

Deep Learning Implementation - Lab 6 : Accelerate Custom Operator

20230683 박한비, 20230308 최승희

Batch Normalization(BN)은 DL 모델 학습 과정에 필수적인 정규화 기법으로, 각 mini-batch 의 평균과 분산을 정규화함으로써 안정적인 학습을 가능하게 한다.

기존의 BN 연산은 PyTorch 나 CUDA 같은 범용 GPU 연산 라이브러리를 기반으로 구현되어 왔지만, 이는 다양한 모델이나 환경에서 최적화 수준이 제한적이다. 더 높은 효율성을 위한 Custom Operator 중 하나인 Triton 은 Python Interface 를 제공하면서, CUDA 수준의 성능 최적화를 지원하는 고성능 GPU 커널 프로그래밍 언어이다.

본 과제에서는 Triton 을 활용하여 BN 의 Forward/Backward 연산을 직접 구현하고, 이를 기존 CUDA 기반 BN 연산과 성능 비교를 통해 우리가 구현한 custom BN operator 의 효과를 분석하고자 한다.

1. Implementation

1. batchnorm_forward_training_kernel

먼저 `program_id(0)`으로 현재 실행 중인 프로그램이 담당할 채널 인덱스 `c` 를 얻고, 인덱스가 존재하는 채널 개수의 범위를 넘어가면 리턴한다. 채널의 총 element 개수(NHW)를 계산하여 `M` 에 저장하고 계산에 Tiling 을 적용하기 위해 `BLOCK_SIZE` 에 따라 각 타일의 로컬 오프셋을 계산해 `offs` 에 저장한다.

`matrix`(채널의 모든 elements)를 `BLOCK_SIZE` 만큼 이동하면서 연산을 진행하고 각 타일에서 계산된 값의 합/값의 제곱의 합을 누적 연산하기 위해 각각 `sum_x`, `sum_x2` 에 저장한다. 최종 `sum_x`, `sum_x2` 를 이용해 평균 `mean` 과 분산 `var` 를 계산한다. 표준편차의 역수 또한 계산해 `invstd` 에 저장한다. 계산된 해당 채널의 배치 통계를 `store()`로 `t1` 에 저장한다. `load()`로 러닝 통계(EMA)도 가져와 업데이트 후 다시 저장하고, $\gamma(g)$ 와 $\beta(b)$ 값도 가져온다.

다시 채널의 모든 elements 를 Tiling 으로 이동하면서 `mean`, `invstd`, `g`, `b` 를 이용해 batch normalization 식을 적용하여(정규화, 스케일, 시프트 적용) 저장한다.

2. batchnorm_forward_inference_kernel

먼저 `program_id(0)`으로 현재 실행 중인 프로그램이 담당할 채널 인덱스 `c` 를 얻고, 인덱스가 존재하는 채널 개수의 범위를 넘어가면 리턴한다. 채널의 총 element

개수(NHW)를 계산하여 M 에 저장하고 계산에 Tiling 을 적용하기 위해 BLOCK_SIZE 에 따라 각 타일의 로컬 오프셋을 계산해 offs 에 저장한다.

training 에서 저장해놓은 러닝 통계 (mean, var)를 로드해오고 invstd 또한 계산한다.
 $\gamma(g)$ 와 $\beta(b)$ 값도 로드해온 후 matrix(채널의 모든 elements)를 BLOCK_SIZE 만큼 이동하면서 mean, invstd, g, b 를 이용해 y 에 batch normalization 식을 적용하고(정규화, 스케일, 시프트 적용) 저장한다.

3. batchnorm_backward_kernel

먼저 program_id(0)을 통해 현재 스레드가 처리할 채널 인덱스 c 를 결정한다.
 $M=N \times \text{spatial_dim}$ 을 통해 해당 채널의 전체 데이터 개수를 구하고, BLOCK_SIZE 를 이용해 타일 단위 연산을 수행하기 위한 로컬 오프셋 offs 를 정의한다. 첫 번째 단계에서는 각 채널별로 출력 기울기(dy)와 입력(x)을 읽어 정규화된 입력값 $\hat{x} = (x - \text{mean}) * \text{invstd}$ 를 계산한다. 각 타일에서 dy 와 $dy * \hat{x}$ 의 합을 각각 sum_dy 와 sum_dy_xhat 에 누적한다. 모든 타일을 순회한 후 최종적으로 계산된 sum_dy 와 sum_dy_xhat 을 grad_beta_ptr 과 grad_gamma_ptr 에 저장한다.

두 번째 단계에서는 입력값에 대한 기울기(dx)를 계산한다. 이를 위해 앞서 계산된 sum_dy, sum_dy_xhat 값을 사용하여 $dx = (g * \text{invstd} / m) * (m * dy - \text{sum_dy} - \hat{x} * \text{sum_dy_xhat})$ 으로 각 입력값의 dx 를 구한다. 계산된 dx 값을 원래 데이터 타입으로 변환해 grad_input_ptr 에 저장한다.

