# Dive into Deep Learning Compiler

*Release 0.1*

**Contributors**

**Oct 16, 2020**

# Contents

**Working in progress**. Check our roadmap[1] for more details.

This project is for readers who are interested in high-performance implementation of their programs utilizing deep learning techniques, especially model inference, but may not have got their hands dirty yet. We assume readers have a minimal background of only having experience on NumPy before. With this in mind, we will explain things from scratch and introduce relative background when needed. Experienced readers, however, should also find the contents useful.

We roughly classify contents into three major parts. In the first part, we will introduce how to implement and optimize operators, such as matrix multiplication and convolution, for various hardware platforms. This is the basic component for deep learning as well as scientific computing in general. In the second part, we will show how to convert neural network models from various deep learning frameworks and further optimize them in the program level. The last part we will address how to deploy the optimized program into various environment such as mobile phones. In addition, at the end of the book, we plan to cover some latest advance of the deep learning compiler domain.

[1] http://bit.ly/2NQ7gh3

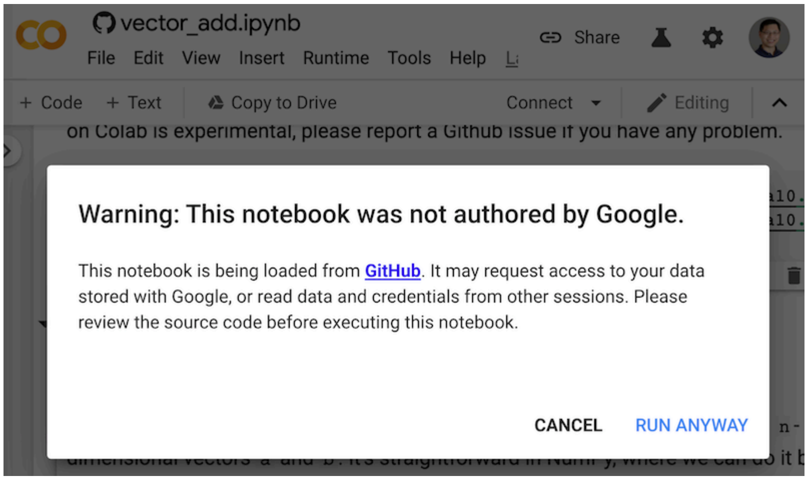
# 1 | Getting Started

Let's start your journey! There's a lot to learn, but every journey starts somewhere. In this part, we'll discuss:

- Installing required libraries so you can run (almost) every chapter by yourself
- Writing an operator that adds two vectors
- Compiling a neural network model to run the inference, and saving the compiled library

## 1.1 Installation

Each section of this book is a Jupyter notebook. The easiest way to run them is clicking the **COLAB** button on the upper right of the HTML page, which will directly you to Google Colab with the corresponding notebook opened. Running the first code cell will connect to a host runtime and show the following warning message. You can click RUN ANYWAY to continue.



on Colab is experimental, please report a Github issue if you have any problem.

Warning: This notebook was not authored by Google.

This notebook is being loaded from GitHub. It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

Fig. 1.1.1: Click RUN ANYWAY to run a section on Colab.

The reset of this section will go through how to set up a Python environment, Jupyter's interactive notebooks, the relevant libraries, and the code needed to run the book you can run them on your machines.

### 1.1.1 Obtaining Source Codes

The source code package containing all notebooks is available at http://tvm.d2l.ai/d2l-tvm.zip. Please download it and extract it into a folder. For example, on Linux/macOS, if you have both `wget` and `unzip` installed, you can do it through:

```
wget http://tvm.d2l.ai/d2l-tvm.zip
unzip d2l-tvm.zip -d d2l-tvm
```

### 1.1.2 Installing Running Environment

If you have both `Python 3.5` or later and `pip` installed, the easiest way to install the running environment is through `pip`. The required packages are

- `d2ltvm` for all dependencies such as Jupyter and saved code blocks
- `tvm` (Chen et al., 2018) for the deep learning compiler we are using
- `mxnet` as the baseline in some chapters

First install `d2ltvm`:

```
pip install git+https://github.com/d2l-ai/d2l-tvm
```

Then compile `tvm` from source codes. TVM doesn't have a pip package because it highly depends on the libraries available on your system. Please follow the instructions on tvm.ai[2] to install `tvm`. The configuration in `config.cmake` this book requires at least

```
set(USE_LLVM ON)
```

If you plan to run on Nvidia GPUs as well, you will also need to

```
set(USE_CUDA ON)
```

Also don't forget to enable `cython`, which accelerates the performance. You just need to run `make cython` in the TVM source folder.

If luckily you are using Ubuntu with `python-3.7`, `llvm-6.0` and `cuda-10.1` installed, you may use the pre-built library that is for evaluating this book:

```
pip install https://tvm-repo.s3-us-west-2.amazonaws.com/tvm-0.7.dev1-cp37-
↪cp37m-linux_x86_64.whl
```

Our code runs on tvm-0.7-dev1 for now.

Finally, install MXNet's CUDA version if GPUs are available (Chen et al., 2015). Assume you are have CUDA 10.1 installed, then

```
pip install mxnet-cu101
```

[2] https://docs.tvm.ai/install/from_source.html

You can change the 101 to match your CUDA version.

Once all packages are installed, you can open the Jupyter notebook by

```
jupyter notebook
```

At this point open http://localhost:8888 (which usually opens automatically) in the browser, then you can view and run the code in each section of the book.

### 1.1.3 Code

Throughout the book, we save reusable code blocks in the d2ltvm package by adding the comment: "# Save to the d2ltvm package." before the code block. For example, the following code snippet shows the libraries imported by d2ltvm.

```
# Save to the d2ltvm package.
import tvm
from tvm import te
import time
import timeit
import numpy as np
from matplotlib import pyplot as plt
from IPython import display
try:
  import mxnet as mx
except:
  pass
```

### 1.1.4 Discussions[3]

## 1.2 Vector Add

Now you have installed all libraries, let's write our first program: summing two n-dimensional vectors a and b. It's straightforward in NumPy, where we can do it by c = a + b.

### 1.2.1 Implementing with NumPy

```
import numpy as np

np.random.seed(0)
n = 100
a = np.random.normal(size=n).astype(np.float32)
b = np.random.normal(size=n).astype(np.float32)
c = a + b
```

Here we create two random vectors with length 100, and sum them element-wisely. Note that NumPy in default uses 64-bit floating-points or 64-bit integers, which is different from 32-bit floating point typically used in deep learning, so we explicitly cast the data type.

[3] https://discuss.tvm.ai/t/getting-started-installation/4706

Although we can use the build-in + operator in NumPy to realize element-wise add, let's try to implement it by only using scalar operators. It will help us understand the implementation with TVM. The following function uses a for-loop to iterate over every element of the vectors, and then add two elements together with the scalar + operator each time.

```
def vector_add(a, b, c):
    for i in range(n):
        c[i] = a[i] + b[i]

d = np.empty(shape=n, dtype=np.float32)
vector_add(a, b, d)
np.testing.assert_array_equal(c, d)
```

Given we will frequently create two random ndarrays and another empty one to store the results in the following chapters, we save this routine to reuse it in the future.

```
# Save to the d2ltvm package.
def get_abc(shape, constructor=None):
    """Return random a, b and empty c with the same shape.
    """
    np.random.seed(0)
    a = np.random.normal(size=shape).astype(np.float32)
    b = np.random.normal(size=shape).astype(np.float32)
    c = np.empty_like(a)
    if constructor:
        a, b, c = [constructor(x) for x in (a, b, c)]
    return a, b, c
```

Note that we fixed the random seed so that we will always get the same results to facilitate the comparison between NumPy, TVM and others. In addition, it accepts an optional `constructor` to convert the data into a different format.

### 1.2.2 Defining the TVM Computation

Now let's implement `vector_add` in TVM. The TVM implementation differs from above in two ways:

1. We don't need to write the complete function, but only to specify how each element of the output, i.e. `c[i]`, is computed
2. TVM is symbolic, we create symbolic variables by specifying their shapes, and define how the program will be computed

In the following program, we first declare the placeholders `A` and `B` for both inputs by specifying their shapes, `(n,)`, through `tvm.te.placeholder`. Both `A` and `B` are `Tensor` objects, which we can feed data later. We assign names to them so we can print an easy-to-read program later.

Next we define how the output `C` is computed by `tvm.compute`. It accepts two arguments, the output shape, and a function to compute each element by giving its index. Since the output is a vector, its elements are indexed by integers. The lambda function defined in `tvm.compute` accepts a single argument `i`, and returns `c[i]`, which is identical to `c[i] = a[i] + b[i]` defined in `vector_add`. One difference is that we don't write the for-loop, which will be filled by TVM later.

```
import tvm
from tvm import te # te stands for tensor expression

# Save to the d2ltvm package.
def vector_add(n):
    """TVM expression for vector add"""
    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')
    return A, B, C

A, B, C = vector_add(n)
type(A), type(C)
```

```
(tvm.te.tensor.Tensor, tvm.te.tensor.Tensor)
```

We can see that A, B, and C are all Tensor objects, which can be viewed as a symbolic version of NumPy's ndarray. We can access the variables' attributes such as data type and shape. But those values don't have concrete values right now.

```
(A.dtype, A.shape), (C.dtype, C.shape)
```

```
(('float32', [100]), ('float32', [100]))
```

The operation that generates the tensor object can be accessed by A.op.

```
type(A.op), type(C.op)
```

```
(tvm.te.tensor.PlaceholderOp, tvm.te.tensor.ComputeOp)
```

We can see that the types of the operations for A and C are different, but they share the same base class Operation, which represents an operation that generates a tensor object.

```
A.op.__class__.__bases__[0]
```

```
tvm.te.tensor.Operation
```

### 1.2.3 Creating a Schedule

To run the computation, we need to specify how to execute the program, for example, the order to access data and how to do multi-threading parallelization. Such an execution plan is called a *schedule*. Since C is the output tensor, let's create a default schedule on its operator and print the pseudo codes.

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

A schedule consists of several stages. Each stage corresponds to an operation to describe how it is scheduled. We can access a particular stage by either s[C] or s[C.op].

```
type(s), type(s[C])
```

```
(tvm.te.schedule.Schedule, tvm.te.schedule.Stage)
```

Later on we will see how to change the execution plan to better utilize the hardware resources to improve its efficiency. Here let's see the default execution plan by printing the C-like pseudo codes.

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

```
produce c {
  for (i, 0, 100) {
    c[i] = (a[i] + b[i])
  }
}
```

The `lower` method accepts the schedule and input and output tensors. The `simple_mode=True` will print the program in a simple and compact way. Note that the program has added proper for-loops according to the output shape. Overall, it's quite similar to the preview function `vector_add`.

Now you see that TVM separates the computation and the schedule. The computation defines how the results are computed, which will not change no matter on what hardware platform you run the program. On the other hand, an efficient schedule are often hardware dependent, but changing a schedule will not impact the correctness. The idea of separating computation from schedule is inherited by TVM from Halide (Ragan-Kelley et al., 2013).

### 1.2.4 Compilation and Execution

Once both computation and schedule are defined, we can compile them into an executable module with `tvm.build`. It accepts the same argument as `tvm.lower`. In fact, it first calls `tvm.lower` to generate the program and then compiles to machine codes.

```
mod = tvm.build(s, [A, B, C])
type(mod)
```

```
tvm.runtime.module.Module
```

It returns an executable module object. Now we can feed data for `A`, `B` and `C` to run it. The tensor data must be `tvm.ndarray.NDArray` object. The easiest way is to create NumPy ndarray objects first and then convert them into TVM ndarray by `tvm.nd.array`. We can convert them back to NumPy by the `asnumpy` method.

```
x = np.ones(2)
y = tvm.nd.array(x)
type(y), y.asnumpy()
```

```
(tvm.runtime.ndarray.NDArray, array([1., 1.]))
```

Now let's construct data and return them as TVM ndarrays.

```
a, b, c = get_abc(100, tvm.nd.array)
```

Do the computation, and verify the results.

```
mod(a, b, c)
np.testing.assert_array_equal(a.asnumpy() + b.asnumpy(), c.asnumpy())
```

### 1.2.5 Argument Constraints

Remember that we specified both inputs to be 100-length vectors when declaring `A` and `B`.

```
A.shape, B.shape, C.shape
```

```
([100], [100], [100])
```

TVM will check if the input shapes satisfy this specification.

```
try:
    a, b, c = get_abc(200, tvm.nd.array)
    mod(a, b, c)
except tvm.TVMError as e:
    print(e)
```

```
Traceback (most recent call last):
  [bt] (1) /var/lib/jenkins/miniconda3/envs/d2l-tvm-0/lib/python3.7/site-
↪packages/tvm/libtvm.so(TVMFuncCall+0x61) [0x7f8eec7a60f1]
  [bt] (0) /var/lib/jenkins/miniconda3/envs/d2l-tvm-0/lib/python3.7/site-
↪packages/tvm/libtvm.so(+0xcab5a1) [0x7f8eec78d5a1]
  File "/home/ubuntu/tvm/src/runtime/library_module.cc", line 89
TVMError: Check failed: ret == 0 (-1 vs. 0) : Assert fail: (100 == int32(arg0.
↪shape[0])), Argument arg0.shape[0] has an unsatisfied constraint
```

The default data type in TVM is `float32`.

```
A.dtype, B.dtype, C.dtype
```

```
('float32', 'float32', 'float32')
```

An error will appear if input with a different data type.

```
try:
    a, b, c = get_abc(100, tvm.nd.array)
    a = tvm.nd.array(a.asnumpy().astype('float64'))
    mod(a, b, c)
except tvm.TVMError as e:
    print(e)
```

```
Traceback (most recent call last):
  [bt] (1) /var/lib/jenkins/miniconda3/envs/d2l-tvm-0/lib/python3.7/site-
↪packages/tvm/libtvm.so(TVMFuncCall+0x61) [0x7f8eec7a60f1]
  [bt] (0) /var/lib/jenkins/miniconda3/envs/d2l-tvm-0/lib/python3.7/site-
↪packages/tvm/libtvm.so(+0xcab5a1) [0x7f8eec78d5a1]
  File "/home/ubuntu/tvm/src/runtime/library_module.cc", line 89
TVMError: Check failed: ret == 0 (-1 vs. 0) : Assert fail: (((tvm_struct_
↪get(arg0, 0, 5) == (uint8)2) && (tvm_struct_get(arg0, 0, 6) == (uint8)32)) &
↪& (tvm_struct_get(arg0, 0, 7) == (uint16)1)), arg0.dtype is expected to be␣
↪float32
```

### 1.2.6 Saving and Loading a Module

A compiled a module can be saved into disk,

```
mod_fname = 'vector-add.tar'
mod.export_library(mod_fname)
```

and then loaded back later.

```
loaded_mod = tvm.runtime.load_module(mod_fname)
```

Verify the results.

```
a, b, c = get_abc(100, tvm.nd.array)
loaded_mod(a, b, c)
np.testing.assert_array_equal(a.asnumpy() + b.asnumpy(), c.asnumpy())
```

### 1.2.7 Summary

Implementing an operator using TVM has three steps:

1. Declare the computation by specifying input and output shapes and how each output element is computed.
2. Create a schedule to (hopefully) fully utilize the machine resources.
3. Compile to the hardware target.

In addition, we can save the compiled module into disk so we can load it back later.

### 1.2.8 Discussions[4]

## 1.3 Neural Network Inference

You have seen how to implement and compile a simple vector addition operator in Section 1.2. Now we will make a big jump to compile a whole pre-trained neural network, which consists of a set of operators, to run the inference.

[4] https://discuss.tvm.ai/t/getting-started-vector-addition/4707

```
import numpy as np
import mxnet as mx
from PIL import Image
import tvm
from tvm import relay
```

Here three additional modules are imported compared to the previous chapter. We will use `PIL` to read images, `mxnet` to obtain pre-trained neural networks, and the `relay` module (Roesch et al., 2019) in TVM to convert and optimize a neural network. `Relay` is the high-level intermediate representation (IR) in TVM to represent a neural network.

### 1.3.1 Obtaining Pre-trained Models

A pre-trained model means a neural network with parameters trained on a data set. Here we download and load a ResNet-18 model by specifying `pretrained=True` from MXNet's model zoo (Chen et al., 2015). If you want to know details about this model, please refer to Chapter 7.6 in D2L[5]. You can find more models on the MXNet model zoo[6] page, or refer to GluonCV[7] and GluonNLP[8] for more computer vision and natural language models.

```
model = mx.gluon.model_zoo.vision.resnet18_v2(pretrained=True)
len(model.features), model.output
```

```
(13, Dense(512 -> 1000, linear))
```

The loaded model is trained on the Imagenet 1K dataset, which contains around 1 million natural object images among 1000 classes. The model has two parts, the main body part `model.features` contains 13 blocks, and the output layer is a dense layer with 1000 outputs.

The following code block loads the text labels for each class in the Imagenet dataset.

```
with open('../data/imagenet1k_labels.txt') as f:
    labels = eval(f.read())
```

### 1.3.2 Pre-processing Data

We first read a sample image. It is resized to the size, i.e. 224 px width and height, which we used to train the neural network.

```
image = Image.open('../data/cat.jpg').resize((224, 224))
image
```

[5] http://d2l.ai/chapter_convolutional-modern/resnet.html
[6] https://mxnet.apache.org/api/python/docs/api/gluon/model_zoo/index.html
[7] https://gluon-cv.mxnet.io/model_zoo/index.html
[8] http://gluon-nlp.mxnet.io/model_zoo/index.html

According to the model zoo page[9]. Image pixels are normalized on each color channel, and the data layout is `(batch, RGB channels, height, width)`. The following function transforms the input image to satisfy the requirement.

```
# Save to the d2ltvm package
def image_preprocessing(image):
    image = np.array(image) - np.array([123., 117., 104.])
    image /= np.array([58.395, 57.12, 57.375])
    image = image.transpose((2, 0, 1))
    image = image[np.newaxis, :]
    return image.astype('float32')

x = image_preprocessing(image)
x.shape
```

```
(1, 3, 224, 224)
```

### 1.3.3 Compile Pre-trained Models

To compile a model, we first express the MXNet model in Relay IR, which the `from_mxnet` method could help. In the method, we provide the model with the input data shape. Some neural networks may require some dimension(s) of the data shape to be determined later. However, in ResNet model the data shape is fixed, which makes it easier for the compiler to achieve high performance. We will mostly stick to fixed data shape in the book. We only touch the dynamic data shape (i.e. some dimension(s) to be determined in runtime) in very late chapters.

```
relay_mod, relay_params = relay.frontend.from_mxnet(model, {'data': x.shape})
type(relay_mod), type(relay_params)
```

[9] https://mxnet.apache.org/api/python/docs/api/gluon/model_zoo/index.html

```
(tvm.ir.module.IRModule, dict)
```

This method will return the program `relay_mod`, which is a `relay` module, and a dictionary of parameters `relay_params` that maps a string key to a TVM ndarray. Next, we lower the module to some lower-level IR which can be consumed by `llvm` backend. LLVM[10] defines an IR that has been adopted by multiple programming languages. The LLVM compiler is then able to compile the generated programs into machine codes for CPUs. We have already used it to compile the vector addition operator in Section 1.2, despite that we didn't explicitly specify it.

In addition, we set the optimization level to the highest level 3. You may get warning messages that not every operator is well optimized, you can ignore it for now. We will get back to it later.

```
target = 'llvm'
with relay.build_config(opt_level=3):
    graph, mod, params = relay.build(relay_mod, target, params=relay_params)
```

```
Cannot find config for target=llvm, workload=('dense_nopack.x86', ('TENSOR',
 ↪ (1, 512), 'float32'), ('TENSOR', (1000, 512), 'float32'), None, 'float32
 ↪'). A fallback configuration is used, which may bring great performance␣
 ↪regression.
```

The compiled module has three parts: `graph` is a json string described the neural network, `mod` is a library that contains all compiled operators used to run the inference, and `params` is a dictionary mapping parameter name to weights.

```
type(graph), type(mod), type(params)
```

```
(str, tvm.runtime.module.Module, dict)
```

You can view `mod` as a TVM module we already seen in Section 1.2.

### 1.3.4 Inference

Now we can create a runtime to run the model inference, namely the forward pass of a neural network. Creating the runtime needs the neural network definition in json (i.e. `graph`) and the library that contains machine code of compiled operators (i.e. `mod`), with a device context that can be constructed from the target. The device is CPU here, specified by `llvm`. Next we load the parameters with `set_input` and run the workload by feeding the input data. Since this network has a single output layer, we can obtain it, a `(1, 1000)` shape matrix, by `get_output(0)`. The final output is a 1000-length NumPy vector.

```
ctx = tvm.context(target)
rt = tvm.contrib.graph_runtime.create(graph, mod, ctx)
rt.set_input(**params)
rt.run(data=tvm.nd.array(x))
scores = rt.get_output(0).asnumpy()[0]
scores.shape
```

[10] https://en.wikipedia.org/wiki/LLVM

```
(1000,)
```

The vector contains the predicted confidence score for each class. Note that the pre-trained model doesn't have the softmax[11] operator, so these scores are not mapped into probabilities in (0, 1). Now we can find the two largest scores and report their labels.

```
a = np.argsort(scores)[-1:-5:-1]
labels[a[0]], labels[a[1]]
```

```
('tiger cat', 'Egyptian cat')
```

### 1.3.5 Saving the Compiled Library

We can save the output of `relay.build` in disk to reuse them later. The following code block saves the json string, library, and parameters.

```
!rm -rf resnet18*

name = 'resnet18'
graph_fn, mod_fn, params_fn = [name+ext for ext in ('.json','.tar','.params')]
mod.export_library(mod_fn)
with open(graph_fn, 'w') as f:
    f.write(graph)
with open(params_fn, 'wb') as f:
    f.write(relay.save_param_dict(params))

!ls -alht resnet18*
```

```
-rw-r--r-- 1 jenkins jenkins  45M Oct 13 10:45 resnet18.params
-rw-r--r-- 1 jenkins jenkins  28K Oct 13 10:45 resnet18.json
-rw-r--r-- 1 jenkins jenkins 157K Oct 13 10:45 resnet18.tar
```

Now we load the saved module back.

```
loaded_graph = open(graph_fn).read()
loaded_mod = tvm.runtime.load_module(mod_fn)
loaded_params = open(params_fn, "rb").read()
```

And then construct the runtime as before to verify the results

```
loaded_rt = tvm.contrib.graph_runtime.create(loaded_graph, loaded_mod, ctx)
loaded_rt.load_params(loaded_params)
loaded_rt.run(data=tvm.nd.array(x))
loaded_scores = loaded_rt.get_output(0).asnumpy()[0]
tvm.testing.assert_allclose(loaded_scores, scores)
```

[11] https://en.wikipedia.org/wiki/Softmax_function

### 1.3.6 Summary

- We can use relay of TVM to convert and compile a neural network into a module for model inference.
- We can save the compiled module into disk to facilitate future deployment.

### 1.3.7 Discussions[12]

## 1.4 Running on a Remote Machine

In this book, we will run and optimize programs on various hardware platforms. One way is to log into the machine with the desired hardware, install required packages and then run the workloads there. It, however, makes maintaining the source codes and data difficult, especially when the targeting hardware is with minimal power. In this section, we will describe another solution: running a daemon on the remote machine and then sending the compiled module and input data to it only for execution.

```python
import d2ltvm
import numpy as np
import mxnet as mx
import tvm
from tvm import te, rpc, relay
from PIL import Image
```

Note that we imported the rpc module from TVM. RPC[13], namely remote procedure call, enables executing a program on a remote place.

### 1.4.1 Setup the Remote Machine

We first need to install TVM runtime module on the remote machine. The installation setup is almost identical to TVM (refer to Section 1.1), except that we only need to build the runtime, i.e. make runtime, instead of the whole TVM library. The runtime size is often less than 1MB, which makes it suitable for device with memory constraints. You also need to enable the proper backend, e.g. CUDA or OpenCL, if necessary.

Once the runtime is installed, we can start the daemon by running the following command on the remote machine.

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

It will start an RPC server which binds the local 9090 port to listen. You should see the following output indicating the server has already started.

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

In addition, you need to check two things on the remote machine.

One is the remote machine's IP. On Linux or macOS, you can get it by ifconfig | grep inet. Also remember to open the 9090 port if there is a firewall.

The other one is the target architecture. It's straightforward for GPUs, we will cover it later. For CPUs, the easiest way is installing LLVM on the remote machine and then checking llvm-config --host-target. The return of the remote machine we are using is x86_64-pc-linux-gnu.

[12] https://discuss.tvm.ai/t/getting-started-neural-network-inference/4708

[13] https://en.wikipedia.org/wiki/Remote_procedure_call

This target triplet has the general format <arch><sub>-<vendor>-<sys>-<abi>, where

- arch: x86, x86_64, arm, thumb, mips, etc.
- sub: for ARM, there are v5, v6m, v7a, v7m, v8, etc.
- vendor: pc, apple, nvidia, ibm, etc.
- sys: linux, win32, darwin, cuda, none, unknown, etc.
- abi: eabi, gnu, android, macho, elf, etc.

For example, it's x86_64-apple-darwin17.7.0 for the MacbookPro I'm using, and armv6k-unknown-linux-gnueabihf for the Raspberry Pi 4B.

### 1.4.2 Compile the Program for the Remote Machine

Let's run the vector addition defined Section 1.2 on the remote machine. Note that we specified the remote machine target through the -target argument for LLVM.

```
n = 100
target = 'llvm -target=x86_64-pc-linux-gnu'

args = d2ltvm.vector_add(n)
s = te.create_schedule(args[-1].op)
mod = tvm.build(s, args, target)
```

Then we save the compiled module to disk, which will be uploaded to the remote machine later.

```
mod_fname = 'vector-add.tar'
mod.export_library(mod_fname)
```

### 1.4.3 Evaluate on the Remote Machine

We first connect to the remote machine with the IP we checked before.

```
remote = rpc.connect('172.31.0.149', 9090)
```

Next, we send the compiled library to the machine and load it into the memory of the remote machine.

```
remote.upload(mod_fname)
remote_mod = remote.load_module(mod_fname)
```

When creating the data, we specify the device context as CPU on the remote machine. The data will be created on the local machine as before, but will be sent to the remote machine later. Note that we used NumPy to create the data, but there is no need to have the remote machine also installed NumPy.

```
ctx = remote.cpu()
a, b, c = d2ltvm.get_abc(n, lambda x: tvm.nd.array(x, ctx=ctx))
```

Since both data and library are ready on the remote machine, let's execute the program on it as well.

```
remote_mod(a, b, c)
```

Finally, the `.asnumpy()` method will send the data back to the local machine and convert to a NumPy array. So we can verify the results as before.

```
np.testing.assert_equal(a.asnumpy()+b.asnumpy(), c.asnumpy())
```

### 1.4.4 Running Neural Network Inference

Let's run the ResNet-18 used in Section 1.3 on the remote machine. As before, we load a sample image and Imagenet 1K labels.

```
image = Image.open('../data/cat.jpg').resize((224, 224))
x = d2ltvm.image_preprocessing(image)
with open('../data/imagenet1k_labels.txt') as f:
    labels = eval(f.read())
```

Then we convert, compile and save the module. Note that we just need to save the shared library which contains the machine code of the compiled operators to the disk.

```
mod_fname = 'resnet18.tar'
model = mx.gluon.model_zoo.vision.resnet18_v2(pretrained=True)
relay_mod, relay_params = relay.frontend.from_mxnet(model, {'data': x.shape})
with relay.build_config(opt_level=3):
    graph, mod, params = relay.build(relay_mod, target, params=relay_params)
mod.export_library(mod_fname)
```

```
Cannot find config for target=llvm -target=x86_64-pc-linux-gnu, workload=(
↪'dense_nopack.x86', ('TENSOR', (1, 512), 'float32'), ('TENSOR', (1000, 512),
↪ 'float32'), None, 'float32'). A fallback configuration is used, which may␣
↪bring great performance regression.
```

Next, we upload the saved library to the remote machine and load it into memory. Then we can create a runtime using the model definition, the remote library and the remote context.

```
remote.upload(mod_fname)
remote_mod = remote.load_module(mod_fname)
remote_rt =  tvm.contrib.graph_runtime.create(graph, remote_mod, ctx)
```

The inference is identical to Section 1.3, where both parameters and input are on the local machine. The runtime will upload them into the remote machine properly.

```
remote_rt.set_input(**params)
remote_rt.run(data=tvm.nd.array(x))
scores = remote_rt.get_output(0).asnumpy()[0]
scores.shape
a = np.argsort(scores)[-1:-5:-1]
labels[a[0]], labels[a[1]]
```

```
('tiger cat', 'Egyptian cat')
```

### 1.4.5 Summary

- We can install a TVM runtime on a remote machine to start an RPC server to accept workloads to run.
- A program can be compiled locally with specifying the remote machine's architecture target (called cross-compilation), and then run on the remote machine via RPC.

### 1.4.6 Discussions[14]

[14] https://discuss.tvm.ai/t/getting-started-running-on-a-remote-machine/4709

# 2 | Expressions for Operators

We start from operators. As you may know, operators are the building blocks of neural network models. A deep neural network can be expressed as a Directed Acyclic Graph (DAG), which consists of nodes being the operators and edges being the data dependency between nodes. Being able to execute the operators efficiently is of course a necessity to high-performance neural network model execution.

In Section 1.2 you have seen how to build the vector addition expression in TVM. This chapter covers more concepts in TVM to construct expressions. Specifically, you'll learn about data types, shapes, indexing, reduction and control flow, based on which you'll be able to construct operators in the next chapter.

## 2.1 Data Types

Every tensor has a data type, which is typically `float32` in deep learning, but also could be `int8` (e.g. for model quantization) and others. The `tvm_vector_add` module we created in Section 1.2 only accepts `float32` tensors. Let's extend it to other data types in this section.

### 2.1.1 Specifying a Data Type

To use a data type different to the default `float32`, we can specify it explicitly when creating the placeholders. In the following code block, we generalize the vector addition expression defined in Section 1.2 to accept an argument `dtype` to specify the data type. In particular, we pass `dtype` to `te.placeholder` when creating `A` and `B`. The result `C` then obtains the same data type as `A` and `B`.

```python
import tvm
from tvm import te
import d2ltvm
import numpy as np

n = 100

def tvm_vector_add(dtype):
    A = te.placeholder((n,), dtype=dtype)
    B = te.placeholder((n,), dtype=dtype)
    C = te.compute(A.shape, lambda i: A[i] + B[i])
    print('expression dtype:', A.dtype, B.dtype, C.dtype)
    s = te.create_schedule(C.op)
    return tvm.build(s, [A, B, C])
```

Let's compile a module that accepts `int32` tensors.

```
mod = tvm_vector_add('int32')
```

```
expression dtype: int32 int32 int32
```

Then we define a method to verify the results with a particular data type. Note that we pass a constructor that modifies the tensor data type by `astype`.

```
def test_mod(mod, dtype):
    a, b, c = d2ltvm.get_abc(n, lambda x: tvm.nd.array(x.astype(dtype)))
    print('tensor dtype:', a.dtype, b.dtype, c.dtype)
    mod(a, b, c)
    np.testing.assert_equal(c.asnumpy(), a.asnumpy() + b.asnumpy())

test_mod(mod, 'int32')
```

```
tensor dtype: int32 int32 int32
```

Let's try other data types as well

```
for dtype in ['float16', 'float64', 'int8','int16', 'int64']:
    mod = tvm_vector_add(dtype)
    test_mod(mod, dtype)
```

```
expression dtype: float16 float16 float16
tensor dtype: float16 float16 float16
expression dtype: float64 float64 float64
tensor dtype: float64 float64 float64
expression dtype: int8 int8 int8
tensor dtype: int8 int8 int8
expression dtype: int16 int16 int16
tensor dtype: int16 int16 int16
expression dtype: int64 int64 int64
tensor dtype: int64 int64 int64
```

### 2.1.2 Converting Elements Data Types

Besides constructing a tensor with a particular data type, we can also cast the data type of a tensor element during the computation. The following method is the same as `tvm_vector_add` except that it casts the data type of A and B in `te.compute`, leaving the data type defined in `te.placeholder` as default (`float32`). Because of the casting done by `astype`, the result `C` will have the data type specified by `dtype`.

```
def tvm_vector_add_2(dtype):
    A = te.placeholder((n,))
    B = te.placeholder((n,))
    C = te.compute(A.shape,
                   lambda i: A[i].astype(dtype) + B[i].astype(dtype))
    print('expression dtype:', A.dtype, B.dtype, C.dtype)
    s = te.create_schedule(C.op)
    return tvm.build(s, [A, B, C])
```

Then we define a similar test function to verify the results.

```
def test_mod_2(mod, dtype):
    a, b, c = d2ltvm.get_abc(n)
    # by default `get_abc` returns NumPy ndarray in float32
    a_tvm, b_tvm = tvm.nd.array(a), tvm.nd.array(b)
    c_tvm = tvm.nd.array(c.astype(dtype))
    print('tensor dtype:', a_tvm.dtype, b_tvm.dtype, c_tvm.dtype)
    mod(a_tvm, b_tvm, c_tvm)
    np.testing.assert_equal(c_tvm.asnumpy(), a.astype(dtype) + b.
↪astype(dtype))

mod = tvm_vector_add_2('int32')
test_mod_2(mod, 'int32')
```

```
expression dtype: float32 float32 int32
tensor dtype: float32 float32 int32
```

### 2.1.3 Summary

- We can specify the data type by `dtype` when creating TVM placeholders.
- The data type of a tensor element can be cast by `astype` in TVM compute.

## 2.2 Shapes

The vector addition module defined in Section 1.2 only accepts vectors with 100-length. It's too restrictive for real scenarios where inputs can have arbitrary shapes. In this section, we will show how to relax this constraint to deal with general cases.

### 2.2.1 Variable Shapes

Remember that we create symbolic placeholders for tensors `A` and `B` so we can feed with data later. We can do the same thing for the shape as well. In particular, the following code block uses `te.var` to create a symbolic variable for an `int32` scalar, whose value can be specified later.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te

n = te.var(name='n')
type(n), n.dtype
```

```
(tvm.tir.expr.Var, 'int32')
```

Now we can use `(n,)` to create a placeholder for an arbitrary length vector.

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

```
produce c {
  for (i, 0, n) {
    c[(i*stride)] = (a[(i*stride)] + b[(i*stride)])
  }
}
```

Compared to the generated pseudo codes in Section 1.2, we can see the upper bound value of the for loop is changed from 100 to `n`.

Now we define a similar test function as before to verify that the compiled module is able to correctly execute on input vectors with different lengths.

```
def test_mod(mod, n):
    a, b, c = d2ltvm.get_abc(n, tvm.nd.array)
    mod(a, b, c)
    print('c.shape:', c.shape)
    np.testing.assert_equal(c.asnumpy(), a.asnumpy() + b.asnumpy())

mod = tvm.build(s, [A, B, C])
test_mod(mod, 5)
test_mod(mod, 1000)
```

```
c.shape: (5,)
c.shape: (1000,)
```

But note that we still place the constraint that `A`, `B`, and `C` must be in the same shape. So an error will occur if it is not satisfied.

### 2.2.2 Multi-dimensional Shapes

You may already notice that a shape is presented as a tuple. A single element tuple means a 1-D tensor, or a vector. We can extend it to multi-dimensional tensors by adding variables to the shape tuple.

The following method builds a module for multi-dimensional tensor addition, the number of dimensions is specified by `ndim`. For a 2-D tensor, we can access its element by `A[i,j]`, similarly `A[i,j,k]` for 3-D tensors. Note that we use `*i` to handle the general multi-dimensional case in the following code.

```
def tvm_vector_add(ndim):
    A = te.placeholder([te.var() for _ in range(ndim)])
    B = te.placeholder(A.shape)
    C = te.compute(A.shape, lambda *i: A[i] + B[i])
    s = te.create_schedule(C.op)
    return tvm.build(s, [A, B, C])
```

Verify that it works beyond vectors.

```
mod = tvm_vector_add(2)
test_mod(mod, (2, 2))

mod = tvm_vector_add(4)
test_mod(mod, (2, 3, 4, 5))
```

```
c.shape: (2, 2)
c.shape: (2, 3, 4, 5)
```

### 2.2.3 Summary

- We can use `te.var()` to specify the dimension(s) of a shape when we don't know the concrete data shape before execution.
- The shape of an $n$-dimensional tensor is presented as an $n$-length tuple.

## 2.3 Index and Shape Expressions

You already know that a shape can be a tuple of symbols such as `(n, m)` and the elements can be accessed via indexing, e.g. `a[i, j]`. In practice, both shapes and indices may be computed through complex expressions. We will go through several examples in this section.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
```

### 2.3.1 Matrix Transpose

Our first example is matrix transpose `a.T`, in which we access `a`'s elements by columns.

```
n = te.var('n')
m = te.var('m')
A = te.placeholder((n, m), name='a')
B = te.compute((m, n), lambda i, j: A[j, i], 'b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  for (i, 0, m) {
    for (j, 0, n) {
      b[((i*stride) + (j*stride))] = a[((j*stride) + (i*stride))]
    }
  }
}
```

Note that the 2-D index, e.g. `b[i,j]` are collapsed to the 1-D index `b[((i*n) + j)]` to follow the C convention.

Now verify the results.

```
a = np.arange(12, dtype='float32').reshape((3, 4))
b = np.empty((4, 3), dtype='float32')
a, b = tvm.nd.array(a), tvm.nd.array(b)

mod = tvm.build(s, [A, B])
mod(a, b)
print(a)
print(b)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
[[ 0.  4.  8.]
 [ 1.  5.  9.]
 [ 2.  6. 10.]
 [ 3.  7. 11.]]
```

### 2.3.2 Reshaping

Next let's use expressions for indexing. The following code block reshapes a 2-D array `a` (`n` by `m` as defined above) to 1-D (just like `a.reshape(-1)` in NumPy). Note how we convert the 1-D index `i` to the 2-D index `[i//m, i%m]`.

```
B = te.compute((m*n, ), lambda i: A[i//m, i%m], name='b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  for (i, 0, (m*n)) {
    b[i] = a[((floordiv(i, m)*stride) + (floormod(i, m)*stride))]
  }
}
```

Since an $n$-D array is essentially listed in the memory as a 1-D array, the generated code does not rearrange the data sequence, but it simplifies the index expression from 2-D (`(i//m)*m + i%m`) to 1-D (`i`) to improve the efficiency.

We can implement a general 2-D reshape function as well.

```
p, q = te.var('p'), te.var('q')
B = te.compute((p, q), lambda i, j: A[(i*q+j)//m, (i*q+j)%m], name='b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  for (i, 0, p) {
    for (j, 0, q) {
      b[((i*stride) + (j*stride))] = a[((floordiv(((i*q) + j), m)*stride) +␣
↪(floormod(((i*q) + j), m)*stride))]
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

When testing the results, we should be aware that we put no constraint on the output shape, which can have an arbitrary shape `(p, q)`, and therefore TVM will not be able to check if $qp = nm$ for us. For example, in the following example we created a `b` with size (20) larger than `a` (12), then only the first 12 elements in `b` are from `a`, others are uninitialized values.

```
mod = tvm.build(s, [A, B])
a = np.arange(12, dtype='float32').reshape((3,4))
b = np.zeros((5, 4), dtype='float32')
a, b = tvm.nd.array(a), tvm.nd.array(b)

mod(a, b)
print(b)
```

```
[[0.000000e+00 1.000000e+00 2.000000e+00 3.000000e+00]
 [4.000000e+00 5.000000e+00 6.000000e+00 7.000000e+00]
 [8.000000e+00 9.000000e+00 1.000000e+01 1.100000e+01]
 [8.722636e-23 3.066461e-41 9.108440e-44 0.000000e+00]
 [8.743578e-23 3.066461e-41 8.741932e-23 3.066461e-41]]
```

### 2.3.3 Slicing

Now let's consider a special slicing operator `a[bi::si, bj::sj]` where `bi`, `bj`, `si` and `sj` can be specified later. Now the output shape needs to be computed based on the arguments. In addition, we need to pass the variables `bi`, `bj`, `si` and `sj` as arguments when compiling the module.

