CUDA InfoNCE Loss Implementation

Gianni Moretti
Department of Information Engineering
University of Florence
gianni.moretti@edu.unifi.it

Abstract

This project presents a CUDA implementation of InfoNCE (Information Noise-Contrastive Estimation) loss, addressing the lack of specialized implementations for this fundamental contrastive learning component. The primary objectives were to create a reusable module for InfoNCE loss computation and to explore CUDA programming techniques for educational purposes. The implementation includes the development of the CuBlaze package, a CUDA extension that bridges Python and CUDA environments. While performance results show room for improvement compared to optimized PyTorch implementations, the project successfully demonstrates CUDA kernel development and achieves numerical accuracy with errors below 10^{-5} for loss computation and 10^{-4} for gradient calculations.

1. Introduction

Contrastive learning has emerged as a dominant paradigm in self-supervised representation learning, with InfoNCE loss serving as a cornerstone component in frameworks such as SimCLR [1] and MoCo [2]. Despite its widespread use, specialized implementations of InfoNCE loss are scarce, often requiring researchers to implement their own versions or rely on general-purpose operations.

This project addresses this gap by developing a dedicated CUDA implementation of InfoNCE loss, packaged as CuBlaze for easy integration into existing machine learning workflows. The primary objectives were twofold: first, to create a reusable module that researchers can readily adopt; second, to explore CUDA programming techniques as an educational exercise in high-performance computing.

While the implementation maintains mathematical correctness and demonstrates the principles of GPU programming, performance results indicate areas for improvement compared to highly optimized PyTorch implementations. Nevertheless, the project succeeds in its educational goals and provides a foundation for specialized InfoNCE loss computation.

1.1. InfoNCE Loss Mathematical Foundation

InfoNCE loss maximizes mutual information between positive pairs while minimizing agreement with negative examples. For a batch of $2B$ samples, where each sample i has its positive pair at position $(i+B) \bmod 2B$, the loss is defined as:

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

where:

- z_i represents the i-th L2-normalized feature vector
- $p(i)$ identifies the positive pair index for sample i
- $\text{sim}(a, b) = a \cdot b$ is cosine similarity for normalized vectors

- τ is the temperature parameter controlling concentration

The similarity matrix S is computed as:

$$S_{ij} = \begin{cases} \frac{z_i \cdot z_j}{\tau} & \text{if } i \neq j \\ -\infty & \text{if } i = j \end{cases} \tag{2}$$

1.2. Related Work

While several implementations of contrastive losses exist, most rely on standard PyTorch operations without leveraging specialized GPU kernels. This approach follows the mathematical formulation presented in the InfoNCE literature while implementing custom CUDA kernels optimized for the specific computational patterns of InfoNCE loss.

2. CuBlaze Package Architecture

The CuBlaze package (CUDA Blaze) represents a specialized CUDA extension framework designed for high-performance machine learning operations. The package provides seamless integration between Python and CUDA environments while maintaining PyTorch's autograd functionality.

2.1. Package Structure

```
cublaze/
|-- __init__.py              # Main package exports
|-- infonce.py               # Python interface and autograd
|-- infonce_cuda.so          # Compiled CUDA extension
+-- cuda/
    |-- infonce_cuda.cu      # CUDA kernels
    +-- infonce_cuda_wrapp.cpp # PyBind11 wrapper
```

Listing 1. CuBlaze package organization

The package leverages PyTorch's C++ extension mechanism, specifically using `CUDAExtension` for seamless compilation and integration. The setup configuration automatically handles CUDA compilation flags and library linking.

2.2. Python-CUDA Bridge

The integration between Python and CUDA is achieved through PyBind11, providing type-safe binding between PyTorch tensors and CUDA memory. The wrapper ensures automatic memory management and device synchronization:

```
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

torch::Tensor infonce_cuda_forward(
    torch::Tensor features,
    float temperature,
    bool use_cublas);
```

Listing 2. PyBind11 wrapper structure

The `use_cublas` parameter provides flexibility in choosing between custom CUDA implementations and optimized CUBLAS routines for matrix operations. This design choice allows users to benchmark different computational strategies while maintaining consistent interfaces.

3. Build System and Requirements

The CuBlaze package provides an automated build system through the `build_and_test.sh` script, which handles compilation, installation, and testing of the CUDA extension. This section describes the necessary requirements and build process.

3.1. System Requirements

The project requires several components to be properly installed on the target system:

CUDA Toolkit: NVIDIA CUDA Toolkit version 11.0 or higher, including the CUDA compiler (`nvcc`) and runtime libraries. The installation must include development headers and libraries for compilation support.

