https://doi.org/10.7840/kics.2025.50.4.683

자원 제약적 디바이스를 위한 도커 이미지 경량화에 대한 연구

임 수 연◆, 홍 용 근°

Research on Lightweighting Docker Images for Resource-Constrained Devices

Suyeon Lim◆, Yong-Geun Hong°

요 약

클라우드 컴퓨팅 분야에서 널리 활용되고 있는 도커 기술은 인공지능 기술을 다양한 도메인과 디바이스에 적용하면서부터 그 활용도는 더 커졌다. 특히, 인공지능 서비스의 애플리케이션 배포 뿐만 아니라 하드웨어와 소프트웨어의 이질성을 극복하고 운용과 설정을 쉽게 하기 위해서 도커 기술은 다양한 디바이스에서도 사용해야 하는 필요성이 생겼다. 지금까지는 크게 고려하지 않았던 자원 제약적 디바이스에서 도커 기술을 사용하기 위해, 도커 이미지 경량화 연구가 필요하다. 본 연구에서는 도커 이미지를 만들 때 사용되는 다양한 베이스 이미지를 활용한 빌드 시간, 이미지 크기, 성능 등을 비교하고 분 석하여 python 기반 애플리케이션을 개발하는 상황에서 가장 적합한 베이스 이미지를 선출하고 도커 이미지를 경량화하기 위한 도커파일의 구성 방안을 제시하고자 한다.

키워드 : 도커 이미지, 도커파일, 경량화, 컨테이너, 자원 제약적
Key Words : docker image, dockerfile, lightweight, container, resource-constrained

ABSTRACT

Docker technology, which is widely used in the field of cloud computing, is becoming increasingly useful as AI technology is applied to various domains and devices. In particular, it is necessary to use docker technology on various devices to overcome hardware and software heterogeneity, and to facilitate the operation and setup, as well as application deployment of AI services. In order to use docker technology on resource-constrained devices, which have been largely ignored, it is necessary to investigate docker image lightweighting. In this study, we compare and analyse the build time, image size and performance of different base images used to create docker images to select the most suitable base image in the context of developing python-based applications and propose a dockerfile configuration plan to lighten docker images.

※ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 정보통신방송표준개발지원사업(RS-2024-00397768)사업으로 지원받은 연구결과입니다.
◆ First Author : Daejeon University Department of AI Convergence, gkflqhflqkq@naver.com, 학생회원
° Corresponding Author : Daejeon University Department of AI Convergence, yghong@dju.kr, 종신회원
논문번호 : 202411-270-D-RN, Received November 6, 2024; Revised November 21, 2024; Accepted December 21, 2024

Ⅰ. 서 론

현대 소프트웨어 개발에서는 개발한 결과물을 빠르고 일관적인 배포가 중요하게 자리 잡고 있다. 이러한 기술들을 충족하기 위하여 가상화(virtualization) 기술 중의 하나인 컨테이너(container) 기술이 널리 사용되고 있다. 특히 2013년 Docker사에서 출시한 도커(docker)는 컨테이너 기반의 애플리케이션 배포를 자동화하고 일관성을 유지하는 도구로 자리 잡고 있다[1].

클라우드 컴퓨팅 분야에서 이미 많이 활용되고 있는 도커는 인공지능 기술을 활용하여 다양한 도메인에서 관련 서비스를 개발하면서부터 그 활용도는 더욱 커지기 시작하였다. 다양한 도메인에서 사용되는 하드웨어와 소프트웨어의 이질성을 극복하고, 개발한 인공지능 서비스를 쉽고 빠르게 도입하기 위해서 도커는 인공지능 서비스 운용에 필요한 파일과 설정을 한개의 도커 이미지(docker image)에 담아서 처리한다. 도커파일(dockerfile)은 도커 이미지를 정의하는 파일로 효율적인 도커 이미지의 생성과 배포의 시작이 된다. 하지만 도커파일을 잘못 구성하면 도커 이미지 크기가 불필요하게 커질 수 있고 그로 인한 빌드 시간과 배포 속도에 영향을 미칠 수 있다[4].

도커 이미지 크기와 빌드 시간이 커질수록 애플리케이션의 배포와 운영에 여러 문제가 발생하게 된다. 크기가 큰 도커 이미지는 전송할 때 시간이 더 소요되고 불필요한 리소스를 차지하게 되는 등 배포 주기를 지연시키게 된다. 또, 보안에서도 도커 이미지가 클수록 잠재적인 취약점이 증가하게 된다. 이런 문제는 클라우드 환경에서 여러 컨테이너를 동시에 운영할 때 취약해지고 전체적인 시스템의 효율성을 저하시킨다. 따라서 도커 이미지의 경량화는 소프트웨어 배포 효율성 뿐만 아니라 보안성, 운영비용 절감 측면에서도 중요한 과제가 될 수 있다[9].

또한, 인공지능 서비스가 자원 제약적 디바이스에서 운영되는 경우, 도커 이미지의 경량화는 더욱 중요한 문제가 된다. 예를 들어, GPU가 없는 저사양 디바이스에서는 처리 속도를 높이기 위해 도커 이미지의 크기를 최소화하는 것이 필수적이다. 경량화된 도커 이미지는 자원 제약적인 환경에서 더 빠르게 실행될 수 있으며, 제한된 자원에서 최적의 성능을 발휘하도록 도와준다.