2. Result

Implementation 을 테스트하기 위해 이미 주어진 test_batchnorm.py 를 활용하였다. test_batchnorm.py 를 실행한 결과 아래와 같은 결과가 출력되는 것을 보아 구현이 성공적으로 이루어졌다는 것을 알 수 있었다.

```
root@C:273908991~/custom_ops_bn_student8 python3 test_batchnorm.py
BatchNorm Custom Operators: Correctness & Performance Tests
=====
CORRECTNESS TESTS
=====
Correctness Test: N=64, C=128, H=56, W=56, Training=True
Triton max diff: 3.81e-06
Triton: ✓ PASSED
=====
Correctness Test: N=64, C=128, H=56, W=56, Training=False
Triton max diff: 4.58e-06
Triton: ✓ PASSED
=====
ALL CORRECTNESS TESTS PASSED
=====
PERFORMANCE BENCHMARKS
=====
Performance Benchmark: N=64, C=128, H=56, W=56, Training=True
1. CUDA Custom Operator (Reference)
  Forward: 1.5534 ms
  Backward: 2.5572 ms
2. Triton Custom Operator (Your Implementation)
  Forward: 0.5867 ms
  Speedup vs CUDA: 2.67x
  Backward: 1.0665 ms
  Speedup vs CUDA: 2.40x
=====
Performance Benchmark: N=64, C=128, H=56, W=56, Training=False
1. CUDA Custom Operator (Reference)
  Forward: 0.3443 ms
  Backward: 2.5885 ms
2. Triton Custom Operator (Your Implementation)
  Forward: 0.4325 ms
  Speedup vs CUDA: 0.80x
  Backward: 0.9692 ms
  Speedup vs CUDA: 2.59x
=====
Tests Complete!
```

1. Correctness

Correctness 테스트는 PyTorch 에서 Batch Normalization 을 제공하는 기존 클래스 torch.nn.BatchNorm2d 와의 비교를 통해 진행된다.

Triton Custom BN Operator 를 활용한 정규화와 torch.nn.BatchNorm2d 를 통한 정규화의 각 output 들의 차이가 설정된 임계값($1e-3$)을 넘지 않는다면 테스트는 통과한다.

```
=====
CORRECTNESS TESTS
=====

Correctness Test: N=64, C=128, H=56, W=56, Training=True
=====

Triton max diff: 3.81e-06
Triton: ✓ PASSED

=====

Correctness Test: N=64, C=128, H=56, W=56, Training=False
=====

Triton max diff: 4.58e-05
Triton: ✓ PASSED

=====
ALL CORRECTNESS TESTS PASSED
=====
```

기존 PyTorch 의 torch.nn.BatchNorm2d 와의 차이가 임계치 미만으로 작은 것을 보아 Triton Custom BN Operator 가 유효한 batch normalization 을 제공한다는 것을 보장한다.

2. Performance

Performance 테스트는 주어진 CUDA Custom BN Operator 과의 성능 비교를 통해 진행된다.

```
=====
PERFORMANCE BENCHMARKS
=====

Performance Benchmark: N=64, C=128, H=56, W=56, Training=True
=====

1. CUDA Custom Operator (Reference)
Forward: 1.1534 ms
Backward: 2.5572 ms

2. Triton Custom Operator (Your Implementation)
Forward: 0.5867 ms
Speedup vs CUDA: 1.97x
Backward: 1.0665 ms
Speedup vs CUDA: 2.40x

=====

Performance Benchmark: N=64, C=128, H=56, W=56, Training=False
=====

1. CUDA Custom Operator (Reference)
Forward: 0.3443 ms
Backward: 2.5085 ms

2. Triton Custom Operator (Your Implementation)
Forward: 0.4326 ms
Speedup vs CUDA: 0.80x
Backward: 0.9692 ms
Speedup vs CUDA: 2.59x
```

각 단계에서 걸린 시간을 성능이라고 나타내었으며, 테스트 결과 및 결과에 대한 분석은 아래와 같다.

- Training 의 forward pass 는 CUDA : 1.1534 ms, Triton : 0.5867 ms 로 Triton 이 1.97 배로 빠르다. 이는 Triton 은 평균·분산 계산 및 정규화를 한 번에 묶어 처리해서 메모리를 적게 오가는 반면 CUDA 는 여러 번 나눠 처리해 왕복·동기화가 Triton 에 비해 많아져 소모 시간이 늘어나기 때문이다.
- Training 의 backward pass 는 CUDA : 2.5572 ms, Triton : 1.0665 ms 로 Triton 이 2.40 배로 빠르다. 이는 값들을 합치는 reduction 이 Triton 은 타일 안에서 먼저 크게 모은 뒤 마지막에만 조금 합쳐(원자연산 최소화) 빨라지고, CUDA 는 자주 원자연산을 써 충돌·대기가 커져 느리기 때문이다.
- Inference 의 forward pass 는 CUDA : 0.3443 ms, Triton : 0.4326 ms 로 Triton 이 0.80 배로 느리다. 이는 이미 저장된 평균·분산을 불러와야 하는데, CUDA 가 더 강한 벡터화가 된 상태이기 때문에 대역폭을 끝까지 뽑아 이런 단순 메모리 작업에서 유리하기 때문이다. Triton 은 타일링/마스킹 오버헤드가 있을 수 있다.
- Inference 의 backward pass 는 CUDA : 2.5085 ms, Triton : 0.9692 ms 로 Triton 이 2.59 배로 빠르다. 이는 Training 의 backward pass 와 비슷하게 reduction 이 많기 때문이다.