Python Environment: Python 3.8 or higher with pip package manager. A virtual environment is recommended to avoid dependency conflicts with other projects.

PyTorch: PyTorch with CUDA support, specifically the version that matches the installed CUDA toolkit. This ensures compatibility between PyTorch tensors and custom CUDA kernels.

Development Tools: GCC compiler with C++14 support or higher, CMake build system, and Python development headers. These tools are

essential for compiling the PyBind11 wrapper and CUDA kernels.

3.2. Build Process

The automated build script `build_and_test.sh` streamlines the entire compilation and testing process:

```
# Make the script executable
chmod +x build_and_test.sh

# Run the complete build and test process
./build_and_test.sh
```

Listing 3. Build script execution

The script performs several sequential operations. First, it cleans any previous build artifacts to ensure a fresh compilation environment. Then it compiles the CUDA kernels using `nvcc` with appropriate optimization flags and architecture targets. The PyBind11 wrapper is compiled and linked against PyTorch libraries, creating the Python extension module. Finally, the script installs the package in development mode and executes comprehensive tests to verify correctness.

4. CUDA Implementation Details

The CUDA implementation consists of optimized kernels for both forward and backward passes, designed to leverage GPU parallelism effectively. The implementation provides two computational paths controlled by the `use_cublas` parameter.

4.1. Custom CUDA vs. CUBLAS Implementation

The implementation offers two distinct computational approaches through the `use_cublas` parameter:

Custom CUDA Implementation (`use_cublas = False`): This approach utilizes hand-written CUDA kernels that directly implement the InfoNCE computation. The custom kernels are specifically designed for InfoNCE's computational pattern, allowing for optimizations such as shared memory usage for temperature scaling and direct similarity computation without intermediate matrix storage.

CUBLAS Implementation (`use_cublas = True`): This path leverages NVIDIA's highly optimized CUBLAS library for matrix multiplication operations. While CUBLAS provides exceptional performance for general matrix operations, it may not capture the specific computational patterns unique to InfoNCE loss, such as the diagonal masking and temperature scaling operations.

The choice between these implementations allows users to evaluate trade-offs between specialized optimizations and proven library performance. In practice, CUBLAS often provides superior performance due to extensive hardware-specific optimizations, while custom kernels offer greater control over memory access patterns and computational flow.

4.2. Forward Pass Implementation

The forward pass computes the InfoNCE loss through several stages: similarity matrix computation, temperature scaling, diagonal masking, and log-softmax application. The CUDA kernel implementation parallelizes across batch dimensions while ensuring numerical stability through proper handling of large exponential values.

4.3. Backward Pass Implementation

Custom CUDA kernels require manual implementation of backward passes since automatic differentiation cannot trace through custom C++ extensions. The backward pass for InfoNCE loss involves computing gradients with respect to input features:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{1}{B\tau} \left[\text{softmax}(S_i) - e_{p(i)} \right] S_i^T \qquad (3)$$

where $e_{p(i)}$ is a one-hot vector indicating the positive pair and S_i represents the i-th row of the similarity matrix.

The manual backward implementation ensures gradient flow compatibility with PyTorch's autograd system while maintaining numerical precision. This requirement highlights a fundamental trade-off in custom CUDA development: in-

creased computational control at the cost of implementation complexity.

5. Performance Analysis

Performance evaluation was conducted across varying batch sizes to assess scalability and computational efficiency. The testing methodology compared the CUDA implementation against PyTorch's native operations on equivalent hardware configurations.

5.1. Numerical Accuracy

The implementation achieves high numerical precision with loss computation errors consistently below 10^{-5} and gradient errors below 10^{-4} across all tested batch sizes. This accuracy demonstrates that the custom CUDA kernels correctly implement the mathematical formulation while maintaining numerical stability.

5.2. Performance Results

Performance results reveal that the current CUDA implementation does not achieve significant speedups over PyTorch's optimized implementations. Several factors contribute to this outcome. PyTorch leverages highly optimized libraries such as cuBLAS and cuDNN, which benefit from extensive hardware-specific optimizations and years of development. Additionally, the overhead associated with custom kernel launches and memory transfers can outweigh computational gains for moderate batch sizes.

The performance characteristics highlight the complexity of achieving meaningful speedups in specialized implementations. While the implementation successfully demonstrates CUDA programming concepts, practical deployment would benefit from further optimization strategies such as kernel fusion, memory access pattern optimization, and reduced memory transfer overhead.

