ML Compiler

Apache/TVM

머신러닝 컴파일러

Feb 29, 2024

김호중 (Kim Ho-Jung) _ godmode2k@hotmail.com

Table of Contents

Source: 사용자 튜토리얼 https://tvm.apache.org/docs/tutorial/introduction.html

Version: v0.16.dev0

처음 사용자를 위한 Apache/TVM 가이드 입니다. 물론 저도 처음입니다. 대부분의 내용은 위에 있는 Source URL (Apache/TVM 공식 온라인 문서)를 번역하였습니다.

마지막 부분에 몇몇 예제를 첨부하였습니다. Apache/TVM을 사용해 보시는 데에 도움이 되었으면 합니다.

1. Apache/TVM

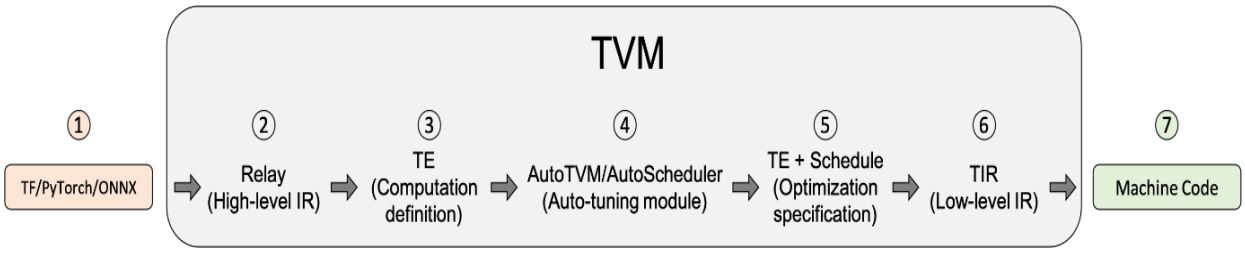
TVM (Tensor Virtual Machine)

1.1 소개

Apache TVM은 CPU, GPU 및 머신러닝 가속기를 위한 오픈소스 머신러닝 컴파일러 프레임워크입니다. 머신러닝 엔지니어가 모든 하드웨어 백엔드에서 최적화하고 효율적으로 계산을 실행할 수 있도록 하는 것을 목표로 합니다.

TVM 및 모델 최적화 개요

아래 다이어그램은 머신 모델이 TVM 최적화 컴파일러 프레임워크로 변환될 때 수행하는 단계를 보여줍니다.



1. Tensorflow, PyTorch 또는 ONNX와 같은 프레임워크에서 모델을 가져옵니다. 임포터 레이어는(Importer Layer) TVM이 Tensorflow, PyTorch 또는 ONNX와 같은 다른 프레임워크에서 모델을 수집할 수 있는 곳입니다. TVM이 각 프론트엔드에 제공하는 지원 수준은 오픈소스 프로젝트를 지속적으로 개선하고 있기 때문에 다양합니다. 모델을 TVM으로 가져오는 데 문제가 있는 경우 ONNX로 변환해 볼 수 있습니다.
2. TVM의 고급 모델 언어인 Relay로 번역합니다. TVM으로 가져온 모델은 Relay에 표시됩니다. 릴레이는 신경망을 위한 함수형 언어이자 중간 표현(IR)이며 다음을 지원합니다
   - 전통적인 데이터 흐름 스타일 표현

- 함수형 스타일, let 바인딩을 통해 완전한 기능을 갖춘 차별화 가능한 언어
- 사용자가 두 가지 프로그래밍 스타일을 혼합할 수 있는 기능

Relay는 그래프 수준 최적화 패스를 적용하여 모델을 최적화합니다.

3. TE(Tensor Expression) 표현으로 낮추기. Lowering은 하이레벨의 표현이 로우레벨의 표현으로 변환되는 경우입니다. 하이레벨의 최적화를 적용한 후 Relay는 FuseOps 패스를 실행하여 모델을 여러 개의 작은 하위 그래프로 분할하고 하위 그래프를 TE 표현으로 낮춥니다. TE(Tensor Expression)는 텐서 계산을 설명하기 위한 도메인 특화 언어입니다. TE는 또한 타일링, 벡터화, 병렬화, 언롤링 및 퓨전과 같은 로우레벨의 루프 최적화를 지정하기 위한 몇 가지 스케줄 (**Schedule**: 프로그램에서 계산의 루프를 변환하는 계산의 변환 집합) 기본 요소를 제공합니다. 릴레이 표현을 TE 표현으로 변환하는 프로세스를 지원하기 위해 TVM에는 일반적인 텐서 연산자(예: conv2d, transpose)의 사전 정의된 템플릿이 있는 TOPI(Tensor Operator Inventory)가 포함되어 있습니다.

4. 자동 튜닝 모듈 AutoTVM 또는 AutoScheduler를 사용하여 최적의 스케줄을 검색합니다. 스케줄은 TE에 정의된 연산자 또는 하위 그래프에 대한 로우레벨 루프 최적화를 지정합니다. 자동 튜닝 모듈은 최적의 스케줄을 검색하고 비용 모델 및 온디바이스 측정과 비교합니다. TVM에는 두 개의 자동 튜닝 모듈이 있습니다.
   - AutoTVM: 템플릿 기반 자동 튜닝 모듈입니다. 검색 알고리즘을 실행하여 사용자 정의 템플릿에서 튜닝 가능한 노브(knobs)에 가장 적합한 값을 찾습니다. 일반 연산자의 경우 해당 템플릿이 이미 TOPI에 제공되어 있습니다.
   - AutoScheduler (또는 Ansor): 템플릿이 없는 자동 튜닝 모듈입니다. 사전 정의된 스케줄 템플릿이 필요하지 않습니다. 대신 계산 정의를 분석하여 검색 공간을 자동으로 생성합니다. 그런 다음 생성된 검색 공간에서 최상의 스케줄을 검색합니다.

5. 모델 컴파일을 위한 최적의 구성을 선택합니다. 튜닝 후 자동 튜닝 모듈은 JSON 형식으로 튜닝 레코드를 생성합니다. 이 단계에서는 각 서브그래프에 가장 적합한 스케줄을 선택합니다.

6. TVM의 로우레벨의 중간 표현인 TIR(Lower to Tensor Intermediate Representation)입니다. 튜닝 단계를 기반으로 최적의 구성을 선택한 후 각 TE 서브그래프는 TIR로 낮아지고 로우레벨의 최적화 패스에 의해 최적화 됩니다. 다음으로, 최적화된 TIR이 하드웨어 플랫폼의 대상 컴파일러로 낮아집니다. 이는 프로덕션에 배포할 수 있는 최적화된 모델을 생성하기 위한 최종 코드 생성 단계입니다. TVM은 다음과 같은 여러 컴파일러 백엔드를 지원합니다.

- LLVM은 표준 x86과 ARM 프로세서, AMDGPU와 NVPTX 코드 생성, 그리고 LLVM에서 지원하는 기타 플랫폼을 포함한 임의의 마이크로프로세서 아키텍처를 대상으로 할 수 있습니다.
- NVIDIA의 컴파일러인 NVCC와 같은 특수 컴파일러.
- TVM의 BYOC(Bring Your Own Codegen) 프레임워크를 통해 구현되는 임베디드 및 특수 대상입니다.

7. 기계어 코드로 컴파일합니다. 이 프로세스가 끝나면 컴파일러별로 생성된 코드를 기계어 코드로 낮출 수 있습니다.

TVM은 모델을 연결 가능한 객체 모듈로 컴파일할 수 있으며, 모델을 동적으로 로드하는 C API와 Python 및 Rust와 같은 다른 언어의 진입점을 제공하는 경량 TVM 런타임으로 실행할 수 있습니다. TVM은 런타임이 단일 패키지의 모델과 결합되는 번들 배포를 빌드할 수도 있습니다.

1.2 필수 구성 요소

1. Apache/TVM

CPU, GPU 및 특수 가속기를 위한 개방형 딥러닝 컴파일러 스택

공식 사이트

https://tvm.apache.org/

https://tvm.apache.org/docs/tutorial/introduction.html

https://tvm.apache.org/docs/install/from_source.html

https://github.com/apache/tvm

예제

https://github.com/apache/tvm/releases/download/v0.15.0/apache-tvm-src-v0.15.0.tar.gz

TVM 소스코드

https://github.com/apache/tvm/archive/refs/tags/v0.15.0.tar.gz

https://github.com/apache/tvm/archive/refs/tags/v0.15.0.zip

Git Clone

$ git clone --recursive https://github.com/apache/tvm.git

2. TLCPack

TLCPack – 텐서 학습 컴파일러 바이너리 패키지. 커뮤니티에서 유지 관리하는 딥 러닝 컴파일러의 바이너리 빌드입니다. TLCPack에는 추가 소스 코드 릴리스가 포함되어 있지 않습니다. Apache TVM에서 소스 코드를 가져와 다른 빌드 구성을 켜서 바이너리를 빌드합니다.

공식사이트

https://tlcpack.ai/

https://github.com/tlc-pack/tlcpack

도커 이미지

https://hub.docker.com/r/tlcpack/ci-cpu

https://hub.docker.com/r/tlcpack/ci-gpu

이미지 다운로드

$ sudo docker pull tlcpack/ci-cpu:20240105-165030-51bdaec6

$ sudo docker pull tlcpack/ci-gpu:20240105-165030-51bdaec6

여기에선 Ubuntu 20.04 LTS 버전이 설치된 상태에서 <tlcpack/ci-cpu> 도커 이미지를 사용합니다.

$ sudo docker pull tlcpack/ci-cpu:20240105-165030-51bdaec6

1.3 빌드

tclpack/ci-cpu:20240105-165030-51bdaec6 도커 이미지를 사용

$ sudo docker pull tlcpack/ci-cpu:20240105-165030-51bdaec6

$ sudo docker run -it --rm -v /work:/work tlcpack/ci-cpu:20240105-165030-51bdaec6

(container) {

apt-get update && apt-get install vim

cd /work

Git Clone

TVM HOME path: /work/tvm

git clone --recursive https://github.com/apache/tvm.git

cd tvm

빌드 환경설정

mkdir build && cd build

cp ../cmake/config.cmake .

vim config.cmake

(**EDIT** config.cmake) {

set(USE_CUDA OFF) 또는 set(USE_CUDA ON)

set(USE_GRAPH_EXECUTOR ON)

set(USE_PROFILER ON)

```
    # 디버그에 IRs 포함, set(USE_RELAY_DEBUG ON)
    set(USE_RELAY_DEBUG OFF) 또는 set(USE_RELAY_DEBUG ON)
    # TVM_LOG_DEBUG. 환경변수 설정
    # export TVM_LOG_DEBUG="ir/transform.cc=1,relay/ir/transform.cc=1"

    # CMake의 찾기 검색으로 LLVM 활성화
    set(USE_LLVM OFF) -> set(USE_LLVM ON)
    # 또는 path를 직접 설정
    #set(USE_LLVM OFF) -> set(USE_LLVM /path/to/llvm-config)
} EDIT config.cmake
```

빌드

(release) # cmake .. (debug) # cmake –DCMAKE_BUILD_TYPE=Debug ..

make -j4

Path 설정

vim $HOME/.bashrc

(**EDIT** $HOME/.bashrc) {

```
    export TVM_HOME=/work/tvm
    export PYTHONPATH=$TVM_HOME/python:${PYTHONPATH}
```

} Path 설정

source $HOME/.bashrc

또는 셸 스크립트를 실행

```
# cat add_tvm_env_path.sh

#!/bin/sh

unset TVM_HOME

unset PYTHONPATH

echo 'export TVM_HOME=/work/tvm' >> $HOME/.bashrc

echo 'export PYTHONPATH=$TVM_HOME/python:${PYTHONPATH}' >> $HOME/.bashrc

source $HOME/.bashrc

# source add_tvm_env_path.sh
```

} (container)

1.4 실행

TVMC

```
# python -m tvm.driver.tvmc
```

1.5 TVMC를 사용한 모델 컴파일 및 최적화

이 섹션에서는 TVM command-line 드라이버인 TVMC를 사용합니다. TVMC는 command-line 인터페이스를 통해 모델의 자동 튜닝, 컴파일, 프로파일링 및 실행을 할 수 있는 도구입니다.

이 섹션을 완료하면 TVMC를 사용하여 다음 작업을 수행합니다.

- TVM 런타임을 위해 사전 학습된 ResNet-50 v2 모델을 컴파일합니다.
- 컴파일된 모델을 통해 실제 이미지를 실행하고 출력 및 모델 성능을 해석합니다.
- TVM을 사용하여 CPU에서 모델을 튜닝합니다.
- TVM에서 수집한 튜닝 데이터를 사용하여 최적화된 모델을 다시 컴파일합니다.
- 최적화된 모델을 통해 이미지를 실행하고 출력과 모델 성능을 비교합니다.

이 섹션의 목표는 TVM 및 TVMC의 기능에 대한 개요를 제공하고 TVM의 작동 방식을 이해하기 위한 단계를 설정하는 것입니다.

TVMC 사용

TVMC는 TVM Python 패키지의 일부인 Python 애플리케이션입니다. Python 패키지를 사용하여 TVM을 설치하면 **tvmc**라는 command-line 애플리케이션이 설치됩니다. 이 명령의 위치는 플랫폼 및 설치 방법에 따라 다릅니다.

또는 **$PYTHONPATH**에 Python 모듈로 TVM이 있는 경우 실행 가능한 python 모듈인 **python -m tvm.driver.tvmc**를 통해 command-line 드라이버 기능에 액세스할 수 있습니다.

단순하게 이 튜토리얼에서는 **TVMC <options>** command-line을 사용하는 tvmc 를 언급 하지만 **python -m tvm.driver.tvmc <options>** 를 사용하여 동일한 결과를 얻을 수 있습니다

다음을 사용하여 도움말 페이지를 확인할 수 있습니다.

```
tvmc --help
```

tvmc에서 사용할 수 있는 TVM의 주요 기능은 부속 명령 **컴파일**, **실행** 그리고 **튜닝**입니다. 지정된 하위 명령의 특정 옵션에 대해 읽으려면 **tvmc <subcommand> --help** 를 사용합니다. 이 튜토리얼에서는 이러한 각 명령을 다루지만 먼저 작업할 미리 학습된 모델을 다운로드해야 합니다.

모델 구하기

이 튜토리얼에서는 ResNet-50 v2를 사용합니다. ResNet-50은 50개 계층으로 구성된 컨벌루션 신경망 (Convolutional Neural Network, CNN)으로 영상을 분류하도록 설계되었습니다. 우리가 사용할 모델은 1,000개의 서로 다른 분류가 있는 100만 개 이상의 이미지에 대해 사전 훈련되었습니다. 네트워크의 입력 영상 크기는 224x224입니다. ResNet-50 모델이 어떻게 구성되어 있는지 자세히 알아보려면 무료로 제공되는 ML 모델 뷰어인 **Netron** (https://netron.app/)을 다운로드하는 것을 추천합니다.

이 튜토리얼에서는 ONNX 형식의 모델을 사용합니다.

wget
https://github.com/onnx/models/raw/b9a54e89508f101a1611cd64f4ef56b9cb62c7cf/vision/classification/resnet/model/resnet50-v2-7.onnx

지원되는 모델 형식

TVMC는 Keras, ONNX, TensorFlow, TFLite 및 Torch로 만든 모델을 지원합니다. 사용 중인 모델 형식을 명시적으로 제공해야 하는 경우 **--model-format** 옵션을 사용합니다. 자세한 내용은 **tvmc compile --help** 를 참조하세요.

TVM에 ONNX 지원 추가

TVM은 시스템에서 사용할 수 있는 ONNX Python 라이브러리에 의존합니다. **pip3 install --user onnx onnxoptimizer** 명령을 사용하여 ONNX를 설치할 수 있습니다. root 액세스 권한이 있고 ONNX를 전역적으로 설치하려는 경우 **--user** 옵션을 제거할 수 있습니다. **onnxoptimizer** 종속성은 선택 사항이며 **onnx>=1.9**에만 사용됩니다.

ONNX 모델을 TVM 런타임으로 컴파일

ResNet-50 모델을 다운로드했으면 다음 단계는 모델을 컴파일하는 것입니다. 이를 위해 **tvmc compile** 을 사용할 것입니다. 컴파일 프로세스에서 얻은 출력은 대상 플랫폼의 동적 라이브러리로 컴파일된 모델의 TAR 패키지입니다. TVM 런타임을 사용하여 대상 장치에서 해당 모델을 실행할 수 있습니다.

```
This may take several minutes depending on your machine
# python -m tvm.driver.tvmc compile ₩
--target "llvm" ₩
--input-shapes "data:[1,3,224,224]" ₩
--output resnet50-v2-7-tvm.tar resnet50-v2-7.onnx
```

모듈에서 **tvmc compile** 이 생성한 파일을 살펴 보겠습니다.

```
# mkdir model
# tar -xvf resnet50-v2-7-tvm.tar -C model
# ls model
```

세 개의 파일이 나열됩니다.

mod.so: TVM 런타임에서 로드할 수 있는 C++ 라이브러리로 표현되는 모델입니다.

mod.json: TVM 릴레이 계산 그래프의 텍스트 표현입니다.

mod.params: 미리 학습된 모델에 대한 매개 변수를 포함하는 파일입니다.

이 모듈은 애플리케이션에서 직접 로드할 수 있으며 모델은 TVM 런타임 API를 통해 실행할 수 있습니다.

올바른 대상 정의

올바른 대상(option **--target**)을 지정하면 대상에서 사용할 수 있는 하드웨어 기능을 활용할 수 있으므로 컴파일된 모듈의 성능에 큰 영향을 미칠 수 있습니다. 자세한 내용은 **Auto-tuning a convolutional network for x86 CPU**을 참조하세요. 선택적 기능과 함께 실행 중인 CPU를 식별하고 대상을 적절하게 설정하는 것이 좋습니다.

https://tvm.apache.org/docs/how_to/tune_with_autotvm/tune_relay_x86.html#tune-relay-x86

TVMC를 사용하여 컴파일된 모듈에서 모델 실행

이제 모델을 이 모듈로 컴파일했으므로 TVM 런타임을 사용하여 예측을 수행할 수 있습니다. TVMC에는 TVM 런타임이 내장되어 있어 컴파일된 TVM 모델을 실행할 수 있습니다.

TVMC를 사용하여 모델을 실행하고 예측하려면 다음 두 가지가 필요합니다.

- 방금 만든 컴파일된 모듈
- 예측을 수행할 모델에 대한 유효한 입력

각 모델은 예상되는 텐서 shapes, 형식 및 데이터 유형과 관련하여 특별합니다. 이러한 이유로 대부분의 모델에는 입력이 유효한지 확인하고 출력을 해석하기 위해 몇 가지 사전 (pre-processing) 및 후처리(post-processing)가 필요합니다. TVMC는 입력과 출력 데이터 모두에 NumPy의 **.npz** 형식을 채택했습니다. 이것은 여러 배열을 파일로 직렬화하기 위해 잘 지원되는 NumPy 형식입니다.

이 튜토리얼에서 입력으로 고양이 이미지를 사용하지만 원하는 이미지로 자유롭게 대체할 수 있습니다.

https://s3.amazonaws.com/model-server/inputs/kitten.jpg

입력 전처리: preprocess.py

ResNet-50 v2 모델의 경우 입력은 ImageNet 형식이어야 합니다. 다음은 ResNet-50 v2에 대한 이미지를 전처리하는 스크립트의 예입니다.

지원되는 버전의 Python 이미지 라이브러리가 설치되어 있어야 합니다. pip3 install --user pillow 를 사용하여 설치 할 수 있습니다.

```
# pip install pillow
# python preprocess.py
```

컴파일된 모듈 실행

모델과 입력 데이터가 모두 준비되면 이제 TVMC를 실행하여 예측을 수행할 수 있습니다.

```
# python -m tvm.driver.tvmc run ₩
--inputs imagenet_cat.npz ₩
--output predictions.npz ₩
resnet50-v2-7-tvm.tar
```

.tar 모델 파일에는 C++ 라이브러리, Relay 모델에 대한 설명 및 모델에 대한 매개 변수가 포함되어 있습니다. TVMC에는 모델을 로드하고 입력에 대한 예측을 수행할 수 있는 TVM 런타임이 포함되어 있습니다. 위의 명령을 실행할 때 TVMC는 NumPy 형식의 모델 출력 텐서가 포함된 새 파일 **predictions.npz**를 출력합니다.

이 예제에서는 컴파일에 사용한 것과 동일한 컴퓨터에서 모델을 실행하고 있습니다. 경우에 따라 RPC 트래커를 통해 원격으로 실행할 수 있습니다. 이러한 옵션에 대한 자세한 내용은 **tvmc run --help**를 확인하세요.

출력 후처리: postprocess.py

앞에서 언급했듯이 각 모델에는 출력 텐서를 제공하는 고유한 방법이 있습니다.

이 경우 모델에 제공된 조회 테이블을 사용하여 ResNet-50 v2의 출력을 사람이 읽을 수 있는 형식으로 렌더링하기 위해 몇 가지 후처리를 실행해야 합니다.

아래 스크립트는 컴파일된 모듈의 출력에서 레이블을 추출하는 후처리의 예를 보여줍니다.

```
# python postprocess.py
```

이 스크립트를 실행하면 다음과 같은 출력이 생성됩니다.

```
# class='n02123045 tabby, tabby cat' with probability=0.610553
# class='n02123159 tiger cat' with probability=0.367179
# class='n02124075 Egyptian cat' with probability=0.019365
# class='n02129604 tiger, Panthera tigris' with probability=0.001273
# class='n04040759 radiator' with probability=0.000261
```

고양이 이미지를 다른 이미지로 바꾸고 ResNet 모델이 어떤 종류의 예측을 수행하는지 확인합니다.

filename: preprocess.py

```
#!python ./preprocess.py
from tvm.contrib.download import download_testdata
from PIL import Image
import numpy as np

img_url = "https://s3.amazonaws.com/model-server/inputs/kitten.jpg"
img_path = download_testdata(img_url, "imagenet_cat.png", module="data")

# Resize it to 224x224
resized_image = Image.open(img_path).resize((224, 224))
img_data = np.asarray(resized_image).astype("float32")

# ONNX expects NCHW input, so convert the array
img_data = np.transpose(img_data, (2, 0, 1))

# Normalize according to ImageNet
imagenet_mean = np.array([0.485, 0.456, 0.406])
imagenet_stddev = np.array([0.229, 0.224, 0.225])
norm_img_data = np.zeros(img_data.shape).astype("float32")
for i in range(img_data.shape[0]):
        norm_img_data[i, :, :] = (img_data[i, :, :] / 255 - imagenet_mean[i]) / imagenet_stddev[i]

# Add batch dimension
img_data = np.expand_dims(norm_img_data, axis=0)

# Save to .npz (outputs imagenet_cat.npz)
np.savez("imagenet_cat", data=img_data)
```

filename: postprocess.py

```
#!python ./postprocess.py
import os.path
import numpy as np

from scipy.special import softmax

from tvm.contrib.download import download_testdata

# Download a list of labels
labels_url = "https://s3.amazonaws.com/onnx-model-zoo/synset.txt"
labels_path = download_testdata(labels_url, "synset.txt", module="data")

with open(labels_path, "r") as f:
    labels = [l.rstrip() for l in f]

output_file = "predictions.npz"

# Open the output and read the output tensor
if os.path.exists(output_file):
    with np.load(output_file) as data:
        scores = softmax(data["output_0"])
        scores = np.squeeze(scores)
        ranks = np.argsort(scores)[::-1]

        for rank in ranks[0:5]:
            print("class='%s' with probability=%f" % (labels[rank], scores[rank]))
```

ResNet 모델 자동 튜닝

이전 모델은 TVM 런타임에서 작동하도록 컴파일되었지만 플랫폼별 최적화는 포함되지 않았습니다. 이 섹션에서는 TVMC를 사용하여 작업 플랫폼을 대상으로 최적화된 모델을 구축하는 방법을 보여줍니다.

어떤 경우에는 컴파일된 모듈을 사용하여 추론을 실행할 때 예상한 성능을 얻지 못할 수도 있습니다. 이와 같은 경우 자동 튜너를 사용하여 모델에 대한 더 나은 구성을 찾고 성능을 향상시킬 수 있습니다.

TVM 튜닝은 주어진 대상에서 더 빠르게 실행되도록 모델을 최적화하는 프로세스를 나타냅니다.

이는 모델의 정확도에 영향을 주지 않고 런타임 성능에만 영향을 미친다는 점에서 학습 또는 미세 조정(fine-tuning)과 다릅니다. 튜닝 프로세스의 일부로 TVM은 다양한 연산자 구현 변형을 실행하여 어떤 것이 가장 성능이 좋은지 확인합니다. 이러한 실행의 결과는 튜닝 레코드 파일에 저장되며 이는 궁극적으로 **tune** 서브커맨드의 출력입니다.

가장 간단한 형태의 튜닝을 수행하려면 다음 세 가지를 제공해야 합니다.

1. 이 모델을 실행하려는 장치의 대상 사양
2. 튜닝 레코드가 저장될 출력 파일의 경로
3. 튜닝할 모델의 경로

```
기본 검색 알고리즘에는 xgboost가 필요하며,
검색 알고리즘 튜닝에 대한 자세한 내용은 아래를 참조하세요.
# pip install xgboost

# python -m tvm.driver.tvmc tune ₩
--target "llvm" ₩
--output resnet50-v2-7-autotuner_records.json ₩
resnet50-v2-7.onnx
```

이 예제에서는 **--target** 플래그에 보다 구체적인 대상을 지정하면 더 나은 결과를 볼 수 있습니다. 예를 들어 Intel i7 프로세서에서는 **--target "llvm -mcpu=skylake"** 를 사용할 수 있습니다. 이 튜닝 예제에서는 LLVM을 지정된 아키텍처의 컴파일러로 사용하여 CPU에서 로컬로 튜닝합니다.

TVMC는 모델의 매개변수 공간에 대한 검색을 수행하여 연산자에 대한 다양한 구성을 시도하고 플랫폼에서 가장 빠르게 실행되는 구성을 선택합니다. 이 검색은 CPU 및 모델 작업을 기반으로 하는 안내 검색이지만 검색을 완료하는 데 몇 시간이 걸릴 수 있습니다. 이 검색의 결과는 **resnet50-v2-7-autotuner_records.json** 파일에 저장되며 이 파일은 나중에 최적화된 모델을 컴파일하는 데 사용됩니다.

튜닝 검색 알고리즘 정의

기본적으로 이 검색은 **XGBoost Grid** 알고리즘을 사용하여 안내됩니다. 모델 복잡성과 사용 가능한 시간에 따라 다른 알고리즘을 선택할 수 있습니다. 전체 목록은 **tvmc tune --help**를 참조하여 확인할 수 있습니다.

```
# python -m tvm.driver.tvmc tune ₩
--target "llvm -mcpu=skylake" ₩
--output resnet50-v2-7-autotuner_records.json ₩
resnet50-v2-7.onnx
```

보통 사용자들이 (a consumer-level) 사용하는 Skylake CPU의 경우 출력은 다음과 같습니다.

```
# [Task  1/24]  Current/Best:    9.65/  23.16 GFLOPS | Progress: (60/1000) | 130.74 s Done.
# [Task  1/24]  Current/Best:    3.56/  23.16 GFLOPS | Progress: (192/1000) | 381.32 s Done.
# [Task  2/24]  Current/Best:   13.13/  58.61 GFLOPS | Progress: (960/1000) | 1190.59 s Done.
# [Task  3/24]  Current/Best:   31.93/  59.52 GFLOPS | Progress: (800/1000) | 727.85 s Done.
# [Task  4/24]  Current/Best:   16.42/  57.80 GFLOPS | Progress: (960/1000) | 559.74 s Done.
# [Task  5/24]  Current/Best:   12.42/  57.92 GFLOPS | Progress: (800/1000) | 766.63 s Done.
# [Task  6/24]  Current/Best:   20.66/  59.25 GFLOPS | Progress: (1000/1000) | 673.61 s Done.
# [Task  7/24]  Current/Best:   15.48/  59.60 GFLOPS | Progress: (1000/1000) | 953.04 s Done.
```

```
# [Task  8/24]  Current/Best:   31.97/  59.33 GFLOPS | Progress: (972/1000) | 559.57 s Done.
# [Task  9/24]  Current/Best:   34.14/  60.09 GFLOPS | Progress: (1000/1000) | 479.32 s Done.
...
# [Task 21/24]  Current/Best:   34.37/  58.28 GFLOPS | Progress: (308/1000) | 225.37 s Done.
# [Task 22/24]  Current/Best:   15.75/  57.71 GFLOPS | Progress: (1000/1000) | 1024.05 s Done.
# [Task 23/24]  Current/Best:   23.23/  58.92 GFLOPS | Progress: (1000/1000) | 999.34 s Done.
# [Task 24/24]  Current/Best:   17.27/  55.25 GFLOPS | Progress: (1000/1000) | 1428.74 s Done.
```

튜닝 세션은 시간이 오래 걸릴 수 있으므로 **tvmc tune**은 반복 횟수(예: **--repeat** 및 **--number**), 사용할 튜닝 알고리즘 등의 측면에서 튜닝 프로세스를 사용자 지정할 수 있는 다양한 옵션을 제공합니다. 자세한 내용은 **tvmc tune --help**를 확인하세요.

