PyTorch Model Performance Analysis and Optimization – Part 3

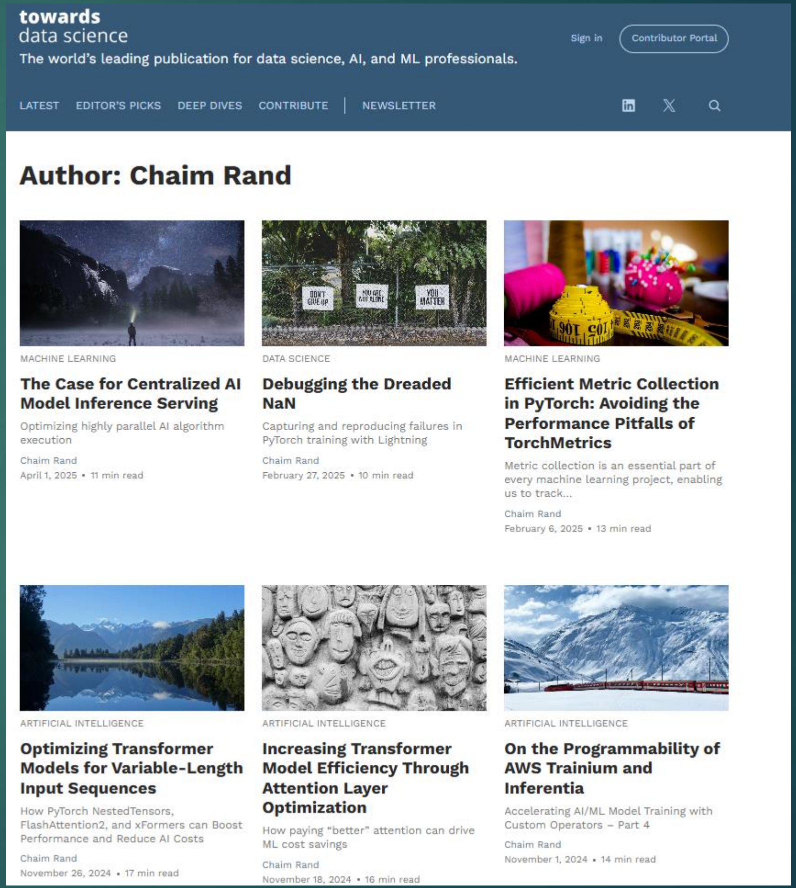
Optimizing Data Transfer With NVIDIA Nsight™ Systems Profiler

Chaim Rand

Jan 2026

Chaim Rand

- AI/ML/CV Algorithm Developer
- Areas of interest
  - Cloud Based AI/ML
  - AI/ML Model Performance Optimization
- Not a CUDA expert
- Blogging Hobbyist
  - https://towardsdatascience.com/author/chaimrand/
  - https://chaimrand.medium.com/



Agenda

- Brief Recap
- Nvidia Nsight Systems Profiler (nsys)
- Data Transfer Bottlenecks
- Case Study 1: CPU-to-GPU
- Case Study 2: GPU-to-CPU
- Case Study 3: GPU-to-GPU

RECAP - Motivation

- AI models are resource intensive and expensive to train/run
- 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 - 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

NVIDIA Nsight Systems (nsys)

- A **system-wide** performance profiler that captures a unified timeline of: CPU threads, CUDA kernels, memory copies, NCCL communication, OS runtime, etc.

Aspect	PyTorch Framework Profiler	NVIDIA Nsight Systems (nsys)
Installation	Bundled with PyTorch	Requires separate installation or dedicated Docker image
Ease of use	Easy (Python API)	More complex (CLI + GUI workflow)
Visibility	PyTorch ops, autograd, and Python control flow	System-level view (OS runtime, CUDA, NCCL drivers)
GPU profiling	Operator-level GPU timing	Fine-grained kernel, stream, and overlap analysis
Call stack	Python / PyTorch operator call stack	CUDA and driver-level call stack
Best for	Model-level and operator optimization	End-to-end performance and scalability debugging

Data Transfer Bottlenecks

- AI/ML workloads involve constant movement of data
  - **CPU → GPU**: Data batches copied from the CPU to the GPU for training/inference
  - **GPU → CPU**: Model outputs (predictions) copied from the GPU to the CPU for storage or delivery to client
  - **GPU → GPU**: Gradients, weights, activations shared in distributed training
- Highly prone to performance bottlenecks
  - GPU lays idle while it waits for data transfer to complete
- Goal: Identify and solve common bottlenecks using nsys profiler

1. Optimizing CPU-to-GPU Data Transfer in AI/ML Training Workloads

A Toy Resnet Model

- Experiment: Train a ResNet-18 image classification model
- Use default settings of PyTorch DataLoader
- Annotate code blocks using NVIDIA Tools Extension (NVTX)
- Run on Amazon EC2 g6e.2xlarge (NVIDIA L40S GPU):

  nsys profile \

    --capture-range=cudaProfilerApi \

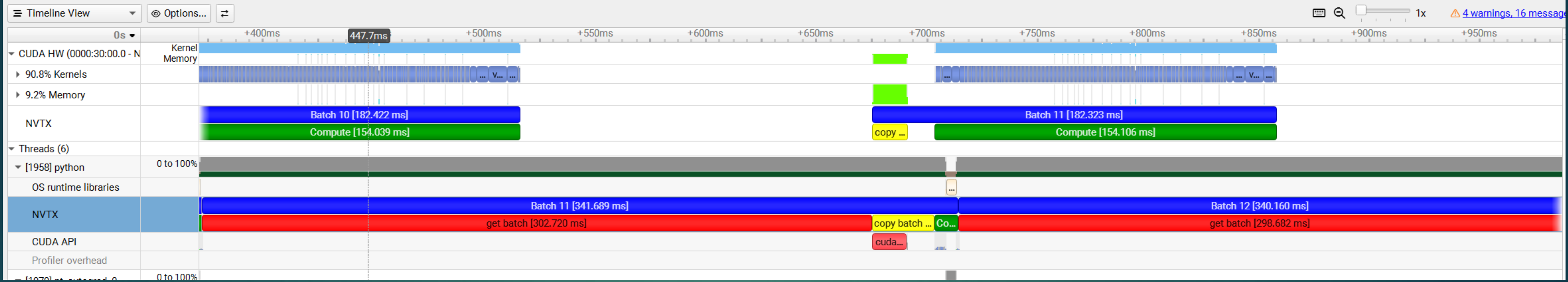
    --trace=cuda,**nvtx**,osrt \

    --output=baseline \

    python train.py
- Copy resultant nsys-rep file to development station for analysis

```python
DEVICE = "cuda"
BATCH_SIZE = 64
IMG_SIZE = 512

def copy_data(batch):
    data, targets = batch
    data_gpu = data.to(DEVICE)
    targets_gpu = targets.to(DEVICE)
    return data_gpu, targets_gpu

train_loader = DataLoader(
    FakeDataset(),
    batch_size=BATCH_SIZE
)
data_iter = iter(train_loader)

for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        with nvtx.annotate("get batch", color="red"):
            batch = next(data_iter)
        with nvtx.annotate("copy batch", color="yellow"):
            batch = copy_data(batch)
        with nvtx.annotate("Compute", color="green"):
            compute_step(model, batch, optimizer)
```