본 연구에서는 도커 이미지를 만들 때 사용되는 다양한 베이스 이미지를 활용한 빌드 시간, 이미지 크기, 성능 등을 비교하고 분석한다. 이를 통해 각각의 베이스 이미지의 장단점을 도출하고 python 기반 애플리케이션을 개발하는 상황에서 가장 가볍고 사용하기 좋은 베이스 이미지를 선출하고 도커 이미지를 경량화하기 위한 도커파일의 구성 방안을 제시하고자 한다. 먼저 2장에서는 도커의 작동 방식과 구성, 도커 이미지에 관한 내용을 기술한다. 3장에서는 도커 이미지 경량화를 위한 실험을 수행하고, 다양한 방법을 적용하여 도커 이미지 크기를 줄이는 과정과 그 결과에 대해 서술한다. 4장에서는 실험 결과를 바탕으로 캐싱 전략, 베이스 이미지 선택, 레이어 최적화, 불필요한 캐시 제거, 다단계 빌드의 활용 등 도커 이미지 최적화의 중요성에 대해 고찰한다. 마지막 5장에서는 연구의 결론을 도출하고, 향후 연구에 관해 기술한다.

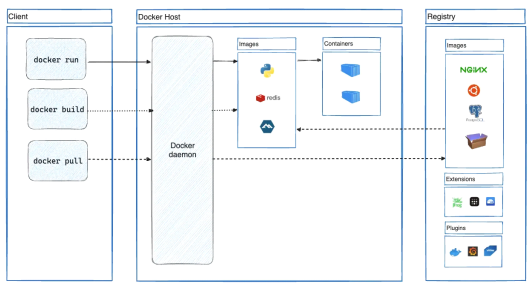
Ⅱ. 기술 배경

2.1 도커

도커는 컨테이너 기술을 기반으로 애플리케이션을 일관성 있게 해주는 오픈 소스 플랫폼이다[1, 9]. 도커를 사용하면 개발, 테스트 과정에서 발생하는 오류를 최소화 할 수 있다. 이런 도커의 특성 덕분에 다양한 오픈 소스 프로젝트들이 도커를 기반으로 운영되고 있다.

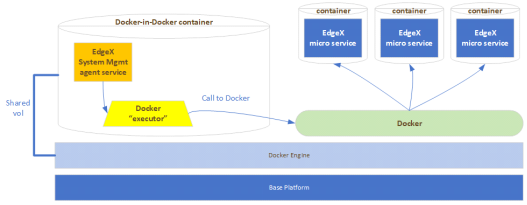
그림 1처럼 도커는 클라이언트-서버 구조로 나누어져 있고 클라이언트에서 명령어를 실행하면 도커 데몬(daemon)이 컨테이너, 이미지 등을 관리하며 돌아가게 된다[1].

도커 기술을 활용한 대표적인 예로 EdgeX Foundry와 ROS(Robot Operating System)를 들 수 있다. EdgeX Foundry에서 배포되는 소프트웨어는 에지 컴퓨팅 환경에서 소프트웨어를 쉽고 빠르게 배포하기 위하여 마이크로서비스 기반의 다수의 도커 이미지를 기반으로 구성되어 있다. ROS는 로봇 뿐만 아니라 자율주행 분야에서 소프트웨어를 효율적으로 배포하기 위하여 도커 이미지로 개발되고 있다. 그림 2와 그림 3에서 보는 것과 같이 여러 오픈소스와 플랫폼에서 도커를 사



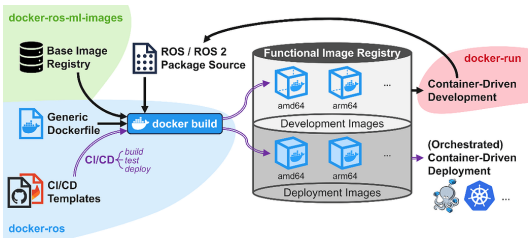
(출처 : 도커 웹사이트)

그림 1. 도커 구조
Fig. 1. Docker architecture



(출처 : EdgeX Foundry 웹사이트)

그림 2. EdgeX Foundry 구조
Fig. 2. EdgeX Foundry architecture



(출처 : ROS 웹사이트)

그림 3. ROS 구조
Fig. 3. ROS architecture

용하는 것을 확인할 수 있다.

2.2 도커 이미지

도커 이미지는 애플리케이션을 실행하기 위한 모든 파일과 설정을 포함하는 파일 시스템이다. 도커 이미지는 도커파일을 기반으로 빌드되고 여러 레이어(Layer)를 쌓는 형태로 구성되어 있다. 도커 이미지는 컨테이너를 실행하는 템플릿 역할을 하며 여러 컨테이너에서 동일한 이미지를 재사용할 수 있다.

2.3 도커 컨테이너

도커 컨테이너는 도커 이미지를 기반으로 실행되는 독립적인 환경이다. 도커 컨테이너는 환경에 필요한 모든 종속성을 포함하고 있고 필요한 리소스만 사용하여 가상환경보다 경량화 되어 있으며 효율을 극대화할 수 있다[2].

2.4 베이스 이미지

모든 도커 이미지는 베이스 이미지로 시작된다. 베이스 이미지는 OS(Operating System), 런타임, 라이브러리 등 기본 환경을 제공하고 거기에 더해 애플리케이션 코드와 추가적인 설정을 더해주면 최종 도커 이미지를 완성할 수 있게 된다. 상황에 맞는 적절한 베이스 이미지를 선택하는 것이 도커 이미지의 크기, 빌드 시간, 성능에 크게 영향을 줄 수 있다[3].

2.5 도커파일

도커파일에서 명령어를 읽어서 도커 이미지를 빌드한다. 도커파일은 소스코드를 빌드하기 위한 명령어가 들어있는 텍스트 파일이다[4].

Ⅲ. 도커 이미지 경량화를 위한 실험

3.1 도커파일에서 캐싱(caching)을 활용하여 재빌드 시 빌드 속도 최적화

도커파일 명령어가 한 줄 씩 실행될 때마다 새로운 레이어가 생성되는데, 도커 이미지를 빌드할 때 자주 변경되는 부분이 도커파일의 상단에 위치하면 레이어 캐싱을 제대로 활용하지 못하게 된다. 이를 방지하려면 자주 변경되는 부분을 도커파일의 하단에 배치해야 하며, 이를 통해 이전에 빌드된 레이어의 캐싱을 활용하여 재빌드 시간을 단축할 수 있다[5].

