NVFuser - A Fusion Code Generator for NVIDIA GPUs

NVFuser is integrated as a backend for TorchScript's Profiling Graph Executor

Enabling NVFuser

NVFuser is not currently the default fuser for NVIDIA GPUs.

Fusions will only show up during the ~3rd iteration of execution, the exact number depends on profiling executor's optimization phases

Enable by Context Manager

```
jit_model = torch.jit.script(model)

with torch.jit.fuser("fuser2") :
    for _ in range(5) :
        outputs = jit_model(inputs)
```

Enable by Specific Functions

1. Disable cpu/gpu fusion for native/nnc fuser

```
torch._C._jit_override_can_fuse_on_cpu(False)
torch._C._jit_override_can_fuse_on_gpu(False)
```

2. Disable nnc fuser

```
torch._C._jit_set_texpr_fuser_enabled(False)
```

3. Enable nvfuser

```
torch._C._jit_set_nvfuser_enabled(True)
```

Simple knobs to change fusion behavior

1. Allow single node fusion `torch._C._jit_set_nvfuser_single_node_mode(True)` Fusion group is only created when two or more compatible ops are grouped together. Turn on single node fusion would allow fusion pass to create fusion group with a single node, this is very handy for testing and could be useful when single node generated kernel out-performs native cuda kernels in framework.
2. Allow horizontal fusion `torch._C._jit_set_nvfuser_horizontal_mode(True)` Fusion pass fuses producer to consumer, horizontal mode allows sibling nodes that shared tensor input to be fused together. This could save input memory bandwidth.

3. Turn off guard for fusion `torch._C._jit_set_nvfuser_guard_mode(False)` This disables the runtime check on fusion group pre-assumptions (tensor meta information / constant inputs / profiled constants), this really is only used for testing as we want to ensure generated kernels are indeed tested and you should avoid using this in training scripts.

Fusion Debugging

Given the following script as an example

```
import torch

def forward(x):
    o = x + 1.0
    o = o.relu()
    return o

shape = (2, 32, 128, 512)
input = torch.rand(*shape).cuda()
t = torch.jit.script(forward)

with torch.jit.fuser("fuser2"):
    for k in range(4):
        o = t(input)
```

TorchScript Based Debugging

1. TorchScript IR Graph

Usage Two easy ways to checkout fusion for graph: The first one is to print out graph in python script after a few runs (for optimization to kick in).

```
print(t.graph_for(input))
```

The second way is to turn on graph dumping in profiling executor via command line below:

```
PYTORCH_JIT_LOG_LEVEL="profiling_graph_executor_impl" python <your pytorch script>
```

Example Output Graph print out is straight forward and you should look for `prim::CudaFusionGroup_X` for fused kernels. While profiling executor dumps many things, but the most important part is `Optimized Graph`. In this example, it shows a Fusion Group, which is an indication that fusion is happening and you should be expecting fused kernel!

```
  Optimized Graph:
  graph(%x.1 : Tensor):
    %12 : bool = prim::CudaFusionGuard[types=[Float(2, 32, 128, 512, strides=[2097152, 65536
```

```
  %11 : Tensor = prim::If(%12)
    block0():
      %o.8 : Tensor = prim::CudaFusionGroup_0[cache_id=0](%x.1)
      -> (%o.8)
    block1():
      %18 : Function = prim::Constant[name="fallback_function", fallback=1]()
      %19 : (Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, dev
      %20 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, devi
      -> (%20)
  return (%11)
with prim::CudaFusionGroup_0 = graph(%2 : Float(2, 32, 128, 512, strides=[2097152, 65536,
  %4 : int = prim::Constant[value=1]()
  %3 : float = prim::Constant[value=1.]() # test.py:6:12
  %o.1 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device=
  %o.5 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device=
  return (%o.5)
```

Note that one thing that could prevents fusion when you are running training is autodiff. Fusion pass only runs within `prim::DifferentiableGraph`, so the first thing you should check is to that targetted ops are within differentiable graph subgraphs. Graph dump could be quite confusing to look at, since it naively dumps all graphs executed by profiling executor and differentiable graphs are executed via a nested graph executor. So for each graph, you might see a few segmented `Optimized Graph` where each corresponds to a differentiable node in the original graph.

2. Cuda Fusion Graphs

Usage Cuda fusion dump gives the input and output graph to fusion pass. This is a good place to check fusion pass logic.

```
PYTORCH_JIT_LOG_LEVEL="graph_fuser" python <your pytorch script>
```

Example Output Running the same script above, in the log, you should be looking for two graphs `Before Fusion` shows the subgraph where fusion pass runs on; `Before Compilation` shows the graph sent to codegen backend, where each `CudaFusionGroup` will trigger codegen runtime system to generate kernel(s) to execute the subgraph.

```
Before Fusion:
graph(%x.1 : Tensor):
  %2 : float = prim::Constant[value=1.]()
  %1 : int = prim::Constant[value=1]()
  %3 : Tensor = prim::profile[profiled_type=Float(2, 32, 128, 512, strides=[2097152, 65536
  %o.10 : Tensor = aten::add(%3, %2, %1) # test.py:6:8
  %5 : Tensor = prim::profile[profiled_type=Float(2, 32, 128, 512, strides=[2097152, 65536
  %o.7 : Tensor = aten::relu(%5) # test.py:7:8
```

```
  %7 : Tensor = prim::profile[profiled_type=Float(2, 32, 128, 512, strides=[2097152, 65536
  %8 : Tensor = prim::profile[profiled_type=Float(2, 32, 128, 512, strides=[2097152, 65536
  return (%7, %8)

Before Compilation:
graph(%x.1 : Tensor):
  %13 : bool = prim::CudaFusionGuard[types=[Float(2, 32, 128, 512, strides=[2097152, 65536
  %12 : Tensor = prim::If(%13)
    block0():
      %o.11 : Tensor = prim::CudaFusionGroup_0(%x.1)
      -> (%o.11)
    block1():
      %o.7 : Tensor = prim::FallbackGraph_1(%x.1)
      -> (%o.7)
  return (%12, %12)
with prim::CudaFusionGroup_0 = graph(%2 : Float(2, 32, 128, 512, strides=[2097152, 65536,
  %4 : int = prim::Constant[value=1]()
  %3 : float = prim::Constant[value=1.]()
  %o.10 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device
  %o.7 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device=
  return (%o.7)
with prim::FallbackGraph_1 = graph(%x.1 : Float(2, 32, 128, 512, strides=[2097152, 65536,
  %1 : int = prim::Constant[value=1]()
  %2 : float = prim::Constant[value=1.]()
  %o.10 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device
  %o.7 : Float(2, 32, 128, 512, strides=[2097152, 65536, 512, 1], requires_grad=0, device=
  return (%o.7)
```