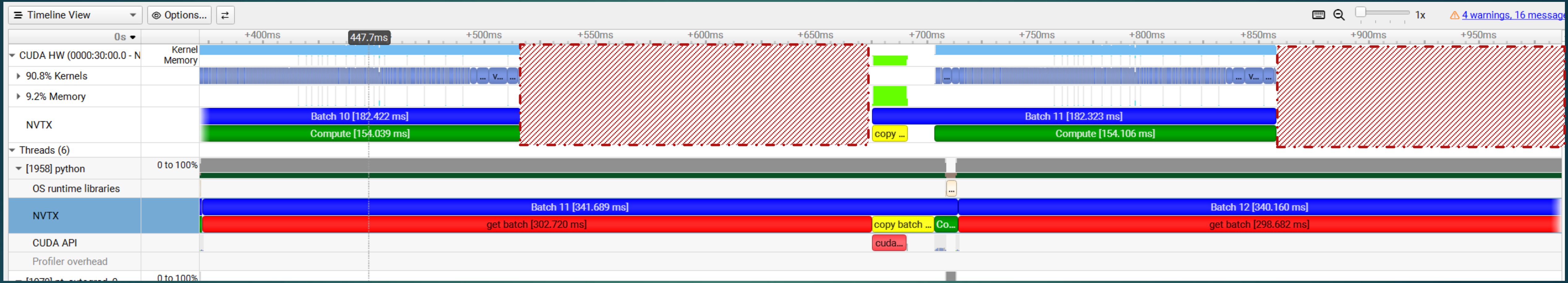
Nsight Systems Profiler Timeline



- Timeline divided into CUDA and CPU sections, each with NVTX section with the colored annotations
- CUDA section distinguishes between the GPU kernel (compute) activity (90.9%) and memory activity (9.1%)

```python
for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        with nvtx.annotate("get batch", color="red"):
            batch = next(data_iter)
        with nvtx.annotate("copy batch", color="yellow"):
            batch = copy_data(batch)
        with nvtx.annotate("Compute", color="green"):
            compute_step(model, batch, optimizer)
```

Nsight Systems Profiler Timeline



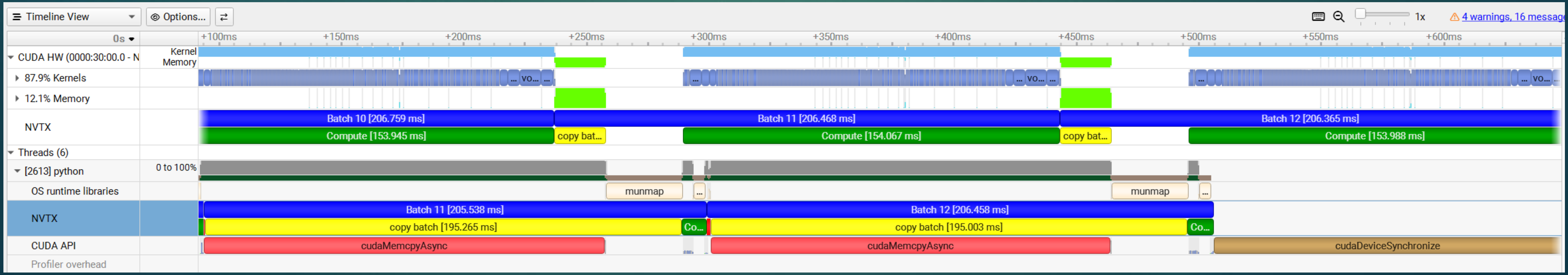
- The GPU is idle for roughly 50% of each training step
- GPU activity for each batch starts immediately after the “get batch” activity has completed on the CPU
  - First a host-to-device memory copy (light green) and then the kernel computations (light blue)
- While GPU process batch N, CPU prepares batch N+1 — leading to a partial overlap of batch N+1 on the CPU with batch N on the GPU.
- The clear bottleneck is in the “get batch” (red) activity - the Dataloader
- **By default, PyTorch performs single process data loading**

```python
for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        with nvtx.annotate("get batch", color="red"):
            batch = next(data_iter)
        with nvtx.annotate("copy batch", color="yellow"):
            batch = copy_data(batch)
        with nvtx.annotate("Compute", color="green"):
            compute_step(model, batch, optimizer)
```

Optimization 1: Multi-Process Data Loading

- Update DataLoader to use multiple workers
- Data batches are prepared in dedicated background processes
- But still a ton of idle time – kernel loading is **blocked** by "copy batch" (in yellow) – more specifically by "munmap"

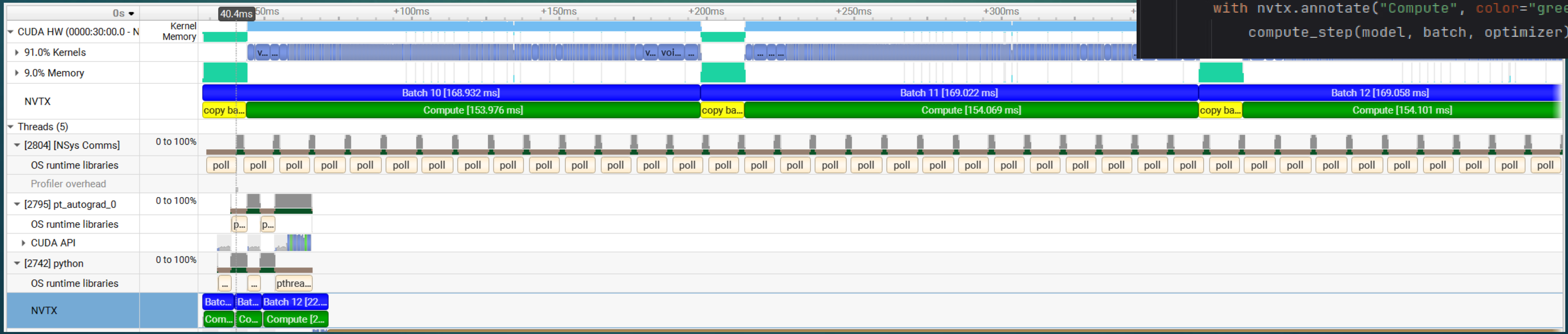
```python
NUM_WORKERS = 8
train_loader = DataLoader(
    FakeDataset(),
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS
)
data_iter = iter(train_loader)

for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        with nvtx.annotate("get batch", color="red"):
            batch = next(data_iter)
        with nvtx.annotate("copy batch", color="yellow"):
            batch = copy_data(batch)
        with nvtx.annotate("Compute", color="green"):
            compute_step(model, batch, optimizer)
```



