Abstract

PyTorch/XLA enables PyTorch users to run their models on XLA devices including Google's Cloud TPUs. The latest improvements in PyTorch/XLA enables training PyTorch models using FSDP to train very large models. In this work, we present benchmarks and Hardware Flops Utilization of training HuggingFace GPT-2 on Cloud TPU v4.

Fully Sharded Data Parallel (FSDP)

FSDP was first introduced in PyTorch 1.11 as a prototype feature and we implemented it for XLA devices as part of the PyTorch/XLA 1.12 release. We provide an FSDP interface with a similar high-level design to the CUDA-based PyTorch FSDP class while also handling several restrictions in XLA (see Design Notes below for more details). This FSDP interface allowed us to easily build models with e.g. 108+ parameters on TPUs and has enabled many research explorations. The API to use FSDP is simple and intuitive. An example of wrapping a module with FSDP can be seen below.

```python
import torch
import torch_xla.core.xla_model as xm
from torch_xla.distributed.fsdp import XLaFullyShardedDataParallel as FSDP
model = FSDP(model(module))
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
output = model(x, y)
loss = output.sum()
loss.backward()
```

Cloud TPU

Cloud TPU v4 Pods are now Generally Available. These machines are powered by the TPU v4 chip which is more than twice as fast as the TPU v3 chip. Like previous TPU generations, these v4 chips are connected together into supercomputers called pods. A single Cloud TPU v4 Pod contains 4,096 v4 chips, and each pod has 10x the interconnect bandwidth per chip at scale compared to any other networking technology. This makes it possible for a TPU v4 Pod to deliver more than 1 exaFLOP per second of computing power with 32Gb of per device HBM.

Cloud TPU VMs enable PyTorch / XLA to run directly on TPU host machines (the machines that are connected to the Cloud TPU accelerators). Users can SSH directly to a Google Compute Engine VM running on each TPU host.

PyTorch/XLA now offers experimental support for PJRT on Cloud TPU v4. PJRT is the runtime API that is shared with JAX.

Experiment Setup

For this benchmarking, HuggingFace GPT-2 model was chosen. With less than ~20 lines of code edits, we run this model on TPU v4.

```
<table>
<thead>
<tr>
<th>PyTorch Version</th>
<th>Nightly at 09/27/2022</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch/XLA Version</td>
<td>Nightly at 09/27/2022</td>
</tr>
<tr>
<td>Accelerator Type</td>
<td>Cloud TPU v4</td>
</tr>
<tr>
<td>Runtime</td>
<td>PJRT</td>
</tr>
<tr>
<td>Model Code</td>
<td>HuggingFace GPT 2</td>
</tr>
<tr>
<td>Precision</td>
<td>BF16</td>
</tr>
</tbody>
</table>
```

Methodology

We first use profiler to capture the step time

```
for i in range(n)
    with torch.autocast(device_type = 'cuda', dtype = torch.bfloat16):
        output = model(x, y)
        loss = output.sum()
        loss.backward()
```

then we use the following formula to calculate the Hardware flops utilization (HFU). We then compare obtained HFU with the one obtained from the most optimized JAX implementation of a similar model.

```
flops_per_step = 6 * global_batch_size * seq_len * num_params
model_flops_utilization = flops_per_step / step_time(s) / chip_count / flops_per_chip
```

Challenges

Host side out of memory

The common practice for PyTorch users is to first build the model layer by layer on CPU and move all of the model weights to the accelerator at the end. As we scale up the model size, we start to run into host size OOM because parameters become too big to fit into the host RAM. The solution we had was to build and model transfer the weights to the accelerator at the end. As we scale up the model size, we start to run into host size OOM because parameters become too big to fit into the host RAM. The solution we had was to build and model transfer the weights to the accelerator at the end.

Device side out of memory

As we grow the model size, weights can't fit into the device memory. The most obvious choice is to increase the accelerator size but we also used the gradient checkpoint technique provided by the FSDP to reduce the peak device memory usage by recomputing some of the gradients during the backward pass.

Results

<table>
<thead>
<tr>
<th>Synthetic model size</th>
<th>Hardware Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>29%</td>
</tr>
<tr>
<td>512</td>
<td>29%</td>
</tr>
<tr>
<td>1024</td>
<td>28%</td>
</tr>
<tr>
<td>2048</td>
<td>27%</td>
</tr>
<tr>
<td>3072</td>
<td>26%</td>
</tr>
</tbody>
</table>

Hardware Utilization shows a drop on 128B parameters. It is likely due to the fact that the model size is increased but the Per Device Batch Size was not. We first run 128B parameters on v4-512 (chip count 256) and get utilization of ~29%. We then run the experiment on v4-1024 with Per Device Batch Size 4 and get a better utilization as reported in the chart above. We expect that running on a larger accelerator size with a larger Per Device Batch Size will result in a much better utilization.

The utilization we get here can be improved significantly. We worked with the XLA team to take a couple of iterations on the HLO but we were not able to fix everything due to the time constraint of this benchmarking project. We will continue working on the model/lowering optimization.

Conclusion

The obtained results demonstrate the ability to scale models using PyTorch/XLA. It further shows that the Hardware Flops Utilization is not far from optimum. We should also note that 128B is not the limit of the PyTorch/XLA + FSDP. We can run with larger models if we reduce the global batch size or use a larger accelerator slice.

Acknowledgements

Thanks to Vaibhav Singh, Chandra Devarakonda, Steven Krawczyk, Qinwen Xu, Will Cromar, Amit Sabne, Allen Wang, Milad Mohammadi for various TPU-related contributions to this benchmarking effort. Thanks to Junmin Hao from AWS for reviewing the PyTorch/XLA FSDP pull request.