```
bi, bj, si, sj = [te.var(name) for name in ['bi', 'bj', 'si', 'sj']]
B = te.compute(((n-bi)//si, (m-bj)//sj), lambda i, j: A[i*si+bi, j*sj+bj],␣
↪name='b')
s = te.create_schedule(B.op)
mod = tvm.build(s, [A, B, bi, si, bj, sj])
```

Now test two cases to verify the correctness.

```
b = tvm.nd.array(np.empty((1, 3), dtype='float32'))
mod(a, b, 1, 2, 1, 1)
np.testing.assert_equal(b.asnumpy(), a.asnumpy()[1::2, 1::1])

b = tvm.nd.array(np.empty((1, 2), dtype='float32'))
mod(a, b, 2, 1, 0, 2)
np.testing.assert_equal(b.asnumpy(), a.asnumpy()[2::1, 0::2])
```

### 2.3.4 Summary

- Both shape dimensions and indices can be expressions with variables.
- If a variable doesn't only appear in the shape tuple, we need to pass it as an argument when compiling.

## 2.4 Reduction Operations

Reduction is an operation to reduce certain dimension(s) of an input tensor, usually to scalar(s), e.g. `np.sum` in NumPy. Reduction is often straightforward to implement with for-loops. But it's a little bit more complicated in TVM since we cannot use a Python for-loop directly. In this section, we will describe how to implement reduction in TVM.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
```

### 2.4.1 Sum

Let's start with summing the rows of a 2-D matrix to reduce it to be a 1-D vector. In NumPy, we can do it with the `sum` method.

```
a = np.random.normal(size=(3,4)).astype('float32')
a.sum(axis=1)
```

```
array([ 1.8948135, -2.4319794,  1.9638997], dtype=float32)
```

As we did before, let's implement this operation from scratch to help understand the TVM expression.

```
def sum_rows(a, b):
    """a is an n-by-m 2-D matrix, b is an n-length 1-D vector
    """
    n = len(b)
    for i in range(n):
        b[i] = np.sum(a[i,:])

b = np.empty((3,), dtype='float32')
sum_rows(a, b)
b
```

```
array([ 1.8948135, -2.4319794,  1.9638997], dtype=float32)
```

It's fairly straightforward, we first iterate on the first dimension, `axis=0`, and then sum all elements on the second dimension to write the results. In NumPy, we can use `:` to slice all elements along that dimension.

Now let's implement the same thing in TVM. Comparing to the vector addition in Section 1.2, we used two new operators here. One is `tvm.reduce_axis`, which create an axis for reduction with range from 0 to `m`. It's functionally similar to the `:` used in `sum_rows`, but we need to explicitly specify the range in TVM. The other one is `tvm.sum`, which sums all elements along the reducing axis `k` and returns a scalar.

```
n, m = te.var('n'), te.var('m')
A = te.placeholder((n, m), name='a')
j = te.reduce_axis((0, m), name='j')
B = te.compute((n,), lambda i: te.sum(A[i, j], axis=j), name='b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  for (i, 0, n) {
    b[(i*stride)] = 0f
    for (j, 0, m) {
      b[(i*stride)] = (b[(i*stride)] + a[((i*stride) + (j*stride))])
    }
  }
}
```

We can see that the generated pseudo codes expand `tvm.sum` into another for loop along axis `k`. As mentioned before, the pseudo codes are C-like, so the index of `a[i,j]` is expanded to `(i*m)+j` by treating `a` as a 1-D array. Also note that `b` is initialized to be all-zero before summation.

Now test the results are as expected.

```
mod = tvm.build(s, [A, B])
c = tvm.nd.array(np.empty((3,), dtype='float32'))
mod(tvm.nd.array(a), c)
np.testing.assert_equal(b, c.asnumpy())
```

We know that `a.sum()` will sum all elements in `a` and returns a scalar. Let's also implement this in TVM. To do it, we need another reduction axis along the first dimension, whose size is `n`. The result is a scalar, namely a 0-rank tensor, can be created with an empty tuple `()`.

```
i = te.reduce_axis((0, n), name='i')
B = te.compute((), lambda: te.sum(A[i, j], axis=(i, j)), name='b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  b[0] = 0f
  for (i, 0, n) {
    for (j, 0, m) {
      b[0] = (b[0] + a[((i*stride) + (j*stride))])
    }
  }
}
```

Let's also verify the results.

```
mod = tvm.build(s, [A, B])
c = tvm.nd.array(np.empty((), dtype='float32'))
mod(tvm.nd.array(a), c)
np.testing.assert_allclose(a.sum(), c.asnumpy(), atol=1e-5)
```

In this case we use `np.testing.assert_allclose` instead of `np.testing.assert_equal` to verify the results as the calculation on `float32` numbers may differ due to the numerical error.

Beyond `tvm.sum`, there are other reduction operators in TVM such as `tvm.min` and `tvm.max`. We can also use them to implement the corresponding reduction operations as well.

### 2.4.2 Commutative Reduction

In mathematics, an operator $\circ$ is commutative if $a \circ b = b \circ a$. TVM allows to define a customized commutative reduction operator through `tvm.comm_reducer`. It accepts two function arguments, one defines how to compute $a \circ b$, the other one specifies the initial value.

Let's use the production by rows, e.g `a.prod(axis=1)`, as an example. Again, we first show how to implement it from scratch.

```
def prod_rows(a, b):
    """a is an n-by-m 2-D matrix, b is an n-length 1-D vector
    """
    n, m = a.shape
    for i in range(n):
        b[i] = 1
        for j in range(m):
            b[i] = b[i] * a[i, j]
```

As can be seen, we need to first initialize the return values to be 1, and then compute the reduction using scalar product `*`. Now let's define these two functions in TVM to serve as the arguments of `te.comm_reducer`. As discussed, the first one defines $a \circ b$ with two scalar inputs. The second one accepts a data type argument to return the initial value of an element. Then we can create the reduction operator.

```
comp = lambda a, b: a * b
init = lambda dtype: tvm.tir.const(1, dtype=dtype)
product = te.comm_reducer(comp, init)
```

The usage of `product` is similar to `te.sum`. Actually, `te.sum` is a pre-defined reduction operator using the same way.

```
n = te.var('n')
m = te.var('m')
A = te.placeholder((n, m), name='a')
k = te.reduce_axis((0, m), name='k')
B = te.compute((n,), lambda i: product(A[i, k], axis=k), name='b')
s = te.create_schedule(B.op)
tvm.lower(s, [A, B], simple_mode=True)
```

```
produce b {
  for (i, 0, n) {
    b[(i*stride)] = 1f
    for (k, 0, m) {
      b[(i*stride)] = (b[(i*stride)]*a[((i*stride) + (k*stride))])
    }
  }
}
```

The generated pseudo codes are similar to the one for summing by rows, except for the initialized value and the reduction arithmetic.

Again, let's verify the results.

```
mod = tvm.build(s, [A, B])
b = tvm.nd.array(np.empty((3,), dtype='float32'))
mod(tvm.nd.array(a), b)
np.testing.assert_allclose(a.prod(axis=1), b.asnumpy(), atol=1e-5)
```

### 2.4.3 Summary

- We can apply a reduction operator, e.g. `te.sum` over a reduction axis `te.reduce_axis`.
- We can implement customized commutative reduction operators by `te.comm_reducer`.

## 2.5 Conditional Expression: `if-then-else`

The `if-then-else` statement is supported through `te.if_then_else`. In this section, we will introduce this expression using computing the lower triangle of an matrix as the example.

```
import tvm
from tvm import te
import numpy as np
import d2ltvm
```

In NumPy, we can easily use `np.tril` to obtain the lower triangle.

```
a = np.arange(12, dtype='float32').reshape((3, 4))
np.tril(a)
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  5.,  0.,  0.],
       [ 8.,  9., 10.,  0.]], dtype=float32)
```

Now let's implement it in TVM with `if_then_else`. It accepts three arguments, the first one is the condition, if true returning the second argument, otherwise returning the third one.

```
n, m = te.var('n'), te.var('m')
A = te.placeholder((n, m))
B = te.compute(A.shape, lambda i, j: te.if_then_else(i >= j, A[i,j], 0.0))
```

Verify the results.

```
b = tvm.nd.array(np.empty_like(a))
s = te.create_schedule(B.op)
print(tvm.lower(s, [A, B], simple_mode=True))
mod = tvm.build(s, [A, B])
mod(tvm.nd.array(a), b)
b
```

```
produce compute {
  for (i, 0, n) {
    for (j, 0, m) {
      compute[((i*stride) + (j*stride))] = tvm_if_then_else((j <= i),␣
↪placeholder[((i*stride) + (j*stride))], 0f)
    }
  }
}
```

```
<tvm.nd.NDArray shape=(3, 4), cpu(0)>
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  5.,  0.,  0.],
       [ 8.,  9., 10.,  0.]], dtype=float32)
```

### 2.5.1 Summary

- We can use `tvm.if_then_else` for the if-then-else statement.

## 2.6 Truth Value Testing: `all` and `any`

In Python, we can use `all` and `any` to get the boolean return of a list of values. `all` returns the logical `and` result while `any` returns the logical `or` result.

```
import numpy as np
import d2ltvm
import tvm
from tvm import te

any((0, 1, 2)), all((0, 1, 2))
```

```
(True, False)
```

TVM provides similar `te.all` and `te.any`, which are useful to construct complex conditional expression for `te.if_then_else`.

The example we will use is padding the matrix `a` with 0s.

```
a = np.ones((3, 4), dtype='float32')
# applying a zero padding of size 1 to a
b = np.zeros((5, 6), dtype='float32')
b[1:-1,1:-1] = a
print(b)
```

```
[[0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

Now let's implement it in TVM. Note that we pass the four condition values into `tvm.any`.

```
p = 1 # padding size
n, m = te.var('n'), te.var('m')
A = te.placeholder((n, m), name='a')
B = te.compute((n+p*2, m+p*2),
                lambda i, j: te.if_then_else(
                    te.any(i<p, i>=n+p, j<p, j>=m+p), 0, A[i-p, j-p]),
                name='b')
```

Verify the results.

```
s = te.create_schedule(B.op)
mod = tvm.build(s, [A, B])
c = tvm.nd.array(np.empty_like(b))
mod(tvm.nd.array(a), c)
print(c)
```

```
[[0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

### 2.6.1 Summary

- We can use `tvm.any` and `tvm.all` to construct complex conditional expressions.

# 3 | Common Operators

In last chapter, we went over how to implement the basic expressions/operators using tvm operators. This chapter will further describe how to implement the typical operators we will encounter in the deep learning models.

## 3.1 Broadcast Add

A broadcast operator process two tensors in different shapes. Normally, one of the operands has a particular dimension to be 1, which will be broadcast along the corresponding dimension of the other operator to perform the given calculation. Common scalar calculations can all be broadcast, such as elementary arithmetic and logical operations. Fig. 3.1.1 illustrates one broadcast add case between two 2-dimensional tensors. Broadcast operators are commonly seen in deep learning workloads, e.g. batch normalization[15].

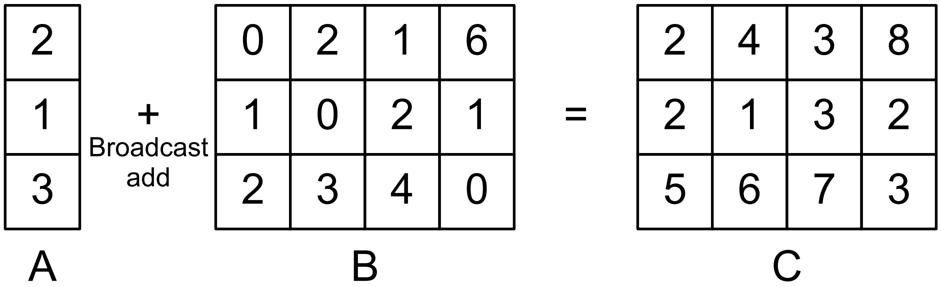


Fig. 3.1.1: One case of broadcast add between two 2-dimensional tensors

In this section we will demonstrate how to perform a broadcast add between two 2-dimensional tensors. The following code defines the computation.

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

# Save to the d2ltvm package.
def broadcast_add(shape1, shape2):
    """Broadcast add between two 2-dimensional tensors

    shape1, shape2 : the shapes of the input tensors
    """
```

(continues on next page)

[15] http://d2l.ai/chapter_convolutional-modern/batch-norm.html

(continued from previous page)

```
    assert len(shape1) == 2 and len(shape2) == 2, \
        "broadcast tensors should both be 2-dimension"
    for i in range(len(shape1)):
        assert shape1[i] == shape2[i] or shape1[i] == 1 or shape2[i] == 1, \
            "tensor shapes do not fit for broadcasting"
    A = te.placeholder(shape1, name='A')
    B = te.placeholder(shape2, name='B')
    m = shape1[0] if shape2[0] == 1 else shape2[0]
    n = shape1[1] if shape2[1] == 1 else shape2[1]
    f = lambda x, y: A[0 if shape1[0]==1 else x, 0 if shape1[1]==1 else y] + \
        B[0 if shape2[0]==1 else x, 0 if shape2[1]==1 else y]
    C = te.compute((m, n), f, name='C')
    return A, B, C
```

Then we use it to perform the broadcast add illustrated in Fig. 3.1.1.

```
m = 3
n = 4
shape1 = (m, 1)
shape2 = (m, n)
A, B, C = broadcast_add(shape1, shape2)
s = te.create_schedule(C.op)
print(tvm.lower(s, [A, B], simple_mode=True))
mod = tvm.build(s, [A, B, C])
```

```
// attr [C] storage_scope = "global"
allocate C[float32 * 12]
produce C {
  for (x, 0, 3) {
    for (y, 0, 4) {
      C[((x*4) + y)] = (A[x] + B[((x*4) + y)])
    }
  }
}
```

The printed pseudocode clearly depicts the process of a broadcast add. We verify the results as follows.

```
# Save to the d2ltvm package.
def get_bcast_data(shape1, shape2, constructor=None):
    """Return random tensors a, b
    and empty tensor c to store broadcast results between a and b

    shape1, shape2: shapes of input tensors
    constructor : user-defined tensor constructor
    """
    np.random.seed(0)
    a = np.random.normal(size=shape1).astype("float32")
    b = np.random.normal(size=shape2).astype("float32")
    out_shape = (shape1[0] if shape2[0] == 1 else shape2[0],
                 shape1[1] if shape2[1] == 1 else shape2[1])
    c = np.empty(out_shape, dtype='float32')
    if constructor:
        a, b, c = [constructor(x) for x in (a, b, c)]
```

(continues on next page)

(continued from previous page)

```
    return a, b, c
a, b, c = get_bcast_data(shape1, shape2, tvm.nd.array)
mod(a, b, c)
np.testing.assert_allclose(np.add(a.asnumpy(), b.asnumpy()), c.asnumpy(),␣
↪atol=1e-5)
```

Note that broadcast is allowed to perform along multiple dimensions.

```
shape1 = (m, 1)
shape2 = (1, n)
A, B, C = broadcast_add(shape1, shape2)
s = te.create_schedule(C.op)
mod = tvm.build(s, [A, B, C])
a, b, c = get_bcast_data(shape1, shape2, tvm.nd.array)
mod(a, b, c)
np.testing.assert_allclose(np.add(a.asnumpy(), b.asnumpy()), c.asnumpy(),␣
↪atol=1e-5)
print(a.shape, b.shape, c.shape)
```

```
(3, 1) (1, 4) (3, 4)
```

Lastly, it is easy to note that when the shapes of two input tensors are indentical, the broadcast add reduces to an element-wise add.

### 3.1.1 Summary

- We can define a broadcast operator in TVM.
- Broadcast be can performed along multiple dimensions.

EE Exercise - Generalize `broadcast_add` defined above to more dimensions and more operators.

## 3.2 Matrix Multiplication

Matrix Multiplication is one of the most widely operators in scientific computing and deep learning, which is typically referred to as *GEMM* (GEneral Matrix Multiply). Let's implement its computation in this section.

Given $A \in \mathbb{R}^{n\times l}$, and $B \in \mathbb{R}^{l\times m}$, if $C = AB$ then $C \in \mathbb{R}^{n\times m}$ and

$$C_{i,j} = \sum_{k=1}^{l} A_{i,k} B_{k,j}. \tag{3.2.1}$$

The elements accessed to compute $C_{i,j}$ are illustrated in Fig. 3.2.1.

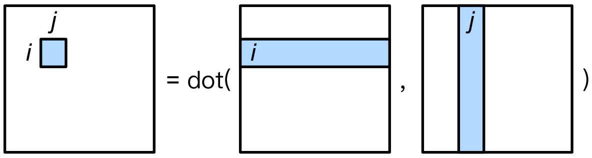


Fig. 3.2.1: Compute $C_{x,y}$ in matrix multiplication.

The following method returns the computing expression of matrix multiplication.

```python
import d2ltvm
import numpy as np
import tvm
from tvm import te

# Save to the d2ltvm package
def matmul(n, m, l):
    """Return the computing expression of matrix multiplication
    A : n x l matrix
    B : l x m matrix
    C : n x m matrix with C = A B
    """
    k = te.reduce_axis((0, l), name='k')
    A = te.placeholder((n, l), name='A')
    B = te.placeholder((l, m), name='B')
    C = te.compute((n, m),
                    lambda x, y: te.sum(A[x, k] * B[k, y], axis=k),
                    name='C')
    return A, B, C
```

Let's compile a module for a square matrix multiplication.

```python
n = 100
A, B, C = matmul(n, n, n)
s = te.create_schedule(C.op)
print(tvm.lower(s, [A, B], simple_mode=True))
mod = tvm.build(s, [A, B, C])
```

```
// attr [C] storage_scope = "global"
allocate C[float32 * 10000]
produce C {
  for (x, 0, 100) {
    for (y, 0, 100) {
      C[((x*100) + y)] = 0f
      for (k, 0, 100) {
        C[((x*100) + y)] = (C[((x*100) + y)] + (A[((x*100) + k)]*B[((k*100) +␣
↪y)]))
      }
    }
  }
}
```

The pseudo code is simply a naive 3-level nested for loop to calculate the matrix multiplication.

And then we verify the results. Note that NumPy may use multi-threading to accelerate its computing, which may result in slightly different results due to the numerical error. There we use `assert_allclose` with a relative large tolerant error to test the correctness.

```python
a, b, c = d2ltvm.get_abc((100, 100), tvm.nd.array)
mod(a, b, c)
np.testing.assert_allclose(np.dot(a.asnumpy(), b.asnumpy()),
                           c.asnumpy(), atol=1e-5)
```

### 3.2.1 Summary

- We can express the computation of matrix multiplication in TVM in one line of code.
- The naive matrix multiplication is a 3-level nested for loop.

## 3.3 Convolution

The convolution (*CONV*) operator is the one of the most expensive and popular operators in neural networks. In this section, we will cover the operator with single input and output channels. Please refer to chapter 6.2[16], 6.3[17], and 6.4[18] in D2L for more explanation about this operator. Here we would not explain much about the convolution-related terms such as padding, channel, stride, convolution kernel, etc.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
```

### 3.3.1 Padding

As a prerequisite to convolution, let's first implement *padding*, which visually surrounds the targeting tensor with a "shell" surrounding it. The padding values are normally 0. Note that we briefly touched padding in Section 2.6 when introducing `te.any`, which was a padding for a 2-D matrix. Here we generalize the padding to work for 2-D convolution on $n$-D tensors, which is usually used in the convolution operators of neural networks. In the general case, we assume the last two dimensions are rows and columns, 0s are only padded on these two dimensions. In particular, if the matrix height (i.e. number of rows) is $n_h$ and width (i.e. number of columns) is $n_w$, then we will pad $p_h$ rows with 0s on top and bottom, and $p_w$ columns with 0s on left and right to make its height and width to $n_h + 2p_h$ and $n_w + 2p_w$, respectively. We have mentioned it once in Section 2.2, but again note that `*X` and `*i` in `te.compute` are used to represent general multi-dimensional tensors.

```
# Save to the d2ltvm package.
def padding(X, ph, pw, val=0):
    """Pad X with the given value in 2-D

    ph, pw : height and width padding
    val : padding value, default 0
    """
    assert len(X.shape) >= 2
    nh, nw = X.shape[-2], X.shape[-1]
    return te.compute(
            (*X.shape[0:-2], nh+ph*2, nw+pw*2),
            lambda *i: te.if_then_else(
                te.any(i[-2]<ph, i[-2]>=nh+ph, i[-1]<pw, i[-1]>=nw+pw),
                val, X[i[:-2]+(i[-2]-ph, i[-1]-pw)]),
            name='PaddedX')
```

Verify the results for a 3-D tensor.

[16] http://numpy.d2l.ai/chapter_convolutional-neural-networks/conv-layer.html
[17] http://numpy.d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html
[18] http://numpy.d2l.ai/chapter_convolutional-neural-networks/channels.html

```
A = te.placeholder((2,3,4))
B = padding(A, 1, 2)
s = te.create_schedule(B.op)
mod = tvm.build(s, [A, B])

a = tvm.nd.array(np.ones((2,3,4), dtype='float32'))
b = tvm.nd.array(np.empty((2,5,8), dtype='float32'))
mod(a, b)
print(b)
```

```
[[[0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 1. 1. 1. 1. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0.]]]
```

### 3.3.2 Convolution

We consider the simple single-channel convolution first. Given an $n_h \times n_w$ data matrix $X$, we first pad 0s into $(n_h + 2p_h) \times (n_w + 2p_w)$. If the kernel matrix $K$ has a size of $k_h \times k_w$, using a stride $s_h$ for height and $s_w$ for width, the output $Y = X \star K$ will have a shape

$$\lfloor (n_h - k_h + 2p_h)/s_h + 1 \rfloor \times \lfloor (n_w - k_w + 2p_w)/s_w + 1 \rfloor. \tag{3.3.1}$$

And the element of $Y$ can be computed $Y_{i,j}$ by

$$Y_{i,j} = \sum_{a=0}^{k_w-1} \sum_{b=0}^{k_h-1} X_{i s_w + a, j s_h + b} K_{a,b} \tag{3.3.2}$$

An example is illustrated in Fig. 3.3.1.

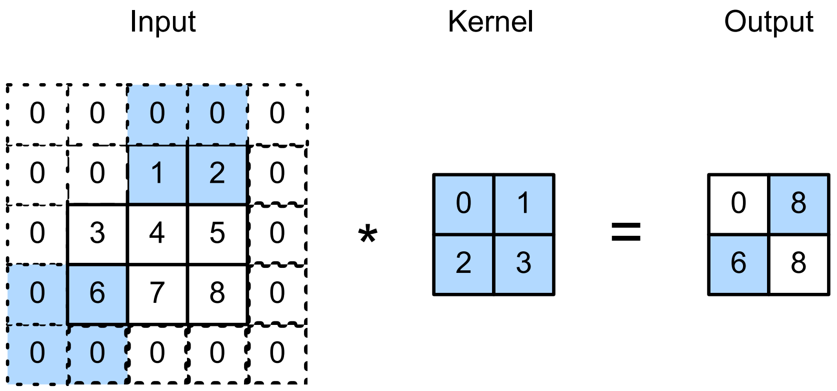


Fig. 3.3.1: The 2-D convolution with paddings for 1, and strides of 3 and 2 for height and width respectively. The shaded portions depicts the two output elements, with the corresponding input and kernel array elements used to calculate them: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

Now let's look at a more general case with multiple channels. Assuming that we have a $c_i \times n_h \times n_w$ 3-D input tensor $X$, and a $c_o \times c_i \times k_h \times k_w$ 4-D kernel tensor $K$, here $c_i$ and $c_o$ are the numbers of input channels and output channels, respectively. Then the output $Y$ has a shape

$$c_o \times \lfloor (h - k_h + 2p_h)/s_h + 1\rfloor \times \lfloor (w - k_w + 2p_w)/s_w + 1\rfloor. \tag{3.3.3}$$

In particular, the $i$-th 2-D matrix $Y_i$, $i = 1, \ldots, c_o$, is computed by

$$Y_i = \sum_{j=1}^{n} X_j \star K_{i,j}, \tag{3.3.4}$$

where $K_{i,j}$ is the 2-D kernel matrix with output channel $i$ and input channel $j$.

In deep learning workloads, especially training, we often concatenate multiple inputs into a batch to process together. A batch of inputs has the shape $n \times c_i \times n_h \times n_w$, where $n$ is the batch size. Applying convolution to a batch means running convolution on the $n$ 3-D tensors separately, and then concatenates results into a 4-D tensor with the first dimension size being $n$.

Note that the input layout we used here is called `NCHW`, which means the 4 dimensions of the input tensors are batch, channel, height and width, respectively. conventionally, `NCHW` means the data is arranged in the memory with `N` being the outer most dimension and `W` being the inner most dimension. Sometimes we use other data layouts such as `NHWC` which may offer a higher performance. We will discuss this in detail later. Similarly, the kernel layout is defined as `KCRS`, which correspond to output channel, input channel, kernel height and width.

Before implementing the convolution, we define a method to calculate the output width or height given the input width or height.

```python
# Save to the d2ltvm package.
def conv_out_size(n, k, p, s):
    """Compute the output size by given input size n (width or height),
    kernel size k, padding p, and stride s
    Return output size (width or height)
    """
    return (n - k + 2 * p)//s + 1
```

Now let's implement the convolution. For simplicity we only consider the single batch case, i.e. `N=1`. In this case, the input data layout can be treated as `CHW`.

```python
# Save to the d2ltvm package.
def conv(oc, ic, nh, nw, kh, kw, ph=0, pw=0, sh=1, sw=1):
    """Convolution

    oc, ic : output and input channels
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding sizes, default 0
    sh, sw : height and width strides, default 1
    """
    # reduction axes
    ric = te.reduce_axis((0, ic), name='ric')
    rkh = te.reduce_axis((0, kh), name='rkh')
    rkw = te.reduce_axis((0, kw), name='rkw')
    # output height and width
    oh = conv_out_size(nh, kh, ph, sh)
    ow = conv_out_size(nw, kw, pw, sw)
```

(continues on next page)

(continued from previous page)

```
    # pad X and then compute Y
    X = te.placeholder((ic, nh, nw), name='X')
    K = te.placeholder((oc, ic, kh, kw), name='K')
    PaddedX = padding(X, ph, pw) if ph * pw != 0 else X
    Y = te.compute(
        (oc, oh, ow),
        lambda c, i, j: te.sum(
            PaddedX[ric, i*sh+rkh, j*sw+rkw] * K[c, ric, rkh, rkw],
            axis=[ric, rkh, rkw]), name='Y')
    return X, K, Y, PaddedX
```

Just as what we created get_abc in Section 1.2, we define a method to get the input and output tensors. Again, we fix the random seed so it returns the same results if calling multiple times.

```
def get_conv_data(oc, ic, n, k, p=0, s=1, constructor=None):
    """Return random 3-D data tensor, 3-D kernel tenor and empty 3-D output
    tensor with the shapes specified by input arguments.

    oc, ic : output and input channels
    n : input width and height
    k : kernel width and height
    p : padding size, default 0
    s : stride, default 1
    constructor : user-defined tensor constructor
    """
    np.random.seed(0)
    data = np.random.normal(size=(ic, n, n)).astype('float32')
    weight = np.random.normal(size=(oc, ic, k, k)).astype('float32')
    on = conv_out_size(n, k, p, s)
    out = np.empty((oc, on, on), dtype='float32')
    if constructor:
        data, weight, out = (constructor(x) for x in [data, weight, out])
    return data, weight, out
```

Now compile a module and compute the results.

```
oc, ic, n, k, p, s = 4, 6, 12, 3, 1, 1
X, K, Y, _ = conv(oc, ic, n, n, k, k, p, p, s, s)
sch = te.create_schedule(Y.op)
mod = tvm.build(sch, [X, K, Y])
print(tvm.lower(sch, [X, K, Y], simple_mode=True))

data, weight, out = get_conv_data(oc, ic, n, k, p, s, tvm.nd.array)
mod(data, weight, out)
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 1176]
produce PaddedX {
  for (i0, 0, 6) {
    for (i1, 0, 14) {
      for (i2, 0, 14) {
        PaddedX[(((i0*196) + (i1*14)) + i2)] = tvm_if_then_else((((((i1 < 1)␣
↪|| (13 <= i1)) || (i2 < 1)) || (13 <= i2)), 0f, X[((((i0*144) + (i1*12)) +␣
↪i2) - 13)])
```

(continues on next page)

(continued from previous page)

```
      }
    }
  }
}
produce Y {
  for (c, 0, 4) {
    for (i, 0, 12) {
      for (j, 0, 12) {
        Y[(((c*144) + (i*12)) + j)] = 0f
        for (ric, 0, 6) {
          for (rkh, 0, 3) {
            for (rkw, 0, 3) {
              Y[(((c*144) + (i*12)) + j)] = (Y[(((c*144) + (i*12)) + j)] +␣
↪(PaddedX[(((((ric*196) + (i*14)) + (rkh*14)) + j) + rkw)]*K[((((c*54) +␣
↪(ric*9)) + (rkh*3)) + rkw)]))
            }
          }
        }
      }
    }
  }
}
```

In the last code block we also print out the pseudo code of a 2-D convolution, which is a naive 6-level nested for loop.

Since NumPy only has a convolution for vectors, we use MXNet's convolution operator as the ground truth. The following code block defines the data generating function and a wrap function to call the convolution operator. Then we can feed the same tensors to compute the results in MXNet.

```
import mxnet as mx

def get_conv_data_mxnet(oc, ic, n, k, p, s, ctx='cpu'):
    ctx = getattr(mx, ctx)()
    data, weight, out = get_conv_data(oc, ic, n, k, p, s,
                                      lambda x: mx.nd.array(x, ctx=ctx))
    data, out = data.expand_dims(axis=0), out.expand_dims(axis=0)
    bias = mx.nd.zeros(out.shape[1], ctx=ctx)
    return data, weight, bias, out

# Save to the d2ltvm package.
def conv_mxnet(data, weight, bias, out, k, p, s):
    mx.nd.Convolution(data, weight, bias, kernel=(k,k), stride=(s,s),
                      pad=(p,p), num_filter=out.shape[1], out=out)

data, weight, bias, out_mx = get_conv_data_mxnet(oc, ic, n, k, p, s)
conv_mxnet(data, weight, bias, out_mx, k, p, s)
```

Lastly, let's compare the results. For a similar reason mentioned in the last chapter, the multi-threading used in MXNet makes us use a relative large tolerant error here.

```
np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 3.3.3 Summary

- We can express the computation of 2-D convolution in TVM in a fairly easy way.
- Deep learning workloads normally operate 2-D convolution on 4-D data tensors and kernel tensors.
- The naive 2-D convolution is a 6-level nested for loop.

## 3.4 Depthwise Convolution

Depthwise convolution is a special kind of convolution commonly used in convolutional neural networks designed for mobile and embedded applications, e.g. MobileNet (Howard et al., 2017).

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
```

### 3.4.1 Compute definition

Let's revisit the 2-D convolution described in Section 3.3 first. The 2-D convolution basically takes a 3-D data (note that for simplicity we set the batch size to be 1) in size `(ic, ih, iw)`, convolves it with a 4-D kernel in size `(oc, ic, kh, kw)`, and produces an output data in size `(oc, oh, ow)`. During the convolution, some padding and stride may be applied.

For depthwise convolution, the convolution computation itself stays the same, as illustrated in Fig. 3.3.1. It differs from the normal 2-D convolution in the way of organizing the convolution. In order to generate an output data in size `(oc, oh, ow)` from input data in size `(ic, ih, iw)`, a two-stage computation is needed. First, we process the input data with `ic` kernels, each of which convolves with the corresponding channel, to produce an intermediate data in size `(ic, oh, ow)`; then we perform the normal, but pointwise, 2-D convolution on the intermediate data in size `(ic, oh, ow)` using a 4-D kernel in size `(oc, ic, 1, 1)` to produce the output data in size `(oc, oh, ow)`, where `padding=0` and `stride=1`.

The computation of the second stage has been covered in Section 3.3. This section only focuses on the computation of the first stage, which is referred to as depthwise convolution. Fig. 3.4.1 illustrates its computation procedure.

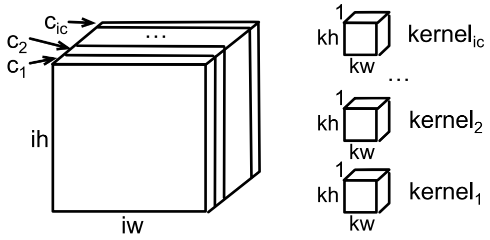


Fig. 3.4.1: Illustration of a depthwise convolution. Each channel of the input data convolves with a dedicated kernel.

From the figure we can see that the shape of the weight is a bit different from the 2-D convolution. The weight for depthwise convolution is 3-D, while it is 4-D for 2-D convolution. Therefore, we modify the get_conv_data slightly to handle the generation of the data for depthwise convolution, and save it for future use.

```
# Save to the d2ltvm package.
def get_conv_data(oc, ic, n, k, p=0, s=1, constructor=None, conv_type='direct
↪'):
    """Return random 3-D data tensor, 3-D kernel tenor and empty 3-D output
    tensor with the shapes specified by input arguments.

    oc, ic : output and input channels
    n : input width and height
    k : kernel width and height
    p : padding size, default 0
    s : stride, default 1
    conv_type: either direct 2D or depthwise, default direct
    constructor : user-defined tensor constructor
    """
    np.random.seed(0)
    data = np.random.normal(size=(ic, n, n)).astype('float32')
    ic_weight = ic
    if conv_type == 'depthwise':
        ic_weight = 1
    weight = np.random.normal(size=(oc, ic_weight, k, k)).astype('float32')
    on = d2ltvm.conv_out_size(n, k, p, s)
    out = np.empty((oc, on, on), dtype='float32')
    if constructor:
        data, weight, out = (constructor(x) for x in [data, weight, out])
    return data, weight, out
```

Comparing to Section 3.3, we added one argument to describe the convolution type, and make the input channel of the weight to be 1 when it is a depthwise convolution. You may wonder why we choose this dimension. The reason is to match the convention brought by the framework.

Then we define the depthwise convolution via TVM. Here, we reuse the padding and conv_out_size methods defined in Section 3.3.

```
from d2ltvm import padding, conv_out_size

# Save to the d2ltvm package.
def depthwise_conv(ic, nh, nw, kh, kw, ph=0, pw=0, sh=1, sw=1):
    """Convolution

    ic : number of channels for both input and output
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding sizes, default 0
    sh, sw : height and width strides, default 1
    """
    # reduction axes
    rkh = te.reduce_axis((0, kh), name='rkh')
    rkw = te.reduce_axis((0, kw), name='rkw')
    # output height and weights
    oh = conv_out_size(nh, kh, ph, sh)
    ow = conv_out_size(nw, kw, pw, sw)
    # pad X and then compute Y
    X = te.placeholder((ic, nh, nw), name='X')
```

(continues on next page)

(continued from previous page)

```
    K = te.placeholder((ic, 1, kh, kw), name='K')
    PaddedX = padding(X, ph, pw) if ph * pw != 0 else X
    Y = te.compute(
        (ic, oh, ow),
        lambda c, i, j: te.sum(
            (PaddedX[c, i*sh+rkh, j*sw+rkw] * K[c, 0, rkh, rkw]),
            axis=[rkh, rkw]), name='Y')

    return X, K, Y, PaddedX
```

After defining the computation of depthwise convolution, we can use the default schedule to compile and execute it as follows. We also print out the pseudo-code of it.

```
ic, n, k, p, s = 256, 12, 3, 1, 1

X, K, Y, _ = depthwise_conv(ic, n, n, k, k, p, p, s, s)
sch = te.create_schedule(Y.op)
mod = tvm.build(sch, [X, K, Y])
print(tvm.lower(sch, [X, K, Y], simple_mode=True))

data, weight, out = get_conv_data(ic, ic, n, k, p, s,
                                  constructor=tvm.nd.array,
                                  conv_type='depthwise')
mod(data, weight, out)
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 50176]
produce PaddedX {
  for (i0, 0, 256) {
    for (i1, 0, 14) {
      for (i2, 0, 14) {
        PaddedX[(((i0*196) + (i1*14)) + i2)] = tvm_if_then_else(((((i1 < 1)␣
↪|| (13 <= i1)) || (i2 < 1)) || (13 <= i2)), 0f, X[((((i0*144) + (i1*12)) +␣
↪i2) - 13)])
      }
    }
  }
}
produce Y {
  for (c, 0, 256) {
    for (i, 0, 12) {
      for (j, 0, 12) {
        Y[(((c*144) + (i*12)) + j)] = 0f
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            Y[(((c*144) + (i*12)) + j)] = (Y[(((c*144) + (i*12)) + j)]␣
↪+ (PaddedX[(((((c*196) + (i*14)) + (rkh*14)) + j) + rkw)]*K[(((c*9) +␣
↪(rkh*3)) + rkw)]))
          }
        }
      }
    }
  }
}
```

### 3.4.2 Depthwise Convolution in General

You may wonder why we want to replace a typical 2-D convolution into a more complicated, two-stage depthwise plus pointwise 2-D convolution. This book doesn't discuss about the choice of algorithms, but from the computational perspective, the main reason is to reduce the number of computation it requires. Assuming that the input data is in size `[ic, ih, iw]`, the kernel is in size `[kh, kw]`, and the output data is in size `[oc, oh, ow]`, a 2-D convolution takes $2 \times ic \times oh \times ow \times kh \times kw \times oc$ FLOPs, while a depthwise plus pointwise 2-D convolution takes $2 \times ic \times oh \times ow \times (kh \times kw + oc)$ FLOPs. It is easy to see that the 2-D convolution normally takes more FLOPs than depthwise plus pointwise 2-D convolution, especially when the kernel size and/or the number of output channels are large. Taking the above example where $ic = 256, oh = ow = 12, kh = kw = 3$, if we set $oc = 512$, the total FLOPs of a 2-D convolution is $339,738,624$, while the depthwise plus pointwise convolution is $38,412,288$, almost one order of magnitude smaller, are much suitable for mobile and embedded applications.

In the MobileNet paper (Howard et al., 2017), the depthwise convolution was described as a separable convolution which separates the channels for convolution. From another aspect, a depthwise convolution can be treated as a special kind of grouped convolution. A `G`-grouped convolution divide the channels into `G` groups and do the convolution group by group independently. This was first introduced in AlexNet to save memory. We can easily figure out that when the number of groups equals the number of channels, a grouped convolution is reduced to a depthwise convolution.

In fact, MXNet uses the same API `mx.nd.Convolution` to process depthwise convolution by specifying the number of groups, as we will show in the next code block.

In addition, a depthwise convolution can be generalized in other ways. For example, we can specify a `multiplier` to increase the number of channels for the output of depthwise convolution, which we are not cover for simplicity.