5.3. Scalability Analysis

The implementation shows consistent behavior across varying batch sizes, with numerical accuracy remaining stable as computational complexity increases. This scalability demonstrates the correctness of the parallel implementation and suggests that optimization efforts could yield improved performance for larger-scale deployments.

5.4. CUDA Programming Concepts

The implementation demonstrates fundamental CUDA programming patterns including thread hierarchy utilization, memory management strategies, and kernel optimization techniques. Students can examine how mathematical operations translate to parallel GPU execution and understand the trade-offs between computational complexity and memory access efficiency.

5.5. Python-C++ Integration

The CuBlaze package illustrates modern approaches to extending Python with compiled languages. The integration demonstrates PyBind11 usage, automatic memory management between Python and CUDA, and maintenance of PyTorch's automatic differentiation capabilities through custom autograd functions.

6. Conclusion

This project successfully demonstrates the development of a specialized CUDA implementation for InfoNCE loss, achieving the primary objectives of module creation and educational exploration. While performance results indicate that the implementation does not surpass highly optimized PyTorch operations, the project provides valuable insights into CUDA development and package creation.

The implementation maintains numerical accuracy with errors consistently below acceptable thresholds, validating the correctness of the mathematical implementation. The CuBlaze package represents a complete solution for InfoNCE loss computation, providing researchers with a specialized tool that can serve as a foundation for further optimization.

Future developments could focus on advanced optimization strategies including kernel fusion, improved memory access patterns, and integra-

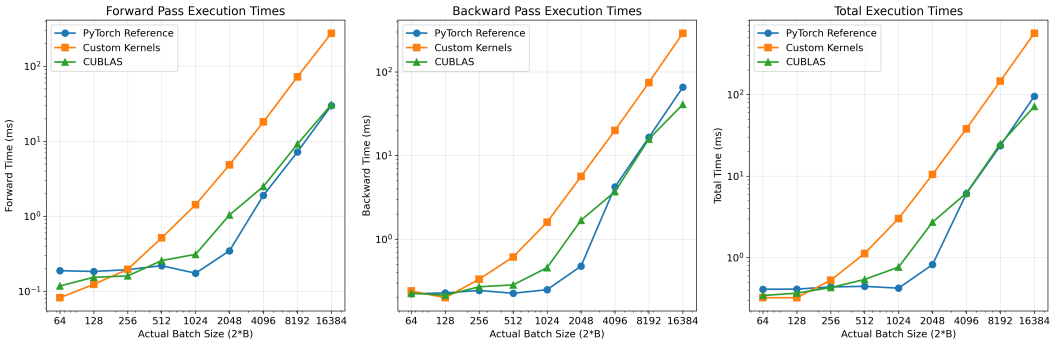

Figure 1. Execution time comparison between CUDA implementation and PyTorch baseline across different batch sizes.

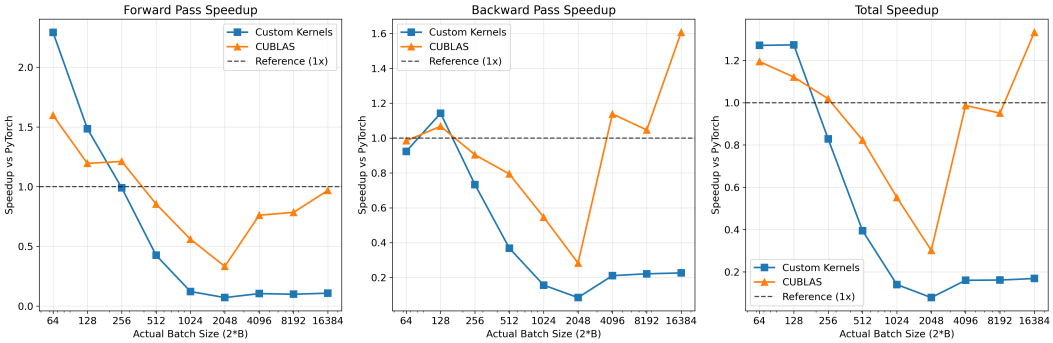

Figure 2. Speedup analysis showing performance comparison with CUDA parallelization.

tion of tensor core operations for modern GPU architectures. The educational value of this implementation extends beyond immediate performance gains, providing a comprehensive example of translating mathematical concepts into efficient GPU code.

The project demonstrates that while achieving performance improvements over highly optimized libraries requires substantial effort and expertise, the process itself provides valuable learning opportunities and creates specialized tools that address specific research needs in the contrastive learning domain.