Optimization 2: Non-blocking Data Copy

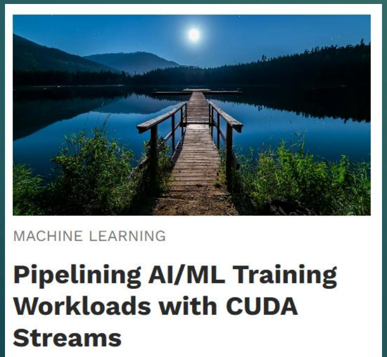
- Set non_blocking=True in the to() operation
  - Requires memory pinning in Data Loader (see documentation for limitations)
- Allows CPU to execute subsequent instructions before memory copy is complete
- Reduced idle time on GPU
- Nothing blocking CPU from queuing all operations
- BUT: memory (light green) and kernel (light blue) operations run sequentially on GPU

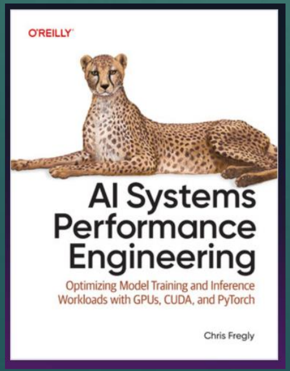
```python
train_loader = DataLoader(
    FakeDataset(),
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS,
    pin_memory=True
)

def copy_data(batch):
    data, targets = batch
    data_gpu = data.to(DEVICE, non_blocking=True)
    targets_gpu = targets.to(DEVICE, non_blocking=True)
    return data_gpu, targets_gpu

for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        with nvtx.annotate("get batch", color="red"):
            batch = next(data_iter)
        with nvtx.annotate("copy batch", color="yellow"):
            batch = copy_data(batch)
        with nvtx.annotate("Compute", color="green"):
            compute_step(model, batch, optimizer)
```



Optimization 3: Pipelining with CUDA Streams

- A CUDA stream is a queue of GPU operations executed sequentially
- We can run two (or more) CUDA streams concurrently
- Memory copy (DMA) and kernel compute (SMs) run on separate GPU engines
  - While SMs execute kernels on batch N on the compute stream, DMA copies batch N+1 on the copy stream
- See chapter 11 of AI Systems Performance Engineering

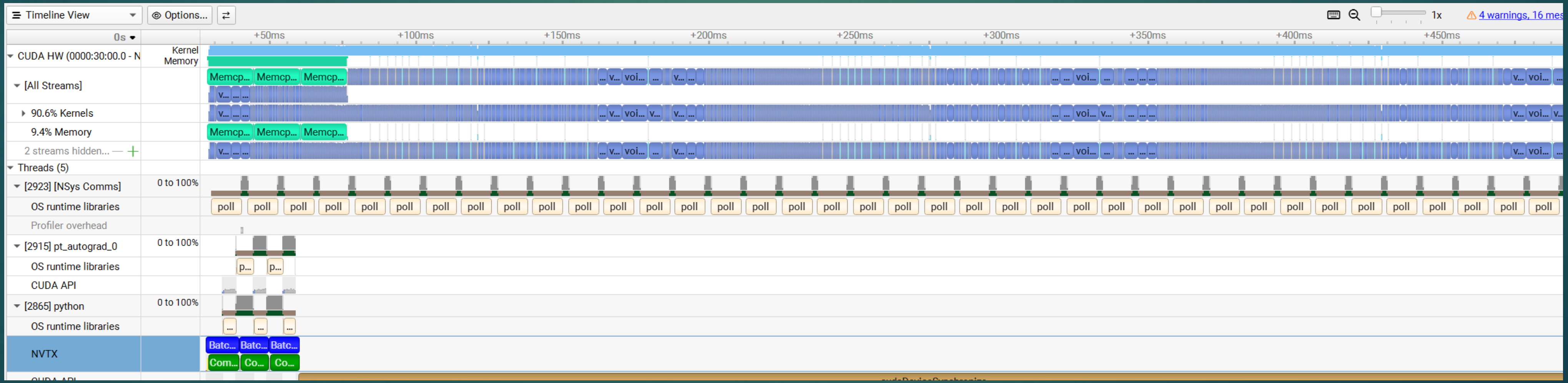




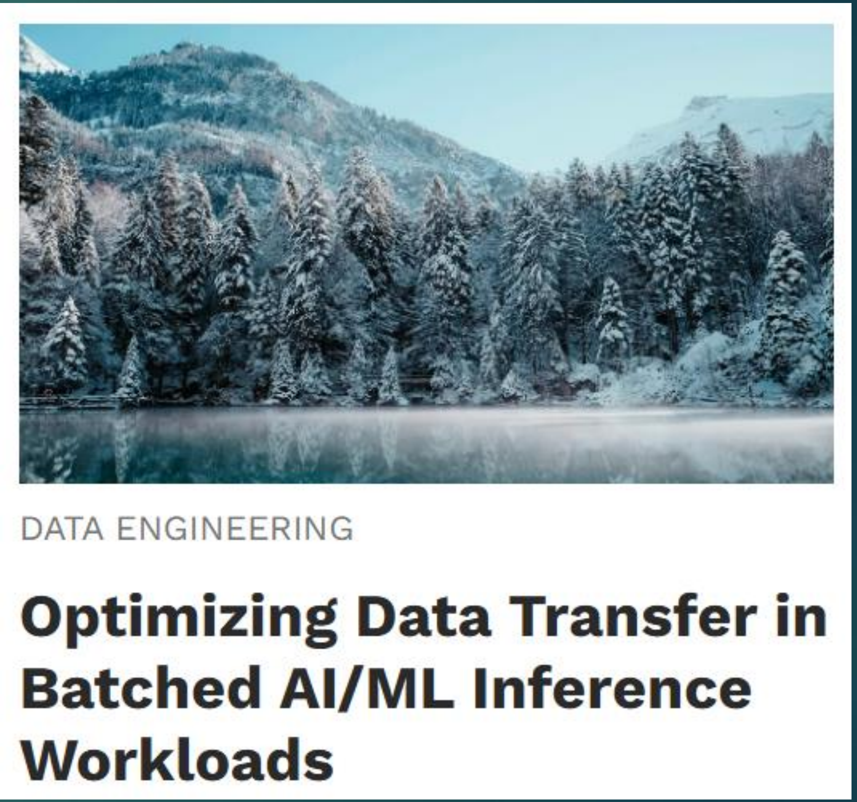

```python
# define two CUDA streams
compute_stream = torch.cuda.Stream()
copy_stream = torch.cuda.Stream()

# extract first batch
next_batch = next(data_iter)
with torch.cuda.stream(copy_stream):
    next_batch = copy_data(next_batch)

for i in range(TOTAL_STEPS):
    with nvtx.annotate(f"Batch {i}", color="blue"):
        # wait for copy stream to complete copy of batch N
        compute_stream.wait_stream(copy_stream)
        batch = next_batch
        # copy batch N+1 on copy stream
        try:
            with nvtx.annotate("get batch", color="red"):
                next_batch = next(data_iter)
            with torch.cuda.stream(copy_stream):
                with nvtx.annotate("copy batch", color="yellow"):
                    next_batch = copy_data(next_batch)
        except:
            # reached end of dataset
            next_batch = None
        # execute model on batch N compute stream
        with torch.cuda.stream(compute_stream):
            with nvtx.annotate("Compute", color="green"):
                compute_step(model, batch, optimizer)
```