### 3.4.3 Comparing to Baseline

We use MXNet's convolution operator as the ground truth to verify the correctness of our depthwise convolution. Before that, we will need to generate data. Like what with have done for TVM, we modify the `get_conv_data_mxnet` method defined in Section 3.3 to take `conv_type`. The data used for depthwise convolution in MXNet can then be generated accordingly.

```python
import mxnet as mx

# Save to the d2ltvm package.
def get_conv_data_mxnet(oc, ic, n, k, p, s, ctx='cpu', conv_type='direct'):
    ctx = getattr(mx, ctx)()
    data, weight, out = get_conv_data(oc, ic, n, k, p, s,
                                      constructor=lambda x: mx.nd.array(x,␣
↪ctx=ctx),
                                      conv_type=conv_type)
    data, out = data.expand_dims(axis=0), out.expand_dims(axis=0)
    bias = mx.nd.zeros(out.shape[1], ctx=ctx)
    return data, weight, bias, out
```

Then we do the computation and compare with the TVM result. Note that the weight size is `[oc, 1, kh, kw]` as the number of groups equals the number of channels, i.e. each kernel only corresponds to one channel of the data, as what we are doing in TVM.

```
# Save to the d2ltvm package.
def depthwise_conv_mxnet(data, weight, bias, out, k, p, s):
    mx.nd.Convolution(data, weight, bias, kernel=(k,k), stride=(s,s),
                      pad=(p,p), num_filter=out.shape[1],
                      out=out, num_group=weight.shape[0])

data, weight, bias, out_mx = get_conv_data_mxnet(ic, ic, n, k, p, s, conv_
↪type='depthwise')
depthwise_conv_mxnet(data, weight, bias, out_mx, k, p, s)

np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 3.4.4 Summary

- Depthwise convolution, together with pointwise convolution, can save a lot of computation and memory compared to normal 2-D convolution.
- Depthwise convolution takes kernels in 3-D, while normal 2-D convolution takes kernels in 4-D.

## 3.5 Pooling

This section talks about how to use TVM to do pooling. Pooling is a common operator in CNN, please refer to chapter 6.5[19] in D2L if you are not familiar with this operator. Here we will skip the why, only focus on how.

There are two types of pooling, `max pooling` which returns the maximal value of a pool, and `avg pooling` which returns the average value of a pool. For simplicity, we work on 2D pooling in this section. Like conv2d, the pooling operator moves the pooling kernel across the feature map with some stride. Sometimes padding is needed to match the required output size. Pooling has significantly less computation than conv2d as it only needs to get the maximal or average value. It is a memory-bound operator.

Fig. 3.5.1 illustrate how 2D `max pooling` and `avg pooling` work, with the following setting: kernel size `[3, 3]`, stride `[1, 1]`, and padding `[1, 1]`.

[19] https://d2l.ai/chapter_convolutional-neural-networks/pooling.html

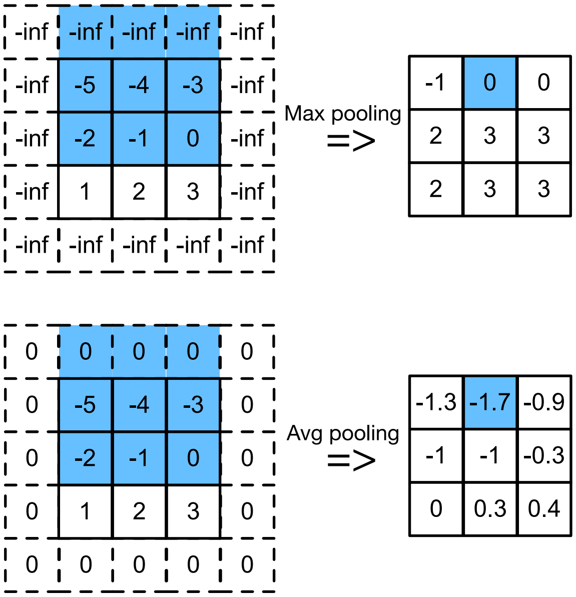


Fig. 3.5.1: 2D max and average poolings. The blue shape indicates a particular pooling step. Note that besides the algorithm, the padding values are also different.

```
import tvm
from tvm import te
import d2ltvm
```

### 3.5.1 Compute definition

The computation manner of `pooling` is similar to `conv`, so you will find the pooling definition code below takes similar arguments as `conv` defined in Section 3.3. The output size of pooling can be calculated by reusing the `conv_out_size` method, too.

We include two types of `pooling` in the same method using different `te.compute`. In the `pool_type` is specified otherwise, the method will throw an error. We use `te.max` to perform `max pooling` and `te.sum` and element-wise division to perform `avg pooling`. In addition, please also note that the padding values of `max pooling` is the `te.min_value` while `avg pooling` being 0.

```
# Save to the d2ltvm package.
def pool(pool_type, c, nh, nw, kh, kw, ph=0, pw=0, sh=1, sw=1):
    """2D pooling

    pool_type: pooling type, 'max' or 'avg'
    c : channels
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding sizes, default 0
    sh, sw : height and width strides, default 1
    """
    # reduction axes
    rkh = te.reduce_axis((0, kh), name='rkh')
    rkw = te.reduce_axis((0, kw), name='rkw')
    # output height and weights
```

(continues on next page)

(continued from previous page)

```
    oh = d2ltvm.conv_out_size(nh, kh, ph, sh)
    ow = d2ltvm.conv_out_size(nw, kw, pw, sw)
    # pad X and then compute Y
    X = te.placeholder((c, nh, nw), name='X')


    if pool_type == 'max':
        PaddedX = d2ltvm.padding(X, ph, pw, val=te.min_value(X.dtype)) \
            if ph * pw != 0 else X
        Y = te.compute((c, oh, ow), \
                            lambda c, h, w: \
                            te.max(PaddedX[c, h*sh+rkh, w*sw+rkw], \
                                axis=[rkh, rkw]), \
                            tag="pool_max", name='PoolMax')
    elif pool_type == 'avg':
        PaddedX = d2ltvm.padding(X, ph, pw) if ph * pw != 0 else X
        tsum = te.compute((c, oh, ow), \
                            lambda c, h, w: \
                            te.sum(PaddedX[c, h*sh+rkh, w*sw+rkw], \
                                axis=[rkh, rkw]), \
                            tag="pool_avg1", name='PoolSum')
        Y = te.compute((c, oh, ow), \
                            lambda c, h, w: \
                            tsum[c, h, w] / (kh*kw), \
                            tag='pool_avg2', name='PoolAvg')
    else:
        raise ValueError("Pool type should be 'avg' or 'max'.")
    return X, Y, PaddedX
```

We then compile the `max pooling` using some toy data sizes. The compute logic is simple as shown in the IR. Again, the `get_conv_data` method in Section 3.3 can be reused to initialize the data. Note that we don't need weights in this case.

```
c, n, k, p, s = 4, 12, 3, 1, 1
X, Y, PaddedX = pool('max', c, n, n, k, k, p, p, s, s)
sch = te.create_schedule(Y.op)
mod = tvm.build(sch, [X, Y])
print(tvm.lower(sch, [X, Y], simple_mode=True))
data, _, out_max = d2ltvm.get_conv_data(c, c, n, k, p, s, tvm.nd.array)
mod(data, out_max)
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 784]
produce PaddedX {
  for (i0, 0, 4) {
    for (i1, 0, 14) {
      for (i2, 0, 14) {
        PaddedX[(((i0*196) + (i1*14)) + i2)] = tvm_if_then_else(((((i1 < 1)␣
↪|| (13 <= i1)) || (i2 < 1)) || (13 <= i2)), -3.40282e+38f, X[((((i0*144) +␣
↪(i1*12)) + i2) - 13)])
      }
    }
  }
```

(continues on next page)

(continued from previous page)

```
}
produce PoolMax {
  for (c, 0, 4) {
    for (h, 0, 12) {
      for (w, 0, 12) {
        PoolMax[(((c*144) + (h*12)) + w)] = -3.40282e+38f
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            PoolMax[(((c*144) + (h*12)) + w)] = max(PoolMax[(((c*144) +␣
↪(h*12)) + w)], PaddedX[(((((c*196) + (h*14)) + (rkh*14)) + w) + rkw)])
          }
        }
      }
    }
  }
}
```

Next, we compile the `avg pooling` using the same toy data sizes. The compute logic is also simple. Check out the computation as well as the padding value difference from the `max pooling`.

```
X, Y, PaddedX = pool('avg', c, n, n, k, k, p, p, s, s)
sch = te.create_schedule(Y.op)
mod = tvm.build(sch, [X, Y])
print(tvm.lower(sch, [X, Y], simple_mode=True))
data, _, out_avg = d2ltvm.get_conv_data(c, c, n, k, p, s, tvm.nd.array)
mod(data, out_avg)
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 784]
// attr [PoolSum] storage_scope = "global"
allocate PoolSum[float32 * 576]
produce PaddedX {
  for (i0, 0, 4) {
    for (i1, 0, 14) {
      for (i2, 0, 14) {
        PaddedX[(((i0*196) + (i1*14)) + i2)] = tvm_if_then_else(((((i1 < 1)␣
↪|| (13 <= i1)) || (i2 < 1)) || (13 <= i2)), 0f, X[((((i0*144) + (i1*12)) +␣
↪i2) - 13)])
      }
    }
  }
}
produce PoolSum {
  for (c, 0, 4) {
    for (h, 0, 12) {
      for (w, 0, 12) {
        PoolSum[(((c*144) + (h*12)) + w)] = 0f
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            PoolSum[(((c*144) + (h*12)) + w)] = (PoolSum[(((c*144) + (h*12))␣
↪+ w)] + PaddedX[(((((c*196) + (h*14)) + (rkh*14)) + w) + rkw)])
          }
        }
      }
```

(continues on next page)

(continued from previous page)

```
      }
    }
  }
  produce PoolAvg {
    for (c, 0, 4) {
      for (h, 0, 12) {
        for (w, 0, 12) {
          PoolAvg[(((c*144) + (h*12)) + w)] = (PoolSum[(((c*144) + (h*12)) +␣
↪w)]*0.111111f)
        }
      }
    }
  }
```

### 3.5.2 MXNet Baseline

We use the pooling functions of MXNet as the baseline to check the correctness of our compiled functions. MXNet computes pooling similarly as what we have done. The only difference is that its input data is in 4D, including batch as the outmost dimension.

```
import mxnet as mx

# Save to the d2ltvm package.
def get_pool_data_mxnet(c, n, k, p, s, ctx='cpu'):
    ctx = getattr(mx, ctx)()
    data, _, out = d2ltvm.get_conv_data(c, c, n, k, p, s,
                                        lambda x: mx.nd.array(x, ctx=ctx))
    data, out = data.expand_dims(axis=0), out.expand_dims(axis=0)
    return data, out

# Save to the d2ltvm package.
def pool_mxnet(pool_type, data, out, k, p, s):
    mx.nd.Pooling(data, kernel=(k,k), stride=(s,s),
                      pad=(p,p), pool_type=pool_type, out=out)

data, out_max_mx = get_pool_data_mxnet(c, n, k, p, s)
pool_mxnet('max', data, out_max_mx, k, p, s)
data, out_avg_mx = get_pool_data_mxnet(c, n, k, p, s)
pool_mxnet('avg', data, out_avg_mx, k, p, s)
```

Finally, we check if our results are close enough to the results produced by MXNet.

```
import numpy as np

np.testing.assert_allclose(out_max_mx[0].asnumpy(), out_max.asnumpy(),␣
↪atol=1e-5)
np.testing.assert_allclose(out_avg_mx[0].asnumpy(), out_avg.asnumpy(),␣
↪atol=1e-5)
```

### 3.5.3 Summary

- 2D pooling handles the data in the similar way as 2D convolution, but the computation itself is much lighter.
- We can define `max pooling` and `avg pooling` easily using TVM expressions.

## 3.6 Batch Normalization

This section talks about how to use TVM to do batch normalization (`batch_norm`). Like pooling, `batch_norm` is also a common operator in CNN. D2L introduces this operator in details[20].

From the calculation perspective, for a given value, `batch_norm` subtracts the $mean$ out of it, and then divide it with the square root of the $variance$, no difference than a regular normalization. It is call `batch_norm` because the mean and variance are attained from the batches of when performed the training. After that, `batch_norm` also applies an affine transformation to the value, i.e. multiplies it with a scale value $gamma$ followed by adding a shift value $beta$. $Gamma$ and $beta$ are attained from the gradient computation of training. Lastly, a small positive value $epsilon$ is added to prevent the divisor to be 0.

In the case of inference, both the mean and variance are determined, so the process of `batch_norm` is just a combination of several simple element-wise operations.

```python
import tvm
from tvm import te
import d2ltvm
import numpy as np
```

### 3.6.1 Compute definition

In practice, we are not going to perform `batch_norm` of one value. Instead, the `batch_norm` will be executed on the output of a convolution, namely, 3-D data in (channel, height, weight). Data in different channels have different values of $mean$, $variance$, $gamma$, and $beta$. The calculation can be expressed as the following formula.

$$out[i,:,:] = \frac{data[i,:,:] - mean[i]}{\sqrt{var[i] + \epsilon}} * gamma[i] + beta[i] \tag{3.6.1}$$

During model training, $mean$ and $var$ are computed from the input $data$. However, in model inference which we focus on here, $mean$ and $var$ are given; therefore we don't need to compute them from $data$.

We will define the compute of this formula. Essentially, `batch_norm` is a combination of a number of simple broadcasting and element-wise calculations. Note that in Section 3.1 we defined a limited `broadcast_add` to perform only broadcast addition for 2-D tensors. If we generalize it to more dimensions and more calculators, we can reuse them to compose the `batch_norm` operator. This is actually what TVM does.

Here, for simplicity, we use TVM basic operators for broadcast calculation. TVM operators are defined in `TOPI`, which stands for Tensor OPerator Inventory. It follows the NumPy convention to override the arithmetic operators (i.e. `+`, `-`, `*`, `/`) for broadcast calculation. The element-wise square root can be found in `TOPI`, too.

The code snippet to define `batch_norm` is as follows.

[20] https://d2l.ai/chapter_convolutional-modern/batch-norm.html

```python
# Save to the d2ltvm package.
import topi

def batch_norm(c, n, eps=1e-5):
    """batch normalization

    c : channels
    N : input width and height
    eps : small positive value to prevent divide 0
    """

    X = te.placeholder((c, n, n), name='X')
    Mean = te.placeholder((c, 1, 1), name='Mean')
    Var = te.placeholder((c, 1, 1), name='Var')
    Gamma = te.placeholder((c, 1, 1), name='Gamma')
    Beta = te.placeholder((c, 1, 1), name='Beta')
    C1 = X - Mean
    C2 = topi.sqrt(Var + eps)
    Y = C1 / C2 * Gamma + Beta
    return X, Mean, Var, Gamma, Beta, Y
```

We can then compile print the IR and compile it. The IR contains several stages but should be easy to follow.

```python
c = 32
n = 28
X, Mean, Var, Gamma, Beta, Y = batch_norm(c, n)

sch = te.create_schedule(Y.op)
mod = tvm.build(sch, [X, Mean, Var, Gamma, Beta, Y])

print(tvm.lower(sch, [X, Mean, Var, Gamma, Beta], simple_mode=True))
```

```
// attr [T_subtract] storage_scope = "global"
allocate T_subtract[float32 * 25088]
// attr [T_add] storage_scope = "global"
allocate T_add[float32 * 32]
produce T_subtract {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (X[(((ax0*784) +␣
↪(ax1*28)) + ax2)] - Mean[ax0])
      }
    }
  }
}
produce T_add {
  for (ax0, 0, 32) {
    T_add[ax0] = (Var[ax0] + 1e-05f)
  }
}
produce compute {
  for (i0, 0, 32) {
    T_add[i0] = sqrt(T_add[i0])
  }
```

(continues on next page)

(continued from previous page)

```
}
produce T_divide {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784)␣
↪+ (ax1*28)) + ax2)]/T_add[ax0])
      }
    }
  }
}
produce T_multiply {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784)␣
↪+ (ax1*28)) + ax2)]*Gamma[ax0])
      }
    }
  }
}
produce T_add {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784)␣
↪+ (ax1*28)) + ax2)] + Beta[ax0])
      }
    }
  }
}
```

To execute it, we will need to create data for `batch_norm`. Similar to the previous sections for getting conv and pooling data, we define a `get_bn_data` method to generate the data of `batch_norm`. One tricky thing is that the variance must be non-negative numbers. Therefore, we move the mean value of the random number generator's normal distribution to 1 (by default mean 0 and standard deviation 1), and get the absolute numbers of generated results.

After getting the data, we can simply call the compiled module to execute.

```
# Save to the d2ltvm package.
def get_bn_data(c, n, constructor=None):
    """Return the batch norm data, mean, variance, gamma and beta tensors.
       Also return the empty tensor for output.

    c : channels
    n : input width and height
    constructor : user-defined tensor constructor
    """
    np.random.seed(0)
    data = np.random.normal(size=(c, n, n)).astype('float32')
    mean = np.random.normal(size=(c, 1, 1)).astype('float32')
    # move the mean of the normal distribution to be 1
    var = np.random.normal(loc=1.0, size=(c, 1, 1)).astype('float32')
```

(continues on next page)

(continued from previous page)

```
    # make sure all variance numbers are not negative
    var = np.absolute(var)
    gamma = np.random.normal(size=(c, 1, 1)).astype('float32')
    beta = np.random.normal(size=(c, 1, 1)).astype('float32')
    out = np.empty((c, n, n), dtype='float32')
    if constructor:
        data, mean, var, gamma, beta, out = \
        (constructor(x) for x in [data, mean, var, gamma, beta, out])
    return data, mean, var, gamma, beta, out

data, mean, var, gamma, beta, out = get_bn_data(c, n, tvm.nd.array)
mod(data, mean, var, gamma, beta, out)
```

### 3.6.2 MXNet Baseline

We use the `batch_norm` function of MXNet as the baseline to check the correctness of our compiled functions. This function in MXNet was defined to be generic for both training and inference. In the inference case that we talk about here, we will need to set the corresponding input arguments properly. One is $use_{g}lobal_{s}tats$, which needs to be set `True` as we will use the input mean and variance for `batch_norm` to compute instead of computing them from the input data (training will do so). The other is $fix\_gamma$, which needs to be set `False` so that the input $gamma$ will be used instead of setting $gamma$ to be all 1.

Lastly, like we have discussed in other cases, MXNet `batch_norm` has input data in 4D, including batch as the outmost dimension. So we will expand this dimension in the data accordingly.

```
import mxnet as mx

# Save to the d2ltvm package.
def get_bn_data_mxnet(c, n, ctx='cpu'):
    ctx = getattr(mx, ctx)()
    data, mean, var, gamma, beta, out = get_bn_data(c, n,
                                        lambda x: mx.nd.array(x, ctx=ctx))
    data, out = data.expand_dims(axis=0), out.expand_dims(axis=0)
    return data, mean, var, gamma, beta, out

# Save to the d2ltvm package.
def batch_norm_mxnet(data, mean, var, gamma, beta, out, eps=1e-5):
    # use_global_stats=True to use the input mean and var instead of computing
    # the mean and var of the input data.
    # fix_gamma=False so that gamma won't be set to 1.
    mx.nd.BatchNorm(data, gamma, beta, mean, var, eps,
                    use_global_stats=True, fix_gamma=False, out=out)

data, mean, var, gamma, beta, out_mx = get_bn_data_mxnet(c, n)
batch_norm_mxnet(data, mean, var, gamma, beta, out_mx)
```

Finally, we check if our results are close enough to the results produced by MXNet.

```
np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 3.6.3 Summary

- From the computation perspective, `batch_norm` is a combination of a number of broadcast and element-wise simple operators, which can be easily attained from TVM's Tensor OPerator Inventory (TOPI).
- In inference, $mean$ and $var$ of `batch_norm` are pre-defined.

# 4 | Operator Optimizations on CPUs

In the past three chapters we mainly focus on the functionality of operators, namely, how to implement the operators to function correctly in TVM. However, getting right results out is not sufficient. Operators could perform poorly even if they execute correctly.

Starting from this chapter, we will talk about the performance optimization to the operators. Specifically, we will work on operator optimizations on CPUs in this chapter, and move on to GPUs in the next chapter.

## 4.1 CPU Architecture

In this section, we will do a brief introduction to the system components that are important for the performance of deep learning and scientific computing on CPUs. For a more comprehensive survey, we recommend this classic textbook[21]. We assume the readers knowing the basic system concepts such as clock rate (frequency), CPU cycle, and cache.

### 4.1.1 Arithmetic Units

A typical general-purpose CPU has hardware units to perform arithmetics on integers (called ALU[22]) and floating-points (called FPU[23]). The performance of various data types depends on the hardware. Let's first check the CPU model we are using.

```
# The following code runs on Linux
!cat /proc/cpuinfo | grep "model name" | head -1
```

```
model name  : Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
```

Now check the performance of a matrix multiplication of different data types.

```
import numpy as np

def benchmark(dtype):
    x = np.random.normal(size=(1000, 1000)).astype(dtype)
    %timeit np.dot(x, x)

benchmark('float32')
benchmark('float64')
```

(continues on next page)

[21] https://www.amazon.com/Computer-Architecture-Quantitative-John-Hennessy/dp/012383872X

[22] https://en.wikipedia.org/wiki/Arithmetic_logic_unit

[23] https://en.wikipedia.org/wiki/Floating-point_unit

(continued from previous page)

```
benchmark('int32')
benchmark('int64')
```

```
2.54 ms ± 5.01 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
4.69 ms ± 37.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
1.12 s ± 601 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.27 s ± 2.26 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

As can be seen, 32-bit floating-point (float32) is 2x faster than 64-bit floating-point (float64). The integer performance is way more slower and there is no much difference between 32-bit integer (int32) and 64-int integer. We will get back to the understand more about these numbers later.

Some operators, however, could be significantly slower than the multiplication and addition `a += b * c` used in matrix multiplication. For example, CPU may need hundreds of cycles to computing transcendental functions such as `exp`. You can see that even 1000 times fewer operations is needed for `np.exp(x)` than `np.dot(x, x)`, the former one takes longer time.

```
x = np.random.normal(size=(1000, 1000)).astype('float32')
%timeit np.exp(x)
```

```
816 µs ± 308 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

### 4.1.2 Parallel Execution

The CPU frequency increased rapidly until the beginning of the 21st century. In 2003, Intel released a Pentium 4[24] CPU with up to 3.8 GHz clock rate. If we check our CPU clock rate,

```
# The following code runs on Linux
!lscpu | grep MHz
```

```
CPU MHz:                  2499.998
```

we can see that it has a lower clock rate compared to the product in 2003, but it might be 100x faster than the Pentium 4 CPU. One secret source is that new CPU models explore a lot more in the territory of parallel execution. Next we briefly discuss two typical parallelizations.

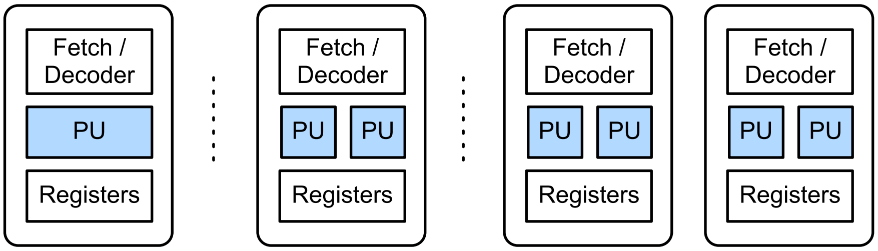


Fig. 4.1.1: Single core vs. single core with SIMD vs. multi-core with SIMD.

[24] https://en.wikipedia.org/wiki/Pentium_4

### SIMD

Single instruction, multiple data (SIMD[25]), refers to processing multiple elements with the same instruction simultaneously. Fig. 4.1.1 illustrates this architecture. In a normal CPU core, there is an instruction fetching and decoding unit. It runs an instruction on the processing unit (PU), e.g. ALU or FPU, to process one element, e.g. float32, each time. With SIMD, we have multiple PUs instead of one. In each time, the fetch-and-decode unit submit the same instruction to every PU to execute simultaneously. If there are $n$ PUs, then we can process $n$ element each time.

Popular SIMD instruction sets include Intel's SSE[26] and AVX[27], ARM's Neon[28] and AMD's 3DNow![29]. Let's check which sets our CPU supports.

```
# The following code runs on Linux
!cat /proc/cpuinfo | grep "flags" | head -1
```

```
flags               : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge␣
↪mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb␣
↪rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc aperfmperf tsc_
↪known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe␣
↪popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm␣
↪3dnowprefetch invpcid_single pti fsgsbase tsc_adjust bmi1 avx2 smep bmi2␣
↪erms invpcid mpx avx512f rdseed adx smap clflushopt clwb avx512cd xsaveopt␣
↪xsavec xgetbv1 ida arat pku
```

As can be seen, the most powerful SIMD instruction set supported is AVX-512, which extends AVX to support executing SIMD on 512-bit width data, e.g. it is able to perform 16 float32 operations or 8 float64 operations each time.

### Multi-cores

SIMD improves the performance of a single core, another way is adding multiple cores to a single CPU processor. Fig. 4.1.1 shows two CPU cores, each of which has 2 PUs.

It looks like that our CPU has 16 cores.

```
# The following code runs on Linux
!cat /proc/cpuinfo | grep "model name" | wc -l
```

```
16
```

But note that modern Intel CPUs normally has hyper-threading[30] which runs 2 hardware threads per core. By hyper-threading, each core is presented as 2 logical cores to the operating system. So even the system shows there are 16 cores, physically our CPU only has 8 cores.

Having two threads sharing the resource of the same core may increase the total throughput but at the expense of increasing the overall latency. In addition the effect of hyper-threading is very much dependent on the appli-

[25] https://en.wikipedia.org/wiki/SIMD
[26] https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
[27] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
[28] https://en.wikipedia.org/wiki/ARM_architecture#Advanced_SIMD_(NEON)
[29] https://en.wikipedia.org/wiki/3DNow!
[30] https://en.wikipedia.org/wiki/Hyper-threading

cation. Therefore, it is not generally recommended to leverage hyper-threading in the deep learning workloads. Later on in the book, you'll see that we only launch 8 threads even if our CPU presents 16 cores.

### Performance

We often use floating point operations per second (FLOPS[31]) to measure the performance of a hardware platform or an executable program. The theoretical peak performance of a single CPU can be computed by

`#physical_cores * #cycles_per_second * #instructions_per_cycle * #operations_per_instruction`

where `#instructions_per_cycle` is also called the SIMD width.

For the CPU we are using, it has 8 physical cores, the max clock rate (i.e. `#cycles_per_second`) is $2.5 \times 10^9$, the AVX-512 computes 16 float32 instructions per cycle, the FMA[32] instruction set in AVX-512 compute `a += b * c` each time, which contains 2 operations. Therefore, the GFLOPS (gigaFLOPS) for single precision (float32) is

```
2.5 * 8 * 16 * 2
```

```
640.0
```

You can modify the above code based on your system information to calculate your CPU peak performance.

Matrix multiplication (*matmul*) is a good benchmark workload for the peak performance, which has $2 \times n^3$ operations in total if all matrices are in shape $[n, n]$. After executing a *matmul*, we can get its (G)FLOPS by dividing its total operations using the averaged executing time. As can be seen, the measured GFLOPS is close to the peak performance (~90% of peak).

```
x = np.random.normal(size=(1000, 1000)).astype('float32')
res = %timeit -o -q np.dot(x, x)
2 * 1000**3 / res.average / 1e9
```

```
774.949674567942
```

## 4.1.3 Memory Subsystem

Another component which significantly impacts the performance is the memory subsystem. The memory size is one of the key specifications of a system. The machine we are using has 240 GB memory.

```
# The following code runs on Linux
!cat /proc/meminfo | grep MemTotal
```

```
MemTotal:       65157056 kB
```

The memory bandwidth, on the other hand, is less noticed but equally important. We can use the mbw[33] tool to test the bandwidth.

[31] https://en.wikipedia.org/wiki/FLOPS

[32] https://en.wikipedia.org/wiki/FMA_instruction_set

[33] http://manpages.ubuntu.com/manpages/xenial/man1/mbw.1.html

```
# The following code runs on Linux
!mbw 256 | grep AVG | grep MEMCPY
```

```
AVG Method: MEMCPY  Elapsed: 0.04812        MiB: 256.00000  Copy: 5320.310␣
↪MiB/s
```

Note that our CPU can execute $640 \times 10^9$ operations on float32 numbers per second. This requires the bandwidth to be at least $640 \times 4 = 2560$ GB/s, which is significantly larger than the measured bandwidth. CPU uses caches to fill in this big bandwidth gap. Let's check the caches our CPU has.

```
# The following code runs on Linux
!lscpu | grep cache
```

```
L1d cache:                32K
L1i cache:                32K
L2 cache:                 1024K
L3 cache:                 36608K
```

As can be seen, there are three levels of caches: L1, L2 and L3 (or LLC, Last Level Cache). The L1 cache has 32KB for instructions and 32KB for data. The L2 cache is 32x larger. The L3 cache is way more larger, but it is still thousands times smaller than the main memory. The benefits of caches are significantly improved access latency and bandwidth. Typically on modern CPUs, the latency to access L1 cache is less than 1 ns, the L2 cache's latency is around 7 ns, and the L3 cache is slower, with a latency about 20 ns, while still faster than the main memory's 100 ns latency.

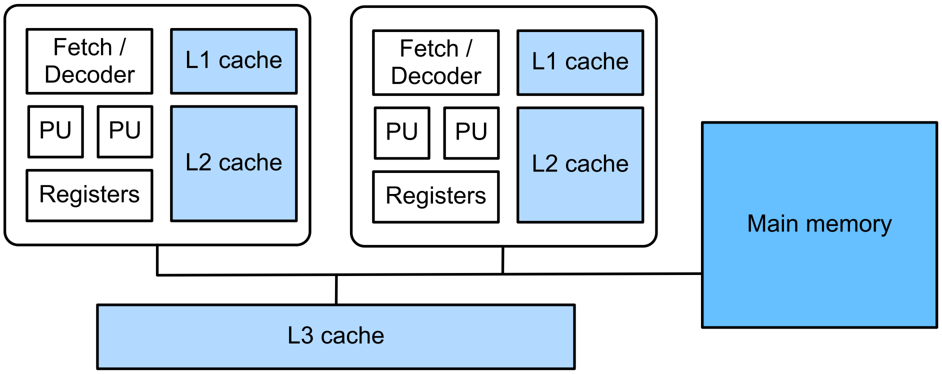


Fig. 4.1.2: The layout of main memory and caches.

A brief memory subsystem layout is illustrated in Fig. 4.1.2. L1 and L2 caches are exclusive to each CPU core, and L3 cache is shared across the cores of the same CPU processor To processing on some data, a CPU will first check if the data exist at L1 cache, if not check L2 cache, if not check L3 cache, if not go to the main memory to retrieve the data and bring it all the way through L3 cache, L2 cache, and L1 cache, finally to the CPU registers. This looks very expensive but luckily in practice, the programs have the data locality patterns[34] which will accelerate the data retrieving procedure. There are two types of locality: temporal locality and spatial locality. Temporal locality means that the data we just used usually would be used in the near future so that they may be still in cache. Spatial locality means that the adjacent data of the ones we just used are likely to be used in the near future. As the system always brings a block of values to the cache each time (see

[34] https://en.wikipedia.org/wiki/Locality_of_reference

the concept of cache lines[35]), those adjacent data may be still in cache when referenced to. Leveraging the advantage brought by data locality is one of the most important performance optimization principles we will describe in detail later.

### 4.1.4 Summary

- CPUs have dedicated units to handle computations on various data types. A CPU's peak performance is determined by the clock rate, the number of cores, and the instruction sets.
- CPUs use multi-level caches to bridge the gap between CPU computational power and main memory bandwidth.
- An efficient program should be effectively parallelized and access data with good temporal and spatial localities.

## 4.2 Function Call Overhead

We are starting to benchmark various schedules since this chapter. Before diving into various execution time numbers, we need to be aware of the overhead of issuing a function call in Python. It's well known that Python is not the fastest language on earth. Prototyping with Python is fast, but we also need to pay the cost that Python does smart things for us under the hook. In this section, we will investigate the overhead to call a function in Python, and demonstrate its impact to our later benchmarking results.

### 4.2.1 Execution Time Measurement

In Python, we often use the `timeit` module to benchmark a workload, especially when its execution time is way less than 1 second. Note that Jupyter has a magic build-in function `%timeit` that makes the usage simpler, but a function using it cannot be saved for future usage, so we will use `timeit` directly.

```
import timeit
import numpy as np
from matplotlib import pyplot as plt
from IPython import display
```

The following codes return the execution time of creating a `[4, 4]` zero matrix by 10 times.

```
timer = timeit.Timer(setup='import numpy as np',
                     stmt='np.zeros((4, 4))')
timer.timeit(10)
```

```
2.514570951461792e-05
```

We can see that the above workload can be done in tens of microsecond, we may need to increase the number of repeats to obtain relatively accurate execution time. The following function will determine the number of repeats needed to run a workload for at least 1 second, and then return the average execution time.

[35] https://en.wikipedia.org/wiki/CPU_cache#CACHE-LINES

```python
# Save to the d2ltvm package.
def bench_workload(workload):
    """Benchmark a workload

    workload: a method that accept a num_repeat argument
    and return its total execution time
    """
    workload(1)  # warmup
    time = workload(1)  # the time to run once
    if time > 1: return time
    # The number of repeats to measure at least 1 second
    num_repeats = max(int(1.0 / time), 5)
    return workload(num_repeats) / num_repeats

print('time for np.zeros((4, 4)):', bench_workload(timer.timeit))
```

```
time for np.zeros((4, 4)): 2.6995336035230484e-07
```

We can see that the newly measured execution time of `np.zeros((4, 4))` is much smaller than the one we first measured, as the former one includes the warmup time.

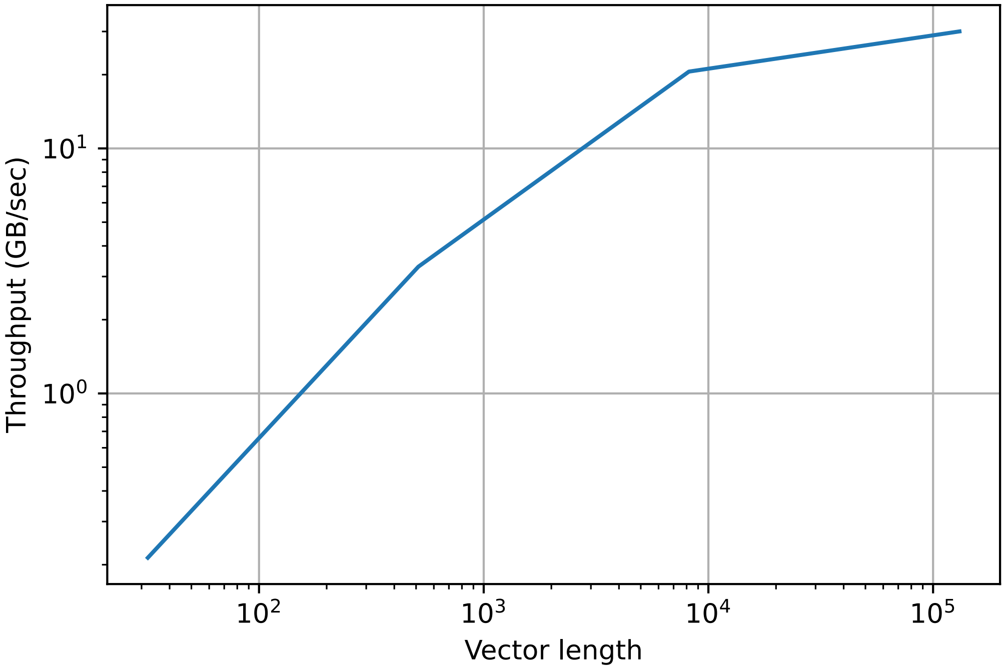
### 4.2.2 The Non-Negligible Overhead

Now we benchmark the `copyto` method, which copies the contents of an ndarray to another one, on various size ndarrays.

```python
def np_setup(n):
    return 'import numpy as np\n' \
           'x = np.random.normal(size=%d).astype("float32")\n' \
           'y = np.empty_like(x)\n'% n

def np_copy(n):
    return timeit.Timer(setup=np_setup(n),
                        stmt='np.copyto(y, x)')

sizes = 2**np.arange(5, 20, 4).astype('int')
exe_times = [bench_workload(np_copy(n).timeit) for n in sizes]
```

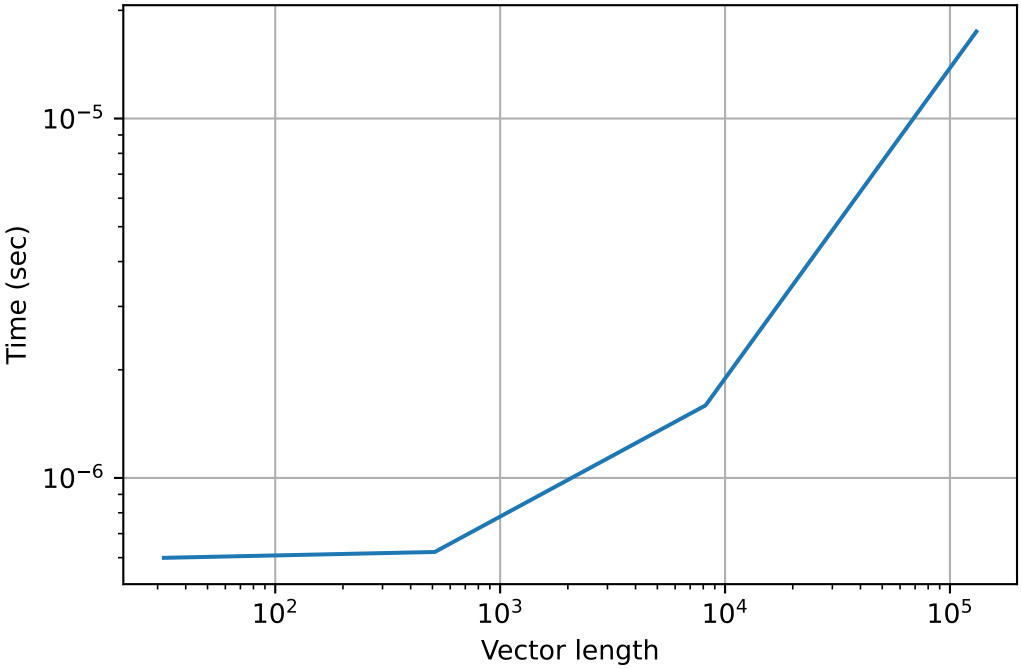
Plot the throughput versus vector length.

```python
display.set_matplotlib_formats('svg')
# one float32 takes 4 bytes
plt.loglog(sizes, sizes*4/exe_times/1e9)
plt.xlabel('Vector length')
plt.ylabel('Throughput (GB/sec)')
plt.grid()
```



We can see that the throughput first increases and then plateaus. The plateau reflects the saturation of the memory bandwidth. However, the behavior of small vectors is not as expected since when the vector length is small, all data should be in the cache which should result in much higher throughput. To examine the reason, let's simply draw the execution time versus vector length.

```
plt.loglog(sizes, exe_times)
plt.xlabel('Vector length')
plt.ylabel('Time (sec)')
plt.grid()
```



We can see that when the vector length is smaller than $10^3$, the execution time barely decrease when the vector length shortens. This is because when the vector length is short, the real `copyto` execution time is too small, and the total execution time is dominated by the function call overhead. The overhead includes any argument preprocessing in the Python function, evoking the foreign function interface, and other Python backend overhead[36]. Therefore, benchmarking too small workloads is not quite meaningful.

### 4.2.3 Overhead of NumPy, TVM and MXNet

Throughout the book, we will benchmark various operators in Numpy, TVM and MXNet. Let's examine their function call overheads. We could roughly estimate it by executing some small workloads where the actual execution time is negligible compared to the overhead. Let's first check NumPy.

```
sizes = 2**np.arange(1, 8).astype('int')
exe_times = np.array([bench_workload(np_copy(n).timeit) for n in sizes])

print('NumPy call overhead: %.1f microsecond' % (exe_times.mean()*1e6))
```

```
NumPy call overhead: 0.6 microsecond
```

The overhead of TVM is higher but in the same order of magnitude.

```
def tvm_copy(n):
    return timeit.Timer(setup=np_setup(n)+'import tvm\n'
                        'x, y = tvm.nd.array(x), tvm.nd.array(y)',
                        stmt='x.copyto(y)')

tvm_times = np.array([bench_workload(tvm_copy(n).timeit) for n in sizes])
print('TVM call overhead: %.1f microsecond'% (tvm_times.mean()*1e6,))
```

```
TVM call overhead: 0.2 microsecond
```

Compared to NumPy and TVM, MXNet has substantially higher overhead. The reason might due to MXNet uses `ctypes` while TVM is compiled with `cython`, and MXNet uses lazy evaluation that brings additional overhead.

```
def mx_copy(n):
    return timeit.Timer(setup=np_setup(n)+'import mxnet as mx\n'
                        'x, y = mx.nd.array(x), mx.nd.array(y)\n'
                        'mx.nd.waitall()'% n,
                        stmt='x.copyto(y); y.wait_to_read()')

mx_times = np.array([bench_workload(mx_copy(n).timeit) for n in sizes])
print('MXNet call overhead: %.1f microsecond'% (mx_times.mean()*1e6,))
```

```
MXNet call overhead: 37.4 microsecond
```

[36] https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

### 4.2.4 Summary

- Warming up a method by running it a few times beforehand is a good practice to measure its execution time.
- The function call overhead might takes several microsecond. Benchmarking too small functions in Python is meaningless.

## 4.3 Vector Add

In this section, we will optimize the vector add defined in Section 1.2 on CPU.