그림 4는 레이어 캐싱이 잘 적용된 도커파일을 보여준다. 여기서는 자주 변경되지 않는 파일을 상단에 배치하여, 재빌드 시 레이어 캐싱을 활용해 빌드 시간을 단축할 수 있다.

반면, 그림 5는 레이어 캐싱이 제대로 적용되지 않은 도커파일을 나타낸다. 예를 들어, 'requirements.txt'와 같은 의존성 파일은 자주 변경되지 않지만, 'test.py'와 같은 애플리케이션 코드 파일은 상대적으로 자주 변경된다. 만약 도커파일이 그림 5처럼 구성되어 있다면, 'test.py'와 같은 자주 변경되는 파일이 상단 부분에 위치하게 된다. 그러면 애플리케이션 파일이 수정될 때마다 도커는 그 아래의 모든 명령어(예: 'requirements.txt'를 추가한 후 'pip install'을 실행하는

```
1 FROM python:3.8
2
3 ADD requirements.txt .
4
5 RUN pip install --no-cache-dir -r requirements.txt
6
7 ADD test.py .
8
9 CMD ["python", "test.py"]
```

그림 4. 레이어 캐싱이 잘 지켜진 도커파일
Fig. 4. Dockerfile with well-preserved layer caching

```
1 FROM python:3.8
2
3 ADD test.py .
4
5 ADD requirements.txt .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 CMD ["python", "test.py"]
```

그림 5. 레이어 캐싱이 잘 지켜지지 않은 도커파일
Fig. 5. Dockerfile with poor layer caching

과정)를 반복하게 된다. 그 결과, 매번 'pip install -r requirements.txt'명령이 실행되어 의존성 설치 과정이 반복되고, 빌드 시간이 불필요하게 길어질 수 있다.

그림 6과 그림 7은 각각 그림4와 그림5의 도커파일을 재빌드한 결과를 보여준다. 자주 변경되지 않는 부분을 상단에 배치하고, 자주 변경되는 부분을 하단에 배치하면, 자주 변경되지 않은 레이어는 캐싱 되어 재사용되고, 변경된 부분만 새로 빌드할 수 있다. 이로 인해 빌드 시간이 크게 줄어든 것을 확인할 수 있었다.

첫 번째 빌드에서는 모든 레이어가 새로 생성되므로 빌드 시간이 길어지지만, 이후 빌드에서는 변경되지 않은 레이어가 캐시로 재사용되어 빌드 시간이 크게 단축되는 것을 확인할 수 있었다(그림 6, 그림 7 참고). 이를 통해 도커파일의 레이어 구조를 최적화하여, 반복 빌드에서 효율성을 극대화할 수 있음을 실험적으로 확인할 수 있다.

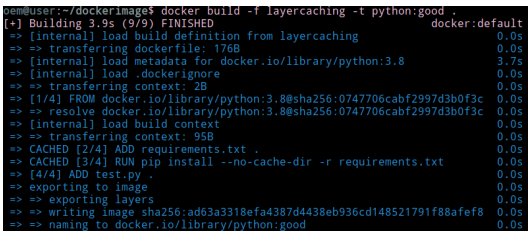


그림 6. 그림 4의 도커파일을 재빌드한 결과
Fig. 6. The result of rebuilding the docker file shown in Figure 4

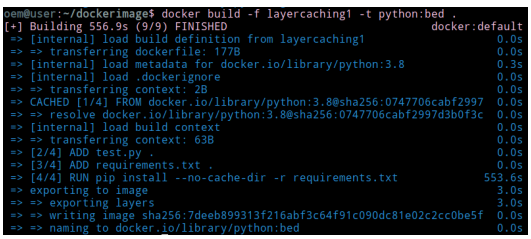


그림 7. 그림 5의 도커파일을 재빌드한 결과
Fig. 7. The result of rebuilding the docker file shown in Figure 5

3.2 최적의 경량화 베이스 이미지 선정 및 빌드 시간, 이미지 크기 비교

본 연구에서 사용하는 베이스 이미지는 다음과 같다

- python:3.8 : python 기반의 표준 이미지로 일반적인 개발 환경에 적합하다. 다양한 라이브러리와 도구가 포함 되어 있어 대부분의 python 애플리케이션 개발에 사용된다[6].
- python:3.8-slim : 경량화된 python 이미지로 최소한으로 필요한 라이브러리만 포함되어 있다. 이미지 크기를 줄이고 불필요한 패키지를 제거해 최적화된 개발 환경을 제공한다[6].
- python:3.8-alpine : Alpine Linux 기반의 경량화된 python 이미지로 이미지 크기가 매우 작아 배포와 네트워크 전송 시 효율적이다[6].

Python 애플리케이션에서 가장 경량화된 베이스 이미지를 선정하기 위해 간단한 코드인 (print('Hello World'))가 실행이 될 수 있게 작성하고('test.py'), 위 베이스 이미지를 사용하여 빌드한 후, 각 이미지의 빌드 시간과 최종 이미지 크기를 비교 분석한다.

실험의 통일성을 위해 그림 8처럼 'requirements.txt'를 동일하게 맞춰주었다.

그림 9는 python:3.8 베이스 이미지를 사용한 도커파일를 보여준다. 도커파일에서는 기본 python 베이스 이미지를 사용하여 간단한 print('Hello World') 코드를 실행할 수 있도록 설정되어 있다. 일반적인 python 환경을 제공한다[6].