Over 2X Speed-up

- Memory copies and kernel compute overlap
- GPU SMs are at full utilization
- Overall performance boost of 2.17X
  - With just a little bit of help from nsys profiler



2. Optimizing GPU-to-CPU Data Transfer in Batched Inference Workloads



A Toy Image Segmentation Model

- Experiment: Run batched inference using a DeepLabV3 model with a ResNet-50 backbone
- Annotate code blocks using NVIDIA Tools Extension (NVTX)
- Run on Amazon EC2 g6e.2xlarge (NVIDIA L40S GPU):

  nsys profile \

  --capture-range=cudaProfilerApi \

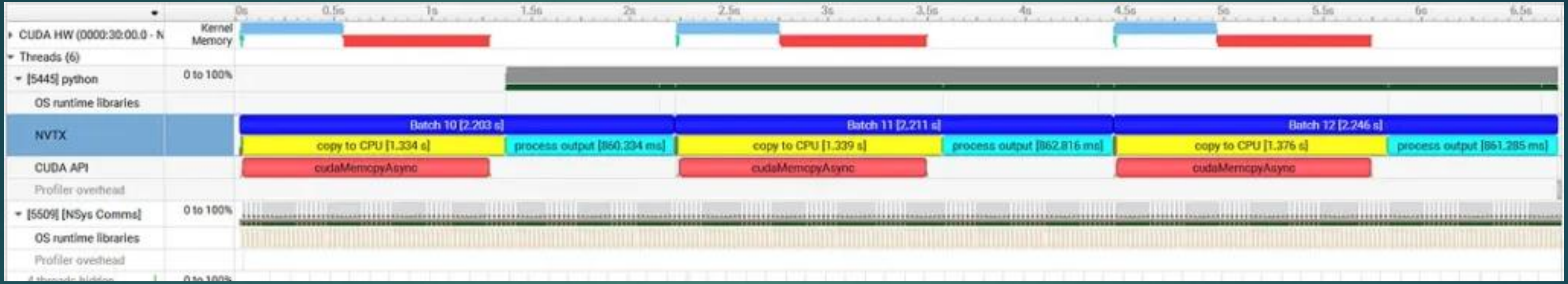
  --trace=cuda,nvtx,osrt \

  --output=baseline \

  python train.py
- Copy resultant nsys-rep file to development station for analysis

```
DEVICE = "cuda"
BATCH_SIZE = 64
IMG_SIZE = 512
N_CLASSES = 21

def to_cpu(output):
    return output.cpu()

def process_output(batch_id, logits):
    # do some post processing on output
    with open('/dev/null', 'wb') as f:
        f.write(logits.numpy().tobytes())

model = deeplabv3_resnet50(weights_backbone=None).to(DEVICE).eval()

with torch.inference_mode():
        with nvtx.annotate(f"Batch {i}", color="blue"):
            with nvtx.annotate("get batch", color="red"):
                batch = next(data_iter)
            with nvtx.annotate("compute", color="green"):
                output = model(batch)
            with nvtx.annotate("copy to CPU", color="yellow"):
                output_cpu = to_cpu(output['out'])
            with nvtx.annotate("process output", color="cyan"):
                process_output(i, output_cpu)
```

Nsight Systems Profiler Timeline

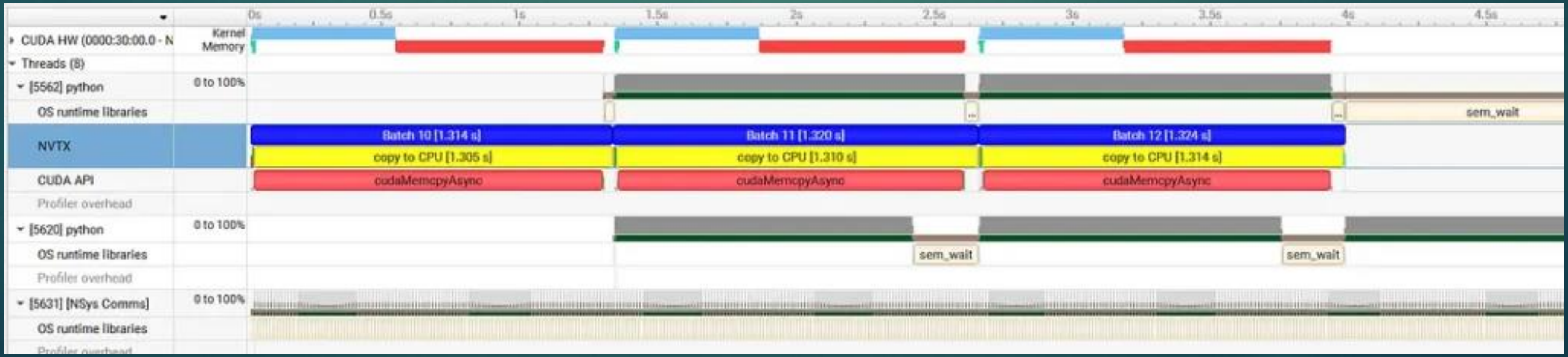


- Large portions of GPU idle time
- Bottleneck source is "process_output" (cyan)
- Our initial implementation runs model inference and output processing in single process

```
with nvtx.annotate(f"Batch {i}", color="blue"):
    with nvtx.annotate("get batch", color="red"):
        batch = next(data_iter)
    with nvtx.annotate("compute", color="green"):
        output = model(batch)
    with nvtx.annotate("copy to CPU", color="yellow"):
        output_cpu = to_cpu(output['out'])
    with nvtx.annotate("process output", color="cyan"):
        process_output(i, output_cpu)
```