### 4.3.1 Setup

```
%matplotlib inline
import d2ltvm
import inspect
from IPython import display
import numpy as np
from matplotlib import pyplot as plt
import timeit
import tvm
from tvm import te
```

We first define reusable plot functions to draw multiple lines, which generalize the plot function defined in Section 4.2.

```
# Save to the d2ltvm package.
def plot(X, Y, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear', fmts=None,
         figsize=(4.5, 3)):
    """Plot multiple lines"""
    display.set_matplotlib_formats('svg')
    plt.rcParams['figure.figsize'] = figsize
    axes = plt.gca()
    X, Y = np.array(X), np.array(Y)
    if X.shape != Y.shape: X = [X] * len(Y)
    if not fmts: fmts = ['-'] * len(X)
    for x, y, fmt in zip(X, Y, fmts):
        axes.plot(x, y, fmt)
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend: axes.legend(legend)
    axes.grid()

# Save to the d2ltvm package
def plot_gflops(sizes, gflops, legend, xlabel='Size'):
    d2ltvm.plot(sizes, gflops, xlabel=xlabel, ylabel='GFLOPS',
```

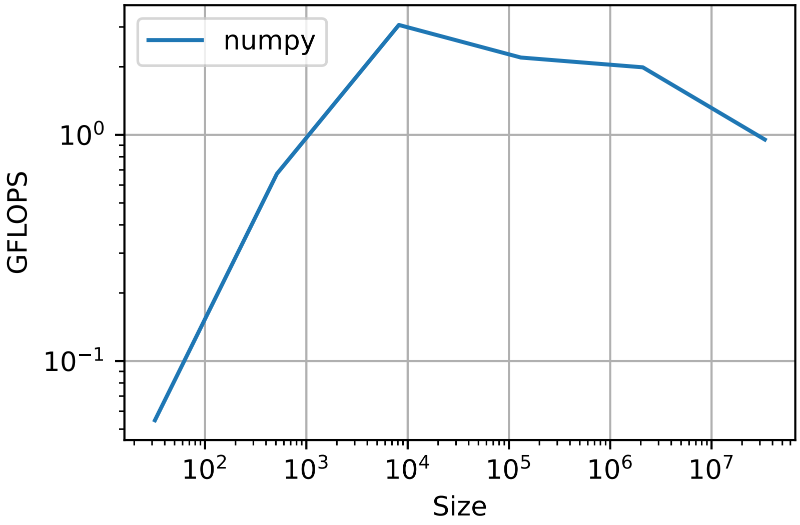
(continues on next page)

(continued from previous page)

```
              xscale='log', yscale='log',
              legend=legend, fmts=['--']*(len(gflops)-1)+['-'])
```

Then we benchmark the performance of NumPy as our baseline. We show the vector size vs measured GFLOPS[37], giga-floating point operations per second, in the following diagram.

```
sizes = 2**np.arange(5, 29, 4)
np_add = lambda n: timeit.Timer(setup='import numpy as np\n'
                                'import d2ltvm\n'
                                'a, b, c = d2ltvm.get_abc(%d)' % n,
                                stmt='np.add(a, b, out=c)')
exe_times = [d2ltvm.bench_workload(np_add(n).timeit) for n in sizes]
np_gflops = sizes / 1e9 / np.array(exe_times)
plot_gflops(sizes, [np_gflops], ['numpy'])
```



As we can see that the performance first increases with the vector length, which is due to the system overhead domination when the workload is small. The performance then decreases when we cannot fit all data into the cache.

### 4.3.2 Default Schedule

In the following code block, we define a reusable method to benchmark TVM performance. It accepts three arguments: 1) a `func` which returns the schedule and its corresponding symbolic tensor arguments; 2) the size list specifying a number of the vector lengths; and 3) the machine target which is CPU-related for this chapter and will be GPU-related in the next chapter.

```
# Save to the d2ltvm package.
def bench_vector_add_tvm(func, sizes, target):
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(a, b, c).mean * nrepeats
```

(continues on next page)

[37] https://en.wikipedia.org/wiki/FLOPS

(continued from previous page)

```
    times = []
    for n in sizes:
        s, (A, B, C) = func(int(n))
        mod = tvm.build(s, [A, B, C], target)
        ctx = tvm.context(target, 0)
        a, b, c = d2ltvm.get_abc(n, lambda x: tvm.nd.array(x, ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
    return sizes / 1e9 / np.array(times)
```

The default schedule is a plain one-level for-loop program.

```
def default(n):
    A, B, C = d2ltvm.vector_add(n)
    s = te.create_schedule(C.op)
    return s, (A, B, C)

s, args = default(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce c {
  for (i, 0, 64) {
    c[i] = (a[i] + b[i])
  }
}
```

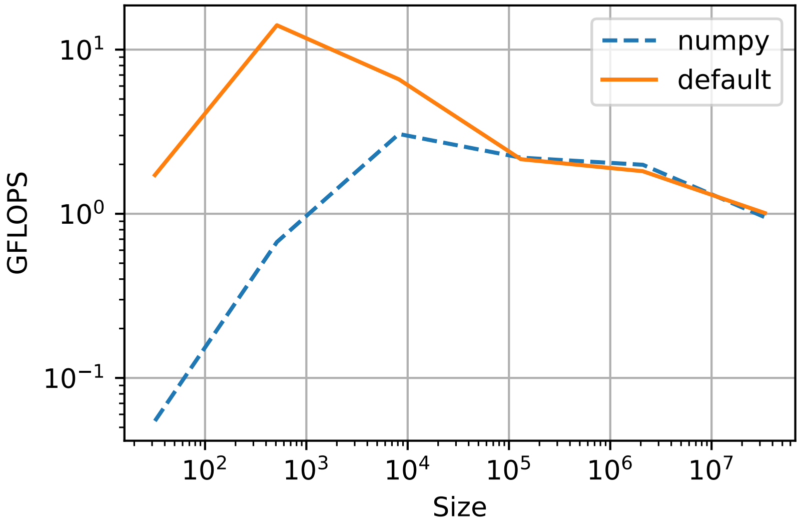
Remember in Section 4.1 we found that our CPU supports AVX-512, we pass `-mcpu=skylake-avx512` to LLVM so that it can generate AVX-512 instructions if possible. In the following codes, we print a few lines of generated LLVM code.

```
target = 'llvm -mcpu=skylake-avx512'
mod = tvm.build(s, args, target)
print(mod.get_source()[:500])
```

```
; ModuleID = 'TVMMod'
source_filename = "TVMMod"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

%0 = type { i8*, %1, i32, %2, i64*, i64*, i64 }
%1 = type { i32, i32 }
%2 = type { i8, i8, i16 }

@__TVMAPISetLastError = linkonce dllexport local_unnamed_addr global␣
↪void (i8*)* null, align 8
@.str = private constant [69 x i8] c"Assert fail: (num_args == 3),␣
↪default_function: num_args should be 300", align 1
@.str.1 = private constant [144 x i8] c"
```

You may find it not quite readable if you are not familiar with LLVM IR. But you don't need to worry much as in most cases, only reading the C-like pseudo code is sufficient to study the performance. Now let's benchmark the default schedule.

```
default_gflops = bench_vector_add_tvm(default, sizes, target)
plot_gflops(sizes, [np_gflops, default_gflops], ['numpy', 'default'])
```



When the vector size is small, the default scheduling outperforms NumPy, which means that in this method the function call overhead of TVM is smaller than NumPy. It's not surprising to find the performance degrades when increasing the vector size as the data cannot fit into the last level cache.

### 4.3.3 Parallelization

One important optimization that is not enabled by default is thread-level parallelization. The vector add operator is an embarrassingly parallel workload[38], we can just change the for-loop into a parallel for-loop. In TVM, we first obtain the scheduler for the output symbol `C` by `s[C]`, and then impose the parallelization of the computation to its first axis, which is `C.op.axis[0]`.

```
def parallel(n):
    s, (A, B, C) = default(n)
    # add thread-level parallelism
    s[C].parallel(C.op.axis[0])
    return s, (A, B, C)

s, args = parallel(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce c {
  parallel (i, 0, 64) {
    c[i] = (a[i] + b[i])
  }
}
```

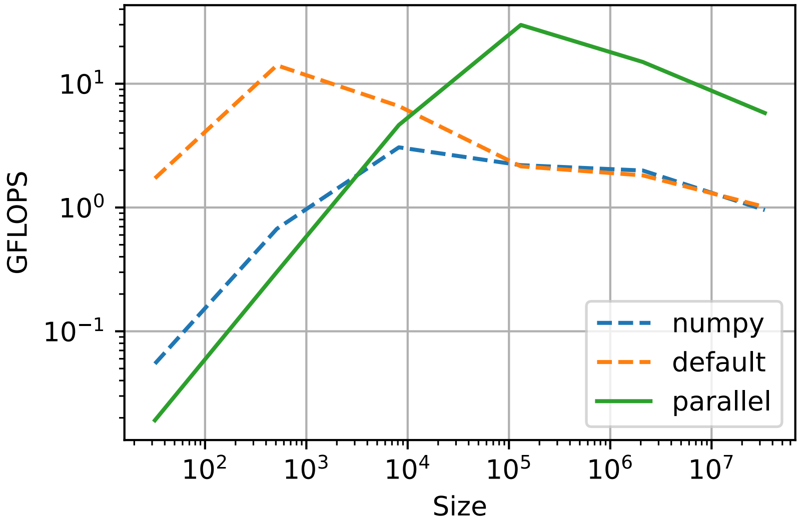
We can see that `for` is changed to `parallel` in the above pseudo codes. It means that the iterations could be executed in parallel by multiple threads. A typical implementation on a system with $t$ CPU cores is we first create $t$ threads with one thread for each core, then thread $i$ will execute blocks $j$ if `j % t = i`. All these

[38] https://en.wikipedia.org/wiki/Embarrassingly_parallel

threads will run simultaneously on their exclusive cores to achieve parallelization, and come back to retrieve another block to execute after finishing one. This is often called the round-robin scheduling[39], which works well if each thread runs at the same speed and every iteration has a roughly same workload. TVM's thread-level parallelization implementation is a special case of round-robin, which evenly divides the to-be-parallelized $n$-length loop into $t$ blocks. Therefore, each thread only needs to process one block. Furthermore, the TVM runtime binds each working thread to a disjoint physical core to avoid resource contention and thread migration overhead. There are other thread-level parallelization schemes such as the more dynamic consumer-producer scheduling[40].

Then we check the parallelization of vectorization and plot the comparison diagram via the following code block.

```
parallel_gflops = bench_vector_add_tvm(parallel, sizes, target)
plot_gflops(sizes, [np_gflops, default_gflops, parallel_gflops],
     ['numpy', 'default', 'parallel'])
```



Comparing the results we obtained before, parallelization significantly improves the performance when the workloads are large, e.g. vector lengths beyond $10^4$. However, the parallelization overhead impact the performance for small workloads, where single thread is even faster. The performance drops at a larger size as multi-core comes in play, leading to a larger amount of L2 cache in total.

### 4.3.4 Vectorization

A single core may have SIMD units to run multiple arithmetic operations at the same time as we saw in Section 4.1. Although one iteration in the above loop only has a single add operation, we can explicitly allocate more operations within an iteration, and ask the compiler to use SIMD instructions to process them.

The way to do it is first splitting the one-level for-loop into a two-level nested for-loop using a `factor`. The inner loop consists of `factor` original iterations that will be grouped together accordingly to execute in SIMD instructions on a core. And iterations in the outer loop still run in parallel.

[39] https://en.wikipedia.org/wiki/Round-robin_scheduling

[40] https://en.wikipedia.org/wiki/Producer–consumer_problem

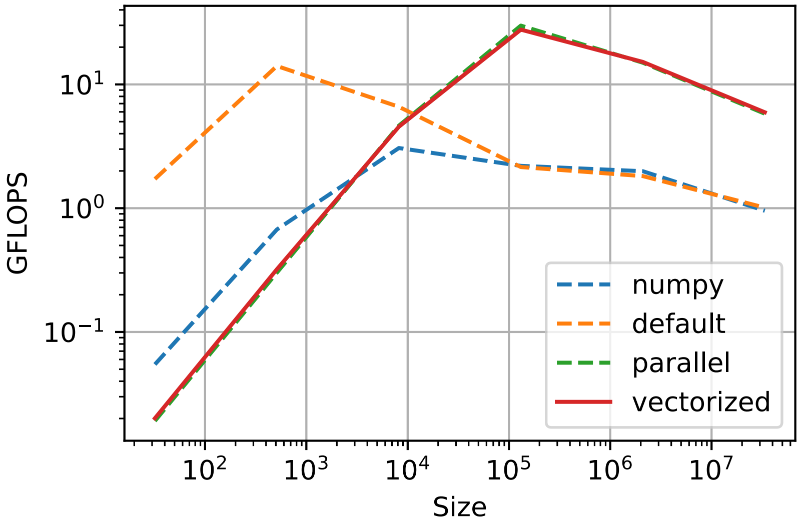
```
def vectorized(n):
    s, (A, B, C) = default(n)
    outer, inner = s[C].split(C.op.axis[0], factor=8)
    s[C].parallel(outer)
    s[C].vectorize(inner)
    return s, (A, B, C)

s, args = vectorized(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce c {
  parallel (i.outer, 0, 8) {
    c[ramp((i.outer*8), 1, 8)] = (a[ramp((i.outer*8), 1, 8)] + b[ramp((i.
↪outer*8), 1, 8)])
  }
}
```

We can see that the outer for-loop is reduced to 8 iterations, while the inner for-loop is vectorized by `ramp` with a stride of 1 and width of 8. The definition of `ramp` is inherited from Halide[41].

Again, we check the performance of vectorization and plot the comparison diagram via the following code block.

```
vectorized_gflops = bench_vector_add_tvm(vectorized, sizes, target)
plot_gflops(sizes, [np_gflops, default_gflops, parallel_gflops, vectorized_
↪gflops],
     ['numpy', 'default', 'parallel', 'vectorized'])
```



The performance of the vectorized version is almost as the plain parallelization version. It's partially because the vector add is bottlenecked by memory bandwidth instead of computation, while SIMD only helps the latter. We will see it helps more on computation intensive workloads such as matrix multiplication later.

[41] https://halide-lang.org/docs/struct_halide_1_1_internal_1_1_ramp.html

### 4.3.5 Summary

- The default scheduling generates naive single-thread CPU program.
- Parallelization improves performance for large workloads.
- We can split a for-loop and then vectorize the inner loop if the system supports SIMD.

## 4.4 Broadcast Add

This section talks about scheduling the broadcast add computation defined in Section 3.1 on CPU. The optimization of it is similar to what we have done for vector add in Section 4.3, as they are both essentially memory-bound element-wise calculation. Hence, instead of "what works", we will talk about "what doesn't work" in this section.

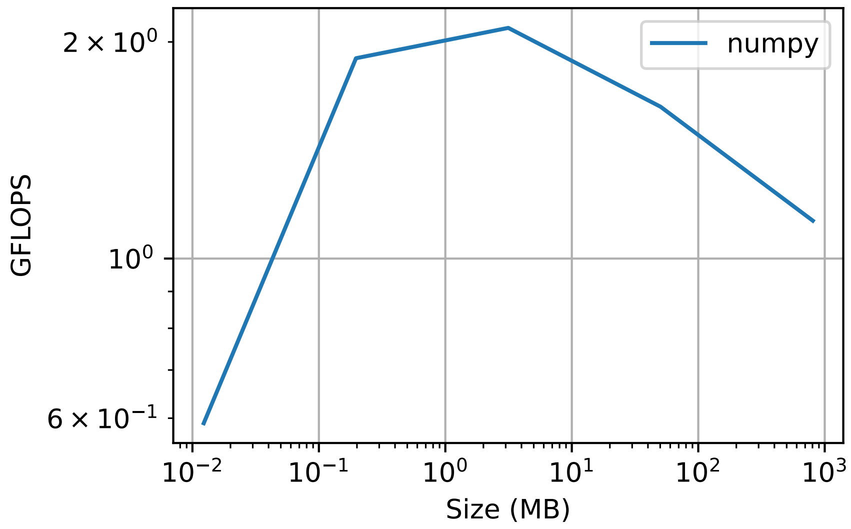
### 4.4.1 Setup

```
%matplotlib inline
import d2ltvm
import inspect
from IPython import display
import numpy as np
from matplotlib import pyplot as plt
import timeit
import tvm
from tvm import te
```

Here we choose the broadcast add depicted in Fig. 3.1.1. The other broadcast patterns do not make essential difference.

First, we define the baseline in `numpy`. We use the real used data sizes in the x axis.

```
sizes = 2**np.arange(5, 15, 2)
np_bcast_add = lambda s1, s2: timeit.Timer(setup='import numpy as np\n'
                                'import d2ltvm\n'
                                'a, b, c = d2ltvm.get_bcast_data(%s, %s)' %␣
↪(s1, s2),
                                stmt='np.add(a, b, out=c)')
exe_times = [d2ltvm.bench_workload(np_bcast_add((n, 1), (n, n)).timeit) for n␣
↪in sizes]
np_gflops = sizes * sizes / 1e9 / np.array(exe_times)
# data size in MB
x_axis_sizes = (sizes * sizes * 2 + sizes * sizes) * 4 / 1e6
d2ltvm.plot_gflops(x_axis_sizes, [np_gflops], ['numpy'], xlabel='Size (MB)')
```



Note that the x axis is denoted as the total data size consumed. The performance drops when the data size is larger than the L2 cache of a single core (1024 KB or 1 MB), which indirectly suggests that `numpy` may use single-thread to perform this operator.

### 4.4.2 Good Schedule

The following code block defines a benchmarking method for broadcast add of a specific pattern in TVM. It follows the same format we discussed in Section 4.3.

```
# Save to the d2ltvm package.
def bench_bcast_add_tvm(func, sizes, target):
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(a, b, c).mean * nrepeats
    times = []
    for n in sizes:
        n = int(n)
        s, (A, B, C) = func(n)
        mod = tvm.build(s, [A, B, C], target)
        ctx = tvm.context(target, 0)
        a, b, c = d2ltvm.get_bcast_data((n, 1), (n, n), lambda x: tvm.nd.
↪array(x, ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
    return sizes * sizes / 1e9 / np.array(times)
```

The good schedule (i.e. what works) follows the similar scheme defined in Section 4.3.

```
target = 'llvm -mcpu=skylake-avx512'
def default(n):
    A, B, C = d2ltvm.broadcast_add((n, 1), (n, n))
    s = te.create_schedule(C.op)
    return s, (A, B, C)

def good_schedule(n):
    s, (A, B, C) = default(n)
```

(continues on next page)

(continued from previous page)

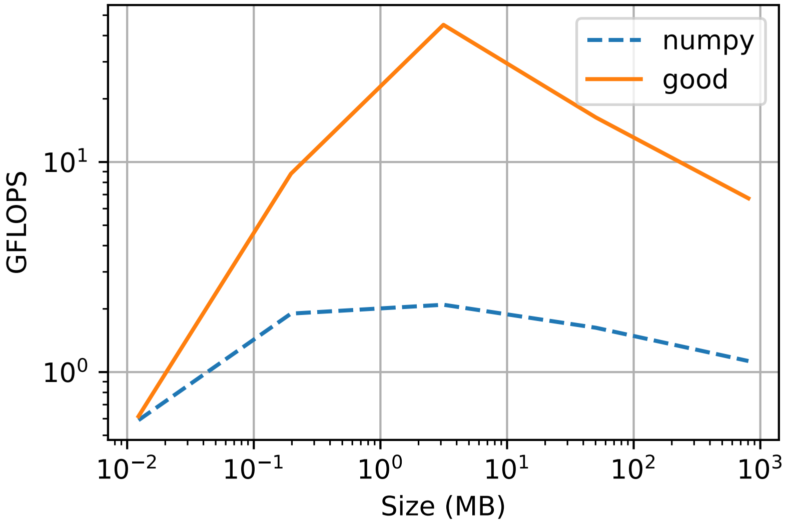
```
    x, y = C.op.axis
    s[C].parallel(x)
    s[C].vectorize(y)
    return s, (A, B, C)

s, args = good_schedule(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce C {
  parallel (x, 0, 64) {
    C[ramp((x*64), 1, 64)] = (x64(A[x]) + B[ramp((x*64), 1, 64)])
  }
}
```

Now the C-like pseudo code should be familiar to you. One notable difference from :numref:ch_vector_add_cpu is that we broadcast `A[x]` to a vectorized register (i.e. `x64(A[x])`) for vectorized add.

Let's benchmark the good schedule.

```
good_gflops = bench_bcast_add_tvm(good_schedule, sizes, target)
d2ltvm.plot_gflops(x_axis_sizes, [np_gflops, good_gflops], ['numpy', 'good'],␣
↪xlabel='Size (MB)')
```



Like the case in Section 4.3, the performance is better and drops at a larger data size as multi-core comes into play, leading to a larger amount L2 cache in total.

### 4.4.3 Bad Schedule

Now let's see how a bad schedule (i.e. what doesn't work) looks like. Note that our data are all stored in row-major and our schedules always make sure that the innermost loop manipulates the data in consecutive space.

However, if we think from the perspective vectorization, we may find that broadcasting `A[x]` to a vectorized register brings overhead. A better way is to have multiple values of `A` to add with multiple values of `B` in a vectorized instruction, which is illustrated in Fig. 4.4.1.

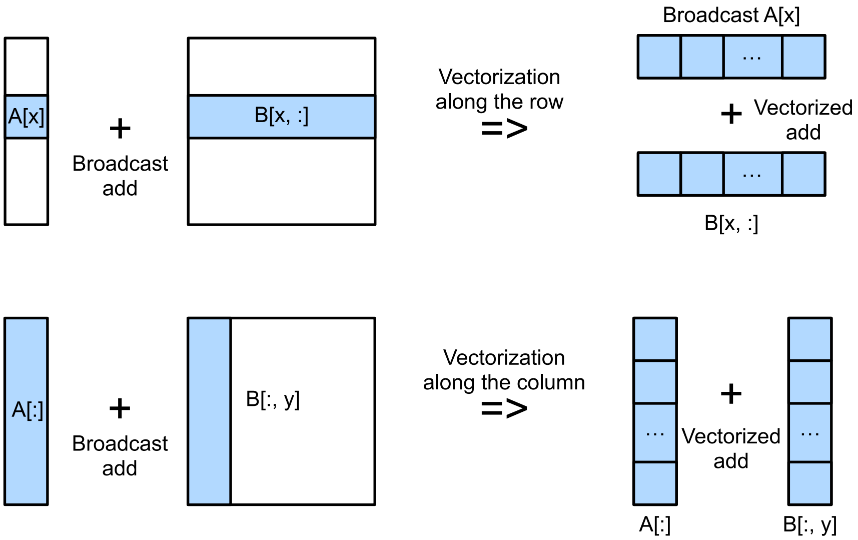


Fig. 4.4.1: Different vectorization strategies of broadcast add.

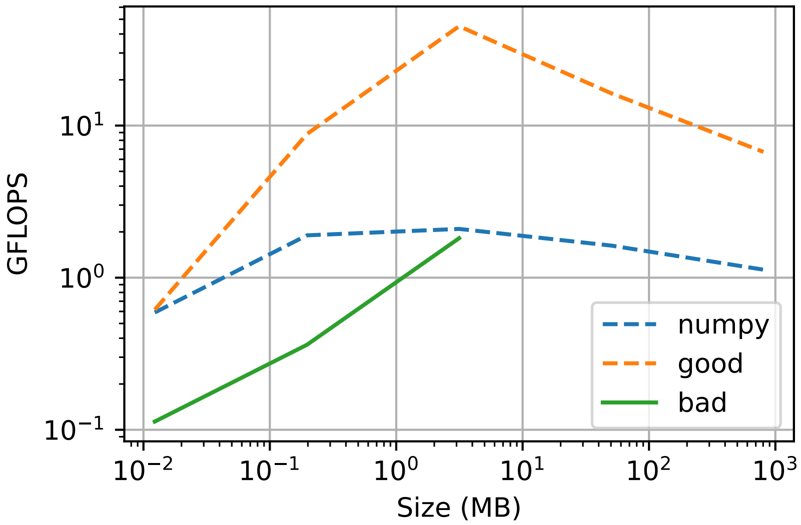
In order to make vectorization along the column, we need to reorder to data access pattern as follows.

```
def bad_schedule(n):
    s, (A, B, C) = default(n)
    x, y = C.op.axis
    s[C].reorder(y, x)
    s[C].parallel(y)
    s[C].vectorize(x)
    return s, (A, B, C)

s, args = bad_schedule(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce C {
  parallel (y, 0, 64) {
    C[ramp(y, 64, 64)] = (A[ramp(0, 1, 64)] + B[ramp(y, 64, 64)])
  }
}
```

Now we eliminate the broadcast of `A[x]` in vectorization. Let's run the benchmark and plot the chart for this schedule.

```
sizes = 2**np.arange(5, 11, 2)
bad_gflops = bench_bcast_add_tvm(bad_schedule, sizes, target)
diff = len(good_gflops)-len(bad_gflops)
bad_gflops = np.append(bad_gflops, [np.nan] * diff)
d2ltvm.plot_gflops(x_axis_sizes, [np_gflops, good_gflops, bad_gflops],
     ['numpy', 'good', 'bad'],  xlabel='Size (MB)')
```



Note that in order to make it finish in a short period of time, we reduce the number of tested sizes for this schedule.

We can see that the performance of the vectorization-favorable schedule is pretty bad. The reason is that we access B in a stride to get the data in the same column, which is much slower than accessing data in a consecutive space. And the thread-level parallelism would make it even worse as different threads compete in getting the data in the same cache line.

The result shows that a good schedule needs to consider multiple performance-related factors together. A scheduling scheme that favors one aspect may lead to bad overall performance as it harms other aspects.

### 4.4.4 Summary

- Like vector add, broadcast add is a memory-bound operator.
- A good schedule needs to consider multiple performance-related factors together.

### 4.4.5 Exercise

- Try to schedule other broadcast add patterns and observe the difference.

## 4.5 Matrix Multiplication

We saw the NumPy `dot` operator nearly reaches the peak performance of our CPU (the Xeon E5-2686 v4) in Section 4.1. In this section, we will investigate multiple scheduling strategies for this operator in TVM.

### 4.5.1 Setup

```
%matplotlib inline
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te

target = 'llvm -mcpu=skylake-avx512'
```

As we did in Section 4.3, we first define a method to measure NumPy performance as our baseline.

```
# Save to the d2ltvm package.
def np_matmul_timer(n):
    timer = timeit.Timer(setup='import numpy as np\n'
                         'import d2ltvm\n'
                         'a, b, c = d2ltvm.get_abc(%s)' % str((n,n)),
                         stmt = 'np.dot(a, b, out=c)')
    return timer.timeit

sizes = 2**np.arange(5, 12, 1)
exe_times = [d2ltvm.bench_workload(np_matmul_timer(n)) for n in sizes]
np_gflops = 2 * sizes **3 / 1e9 / np.array(exe_times)
```

### 4.5.2 Default Schedule

The default schedule consists of three nested for-loops.

```
def default(n):
    A, B, C = d2ltvm.matmul(n, n, n)
    return te.create_schedule(C.op), (A, B, C)

s, args = default(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce C {
  for (x, 0, 64) {
    for (y, 0, 64) {
      C[((x*64) + y)] = 0f
      for (k, 0, 64) {
        C[((x*64) + y)] = (C[((x*64) + y)] + (A[((x*64) + k)]*B[((k*64) +␣
↪y)]))
      }
    }
```

(continues on next page)

(continued from previous page)

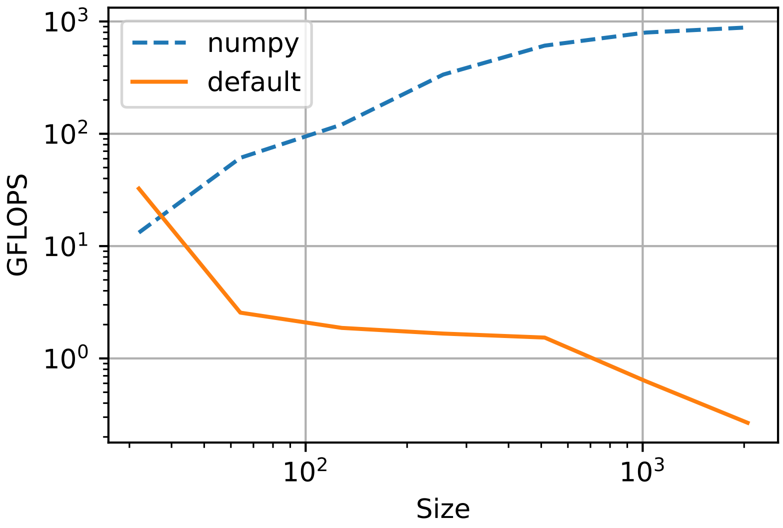
```
  }
}
```

To benchmark its performance, we also define a reusable method as we did in Section 4.3.

```
# Save to the d2ltvm package.
def bench_matmul_tvm(func, sizes, target):
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(a, b, c).mean * nrepeats
    times = []
    for n in sizes:
        s, (A, B, C) = func(int(n))
        mod = tvm.build(s, [A, B, C], target)
        ctx = tvm.context(target, 0)
        a, b, c = d2ltvm.get_abc((n, n), lambda x: tvm.nd.array(x, ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
    return 2 * sizes**3 / 1e9 / np.array(times)
```

The default schedule follows the computation illustrated in Fig. 3.2.1. It's not surprising to see that the default schedule doesn't perform well, especially for large matrices that cannot fit into the cache.

```
default_gflops = bench_matmul_tvm(default, sizes, target)
d2ltvm.plot_gflops(sizes, [np_gflops, default_gflops], ['numpy', 'default'])
```



### 4.5.3 Reordering Axes

The first problem we can see from Fig. 3.2.1 is that matrix $B$ is accessed column by column while its elements are stored by rows (i.e. matrix $B$ is in row-major[42]). In other words, in the pseudo code above, we iterate axis `y` before axis `k`. Simply switching these two for-loops will make all elements read and written sequentially. Fig. 4.5.1 illustrates the changed the data access pattern.

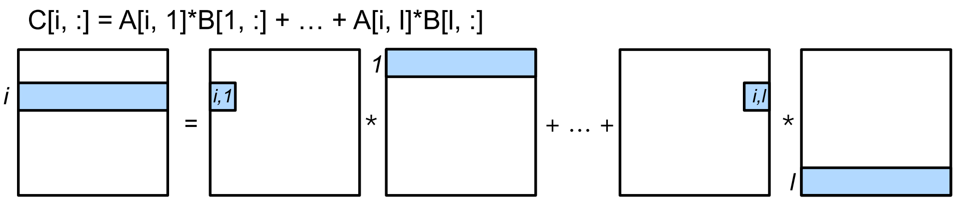


Fig. 4.5.1: Reorder axes in matrix multiplication.

To implement it, we change the axes order from (`x`, `y`, `k`) to (`x`, `k`, `y`) by the `reorder` primitive. The corresponding pseudo code verifies that we are processing all matrices row by row now.

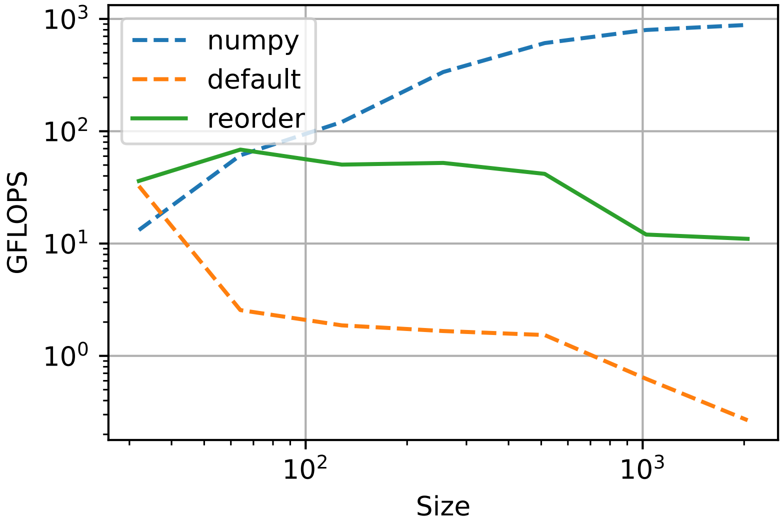
```
def reorder(n):
    s, (A, B, C) = default(n)
    (x, y), (k,) = C.op.axis, C.op.reduce_axis
    s[C].reorder(x, k, y)
    return s, (A, B, C)

s, args = reorder(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce C {
  for (x, 0, 64) {
    for (y.init, 0, 64) {
      C[((x*64) + y.init)] = 0f
    }
    for (k, 0, 64) {
      for (y, 0, 64) {
        C[((x*64) + y)] = (C[((x*64) + y)] + (A[((x*64) + k)]*B[((k*64) +␣
↪y)]))
      }
    }
  }
}
```

We can see that the reordering significantly improves the performance compared to the default schedule.

```
reorder_gflops = bench_matmul_tvm(reorder, sizes, target)
d2ltvm.plot_gflops(sizes, [np_gflops, default_gflops, reorder_gflops],
            ['numpy', 'default', 'reorder'])
```

[42] https://en.wikipedia.org/wiki/Row-_and_column-major_order



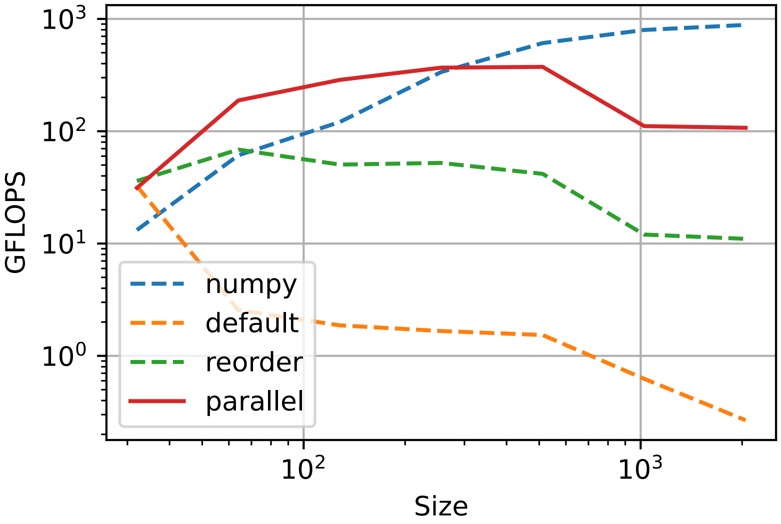
### 4.5.4 Parallelization

Let's revisit the pseudo code above. In the outermost for-loop `for (x, 0, 64)`, each time we compute the results of a row in $C$. Each row can be computed in parallel, so we can make the schedule parallelize axis `x`.

```
def parallel(n):
    s, (A, B, C) = reorder(n)
    s[C].parallel(C.op.axis[0])
    return s, (A, B, C)

s, args = parallel(64)
print(tvm.lower(s, args, simple_mode=True))
```

```
produce C {
  parallel (x, 0, 64) {
    for (y.init, 0, 64) {
      C[((x*64) + y.init)] = 0f
    }
    for (k, 0, 64) {
      for (y, 0, 64) {
        C[((x*64) + y)] = (C[((x*64) + y)] + (A[((x*64) + k)]*B[((k*64) +␣
↪y)]))
      }
    }
  }
}
```

```
parallel_gflops = bench_matmul_tvm(parallel, sizes, target)
d2ltvm.plot_gflops(sizes, [np_gflops, default_gflops, reorder_gflops,␣
↪parallel_gflops],
            ['numpy', 'default', 'reorder', 'parallel'])
```



Parallelization improves the performance again. But we can see that there is still a gap compared to NumPy on large matrices, specially when they cannot fit into the L2 cache. We will try to resolve it in the next section.

### 4.5.5 Summary

- Reordering the for-loops in matrix multiplication properly improves the performance.
- Proper thread-level parallelization also improves the performance.

### 4.5.6 Exercises

1. Change the number of threads
2. Try to order the axes in method `parallel` differently
3. Benchmark matrix multiplication in larger sizes

## 4.6 Improve Cache Efficiency by Blocking

In Section 4.5 we saw that properly reordering the loop axes to get more friendly memory access pattern, together with thread-level parallelization, could dramatically improve the performance for matrix multiplication. The results show that for small-scale matrices, our performance outperforms the NumPy baseline. However, for large matrices, we need to carefully consider the cache hierarchy discussed in Section 4.1.

```
%matplotlib inline
import tvm
from tvm import te
import numpy as np
import d2ltvm

target = 'llvm -mcpu=skylake-avx512'
```

Before we started, let's rerun the benchmark for NumPy as our baseline.

```
sizes = 2**np.arange(5, 12, 1)
exe_times = [d2ltvm.bench_workload(d2ltvm.np_matmul_timer(n)) for n in sizes]
np_gflops = 2 * sizes **3 / 1e9 / np.array(exe_times)
```

### 4.6.1 Blocked Matrix Multiplication

One commonly used strategy is tiling matrices into small blocks that can be fitted into the cache. The math behind it is that a block of $C$, e.g. `C[x:x+tx, y:y+ty]` by the NumPy notation, can be computed by the corresponding rows of $A$ and columns of $B$. That is

```
C[x:x+tx, y:y+ty] = np.dot(A[x:x+tx,:], B[:,y:y+ty])
```

We can further decompose this matrix multiplication into multiple small ones

```
C[x:x+tx, y:y+ty] = sum(np.dot(A[x:x+tx,k:k+tk], B[k:k+tk,y:y+ty]) for k in range(0,n,tk))
```

This computation is also illustrated in Fig. 4.6.1.

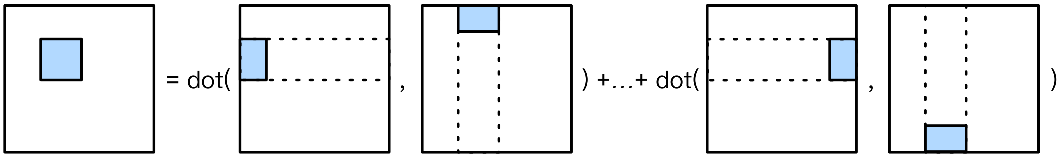


Fig. 4.6.1: Blocked tiling for matrix multiplication.

In each submatrix computation, we need to write a `[tx, ty]` shape matrix, and reach two matrices with shapes `[tx, tk]` and `[tk, ty]`. We can compute such a computation in a single CPU core. If we properly choose the tiling sizes `tx`, `ty` and `tk` to fit into the L1 cache, which is 32KB for our CPU (refer to Section 4.1). The reduced cache miss then should improve the performance.

Let's implement this idea. In the following code block, we choose `tx=ty=32` and `tk=4` so that the submatrix to write has a size of `32*32*4=4KB` and the total size of the two submatrices to read is `2*32*4*4=1KB`. The three matrices together can fit into our L1 cache easily. The tiling is implemented by the `tile` primitive.

