

2.4 텐서의 GPU 활용: Docker 를 이용한 완벽한 개발 환경 구축

딥러닝 모델의 학습은 수백만, 수십억 개의 **파라미터**에 대한 수많은 **행렬 연산**을 포함합니다. 이러한 연산을 CPU 만으로 처리하는 것은 엄청난 시간이 소요됩니다. **GPU**(Graphics Processing Unit)는 수천 개의 작은 코어를 가지고 있어, 단순하고 반복적인 계산을 대규모로 병렬 처리하는 데 특화되어 있습니다. 이 때문에 **딥러닝**의 행렬 곱셈과 같은 연산을 CPU 대비 수십 배에서 수백 배까지 가속할 수 있습니다.

파이토치는 이러한 GPU 의 능력을 최대한 활용할 수 있도록 설계되었습니다. 개발자는 간단한 코드 몇 줄만으로 **데이터와 모델을 GPU 로 옮겨** 학습 속도를 극적으로 향상시킬 수 있습니다. 하지만 GPU 를 사용하기 위해서는 복잡한 드라이버와 라이브러리(CUDA 등) 설치가 필요하며, 이는 종종 개발 환경 설정의 가장 큰 걸림돌이 됩니다. 이러한 문제를 해결하고, 재현 가능하며 이식성 높은 개발 환경을 구축하기 위해 도커(Docker)를 활용하는 것이 현대적인 개발 방식의 표준으로 자리 잡고 있습니다.

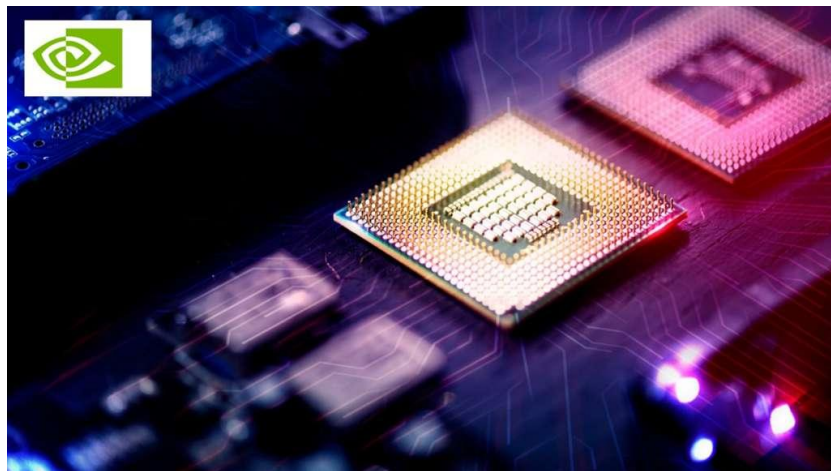


그림 2-1: 딥러닝 연산을 가속하는 핵심 하드웨어인 NVIDIA GPU. 파이토치는 CUDA 를 통해 GPU 의 병렬 처리 능력을 활용합니다.

2.4.1 왜 도커(Docker)인가? "제 컴퓨터에서는 되는데..." 문제의 완벽한 해결책

도커는 애플리케이션을 **컨테이너(Container)**라는 격리된 환경에 모든 실행 환경과 함께 패키징하여 실행하는 기술입니다. 컨테이너는 운영체제, 파이썬 버전, 파이토치 버전, CUDA 버전, 그리고 필요한 모든 라이브러리와 종속성을 포함하므로, "제 컴퓨터에서는 되는데, 다른

팀원 컴퓨터나 서버에서는 안 돼요"와 같은 고질적인 문제를 원천적으로 방지합니다. 딥러닝 GPU 개발 환경에서 도커를 사용하면 다음과 같은 강력한 이점을 얻을 수 있습니다.

- 완벽한 환경 격리 및 **재현성**: 여러분의 **컴퓨터(호스트 머신)**에는 NVIDIA **드라이버**만 깔끔하게 설치하고, 복잡한 CUDA Toolkit, **cuDNN**, 파이토치 등은 모두 도커 컨테이너 안에 설치합니다. 이를 통해 호스트 시스템을 깨끗하게 유지하고, 프로젝트마다 정확히 동일한 버전의 환경을 코드로 관리하며 구성할 수 있습니다.
- 압도적으로 간편한 설정: NVIDIA 에서 공식적으로 제공하는, CUDA 및 파이토치가 사전 설치된 도커 이미지를 사용하면, 복잡한 설치 과정 없이 명령어 한 줄로 검증된 개발 환경을 즉시 실행할 수 있습니다.
- 뛰어난 이식성: 도커 이미지는 어디서든 동일하게 동작합니다. 여러분의 로컬 PC 에서 개발한 환경을 그대로 클라우드 서버(AWS, GCP 등)나 **다른 동료의 컴퓨터로 옮겨 실행할 수 있어** 협업과 배포가 매우 용이해집니다.