그림 10은 python:3.8 베이스 이미지를 기반으로 한 도커파일(그림 9)을 빌드하는 과정에서의 출력 로그를 보여준다. 그림 10을 보면 빌드 시간이 147.1초 걸린

```
1 pandas
2 opencv-python
3 numpy
4 tensorflow
```

그림 8. 'requirements.txt' 코드
Fig. 8. 'requirements.txt' code

```
1 FROM python:3.8
2
3 COPY requirements.txt .
4
5 RUN pip install --no-cache-dir -r requirements.txt
6
7 COPY . .
8
9 CMD ["python", "test.py"]
```

그림 9. 베이스 이미지 python:3.8 도커파일
Fig. 9. Base image python:3.8 dockerfile

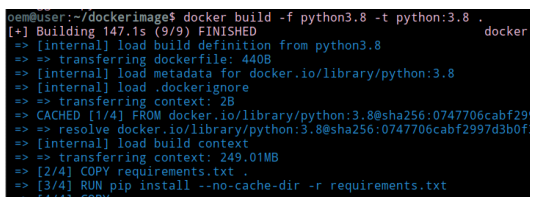


그림 10. 베이스 이미지 python:3.8 빌드
Fig. 10. Base image python 3.8 build

것을 확인할 수 있다.

그림 11을 보면 그림 10이 빌드된 후 최종적으로 생성된 도커 이미지의 크기를 보여준다. 이 도커 이미지의 크기는 3.05GB으로 나온 것을 확인할 수 있다.

그림 12는 python:3.8-slim 베이스 이미지를 사용한 도커파일을 보여준다. python:3.8 베이스 이미지보다 경량화된 python:3.8-slim 베이스 이미지는 불필요한 라이브러리와 패키지를 제거하여 필요한 최소한의 환경만을 포함하고 있다. 도커파일은 동일하게 print('Hello World') 코드를 실행할 수 있도록 설정되어 있다.

그림 13은 python:3.8-slim 베이스 이미지를 기반으로 한 도커파일(그림 12)을 빌드하는 과정에서의 출력 로그를 보여준다. 그림 13을 보면 빌드 시간이 164.9초 걸린 것을 확인할 수 있다.

그림 14를 보면 그림 13이 빌드된 후 최종 생성된 도커 이미지의 크기를 보여준다. 이 도커 이미지의 크기는 1.93GB로, 표준 python:3.8 베이스 이미지보다 상당히 작은 크기임을 확인할 수 있고 python:3.8 베이스 이미지보다 python:3.8-slim 베이스 이미지가 경량화에 얼마나 효과적인지를 나타낸다.

그림 15는 python:3.8-alpine 베이스 이미지를 사용한 도커파일을 보여준다. python:3.8-alpine 베이스 이미지는 Alpine Linux 기반으로 매우 작은 크기의 경량화된 python 베이스 이미지를 제공한다. 도커파일은 동일하게 print('Hello World') 코드를 실행할 수 있도록 설정되어 있다.

그림 16은 python:3.8-alpine 베이스 이미지를 사용하여 도커파일(그림 15)을 빌드하는 과정 에서의 출력 로그를 보여준다. 그림 16을 보면 빌드 시간이 2,423.1초 걸린 것을 확인할 수 있다.

그림 17을 보면 그림 16이 빌드된 후 최종 생성된 도커 이미지의 크기를 보여준다. 이 도커 이미지의 크기는 700MB로, 실험한 모든 베이스 이미지 중에서 가장 작은 크기를 가지고 있다.

많은 python 라이브러리는 내부적으로 C로 구현되어 있으며, 인터페이스만 python으로 제공되어 있는 경우가 많다. 대표적으로 NumPy, Keras, Pandas와 같은 라이브러리들이 이에 해당된다. python을 돌리기 위해서는 glibc 라는 C언어 컴파일을 위한 시스템 라이브러리를 가지고 있어야 하지만 python:3.8-alpine은 glibc를 사용하지 않고 musl를 사용하고 있다. python:3.8-alpine 버전을 사용하면 binary wheel을 사용하지 못하므로 모든 python 패키지의 C코드를 컴파일 해야 한다. 그래서 빌드 시간이 가장 많이 걸린다[7, 8].

```
oem@user:~/dockerimage$ docker images python:3.8
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
python       3.8       906dc7742d73   13 minutes ago   3.05GB
```

그림 11. 베이스 이미지 python:3.8 이미지 크기
Fig. 11. Base image python:3.8 image size

```
1 FROM python:3.8-slim
2
3 ADD requirements.txt .
4
5 RUN pip install --no-cache-dir -r requirements.txt
6
7 ADD test.py .
8
9 CMD ["python", "test.py"]
```

그림 12. 베이스 이미지 python:3.8-slim 도커파일
Fig. 12. Base image python:3.8-slim dockerfile

```
(tfserving) oem@user:~/dockerimage$ docker build -f pythonslim -t python:slim .
[+] Building 164.9s (9/9) FINISHED
 => [internal] load build definition from pythonslim
 => => transferring dockerfile: 477B
 => [internal] load metadata for docker.io/library/python:3.8-slim
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build context
 => => transferring context: 63B
```

그림 13. 베이스 이미지 python:3.8-slim 빌드
Fig. 13. Base image python 3.8-slim build

```
(tfserving) oem@user:~/dockerimage$ docker images python:slim
REPOSITORY   TAG       IMAGE ID       CREATED             SIZE
python       slim      c2e8b77923eb   About an hour ago   1.93G
```

그림 14. 베이스 이미지 python:3.8-slim 이미지 크기
Fig. 14. Base image python:3.8-slim image size

```
1 FROM python:3.8-alpine
2
3 RUN apk add --no-cache gcc musl-dev libffi-dev openssl-dev build-base
4
5 ADD requirements.txt .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 ADD test.py .
10
11 CMD ["python", "test.py"]
```

그림 15. 베이스 이미지 python:3.8-alpine 도커파일
Fig. 15. Base image python:3.8-alpine dockerfile

```
(tfserving) oem@user:~/dockerimage$ docker build -f pythonalpine -t python:alpine .
[+] Building 2423.1s (10/10) FINISHED
 => [internal] load build definition from pythonalpine
 => => transferring dockerfile: 618B
 => [internal] load metadata for docker.io/library/python:3.8-alpine
 => [internal] load .dockerignore
 => => transferring context: 2B
 => CACHED [1/5] FROM docker.io/library/python:3.8-alpine@sha256:3bd7ea88cb637e09d6c7
```