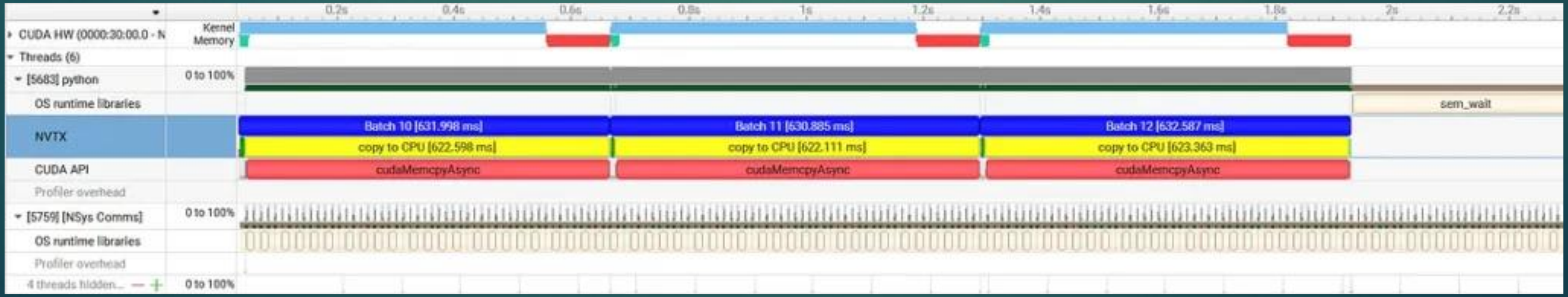
Optimization 1: Multi-Worker Output Processing

- Requires more manual labor than in CPU-to-GPU direction
  - Explicit definition of multiprocessing workers and an output queue
  - See blog post for details
- A small block of idle time – correlated (again) to memory operation(“munmap”)



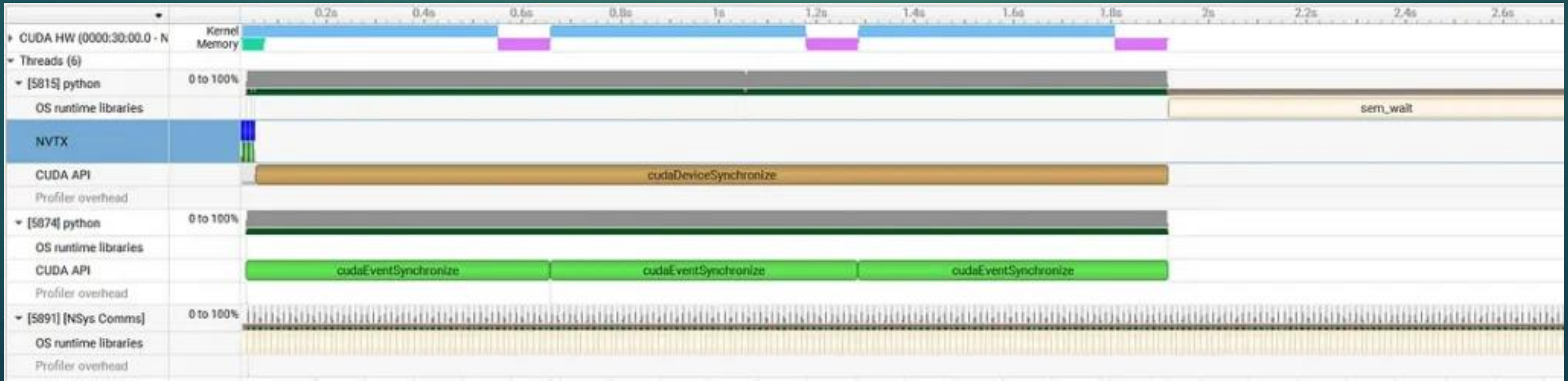
Optimization 2: Buffer Pool Pre-allocation

- Optimize memory management using a pre-allocated buffer pool
  - Buffers managed with a dedicated queue
- Throughput increases by ~2X
- But kernel loading is blocked by the **synchronous/blocking** data copy call



Optimization 3: Non-blocking Data Copy

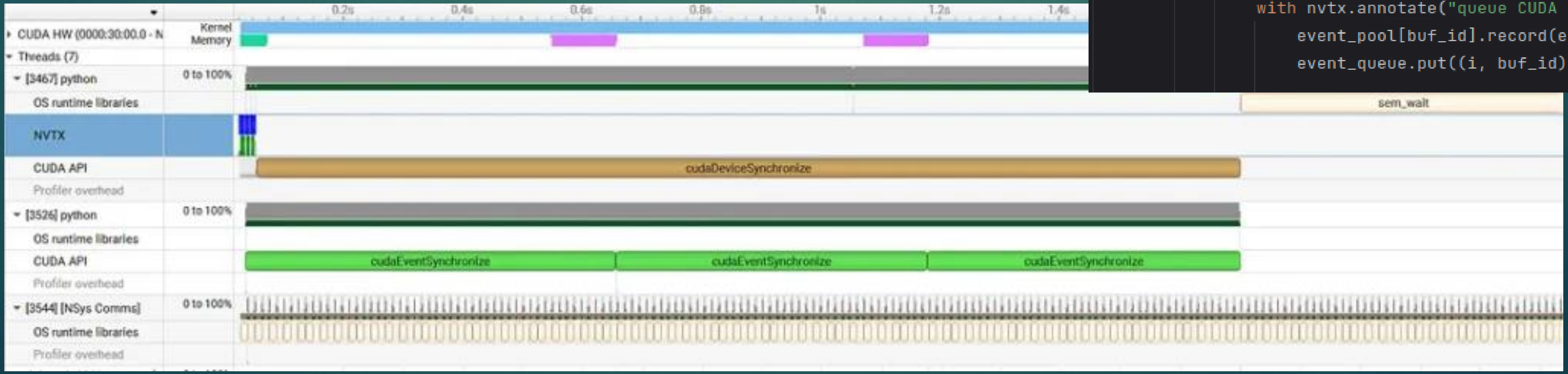
- Allows CPU to execute subsequent instructions before memory copy is complete
  - More nuanced than CPU-to-GPU direction
  - Requires CUDA events to ensure data integrity
  - See post for details
- Nothing blocking CPU from queuing all operations
- BUT: memory (pink) and kernel (light blue) operations run sequentially on GPU