그림 2-2: 도커를 사용하면 복잡한 라이브러리들을 컨테이너 안에 격리하여, 깨끗하고 재현 가능한 개발 환경을 구축할 수 있습니다.

2.4.2 도커를 이용한 GPU 개발 환경 구축 (실전편)

이제부터 실제 딥러닝 프로젝트에서 사용하는 방식 그대로, Docker Compose 를 활용하여 GPU 개발 환경을 구축하는 방법을 단계별로 상세하게 안내하겠습니다. 이 과정을 따라오면 여러분은 현업 수준의 개발 환경을 갖추게 될 것입니다.

1 단계: 사전 준비 사항 확인 및 설치

컨테이너가 여러분의 GPU 를 사용하기 위해서는 몇 가지 필수 소프트웨어를 **호스트 머신**에 설치해야 합니다.

1. NVIDIA 드라이버 설치: 가장 먼저, 여러분의 컴퓨터에 장착된 NVIDIA GPU 에 맞는 최신 드라이버를 설치해야 합니다. **터미널(Windows에서는 PowerShell 또는 CMD)**을 열고 **nvidia-smi** 명령어를 실행했을 때, 아래와 같이 GPU 정보가 정상적으로 출력되어야 합니다. 만약 명령어가 실행되지 않는다면 NVIDIA 드라이버 다운로드 페이지에서 드라이버를 먼저 설치하세요.

2. **도커 데스크톱(Docker Desktop)** 설치: Docker 공식 웹사이트에서 여러분의 운영체제(Windows, macOS, Linux)에 맞는 Docker Desktop 을 다운로드하여 설치합니다. 설치 과정은 매우 간단하며, 몇 번의 클릭만으로 완료됩니다.

3. NVIDIA **Container** Toolkit 설치: 이것이 바로 도커 컨테이너가 호스트 머신의 GPU 에 접근할 수 있도록 해주는 '다리' 역할을 하는 핵심 도구입니다. 공식 설치 가이드를 따라 설치합니다. 이 툴킷이 설치되어 있어야 **docker run** 또는 **docker-compose** 명령어에서 GPU 옵션을 사용할 수 있습니다.

2 단계: 프로젝트 폴더 및 설정 파일 생성

이제 프로젝트를 위한 폴더를 만들고, 그 안에 도커 환경을 정의하는 세 개의 핵심 파일을 생성하겠습니다. VS Code 를 사용하여 이 과정을 진행하는 것을 추천합니다.

1. 프로젝트 폴더(예: **my-pytorch-project**)를 생성하고 VS Code 로 해당 폴더를 엽니다.
2. 폴더 내에 다음 세 개의 파일을 생성합니다.
 - **Dockerfile**: 우리만의 커스텀 도커 이미지를 만드는 설계도
 - **requirements.txt**: 설치할 파이썬 라이브러리 목록
 - **docker-compose.yml**: 컨테이너의 실행 옵션(포트, 볼륨, GPU 등)을 정의하는 파일

3 단계: Dockerfile 작성 (이미지 설계도)

Dockerfile 은 우리의 딥러닝 환경의 기반이 될 도커 이미지를 어떻게 만들지 단계별로 지시하는 파일입니다. 아래 내용을 **Dockerfile** 에 그대로 복사하여 붙여넣으세요. 코드 한 줄 한 줄의 의미를 자세히 설명하겠습니다.

```
# 1. 베이스 이미지 선택: NVIDIA 의 공식 CUDA 이미지를 사용합니다.
# CUDA 12.1, cuDNN 8 이 포함된 Ubuntu 22.04 개발자용 이미지입니다.
# 이 이미지를 기반으로 우리만의 환경을 구축합니다.
FROM nvidia/cuda:12.1.1-cudnn8-devel-ubuntu22.04

# 2. 불필요한 상호작용 방지 및 시간대 설정 (환경 변수)
# apt-get 설치 시 질문을 묻지 않도록 설정하고, 시간대를 서울로 지정합니다.
ENV DEBIAN_FRONTEND=noninteractive
ENV TZ=Asia/Seoul
```

3. Python 및 필수 시스템 패키지 설치

RUN 명령어는 컨테이너 내부에서 셸 명령을 실행합니다.