그림 16. 베이스 이미지 python:3.8-alpine 빌드
Fig. 16. Base image python 3.8-alpine build

```
(tfserving) oem@user:~/dockerimage$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED             SIZE
python       alpine    eb15614d6d2c   43 seconds ago      700MB
```

그림 17. 베이스 이미지 python:3.8-alpine 이미지 크기
Fig. 17. Base image python:3.8-alpine image size

3.3 도커파일 경량화 실험

3.3.1 && 연산자를 사용하여 레이어 수 줄이기

&& 연산자는 여러 명령어를 하나의 RUN 명령어로 결합하여 레이어 수를 줄이는 데 사용된다. 이는 도커 이미지의 크기를 줄이는 데에 큰 도움을 줄 수 있다.

그림 18은 RUN 명령어가 독립적으로 실행되며 각 명령어가 새로운 레이어를 생성한다.

그림 19는 RUN 명령어를 &&으로 결합하여 여러 줄의 명령어를 하나의 레이어로 묶어 실행되도록 한 것이다.

그림 20은 && 연산자를 사용하지 않는 도커파일을 빌드한 결과를 보여준다.

```
1 FROM python:3.8-slim
2
3 WORKDIR /app
4
5 COPY test.py .
6
7 RUN pip install requests
8 RUN pip install numpy
9 RUN pip install pandas
10 RUN pip install matplotlib
11 RUN pip install tensorflow
12
13 CMD ["python", "test.py"]
```

그림 18. && 연산자를 사용하지 않은 도커파일
Fig. 18. Base image python:3.8-alpine dockerfile

```
1 FROM python:3.8-slim
2
3 WORKDIR /app
4
5 COPY test.py .
6
7 RUN pip install requests && \
8     pip install numpy && \
9     pip install pandas && \
10    pip install matplotlib && \
11    pip install tensorflow
12
13 CMD ["python", "test.py"]
```

그림 19. && 연산자를 사용한 도커파일
Fig. 19. Dockerfile with && operator

```
oem@user:~/dockerimage$ docker build -f basedockerfile -t python3:base .
[+] Building 199.3s (13/13) FINISHED                                docker:default
 => [internal] load build definition from basedockerfile                   0.0s
 => => transferring dockerfile: 247B                                       0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim         7.3s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [1/8] FROM docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133   0.0s
 => => resolve docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133   0.0s
 => [internal] load build context                                          0.0s
 => => transferring context: 28B                                           0.0s
 => CACHED [2/8] WORKDIR /app                                              0.0s
 => CACHED [3/8] COPY test.py .                                            0.0s
 => [4/8] RUN pip install requests                                         2.0s
 => [5/8] RUN pip install numpy                                            6.8s
 => [6/8] RUN pip install pandas                                           6.6s
 => [7/8] RUN pip install matplotlib                                       8.4s
 => [8/8] RUN pip install tensorflow                                     158.7s
 => exporting to image                                                     9.3s
 => => exporting layers                                                    9.3s
 => => writing image sha256:0c2f07a922fe1fabd95544796571344435f65b68ad9ab   0.0s
 => => naming to docker.io/library/python3:base                            0.0s
```

그림 20. 그림 18의 도커파일을 빌드한 모습
Fig. 20. Build the dockerfile shown in Figure 18

그림 21은 && 연산자를 사용하여 여러 명령어를 하나의 RUN 명령어로 결합한 도커파일을 빌드한 결과를 보여준다. 명령어들이 하나의 레이어로 결합되어 빌드되며, 이로 인해 빌드 시간이 줄어드는 것을 보여준다.

본 3-3-1절 실험을 통해 && 연산자를 사용하여 도커파일의 레이어 수를 줄이면 빌드 시간이 대략 36초 단축되는 것을 확인했다. 또한, 그림 22의 결과를 보면 && 연산자를 사용한 도커파일의 도커 이미지(2.43GB)가 && 연산자 사용하지 않은 도커파일의 도커 이미지(2.5 GB)에 비해 0.07GB만큼 이미지 크기가 더 작아지는 결과를 볼 수 있다.

```
oem@user:~/dockerimage$ docker build -f dockerfiledusruf -t python3:dusruf .
[+] Building 163.8s (9/9) FINISHED                                  docker:default
 => [internal] load build definition from dockerfiledusruf                 0.0s
 => => transferring dockerfile: 269B                                       0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim         4.5s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [1/4] FROM docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133   0.0s
 => => resolve docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133   0.0s
 => [internal] load build context                                          0.0s
 => => transferring context: 28B                                           0.0s
 => CACHED [2/4] WORKDIR /app                                              0.0s
 => CACHED [3/4] COPY test.py .                                            0.0s
 => [4/4] RUN pip install requests &&     pip install numpy &&     pip   148.8s
 => exporting to image                                                    10.5s
 => => exporting layers                                                   10.5s
 => => writing image sha256:c7a0495f58895b3c350e54d8ab1256324ec9a707089bd   0.0s
 => => naming to docker.io/library/python3:dusruf                          0.0s
```

그림 21. 그림 19의 도커파일을 빌드한 모습
Fig. 21.Build the dockerfile shown in Figure 19

```
oem@user:~/dockerimage$ docker images python3
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
python3      dusruf    c7a0495f5889   15 minutes ago   2.43GB
python3      base      0c2f07a922fe   25 minutes ago   2.5GB
```

그림 22. 3-3-1 실험의 도커 이미지 크기
Fig. 22. Docker image size for experiments in 3-3-1

3.3.2 불필요한 캐시를 삭제하는 실험

도커 이미지의 크기를 줄이기 위한 방법 중 하나는 불필요한 캐시 파일을 삭제하는 것이다. 캐시는 rm -rf /var/lib/apt/lists/* 명령어를 통해 삭제할 수 있다. 도커파일에서 apt-get update 명령어를 사용하여 패키지 목록을 업데이트하고, apt-get install 명령어로 패키지를 설치한 후, 패키지 목록이 /var/lib/apt/lists/ 디렉토리에 저장되는데 이 디렉토리의 내용은 패키지 설치가 끝난 후 더 이상 필요하지 않으므로, 이 캐시 파일을 삭제하여 도커 이미지의 크기를 줄이는 데에 도움을 준다.