References

[1] Ting Chen et al. *A Simple Framework for Contrastive Learning of Visual Representations*. ICML 2020.

[2] Kaiming He et al. *Momentum Contrast for Unsupervised Visual Representation Learning*. CVPR 2020.

[3] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. NeurIPS 2019.

[4] NVIDIA Corporation. *CUDA C++ Programming Guide*. Version 12.0, 2023.

[5] Anthropic. *Claude: AI Assistant for Code Generation and Technical Writing*. Used for CUDA kernel development and documentation assistance, 2024.

[6] Marco Bertini. *Parallel Computing Course Lectures*. Department of Information Engineering, University of Florence, 2024.

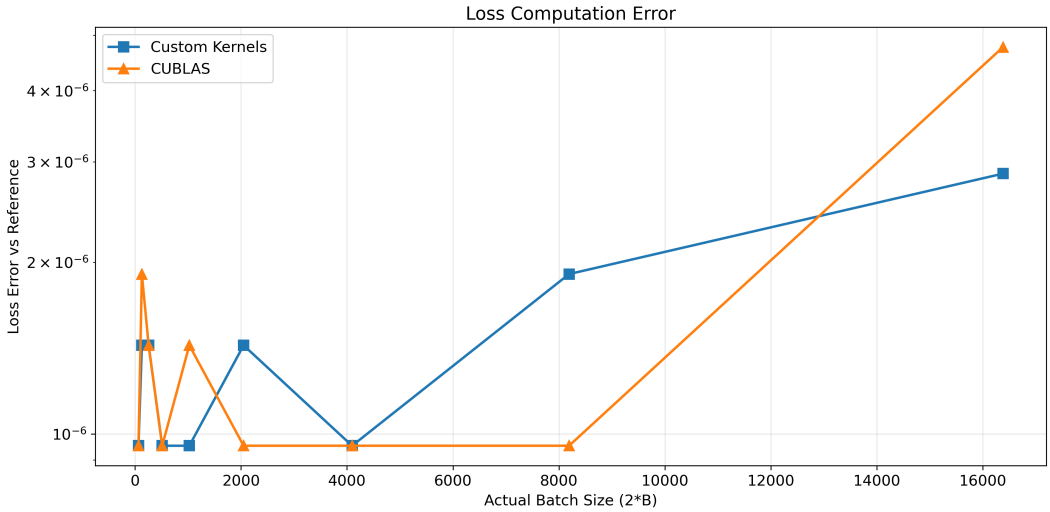

Figure 3. Loss computation accuracy showing numerical precision of CUDA implementation compared to PyTorch reference.

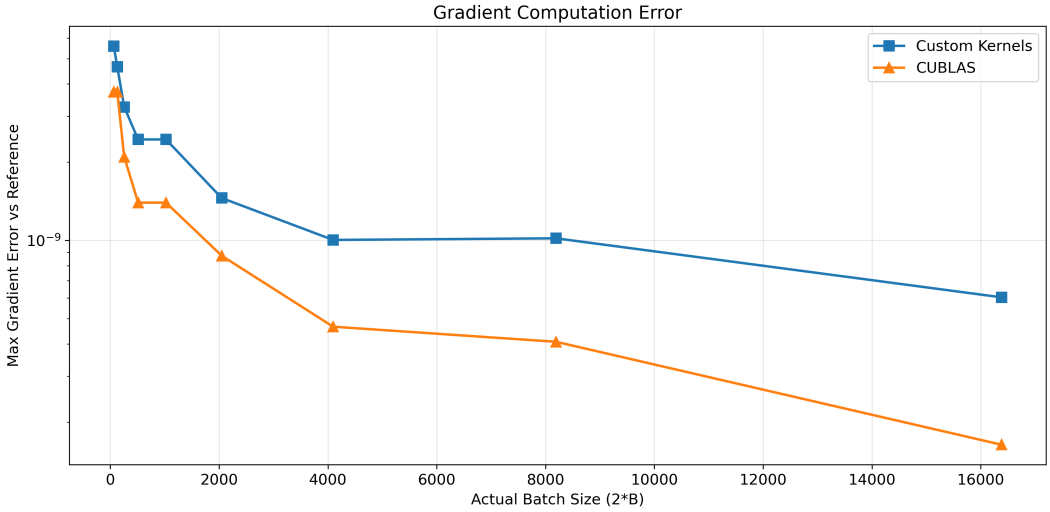

Figure 4. Gradient computation accuracy demonstrating backward pass precision across different batch sizes.