RUN apt-get update && apt-get install -y ₩

python3.10 ₩

python3-pip ₩

python3.10-venv ₩

matplotlib 등 GUI 관련 라이브러리에 필요한 패키지

python3-tk ₩

matplotlib 에서 한글을 사용하기 위한 폰트

fonts-nanum ₩

이미지 크기 최적화를 위해 설치 후 apt 캐시 삭제

&& rm -rf /var/lib/apt/lists/*

4. 작업 디렉토리 설정

컨테이너 내부에서 명령이 실행될 기본 경로를 /workspace 로 지정합니다.

WORKDIR /workspace

5. 파이썬 가상 환경 생성

시스템 파이썬과 격리된 우리만의 파이썬 환경을 만듭니다.

ENV VIRTUAL_ENV=/opt/venv/py310

RUN python3.10 -m venv \$VIRTUAL_ENV

생성된 가상 환경을 기본 파이썬으로 사용하도록 경로를 설정합니다.

ENV PATH="\$VIRTUAL_ENV/bin:\$PATH"

6. pip 업그레이드 및 requirements.txt 복사/설치

pip 자체를 최신 버전으로 업그레이드합니다.

RUN pip install --upgrade pip

호스트 컴퓨터의 requirements.txt 파일을 컨테이너의 /workspace 로 복사합니다.

COPY requirements.txt .

복사된 requirements.txt 에 명시된 모든 파이썬 라이브러리를 설치합니다.

--no-cache-dir 옵션은 불필요한 캐시를 남기지 않아 이미지 크기를 줄여줍니다.

RUN pip install --no-cache-dir -r requirements.txt

7. JupyterLab 실행 명령

```
# 컨테이너가 시작될 때 자동으로 실행될 기본 명령입니다.  
# 외부 접속 허용, 8888 포트 사용, 비밀번호 없이 접속하도록 설정합니다.  
CMD ["jupyter", "lab", "--ip=0.0.0.0", "--port=8888", "--allow-root", "--no-browser", "--  
ServerApp.token="", "--ServerApp.password=""]
```

Dockerfile 상세 해설:

- FROM: 모든 Dockerfile 의 시작점입니다.`nvidia/cuda:12.1.1-cudnn8-devel-ubuntu22.04` 는 NVIDIA 가 제공하는 공식 이미지로, GPU 가속에 필요한 CUDA Toolkit 과 cuDNN 라이브러리가 이미 설치된 Ubuntu 22.04 환경입니다. 'devel' 버전은 컴파일러 등 개발 도구가 포함되어 있어 라이브러리 설치 시 발생할 수 있는 문제를 최소화합니다.
- ENV: 환경 변수를 설정합니다. 컨테이너의 시간대를 설정하거나, 설치 과정에서 불필요한 확인 창이 뜨는 것을 방지합니다.
- RUN: 셸 명령어를 실행하여 패키지를 설치합니다.`apt-get update` 로 패키지 목록을 갱신하고,`apt-get install -y` 로 필요한 시스템 도구들(파이썬, pip 등)을 설치합니다.&&를 사용하여 여러 명령을 한 줄에 연결하면 도커 이미지의 레이어 수를 줄여 효율성을 높일 수 있습니다.
- WORKDIR: 컨테이너 내의 '현재 폴더'를 지정합니다. 이후의 `COPY`,`RUN` 명령어들은 이 디렉토리를 기준으로 실행됩니다.
- 가상 환경:`python -m venv` 를 사용하여 독립된 파이썬 환경을 만드는 것은 좋은 습관입니다. 이를 통해 시스템 전체에 영향을 주지 않고 프로젝트별 의존성을 관리할 수 있습니다.
- COPY: 호스트 컴퓨터의 파일을 컨테이너 내부로 복사합니다.
여기서는 `requirements.txt` 파일을 복사하여 파이썬 라이브러리를 설치할 준비를 합니다.
- CMD: 컨테이너가 실행될 때 기본적으로 수행할 명령을 지정합니다. 여기서는 JupyterLab 을 실행하여 웹 브라우저를 통해 코드를 작성하고 실행할 수 있는 환경을 제공합니다.

