Master Degree in Artificial Intelligence

InfoNCE Loss CUDA Implementation

Parallel Programming for Machine Learning 2025

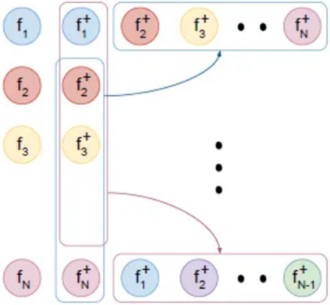
Gianni Moretti

Introduction

What is InfoNCE Loss?

The InfoNCE (Information Noise-Contrastive Estimation) loss works by comparing each sample in a batch with its positive pair (a different view or augmentation of the same data point) and a set of negative samples (other data points in the batch).

- Cornerstone of contrastive learning methods (SimCLR, MoCo)
- Maximizes mutual information between positive pairs
- Pushes negative examples apart in representation space



Mathematical Foundation

For a batch of $2B$ samples with positive pairs, InfoNCE is defined as:

$$\mathcal{L}_{\text{InfoNCE}} = -\frac{1}{2B}\sum_{i=1}^{2B}\log\frac{\exp(\text{sim}(z_i, z_{p(i)})/\tau)}{\sum_{j=1,j\neq i}^{2B}\exp(\text{sim}(z_i, z_j)/\tau)} \tag{1}$$

Key Components:

- z_i: L2-normalized feature vector
- $p(i)$: positive pair index for sample i
- $\text{sim}(a, b) = a \cdot b$: cosine similarity
- τ: temperature parameter controlling concentration

Classic PyTorch Implementation

```
def info_nce_loss(features, temperature=0.5):
    device = features.device
    batch_size = features.shape[0] // 2
    # Compute similarity matrix (dot product)
    sim_matrix = torch.matmul(features, features.T)  # (2B, 2B)
    # Remove self-similarity by masking the diagonal
    mask = torch.eye(2 * batch_size, dtype=torch.bool, device=device)
    sim_matrix = sim_matrix.masked_fill(mask, float('-inf'))
    # Labels: positive pair for i is at (i + B) % (2B)
    labels = torch.arange(batch_size, device=device)
    labels = torch.cat([labels + batch_size, labels])
    # Scale similarities by temperature
    sim_matrix /= temperature
    # Apply cross entropy loss
    loss = F.cross_entropy(sim_matrix, labels)
    return loss
```

Project Goals

- Implement specialized CUDA InfoNCE Loss computation
- Create CuBlaze package for Python integration
- Compare custom CUDA vs CUBLAS implementations
- Demonstrate Python-CUDA-PyTorch integration
- Educational exploration of GPU programming techniques
- Achieve numerical accuracy while maintaining performance

Implementation Overview

Technologies Used:

- **Language**: CUDA C++ for GPU kernels
- **Integration**: PyBind11 for Python-C++ binding
- **Framework**: PyTorch with autograd support
- **Libraries**: CUBLAS for optimized matrix operations
- **Build System**: PyTorch C++ extensions

Mathematical Foundation

InfoNCE Loss gradient derivation problem

- PyTorch’s autograd system cannot automatically compute gradients when custom CUDA kernels are used.
- This is because custom CUDA code interrupts the standard computation graph, breaking the autograd chain.
- As a result, PyTorch is unable to trace operations through C++/CUDA extensions.
- To ensure correct gradient computation and maintain compatibility with PyTorch, we must manually implement the backward pass.

InfoNCE Loss Derivation

Forward Pass - Loss Computation:

Given normalized feature vectors $Z = \{Z_1, Z_2, \ldots, Z_N\}$ with $N = 2B$:

$$L_{ij} = \frac{1}{\tau} Z_i \cdot Z_j \tag{2}$$

$$P_{ij} = \frac{\exp(L_{ij})}{\sum_{k=1}^{N} \exp(L_{ik})} \tag{3}$$

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \log P_{i,p(i)} \tag{4}$$

where $p(i) = (i + B) \bmod N$ identifies positive pairs.

InfoNCE Loss Derivation

Numerical Stability:

To ensure numerical stability during the computation of the softmax, a *max reduction* is applied to each row of the similarity matrix L_{ij}. Specifically, for each i, the maximum value $\max_j L_{ij}$ is subtracted from all logits before applying the exponential:

$$P_{ij} = \frac{\exp(L_{ij} - \max_j L_{ij})}{\sum_{k=1}^{N} \exp(L_{ik} - \max_j L_{ij})}$$

This prevents overflow and improves the stability of the softmax computation.

