Technical Review on PyTorch 2.0 and Triton

Keren Zhou
George Mason University
kzhou6@gmu.edu
Transform DNNs to Low Level Code

```python
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Graph</th>
<th>Kernel</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>XLA/HLO</td>
<td>CUDA</td>
<td>GPU</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>TVM/Relay</td>
<td>HIP</td>
<td>CPU</td>
</tr>
<tr>
<td>JAX</td>
<td>PyTorch/fx</td>
<td>OpenCL</td>
<td>FPGA</td>
</tr>
</tbody>
</table>
## Transform DNNs to Low Level Code

### Diagram

```
   a  b  
   ↑  ↑  
  mm   
  ↓  ↓  
 add  c  
  ↓  ↓  
e
```

### Table

<table>
<thead>
<tr>
<th>Model</th>
<th>Graph</th>
<th>Kernel</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>XLA/HLO</td>
<td>CUDA</td>
<td>GPU</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>TVM/Relay</td>
<td>HIP</td>
<td>CPU</td>
</tr>
<tr>
<td>JAX</td>
<td>PyTorch/fx</td>
<td>OpenCL</td>
<td>FPGA</td>
</tr>
</tbody>
</table>
Transform DNNs to Low Level Code

```c
__global__
void mm(float *a, float *b, float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Graph</th>
<th>Kernel</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>XLA/HLO</td>
<td>CUDA</td>
<td>GPU</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>TVM/Relay</td>
<td>HIP</td>
<td>CPU</td>
</tr>
<tr>
<td>JAX</td>
<td>PyTorch/fx</td>
<td>OpenCL</td>
<td>FPGA</td>
</tr>
</tbody>
</table>
# Transform DNNs to Low Level Code

<table>
<thead>
<tr>
<th>Model</th>
<th>Graph</th>
<th>Kernel</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>XLA/HLO</td>
<td>CUDA</td>
<td>GPU</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>TVM/Relay</td>
<td>HIP</td>
<td>CPU</td>
</tr>
<tr>
<td>JAX</td>
<td>PyTorch/fx</td>
<td>OpenCL</td>
<td>FPGA</td>
</tr>
</tbody>
</table>
Transform DNNs to Low Level Code

Model | Graph | Kernel | Device
--- | --- | --- | ---
PyTorch | XLA/HLO | CUDA | GPU
TensorFlow | TVM/Relay | HIP | CPU
JAX | PyTorch/fx | OpenCL | FPGA

__global__
void mm(float *a, float *b, float *c) {
    float *a_tile;
    float *b_tile;
    ...
}

```c
__global__
void mm(float *a, float *b, float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```
PYTORCH 2.0
Features

- **TorchDynamo**
  - Captures PyTorch programs safely using Python Frame Evaluation Hooks

- **AOTAutograd**
  - Generating ahead-of-time backward traces

- **PrimTorch**
  - Canonicalizes ~2000+ PyTorch operators down to a closed set of ~250 primitive operators

- **TorchInductor**
  - Deep learning compiler that generates fast code for multiple accelerators and backends
  - For NVIDIA and AMD GPUs, it uses OpenAI Triton as a key building block
Overview

PT2 for Backend Integration

Frontend
- User Model Script
- Dynamo
- Legacy FX Tracer
  - Migrate
- FX Graph in Torch IR
- AOTAutograd

Backend
- FX Graph in Aten/Prims IR
- Integration Interface

Codegen Backends
- Inductor
- nvFuser
- AITemplate
- Others
- Inductor Loop-level IR
- Triton
- C++/OpenMP
- Others

Legend:
- Mid-layer Integration at Aten FX graph layer
- Low-level Integration with Inductor
Graph Tracers Prior to PyTorch 2.0

- `torch.jit.trace`
  - Tracing at C++ level
    - Does not capture any control flow done in Python

- `torch.jit.script`
  - Static Python AST analysis (i.e., visit_<syntax_name>)
    - An unimplemented component of Python makes the entire program unfit for capture

- Lazy tensors
  - Hashing the graph to avoid recompilation
    - Recompilation if any part of the graph is changed

- `torch.fx.symbolic_trace`
  - Tracing at python level using proxy objects
    - Silently incorrect results due to random functions and global variables
PEP 523 - Adding a frame evaluation API to CPython

- Expand CPython’s C API to allow a per-interpreter function to handle the evaluation of frame
  - seval_frame = _PyEval_EvalFrameDefault by default

```c
typedef struct {
    ...
    _PyFrameEvalFunction eval_frame;
} PyInterpreterState;