4 단계: `requirements.txt` 작성 (파이썬 라이브러리 목록)

`requirements.txt` 파일에는 이 프로젝트에서 사용할 파이썬 라이브러리들을 명시합니다. `Dockerfile` 은 이 파일을 참조하여 `pip install` 을 실행합니다.

```
# PyTorch Core Libraries (CUDA 12.1)  
# 버전을 명시하여 재현성을 보장하는 것이 매우 중요합니다.  
torch==2.3.1
```

```
torchvision==0.18.1
torchaudio==2.3.1
--extra-index-url https://download.pytorch.org/whl/cu121
```

```
# Jupyter Environment
jupyter
jupyterlab
ipywidgets
```

```
# Data Science & ML Libraries
numpy
pandas
matplotlib
scikit-learn
```

중요: 파이토치 설치 명령어의 비밀

--extra-index-url <https://download.pytorch.org/whl/cu121> 부분은 매우 중요합니다. 이는 pip 이 기본 패키지 저장소(PyPI) 외에, 파이토치가 제공하는 CUDA 12.1 용 특별 빌드 저장소에서도 패키지를 찾도록 지시합니다. 이 부분이 없으면 GPU 를 지원하지 않는 일반 버전의 파이토치가 설치될 수 있습니다.

5 단계: *docker-compose.yml* 작성 (컨테이너 오케스트레이션)

docker-compose.yml 파일은 *Dockerfile* 로 빌드된 이미지를 사용하여 컨테이너를 어떻게 실행할지 상세하게 정의합니다. 포트 연결, 폴더 공유, GPU 할당 등 실질적인 실행 환경을 설정하는 역할을 합니다.

```
# Docker Compose 파일 형식 버전을 지정합니다. '3.8' 이상을 권장합니다.
version: '3.8'
```

```
# services 는 실행할 컨테이너들의 묶음입니다.
```

```
services:
```

```
  # 서비스의 이름을 'pytorch-dev'로 지정합니다. 이 이름이 컨테이너 이름의 일부가 됩니다.
```

```
    pytorch-dev:
```

```
# build: 현재 디렉토리('.')의 Dockerfile 을 사용하여 이미지를 빌드하라는 의미입니다.
build: .
# ports: 호스트와 컨테이너의 포트를 연결합니다.
# "8888:8888" -> 호스트의 8888 번 포트에 들어오는 요청을 컨테이너의 8888 번 포트에
전달합니다.
# 이를 통해 웹 브라우저에서 localhost:8888 로 JupyterLab 에 접속할 수 있습니다.
ports:
  - "8888:8888"
# volumes: 호스트의 폴더를 컨테이너의 폴더에 연결(마운트)합니다.
# 이를 통해 컨테이너가 종료되어도 작업 내용이 보존되고, 호스트에서 코드를 수정하면
컨테이너에 즉시 반영됩니다.
volumes:
  # 호스트의 현재 폴더(/src)를 컨테이너의 /workspace/src 폴더에 연결합니다.
  # 소스 코드를 저장할 공간입니다.
  - ./src:/workspace/src
  # Hugging Face 모델 등 대용량 파일을 저장할 캐시 폴더를 연결합니다.
  # 모델을 다시 다운로드하는 것을 방지하여 시간을 절약합니다.
  - ./cache/huggingface:/root/.cache/huggingface
# environment: 컨테이너 내부에서 사용할 환경 변수를 설정합니다.
environment:
  - JUPYTER_ENABLE_LAB=yes
  # Hugging Face 라이브러리가 캐시를 저장할 경로를 위 볼륨 경로와 일치시킵니다.
  - TRANSFORMERS_CACHE=/root/.cache/huggingface
# deploy: 서비스 배포와 관련된 설정을 정의합니다. GPU 사용을 위해 필수적입니다.
deploy:
  resources:
    reservations:
      devices:
        # driver: nvidia -> NVIDIA GPU 드라이버를 사용하도록 지정합니다.
        - driver: nvidia
        # count: all -> 시스템에 있는 모든 GPU 를 컨테이너에 할당합니다.
        count: all
        # capabilities: [gpu] -> 컨테이너가 GPU 기능을 사용할 수 있도록 허용합니다.
        capabilities: [gpu]
```