Gradient Computation (Backward Pass)

Chain Rule Application:

For loss $\ell_i = -\log P_{i,p(i)}$, the gradient with respect to logits is:

$$\frac{\partial \ell_i}{\partial L_{ij}} = P_{ij} - \mathbb{1}_{j=p(i)} \tag{5}$$

Final Gradient Formula:

$$\frac{\partial \mathcal{L}}{\partial Z_k} = \frac{1}{N\tau}\left(\sum_j G_{kj} Z_j + \sum_i G_{ik} Z_i\right) \tag{6}$$

where $G_{ij} = P_{ij} - \mathbb{1}_{j=p(i)}$.

Gradient Computation (Backward Pass)

In matrix form:

$$\nabla_Z \mathcal{L} = \frac{1}{N\tau}(G + G^T)Z \tag{7}$$

Implementation Details

CuBlaze Package Architecture

Package Structure

Complete Python-CUDA integration solution

```
cublaze/
|-- __init__.py
|-- infonce.py
|-- infonce_cuda.so
+-- cuda/
    |-- infonce_cuda.cu
    +-- infonce_cuda_wrapp.cpp
```

- PyTorch autograd compatibility
- GPU-optimized kernels

Python-CUDA Integration

PyBind11 Wrapper

Type-safe binding between PyTorch and CUDA

```cpp
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>
torch::Tensor infonce_cuda_forward(
    torch::Tensor features,
    float temperature,
    bool use_cublas);
torch::Tensor infonce_cuda_backward(
    torch::Tensor features,
    torch::Tensor similarity_matrix,
    torch::Tensor labels,
    // ... other parameters
);
```

PyTorch Integration

InfoNCEFunction class for autograd support

```python
class InfoNCEFunction(Function):
    @staticmethod
    def forward(ctx, features, temperature, use_cublas):
        # Save parameters for backward pass
        ctx.temperature = temperature
        ctx.use_cublas = use_cublas
        # CUDA forward computation
        loss, similarity_matrix, labels, max_vals, sum_exps =
            infonce_cuda.infonce_forward(features, temperature, use_cublas)
        # Save tensors for backward pass
        ctx.save_for_backward(features, similarity_matrix, labels, max_vals, sum_exps)
        return loss
```

PyTorch Integration

backward pass implementation

```python
    @staticmethod
    def backward(ctx, grad_output):
        # Retrieve saved tensors
        features, similarity_matrix, labels, max_vals, sum_exps = \
            ctx.saved_tensors

        # CUDA backward computation
        grad_features = infonce_cuda.infonce_backward(
            features, similarity_matrix, labels,
            max_vals, sum_exps, ctx.temperature,
            grad_output, ctx.use_cublas)

        return grad_features, None, None
```

CUDA Implementation Strategies

Two Computational Approaches

Custom CUDA vs CUBLAS Implementation

Custom CUDA (use_cublas=False)

- Hand-written GPU kernels
- Specialized for InfoNCE patterns
- Direct similarity computation
- Educational value for GPU programming

CUBLAS Library (use_cublas=True)

- Highly optimized matrix operations
- Hardware-specific optimizations
- Production-ready performance

Dynamic CUDA Block Size Selection

Block size selection is dynamic and adapts to the matrix size being processed.

- The CUDA kernel launch configuration (block size and grid size) is chosen at runtime based on the problem dimensions.
- This strategy ensures efficient GPU utilization and performance portability across different batch sizes and feature dimensions.

```
__host__ int calculate_optimal_block_size_1d(int total_elements)
__host__ dim3 calculate_optimal_block_size_2d(int dim1,int dim2)
__host__ int calculate_grid_size_1d(int total_elements,int block_size)
__host__ dim3 calculate_grid_size_2d(int dim1,int dim2,dim3 block_size)
```

Forward-Pass Caching for Efficient Backward

The forward pass computes and stores:

- The similarity matrix
- The maximum value of each row of the similarity matrix (for numerical stability)
- The sum of exponentials for each row (softmax denominator)
- The positive pair labels for each sample

Forward-Pass Caching for Efficient Backward

Why Cache These Values?