어떤 상황에서는 특정 작업(즉, 가장 관련성이 높은 작업)만 튜닝하여 더 간단한 작업 부하를 튜닝하는 데 낭비되는 시간을 줄이는 것이 좋습니다. 플래그 –task는 튜닝에 사용되는 작업을 제한하는 다양한 옵션을 제공합니다(예: –task 20,22 또는 –task 16-). 사용 가능한 모든 작업은 **–task list**을 사용하여 출력 할 수 있습니다.

데이터 튜닝을 사용하여 최적화된 모델 컴파일

위의 튜닝 프로세스의 출력으로 **resnet50-v2-7-autotuner_records.json**에 저장된 튜닝 레코드를 얻었습니다.

이 파일은 다음 두 가지 방법으로 사용할 수 있습니다.

1. 추가 튜닝에 대한 입력으로(**tvmc tune --tuning-records**를 통해).
2. 컴파일러에 대한 입력으로

컴파일러는 결과를 사용하여 지정된 대상의 모델에 대한 고성능 코드를 생성합니다.

이를 위해 **tvmc compile --tuning-records** 를 사용할 수 있습니다. 자세한 내용은 **tvmc compile --help** 를 확인하세요.

이제 모델에 대한 튜닝 데이터가 수집되었으므로 최적화된 연산자를 사용하여 모델을 다시 컴파일하여 계산 속도를 높일 수 있습니다.

```
# python -m tvm.driver.tvmc compile ₩
--target "llvm" ₩
--tuning-records resnet50-v2-7-autotuner_records.json ₩
--output resnet50-v2-7-tvm_autotuned.tar ₩
resnet50-v2-7.onnx
```

최적화된 모델이 실행되고 동일한 결과가 생성되는지 확인합니다.

```
# python -m tvm.driver.tvmc run ₩
--inputs imagenet_cat.npz ₩
--output predictions.npz ₩
resnet50-v2-7-tvm_autotuned.tar
```

출력 후처리: postprocess.py

```
# python postprocess.py
```

예측이 동일한지 확인:

```
# class='n02123045 tabby, tabby cat' with probability=0.610550
# class='n02123159 tiger cat' with probability=0.367181
# class='n02124075 Egyptian cat' with probability=0.019365
# class='n02129604 tiger, Panthera tigris' with probability=0.001273
```

```
# class='n04040759 radiator' with probability=0.000261
```

튜닝된 모델과 튜닝되지 않은 모델 비교

TVMC는 모델 간의 기본 성능 벤치마킹을 위한 도구를 제공합니다. 반복 횟수를 지정하고 TVMC가 모델 런타임에 대해 보고하도록 지정할 수 있습니다(런타임 시작과 무관). 튜닝을 통해 모델 성능이 얼마나 향상되었는지 대략적으로 알 수 있습니다.

예를 들어 Intel i7 테스트 시스템에서 튜닝된 모델이 튜닝되지 않은 모델보다 47% 더 빠르게 실행되는 것을 볼 수 있습니다.

```
# python -m tvm.driver.tvmc run ₩
--inputs imagenet_cat.npz ₩
--output predictions.npz ₩
--print-time ₩
--repeat 100 ₩
resnet50-v2-7-tvm_autotuned.tar
```

```
# Execution time summary:
# mean (ms)     max (ms)      min (ms)      std (ms)
#       92.19       115.73         89.85          3.15
```

```
# python -m tvm.driver.tvmc run ₩
--inputs imagenet_cat.npz ₩
--output predictions.npz ₩
--print-time ₩
--repeat 100 ₩
resnet50-v2-7-tvm.tar
```

```
# Execution time summary:
# mean (ms)     max (ms)      min (ms)      std (ms)
```

```
#       193.32          219.97          185.04          7.11
```

최종 비고

이 튜토리얼에서는 TVM용 command-line 드라이버인 TVMC를 소개했습니다. 모델을 컴파일, 실행 및 튜닝하는 방법을 시연했습니다. 또한 입력과 출력의 사전 및 후처리의 필요성에 대해서도 논의했습니다. 튜닝 프로세스 후 최적화되지 않은 모델과 최적화 모델의 성능을 비교하는 방법을 시연했습니다.

여기서는 로컬에서 ResNet-50 v2를 사용하는 간단한 예제를 제시했습니다. 그러나 TVMC는 교차 컴파일, 원격 실행 및 프로파일링/벤치마킹을 포함한 더 많은 기능을 지원합니다.

사용 가능한 다른 옵션을 보려면 **tvmc --help**를 참조하세요.

다음 튜토리얼 <tvmc_python>에서는 TVM에 대한 Python 인터페이스를 소개하고, 그 이후의 튜토리얼인 Python 인터페이스로 모델 컴파일 및 최적화 <autotvm_relay_x86>에서는 Python 인터페이스를 사용하여 동일한 컴파일 및 최적화 단계를 다룹니다.

Download Python source code: tvmc_command_line_driver.py

https://tvm.apache.org/docs/_downloads/233ceda3a682ae5df93b4ce0bcfbf870/tvmc_command_line_driver.py

Download Jupyter notebook: tvmc_command_line_driver.ipynb

https://tvm.apache.org/docs/_downloads/efe0b02e219b28e0bd85fbdda35ba8ac/tvmc_command_line_driver.ipynb

1.6 TVMC Python 사용 시작하기: TVM을 위한 고급 API

(container) {

mkdir myscripts && cd myscripts

wget https://github.com/onnx/models/raw/b9a54e89508f101a1611cd64f4ef56b9cb62c7cf/vision/classification/resnet/model/resnet50-v2-7.onnx

mv resnet50-v2-7.onnx my_model.onnx

touch tvmcpythonintro.py

filename: tvmcpythonintro.py

```
from tvm.driver import tvmc


# Step 1: Load

model = tvmc.load('my_model.onnx')

# Step 1: Load + shape_dict

#model = tvmc.load('my_model.onnx', shape_dict={'input1' : [1, 2, 3, 4], 'input2' : [1, 2, 3, 4]})


# target: cuda (NVIDIA GPU), llvm (CPU), llvm -mcpu=cascadelake (Intel CPU)

package = tvmc.compile(model, target="llvm") # Step 2: Compile


# device: CPU, CUDA, CL, Metal, and Vulkan

result = tvmc.run(package, device="cpu") # Step 3: Run
```

} (container)

 # Step 0: Imports

```
from tvm.driver import tvmc
```

 # Step 1: Load a model

이 단계에서는 지원되는 프레임워크의 머신러닝 모델을 Relay라는 TVM의 고급 그래프 표현 언어로 변환합니다. 이것은 tvm의 모든 모델에 대해 통일된 시작점을 갖기 위한 것입니다. 현재 지원하는 프레임워크는 Keras, ONNX, Tensorflow, TFLite 그리고 PyTorch입니다.

```
model = tvmc.load('my_model.onnx') #Step 1: Load
```

릴레이를 보려면 model.summary()를 실행할 수 있습니다.

모든 프레임워크는 shape_dict 인수로 입력 shapes를 덮어쓰는 것을 지원합니다. 대부분의 프레임워크에서 이는 선택 사항이지만 Pytorch의 경우 TVM이 자동으로 검색할 수 없기 때문에 필요합니다.

```
# Step 1: Load + shape_dict
#model = tvmc.load('my_model.onnx', shape_dict={'input1' : [1, 2, 3, 4], 'input2' : [1, 2, 3, 4]})
```

모델의 input/shape_dict 를 확인하는 제안된 방법은 **Netron**(https://netron.app/)을 사용하는 것입니다. 모델을 연 후 첫 번째 노드를 클릭하여 입력 섹션에서 name과 shape를 확인합니다.

 # Step 2: Compile

이제 모델이 Relay에 있으므로 다음 단계는 실행할 원하는 하드웨어로 컴파일하는 것입니다. 이 하드웨어를 대상(a target)이라고 합니다. 이 컴파일 프로세스는 Relay의 모델을 대상 컴퓨터가 이해할 수 있는 하위 수준 언어로 변환합니다.

모델을 컴파일하려면 **tvm.target** 문자열이 필요합니다. tvm.targets 및 해당 옵션에 대한 자세한 내용은 설명서를 참조하세요. (https://tvm.apache.org/docs/api/python/target.html)

1. cuda (NVIDIA GPU)
2. llvm (CPU)
3. llvm -mcpu=cascadelake (Intel CPU)

```
package = tvmc.compile(model, target="llvm") #Step 2: Compile
```

컴파일 단계에서는 패키지를 반환합니다.

 # Step 3: Run

이제 컴파일된 패키지를 하드웨어 대상에서 실행할 수 있습니다. 장치 입력 옵션은 CPU, CUDA, CL, Metal 그리고 Vulkan입니다.

```
# device: CPU, Cuda, CL, Metal, and Vulkan
result = tvmc.run(package, device="cpu") #Step 3: Run
```

Tune [Optional & Recommended]

튜닝을 통해 실행 속도를 더욱 향상시킬 수 있습니다. 이 선택적 단계에서는 기계 학습을 사용하여 모델(함수) 내의 각 작업을 살펴보고 더 빠르게 실행할 수 있는 방법을 찾으려고 합니다. 우리는 비용 모델과 가능한 Schedule을 벤치마킹하여 이를 수행합니다.

대상은 compile과 동일합니다.

```
tvmc.tune(model, target="llvm") #Step 1.5: Optional Tune
```

터미널 출력은 다음과 같아야 합니다.

[Task 1/13] Current/Best: 82.00/ 106.29 GFLOPS | Progress: (48/769) | 18.56 s

[Task 1/13] Current/Best: 54.47/ 113.50 GFLOPS | Progress: (240/769) | 85.36 s

.....

무시할 수 있는 UserWarnings가 있을 수 있습니다. 이렇게 하면 최종 결과가 더 빨라지지만 튜닝하는 데 몇 시간이 걸릴 수 있습니다.

아래의 '튜닝 결과 저장' 섹션을 참조하세요. 결과를 적용하려면 튜닝 결과를 컴파일에 전달해야 합니다.

```
#tvmc.compile(model, target="llvm", tuning_records = "records.log") #Step 2: Compile
```

저장하고 터미널에서 프로세스를 시작합니다.

```
# python my_tvmc_script.py
```

참고: 팬이 매우 활발해질 수 있습니다.

예제 결과:

Time elapsed for training: 18.99 s

Execution time summary:

mean (ms)	max (ms)	min (ms)	std (ms)
25.24	26.12	24.89	0.38

Output Names:

['output_0']

추가 TVMC 기능

모델 저장

나중을 위해 더 빠르게 하려면 모델을 로드한 후(1단계) Relay 버전을 저장합니다. 그러면 나중에 포함된 구문에서 사용할 수 있도록 저장한 위치에 모델이 나타납니다.

```
model = tvmc.load('my_model.onnx') #Step 1: Load
model.save(desired_model_path)
```

패키지 저장

모델을 컴파일한 후(2단계) 패키지도 저장할 수 있습니다.

```
tvmc.compile(model, target="llvm", package_path="whatever") #Step 2: Compile

new_package = tvmc.TVMCPackage(package_path="whatever")
result = tvmc.run(new_package, device="cpu") #Step 3: Run
```

자동 스케줄러 사용

차세대 tvm을 사용하여 잠재적으로 더 빠른 실행 속도 결과를 얻을 수 있습니다. Schedule의 검색 공간은 손으로 써야 했던 이전과 달리 자동으로 생성됩니다.

자세히 알아보기:

1: Introducing TVM Auto-scheduler (a.k.a. Ansor)

https://tvm.apache.org/2021/03/03/intro-auto-scheduler

2: Ansor: Generating High-Performance Tensor Programs for Deep Learning
https://arxiv.org/abs/2006.06762

```
tvmc.tune(model, target="llvm", enable_autoscheduler = True)
```

튜닝 결과 저장

튜닝 결과는 나중에 다시 사용할 수 있도록 파일에 저장할 수 있습니다.

Method 1:

```
log_file = "hello.json"

# 튜닝 실행
tvmc.tune(model, target="llvm", tuning_records=log_file)
...
# 나중에 튜닝을 실행하고 튜닝 결과를 재사용합니다.
tvmc.tune(model, target="llvm", prior_records=log_file)
```

Method 2:

```
# 튜닝 실행
tuning_records = tvmc.tune(model, target="llvm")
...
# 나중에 튜닝을 실행하고 튜닝 결과를 재사용합니다.
tvmc.tune(model, target="llvm", prior_records=tuning_records)
```

더 복잡한 모델 튜닝:

T의 출력이 .....T.T..T..T..T.T.T.T.T.T. 처럼 보이는 것을 발견하면 검색 시간 프레임을 늘립니다.

```
tvmc.tune(model,trials=10000,timeout=10,)
```

원격 장치에 대한 모델 컴파일:

RPC(Remote Procedural Call)는 로컬 컴퓨터에 없는 하드웨어에 대해 컴파일하려는 경우에 유용합니다. tvmc 메서드가 이를 지원합니다. RPC 서버를 설정하려면 이 문서의 '디바이스에서 RPC 서버 설정' 섹션을 참조하세요.

https://tvm.apache.org/docs/tutorials/get_started/cross_compilation_and_rpc.html

TVMC 스크립트 내에 다음을 포함하고 그에 따라 조정합니다.

```
tvmc.tune(
        model,
        target=target, # Compilation target as string // Device to compile for
        target_host=target_host, # Host processor
        hostname=host_ip_address, # The IP address of an RPC tracker, used when benchmarking remotely.
        port=port_number, # The port of the RPC tracker to connect to. Defaults to 9090.
        rpc_key=your_key, # The RPC tracker key of the target device. Required when rpc_tracker is provided
)
```

Download Python source code: tvmc_python.py

https://tvm.apache.org/docs/_downloads/10724e9ad9c29faa223c1d5eab6dbef9/tvmc_python.py

Download Jupyter notebook: tvmc_python.ipynb

https://tvm.apache.org/docs/_downloads/8d55b8f991fb704002f768367ce2d1a2/tvmc_python.ipynb

1.7 Python 인터페이스(AutoTVM)를 사용하여 모델 컴파일 및 최적화

TVMC 튜토리얼(https://tvm.apache.org/docs/tutorial/tvmc_command_line_driver)에서는 TVM, TVMC의 command-line 인터페이스를 사용하여 사전 훈련된 비전 모델인 ResNet-50 v2를 컴파일, 실행 및 튜닝하는 방법을 다루었습니다. TVM은 단순한 command-line 도구에 가깝지만 기계 학습 모델 작업에 엄청난 유연성을 제공하는 다양한 언어에 사용할 수 있는 API가 포함된 최적화 프레임워크입니다.

이 튜토리얼에서는 TVMC에서 했던 것과 동일한 내용을 다루지만 Python API를 사용하여 수행하는 방법을 보여줍니다. 이 섹션을 완료하면 TVM용 Python API를 사용하여 다음 작업을 수행합니다.

- TVM 런타임을 위해 사전 학습된 ResNet-50 v2 모델을 컴파일합니다.
- 컴파일된 모델을 통해 실제 이미지를 실행하고 출력 및 모델 성능을 해석합니다.
- TVM을 사용하여 CPU에서 모델링하는 모델을 튜닝합니다.
- TVM에서 수집한 튜닝 데이터를 사용하여 최적화된 모델을 다시 컴파일합니다.
- 최적화된 모델을 통해 이미지를 실행하고 출력과 모델 성능을 비교합니다.

이 섹션의 목표는 TVM의 기능과 Python API를 통해 이를 사용하는 방법에 대한 개요를 제공하는 것입니다.

TVM은 딥러닝 컴파일러 프레임워크로, 딥러닝 모델 및 연산자 작업에 사용할 수 있는 다양한 모듈이 있습니다. 이 튜토리얼에서는 Python API를 사용하여 모델을 로드, 컴파일 및 최적화하는 방법을 살펴보겠습니다.

먼저 모델 로드 및 변환을 위한 **onnx**, 테스트 데이터 다운로드를 위한 helper 유틸리티, 이미지 데이터 작업을 위한 Python 이미지 라이브러리, 이미지 데이터의 사전 및 후처리를 위한 **numpy**, TVM Relay 프레임워크 및 TVM Graph Executor를 비롯한 여러 종속성을 import 합니다.

```
import onnx
from tvm.contrib.download import download_testdata
from PIL import Image
import numpy as np
import tvm.relay as relay
import tvm
from tvm.contrib import graph_executor
```

ONNX 모델 다운로드 및 로드

이 튜토리얼에서는 ResNet-50 v2를 사용합니다. ResNet-50은 50개 계층으로 구성된 컨벌루션 신경망 (Convolutional Neural Network, CNN)으로 영상을 분류하도록 설계되었습니다. 우리가 사용할 모델은 1,000개의 서로 다른 분류가 있는 100만 개 이상의 이미지에 대해 사전 훈련되었습니다. 네트워크의 입력 영상 크기는 224x224입니다. ResNet-50 모델이 어떻게 구성되어 있는지 자세히 알아보려면 무료로 제공되는 ML 모델 뷰어인 **Netron** (https://netron.app/)을 다운로드하는 것을 추천합니다.

TVM은 사전 훈련된 모델을 다운로드할 수 있는 helper 라이브러리를 제공합니다. 모듈을 통해 모델 URL, 파일 이름 및 모델 유형을 제공하면 TVM은 모델을 다운로드하여 디스크에 저장합니다. ONNX 모델 인스턴스의 경우 ONNX 런타임을 사용하여 메모리에 로드할 수 있습니다.

다른 모델 형식으로 작업하기

TVM은 널리 사용되는 많은 모델 형식을 지원합니다. 목록은 TVM 문서의 **Compile Deep Learning Models** 섹션에서 찾을 수 있습니다.

https://tvm.apache.org/docs/how_to/compile_models/index.html#tutorial-frontend

```
model_url = (
    "https://github.com/onnx/models/raw/main/"
    "vision/classification/resnet/model/"
    "resnet50-v2-7.onnx"
```

```
)

model_path = download_testdata(model_url, "resnet50-v2-7.onnx", module="onnx")
onnx_model = onnx.load(model_path)

# Seed numpy's RNG to get consistent results
np.random.seed(0)
```

테스트 이미지 다운로드, 전처리 및 로드하기

각 모델은 예상되는 텐서 shapes, 형식 및 데이터 유형과 관련하여 특별합니다. 이러한 이유로 대부분의 모델에는 입력이 유효한지 확인하고 출력을 해석하기 위해 몇 가지 사전 및 후처리가 필요합니다. TVMC는 입력과 출력 데이터 모두에 NumPy의 **.npz** 형식을 채택했습니다. 이 튜토리얼의 입력으로 고양이 이미지를 사용하지만 원하는 이미지로 자유롭게 대체할 수 있습니다.

이미지 데이터를 다운로드한 다음 numpy 배열로 변환하여 모델에 대한 입력값으로 사용합니다.

```
img_url = "https://s3.amazonaws.com/model-server/inputs/kitten.jpg"
img_path = download_testdata(img_url, "imagenet_cat.png", module="data")

# Resize it to 224x224
resized_image = Image.open(img_path).resize((224, 224))
img_data = np.asarray(resized_image).astype("float32")

# Our input image is in HWC layout while ONNX expects CHW input, so convert the array
img_data = np.transpose(img_data, (2, 0, 1))

# Normalize according to the ImageNet input specification
imagenet_mean = np.array([0.485, 0.456, 0.406]).reshape((3, 1, 1))
```

```
imagenet_stddev = np.array([0.229, 0.224, 0.225]).reshape((3, 1, 1))
norm_img_data = (img_data / 255 - imagenet_mean) / imagenet_stddev

# Add the batch dimension, as we are expecting 4-dimensional input: NCHW.
img_data = np.expand_dims(norm_img_data, axis=0)
```

Relay로 모델 컴파일

다음 단계는 ResNet 모델을 컴파일하는 것입니다. 먼저 **from_onnx** 임포터를 사용하여 릴레이할 모델을 임포트합니다. 그런 다음 표준 최적화를 사용하여 모델을 TVM 라이브러리로 빌드합니다. 마지막으로 라이브러리에서 TVM 그래프 런타임 모듈을 만듭니다.

```
target = "llvm"
```

올바른 대상을 지정하면 대상에서 사용할 수 있는 하드웨어 기능을 활용할 수 있으므로 컴파일된 모듈의 성능에 큰 영향을 미칠 수 있습니다. 자세한 내용은 **Auto-tuning a convolutional network for x86 CPU** 을 참조하세요.

https://tvm.apache.org/docs/how_to/tune_with_autotvm/tune_relay_x86.html#tune-relay-x86

선택적 기능과 함께 실행 중인 CPU를 식별하고 대상을 적절하게 설정하는 것이 좋습니다. 예를 들어 일부 프로세서의 경우 **target = "llvm -mcpu=skylake"** 또는 AVX-512 벡터 명령 집합이 있는 프로세서의 경우 **target = "llvm -mcpu=skylake-avx512"**입니다.

```
# The input name may vary across model types. You can use a tool
# like Netron to check input names
input_name = "data"
shape_dict = {input_name: img_data.shape}

mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)

with tvm.transform.PassContext(opt_level=3):
    lib = relay.build(mod, target=target, params=params)
```

```
dev = tvm.device(str(target), 0)
module = graph_executor.GraphModule(lib["default"](dev))
```

TVM 런타임에서 실행

이제 모델을 컴파일했으므로 TVM 런타임을 사용하여 예측을 수행할 수 있습니다. TVM을 사용하여 모델을 실행하고 예측을 수행하려면 다음 두 가지가 필요합니다.

- 방금 제작한 컴파일된 모델
- 예측을 수행할 모델에 대한 유효한 입력

```
dtype = "float32"
module.set_input(input_name, img_data)
module.run()
output_shape = (1, 1000)
tvm_output = module.get_output(0, tvm.nd.empty(output_shape)).numpy()
```

기본 성능 데이터 수집

이 최적화되지 않은 모델과 관련된 몇 가지 기본 성능 데이터를 수집하여 나중에 튜닝된 모델과 비교하려고 합니다. CPU 노이즈를 설명하기 위해 여러 배치에서 여러 번 반복하여 계산을 실행한 다음 평균, 중앙값 및 표준 편차에 대한 몇 가지 기본 통계를 수집합니다.

```
import timeit

timing_number = 10
timing_repeat = 10
```

```
unoptimized = (
    np.array(timeit.Timer(lambda: module.run()).repeat(
                repeat=timing_repeat, number=timing_number))
    * 1000
    / timing_number
)
unoptimized = {
    "mean": np.mean(unoptimized),
    "median": np.median(unoptimized),
    "std": np.std(unoptimized),
}

print(unoptimized)
```

Out:

```
{'mean': 468.8155529099992, 'median': 468.2440652000025, 'std': 2.1034826796360457}
```

출력 후처리

앞서 언급했듯이 각 모델에는 출력 텐서를 제공하는 고유한 방법이 있습니다.

이 경우 모델에 제공된 조회 테이블을 사용하여 ResNet-50 v2의 출력을 사람이 읽을 수 있는 형식으로 렌더링하기 위해 몇 가지 후처리를 실행해야 합니다.

```
from scipy.special import softmax

# Download a list of labels
labels_url = "https://s3.amazonaws.com/onnx-model-zoo/synset.txt"
labels_path = download_testdata(labels_url, "synset.txt", module="data")

with open(labels_path, "r") as f:
    labels = [l.rstrip() for l in f]
```

```
# Open the output and read the output tensor
scores = softmax(tvm_output)
scores = np.squeeze(scores)
ranks = np.argsort(scores)[::-1]
for rank in ranks[0:5]:
    print("class='%s' with probability=%f" % (labels[rank], scores[rank]))
```

Out:

```
class='n02123045 tabby, tabby cat' with probability=0.621103
class='n02123159 tiger cat' with probability=0.356379
class='n02124075 Egyptian cat' with probability=0.019712
class='n02129604 tiger, Panthera tigris' with probability=0.001215
class='n04040759 radiator' with probability=0.000262
```

그러면 다음과 같은 출력이 생성됩니다.

```
# class='n02123045 tabby, tabby cat' with probability=0.610553
# class='n02123159 tiger cat' with probability=0.367179
# class='n02124075 Egyptian cat' with probability=0.019365
# class='n02129604 tiger, Panthera tigris' with probability=0.001273
# class='n04040759 radiator' with probability=0.000261
```

모델 튜닝

이전 모델은 TVM 런타임에서 작동하도록 컴파일되었지만 플랫폼별 최적화는 포함되지 않았습니다. 이 섹션에서는 TVM을 사용하여 작업 플랫폼을 대상으로 최적화된 모델을 구축하는 방법을 보여줍니다.

어떤 경우에는 컴파일된 모듈을 사용하여 추론을 실행할 때 예상한 성능을 얻지 못할 수도 있습니다. 이와 같은 경우 자동 튜너를 사용하여 모델에 대한 더 나은 구성을 찾고 성능을 향상시킬

수 있습니다.

TVM 튜닝은 주어진 대상에서 더 빠르게 실행되도록 모델을 최적화하는 프로세스를 나타냅니다. 이는 모델의 정확도에 영향을 주지 않고 런타임 성능에만 영향을 미친다는 점에서 학습 또는 미세 조정 (fine-tuning)과 다릅니다.

튜닝 프로세스의 일환으로 TVM은 다양한 연산자 구현 변형을 실행하여 어떤 것이 가장 성능이 좋은지 확인합니다. 이러한 실행의 결과는 튜닝 레코드 파일에 저장됩니다.

가장 간단한 형태의 튜닝을 수행하려면 다음 세 가지를 제공해야 합니다.

- 이 모델을 실행하려는 장치의 대상 사양
- 튜닝 레코드가 저장될 출력 파일의 경로
- 튜닝할 모델의 경로입니다.

```
import tvm.auto_scheduler as auto_scheduler
from tvm.autotvm.tuner import XGBTuner
from tvm import autotvm
```

Runner에 대한 몇 가지 기본 매개 변수를 설정합니다. **Runner**는 특정 매개 변수 집합으로 생성된 컴파일된 코드를 가져와서 성능을 측정합니다.

number는 테스트할 다양한 구성의 수를 지정하고, **repeat**는 각 구성에 대해 수행할 측정 횟수를 지정합니다.

min_repeat_ms는 구성 테스트를 실행해야 하는 기간을 지정하는 값입니다. 반복 횟수가 이 시간 이하로 떨어지면 증가합니다. 이 옵션은 GPU에서 정확한 튜닝을 위해 필요하며 CPU 튜닝에는 필요하지 않습니다. 이 값을 0으로 설정하면 비활성화됩니다.

timeout은 테스트된 각 구성에 대해 학습 코드를 실행하는 시간에 대한 상한을 설정합니다.

```
number = 10
repeat = 1
min_repeat_ms = 0   # since we're tuning on a CPU, can be set to 0
timeout = 10   # in seconds
```

```
# create a TVM runner
runner = autotvm.LocalRunner(
    number=number,
    repeat=repeat,
    timeout=timeout,
    min_repeat_ms=min_repeat_ms,
    enable_cpu_cache_flush=True,
)
```

튜닝 옵션을 유지하기 위한 간단한 구조를 만듭니다. 검색을 위해 XGBoost 알고리즘을 사용합니다. 프로덕션 작업의 경우 시도 횟수를 여기에 사용된 값 20보다 크게 설정할 수 있습니다.

CPU의 경우 1500, GPU 3000-4000을 권장합니다. 필요한 시도 횟수는 특정 모델 및 프로세서에 따라 달라질 수 있으므로 튜닝 시간과 모델 최적화 간의 최상의 균형을 찾기 위해 다양한 값에서 성능을 평가하는 데 시간을 할애하는 것이 좋습니다.

튜닝을 실행하는 데 시간이 많이 걸리기 때문에 시도 횟수를 10으로 설정하지만 이렇게 작은 값은 권장하지 않습니다. **early_stopping** 파라미터는 검색을 조기에 중지하는 조건을 적용하기 전에 실행할 최소 추적 수입니다.

측정 옵션은 평가 코드가 빌드되는 위치와 실행될 위치를 나타냅니다. 이 경우 방금 만든 **LocalRunner**와 **LocalBuilder**를 사용합니다. **tuning_records** 옵션은 튜닝 데이터를 쓸 파일을 지정합니다.

```
tuning_option = {
    "tuner": "xgb",
    "trials": 20,
    "early_stopping": 100,
    "measure_option": autotvm.measure_option(
        builder=autotvm.LocalBuilder(build_func="default"), runner=runner
    ),
    "tuning_records": "resnet-50-v2-autotuning.json",
}
```

튜닝 검색 알고리즘 정의

기본적으로 이 검색은 **XGBoost Grid** 알고리즘을 사용합니다. 모델 복잡성과 사용 가능한 시간에 따라 다른 알고리즘을 선택할 수 있습니다.

