PyTorch

Table of Contents

Tracing Time vs. Execution Time in PyTorch/XLA

When working with PyTorch/XLA, it's essential to understand that operations on XLA tensors are not typically executed immediately in the way they are with standard PyTorch tensors on CPU or CUDA devices (which operate in "eager mode"). PyTorch/XLA employs a "lazy execution" model. This means that when you write PyTorch code using XLA tensors, you are primarily defining or tracing a computation graph. The compilation of the currently traced graph and it's subsequent execution on the device is deferred until a specific trigger point.

This leads to two distinct types of "time" to consider:

1. Host-Side Time: The period during which your CPU (host) prepares the computation. This includes:
    - Tracing Time: The period during which PyTorch/XLA records your operations and builds the computation graph.
    - Compilation Time: The time the host-side XLA compiler takes to transform the traced graph into optimized device code. This is most significant on the first execution of a new graph or if the graph changes.

2. Device Time: The period during which the XLA device (e.g., TPU) actually performs the computations. This is primarily:
    - Execution Time: The time the XLA device spends running the compiled code.

Illustrating a Common Pitfall: Measuring Only Tracing Time

When you write PyTorch code using XLA tensors (e.g., tensors on a TPU), PyTorch/XLA doesn't execute each operation on the device right away. It traces these operations, adding them to an internal computation graph. **If you measure the duration of code that only performs XLA operations without an explicit instruction to wait for the device, you are primarily measuring this tracing time plus Python overhead.**

Consider the following conceptual code:

```
# Assume 'a' and 'b' are XLA tensors
# start_time = time.perf_counter()

# This operation is recorded in PyTorch/XLA's graph
result = torch.matmul(a, b)

#!!! INCORRECT PROFILING: compilation and execution are deferred !!!
# end_time = time.perf_counter()
# elapsed_time = end_time - start_time
```

The `elapsed_time` here would predominantly reflect how long it took PyTorch/XLA to trace the matmul operation. The actual matrix multiplication on the XLA device, along with its compilation, is not started.

Measuring End-to-End Performance

To correctly profile the performance of your code on the XLA device, you must ensure that your timing captures host-side compilation and devide execution. This involves:

1. Ensure the traced computational graph is compiled, if it's the first time this graph is seen or if it is changed, and sent to the device for execution.
2. Make sure the Python script waits until the XLA device has completed all its assigned computations before taking the final timestamp.

This is exemplified, using `torch_xla.sync(wait=True)`, in the following conceptual code:

```
# Assume 'a' and 'b' are XLA tensors
# start_time = time.perf_counter()

result = torch.matmul(a, b)

# CORRECT PROFILING: Explicitly wait for the XLA device to finish.
torch_xla.sync(wait=True)

# end_time = time.perf_counter()
# elapsed_time = end_time - start_time
```

By performing warm-up iterations (running the target code with `torch_xla.sync(wait=True)` before the timed section), you can often isolate the steady-state execution time from the initial compilation time for more consistent benchmarking.

Triggering Execution and Ensuring Completion

Several mechanisms trigger graph execution and/or ensure completion:

1. `torch_xla.sync(wait=True)`: This is the most direct method for benchmarking. It ensures all pending XLA operations are launched and, crucially, blocks the Python script until the device finishes.
2. `Data Access/Transfer`: Operations like `tensor.cpu()`, `tensor.item()`, or printing an XLA tensor require the actual data. To provide it, PyTorch/XLA must execute the graph that produces the tensor and wait for its completion.
3. `torch_xla.core.xla_model.mark_step()`: Commonly used in loops, this signals to XLA that a logical unit of work (e.g., a training step) is defined and can be executed. It triggers execution but doesn't inherently block Python until device completion.
4. `torch_xla.core.xla_model.optimizer_step(optimizer)`: Reduces gradients, applies optimizer updates, and conditionally triggers `mark_step` via its barrier argument (default False, as data loaders often handle `mark_step`).

5. `torch_xla.core.xla_model.unlazy(tensors)` : Blocks until specified tensors are materialized.

Case Study: Correctly Profiling Loops with `mark_step`

A common scenario involves loops, such as in model training, where `mark_step` is used. Consider this structure:

```
def run_model_step():
    #... XLA tensor operations...
    pass

# start_loop_time = time.perf_counter()
for step in range(num_steps):
  run_model_step()  # Operations are traced
  torch_xla.core.xla_model.mark_step()    # Graph for this step is submitted for execution

#!!! INCORRECT PROFILING APPROACH FOR TOTAL TIME!!!
# end_loop_time = time.perf_counter()
# elapsed_loop_time = end_loop_time - start_loop_time
```

The `elapsed_loop_time` in this case primarily measures the cumulative host-side time. This includes:

1. The time spent in `run_model_step()` for each iteration (largly tracing).
2. The time taken by `mark_step` in each iteration to trigger the host-side compilation (if the graph is new or changed) and dispatch the graph for that step to the XLA device for execution.

Crucially, `mark_step` typically works asynchronously: The Python loop proceed to trace the next step while the device is still performing its execution for the current or previous step. Thus, `elapsed_loop_time` does not guarantee inclusion of the full device execution time for all `num_steps` if the device work lags behind the Python loop.

In order to measure total loop time (including all device execution), `torch_xla.sync(wait=True)` has to be added after the loop and before taking the final timestamp.

```
# start_loop_time = time.perf_counter()
for step in range(num_steps):
  run_model_step()
  torch_xla.core.xla_model.mark_step()

# CORRECT PROFILING: Wait for ALL steps to complete on the device.
torch_xla.sync(wait=True)

end_loop_time = time.perf_counter()
elapsed_loop_time = end_loop_time2 - start_loop_time
```

‹ Previous Next ›