PyObject *
PyEval_EvalFrameEx(PyFrameObject *frame, int throwflag)
{
    PyThreadState *tstate = PyThreadState_GET();
    return tstate->interp->eval_frame(frame, throwflag);
}
```
TorchDynamo

Default Python Behavior

```
foo(...)  
PyFrameObject  PyCodeObject

_PyEval_EvalFrameDefault()  
```

TorchDynamo Behavior

```
foo(...)  
PyFrameObject  PyCodeObject

Guards

Transformed PyCodeObject (non-torch.* bits)

FX Graphs (torch.* bits)

User-defined Compiler

Cached

Compiled Function

_PyEval_EvalFrameDefault()
```
TorchInductor

- The default “user-defined” compiler
  - Implemented in Python

- Decomposition
  - \( \log_2 \rightarrow \log \times \log_2\text{scale} \)

- Lowering
  - Use Python functions to define the bodies of loops

- Scheduling
  - Determine which kernels should be fused to achieve the best performance

- Code generation
  - GPU
    - IR->Triton Python code
  - CPU
    - IR->OpenMP/C++
Usage

- `torch.compile`
  - model=None
    - required
  - fullgraph=False
  - dynamic=False
  - backend='inductor'
  - mode=None
    - reduce-overhead
    - max-autotune
  - options=None
  - disable=False

- Function
  
  compiled_module = `torch.compile(module, ...)`

- Decorator
  
  ```python
  @torch.compile(fullgraph=True)
  def foo(x):
    return torch.sin(x) + torch.cos(x)
  ```
Example

```python
import torch._dynamo
import torch

def f(x):
    return torch.sin(x)**2 + torch.cos(x)**2

x = torch.ones(256, requires_grad=True, device='cuda')
y = torch.ones_like(x)

torch._dynamo.reset()
compiled_f = torch.compile(f)
out = torch.nn.functional.mse_loss(compiled_f(x), y).backward()
```

Example - Prims IR

class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[256]):
        # File: /home/keren/code/test.py:7, code: return torch.sin(x)**2 + torch.cos(x)**2
        sin: f32[256] = torch.ops.aten.sin.default(primals_1)
        pow_1: f32[256] = torch.ops.aten.pow.Tensor_Scalar(sin, 2)
        cos: f32[256] = torch.ops.aten.cos.default(primals_1)
        pow_2: f32[256] = torch.ops.aten.pow.Tensor_Scalar(cos, 2)
        add: f32[256] = torch.ops.aten.add.Tensor(pow_1, pow_2); pow_1 = pow_2 = None
        return [add, sin, primals_1, cos]
Example - Triton Code

```python
@pointwise(size_hints=[256], filename=__file__, meta={'signature': {0: '*fp32', 1: '*fp32', 2: 'i32'}, 'device': 0,
'constants': {}, 'mutated_arg_names': [], 'configs': [instance_descriptor(divisible_by_16=(0, 1, 2),
equal_to_1=())])}
@triton.jit
def triton_(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xnumel = 256
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tl.sin(tmp0)
    tmp2 = tmp1 * tmp1
    tmp3 = tl.cos(tmp0)
    tmp4 = tmp3 * tmp3
    tmp5 = tmp2 + tmp4
    tl.store(out_ptr0 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp5, xmask)
```
Benefits

• Robustness
  • Capture a single graph for most models
  • Fallback to partial graphs is needed

• Speed
  • ~1.5x faster than the eager mode
Handwritten Low Level Code VS Automated Generation

- **Low flexibility**
  - Fine-tune for every shape/data type/algorithm
  - Employ assembly instructions
  - ...

- **High performance**
  - Apply sophisticated instruction/operator scheduling
  - Simplify code
  - ...

- **High flexibility**
  - Build upon existing operators
  - No need to recompile
  - ...

- **Low performance**
  - Not fine-tuned for specific shapes
  - Intermediate memory movement
  - ...
Triton is a Python-Like Language

- PyTorch compatible
  - Inputs can be PyTorch tensors or custom data-structures (e.g., tensors of pointers)
- Python syntax
  - All standard python control flow structure (for/if/while/return) are supported
  - Python code is lowered to Triton IR
The Programming Language Design Triangle

- Triton focuses on usability and performance
  - The language features supported by triton is a subset of Python
    - No dict
    - No meta-programming
    - No slicing
    - No indexing
    - ...

Expressiveness | Usability | Performance
CUDA Terminologies

• Parallelism
  • Grid
    • One for each kernel (Pre-Hopper)
  • Block/Warp/Thread

• Memory
  • Global
    • Visible to all threads
  • Shared
    • Private to each block
  • Local
    • Private to each thread
# CUDA vs Triton

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>Triton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Global/Shared/Local</td>
<td>Automatic</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Threads/Blocks/Warps</td>
<td>Mostly Blocks</td>
</tr>
<tr>
<td>Tensor Core</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>Vectorization</td>
<td>.8/.16/.32/.64/.128</td>
<td>Automatic</td>
</tr>
<tr>
<td>Async SIMT</td>
<td>Support</td>
<td>Limited</td>
</tr>
<tr>
<td>Device Function</td>
<td>Support</td>
<td>Support</td>
</tr>
</tbody>
</table>

Using Triton, you only need to know that a program is divided into multiple blocks.
Vector Addition (Single Block)

\[ Z[:,] = X[:,] + Y[:,] \]

→ Without boundary check