튜닝 파라미터 설정

이 예제에서는 시간 관계상 시행 횟수와 조기 중지를 20과 100으로 설정했습니다. 이러한 값을 더 높게 설정하면 성능이 더 향상될 수 있지만 이렇게 하면 튜닝에 소요되는 시간이 희생됩니다. 수렴에 필요한 시행 횟수는 모델 및 대상 플랫폼의 세부 사항에 따라 달라집니다.

```
# begin by extracting the tasks from the onnx model
tasks = autotvm.task.extract_from_program(mod["main"], target=target, params=params)

# Tune the extracted tasks sequentially.
for i, task in enumerate(tasks):
    prefix = "[Task %2d/%2d] " % (i + 1, len(tasks))

    # choose tuner
    tuner = "xgb"

    # create tuner
    if tuner == "xgb":
        tuner_obj = XGBTuner(task, loss_type="reg")
    elif tuner == "xgb_knob":
        tuner_obj = XGBTuner(task, loss_type="reg", feature_type="knob")
    elif tuner == "xgb_itervar":
        tuner_obj = XGBTuner(task, loss_type="reg", feature_type="itervar")
    elif tuner == "xgb_curve":
        tuner_obj = XGBTuner(task, loss_type="reg", feature_type="curve")
    elif tuner == "xgb_rank":
        tuner_obj = XGBTuner(task, loss_type="rank")
    elif tuner == "xgb_rank_knob":
        tuner_obj = XGBTuner(task, loss_type="rank", feature_type="knob")
    elif tuner == "xgb_rank_itervar":
        tuner_obj = XGBTuner(task, loss_type="rank", feature_type="itervar")
```

```
    elif tuner == "xgb_rank_curve":
        tuner_obj = XGBTuner(task, loss_type="rank", feature_type="curve")
    elif tuner == "xgb_rank_binary":
        tuner_obj = XGBTuner(task, loss_type="rank-binary")
    elif tuner == "xgb_rank_binary_knob":
        tuner_obj = XGBTuner(task, loss_type="rank-binary", feature_type="knob")
    elif tuner == "xgb_rank_binary_itervar":
        tuner_obj = XGBTuner(task, loss_type="rank-binary", feature_type="itervar")
    elif tuner == "xgb_rank_binary_curve":
        tuner_obj = XGBTuner(task, loss_type="rank-binary", feature_type="curve")
    elif tuner == "ga":
        tuner_obj = GATuner(task, pop_size=50)
    elif tuner == "random":
        tuner_obj = RandomTuner(task)
    elif tuner == "gridsearch":
        tuner_obj = GridSearchTuner(task)
    else:
        raise ValueError("Invalid tuner: " + tuner)

    tuner_obj.tune(
        n_trial=min(tuning_option["trials"], len(task.config_space)),
        early_stopping=tuning_option["early_stopping"],
        measure_option=tuning_option["measure_option"],
        callbacks=[
            autotvm.callback.progress_bar(tuning_option["trials"], prefix=prefix),
            autotvm.callback.log_to_file(tuning_option["tuning_records"]),
        ],
    )
```

Out:

```
[Task  1/25]  Current/Best:    0.00/    0.00 GFLOPS | Progress: (0/20) | 0.00 s
[Task  1/25]  Current/Best:    6.77/   14.96 GFLOPS | Progress: (4/20) | 9.89 s
[Task  1/25]  Current/Best:   14.58/   18.60 GFLOPS | Progress: (8/20) | 13.14 s
[Task  1/25]  Current/Best:    5.77/   18.60 GFLOPS | Progress: (12/20) | 16.97 s
[Task  1/25]  Current/Best:   14.90/   23.70 GFLOPS | Progress: (16/20) | 20.35 s
[Task  1/25]  Current/Best:    7.56/   23.70 GFLOPS | Progress: (20/20) | 23.93 s Done.
```

```
...
[Task 25/25]  Current/Best:      8.14/     8.14 GFLOPS | Progress: (8/20) | 17.20 s
[Task 25/25]  Current/Best:      5.49/     8.14 GFLOPS | Progress: (12/20) | 18.57 s
[Task 25/25]  Current/Best:      7.60/     8.14 GFLOPS | Progress: (16/20) | 20.32 s
[Task 25/25]  Current/Best:      2.74/     8.51 GFLOPS | Progress: (20/20) | 26.53 s Done.
```

이 튜닝 프로세스의 출력은 다음과 같습니다.

```
# [Task   1/24]  Current/Best:     10.71/    21.08 GFLOPS | Progress: (60/1000) | 111.77 s Done.
# [Task   1/24]  Current/Best:      9.32/    24.18 GFLOPS | Progress: (192/1000) | 365.02 s Done.
# [Task   2/24]  Current/Best:     22.39/ 177.59 GFLOPS | Progress: (960/1000) | 976.17 s Done.
# [Task   3/24]  Current/Best:     32.03/ 153.34 GFLOPS | Progress: (800/1000) | 776.84 s Done.
# [Task   4/24]  Current/Best:     11.96/ 156.49 GFLOPS | Progress: (960/1000) | 632.26 s Done.
# [Task   5/24]  Current/Best:     23.75/ 130.78 GFLOPS | Progress: (800/1000) | 739.29 s Done.
...
# [Task 20/24]  Current/Best:     30.47/ 205.92 GFLOPS | Progress: (980/1000) | 471.00 s Done.
# [Task 21/24]  Current/Best:     46.91/ 227.99 GFLOPS | Progress: (308/1000) | 219.18 s Done.
# [Task 22/24]  Current/Best:     13.33/ 207.66 GFLOPS | Progress: (1000/1000) | 761.74 s Done.
# [Task 23/24]  Current/Best:     53.29/ 192.98 GFLOPS | Progress: (1000/1000) | 799.90 s Done.
# [Task 24/24]  Current/Best:     25.03/ 146.14 GFLOPS | Progress: (1000/1000) | 1112.55 s Done.
```

데이터 튜닝을 사용하여 최적화된 모델 컴파일

위의 튜닝 프로세스의 출력으로 **resnet-50-v2-autotuning.json**에 저장된 튜닝 레코드를 얻었습니다. 컴파일러는 결과를 사용하여 지정된 대상의 모델에 대한 고성능 코드를 생성합니다.

이제 모델에 대한 튜닝 데이터가 수집되었으므로 최적화된 연산자를 사용하여 모델을 다시 컴파일하여 계산 속도를 높일 수 있습니다.

```
with autotvm.apply_history_best(tuning_option["tuning_records"]):
    with tvm.transform.PassContext(opt_level=3, config={}):
        lib = relay.build(mod, target=target, params=params)
```

```
dev = tvm.device(str(target), 0)
module = graph_executor.GraphModule(lib["default"](dev))
```

Out:

```
Done.
```

최적화된 모델이 실행되고 동일한 결과가 생성되는지 확인합니다.

```
dtype = "float32"
module.set_input(input_name, img_data)
module.run()
output_shape = (1, 1000)
tvm_output = module.get_output(0, tvm.nd.empty(output_shape)).numpy()

scores = softmax(tvm_output)
scores = np.squeeze(scores)
ranks = np.argsort(scores)[::-1]
for rank in ranks[0:5]:
    print("class='%s' with probability=%f" % (labels[rank], scores[rank]))
```

Out:

```
class='n02123045 tabby, tabby cat' with probability=0.621104
class='n02123159 tiger cat' with probability=0.356378
class='n02124075 Egyptian cat' with probability=0.019712
class='n02129604 tiger, Panthera tigris' with probability=0.001215
class='n04040759 radiator' with probability=0.000262
```

예측이 동일한지 확인:

```
# class='n02123045 tabby, tabby cat' with probability=0.610550
# class='n02123159 tiger cat' with probability=0.367181
# class='n02124075 Egyptian cat' with probability=0.019365
# class='n02129604 tiger, Panthera tigris' with probability=0.001273
# class='n04040759 radiator' with probability=0.000261
```

튜닝된 모델과 튜닝되지 않은 모델 비교

이 최적화된 모델과 관련된 몇 가지 기본 성능 데이터를 수집하여 최적화되지 않은 모델과 비교하려고 합니다. 기본 하드웨어, 반복 횟수 및 기타 요인에 따라 최적화된 모델과 최적화되지 않은 모델을 비교할 때 성능이 향상되는 것을 볼 수 있습니다.

```
import timeit

timing_number = 10
timing_repeat = 10
optimized = (
      np.array(timeit.Timer(lambda: module.run()).repeat(
                  repeat=timing_repeat, number=timing_number))
      * 1000
      / timing_number
)
optimized = {"mean": np.mean(optimized), "median": np.median(optimized),
                   "std": np.std(optimized)}



print("optimized: %s" % (optimized))
print("unoptimized: %s" % (unoptimized))
```

Out:

```
optimized: {'mean': 421.6479147799964, 'median': 421.39902544998904, 'std': 2.945366295636001}
unoptimized: {'mean': 468.8155529099992, 'median': 468.2440652000025,
                    'std': 2.1034826796360457}
```

최종 비고

이 튜토리얼에서는 TVM Python API를 사용하여 모델을 컴파일, 실행 및 튜닝하는 방법에 대한 간단한 예를 제공했습니다. 또한 입력과 출력의 사전 및 후처리의 필요성에 대해서도 논의했습니다. 튜닝 프로세스 후 최적화되지 않은 모델과 최적화 모델의 성능을 비교하는 방법을 시연했습니다.

여기서는 로컬에서 ResNet-50 v2를 사용하는 간단한 예제를 제시했습니다. 그러나 TVM은 교차 컴파일, 원격 실행 및 프로파일링/벤치마킹을 포함한 더 많은 기능을 지원합니다.

스크립트의 총 실행 시간 : (13 분 9.373 초)

Download Python source code: autotvm_relay_x86.py

https://tvm.apache.org/docs/_downloads/57a45d9bef1af358191e7d50043e652c/autotvm_relay_x86.py

Download Jupyter notebook: autotvm_relay_x86.ipynb

https://tvm.apache.org/docs/_downloads/2f91b1346a0ba21b800081aa15fdaac2/autotvm_relay_x86.ipynb

1.8 Tensor 표현식을 사용한 연산자 작업

이 튜토리얼에서는 TVM이 텐서 표현식(TE)과 함께 작동하여 텐서 계산을 정의하고 루프 최적화를 적용하는 방법에 주의를 기울일 것입니다. TE는 순수 함수형 언어로 텐서 계산을 표현합니다 (즉, 각 표현식에는 부작용이 없음).

TVM 전체의 컨텍스트에서 볼 때 Relay는 계산을 연산자 집합으로 설명하며, 이러한 각 연산자는 각 TE 식이 입력 텐서를 사용하여 출력 텐서를 생성하는 TE 식으로 나타낼 수 있습니다.

이 튜토리얼은 TVM의 Tensor Expression 언어에 대한 내용입니다. TVM은 효율적인 커널 구성을 위해 도메인별 텐서 표현식을 사용합니다. 텐서 표현식 언어를 사용하는 두 가지 예를 통해 기본 워크플로를 보여줍니다.

첫 번째 예제에서는 TE와 벡터 덧셈을 사용한 스케줄링을 소개합니다. 두 번째는 TE를 사용한 행렬 곱셈의 단계별 최적화를 통해 이러한 개념을 확장합니다. 이 행렬 곱셈 예제는 향후 TVM의 고급 기능을 다루는 튜토리얼의 비교 기준이 될 것입니다.

예제 1: CPU용 TE에서 벡터 덧셈 쓰기 및 스케줄링

벡터 덧셈을 위한 TE를 구현한 다음 CPU를 대상으로 하는 Schedule을 구현하는 Python의 예를 살펴보겠습니다. 먼저 TVM 환경을 초기화합니다.

```
import tvm
import tvm.testing
from tvm import te
import numpy as np
```

대상 CPU를 식별하고 지정할 수 있으면 더 나은 성능을 얻을 수 있습니다. LLVM을 사용하는 경우 **llc --version** 명령에서 CPU 유형을 가져올 수 있으며, **/proc/cpuinfo**에서 프로세서가 지원할

수 있는 추가 확장을 확인할 수 있습니다. 예를 들어 AVX-512 명령이 있는 CPU에 **llvm -mcpu=skylake-avx512**를 사용할 수 있습니다.

```
tgt = tvm.target.Target(target="llvm", host="llvm")
```

벡터 계산 설명

벡터 덧셈 계산에 대해 설명합니다. TVM은 텐서 의미 체계를 채택하며 각 중간 결과는 다차원 배열로 표시됩니다. 사용자는 텐서를 생성하는 계산 규칙을 설명해야 합니다. 먼저 shape를 나타내기 위해 기호 변수 **n**을 정의합니다.

그런 다음 주어진 shape **(n,)**으로 두 개의 placeholder Tensor **A**와 **B**를 정의합니다. 그런 다음 **compute** 작업을 사용하여 결과 텐서 **C**를 설명합니다. **compute**는 지정된 텐서 shape을 따르는 출력과 람다 함수로 정의된 텐서의 각 위치에서 수행할 계산을 정의합니다.

n은 변수이지만 **A**, **B** 및 **C** 텐서 간에 일관된 shape을 정의합니다. 이 단계에서는 계산이 어떻게 수행되어야 하는지만 선언하기 때문에 실제 계산이 발생하지 않는다는 것을 기억하세요.

```
n = te.var("n")
A = te.placeholder((n,), name="A")
B = te.placeholder((n,), name="B")
C = te.compute(A.shape, lambda i: A[i] + B[i], name="C")
```

람다 함수

te.compute 메서드에 대한 두 번째 인수는 계산을 수행하는 함수입니다. 이 예제에서는 **람다(lambda)** 함수라고도 하는 익명 함수를 사용하여 계산을 정의하며, 이 경우 **A**와 **B**의 **i**번째 요소를 추가합니다.

계산에 대한 기본 스케줄 만들기

위의 줄은 계산 규칙을 설명하지만 다양한 장치에 맞게 다양한 방법으로 C를 계산할 수 있습니다. 여러 축이 있는 텐서의 경우 먼저 반복할 축을 선택하거나 계산을 여러 스레드로 분할할 수 있습니다.

TVM을 사용하려면 사용자가 계산을 수행하는 방법에 대한 설명인 Schedule을 제공해야 합니다. TE 내에서 작업을 예약하면 루프 순서를 변경하고, 여러 스레드에 걸쳐 계산을 분할하고, 데이터 블록을 함께 그룹화하는 등의 작업을 수행할 수 있습니다.

Schedule 이면의 중요한 개념은 계산이 수행되는 방식만 설명하므로 동일한 TE에 대해 다른 Schedule이 동일한 결과를 생성한다는 것입니다.

TVM을 사용하면 행 우선 순위 (row-major order)를 반복하여 C를 계산하는 단순한 Schedule을 만들 수 있습니다.

```
for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i];
}
```

```
s = te.create_schedule(C.op)
```

기본 스케줄 컴파일 및 평가

TE 표현식과 Schedule을 사용하여 대상 언어 및 아키텍처(이 경우 LLVM 및 CPU)에 대한 실행 가능한 코드를 생성할 수 있습니다. TVM에 Schedule, Schedule에 있는 TE 표현식 목록, 대상 및 호스트, 생성 중인 함수의 이름을 제공합니다. 출력의 결과는 Python에서 직접 호출할 수 있는 type-erased 함수입니다.

다음 줄에서는 **tvm.build**를 사용하여 함수를 만듭니다. build 함수는 Schedule, 함수의 원하는 서명(입력 및 출력 포함) 및 컴파일하려는 대상 언어를 사용합니다.

```
fadd = tvm.build(s, [A, B, C], tgt, name="myadd")
```

함수를 실행하고 출력을 numpy의 동일한 계산과 비교해 보겠습니다. 컴파일된 TVM 함수는 모든 언어에서 호출할 수 있는 간결한 C API를 노출합니다. 먼저 TVM이 Schedule을 컴파일할 수 있는 장치(이 예에서는 CPU)인 장치를 만듭니다.

이 경우 디바이스는 LLVM CPU 대상입니다. 그런 다음 장치에서 텐서를 초기화하고 사용자 지정 추가 작업을 수행할 수 있습니다. 계산이 올바른지 확인하기 위해 c 텐서의 출력 결과를 numpy가 수행한 것과 동일한 계산과 비교할 수 있습니다.

```
dev = tvm.device(tgt.kind.name, 0)

n = 1024
a = tvm.nd.array(np.random.uniform(size=n).astype(A.dtype), dev)
b = tvm.nd.array(np.random.uniform(size=n).astype(B.dtype), dev)
c = tvm.nd.array(np.zeros(n, dtype=C.dtype), dev)
fadd(a, b, c)
tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())
```

이 버전이 numpy와 비교하여 얼마나 빠른지 비교하려면 TVM 생성 코드의 프로필을 실행하는 helper 함수를 만드세요.

```
import timeit

np_repeat = 100
np_running_time = timeit.timeit(
      setup="import numpy₩n"
      "n = 32768₩n"
      'dtype = "float32"₩n'
      "a = numpy.random.rand(n, 1).astype(dtype)₩n"
      "b = numpy.random.rand(n, 1).astype(dtype)₩n",
      stmt="answer = a + b",
      number=np_repeat,
)
print("Numpy running time: %f" % (np_running_time / np_repeat))
```

```
def evaluate_addition(func, target, optimization, log):
    dev = tvm.device(target.kind.name, 0)
    n = 32768
    a = tvm.nd.array(np.random.uniform(size=n).astype(A.dtype), dev)
    b = tvm.nd.array(np.random.uniform(size=n).astype(B.dtype), dev)
    c = tvm.nd.array(np.zeros(n, dtype=C.dtype), dev)

    evaluator = func.time_evaluator(func.entry_name, dev, number=10)
    mean_time = evaluator(a, b, c).mean
    print("%s: %f" % (optimization, mean_time))

    log.append((optimization, mean_time))


log = [("numpy", np_running_time / np_repeat)]
evaluate_addition(fadd, tgt, "naive", log=log)
```

Out:

```
Numpy running time: 0.000009
naive: 0.000008
```

병렬 처리를 사용하도록 스케줄 업데이트

이제 TE의 기본 사항을 설명했으므로 Schedule이 수행하는 작업과 Schedule을 사용하여 다양한 아키텍처에 대한 텐서 표현식을 최적화하는 방법에 대해 자세히 살펴보겠습니다. Schedule은 다양한 방법으로 표현식을 변환하기 위해 표현식에 적용되는 일련의 단계입니다.

Schedule이 TE의 표현식에 적용되면 입력과 출력은 동일하게 유지되지만 컴파일될 때 표현식의 구현이 변경될 수 있습니다. 이 텐서 추가는 기본 Schedule에서 연속적으로 실행되지만 모든 프로세서 스레드에서 쉽게 병렬화할 수 있습니다. 병렬 Schedule 작업을 계산에 적용할 수 있습니다.

```
s[C].parallel(C.op.axis[0])
```

tvm.lower 명령은 해당 Schedule과 함께 TE의 IR(Intermediate Representation)을 생성합니다. 다른 Schedule 작업을 적용할 때 표현식을 낮추면 계산 순서에 미치는 영향을 볼 수 있습니다. **simple_mode=True** 플래그를 사용하여 읽을 수 있는 C 스타일 문을 반환합니다.

```
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.handle, B: T.handle, C: T.handle):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        n = T.int32()
        A_1 = T.match_buffer(A, (n,), strides=("stride",), buffer_type="auto")
        B_1 = T.match_buffer(B, (n,), strides=("stride",), buffer_type="auto")
        C_1 = T.match_buffer(C, (n,), strides=("stride",), buffer_type="auto")
        for i in T.parallel(n):
            C_2 = T.Buffer((C_1.strides[0] * n,), data=C_1.data, buffer_type="auto")
            A_2 = T.Buffer((A_1.strides[0] * n,), data=A_1.data, buffer_type="auto")
            B_2 = T.Buffer((B_1.strides[0] * n,), data=B_1.data, buffer_type="auto")
            C_2[i * C_1.strides[0]] = A_2[i * A_1.strides[0]] + B_2[i * B_1.strides[0]]
```

이제 TVM이 독립 스레드에서 이러한 블록을 실행할 수 있습니다. 병렬 작업이 적용된 이 새 Schedule을 컴파일하고 실행해 보겠습니다.

```
fadd_parallel = tvm.build(s, [A, B, C], tgt, name="myadd_parallel")
```

```
fadd_parallel(a, b, c)

tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())

evaluate_addition(fadd_parallel, tgt, "parallel", log=log)
```

Out:

```
parallel: 0.000008
```

벡터화를 사용하도록 스케줄 업데이트

최신 CPU에는 부동 소수점 값에 대해 SIMD 연산을 수행할 수 있는 기능도 있으며 이를 활용하기 위해 계산 표현식에 다른 Schedule을 적용할 수 있습니다. 이를 위해서는 여러 단계가 필요합니다: 먼저 분할 스케줄링 프리미티브를 사용하여 Schedule을 내부 루프와 외부 루프로 분할해야 합니다.

내부 루프는 벡터화를 사용하여 벡터화 스케줄링 프리미티브를 사용하는 SIMD 명령어를 사용할 수 있으며, 외부 루프는 병렬 스케줄링 프리미티브를 사용하여 병렬화할 수 있습니다. CPU의 스레드 수로 분할 계수를 선택합니다.

```
# Recreate the schedule, since we modified it with the parallel operation in
# the previous example
n = te.var("n")
A = te.placeholder((n,), name="A")
B = te.placeholder((n,), name="B")
C = te.compute(A.shape, lambda i: A[i] + B[i], name="C")

s = te.create_schedule(C.op)

# This factor should be chosen to match the number of threads appropriate for
# your CPU. This will vary depending on architecture, but a good rule is
# setting this factor to equal the number of available CPU cores.
```

```
factor = 4

outer, inner = s[C].split(C.op.axis[0], factor=factor)
s[C].parallel(outer)
s[C].vectorize(inner)

fadd_vector = tvm.build(s, [A, B, C], tgt, name="myadd_parallel")

evaluate_addition(fadd_vector, tgt, "vector", log=log)

print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
vector: 0.000039
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.handle, B: T.handle, C: T.handle):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        n = T.int32()
        A_1 = T.match_buffer(A, (n,), strides=("stride",), buffer_type="auto")
        B_1 = T.match_buffer(B, (n,), strides=("stride",), buffer_type="auto")
        C_1 = T.match_buffer(C, (n,), strides=("stride",), buffer_type="auto")
        for i_outer in T.parallel((n + 3) // 4):
            for i_inner_s in range(4):
                if T.likely(i_outer * 4 + i_inner_s < n):
                    C_2 = T.Buffer((C_1.strides[0] * n,), data=C_1.data, buffer_type="auto")
                    A_2 = T.Buffer((A_1.strides[0] * n,), data=A_1.data, buffer_type="auto")
                    B_2 = T.Buffer((B_1.strides[0] * n,), data=B_1.data, buffer_type="auto")
                    cse_var_1: T.int32 = i_outer * 4 + i_inner_s
                    C_2[cse_var_1 * C_1.strides[0]] =
                        A_2[cse_var_1 * A_1.strides[0]] + B_2[cse_var_1 * B_1.strides[0]]
```

다른 스케줄 비교

이제 다른 Schedule을 비교할 수 있습니다

```
baseline = log[0][1]
print("%s₩t%s₩t%s" % ("Operator".rjust(20), "Timing".rjust(20), "Performance".rjust(20)))
for result in log:
    print(
        "%s₩t%s₩t%s"
        % (result[0].rjust(20), str(result[1]).rjust(20), str(result[1] / baseline).rjust(20))
    )
```

Out:

Operator	Timing	Performance
numpy	8.686600012879354e-06	1.0
naive	7.9005e-06	0.9095042926215289
parallel	8.158400000000001e-06	0.9391936992498554
vector	3.91403e-05	4.505825057210863

코드 전문화

눈치채셨겠지만, **A**, **B**, **C**의 선언은 모두 동일한 shape 인수인 n을 사용합니다. TVM은 출력된 장치 코드에서 볼 수 있듯이 이를 활용하여 커널에 단일 shape 인수만 전달합니다. 이것은 전문화의 한 형태입니다.

호스트 측에서 TVM은 매개변수의 제약 조건을 확인하는 검사 코드를 자동으로 생성합니다. 따라서 shapes 다른 배열을 **fadd**에 전달하면 오류가 발생합니다. 우리는 더 많은 전문화를 할 수 있습니다. 예를 들어, 계산 선언에 **n = te.var("n")** 대신 **n = tvm.runtime.convert(1024)**를 쓸 수 있습니다. 생성된 함수는 길이가 1024인 벡터만 사용합니다.

벡터 덧셈 연산자를 정의, Schedule 및 컴파일한 다음 TVM 런타임에서 실행할 수 있었습니다. 연산자를 라이브러리로 저장한 다음 나중에 TVM 런타임을 사용하여 로드할 수 있습니다.

GPU에 대한 벡터 덧셈 타겟팅 (선택 사항)

TVM은 여러 아키텍처를 대상으로 할 수 있습니다. 다음 예제에서는 GPU에 벡터 덧셈 컴파일을 목표로 합니다.

```
# If you want to run this code, change ``run_cuda = True``
# Note that by default this example is not run in the docs CI.

run_cuda = False
if run_cuda:
    # Change this target to the correct backend for you gpu. For example: cuda (NVIDIA GPUs),
    # rocm (Radeon GPUS), OpenCL (opencl).
    tgt_gpu = tvm.target.Target(target="cuda", host="llvm")

    # Recreate the schedule
    n = te.var("n")
    A = te.placeholder((n,), name="A")
    B = te.placeholder((n,), name="B")
    C = te.compute(A.shape, lambda i: A[i] + B[i], name="C")
    print(type(C))

    s = te.create_schedule(C.op)

    bx, tx = s[C].split(C.op.axis[0], factor=64)


################################################################################
########
    # Finally we must bind the iteration axis bx and tx to threads in the GPU
    # compute grid. The naive schedule is not valid for GPUs, and these are
    # specific constructs that allow us to generate code that runs on a GPU.
```

```
    s[C].bind(bx, te.thread_axis("blockIdx.x"))
    s[C].bind(tx, te.thread_axis("threadIdx.x"))


######################################################################
    # Compilation
    # -----------
    # After we have finished specifying the schedule, we can compile it
    # into a TVM function. By default TVM compiles into a type-erased
    # function that can be directly called from the python side.
    #
    # In the following line, we use tvm.build to create a function.
    # The build function takes the schedule, the desired signature of the
    # function (including the inputs and outputs) as well as target language
    # we want to compile to.
    #
    # The result of compilation fadd is a GPU device function (if GPU is
    # involved) as well as a host wrapper that calls into the GPU
    # function. fadd is the generated host wrapper function, it contains
    # a reference to the generated device function internally.

    fadd = tvm.build(s, [A, B, C], target=tgt_gpu, name="myadd")


######################################################################
########
    # The compiled TVM function exposes a concise C API that can be invoked from
    # any language.
    #
    # We provide a minimal array API in python to aid quick testing and prototyping.
    # The array API is based on the `DLPack <https://github.com/dmlc/dlpack>`_ standard.
    #
    # - We first create a GPU device.
    # - Then tvm.nd.array copies the data to the GPU.
    # - ``fadd`` runs the actual computation
    # - ``numpy()`` copies the GPU array back to the CPU (so we can verify correctness).
    #
```

```
    # Note that copying the data to and from the memory on the GPU is a required step.

    dev = tvm.device(tgt_gpu.kind.name, 0)

    n = 1024
    a = tvm.nd.array(np.random.uniform(size=n).astype(A.dtype), dev)
    b = tvm.nd.array(np.random.uniform(size=n).astype(B.dtype), dev)
    c = tvm.nd.array(np.zeros(n, dtype=C.dtype), dev)
    fadd(a, b, c)
    tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())


################################################################################
########
    # Inspect the Generated GPU Code
    # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    # You can inspect the generated code in TVM. The result of tvm.build is a TVM
    # Module. fadd is the host module that contains the host wrapper, it also
    # contains a device module for the CUDA (GPU) function.
    #
    # The following code fetches the device module and prints the content code.

    if (
        tgt_gpu.kind.name == "cuda"
        or tgt_gpu.kind.name == "rocm"
        or tgt_gpu.kind.name.startswith("opencl")
    ):
        dev_module = fadd.imported_modules[0]
        print("-----GPU code-----")
        print(dev_module.get_source())
    else:
        print(fadd.get_source())
```

컴파일된 모듈 저장 및 로드

런타임 컴파일 외에도 컴파일된 모듈을 파일에 저장하고 나중에 다시 로드할 수 있습니다.

다음 코드는 먼저 다음 단계를 수행합니다.

- 컴파일된 호스트 모듈을 오브젝트 파일에 저장합니다.
- 그런 다음 장치 모듈을 ptx 파일에 저장합니다.
- cc.create_shared는 컴파일러(gcc)를 호출하여 공유 라이브러리를 만듭니다.

```
from tvm.contrib import cc
from tvm.contrib import utils

temp = utils.tempdir()
fadd.save(temp.relpath("myadd.o"))
if tgt.kind.name == "cuda":
    fadd.imported_modules[0].save(temp.relpath("myadd.ptx"))
if tgt.kind.name == "rocm":
    fadd.imported_modules[0].save(temp.relpath("myadd.hsaco"))
if tgt.kind.name.startswith("opencl"):
    fadd.imported_modules[0].save(temp.relpath("myadd.cl"))
cc.create_shared(temp.relpath("myadd.so"), [temp.relpath("myadd.o")])
print(temp.listdir())
```

Out:

```
['myadd.o', 'myadd.so']
```

모듈 스토리지 형식

CPU(호스트) 모듈은 공유 라이브러리(.so)로 직접 저장됩니다. 디바이스 코드에는 여러 사용자 지정 형식이 있을 수 있습니다. 이 예제에서 디바이스 코드는 ptx와 메타 데이터 json 파일에 저장됩니다. 가져오기를 통해 별도로 로드하고 연결할 수 있습니다.

