CuBridge : A GPU-Accelerated Tensor Library for Java

조선대학교 3학년
컴퓨터공학과
배준호

0. 요약 (Abstract)

수치 연산 및 딥러닝 분야에서는 Python 기반의 텐서 연산 프레임워크들이 사실상 표준처럼 활용되고 있다. 다양한 라이브러리가 지속적으로 발전하며 견고한 생태계를 형성하고 있으나, 이에 비해 Java 환경에서는 유사한 기능을 제공하는 시스템이 상대적으로 부족한 실정이다. 특히 고성능 GPU 연산을 포함한 텐서 기반 계산을 Java에서 직접 제어할 수 있는 경량 시스템은 아직 널리 보급되지 않았다.

본 연구에서는 Java 환경에서 독립적으로 작동하며 텐서 연산 기능을 수행할 수 있는 시스템인 CuBridge를 설계하고 구현하였다. CuBridge는 연산의 정의와 실행을 명확히 분리하는 구조를 채택하였으며, Java에서는 연산 흐름을 단순히 지시하고, 실제 계산은 C++ 및 CUDA를 기반으로 수행된다. 전체 구조는 JNI 기반의 동적 DLL 연동 방식으로 구성되어 있으며, 실행 환경에 따라 연산 경로가 GPU 또는 CPU로 자동 분기된다.

모든 연산은 put -> cal -> get의 일관된 흐름을 따르며, 단항·이항·축 연산과 행렬 내적 외에도, Affine 변환, Softmax, 손실 함수 등 신경망 학습에 필요한 고수준 연산을 자체적으로 지원한다.

본 논문에서는 CuBridge의 설계 철학과 내부 구조, 주요 함수 체계를 소개하고, ND4J, JCuda 등 Java 기반 수치 연산 라이브러리들과의 비교를 통해 구조적 단순성과 연산 구성의 직관성 측면에서 CuBridge가 지니는 차별점을 논의한다. 또한 실제 연산 예제와 성능 평가 결과를 바탕으로, CuBridge가 Java 환경에서 수치 및 신경망 연산을 위한 기반 시스템으로서 가질 수 있는 가능성을 제시하고자 한다.

목차

1. 서론 (Introduction)

1.1 연구 배경

최근 인공신경망을 비롯한 머신러닝 기술은 빠른 속도로 발전하고 있으며, 복잡한 학습 구조와 모델 설계를 지원하는 다양한 프레임워크들이 등장하고 있다. 이러한 고수준 시스템들의 기반에는, 단순하면서도 강력한 수치 연산 라이브러리가 핵심적으로 자리 잡고 있다. 일반적으로는 Python 환경에서 NumPy, Pandas와 같은 고성능 연산 도구들이 기본적으로 사용되며, PyTorch나 TensorFlow 또한 내부적으로 이들 연산 구조에 상당 부분 의존하고 있다.

이와 같은 흐름 속에서 NumPy, CuPy와 같은 Python 중심의 텐서 연산 시스템은 사실상 업계 표준으로 자리매김하였으며, 대부분의 연구 및 개발 환경이 해당 생태계를 전제로 구성되고 있는 실정이다. Python은 배우기 쉽고 다양한 기능을 제공하는 언어로서 많은 장점을 지니고 있으나, 그만큼 생태계에 대한 의존성도 크게 작용한다.

본 논문은 이러한 Python 중심의 환경에서 벗어나, 보다 범용적이고 독립적인 연산 시스템의 가능성을 모색하고자 하였으며, 그 구현 기반으로 Java를 선택하였다. Java는 상대적으로 저수준 메모리 제어가 가능하면서도 플랫폼 독립적인 구조를 갖추고 있으며, 시스템 연동성, 보안성, 네트워크 처리 능력 등의 측면에서 산업 현장에서도 꾸준히 활용되고 있다. 이에 따라 Java 기반의 GPU 연산 시스템을 구축할 수 있다면, Python 생태계에 속하지 않는 환경에서도 효율적인 텐서 연산 및 학습 시스템을 구현할 수 있을 것이라는 가정에서 본 연구를 시작하게 되었다.

1.2 자바와 쿠다 결합의 장점

Java는 플랫폼 독립성과 객체지향 구조, 풍부한 생태계를 바탕으로 다양한 산업 및 시스템 개발 분야에서 폭넓게 활용되고 있는 언어이다. 특히 정적 타입에 기반한 안정적인 메모리 모델과 보안성은 대규모 프로젝트나 분산 시스템 개발에서 강점을 제공하며, 자동 가비지 컬렉션과 네트워크 기반 서버 환경에 최적화된 기능 또한 실용적인 장점으로 작용한다. Python에 비해 실행 속도가 더 빠른 경우도 존재하며, 무엇보다도 산업 현장에서 널리 사용되고 있다는 점은 Java의 실용성과 호환성을 더욱 부각시킨다.

한편, CUDA는 NVIDIA GPU를 기반으로 하는 병렬 연산 플랫폼으로, 대규모 수치 계산이나 딥러닝 연산에서 매우 높은 처리 성능을 제공하는 기술이다. 이러한 두 환경을 결합할 경우, Java의 안정성과 호환성, 그리고 CUDA의 고성능 병렬 연산 능력을 동시에 활용할 수 있는 연산 구조를 구성할 수 있다. Java에서 복잡한 연산 흐름을 명확하게 정의하고, 실제 연산은 CUDA를 통해 고속으로 수행함으로써, 연산 효율성과 유지보수성을 함께 확보할 수 있다.

또한 Java는 GUI, 네트워크, 파일 입출력, 데이터베이스 연동 등 다양한 외부 시스템과의 연결에 강점을 갖고 있어, 계산 결과를 시각화하거나 외부 애플리케이션과의 연동 측면에서도 용이하다. 반면 CUDA는 연산 처리에 특화되어 있으나, 애플리케이션 전체의 구성이나 확장성 측면에서는 한계를 지닌다. 이와 같은 특성은 두 기술을 상호 보완적으로

결합할 수 있는 가능성을 시사한다.

물론 Java와 CUDA의 결합을 시도한 기존 사례도 존재한다. 대표적으로 JCuda는 CUDA 기능을 Java 환경에서 사용할 수 있도록 구현되었으나, 개발자가 포인터를 직접 다루어야 한다는 점에서 Java의 타입 안정성과 추상화를 크게 훼손하는 문제가 있었다. 이는 Java 언어의 핵심 철학과는 다소 상충되며, 메모리 접근 및 해제를 개발자가 명시적으로 처리해야 한다는 점에서 진입 장벽이 존재하였다. 더불어 CUDA 커널을 직접 작성하고, 복잡한 컴파일 및 링크 설정을 별도로 관리해야 하는 부담도 있었다. 이로 인해 JCuda는 Java와 CUDA를 단순히 연결하는 래퍼 수준에 머무르고 있으며, 전체 연산 흐름을 Java 수준에서 일관되게 구성하기 어려운 구조적 한계를 보였다.

이러한 배경을 바탕으로, 본 연구에서는 Java의 실용성과 CUDA의 성능을 유지하면서도, Java 코드만으로 직관적이고 일관된 연산 흐름을 구성할 수 있는 구조를 지향하였다.

1.3 본 논문의 기여

본 논문에서 제안하는 CuBridge는 Java와 CUDA의 결합에 있어 기존 방식들과는 다른 구조적 특성을 갖는다. 특히 CUDA의 병렬 연산 성능을 활용하면서도, Java 개발자가 복잡한 커널 코드나 메모리 관리를 직접 다루지 않도록 구성된 추상화 계층이라는 점에서 의의를 지닌다. 주요 특징은 다음과 같다.

1. 연산 흐름의 단순화

기존 JCuda와 같은 시스템은 Java에서 CUDA 함수를 직접 호출하는 방식에 머무르는 경우가 많았다. 이에 비해 CuBridge는 연산의 정의와 실행을 명확히 분리하였으며, 사용자는 put -> cal -> get이라는 일관된 인터페이스만으로 연산을 수행할 수 있도록 구성하였다. Java 코드 내에서 연산 흐름을 선언형으로 구성할 수 있으며, 별도의 커널 작성 없이 직관적인 코드 작성이 가능하다.

2. Java 철학의 유지

CuBridge는 Java의 객체지향 구조와 자동 메모리 관리 철학을 유지한다. 개발자가 포인터를 직접 다루지 않아도 텐서를 구성할 수 있으며, 내부적으로는 Java의 가비지 컬렉션과 타입 안정성(Type Safety)을 활용하여 메모리 관리 부담을 최소화하였다. 연산 결과의 재사용이나 중간 결과의 누적 또한 큐 기반으로 자동 관리되어, 명시적인 자원 해제가 필요하지 않다.

3. 유연한 변수 관리

CuBridge는 문자열 기반 변수명을 통해 텐서를 관리하며, 사용자가 이름을 명시하지 않아도 내부적으로 자동 이름이 할당된다. 동일한 이름의 변수에 연산 결과를 재할당하거나 중복된 연산이 발생하는 경우에도, 시스템이 내부적으로 이를 처리하여 일관된 흐름을 유지한다. 이를 통해 사용자는 연산 순서나 데이터 흐름에 대해 선언적 수준에서만 설계하면 된다.

4. 환경 적응형 실행

CuBridge는 실행 환경을 자동으로 감지하여, CUDA가 존재하는 경우에는 GPU 모드로, 그렇지 않은 경우에는 CPU 모드로 전환된다. 이 전환 과정은 사용자 개입 없이 자동으로 처리되며, 동일한 Java 코드가 다양한 시스템 환경에서 동일하게 작동할 수 있도록 하여 높은 이식성과 실용성을 확보하였다.

이와 같은 구조를 통해 CuBridge는 GPU의 고성능 연산 능력을 활용하면서도, Java 언어의 개발 철학을 해치지 않는 범위 내에서 직관적이고 확장성 있는 수치 연산 플랫폼을 제공한다. 이는 Java 기반 신경망 설계 및 수치 계산 시스템의 구현에 있어 하나의 실용적인 대안을 제시할 수 있을 것으로 기대된다.

2. 관련 연구

2.1 기존 라이브러리 소개 및 단점

Java 환경에서 수치 연산 또는 GPU 연산을 구현하려는 시도는 과거부터 다양하게 이루어져 왔다. 대표적으로 ND4J(Numerical Data for Java)는 다차원 배열 연산을 지원하는 Java 기반 텐서 연산 라이브러리이며, JCuda는 CUDA의 기능을 Java에서 사용할 수 있도록 바인딩하여 GPU 연산을 가능하게 한 프로젝트이다. 이외에도 DeepLearning4J, Aparapi, Rootbeer 등 Java 기반 GPGPU 프레임워크들이 다수 개발되어 왔다. 그러나 이러한 라이브러리들은 다음과 같은 구조적 한계를 공통적으로 지닌다.

1. JCuda의 저수준 제어 부담

JCuda는 CUDA의 핵심 기능을 Java에서 직접 호출할 수 있도록 바인딩을 제공하지만, CUDA의 포인터 구조나 메모리 모델을 Java 코드에서 직접 다루어야 한다. 이는 Java 언어가 지향하는 추상성과 안정성 철학에 어긋나며, Java 개발자 입장에서는 익숙하지 않은 저수준 메모리 접근과 디버깅 부담을 야기하게 된다.

2. ND4J의 내부 복잡성

ND4J는 고수준 API를 제공하지만, 내부 구현이 복잡하고 연산 흐름의 추상화가 충분히 이루어지지 않아 연산 흐름을 직관적으로 파악하거나 제어하기 어렵다. 특히 GPU 연산을 사용하기 위해서는 별도의 구성 파일과 환경 설정이 요구되며, 버전 간 호환성 문제 또한 자주 발생한다.

3. DeepLearning4J의 추상화 한계

DeepLearning4J는 신경망 전체 구조를 관리하는 프레임워크로서, 연산 단위보다는 모델 수준의 학습 구성에 초점을 맞추고 있다. 따라서 개별 연산의 세부 조작이나 사용자 정의 흐름 제어에는 제약이 따르며, 단순한 수치 연산 플랫폼으로 활용하기에는 한계가 존재한다.

4. 환경 의존성과 유지 관리 문제

이러한 시스템들은 대부분 타 라이브러리에 강하게 종속되어 있으며, GPU가 없는 환경이나 JNI를 사용할 수 없는 환경에서는 기능이 제한되거나 정상적으로 동작하지 않는 경우가 많다. 특히, 현재 시점 기준으로는 JCuda 및 ND4J를 포함한 주요 Java 기반 GPGPU 라이브러리들은 CUDA 최신 버전과의 호환이 중단된 상태이며, 대부분의 프로젝트가 활발한 유지보수 없이 개발이 중지된 것으로 확인된다. 이로 인해 최신 CUDA 환경에서 안정적으로 작동하는 Java 기반 GPU 가속 라이브러리는 사실상 존재하지 않는 상황이다.

이처럼 기존 Java 연산 시스템들은 각기 다른 접근 방식으로 GPU 연산 및 병렬 연산을 지원하고자 하였으나, 여러 공통적인 구조적 한계를 지니고 있다. 우선, 대부분의 시스템은 포인터, 버퍼 주소, 메모리 크기 등의 자원을 개발자가 직접 제어해야 하는 구조로 되어 있어, Java 언어의 본래 추상성과 안정성 철학에 부합하지 않는 저수준 제어를 요구한다. 또한 CUDA 및 JNI 환경을 전제로 설계되어 있어, 해당 환경이 구축되어 있지 않은 경우에는 기능 수행이 제한되거나 실행 자체가 불가능한 경우도 존재한다.

더불어 연산 흐름이 복잡하게 얽혀 있는 경우가 많아, 수치 연산을 직관적으로 설계하거나 디버깅하기 어렵다는 한계가 있다. 특히 Java 개발자 입장에서는 CUDA 또는 GPGPU 관련 개념을 별도로 학습해야 하기 때문에 진입 장벽이 높으며, 이러한 부담은 실제 적용 가능성을 낮추는 요인이 되기도 한다. 마지막으로, 기존 시스템의 상당수가 최신 CUDA 버전에 대한 호환성을 유지하지 못하고 있는 상황으로, 빠르게 변화하는 하드웨어 및 드라이버 환경에 대응하기 어려운 구조적 단절 문제가 존재한다.