- These values are saved in memory and passed to the backward CUDA kernel.
- This avoids redundant computation and ensures that the backward pass is both correct and efficient.
- The approach is especially important for large batches, where recomputing softmax or labels would be costly.
- This design is compatible with PyTorch’s autograd, as all necessary tensors are saved in the context object.

Build System and Compilation

Automated Build Process

Complete development workflow automation

```bash
# Automated build and test script
./build_and_test.sh
```

System Requirements:

- CUDA Toolkit 11.0+
- PyTorch with CUDA support
- PyBind11 development headers
- GCC with C++14 support

CUDA Kernel Implementation

Forward Pass CUDA Kernel

infonce_forward_kernel

```
__global__ void infonce_forward_kernel(const float* similarity_matrix,
    const int* labels,
                                        float* loss, float* max_vals,
    float* sum_exps,
                                        int batch_size) {
    extern __shared__ float shared_loss[];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int tid = threadIdx.x;

    // Initialize shared memory
    shared_loss[tid] = 0.0f;

```

CUDA Kernel Implementation

```
    if (i < batch_size) {
        //1. Find max value and sum of exponentials for this row
        //.....
        //2. Compute sum of exponentials
        //.....

        //Save max_val and sum_exp for backward pass
        max_vals[i] = max_val;
        sum_exps[i] = sum_exp;

        // Calculate loss for this row
        int positive_idx = labels[i];
        float positive_logit = similarity_matrix[i * batch_size + positive_idx];
        float log_prob = (positive_logit - max_val) - logf(sum_exp);

```

CUDA Kernel Implementation

```
        // Store local loss in shared memory (normalized by batch size)
        shared_loss[tid] = -log_prob / batch_size;
    }

    __syncthreads();

    if (tid == 0) {
        float block_loss = 0.0f;
        for (int i = 0; i < blockDim.x; i++) {
            block_loss += shared_loss[i];
        }
        atomicAdd(loss, block_loss);
    }
}
```

Backward Pass Implementation

CUDA Kernel Implementation

Manual gradient computation for autograd compatibility

```
__global__ void infonce_backward_kernel(const float* similarity_matrix, const int* labels,
                                        const float* max_vals, const float* sum_exps,
                                        float* grad_matrix, int batch_size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int total_elements = batch_size * batch_size;
    if (tid < total_elements) {
        int i = tid / batch_size;
        int j = tid % batch_size;

        // Use pre-computed values from forward pass
        float max_val = max_vals[i];
        float sum_exp = sum_exps[i];
```

CUDA Kernel Implementation

```
        // Calculate gradient: P_ij - 1_{j=p(i)}
        int positive_idx = labels[i];
        float val = similarity_matrix[tid];

        if (val != -INFINITY) {
            float prob = expf(val - max_val) / sum_exp;
            float grad_val = prob - (j == positive_idx ? 1.0f : 0.0f);
            grad_matrix[tid] = grad_val / batch_size;
        } else {
            grad_matrix[tid] = 0.0f;
        }
    }
}
```

Usage Example

Simple integration into existing PyTorch code

```python
import torch
from cublaze import InfoNCELoss
# Initialize the loss function
infonce_loss = InfoNCELoss(temperature=0.5, use_cublas=False)
# Prepare normalized features (2B, D)
# Each sample i has positive pair at (i + B) % 2B
features = torch.randn(64, 256).cuda()
features = F.normalize(features, dim=1)  # L2 normalization required
# Compute loss with gradients
loss = infonce_loss(features)
loss.backward()
# Gradients are automatically computed
print(f"Loss: {loss.item():.4f}")
print(f"Feature gradients shape: {features.grad.shape}")
```

Experimental Setup

Experimental Setup

Testing Environment:

- GPU: NVIDIA with CUDA support
- Multiple batch sizes: 16 to 8192 samples
- Feature dimensions: 256
- Temperature values: 0.5
- Multiple runs for statistical accuracy

Test Configurations:

- **Custom CUDA**: use_cublas=False
- **CUBLAS Library**: use_cublas=True
- **PyTorch Baseline**: Native implementation

Experimental Setup

Primary Measurements:

- Execution time per forward/backward pass
- Numerical accuracy vs PyTorch reference
- Scalability with batch size