After tiling, we merge the outer width and height axes into a single one using the `fuse` primitive, so we can parallelize it. It means that we will compute blocks in parallel. Within a block, we split the reduced axis, reorder the axes as we did in:numref:`ch_matmul_cpu`, and then vectorize the innermost axis using SIMD instructions, and unroll the second innermost axis using the `unroll` primitive, namely the inner reduction axis.

```
tx, ty, tk = 32, 32, 4  # tile sizes

def block(n):
    A, B, C = d2ltvm.matmul(n, n, n)
    s = te.create_schedule(C.op)
    # Tile by blocks, and then parallelize the computation of each block
    xo, yo, xi, yi = s[C].tile(*C.op.axis, tx, ty)
    xy = s[C].fuse(xo, yo)
    s[C].parallel(xy)
    # Optimize the computation of each block
    ko, ki = s[C].split(s[C].op.reduce_axis[0], factor=tk)
```

(continues on next page)

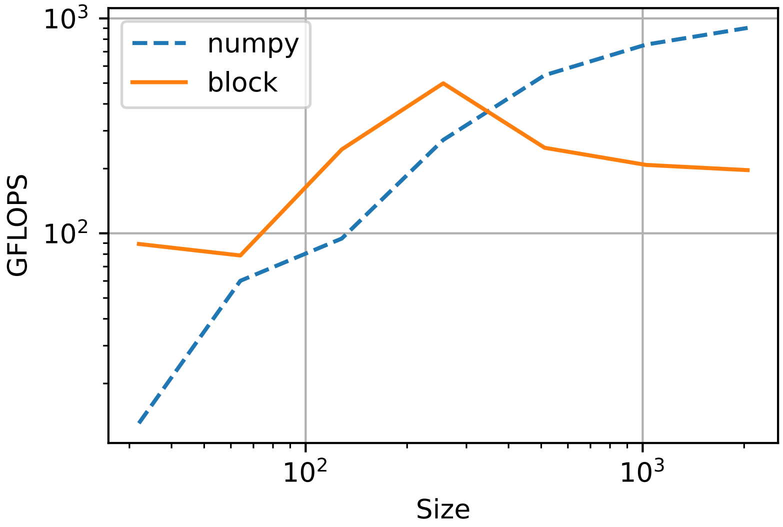
(continued from previous page)

```
    s[C].reorder(ko, xi, ki, yi)
    s[C].vectorize(yi)
    s[C].unroll(ki)
    return s, (A, B, C)

s, (A, B, C) = block(64)
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

```
produce C {
  parallel (x.outer.y.outer.fused, 0, 4) {
    for (x.inner.init, 0, 32) {
      C[ramp((((floordiv(x.outer.y.outer.fused, 2)*2048) + (x.inner.init*64))␣
↪+ (floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)] = x32(0f)
    }
    for (k.outer, 0, 16) {
      for (x.inner, 0, 32) {
        C[ramp((((floordiv(x.outer.y.outer.fused, 2)*2048) + (x.inner*64)) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)] = (C[ramp((((floordiv(x.
↪outer.y.outer.fused, 2)*2048) + (x.inner*64)) + (floormod(x.outer.y.
↪outer.fused, 2)*32)), 1, 32)] + (x32(A[(((floordiv(x.outer.y.outer.
↪fused, 2)*2048) + (x.inner*64)) + (k.outer*4))])*B[ramp(((k.outer*256) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)]))
        C[ramp((((floordiv(x.outer.y.outer.fused, 2)*2048) + (x.inner*64)) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)] = (C[ramp((((floordiv(x.
↪outer.y.outer.fused, 2)*2048) + (x.inner*64)) + (floormod(x.outer.y.outer.
↪fused, 2)*32)), 1, 32)] + (x32(A[((((floordiv(x.outer.y.outer.fused,␣
↪2)*2048) + (x.inner*64)) + (k.outer*4)) + 1)])*B[ramp((((k.outer*256) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)) + 64), 1, 32)]))
        C[ramp((((floordiv(x.outer.y.outer.fused, 2)*2048) + (x.inner*64)) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)] = (C[ramp((((floordiv(x.
↪outer.y.outer.fused, 2)*2048) + (x.inner*64)) + (floormod(x.outer.y.outer.
↪fused, 2)*32)), 1, 32)] + (x32(A[((((floordiv(x.outer.y.outer.fused,␣
↪2)*2048) + (x.inner*64)) + (k.outer*4)) + 2)])*B[ramp((((k.outer*256) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)) + 128), 1, 32)]))
        C[ramp((((floordiv(x.outer.y.outer.fused, 2)*2048) + (x.inner*64)) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)), 1, 32)] = (C[ramp((((floordiv(x.
↪outer.y.outer.fused, 2)*2048) + (x.inner*64)) + (floormod(x.outer.y.outer.
↪fused, 2)*32)), 1, 32)] + (x32(A[((((floordiv(x.outer.y.outer.fused,␣
↪2)*2048) + (x.inner*64)) + (k.outer*4)) + 3)])*B[ramp((((k.outer*256) +␣
↪(floormod(x.outer.y.outer.fused, 2)*32)) + 192), 1, 32)]))
      }
    }
  }
}
```

From the generated C-like codes, we can see that `parallel` is placed on the `x.outer`, i.e. `xo`, axis. The vectorization translated the axis `yi`, whose length is 32, into `ramp` with a stride 1 and width 32. Besides, the axis `ki` is also unrolled into 4 sequential operations to reduce the cost of the for-loop.

```
blocked_gflops = d2ltvm.bench_matmul_tvm(block, sizes, target)
d2ltvm.plot_gflops(sizes, [np_gflops, blocked_gflops], ['numpy', 'block'])
```



The benchmark results show that our program is as good as NumPy for small matrices, but still doesn't do well for large ones. One major reason is because both read and write of these submatrices are not continuous after tiling.

### 4.6.2 Write Cache

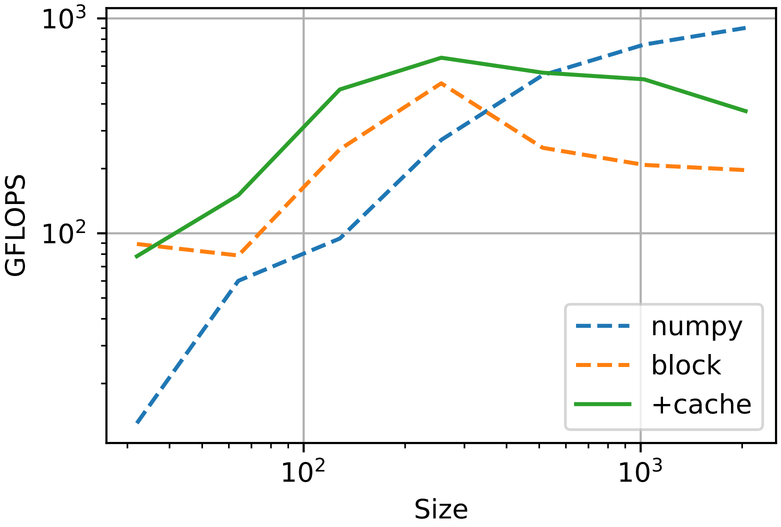
The non-continuous write issue is severer than the non-continuous read. This is because we read once of each submatrix of A and B, but need to write by n times for the submatrix of C. In the following code block, we first write the results into a local buffer for each submatrix computation, and then write them back to C. It can be done by the cache_write method. We specify the buffer being used for each block by placing it within the yo axis using the compute_at primitive. The rest optimization is the same as before, but note that we need to use s[Cached] instead of s[C] to optimize the submatrix computation.

```
def cached_block(n):
    A, B, C = d2ltvm.matmul(n, n, n)
    s = te.create_schedule(C.op)
    # Create a write cache for C
    CachedC = s.cache_write(C, 'local')
    # Same as before, first tile by blocks, and then parallelize the
    # computation of each block
    xo, yo, xi, yi = s[C].tile(*C.op.axis, tx, ty)
    xy = s[C].fuse(xo, yo)
    s[C].parallel(xy)
    # Use the write cache for the output of the xy axis, namely a block.
    s[CachedC].compute_at(s[C], xy)
    # Same as before to optimize the computation of a block .
    xc, yc = s[CachedC].op.axis
    ko, ki = s[CachedC].split(CachedC.op.reduce_axis[0], factor=tk)
    s[CachedC].reorder(ko, xc, ki, yc)
    s[CachedC].unroll(ki)
    s[CachedC].vectorize(yc)
    return s, (A, B, C)

s, (A, B, C) = cached_block(512)
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

```
produce C {
  parallel (x.outer.y.outer.fused, 0, 256) {
    // attr [C.local] storage_scope = "local"
    allocate C.local[float32 * 1024]
    produce C.local {
      for (x.c.init, 0, 32) {
        C.local[ramp((x.c.init*32), 1, 32)] = x32(0f)
      }
      for (k.outer, 0, 128) {
        for (x.c, 0, 32) {
          C.local[ramp((x.c*32), 1, 32)] = (C.local[ramp((x.c*32), 1, 32)]␣
↪+ (x32(A[(((floordiv(x.outer.y.outer.fused, 16)*16384) + (x.c*512)) +␣
↪(k.outer*4))])*B[ramp(((k.outer*2048) + (floormod(x.outer.y.outer.fused,␣
↪16)*32)), 1, 32)]))
          C.local[ramp((x.c*32), 1, 32)] = (C.local[ramp((x.c*32), 1, 32)]␣
↪+ (x32(A[((((floordiv(x.outer.y.outer.fused, 16)*16384) + (x.c*512)) + (k.
↪outer*4)) + 1)])*B[ramp((((k.outer*2048) + (floormod(x.outer.y.outer.fused,␣
↪16)*32)) + 512), 1, 32)]))
          C.local[ramp((x.c*32), 1, 32)] = (C.local[ramp((x.c*32), 1, 32)]␣
↪+ (x32(A[((((floordiv(x.outer.y.outer.fused, 16)*16384) + (x.c*512)) + (k.
↪outer*4)) + 2)])*B[ramp((((k.outer*2048) + (floormod(x.outer.y.outer.fused,␣
↪16)*32)) + 1024), 1, 32)]))
          C.local[ramp((x.c*32), 1, 32)] = (C.local[ramp((x.c*32), 1, 32)]␣
↪+ (x32(A[((((floordiv(x.outer.y.outer.fused, 16)*16384) + (x.c*512)) + (k.
↪outer*4)) + 3)])*B[ramp((((k.outer*2048) + (floormod(x.outer.y.outer.fused,␣
↪16)*32)) + 1536), 1, 32)]))
        }
      }
    }
    for (x.inner, 0, 32) {
      for (y.inner, 0, 32) {
        C[((((floordiv(x.outer.y.outer.fused, 16)*16384) + (x.inner*512))␣
↪+ (floormod(x.outer.y.outer.fused, 16)*32)) + y.inner)] = C.local[((x.
↪inner*32) + y.inner)]
      }
    }
  }
}
```

Note from the generated pseudo codes that we initialize `C.local` within the `yo` axis, and the size of `C.local` is `tx * ty = 1024`.

```
cached_gflops = d2ltvm.bench_matmul_tvm(cached_block, sizes, target)
d2ltvm.plot_gflops(sizes, [np_gflops, blocked_gflops, cached_gflops],
            ['numpy', 'block', '+cache'])
```



We can see the the write cache improves the performance for matrix multiplication on large sizes.

### 4.6.3 Summary

- Blocked tiling improves cache efficiency for matrix multiplication.
- Data to be frequently read and written should be placed in a buffer explicitly to reduce cache misses.

### 4.6.4 Exercises

1. Try different hyperparameters for `tx`, `ty` and `tx`.
2. Try different axis orders.
3. Benchmark on larger matrices, observe if there is still performance gap between NumPy. If so, try to explain the reason.
4. Evaluate the correctness of the computed results.

## 4.7 Convolution

In this section, we will optimize the convolution operator defined in Section 3.3 on CPUs. Specifically, this is a 2-D convolution operator.

### 4.7.1 Setup

```
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te

target = 'llvm -mcpu=skylake-avx512'
```

Let's first define our performance baseline, which is the convolution operator provided by MXNet, backing up by Intel MKL-DNN library. MKL-DNN uses OpenMP compiled by Intel compiler to implement the thread-level parallelization. In order to match the thread behavior of TVM described in Section 4.3, we specify the following Intel OpenMP environment variables. Without setting them, sometimes the multi-thread performance of MKL-DNN kernels may not be stable.

```
import os
os.environ['KMP_AFFINITY']='granularity=fine,noduplicates,compact,1,0'
```

To simplify the measurement, we use the same number of input and output channels, and the same size of height and width of input and kernel tensors. The padding will be `(kernel - 1) // 2` and the stride will be 1 so that the output will have the same width and height as the input, i.e. `SAME` padding.

```
# Save to the d2ltvm package.
def conv_gflop(oc, ic, n, k, p, s):
    """Compute the #floating point operations in a convolution.

    The arguments are output channels oc, input channels ic, input size n,
    kernel size k, padding p and stride s.
    """
    on = d2ltvm.conv_out_size(n, k, p, s)
    return 2 * oc * ic * on * on * k * k / 1e9

# Save to the d2ltvm package.
def conv_timer_mxnet(c, n, k, ctx):
    """Benchmark convolution in MXNet

    c : input, output channels
    n : input width and height
    k : kernel width and height
    """
    timer = timeit.Timer(
        setup='import d2ltvm\n'
        'import mxnet as mx\n'
        'c, n, k, p, s = %d, %d, %d, %d, 1\n'
        'data, weight, bias, out = d2ltvm.get_conv_data_mxnet(\n'
        '    c, c, n, k, p, s, "%s")'%(c, n, k, (k-1)//2, ctx),
        stmt='d2ltvm.conv_mxnet(data, weight, bias, out, k, p, s);'
        'out.wait_to_read()')
    return timer.timeit

# Save to the d2ltvm package.
def bench_conv_mxnet(sizes, ctx='cpu'):
    """Return the GFLOPS of MXNet convolution"""
```

(continues on next page)

(continued from previous page)

```
    return [conv_gflop(c, c, n, k, (k-1)//2, 1) /
            d2ltvm.bench_workload(conv_timer_mxnet(c, n, k, ctx))
            for c, n, k in sizes]
```

Then benchmark its performance with various numbers of channels, when the input and kernel width/height are fixed to be 64 and 3, respectively.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]
mxnet_gflops = bench_conv_mxnet(sizes)
```

Similar to the `bench_matmul_tvm` method implemented in Section 4.5, we implement the reusable method to benchmark the convolution operator. Note that the first argument `func` here is a function that takes input arguments `oc, ic, n, k, p, s` and returns the schedule as well as its corresponding tensor symbolic arguments.

```
# Save to the d2ltvm package.
def bench_conv_tvm(func, sizes, target):
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(x, k, y).mean * nrepeats
    gflops, times = [], []
    for (c, n, k) in sizes:
        args = c, c, n, k, (k-1)//2, 1 # oc, ic, n, k, p, s
        s, (X, K, Y) = func(*args)
        mod = tvm.build(s, [X, K, Y], target)
        ctx = tvm.context(target, 0)
        x, k, y = d2ltvm.get_conv_data(
            *args, lambda x: tvm.nd.array(x, ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
        gflops.append(conv_gflop(*args))
    return np.array(gflops) / np.array(times)
```

## 4.7.2 Blocked Convolution

Just as we tiled matrices into blocks to improve the cache efficiency for matrix multiplication in Section 4.6, we can do the same thing for convolution. Let's consider the case with a single output channel for simplicity.

Consider a block of $Y$, denoted as `Y[i:i+tw, j:j+th]`, where `th` and `tw` are the tile sizes for height and width, respectively. We know the it can be computed by applying the convolution on a block of $X$ with kernel $K$. In particular, assuming the width and height of $K$ are `kw` and `kh`, then

```
Y[i:i+tw, j:j+th] = conv(X[i:i+tw+kw-1, j:j+th+kh-1], K)
```

The elements that needed for a block are illustrated in Fig. 4.7.1. If we choose proper `tw` and `th`, we can fit data into the cache to improve its efficiency. This is easy to be generated to multiple output channels as it simply adds the number of kernels accordingly.

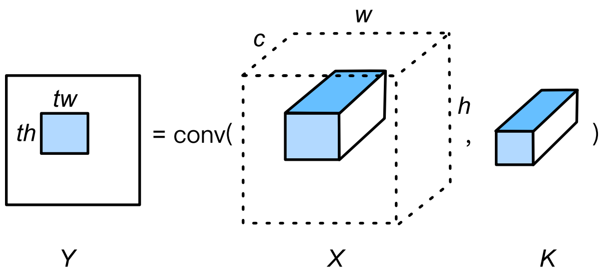


Fig. 4.7.1: Compute a block of $Y$ in convolution.

The implementation of this schedule is similar to the one for matrix multiplication defined in Section 4.6. But also note the following two differences:

1. We parallelize the outer data blocks across the output channels.
2. We move the inner height and width axes before the reduction axes, since they are more effectively to be optimized through vectorization and unrolling.

```python
th, tw = 8, 8  # Tile sizes for height and weight

def cached_block(oc, ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.conv(oc, ic, n, n, k, k, p, p, s, s)
    s = te.create_schedule(Y.op)
    CachedY = s.cache_write(Y, 'local')
    # Compute the output block for every output channel in parallel
    oc, h, w = Y.op.axis
    ho, wo, hi, wi = s[Y].tile(h, w, th, tw)
    ochw = s[Y].fuse(oc, ho, wo)
    s[Y].parallel(ochw)
    # Cache the output block, and move the inner height and width axes
    # to innermost, so we can vectorize and unroll them
    s[CachedY].compute_at(s[Y], ochw)
    _,  ch, cw = CachedY.op.axis
    ric, rkh, rkw = CachedY.op.reduce_axis
    s[CachedY].reorder(ric, rkh, rkw, ch, cw)
    s[CachedY].vectorize(cw)
    s[CachedY].unroll(ch)
    # Schedule the padding by adding thread-level parallelism
    if PaddedX != X:
        s[PaddedX].parallel(PaddedX.op.axis[0])
    return s, (X, K, Y)

s, args = cached_block(32, 32, 64, 3, 1, 1)
tvm.lower(s, args, simple_mode=True)
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 139392]
produce PaddedX {
  parallel (i0, 0, 32) {
    for (i1, 0, 66) {
      for (i2, 0, 66) {
        PaddedX[(((i0*4356) + (i1*66)) + i2)] = tvm_if_then_else((((((i1 < 1)␣
↪|| (65 <= i1)) || (i2 < 1)) || (65 <= i2)), 0f, X[((((i0*4096) + (i1*64)) +␣
↪i2) - 65)])
```

(continues on next page)

(continued from previous page)

```
      }
    }
  }
}
produce Y {
  parallel (c.i.outer.fused.j.outer.fused, 0, 2048) {
    // attr [Y.local] storage_scope = "local"
    allocate Y.local[float32 * 64]
    produce Y.local {
      Y.local[ramp(0, 1, 8)] = x8(0f)
      Y.local[ramp(8, 1, 8)] = x8(0f)
      Y.local[ramp(16, 1, 8)] = x8(0f)
      Y.local[ramp(24, 1, 8)] = x8(0f)
      Y.local[ramp(32, 1, 8)] = x8(0f)
      Y.local[ramp(40, 1, 8)] = x8(0f)
      Y.local[ramp(48, 1, 8)] = x8(0f)
      Y.local[ramp(56, 1, 8)] = x8(0f)
      for (ric, 0, 32) {
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            Y.local[ramp(0, 1, 8)] = (Y.local[ramp(0, 1, 8)] +␣
↪(PaddedX[ramp((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.fused,␣
↪64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(8, 1, 8)] = (Y.local[ramp(8, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,␣
↪8)*8)) + rkw) + 66), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.fused,
↪ 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(16, 1, 8)] = (Y.local[ramp(16, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 132), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(24, 1, 8)] = (Y.local[ramp(24, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 198), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(32, 1, 8)] = (Y.local[ramp(32, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 264), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(40, 1, 8)] = (Y.local[ramp(40, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 330), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
            Y.local[ramp(48, 1, 8)] = (Y.local[ramp(48, 1, 8)] +␣
↪(PaddedX[ramp(((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 396), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)])))
```

(continues on next page)

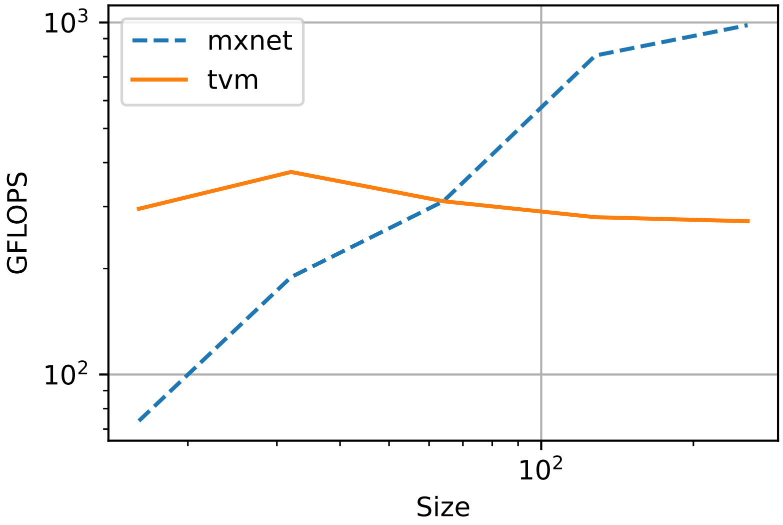
(continued from previous page)

```
            Y.local[ramp(56, 1, 8)] = (Y.local[ramp(56, 1, 8)] +␣
↪(PaddedX[ramp((((((ric*4356) + (floordiv(floormod(c.i.outer.fused.j.outer.
↪fused, 64), 8)*528)) + (rkh*66)) + (floormod(c.i.outer.fused.j.outer.fused,
↪ 8)*8)) + rkw) + 462), 1, 8)]*x8(K[((((floordiv(c.i.outer.fused.j.outer.
↪fused, 64)*288) + (ric*9)) + (rkh*3)) + rkw)]))))
          }
        }
      }
    }
    for (i.inner, 0, 8) {
      for (j.inner, 0, 8) {
        Y[((((floordiv(c.i.outer.fused.j.outer.fused, 8)*512) + (i.inner*64))␣
↪+ (floormod(c.i.outer.fused.j.outer.fused, 8)*8)) + j.inner)] = Y.local[((i.
↪inner*8) + j.inner)]
      }
    }
  }
}
```

The output pseudo code becomes a bit long, but as long as you understand the scheduling primitives we used above, the code should make sense if you take a close look.

Now let's benchmark its performance and compare with our baseline.

```
tvm_gflops = bench_conv_tvm(cached_block, sizes, target)
d2ltvm.plot_gflops(channels, [mxnet_gflops, tvm_gflops], ['mxnet', 'tvm'])
```



As we can seen that our schedule does well for small channel sizes, while its performance decrease for large channel sizes. We will optimize the later in Section 4.8.

### 4.7.3 Summary

- We tile the input and output data of convolution as we did for matrix multiplication for better cache efficiency.

### 4.7.4 Exercises

- Try different tiling sizes.

## 4.8 Packed Convolution

We observed in Section 4.7 that the performance degrades when increasing the channel size for the convolution operator. In this section, we will consider tiling the channel axes and packing the data to improve the performance.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
import os

os.environ['KMP_AFFINITY']='granularity=fine,noduplicates,compact,1,0'
target = 'llvm -mcpu=skylake-avx512'
```

### 4.8.1 Packing Data and Weight

In Section 4.7, we first tiled the width and height axes and then moved the inner axes as the innermost dimensions during computing. It's the same idea to tile the channels. But we do one additional step to actually move the data in the inner channel loop to the last dimension, i.e. do data layout transformation. So we pay the data movement cost once, while improving the data accessing performance significantly.

The following code block splits the channel dimension and move the data in NumPy.

```
c, n, tc = 4, 2, 2  # channel, height/width, and tiling size
x = np.arange(c*n*n).reshape((c, n, n)).astype('float32')
print('input shape', x.shape, '\n', x)
y = x.reshape(c//tc, n, n, tc).transpose(0, 2, 3, 1)
print('packed shape', y.shape, '\n', y)
```

```
input shape (4, 2, 2)
 [[[ 0.  1.]
  [ 2.  3.]]

 [[ 4.  5.]
  [ 6.  7.]]

 [[ 8.  9.]
  [10. 11.]]
```

(continues on next page)

(continued from previous page)

```
 [[12. 13.]
  [14. 15.]]]
packed shape (2, 2, 2, 2)
 [[[[ 0.   4.]
   [ 1.   5.]]

  [[ 2.   6.]
   [ 3.   7.]]]


 [[[ 8. 12.]
   [ 9. 13.]]

  [[10. 14.]
   [11. 15.]]]]
```

Now let's implement the pack computation in TVM.

```python
def conv_pack(oc, ic, nh, nw, kh, kw, ph, pw, toc, tic):
    """Pack data and weight for convolution

    oc, ic : output and input channels
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding
    toc, tic : the tiling sizes of the output and input channels
    """
    X = te.placeholder((ic, nh, nw), name='X')
    K = te.placeholder((oc, ic, kh, kw), name='K')
    PaddedX = d2ltvm.padding(X, ph, pw) if ph * pw != 0 else X
    # pack X and K
    assert ic % tic == 0 and oc % toc == 0
    PackedX = te.compute(
        (ic//tic, nh+ph*2, nw+pw*2, tic),
        lambda ic_out, x, y, ic_in: PaddedX[ic_out*tic + ic_in, x, y],
        name='PackedX')
    PackedK = te.compute(
        (oc//toc, ic//tic, kh, kw, tic, toc),
        lambda oc_out, ic_out, x, y, ic_in, oc_in: K[
            oc_out*toc + oc_in, ic_out*tic + ic_in, x, y],
        name='PackedK')
    return X, K, PaddedX, PackedX, PackedK
```

Verify the results by re-implementing the previous example.

```python
X, _, _, PackedX, _ = conv_pack(c, c, n, n, 1, 1, 0, 0, tc, tc)
mod = tvm.build(te.create_schedule(PackedX.op), [X, PackedX])
packed_x = tvm.nd.array(np.empty((c//tc, n, n, tc), dtype='float32'))
mod(tvm.nd.array(x), packed_x)
np.testing.assert_equal(packed_x.asnumpy(), y)
```

Of note, in Section 3.3 we defined the layout of the input data to be `NCHW`. For the packed data `PackedX` here, we follow the convention defined in (Liu et al., 2019) to define its layout as `NCHW[x]c`, where `x` describes the size of `c`, which is 2 (the value of `tc` above) in this case.

### 4.8.2 Computation

Since we changed the data layout, we need to re-implement the convolution computation accordingly.

```
def conv(oc, ic, nh, nw, kh, kw, ph, pw, sh, sw, toc, tic):
    """2-D conv

    oc, ic : output and input channels.
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding
    sh, sw : height and width strides
    toc, tic : the tiling sizes of output channel and input channel
    """
    X, K, PaddedX, PackedX, PackedK = conv_pack(
        oc, ic, nh, nw, kh, kw, ph, pw, toc, tic)
    # reduction axes
    ric_in = te.reduce_axis((0, tic), name='ric_in')
    ric_out = te.reduce_axis((0, ic//tic), name='ric_out')
    rkh = te.reduce_axis((0, kh), name='rkh')
    rkw = te.reduce_axis((0, kw), name='rkw')
    # output height and weights
    oh = d2ltvm.conv_out_size(nh, kh, ph, sh)
    ow = d2ltvm.conv_out_size(nw, kw, pw, sw)
    # Compuated Y in the packed layout
    PackedY = te.compute(
        (oc//toc, oh, ow, toc),
        lambda oc_out, x, y, oc_in: te.sum(
            PackedX[ric_out, x*sh+rkh, y*sw+rkw, ric_in] *
            PackedK[oc_out, ric_out, rkh, rkw, ric_in, oc_in],
            axis=[ric_out, rkh, rkw, ric_in]), name='Y')
    # Unpack the result
    Y = te.compute((oc, oh, ow),
                   lambda oc, x, y: PackedY[oc//toc, x, y, oc%toc],
                   name='Y')
    return X, K, Y, PaddedX, PackedX, PackedK, PackedY
```

Let’s compile it using the default scheduling and compute the results.

```
oc, ic, n, k, p, s, toc, tic = 4, 6, 12, 3, 1, 1, 2, 3
X, K, Y, _, _, _, _ = conv(oc, ic, n, n, k, k, p, p, s, s, toc, tic)
mod = tvm.build(te.create_schedule(Y.op), [X, K, Y])

data, weight, out = d2ltvm.get_conv_data(oc, ic, n, k, p, s, tvm.nd.array)
mod(data, weight, out)
```

And then verify the result.

```
data, weight, bias, out_mx = d2ltvm.get_conv_data_mxnet(oc, ic, n, k, p, s)
d2ltvm.conv_mxnet(data, weight, bias, out_mx, k, p, s)
np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 4.8.3 Schedule

The optimization strategy here is similar to Section 4.7. The major differences are

1. the innermost axis is the inner axis split from the output channel because the elements have already sit on the last dimension after packing.
2. We only split the width dimension instead of both width and height dimensions.
3. We need to schedule the packing and unpacking computations as well.

```
# tiling sizes for output channel, input channel, and width
toc, tic, tw = 16, 16, 4

def cached_block(oc, ic, n, k, p, s):
    X, K, Y, PaddedX, PackedX, PackedK, PackedY = conv(
        oc, ic, n, n, k, k, p, p, s, s, toc, tic)
    s = te.create_schedule(Y.op)
    CachedY = s.cache_write(PackedY, 'local')
    oc_out, h, w, oc_in = s[PackedY].op.axis
    oc_out_h = s[PackedY].fuse(oc_out, h)
    # Parallel on the first two dimensions oc_out and h
    s[PackedY].parallel(oc_out_h)
    # Optimize the computation of a cached output block
    w_out, w_in = s[PackedY].split(w, factor=tw)  # Split the columns
    s[CachedY].compute_at(s[PackedY], w_out)
    _, _, cw, oc_in = CachedY.op.axis
    ric_out, rkh, rkw, ric_in = CachedY.op.reduce_axis
    s[CachedY].reorder(ric_out, rkh, rkw, ric_in, cw, oc_in)
    s[CachedY].unroll(ric_in)
    s[CachedY].unroll(cw)
    s[CachedY].vectorize(oc_in)
    # Schedule the padding by adding thread-level parallelism
    if PaddedX != X:
        s[PaddedX].parallel(PaddedX.op.axis[0])
    # Optimize the packing of X and K
    s[PackedX].parallel(s[PackedX].fuse(*PackedX.op.axis[0:2]))
    s[PackedX].unroll(PackedX.op.axis[-1])
    s[PackedK].parallel(s[PackedK].fuse(*PackedK.op.axis[0:2]))
    s[PackedK].unroll(PackedK.op.axis[-1])
    # Optimize the unpacking of Y
    s[Y].parallel(s[Y].fuse(*Y.op.axis[0:2]))
    s[Y].unroll(Y.op.axis[-1])
    return s, (X, K, Y)

s, args = cached_block(32, 32, 64, 3, 1, 1)
# Uncomment the following line to see the long
# psuedo codes because of unrolling.
# tvm.lower(s, args, simple_mode=True)
```

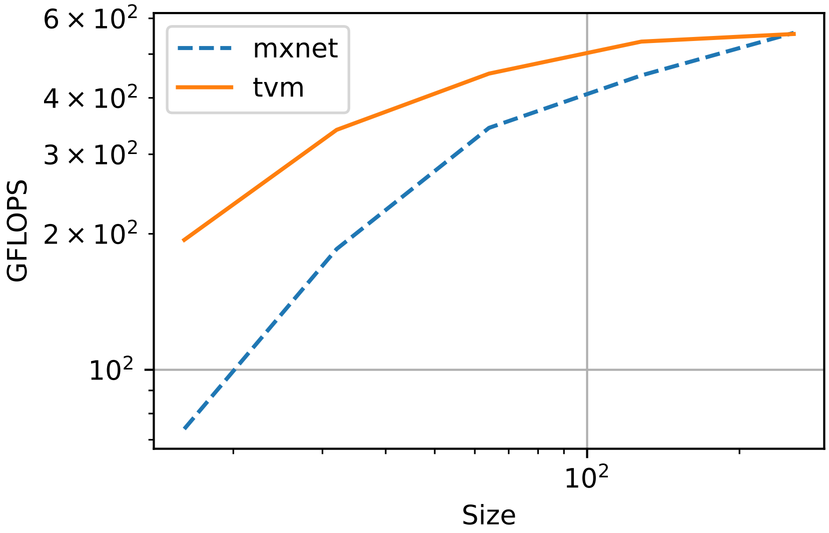
Let's benchmark then same workloads as Section 4.7 and compare to our MXNet baseline. As you can see, the results are significantly improved.

```
channels = 2**np.arange(4, 9)
sizes = [(int(c), 64, 3) for c in channels] # a list of (c, n, k)
tvm_gflops = d2ltvm.bench_conv_tvm(cached_block, sizes, target)
```

(continues on next page)

(continued from previous page)

```
mxnet_gflops = d2ltvm.bench_conv_mxnet(sizes)
d2ltvm.plot_gflops(channels, [mxnet_gflops, tvm_gflops], ['mxnet', 'tvm'])
```



### 4.8.4 Summary

- We often tile input and output channels to for better cache efficiency and pack data accordingly.

### 4.8.5 Exercises

- What if the number of channels cannot be divided by the tiling size?
- Try different tiling sizes.

## 4.9 Depthwise Convolution

In this section, we will follow the packing idea presented in Section 4.8 to re-defined the computation of depthwise convolution and schedule it to run efficiently on CPUs. Similar to the 2-D convolution, tiling the data along the channel dimension and packing it into `NCHW[x]c` benefit the performance significantly.

```
import d2ltvm
import numpy as np
import tvm
from tvm import te
import timeit
import os
os.environ['KMP_AFFINITY']='granularity=fine,noduplicates,compact,1,0'

target = 'llvm -mcpu=skylake-avx512'
```

### 4.9.1 Packing Data and Weight

Recall the depthwise convolution described in Section 3.4, it differs from the 2-D convolution by having each channel of the input data convolved with a separated kernel. Therefore, the packing mechanism of input data is exactly the same as we did in Section 4.8. Kernel is a bit different, as the size is in `[oc, 1, kh, kw]`, which means that there is no need to tile the input channel.

In other words, in the packing method below, we only pass one argument `c` to depict the channel, and another argument `tc` to depict the tiling size of channels. Other than that, it works very similarly as the `conv_pack` method defined in Section 4.8.

```python
def depthwise_conv_pack(c, nh, nw, kh, kw, ph, pw, tc):
    """Pack data and weight for depthwise convolution
       Note that the input channel of kernel is specified as 1,
       and the output channel of kernel equals the input channel of data

    c : input channel of data and output channel of kernel
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding
    tc : the tiling size of channels
    """
    X = te.placeholder((c, nh, nw), name='X')
    K = te.placeholder((c, 1, kh, kw), name='K')
    PaddedX = d2ltvm.padding(X, ph, pw) if ph * pw != 0 else X
    # make sure the channel tiling is valid
    if c < tc:
        tc = c
    assert c % tc == 0
    # pack X and K
    PackedX = te.compute(
        (c//tc, nh+ph*2, nw+pw*2, tc),
        lambda c_out, x, y, c_in: PaddedX[c_out*tc + c_in, x, y],
        name='PackedX')
    PackedK = te.compute(
        (c//tc, 1, kh, kw, 1, tc),
        lambda c_out, _, x, y, __, c_in: K[
            c_out*tc + c_in, 0, x, y],
        name='PackedK')
    return X, K, PaddedX, PackedX, PackedK
```

### 4.9.2 Computation

Like in Section 4.8, we also need to re-implement the depthwise convolution computation accordingly.

```python
def depthwise_conv(c, nh, nw, kh, kw, ph, pw, sh, sw, tc):
    """depthwise conv

    c : number of channels for both input and output.
    nh, nw : input width and height
    kh, kw : kernel width and height
    ph, pw : height and width padding
    sh, sw : height and width strides
    tc : the tiling sizes of channels
```

(continues on next page)

(continued from previous page)

```
    """
    X, K, PaddedX, PackedX, PackedK = depthwise_conv_pack(
        c, nh, nw, kh, kw, ph, pw, tc)
    # reduction axes
    rkh = te.reduce_axis((0, kh), name='rkh')
    rkw = te.reduce_axis((0, kw), name='rkw')
    # output height and weights
    oh = d2ltvm.conv_out_size(nh, kh, ph, sh)
    ow = d2ltvm.conv_out_size(nw, kw, pw, sw)
    # compute Y in the packed layout
    PackedY = te.compute(
        (c//tc, oh, ow, tc),
        lambda c_out, x, y, c_in: te.sum(
            (PackedX[c_out, x*sh+rkh, y*sw+rkw, c_in] *
             PackedK[c_out, 0, rkh, rkw, 0, c_in]),
            axis=[rkh, rkw]), name='PackedY')

    # Unpack the result
    Y = te.compute((c, oh, ow),
                    lambda c, x, y: PackedY[c//tc, x, y, c%tc],
                    name='Y')
    return X, K, Y, PaddedX, PackedX, PackedK, PackedY
```

Let's quickly compile it using the default scheduling to compute the results.

```
c, n, k, p, s, tc = 32, 64, 3, 1, 1, 16

X, K, Y, _, _, _, _ = depthwise_conv(c, n, n, k, k, p, p, s, s, tc)
mod = tvm.build(te.create_schedule(Y.op), [X, K, Y])

data, weight, out = d2ltvm.get_conv_data(c, c, n, k, p, s, tvm.nd.array, conv_
↪type='depthwise')
mod(data, weight, out)
```

And then verify the result.

```
data, weight, bias, out_mx = d2ltvm.get_conv_data_mxnet(c, c, n, k, p, s,␣
↪conv_type='depthwise')
d2ltvm.depthwise_conv_mxnet(data, weight, bias, out_mx, k, p, s)
np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 4.9.3 Schedule

The optimization strategy here is almost identical to `cache_block` defined in Section 4.8. The main difference is in the channels, i.e. we don't need to reduce along the input channel dimension due to the compute nature of depthwise convolution.

The tiling sizes for channel and width are set to make sure that the working set of the inner loop which calculates the cached output fits in the cache.

```
# tiling sizes for channel and width
tc, tw = 16, 4
```

(continues on next page)

(continued from previous page)

```
def depthwise_cached_block(c, n, k, p, s):
    X, K, Y, PaddedX, PackedX, PackedK, PackedY = depthwise_conv(
        c, n, n, k, k, p, p, s, s, tc)
    sch = te.create_schedule(Y.op)

    CachedY = sch.cache_write(PackedY, 'global')

    c_out, h, w, c_in = sch[PackedY].op.axis
    w_out, w_in = sch[PackedY].split(w, factor=tw)
    sch[PackedY].reorder(c_out, h, w_out, w_in, c_in)
    c_out_h = sch[PackedY].fuse(c_out, h)
    sch[PackedY].parallel(c_out_h)
    sch[CachedY].compute_at(sch[PackedY], w_out)

    cc_out, ch, cw, cc_in = sch[CachedY].op.axis
    kh, kw = sch[CachedY].op.reduce_axis
    sch[CachedY].reorder(cc_out, ch, kh, kw, cw, cc_in)
    sch[CachedY].vectorize(cc_in)
    sch[CachedY].unroll(cw)

    # Schedule the padding by adding thread-level parallelism
    if PaddedX != X:
        sch[PaddedX].parallel(PaddedX.op.axis[0])
    # Optimize the packing of X and K
    sch[PackedX].parallel(sch[PackedX].fuse(*PackedX.op.axis[0:2]))
    sch[PackedX].unroll(PackedX.op.axis[-1])
    sch[PackedK].parallel(sch[PackedK].fuse(*PackedK.op.axis[0:2]))
    sch[PackedK].unroll(PackedK.op.axis[-1])
    # Optimize the unpacking of Y
    sch[Y].parallel(sch[Y].fuse(*Y.op.axis[0:2]))
    sch[Y].unroll(Y.op.axis[-1])
    return sch, (X, K, Y)

# c, n, k, p, s were defined in the previous code block
sch, args = depthwise_cached_block(c, n, k, p, s)
# Uncomment the following line to see the long
# psuedo codes because of unrolling.
# tvm.lower(sch, args, simple_mode=True)
```

As the scheduling is vastly changed, let's do another round of sanity check.

```
mod = tvm.build(sch, args, target)
ctx = tvm.context(target, 0)
data, weight, out = d2ltvm.get_conv_data(
            c, c, n, k, p, s, lambda x: tvm.nd.array(x, ctx=ctx), conv_type=
↪'depthwise')
mod(data, weight, out)

data, weight, bias, out_mx = d2ltvm.get_conv_data_mxnet(c, c, n, k, p, s,␣
↪conv_type='depthwise')
d2ltvm.depthwise_conv_mxnet(data, weight, bias, out_mx, k, p, s)
np.testing.assert_allclose(out_mx[0].asnumpy(), out.asnumpy(), atol=1e-5)
```

### 4.9.4 Benchmark

Finally, let's benchmark the results against MXNet.

The following benchmarking method is very similar to `bench_conv_tvm` defined in Section 4.7, with two differences:

1. The signature of the convolution functions (depthwise convolution only takes one channel input).
2. The way to compute the FLOPs of computation (the input channel dimension of depthwise convolution is 1).