2.2 타 언어 라이브러리 소개

Python 기반의 수치 연산 라이브러리는 현재 연구 및 산업 전반에서 사실상 표준처럼 자리잡고 있다. 대표적으로 NumPy는 고성능 배열 연산의 핵심 도구로 널리 활용되고 있으며, CuPy는 그 구조를 GPU 환경에 맞추어 구현한 대체 라이브러리로 사용되고 있다. PyTorch나 TensorFlow와 같은 딥러닝 프레임워크 역시, 내부적으로는 이러한 저수준 텐서 연산 시스템을 기반으로 다양한 고수준 기능을 제공하고 있다.

이러한 프레임워크들은 높은 사용성과 체계적인 문서화, 방대한 커뮤니티를 바탕으로 지속적인 발전을 이루고 있으며, 학습 곡선이 완만하고 풍부한 예제가 제공되어 실험 및 연구 환경에 적합한 장점을 지닌다. 그러나 구조적으로 Python 언어에 밀접하게 결합되어 있어, 연산 속도, 멀티스레딩, 시스템 수준 제어 등의 측면에서는 일정한 제약이 존재하며, 독립적이고 경량화된 환경에서 활용하기에는 다소 무겁거나 복잡한 경우도 있다.

또한 많은 수치 연산 라이브러리들은 Python 문법과 객체 모델에 강하게 의존하여 설계되었기 때문에, 다른 언어로의 이식이나 임베디드 시스템과 같은 제한된 환경에 통합하는 데 있어 어려움이 따를 수 있다. 이러한 한계로 인해, 일부 상황에서는 Python 외의 언어 환경에서도 동등한 수준의 연산 성능과 유연한 API를 제공할 수 있는 대안의 필요성이 제기되고 있다.

CuBridge는 이러한 필요를 바탕으로, Java 환경에서도 수치 연산 및 GPU 가속 연산을

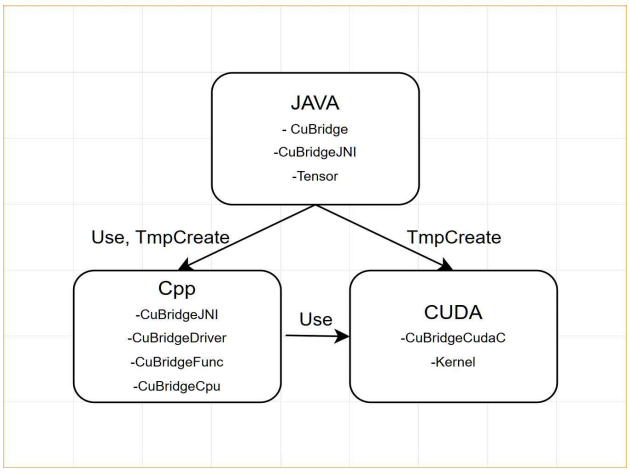
수행할 수 있는 프레임워크를 구축하고자 설계되었다. 전체적인 구조와 연산 흐름은 NumPy 및 CuPy와 같은 Python 계열 라이브러리의 철학을 참고하되, Java 언어의 문법 구조와 메모리 모델에 맞게 독립적으로 재구성되었다.

즉, CuBridge는 Python 기반 시스템을 개량하거나 대체하려는 것이 아니라, 해당 시스템들의 연산 철학과 구조적 설계를 참고하여 Java 언어에 적합하게 구현한 독립적인 시도라 할 수 있다. 이는 Java 환경에서 수치 연산이 필요한 개발자에게 새로운 선택지를 제공하고, Python 생태계를 보완하는 병렬적 시스템으로 작동할 수 있을 것으로 기대된다.

3. 시스템 개요

3.1 구조 개괄

CuBridge는 연산 지시와 실행을 분리하는 철학을 중심으로 설계되었다. 전체 구조는 Java 영역의 프론트엔드와 C++/CUDA 영역의 백엔드로 이원화되어 있으며, 이 둘은 JNI를 통해 연결된다. 사용자는 Java 코드에서 연산 흐름을 선언하며, 백엔드에서 해당 연산이 처리되는 방식으로 동작한다. 클래스간 상세 관계는 부록 A에 첨부한다.



1. 프론트엔드 (Java):

사용자가 직접 사용하는 API 계층으로, Tensor, CuBridge, CuBridgeJNI 등의 클래스로 구성된다. 연산은 put -> cal -> get의 구조적 흐름을 따르며, 함수들은 사용자 친화적인 인터페이스와 다양한 매개변수 오버로딩, 자동 변수 명명, 선언형 연산 흐름, 함수 체이닝을 통해 높은 직관성과 유연성을 제공한다.

2. 백엔드 (C++/CUDA):

실제 연산을 수행하는 계층이다. 프론트엔드에서 전달된 지시는 백엔드 내부 큐에 적재되고, 함수 실행 시 큐에서 필요한 텐서를 꺼내 연산을 처리한다. 각 연산은 크게 8단계의 과정을 거치며, 이를 통해 안정성과 신뢰성을 보장한다. 자동으로 컴퓨터 환경을 감지하여 CUDA의 유무를 판단하고, 실제 연산 실행 시 GPU / CPU 중 하나를 선택하여 실행한다.

3.2 주요 용어

1. Tensor (Java 측)

CuBridge 프론트엔드(Java 영역)에서 사용되는 주요 데이터 객체로, 사용자가 직접 생성 및 접근하는 단위이다. 내부적으로는 배열 형태의 데이터를 나타내며, data, shape, len 등의 속성을 가진다. 연산 흐름 정의 시 입력 및 출력 변수로 사용되며, 사용자 입장에서의 추상화된 텐서 개념을 구현한다.

2. Tensor (C++ 측)

CuBridge 백엔드(C++/CUDA 영역)에서 실질적인 연산 대상으로 사용되는 객체이다. Java 측 Tensor 객체로부터 전달된 정보를 바탕으로 생성되며, 내부적으로는 name, data, shape, dLen, sLen, usageCount, isBroadcast 등의 속성을 포함한다. 모든 연산 함수는 이 객체를 기반으로 실행된다.

3. TensorQueue

백엔드에서 사용되는 텐서 입력 대기열 구조로, 프론트엔드에서 put() 함수를 통해 전달된 텐서들이 순차적으로 저장된다. 모든 연산 함수는 실행 시 해당 큐를 참조하여 필요한 텐서가 존재하는지 여부를 판단하고, 연산 준비를 수행한다.

4. TensorBuffer

TensorQueue에서 존재 여부가 확인된 텐서를 임시로 저장하는 백엔드의 버퍼 구조이다. 실제 연산이 이루어지는 공간으로, 모든 연산 함수는 해당 버퍼에서 텐서를 불러와 연산을 수행한다. 연산 중 텐서의 수정, 교환, 결과 반영 등은 이 버퍼를 통해 이루어진다.

4. CuBridge 프론트엔드

4.1 Tensor

Tensor 클래스는 CuBridge 프론트엔드(Java 영역)에서 모든 연산의 기준이 되는 핵심 데이터 객체로, 다차원 수치 배열을 표현한다. 내부적으로는 data, shape, len 등의 멤버 변수를 통해 텐서의 구조와 내용을 관리하며, CuBridge에서 정의되는 모든 연산은 이 객체를 기반으로 입출력이 수행된다. 즉, 사용자는 해당 클래스를 활용함으로써 다차원 행렬 데이터를 손쉽게 정의하고 처리할 수 있다.

1. 생성자

CuBridge의 Tensor 클래스는 다양한 상황에 대응할 수 있도록 여러 종류의 생성자를 제공한다. 이는 다양한 형태의 데이터를 손쉽게 Tensor 객체로 감쌀 수 있도록 하기 위함이다.

순번	함수 헤더	설명
1	Tensor()	기본 생성자. 별도의 데이터 없이 객체만 생성되며, 이후 정적 함수 호출 등에 활용된다.

2	Tensor(double... data)	실수형 데이터를 단일 값 또는 배열 형태로 입력받아 텐서를 생성한다. 자동으로 1차원 형태로 간주된다.
3	Tensor(double[] data, int... shape)	데이터와 함께 다차원 shape을 지정하여 직접 텐서를 구성한다.
4	Tensor(String path)	외부 CSV 파일을 불러와 2차원 텐서를 생성한다. 이때 모든 값은 실수형이어야 하며, 첫 열은 라벨로 간주되어 자동 제거된다.
5	Tensor(String[][] data)	문자열 배열을 직접 입력하여 2차원 텐서를 구성한다. 외부 텍스트 파일 등을 통해 불러온 데이터를 처리할 때 사용된다.
6	Tensor(String[][] data, double norm)	위와 동일한 문자열 입력에 대해 정규화를 적용할 수 있도록, 전체 값을 norm으로 나눈 상태로 생성한다.

2. 주요 유틸리티 함수

유틸리티 함수들은 CuBridge 사용자에게 텐서의 상태를 손쉽게 확인하고, 필요한 정보를 직관적으로 추출할 수 있는 수단을 제공한다. 특히 Java 언어의 일반적인 디버깅 흐름과 호환되도록 구성되어 있어, 텐서 연산 전후의 상태 점검이나 로깅에도 효과적으로 활용될 수 있다.

순번	함수 헤더	설명
1	double[] toArray()	텐서 내부에 저장된 데이터를 1차원 실수 배열 형태로 반환한다. 외부 연산이나 사용자 정의 로직에서 텐서 데이터를 직접 활용할 수 있도록 돕는다.
2	int getSize()	텐서에 포함된 전체 원소 수를 반환한다. 연산 시 크기 검증이나 메모리 제어 등에 활용될 수 있다.
3	int[] getShape()	텐서의 다차원 구조(shape)를 배열 형태로 반환한다. 축별 차원을 확인하거나 조건 분기 등에 사용된다.
4	void printSize()	텐서의 shape를 콘솔에 출력한다. 디버깅 또는 구조 확인 시 활용된다.
5	void printData()	텐서의 전체 데이터를 콘솔에 출력한다. 입력값 또는 연산 결과를 육안으로 확인할 때 유용하다.
6	double[] head()	텐서 데이터의 앞부분 일부(기본적으로 6개 요소)를 1차원 배열로 반환한다. 전체 출력이 부담스러운 대용량 텐서에 대해 샘플을 빠르게 확인할 수 있도록 한다.

3. 주요 정적 함수

정적 함수들은 Java 코드 상에서 간결하게 텐서를 초기화할 수 있도록 지원하며, 다양한 실험 환경이나 초기값 조건에 맞춰 유연하게 사용할 수 있는 기본 생성 도구들이다. 이를 통해 CuBridge는 NumPy와 유사한 초기화 함수 구성을 Java 문법에 맞게 제공하며, 수치 연산 실험의 시작점을 효율적으로 정의할 수 있도록 돕는다.

순번	함수 헤더	설명
1	Tensor.filled(double value, int... shape)	주어진 shape에 따라, 모든 원소를 동일한 value 값으로 초기화한 텐서를 생성한다. 특정 상수값을 기준으로 텐서를 구성하거나 마스크, 오프셋 등의 연산에 활용할 수 있다.
2	Tensor.rand(int... shape)	지정된 shape에 따라 0 이상 1 미만의 균등분포(uniform distribution)를 갖는 난수 값으로 텐서를 초기화한다. 초기화 실험이나 무작위 입력이 필요한 경우 사용된다.
3	Tensor.randn(int... shape)	지정된 shape에 따라 평균 0, 표준편차 1의 정규분포(standard normal distribution)를 따르는 값으로 텐서를 초기화한다. 딥러닝 가중치 초기화 등에 자주 활용된다.
4	Tensor.zeros(int... shape) Tensor.ones(int... shape)	각각 모든 원소를 0 또는 1로 초기화한 텐서를 생성한다. 기본값 설정이나 조건 비교 연산 등 다양한 상황에서 활용된다.

5	Tensor.arange(double start, double end, double step)	시작값(start)부터 종료값(end) 이전까지 일정 간격(step)으로 증가하는 값을 갖는 1차원 텐서를 생성한다. 반복 인덱스, 위치 벡터 등 다양한 정렬형 연산에 적합하다.
6	Tensor.linspace(double start, double end, int count)	지정된 구간 [start, end]를 균등하게 나눈 count개의 값을 갖는 1차원 텐서를 생성한다. 연속 구간 샘플링이나 그래프 축 구성 등에 활용된다.

이처럼 Tensor 클래스는 CuBridge 내 모든 연산의 입출력 단위를 정의하는 핵심 요소로, 다양한 생성 방식과 편의 기능을 통해 사용자 친화적인 연산 환경을 제공하며, 프론트엔드와 백엔드 간의 일관된 데이터 흐름을 유지하는 기반이 된다.

4.2 CuBridge

CuBridge 클래스는 사용자가 연산 흐름을 직접 작성하는 주요 API 객체로, 프론트엔드(Java 영역)에서 중심적인 역할을 수행한다. 모든 연산은 이 클래스를 통해 선언되며, 내부적으로는 Java에서 정의된 연산 명령이 C++/CUDA 기반 백엔드로 전달되기 전, 적절한 형태로 변환 및 정리되는 과정을 거친다.

사용자는 이 클래스를 통해 입력 데이터를 등록하고, 원하는 연산을 선언하며, 결과를 조회하는 일련의 과정을 단순하고 직관적으로 구성할 수 있다. 전체 연산 흐름은 put -> cal -> get이라는 일관된 구조를 따르며, 이를 통해 연산 정의를 선언적으로 구성할 수 있다는 점에서 높은 가독성과 간결함을 제공한다.

특히 CuBridge 클래스는 대부분의 연산 함수에서 함수 오버로딩(overloading)을 적극적으로 지원한다. 동일한 연산에 대해 입력 형식이나 인자의 개수에 따라 다양한 호출 방식이 제공되며, 출력 변수명을 생략한 경우에는 내부적으로 자동 명명되어 관리된다. 이러한 구조는 실험용 코드 작성이나 반복적인 연산 설계에 높은 효율성을 제공하며, 사용자가 매번 변수명을 직접 지정하지 않더라도 연산 흐름을 유연하게 설계할 수 있도록 돕는다.