컴파일된 모듈 로드

파일 시스템에서 컴파일된 모듈을 로드하고 코드를 실행할 수 있습니다. 다음 코드는 호스트와 장치 모듈을 별도로 로드하고 함께 연결합니다. 새로 로드된 함수가 작동하는지 확인할 수 있습니다.

```
fadd1 = tvm.runtime.load_module(temp.relpath("myadd.so"))
if tgt.kind.name == "cuda":
    fadd1_dev = tvm.runtime.load_module(temp.relpath("myadd.ptx"))
    fadd1.import_module(fadd1_dev)

if tgt.kind.name == "rocm":
    fadd1_dev = tvm.runtime.load_module(temp.relpath("myadd.hsaco"))
    fadd1.import_module(fadd1_dev)

if tgt.kind.name.startswith("opencl"):
    fadd1_dev = tvm.runtime.load_module(temp.relpath("myadd.cl"))
    fadd1.import_module(fadd1_dev)

fadd1(a, b, c)
tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())
```

모든 것을 하나의 라이브러리로 통합

위의 예에서는 장치와 호스트 코드를 별도로 저장합니다. TVM은 또한 모든 것을 하나의 공유 라이브러리로 내보내기를 지원합니다. 내부적으로 디바이스 모듈을 이진 Blob으로 압축하고 호스트 코드와 함께 연결합니다. 현재 Metal, OpenCL 및 CUDA 모듈의 패킹을 지원합니다.

```
fadd.export_library(temp.relpath("myadd_pack.so"))
fadd2 = tvm.runtime.load_module(temp.relpath("myadd_pack.so"))
fadd2(a, b, c)
tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())
```

런타임 API 및 스레드 안전성

TVM의 컴파일된 모듈은 TVM 컴파일러에 종속되지 않습니다. 대신 최소 런타임 라이브러리에만 의존합니다. TVM 런타임 라이브러리는 장치 드라이버를 래핑하고 컴파일된 함수에 스레드로부터 안전하고 장치에 구애받지 않는 호출을 제공합니다.

즉, 해당 GPU에 대한 코드를 컴파일한 경우 모든 GPU의 모든 스레드에서 컴파일된 TVM 함수를 호출할 수 있습니다.

OpenCL 코드 생성

TVM은 여러 백엔드에 코드 생성 기능을 제공합니다. CPU 백엔드에서 실행되는 OpenCL 코드 또는 LLVM 코드를 생성할 수도 있습니다.

다음 코드 블록은 OpenCL 코드를 생성하고, OpenCL 장치에 배열을 생성하고, 코드의 정확성을 확인합니다.

```
if tgt.kind.name.startswith("opencl"):
    fadd_cl = tvm.build(s, [A, B, C], tgt, name="myadd")
    print("------opencl code------")
    print(fadd_cl.imported_modules[0].get_source())
    dev = tvm.cl(0)
    n = 1024
    a = tvm.nd.array(np.random.uniform(size=n).astype(A.dtype), dev)
    b = tvm.nd.array(np.random.uniform(size=n).astype(B.dtype), dev)
    c = tvm.nd.array(np.zeros(n, dtype=C.dtype), dev)
    fadd_cl(a, b, c)
    tvm.testing.assert_allclose(c.numpy(), a.numpy() + b.numpy())
```

TE 스케줄링 프리미티브

TVM에는 다양한 스케줄링 프리미티브가 포함되어 있습니다.

- split: 정의된 계수에 따라 지정된 축을 두 개의 축으로 분할합니다.
- tile: 타일은 정의된 요인에 따라 계산을 두 축으로 분할합니다.
- fuse: 한 계산의 두 개의 연속된 축을 융합합니다.
- reorder: 계산의 축을 정의된 순서로 reorder할 수 있습니다.
- bind: GPU 프로그래밍에 유용한 특정 스레드에 계산을 바인딩할 수 있습니다.
- compute_at: 기본적으로 TVM은 기본적으로 함수의 가장 바깥쪽 수준 또는 루트에서 텐서를 계산합니다. compute_at는 하나의 텐서가 다른 연산자에 대한 계산의 첫 번째 축에서 계산되어야 함을 지정합니다.
- compute_inline: 인라인으로 표시되면 계산이 확장된 다음 텐서가 필요한 주소에 삽입됩니다.
- compute_root: 계산을 함수의 가장 바깥쪽 계층 또는 루트로 이동합니다. 즉, 다음 단계로 이동하기 전에 계산 단계가 완전히 계산됩니다.

이러한 프리미티브에 대한 자세한 설명은 **Schedule Primitives** 문서 페이지에서 찾을 수 있습니다.

https://tvm.apache.org/docs/how_to/work_with_schedules/schedule_primitives.html#schedule-primitives

예제 2: TE를 사용한 행렬 곱셈 수동 최적화

이제 우리는 단 18줄의 Python 코드로 TVM이 일반적인 행렬 곱셈 연산의 속도를 18배 높이는 방법을 보여주는 두 번째 고급 예제를 고려할 것입니다.

행렬 곱셈은 계산 집약적인 작업입니다. 좋은 CPU 성능을 위한 두 가지 중요한 최적화가 있습니다.

1. 메모리 액세스의 캐시 적중률을 높입니다. 복잡한 수치 계산과 핫스팟 메모리 액세스는 모두 높은 캐시 적중률로 가속화될 수 있습니다. 이를 위해서는 원본 메모리 액세스 패턴을 캐시 정책에 맞는 패턴으로 변환해야 합니다.
2. SIMD(Single Instruction Multi-Data)는 벡터 처리 장치라고도 합니다. 단일 값을 처리하는 대신 각 주기에서 SIMD는 작은 데이터 배치를 처리할 수 있습니다. 이를 위해서는 LLVM 백엔드가 SIMD로 낮출 수 있도록 루프 본문의 데이터 액세스 패턴을 균일한 패턴으로 변환해야 합니다.

이 튜토리얼에 사용된 기술은 이 **repository**에 언급된 트릭의 하위 집합입니다. 그 중 일부는 TVM 추상화에 의해 자동으로 적용되었지만 일부는 TVM 제약 조건으로 인해 자동으로 적용되지 않습니다. (https://github.com/flame/how-to-optimize-gemm)

준비 및 성능 기준

행렬 곱셈의 numpy 구현에 대한 성능 데이터를 수집하는 것으로 시작합니다.

```
import tvm
import tvm.testing
from tvm import te
import numpy

# The size of the matrix
```

```
# (M, K) x (K, N)
# You are free to try out different shapes, sometimes TVM optimization outperforms numpy with
MKL.
M = 1024
K = 1024
N = 1024

# The default tensor data type in tvm
dtype = "float32"

# You will want to adjust the target to match any CPU vector extensions you
# might have. For example, if you're using using Intel AVX2 (Advanced Vector
# Extensions) ISA for SIMD, you can get the best performance by changing the
# following line to ``llvm -mcpu=core-avx2``, or specific type of CPU you use.
# Recall that you're using llvm, you can get this information from the command
# ``llc --version`` to get the CPU type, and you can check ``/proc/cpuinfo``
# for additional extensions that your processor might support.

target = tvm.target.Target(target="llvm", host="llvm")
dev = tvm.device(target.kind.name, 0)

# Random generated tensor for testing
a = tvm.nd.array(numpy.random.rand(M, K).astype(dtype), dev)
b = tvm.nd.array(numpy.random.rand(K, N).astype(dtype), dev)

# Repeatedly perform a matrix multiplication to get a performance baseline
# for the default numpy implementation
np_repeat = 100
np_running_time = timeit.timeit(
      setup="import numpy₩n"
      "M = " + str(M) + "₩n"
      "K = " + str(K) + "₩n"
      "N = " + str(N) + "₩n"
      'dtype = "float32"₩n'
      "a = numpy.random.rand(M, K).astype(dtype)₩n"
      "b = numpy.random.rand(K, N).astype(dtype)₩n",
      stmt="answer = numpy.dot(a, b)",
```

```
    number=np_repeat,
)
print("Numpy running time: %f" % (np_running_time / np_repeat))

answer = numpy.dot(a.numpy(), b.numpy())
```

Out:

```
Numpy running time: 0.018567
```

이제 TVM TE를 사용하여 기본 행렬 곱셈을 작성하고 numpy 구현과 동일한 결과를 생성하는지 확인합니다. 또한 Schedule 최적화의 성능을 측정하는 데 도움이 되는 함수를 작성합니다.

```
# TVM Matrix Multiplication using TE
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k), name="C")

# Default schedule
s = te.create_schedule(C.op)
func = tvm.build(s, [A, B, C], target=target, name="mmult")

c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), dev)
func(a, b, c)
tvm.testing.assert_allclose(c.numpy(), answer, rtol=1e-5)



def evaluate_operation(s, vars, target, name, optimization, log):
    func = tvm.build(s, [A, B, C], target=target, name="mmult")
    assert func

    c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), dev)
    func(a, b, c)
    tvm.testing.assert_allclose(c.numpy(), answer, rtol=1e-5)

    evaluator = func.time_evaluator(func.entry_name, dev, number=10)
```

```
    mean_time = evaluator(a, b, c).mean
    print("%s: %f" % (optimization, mean_time))
    log.append((optimization, mean_time))


log = []

evaluate_operation(s, [A, B, C], target=target, name="mmult", optimization="none", log=log)
```

Out:

```
none: 3.435423
```

TVM lower 함수를 사용하여 연산자와 기본 Schedule의 중간 표현 (IR)을 살펴보겠습니다. 구현이 본질적으로 A 및 B 행렬의 인덱스에 대해 세 개의 중첩 루프를 사용하는 행렬 곱셈의 네이티브 구현이라는 점에 유의하세요.

```
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"),
             B: T.Buffer((1024, 1024), "float32"), C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        for x, y in T.grid(1024, 1024):
            C_1 = T.Buffer((1048576,), data=C.data)
            C_1[x * 1024 + y] = T.float32(0)
            for k in range(1024):
                cse_var_2: T.int32 = x * 1024
                cse_var_1: T.int32 = cse_var_2 + y
                A_1 = T.Buffer((1048576,), data=A.data)
```

```
        B_1 = T.Buffer((1048576,), data=B.data)
        C_1[cse_var_1] = C_1[cse_var_1] + A_1[cse_var_2 + k] * B_1[k * 1024 + y]
```

최적화 1: 블로킹

캐시 적중률을 높이는 중요한 트릭은 블록 내부가 메모리 지역성이 높은 작은 이웃이 되도록 메모리 액세스를 구조화하는 차단입니다. 이 튜토리얼에서는 블록 계수 32를 선택합니다. 그러면 메모리의 32 * 32 * sizeof(float) 영역을 채우는 블록이 생성됩니다. 이는 L1 캐시에 대한 참조 캐시 크기 32KB와 관련하여 4KB의 캐시 크기에 해당합니다.

먼저 **C** 작업에 대한 기본 Schedule을 만든 다음 지정된 블록 계수를 사용하여 **tile** 스케줄링 프리미티브를 적용하고 스케줄링 프리미티브는 결과 루프 순서를 가장 바깥쪽에서 가장 안쪽으로 벡터 **[x_outer, y_outer, x_inner, y_inner]**로 반환합니다.

그런 다음 연산 출력에 대한 축소 축을 가져오고 4의 계수를 사용하여 분할 연산을 수행합니다. 이 요소는 현재 작업 중인 블로킹 최적화에 직접적인 영향을 미치지는 않지만 나중에 벡터화를 적용할 때 유용할 것입니다.

이제 연산이 블록되었으므로 계산을 다시 정렬하여 축소 연산을 계산의 가장 바깥쪽 루프에 배치하여 블록된 데이터가 캐시에 남아 있도록 할 수 있습니다. 이렇게 하면 Schedule이 완료되고 네이티브 Schedule과 비교하여 성능을 빌드하고 테스트할 수 있습니다.

```
bn = 32

# Blocking by loop tiling
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

# Hoist reduction domain outside the blocking loop
```

```
s[C].reorder(xo, yo, ko, ki, xi, yi)

evaluate_operation(s, [A, B, C], target=target, name="mmult", optimization="blocking", log=log)
```

Out:

```
blocking: 0.299429
```

캐싱을 활용하도록 계산 순서를 다시 지정하면 계산 성능이 크게 향상되는 것을 볼 수 있습니다. 이제 내부 표현을 출력하고 원본과 비교합니다.

```
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"),
             B: T.Buffer((1024, 1024), "float32"), C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        for x_outer, y_outer in T.grid(32, 32):
            C_1 = T.Buffer((1048576,), data=C.data)
            for x_inner_init, y_inner_init in T.grid(32, 32):
                C_1[x_outer * 32768 + x_inner_init * 1024 + y_outer
                    * 32 + y_inner_init] = T.float32(0)
            for k_outer, k_inner, x_inner, y_inner in T.grid(256, 4, 32, 32):
                cse_var_3: T.int32 = y_outer * 32
                cse_var_2: T.int32 = x_outer * 32768 + x_inner * 1024
                cse_var_1: T.int32 = cse_var_2 + cse_var_3 + y_inner
                A_1 = T.Buffer((1048576,), data=A.data)
                B_1 = T.Buffer((1048576,), data=B.data)
                C_1[cse_var_1] = C_1[cse_var_1] + A_1[cse_var_2 + k_outer * 4
                    + k_inner] * B_1[k_outer * 4096 + k_inner * 1024 + cse_var_3 + y_inner]
```

최적화 2: 벡터화

또 다른 중요한 최적화 트릭은 벡터화입니다. 메모리 액세스 패턴이 균일하면 컴파일러는 이 패턴을 감지하고 연속 메모리를 SIMD 벡터 프로세서에 전달할 수 있습니다. TVM에서는 **벡터화** 인터페이스를 사용하여 이 하드웨어 기능을 활용하여 컴파일러에 이 패턴을 알릴 수 있습니다.

```
# Apply the vectorization optimization
s[C].vectorize(yi)

evaluate_operation(s, [A, B, C], target=target, name="mmult", optimization="vectorization", log=log)

# The generalized IR after vectorization
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
vectorization: 0.280275
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
             C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        for x_outer, y_outer in T.grid(32, 32):
            C_1 = T.Buffer((1048576,), data=C.data)
            for x_inner_init in range(32):
                C_1[x_outer * 32768 + x_inner_init * 1024 + y_outer * 32:x_outer
```

```
                    * 32768 + x_inner_init * 1024 + y_outer * 32 + 32]
                    = T.Broadcast(T.float32(0), 32)
            for k_outer, k_inner, x_inner in T.grid(256, 4, 32):
                cse_var_3: T.int32 = y_outer * 32
                cse_var_2: T.int32 = x_outer * 32768 + x_inner * 1024
                cse_var_1: T.int32 = cse_var_2 + cse_var_3
                A_1 = T.Buffer((1048576,), data=A.data)
                B_1 = T.Buffer((1048576,), data=B.data)
                C_1[cse_var_1:cse_var_1 + 32] = C_1[cse_var_1:cse_var_1 + 32]
                    + T.Broadcast(A_1[cse_var_2 + k_outer * 4 + k_inner], 32)
                    * B_1[k_outer * 4096 + k_inner * 1024 + cse_var_3:k_outer * 4096
                    + k_inner * 1024 + cse_var_3 + 32]
```

최적화 3: 루프 순열

위의 IR을 보면 내부 루프 행 데이터가 벡터화되고 B가 PackedB로 변환되는 것을 볼 수 있습니다(이는 내부 루프의 (float32x32*)B2 부분에서 분명합니다). 이제 PackedB의 순회가 순차적입니다. 그래서 우리는 A의 액세스 패턴을 살펴볼 것입니다. 현재 Schedule에서 A는 캐시에 친숙하지 않은 열별로 액세스됩니다. ki와 내부 축 xi의 중첩 루프 순서를 변경하면 A 행렬에 대한 액세스 패턴이 더 캐시 친화적이 됩니다.

```
s = te.create_schedule(C.op)
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

# re-ordering
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)

evaluate_operation(
    s, [A, B, C], target=target, name="mmult", optimization="loop permutation", log=log
```

```
)

# Again, print the new generalized IR
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

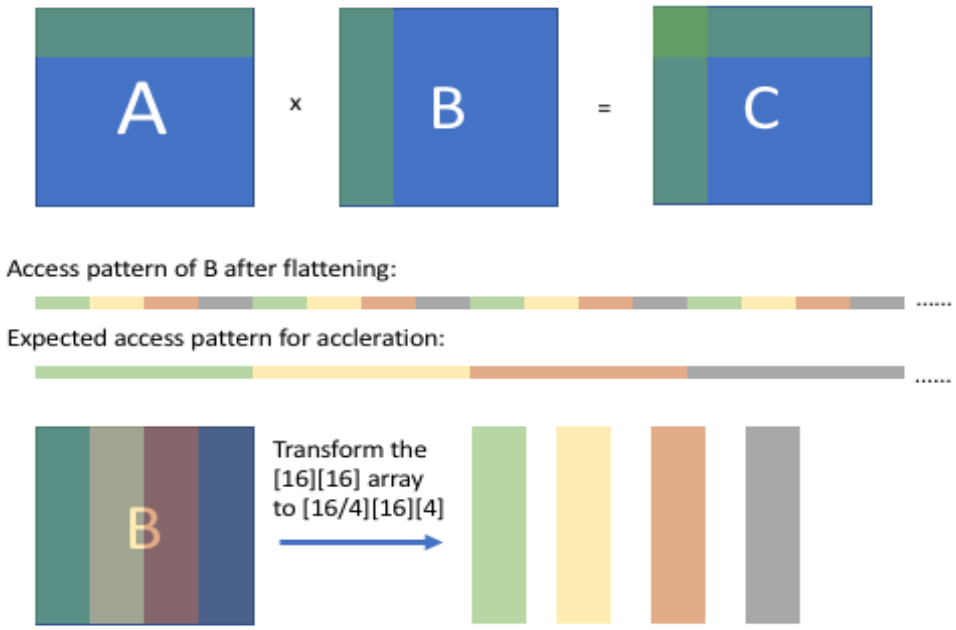
Out:

```
loop permutation: 0.115154
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
             C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        for x_outer, y_outer in T.grid(32, 32):
            C_1 = T.Buffer((1048576,), data=C.data)
            for x_inner_init in range(32):
                C_1[x_outer * 32768 + x_inner_init * 1024 + y_outer * 32:x_outer * 32768
                    + x_inner_init * 1024 + y_outer * 32 + 32] = T.Broadcast(T.float32(0), 32)
            for k_outer, x_inner, k_inner in T.grid(256, 32, 4):
                cse_var_3: T.int32 = y_outer * 32
                cse_var_2: T.int32 = x_outer * 32768 + x_inner * 1024
                cse_var_1: T.int32 = cse_var_2 + cse_var_3
                A_1 = T.Buffer((1048576,), data=A.data)
                B_1 = T.Buffer((1048576,), data=B.data)
                C_1[cse_var_1:cse_var_1 + 32] = C_1[cse_var_1:cse_var_1 + 32]
                    + T.Broadcast(A_1[cse_var_2 + k_outer * 4 + k_inner], 32)
                    * B_1[k_outer * 4096 + k_inner * 1024 + cse_var_3:k_outer * 4096
                    + k_inner * 1024 + cse_var_3 + 32]
```

최적화 4: 어레이 패킹

또 다른 중요한 트릭은 배열 패킹입니다. 이 트릭은 배열의 스토리지 차원을 재정렬하여 평면화 후 특정 차원의 연속 액세스 패턴을 순차 패턴으로 변환하는 것입니다.



위의 그림과 같이 계산을 차단한 후 B의 배열 액세스 패턴(평면화 후)을 관찰할 수 있으며, 이는 규칙적이지만 불연속적입니다. 우리는 약간의 변환 후에 지속적인 액세스 패턴을 얻을 수 있을 것으로 기대합니다. **[16][16]** 배열을 **[16/4][16][4]** 배열로 재정렬하면 압축된 배열에서 해당 값을 가져올 때 B의 액세스 패턴이 순차적으로 됩니다.

이를 위해 B의 새로운 패킹을 고려하여 새로운 기본 Schedule로 시작해야 합니다. TE는 최적화된 연산자를 작성하기 위한 강력하고 표현력이 뛰어난 언어이지만 작성하려는 기본 알고리즘, 데이터 구조 및 하드웨어 대상에 대한 지식이 필요한 경우가 많습니다. 튜토리얼의 뒷부분에서는 TVM이 이러한 부담을 덜어주기 위한 몇 가지 옵션에 대해 논의할 것입니다. 어쨌든 새로운 최적화 된 Schedule로 넘어 갑시다.

```
# We have to re-write the algorithm slightly.
packedB = te.compute((N / bn, K, bn), lambda x, y, z: B[y, x * bn + z], name="packedB")
C = te.compute(
```

```
    (M, N),
    lambda x, y: te.sum(A[x, k] * packedB[y // bn, k, tvm.tir.indexmod(y, bn)], axis=k),
    name="C",
)

s = te.create_schedule(C.op)

xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)

x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)

evaluate_operation(s, [A, B, C], target=target, name="mmult",
                           optimization="array packing", log=log)

# Here is the generated IR after array packing.
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
array packing: 0.105407
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
                     C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        packedB = T.allocate([32768], "float32x32", "global")
```

```
        packedB_1 = T.Buffer((32768,), "float32x32", data=packedB)
        for x in T.parallel(32):
            for y in range(1024):
                B_1 = T.Buffer((1048576,), data=B.data)
                packedB_1[x * 1024 + y] = B_1[y * 1024 + x * 32:y * 1024 + x * 32 + 32]
        for x_outer, y_outer in T.grid(32, 32):
            C_1 = T.Buffer((1048576,), data=C.data)
            for x_inner_init in range(32):
                C_1[x_outer * 32768 + x_inner_init * 1024 + y_outer * 32:x_outer * 32768
                    + x_inner_init * 1024 + y_outer * 32 + 32] = T.Broadcast(T.float32(0), 32)
            for k_outer, x_inner, k_inner in T.grid(256, 32, 4):
                cse_var_3: T.int32 = x_outer * 32768 + x_inner * 1024
                cse_var_2: T.int32 = k_outer * 4
                cse_var_1: T.int32 = cse_var_3 + y_outer * 32
                A_1 = T.Buffer((1048576,), data=A.data)
                C_1[cse_var_1:cse_var_1 + 32] = C_1[cse_var_1:cse_var_1 + 32]
                    + T.Broadcast(A_1[cse_var_3 + cse_var_2 + k_inner], 32)
                    * packedB_1[y_outer * 1024 + cse_var_2 + k_inner]
```

최적화 5: 캐싱을 통한 블록 쓰기 최적화

지금까지 모든 최적화는 **C** 행렬을 계산하기 위해 **A** 및 **B** 행렬의 데이터에 효율적으로 액세스하고 계산하는 데 중점을 두었습니다. 블로킹 최적화 후 연산자는 결과를 **C** 블록에 기록하며 액세스 패턴은 순차적이지 않습니다. 순차 캐시 배열을 사용하여 **cache_write**, **compute_at** 및 **unroll**'의 조합을 사용하여 블록 결과를 저장하고 모든 블록 결과가 준비되면 '**C**'에 기록하여 이 문제를 해결할 수 있습니다.

```
s = te.create_schedule(C.op)

# Allocate write cache
CC = s.cache_write(C, "global")
```

```
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)

# Write cache is computed at yo
s[CC].compute_at(s[C], yo)

# New inner axes
xc, yc = s[CC].op.axis

(k,) = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)

x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)

evaluate_operation(s, [A, B, C], target=target, name="mmult",
                           optimization="block caching", log=log)

# Here is the generated IR after write cache blocking.
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
block caching: 0.111790
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
                C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        packedB = T.allocate([32768], "float32x32", "global")
```

```
        C_global = T.allocate([1024], "float32", "global")
        packedB_1 = T.Buffer((32768,), "float32x32", data=packedB)
        for x in T.parallel(32):
            for y in range(1024):
                B_1 = T.Buffer((1048576,), data=B.data)
                packedB_1[x * 1024 + y] = B_1[y * 1024 + x * 32:y * 1024 + x * 32 + 32]
        for x_outer, y_outer in T.grid(32, 32):
            C_global_1 = T.Buffer((1024,), data=C_global)
            for x_c_init in range(32):
                C_global_1[x_c_init * 32:x_c_init * 32 + 32] = T.Broadcast(T.float32(0), 32)
            for k_outer, x_c in T.grid(256, 32):
                cse_var_4: T.int32 = k_outer * 4
                cse_var_3: T.int32 = x_c * 32
                cse_var_2: T.int32 = y_outer * 1024 + cse_var_4
                cse_var_1: T.int32 = x_outer * 32768 + x_c * 1024 + cse_var_4
                A_1 = T.Buffer((1048576,), data=A.data)
                C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                    + T.Broadcast(A_1[cse_var_1], 32) * packedB_1[cse_var_2]
                C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                    + T.Broadcast(A_1[cse_var_1 + 1], 32) * packedB_1[cse_var_2 + 1]
                C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                    + T.Broadcast(A_1[cse_var_1 + 2], 32) * packedB_1[cse_var_2 + 2]
                C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                    + T.Broadcast(A_1[cse_var_1 + 3], 32) * packedB_1[cse_var_2 + 3]
            for x_inner, y_inner in T.grid(32, 32):
                C_1 = T.Buffer((1048576,), data=C.data)
                C_1[x_outer * 32768 + x_inner * 1024 + y_outer * 32 + y_inner]
                    = C_global_1[x_inner * 32 + y_inner]
```

최적화 6: 병렬화

지금까지 우리의 계산은 단일 코어만 사용하도록 설계되었습니다. 거의 모든 최신 프로세서에는 다중 코어가 있으며 계산을 병렬로 실행하면 계산이 도움이 될 수 있습니다. 최종 최적화는 스레드 수준 병렬화를 활용하는 것입니다.

```
# parallel
s[C].parallel(xo)

x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)

evaluate_operation(
    s, [A, B, C], target=target, name="mmult", optimization="parallelization", log=log
)

# Here is the generated IR after parallelization.
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

Out:

```
parallelization: 0.131911
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
             C: T.Buffer((1024, 1024), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        packedB = T.allocate([32768], "float32x32", "global")
        packedB_1 = T.Buffer((32768,), "float32x32", data=packedB)
        for x in T.parallel(32):
            for y in range(1024):
                B_1 = T.Buffer((1048576,), data=B.data)
                packedB_1[x * 1024 + y] = B_1[y * 1024 + x * 32:y * 1024 + x * 32 + 32]
        for x_outer in T.parallel(32):
            C_global = T.allocate([1024], "float32", "global")
            for y_outer in range(32):
                C_global_1 = T.Buffer((1024,), data=C_global)
```

```
                for x_c_init in range(32):
                    C_global_1[x_c_init * 32:x_c_init * 32 + 32] = T.Broadcast(T.float32(0), 32)
                for k_outer, x_c in T.grid(256, 32):
                    cse_var_4: T.int32 = k_outer * 4
                    cse_var_3: T.int32 = x_c * 32
                    cse_var_2: T.int32 = y_outer * 1024 + cse_var_4
                    cse_var_1: T.int32 = x_outer * 32768 + x_c * 1024 + cse_var_4
                    A_1 = T.Buffer((1048576,), data=A.data)
                    C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                        + T.Broadcast(A_1[cse_var_1], 32) * packedB_1[cse_var_2]
                    C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                        + T.Broadcast(A_1[cse_var_1 + 1], 32) * packedB_1[cse_var_2 + 1]
                    C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                        + T.Broadcast(A_1[cse_var_1 + 2], 32) * packedB_1[cse_var_2 + 2]
                    C_global_1[cse_var_3:cse_var_3 + 32] = C_global_1[cse_var_3:cse_var_3 + 32]
                        + T.Broadcast(A_1[cse_var_1 + 3], 32) * packedB_1[cse_var_2 + 3]
                for x_inner, y_inner in T.grid(32, 32):
                    C_1 = T.Buffer((1048576,), data=C.data)
                    C_1[x_outer * 32768 + x_inner * 1024 + y_outer * 32 + y_inner]
                        = C_global_1[x_inner * 32 + y_inner]
```

행렬 곱셈 예제 요약

단 18줄의 코드로 위의 간단한 최적화를 적용한 후 생성된 코드는 MKL(Math Kernel Library)을 사용한 numpy의 성능에 접근하기를 시작할 수 있습니다. 작업하면서 성능을 기록했기 때문에 결과를 비교할 수 있습니다.

```
baseline = log[0][1]
print("%s₩t%s₩t%s" % ("Operator".rjust(20), "Timing".rjust(20), "Performance".rjust(20)))
for result in log:
    print(
        "%s₩t%s₩t%s"
        % (result[0].rjust(20), str(result[1]).rjust(20), str(result[1] / baseline).rjust(20))
```

)

Out:

Operator	Timing	Performance
none	3.4354234775999997	1.0
blocking	0.2994290564	0.08715928570447516
vectorization	0.28027533790000003	0.08158392690958771
loop permutation	0.11515386050000001	0.033519553339155427
array packing	0.10540671910000002	0.030682307374122496
block caching	0.11179033520000001	0.03254048181509698
parallelization	0.13191064889999998	0.03839720190541204

웹 페이지의 출력은 비독점적 Docker 컨테이너의 실행 시간을 반영하며 신뢰할 수 없는 것으로 간주되어야 합니다. 튜토리얼을 직접 실행하여 TVM으로 달성되는 성능 향상을 관찰하고 각 예제를 주의 깊게 살펴보고 행렬 곱셈 연산에 대한 반복적인 개선 사항을 이해하는 것이 좋습니다.