```python
import triton.language as tl
import triton

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
```
Vector Addition (Boundary Check)

\[ Z[:] = X[:] + Y[:] \]

\rightarrow \text{With boundary check}

```python
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)
    offsets += tl.program_id(0) * 1024
    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
```

```bash
N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
```
Vector Addition (Autotune)

\[ Z[:] = X[:] + Y[:] \]

- Each block computes TILE elements
- @triton.autotune
- Select the best config based on the execution time
- We don’t want to build complex autotune policies into Triton

```python
@triton.autotune(configs=[
    triton.Config(TILE=128),
    triton.Config(TILE=256)
])
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, TILE)
    offsets += tl.program_id(0)*TILE
    # create TILE pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load TILE elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back TILE elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args["TILE"]), )
_add[grid](z, x, y, N)
```
Triton JIT-Compilation Workflow

Hashing

- Tensors
- Constants
- Scalars

Triton Kernel

Optimization & Analysis

- MLIR Dialects

PTX
Optimization Passes

- **MLIR general optimizations**
  - CSE, DCE, Inlining, ...

- **TritonGPU specific optimizations**
  - Pipeline
  - Prefetch
  - Matmul accelerate
  - Coalesce
  - Remove layout

- **TritonNVIDIA GPU specific optimizations**
  - TMA Materialization
  - TMA Multicast
  - Async Dot
  - Warp Specialization
Layout Encoding in TritonGPU

- A specification that maps data distribution to threads to better utilize the underlying hardware
  - Suppose we have a 2x2 tensor and 8 threads
    - Layout(0, 0) = {0, 4}
    - Layout(0, 1) = {1, 5}
    - Layout(1, 0) = {2, 6}
    - Layout(1, 1) = {3, 7}
  - It means that
    - data(0, 0) is stored on thread 0 and thread 4
    - data(0, 1) is stored on thread 1 and thread 5
    - data(1, 0) is stored on thread 2 and thread 6
    - data(1, 1) is stored on thread 3 and thread 7
Blocked Layout

- The most basic layout in Triton
- Assign a default layout initially
- Optimize the layout based on global memory load/store ops
- A 2d blocked layout example
  - sizePerThread = \{2, 2\}
  - threadsPerWarp = \{8, 4\}
  - warpsPerCTA = \{1, 2\}
  - CTAsPerCGA = \{1, 1\}
  - order = \{1, 0\}
    - Row major
Shared Layout

• Specify how data is stored on shared memory
  • Use 2D-swizzling or padding to avoid bank conflicts

• Triton doesn’t manage shared memory explicitly
  • Shared memory is only used when involving data exchange across threads
    • Convert from one layout to another
Dot Operand Layout

- **mma.m16n8k16**
  - \( A [m,k] \times B [k,n] + C [m, n] = D [m, n] \)

![Diagram of Dot Operand Layout]

- **A**: fp16
- **B**: fp16
MMA Layout

- **mma.m16n8k16**
  - $A[m,k] \times B[k,n] + C[m,n] = D[m,n]$
Analysis Passes

- Shared memory
  - Alias
  - Liveness
  - Barrier
- Pointer alignment
  - Axisinfo
- Call graph
  - “noinline” functions
Ecosystem

- Deepspeed
- Tinygrad
- Kernl.ai
- PyTorch
- JAX
- OpenXLA
- IREE

Tools:
- Runtime
- Debugger
- Profiler
- Language
- Backends
Dev Time VS Performance

Dev Time

Performance

- TensorFlow
- JAX
- PyTorch V1
- PyTorch V2
- TVM
- CUTLASS
- CUDA
- SASS
- Triton
Triton Performance

- It takes <25 lines of code to write a Triton kernel on par with cuBLAS
- Arbitrary ops can be “fused” before/after the GEMM while the data is still on-chip
  - leading to large speedups over PyTorch/cublas