또한 CuBridge는 Java의 Tensor 객체를 C++ 백엔드에서 사용할 수 있도록 변환하는 중요한 중간 처리 기능도 수행한다. 사용자가 정의한 Tensor는 단순히 JNI를 통해 직접 전달될 수 없기 때문에, put() 함수 내부에서는 해당 객체를 구성하는 요소들(데이터 배열, shape, 변수명 등)을 해석한 뒤, C++ 측 구조에 맞춰 전달하는 전처리 과정을 수행한다. 이를 통해 CuBridge는 Java의 추상 객체 모델과 CUDA 기반의 저수준 구조를 자연스럽게 연결하는 역할을 담당한다.

이처럼 CuBridge 클래스는 선언적 연산 설계, 함수 오버로딩, 데이터 구조 변환 등 다양한 측면에서 사용자의 부담을 최소화하고, 백엔드와의 연계를 직관적으로 수행할 수 있도록 설계되어 있다. 이후 소개되는 연산 함수들 역시 이 구조를 바탕으로 동작한다.

1. 환경 제어 함수

CuBridge는 연산 실행 시 시스템 환경을 자동으로 감지하여 GPU(CUDA) 또는 CPU 경로로 연산을 분기한다. CuBridge는 현재 연산이 어떠한 경로를 사용하는지 확인하고, 그 경로를 사용자 임의로 조작할 수 있도록 함수를 지원한다.

순번	함수 헤더	설명
1	selectGPU()	연산을 GPU 경로로 강제 설정한다.
2	selectCPU()	연산을 CPU 경로로 강제 설정한다.
3	envReset()	경로를 초기화하고 CUDA 지원 여부를 다시 탐색한다.
4	getEnvironmentStatus()	현재 연산 환경 및 내부 플래그 상태를 반환한다.

2. 텐서 관리 함수

CuBridge에서는 연산 흐름의 전후로 입력 텐서를 등록하거나 연산 결과를 반환받는 작업이 필요하다. 이를 위해 텐서의 등록 및 조회를 위한 다양한 함수를 제공하며, 특히 텐서 등록을 위한 put() 함수는 여러 형태의 오버로딩을 지원하여 입력 방식에 유연하게 대응할 수 있도록 구성되어 있다. 간단한 스칼라 값은 텐서 객체를 생성하지 않고도 직접 입력할 수 있으며, 이 외에도 백엔드에서의 텐서 존재 여부를 확인하거나 내부 연산 큐 전체를 초기화하는 기능도 함께 제공된다.

순번	함수 헤더	설명
1	put(int data, String name, int usageCount) put(double data, String name, int usageCount)	스칼라 값을 등록한다. 내부적으로 텐서로 랩핑되며, 기본적으로 브로드캐스팅을 허용한다. 이름과 사용횟수를 옵션으로 가진다.
2	put(Tensor data, String name, int usageCount, boolean broadcast)	텐서를 등록한다. 이름, 사용횟수, 브로드캐스팅 여부를 옵션으로 가진다.
3	get(String name)	이름에 해당하는 텐서를 하나 반환한다. 이름이 비어있다면 내부 큐의 가장 최선단의 텐서를 반환한다.
4	clear()	현재까지 등록된 모든 큐의 내용을 초기화한다.
5	visualQueue()	벡엔드의 큐에 등록된 텐서 목록을 콘솔에 출력한다.
6	visualBuffer()	벡엔드의 버퍼에 등록된 텐서 목록을 콘솔에 출력한다.

3. 연산 함수 유형

CuBridge에서 정의된 연산 함수들은 전체 시스템 구조 중에서도 가장 핵심적인 요소로, 텐서 연산의 흐름을 실제로 구성하고 수행하는 역할을 담당한다. 이러한 함수들은 단순한 수치 계산을 넘어, 전체 연산 흐름의 구성과 신경망을 비롯한 학습 알고리즘의 설계에 직접적으로 활용되므로, CuBridge의 철학과 기능적 특성을 가장 명확하게 반영하는 부분이라 할 수 있다.

또한 CuBridge의 연산 함수들은 텐서의 이름이나 입력을 명시하지 않고도 호출이 가능하도록 설계되어 있다. 입력 텐서가 지정되지 않은 경우에는 내부 큐의 최상단 텐서를 자동으로 선택하여 연산이 수행되며, 출력 이름이 생략된 경우에는 시스템이 자동으로 임시 이름을 생성하여 결과를 저장한다. 이러한 자동 처리 구조는 모든 연산 함수에 대해 다양한 형태의 오버로딩으로 구현되어 있으며, 사용자가 자연스럽게 코드를 작성하는 것만으로도 높은 편의성과 유연성을 확보할 수 있도록 돕는다.

CuBridge는 연산의 성격 및 적용 축(axis)에 따라 함수들을 체계적으로 분류하고 있으며, 각 유형별 상세 함수 목록은 본 논문 부록 C에 별도로 제시하고 해당 절에서는 그 개념을 설명한다.

순번	함수 분류	설명
1	단항 연산 (Unary)	입력 텐서 하나에 대해 요소별(element-wise)로 작용하는 기본 연산들이다. 대표적으로 절댓값(abs), 부호 반전(neg), 제곱근(sqrt), 로그(log), 시그모이드(sigmoid) 등이 있으며, 대부분의 경우 입력 텐서의 shape과 동일한 출력 텐서를 생성한다.
2	이항 연산 (Binary)	두 개의 입력 텐서를 받아 각 요소 간 연산을 수행한다. 덧셈(add), 곱셈(mul), 나눗셈(div) 외에도, 크기 비교(gt, lt), 동일성 검사(eq) 등의 논리 연산도 포함된다. 텐서 간 shape이 일치하지 않을 경우, 자동 브로드캐스트 기능이 적용되어 유연한 연산이 가능하다.
3	축 종속 연산 (Axis-Based)	지정된 축을 기준으로 하위 모든 축의 값을 함께 고려하여 연산을 수행하는 방식이다. 대표적으로 mean, var, std와 같은 함수는 특정 축을 기준으로 하위의 모든 데이터를 통합하여 평균이나 분산 등의 값을 계산한다. 이처럼 연산 결과는 지정 축을 따라 통합되며, 지정 축 이하의 정보가 하나의 값으로 축소되는 구조를 갖는다.
4	축 독립 연산 (Axis-Free)	축 독립 연산은 지정된 하나의 축에만 독립적으로 작용하며, 그 외의 축 구조는 그대로 유지되는 방식이다. 예를 들어, accumulate는 특정 축을 따라 누적합을 계산하고, expand는 지정 축의 크기를 확장하여 동일한 값을 복사하는 연산을 수행한다. 이들 함수는 모두 하나의 축에만 선택적으로 작용하며, 다른 축에는 영향을 미치지 않는다는 점에서, 국소적인 축 조작 연산으로 분류된다.
5	행렬 변환 연산 (Matrix Operation)	행렬 변환 연산은 텐서의 구조 자체를 변경하거나, 연산 수행에 적합한 형태로 재구성하기 위해 사용된다. 대표적으로 transpose는 텐서의 차원 순서를 변경하며, dot, matmul과 같은 함수는 다차원 행렬 간의 내적 또는 행렬 곱 연산을 수행한다. 또한 im2col 및 col2im 함수는 CNN 연산에서 주로 사용되는 변환 연산으로, 이미지 데이터를 행렬 형태로 재배열하거나 다시 복원하는 데 사용되며, 이들 역시 행렬 변환 범주에 속한다. 한편 broadcast 함수는 두 텐서의 shape을 일치시키기 위해 축의 크기를 자동으로 확장하는 기능을 수행하며, 구조적으로는 행렬 변환 연산의 일종으로 분류된다. 다만 해당 함수는 사용자에 의해 직접 호출되지 않고, 이항 연산이나 기타 내부 연산 과정 중 자동으로 적용된다.
6	신경망 연산	신경망 연산은 인공신경망 학습에 특화된 고수준 연산 함수들로, 내부적으로는 여러 기본 연산의 연속적인 호출로 구성되어 있다. affine, softmax, mse, cee 등의 함수가 이에 해당된다. 이러한 연산들은 CuBridge 내부에 미리 정의되어 있어, 사용자가 별도로 수식을 구성하지 않더라도 곧바로 호출이 가능하며, 연산 흐름은 백엔드 단계에서 최적화된 방식으로 고속 실행된다. 이를 통해 신경망 구조 설계와 학습 연산 구성이 한층 간결하고 직관적으로 이루어질 수 있다.

이상과 같이 CuBridge는 연산 흐름을 중심으로 한 구조적 설계, 유연한 텐서 관리, 함수 오버로딩 및 자동 처리 시스템을 통해 Java 환경에서도 효율적이고 직관적인 수치 연산을 가능하게 한다. 특히 다양한 연산 유형을 체계적으로 분류하고, 축 기반 연산 및 고수준 신경망 연산까지 포괄함으로써, 단순한 연산 라이브러리를 넘어 신경망 구성의 실질적인 기반 도구로서 활용 가능한 확장성을 갖추고 있다.

다만 CuBridge 클래스 자체가 곧바로 백엔드 연산을 수행하는 것은 아니며, Java와 C++ 사이의 연동을 위해 중간 계층으로서의 네이티브 함수 연결 클래스가 필요하다. 이 역할을 담당하는 것이 바로 CuBridgeJNI 클래스이며, 다음 장에서는 해당 클래스의 구조와 역할에 대해 설명한다.

4.3 CuBridge_JNI

CuBridge 시스템은 Java에서 선언된 연산 명령을 C++ 및 CUDA 환경으로 전달하기 위해 Java Native Interface(JNI)를 활용한다. JNI는 Java와 네이티브 코드(C/C++) 사이의 연결을 담당하는 표준 인터페이스로, Java의 추상성과 이식성을 유지하면서도 고성능 하드웨어 자원에 접근할 수 있도록 설계된 기술이다.

일반적으로 Java는 JVM 상에서 실행되기 때문에 운영체제나 하드웨어에 직접 접근하기 어렵다. 반면 C/C++은 포인터 기반 메모리 제어, GPU 연산 등 저수준 시스템 제어에 적합하다. 이를 결합하기 위해 Java는 native 키워드를 통해 외부 네이티브 라이브러리를 선언하고, JNI를 이용해 해당 라이브러리와 연결하여 연산을 수행할 수 있다. CuBridge는 이러한 구조를 기반으로 Java와 C++/CUDA 간 연산 경로를 연결하고 있다.

CuBridge에서는 CuBridgeJNI 클래스가 프론트엔드(Java)와 백엔드(C++/CUDA)를 잇는 중간 계층으로서의 역할을 수행한다. 사용자는 CuBridge 클래스에서 연산 흐름을 선언하지만, 실제 연산은 네이티브 환경에서 수행되므로, 이를 안정적으로 중계하는 구조가 필요하다. CuBridgeJNI는 모든 JNI 호출을 내부적으로 감추고, 사용자는 선언적 API만을 통해 연산을 구성할 수 있도록 설계되어 있다. 이를 통해 JNI 문법이나 시스템 의존 설정을 고려하지 않고도 고성능 연산을 쉽게 사용할 수 있다.

이 클래스는 다음과 같은 주요 네이티브 함수 등을 통해 연산 명령을 전달한다.

```
static native void clear();
static native boolean put(double[] data, int[] shape, int dataLen,
        int shapeLen, int usageNum, String name, boolean isBroad);
static native boolean pop(String name);
static native boolean abs(String a, String out);
static native boolean add(String a, String b, String out);
```

이 함수들은 모두 CuBridge 내부에서 자동으로 호출되며, 사용자가 직접 접근할 필요는 없다.

JNI의 함수들은 모두 CuBridge 클래스 내부에서 알맞게 맵핑되어, 고수준 API 호출만으로도 네이티브 연산이 자동으로 실행되며, 전체 흐름은 투명하고 유연하게 처리된다.

CuBridge의 또 다른 특징은 동적 DLL 로딩 자동화 구조이다. 일반적으로 JNI 기반 시스템은 IDE에서 DLL 파일 경로를 직접 지정하거나 환경 변수 등록이 필요하지만, CuBridge는 jar 내부에 포함된 DLL 파일을 실행 시 자동 추출하고, 운영체제별 임시 폴더에 저장한 뒤 동적으로 로딩하는 방식을 채택하고 있다.

초기 실행 과정은 다음과 같이 구성된다.

1. **CuBridge의 jar 파일에서 백엔드 DLL 자동 추출**
2. **운영체제 임시 폴더에 저장**
3. **System.load()를 이용해 DLL 로딩**
4. **JNI를 통해 C++ 연산 함수들과 연결**

이 구조는 플랫폼 독립성과 사용자 편의성을 모두 확보하는 데 크게 기여한다. 개발자는 별도의 설치나 환경 설정 없이 동일한 jar 파일 하나를 import 하는 것 만으로 다양한 환경에서 CuBridge를 실행할 수 있으며, 이식성과 유지보수 측면에서도 높은 유연성을 제공한다.

결과적으로 CuBridgeJNI는 단순한 JNI 호출 클래스에 그치지 않고, CuBridge 전체 구조의 유연성과 이식성을 실질적으로 뒷받침하는 핵심 계층이라 할 수 있다. Java와 CUDA 사이의 경계를 자연스럽게 연결함으로써, 사용자는 Java의 객체지향적 구조를 유지한 채로도 GPU 기반 고속 연산을 효과적으로 수행할 수 있다.

5. CuBridge 백엔드

5.0 JNI 링크

Java 측에서 CuBridgeJNI 클래스가 네이티브 연산 호출을 담당하듯, C++ 측에도 이를 대응하는 JNI 함수들이 존재한다. 이 JNI 링크 계층은 Java에서 전달된 데이터를 C++의 내부 형식에 맞게 변환하고, 해당 연산에 대응하는 백엔드 함수를 호출하는 역할을 수행한다. 결과적으로 이 계층은 CuBridge의 모든 입력과 출력을 중계하는 핵심 통신 창구라 할 수 있다.

해당 계층은 다음과 같은 절차를 통해 프론트엔드와 백엔드를 연결한다:

1. 자료형 변환

Java에서 전달된 배열 및 객체는 JNI를 통해 C++의 자료형과 포인터 형식으로 변환되며, 연산 종료 후 결과는 다시 Java 형식으로 역변환된다. 이를 통해 양 방향의 데이터 흐름이 원활히 이루어진다.