최종 메모 및 요약

앞서 언급했듯이 TE 및 스케줄링 프리미티브를 사용하여 최적화를 적용하는 방법에는 기본 아키텍처와 알고리즘에 대한 약간의 지식이 필요할 수 있습니다. 그러나 TE는 잠재적인 최적화를 검색할 수 있는 더 복잡한 알고리즘의 기초 역할을 하도록 설계되었습니다. TE에 대한 이 소개를 통해 얻은 지식을 바탕으로 이제 TVM이 Schedule 최적화 프로세스를 자동화할 수 있는 방법을 살펴볼 수 있습니다.

이 튜토리얼에서는 벡터 더하기 및 행렬 곱셈 예제를 사용하여 TVM 텐서 표현식(TE) 워크플로우를 살펴보았습니다. 일반적인 워크플로는 다음과 같습니다.

- 일련의 연산을 통해 계산을 표현합니다.
- 스케줄 프리미티브 사용을 계산하는 방법을 표현합니다.

- 원하는 타깃 함수로 컴파일합니다.
- 선택적으로 나중에 로드할 함수를 저장합니다.

다음 튜토리얼에서는 행렬 곱셈 예제를 확장하고, 특정 플랫폼에 대한 계산을 자동으로 최적화할 수 있는 튜닝 가능한 매개 변수를 사용하여 행렬 곱셈 및 기타 연산의 일반 템플릿을 빌드하는 방법을 보여줍니다.

스크립트의 총 실행 시간 : (1 분 0.554 초)

Download Python source code: tensor_expr_get_started.py

https://tvm.apache.org/docs/_downloads/40a01cffb015a67aaec0fad7e27cf80d/tensor_expr_get_started.py

Download Jupyter notebook: tensor_expr_get_started.ipynb

https://tvm.apache.org/docs/_downloads/4459ebf5b03d332f7f380abdaef81c05/tensor_expr_get_started.ipynb

1.9 스케줄 템플릿과 AutoTVM을 통한 연산자 최적화

이 튜토리얼에서는 TVM TE(Tensor Expression) 언어를 사용하여 AutoTVM에서 최적의 Schedule을 찾기 위해 검색할 수 있는 Schedule 템플릿을 작성하는 방법을 보여줍니다. 이 프로세스를 자동 튜닝이라고 하며, 이는 텐서 계산 최적화 프로세스를 자동화하는 데 도움이 됩니다.

이 튜토리얼은 TE를 사용하여 행렬 곱셈을 작성하는 방법에 대한 이전 튜토리얼을 기반으로 합니다.

https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html

자동 튜닝에는 두 단계가 있습니다.

- 첫 번째 단계는 검색 공간을 정의하는 것입니다.
- 두 번째 단계는 이 공간을 탐색하기 위해 검색 알고리즘을 실행하는 것입니다.

이 튜토리얼에서는 TVM에서 이 두 단계를 수행하는 방법을 알아볼 수 있습니다. 전체 워크플로는 행렬 곱셈 예제로 설명됩니다.

Note

이 튜토리얼은 Windows 또는 최신 버전의 macOS에서 실행되지 않습니다. 실행하려면 이 튜토리얼의 본문을 **if __name__ == "__main__":** 블록으로 래핑해야 합니다.

종속성 설치

TVM에서 autotvm 패키지를 사용하려면 몇 가지 추가 종속성을 설치해야 합니다.

```
pip3 install --user psutil xgboost cloudpickle
```

튜닝에서 TVM을 더 빠르게 실행하려면 cython을 TVM의 FFI (Foreign Function Interface, 외부 함수 인터페이스)로 사용하는 것이 좋습니다. TVM의 루트 디렉터리에서 다음을 실행합니다.

```
pip3 install --user cython
sudo make cython3
```

이제 Python 코드로 돌아갑니다. 먼저 필요한 패키지를 가져옵니다.

```
import logging
import sys

import numpy as np
import tvm
from tvm import te
import tvm.testing

# the module is called `autotvm`
from tvm import autotvm
```

TE를 사용한 기본 행렬 곱셈

TE를 사용한 행렬 곱셈의 기본 구현을 상기해 보세요. 여기에 몇 가지 변경 사항을 적어 둡니다. 곱셈을 Python 함수 정의로 래핑합니다. 단순화를 위해 재정렬의 블록 크기를 정의하는 고정 값을 사용하여 분할 최적화에 주의를 기울일 것입니다.

```
def matmul_basic(N, L, M, dtype):

    A = te.placeholder((N, L), name="A", dtype=dtype)
    B = te.placeholder((L, M), name="B", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)
```

```
    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    yo, yi = s[C].split(y, 8)
    xo, xi = s[C].split(x, 8)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

AutoTVM을 사용한 행렬 곱셈

이전 Schedule 코드에서는 상수 "8"을 바둑판식 배열 계수로 사용합니다. 그러나 최상의 타일링 계수는 실제 하드웨어 환경과 입력 shape에 따라 달라지기 때문에 최상의 타일링 요소가 아닐 수 있습니다.

더 넓은 범위의 입력 shapes 및 타겟 하드웨어에서 Schedule 코드를 이식할 수 있도록 하려면 후보 값 세트를 정의하고 타겟 하드웨어의 측정 결과에 따라 가장 적합한 값을 선택하는 것이 좋습니다.

autotvm에서는 튜닝 가능한 매개변수 또는 이러한 종류의 값에 대한 "노브" (knob)를 정의할 수 있습니다.

기본 행렬 곱셈 템플릿

분할 스케줄링 작업의 블록 크기에 대해 튜닝 가능한 매개변수 세트를 생성하는 방법의 예부터 시작합니다.

```
# Matmul V1: List candidate values
@autotvm.template("tutorial/matmul_v1")   # 1. use a decorator
def matmul_v1(N, L, M, dtype):
    A = te.placeholder((N, L), name="A", dtype=dtype)
    B = te.placeholder((L, M), name="B", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)

    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    # 2. get the config object
    cfg = autotvm.get_config()

    # 3. define search space
    cfg.define_knob("tile_y", [1, 2, 4, 8, 16])
    cfg.define_knob("tile_x", [1, 2, 4, 8, 16])

    # 4. schedule according to config
    yo, yi = s[C].split(y, cfg["tile_y"].val)
    xo, xi = s[C].split(x, cfg["tile_x"].val)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

여기서는 이전 Schedule 코드를 네 번 수정하고 튜닝 가능한 "템플릿"을 얻습니다. 수정 사항을 하나씩 설명하겠습니다.

1. 데코레이터를 사용하여 이 함수를 단순 템플릿으로 표시합니다.
2. 구성 객체 가져오기: 이 **cfg**를 이 함수의 인수로 간주할 수 있지만 다른 방식으로 얻습니다. 이 인수를 사용하면 이 함수는 더 이상 결정적 Schedule이 아닙니다. 대신 이 함수에 다른 구성을 전달하고 다른 Schedule을 가져올 수 있습니다. 이와 같은 구성 개체를 사용하는 함수를

"템플릿"이라고 합니다.

템플릿 함수를 더 간결하게 만들기 위해 단일 함수 내에서 매개변수 검색 공간을 정의하기 위해 두 가지 작업을 수행할 수 있습니다.

1. 설정된 값에 걸쳐 검색 공간을 정의합니다. 이것은 **cfg**를 ConfigSpace 객체로 만들어 수행됩니다. 이 기능에서 튜닝 가능한 모든 노브를 수집하고 검색 공간을 구축합니다.
2. 이 공간의 엔터티에 따라 Schedule 합니다. 이 작업은 **cfg**를 ConfigEntity 객체로 만들어 수행됩니다. ConfigEntity 인 경우 모든 공간 정의 API (즉, **cfg.define_XXXXX(...)**)를 무시합니다. 대신, 모든 튜닝 가능한 노브에 대한 결정론적 값을 저장하고 이 값에 따라 Schedule 합니다.

자동 튜닝 중에 먼저 **ConfigSpace** 개체와 함께 이 템플릿을 호출하여 검색 공간을 구축합니다. 그런 다음 빌드된 공간에서 다른 ConfigEntity를 사용하여 이 템플릿을 호출하여 다른 Schedule을 가져옵니다. 마지막으로 다른 Schedule에서 생성된 코드를 측정하고 가장 적합한 코드를 선택합니다.

3. 두 개의 튜닝 가능한 노브를 정의합니다. 첫 번째는 5개의 가능한 값으로 **tile_y**됩니다. 두 번째는 가능한 값의 동일한 목록으로 **tile_x**됩니다. 이 두 노브는 독립적이므로 크기가 25 = 5x5 인 검색 공간에 걸쳐 있습니다.
4. 구성 노브는 **분할** Schedule 작업으로 전달되어 이전에 **cfg**에서 정의한 5x5 결정적 값에 따라 Schedule할 수 있습니다.

고급 매개변수 API를 사용한 행렬 곱셈 템플릿

이전 템플릿에서는 노브에 사용할 수 있는 모든 값을 수동으로 나열했습니다. 이것은 공간을 정의하는 가장 낮은 수준의 API이며 검색할 매개 변수 공간의 명시적 열거를 제공합니다. 그러나 검색 공간을 더 쉽고 스마트하게 정의할 수 있는 또 다른 API 집합도 제공합니다. 가능하면 이 상위 수준 API를 사용하는 것이 좋습니다

다음 예에서는 **ConfigSpace.define_split**를 사용하여 분할 노브를 정의합니다. 축을 분할하고 공간을 구성하는 가능한 모든 방법을 열거합니다.

또한 재정렬 노브에 대한 **ConfigSpace.define_reorder**과 언롤, 벡터화, 스레드 바인딩과 같은 주석에 대한 **ConfigSpace.define_annotate**도 있습니다. 상위 수준 API가 요구 사항을 충족할 수 없는 경우 항상 하위 수준 API를 사용하도록 대체할 수 있습니다.

```
@autotvm.template("tutorial/matmul")
def matmul(N, L, M, dtype):
    A = te.placeholder((N, L), name="A", dtype=dtype)
    B = te.placeholder((L, M), name="B", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)

    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    ##### define space begin #####
    cfg = autotvm.get_config()
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2)
    ##### define space end #####

    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

cfg.define_split에 대한 추가 설명

이 템플릿에서 **cfg.define_split("tile_y", y, num_outputs=2)**는 y 길이의 인수를 사용하여 y 축을

두 축으로 분할할 수 있는 가능한 모든 조합을 열거합니다. 예를 들어, y의 길이가 32이고 32의 인수를 사용하여 두 개의 축으로 분할하려는 경우 (외부 축의 길이, 내부 축의 길이) 쌍, 즉 (6, 32), (16, 2), (8, 4), (4, 8), (2, 16) 또는 (1, 32). 이것들은 모두 tile_y의 6가지 가능한 값입니다.

스케줄링 중에 **cfg["tile_y"]**는 **SplitEntity** 객체입니다. 외부 축과 내부 축의 길이를 **cfg['tile_y'].size**(두 개의 요소가 있는 tuple)에 저장합니다. 이 템플릿에서는 **yo, yi = cfg['tile_y'].apply(s, C, y)**를 사용하여 적용합니다. 실제로 이것은 **yo, yi = s[C].split(y, cfg["tile_y"].size[1])** 또는 **yo, yi = s[C].split(y, nparts=cfg['tile_y"].size[0])** 과 같습니다.

cfg.apply API를 사용하면 다단계 분할(즉, num_outputs >= 3인 경우)이 더 쉬워진다는 장점이 있습니다.

2단계: AutoTVM을 사용하여 행렬 곱셈 최적화

1단계에서는 분할 Schedule에 사용되는 블록 크기를 매개변수화할 수 있는 행렬 곱셈 템플릿을 작성했습니다. 이제 이 매개변수 공간에 대한 검색을 수행할 수 있습니다. 다음 단계는 이 공간의 탐색을 안내할 튜너를 선택하는 것입니다.

TVM의 자동 튜너

튜너에 대한 작업은 다음 의사코드 (pseudo code)로 설명할 수 있습니다

```
ct = 0
while ct < max_number_of_trials:
    propose a batch of configs
    measure this batch of configs on real hardware and get results
    ct += batch_size
```

다음 구성 배치를 제안할 때 튜너는 다른 전략을 취할 수 있습니다. TVM에서 제공하는 튜너 전략 중 일부는 다음과 같습니다.

- **tvm.autotvm.tuner.RandomTuner**: 공간을 임의의 순서로 열거합니다.
- **tvm.autotvm.tuner.GridSearchTuner**: 그리드 검색 순서로 공간을 열거합니다.
- **tvm.autotvm.tuner.GATuner**: 유전자 알고리즘을 사용하여 공간 검색
- **tvm.autotvm.tuner.XGBTuner**: 모델 기반 방법을 사용합니다. XGBoost 모델을 훈련시켜 IR이 낮아진 속도를 예측하고 예측에 따라 다음 배치를 선택합니다.

공간의 크기, 시간 예산 및 기타 요인에 따라 튜너를 선택할 수 있습니다. 예를 들어 공간이 매우 작은 경우(1000개 미만) 그리드 검색 튜너 또는 임의 튜너로 충분합니다. 공간이 10^9 수준인 경우(CUDA GPU에서 conv2d 연산자의 공간 크기) XGBoostTuner는 보다 효율적으로 탐색하고 더 나은 구성을 찾을 수 있습니다.

튜닝 시작

여기서 행렬 곱셈 예제를 계속합니다. 먼저 튜닝 작업을 만듭니다. 초기화된 검색 공간도 검사할 수 있습니다. 이 경우 512x512 정사각형 행렬 곱셈의 경우 공간 크기는 10x10=100입니다. 작업 및 검색 공간은 선택한 튜너와 독립적입니다.

```
N, L, M = 512, 512, 512
task = autotvm.task.create("tutorial/matmul", args=(N, L, M, "float32"), target="llvm")
print(task.config_space)
```

Out:

```
ConfigSpace (len=100, range_length=100, space_map=
    0 tile_y: Split(policy=factors, product=512, num_outputs=2) len=10
    1 tile_x: Split(policy=factors, product=512, num_outputs=2) len=10
)
```

그런 다음 생성된 코드를 측정하고 튜너를 선택하는 방법을 정의해야 합니다. 우리 공간은 작기 때문에 랜덤 튜너도 괜찮습니다.

데모를 위해 이 튜토리얼에서는 10번의 시도만 수행합니다. 실제로 시간 예산에 따라 더 많은 시도를 수행할 수 있습니다. 튜닝 결과를 로그 파일에 기록합니다. 이 파일은 나중에 튜너에서 검색

한 최상의 구성을 선택하는 데 사용할 수 있습니다.

```
# logging config (for printing tuning log to the screen)
logging.getLogger("autotvm").setLevel(logging.DEBUG)
logging.getLogger("autotvm").addHandler(logging.StreamHandler(sys.stdout))
```

구성을 측정하는 두 단계는 빌드와 실행입니다. 기본적으로 모든 CPU 코어를 사용하여 프로그램을 컴파일합니다. 그런 다음 순차적으로 측정합니다. 분산을 줄이기 위해 5개의 측정값을 수행하고 평균을 냅니다.

```
measure_option = autotvm.measure_option(builder="local", runner=autotvm.LocalRunner(number=5))

# Begin tuning with RandomTuner, log records to file `matmul.log`
# You can use alternatives like XGBTuner.
tuner = autotvm.tuner.RandomTuner(task)
tuner.tune(
    n_trial=10,
    measure_option=measure_option,
    callbacks=[autotvm.callback.log_to_file("matmul.log")],
)
```

Out:

```
waiting for device...

device available

Get devices for measurement successfully!

No: 1   GFLOPS: 0.52/0.52   result: MeasureResult(costs=(0.5208916562000001,), error_no=MeasureErrorNo.NO_ERROR, all_cost=8.584944486618042, timestamp=1708497900.9921024)   [('tile_y', [-1, 512]), ('tile_x', [-1, 1])],None,9

No: 2   GFLOPS: 2.95/2.95   result: MeasureResult(costs=(0.0909711818,), error_no=MeasureErrorNo.NO_ERROR, all_cost=1.6896717548370361, timestamp=1708497902.7007704)   [('tile_y', [-1, 1]), ('tile_x', [-1, 2])],None,10

No: 3   GFLOPS: 12.47/12.47   result: MeasureResult(costs=(0.021529155,),
```

```
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.600501298904419,
timestamp=1708497903.2990787) [('tile_y', [-1, 64]), ('tile_x', [-1, 512])],None,96

No: 4   GFLOPS: 3.84/12.47      result: MeasureResult(costs=(0.0699601004,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=1.363755226135254,
timestamp=1708497904.6681912)       [('tile_y', [-1, 64]), ('tile_x', [-1, 8])],None,36

No: 5   GFLOPS: 3.05/12.47      result: MeasureResult(costs=(0.0881226858,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=1.6694049835205078,
timestamp=1708497906.4790328)       [('tile_y', [-1, 16]), ('tile_x', [-1, 2])],None,14

No: 6   GFLOPS: 10.68/12.47      result: MeasureResult(costs=(0.0251454202,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.6708171367645264,
timestamp=1708497907.1323538)       [('tile_y', [-1, 8]), ('tile_x', [-1, 32])],None,53

No: 7   GFLOPS: 11.95/12.47      result: MeasureResult(costs=(0.0224579592,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.6492056846618652,
timestamp=1708497907.7506113)       [('tile_y', [-1, 4]), ('tile_x', [-1, 128])],None,72

No: 8   GFLOPS: 8.63/12.47      result: MeasureResult(costs=(0.031087045400000003,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.757709264755249,
timestamp=1708497908.4990005)       [('tile_y', [-1, 16]), ('tile_x', [-1, 8])],None,34

No: 9   GFLOPS: 12.64/12.64      result: MeasureResult(costs=(0.021240025399999998,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.5586652755737305,
timestamp=1708497909.1680236)       [('tile_y', [-1, 16]), ('tile_x', [-1, 16])],None,44

No: 10  GFLOPS: 10.07/12.64      result: MeasureResult(costs=(0.0266637334,),
error_no=MeasureErrorNo.NO_ERROR, all_cost=0.6371240615844727,
timestamp=1708497909.8454962)       [('tile_y', [-1, 2]), ('tile_x', [-1, 32])],None,51
```

튜닝이 완료되면 로그 파일에서 가장 잘 측정된 성능을 가진 구성을 선택하고 해당 매개변수로 Schedule을 컴파일할 수 있습니다. 또한 Schedule이 정답을 생성하고 있는지 빠르게 확인합니다. **autotvm.apply_history_best** 컨텍스트에서 직접 **matmul** 함수를 호출할 수 있습니다. 이 함수를 호출하면 인수로 디스패치 컨텍스트를 쿼리하고 동일한 인수로 최상의 구성을 얻습니다.

```
# apply history best from log file
```

```
with autotvm.apply_history_best("matmul.log"):
      with tvm.target.Target("llvm"):
            s, arg_bufs = matmul(N, L, M, "float32")
            func = tvm.build(s, arg_bufs)

# check correctness
a_np = np.random.uniform(size=(N, L)).astype(np.float32)
b_np = np.random.uniform(size=(L, M)).astype(np.float32)
c_np = a_np.dot(b_np)

c_tvm = tvm.nd.empty(c_np.shape)
func(tvm.nd.array(a_np), tvm.nd.array(b_np), c_tvm)

tvm.testing.assert_allclose(c_np, c_tvm.numpy(), rtol=1e-4)
```

Out:

```
Finish loading 10 records
```

최종 메모 및 요약

이 튜토리얼에서는 TVM이 매개 변수 공간을 검색하고 최적화된 Schedule 구성을 선택할 수 있도록 하는 연산자 템플릿을 구축하는 방법을 살펴보았습니다. 이것이 어떻게 작동하는지 더 깊이 이해하려면 **Tensor Expressions 시작하기 <tensor_expr_get_started>_ 튜토리얼**에 설명된 Schedule 작업을 기반으로 Schedule에 새 검색 매개 변수를 추가하여 이 예제를 확장하는 것이 좋습니다. 다음 섹션에서는 사용자가 사용자 정의 템플릿을 제공할 필요 없이 TVM이 일반 연산자를 최적화하는 방법인 AutoScheduler를 시연합니다.

Download Python source code: autotvm_matmul_x86.py

https://tvm.apache.org/docs/_downloads/8e7bbc9dbdda76ac573b24606b41c006/autotvm_matmul_x86.py

Download Jupyter notebook: autotvm_matmul_x86.ipynb

https://tvm.apache.org/docs/_downloads/37bbf9e2065ec8deeb64a8d9fa0755bc/autotvm_matmul_x86.ipynb

1.10 자동 스케줄링을 통한 연산자 최적화

이 튜토리얼에서는 TVM의 자동 스케줄링 기능이 사용자 지정 템플릿을 작성할 필요 없이 최적의 Schedule을 찾는 방법을 보여줍니다.

검색 공간을 정의하기 위해 수동 템플릿에 의존하는 템플릿 기반 **AutoTVM**과 달리 자동 스케줄러에는 템플릿이 필요하지 않습니다. 사용자는 Schedule 명령이나 템플릿 없이 계산 선언을 작성하기만 하면 됩니다. 자동 스케줄러는 자동으로 큰 검색 공간을 생성하고 공간에서 좋은 Schedule을 찾을 수 있습니다.

https://tvm.apache.org/docs/tutorial/autotvm_matmul_x86.html

이 튜토리얼에서는 행렬 곱셈을 예로 사용합니다.

Note

이 튜토리얼은 Windows 또는 최신 버전의 macOS에서 실행되지 않습니다. 실행하려면 이 튜토리얼의 본문을 **if __name__ == "__main__":** 블록으로 래핑해야 합니다.

```
import numpy as np
import tvm
from tvm import te, auto_scheduler
```

행렬 곱셈 정의하기

시작하려면 바이어스 (bias) 추가를 사용하여 행렬 곱셈을 정의합니다. 이것은 TVM Tensor Expression 언어에서 사용할 수 있는 표준 연산을 사용합니다. 가장 큰 차이점은 함수 정의의 맨 위에 register_workload 데코레이터를 사용한다는 것입니다. 함수는 입력/출력 텐서 목록을 반환해야 합니다. 이러한 텐서에서 자동 스케줄러는 전체 계산 그래프를 가져올 수 있습니다.

```
@auto_scheduler.register_workload  # Note the auto_scheduler decorator
def matmul_add(N, L, M, dtype):
    A = te.placeholder((N, L), name="A", dtype=dtype)
```

```
    B = te.placeholder((L, M), name="B", dtype=dtype)
    C = te.placeholder((N, M), name="C", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    matmul = te.compute(
        (N, M),
        lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),
        name="matmul",
        attrs={"layout_free_placeholders": [B]},   # enable automatic layout transform for tensor B
    )
    out = te.compute((N, M), lambda i, j: matmul[i, j] + C[i, j], name="out")

    return [A, B, C, out]
```

검색 작업 만들기

함수가 정의되면 이제 auto_scheduler가 검색할 작업을 생성할 수 있습니다. 이 행렬 곱셈에 대한 특정 매개 변수(이 경우 크기가 1024x1024인 두 정사각형 행렬의 곱셈)를 지정합니다. 그런 다음 N=L=M=1024 및 dtype="float32"를 사용하여 검색 작업을 만듭니다.

맞춤 타겟으로 성능 개선

TVM이 특정 하드웨어 플랫폼을 최대한 활용하려면 CPU 기능을 수동으로 지정해야 합니다. 예를 들어:

- AVX2를 사용하도록 설정하려면 아래 **llvm**을 **llvm -mcpu=core-avx2**로 바꿉니다.
- AVX-512를 사용하도록 설정하려면 아래 **llvm**을 **llvm -mcpu=skylake-avx512**로 바꿉니다.

```
target = tvm.target.Target("llvm")
N = L = M = 1024
task = tvm.auto_scheduler.SearchTask(func=matmul_add, args=(N, L, M, "float32"), target=target)
```

```
# Inspect the computational graph
print("Computational DAG:")
print(task.compute_dag)
```

Out:

```
Computational DAG:
A = PLACEHOLDER [1024, 1024]
B = PLACEHOLDER [1024, 1024]
matmul(i, j) += (A[i, k]*B[k, j])
C = PLACEHOLDER [1024, 1024]
out(i, j) = (matmul[i, j] + C[i, j])
```

자동 스케줄러에 대한 매개 변수 설정

다음으로 자동 스케줄러에 대한 매개 변수를 설정합니다.

- **num_measure_trials**는 검색 중에 사용할 수 있는 측정 시도 횟수입니다. 빠른 데모를 위해 이 튜토리얼에서는 10번의 시도만 합니다. 실제로 1000은 검색이 수렴하기에 좋은 값입니다. 시간 예산에 따라 더 많은 시도를 할 수 있습니다.
- 또한 **RecordToFile**을 사용하여 측정 기록을 파일 **matmul.json**에 기록합니다. 측정 레코드는 기록을 가장 잘 쿼리하고 검색을 다시 시작하고 나중에 더 많은 분석을 수행하는 데 사용할 수 있습니다.
- 더 많은 매개 변수는 **TuningOptions** 를 참조하세요.

```
log_file = "matmul.json"
tune_option = auto_scheduler.TuningOptions(
    num_measure_trials=10,
    measure_callbacks=[auto_scheduler.RecordToFile(log_file)],
    verbose=2,
)
```

검색 실행

이제 모든 입력이 준비되었습니다. 아주 간단하지 않나요? 검색을 시작하고 자동 스케줄러가 마법을 부리도록 할 수 있습니다. 몇 번의 측정 시도 후 로그 파일에서 최상의 Schedule을 로드하고 적용할 수 있습니다.

```
# Run auto-tuning (search)
task.tune(tune_option)
# Apply the best schedule
sch, args = task.apply_best(log_file)
```

Out:

```
.T
```

최적화된 스케줄 검사

자동 Schedule 후 IR을 볼 수 있도록 Shcedule을 낮출 수 있습니다. 자동 스케줄러는 다단계 타일링, 레이아웃 변환, 병렬화, 벡터화, 언롤링 및 연산자 융합을 포함한 최적화를 올바르게 수행합니다.

```
print("Lowered TIR:")
print(tvm.lower(sch, args, simple_mode=True))
```

Out:

```
Lowered TIR:
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
```

```
def main(A: T.Buffer((1024, 1024), "float32"), B: T.Buffer((1024, 1024), "float32"),
         C: T.Buffer((1024, 1024), "float32"), out: T.Buffer((1024, 1024), "float32")):
    T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
    auto_scheduler_layout_transform = T.allocate([1048576], "float32", "global")
    auto_scheduler_layout_transform_1 =
        T.Buffer((1048576,), data=auto_scheduler_layout_transform)
    for ax0_ax1_fused_ax2_fused in T.parallel(128):
        for ax4, ax6, ax7 in T.grid(256, 4, 8):
            B_1 = T.Buffer((1048576,), data=B.data)
            auto_scheduler_layout_transform_1[ax0_ax1_fused_ax2_fused * 8192 + ax4
                * 32 + ax6 * 8 + ax7] =
                    B_1[ax4 * 4096 + ax6 * 1024 + ax0_ax1_fused_ax2_fused * 8 + ax7]
    for i_outer_outer_j_outer_outer_fused in T.parallel(16384):
        matmul = T.allocate([4], "float32x8", "global")
        for i_outer_inner in range(2):
            matmul_1 = T.Buffer((4,), "float32x8", data=matmul)
            matmul_1[0] = T.Broadcast(T.float32(0), 8)
            matmul_1[1] = T.Broadcast(T.float32(0), 8)
            matmul_1[2] = T.Broadcast(T.float32(0), 8)
            matmul_1[3] = T.Broadcast(T.float32(0), 8)
            for k_outer, k_inner in T.grid(256, 4):
                cse_var_2: T.int32 = i_outer_outer_j_outer_outer_fused % 128 * 8192
                    + k_outer * 32 + k_inner * 8
                cse_var_1: T.int32 = i_outer_outer_j_outer_outer_fused // 128 * 8192
                    + i_outer_inner * 4096 + k_outer * 4 + k_inner
                A_1 = T.Buffer((1048576,), data=A.data)
                matmul_1[0] = matmul_1[0] + T.Broadcast(A_1[cse_var_1], 8)
                    * auto_scheduler_layout_transform_1[cse_var_2:cse_var_2 + 8]
                matmul_1[1] = matmul_1[1] + T.Broadcast(A_1[cse_var_1 + 1024], 8)
                    * auto_scheduler_layout_transform_1[cse_var_2:cse_var_2 + 8]
                matmul_1[2] = matmul_1[2] + T.Broadcast(A_1[cse_var_1 + 2048], 8)
                    * auto_scheduler_layout_transform_1[cse_var_2:cse_var_2 + 8]
                matmul_1[3] = matmul_1[3] + T.Broadcast(A_1[cse_var_1 + 3072], 8)
                    * auto_scheduler_layout_transform_1[cse_var_2:cse_var_2 + 8]
            for i_inner in range(4):
                cse_var_3: T.int32 = i_outer_outer_j_outer_outer_fused // 128 * 8192
                    + i_outer_inner * 4096 + i_inner * 1024
```

```
            + i_outer_outer_j_outer_outer_fused % 128 * 8
        out_1 = T.Buffer((1048576,), data=out.data)
        C_1 = T.Buffer((1048576,), data=C.data)
        out_1[cse_var_3:cse_var_3 + 8] = matmul_1[i_inner]
            + C_1[cse_var_3:cse_var_3 + 8]
```

정확성 확인 및 성능 평가

바이너리를 빌드하고 정확성과 성능을 확인합니다.