General ideals of debug no-fusion

Currently there we have a few consumers that utilizes nvfuser via lowering computations to TorchScript and executing that through a ProfilingExecutor.

Without going into too much details about how the integration is done, a few notes on debugging no-fusion on ProfilingExecutor:

1. Run TorchScript module multiple times (5 could be a lucky number) to enable fusion. Because ProfilingExecutor takes the first (few) runs for profiling, later optimization (including the fusion pass the enables nvfuser) relies on profiling information to run, so your initial runs are not going to trigger fused kernels. Note that the number of profiling runs is dependent on your model.
2. Fused kernel should show up in TorchScript IR as `prim::CudaFusionGroup`. You can look at your TorchScript optimized graph to see if fusion is happening `jit_model.graph_for(*inputs)`.
3. If your scripted model has inputs requiring gradient, fusion is only

happening for graphs inside `prim::DifferentiableGraph`. There are many reasons why your graph is not autodiff-able. Take a look at `/torch/csrc/jit/runtime/symbolic_scripts.cpp`, which lists all autodiff-able ops (note that this is a different list from autograd-supported ops). There's also a threshold where tiny autodiff graph are inlined/reverted, which could be disabled via `torch._C._debug_set_autodiff_subgraph_inlining(False)`.

General ideals of debug nvfuser mal-functioning

Assuming we have ProfilingExecutor things worked out properly, that is, you see a region that's supposed to be fused but did not ended up in a fused kernel, here's ways to dig deeper:

1. Dump fusion pass result: `PYTORCH_JIT_LOG_LEVEL=graph_fuser python your_script.py &> log`

   Looks for graph dumped with `Before Fusion` & `Before Compilation`, which shows the portion of graph where fusion pass runs on and the result of fusion (`CudaFusionGroup`).

2. Check out which ops are not fused and roughly why: `PYTORCH_JIT_LOG_LEVEL=">partition:graph_fuser python your_script.py &> log`

   Enabling GRAPH_UPDATE from partition.cpp dumps a log when a given node is rejected by fusion.

3. Disabling FALLBACK path: If you see a warning where a FALLBACK path has been taken while executing your model with nvfuser enabled, it's indicating that either codegen or fusion pass has failed unexpectedly. This is likely to cause regression on model performance, even though it's still functionally correct. We recommend to disable FALLBACK path, so error would be reported properly to open an informative issue.

   ```
   PYTORCH_NVFUSER_DISABLE_FALLBACK=1 python your_script.py &> log
   ```

4. Pin point kernel/fusion pattern that's causing error: With a larger model that includes multiple fusion patterns, it could be tricky to figure out which exact fusion is causing FALLBACK and build up a minimal python repro. One quick thing to try is to run the example with a few knobs turned on:

   ```
   PYTORCH_NVFUSER_DISABLE_FALLBACK=1 \
   PYTORCH_JIT_LOG_LEVEL=">partition:graph_fuser:>>kernel_cache" \
   python your_script.py &> log
   ```

   This logs all TorchScript IR parsed to codegen IR as well as kernel generated and executed by nvfuser. Since fallback path is disabled, it's likely that the last log would indicate the failing fusion.

Hint: look for last `Before Compilation:` that indicates a parsing failure, or `running GraphCache: xxxxx`, which indicates jit compilation/execution failure (also search for the GraphCache address, which would should have dumped a TorchScript IR earlier.

Query nvfuser codegen kernels

There're a few debug dump that could be turned on via environment variables. Look for `PYTORCH_NVFUSER_DUMP` inside `[pytorch_source_path]/torch/csrc/jit/codegen/cuda/utils.cpp` A few useful ones are: 1. `dump_eff_bandwidth`: print out effective bandwidth of each generated kernel. This naively measure the kernel time divided by I/O buffer size and is a good/simple metric of performance for bandwidth bound kernels 2. `cuda_kernel`: print out generated cuda kernels 3. `launch_param`: print out launch config of generated kernels 4. `print_args`: print out input output tensors of executed codegen kernels

FAQs

1. There's regression after turning on nvfuser.

First thing is to check that you have fusion kernel running properly. Try to run your model with fallback disabled to see if you hit any errors that caused fallback via `export PYTORCH_NVFUSER_DISABLE_FALLBACK=1`.

2. I didn't see any speedup with nvfuser.

Check if there is fusion in your script model. Run your script with `PYTORCH_JIT_LOG_LEVEL="graph_fuser"`, you should see some log dump of before/after graph regarding fusion pass. If nothing shows up in the log, that means something in TorchScript is not right and fusion pass are not executed. Check General ideals of debug no-fusion for more details.