이제 프로젝트 폴더 구조는 다음과 같을 것입니다.

```
my-pytorch-project/
├── Dockerfile
├── docker-compose.yml
├── requirements.txt
├── src/          # 소스 코드를 저장할 폴더 (미리 만들어주세요)
└── cache/        # 캐시 파일을 저장할 폴더 (미리 만들어주세요)
```

6 단계: 환경 실행 및 VS Code 연동

모든 준비가 끝났습니다. 이제 터미널에서 다음 명령어를 실행하여 우리의 딥러닝 환경을 빌드하고 실행해 봅시다.

```
# my-pytorch-project 폴더에서 아래 명령어를 실행합니다.
docker-compose up --build
```

- `docker-compose up:docker-compose.yml` 파일에 정의된 서비스를 시작합니다.
- `--build`: 이미지를 빌드하거나, `Dockerfile` 이 변경된 경우 이미지를 다시 빌드합니다. 처음 실행할 때는 반드시 이 옵션을 사용해야 합니다.

이 명령을 실행하면 터미널에 수많은 로그가 출력되면서 이미지 빌드 및 라이브러리 설치가 진행됩니다. 이 과정은 처음 실행 시 몇 분 정도 소요될 수 있습니다. 모든 과정이 성공적으로 끝나면, 터미널에 JupyterLab 접속 URL 과 함께 서버가 실행 중이라는 메시지가 나타납니다.

이제 가장 중요한 VS Code 연동 단계입니다.

1. VS Code Docker 확장 프로그램 설치: VS Code 좌측의 확장 프로그램 탭에서 'Docker'를 검색하여 Microsoft 에서 제공하는 공식 확장 프로그램을 설치합니다.
2. 컨테이너에 연결: Docker 확장 프로그램을 설치하면 사이드바에 고래 모양 아이콘이 생깁니다. 이 아이콘을 클릭하면 현재 실행 중인 컨테이너 목록이 나타납니다. 방금 실행한 컨테이너(이름에 'pytorch-dev'가 포함됨)를 찾아 마우스 오른쪽 버튼을 클릭하고 'Attach Visual Studio Code'를 선택합니다.
3. 새로운 VS Code 창 열림: 잠시 후, 새로운 VS Code 창이 열립니다. 이 창은 이제 여러분의 로컬 컴퓨터가 아닌, 도커 컨테이너 내부에 직접 연결된 상태입니다. 창의 왼쪽 하단을 보면 "Container pytorch-dev..." 와 같이 현재 컨테이너에 연결되어 있음을 확인할 수 있습니다.

4. 작업 시작:

- 폴더 열기: VS Code 상단 메뉴에서 'File' > 'Open Folder...'를 선택하고, 경로에 `/workspace` 를 입력합니다. 그러면 `docker-compose.yml` 에서 마운트한 `src` 폴더와 `cache` 폴더가 보일 것입니다.
- 터미널 사용: VS Code 에서 터미널을 열면(`Ctrl+``), 이 터미널은 컨테이너 내부의 셸입니다. 여기서 `python`, `pip list`, `nvidia-smi` 등의 명령어를 실행하면 `Dockerfile` 에서 설정한 환경이 그대로 적용되는 것을 확인할 수 있습니다.
- 코드 작성 및 실행: `src` 폴더 안에 파이썬 파일(예: `gpu_test.py`)을 만들고 코드를 작성하세요. VS Code 의 파이썬 확장 프로그램이 컨테이너의 가상 환경(`/opt/venv/py310/bin/python`)을 자동으로 인식하여 코드 자동 완성, 디버깅 등을 완벽하게 지원합니다. 코드를 실행하면 컨테이너의 GPU 를 사용하여 연산이 수행됩니다.

이로써 여러분은 호스트 컴퓨터는 깨끗하게 유지하면서, 프로젝트에 필요한 모든 것이 완벽하게 격리되고 재현 가능한 전문가 수준의 딥러닝 개발 환경을 갖추게 되었습니다. 코드를 수정하고 싶으면 호스트의 `src` 폴더에서 작업하면 되고, 실행은 컨테이너에 연결된 VS Code 에서 하면 됩니다.

2.4.3 파이토치에서 GPU 사용하기

GPU 환경이 준비되었다면, 파이토치 코드에서 GPU 를 사용하는 것은 매우 간단합니다. 핵심은 device 객체를 만들고, 텐서와 모델을 해당 device 로 보내는 것입니다.