2. 연산 함수 중계

프론트엔드에서 호출되는 연산 함수는 add(), sub(), abs() 등 직관적인 이름을 사용하지만, 백엔드에서는 내부적으로 binaryFunc(), unaryFunc() 등의 범용 함수로 처리된다. JNI 링크 계층은 이들 함수 간의 매핑을 수행하여, 알맞은 연산이 실행되도록 중계한다.

3. DLL의 진입점 역할

해당 JNI 계층은 DLL 수준에서의 메인 진입점 역할을 수행한다. 프론트엔드로부터 전달된 모든 입력은 이 계층을 통해 큐 및 버퍼로 전달되며, 백엔드에서 생성된 출력 또한 이 계층을 통해 Java로 반환된다. 따라서 CuBridge의 실행 흐름은 항상 이 JNI 링크 계층을 중심으로 시작되고 종료된다.

이러한 구조는 Java와 C++ 사이의 명확한 계층 분리를 유지하면서도, 높은 수준의 연산 성능과 구조적 유연성을 동시에 확보하는 기반이 된다.

5.1 큐 기반 텐서 관리

CuBridge의 백엔드 연산은 TensorQueue와 TensorBuffer라는 두 개의 자료 구조를 중심으로 순환적으로 이루어진다. 본 논문에서는 이를 각각 큐와 버퍼로 지칭한다.

프론트엔드에서 텐서가 삽입되거나, 연산을 통해 새로운 결과가 생성되면 해당 텐서는 우선적으로 큐에 저장된다. 이때 동일한 이름의 텐서가 이미 존재하는 경우에는 중복 등록이 자동으로 방지되며, 사용자가 이름을 명시하지 않은 경우에는 시스템이 내부적으로 고유한 임시 이름을 생성하여 부여한다.

연산 함수는 큐에 저장된 텐서에 직접 접근하지 않고, 필요한 텐서를 우선적으로 버퍼로 복사하여 사용한다. 이 구조는 연산 도중 예기치 않은 중단이나 중복 접근으로부터 원본 데이터를 보호하고, 함수 단위의 데이터 독립성과 실행 안정성을 확보하기 위해 설계되었다. 이러한 흐름은 이후 5.2절에서 상세히 설명한다.

5.2 실행 함수의 내부 8단계

CuBridge의 백엔드 연산은 모든 함수에서 동일한 8단계 절차에 따라 수행된다. 이 구조는 연산 흐름의 일관성을 확보하고, 디버깅 및 확장 과정에서도 높은 안정성과 예측 가능성을 제공하도록 설계되었다. 특히 각 단계 내부에서 환경에 따른 자동 분기가 함께 처리되기 때문에, 실행 함수는 단순한 순차 호출만으로도 텐서 연산을 안정적으로 수행할 수 있다. 상세한 도식은 부록 B에서 제시하며, 각 단계의 설명은 다음과 같다.

1. 입력 텐서 유무 확인

연산에 필요한 텐서가 큐에 존재하는지를 먼저 검사한다. 입력 이름이 명시되지 않은 경우에는 큐의 최상단에 위치한 텐서를 자동으로 선택하며, 선택된 텐서는 복사되어 복사본이 버퍼로 이동되고, 원본은 큐에서 삭제된다. 단, 해당 텐서의 usageCount 값이 2 이상인 경우에는 이후 다른 연산에서 재사용될 수 있다고 판단하여, 삭제하는 대신 usageCount 값을 1 감소시킨 뒤 원본을 다시 큐에 삽입한다.

2. 텐서 로드

1단계에서 선택된 텐서는 복사되어 버퍼에 저장되며, 이후 모든 연산은 큐가 아닌 버퍼를 대상으로 수행된다. 이는 연산 함수가 큐의 텐서를 직접 사용할 경우 발생할 수 있는 값의 손상이나 예기치 않은 변경을 방지하기 위한 조치이다. 또한 연산 도중 함수가 예외적으로 중단되거나 실패하더라도, 큐에 보관된 다른 원본 텐서들이 그대로 유지되므로 후속 연산에서의 오류를 최소화할 수 있다.

이러한 이유로 연산 함수와 큐 사이에 중간 버퍼를 두고 복사하는 구조를 취하며, 이 단계에서는 버퍼에 저장된 텐서의 데이터, shape 등 각 정보를 함수 내부의 지역 변수로 순차적으로 불러와 활용한다.

3. 연산 조건 확인

각 연산에 따라 입력 텐서가 요구하는 조건을 만족하는지를 먼저 검사한다. 예를 들어, 내적(dot product)의 경우 두 텐서의 특정 차원이 일치해야 하며, 이항 연산에서는 각 축

의 매칭 여부와 브로드캐스팅 가능성을 함께 판단한다. 특히 CuBridge는 NumPy와 달리, 입력 텐서 간에 N배수 크기 차이가 존재하더라도 브로드캐스팅이 가능하도록 설계되어 있어 보다 유연한 연산 구성이 가능하다.

이 단계에서는 이러한 조건 검사를 바탕으로 필요한 경우 자동 브로드캐스팅을 수행하며, 이를 통해 이후 연산의 안정성과 일관성을 확보한다.

4. 출력 텐서 크기 계산

입력 텐서의 크기와 연산 조건을 바탕으로, 출력 텐서의 shape을 사전에 계산한다. CuBridge는 텐서의 실제 데이터와 별개로 축 정보를 독립적으로 관리하기 때문에, 연산에 앞서 정확한 shape 계산 과정이 필수적으로 요구된다. 단항 및 이항 연산의 경우 일반적으로 입력과 동일한 shape 또는 브로드캐스팅을 거친 크기를 갖지만, 행렬 연산이나 축 연산처럼 축 구조가 변경되는 연산에서는 별도의 shape 계산 로직이 필요하다.

또한 이 단계에서는 shape 외에도 연산에 필요한 각종 부가 정보(출력 원소 수, 축별 연산 단위 등)도 함께 계산되어, 이후 단계에서 연산이 안정적으로 수행될 수 있도록 준비된다.

5. 메모리 적재 (loadMemory)

연산에 필요한 텐서 데이터는 적절한 메모리 영역(CPU 또는 GPU)으로 복사된다. 이 과정은 시스템 환경을 자동으로 감지하여, 실행 대상에 따라 메모리 경로를 분기 처리한다. 이에 대한 구체적인 분기 로직은 5.3절에서 다룬다.

6. 연산 수행

적재된 데이터를 바탕으로 실제 연산이 수행된다. 이 단계에서는 환경에 따라 설정된 함수 포인터가 호출되며, 해당 포인터는 CUDA 커널 또는 CPU 연산 함수 중 하나를 실행한다. 실행 환경에 따른 연산 흐름의 차이 역시 5.3절에서 자세히 설명한다.

7. 메모리 회수 (pickMemory)

연산이 완료된 결과는 다시 RAM으로 복사되어 내부 텐서 형태로 회수된다. 이때 연산 환경에 따라 출력 텐서의 위치와 복사 경로가 달라지며, 정확한 회수 방식은 GPU 또는 CPU 메모리 전략에 따라 분기된다. 관련 내용은 5.3절에서 구체적으로 서술한다.

8. 출력 저장 및 메모리 정리

연산을 통해 생성되어 RAM으로 복사된 출력 텐서는, 최종적으로 축 정보 등 부가 데이터와 함께 전역 큐에 저장된다. 이때 출력 텐서의 이름이 기존에 존재하는지 확인하며, 동일한 이름이 이미 큐에 있을 경우에는 shape를 비교하여 덮어쓰기가 가능한지를 판단한다. 만약 크기가 일치할 경우, 기존 값을 대체하거나 누적하는 방식으로 처리할 수 있으며, 이를 통해 += 형태의 연산 흐름도 유연하게 구현 가능하다.

모든 출력이 큐에 정상적으로 등록된 이후에는, 함수 내부에서 사용된 지역 변수와 임시 메모리들을 정리하여 메모리 누수가 발생하지 않도록 하고, 해당 연산 함수의 실행을 종료한다.

이와 같은 8단계 흐름은 CuBridge의 모든 연산 함수에 일관되게 적용되며, 각 단계는 독립적이고 예외 안전하게 구성되어 있어 시스템 전체의 안정성과 디버깅 용이성을 크게 향상시킨다.

또한, 전체 실행 흐름에는 플래그 기반의 제어 구조가 함께 적용되어 있어, 연산 중 어느 한 단계에서 오류가 발생하더라도 이후 단계는 자동으로 우회(bypass) 처리된다. 이 경우 시스템은 연산 중단 후 즉시 오류를 보고하고, 관련 메모리를 정리하며, 큐 내 변수 상태를 복구하여 전체 구조의 일관성을 유지하도록 설계되어 있다. 이러한 예외 대응 방식은 CuBridge가 신뢰성 있는 연산 플랫폼으로 기능할 수 있는 중요한 기반이 된다.

5.3 환경에 따른 연산 분기

CuBridge는 GPU 연산을 지원함과 동시에, CUDA 환경이 없는 시스템에서도 동일한 연산 흐름을 유지할 수 있도록 설계되었다. 이를 위해 시스템 초기화 시점에 운영체제에 설치된 CUDA 런타임 DLL의 존재 여부를 먼저 확인하고, 그 결과를 바탕으로 연산 환경(GPU/CPU)을 자동으로 분기하는 구조를 갖는다.

가장 먼저 isCudaSupport() 함수는 atomicAdd()가 지원되는 CUDA 11.0부터 12.8까지의 다양한 버전을 대상으로, 해당 CUDA 런타임 DLL이 설치되어 있는지를 검사한다. 이 중 하나라도 성공적으로 로드되면 CuBridge는 CUDA 연산이 가능한 환경이라고 판단하고, 다음 단계로 진입한다.

하지만 CUDA 런타임이 존재한다고 해서 GPU가 실제 연산에 적합한지는 확정할 수 없다. 따라서 이후에는 GetProcAddress()를 통해 cudaGetDeviceCount 등 디바이스 정보 질의 함수들의 포인터를 직접 바인딩하고, 이를 실행하여 연산 가능한 디바이스가 실제로 존재하는지를 점검한다. 장치가 없거나 조건에 맞지 않거나, 해당 함수 호출에 실패할 경우 CuBridge는 GPU 사용이 불가능하다고 판단하고 자동으로 CPU 모드로 전환된다.

반면, CUDA 환경이 정상적으로 감지되면 CuBridge는 CuBridgeCudaC.dll을 로드하고, 연산에 공통적으로 사용되는 핵심 함수들을 GetProcAddress()를 통해 동적으로 불러와 포인터에 저장한다.

이때 로딩되는 함수들과 그 역할은 다음과 같다.

1. **_cudaAlloc**: GPU 메모리 영역에 텐서를 위한 공간을 동적으로 할당한다.
2. **_cudaCopy**: GPU와 CPU 간에 데이터를 복사한다.
3. **_cudaFree()**: 사용된 GPU 메모리를 해제하여 누수를 방지한다.
4. **_getCudaInfo**: GPU 디바이스의 성능, 메모리 용량 등의 정보를 조회한다.

이 과정이 모두 완료되면 내부 플래그인 isGPU가 1로 설정되며, GPU 기반 연산이 활성화된다.

이후 updateComputePolicy() 함수가 호출되며, 현재 환경 설정에 따라 연산 함수 포인터의 바인딩이 이루어진다. GPU 모드일 경우 GpuInit()과 GpuBind()가 호출되어, CUDA 커널 함수 포인터들이 CuBridgeFunc 내부의 맵에 저장된다.

반대로 GPU 모드가 비활성화된 경우에는 CpuInit()과 CpuBind()를 통해 동일한 연산 이름들에 대해 CPU용 함수 포인터가 바인딩된다. 이 구조를 통해 실제 연산 수행 시에는 단순히 해당 맵을 호출하는 방식만으로 GPU/CPU 커널 전환이 가능하다.

이처럼 CuBridge는 GPU와 CPU 간의 전환을 단순한 예외 처리 방식이 아닌, 초기 감지 단계에서 환경 상태에 따라 전용 초기화 루틴을 분기 호출하고, 그 결과에 따라 함수 포인터를 바인딩하는 구조로 설계되어 있다. 이러한 환경 판단 및 모드 설정이 완료되면, CuBridge의 모든 연산 함수는 5.2절에서 설명한 고정된 8단계 실행 절차에 따라 일관되게 실행된다. 그 내용은 다음과 같다.

5. 메모리 적재 (loadMemory)

연산에 필요한 텐서 데이터는 현재 시스템 환경에 따라 적절한 메모리 영역(CPU 또는 GPU)으로 복사된다. GPU 모드에서는 데이터를 VRAM으로 전송하고, CPU 모드에서는 전송 과정을 생략한 채 RAM 상에서 직접 데이터를 처리한다. 이 과정은 환경 감지 결과를 기반으로 자동 분기되며, 모든 연산 수행 직전에 공통적으로 호출된다.

6. 연산 수행

메모리에 적재된 데이터를 바탕으로 실제 연산이 수행된다. 이 단계에서는 환경에 따라 사전에 설정된 함수 포인터가 호출되며, 해당 포인터는 CUDA 커널 또는 CPU 연산 함수 중 하나를 실행한다. 연산의 종류(unary, binary, axis 등)에 따라 알맞은 포인터가 자동으로 연결되어 실행되며, 예를 들어 단항 연산의 경우 다음과 같이 호출된다.

(*funcUnary)[funcName](a, out, o_dlen)

이처럼 연산 수행은 연산 타입에 따라 이미 바인딩 된 맵에서 포인터를 검색하여 실행되므로, 환경에 관계없이 동일한 방식으로 처리된다.

7. 메모리 회수 (pickMemory)

연산이 완료되면 그 결과는 다시 RAM으로 복사되어 CuBridge 내부에서 사용하는 텐서 형식으로 회수된다. GPU 모드에서는 VRAM에서 RAM으로의 결과 전송이 이루어지고, CPU 모드에서는 별도의 전송 없이 즉시 결과를 사용할 수 있다. 이 구조 덕분에 모든 연산 함수는 동일한 처리 순서를 따를 수 있으며, 환경 차이와 무관하게 코드의 일관성과 안정성이 유지된다.