```
func = tvm.build(sch, args, target)
a_np = np.random.uniform(size=(N, L)).astype(np.float32)
b_np = np.random.uniform(size=(L, M)).astype(np.float32)
c_np = np.random.uniform(size=(N, M)).astype(np.float32)
out_np = a_np.dot(b_np) + c_np

dev = tvm.cpu()
a_tvm = tvm.nd.array(a_np, device=dev)
b_tvm = tvm.nd.array(b_np, device=dev)
c_tvm = tvm.nd.array(c_np, device=dev)
out_tvm = tvm.nd.empty(out_np.shape, device=dev)
func(a_tvm, b_tvm, c_tvm, out_tvm)

# Check results
np.testing.assert_allclose(out_np, out_tvm.numpy(), rtol=1e-3)

# Evaluate execution time.
evaluator = func.time_evaluator(func.entry_name, dev, min_repeat_ms=500)
print(
    "Execution time of this operator: %.3f ms"
    % (np.median(evaluator(a_tvm, b_tvm, c_tvm, out_tvm).results) * 1000)
)
```

Out:

```
Execution time of this operator: 94.879 ms
```

레코드 파일 사용

검색하는 동안 모든 측정 기록은 **matmul.json**.파일에 기록됩니다. 측정 레코드는 검색 결과를 다시 적용하고, 검색을 다시 시작하고, 기타 분석을 수행하는 데 사용할 수 있습니다.

다음은 파일에서 최상의 Schedule을 로드하고 동등한 Python Schedule API를 출력하는 예입니다. 이는 자동 스케줄러의 동작을 디버깅하고 학습하는 데 사용할 수 있습니다.

```
print("Equivalent python schedule:")
print(task.print_best(log_file))
```

Out:

```
Equivalent python schedule:
matmul_i, matmul_j, matmul_k = tuple(matmul.op.axis) + tuple(matmul.op.reduce_axis)
out_i, out_j = tuple(out.op.axis) + tuple(out.op.reduce_axis)
matmul_i_o_i, matmul_i_i = s[matmul].split(matmul_i, factor=4)
matmul_i_o_o_i, matmul_i_o_i = s[matmul].split(matmul_i_o_i, factor=1)
matmul_i_o_o_o, matmul_i_o_o_i = s[matmul].split(matmul_i_o_o_i, factor=2)
matmul_j_o_i, matmul_j_i = s[matmul].split(matmul_j, factor=8)
matmul_j_o_o_i, matmul_j_o_i = s[matmul].split(matmul_j_o_i, factor=1)
matmul_j_o_o_o, matmul_j_o_o_i = s[matmul].split(matmul_j_o_o_i, factor=1)
matmul_k_o, matmul_k_i = s[matmul].split(matmul_k, factor=4)
s[matmul].reorder(matmul_i_o_o_o, matmul_j_o_o_o, matmul_i_o_o_i, matmul_j_o_o_i, matmul_k_o,
matmul_i_o_i, matmul_j_o_i, matmul_k_i, matmul_i_i, matmul_j_i)
out_i_o_i, out_i_i = s[out].split(out_i, factor=4)
out_i_o_o, out_i_o_i = s[out].split(out_i_o_i, factor=2)
out_j_o_i, out_j_i = s[out].split(out_j, factor=8)
out_j_o_o, out_j_o_i = s[out].split(out_j_o_i, factor=1)
s[out].reorder(out_i_o_o, out_j_o_o, out_i_o_i, out_j_o_i, out_i_i, out_j_i)
s[matmul].compute_at(s[out], out_j_o_i)
```

```
out_i_o_o_j_o_o_fused = s[out].fuse(out_i_o_o, out_j_o_o)
s[out].parallel(out_i_o_o_j_o_o_fused)
s[matmul].pragma(matmul_i_o_o_o, "auto_unroll_max_step", 8)
s[matmul].pragma(matmul_i_o_o_o, "unroll_explicit", True)
s[matmul].vectorize(matmul_j_i)
s[out].vectorize(out_j_i)
```

더 복잡한 예는 검색을 다시 시작하는 것입니다. 이 경우 검색 정책 및 비용 모델을 직접 생성하고 로그 파일을 사용하여 검색 정책 및 비용 모델의 상태를 다시 시작해야 합니다. 아래 예에서는 상태를 다시 시작하고 5회 더 시행합니다.

```
def resume_search(task, log_file):
    print("Resume search:")
    cost_model = auto_scheduler.XGBModel()
    cost_model.update_from_file(log_file)
    search_policy = auto_scheduler.SketchPolicy(
        task, cost_model, init_search_callbacks=[auto_scheduler.PreloadMeasuredStates(log_file)]
    )
    tune_option = auto_scheduler.TuningOptions(
        num_measure_trials=5, measure_callbacks=[auto_scheduler.RecordToFile(log_file)]
    )
    task.tune(tune_option, search_policy=search_policy)


resume_search(task, log_file)
```

Out:

```
Resume search:
*E
```

최종 메모 및 요약

이 튜토리얼에서는 TVM 자동 스케줄러를 사용하여 검색 템플릿을 지정할 필요 없이 행렬 곱셈을 자동으로 최적화하는 방법을 살펴보았습니다. TVM이 계산 작업을 최적화하는 방법을 보여주는 TE(Tensor Expression) 언어에서 시작하는 일련의 예제를 마칩니다.

스크립트의 총 실행 시간 : (1 분 42.985 초)

Download Python source code: auto_scheduler_matmul_x86.py

https://tvm.apache.org/docs/_downloads/eac4389b114db015e95cb3cdf8b86b83/auto_scheduler_matmul_x86.py

Download Jupyter notebook: auto_scheduler_matmul_x86.ipynb

https://tvm.apache.org/docs/_downloads/246d4b8509474fd9046e69f6cc9b7f87/auto_scheduler_matmul_x86.ipynb

1.11 TensorIR에 대한 Blitz 과정

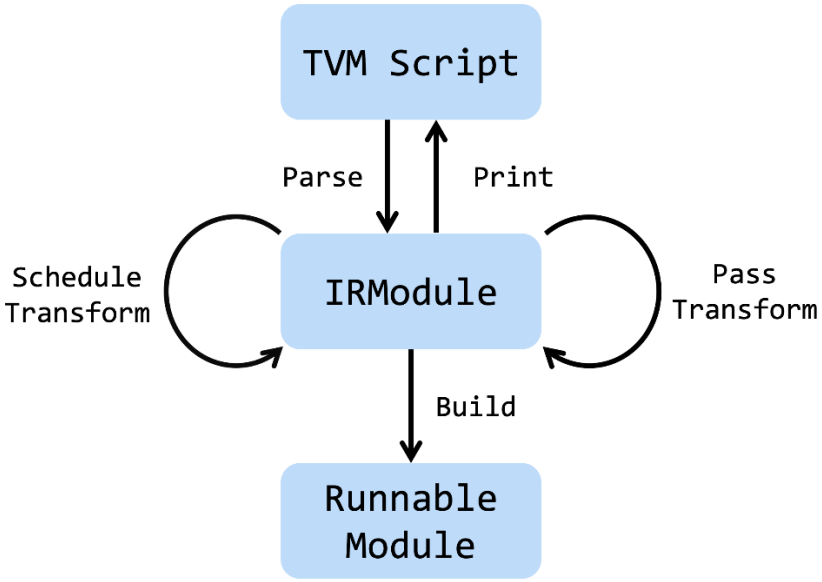
TensorIR은 크게 두 가지 목적을 수행하는 딥 러닝 프로그램을 위한 도메인 특화 언어 (Domain Specific Language, DSL)입니다.

- 다양한 하드웨어 백엔드에서 프로그램을 변환하고 최적화하기 위한 구현입니다.
- 자동 텐서화 프로그램 (automatic _tensorized_ program) 최적화를 위한 추상화.

```
import tvm
from tvm.ir.module import IRModule
from tvm.script import tir as T
import numpy as np
```

IRModule

IRModule은 딥러닝 프로그램을 포함하는 TVM의 중심 데이터 구조입니다. IR 변환 및 모델 구축의 기본 관심 대상입니다.



Life of IRModule in TensorIR

TVMScript에서 만들 수 있는 IRModule의 수명 주기입니다. TensorIR Schedule 프리미티브와 패스(Pass)는 IRModule을 변환하는 두 가지 주요 방법입니다. 또한 IRModule에 대한 일련의 변환도 허용됩니다. **모든** 단계에서 IRModule을 TVMScript로 출력할 수 있습니다. 모든 변환 및 최적화가 완료되면 IRModule을 실행 가능한 모듈로 빌드하여 대상 장치에 배포할 수 있습니다.

TensorIR 및 IRModule의 설계를 기반으로 새로운 프로그래밍 방법을 만들 수 있습니다.

- TVMScript로 python-AST 기반 구문으로 프로그램을 작성합니다.
- Python API를 사용하여 프로그램을 변환하고 최적화합니다.
- 명령형 스타일 변환 API를 사용하여 대화형으로 성능을 검사하고 사용해 봅니다.

IRModule 만들기

IRModule은 TVM IR에 대한 왕복 가능한 (round-trippable) 구문인 TVMScript를 작성하여 만들 수 있습니다.

Tensor 표현식(Tensor 표현식을 사용하여 연산자 작업)으로 계산 표현식을 생성하는 것과 달리

TensorIR을 사용하면 Python AST에 포함된 언어인 TVMScript를 통해 프로그래밍할 수 있습니다. 새로운 방법을 사용하면 복잡한 프로그램을 작성하고 Schedule 및 최적화할 수 있습니다.

https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html#tutorial-tensor-expr-get-started

다음은 벡터 덧셈에 대한 간단한 예제입니다.

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def main(a: T.handle, b: T.handle):
        # We exchange data between function by handles, which are similar to pointer.
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        # Create buffer from handles.
        A = T.match_buffer(a, (8,), dtype="float32")
        B = T.match_buffer(b, (8,), dtype="float32")
        for i in range(8):
            # A block is an abstraction for computation.
            with T.block("B"):
                # Define a spatial block iterator and bind it to value i.
                vi = T.axis.spatial(8, i)
                B[vi] = A[vi] + 1.0


ir_module = MyModule
print(type(ir_module))
print(ir_module.script())
```

Out:

```
<class 'tvm.ir.module.IRModule'>
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
```

```
    def main(A: T.Buffer((8,), "float32"), B: T.Buffer((8,), "float32")):
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i in range(8):
            with T.block("B"):
                vi = T.axis.spatial(8, i)
                T.reads(A[vi])
                T.writes(B[vi])
                B[vi] = A[vi] + T.float32(1)
```

또한 텐서 표현식 DSL을 사용하여 간단한 연산자를 작성하고 IRModule로 변환할 수도 있습니다.

```
from tvm import te

A = te.placeholder((8,), dtype="float32", name="A")
B = te.compute((8,), lambda *i: A(*i) + 1.0, name="B")
func = te.create_prim_func([A, B])
ir_module_from_te = IRModule({"main": func})
print(ir_module_from_te.script())
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((8,), "float32"), B: T.Buffer((8,), "float32")):
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i0 in range(8):
            with T.block("B"):
                v_i0 = T.axis.spatial(8, i0)
                T.reads(A[v_i0])
                T.writes(B[v_i0])
                B[v_i0] = A[v_i0] + T.float32(1)
```

IRModule 빌드 및 실행

IRModule을 특정 대상 백엔드가 있는 실행 가능한 모듈로 빌드할 수 있습니다.

```
mod = tvm.build(ir_module, target="llvm")  # The module for CPU backends.
print(type(mod))
```

Out:

```
<class 'tvm.driver.build_module.OperatorModule'>
```

입력 배열과 출력 배열을 준비한 다음 모듈을 실행합니다.

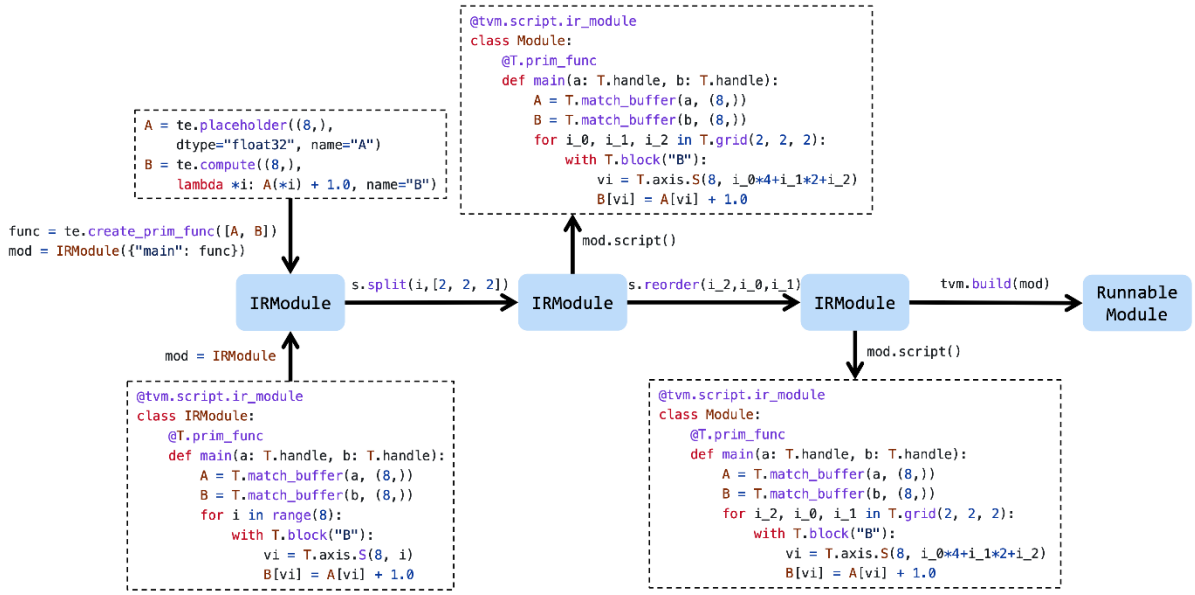
```
a = tvm.nd.array(np.arange(8).astype("float32"))
b = tvm.nd.array(np.zeros((8,)).astype("float32"))
mod(a, b)
print(a)
print(b)
```

Out:

```
[0. 1. 2. 3. 4. 5. 6. 7.]
[1. 2. 3. 4. 5. 6. 7. 8.]
```

IRModule 변환

IRModule은 프로그램 최적화를 위한 중앙 데이터 구조로 Schedule에 의해 변환될 수 있습니다. Schedule에는 프로그램을 대화형으로 변환하는 여러 기본 메서드가 포함되어 있습니다. 각 프리미티브는 특정 방식으로 프로그램을 변환하여 추가 성능 최적화를 제공합니다.



TensorIR interactive optimization flow

위의 이미지는 텐서 프로그램을 최적화하기 위한 일반적인 워크플로입니다. 먼저 TVMScript 또는 Tensor Expression에서 생성된 초기 IRModule에 대한 Schedule을 만들어야 합니다. 그런 다음 Schedule 기본 형식의 시퀀스가 성능을 향상시키는 데 도움이 됩니다. 그리고 마침내 우리는 이를 실행 가능한 모듈로 낮춰서 빌드할 수 있습니다.

여기서는 매우 간단한 변환을 보여 줍니다. 먼저 ir_module 입력에 대한 Schedule을 만듭니다.

```
sch = tvm.tir.Schedule(ir_module)
print(type(sch))
```

Out:

```
<class 'tvm.tir.schedule.schedule.Schedule'>
```

루프를 3개의 루프로 바둑판식으로 배열하고 결과를 출력합니다.

```
# Get block by its name
block_b = sch.get_block("B")
# Get loops surrounding the block
(i,) = sch.get_loops(block_b)
```

```
# Tile the loop nesting.
i_0, i_1, i_2 = sch.split(i, factors=[2, 2, 2])
print(sch.mod.script())
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((8,), "float32"), B: T.Buffer((8,), "float32")):
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i_0, i_1, i_2 in T.grid(2, 2, 2):
            with T.block("B"):
                vi = T.axis.spatial(8, i_0 * 4 + i_1 * 2 + i_2)
                T.reads(A[vi])
                T.writes(B[vi])
                B[vi] = A[vi] + T.float32(1)
```

루프를 재정렬할 수도 있습니다. 이제 루프 i_2 를 i_1 외부로 이동합니다.

```
sch.reorder(i_0, i_2, i_1)
print(sch.mod.script())
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((8,), "float32"), B: T.Buffer((8,), "float32")):
        T.func_attr({"tir.noalias": T.bool(True)})
```

```
        # with T.block("root"):
        for i_0, i_2, i_1 in T.grid(2, 2, 2):
            with T.block("B"):
                vi = T.axis.spatial(8, i_0 * 4 + i_1 * 2 + i_2)
                T.reads(A[vi])
                T.writes(B[vi])
                B[vi] = A[vi] + T.float32(1)
```

GPU 프로그램으로 변환

GPU에 모델을 배포하려면 스레드 바인딩이 필요합니다. 다행히도 프리미티브를 사용하고 점진적으로 변환할 수도 있습니다.

```
sch.bind(i_0, "blockIdx.x")
sch.bind(i_2, "threadIdx.x")
print(sch.mod.script())
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((8,), "float32"), B: T.Buffer((8,), "float32")):
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i_0 in T.thread_binding(2, thread="blockIdx.x"):
            for i_2 in T.thread_binding(2, thread="threadIdx.x"):
                for i_1 in range(2):
                    with T.block("B"):
                        vi = T.axis.spatial(8, i_0 * 4 + i_1 * 2 + i_2)
                        T.reads(A[vi])
```

```
                T.writes(B[vi])
                B[vi] = A[vi] + T.float32(1)
```

스레드를 바인딩한 후 이제 **CUDA** 백엔드를 사용하여 IRModule을 빌드합니다.

```
ctx = tvm.cuda(0)
cuda_mod = tvm.build(sch.mod, target="cuda")
cuda_a = tvm.nd.array(np.arange(8).astype("float32"), ctx)
cuda_b = tvm.nd.array(np.zeros((8,)).astype("float32"), ctx)
cuda_mod(cuda_a, cuda_b)
print(cuda_a)
print(cuda_b)
```

Out:

```
[0. 1. 2. 3. 4. 5. 6. 7.]
[1. 2. 3. 4. 5. 6. 7. 8.]
```

Download Python source code: tensor_ir_blitz_course.py

https://tvm.apache.org/docs/_downloads/5c7000b5aef924e29ec975ec3002ea03/tensor_ir_blitz_course.py

Download Jupyter notebook: tensor_ir_blitz_course.ipynb

https://tvm.apache.org/docs/_downloads/c9bb7875c6ca5b2da162e177d3c9aac0/tensor_ir_blitz_course.ipynb

1.12 크로스 컴파일과 RPC

이 가이드에서는 TVM에서 RPC를 사용한 크로스 컴파일 (Cross Compilation) 및 원격 디바이스 실행을 소개합니다.

크로스 컴파일 및 RPC를 사용하면 **로컬 컴퓨터에서 프로그램을 컴파일한 다음 원격 디바이스에서 실행**할 수 있습니다. Raspberry Pi 및 모바일 플랫폼과 같이 원격 디바이스 리소스가 제한된 경우에 유용합니다. 이 튜토리얼에서는 CPU 예제에 Raspberry Pi를 사용하고 OpenCL 예제에 Firefly-RK3399를 사용합니다.

디바이스에서 TVM 런타임 빌드

첫 번째 단계는 원격 디바이스에서 TVM 런타임을 빌드하는 것입니다.

Note

이 섹션과 다음 섹션의 모든 지침은 대상 디바이스(예: Raspberry Pi)에서 실행되어야 합니다. 대상이 Linux를 실행 중이라고 가정합니다.

로컬 컴퓨터에서 컴파일을 수행하기 때문에 원격 디바이스는 생성된 코드를 실행하는 데만 사용됩니다. 원격 디바이스에서 TVM 런타임을 빌드하기만 하면 됩니다.

```
git clone --recursive https://github.com/apache/tvm tvm
cd tvm
make runtime -j2
```

런타임을 성공적으로 빌드한 후 **~/.bashrc** 파일에 환경 변수를 설정해야 합니다. **vi ~/.bashrc** 사용하여 **~/.bashrc** 를 편집하고 아래 줄을 추가 할 수 있습니다 (TVM 디렉토리가 **~/tvm**에 있다고 가정).

```
export PYTHONPATH=$PYTHONPATH:~/tvm/python
```

환경 변수를 업데이트하려면 **source ~/.bashrc**를 실행합니다.

디바이스에서 RPC 서버 설정

RPC 서버를 시작하려면 원격 디바이스(이 예제에서는 Raspberry Pi)에서 다음 명령을 실행합니다.

```
python -m tvm.exec.rpc_server --host 0.0.0.0 --port=9090
```

아래 줄이 보이면 RPC 서버가 디바이스에서 성공적으로 시작되었음을 의미합니다.

```
INFO:root:RPCServer: bind to 0.0.0.0:9090
```

로컬 머신에서 커널 선언 및 크로스 컴파일

Note

이제 전체 TVM(LLVM 포함)이 설치된 로컬 컴퓨터로 돌아갑니다.

여기서 우리는 로컬 컴퓨터에서 간단한 커널을 선언 할 것입니다.

```
import numpy as np

import tvm
from tvm import te
from tvm import rpc
from tvm.contrib import utils
```

```
n = tvm.runtime.convert(1024)
A = te.placeholder((n,), name="A")
B = te.compute((n,), lambda i: A[i] + 1.0, name="B")
s = te.create_schedule(B.op)
```

그런 다음 커널을 크로스 컴파일합니다. Raspberry Pi 3B의 경우 대상은 **'llvm -mtriple=armv7l-linux-gnueabihf'**여야 하지만 여기서는 'llvm'을 사용하여 웹 페이지 빌드 서버에서 이 튜토리얼을 실행할 수 있도록 합니다. 다음 블록의 자세한 참고 사항을 참조하세요.

```
local_demo = True

if local_demo:
    target = "llvm"
else:
    target = "llvm -mtriple=armv7l-linux-gnueabihf"

func = tvm.build(s, [A, B], target=target, name="add_one")
# save the lib at a local temp folder
temp = utils.tempdir()
path = temp.relpath("lib.tar")
func.export_library(path)
```

Note

실제 원격 디바이스에서 이 튜토리얼을 실행하려면 **local_demo**를 False로 변경하고 **빌드의 target**을 디바이스에 적합한 대상 트리플 (triple)로 바꿉니다. 장치마다 다를 수 있는 대상 트리플입니다. 예를 들어 Raspberry Pi 3B의 경우 **'llvm -mtriple=armv7l-linux-gnueabihf'**이고 RK3399의 경우 **'llvm -mtriple=aarch64-linux-gnu'**입니다.

일반적으로 장치에서 **gcc -v** 를 실행하고 **Target:** (여전히 느슨한 구성 일 수 있음)으로 시작하는 줄을 찾아 대상을 쿼리 할 수 있습니다.

-mtriple 외에도 다음과 같은 다른 컴파일 옵션을 설정할 수도 있습니다.

- -mcpu=<cpuname>

    현재 아키텍처에서 코드를 생성할 특정 칩을 지정합니다. 기본적으로 이는 대상 트리플에서 유추되고 현재 아키텍처로 자동 감지됩니다.

- -mattr=a1,+a2,-a3,...

  대상의 특정 특성(예: SIMD 작업 사용 여부)을 재정의하거나 제어합니다. 기본 속성 집합은 현재 CPU에 의해 설정됩니다. 사용 가능한 속성 목록을 얻으려면 다음을 수행 할 수 있습니다.

```
llc -mtriple=<your device target triple> -mattr=help
```

이러한 옵션은 llc와 일치합니다. 사용 가능한 특정 기능을 포함하도록 대상 트리플 및 기능 세트를 설정하는 것이 좋습니다., 보드의 기능을 최대한 활용할 수 있도록. 크로스 컴파일 속성에 대한 자세한 내용은 LLVM 크로스 컴파일 가이드에서 확인할 수 있습니다.

https://clang.llvm.org/docs/CrossCompilation.html

RPC를 통해 원격으로 CPU 커널 실행

원격 장치에서 생성된 CPU 커널을 실행하는 방법을 보여줍니다. 먼저 원격 장치에서 RPC 세션을 가져옵니다.

```
if local_demo:
    remote = rpc.LocalSession()
else:
    # The following is my environment, change this to the IP address of your target device
    host = "10.77.1.162"
    port = 9090
    remote = rpc.connect(host, port)
```

lib를 원격 장치에 업로드한 다음 디바이스 로컬 컴파일러를 호출하여 다시 연결합니다. 이제 **func**는 원격 모듈 객체입니다.

```
remote.upload(path)
func = remote.load_module("lib.tar")
```

```
# create arrays on the remote device
dev = remote.cpu()
a = tvm.nd.array(np.random.uniform(size=1024).astype(A.dtype), dev)
b = tvm.nd.array(np.zeros(1024, dtype=A.dtype), dev)
# the function will run on the remote device
func(a, b)
np.testing.assert_equal(b.numpy(), a.numpy() + 1)
```

원격 디바이스에서 커널의 성능을 평가하려면 네트워크 오버헤드를 피하는 것이 중요합니다. **time_evaluator** 함수는 함수를 여러 번 실행하고, 원격 디바이스에서 실행당 비용을 측정하고, 측정된 비용을 반환하는 원격 함수를 반환합니다. 네트워크 오버헤드는 제외됩니다.

```
time_f = func.time_evaluator(func.entry_name, dev, number=10)
cost = time_f(a, b).mean
print("%g secs/op" % cost)
```

Out:

```
1.176e-07 secs/op
```

RPC를 통해 원격으로 OpenCL 커널 실행

원격 OpenCL 장치의 경우 워크플로는 위와 거의 동일합니다. 커널을 정의하고, 파일을 업로드하고, RPC를 통해 실행할 수 있습니다.

Note

Raspberry Pi는 OpenCL을 지원하지 않으며 다음 코드는 Firefly-RK3399에서 테스트되었습니다. 이 튜토리얼에 따라 RK3399용 OS 및 OpenCL 드라이버를 설정할 수 있습니다.

https://gist.github.com/mli/585aed2cec0b5178b1a510f9f236afa2

또한 rk3399 보드에서 OpenCL이 활성화된 런타임을 빌드해야 합니다. TVM 루트 디렉터리에서

다음을 실행합니다.

```
cp cmake/config.cmake .
sed -i "s/USE_OPENCL OFF/USE_OPENCL ON/" config.cmake
make runtime -j4
```

다음 함수는 OpenCL 커널을 원격으로 실행하는 방법을 보여줍니다

```
def run_opencl():
    # NOTE: This is the setting for my rk3399 board. You need to modify
    # them according to your environment.
    opencl_device_host = "10.77.1.145"
    opencl_device_port = 9090
    target = tvm.target.Target("opencl", host="llvm -mtriple=aarch64-linux-gnu")

    # create schedule for the above "add one" compute declaration
    s = te.create_schedule(B.op)
    xo, xi = s[B].split(B.op.axis[0], factor=32)
    s[B].bind(xo, te.thread_axis("blockIdx.x"))
    s[B].bind(xi, te.thread_axis("threadIdx.x"))
    func = tvm.build(s, [A, B], target=target)

    remote = rpc.connect(opencl_device_host, opencl_device_port)

    # export and upload
    path = temp.relpath("lib_cl.tar")
    func.export_library(path)
    remote.upload(path)
    func = remote.load_module("lib_cl.tar")

    # run
    dev = remote.cl()
    a = tvm.nd.array(np.random.uniform(size=1024).astype(A.dtype), dev)
    b = tvm.nd.array(np.zeros(1024, dtype=A.dtype), dev)
    func(a, b)
    np.testing.assert_equal(b.numpy(), a.numpy() + 1)
    print("OpenCL test passed!")
```

요약

이 튜토리얼에서는 TVM의 크로스 컴파일 및 RPC 기능에 대한 자세한 내용을 제공합니다.

- 원격 디바이스에 RPC 서버를 설정합니다.
- 로컬 머신에서 커널을 크로스 컴파일하도록 타겟 디바이스 구성을 설정합니다.
- RPC API를 통해 원격으로 커널을 업로드하고 실행합니다.

Download Python source code: cross_compilation_and_rpc.py

https://tvm.apache.org/docs/_downloads/766206ab8f1fd80ac34d9816cb991a0d/cross_compilation_and_rpc.py

Download Jupyter notebook: cross_compilation_and_rpc.ipynb

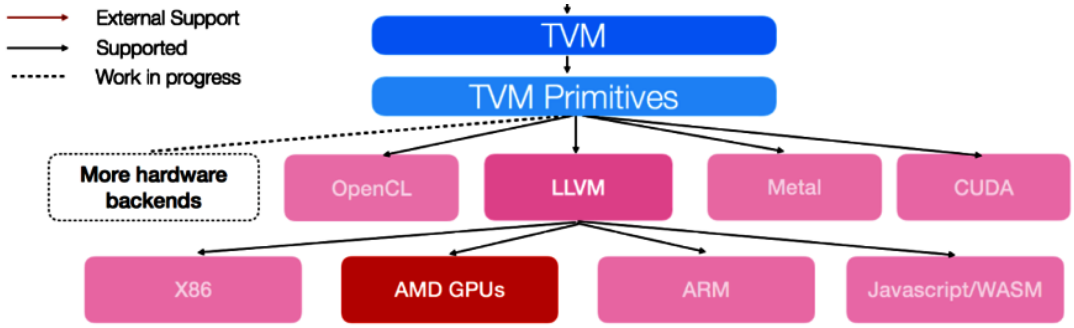
https://tvm.apache.org/docs/_downloads/f289ca2466fcf79c024068c1f8642bd0/cross_compilation_and_rpc.ipynb

1.13 딥러닝 모델 컴파일을 위한 빠른 시작 튜토리얼

이 예제에서는 Relay Python 프런트엔드를 사용하여 신경망을 빌드하고 TVM을 사용하여 NVIDIA GPU용 런타임 라이브러리를 생성하는 방법을 보여줍니다. CUDA 및 LLVM을 사용하도록 설정하여 TVM을 빌드해야 합니다.