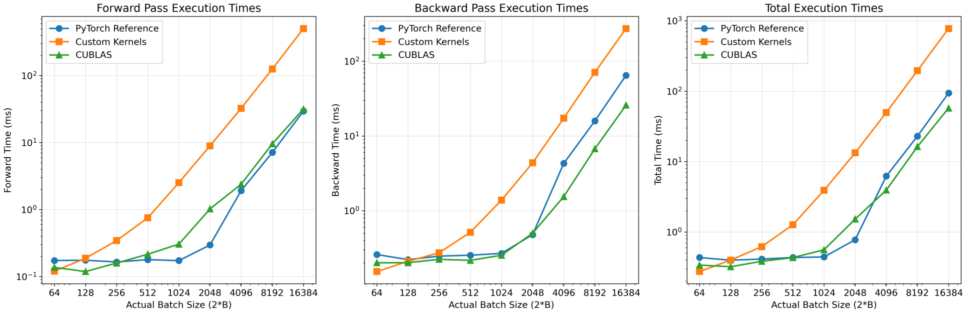
Accuracy Metrics:

- Loss computation error (target: $< 10^{-5}$)
- Gradient computation error (target: $< 10^{-4}$)
- Cross-validation with PyTorch implementation

Performance Results

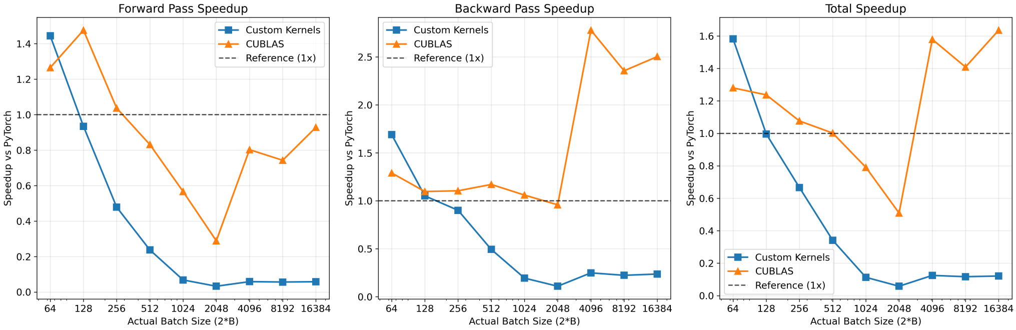
Execution Time Analysis

Batch Size Impact on Execution Time:



Execution Time Analysis

Batch Size Impact on Speedup:

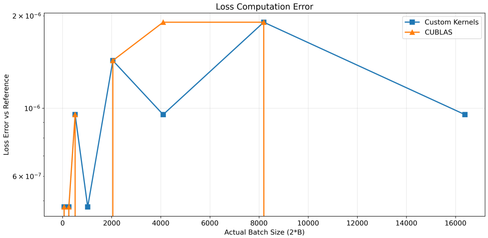


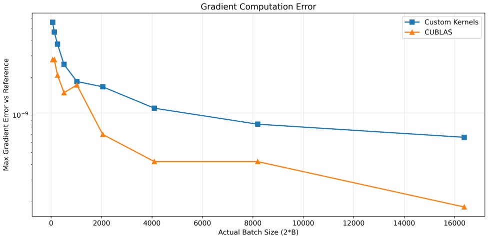
Execution Time Analysis

Key Findings:

- Current CUDA implementation shows higher execution times than PyTorch
- PyTorch leverages highly optimized libraries (cuBLAS, cuDNN)
- Custom kernels face overhead from memory transfers and kernel launches
- Optimization opportunities exist for specialized implementations (chacing, kernel fusion)

Numerical Accuracy Results





Numerical Accuracy Results

Accuracy Achievements:

- Loss computation errors consistently below 10^{-5}
- Gradient errors maintained below 10^{-4}
- Excellent numerical stability across all batch sizes
- Validates correctness of manual backward implementation

Conclusions and Future Work

Achieved Objectives:

- ✓ Created complete CuBlaze package
- ✓ Demonstrated Python-CUDA-PyTorch integration
- ✓ Maintained numerical accuracy (errors $< 10^{-5}$ loss, $< 10^{-4}$ gradients)
- ✓ Implemented custom autograd functionality
- ✓ Provided foundation for specialized InfoNCE implementations

Conclusions and Future Work

Future Optimization Opportunities:

- Kernel fusion for reduced memory transfers
- Shared memory utilization optimization
- Advanced memory access pattern optimization (warp-level optimizations)

Thanks for your attention

Gianni Moretti