결과적으로 CuBridge의 모든 연산 함수는

환경 감지 -> 초기화 -> 함수 포인터 바인딩 -> load -> func -> pick

의 일관된 실행 흐름을 따른다. 이를 통해 사용자 코드는 GPU 또는 CPU 환경 여부를 전혀 신경 쓰지 않고도 동일한 연산 함수를 사용할 수 있으며, 환경 분기에 따른 내부 처리를 CuBridge가 자동으로 책임지는 구조가 완성된다.

이와 같이 CuBridge의 백엔드는 환경에 따른 자동 분기, 안정적인 텐서 관리, 일관된 연산 흐름 구조를 통해 고성능 GPU 연산과 범용 CPU 연산 모두를 유연하게 지원한다. 특히 초기화 단계에서의 정밀한 환경 감지와 함수 포인터 기반의 동적 연산 바인딩 구조

는, 사용자가 복잡한 시스템 조건을 직접 고려하지 않아도 되도록 하여, 실용성과 이식성을 동시에 만족시키는 핵심 기반이 된다. 이러한 구조는 단순한 수치 연산을 넘어 신경망 모델이나 대규모 계산에도 안정적인 백엔드로 기능할 수 있는 강력한 토대를 제공한다.

6. 실험 및 비교

6.0 실험 배경

CuBridge는 Python의 NumPy를 기반으로 연산 흐름과 인터페이스 구조를 설계하였다. 특히 텐서 생성, 브로드캐스팅, 함수 호출 방식 등에서 NumPy의 직관적인 설계를 참고하였으며, 이는 전체 시스템의 선언적 구조와 사용자 친화적인 연산 구성에 큰 영향을 주었다. 그러나 CuBridge의 목표는 NumPy를 단순히 모방하거나 대체하는 데 있지 않으며, Java 환경 내에서 GPU 병렬 연산이 가능한 독립적 대안을 구축하는 데 있다.

한편, Java 생태계에서 GPU 연산을 지원하는 수치 연산 프레임워크는 매우 제한적이며, 대표적으로 JCuda와 ND4J가 존재했으나 두 라이브러리 모두 최신 CUDA 버전에 대한 지원이 중단되었고, 공식 개발 또한 사실상 종료된 상태이다. 예를 들어, JCuda는 CUDA 10.1 이하 환경에서만 작동하며, ND4J 역시 구버전에서 개발이 중단되어 CUDA 연동 기능은 현재 정상적인 실행을 보장하지 않는다. 이로 인해 기존 Java 기반 GPU 연산 라이브러리의 공백은 심화되고 있으며, CuBridge는 이러한 공백을 실질적으로 채울 수 있는 새로운 대안으로 제안된다.

이러한 배경을 바탕으로 본 실험은 다음과 같은 세 가지 비교군을 설정하였다.

1. NumPy, CuPy: Python 기반의 대표적인 수치 연산 라이브러리로, CuBridge의 설계 참고 기준으로 사용되며, 실험 1에서는 연산 구조 및 선언 방식의 기준군(reference group)으로, 실험 2와 3에서는 대조군으로 간주한다.

2, JCuda: Java에서 GPU 연산을 지원했던 고전적 방식의 라이브러리로, CuBridge와 가장 유사한 목적을 가졌던 대조군(comparison group)이지만, 실제 실행 가능성은 제한적이다.

3. CuBridge: 본 논문에서 제안하는 실험군으로, Java 기반의 선언형 인터페이스, 자동 백엔드 전환 구조, 고정된 연산 흐름 등을 평가 대상으로 삼는다.

본 논문에서는 다음의 네 가지 실험을 통해 CuBridge의 실용성과 구조적 우위를 검증하고자 한다.

1 코드량 및 구조 비교
2 대규모 연산 성능 측정
3 브로드캐스트 처리 능력 비교
4 실질 연계 연산 성공 가능성 확인

6.1 코드 구성 비교

먼저 NumPy, JCuda, CuBridge의 연산 구성 방식을 비교하였다. 동일한 행렬 내적 연산을 예시로 하여, 각 프레임워크가 연산 흐름을 구현하는 방식의 차이를 분석하였다. 비교 항목으로는 전체 코드량, 추상화 수준, 함수 호출 방식, 그리고 코드의 선언적 구조 여부 등을 설정하였다. 세 가지 방식의 코드 예시는 아래 표와 같다.

<table>
<tr><th>Numpy</th><th>Jcuda</th><th>CuBridge</th></tr>
<tr><td rowspan="2">import numpy as np

a = np.random.rand(128, 64)
b = np.random.rand(64, 32)
out = np.dot(a, b)
print(out)</td><td rowspan="2">import jcuda.*;
import jcuda.runtime.*;

int m = 128, k = 64, n = 32;
float[] hostA = new float[m * k];
float[] hostB = new float[k * n];
float[] hostC = new float[m * n];
for (int i = 0; i < hostA.length; i++)
hostA[i] = (float)Math.random();
for (int i = 0; i < hostB.length; i++) hostB[i] = (float)Math.random();

Pointer dA = new Pointer(), dB = new Pointer(), dC = new Pointer();
cudaMalloc(dA, m * k * Sizeof.FLOAT);
cudaMalloc(dB, k * n * Sizeof.FLOAT);
cudaMalloc(dC, m * n * Sizeof.FLOAT);

cudaMemcpy(dA, Pointer.to(hostA), m * k * Sizeof.FLOAT, cudaMemcpyHostToDevice);
cudaMemcpy(dB, Pointer.to(hostB), k * n * Sizeof.FLOAT, cudaMemcpyHostToDevice);

// 행렬 곱 커널 실행 코드 생략 (보통 cublasSgemm 또는 사용자 정의 커널 별도 필요)

cudaMemcpy(Pointer.to(hostC), dC, m * n * Sizeof.FLOAT, cudaMemcpyDeviceToHost);

System.out.println(Arrays.toString(Arrays.copyOf(hostC, 10)));</td><td>CuBridge cb = CuBridge.getInstance();

Tensor A = Tensor.rand(new int[] {128, 64});
Tensor B = Tensor.rand(new int[] {64, 32});

cb.put(A, "a").put(B, "b");
cb.dot("a", "b", "c");

Tensor C = cb.get("c");

C.printData();</td></tr>
<tr><td>최대한 간결히 한 경우</td></tr>
<tr><td colspan="2"></td><td>CuBridge cb = CuBridge.getInstance();

Tensor A = Tensor.rand(new int[] {128, 64});
Tensor B = Tensor.rand(new int[] {64, 32});

cb.put(A).put(B)
.dot()
.get()
.printData();</td></tr>
</table>

이와 같이 CuBridge는 JCuda에 비해 훨씬 간결한 구문으로 연산 흐름을 표현할 수 있었으며, 특히나 변수 생략 및 복잡한 메모리 할당과 커널 직접 호출 과정을 추상화함으로써 개발자 편의성을 크게 높였다. 또한 NumPy에 비해서도 Java 언어의 정적 타입 구조를 유지하면서도 선언형 스타일을 구현할 수 있어, 전체적인 구성 간결성과 직관성 측면에서 유사한 수준을 달성하였다. 이로써 CuBridge는 Java 기반 환경에서도 Python 수준의 생산성을 확보할 수 있음을 보여준다.

6.2 각 라이브러리 별 내적 연산 속도

본 실험은 동일한 크기를 갖는 두 개의 1차원 배열에 대해 내적 연산(dot product) 을 수행하여, CuBridge의 대규모 수치 연산 성능을 NumPy, CuPy, 그리고 순수 Java와 비교하기 위해 설계되었다. 단순한 덧셈이 아닌 내적을 선택한 이유는, 이 연산이 메모리 접근과 곱셈·합산을 동시에 포함하고 있어, 연산 최적화 구조의 실제 효과를 보다 명확하게 측정할 수 있기 때문이다.

비교군으로는 Python 기반의 NumPy, GPU 가속을 지원하는 CuPy, Java의 순수 반복문을 활용한 구현(Java for-loop), 그리고 본 논문에서 제안하는 CuBridge가 포함되며, 각 시스템에서 동일한 1억 원소의 배열에 대해 내적을 수행한다.

참고용으로, JCuda 기반 내적 구현도 커널 코드 및 전체 흐름을 별도로 작성하여 코드 비교 대상으로는 제시하였으나, 해당 라이브러리는 CUDA 10.1 이하에서만 동작하며 최신 CUDA 환경에서는 실행이 불가능한 경우가 많아, 실제 성능 측정에서는 제외하였다.

실험에 사용된 코드와 결과는 아래와 같으며, 모든 구현은 연산 정확성과 동일 조건을 보장하도록 구성되었다.

실험 코드는 다음과 같다.

1. Jcuda에서의 연산 코드

AddKernel.cu	Jcuda

extern "C"
__global__ void dotArrays(const float* A, const float* B, float* C, int size) {
  __shared__ float temp[256];
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  int tid = threadIdx.x;
 	
  temp[tid] = (idx < size) ? A[idx] * B[idx] : 0;
 	
  __syncthreads();
 	
  // 병렬 감소
  for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
  if (tid < stride) temp[tid] += temp[tid + stride];
  __syncthreads();
  }
 	
  if (tid == 0) atomicAdd(C, temp[0]);
}
//이후 nvcc 커맨드로 컴파일하여 .ptx로 실행
	int n = 100_000_000;
	int size = n * Sizeof.FLOAT;
	
	// JCuda 초기화
	JCudaDriver.setExceptionsEnabled(true);
	cuInit(0);
	CUdevice device = new CUdevice();
	cuDeviceGet(device, 0);
	CUcontext context = new CUcontext();
	cuCtxCreate(context, 0, device);
	
	// 입력 배열 생성
	float[] hostA = new float[n];
	float[] hostB = new float[n];
	for (int i = 0; i < n; i++) {
	  hostA[i] = i;
	  hostB[i] = i;
	}
	
	// GPU 메모리 할당
	CUdeviceptr devA = new CUdeviceptr();
	CUdeviceptr devB = new CUdeviceptr();
	CUdeviceptr devC = new CUdeviceptr();
	cuMemAlloc(devA, size);
	cuMemAlloc(devB, size);
	cuMemAlloc(devC, Sizeof.FLOAT); // 결과는 float 하나만!
	
	// 데이터 복사 (Host -> Device)
	cuMemcpyHtoD(devA, Pointer.to(hostA), size);
	cuMemcpyHtoD(devB, Pointer.to(hostB), size);
	cuMemsetD32(devC, 0, 1); // 결과 초기화
	
	// 커널 로드 및 함수 찾기
	CUmodule module = new CUmodule();
	cuModuleLoad(module, "AddKernel.ptx");
	CUfunction function = new CUfunction();
	cuModuleGetFunction(function, module, "dotArrays");
	
	// 커널 실행 설정
	int blockSize = 256;
	int gridSize = (n + blockSize - 1) / blockSize;

```
Pointer kernelParameters = Pointer.to(
    Pointer.to(devA),
    Pointer.to(devB),
    Pointer.to(devC),
    Pointer.to(new int[]{n}));

// 커널 실행
cuLaunchKernel(function,
    gridSize, 1, 1,
    blockSize, 1, 1,
    0, null, kernelParameters, null);
cuCtxSynchronize();

// 결과 복사 (Device -> Host)
float[] result = new float[1];
cuMemcpyDtoH(Pointer.to(result), devC, Sizeof.FLOAT);

System.out.println("Dot Product: " + result[0]);

// 메모리 해제
cuMemFree(devA);
cuMemFree(devB);
cuMemFree(devC);
```

2. 실제 실험 코드

Numpy

```
import numpy as np


A = np.random.rand(10000, 10000)
B = np.random.rand(10000, 10000)
C = np.dot(A, B)
```

Cupy

```
import cupy as cp


A = cp.random.rand(10000, 10000)
B = cp.random.rand(10000, 10000)
C = cp.dot(A, B)
cp.cuda.Device(0).synchronize()
```

Java

```
double[] arrA = A.toArray();
double[] arrB = B.toArray();

int N = 10_000;
double[][] matA = new double[N][N];
double[][] matB = new double[N][N];
double[][] matC = new double[N][N];

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        matA[i][j] = arrA[i * N + j];
        matB[i][j] = arrB[i * N + j];
    }
}

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        double sum = 0;
        for (int k = 0; k < N; k++)
            sum += matA[i][k] * matB[k][j];
        matC[i][j] = sum;
    }
}
```

CuBridge

```
CuBridge cb = CuBridge.getInstance();

Tensor A = Tensor.rand(new int[] {10_000, 10_000});
Tensor B = Tensor.rand(new int[] {10_000, 10_000});

cb.put(A).put(B);
cb.dot();
cb.get();
```

3. 실험 결과(단위:ms)

순번	종류	전체시간	연산시간
1	Numpy	7546	6493
2	Cupy	23355	17067
3	CuBridge	40159	35723
4	Java	300000+	300000+

4. 실험 결과 분석

NumPy는 CPU 기반임에도 불구하고 가장 빠른 연산 속도를 보였다. 이는 내부적으로 고도로 최적화된 BLAS(C 기반) 루틴을 사용하며, 배열 연산에서 별도의 메모리 복사 없이 바로 연산이 가능한 구조 때문이다.

CuPy는 GPU를 사용하는 라이브러리로, 연산 자체는 빠르지만 GPU 메모리와 RAM 간의 입출력 복사 비용으로 인해 전체 시간에서 손해를 보는 구조를 보였다. 연산 시간만 따졌을 경우 CuBridge보다 빠르지만, 초기화 및 전송 오버헤드가 전체 성능에 영향을 미쳤다.

CuBridge는 Java 기반으로 구현된 시스템임에도 불구하고, GPU 연산을 안정적으로 수행하며 상당히 준수한 연산 속도를 보였다. 특히 연산 시간은 CuPy보다 약간 느리지만, Java의 구조적 제약과 JNI 연동 및 내부 함수 절차를 감안하면 GPU 연산을 성공적으로 통합한 사례로 평가할 수 있다.

반면, 순수 Java의 3중 for문 기반 내적 연산은 10분이 넘어도 종료되지 않는 수준으로, 비교 자체가 어려울 정도로 비효율적이었다. 이는 Java의 배열 연산 최적화 부재와 단일 스레드 기반 구조의 한계를 보여준다.