We don't unify the benchmarking of depthwise convolution into the `bench_conv_tvm` method in order to reduce the possible confusion it may cause.

```
# Save to the d2ltvm package.
def bench_depthwise_conv_tvm(func, sizes, target):
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(x, k, y).mean * nrepeats
    gflops, times = [], []
    for (c, n, k) in sizes:
        args = c, n, k, (k-1)//2, 1 # c, n, k, p, s
        s, (X, K, Y) = func(*args)
        mod = tvm.build(s, [X, K, Y], target)
        ctx = tvm.context(target, 0)
        x, k, y = d2ltvm.get_conv_data(
            args[0], *args, lambda x: tvm.nd.array(x, ctx=ctx), conv_type=
↪'depthwise')
        times.append(d2ltvm.bench_workload(workload))
        gflops.append(d2ltvm.conv_gflop(1, *args))
    return np.array(gflops) / np.array(times)
```

Similarly, the timing methods for depthwise convolution in MXNet are largely duplicated from the corresponding methods defined in Section 4.7.

```
# Save to the d2ltvm package.
def depthwise_conv_timer_mxnet(c, n, k, ctx):
    """Benchmark convolution in MXNet

    c : input, output channels
    n : input width and height
    k : kernel width and height
    """
    timer = timeit.Timer(
        setup='import d2ltvm\n'
        'import mxnet as mx\n'
        'c, n, k, p, s = %d, %d, %d, %d, 1\n'
        'data, weight, bias, out = d2ltvm.get_conv_data_mxnet(\n'
        '    c, c, n, k, p, s, "%s", "%s")'%(c, n, k, (k-1)//2, ctx,
↪'depthwise'),
        stmt='d2ltvm.depthwise_conv_mxnet(data, weight, bias, out, k, p, s);'
        'out.wait_to_read()')
    return timer.timeit

# Save to the d2ltvm package.
```

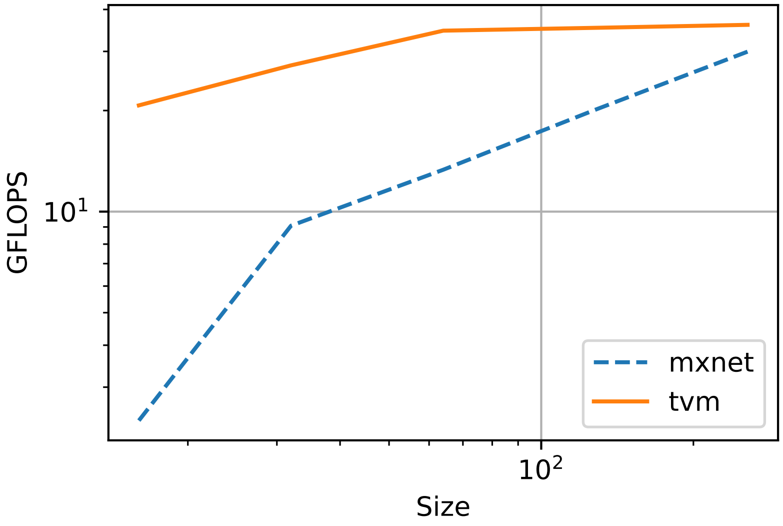
(continues on next page)

(continued from previous page)

```
def bench_depthwise_conv_mxnet(sizes, ctx='cpu'):
    """Return the GFLOPS of MXNet convolution"""
    return [d2ltvm.conv_gflop(1, c, n, k, (k-1)//2, 1) /
            d2ltvm.bench_workload(depthwise_conv_timer_mxnet(c, n, k, ctx))
            for c, n, k in sizes]
```

Now, let's benchmark against our MXNet baseline. We see that our depthwise convolution performance consistently outperform MXNet. As depthwise convolution is a memory-bound operator, we see that the performance saturates after channel size gets to 128.

```
channels = 2**np.arange(4, 9)
sizes = [(int(c), 64, 3) for c in channels] # a list of (c, n, k)
tvm_gflops = bench_depthwise_conv_tvm(depthwise_cached_block, sizes, target)
mxnet_gflops = bench_depthwise_conv_mxnet(sizes)
d2ltvm.plot_gflops(channels, [mxnet_gflops, tvm_gflops], ['mxnet', 'tvm'])
```



## 4.9.5 Summary

- We can get good performance out of depthwise convolution by following the same rules of optimizing 2-D convolution.

### 4.9.6 Exercises

- Try different tiling sizes.

## 4.10 Pooling

In this section, we will optimize the vector add defined in Section 3.5 on CPU.

```
%matplotlib inline
import d2ltvm
import inspect
from IPython import display
import numpy as np
from matplotlib import pyplot as plt
import timeit
import tvm
from tvm import te

target = 'llvm -mcpu=skylake-avx512'
```

### 4.10.1 Max Pooling

#### Schedule

By now, you should be familiar with the basic optimization tricks that we can do on CPU. Let's print out the IR of `max pooling` again to observe.

```
# channel, input height and width, kernel height and width
size = (64, 64, 3)

def default_max(size):
    c, n, k = size[:]
    X, Y, PaddedX = d2ltvm.pool('max', c, n, n, k, k, 1, 1, 1, 1)
    sch = te.create_schedule(Y.op)
    return sch, (X, Y)

sch, args = default_max(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 278784]
produce PaddedX {
  for (i0, 0, 64) {
    for (i1, 0, 66) {
      for (i2, 0, 66) {
        PaddedX[(((i0*4356) + (i1*66)) + i2)] = tvm_if_then_else(((((i1 < 1)␣
↪|| (65 <= i1)) || (i2 < 1)) || (65 <= i2)), -3.40282e+38f, X[((((i0*4096) +␣
↪(i1*64)) + i2) - 65)])
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
produce PoolMax {
  for (c, 0, 64) {
    for (h, 0, 64) {
      for (w, 0, 64) {
        PoolMax[(((c*4096) + (h*64)) + w)] = -3.40282e+38f
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            PoolMax[(((c*4096) + (h*64)) + w)] = max(PoolMax[(((c*4096) +␣
↪(h*64)) + w)], PaddedX[(((((c*4356) + (h*66)) + (rkh*66)) + w) + rkw)])
          }
        }
      }
    }
  }
}
```

You may have already figured out we can do some vectorization and parallelization for the outer loops. As `pooling` is a memory-bound operator, we don't have much to do for optimizing the cache.

However, there is one more thing that we can optimize. Note that the compute of `PoolMax` takes `PaddedX` as the input, which is the output of last compute. We can compute `PaddedX` inline in the stage of `PoolMax` by `te.schedule.AutoInlineInjective(sch)`. Doing this would prevent data from writing to and then reading from `PaddedX`. The simple scheduling is as follows.

```
def optimized_max(size):
    sch, (X, Y) = default_max(size)
    te.schedule.AutoInlineInjective(sch)
    c, h, w = Y.op.axis[0:3]
    fused = sch[Y].fuse(c, h)
    sch[Y].parallel(fused)
    sch[Y].vectorize(w)
    return sch, (X, Y)

sch, args = optimized_max(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce PoolMax {
  parallel (c.h.fused, 0, 4096) {
    PoolMax[ramp((c.h.fused*64), 1, 64)] = x64(-3.40282e+38f)
    for (rkh, 0, 3) {
      for (rkw, 0, 3) {
        for (w.s, 0, 64) {
          PoolMax[((c.h.fused*64) + w.s)] = max(PoolMax[((c.h.fused*64) + w.
↪s)], tvm_if_then_else((((((rkh + floormod(c.h.fused, 64)) < 1) || (65 <=␣
↪(rkh + floormod(c.h.fused, 64)))) || ((w.s + rkw) < 1)) || (65 <= (w.s +␣
↪rkw))), -3.40282e+38f, X[(((((c.h.fused*64) + (rkh*64)) + w.s) + rkw) -␣
↪65)]))
        }
      }
    }
  }
}
```

The optimized IR does both the parallelization and vectorization.

### Benchmarking

We now benchmark the default and optimized scheduling of `max pooling`. In order to do this, we first define the benchmarking method of pooling. It is analogous to the benchmarking methods we defined in for other operators before. The main difference is that as `pooling` performs little computation, we use execution time instead of FLOPs to benchmark the results.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]

# Save to the d2ltvm package.
def bench_pooling_tvm(func, sizes, target):
    """Benchmark pooling in TVM

    func : the scheduling method
    sizes : the data size list, each of which is a (channel, input_hw, kernel_
↪hw) triplet
    target : the TVM target, e.g. llvm or cuda
    """
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(data, out_max).mean * nrepeats
    times = []
    for size in sizes:
        sch, args = func(size)
        mod = tvm.build(sch, args, target)
        ctx = tvm.context(target, 0)
        data, _, out_max = d2ltvm.get_conv_data(size[0], size[0], size[1],␣
↪size[2], 1, 1,
                                                lambda x: tvm.nd.array(x,␣
↪ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
    return np.array(times)

default_max_times = bench_pooling_tvm(default_max, sizes, target)
optimized_max_times = bench_pooling_tvm(optimized_max, sizes, target)
```

Then we define the benchmarking method to perform pooling in MXNet. Like the TVM pooling benchmarking method, we collect execution time instead of FLOPs.

```
# Save to the d2ltvm package.
def pooling_timer_mxnet(pool_type, c, n, k, ctx):
    """Benchmark pooling in MXNet

    c : channels
    n : input width and height
    k : kernel width and height
    """
    timer = timeit.Timer(
        setup='import d2ltvm\n'
        'import mxnet as mx\n'
```

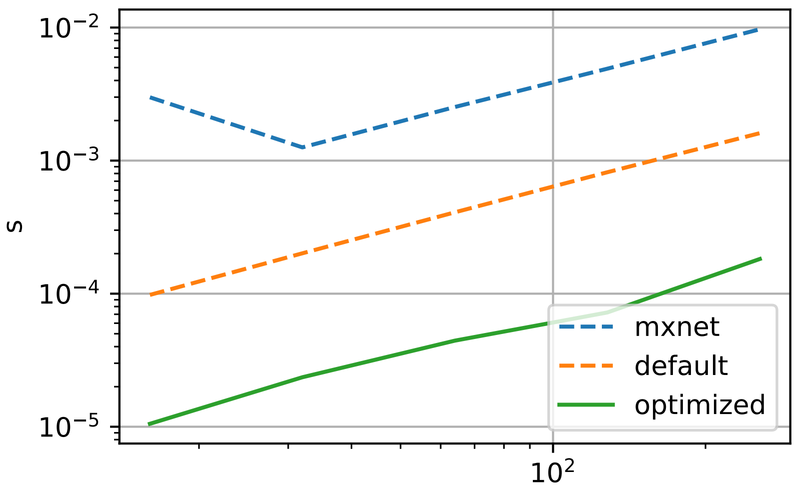
(continues on next page)

(continued from previous page)

```
        'c, n, k, p, s = %d, %d, %d, 1, 1\n'
        'data, out = d2ltvm.get_pool_data_mxnet(\n'
        '    c, n, k, p, s, "%s")'%(c, n, k, ctx),
        stmt='d2ltvm.pool_mxnet("%s", data, out, k, p, s);'
        'out.wait_to_read()'%(pool_type))
    return timer.timeit

# Save to the d2ltvm package.
def bench_pooling_mxnet(pool_type, sizes, ctx='cpu'):
    """Return the execution times of MXNet pooling"""
    return [d2ltvm.bench_workload(pooling_timer_mxnet(pool_type, c, n, k,␣
↪ctx))
            for c, n, k in sizes]

mxnet_max_times = bench_pooling_mxnet('max', sizes)
```

Then we plot the results to compare. Note that we are plotting the execution time, so lower is better.

```
times = [mxnet_max_times, default_max_times, optimized_max_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'default', 'optimized'], fmts=['--']*(len(times)-
↪1)+['-'])
```



From the diagram we see that even using the default schedule, TVM outperforms MXNet significantly. This is because the pooling runs very fast, making the function call overhead of MXNet discussed in Section 4.2 profound.

## 4.10.2 Avg Pooling

### Schedule

Now let's move to `avg pooling`. Again, we print out the IR of `avg pooling` to observe.

```python
def default_avg(size):
    c, n, k = size[:]
    X, Y, PaddedX = d2ltvm.pool('avg', c, n, n, k, k, 1, 1, 1, 1)
    sch = te.create_schedule(Y.op)
    return sch, (X, Y)

sch, args = default_avg(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
// attr [PaddedX] storage_scope = "global"
allocate PaddedX[float32 * 278784]
// attr [PoolSum] storage_scope = "global"
allocate PoolSum[float32 * 262144]
produce PaddedX {
  for (i0, 0, 64) {
    for (i1, 0, 66) {
      for (i2, 0, 66) {
        PaddedX[(((i0*4356) + (i1*66)) + i2)] = tvm_if_then_else(((((i1 < 1) || (65 <= i1)) || (i2 < 1)) || (65 <= i2)), 0f, X[((((i0*4096) + (i1*64)) + i2) - 65)])
      }
    }
  }
}
produce PoolSum {
  for (c, 0, 64) {
    for (h, 0, 64) {
      for (w, 0, 64) {
        PoolSum[(((c*4096) + (h*64)) + w)] = 0f
        for (rkh, 0, 3) {
          for (rkw, 0, 3) {
            PoolSum[(((c*4096) + (h*64)) + w)] = (PoolSum[(((c*4096) + (h*64)) + w)] + PaddedX[(((((c*4356) + (h*66)) + (rkh*66)) + w) + rkw)])
          }
        }
      }
    }
  }
}
produce PoolAvg {
  for (c, 0, 64) {
    for (h, 0, 64) {
      for (w, 0, 64) {
        PoolAvg[(((c*4096) + (h*64)) + w)] = (PoolSum[(((c*4096) + (h*64)) + w)]*0.111111f)
      }
    }
  }
}
```

Similarly, we can do parallelization, vectorization, and merging stages of `PaddedX` and `PoolSum` using `te.schedule.AutoInlineInjective(sch)`. In addition, we want to compute an element of `PoolAvg` right after getting the corresponding element of `PoolSum`, so that we can immediately use `PoolSum` after it is produced. This can be realized by the `compute_at()` scheduling premitive. As a side note, there is another compiling optimization already done in the default scheduling, which is converting the division operation into multiplying the reciprocal of the divisor. In this case, it converts dividing 9 into multiplying 0.111111. The reason is that the division instruction in the processor is much more expensive than the multiplication instruction.

```python
def schedule_avg(size):
    sch, (X, Y) = default_avg(size)
    te.schedule.AutoInlineInjective(sch)
    c, h, w = Y.op.axis[0:3]
    fused = sch[Y].fuse(c, h)
    sch[Y].parallel(fused)
    sch[Y].vectorize(w)
    PoolSum = Y.op.input_tensors[0]
    sch[PoolSum].compute_at(sch[Y], Y.op.axis[2])
    return sch, (X, Y)

sch, args = schedule_avg(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce PoolAvg {
  parallel (c.h.fused, 0, 4096) {
    // attr [PoolSum] storage_scope = "global"
    allocate PoolSum[float32 * 64]
    produce PoolSum {
      PoolSum[ramp(0, 1, 64)] = x64(0f)
      for (rkh, 0, 3) {
        for (rkw, 0, 3) {
          for (w.s, 0, 64) {
            PoolSum[w.s] = (PoolSum[w.s] + tvm_if_then_else((((((rkh +␣
↪floormod(c.h.fused, 64)) < 1) || (65 <= (rkh + floormod(c.h.fused, 64))))␣
↪|| ((w.s + rkw) < 1)) || (65 <= (w.s + rkw))), 0f, X[(((((c.h.fused*64) +␣
↪(rkh*64)) + w.s) + rkw) - 65)]))
          }
        }
      }
    }
    PoolAvg[ramp((c.h.fused*64), 1, 64)] = (PoolSum[ramp(0, 1, 64)]*x64(0.
↪111111f))
  }
}
```

From the optimized IR we can see that the stage of `PoolSum` is merged into the stage of `PoolAvg`, which produces better data locality.

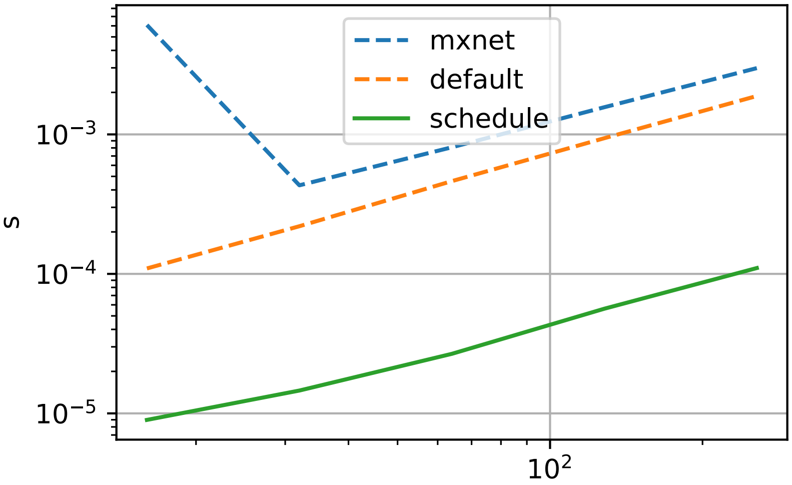
### Benchmarking

We now benchmark the default and optimized scheduling of `avg pooling`, as well as the MXNet performance.

```
default_avg_times = bench_pooling_tvm(default_avg, sizes, target)
schedule_avg_times = bench_pooling_tvm(schedule_avg, sizes, target)

mxnet_avg_times = bench_pooling_mxnet('avg', sizes)
```

Then we plot the results to compare. Again, as we are plotting the execution time, lower is better.

```
times = [mxnet_avg_times, default_avg_times, schedule_avg_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'default', 'schedule'], fmts=['--']*(len(times)-
↪1)+['-'])
```



From the diagram we see similar behavior as the `max pooling` operator.

Overall, we see that `pooling` only takes a negeligible amount of time. So it is normally not a focus in optimizing deep learning workloads.

Another interesting optimization idea of `pooling` is to fuse it to other operators like `convolution`. Theoretically, this can result in better data locality. However, this is not easy to implement as the reduction operation in `pooling` introduces dependencies, i.e. a specific set of data should be ready before the `pooling` can be executed. Handling this dependency in other operators is tedious. In practice, only on specialized accelerators where all memory movement is managed explicitly, the compiler would perform operator fusion between `pooling` and others.

### 4.10.3 Summary

- Pooling can be optimized on CPUs via parallelization and vectorization.
- The padding stage can be computed inline in the latter stage using AutoInlineInjective.
- We can use compute_at scheduling premitive to merge stages for better data locality.

## 4.11 Batch Normalization

This section talks about scheduling the batch normalization computation defined in Section 3.6 on CPU.

### 4.11.1 Setup

```
%matplotlib inline
import d2ltvm
import inspect
from IPython import display
import numpy as np
from matplotlib import pyplot as plt
import timeit
import tvm
from tvm import te
import topi

target = 'llvm -mcpu=skylake-avx512'
```

### 4.11.2 Schedule

We first review the default scheduling of batch normalization and its IR, as shown in Section 3.6.

```
# a tuple of channel and input height/width
size = (32, 28)

def default_bn(size):
    c, n = size[:]
    X, Mean, Var, Gamma, Beta, Y = d2ltvm.batch_norm(c, n)
    sch = te.create_schedule(Y.op)
    return sch, (X, Mean, Var, Gamma, Beta, Y)

sch, args = default_bn(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
// attr [T_subtract] storage_scope = "global"
allocate T_subtract[float32 * 25088]
// attr [T_add] storage_scope = "global"
allocate T_add[float32 * 32]
produce T_subtract {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
```

(continues on next page)

(continued from previous page)

```
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (X[(((ax0*784) +␣
↪(ax1*28)) + ax2)] - Mean[ax0])
      }
    }
  }
}
produce T_add {
  for (ax0, 0, 32) {
    T_add[ax0] = (Var[ax0] + 1e-05f)
  }
}
produce compute {
  for (i0, 0, 32) {
    T_add[i0] = sqrt(T_add[i0])
  }
}
produce T_divide {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784)␣
↪+ (ax1*28)) + ax2)]/T_add[ax0])
      }
    }
  }
}
produce T_multiply {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_subtract[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784)␣
↪+ (ax1*28)) + ax2)]*Gamma[ax0])
      }
    }
  }
}
produce T_add {
  for (ax0, 0, 32) {
    for (ax1, 0, 28) {
      for (ax2, 0, 28) {
        T_add[(((ax0*784) + (ax1*28)) + ax2)] = (T_subtract[(((ax0*784) +␣
↪(ax1*28)) + ax2)] + Beta[ax0])
      }
    }
  }
}
```

We can easily tell that the multiple stages of the computation can be fused injectively together. Like we have done in Section 4.10, `te.schedule.AutoInlineInjective(sch)` is used for it. Essentially, `AutoInlineInjective` traverses the stages of the schedule and fuses all stages that are fusable injectively. The resulting stage is a three-level nested loop of channel, input height, and input width.

In addition, we can fuse the first two axes into one and parallelize it in multi-thread. We can also vectorize the innermost axis. The optimized scheduling scheme looks a lot like the scheduling of `max_pooling` in

Section 4.10.

```
def optimized_bn(size):
    sch, (X, Mean, Var, Gamma, Beta, Y) = default_bn(size)
    te.schedule.AutoInlineInjective(sch)
    c, h, w = Y.op.axis[0:3]
    fused = sch[Y].fuse(c, h)
    sch[Y].parallel(fused)
    sch[Y].vectorize(w)
    return sch, (X, Mean, Var, Gamma, Beta, Y)

sch, args = optimized_bn(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce T_add {
  parallel (ax0.ax1.fused, 0, 896) {
    T_add[ramp((ax0.ax1.fused*28), 1, 28)] = ((((X[ramp((ax0.ax1.fused*28), 1,
↪ 28)] - x28(Mean[floordiv(ax0.ax1.fused, 28)]))/x28(sqrt((Var[floordiv(ax0.
↪ax1.fused, 28)] + 1e-05f))))*x28(Gamma[floordiv(ax0.ax1.fused, 28)])) +␣
↪x28(Beta[floordiv(ax0.ax1.fused, 28)]))
  }
}
```

### 4.11.3 Benchmarking

First, we define the method to benchmark batch normalization of TVM as usual.

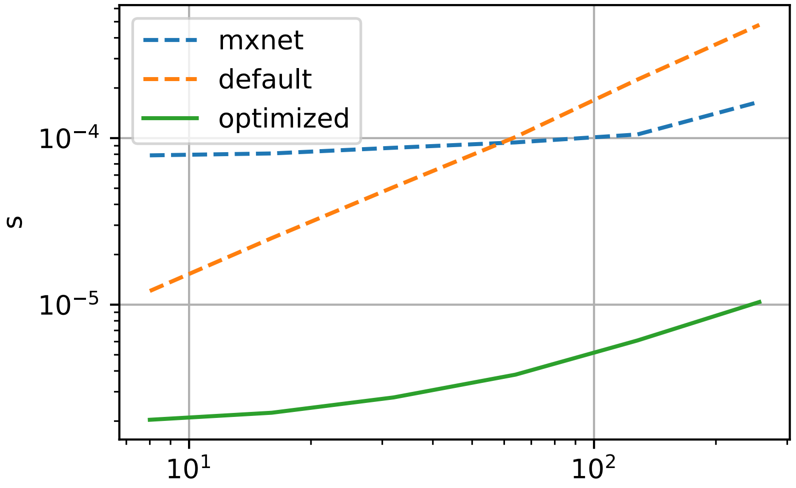
```
# Save to the d2ltvm package.
def bench_bn_tvm(func, sizes, target):
    """Benchmark batch normalization in TVM

    func : the scheduling method
    sizes : the data size list, each of which is a (channel, input_hw) tuple
    target : the TVM target, e.g. llvm or cuda
    """
    def workload(nrepeats):
        timer = mod.time_evaluator(mod.entry_name, ctx=ctx, number=nrepeats)
        return timer(data, mean, var, gamma, beta, out).mean * nrepeats
    times = []
    for size in sizes:
        sch, args = func(size)
        mod = tvm.build(sch, args, target)
        ctx = tvm.context(target, 0)
        data, mean, var, gamma, beta, out = d2ltvm.get_bn_data(size[0],␣
↪size[1],
                                                               lambda x: tvm.
↪nd.array(x, ctx=ctx))
        times.append(d2ltvm.bench_workload(workload))
    return np.array(times)
```

Then, we use MXNet as the baseline, and define the method to benchmark its performance.

```python
# Save to the d2ltvm package.
def bn_timer_mxnet(c, n, ctx):
    """Benchmark batch normalization in MXNet

    c : channels
    n : input width and height
    ctx : compute ctx, e.g., cpu or gpu
    """
    timer = timeit.Timer(
        setup='import d2ltvm\n'
        'import mxnet as mx\n'
        'c, n = %d, %d\n'
        'data, mean, var, gamma, beta, out = d2ltvm.get_bn_data_mxnet(\n'
        '    c, n, "%s")'%(c, n, ctx),
        stmt='d2ltvm.batch_norm_mxnet(data, mean, var, gamma, beta, out);'
        'out.wait_to_read()')
    return timer.timeit

# Save to the d2ltvm package.
def bench_bn_mxnet(sizes, ctx='cpu'):
    """Return the execution times of MXNet batch norm"""
    return [d2ltvm.bench_workload(bn_timer_mxnet(c, n, ctx))
            for c, n in sizes]
```

Finally, we define a number of different channel numbers (with the fixed input size $28 \times 28$) to plot the benchmarking results of our baseline, default scheduling and optimized scheduling.

```python
channels = 2**np.arange(3, 9, 1)
# a list of (c, n)
sizes = [(int(c), 28) for c in channels]

mxnet_times = bench_bn_mxnet(sizes)
default_times = bench_bn_tvm(default_bn, sizes, target)
optimized_times = bench_bn_tvm(optimized_bn, sizes, target)

times = [mxnet_times, default_times, optimized_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'default', 'optimized'], fmts=['--']*(len(times)-
↪1)+['-'])
```



From the diagram we can see multiple things. First, for this kind of small operators, MXNet's execution time is dominated by the function invoking overhead (Section 4.2). Second, TVM's invoking overhead is small, making even the default schedule outperforms the MXNet baseline. Third, after fusing stages and doing proper parallelization and vectorization, we can have a much better performance for batch normalization.

### 4.11.4 Summary

- Like Pooling, the optimization of batch normalization on CPU is all about stage fusion and parallelization/vectorization.

# 5 | Operator Optimizations on GPUs

The chapter talks about the operator optimization on Nvidia GPUs. Basically, we follow the some logic of last chapter, starting from introducing the architecture of GPUs, followed by the optimization of some typical operators.

## 5.1 GPU Architecture

High-end GPUs often provide a significantly better performance over high-end CPUs. Although the terminologies and programming paradigms are different between GPUs and CPUs, their architectures are similar to each other, with GPU having a wider SIMD width and more cores. In this section, we will brief review the GPU architecture in comparison to the CPU architecture presented in Section 4.1.

(FIXME, changed from V100 to T4 in CI…, also changed cpu…)

The system we are using has a Tesla T4[43] GPU, which is based on Turing architecture. Tesla T4 is a GPU card based on the Turing architecture and targeted at deep learning model inference acceleration.

```
!nvidia-smi -q -i 0 | grep "Product Name"
```

```
    Product Name                          : Tesla T4
```

[43] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

### 5.1.1 Streaming Multiprocessor

A streaming multiprocessor (SM) roughly equals a CPU core. The SM used by T4 is illustrated in Fig. 5.1.1.

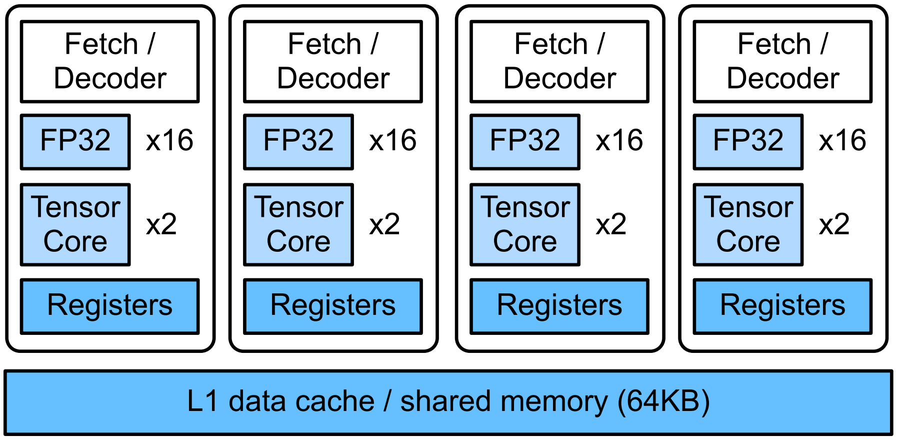


Fig. 5.1.1: A streaming multiprocessor in Tesla T4

As can be seen, an SM is partitioned into 4 processing blocks. In each block, there are 16 arithmetic units (AU) for processing float32 numbers, which are also called FP32 CUDA cores. In total, an SM has 64 FP32 AUs, which are able to execute 64 float32 operators (e.g. FMA) in each time. Besides the register files and the instruction loader/decoders, an SM has 8 tensor cores. Each tensor core is able to execute a $4 \times 4$ float16 (or int8/int4) matrix product in each time. So each one, we can call it FP16 AU, counts for $2 \times 4^3 = 128$ operators per clock. It is worth noting that in this chapter we won't use the tensor core. We will talk about utilizing it in the later chapter.

Another difference is that the SM only has an L1 cache, which is similar to CPU's L1 cache. However, we can use this storage as a shared memory for all threads running on the SM. We know that the cache is controlled by both hardware and operating system, while we can explicitly allocate and reclaim space on the shared memory, which gives us more flexibility to do performance optimization.

### 5.1.2 GPU Architecture

Our Tesla T4 card contains 40 SMs with a 6MB L2 cache shared by all SMs. It also ships with 16GB high-bandwidth memory (GDDR6) that is connected to the processor. The overall architecture is illustrated in Fig. 5.1.2.

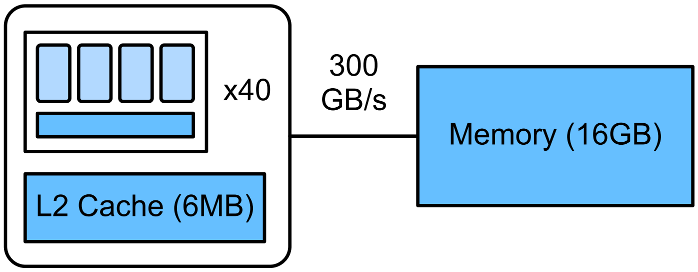


Fig. 5.1.2: The Tesla T4 Architecture

More broadly, we compare the specification difference between the CPU and GPUs used in this book in Fig.

5.1.2, where GPUs includes Tesla P100[44] (used in Colab), Tesla V100[45] (equipped in Amazon EC2 P3 instance), and Tesla T4[46] (equipped in Amazon EC2 G4 instance).

| Hardware | Intel E5-2686 v4 | Tesla P100 | Tesla V100 | Tesla T4 |
|---|---|---|---|---|
| Clock rate (GHz) | **3** | 1.48 | 1.53 | 1.59 |
| # cores | 16 | 56 | **80** | 40 |
| # FP64 AUs per core | 4 | **32** | **32** | x |
| # FP32 AUs per core | 8 | **64** | **64** | **64** |
| # FP16 AUs per core | x | x* | **8** | **8** |
| cache per core (KB) | **320** | 64 | 128 | 64 |
| shared cache (MB) | **45** | 4 | 6 | 6 |
| Memory (GB) | **240** | 16 | 16 | 16 |
| Max memory bandwidth (GB/sec) | 72 | 732 | **900** | 300 |
| FP64 TFLOPS | 0.38 | 4.7 | **7.8** | x |
| FP32 TFLOPS | 0.77 | 9.3 | **15.7** | 8.1 |
| FP16 TFLOPS | x | 18.7 | **125.3** | 65 |

Table: Compare the commonly used CPUs and GPUs, x means not supported. *: Tesla P100 processes FP16 using FP32 CUDA cores.

### 5.1.3 Summary

- GPUs have conceptually similar architecture as CPUs, but are much faster.

## 5.2 Vector Add

In this section, we will optimize the vector add defined in Section 1.2 on GPUs. As we have seen in Section 5.1, the overall architecture of GPUs is not significantly different from CPUs, but GPUs often have more cores and wider SIMD units. If you are new to GPU programming, we recommend you to first read how we optimized the vector add on CPUs in Section 4.3 for more contexts.

```
%matplotlib inline
import tvm
from tvm import te
import numpy as np
import d2ltvm
import mxnet as mx
import timeit
```

Since NumPy only runs on CPUs, we use MXNet as our baseline. The following code block plots the performance baseline as a function of vector length.

```
sizes = 2**np.arange(10, 30, 3)
```

(continues on next page)

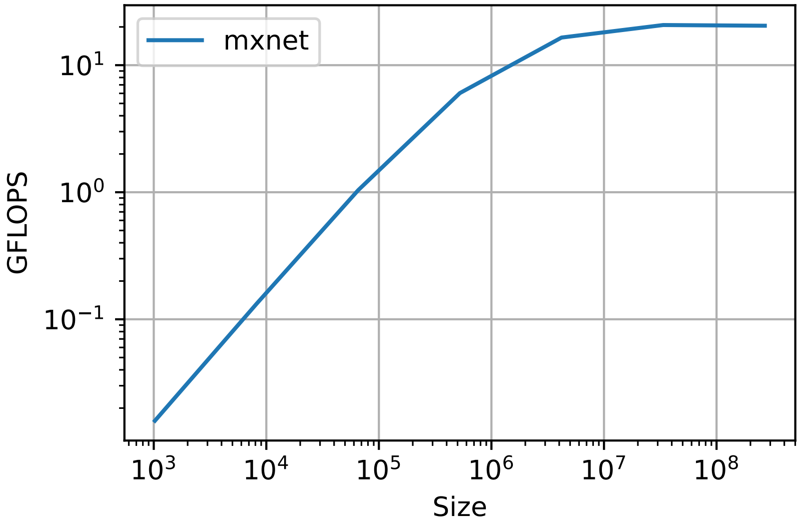
[44] https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[45] https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[46] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

(continued from previous page)

```python
mx_add = lambda n: timeit.Timer(
    setup='import mxnet as mx\n'
    'import d2ltvm\n'
    'mx_arr = lambda x: mx.nd.array(x, ctx=mx.gpu())\n'
    'a, b, c = d2ltvm.get_abc(%d, mx_arr)' % n,
    stmt='mx.nd.elemwise_add(a, b, out=c); c.wait_to_read()')

exe_times = [d2ltvm.bench_workload(mx_add(n).timeit) for n in sizes]
mx_gflops = sizes / 1e9 / np.array(exe_times)
d2ltvm.plot_gflops(sizes, [mx_gflops], ['mxnet'])
```



### 5.2.1 CUDA Programming

CUDA is a programming interface proposed by Nvidia for high-performance parallel programming on GPUs. TVM provides an abstraction level above CUDA so that we don't bother to know the details about CUDA programming. We still, however, need to understand several concepts that are specific to CUDA.

The first one is *CUDA kernel*, or simply *kernel*. A kernel is a small program or a function. In TVM, it's often the Lambda expression to compute elements of the output tensor, e.g. `lambda i: A[i] + B[i]` in vector add.

The second one is *CUDA thread*, or simply *thread*. A thread is an abstracted entity that represents the execution of a kernel. Note that it's a programming abstract, not the aforementioned logic or hardware thread which actually runs. So if the vector length is 1 million, we will have 1 million threads, but apparently we cannot run all of them simultaneously.

All these threads are grouped into blocks. If the block size is 100, 1 million threads would be partitioned into 10,000 thread blocks. The maximal number of threads in a block is 512 before CUDA 10, and 1024 after. Each thread has a unique ID. In the 1-D indexing case, the `i`-th block is indexed by `blockIdx.x`. The number of threads in each block is `blockDim.x` and the `i`-th thread within a block is indexed by `threadIdx.x`. Therefore, the overall index of a thread can be calculated as

$$i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \tag{5.2.1}$$

It can be extended to 2-D and 3-D indexing schemes for both block and thread, where we just simply add `.y` and `.z` fields accordingly. Note that `x` is the innermost dimension while `z` is the outermost, and block is outer than thread. Multi-dimensional blocks make a grid (`gridDim.x` describes the number of blocks along the x axis), just as multi-dimensional threads make a block. So the overall index of a thread in 1-D block and 3-D thread case can be calculated as

$$\begin{aligned} i =& \text{blockIdx.x} \times \text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z} \\ &+ \text{threadIdx.z} \times \text{blockDim.x} \times \text{blockDim.y} \\ &+ \text{threadIdx.y} \times \text{blockDim.x} \\ &+ \text{threadIdx.x} \end{aligned} \tag{5.2.2}$$

During execution, all threads in a single block will be executed in the same core, or streaming multiprocessor (SM). We could assume that they will be executed simultaneously so we can synchronize these threads in the middle of a kernel. Different blocks may run on different cores.

In analogy to CPUs, we parallelize thread blocks on a GPU is like we parallelize over CPU threads, while a thread block runs the workload that could be vectorized. Remember that in the last part of Section 4.3 we split the for-loop for vector add for parallelization and vectorization separately, we can do it similarly in GPUs programming.

```
nt = 64  # number of threads in a block

def parallel(n):
    A, B, C = d2ltvm.vector_add(n)
    s = te.create_schedule(C.op)
    bx, tx = s[C].split(C.op.axis[0], factor=nt)
    s[C].bind(bx, te.thread_axis("blockIdx.x"))
    s[C].bind(tx, te.thread_axis("threadIdx.x"))
    return s, (A, B, C)

s, args = parallel(256)
tvm.lower(s, args, simple_mode=True)
```

```
produce c {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 4
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 64
  c[((blockIdx.x*64) + threadIdx.x)] = (a[((blockIdx.x*64) + threadIdx.x)] +␣
↪b[((blockIdx.x*64) + threadIdx.x)])
}
```

Compared to `vectorized` defined in Section 4.3, there are two major differences.

One is that we bind axes into block and thread indexing axes instead of call `parallel` and `vectorize`. The inner axis length is 64, specified by `nt`. Binding the `threadIdx.x` thread axis on it means that we will create 64 threads in a thread block. Similarly, binding the `blockIdx.x` thread axis on the outer axis leads to `n/nt` thread blocks.

The other one is that the pseudo codes only have the kernel, instead of the whole program with for-loops. The indexing is obtained by the 1-D indexing scheme we shown before.

If you have written CUDA codes before, you may want to check the generated CUDA program. Note that the target is set to be `cuda`, instead of `llvm` for CPUs, during compiling.

```
mod = tvm.build(s, args, 'cuda')
dev_mod = mod.imported_modules[0]
print(dev_mod.get_source())
```

```
extern "C" __global__ void default_function_kernel0(void** __restrict__ c,␣
↪void** __restrict__ a, void** __restrict__ b) {
  (( float*)c)[(((((int)blockIdx.x) * 64) + ((int)threadIdx.x)))] = (((␣
↪float*)a)[(((((int)blockIdx.x) * 64) + ((int)threadIdx.x)))] + ((␣
↪float*)b)[(((((int)blockIdx.x) * 64) + ((int)threadIdx.x)))]);
}
```

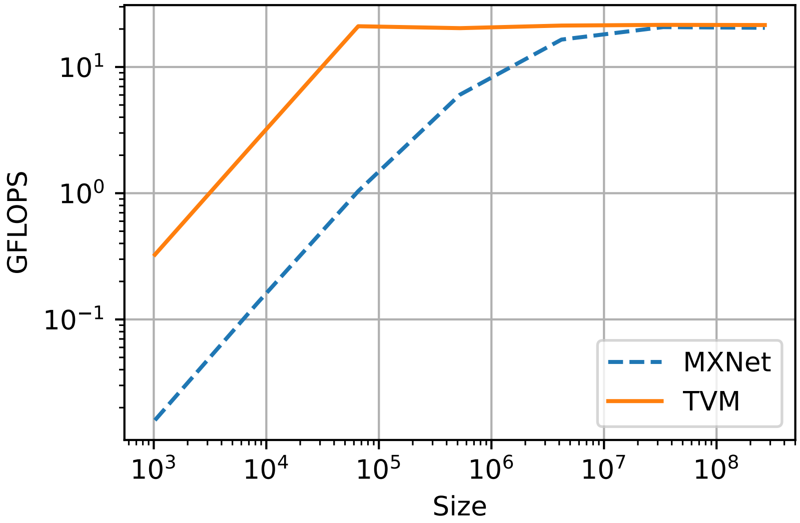
As mentioned in Section 5.1, GPUs have their own on-board memories. To execute an operation on a GPU, we need to copy the data from the main memory to its memory. For Nvidia GPUs, the context for the $i$-th GPU can be presented by either `tvm.context('cuda', i)` or simply `tvm.gpu(i)`. The following code block copies data to the first GPU of the system. Note the `gpu(0)` shown in the first line.