TVM의 지원되는 하드웨어 백엔드 개요

아래 이미지는 현재 TVM에서 지원하는 하드웨어 백엔드를 보여줍니다.



이 튜토리얼에서는 CUDA 및 LLVM을 대상 백엔드로 선택합니다. 먼저 Relay와 TVM을 가져 오겠습니다.

```
import numpy as np

from tvm import relay
from tvm.relay import testing
import tvm
from tvm import te
from tvm.contrib import graph_executor
import tvm.testing
```

Relay에서 신경망 정의하기

먼저 릴레이 Python 프론트엔드가 있는 신경망을 정의하겠습니다. 간단히 하기 위해 Relay에서 미리 정의된 resnet-18 네트워크를 사용합니다. 매개변수는 Xavier initializer (weight initialization 방법) 로 초기화됩니다. Relay는 MXNet, CoreML, ONNX 및 Tensorflow와 같은 다른 모델 형식도 지원합니다.

이 튜토리얼에서는 디바이스에서 추론을 수행하고 배치 크기가 1로 설정되어 있다고 가정합니다. 입력 영상은 224 * 224 크기의 RGB 컬러 영상입니다. tvm.relay.expr.TupleWrapper.astext()를 호출하여 네트워크 구조를 표시할 수 있습니다.

```
batch_size = 1
num_class = 1000
image_shape = (3, 224, 224)
data_shape = (batch_size,) + image_shape
out_shape = (batch_size, num_class)

mod, params = relay.testing.resnet.get_workload(
      num_layers=18, batch_size=batch_size, image_shape=image_shape
)

# set show_meta_data=True if you want to show meta data
print(mod.astext(show_meta_data=False))
```

Out:

```
#[version = "0.0.5"]
def @main(%data: Tensor[(1, 3, 224, 224), float32] /* ty=Tensor[(1, 3, 224, 224), float32] */,
%bn_data_gamma: Tensor[(3), float32] /* ty=Tensor[(3), float32] */,
%bn_data_beta: Tensor[(3), float32] /* ty=Tensor[(3), float32] */,
...
%fc1_bias: Tensor[(1000), float32] /* ty=Tensor[(1000), float32] */) -> Tensor[(1, 1000), float32] {
  %0 = nn.batch_norm(%data, %bn_data_gamma, %bn_data_beta,
  %bn_data_moving_mean, %bn_data_moving_var, epsilon=2e-05f, scale=False)
  /* ty=(Tensor[(1, 3, 224, 224), float32], Tensor[(3), float32], Tensor[(3), float32]) */;
```

```
  %1 = %0.0 /* ty=Tensor[(1, 3, 224, 224), float32] */;

  %2 = nn.conv2d(%1, %conv0_weight, strides=[2, 2], padding=[3, 3, 3, 3],
  channels=64, kernel_size=[7, 7])
  /* ty=Tensor[(1, 64, 112, 112), float32] */;

  %3 = nn.batch_norm(%2, %bn0_gamma, %bn0_beta, %bn0_moving_mean,
  %bn0_moving_var, epsilon=2e-05f)
  /* ty=(Tensor[(1, 64, 112, 112), float32], Tensor[(64), float32], Tensor[(64), float32]) */;

  %4 = %3.0 /* ty=Tensor[(1, 64, 112, 112), float32] */;
  %5 = nn.relu(%4) /* ty=Tensor[(1, 64, 112, 112), float32] */;

  %6 = nn.max_pool2d(%5, pool_size=[3, 3], strides=[2, 2], padding=[1, 1, 1, 1])
  /* ty=Tensor[(1, 64, 56, 56), float32] */;

  %7 = nn.batch_norm(%6, %stage1_unit1_bn1_gamma, %stage1_unit1_bn1_beta,
  %stage1_unit1_bn1_moving_mean, %stage1_unit1_bn1_moving_var, epsilon=2e-05f)
  /* ty=(Tensor[(1, 64, 56, 56), float32], Tensor[(64), float32], Tensor[(64), float32]) */;

  %8 = %7.0 /* ty=Tensor[(1, 64, 56, 56), float32] */;
  %9 = nn.relu(%8) /* ty=Tensor[(1, 64, 56, 56), float32] */;

  %10 = nn.conv2d(%9, %stage1_unit1_conv1_weight,
  padding=[1, 1, 1, 1], channels=64, kernel_size=[3, 3])
  /* ty=Tensor[(1, 64, 56, 56), float32] */;
...
  %88 = nn.dense(%87, %fc1_weight, units=1000) /* ty=Tensor[(1, 1000), float32] */;
  %89 = nn.bias_add(%88, %fc1_bias, axis=-1) /* ty=Tensor[(1, 1000), float32] */;
  nn.softmax(%89) /* ty=Tensor[(1, 1000), float32] */
}
```

컴파일

다음 단계는 Relay/TVM 파이프라인을 사용하여 모델을 컴파일하는 것입니다. 사용자는 컴파일의

최적화 수준을 지정할 수 있습니다. 현재 이 값은 0에서 3 사이일 수 있습니다. 최적화 단계에는 연산자 융합, 사전 계산, 레이아웃 변환 등이 포함됩니다.

relay.build()는 JSON 형식의 실행 그래프, 대상 하드웨어에서 이 그래프를 위해 특별히 컴파일된 함수의 TVM 모듈 라이브러리, 모델의 매개 변수 blob의 세 가지 구성 요소를 반환합니다. 컴파일하는 동안 Relay는 그래프 수준 최적화를 수행하고 TVM은 텐서 수준 최적화를 수행하므로 모델 제공에 최적화된 런타임 모듈이 생성됩니다.

먼저 NVIDIA GPU용으로 컴파일합니다. 그 뒤에서 **relay.build()**는 먼저 pruning, fusing, 등과 같은 여러 그래프 수준 최적화를 수행한 다음 연산자(즉, 최적화된 그래프의 노드)를 TVM 구현에 등록하여 **tvm.module**을 생성합니다. 모듈 라이브러리를 생성하기 위해 TVM은 먼저 높은 수준의 IR을 지정된 대상 백엔드의 하위 내장 IR(이 예에서는 CUDA)로 전송합니다. 그러면 기계어가 모듈 라이브러리로 생성됩니다.

```
opt_level = 3
target = tvm.target.cuda()
with tvm.transform.PassContext(opt_level=opt_level):
    lib = relay.build(mod, target, params=params)
```

Out:

```
/workspace/python/tvm/target/target.py:446: UserWarning: Try specifying cuda arch by adding 'arch=sm_xx' to your target.
  warnings.warn("Try specifying cuda arch by adding 'arch=sm_xx' to your target.")
```

라이브러리 생성 실행

이제 그래프 실행기를 만들고 NVIDIA GPU에서 모듈을 실행할 수 있습니다.

```
# create random input
```

```
dev = tvm.cuda()
data = np.random.uniform(-1, 1, size=data_shape).astype("float32")
# create module
module = graph_executor.GraphModule(lib["default"](dev))
# set input and parameters
module.set_input("data", data)
# run
module.run()
# get output
out = module.get_output(0, tvm.nd.empty(out_shape)).numpy()

# Print first 10 elements of output
print(out.flatten()[0:10])
```

Out:

```
[0.00089283 0.00103331 0.0009094  0.00102275 0.00108751 0.00106737
 0.00106262 0.00095838 0.00110792 0.00113151]
```

컴파일된 모듈 저장 및 로드(Save and Load Compiled Module)

또한 graph, lib 및 매개 변수를 파일에 저장하고 배포 환경에서 다시 로드할 수 있습니다.

```
# save the graph, lib and params into separate files
from tvm.contrib import utils

temp = utils.tempdir()
path_lib = temp.relpath("deploy_lib.tar")
lib.export_library(path_lib)
print(temp.listdir())
```

Out:

```
['deploy_lib.tar']
```

```
# load the module back.
loaded_lib = tvm.runtime.load_module(path_lib)
input_data = tvm.nd.array(data)

module = graph_executor.GraphModule(loaded_lib["default"](dev))
module.run(data=input_data)
out_deploy = module.get_output(0).numpy()

# Print first 10 elements of output
print(out_deploy.flatten()[0:10])

# check whether the output from deployed module is consistent with original one
tvm.testing.assert_allclose(out_deploy, out, atol=1e-5)
```

Out:

```
[0.00089283 0.00103331 0.0009094  0.00102275 0.00108751 0.00106737
 0.00106262 0.00095838 0.00110792 0.00113151]
```

Download Python source code: relay_quick_start.py

https://tvm.apache.org/docs/_downloads/cc6d9aebd24d54d81752590cbc8f99f9/relay_quick_start.py

Download Jupyter notebook: relay_quick_start.ipynb

https://tvm.apache.org/docs/_downloads/3dd2108354ac3028c96bcd6a0c7899dd/relay_quick_start.ipynb

1.14 UMA를 사용하여 하드웨어 가속기를 TVM-ready로 만들기

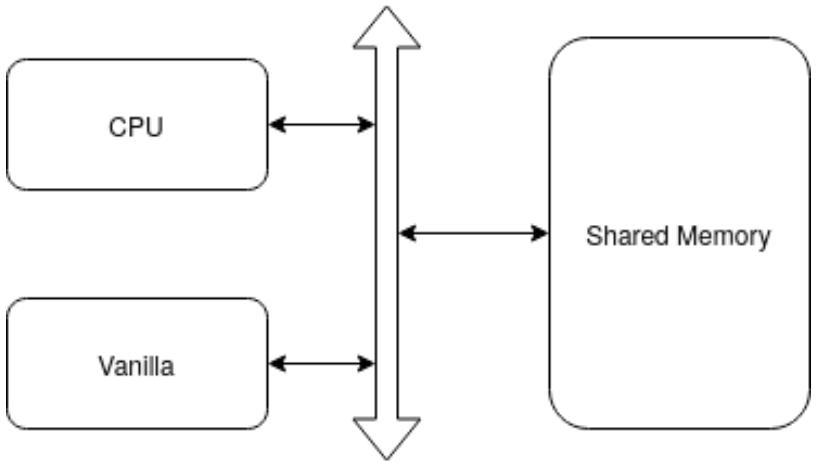
UMA(Universal Modular Accelerator Interface)에 대한 입문 튜토리얼입니다. UMA는 새로운 하드웨어 가속기를 TVM에 통합하기 위한 사용하기 쉬운 API를 제공합니다.

이 튜토리얼에서는 UMA를 사용하여 하드웨어 가속기를 TVM을 지원하는 방법을 단계별로 설명합니다. 이 문제에 대한 만능 솔루션은 없지만 UMA는 여러 하드웨어 가속기 클래스를 TVM에 통합하기 위해 안정적인 Python 전용 API를 제공하는 것을 목표로 합니다.

이 튜토리얼에서는 복잡성이 증가하는 세 가지 사용 사례에서 UMA API에 대해 알아봅니다. 이러한 사용 사례에서는 세 가지 모의 가속기인 **Vanilla**, **Strawberry** 및 **Chocolate**이 도입되고 UMA를 사용하여 TVM에 통합됩니다.

Vanilla

Vanilla는 MAC 어레이로 구성된 간단한 가속기이며 내부 메모리가 없습니다. Conv2D 레이어 만 처리 할 수 있으며 다른 모든 레이어는 CPU에서 실행되며 Vanilla도 조율합니다. CPU와 Vanilla 모두 공유 메모리를 사용합니다.



Vanilla에는 Conv2D 작업 (동일 패딩 포함)을 수행하기위한 C 인터페이스 **vanilla_conv2dnchw(...)**가 있으며 입력 기능 맵, 가중치 및 결과와 **Conv2D (oc, iw, ih, ic, kh, kw**)의 크기에 대한 포인터를 허용합니다.

```
int vanilla_conv2dnchw(float* ifmap, float*  weights, float*  result, int oc, int iw, int ih, int ic, int kh, int kw);
```

스크립트 **uma_cli**는 새 액셀러레이터를 위해 UMA-API에 대한 API 호출을 사용하여 코드 스켈레톤(skeletons)을 만듭니다.

Vanilla의 경우 다음과 같이 사용합니다. (**--tutorial vanilla**는 이 튜토리얼 부분에 필요한 모든 추가 파일을 추가합니다)

```
pip install inflection
cd $TVM_HOME/apps/uma
python uma_cli.py --add_hardware vanilla_accelerator --tutorial vanilla
```

uma_cli.py 는 **vanilla_accelerator** 디렉토리에 이러한 파일을 생성합니다.

```
backend.py  codegen.py  conv2dnchw.cc  passes.py  patterns.py  run.py  strategies.py
```

Vanilla 백엔드

Vanilla에 대해 생성된 백엔드는 vanilla_accelerator/backend.py 에서 찾을 수 있습니다.

```
class VanillaAcceleratorBackend(UMABackend):
    """UMA backend for VanillaAccelerator."""

    def __init__(self):
        super().__init__()

        self._register_pattern("conv2d", conv2d_pattern())
        self._register_tir_pass(PassPhase.TIR_PHASE_0, VanillaAcceleratorConv2DPass())
        self._register_codegen(fmt="c", includes=gen_includes)
```

```
    @property
    def target_name(self):
        return "vanilla_accelerator"
```

오프로드된 패턴 정의

Conv2D가 Vanilla로 오프로드되도록 지정하기 위해 **vanilla_accelerator/patterns.py** 에서 릴레이 데이터플로우 패턴(**DFPattern**: https://tvm.apache.org/docs/reference/langref/relay_pattern.html)으로 설명됩니다.

```
def conv2d_pattern():
    pattern = is_op("nn.conv2d")(wildcard(), wildcard())
    pattern = pattern.has_attr({"strides": [1, 1]})
    return pattern
```

입력 그래프의 Conv2D 작업을 Vanilla의 저수준 함수 호출 **vanilla_conv2dnchw(...)** 에 매핑하기 위해 TIR 패스 **VanillaAcceleratorConv2DPass**(이 튜토리얼의 뒷부분에서 설명)가 **VanillaAcceleratorBackend**에 등록됩니다.

Codegen

vanilla_accelerator/codegen.py 파일은 정적 C-Code (C language code)를 정의하고 **gen_includes**에서 TVM's C-Codegen에 의해 생성된 결과 C-Code에 추가됩니다. 여기에 바닐라의 저수준 라이브러리 ''**vanilla_conv2dnchw()**''를 포함하도록 C-Code 가 추가됩니다.

```
def gen_includes() -> str:
    topdir = pathlib.Path(__file__).parent.absolute()

    includes = ""
    includes += f'#include "{topdir}/conv2dnchw.cc"'
    return includes
```

위의 **VanillaAcceleratorBackend**에서 볼 수 있듯이 **self._register_codegen** 을 사용하여 UMA에 등록됩니다.

```
self._register_codegen(fmt="c", includes=gen_includes)
```

신경망 빌드 및 Vanilla에서 실행

UMA의 기능을 시연하기 위해 단일 Conv2D 레이어에 대한 C-Code 를 생성하고 Vanilla 가속기에서 실행합니다. **vanilla_accelerator/run.py** 파일은 Vanilla의 C-API를 사용하여 Conv2D 레이어를 실행하는 데모를 제공합니다.

```
def main():
    mod, inputs, output_list, runner = create_conv2d()

    uma_backend = VanillaAcceleratorBackend()
    uma_backend.register()
    mod = uma_backend.partition(mod)
    target = tvm.target.Target("vanilla_accelerator", host=tvm.target.Target("c"))

    export_directory = tvm.contrib.utils.tempdir(keep_for_debug=True).path
    print(f"Generated files are in {export_directory}")
    compile_and_run(
        AOTModel(module=mod, inputs=inputs, outputs=output_list),
        runner,
        interface_api="c",
        use_unpacked_api=True,
        target=target,
        test_dir=str(export_directory),
    )


main()
```

vanilla_accelerator/run.py 를 실행하면 출력 파일이 MLF (Model Library Format)으로 생성됩니다.

Output:

```
Generated files are in /tmp/tvm-debug-mode-tempdirs/2022-07-13T13-26-22___x5u76h0p/00000
```

생성된 파일을 살펴보겠습니다.

Output:

```
cd /tmp/tvm-debug-mode-tempdirs/2022-07-13T13-26-22___x5u76h0p/00000
cd build/
ls -1

codegen
lib.tar
metadata.json
parameters
runtime
src
```

생성된 C-Code 를 평가하려면 **codegen/host/src/default_lib2.c**로 이동하세요.

```
cd codegen/host/src/
ls -1

default_lib0.c
default_lib1.c
default_lib2.c
```

default_lib2.c에서 생성 된 코드가 Vanilla의 C-API를 호출하고 Conv2D 레이어를 실행하는 것을 볼 수 있습니다.

```
TVM_DLL int32_t tvmgen_default_vanilla_accelerator_main_0(float* placeholder, float* placeholder1, float* conv2d_nchw, uint8_t* global_workspace_1_var) {
    vanilla_accelerator_conv2dnchw(placeholder, placeholder1, conv2d_nchw, 32, 14, 14, 32, 3, 3);
    return 0;
}
```

Strawberry

Coming soon ...

Chocolate

Coming soon ...

커뮤니티 의견 요청

이 튜토리얼이 액셀러레이터에 **맞지 않는 경우** TVM 토론 포럼의 UMA 스레드에 요구 사항을 추가합니다. UMA 인터페이스를 사용하여 추가 클래스의 AI 하드웨어 가속기를 TVM에서 사용할 수 있도록 하는 방법에 대한 지침을 제공하기 위해 이 튜토리얼을 확장하고자 합니다.

References

[UMA-RFC] UMA: Universal Modular Accelerator Interface, TVM RFC, June 2022.

https://github.com/apache/tvm-rfcs/blob/main/rfcs/0060_UMA_Unified_Modular_Accelerator_Interface.md

[DFPattern] Pattern Matching in Relay

https://tvm.apache.org/docs/reference/langref/relay_pattern.html

Download Python source code: uma.py

https://tvm.apache.org/docs/_downloads/f9c6910c7b4a120c51a9bf48f34f3ad7/uma.py

Download Jupyter notebook: uma.ipynb

https://tvm.apache.org/docs/_downloads/6e0673ce1f08636c34d0b9a73ea114f7/uma.ipynb

1.15 TOPI 소개

TOPI(TVM Operator Inventory)에 대한 소개 튜토리얼입니다. TOPI는 TVM보다 추상화가 높은 numpy 스타일의 제네릭 연산 및 Schedule을 제공합니다. 이 튜토리얼에서는 TOPI가 TVM에서 보일러플레이트 코드 (boilerplate code, 상용구 코드)를 작성하지 않아도 되는 방법을 살펴보겠습니다.

```
import tvm
import tvm.testing
from tvm import te
from tvm import topi
import numpy as np
```

기본 예제

행의 합 연산을 다시 살펴 보겠습니다 (**B = numpy.sum(A, axis=1)**'과 동일) 2차원 TVM 텐서 A의 행 합을 계산하려면 다음과 같이 기호 연산과 Schedule을 지정해야 합니다

```
n = te.var("n")
m = te.var("m")
A = te.placeholder((n, m), name="A")
k = te.reduce_axis((0, m), "k")
B = te.compute((n,), lambda i: te.sum(A[i, k], axis=k), name="B")
s = te.create_schedule(B.op)
```

사람이 읽을 수 있는 형식으로 IR 코드를 검사하기 위해 다음을 수행할 수 있습니다.

```
print(tvm.lower(s, [A], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T
```

```
@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.handle):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        n, m = T.int32(), T.int32()
        A_1 = T.match_buffer(A, (n, m), strides=("stride", "stride"), buffer_type="auto")
        B = T.allocate([n], "float32", "global")
        for i in range(n):
            B_1 = T.Buffer((n,), data=B)
            B_1[i] = T.float32(0)
            for k in range(m):
                A_2 = T.Buffer((A_1.strides[0] * n,), data=A_1.data, buffer_type="auto")
                B_1[i] = B_1[i] + A_2[i * A_1.strides[0] + k * A_1.strides[1]]
```

그러나 이러한 일반적인 연산을 위해 우리는 축소 축을 직접 정의해야할뿐만 아니라 **te.compute**를 사용하여 명시적 계산을 해야했습니다. 더 복잡한 작업을 위해 얼마나 많은 세부 정보를 제공해야 하는지 상상해 보세요. 다행히도 이 두 줄을 **numpy.sum**과 마찬가지로 간단하게 **topi.sum**으로 바꿀 수 있습니다

```
C = topi.sum(A, axis=1)
ts = te.create_schedule(C.op)
print(tvm.lower(ts, [A], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.handle):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        n, m = T.int32(), T.int32()
        A_1 = T.match_buffer(A, (n, m), strides=("stride", "stride"), buffer_type="auto")
```

```
        A_red = T.allocate([n], "float32", "global")
        for ax0 in range(n):
            A_red_1 = T.Buffer((n,), data=A_red)
            A_red_1[ax0] = T.float32(0)
            for k1 in range(m):
                A_2 = T.Buffer((A_1.strides[0] * n,), data=A_1.data, buffer_type="auto")
                A_red_1[ax0] = A_red_1[ax0] + A_2[ax0 * A_1.strides[0] + k1 * A_1.strides[1]]
```

Numpy 스타일 연산자 오버로딩

topi.broadcast_add 사용하여 올바른 (특정하게 브로드캐스팅 가능한) shapes 를 갖는 두 개의 텐서를 추가할 수 있습니다. 더 짧은 TOPI는 이러한 일반적인 작업에 대한 연산자 오버로딩을 제공합니다. 예를 들어,

```
x, y = 100, 10
a = te.placeholder((x, y, y), name="a")
b = te.placeholder((y, y), name="b")
c = a + b  # same as topi.broadcast_add
d = a * b  # same as topi.broadcast_mul
```

동일한 구문으로 오버로드된 TOPI는 기본 형식(int, float)을 텐서 **d - 3.14**로 브로드캐스트하는 것을 처리합니다.

제네릭 Schedule 및 융합 작업

지금까지 우리는 TOPI가 하위 수준 API에서 명시적 계산을 작성하지 않도록 하는 방법에 대한 예를 살펴보았습니다. 하지만 여기서 끝이 아닙니다. 그래도 우리는 이전과 같이 Schedule을 잡았습니다. 또한 TOPI는 주어진 컨텍스트에 따라 더 높은 수준의 스케줄링 레시피를 제공합니다. 예를 들어, CUDA의 경우 **topi.sum**으로 끝나는 다음 일련의 작업을 오직

topi.generic.schedule_reduce 를 사용하여 Schedule 할 수 있습니다.

```
e = topi.elemwise_sum([c, d])
f = e / 2.0
g = topi.sum(f)
with tvm.target.cuda():
    sg = topi.cuda.schedule_reduce(g)
    print(tvm.lower(sg, [a, b], simple_mode=True))
```

Out:

```
/workspace/python/tvm/target/target.py:446: UserWarning: Try specifying cuda arch by adding 'arch=sm_xx' to your target.
  warnings.warn("Try specifying cuda arch by adding 'arch=sm_xx' to your target.")
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(a: T.Buffer((100, 10, 10), "float32"), b: T.Buffer((10, 10), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        T_divide_red = T.allocate([1], "float32", "global")
        threadIdx_x = T.launch_thread("threadIdx.x", 1024)
        T_divide_red_rf = T.allocate([1], "float32", "local")
        reduce_temp0 = T.allocate([1], "float32", "local")
        T_divide_red_rf_1 = T.Buffer((1,), data=T_divide_red_rf, scope="local", align=4)
        T_divide_red_rf_1[0] = T.float32(0)
        for k0_k1_fused_k2_fused_outer in range(10):
            if T.likely(k0_k1_fused_k2_fused_outer * 64 + threadIdx_x // 16 < 625 and
                k0_k1_fused_k2_fused_outer * 64 + threadIdx_x // 16 < 625 and
                k0_k1_fused_k2_fused_outer * 64 + threadIdx_x // 16 < 625):
                a_1 = T.Buffer((10000,), data=a.data)
                b_1 = T.Buffer((100,), data=b.data)
                T_divide_red_rf_1[0] = T_divide_red_rf_1[0] +
                    (a_1[k0_k1_fused_k2_fused_outer * 1024 + threadIdx_x]
                        + b_1[(k0_k1_fused_k2_fused_outer * 24 + threadIdx_x) % 100]
                        + a_1[k0_k1_fused_k2_fused_outer * 1024 + threadIdx_x]
```

```
                    * b_1[(k0_k1_fused_k2_fused_outer * 24 + threadIdx_x) % 100])
                    * T.float32(0.5)
        reduce_temp0_1 = T.Buffer((1,), data=reduce_temp0, scope="local")
        with T.attr(T.comm_reducer(lambda x, y: x + y, [T.float32(0)]), "reduce_scope",
            T.reinterpret("handle", T.uint64(0))):
            T.tvm_thread_allreduce(T.uint32(1), T_divide_red_rf_1[0], T.bool(True),
                                   reduce_temp0_1[0], threadIdx_x)
        if threadIdx_x == 0:
            T_divide_red_1 = T.Buffer((1,), data=T_divide_red, align=4)
            T_divide_red_1[0] = reduce_temp0_1[0]
```

보시다시피, 예정된 계산 단계가 누적되어 다음과 같이 검사할 수 있습니다.

```
print(sg.stages)
```

Out:

```
[stage(a, placeholder(a, 0x10aff7e0)), stage(b, placeholder(b, 0x30e06510)), stage(T_add, compute(T_add, body=[a[ax0, ax1, ax2] + b[ax1, ax2]], axis=[T.iter_var(ax0, T.Range(0, 100), "DataPar", ""), T.iter_var(ax1, T.Range(0, 10), "DataPar", ""), T.iter_var(ax2, T.Range(0, 10), "DataPar", "")], reduce_axis=[], tag=broadcast, attrs={})), stage(T_multiply, compute(T_multiply, body=[a[ax0, ax1, ax2] * b[ax1, ax2]], axis=[T.iter_var(ax0, T.Range(0, 100), "DataPar", ""), T.iter_var(ax1, T.Range(0, 10), "DataPar", ""), T.iter_var(ax2, T.Range(0, 10), "DataPar", "")], reduce_axis=[], tag=broadcast, attrs={})), stage(T_elemwise_sum, compute(T_elemwise_sum, body=[T_add[ax0, ax1, ax2] + T_multiply[ax0, ax1, ax2]], axis=[T.iter_var(ax0, T.Range(0, 100), "DataPar", ""), T.iter_var(ax1, T.Range(0, 10), "DataPar", ""), T.iter_var(ax2, T.Range(0, 10), "DataPar", "")], reduce_axis=[], tag=elemwise, attrs={})), stage(T_divide, compute(T_divide, body=[T_elemwise_sum[ax0, ax1, ax2] / T.float32(2)], axis=[T.iter_var(ax0, T.Range(0, 100), "DataPar", ""), T.iter_var(ax1, T.Range(0, 10), "DataPar", ""), T.iter_var(ax2, T.Range(0, 10), "DataPar", "")], reduce_axis=[], tag=elemwise, attrs={})), stage(T_divide_red.rf, compute(T_divide_red.rf, body=[T.reduce(T.comm_reducer(lambda x, y: x + y, [T.float32(0)]), source=[T_divide[(k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024) // 10 // 10, (k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024) // 10 % 10, (k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024) % 10]], init=[], axis=[T.iter_var(k0_k1_fused_k2_fused_outer, T.Range(0, 10), "CommReduce", "")], condition=T.likely((k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024) // 10 // 10 < 100 and (k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024) // 10 < 1000 and k0_k1_fused_k2_fused_inner + k0_k1_fused_k2_fused_outer * 1024 < 10000), value_index=0)],
```

```
axis=[T.iter_var(k0_k1_fused_k2_fused_inner, T.Range(0, 1024), "DataPar", "")],
reduce_axis=[T.iter_var(k0_k1_fused_k2_fused_outer, T.Range(0, 10), "CommReduce", "")], tag=,
attrs={})), stage(T_divide_red, compute(T_divide_red.repl, body=[T.reduce(T.comm_reducer(lambda
x, y: x + y, [T.float32(0)]), source=[T_divide_red.rf[k0_k1_fused_k2_fused_inner_v]], init=[],
axis=[T.iter_var(k0_k1_fused_k2_fused_inner_v, T.Range(0, 1024), "CommReduce", "")],
condition=T.bool(True), value_index=0)], axis=[],
reduce_axis=[T.iter_var(k0_k1_fused_k2_fused_inner_v, T.Range(0, 1024), "CommReduce", "")], tag=,
attrs={}))]
```

다음과 같이 **numpy** 결과와 비교하여 정확성을 테스트 할 수 있습니다

```
func = tvm.build(sg, [a, b, g], "cuda")
dev = tvm.cuda(0)
a_np = np.random.uniform(size=(x, y, y)).astype(a.dtype)
b_np = np.random.uniform(size=(y, y)).astype(b.dtype)
g_np = np.sum(np.add(a_np + b_np, a_np * b_np) / 2.0)
a_nd = tvm.nd.array(a_np, dev)
b_nd = tvm.nd.array(b_np, dev)
g_nd = tvm.nd.array(np.zeros(g_np.shape, dtype=g_np.dtype), dev)
func(a_nd, b_nd, g_nd)
tvm.testing.assert_allclose(g_nd.numpy(), g_np, rtol=1e-5)
```