그림 23은 apt-get update와 apt-get install 명령어를 사용한 후, 캐시 파일을 삭제하지 않은 도커파일을 보여준다. 이러한 도커파일은 불필요한 캐시 파일을 포함하여 최종 이미지의 크기가 불필요하게 커질 수 있다.

그림 24는 apt-get update와 apt-get install 명령어를 사용한 후, rm -rf /var/lib/apt/lists/* 명령어를 추가하여

```
1 FROM python:3.8-slim
2
3 WORKDIR /app
4
5 RUN pip install requests && \
6     pip install numpy && \
7     pip install pandas && \
8     pip install matplotlib && \
9     pip install tensorflow
10
11 RUN apt-get update && \
12     apt-get install -y curl git
13
14 COPY test.py .
15
16 CMD ["python", "test.py"]
```

그림 23. 불필요한 캐시를 삭제하지 않는 도커파일
Fig. 23. Dockerfile that don't delete unnecessary cache

```
1 FROM python:3.8-slim
2
3 WORKDIR /app
4
5 RUN pip install requests && \
6     pip install numpy && \
7     pip install pandas && \
8     pip install matplotlib && \
9     pip install tensorflow
10
11 RUN apt-get update && \
12     apt-get install -y curl git && \
13     rm -rf /var/lib/apt/lists/*
14
15 COPY test.py .
16
17 CMD ["python", "test.py"]
```

그림 24. 불필요한 캐시를 삭제한 도커파일
Fig. 24. Dockerfile with unnecessary cache deleted

캐시 파일을 삭제한 도커파일을 보여준다. 이러한 도커파일은 불필요한 파일을 제거하여 최종 이미지의 크기를 줄이는데 도움을 줄 수 있다.

그림 25는 그림 23의 도커파일를 빌드한 모습이다. 불필요한 캐시 파일이 포함된 상태로 빌드되어 최종 이미지의 크기가 불필요하게 커질 수 있다. 실제로 최종 이미지 크기는 2.55GB가 나왔다.

그림 26은 그림 24의 도커파일를 빌드한 모습이다. 불필요한 파일이 제거된 상태로 빌드되어 최종 이미지의 크기가 작아지게 도움을 준다. 실제로 최종 이미지 크기는 2.53GB가 나왔다.

그림 27의 이미지 크기를 보면 불필요한 캐시 파일을 삭제한 도커파일이 그렇지 않은 경우에 비해 최종

```
oem@user:~/dockerimage$ docker build -f cachnodelete -t python3:cachnodelete .
[+] Building 35.9s (10/10) FINISHED
 => [internal] load build definition from cachnodelete
 => => transferring dockerfile: 363B
 => [internal] load metadata for docker.io/library/python:3.8-slim
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/5] FROM docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133fa57e2c
 => => resolve docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133fa57e2c
 => [internal] load build context
 => => transferring context: 28B
 => CACHED [2/5] WORKDIR /app
```

그림 25. 그림 23의 도커파일를 빌드한 모습
Fig. 25. Build the dockerfile shown in Figure 23

```
oem@user:~/dockerimage$ docker build -f cachdelete -t python3:cachdelete .
[+] Building 5.0s (10/10) FINISHED
 => [internal] load build definition from cachdelete
 => => transferring dockerfile: 320B
 => [internal] load metadata for docker.io/library/python:3.8-slim
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/5] FROM docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133fa57e
 => => resolve docker.io/library/python:3.8-slim@sha256:f8b4609a66cdaa133fa57e
 => [internal] load build context
 => => transferring context: 28B
```

그림 26. 그림 24의 도커파일를 빌드한 모습
Fig. 26. Build the dockerfile shown in Figure 24

```
oem@user:~/dockerimage$ docker images python3
REPOSITORY   TAG           IMAGE ID       CREATED         SIZE
python3      cachdelete    2610c594d1d1   20 hours ago    2.53GB
python3      cachnodelete  918c372516ca   20 hours ago    2.55GB
```

그림 27. 3-3-2 실험의 도커 이미지 크기
Fig. 27. Docker image size for experiments in 3-3-2

이미지 크기가 더 작아진 것을 확인할 수 있었다. 불필요한 캐시 파일을 삭제하지 않은 경우 이미지 크기는 2.55GB로 나왔고 불필요한 캐시 파일을 삭제한 경우의 이미지 크기는 2.53GB로 줄어든 걸 확인할 수 있다.

3.3.3 다단계 빌드를 활용한 실험

다단계 빌드는 빌드 과정에서 필요한 도구와 라이브러리는 중간 단계에서만 사용하고, 최종 도커 이미지에 꼭 필요한 것만 남기기 때문에 도커 이미지 크기를 경량화 시킬 수 있다.

그림 28은 다단계 빌드를 사용하지 않은 도커파일을 보여주고 있다. 모든 빌드 도구와 라이브러리가 최종 이미지에 포함되어 있다.

그림 29는 다단계 빌드를 사용한 도커파일을 보여주고 있다. 중간 단계에서는 코드 빌드에 필요한 모든 도구와 라이브러리가 포함되지만, 최종 단계에서는 꼭 필요한 파일만 포함 되어 있기 때문에 최종 도커 이미지의 크기 경량화하는 것에 도움을 준다.