Optimization 4: Pipelining with CUDA Streams

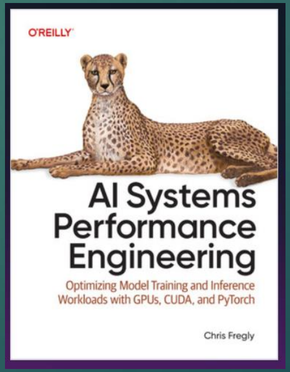
- Perform the GPU-to-CPU data copy on a dedicated stream
- Memory copies and kernel compute overlap
- GPU SMs are at full utilization
- Overall performance boost of 4.11X
  - With a little bit of help from nsys profiler

```python
egress_stream = torch.cuda.Stream()

with torch.inference_mode():
        with nvtx.annotate(f"Batch {i}", color="blue"):
            with nvtx.annotate("get batch", color="red"):
                batch = next(data_iter)
            with nvtx.annotate("compute", color="green"):
                output = model(batch)

            # on separate stream
            with torch.cuda.stream(egress_stream):
                # wait for default stream to complete compute
                egress_stream.wait_stream(torch.cuda.default_stream())
                with nvtx.annotate("copy to CPU", color="yellow"):
                    output_cpu, buf_id = to_cpu(output['out'])
                with nvtx.annotate("queue CUDA event", color="cyan"):
                    event_pool[buf_id].record(egress_stream)
                    event_queue.put((i, buf_id))
```



3. Optimizing Data Transfer in Distributed AI/ML Training Workloads (blog post pending): **GPU-to-GPU**

See chapter 4 in AI Systems Performance Engineering



Instance Selection for Distributed Training

- Appropriate instance selection can be critical for success
- Distributed training relies on data transfer between GPUs
- Run ***nvidia-smi topo -m*** to see how GPUs are linked:
  - On g6e.48xlarge GPUs are linked by PCIe – could suffice for workloads with a low communication-to-compute ration
  - On p4d.24xlarge GPUs are linked by NVLink – essential for workloads with high communication rates

DEEP LEARNING

Instance Selection for Deep Learning

How to choose the best machine for your ML workload

```
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      NODE    NODE    NODE    SYS     SYS     SYS     SYS     0-47,96-143     0               N/A
GPU1    NODE     X      NODE    NODE    SYS     SYS     SYS     SYS     0-47,96-143     0               N/A
GPU2    NODE    NODE     X      NODE    SYS     SYS     SYS     SYS     0-47,96-143     0               N/A
GPU3    NODE    NODE    NODE     X      SYS     SYS     SYS     SYS     0-47,96-143     0               N/A
GPU4    SYS     SYS     SYS     SYS      X      NODE    NODE    NODE    48-95,144-191   1               N/A
GPU5    SYS     SYS     SYS     SYS     NODE     X      NODE    NODE    48-95,144-191   1               N/A
GPU6    SYS     SYS     SYS     SYS     NODE    NODE     X      NODE    48-95,144-191   1               N/A
GPU7    SYS     SYS     SYS     SYS     NODE    NODE    NODE     X      48-95,144-191   1               N/A

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

GPU Topology of g6e.48xlarge (by Author)

```
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity    NUMA Affinity   GPU NUM
GPU0     X      NV12    NV12    NV12    NV12    NV12    NV12    NV12    0-23,48-71      0               N/A
GPU1    NV12     X      NV12    NV12    NV12    NV12    NV12    NV12    0-23,48-71      0               N/A
GPU2    NV12    NV12     X      NV12    NV12    NV12    NV12    NV12    0-23,48-71      0               N/A
GPU3    NV12    NV12    NV12     X      NV12    NV12    NV12    NV12    0-23,48-71      0               N/A
GPU4    NV12    NV12    NV12    NV12     X      NV12    NV12    NV12    24-47,72-95     1               N/A
GPU5    NV12    NV12    NV12    NV12    NV12     X      NV12    NV12    24-47,72-95     1               N/A
GPU6    NV12    NV12    NV12    NV12    NV12    NV12     X      NV12    24-47,72-95     1               N/A
GPU7    NV12    NV12    NV12    NV12    NV12    NV12    NV12     X      24-47,72-95     1               N/A

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

GPU Topology of p4d.24xlarge (by Author)

A Toy Vision Transformer Model

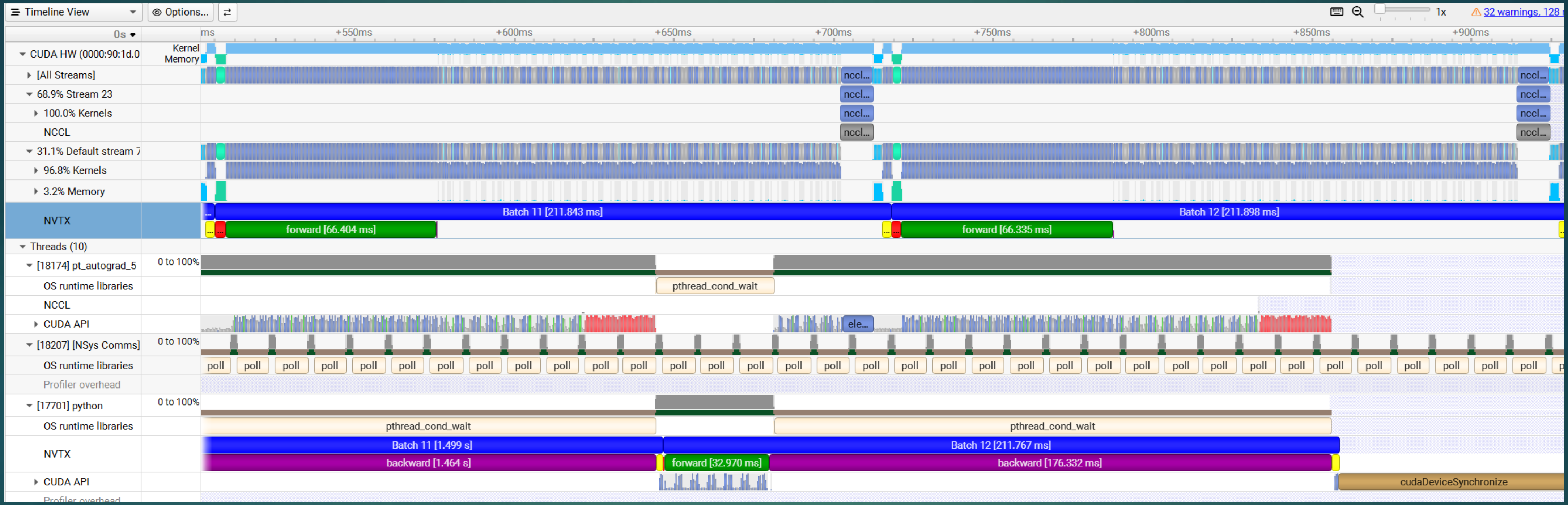
- Experiment: Run data-distributed training on a [Vision Transformer (ViT)-L/32]() image-classification model (~306 million parameters)
  - Relatively high communication/compute ratio
- Annotate code blocks using NVIDIA Tools Extension (NVTX)
- Naively configure DDP to use a single DDP bucket
- Run with NCCL option:

```
torchrun --nproc_per_node=8 \
  --no-python \
 nsys profile \
  --capture-range=cudaProfilerApi \
  --capture-range-end=stop \
  --trace=cuda,nvtx,osrt,nccl \
  --output=ddp-8gpu_%q{RANK} \
  python train.py
```