TOPI는 또한 최적화된 Schedule로 _softmax_와 같은 일반적인 신경망 작업을 제공합니다

```
tarray = te.placeholder((512, 512), name="tarray")
softmax_topi = topi.nn.softmax(tarray)
with tvm.target.Target("cuda"):
    sst = topi.cuda.schedule_softmax(softmax_topi)
    print(tvm.lower(sst, [tarray], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
```

```
@T.prim_func
def main(tarray: T.Buffer((512, 512), "float32")):
    T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
    T_softmax_norm = T.allocate([65536], "float32x4", "global")
    blockIdx_x = T.launch_thread("blockIdx.x", 512)
    normal_reduce_temp0 = T.allocate([1], "float32", "local")
    reduce_temp0 = T.allocate([1], "float32", "local")
    T_softmax_exp = T.allocate([512], "float32", "warp")
    normal_reduce_temp0_1 = T.allocate([1], "float32", "local")
    reduce_temp0_1 = T.allocate([1], "float32", "local")
    threadIdx_x = T.env_thread("threadIdx.x")
    T_softmax_exp_1 = T.Buffer((512,), data=T_softmax_exp, scope="warp")
    with T.launch_thread(threadIdx_x, 32):
        normal_reduce_temp0_2 = T.Buffer((1,), data=normal_reduce_temp0, scope="local")
        normal_reduce_temp0_2[0] = T.float32(-3.4028234663852886e+38)
        tarray_1 = T.Buffer((262144,), data=tarray.data)
        for k_inner in range(16):
            normal_reduce_temp0_2[0] = T.max(normal_reduce_temp0_2[0],
                                             tarray_1[blockIdx_x * 512 + threadIdx_x * 16 + k_inner])
        with T.attr(T.comm_reducer(lambda x, y: T.max(x, y),
                    [T.float32(-3.4028234663852886e+38)]), "reduce_scope",
                    T.reinterpret("handle", T.uint64(0))):
            reduce_temp0_2 = T.Buffer((1,), data=reduce_temp0, scope="local")
            T.tvm_thread_allreduce(T.uint32(1), normal_reduce_temp0_2[0], T.bool(True),
                                   reduce_temp0_2[0], threadIdx_x)
        for i1_inner_outer in range(4):
            cse_var_1: T.int32 = i1_inner_outer * 4
            reduce_temp0_2 = T.Buffer((1,), data=reduce_temp0, scope="local", align=4)
            T_softmax_exp_1[threadIdx_x * 16 + cse_var_1:threadIdx_x * 16 + cse_var_1 + 4]
                = T.exp(tarray_1[blockIdx_x * 512 + threadIdx_x * 16 + cse_var_1:blockIdx_x
                    * 512 + threadIdx_x * 16 + cse_var_1 + 4]
                    - T.Broadcast(reduce_temp0_2[0], 4))
    T.launch_thread(threadIdx_x, 32)
    normal_reduce_temp0_2 = T.Buffer((1,), data=normal_reduce_temp0_1, scope="local")
    normal_reduce_temp0_2[0] = T.float32(0)
    for k_inner in range(16):
        normal_reduce_temp0_2[0] = normal_reduce_temp0_2[0]
```

```
                                    + T_softmax_exp_1[threadIdx_x * 16 + k_inner]
        with T.attr(T.comm_reducer(lambda x, y: x + y, [T.float32(0)]), "reduce_scope",
                    T.reinterpret("handle", T.uint64(0))):
            reduce_temp0_2 = T.Buffer((1,), data=reduce_temp0_1, scope="local")
            T.tvm_thread_allreduce(T.uint32(1), normal_reduce_temp0_2[0], T.bool(True),
                                   reduce_temp0_2[0], threadIdx_x)
        for i1_inner_outer in range(4):
            T_softmax_norm_1 = T.Buffer((65536,), "float32x4", data=T_softmax_norm)
            reduce_temp0_2 = T.Buffer((1,), data=reduce_temp0_1, scope="local", align=4)
            T_softmax_norm_1[blockIdx_x * 128 + threadIdx_x * 4 + i1_inner_outer]
                = T_softmax_exp_1[threadIdx_x * 16 + i1_inner_outer * 4:threadIdx_x * 16
                    + i1_inner_outer * 4 + 4] / T.Broadcast(reduce_temp0_2[0], 4)
```

컨볼루션 융합

topi.nn.conv2d와 **topi.nn.relu**를 함께 융합할 수 있습니다.

Note

TOPI 함수는 모두 제네릭 함수입니다. 성능을 최적화하기 위해 서로 다른 백엔드에 대해 서로 다른 구현이 있습니다. 각 백엔드에 대해 컴퓨팅 선언 및 Schedule 모두에 대한 대상 범위에서 호출해야 합니다. TVM은 대상 정보로 호출할 올바른 함수를 선택합니다.

```
data = te.placeholder((1, 3, 224, 224))
kernel = te.placeholder((10, 3, 5, 5))

with tvm.target.Target("cuda"):
    conv = topi.cuda.conv2d_nchw(data, kernel, 1, 2, 1)
    out = topi.nn.relu(conv)
    sconv = topi.cuda.schedule_conv2d_nchw([out])
    print(tvm.lower(sconv, [data, kernel], simple_mode=True))
```

Out:

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(placeholder: T.Buffer((1, 3, 224, 224), "float32"), placeholder_1: T.Buffer((10, 3, 5, 5), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "tir.noalias": T.bool(True)})
        compute = T.allocate([501760], "float32", "global")
        blockIdx_z = T.launch_thread("blockIdx.z", 5)
        conv2d_nchw = T.allocate([14], "float32", "local")
        pad_temp_shared = T.allocate([112], "float32", "shared")
        placeholder_shared = T.allocate([2], "float32", "shared")
        blockIdx_y = T.launch_thread("blockIdx.y", 224)
        blockIdx_x = T.launch_thread("blockIdx.x", 2)
        threadIdx_z = T.launch_thread("threadIdx.z", 1)
        threadIdx_y = T.launch_thread("threadIdx.y", 1)
        threadIdx_x = T.launch_thread("threadIdx.x", 16)
        conv2d_nchw_1 = T.Buffer((4,), data=conv2d_nchw, scope="local", align=8)
        conv2d_nchw_1[0] = T.float32(0)
        conv2d_nchw_1[2] = T.float32(0)
        conv2d_nchw_1[4] = T.float32(0)
        conv2d_nchw_1[6] = T.float32(0)
        conv2d_nchw_1[8] = T.float32(0)
        conv2d_nchw_1[10] = T.float32(0)
        conv2d_nchw_1[12] = T.float32(0)
        conv2d_nchw_1[1] = T.float32(0)
        conv2d_nchw_1[3] = T.float32(0)
        conv2d_nchw_1[5] = T.float32(0)
        conv2d_nchw_1[7] = T.float32(0)
        conv2d_nchw_1[9] = T.float32(0)
        conv2d_nchw_1[11] = T.float32(0)
        conv2d_nchw_1[13] = T.float32(0)
        for rc_outer, ry_outer in T.grid(3, 5):
            threadIdx_x_1 = T.env_thread("threadIdx.x")
```

```
                pad_temp_shared_1 = T.Buffer((112,), data=pad_temp_shared, scope="shared")
                placeholder_2 = T.Buffer((150528,), data=placeholder.data)
                with T.launch_thread("threadIdx.z", 1) as threadIdx_z_1:
                    threadIdx_y_1 = T.launch_thread("threadIdx.y", 1)
                    T.launch_thread(threadIdx_x_1, 16)
                    pad_temp_shared_1[threadIdx_x_1 * 7] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and 1 <= blockIdx_x * 56 + threadIdx_x_1 * 7 // 2, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 450], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 1] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and 1 <= blockIdx_x * 56 + (threadIdx_x_1 * 7 + 1) // 2, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 449], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 2] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 448], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 3] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 447], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 4] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 446], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 5] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 445], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 6] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 444], T.float32(0))
                threadIdx_x_2 = T.env_thread("threadIdx.x")
                placeholder_shared_1 = T.Buffer((2,), data=placeholder_shared, scope="shared", align=8)
                placeholder_3 = T.Buffer((750,), data=placeholder_1.data)
                with T.launch_thread("threadIdx.z", 1) as threadIdx_z_1:
                    threadIdx_y_1 = T.launch_thread("threadIdx.y", 1)
                    T.launch_thread(threadIdx_x_2, 16)
                    if T.likely(threadIdx_x_2 < 2):
                        placeholder_shared_1[threadIdx_x_2] = placeholder_3[blockIdx_z * 150 +
```

```
threadIdx_x_2 * 75 + rc_outer * 25 + ry_outer * 5]
                conv2d_nchw_1[0] = conv2d_nchw_1[0] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[0]
                conv2d_nchw_1[2] = conv2d_nchw_1[2] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[0]
                conv2d_nchw_1[4] = conv2d_nchw_1[4] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[0]
                conv2d_nchw_1[6] = conv2d_nchw_1[6] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[0]
                conv2d_nchw_1[8] = conv2d_nchw_1[8] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[0]
                conv2d_nchw_1[10] = conv2d_nchw_1[10] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[0]
                conv2d_nchw_1[12] = conv2d_nchw_1[12] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[0]
                conv2d_nchw_1[1] = conv2d_nchw_1[1] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[1]
                conv2d_nchw_1[3] = conv2d_nchw_1[3] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[1]
                conv2d_nchw_1[5] = conv2d_nchw_1[5] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[1]
                conv2d_nchw_1[7] = conv2d_nchw_1[7] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[1]
                conv2d_nchw_1[9] = conv2d_nchw_1[9] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[1]
                conv2d_nchw_1[11] = conv2d_nchw_1[11] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[1]
                conv2d_nchw_1[13] = conv2d_nchw_1[13] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[1]
                threadIdx_z_1 = T.env_thread("threadIdx.z")
                threadIdx_y_1 = T.env_thread("threadIdx.y")
                with T.launch_thread(threadIdx_z_1, 1):
                    T.launch_thread(threadIdx_y_1, 1)
                    T.launch_thread(threadIdx_x_1, 16)
                    pad_temp_shared_1[threadIdx_x_1 * 7] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and 1 <= blockIdx_x * 56 + (threadIdx_x_1 * 7 + 1) // 2, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 +
```

```
threadIdx_x_1 * 7 - 449], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 1] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 448], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 2] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 447], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 3] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 446], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 4] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 445], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 5] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 444], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 6] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 443], T.float32(0))
                threadIdx_z_2 = T.env_thread("threadIdx.z")
                threadIdx_y_2 = T.env_thread("threadIdx.y")
                with T.launch_thread(threadIdx_z_2, 1):
                    T.launch_thread(threadIdx_y_2, 1)
                    T.launch_thread(threadIdx_x_2, 16)
                    if T.likely(threadIdx_x_2 < 2):
                        placeholder_shared_1[threadIdx_x_2] = placeholder_3[blockIdx_z * 150 +
threadIdx_x_2 * 75 + rc_outer * 25 + ry_outer * 5 + 1]
                conv2d_nchw_1[0] = conv2d_nchw_1[0] + pad_temp_shared_1[threadIdx_x] *
placeholder_shared_1[0]
                conv2d_nchw_1[2] = conv2d_nchw_1[2] + pad_temp_shared_1[threadIdx_x + 16] *
placeholder_shared_1[0]
                conv2d_nchw_1[4] = conv2d_nchw_1[4] + pad_temp_shared_1[threadIdx_x + 32] *
placeholder_shared_1[0]
                conv2d_nchw_1[6] = conv2d_nchw_1[6] + pad_temp_shared_1[threadIdx_x + 48] *
placeholder_shared_1[0]
                conv2d_nchw_1[8] = conv2d_nchw_1[8] + pad_temp_shared_1[threadIdx_x + 64] *
placeholder_shared_1[0]
```

```
            conv2d_nchw_1[10] = conv2d_nchw_1[10] + pad_temp_shared_1[threadIdx_x + 80] *
placeholder_shared_1[0]
            conv2d_nchw_1[12] = conv2d_nchw_1[12] + pad_temp_shared_1[threadIdx_x + 96] *
placeholder_shared_1[0]
            conv2d_nchw_1[1] = conv2d_nchw_1[1] + pad_temp_shared_1[threadIdx_x] *
placeholder_shared_1[1]
            conv2d_nchw_1[3] = conv2d_nchw_1[3] + pad_temp_shared_1[threadIdx_x + 16] *
placeholder_shared_1[1]
            conv2d_nchw_1[5] = conv2d_nchw_1[5] + pad_temp_shared_1[threadIdx_x + 32] *
placeholder_shared_1[1]
            conv2d_nchw_1[7] = conv2d_nchw_1[7] + pad_temp_shared_1[threadIdx_x + 48] *
placeholder_shared_1[1]
            conv2d_nchw_1[9] = conv2d_nchw_1[9] + pad_temp_shared_1[threadIdx_x + 64] *
placeholder_shared_1[1]
            conv2d_nchw_1[11] = conv2d_nchw_1[11] + pad_temp_shared_1[threadIdx_x + 80] *
placeholder_shared_1[1]
            conv2d_nchw_1[13] = conv2d_nchw_1[13] + pad_temp_shared_1[threadIdx_x + 96] *
placeholder_shared_1[1]
            with T.launch_thread(threadIdx_z_1, 1):
                T.launch_thread(threadIdx_y_1, 1)
                T.launch_thread(threadIdx_x_1, 16)
                pad_temp_shared_1[threadIdx_x_1 * 7] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 448], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 1] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 447], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 2] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 446], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 3] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 445], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 4] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 444], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 5] = T.if_then_else(2 <= blockIdx_y +
```

```
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 443], T.float32(0))
                    pad_temp_shared_1[threadIdx_x_1 * 7 + 6] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 442], T.float32(0))
                with T.launch_thread(threadIdx_z_2, 1):
                    T.launch_thread(threadIdx_y_2, 1)
                    T.launch_thread(threadIdx_x_2, 16)
                    if T.likely(threadIdx_x_2 < 2):
                        placeholder_shared_1[threadIdx_x_2] = placeholder_3[blockIdx_z * 150 + threadIdx_x_2 * 75 + rc_outer * 25 + ry_outer * 5 + 2]
                conv2d_nchw_1[0] = conv2d_nchw_1[0] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[0]
                conv2d_nchw_1[2] = conv2d_nchw_1[2] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[0]
                conv2d_nchw_1[4] = conv2d_nchw_1[4] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[0]
                conv2d_nchw_1[6] = conv2d_nchw_1[6] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[0]
                conv2d_nchw_1[8] = conv2d_nchw_1[8] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[0]
                conv2d_nchw_1[10] = conv2d_nchw_1[10] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[0]
                conv2d_nchw_1[12] = conv2d_nchw_1[12] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[0]
                conv2d_nchw_1[1] = conv2d_nchw_1[1] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[1]
                conv2d_nchw_1[3] = conv2d_nchw_1[3] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[1]
                conv2d_nchw_1[5] = conv2d_nchw_1[5] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[1]
                conv2d_nchw_1[7] = conv2d_nchw_1[7] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[1]
                conv2d_nchw_1[9] = conv2d_nchw_1[9] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[1]
                conv2d_nchw_1[11] = conv2d_nchw_1[11] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[1]
```

```
            conv2d_nchw_1[13] = conv2d_nchw_1[13] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[1]
            with T.launch_thread(threadIdx_z_1, 1):
                T.launch_thread(threadIdx_y_1, 1)
                T.launch_thread(threadIdx_x_1, 16)
                pad_temp_shared_1[threadIdx_x_1 * 7] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 447], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 1] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 446], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 2] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 445], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 3] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 444], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 4] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 443], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 5] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 442], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 6] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and blockIdx_x * 16 + threadIdx_x_1 < 31, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 441], T.float32(0))
            with T.launch_thread(threadIdx_z_2, 1):
                T.launch_thread(threadIdx_y_2, 1)
                T.launch_thread(threadIdx_x_2, 16)
                if T.likely(threadIdx_x_2 < 2):
                    placeholder_shared_1[threadIdx_x_2] = placeholder_3[blockIdx_z * 150 + threadIdx_x_2 * 75 + rc_outer * 25 + ry_outer * 5 + 3]
            conv2d_nchw_1[0] = conv2d_nchw_1[0] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[0]
            conv2d_nchw_1[2] = conv2d_nchw_1[2] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[0]
```

```
            conv2d_nchw_1[4] = conv2d_nchw_1[4] + pad_temp_shared_1[threadIdx_x + 32] *
placeholder_shared_1[0]
            conv2d_nchw_1[6] = conv2d_nchw_1[6] + pad_temp_shared_1[threadIdx_x + 48] *
placeholder_shared_1[0]
            conv2d_nchw_1[8] = conv2d_nchw_1[8] + pad_temp_shared_1[threadIdx_x + 64] *
placeholder_shared_1[0]
            conv2d_nchw_1[10] = conv2d_nchw_1[10] + pad_temp_shared_1[threadIdx_x + 80] *
placeholder_shared_1[0]
            conv2d_nchw_1[12] = conv2d_nchw_1[12] + pad_temp_shared_1[threadIdx_x + 96] *
placeholder_shared_1[0]
            conv2d_nchw_1[1] = conv2d_nchw_1[1] + pad_temp_shared_1[threadIdx_x] *
placeholder_shared_1[1]
            conv2d_nchw_1[3] = conv2d_nchw_1[3] + pad_temp_shared_1[threadIdx_x + 16] *
placeholder_shared_1[1]
            conv2d_nchw_1[5] = conv2d_nchw_1[5] + pad_temp_shared_1[threadIdx_x + 32] *
placeholder_shared_1[1]
            conv2d_nchw_1[7] = conv2d_nchw_1[7] + pad_temp_shared_1[threadIdx_x + 48] *
placeholder_shared_1[1]
            conv2d_nchw_1[9] = conv2d_nchw_1[9] + pad_temp_shared_1[threadIdx_x + 64] *
placeholder_shared_1[1]
            conv2d_nchw_1[11] = conv2d_nchw_1[11] + pad_temp_shared_1[threadIdx_x + 80] *
placeholder_shared_1[1]
            conv2d_nchw_1[13] = conv2d_nchw_1[13] + pad_temp_shared_1[threadIdx_x + 96] *
placeholder_shared_1[1]
            with T.launch_thread(threadIdx_z_1, 1):
                T.launch_thread(threadIdx_y_1, 1)
                T.launch_thread(threadIdx_x_1, 16)
                pad_temp_shared_1[threadIdx_x_1 * 7] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 446], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 1] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 445], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 2] = T.if_then_else(2 <= blockIdx_y +
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 +
ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 444], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 3] = T.if_then_else(2 <= blockIdx_y +
```

```
ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 443], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 4] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 442], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 5] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and blockIdx_x * 16 + threadIdx_x_1 < 31, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 441], T.float32(0))
                pad_temp_shared_1[threadIdx_x_1 * 7 + 6] = T.if_then_else(2 <= blockIdx_y + ry_outer and blockIdx_y + ry_outer < 226 and blockIdx_x * 112 + threadIdx_x_1 * 7 < 216, placeholder_2[rc_outer * 50176 + blockIdx_y * 224 + ry_outer * 224 + blockIdx_x * 112 + threadIdx_x_1 * 7 - 440], T.float32(0))
            with T.launch_thread(threadIdx_z_2, 1):
                T.launch_thread(threadIdx_y_2, 1)
                T.launch_thread(threadIdx_x_2, 16)
                if T.likely(threadIdx_x_2 < 2):
                    placeholder_shared_1[threadIdx_x_2] = placeholder_3[blockIdx_z * 150 + threadIdx_x_2 * 75 + rc_outer * 25 + ry_outer * 5 + 4]
            conv2d_nchw_1[0] = conv2d_nchw_1[0] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[0]
            conv2d_nchw_1[2] = conv2d_nchw_1[2] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[0]
            conv2d_nchw_1[4] = conv2d_nchw_1[4] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[0]
            conv2d_nchw_1[6] = conv2d_nchw_1[6] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[0]
            conv2d_nchw_1[8] = conv2d_nchw_1[8] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[0]
            conv2d_nchw_1[10] = conv2d_nchw_1[10] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[0]
            conv2d_nchw_1[12] = conv2d_nchw_1[12] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[0]
            conv2d_nchw_1[1] = conv2d_nchw_1[1] + pad_temp_shared_1[threadIdx_x] * placeholder_shared_1[1]
            conv2d_nchw_1[3] = conv2d_nchw_1[3] + pad_temp_shared_1[threadIdx_x + 16] * placeholder_shared_1[1]
```

```
            conv2d_nchw_1[5] = conv2d_nchw_1[5] + pad_temp_shared_1[threadIdx_x + 32] * placeholder_shared_1[1]
            conv2d_nchw_1[7] = conv2d_nchw_1[7] + pad_temp_shared_1[threadIdx_x + 48] * placeholder_shared_1[1]
            conv2d_nchw_1[9] = conv2d_nchw_1[9] + pad_temp_shared_1[threadIdx_x + 64] * placeholder_shared_1[1]
            conv2d_nchw_1[11] = conv2d_nchw_1[11] + pad_temp_shared_1[threadIdx_x + 80] * placeholder_shared_1[1]
            conv2d_nchw_1[13] = conv2d_nchw_1[13] + pad_temp_shared_1[threadIdx_x + 96] * placeholder_shared_1[1]
        compute_1 = T.Buffer((501760,), data=compute)
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x] = T.max(conv2d_nchw_1[0], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 16] = T.max(conv2d_nchw_1[2], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 32] = T.max(conv2d_nchw_1[4], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 48] = T.max(conv2d_nchw_1[6], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 64] = T.max(conv2d_nchw_1[8], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 80] = T.max(conv2d_nchw_1[10], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 96] = T.max(conv2d_nchw_1[12], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50176] = T.max(conv2d_nchw_1[1], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50192] = T.max(conv2d_nchw_1[3], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50208] = T.max(conv2d_nchw_1[5], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50224] = T.max(conv2d_nchw_1[7], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50240] = T.max(conv2d_nchw_1[9], T.float32(0))
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x + 50256] = T.max(conv2d_nchw_1[11], T.float32(0))
```

```
        compute_1[blockIdx_z * 100352 + blockIdx_y * 224 + blockIdx_x * 112 + threadIdx_x +
50272] = T.max(conv2d_nchw_1[13], T.float32(0))
```

요약

이 튜토리얼에서 다음을 알아보았습니다

- numpy 스타일 연산자를 사용한 일반적인 작업에 TOPI API를 사용하는 방법.
- TOPI가 컨텍스트에 대한 제네릭 Schedule 및 연산자 융합을 촉진하여 최적화된 커널 코드를 생성하는 방법.

Download Python source code: intro_topi.py

https://tvm.apache.org/docs/_downloads/3a9b1d387f618487c8ccf6b8b78ae179/intro_topi.py

Download Jupyter notebook: intro_topi.ipynb

https://tvm.apache.org/docs/_downloads/63f9e50204143ea3c2d3593c72439b3d/intro_topi.ipynb

1.16 Examples

MNIST model

Training and Save Weights (filename: 'mnist_weights.h5')

```
# python mnist_model_save_weights.py
```

Compile

```
# python tvm_mnist.py
```

filename: mnist_model_save_weights.py

```
# Source: https://aben20807.github.io/posts/20190616-build-mnist-with-tvm/

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
```

```
model.add(Conv2D(32, kernel_size=(3, 3),activation='relu',input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adadelta(),metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=epochs,
          verbose=1, validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
model.save_weights('mnist_weights.h5')
```

filename: tvm_mnist.py

```
# Source-based: https://aben20807.github.io/posts/20190616-build-mnist-with-tvm/
# Note: <save weights> and <handwritten number image> are required

#import nnvm
#
# FIX: lives in the TVM repo.

import tvm
import tvm.relay as relay

#from scipy.misc import imread, imresize
#
# FIX:
# pip install imageio
# pip install opencv-python
from imageio import imread
from cv2 import resize as imresize

import numpy as np
import keras
from keras.models import load_model
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, InputLayer
from keras.layers import Conv2D, MaxPooling2D
num_classes = 10
input_shape = (28, 28, 1)

x = imread('test3.png',mode='L')
# Compute a bit-wise inversion so black becomes white and vice versa
x = np.invert(x)
# Make it the right size
x = imresize(x,(28,28))
# Convert to a 4D tensor to feed into our model
x = x.reshape(1,28,28,1)
x = x.astype('float32')
```

```
x /= 255

# model = load_model('cnn.h5')
# Construct a MNIST model
model = Sequential()
model.add(InputLayer(input_shape=input_shape))
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# Load the weights that we get from last program
model.load_weights('mnist_weights.h5')

shape_dict = {'input_1': (1, 1, 28, 28)}
func, params = relay.frontend.from_keras(model, shape_dict)
target = "llvm"
ctx = tvm.cpu(0)
with relay.build_config(opt_level=3):
      executor = relay.build_module.create_executor('graph', func, ctx, target)

# Pperform the prediction
dtype = 'float32'

#tvm_out = executor.evaluate(func)(tvm.nd.array(x.astype(dtype)), **params)
#
# FIX:
print( func.astext(show_meta_data=False) )
tvm_out = executor.evaluate()(tvm.nd.array(x.astype(dtype)), **params)

print(np.argmax(tvm_out.asnumpy()[0]))
```

Issues

InputLayer: without InputLayer as the first layer

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Results: mod metadata and errors in TVM

```
#[version = "0.0.5"]
def @main(%conv2d_input, %v_param_1: Tensor[(32, 1, 3, 3), float32], %v_param_2: Tensor[(32), float32], %v_param_3: Tensor[(64, 32, 3, 3), float32], %v_param_4: Tensor[(64), float32], %v_param_5: Tensor[(128, 9216), float32], %v_param_6: Tensor[(128), float32], %v_param_7: Tensor[(10, 128), float32], %v_param_8: Tensor[(10), float32]) {
  %0 = nn.conv2d(%conv2d_input, %v_param_1, padding=[0, 0, 0, 0], channels=32, kernel_size=[3, 3]);
  %1 = nn.bias_add(%0, %v_param_2);
  %2 = nn.relu(%1);
  %3 = nn.conv2d(%2, %v_param_3, padding=[0, 0, 0, 0], channels=64, kernel_size=[3, 3]);
  %4 = nn.bias_add(%3, %v_param_4);
  %5 = nn.relu(%4);
  %6 = nn.max_pool2d(%5, pool_size=[2, 2], strides=[2, 2], padding=[0, 0, 0, 0]);
  %7 = transpose(%6, axes=[0, 2, 3, 1]);
  %8 = nn.batch_flatten(%7);
  %9 = nn.dense(%8, %v_param_5, units=128);
  %10 = nn.bias_add(%9, %v_param_6);
  %11 = nn.relu(%10);
  %12 = nn.dense(%11, %v_param_7, units=10);
  %13 = nn.bias_add(%12, %v_param_8);
```

```
   nn.softmax(%13, axis=1)
}
The type inference pass was unable to infer a type for this expression.
This usually occurs when an operator call is under constrained in some way, check other reported errors for hints of what may of happened.
...
tvm.error.DiagnosticError: Traceback (most recent call last):
6: tvm::runtime::PackedFuncObj::Extractor<tvm::runtime::PackedFuncSubObj
<tvm::runtime::TypedPackedFunc<tvm::IRModule
...
```

InputLayer: with InputLayer as the first layer

```
model = Sequential()
#
# ADD InputLayer as the first layer
input_shape = (28, 28, 1)
model.add(InputLayer(input_shape=input_shape))
#
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Results: mod metadata and no errors in TVM

```
#[version = "0.0.5"]
def @main(%input_1: Tensor[(1, 1, 28, 28), float32], %v_param_1: Tensor[(32, 1, 3, 3), float32], %v_param_2: Tensor[(32), float32], %v_param_3: Tensor[(64, 32, 3, 3), float32], %v_param_4: Tensor[(64), float32], %v_param_5: Tensor[(128, 9216), float32], %v_param_6: Tensor[(128), float32], %v_param_7: Tensor[(10, 128), float32], %v_param_8: Tensor[(10), float32]) {
```

```
  %0 = nn.conv2d(%input_1, %v_param_1, padding=[0, 0, 0, 0], channels=32, kernel_size=[3, 3]);
  %1 = nn.bias_add(%0, %v_param_2);
  %2 = nn.relu(%1);
  %3 = nn.conv2d(%2, %v_param_3, padding=[0, 0, 0, 0], channels=64, kernel_size=[3, 3]);
  %4 = nn.bias_add(%3, %v_param_4);
  %5 = nn.relu(%4);
  %6 = nn.max_pool2d(%5, pool_size=[2, 2], strides=[2, 2], padding=[0, 0, 0, 0]);
  %7 = transpose(%6, axes=[0, 2, 3, 1]);
  %8 = nn.batch_flatten(%7);
  %9 = nn.dense(%8, %v_param_5, units=128);
  %10 = nn.bias_add(%9, %v_param_6);
  %11 = nn.relu(%10);
  %12 = nn.dense(%11, %v_param_7, units=10);
  %13 = nn.bias_add(%12, %v_param_8);
  nn.softmax(%13, axis=1)
}
```

C/C++

Android