그림 30을 보면 다단계를 이용하지 않은 도커 이미지보다 다단계를 이용한 도커 이미지의 크기가 훨씬 작아진 걸 확인할 수 있다. 실제로 다단계 빌드를 사용하지 않는 도커 이미지의 크기는 1.91GB가 나왔고 다단계 빌드를 사용한 도커 이미지의 크기는 209MB로

```
1 FROM python:3.8-slim
2
3 WORKDIR /app
4
5 RUN pip install --no-cache-dir requests \
6     numpy \
7     pandas \
8     matplotlib \
9     tensorflow
10
11 RUN apt-get update && \
12     apt-get install -y --no-install-recommends curl git && \
13     rm -rf /var/lib/apt/lists/*
14
15 COPY test.py /app
16
17 CMD ["python","test.py"]
```

그림 28. 다단계빌드를 사용하지 않은 도커파일
Fig. 28. Dockerfile without using a multi-stage build

```
FROM python:3.8-slim AS build

WORKDIR /app

RUN pip install --no-cache-dir requests \
    numpy \
    pandas \
    matplotlib \
    tensorflow

FROM python:3.8-slim

WORKDIR /app

RUN apt-get update && \
    apt-get install -y --no-install-recommends curl git && \
    rm -rf /var/lib/apt/lists/*


COPY --from=build /usr/local/bin /usr/local/bin

COPY test.py .

CMD ["python", "test.py"]
```

그림 29. 다단계빌드를 사용한 도커파일
Fig. 29. Dockerfile with multi-stage build

```
oem@user:~/dockerimage$ docker images python3
REPOSITORY   TAG           IMAGE ID       CREATED          SIZE
python3      twostage      05ddf9aee226   2 minutes ago    209MB
python3      onestage      8df090074889   14 minutes ago   1.91GB
```

그림. 30. 3-3-3 실험의 도커 이미지 크기
Fig. 30. Docker image size for experiments in 3-3-3

도커 이미지의 크기가 크게 경량화된 것을 확인할 수 있었다.

Ⅳ. 도커 이미지 경량화 실험에 대한 고찰

4.1 캐싱의 중요성

도커는 빌드 과정에서 각 단계의 결과를 캐시하여 빌드 시간을 단축할 수 있다. 이 캐싱은 동일한 도커파일이 반복될 때 매우 유용하지만, 캐시를 효과적으로 활용하려면 명령어의 순서가 중요하다. 자주 변경되지 않는 명령어는 상단에 배치하고, 자주 변경되는 명령어는 하단에 배치하여 레이어 캐시를 효율적으로 활용할 수 있다. 예를 들어, 종속성 설치와 같은 작업을 먼저 수행하고, 이후 애플리케이션 소스 코드를 복사하면 종속성 변경에 따라 전체 도커 이미지를 재빌드하지 않아도 되는 것을 확인할 수 있었다[5].

4.2 베이스 이미지의 중요성

- **이미지 크기** : python:3.8-alpine 베이스 이미지는 가장 작은 크기를 제공하여, 저장 공간 측면에서 가장 효율적이다. python:3.8 베이스 이미지는 상대적으로 큰 베이스 이미지의 크기를 가지고 있는 것을 확인했다.
- **빌드 시간** : python:3.8-alpine 베이스 이미지는 가장 긴 빌드 시간을 기록했다. 이는 Alpine Linux의 musl 사용으로 인해 많은 라이브러리의 C 코드 컴파일이 필요하기 때문이다[7]. python:3.8 베이스 이미지의 빌드 시간은 상대적으로 짧은 것을 확인했다.

표 1. 3-2-2 실험의 베이스 이미지의 크기와 빌드 시간 비교
Table 1. Comparing the size and build time of the base image for the experiment in Section 3-2-2

Base image	Size(MB)	Time(second)
Python:3.8	3,123.2	147.1
Python:3.8-slim	1,970.52	164.9
Python:3.8-alpine	700	2,423.1

표 1은 3-3-2절에 있는 실험의 도커 python 베이스 이미지의 크기와 빌드 시간을 비교한 것이다. python:3.8 베이스 이미지와 python:3.8-slim 베이스 이미지는 상대적으로 짧은 빌드 시간을 나타냈다. 따라서, python:3.8-slim 베이스 이미지는 빌드 시간과 이미지 크기 간의 균형을 잘 맞춘 옵션으로, 대부분의 일반적인 python 애플리케이션에 적합하다. python:3.8-alpine 베이스 이미지는 가장 경량화된 이미지이지만, C 라이브러리와의 호환성 문제로 인해 긴 빌드 시간을 고려해야 한다. python:3.8 베이스 이미지는 기능이 풍부하지만, 도커 이미지 크기 면에서 가장 무겁다[7,8].

- python:3.8 베이스 이미지의 크기는 3123.2MB로 가장 크고 빌드 시간은 147초 가장 짧게 나왔다.
- python:3.8-slim 베이스 이미지의 크기는 1979.52MB로 상대적으로 작게 나왔고 빌드 시간은 164초로 나왔다.
- python:3.8-alpine 베이스 이미지의 크기는 700MB로 가장 작게 나왔지만 빌드 시간은 2423.1초로 가장 길게 나왔다.

표 2. 3-2-2 실험의 베이스 이미지의 크기와 빌드 시간 비교 시각화 그래프
Table 2. Graph of Comparing the size and build time of the base image for the experiment in Section 3-2-2

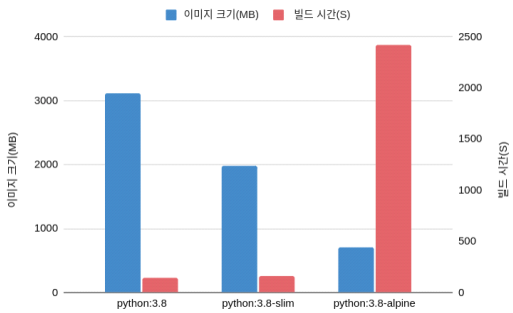


표 2는 표 1의 데이터를 시각적으로 표현한 그래프이다.