- Copy resultant nsys-rep files to development station for analysis (multi-report view)

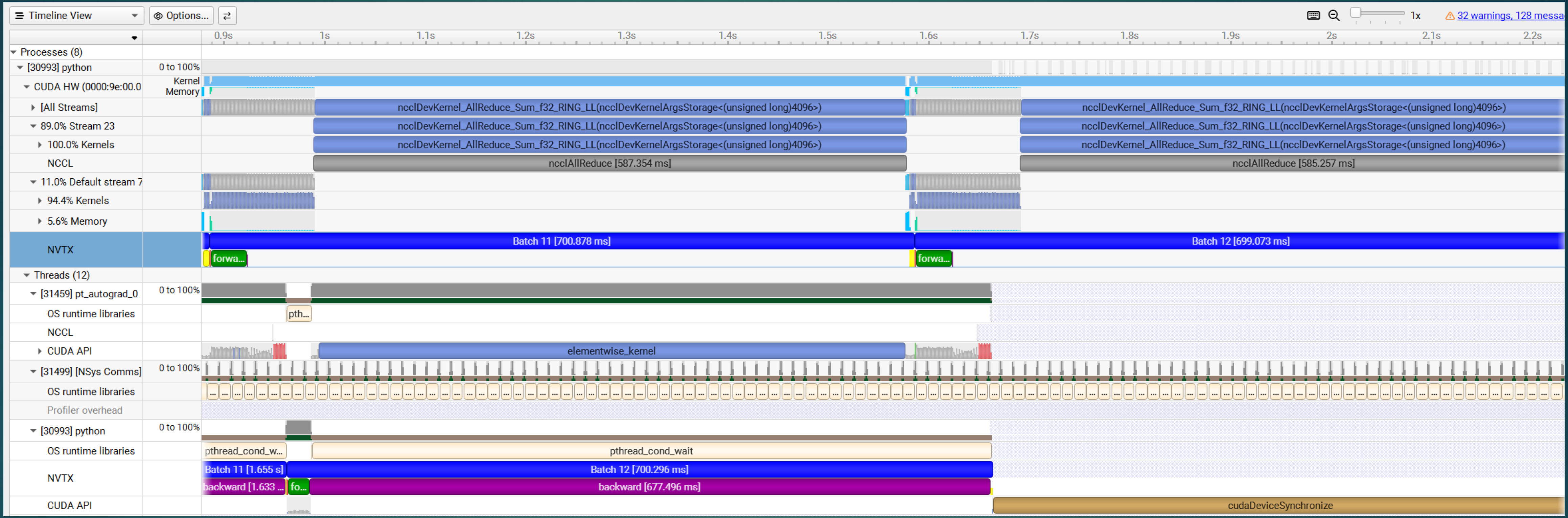
```python
WORLD_SIZE = int(os.environ.get("WORLD_SIZE", 1))
BATCH_SIZE = 32
N_WORKERS = 8

def get_model():
    return vit_l_32(weights=None)

def get_data_iter(rank, world_size):  1 usage
    dataset = FakeDataset()
    sampler = DistributedSampler(dataset, num_replicas=world_size,
                                 rank=rank, shuffle=True)
    train_loader = DataLoader(dataset, batch_size=BATCH_SIZE,
                              sampler=sampler, num_workers=N_WORKERS,
                              pin_memory=True)
    return iter(train_loader)

def configure_ddp(model, rank):
    dist.init_process_group("nccl")
    model = DDP(model,
                device_ids=[rank],
                bucket_cap_mb=2000)
    return model

def train():
    # detect the env vars set by torchrun
    rank = int(os.environ.get("RANK", 0))
    local_rank = int(os.environ.get("LOCAL_RANK", 0))
    torch.cuda.set_device(local_rank)
    model = get_model().to(local_rank)
    criterion = nn.CrossEntropyLoss().to(rank)
    model = configure_ddp(model, rank)
    optimizer = optim.SGD(model.parameters())
    data_iter = get_data_iter(rank, WORLD_SIZE)
    model.train()
    for i in range(TOTAL_STEPS):
        with nvtx.annotate(f"Batch {i}", color="blue"):
            with nvtx.annotate("get batch", color="red"):
                data, target = next(data_iter)
                data = data.to(local_rank, non_blocking=True)
                target = target.to(local_rank, non_blocking=True)
            with nvtx.annotate("forward", color="green"):
                output = model(data)
                loss = criterion(output, target)
            with nvtx.annotate("backward", color="purple"):
                optimizer.zero_grad()
                loss.backward()
            with nvtx.annotate("optimizer step", color="yellow"):
                optimizer.step()
    dist.destroy_process_group()
```

p4d Profiler Timeline



- In the CUDA NCCL row we see gradient all reduce command
- Model execution is ~192 ms, communication is ~10 ms