```
ctx = tvm.gpu(0)
tvm.nd.array(np.zeros((3,3)), ctx)
```

```
<tvm.nd.NDArray shape=(3, 3), gpu(0)>
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Remember that the `bench_vector_add_tvm` method already takes care of the context. Now let's benchmark the above schedule against MXNet.

```
tvm_gflops = d2ltvm.bench_vector_add_tvm(parallel, sizes, 'cuda')
d2ltvm.plot_gflops(sizes, [mx_gflops, tvm_gflops], ['MXNet', 'TVM'])
```



We can see that TVM is faster for small workloads. That is because TVM uses `cython` for the foreign function interface, which is significantly faster than the `ctypes` used in MXNet.

Note that if you are running the above code on GPUs with faster memory, such as V100, you may find TVM performs worse compared to MXNet for large vectors. You can improve it by increasing the number of threads `nt`.

### 5.2.2 Summary

- GPU introduces a thread-block abstraction for parallel programming. We can bind axes to threads or thread-blocks through `bind`.
- We need to allocate each streaming multiprocessor (SM) enough workloads for good performance.

## 5.3 Broadcast Add

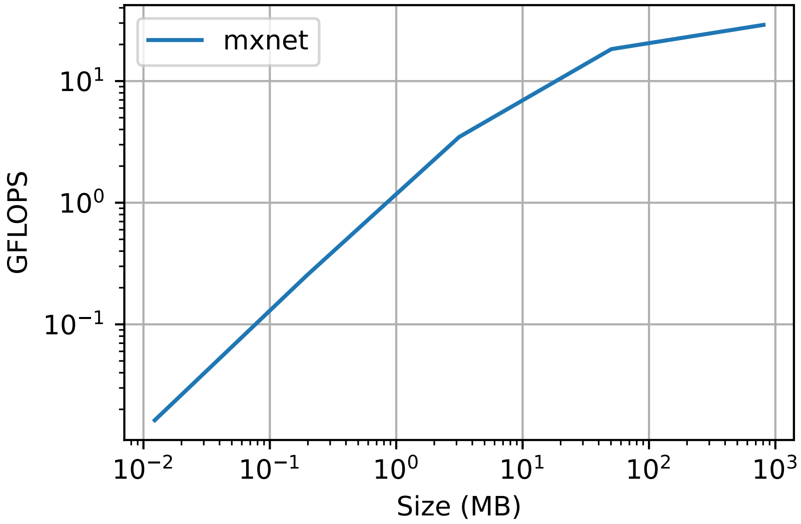
This section talks about scheduling the broadcast add computation defined in Section 3.1 on GPU. Similar to the CPU case, we can extend what we have done for vector add on GPU in Section 5.2 with some minor modification.

### 5.3.1 Setup

```
%matplotlib inline
import tvm
from tvm import te
import numpy as np
import d2ltvm
import mxnet as mx
import timeit
```

Like Section 5.2, we use MXNet as our baseline. The following code block plots the performance baseline as a function of consumed data size. The broadcast pattern is the same as Fig. 3.1.1 illustrates.

```
sizes = 2**np.arange(5, 15, 2)

mx_bcast_add = lambda s1, s2: timeit.Timer(
    setup='import mxnet as mx\n'
    'import d2ltvm\n'
    'mx_arr = lambda x: mx.nd.array(x, ctx=mx.gpu())\n'
    'a, b, c = d2ltvm.get_bcast_data(%s, %s, mx_arr)' % (s1, s2),
    stmt='mx.nd.broadcast_add(a, b, out=c); c.wait_to_read()')

exe_times = [d2ltvm.bench_workload(mx_bcast_add((n, 1), (n, n)).timeit) for n␣
↪in sizes]
mx_gflops = sizes * sizes / 1e9 / np.array(exe_times)
# data size in MB
x_axis_sizes = (sizes * sizes * 2 + sizes * sizes) * 4 / 1e6
d2ltvm.plot_gflops(x_axis_sizes, [mx_gflops], ['mxnet'], xlabel='Size (MB)')
```



### 5.3.2 Continuous scheduling

Like in Section 5.2, we will need to bind some axes to CUDA threads and blocks. For broadcast add depicted in Fig. 3.1.1, the loop pattern is straightforward, where the inner loop goes through the columns of a particular row of `B`, and the outer loop goes through all rows. One way to think about the thread binding could be to bind the CUDA threads along column axis `y` and blocks along row axis `x`. However, when the number of columns exceeds 1024, it will return an error as violating the maximal number of threads in a block. Therefore, we will need to restrict the number of threads as `nt`. There is no constraint on the number of blocks (`nb`), but generally, we want one thread to take care of multiple calculations if the processed data is large to reduce the overhead.

We first specify `nt` and `nb`.

```
nt = 64  # number of threads in a block
nb = 256 # number of blocks

with tvm.target.create('cuda'):
    assert nt <= tvm.target.Target.current(allow_none=False).max_num_threads,␣
 ↪\
        'the number of threads in a block exceed the hardware limit'
```

To facilitate the splitting and binding, we can first fuse the two loops of broadcast add together. Then we bind the threads accordingly. Note that we only further split the data to let one thread process multiple calculations when the data size is large enough, i.e. `need_further_split=True`.

```
def continuous_parallel(n):
    A, B, C = d2ltvm.broadcast_add((n,1), (n,n))
    total_size = n * n
    need_further_split = total_size > nb * nt
    s = te.create_schedule(C.op)
    x, y = C.op.axis
    fused = s[C].fuse(x, y)
    if need_further_split:
        bx, tx = s[C].split(fused, nparts=nb)
        tx, xi = s[C].split(tx, nparts=nt)
```

(continues on next page)

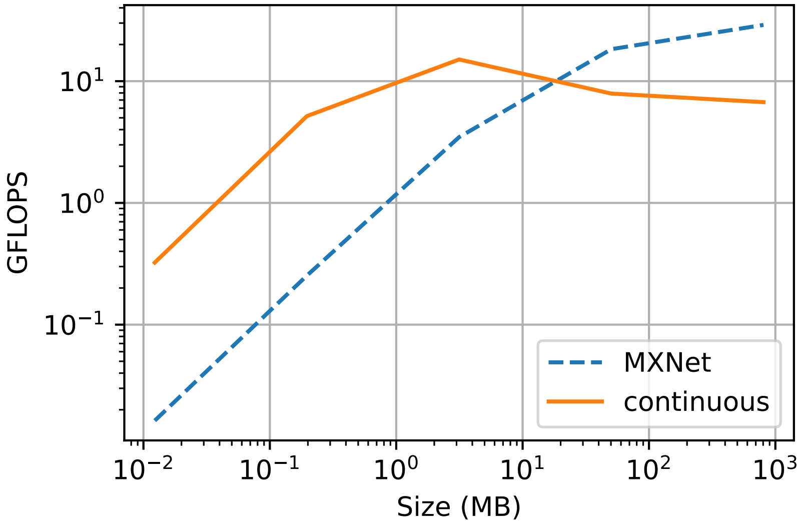
(continued from previous page)

```
        s[C].bind(bx, te.thread_axis("blockIdx.x"))
        s[C].bind(tx, te.thread_axis("threadIdx.x"))
    else:
        bx, tx = s[C].split(fused, factor=nt)
        s[C].bind(bx, te.thread_axis("blockIdx.x"))
        s[C].bind(tx, te.thread_axis("threadIdx.x"))
    return s, (A, B, C)

s, args = continuous_parallel(256)
tvm.lower(s, args, simple_mode=True)
```

```
produce C {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 256
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 64
  for (x.y.fused.inner.inner, 0, 4) {
    C[(((blockIdx.x*256) + (threadIdx.x*4)) + x.y.fused.inner.inner)] =␣
↪(A[blockIdx.x] + B[(((blockIdx.x*256) + (threadIdx.x*4)) + x.y.fused.inner.
↪inner)])
  }
}
```

The pseudo-code above looks reasonable. Each thread works on a number of calculations over a continuous data chunk. Now let's compile it to execute.

```
mod = tvm.build(s, args, 'cuda')

tvm_continuous_gflops = d2ltvm.bench_bcast_add_tvm(continuous_parallel, sizes,
↪ 'cuda')
d2ltvm.plot_gflops(x_axis_sizes, [mx_gflops, tvm_continuous_gflops],
                   ['MXNet', 'continuous'], xlabel='Size (MB)')
```



We see that the continuous scheduling outperforms MXNet only for small workloads due to lighter overhead. Its performance becomes inferior to MXNet while increasing the data size. This is because the continuous scheduling causes serious bank conflict, which we will discuss in Section 5.5 and solve it in a more systematic way.

### 5.3.3 Alternate scheduling

For now, let's try to avoid one thread processing a continuous data chunk. Instead, we make each thread operate on the scattered data. Fig. 5.3.1 depicts the difference between continuous and alternate schedulings.

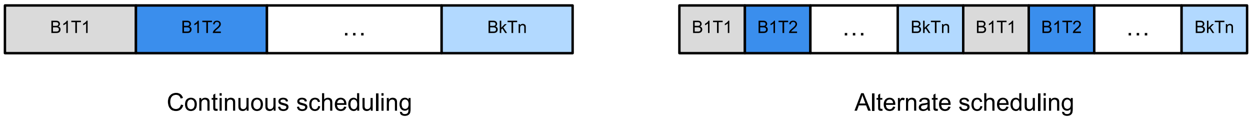


Fig. 5.3.1: Difference between continuous and alternate schedulings.

```
def alternate_parallel(n):
    A, B, C = d2ltvm.broadcast_add((n,1), (n,n))
    total_size = n * n
    need_further_split = total_size > nb * nt
    s = te.create_schedule(C.op)
    x, y = C.op.axis
    fused = s[C].fuse(x, y)
    if need_further_split:
        xo, xi = s[C].split(fused, factor=nb * nt)
        bx, tx = s[C].split(xi, factor=nt)
        # bring the outermost axis to the innermost
        # for alternate data access of a CUDA thread
        s[C].reorder(bx, tx, xo)
        s[C].bind(bx, te.thread_axis("blockIdx.x"))
        s[C].bind(tx, te.thread_axis("threadIdx.x"))
    else:
        bx, tx = s[C].split(fused, factor=nt)
        s[C].bind(bx, te.thread_axis("blockIdx.x"))
        s[C].bind(tx, te.thread_axis("threadIdx.x"))
    return s, (A, B, C)

s, args = alternate_parallel(256)
tvm.lower(s, args, simple_mode=True)
```

```
produce C {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 256
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 64
  for (x.y.fused.outer, 0, 4) {
    C[(((x.y.fused.outer*16384) + (blockIdx.x*64)) + threadIdx.x)] = (A[((x.y.
↪fused.outer*64) + floordiv(((blockIdx.x*64) + threadIdx.x), 256))] + B[(((x.
↪y.fused.outer*16384) + (blockIdx.x*64)) + threadIdx.x)])
  }
}
```

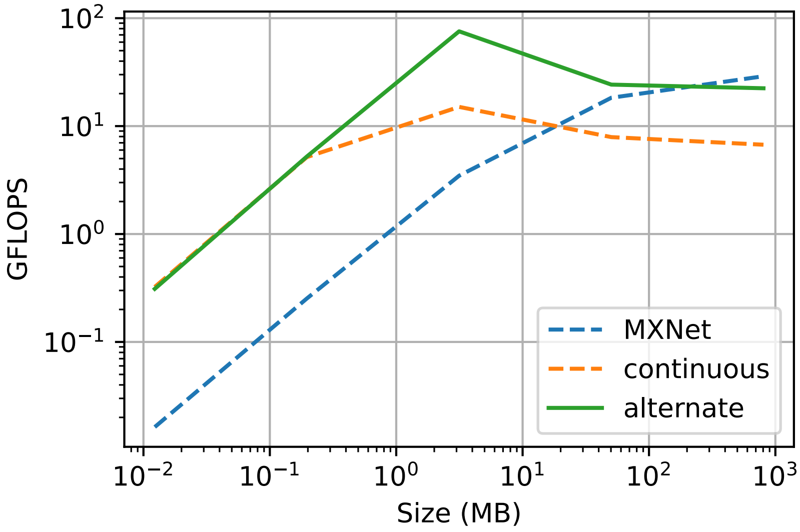
Comparing with the pseudo-code above, we can easily see that the data access changes. Now let's compile it to execute.

```
mod = tvm.build(s, args, 'cuda')

tvm_alternate_gflops = d2ltvm.bench_bcast_add_tvm(alternate_parallel, sizes,
↪'cuda')
```

(continues on next page)

(continued from previous page)

```
d2ltvm.plot_gflops(x_axis_sizes, [mx_gflops, tvm_continuous_gflops, tvm_
↪alternate_gflops],
                    ['MXNet', 'continuous', 'alternate'], xlabel='Size (MB)')
```



The performance is much better than continuous scheduling once `need_further_split=True`. We also observe that the performance drops notably after the data size exceeds the L2 cache size (6 MB) shared by all SMs.

Lastly, it is worthwhile to note that the two values `nt` and `nb` are tunable based on the workload and platform. It is not guaranteed that what we have chosen here is best for all cases we tested out. And there is no combination which works well across all cases. Therefore, a good compiler will need to allow tuning on this kind of parameters.

### 5.3.4 Summary

- It is favorable for CUDA threads to access data alternately.

### 5.3.5 Exercise

- Vary nt and nb to observe the performance difference.
- Try to schedule other broadcast add patterns and observe the difference.

## 5.4 Matrix Multiplication

In this section, we will extend Section 4.6 to optimize matrix multiplication on GPUs.

```
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te
```

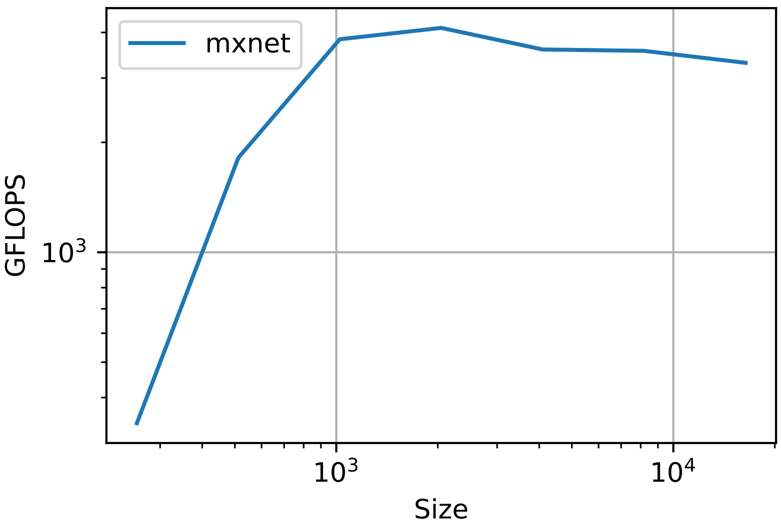
### 5.4.1 Setup

We will use MXNet as our baseline, which calls cuBLAS to execute the matrix multiplication.

```
# Save to the d2ltvm package.
def matmul_timer_mxnet(n, ctx):
    """The matrix multiplication timer for MXNet

    n : width and height of inputs
    ctx : device
    """
    timer = timeit.Timer(
        setup='import d2ltvm\n'
        'import mxnet as mx\n'
        'a, b, c, = d2ltvm.get_abc((%d, %d), lambda x: mx.nd.array(x, ctx=mx.
↪%s()))\n'
        'mx.nd.waitall()' % (n, n, ctx),
        stmt='mx.nd.dot(a, b, out=c); c.wait_to_read()')
    return timer.timeit
```

Then we compute its GFLOPS and plot the performance baseline as a function of matrix size.

```
sizes = 2**np.arange(8, 15, 1)
exe_times = [d2ltvm.bench_workload(matmul_timer_mxnet(int(n), 'gpu'))
         for n in sizes]
mx_gflops = 2 * sizes ** 3 / 1e9 / np.array(exe_times)
d2ltvm.plot_gflops(sizes, [mx_gflops], ['mxnet'])
```



## 5.4.2 Blocked Matrix Multiplication on GPU

We will follow Section 4.6 to split the matrix $C$ into blocks, and have each core (streaming multiprocessor) to compute a block at a time. It can be done by assigning a block to a thread block as we did in Section 5.2 (don't confuse the matrix block with thread block here). As mentioned in Section 5.1, the GPU core has a finer architecture, we need to split a block further for every CUDA thread in the thread block. The simplest 1-D case was already illustrated in Section 5.2. This section will explore the local memory within a core using 2-D thread indexing.

### Shared Memory

Within a GPU core, there is a shared memory (or L1 cache) that can be accessed by all threads, which is managed by the compiler and hardware. Unlike CPU cache, we can allocate space directly on the shared memory just as the same as on the main memory and the global GPU memory.

In the TVM abstraction, we also call it cache to simplify the concept. The TVM scheduling primitive `cache_read` can create a read-only cache for $A$ that will be used by $C$ on the shared memory, i.e. `s.cache_read(A, "shared", [C])`.

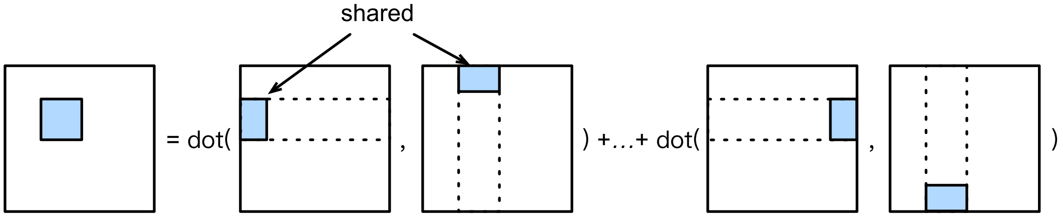


Fig. 5.4.1: Blocked tiling for matrix multiplication with blocks denoted in blue on shared memory.

In Section 4.6, we created a write cache of an output block. Here, we will explore the opportunity to create read caches for input blocks. We redraw Fig. 4.6.1 in Fig. 5.4.1, it shows how to compute an output block through a series of matrix multiplications over input blocks. Since we will use all threads in a thread block to calculate this output block, we can cache input blocks in the shared memory. Now we can rewrite the block computation in Section 4.6 as:

```
for k in range(0, n, tk):
    A_shared = A[y:y+ty, k:k+tk]  # cache in shared memory
    B_shared = B[k:k+tk, x:x+tx]  # cache in shared memory
    # use all threads in the thread block
    C[y:y+ty, x:x+tx] += dot(A_shared, B_shared)
```

Here `tx`, `ty` and `tk` are the tile sizes. The only difference is that we put the input blocks in the shared memory of a GPU core.

Assuming `tx=64`, `ty=128` and `tk=32`, then for each core, we will cache two matrices with sizes $128 \times 32$ and $32 \times 64$ on the shared memory, with a total size 24 KB. We can query the shared memory size per block in KB of the GPU we are using to make sure that these two matrices can fit into the shared memory. Note that this is different from the shared memory size per SM we reported in :numref"`ch_gpu_arch`.

```
ctx = tvm.gpu()
ctx.max_shared_memory_per_block/1024
```

```
48.0
```

### Thread Block and Registers

Next let's explore how to compute an output block using one GPU core in parallel efficiently. We can use the same idea: further splitting the output block into smaller block tiles, and having each thread to compute one tile. Fig. 5.4.2 shows splitting a $128 \times 64$ output block into 256 ($16 \times 16$) tiles, each of which is an $8 \times 4$ matrix. Then we will create 256 threads within this thread block. Since the output is a matrix, we use a 2-D thread indexing, with `blockDim.x = blockDim.y = 16`. In addition, we will move the inputs, two vectors with lengths of 8 and 4, respectively, and the output, an $8 \times 4$ matrix, for each thread into the registers.

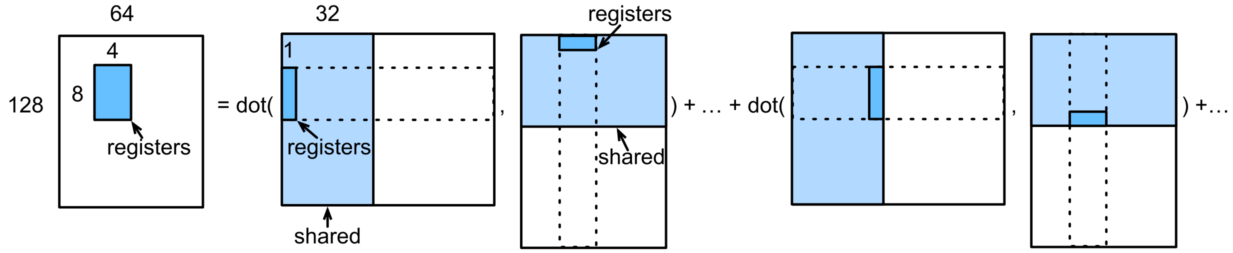


Fig. 5.4.2: Further blocked tiling for matrix multiplication to put small tiles into registers.

Registers are local memory to a CUDA thread that is running. Accessing the registers are faster than the shared memory. So our goal is to make sure the data that we want to use each time can fit into the registers. In our case, each thread has three tensors with sizes $8 \times 1$, $1 \times 4$ and $8 \times 4$, respectively. These lead to in total 46 32-bit floats. In the Tesla T4 GPU that we are using, each block has 65,536 32-bit registers shared by up to 1024 threads. Therefore, we can easily fit the data to the registers.

### Cooperative Fetching

Finally, loading the blocks of A_shared and B_shared into the shared memory is time consuming. We can accelerate it through multi-threading, namely using all threads in a thread block to load it.

## 5.4.3 Implementation

We first implement utility methods which split an axis with a list of factors, and bind a list of axes with threads.

```python
# Save into the d2ltvm package.
def split(stage, axis, factors):
    """Split an axis by a list of factors in a reverse order
    """
    axes = []
    for f in reversed(factors):
        axis, x = stage.split(axis, f)
        axes.append(x)
    return list(reversed(axes+[axis]))

# Save into the d2ltvm package.
def bind_thread(stage, axes, tags):
    """Bind a list of axes to thread axes
    """
    for axis, tag in zip(axes, tags):
        stage.bind(axis, te.thread_axis(tag))
```

Next we specify the hyperparameters with values we described before.

```python
block_size = 16  # the number of threads for one dimension in a thread block.
tx, ty, tk = 8, 4, 32  # tile sizes for one CUDA thread
```

Now we can implement our schedule. There are three things worth mentioning:

1. we denote by x the rows and y the columns, so an element can be assessed by C[x,y].
2. As mentioned above, in CUDA thread indexing, x is used for the innermost dimension, which is the matrix column in our case. Therefore you will see we bind axis yb (split from y) to blockIdx.x instead of blockIdx.y.
3. W need to partition the axes of A_shared and B_shared into block_size parts, so we can reuse the threads bound to xo and yo for cooperative fetching. Otherwise TVM may not properly synchronize threads which leads to wrong results.

```python
def matmul_gpu(n):
    A, B, C = d2ltvm.matmul(n, n, n)
    s = te.create_schedule(C.op)
    # Create caches
    A_shared = s.cache_read(A, "shared", [C])
    A_local  = s.cache_read(A_shared, "local", [C])
    B_shared = s.cache_read(B, "shared", [C])
    B_local  = s.cache_read(B_shared, "local", [C])
    C_local = s.cache_write(C, "local")
    # Split each axis into block axis, thread axis, and inner axis
    x, y = s[C].op.axis
```

(continues on next page)

(continued from previous page)

```
    xb, xo, xi = split(s[C], x, (block_size, tx))
    yb, yo, yi = split(s[C], y, (block_size, ty))
    s[C].reorder(xb, yb, xo, yo, xi, yi)
    # Note that we bind yb to blockIdx.x instead of blockIdx.y
    bind_thread(s[C], (yb, xb, yo, xo),
                ("blockIdx.x", "blockIdx.y", "threadIdx.x", "threadIdx.y"))
    # Schedule C_local
    s[C_local].compute_at(s[C], yo)
    yi, xi = s[C_local].op.axis
    k, = s[C_local].op.reduce_axis
    ko, ki = s[C_local].split(k, tk)
    s[C_local].reorder(ko, ki, yi, xi)
    # Optimize read caches of A and B with cooperative fetching
    def optimize_read_cache(shared, local):
        s[shared].compute_at(s[C_local], ko)
        s[local].compute_at(s[C_local], ki)
        y, x = s[shared].op.axis
        # Note that we must split into block_size parts to reuse
        # the previous axis threads
        yo, yi = s[shared].split(y, nparts=block_size)
        xo, xi = s[shared].split(x, nparts=block_size)
        s[shared].reorder(yo, xo, yi, xi)
        bind_thread(s[shared], (yo, xo), ("threadIdx.y", "threadIdx.x"))
    optimize_read_cache(A_shared, A_local)
    optimize_read_cache(B_shared, B_local)
    return s, (A, B, C)
```

Let's verify the correctness of the schedule. First we print the pseudo codes. Since we didn't unroll the loops, the pseudo codes are relative compact and we can check the allocated cache sizes and how each stage is computed.

```
n = 2048
s, args = matmul_gpu(n)
tvm.lower(s, args, simple_mode=True)
```

```
produce C {
  // attr [iter_var(blockIdx.y, , blockIdx.y)] thread_extent = 16
  // attr [C.local] storage_scope = "local"
  allocate C.local[float32 * 32]
  // attr [A.shared] storage_scope = "shared"
  allocate A.shared[float32 * 4096]
  // attr [B.shared] storage_scope = "shared"
  allocate B.shared[float32 * 2048]
  // attr [A.shared.local] storage_scope = "local"
  allocate A.shared.local[float32 * 8]
  // attr [B.shared.local] storage_scope = "local"
  allocate B.shared.local[float32 * 4]
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 32
  // attr [iter_var(threadIdx.y, , threadIdx.y)] thread_extent = 16
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 16
  produce C.local {
    for (x.c.init, 0, 8) {
      for (y.c.init, 0, 4) {
        C.local[((x.c.init*4) + y.c.init)] = 0f
      }
```

(continues on next page)

(continued from previous page)

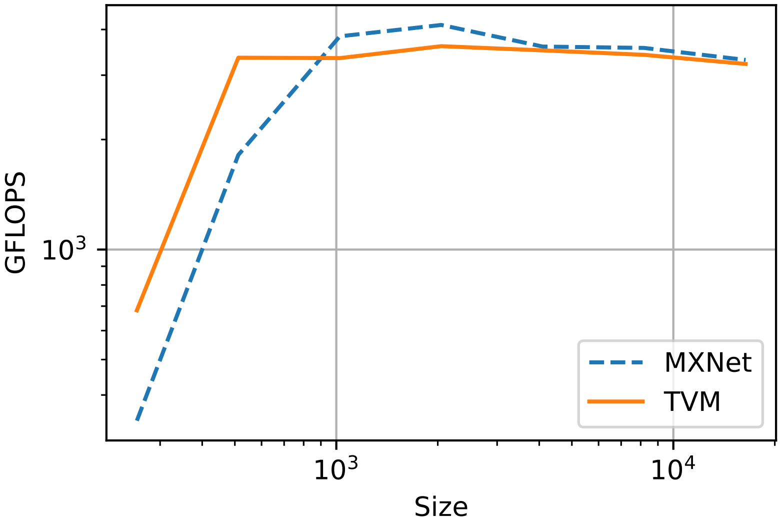
```
  }
  for (k.outer, 0, 64) {
   produce A.shared {
    // attr [iter_var(threadIdx.y, , threadIdx.y)] thread_extent = 16
    // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 16
    for (ax0.inner, 0, 8) {
     for (ax1.inner, 0, 2) {
      A.shared[((((threadIdx.y*256) + (ax0.inner*32)) + (threadIdx.
↪x*2)) + ax1.inner)] = A[((((((blockIdx.y*262144) + (threadIdx.y*16384))␣
↪+ (ax0.inner*2048)) + (k.outer*32)) + (threadIdx.x*2)) + ax1.inner)]
     }
    }
   }
   produce B.shared {
    // attr [iter_var(threadIdx.y, , threadIdx.y)] thread_extent = 16
    // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 16
    for (ax0.inner, 0, 2) {
     for (ax1.inner, 0, 4) {
      B.shared[((((threadIdx.y*128) + (ax0.inner*64)) + (threadIdx.
↪x*4)) + ax1.inner)] = B[((((((k.outer*65536) + (threadIdx.y*4096)) + (ax0.
↪inner*2048)) + (blockIdx.x*64)) + (threadIdx.x*4)) + ax1.inner)]
     }
    }
   }
   for (k.inner, 0, 32) {
    produce A.shared.local {
     for (ax0, 0, 8) {
      A.shared.local[ax0] = A.shared[(((threadIdx.y*256) + (ax0*32)) +␣
↪k.inner)]
     }
    }
    produce B.shared.local {
     for (ax1, 0, 4) {
      B.shared.local[ax1] = B.shared[(((k.inner*64) + (threadIdx.x*4))␣
↪+ ax1)]
     }
    }
    for (x.c, 0, 8) {
     for (y.c, 0, 4) {
      C.local[((x.c*4) + y.c)] = (C.local[((x.c*4) + y.c)] + (A.shared.
↪local[x.c]*B.shared.local[y.c]))
     }
    }
   }
  }
 }
 for (x.inner, 0, 8) {
  for (y.inner, 0, 4) {
   C[((((((blockIdx.y*262144) + (threadIdx.y*16384)) + (x.inner*2048)) +␣
↪(blockIdx.x*64)) + (threadIdx.x*4)) + y.inner)] = C.local[((x.inner*4) + y.
↪inner)]
  }
 }
}
```

Next we compare the results against NumPy results to check the correctness.

```
target, ctx = 'cuda', tvm.gpu()
mod = tvm.build(s, args, target)
a, b, c, = d2ltvm.get_abc((n, n), lambda x: tvm.nd.array(x, ctx=ctx))
mod(a, b, c)
np.testing.assert_allclose(
    c.asnumpy(), np.dot(a.asnumpy(), b.asnumpy()), atol=1e-2)
```

Finally, we measure the performance to compare with our baseline. You can see that our schedule works well for small matrices but is constantly slower for large ones. The reason might due to 1) we didn't consider bank conflict when reading share memory; 2) there's other optimization opportunity that we didn't investigate; 3) previous works show that pure assembly codes, which cuBLAS uses, provide more room to optimize and often outperform CUDA codes (Nath et al., 2010; Lai & Seznec, 2013).

```
tvm_gflops = d2ltvm.bench_matmul_tvm(matmul_gpu, sizes, 'cuda')
d2ltvm.plot_gflops(sizes, [mx_gflops, tvm_gflops], legend=['MXNet', 'TVM'])
```



### 5.4.4 Summary

- We use a two-level block tiling to parallelize matrix multiplication on GPUs.
- We load data used by a thread block into share memory, and data used by a CUDA thread into registers.
- The shared data within a thread block is loaded by cooperative fetching.

## 5.5 Convolution

In this section, we will extend Section 4.8 to optimize convolution on GPUs.

```
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te
```

(continues on next page)

(continued from previous page)

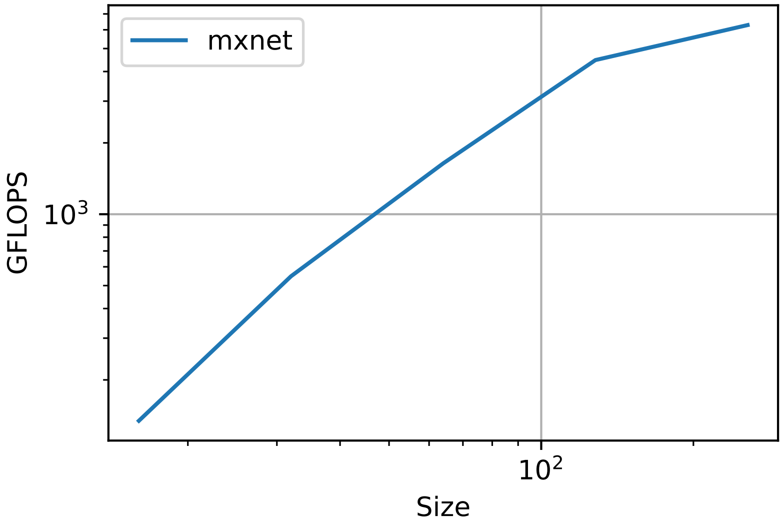
```
target = 'cuda'
```

### 5.5.1 Setup

As usual, we will use MXNet as our baseline, which calls cuDNN to execute the convolution. Like what we have done on CPUs, we benchmark the performance with various numbers of channels, when the input and kernel width/height are fixed to be 64 and 3, respectively. The benchmark method `bench_conv_mxnet` has already been defined in Section 4.8. The only change that we need to make is to specify to the method that the target device is `GPU`.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]
mx_gflops = d2ltvm.bench_conv_mxnet(sizes, 'gpu')
d2ltvm.plot_gflops(channels, [mx_gflops], ['mxnet'])
mx_gflops
```

```
[134.41320431231432,
 547.5253539941237,
 1639.1830964521912,
 4471.396488551705,
 6281.556088500076]
```



The results above show that on GPUs the performance of convolution increases while the number of channels increases. For showing the gradual performance improvement brought by TVM scheduling, we now fix the channel size to be 64, where MXNet could get to the performance of about 1700 GFLOPS or 1.7 TGLOPS. Please keep this in mind while we are working on the TVM scheduling.

```
sizes = [(64, 64, 3)]
```

### 5.5.2 Default schedule of CONV

We then describe the computation of convolution in TVM using `conv` method, which is defined in Section 4.8. For a default schedule, we can simply bind two axes of the convolution loop nest into block and thread indexing axes.

```
def default_sch(oc, ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.conv(oc, ic, n, n, k, k, p, p, s, s)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()
    _, y, x = sch[Y].op.axis
    sch[Y].bind(y, te.thread_axis("blockIdx.x"))
    sch[Y].bind(x, te.thread_axis("threadIdx.x"))
    print(tvm.lower(sch, [X, K, Y], simple_mode=True))
    return sch, (X, K, Y)

tvm_gflops = d2ltvm.bench_conv_tvm(default_sch, sizes, target)
tvm_gflops
```

```
produce Y {
  for (c, 0, 64) {
    // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 64
    // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 64
    Y[(((c*4096) + (blockIdx.x*64)) + threadIdx.x)] = 0f
    for (ric, 0, 64) {
      for (rkh, 0, 3) {
        for (rkw, 0, 3) {
          Y[(((c*4096) + (blockIdx.x*64)) + threadIdx.x)] = (Y[(((c*4096) +␣
↪(blockIdx.x*64)) + threadIdx.x)] + (tvm_if_then_else((((((blockIdx.x + rkh)
↪< 1) || (65 <= (blockIdx.x + rkh))) || ((threadIdx.x + rkw) < 1)) || (65 <=␣
↪(threadIdx.x + rkw))), 0f, X[((((((ric*4096) + (blockIdx.x*64)) + (rkh*64))␣
↪+ threadIdx.x) + rkw) - 65)])*K[((((c*576) + (ric*9)) + (rkh*3)) + rkw)]))
        }
      }
    }
  }
}
```

```
array([129.81401547])
```

The default scheduling gives us the performance around 100 GFLOPS.

### 5.5.3 Tiling

As described in the last section, we can do tiling and bring the data to the shared and local memory of the GPU explicitly to improve the performance. Specifically, we tile three dimensions of the output (channel, height and width) as well as the three reduce dimensions (input channel, kernel height and kernel width). For the output dimensions, we split each of them into three parts for block binding, thread binding and CUDA kernel processing, respectively. The splitting factors are chosen to make sure the data can be fit into the shared and local memory of the GPU.

In our case, we specify the output tile (`YL`) and its corresponding input tiles (`XL` and `KL`) to the local memory, and the tiled input data (`XX`) and kernel (`KK`) to the shared memory. Fig. 5.5.1 shows how the tiling works for convolution.

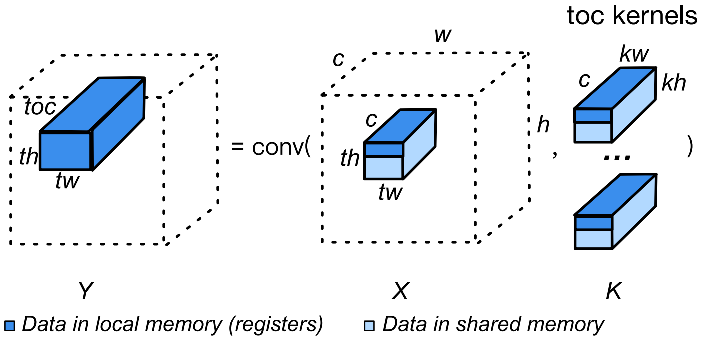


Fig. 5.5.1: Blocked tiling for convolution to put small tiles into shared and local memory of GPU.

Under our tiling factors below, each thread needs to access 64(`YL`)+48(`XL`)+24(`KL`)= 136 32-bit floats. And in our setting each block contains $4 \times 2 \times 16 = 128$ threads, making the total occupied local memory registers to be $128 \times 136 = 17,408$, which is easy to be fit into one SM. Similarly, we can reason that `XX` and `KK` are fittable to the shared memory.

In addition, as in the last section, we use cooperative fetching to local the data from GPU host memory to the shared memory in parallel.