결론적으로, CuBridge는 Java 환경에서 GPU 병렬 연산을 안정적으로 수행할 수 있는 현실적인 대안으로 작동하며, Python 기반의 수치 연산 라이브러리들과 비교해도 선언적 구조 및 인터페이스 측면에서 경쟁력을 갖추고 있음을 보여주었다.

6.3 브로드캐스팅 비교

이 절에서는 CuBridge의 브로드캐스트 기능이 NumPy 대비 어떤 구조적 차이를 가지며, 연산 적용 범위가 어떻게 확장되었는지를 실험을 통해 확인하고자 한다.

NumPy는 기본적으로 각 축의 크기가 일치하거나, 크기 1인 축에 한해 자동 확장을 허용한다. 하지만, 이 구조는 N 배수 형태의 축 차이를 가진 경우에는 연산이 불가능하다는 한계가 있다.

이에 반해 CuBridge는 자체 구현한 브로드캐스트 로직을 통해, N 배수 관계의 축이라면 자동으로 확장하여 연산할 수 있도록 설계되어 있다. 이는 다양한 입력 조합에 대해 더 유연한 텐서 연산을 지원하게 하며, 복잡한 다축 구조를 가진 실제 데이터에서도 활용 가능성을 넓힌다.

이를 검증하기 위해, NumPy에서 연산이 불가능한 (2, 1, 6)과 (1, 4, 2) 형태의 텐서를 사용하여 CuBridge가 정상적으로 연산을 수행할 수 있는지를 실험하였다.

실험 코드는 다음과 같다.

Numpy	CuBridge

import numpy as np
 	
a = np.ones((2, 1, 6))
b = np.ones((1, 4, 2))
c = a + b # ValueError 발생
print(c) # 실행 불가
	Tensor A = Tensor.ones(new int[]{2, 1, 6});
	Tensor B = Tensor.ones(new int[]{1, 4, 2});
	
	cb.put(A).put(B)
	  .add() // 브로드캐스팅 성공
	  .get()
	  .printData(); // (2, 4, 6) 으로 브로드캐스트된 결과 출력

실험 결과, 넘파이에서는

[ValueError: operands could not be broadcast together with shapes (2,1,6) (1,4,2)]

에러가 발생하여 실행이 중단되었다. 이는 NumPy의 브로드캐스팅 규칙이 1을 기준으로 한 축 확장만을 허용하기 때문이며, N배수 크기에 대해서는 연산을 지원하지 않음을 보여준다.

반면, CuBridge에서는 동일한 입력에 대해 연산이 정상적으로 완료되었으며, 결과로

```
Tensor(shape=[2, 4, 6]):
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
[  2.000,  2.000,  2.000,  2.000,  2.000,  2.000 ]
```

가 출력되었다. 이는 실제로 (2, 1, 6)과 (1, 4, 2)의 입력이 내부적으로 (2, 4, 6)으로 브로드캐스트되어 연산이 수행되었음을 의미하며, CuBridge의 확장된 브로드캐스트 구조가 실질적으로 동작함을 입증한다.

6.4 선형 회귀 모델 구현

본 실험에서는 CuBridge만을 활용하여 선형 회귀(Linear Regression) 모델을 학습시키는 전체 과정을 구성하였다. 입력 텐서 생성, 가중치 초기화, 예측값 계산(forward), 손실(loss) 계산, 그리고 경사 하강법에 의한 파라미터 갱신(backward & update) 등, 모든 단계가 CuBridge의 연산 함수만으로 구현되었다.

외부 라이브러리나 머신러닝 프레임워크의 도움 없이도 완전한 학습 루프가 작성 가능하다는 점은, CuBridge가 단순한 수치 연산 모듈을 넘어 모델 학습을 위한 실행 엔진으로도 기능할 수 있음을 시사한다. 이는 CuBridge가 제공하는 연산 인터페이스의 표현력과 연산 흐름의 유연성을 실증적으로 보여주는 예라 할 수 있다.

실험 코드는 다음과 같다. 전체 상세 원문은 부록 E에서 다룬다.

CuBridge

```
CuBridge cb = CuBridge.getInstance();

//1. 데이터 생성
int N = 1000;
int D = 3;

Tensor X = Tensor.rand(new int[]{N, D});

//2. 정답 가중치와 편향 생성 및 정답 계산
Tensor Wt = Tensor.rand(new int[]{D, 1});
Tensor Bt = Tensor.rand(new int[]{1, 1});

cb.put(X, "X").put(Wt, "Wt").put(Bt, "Bt");
Tensor Yt = cb.affine("X", "Wt", "Bt", "Yt").get("Yt");

//3. 랜덤 가중치와 편향 생성
Tensor W = Tensor.rand(new int[]{D, 1});
Tensor B = Tensor.rand(new int[]{1, 1});

//7. 반복
for(int i = 0; i < 5000; i++){
            //4. 예측 계산
```

```
            cb.put(X, "X").put(W, "W").put(B, "B");
            Tensor Yp = cb.affine("X", "W", "B", "Yp").get("Yp");

            //5. MSE 비용 연산
            cb.put(Yp, "Yp").put(Yt, "Yt");
            cost = cb.mse("Yp", "Yt", "cost").get("cost");

            //6. 역전파
            cb.put(Yp, "Yp").put(Yt, "Yt")
                .put(X, "X").put(W, "W").put(B, "B")
                .put(0.01, "RATE", 2).put(N, "N");

            //6-1. 델타 계산
            cb.sub("Yp", "Yt", "delta").div("delta", "N", "delta").duple("delta", 2);

            //6-2. dw db 계산
            cb.transpose("X", "tX").dot("tX", "delta", "dw")
                .mul("dw", "RATE", "dw");

            cb.accumulate("delta", "db", 0)
                .mul("db", "RATE", "db");

            //6-3. 값 갱신
            cb.sub("W", "dw", "W").sub("B", "db", "B");

            W = cb.get("W");
            B = cb.get("B");
}
```

실험 결과는 다음과 같다.

순번	텐서	값
1	Step 0 W, B	Tensor(shape=[3, 1]): [0.111] [0.170] [0.988] Tensor(shape=[1, 1]): [0.368]
2	Step 5000 W, B	Tensor(shape=[3, 1]): [0.753] [0.668] [0.427] Tensor(shape=[1, 1]): [0.510]
3	Correct W, B	Tensor(shape=[3, 1]): [0.771] [0.682] [0.420] Tensor(shape=[1, 1]): [0.497]
4	Cost Before	Tensor(shape=[1]): [0.270]
5	Cost After	[0.000]

보는 바와 같이 5000회의 학습을 수행한 결과, 무작위로 초기화되었던 W와 B가 정답 Wt, Bt에 수렴하는 방향으로 조정되었고, 비용 함수 값 역시 약 0.27에서 0.000 이하로 감소하였다.

이는 해당 선형 회귀 문제에 대해 학습이 실질적으로 수행되었음을 의미하며, CuBridge가 고수준 연산 흐름의 조합만으로도 학습 가능한 알고리즘을 구현할 수 있음을 실험적으로 검증한 결과라 할 수 있다.

7. 설계 중점

7.0 CuBridge의 설계 철학 및 실행 구조

CuBridge는 단순한 수치 연산 도구를 넘어서, 대규모 텐서 기반 연산의 선언과 실행을 분리하고, 환경에 독립적인 계산 흐름을 제공하는 고수준 연산 프레임워크로 설계되었다. 이를 실현하기 위해, 전체 시스템은 다음의 두 가지 핵심 구조를 중심으로 구축되었다.

1. **지시-시행 알고리즘 (Directive-Execution Algorithm)**
2. **복사-독립 알고리즘 (Copy-Isolation Algorithm)**

이 장에서는 위 두 구조가 CuBridge 전체 시스템 내에서 어떻게 구현되고, 각각의 기술적 효과와 구조적 장점을 어떻게 실현하고 있는지를 설명한다.

7.1 지시-시행 알고리즘 (Directive-Execution Algorithm)

CuBridge는 전체 연산 흐름의 명확성과 구조적 유연성을 확보하기 위해, 선언과 실행을 분리한 지시-시행(Directive-Execution) 구조를 채택하였다. 이 구조는 '무엇을 할 것인가'에 대한 상위 선언(지시) 과, '어떻게 수행할 것인가'에 대한 하위 실행(시행) 을 명확히 구분함으로써, 복잡한 연산도 체계적이고 일관된 방식으로 구성할 수 있도록 한다.

CuBridge에서의 연산은 전역 큐와 버퍼에 저장된 텐서를 중심으로 수행된다. 사용자는 CuBridge 클래스의 상위 함수들을 통해 연산 이름과 입력 텐서의 이름만을 지정하면 되며, 이 호출은 단순히 실행 명령을 하위 계층에 전달하는 지시 역할을 한다.

실제 연산은 CuBridge 내부의 하위 함수가 담당한다. 하위 계층은 입력 텐서를 전역 큐에서 불러오고, 현재 환경(GPU 또는 CPU)에 따라 자동으로 분기하여, 지정된 연산을 적절한 경로에서 실행한다.

결과적으로 CuBridge는 상위 함수와 하위 함수, 프론트엔드와 백엔드의 역할을 명확히 나누며, 모든 연산이 지시 -> 시행 구조를 따르도록 설계되어 있어, 구조적 일관성과 실행 안정성을 동시에 확보한다.

이러한 구조는 다음의 설계 원칙에 기반한다.

1. 전역 흐름 관리

여러 함수에서 공유되는 데이터는 단순한 매개변수 전달이 아닌, 전역 큐와 버퍼 등 통합된 구조 안에서 관리되며, 전체 연산 흐름은 이 기반 위에서 일관되게 조정된다.

2. 상위 함수의 책임 제한

사용자 API 등 상위 함수는 연산을 직접 수행하지 않으며, 수행할 연산의 종류와 입력만을 명시적으로 선언하고 하위에 위임한다.

3. 기능 최소 단위화

모든 연산 기능은 가능한 한 작고 독립적인 단위로 구현되며, 이들을 조합함으로써 다양한 연산 흐름을 유연하고 확장성 있게 구성할 수 있도록 한다.

4. 구조의 계층화

단순한 명령-실행의 직선 구조를 넘어서, 상위 지시가 중간 단계의 조합 또는 전처리를 거쳐 하위 실행 함수로 연결되는 계층적 호출 체계를 갖춘다.

결과적으로 CuBridge는, 선언과 실행이 명확히 분리된 구조를 통해 함수 호출의 일관성을 유지하면서도, 다양한 실행 환경에 자동으로 적응하는 유연한 실행 구조를 구현하였다. 이러한 구조는 연산 흐름의 재사용, 기능 간 조합, 환경 독립성 확보 등 여러 측면에서 시스템의 확장성과 안정성을 동시에 보장하는 핵심 메커니즘으로 작용한다.

7.2 복사-독립 알고리즘 (Copy-Isolation Algorithm)

CuBridge는 메모리 안정성, 디버깅 용이성, 그리고 연산 간 독립성을 확보하기 위해, 모든 연산 함수의 데이터 처리 흐름을 복사-독립(Copy-Isolation) 구조를 채택하였다. 이 구조는 각 함수가 사용하는 데이터를 철저히 복사하고 지역적으로 관리함으로써, 메모리 충돌이나 누수 문제를 방지하고 연산 흐름의 예측 가능성을 높이는 것을 목적으로 한다.

연산 함수는 큐 또는 버퍼에서 필요한 텐서를 복사한 뒤, 해당 복사본을 지역 변수로 사용하여 연산을 수행한다. 이 과정에서 원본 텐서는 절대 직접 수정되지 않으며, 함수 종료 시 지역 변수는 자동으로 소멸한다. 다른 함수로 텐서를 전달하거나 반환할 경우에도 복사본이 사용되며, 이후 연산은 오직 이 복사본을 기준으로 이루어진다.

이러한 구조는 다음의 설계 원칙에 기반한다.

1. 지역 변수 우선 사용

함수 내부에서는 항상 입력 데이터를 복사하여 처리하며, 외부 데이터를 직접 참조하거나 수정하지 않는다.

2. 외부 전달 시 복사 시행

다른 함수로 직접 데이터를 넘길 때에는 반드시 복사본을 전달하며, 함수 내부의 지역 변수는 함수 종료 시 자동 정리된다.

3. 전역 공유 구조 제한

연산 흐름 간 공유가 필요한 경우에만 큐나 버퍼 등 전역 구조를 이용하고, 그 안에서도 유효성 검사를 통해 일관성을 유지한다.

4. 변수 수명의 체계적 연결

모든 변수는 자신이 생성된 함수 내에서 수명이 종료되며, 이후 필요한 경우 복사되어 다음 흐름으로 전달되는 '쇠사슬형 구조'를 통해 메모리 흐름을 형성한다.

이러한 구조는 다중 연산이 반복되거나 병렬 연산이 필요한 환경에서도 안정적으로 동작하며, 변수 오염이나 메모리 해제 오류를 원천적으로 차단한다. 또한 복사 흐름이 명시적이고 일관되기 때문에 디버깅과 테스트 과정에서도 큰 장점을 제공한다.

나아가, 빈번한 메모리 할당과 해제가 발생하는 상황에서도 메모리 누수 없이 안정적인 공간 관리를 가능하게 하여, 연산 규모가 커질수록 그 효과가 더욱 두드러진다.

8. 결론

8.1 CuBridge 기여 요약

CuBridge는 Java 언어 기반에서 GPU 병렬 연산을 직접 수행할 수 있도록 설계된 경량 수치 연산 프레임워크이다. 단순히 기존 CUDA 코드를 Java에서 호출하는 수준을 넘어서, Java 코드만으로도 고성능 연산 흐름을 선언하고 실행할 수 있도록 구성되어 있으며, GPU가 없는 환경에서도 동일한 구조로 작동할 수 있는 유연성을 갖춘 것이 핵심적인 특징이다.

사용자는 CuBridge를 통해 텐서를 생성하고 다양한 수치 연산을 수행할 수 있으며, 이 과정에서 메모리 주소나 커널 함수, 포인터 등의 복잡한 개념을 직접 다룰 필요가 없다. CuBridge는 이러한 내부 구현을 철저히 추상화하여, Java 개발자가 익숙한 객체지향 방식과 자동 메모리 관리를 그대로 유지한 채 GPU 연산을 사용할 수 있게 한다.