컨테이너에 연결된 VS Code 에서 `src/gpu_test.py` 파일을 만들고 다음 코드를 작성해 보세요.

```
import torch
import time
```

```
# 1. 장치 설정: 코드를 장치 독립적으로 만드는 핵심 패턴
# torch.cuda.is_available()을 확인하여 GPU 사용 가능 여부를 판단합니다.
# 가능하면 'cuda' 장치를, 불가능하면 'cpu' 장치를 사용하도록 설정합니다.
# 이렇게 하면 코드가 어떤 환경에서도 유연하게 동작할 수 있습니다.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
if torch.cuda.is_available():
    print(f"Device name: {torch.cuda.get_device_name(0)}")
```

2. 텐서를 특정 장치로 생성하거나 이동

처음부터 GPU 에 텐서 생성

```
tensor_on_gpu = torch.randn(1000, 1000, device=device)
print(f"\n 텐서가 생성된 장치: {tensor_on_gpu.device}")
```

CPU 에서 생성 후 GPU 로 이동

```
tensor_on_cpu = torch.randn(1000, 1000)
print(f"이동 전 텐서 장치: {tensor_on_cpu.device}")
tensor_on_gpu_moved = tensor_on_cpu.to(device)
print(f"이동 후 텐서 장치: {tensor_on_gpu_moved.device}")
```

3. GPU 연산 속도 비교

큰 행렬 곱셈 연산을 CPU 와 GPU 에서 각각 수행하고 시간을 측정합니다.

size = 4096

```
cpu_a = torch.randn(size, size, device='cpu')
cpu_b = torch.randn(size, size, device='cpu')
```

CPU 연산 시간 측정

```
start_time_cpu = time.time()
result_cpu = torch.matmul(cpu_a, cpu_b)
end_time_cpu = time.time()
cpu_time = end_time_cpu - start_time_cpu
print(f"\nCPU 연산 시간: {cpu_time:.5f} 초")
```

GPU 연산 시간 측정

```
if device.type == 'cuda':
    gpu_a = cpu_a.to(device)
    gpu_b = cpu_b.to(device)
```

GPU 워밍업 (초기 커널 로딩 시간 제외)

```
_ = torch.matmul(gpu_a, gpu_b)
```

GPU 연산은 비동기적으로 처리될 수 있으므로, 정확한 시간 측정을 위해

torch.cuda.synchronize()를 호출하여 GPU 의 모든 연산이 끝날 때까지 기다립니다.

```
torch.cuda.synchronize()
```

```
start_time_gpu = time.time()
result_gpu = torch.matmul(gpu_a, gpu_b)
torch.cuda.synchronize()
end_time_gpu = time.time()
gpu_time = end_time_gpu - start_time_gpu
print(f"GPU 연산 시간: {gpu_time:.5f} 초")

# 속도 향상률 계산
speedup = cpu_time / gpu_time
print(f"GPU 가 CPU 보다 약 {speedup:.2f}배 빠릅니다.")
```

VS Code 의 컨테이너 터미널에서 `python src/gpu_test.py` 를 실행하면, GPU 가 CPU 보다 행렬 곱셈에서 얼마나 빠른지 직접 확인할 수 있습니다. 이것이 바로 딥러닝에서 GPU 가 필수적인 이유입니다.

코드 핵심 요약:

- `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`: 이 한 줄의 코드는 장치 독립적인(device-agnostic) 코드를 작성하는 핵심 패턴입니다. 코드를 수정하지 않고도 GPU 가 있는 환경과 없는 환경 모두에서 실행할 수 있게 해줍니다.
- `torch.randn(..., device=device)`: 텐서를 생성할 때 `device` 인자를 주면 처음부터 해당 장치(GPU 또는 CPU)의 메모리에 텐서가 생성됩니다.
- `tensor.to(device)`: 이미 생성된 텐서를 다른 장치로 복사합니다. CPU ↔ GPU 이동 모두에 사용됩니다. 중요: 텐서 간의 연산은 모든 텐서가 동일한 장치에 있을 때만 가능합니다.
- `torch.cuda.synchronize()`: CPU 는 GPU 에 연산 명령을 내리고 바로 다음 코드를 실행하는 비동기(asynchronous) 방식으로 동작합니다. 따라서 정확한 GPU 연산 시간을 측정하려면, 이 함수를 호출하여 GPU 의 모든 작업이 완료될 때까지 CPU 가 기다리도록 해야 합니다.