```
tile_c = [4, 8]
tile_h = [2, 2]
tile_w = [16, 4]
tile_rc = [1, 1]
tile_rh = [1, 1]
tile_rw = [1, 3]

# Save to the d2ltvm package.
def split_axis(factors, sch, op, axis):
        """Splitting an axis into factors

        Parameters
        ----------
        factors: array of integers
            The factors that the split applies
        sch: tvm.te.schedule.Schedule
            The tvm schedule
        op: tvm.te.tensor.Operation
            The stage to be applied
        axis: tvm.te.schedule.IterVar
            axis to split

        Returns
        -------
        axes : list of Axis
            The transformed axes.
        """
        ret = []
        for i in range(0, len(factors)):
            ax0, ax1 = sch[op].split(axis, factor=int(np.prod(factors[i:])))
            ret.append(ax0)
```

(continues on next page)

(continued from previous page)

```
            axis = ax1
        return ret + [axis]

def tiling(oc, ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.conv(oc, ic, n, n, k, k, p, p, s, s)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()

    YL = sch.cache_write(Y, 'local')

    # create cache stage
    XX = sch.cache_read(PaddedX, 'shared', [YL])
    KK = sch.cache_read(K, 'shared', [YL])
    XL = sch.cache_read(XX, 'local', [YL])
    KL = sch.cache_read(KK, 'local', [YL])

    c, h, w = sch[Y].op.axis

    bc, tc, ic = split_axis(tile_c, sch, Y, c)
    bh, th, ih = split_axis(tile_h, sch, Y, h)
    bw, tw, iw = split_axis(tile_w, sch, Y, w)

    sch[Y].bind(bc, te.thread_axis("blockIdx.z"))
    sch[Y].bind(bh, te.thread_axis("blockIdx.y"))
    sch[Y].bind(bw, te.thread_axis("blockIdx.x"))
    sch[Y].bind(tc, te.thread_axis("threadIdx.z"))
    sch[Y].bind(th, te.thread_axis("threadIdx.y"))
    sch[Y].bind(tw, te.thread_axis("threadIdx.x"))
    sch[Y].reorder(bc, bh, bw, tc, th, tw, ic, ih, iw)

    sch[YL].compute_at(sch[Y], tw)

    # tile reduction axes
    c, h, w = sch[YL].op.axis
    rc, rh, rw = sch[YL].op.reduce_axis
    rco, rcm, rci = split_axis(tile_rc, sch, YL, rc)
    rho, rhm, rhi = split_axis(tile_rh, sch, YL, rh)
    rwo, rwm, rwi = split_axis(tile_rw, sch, YL, rw)
    sch[YL].reorder(rco, rho, rwo, rcm, rhm, rwm, rci, rhi, rwi, c, h, w)

    sch[XX].compute_at(sch[YL], rwo)
    sch[KK].compute_at(sch[YL], rwo)
    sch[XL].compute_at(sch[YL], rwm)
    sch[KL].compute_at(sch[YL], rwm)

    # cooperative fetching
    for load in [XX, KK]:
        args = sch[load].op.axis
        fused = sch[load].fuse(*args)
        # align thread layout
        tz, fused = sch[load].split(fused, nparts=tile_c[0])
        ty, fused = sch[load].split(fused, nparts=tile_h[0])
        tx, _ = sch[load].split(fused, nparts=tile_w[0])
        sch[load].bind(tz, te.thread_axis("threadIdx.z"))
        sch[load].bind(ty, te.thread_axis("threadIdx.y"))
```

(continues on next page)

(continued from previous page)

```
        sch[load].bind(tx, te.thread_axis("threadIdx.x"))

    return sch, (X, K, Y)

tvm_gflops = d2ltvm.bench_conv_tvm(tiling, sizes, target)
tvm_gflops
```

```
array([1871.96109223])
```

The performance increases over one order of magnitude which is already on par with our baseline. We can still improve it by solving some bank conflict issue when accessing the data.

## 5.5.4 Optimizing the data access on GPUs

### Bank Conflict

In the scheduling above, all threads will read `XX` and `KK` simultaneously, which may harm the performance. To understand it, we need to dive a little bit into the shared memory architecture and how threads are executed.

Remember that we created 128 threads for a thread block. Due to resource limits, we cannot execute all of them at the same time. Instead, each time we select a group of threads and run time simultaneously, and then switch to another group quickly. Such a group is called a *warp*, which contains 32 threads, and each having a consecutive thread index.

Each thread in a warp can access data in the shared memory simultaneously. So the shared memory is designed to support parallel load and store. The basic unit of the shared memory is *word*. Each word has 4 bytes, which can hold a single 32-bit floating number. Words are grouped into 32 banks. The $j$-th word is in the $i$-th bank if `j % 32 == i`.

Each bank can only handle a single request at a time, while these 32 banks are processed in parallel. So the shared memory performs fastest when each bank gets one request from a single thread in the warp, therefore we can access 32 words at the same time. However, if there are two threads requesting data from the same bank, we need to serialize these two requests. This is called a *bank conflict*. A special case is that if all (or some) threads of a warp request the same word in a bank, then the word will only read once and broadcast (multicast) to each thread, so there is no bank conflict. Fig. 5.5.2 illustrated these three cases.

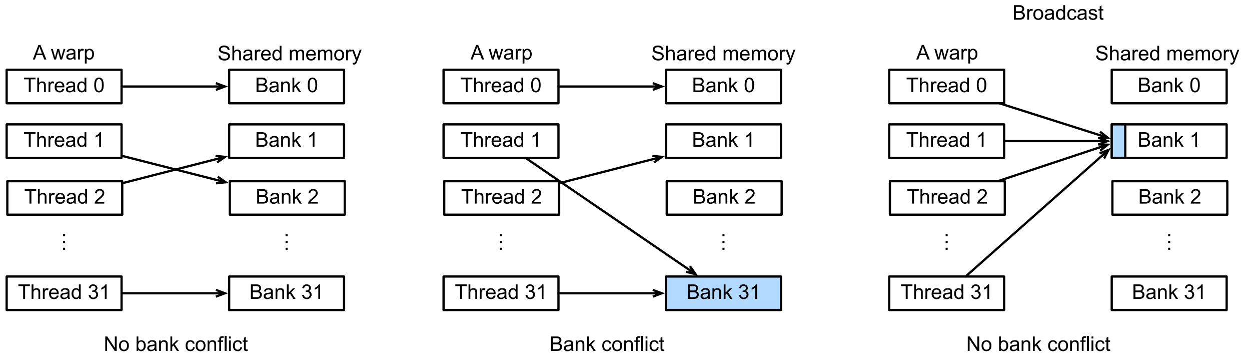


Fig. 5.5.2: Accessing shared memory

### Data Access Pattern

Now let's analyze the read pattern of XX and KK. Note from Fig. 5.5.1 that a thread reads a row segment of XX with a length of 4 consecutive numbers, which spread in 4 adjacent banks which causes severe bank conflict. The same thing applies to the reading of KK.

One way to mitigate this is to let the thread read in columns instead of rows, so each reading would have a stride, making the numbers to be read from one thread more spread among banks. Fig. 5.5.3 shows the difference of the reading patterns from shared memory to local memory.

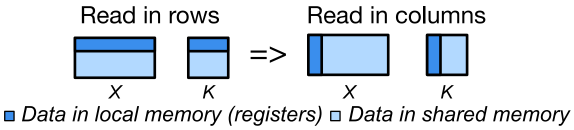


Fig. 5.5.3: Different reading patterns from shared memory to local memory

### Virtual Threads

In addition, TVM provides another mechanism, called *virtual thread*, to further increase the data access stride to mitigate bank conflict. Let's revisit the thread structure we defined above. Each block has 16 threads in the x dimension, each thread processes 4 elements. It conceptually gets data in the pattern depicted in the left of Fig. 5.5.4.

In order to let threads to process data in a spread manner, we can use virtual threads to obtain strided data chunks. For example, we can first split the data into 2 parts to bind to 2 virtual threads. Then we further split the each part into 16 pieces to bind to 16 CUDA threads. In practice, the $i$-th CUDA thread in all virtual threads will be merged into a single one, so we will only get 16 CUDA threads instead of $16 \times 2$ threads. In this case, each thread processes two spread data pieces as depicted in the right of Fig. 5.5.4.

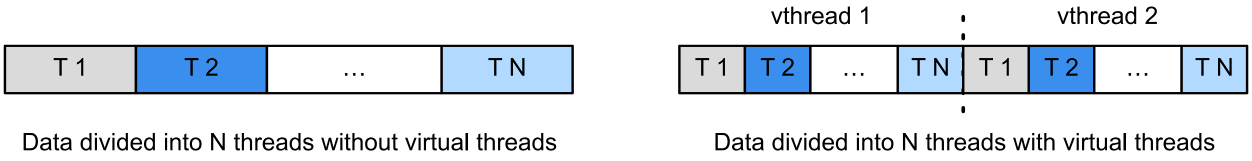

T 1
T 2
...
T N
Data divided into N threads without virtual threads
vthread 1
vthread 2
T 1
T 2
...
T N
T 1
T 2
...
T N
Data divided into N threads with virtual threads

Fig. 5.5.4: Virtual thread binding

```
tile_c = [1, 4, 8]
tile_h = [1, 2, 2]
tile_w = [2, 16, 2] # making 2 virtual thread along the ow dimension
tile_rc = [1, 1]
tile_rh = [1, 3] # making the data access in columns
tile_rw = [1, 1]

def vthread(oc, ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.conv(oc, ic, n, n, k, k, p, p, s, s)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()

    YL = sch.cache_write(Y, 'local')
```

(continues on next page)

(continued from previous page)

```
    # create cache stage
    XX = sch.cache_read(PaddedX, 'shared', [YL])
    KK = sch.cache_read(K, 'shared', [YL])
    XL = sch.cache_read(XX, 'local', [YL])
    KL = sch.cache_read(KK, 'local', [YL])

    c, h, w = sch[Y].op.axis

    bc, vc, tc, ic = split_axis(tile_c, sch, Y, c)
    bh, vh, th, ih = split_axis(tile_h, sch, Y, h)
    bw, vw, tw, iw = split_axis(tile_w, sch, Y, w)

    sch[Y].bind(bc, te.thread_axis("blockIdx.z"))
    sch[Y].bind(bh, te.thread_axis("blockIdx.y"))
    sch[Y].bind(bw, te.thread_axis("blockIdx.x"))
    sch[Y].bind(vc, te.thread_axis("vthread"))
    sch[Y].bind(vh, te.thread_axis("vthread"))
    sch[Y].bind(vw, te.thread_axis("vthread"))
    sch[Y].bind(tc, te.thread_axis("threadIdx.z"))
    sch[Y].bind(th, te.thread_axis("threadIdx.y"))
    sch[Y].bind(tw, te.thread_axis("threadIdx.x"))
    sch[Y].reorder(bc, bh, bw, vc, vh, vw, tc, th, tw, ic, ih, iw)

    sch[YL].compute_at(sch[Y], tw)

    # tile reduction axes
    c, h, w = sch[YL].op.axis
    rc, rh, rw = sch[YL].op.reduce_axis
    rco, rcm, rci = split_axis(tile_rc, sch, YL, rc)
    rho, rhm, rhi = split_axis(tile_rh, sch, YL, rh)
    rwo, rwm, rwi = split_axis(tile_rw, sch, YL, rw)
    sch[YL].reorder(rco, rho, rwo, rcm, rhm, rwm, rci, rhi, rwi, c, h, w)

    sch[XX].compute_at(sch[YL], rwo)
    sch[KK].compute_at(sch[YL], rwo)
    sch[XL].compute_at(sch[YL], rwm)
    sch[KL].compute_at(sch[YL], rwm)

    # cooperative fetching
    for load in [XX, KK]:
        args = sch[load].op.axis
        fused = sch[load].fuse(*args)
        # align thread layout
        tz, fused = sch[load].split(fused, nparts=tile_c[1])
        ty, fused = sch[load].split(fused, nparts=tile_h[1])
        tx, _ = sch[load].split(fused, nparts=tile_w[1])
        sch[load].bind(tz, te.thread_axis("threadIdx.z"))
        sch[load].bind(ty, te.thread_axis("threadIdx.y"))
        sch[load].bind(tx, te.thread_axis("threadIdx.x"))

    return sch, (X, K, Y)

tvm_gflops = d2ltvm.bench_conv_tvm(vthread, sizes, target)
tvm_gflops
```

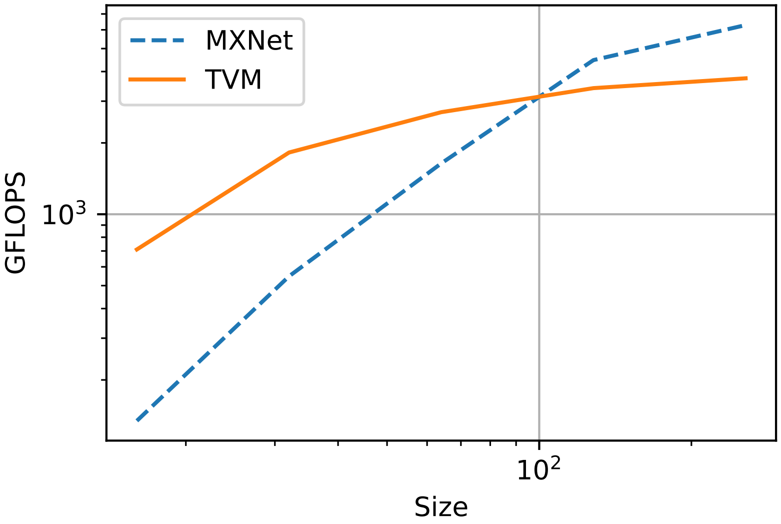
```
array([2705.41367495])
```

After carefully optimizing the data access, the performance we get outperforms our baseline at `channel=64`.

Now let's vary the number of channels to test out the convolution performance obtained by TVM more comprehensively. And then we can plot the chart to compare MXNet and TVM.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]
target = 'cuda'
tvm_gflops = d2ltvm.bench_conv_tvm(vthread, sizes, target)
d2ltvm.plot_gflops(channels, [mx_gflops, tvm_gflops], legend=['MXNet', 'TVM'])
tvm_gflops
```

```
array([ 710.40119273, 1822.09562259, 2695.62647693, 3405.25584565,
       3746.19560406])
```



From the figure we see that TVM outperforms MXNet in smaller channel sizes but MXNet becomes better as the number of channels increases. This is mostly because cuDNN used by MXNet has manually optimized implementation for different data shapes, but here we use only one scheduling strategy for convolution kernels in different sizes.

You may wonder how we can choose different schedules for convolutions of different data sizes, and even better, if we can automate the choice of schedules given a specific set of data shapes for convolution. We will talk about these techniques later.

In addition, there are ways that one can do to further increase the performance. For example, we can try to avoid all bank conflict by making the data reading stride always a multiple of 32. However, these tricks may be ad hoc and require intensive programming efforts. Our goal is to come up with some more high-level and generic scheduling scheme to achieve the reasonable performance.

### 5.5.5 Summary

- We leverage the memory hierarchy of GPU to tile the data for better convolution performance.
- We carefully manipulate the data access pattern to mitigate bank conflict which harms the performance.

### 5.5.6 Exercise

- Try our different factors to split the axes and observe the performance difference.
- Vary the size of input data and observe the performance difference.

## 5.6 Depthwise Convolution

In this section, we will talk about how to optimize depthwise convolution on GPUs.

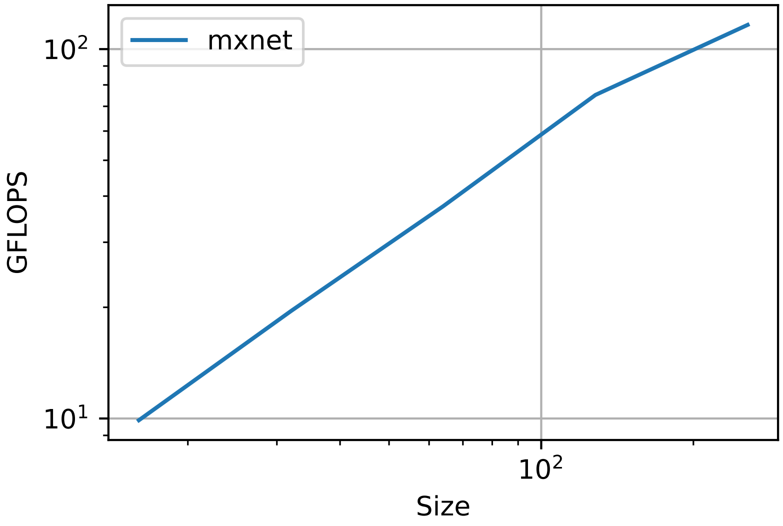
```
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te

target = 'cuda'
```

### 5.6.1 Setup

The baseline of depthwise convolution on GPUs is given by MXNet, which relies on cuDNN for high performance. Again, we benchmark the performance with various numbers of channels, when the input and kernel width/height are fixed to be 64 and 3, respectively. The benchmark method `depthwise_conv_timer_mxnet` has already been defined in Section 4.9. The only change that we need to make is to specify to the method that the target device is `GPU`.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]
mx_gflops = d2ltvm.bench_depthwise_conv_mxnet(sizes, 'gpu')
d2ltvm.plot_gflops(channels, [mx_gflops], ['mxnet'])
mx_gflops
```

```
[9.891531650894763,
 19.543424128359018,
 37.55850310833435,
 75.14255278388472,
 116.29688859027996]
```



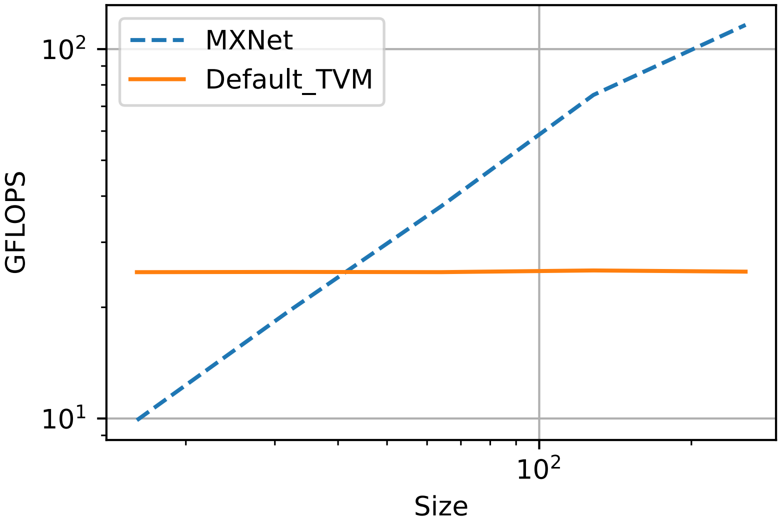
It is expected to see that the performance of depthwise convoluyion on GPUs increases while the number of channels increases.

### 5.6.2 Default schedule of Depthwise Convolution

In order to show the effectiveness of scheduling, we first apply a default schedule, which does nothing but only binds the axes to GPU thread and block.

```
def default_sch(ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.depthwise_conv(ic, n, n, k, k, p, p, s, s)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()
    _, y, x = sch[Y].op.axis
    sch[Y].bind(y, te.thread_axis("blockIdx.x"))
    sch[Y].bind(x, te.thread_axis("threadIdx.x"))
    return sch, (X, K, Y)

default_tvm_gflops = d2ltvm.bench_depthwise_conv_tvm(default_sch, sizes,␣
↪target)
d2ltvm.plot_gflops(channels, [mx_gflops, default_tvm_gflops], legend=['MXNet',
↪ 'Default_TVM'])
default_tvm_gflops
```

```
array([24.89613593, 24.93398427, 24.90192682, 25.16284254, 24.97764198])
```



The default scheduling gives us the performance around 25 GFLOPS for every data shape we investigate, indicating that the compute power is not actually used.

### 5.6.3 Scheduling of Depthwise Convolution

We work on the scheduling of depthwise convolution from the following aspects. Note that none of them is new, all covered in the previous sections.

Remember that the depthwise convolution convolves each input channel with a dedicated kernel, we can simply assign each channel to a different CUDA block. By doing this, we make different SMs work on different portions of the data, avoiding data contention across channels.

In terms of tiling, we followed the same trick done in Section 5.5 to bring some data to the shared and local memory of the GPU. You can follow the analytics approach described in Section 5.5 to calculate the cached data size and make sure the data fits in the cache. And we continue doing the cooperative fetching as before.

There is another key point for getting the good performance out of a GPU, which is mitigating the bank conflict described in Section 5.5. Unlike using the virtual thread in Section 5.5, this time we manipulate the data access pattern as illustrated in Fig. 5.5.3. That is, we read the data in columns to bring them into the local memory.

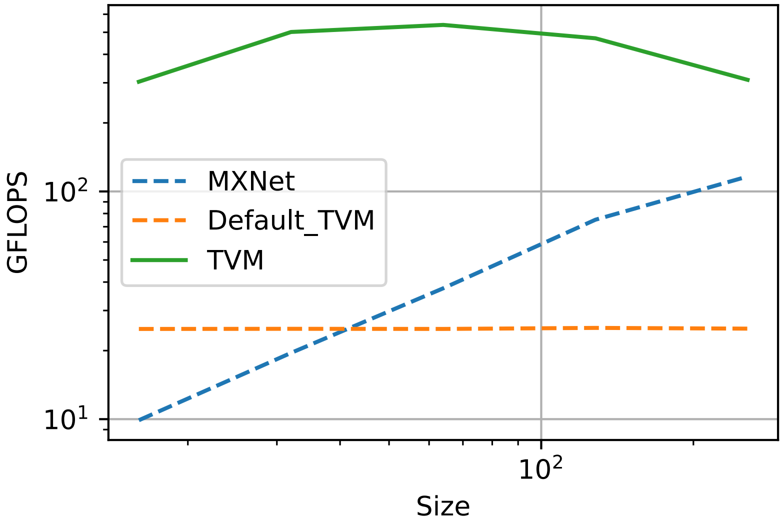
```
tile_c = [1, 1]  # making each block take 1 channel
tile_h = [2, 8]  # making each thread take 8 rows
tile_w = [64, 1] # making each thread take 1 column

def schedule(ic, n, k, p, s):
    X, K, Y, PaddedX = d2ltvm.depthwise_conv(ic, n, n, k, k, p, p, s, s)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()

    YL = sch.cache_write(Y, 'local')
    # create cache stage
    XX = sch.cache_read(PaddedX, 'shared', [YL])
    KK = sch.cache_read(K, 'shared', [YL])
    XL = sch.cache_read(XX, 'local', [YL])
    KL = sch.cache_read(KK, 'local', [YL])
```

(continues on next page)

(continued from previous page)

```
    # tile and bind spatial axes
    c, h, w = sch[Y].op.axis
    bc, tc, ic = d2ltvm.split_axis(tile_c, sch, Y, c)
    bh, th, ih = d2ltvm.split_axis(tile_h, sch, Y, h)
    bw, tw, iw = d2ltvm.split_axis(tile_w, sch, Y, w)

    sch[Y].bind(bc, te.thread_axis("blockIdx.z"))
    sch[Y].bind(bh, te.thread_axis("blockIdx.y"))
    sch[Y].bind(bw, te.thread_axis("blockIdx.x"))
    sch[Y].bind(tc, te.thread_axis("threadIdx.z"))
    sch[Y].bind(th, te.thread_axis("threadIdx.y"))
    sch[Y].bind(tw, te.thread_axis("threadIdx.x"))
    sch[Y].reorder(bc, bh, bw, tc, th, tw, ic, ih, iw)

    sch[YL].compute_at(sch[Y], tw)

    sch[XX].compute_at(sch[Y], bw)
    sch[KK].compute_at(sch[Y], bw)
    sch[XL].compute_at(sch[Y], tw)
    sch[KL].compute_at(sch[Y], tw)

    # cooperative fetching
    for load in [XX, KK]:
        args = sch[load].op.axis
        fused = sch[load].fuse(*args)
        # align thread layout
        tz, fused = sch[load].split(fused, nparts=tile_c[0])
        ty, fused = sch[load].split(fused, nparts=tile_h[0])
        tx, _ = sch[load].split(fused, nparts=tile_w[0])
        sch[load].bind(tz, te.thread_axis("threadIdx.z"))
        sch[load].bind(ty, te.thread_axis("threadIdx.y"))
        sch[load].bind(tx, te.thread_axis("threadIdx.x"))
    return sch, (X, K, Y)

tvm_gflops = d2ltvm.bench_depthwise_conv_tvm(schedule, sizes, target)
d2ltvm.plot_gflops(channels, [mx_gflops, default_tvm_gflops, tvm_gflops],␣
↪legend=['MXNet', 'Default_TVM', 'TVM'])
tvm_gflops
```

```
array([303.29601158, 501.50329581, 538.83357121, 470.61253687,
       309.149283  ])
```



We can see that after properly scheduling the computation, TVM can boost the performance of depthwise convolution by over one order of magnitude, which is also much better that the MXNet baseline.

It is worthwhile noting that, in the real workloads, depthwise convolution consumes only a little computation, which can be finished in microseconds. Therefore, although TVM outperforms MXNet for quite a lot, the real executing time difference is marginal. This somewhat reflects the famous Amdahl's law[47], i.e. in the real use case, we should first focus on optimizing the hot spots which takes the majority of executing time.

### 5.6.4 Summary

- Optimizing the depthwise convolution on GPUs has no major difference from optimizing other operators. The same techniques, e.g. parallelization, tiling, apply.

### 5.6.5 Exercise

- Try to use virtual thread to mitigate the bank conflict in depthwise convolution.
- Vary the size of input data and observe the performance difference.

## 5.7 Pooling

In this section, we will optimize pooling on GPUs. It is relatively easy compared to the schedulings we discussed for `matmul` and `conv`.

```
import d2ltvm
import numpy as np
import timeit
import tvm
from tvm import te

target = 'cuda'
```

[47] https://en.wikipedia.org/wiki/Amdahl%27s_law

### 5.7.1 Scheduling

#### Max Pooling

Like scheduling on CPUs at Section 4.10, we first use `compute_inline` to inject the padding compute into the pooling stage of `max pooling`. After that, what we are scheduling is essentially two-level for loop traversal of a two-level reduction.

It is easy to see that we can attach the two-level traversal to CUDA blocks and threads respectively. As CUDA threads in the same CUDA block will be run in the same SM (review Section 5.2 if you forget those concepts) and `max pooling` is a memory bound operator, we maximize the number of threads in the block.

Another scheduling approach we usually take in GPUs is managing the memory hierarchy. In this case, we want to bring the two-level reduction to the local memory for computation.

Finally, we can print out the IR to check.

```python
# attain the maximal number of threads of a CUDA block
nt = 0
with tvm.target.create(target):
    nt = tvm.target.Target.current(allow_none=False).max_num_threads

def schedule_max(size):
    c, n, k = size[:]
    X, Y, PaddedX = d2ltvm.pool('max', c, n, n, k, k, 1, 1, 1, 1)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()
    # traversal axes binding
    fused = sch[Y].fuse(*sch[Y].op.axis)
    bx, tx = sch[Y].split(fused, factor=nt)
    sch[Y].bind(bx, te.thread_axis("blockIdx.x"))
    sch[Y].bind(tx, te.thread_axis("threadIdx.x"))

    return sch, (X, Y)

# (channel, input width and height, kernel width and height)
size = (64, 64, 3)
sch, args = schedule_max(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce PoolMax {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 256
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 1024
  PoolMax[((blockIdx.x*1024) + threadIdx.x)] = -3.40282e+38f
  for (rkh, 0, 3) {
    for (rkw, 0, 3) {
      PoolMax[((blockIdx.x*1024) + threadIdx.x)] = max(PoolMax[((blockIdx.
↪x*1024) + threadIdx.x)], tvm_if_then_else((((((rkh + floormod(((blockIdx.
↪x*16) + floordiv(threadIdx.x, 64)), 64)) < 1) || (65 <= (rkh +␣
↪floormod(((blockIdx.x*16) + floordiv(threadIdx.x, 64)), 64)))) || ((rkw␣
↪+ floormod(threadIdx.x, 64)) < 1)) || (65 <= (rkw + floormod(threadIdx.x,␣
↪64)))), -3.40282e+38f, X[(((((blockIdx.x*1024) + (rkh*64)) + threadIdx.x) +␣
↪rkw) - 65)]))
    }
  }
}
```

### Avg Pooling

`Avg pooling` is similar to `max pooling` except that there are two stages for the pooling computation. As discussed at Section 4.10, we used the `compute_at` scheduling primitive to merge the two stages. Other than that, `avg pooling` reuses the scheduling scheme of `max pooling`.

We also print out the IR for observation.

```python
def schedule_avg(size):
    c, n, k = size[:]
    X, Y, PaddedX = d2ltvm.pool('avg', c, n, n, k, k, 1, 1, 1, 1)
    sch = te.create_schedule(Y.op)
    sch[PaddedX].compute_inline()

    # traversal axes binding
    fused = sch[Y].fuse(*sch[Y].op.axis)
    bx, tx = sch[Y].split(fused, factor=nt)
    sch[Y].bind(bx, te.thread_axis("blockIdx.x"))
    sch[Y].bind(tx, te.thread_axis("threadIdx.x"))

    # merging two stages
    PoolSum = Y.op.input_tensors[0]
    sch[PoolSum].compute_at(sch[Y], tx)
    return sch, (X, Y)

# (channel, input width and height, kernel width and height)
size = (64, 64, 3)
sch, args = schedule_avg(size)
mod = tvm.build(sch, args, target)
print(tvm.lower(sch, args, simple_mode=True))
```

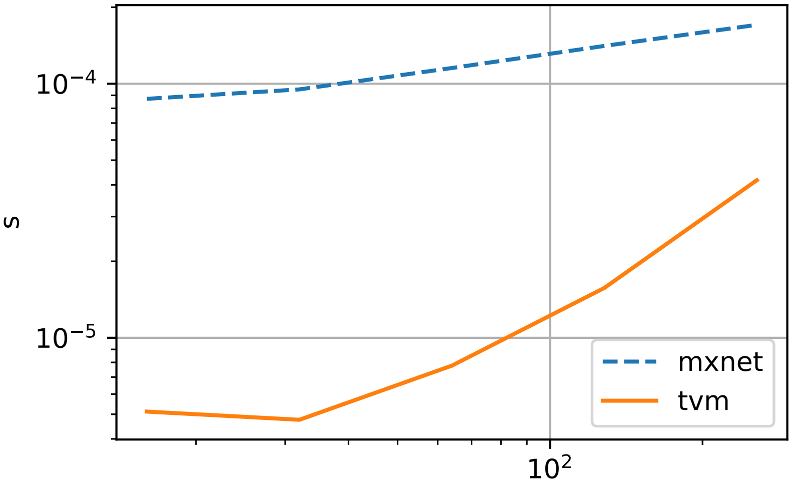
```
produce PoolAvg {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 256
  // attr [PoolSum] storage_scope = "local"
  allocate PoolSum[float32 * 1]
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 1024
  produce PoolSum {
    PoolSum[0] = 0f
    for (rkh, 0, 3) {
      for (rkw, 0, 3) {
        PoolSum[0] = (PoolSum[0] + tvm_if_then_else((((((rkh +␣
 ↪floormod(((blockIdx.x*16) + floordiv(threadIdx.x, 64)), 64)) < 1) ||␣
 ↪(65 <= (rkh + floormod(((blockIdx.x*16) + floordiv(threadIdx.x, 64)),
 ↪ 64)))) || ((rkw + floormod(threadIdx.x, 64)) < 1)) || (65 <= (rkw +␣
 ↪floormod(threadIdx.x, 64)))), 0f, X[(((((blockIdx.x*1024) + (rkh*64)) +␣
 ↪threadIdx.x) + rkw) - 65)]))
      }
    }
  }
  PoolAvg[((blockIdx.x*1024) + threadIdx.x)] = (PoolSum[0]*0.111111f)
}
```

### 5.7.2 Benchmarking

We use the pooling implementation of GPU in MXNet as the baseline. The benchmarking code can be reused from Section 4.10. Note that for TVM, we set `target` to be CUDA; for MXNet, we set `ctx` to be gpu.

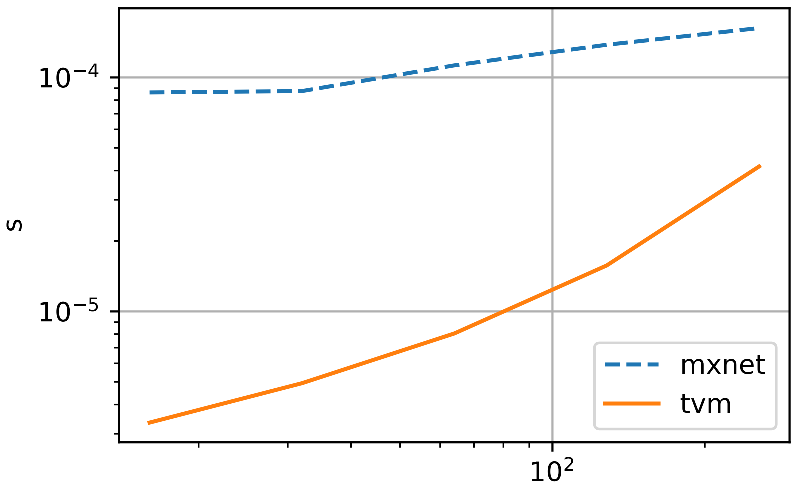
First, compare the `max pooling`.

```
channels = 2**np.arange(4, 9)
# a list of (c, n, k)
sizes = [(int(c), 64, 3) for c in channels]

tvm_max_times = d2ltvm.bench_pooling_tvm(schedule_max, sizes, target)
mxnet_max_times = d2ltvm.bench_pooling_mxnet('max', sizes, ctx='gpu')

times = [mxnet_max_times, tvm_max_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'tvm'], fmts=['--']*(len(times)-1)+['-'])
```



Then, compare the `avg pooling`.

```
tvm_avg_times = d2ltvm.bench_pooling_tvm(schedule_avg, sizes, target)
mxnet_avg_times = d2ltvm.bench_pooling_mxnet('avg', sizes, ctx='gpu')

times = [mxnet_avg_times, tvm_avg_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'tvm'], fmts=['--']*(len(times)-1)+['-'])
```



Note that we are plotting execution times, so low is better. Both results show that TVM completes pooling computation much faster than MXNet.

### 5.7.3 Summary

- Scheduling pooling on GPUs are analogous to scheduling on CPUs, similar tricks can be used, e.g. `compute_inline` and `compute_at`.
- Other than that, GPU specific optimization tricks are also used, e.g., thread and block binding.

### 5.7.4 Exercise

- Use `cache_read` and `cache_write` primitives to further schedule pooling, observe if there is any improvement, and think about the reason.

## 5.8 Batch Norm

This section talks about batch normalization defined in Section 3.6 on GPU.

### 5.8.1 Setup

```
%matplotlib inline
import tvm
from tvm import te
import numpy as np
import d2ltvm
import mxnet as mx
import timeit

target = 'cuda'
```

### 5.8.2 Schedule

In order to schedule the batch normalization on GPU, we first fuse the stages using `te.schedule.AutoInlineInjective`. By doing so, we end up one loop nest that we can easily parallelize (via binding CUDA blocks) and vectorize (via binding CUDA threads). Each CUDA thread operates on a series of numbers as what we did for convolution in Section 5.5. This is the simple default schedule we can get for batch normalization. The code snippet below does it and prints out the corresponding IR. It should be clear to you so far.

```
size = (32, 112)

def default_bn(size):
    c, n = size[:]
    X, Mean, Var, Gamma, Beta, Y = d2ltvm.batch_norm(c, n)
    sch = te.create_schedule(Y.op)
    te.schedule.AutoInlineInjective(sch)
    c, h, w = Y.op.axis[0:3]
    sch[Y].bind(c, te.thread_axis("blockIdx.x"))
    sch[Y].bind(h, te.thread_axis("threadIdx.x"))
    return sch, (X, Mean, Var, Gamma, Beta, Y)

sch, args = default_bn(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce T_add {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 32
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 112
  for (ax2, 0, 112) {
    T_add[(((blockIdx.x*12544) + (threadIdx.x*112)) + ax2)] =␣
↪((((X[(((blockIdx.x*12544) + (threadIdx.x*112)) + ax2)] - Mean[blockIdx.
↪x])/sqrt((Var[blockIdx.x] + 1e-05f)))*Gamma[blockIdx.x]) + Beta[blockIdx.x])
  }
}
```

An alternative scheduling scheme is to fuse all axes of the loop nest and restructure it into a 2-level for loop, where the inner one binds to CUDA threads and the outer one binds to CUDA blocks. The number of CUDA threads of a block is set to be the maximum, as we did in Section 5.7. The code snippet below does it and prints out the corresponding IR.

```
nt = 0
with tvm.target.create(target):
    nt = tvm.target.Target.current(allow_none=False).max_num_threads

def optimized_bn(size):
    c, n = size[:]
    X, Mean, Var, Gamma, Beta, Y = d2ltvm.batch_norm(c, n)
    sch = te.create_schedule(Y.op)
    te.schedule.AutoInlineInjective(sch)
    fused = sch[Y].fuse(*sch[Y].op.axis)
    bx, tx = sch[Y].split(fused, factor=nt)
    sch[Y].bind(bx, te.thread_axis("blockIdx.x"))
    sch[Y].bind(tx, te.thread_axis("threadIdx.x"))
    return sch, (X, Mean, Var, Gamma, Beta, Y)
```

(continues on next page)

(continued from previous page)

```
sch, args = optimized_bn(size)
print(tvm.lower(sch, args, simple_mode=True))
```

```
produce T_add {
  // attr [iter_var(blockIdx.x, , blockIdx.x)] thread_extent = 392
  // attr [iter_var(threadIdx.x, , threadIdx.x)] thread_extent = 1024
  T_add[((blockIdx.x*1024) + threadIdx.x)] = ((((X[((blockIdx.x*1024)␣
↪+ threadIdx.x)] - Mean[floordiv(((blockIdx.x*1024) + threadIdx.x),␣
↪12544)])/sqrt((Var[floordiv(((blockIdx.x*1024) + threadIdx.x), 12544)]␣
↪+ 1e-05f)))*Gamma[floordiv(((blockIdx.x*1024) + threadIdx.x), 12544)]) +␣
↪Beta[floordiv(((blockIdx.x*1024) + threadIdx.x), 12544)])
}
```

In this specific case, as the total number of entries ($32 \times 28 \times 28 = 25088$) cannot be divided by `max_num_threads` (1024), we see the `if` statement in the IR to guard the boundary. The term `likely` in the `if` statement indicates the compiler for better speculation. Another new keyword we see in the IR is `floordiv`, which is introduced to guide the access pattern. Note that for batch normalization, we have the same `Mean`, `Var`, `Gamma`, and `Beta` values for the same channel. When different channels are located in the same CUNDA block, we will need an extra calculation to locate the correct values.

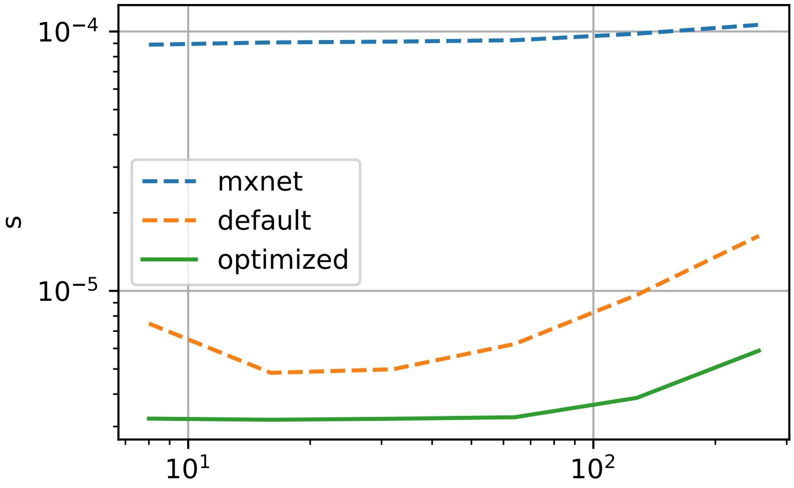
In general, we want to avoid the above additional steps if possible. However, in this case, as batch normalization is memory-bound, having more CUDA threads in a block, and correspondingly having more data shared in a block, is more important.

You may also notice that we do not use the `cache_read` and `cache_write` primitives to optimize the schedule. This is also due to the memory-bound nature of batch normalization, bringing data to local memory would not help much.

### 5.8.3 Benchmark

We use MXNet as the baseline to check the performance of batch normalization we achieve using TVM. The benchmarking methods were defined in Section 4.11. We plot the absolute times of the MXNet baseline and the two scheduling schemes described above, as functions of varied channels.

```
channels = 2**np.arange(3, 9, 1)
sizes = [(int(c), 28) for c in channels]
default_times = d2ltvm.bench_bn_tvm(default_bn, sizes, target)
optimized_times = d2ltvm.bench_bn_tvm(optimized_bn, sizes, target)
mxnet_times = d2ltvm.bench_bn_mxnet(sizes, ctx='gpu')
times = [mxnet_times, default_times, optimized_times]
d2ltvm.plot(channels, times, ylabel='s',
            xscale='log', yscale='log',
            legend=['mxnet', 'default', 'optimized'], fmts=['--']*(len(times)-
↪1)+['-'])
```



The results show that even with `if` statement and `floordiv`, the optimized schedule executes faster than the default the schedule for batch normalization. The batch normalization performance of MXNet is much worse, likely to be dominated by the function call overhead.

### 5.8.4 Summary

- Optimizing batch normalization on GPU is all about block and thread binding.
- For small operators, having additional steps in IR is acceptable as long as the number of CUDA threads in a CUDA block can be maximized.

### 5.8.5 Exercise

- Try to eliminate the `if` statement and `floordiv` from the IR generated by the optimized scheduling scheme of batch normalization, and check out the performance.
- Try to use `cache_write` to schedule batch normalization and observe the performance difference, if any, it brings.

# 6 | Neural Networks

A place holder

# 7 | Deployment

A place holder

# 8 Discussions[48]

[48] https://discuss.tvm.ai/t/d2l-tvm-a-tvm-introduction-book/4305

# Bibliography

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., … Zhang, Z. (2015). Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., … others. (2018). Tvm: an automated end-to-end optimizing compiler for deep learning. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (pp. 578–594).

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (2017). Mobilenets: efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Lai, J., & Seznec, A. (2013). Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (pp. 1–10).

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., & Wang, Y. (2019). Optimizing cnn model inference on cpus. *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (pp. 1025–1040).

Nath, R., Tomov, S., & Dongarra, J. (2010). An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, *24*(4), 511–515.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 519–530). ACM.

Roesch, J., Lyubomirsky, S., Kirisame, M., Pollock, J., Weber, L., Jiang, Z., … Tatlock, Z. (2019). Relay: a high-level ir for deep learning. *arXiv preprint arXiv:1904.08368*.