또한 연산의 구조는 put -> cal -> get이라는 명확한 흐름을 따르며, 각 연산은 자동 변수 명명, 내부 큐 관리, 환경 감지에 기반한 실행 경로 분기 등 일관된 체계 속에서 이루어진다. 특히 브로드캐스팅 지원 방식의 확장, 지시-시행 알고리즘, 복사-독립 메모리 구조와 같은 설계 방식은 시스템 전체의 안정성과 확장성을 높이는 중요한 기반이 되었다.

CuBridge는 딥러닝 프레임워크처럼 거대한 시스템은 아니지만, 단순한 수치 연산부터 신경망 학습의 기초까지 Java 코드만으로 구성할 수 있는 실용적인 실행 기반을 제공한다. 기존의 JCuda나 ND4J가 제공하지 못했던 직관성과 환경 독립성, 그리고 개발 난이도의 완화는 CuBridge의 가장 실질적인 기여로 평가된다.

결과적으로 CuBridge는 다음과 같은 역할을 수행한다.

1. Java 환경에서 CUDA의 성능을 활용할 수 있는 고성능 연산 프레임워크
2. 복잡한 메모리 제어나 포인터 없이도 GPU 연산을 구성할 수 있는 추상화된 계산 모델
3. 선언형으로 작성되는 연산 흐름과 유연한 연산 구성이 가능한 계산 인터페이스
4. CPU/GPU 환경에 자동 적응하여 동일한 코드로 실행 가능한 범용성
5. 지시-시행 및 복사-독립 구조를 통해 높은 유지보수성과 확장성 확보

이러한 기여를 통해 CuBridge는 Python 중심의 연산 생태계에 대응하는 Java 기반 대안으로서의 가능성을 실험하였으며, 수치 연산 도구로서 실질적인 확장성과 실용성을 갖춘 새로운 선택지로 제안하고자 한다.

8.2 기술 한계와 개선 방향

CuBridge는 Java 환경에서 GPU 연산을 효율적으로 수행할 수 있도록 설계된 경량 프레임워크이지만, 현재 버전(1.0)에는 다음과 같은 기술적 한계와 향후 개선 가능성이 존재한다. 다음은 주요 한계점과 이에 대한 개선 계획이다.

1. 연산 함수의 제한

현재 CuBridge에서 지원하는 연산은 개발자가 직접 구현한 단항, 이항, 축 기반, 행렬 연산 및 일부 신경망 연산에 국한된다. 수식 조합, 고차원 수치 해석, 복합 함수 등은 아직 지원되지 않는다.

이에 따라 향후 버전에서는 Java 측에서 사용자 정의 연산을 체인 형태로 조합할 수 있는 매크로 함수 기능을 도입하고, 다양한 수치 함수 및 통계 함수들을 내장함으로써 연산 커버리지를 점진적으로 확대할 계획이다.

2. GPU 자원 활용의 제약

현재 구조는 단일 GPU 환경을 기준으로 최적화되어 있으며, 하나의 연산에 전체 GPU 자원을 할당하는 방식으로 설계되어 있다. 이로 인해 다중 요청이나 대규모 병렬 연산 환경에서는 자원 효율성이 떨어질 수 있다. 또한, 현재 버전에서는 VRAM 용량을 초과하는 연산을 처리할 수 없기 때문에, 연산 입력 크기가 메모리 한계를 넘어서면 실행 자체가 불가능하다.

이를 보완하기 위해 향후에는 연산 단위 분할뿐만 아니라 메모리 적재 자체를 분할하여 순차 실행하는 구조를 도입할 계획이다. 아울러 GPU 자원 스케줄링, 멀티 GPU 분산 처리 구조 등도 함께 적용함으로써, 백엔드 커널 실행 및 메모리 관리 체계를 보다 유연하게 재설계할 예정이다.

3. 자동 미분 미지원

현재 CuBridge는 신경망 연산에 필요한 순전파(Forward) 계산만 지원하며, 역전파(Backward) 계산은 수동으로 구현해야 한다. 이는 모델 학습 자동화나 미분 기반 최적화에 제약을 주는 요소이다.

이를 해결하기 위해, 향후 Java 측에서 'Auto' 클래스 기반의 연산 추적(Tracking) 기능을 제공하여, 연산 기록 기반의 자동 미분 기능을 지원할 예정이다.

4. 상수 정의 및 재사용의 불편함

현재 CuBridge에서는 모든 수치 입력을 Tensor 형태로 명시적으로 선언해야 하며, 1, π, e와 같은 자주 사용하는 상수들도 매번 수동으로 생성해야 한다.

이를 보완하기 위해 향후 버전에서는 CuBridge 내부에 자주 쓰이는 상수를 미리 정의해 두어, 더욱 간편한 사용을 지원할 예정이다.

8.3 추후 개발 로드맵

CuBridge는 현재 1.1 버전까지 개발이 완료된 상태이며, 이후 버전은 병렬 처리, 서버 연산 위임, 분산 클러스터 등 점진적인 확장 구조를 지향하고 있다. 각 버전은 명확한 개발 목표를 갖고 있으며, 아래는 현재까지의 버전 및 계획된 주요 로드맵을 요약한 것이다.

순번	버전	목표	내용
1	beta	Java와 CUDA 간의 기본 연동 구조 검증	1. JNI 구조 설계 2. Tensor 및 CuBridge 클래스 들의 기초 구현 3. DLL 분리 및 기본 연산 함수 구성
2	1.0	기본 연산 및 실행 흐름 안정화	1. put -> cal -> get 선언형 연산 인터페이스 확립 2. 단항·이항·행렬·축 연산, Affine,Softmax,CEE 등 고수준 연산 포함 3. CudaC.dll 분리를 통한 환경적응 개선 4. 자동 환경 감지 기반 CPU/GPU 전환 개선 5. 큐 및 버퍼 기반 텐서 관리 구조, 함수 실행 8단계 정립
3	1.1	신경망 연산 준비 및 전처리 기능 확장	1. im2col, col2im, reshape 등 CNN 연산 보조 함수 추가 2. rad2deg, deg2rad 등 수학 유틸리티 함수 추가 3. CSV 기반 문자열 텐서 로딩 기능 및 정규화 기능 지원
예정 (상세 로드맵은 부록 D에서 제시)			
4	2.0	완전 병렬 연산 및 다중 GPU 확장	1. 거대 연산을 블록 및 스레드, vram 적재 수준에서 병렬 분할 2. 연산 스케줄러 도입: 입력 크기 기반 연산 조각 자동 분배 3. 다중 GPU 환경에서의 연산 분산 및 결과 통합 처리 4. 커널 호출 동기화 및 GPU-CPU 전송 최적화
5	3.0	연산 처리를 서버에 위임할 수 있는 구조 구축	1. 연산 요청을 클라이언트에서 서버로 전송하고 결과만 수신 2. JSON 기반 REST 인터페이스 도입 3. 서버 내 텐서 캐싱 및 요청 큐 최적화 구조 4. 다중 클라이언트와 서버간 연결 지원
6	4.0	분산 노드 기반 클러스터 연산 구조 확장	1. 텐서 단위 분할 후 다중 노드에 작업 분배 2. DAG 기반 네트워크상 연산 그래프 처리 3. 노드 간 동기화, 연산 병합, 오류 복구 구조 내장 4. 클러스터 구조를 통한 그리드 네트워킹 지원

8.4 결론

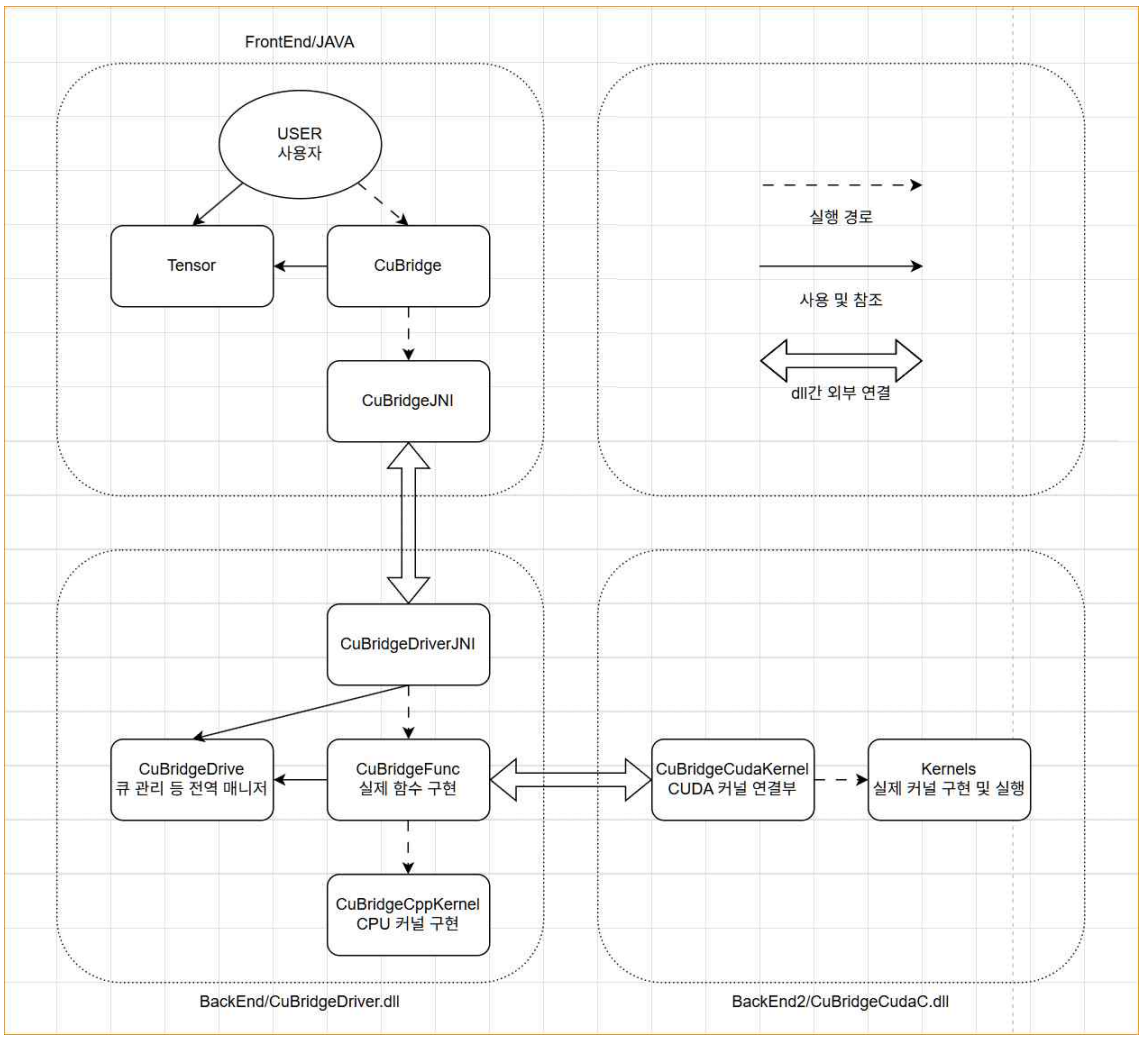
CuBridge는 Java 언어의 안정성과 CUDA의 병렬 연산 성능을 결합하여, 수치 계산의 새로운 가능성을 실험한 결과물이다. 본 연구는 거대한 프레임워크를 지향하지 않으면서도 Java 환경 내에서 독립적인 수치 연산 기반을 구축할 수 있다는 점을 입증하고자 하였다. 특히, 연산 흐름의 선언적 구성, GPU/CPU 환경에 따른 자동 전환, 내부 메모리 구조의 안정성 등은 이후 다양한 프로젝트에서 재사용 가능한 핵심 설계 자산이 될 수 있다.

향후 CuBridge는 단순한 라이브러리를 넘어, 병렬 연산의 확장, 서버 기반 연산 처리, 분산 시스템 연동 등 보다 넓은 적용 범위로 확장될 계획이다. 비록 지금은 초기 단계의 경량 프레임워크이지만, Java 기반 과학 연산 생태계에서 실질적 기여를 할 수 있는 실행 기반으로 발전해 나가고자 한다. 본 연구가 Java 환경에서 수치 연산과 GPU 활용의 새로운 방향을 모색하는 출발점이 되기를 기대한다.

