storage location
  - While also taking cost and flexibility into consideration
- Maximize storage network out capacity
  - E.g., Consider S3 data partitions to reduce throttling
- Consider compressing data in storage to reduce network payload
  - Although decompression may increase CPU utilization
- Consider grouping samples into larger files – to reduce overhead of file retrieval
- Use optimized data retrieval utilities (e.g., s5cmd for S3 object files)
- Tune file retrieval configurations (e.g., chunk size)

TTTs – Addressing Data Processing Bottlenecks

- Tune the number of data loading workers and prefetch factor
- Whenever possible, move data-proc to data preparation phase
- Choose an instance type with better CPU/GPU compute ratio
- Optimize order of operators
  - E.g., crop then blur rather than blur and then crop
- Use optimized Python libraries/utilities (Jax/Numba JIT)
- Create custom PyTorch CPU operators
  - E.g., Only read the desired crop from file
- Consider adding auxiliary CPUs (data servers) – (e.g., Ray Data)
- Prepend some of the “GPU-friendly” data-processing to GPU compute graph
- Tune OS level configurations (thread management, memory allocation, etc.)

TTTs – For minimizing Payload of host to device data copy

- Postpone int8 to float32 datatype conversions to the GPU - reduces memory copy by a factor of 4.
- If your model is using lower precision floats (e.g., fp16/bfloat16) cast the floats on the CPU to reduce payload by half.
- Postpone unpacking one-hot vectors to the GPU – i.e., keep them as label ids until the last possible moment.
- If you have many binary values, consider using bitmasks to compress the payload. E.g., if you have 8 binary maps, consider compressing them into a single uint8.
- If your input data is sparse, consider PyTorch sparse data representations.
- While zero-padding is a popular technique for dealing with variable sized input samples, it can significantly increase the size of the memory copy. Consider alternative options (e.g., see here).
- Make sure you are not copying data that you do not actually need on the GPU!!

Key Takeaways

- AI/ML developers must take responsibility for the runtime performance of their models
- Integrate profiling analysis and optimization into AI/ML development workflow
  - Use data-caching to identify bottlenecks in data input pipeline
- You do not need to be a CUDA/optimization expert to boost runtime performance and reduce AI/ML costs