4.3 레이어 최적화의 중요성

도커파일은 RUN, COPY, ADD 명령어마다 새로운 레이어를 생성된다. 레이어 수를 줄이는 것은 도커 이미지의 경량화 뿐만 아니라 빌드 속도에도 영향을 주고 있다. 여러 RUN 명령어를 && 연산자를 사용하여 하나로 결합하면 레이어 수를 줄일 수 있다.

4.4 불필요한 캐시 제거의 중요성

도커 이미지의 최종 크기를 줄이는 데 있어 불필요한 캐시 제거는 효과적인 방법이다. --no-cache-dir 플래그를 사용하여 pip 다운로드 캐시를 제거하면, 최종 이미지에서 불필요한 파일이 줄어들어 경량화 할 수 있다.

4.5 다단계 빌드 활용의 중요성

다단계 빌드는 복잡한 빌드 환경을 단순화하고 최종 도커 이미지를 경량화 하는 데 유용하다. 필요한 도구와 파일을 빌드 과정에서만 사용하고 최종 도커 이미지에서 제외할 수 있기 때문에 도커 이미지의 크기가 줄어드는 것을 확인할 수 있다.

V. 결 론

본 논문에서는 자원 제약적 디바이스에서 도커를 활용하기 위한 도커 이미지 경량화 연구를 통해 도커 이미지의 경량화와 빌드 성능 최적화를 위한 다양한 전략의 효과를 확인할 수 있었다. 캐싱을 효율적으로 활용하기 위해 명령어 순서를 최적화하여 빌드 시간을 단축할 수 있음을 확인하였다. 자주 변경되지 않는 명령어를 상단에 배치하고, 자주 변경되는 명령어를 하단에 배치함으로써 캐시를 효과적으로 활용할 수 있었다. 또한, 적절한 베이스 이미지를 선택하는 것이 중요하다는 것을 확인할 수 있었다. 예를 들어, python:3.8-alpine 베이스 이미지는 가장 작은 크기를 제공하지만 빌드 시간이 길다는 단점이 있다. 반면, python:3.8-slim 베이스 이미지는 빌드 시간과 이미지 크기 간의 균형을 잘 맞춘 옵션으로, 대부분의 일반적인 python 애플리케이션에 적합하다. RUN 명령어를 && 연산자로 결합하여 레이어 수를 줄임으로써 도커 이미지의 크기와 빌드 속도를 개선할 수 있었다. apt 패키지 관리자의 캐시 파일을 삭제하여 도커 이미지의 크기를 줄이는 효과를 확인하였다. 또한, 다단계 빌드를 활용하여 빌드 과정에서만 필요한 도구와 파일을 최종 이미지에서 제외함으로써 도커 이미지의 크기를 크게 줄일 수 있었다.

향후 연구에서는 객체 인식과 같은 무거운 python 기반 딥러닝/머신러닝 애플리케이션을 대상으로 한 도커 이미지 최적화 방안에 대해 연구할 필요가 있다고 생각한다. 이러한 애플리케이션은 더 많은 라이브러리와 종속성을 필요로 하므로, 더욱 효율적인 캐싱 전략과 레이어 최적화 방안이 요구된다. 이를 통해 도커 이미지를 더욱 경량화하고 빌드 성능을 최적화할 수 있을 것이다.

References

[1] *What is docker?*, Retrieved Oct. 14, 2024, from https://docs.docker.com/get-started/docker-overview/

[2] *What is a Container?*, Retrieved Oct. 14, 2024, from https://www.docker.com/resources/what-container/

[3] *Docker base images*, Retrieved Oct. 14, 2024, from https://docs.docker.com/build/building/base-images/

[4] *Dockerfile overview*, Retrieved Oct. 14, 2024, from https://docs.docker.com/build/concepts/dockerfile/

[5] *Docker build cache*, Retrieved Oct. 14, 2024, from https://docs.docker.com/ build/cache/

[6] *Docker Official Images*, Retrieved Oct. 14, 2024, from https://docs.docker.com/trusted-content/official-images/

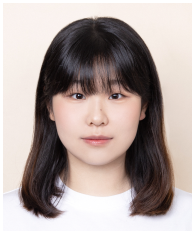
[7] *Using Alpine can make Python docker builds 50×slower*, Retrieved Oct. 14, 2024, from https://pythonspeed.com/articles/alpine-docker-python

[8] *musl-libc-Alpine's Greatest Weakness*, Retrieved Oct. 14, 2024, from https:// www.linkedin.com/pulse/musl-libc-alpines-greatest-weakness-rogan-lynch/

[9] S.-Y. Bae, "A comparative analysis of domestic and foreign docker container-based research trends," *J. Korea Contents Assoc.,* vol. 22, no. 10, pp. 742-753, 2022. (https://doi.org/10.5392/JKCA.2022.22.10.742)

[10] *docker image ls*, Retrieved Oct. 14, 2024, from https://docs.docker.com/reference/cli/docker/image/ls/

임 수 연 (Suyeon Lim)



2025년 2월 : 대전대학교 AI융합학과 학사

2025년 3월~현재 : 대전대학교 컴퓨터공학과 석사과정

<관심분야> 컴퓨터 비전, 자율주행, 자연어 처리

[ORCID:0009-0004-2611-9189]

홍 용 근 (Yong-Geun Hong)

1997년 2월 : 경북대학교 컴퓨터공학과 학사

1999년 2월 : 경북대학교 컴퓨터공학과 석사

2013년 2월 : 경북대학교 컴퓨터공학과 박사

2001년~2020년 : 한국전자통신연구원 실장/책임연구원

2021년~현재 : 대전대학교 AI소프트웨어학부 부교수

<관심분야> 온디바이스 AI, 지능형 사물인터넷, 자율주행

[ORCID:0000-0003-2974-3820]