부록(Appendix)

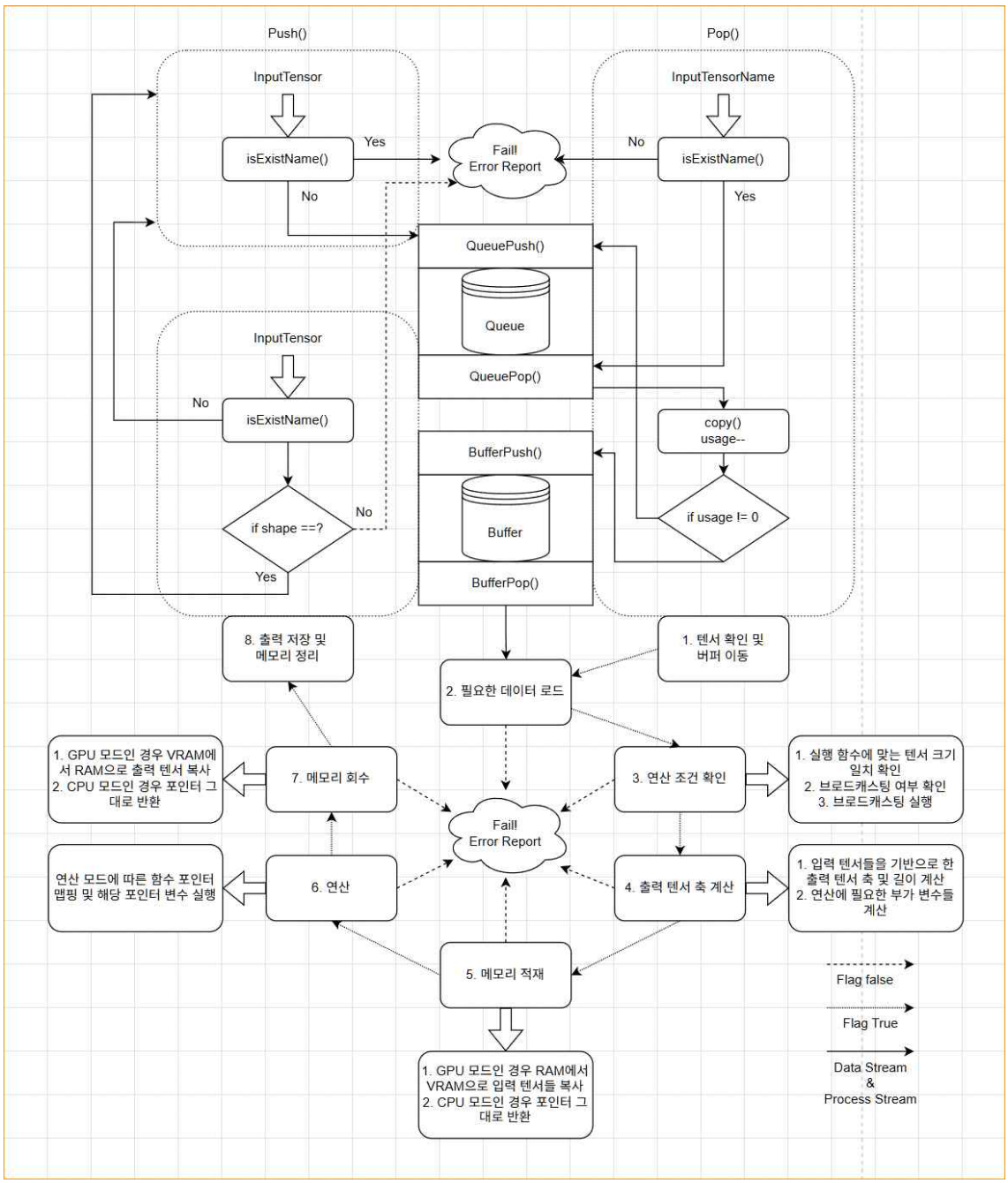
A. 클래스 간 구조도

해당 부록에서는 각 단계에 내장된 클래스 간 관계를 도식한다.



B. DLL 내부 시스템 흐름

해당 부록에서는 함수 내부 8단계와 데이터 push, pop의 알고리즘을 도식한다.



C. 전체 함수 정리

해당 단계에서는 사용자가 직접 다루는 프론트엔드의 API 함수들을 다룬다.

모든 함수들은 각 매개변수가 생략된 버전의 함수들이 오버로딩 되어 있어, 실제 사용 시에는 의도에 따라 매개변수 생략이 가능하다. 입력 매개변수가 생략된 경우에는 내부 큐의 최선단을 가져와 연산하고, 출력 매개변수가 생략된 경우에는 내부적으로 랜덤한 임시 이름을 지정해 큐에 저장한다.

해당 부록에서는 모두 원본으로 표기한다.

CuBridge		
순번	분류	함수
1	쿠브릿지 사용 시 필수 호출	getInstance()
2	환경 설정	selectCPU()
3		selectGPU()
4		envReset()
5	환경 질의	getEnviromentsStatus()
6	초기화	clear()
7	내부 큐 및 버퍼 내용 확인	visualQueue(String name)
8		visualBuffer(String name)
9	정수 입력	put(int data, String name, int usage)
10	실수 입력	put(double data, String name, int usage)
11	텐서 입력	put(Tensor data, String name, int usage)
12	텐서 출력	get(String name)
13	내부 텐서 사용 횟수 조작	duple(String name, int usage)
14	내부 텐서 브로드캐스팅 가능 여부 조작	broad(String name, boolean broad)
15	내부 텐서 축 정보 조작	reshape(String name, int[] shape)
16	단항 연산자 (Unary)	abs(String a, String out)
17		neg(String a, String out)
18		square(String a, String out)
19		sqrt(String a, String out)
20		log(String a, String out)
21		log2(String a, String out)
22		ln(String a, String out)
23		reciprocal(String a, String out)
24		sin(String a, String out)
25		cos(String a, String out)
26		tan(String a, String out)
27		step(String a, String out)
28		sigmoid(String a, String out)
29		tanh(String a, String out)
30		relu(String a, String out)
31		leakrelu(String a, String out)
32		softplus(String a, String out)
33		exp(String a, String out)
34		deg2rad(String a, String out)
35		rad2deg(String a, String out)
36		round(String a, String out)
37		ceil(String a, String out)
38		floor(String a, String out)
39		not(String a, String out)

순번	분류	함수
40	이항 연산자 (Binary)	add(String a, String b, String out)
41		sub(String a, String b, String out)
42		mul(String a, String b, String out)
43		div(String a, String b, String out)
44		pow(String a, String b, String out)
45		mod(String a, String b, String out)
46		gt(String a, String b, String out)
47		lt(String a, String b, String out)
48		ge(String a, String b, String out)
49		le(String a, String b, String out)
50		eq(String a, String b, String out)
51		ne(String a, String b, String out)
52		and(String a, String b, String out)
53		or(String a, String b, String out)
54	축 종속 연산자 (Axis-Based) (하위 축 포함 연산)	sum(String a, String out, int axis)
55		mean(String a, String out, int axis)
56		var(String a, String out, int axis)
57		std(String a, String out, int axis)
58		max(String a, String out, int axis)
59		min(String a, String out, int axis)
60	축 독립 연산자 (Axis-Free) (지정 축만 연산)	accumulate(String a, String out, int axis)
61		compress(String a, String out, int axis)
62		expand(String a, String out, int axis)
63		axisMax(String a, String out, int axis)
64		axisMin(String a, String out, int axis)
65		argMax(String a, String out, int axis)
66		argMin(String a, String out, int axis)
67	행렬 변환 연산자 (Matrix Operation)	transpose(String a, String out, int axis1, int axis2)
68		dot(String a, String b, String out)
69		matmul(String a, String b, String out)
70		im2col1D(String input, String kernel, String out, int pad, int stride)
71		col2im1D(String input, String kernel, String out, int oL, int pad, int stride)
72		im2col2D(String input, Strin kernel, String out, int padH, int padW, int strideH, int strideW)
73		col2im2D(String input, Strin kernel, String out, int oH, int oW, int padH, int padW, int strideH, int strideW)
74	신경망 연산자	mse(String yp, String y, String out)
75		cee(String yp, String y, String out)
76		affine(String x, String w, String b, String out)
77		softmax(String name, String out, int axis)

D. 이후 버전별 상세 계획 로드맵

해당 부록에서는 지금까지의 버전 특징 및 이후 상세 버전의 계획을 설명한다.

순번	버전	주요 목표	상세 설명
1	Beta	Java와 CUDA 간의 기본 연동 구조 검증	1. JNI 구조 설계 2. Tensor 및 CuBridge 내부 클래스 들의 기초 구현 3. DLL 분리 및 기본 연산 함수 구성 4. 환경적응 및 CuBridge 벡엔드 엔진 구동의 검증 5. 장치 성능에 따른 데이터의 RAM/VRAM 선택 저장 구현 6. 큐, 버퍼, 상수 전용 맵의 3중 저장소 구현
2	1.0	기본 연산 및 실행 흐름 안정화	1. put -> cal -> get 선언형 연산 인터페이스 확립 2. 77개의 원본 함수, 오버로딩 포함 약 300여개의 API 함수 정립 3. CuBridgeCudaC.dll 분리를 통한 환경적응형 구조 완성 4. 함수포인터 구현 및 함수 실행 8단계 정립 5. RAM/VRAM 선택 저장 및 상수 전용 맵 삭제를 통한 최적화
3	1.1	합성곱 신경망 연산 준비 및 전처리 기능 확장	1. im2col, col2im, reshape 등 CNN 연산 보조 함수 추가 2. rad2deg, deg2rad 등 수학 유틸리티 함수 추가 3. 문자열 기반 텐서 로딩 기능 및 정규화 기능 지원
4	1.2	상수 저장 기능 추가	1. 내부적으로 자주 사용되는 상수들 미리 저장 및 제공
5	1.3	비용함수, norm등 거리함수 추가	1. cee, mse 이외의 비용함수들 구현 및 CostFunc로 분리 2. norm, Euclidean distance등의 크기/거리함수 구현
6	1.4	복합함수 추가	1. 복합적인 고급 선형대수 연산들 추가
7	1.5	Autograd 추가	1. 연산 선언기록 저장 및 역전파 자동 구축 실행 구현
8	1.6	고급 대수연산 및 매크로 함수 추가	1. 고급 선형대수 연산들 추가 2. 기본 연산들의 활용 및 for 등 반복 연산이 포함 된 매크로 함수 구현
9	2.0	완전 병렬 연산 및 다중 GPU 확장	1. 거대 연산을 블록 및 스레드, vram 적재 수준에서 병렬 분할 2. 다중 GPU 환경에서의 연산 분산 및 결과 통합 처리 구현 3. 연산 스케줄러 도입: 다중 GPU와 다중작업 필요시 입력 크기 기반 연산 조각 자동 분배 최적화 4. 커널 호출 동기화 및 GPU-CPU 전송 최적화 5. 스레딩 도입을 통한 연산/커널의 완전한 병렬화 구현
10	3.0	연산 처리를 서버에 위임할 수 있는 구조 구축	1. 서버-클라이언트 구조 구축 및 연결 과정 정립 2. 연산 요청 클라이언트에서 서버로 전송 및 결과 수신 3. JSON 기반 REST 인터페이스 도입 4. CubBridge의 서버/클라이언트/독립 모드에 따른 내부 API 바이패스 구조 구축 5. 서버 내 텐서 캐싱 및 요청 큐 최적화 6. 다중 클라이언트와 단일 서버 연결 지원 및 최적화
11	4.0	분산 노드 기반 클러스터 연산 구조 확장	1. 단일 클라이언트와 다중 서버 구조 구축 및 연결 과정 정립 2. 내부 바이패스 구조 구현 3. 모드에 따른 텐서 단위 분할 후 다중 노드에 작업 분배 최적화 4. 데이터 분배 및 연산과 결과 종합 알고리즘 구현 5. 서버/클라이언트의 이중구조 병행을 통한 트리형 구조 구축 및 연결 과정 정립 6. 연산 흐름의 DAG화 및 데이터 분배/결과 병합 알고리즘 구현 7. 노드 간 동기화, 연산 병합, 오류 복구 구조 구현 8. 트리형 -> 클러스터 -> 그리드 네트워킹 지원

E. 샘플 테스트 시나리오(선형회귀)

CuBridge

```
CuBridge cb = CuBridge.getInstance();

//1. 데이터 생성
int N = 1000;
int D = 3;

Tensor X = Tensor.rand(new int[]{N, D});

Tensor cost_before = null, cost_after, cost = null;

//2. 정답 가중치와 편향 생성 및 정답 계산
Tensor Wt = Tensor.rand(new int[]{D, 1});
Tensor Bt = Tensor.rand(new int[]{1, 1});

cb.put(X, "X").put(Wt, "Wt").put(Bt, "Bt");
Tensor Yt = cb.affine("X", "Wt", "Bt", "Yt").get("Yt");

//3. 랜덤 가중치와 편향 생성
Tensor W = Tensor.rand(new int[]{D, 1});
Tensor B = Tensor.rand(new int[]{1, 1});

System.out.println("Step 0 W, B");
W.printData();
B.printData();
System.out.println();

//7. 반복
for(int i = 0; i < 5000; i++){
        //4. 예측 계산
        cb.put(X, "X").put(W, "W").put(B, "B");
        Tensor Yp = cb.affine("X", "W", "B", "Yp").get("Yp");

        //5. MSE 비용 연산
        cb.put(Yp, "Yp").put(Yt, "Yt");
        cost = cb.mse("Yp", "Yt", "cost").get("cost");

        if(i == 0)
                cost_before = new Tensor(cost.toArray());

        //6. 역전파
        cb.put(Yp, "Yp").put(Yt, "Yt")
          .put(X, "X").put(W, "W").put(B, "B")
          .put(0.01, "RATE", 2).put(N, "N");

        //5-1. 델타 계산
        cb.sub("Yp", "Yt", "delta").div("delta", "N", "delta").duple("delta", 2);

        //5-2. dw db 계산
        cb.transpose("X", "tX").dot("tX", "delta", "dw")
          .mul("dw", "RATE", "dw");

        cb.accumulate("delta", "db", 0)
          .mul("db", "RATE", "db");

        //5-3. 값 갱신
        cb.sub("W", "dw", "W").sub("B", "db", "B");

        W = cb.get("W");
        B = cb.get("B");
}

cost_after = new Tensor(cost.toArray());

System.out.println("Step 5000 W, B");
W.printData();
B.printData();
```

```
System.out.println();

System.out.println("Correct W, B");
Wt.printData();
Bt.printData();
System.out.println();

System.out.println("Cost Before/after");
cost_before.printData();
cost_after.printData();
System.out.println();
```

F. 출처

1, T. E. Oliphant, A Guide to NumPy, Trelgol Publishing, 2006.
Python 기반 수치 연산 라이브러리 NumPy의 구조 설명서

2. NumPy documentation
Numpy 공식 문서 페이지
https://numpy.org/doc/stable/index.html

3. NVIDIA, CUDA Toolkit Documentation, 2024.
CUDA 커널, 메모리, 연산 흐름에 대한 공식 문서

4. JCuda.org, JCuda: Java bindings for the CUDA runtime and driver API, 2023.
Java용 CUDA 바인딩 라이브러리

5. Oracle, Java Native Interface (JNI) Specification, 2024.
Java와 native 코드 연동 방식에 대한 공식 명세

6. Skymind Inc., ND4J: N-Dimensional Arrays for Java, 2022.
Java 기반 N차원 배열 연산 라이브러리
https://deeplearning4j.org/docs/latest/nd4j-overview

7. Peter Abeles, Efficient Java Matrix Library (EJML), 2023.
Java용 경량 행렬 연산 라이브러리
https://ejml.org/

8. 배준호, CuBridge : A GPU-Accelerated Tensor Library for Java, 2025
CuBridge 공식 구현 저장소
https://github.com/jun-Bridge/CuBridge