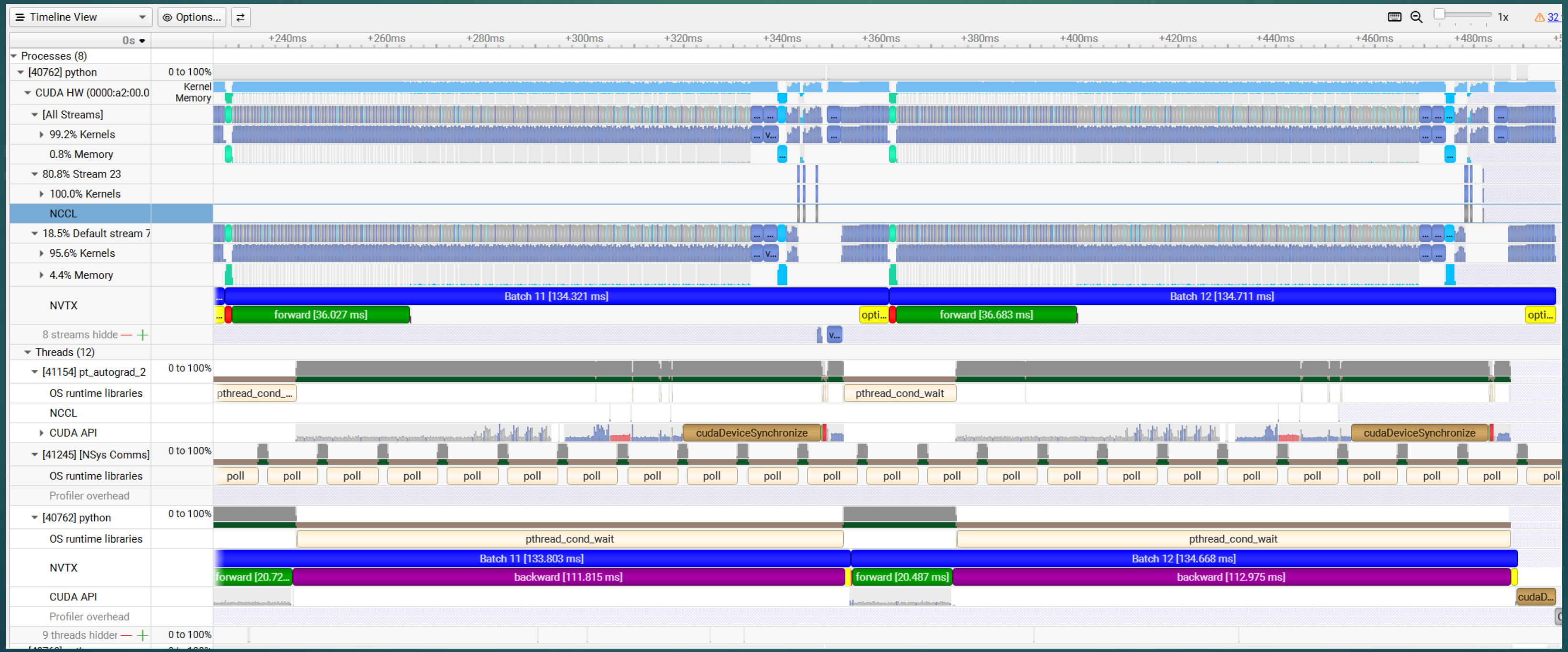
g6e Profiler Timeline



- Model execution is faster on L40S – 100 ms (more modern GPU architecture)
- But NCCL kernel dominates step – 588 ms
- Slow GPU-to-GPU throughput results in significant bottleneck

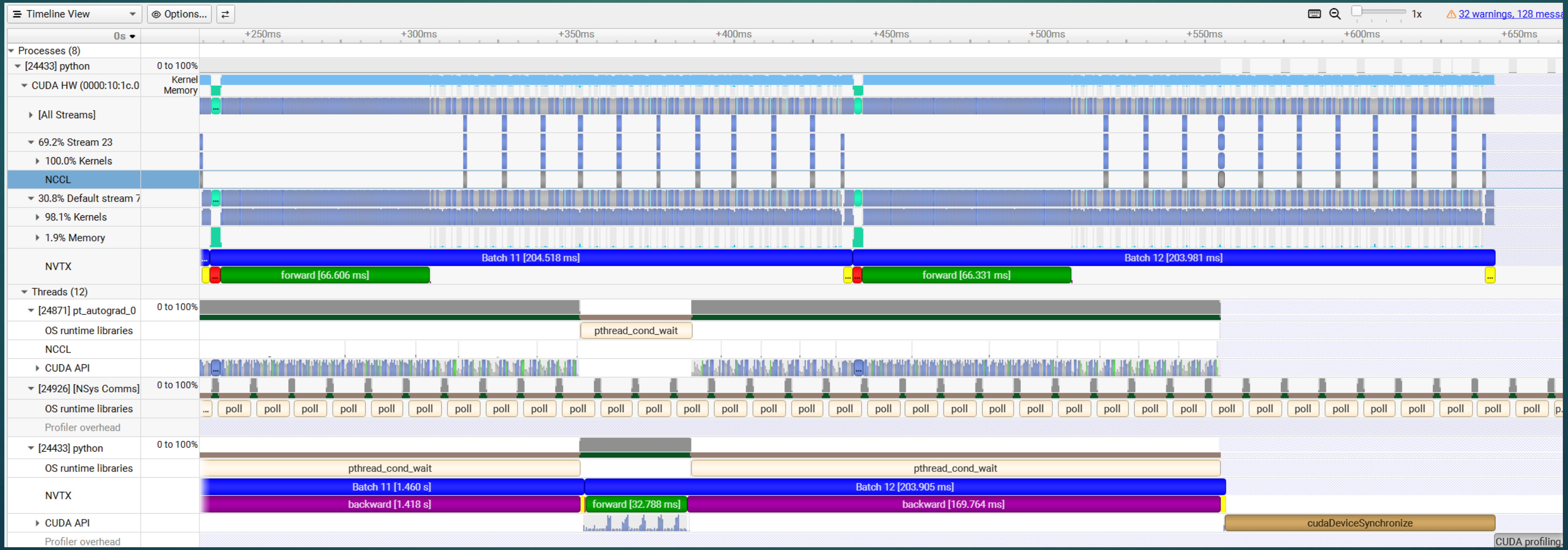
Optimization 1: Gradient Compression

- Reduce payload of data transfer
- Compression scheme examples, fp16, bf16, PowerSGD (requires tuning) – all "lossy"
- Significant (5X) improvement on g6e (shown below)
- Harms throughput on p4d (due to overhead of compression)



Optimization 2: Parallelize Communication with Backward

- Tune *bucket_cap_mb* flag of DDP container
- On p4d optimal value (*bucket_cap_mb*=100) – boosts throughput by 4%
  - Multiple small NCCL calls instead of a single large one



Key Takeaways

- Use NVIDIA Nsight System Profiler to analyze **system-wide** performance of AI/ML workloads
- Data transfer are often a source of performance bottlenecks / resource under-utilization
  - Often present opportunities for optimization – with the help of PyTorch and nsys profilers.
- AI/ML developers must take responsibility for the runtime performance of their models
  - Integrate profiling analysis and optimization into AI/ML development workflow
  - You do not need to be a CUDA/optimization expert to boost runtime performance and reduce AI/ML costs